On the Construction of Rectilinear Steiner Minimum Trees among Obstacles

HUANG, Tao

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
January 2013

Abstract

Rectilinear Steiner minimum tree (RSMT) problem asks for a shortest tree spanning a set of given terminals using only horizontal and vertical lines. Construction of RSMTs is an important problem in VLSI physical design. It is useful for both the detailed and global routing steps, and it is important for congestion, wire length and timing estimations during the floorplanning or placement step. The original RSMT problem assumes no obstacle in the routing region. However, in today's designs, there can be many routing blockages, like macro cells, IP blocks and pre-routed nets. Therefore, the RSMT problem with blockages has become an important problem in practice and has received a lot of research attentions in the recent years. The RSMT problem has been shown to be NP-complete, and the introduction of obstacles has made this problem even more complicated.

In the first part of this thesis, we propose an exact algorithm, called ObSteiner, for the construction of obstacle-avoiding RSMT (OARSMT) in the presence of complex rectilinear obstacles. Our work is developed based on the GeoSteiner approach in which full Steiner trees (FSTs) are first constructed and then combined into a RSMT. We modify and extend the algorithm to allow rectilinear obstacles in the routing region. We prove that by adding virtual terminals to each routing obstacle, the FSTs in the presence of obstacles will follow some very simple structures. A two-phase approach is then developed for the construction of OARSMTs. In the first phase, we generate a set of FSTs. In the second phase, the FSTs generated in the first phase are used to construct an OARSMT. Experimental results show that ObSteiner is able to handle problems with hundreds of terminals in the presence of up to two thousand obstacles, generating an optimal solution in a reasonable amount of time.

In the second part of this thesis, we propose the OARSMT problem with slew constraints over obstacles. In modern VLSI designs, obstacles usually block a fraction of metal layers only making it possible to route over the obstacles. However, since buffers cannot be place on top of any obstacle, we should avoid routing long wires over obstacles. Therefore, we impose the slew constraints for the interconnects that are routed over obstacles. To deal with this problem, we analyze the optimal solutions and prove that the internal trees with signal direction over an obstacle will follow some simple structures. Based on this observation, we propose an exact algorithm, called ObSteiner with slew constraints, that is able to find an optimal solution in the extended Hanan grid. Experimental results show that the proposed algorithm is able to reduce nearly 5% routing resources on average in comparison with the OARSMT algorithm and is also very much faster.

Acknowledgments

First, my greatest thanks and appreciation go to my supervisor Evangeline F. Y. Young. Without her insightful guidance, advise, and continuous encouragement, this thesis would not have been possible. I have learned a lot from her kindness, patience, and positive attitude towards life. All these would be invaluable through out my life. I'm especially grateful for the days back to 2009 when she introduced the Steiner tree problem to me. Little did I know that this elegant problem would bring me so much fun, hard work, frustration, satisfaction, and fulfilment.

Also I would like to thank my fellow labmates Xiao Linfu, Qian Zaichen, Xiao Zigang, Jiang Yan, Tian Haitong, Cui Guxin, He Xu, Qian Fuqiang, Chow Wing Kai, Kuang Jiang, and Cai Wenzan. Thanks for all your helps during my Ph.D. study. I won't forget the late nights we worked together for the contests, and all the fun we have had in the last three years.

Finally, I want to express my heartfelt thanks to my parents Huang Zengzhang and Huang Ruizhu, for their love, understanding, and constant support. Their encouragement has always been a powerful source of inspiration and energy. Without them, this dissertation would not exist. I would also like to express my deepest gratitude to Liang Yuting for everything. Thank you for being there for me from the very beginning.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Contents

1.1 The rectilinear Steiner minimum tree problem

The Steiner minimum tree (SMT) problem asks for a shortest network that spans a set of given points in a metric space. The set of given points are usually referred to as *terminals* and new auxiliary *Steiner points* can be introduced so that the total length of the network can be reduced. The history of the SMT problem started with Fermat (1601-1665) who proposed the problem: given three points in a plane, find a fourth point such that the sum of its distances to the three given points is a minimum. Courant and Robbins [13] in their famous book "What Is Mathematics?" first named the problem after Steiner (1796-1863) who solved the problem

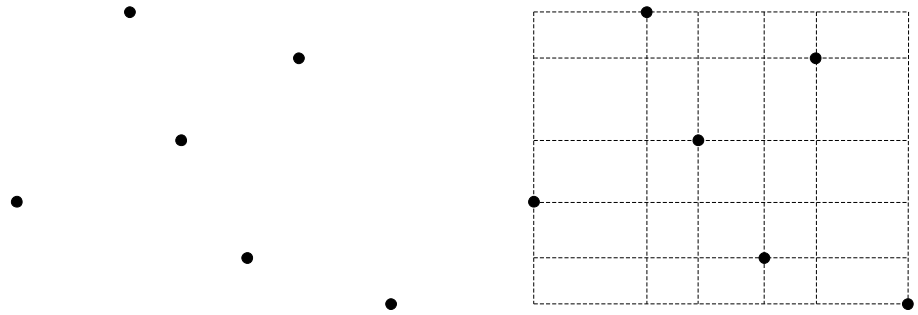


Figure 1.1: Hanan grid.

of joining three villages by a system of roads having minimum total length. The popularity of this book has raised the research interests in the SMT problem.

The formulation of the SMT problem is as follows:

The Steiner minimum tree problem: Given a set V of n terminals in the space L_p[1]. Find a shortest tree embedded in the space that spans V.

The original SMT problem considers the Euclidean space (i.e. L_2 space). The rectilinear Steiner tree problem (i.e. in L_1 space) is firstly considered by Hanan [25]. The problem is equivalent to finding a tree connecting all the terminals by using only horizontal and vertical lines. An optimal solution to this problem is called a rectilinear Steiner minimum tree (RSMT). Hanan prove that there is at least one RSMT that is contained in the Hanan grid. The Hanan grid, as shown in Fig. 1.1, can be obtained by constructing horizontal and vertical lines through each terminal and the intersections of these lines are thus candidate Steiner points. Although there is a finite number of candidate Steiner points in the Hanan grid, it is still a very difficult problem to select a subset of them to construct a RSMT. In fact, the RSMT problem is shown to be NP-complete by Garey and Johnson [39]. Moreover, they also showed that the Euclidean Steiner minimum tree (ESMT) problem is NP-hard.

[1]The distance between two points in the L_p space can be calculated by $d(u,v) = (|u_x - v_x|^p + |u_y - v_y|^p)^{\frac{1}{p}}$

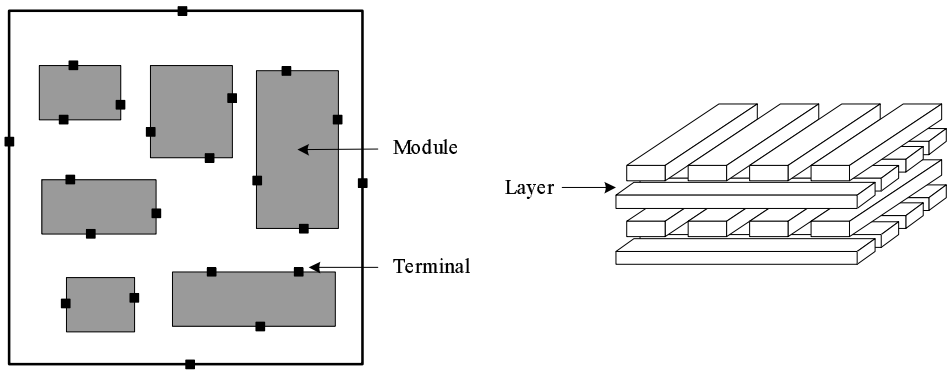


Figure 1.2: An example of the routing problem.

1.2 Applications

The RSMT problem has many applications in VLSI physical design.

In the VLSI physical design flow, one important step is routing. The specification of a routing problem usually consists of a set of modules, a netlist, and the area available for routing. Each module has a set of terminals and is fixed in position. A netlist is a set of nets. Each net consists of a set of terminals that need to be made electronically equivalent (i.e. connected by wires). In modern VLSI design, there exist multiple routing layers, and each routing layer has a predefined direction (either horizontal or vertical) and routing capacity. Connectivity between layers can be achieved by vias. The objective of routing is to create an interconnection among the terminals of same nets such that the total wire length (i.e. routing resource) is minimized. For high performance design, it is also necessary to consider other requirements such as timing budget, signal integrity, and manufacturability issues. An example of the routing problem is shown in Fig. 1.2.

In VLSI deign, routing is usually performed in two stages: global routing followed by detailed routing. The task of global routing is to first partition the routing region into tiles and then determine a loose tile-to-tile route for each net. In this stage, terminals within the same tile are assumed to be at the center of the tile. It

is also common to represent a 3D routing problem as a 2D problem and perform layer assignment as a post-processing step. Therefore, the routing of a net can be realized by constructing a RSMT. A common approach for global routing is to first generate RSMTs for all the nets [38]. Since, RSMT only minimizes the wire length, it is possible that in some tiles, the number of wires may exceed the routing capacity creating some congested regions. In such cases, nets that are routed through the congested region will be ripped up and rerouted by using congestion-aware RSMT [40] or the maze routing algorithm. Given a global routing solution, detailed routing determines the actual geometric layout of each net (i.e. exact tracks, via position, and layer) within the assigned routing regions. In this stage, the RSMTs can also be used to guide the routing [57] to minimize the wire length and via usage.

Despite extensive applications in the routing stage, RSMTs can also find its application in an even earlier stage in VLSI design flow, such as floorplanning and placement. In floorplanning and placement, modules are not fixed and their positions are to be determined. A solution to the problem is a layout that specifies the location of each module such that there is no overlap. A good floorplanning or placement solution should be routable (i.e. be successfully routed in the later routing stage) by using the smallest amount of routing resources. This necessitates congestion and wire length estimations during floorplanning and placement. The estimation can be done by performing routing, but it is computationally too expensive. Therefore, using RSMTs as an approximation becomes an efficient alternative and is adopted by many estimation approaches [28]. Another target of floorplanning and placement is to achieve good timing. As deep submicron technology advances, interconnect delay is becoming increasingly dominant over transistor and logic delay. Timing estimation has to consider both interconnect and gate delays

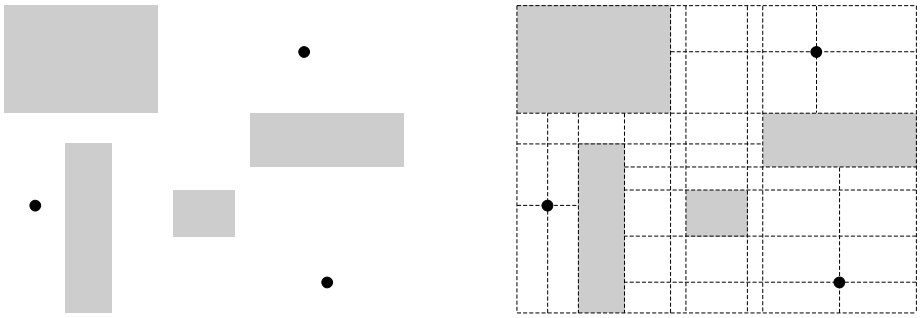


Figure 1.3: Escape graph.

in order to be accurate. This requires actual topology of each net which is usually approximated by using RSMTs [21].

1.3 Obstacle consideration

A more general version of the RSMT problem is to consider obstacles. An obstacle is a rectilinear polygon, i.e., all boundary edges of an obstacle are either horizontal or vertical. The RSMT problem in the presence of obstacles is of practical interest because such obstacles exist in modern VLSI designs (e.g. macro cells, IP blocks, and pre-routed nets).

In the routing region, an obstacle blocks some metal layers. If the obstacle blocks all the metal layers, the routing tree has to avoid it. A RSMT that avoids obstacles is called an obstacle-avoiding RSMT (OARSMT). Analogous to the Hanan grid for the RSMT problem, Ganley and Cohoon [31] proposed an escape graph for the OARSMT problem. The escape graph consists of two types of segments. The first type is the segments that extend from the terminals in the vertical and horizontal directions, until an obstacle boundary is met. The second type of segments can be obtained by extending the boundary segments of each obstacle until an obstacle boundary is met. An example of the escape graph is shown in Fig. 1.3.

It is proven in [31] that for any OARSMT problem, there is at least one optimal solution composed only of the escape segments in the escape graph.

If an obstacle blocks only a fraction of metal layers, then routing wires on top of obstacles is possible. However, if a long wire is routed over an obstacle, there will be signal integrity problems because buffers cannot be placed on top of any obstacle. As deep submicron technology advances, interconnect delay is becoming increasingly dominant over transistor and logic delay [8]. High interconnect resistance will cause signal integrity to degrade rapidly in a long connection. This problem is usually solved by inserting buffers that break a long wire into small segments. Notice that buffers cannot be placed on top of any obstacle. Therefore, although routing over obstacles is possible, one should be aware of the signal integrity issue and avoid routing long wires on top of obstacles that may lead to complicated post-routing electrical fixups. In this case, the OARSMT can be an option. However, avoiding all obstacles may result in an unnecessary resource wastage. A smarter router should be able to avoid some of the obstacles that cause problems, while allowing wires to cross the others.

1.4 Thesis outline

This dissertation studies the RSMT problem in the presence of obstacles.

In Chapter 2, we do a literature review of the RSMT and OARSMT problem. We introduce a set of heuristics and exact algorithms including the state-of-the-art for the RSMT and OARSMT problem.

In Chapter 3, we propose an exact algorithm, called ObSteiner, for the construction of OARSMTs among complex rectilinear obstacles. ObSteiner is a two-phase approach in which the optimal solution is constructed by the concatenation of full

Steiner trees (FSTs) among complex obstacles. We first show that, by adding virtual terminals, the FSTs among complex obstacles can be greatly simplified, thus providing the theoretical foundations for the exact approach. We then describe the two-phase algorithm in detail including the FST generation phase, the FST pruning procedure, and the FST concatenation phase. ObSteiner is able to handle complex obstacles including both convex and concave ones. Experimental results show that benchmarks with hundreds of terminals among a large number of obstacles can be solved optimally in a reasonable amount of time.

In Chapter 4, we study a variant of the RSMT problem in the presence of obstacles that allows wires to be routed over obstacles. In modern designs, obstacles usually block the device layer and a fraction of metal layers only. Therefore, routing wires on top of obstacles is possible. However, if a large amount of wires are routed over an obstacle, it may cause signal integrity problems because buffers cannot be placed on top of any obstacle. To tackle this problem, we impose slew constraints on the interconnects that are routed over an obstacle. This is called the OARSMT problem with slew constraints over obstacles. We first analyze an optimal solution to this problem and find that the tree structures over obstacles with slew constraints will follow some very simple forms. Based on this observation, we propose an algorithm, called ObSteiner with slew constraints, to find an optimal solution embedded in the extended Hanan grid. The solutions can guarantee the interconnect performance and avoid post-routing electrical fixups due to slew violations. We also show that the solutions provided by our algorithm can save over 5% routing resources on average in comparison with the OARSMTs that avoid all obstacles.

In Chapter 5, a conclusion of this thesis is drawn.

1.5 Thesis contributions

The contributions of this dissertation can be summarized as follows.

For the OARSMT problem:

1. This is the first work to propose a geometric approach to exactly solve the OARSMT problem when there are complex rectilinear obstacles. ObSteiner is able to handle both convex and concave rectilinear obstacles, while previous exact algorithm can only handle rectangular obstacles.

2. We design an efficient pruning procedure which can greatly reduce the size of the solution space and therefore improve the performance of the algorithm. For the second phase of the algorithm, we propose a new formulation for the concatenation of FSTs. In the branch-and-cut search, we develop new separation algorithm to adapt to the presence of virtual terminals. We also adopt an incremental way to handle obstacles. An obstacle will be considered only if it is necessary. By using ObSteiner, benchmarks with up to two thousand obstacles can be solved to optimal in a reasonable amount of time, while previous exact algorithm can only deal with benchmarks with around twenty obstacles.

3. Based on the theorem we developed in this thesis, we further propose a simple graph model that can transfer the geometric OARSMT problem into a graph problem. We prove that the proposed graph model contains at least one optimal solution and is also simpler (in terms of the number of edges and nodes) than the simplest graph model in the literature.

For the OARSMT with slew constraints over obstacles:

1. We formulate the OARSMT problem with slew constraints over obstacles. The solution to this problem is a resource efficient Steiner tree that anticipates good interconnect performance.

2. We analyze an optimal solution to this problem and find that the slew constrained tree structures over obstacles will follow some very simple forms.

3. We propose an algorithm that can find an optimal solution embedded in the extended Hanan grid and show that the solutions provided by our algorithm can save a significant amount of routing resources and run time in comparison with the state-of-the-art optimal OARSMT algorithm.

Finally, a combination of the above researches provides a powerful tool for solving the RSMT problem in the presence of obstacles. With our optimal methods, we can easily compare the performance of different approaches and see how far a heuristic solution is away from the optimum. The works presented in this dissertation give key insights into this difficult problem.

CHAPTER 2

Background

Contents

2.1 RSMT algorithms

2.1.1 Heuristics

The RSMT problem is NP-complete. It means that efficient polynomial time exact algorithm may not exist. Therefore, many researches of the RSMT problem have been focused on the development of heuristics. Early heuristics are mainly based on improving over a RMST. Starting in 1990s, a new class of RSMT heuristics that do not rely on the RMST has been proposed. Two typical examples are iterated

one Steiner and batched iterated one Steiner. Recently, a look up table based algorithm called FLUTE is proposed. Comparing with the other heuristics, FLUTE can provide the best tradeoff between runtime and accuracy, and therefore is the state-of-the-art algorithm. In this section, a brief introduction to these approaches is presented.

2.1.1.1 RMST based heuristics

Let $|RSMT(V)|$ and $|RMST(V)|$ be the length of the RSMT and RMST over V respectively, The rectilinear Steiner ratio is defined as

$$\rho(L_1) = \inf_V \left\{ \frac{|RSMT(V)|}{|RMST(V)|} \right\} \tag{2.1}$$

where V is a set of points in the rectilinear plane. That is, the rectilinear Steiner ratio is the largest possible ratio between the length of a RSMT and the length of a RMST in the rectilinear plane. It has been proved that the rectilinear Steiner ratio is $\frac{2}{3}$ [26]. This means that any heuristic based on improving over a RMST can guarantee a worst-case performance ratio of $\frac{3}{2}$. Therefore, many RSMT heuristics in the literature use RMST-based strategies.

A RMST can be computed in $O(n\log n)$ time. The first RMST algorithm with this complexity is proposed by Hwang [27] and the algorithm is based on the construction of the rectilinear Voronoi diagram. Hwang showed that the rectilinear Voronoi diagram can be built in $O(n\log n)$ time. It can also be verified that a RMST can be computed in $O(n)$ time by using the Voronoi diagram, and therefore the complexity of finding a RMST is $O(n\log n)$. However, the computation of Voronoi diagram can be tedious. A simpler way is to use the nearest neighbors of each terminal. For each terminal we divide its surrounding area into eight regions

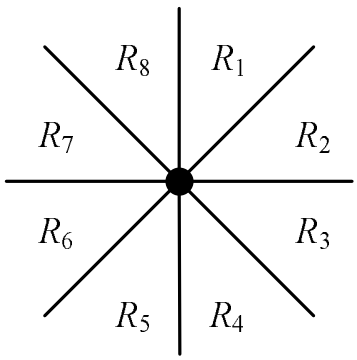


Figure 2.1: Eight regions of a terminal.

separated by lines that intersect at a 45-degree angle, as shown in Fig. 2.1. The following theorem is firstly proposed by Yao [58].

Theorem 2.1. *In a RMST, if two terminals v and u are connected, then v is the nearest to u in one of the eight regions of u.*

Theorem 2.1 shows that for the construction of RMST, only the edges connecting nearest neighbors in the eight regions need to be considered. Finding the nearest neighbor of all terminals in all eight regions can be done in $O(n\log n)$ time [20, 24]. Since there are at most $8n$ edges, a RMST can be therefore found in $O(n\log n)$ time by using either the Prim's or the Kruskal's algorithm.

With a RMST as a starting point, a direct way to improve and obtain a RSMT is to remove overlapping segments by introducing Steiner points. These approaches are called Steinerization. Early overlap removal schemes all make use of simple heuristics. A pair of edges sharing a common terminals are chosen arbitrarily. If there is overlap, they are embedded by adding a Steiner point. This process terminates until all pairs of neighboring edges are explored. A comparison between different ways on selecting pairs of edges to be processed is done by Richards [43]. Later, Ho *et al.* [32] gave a polynomial time algorithm to find an optimal embedding. The algorithm starts with a special kind of RMST called separable RMST. A

RMST is separable if and only if for any pair of non-adjacent edges in the tree, any staircase layouts of the two edges will not intersect or overlap. They first gave a $O(n^2)$ time algorithm for the construction of separable RMST. Based on the separable RMST, an $O(n)$ time optimal algorithm is proposed with the assumption that each edge has at most one corner (i.e. L-shaped). The algorithm starts by making a terminal as the root of the tree and solve the problem in a bottom-up fashion. The key observation is that the optimal solution of a subtree depends only on how the edge connecting the root node of the subtree and its parent, is embedded. Since only L-shaped edges are considered, there are two options for embedding. Therefore, an $O(n)$ dynamic programming algorithm can find optimal solution. Ho *et al.* further extended the algorithm to handle the case when each edge has at most two corners (i.e. Z-shaped). The difference is that there can be more embedding options for each subtree. Ho *et al.* showed that the corresponding dynamic programming algorithm has a time complexity of $O(n^7)$. Finally, they proved that the resulting RSMT after optimal Z-shaped embedding is also optimal when there is no restriction on edge shapes.

Another way to improve over a RMST is to add some new edges to replace longer ones repeatedly. These approaches are called edge-substitution. Borah *et al.* [37] proposed an edge-based heuristic that starts with a RMST and incrementally improves the cost by connecting a node[1] to a neighboring edge and removing the longest edge in the loop thus formed. The reduction in the cost of the tree due to this operation is the gain. The algorithm works in an iterative manner. In each iteration, a set of such (node, edge) pairs are found and updates are applied to the tree starting from the (node, edge) pairs with the largest gain. Borah *et al.* showed that finding all possible (node, edge) pairs with positive gain can be done in $O(n\log n)$

[1] A node can be a terminal or a Steiner point

time and applying the updates to the tree requires only $O(n)$ time. They further showed that three iterations are sufficient in most cases. Therefore, the complexity of the algorithm is $O(n\log n)$. Zhou *et al.* [23] extended the edge-based heuristic by using a spanning graph [24]. A spanning graph is an undirected graph over the points that contains at least one MST. They showed that finding potential (node, edge) pairs in the spanning graph can be more efficient. They also proposed a simpler way to find the longest edge on the loop formed by connecting a node to an edge with a binary tree merging approach. Although, the run time is dominated by the spanning graph and RMST generation, which take $O(n\log n)$ time, a good practical performance can be achieved.

2.1.1.2 Iterated 1-Steiner

While the RMST-based heuristics can guarantee a worst case performance ratio of $\frac{3}{2}$, it is still a problem to find such a heuristic method with performance ratio strictly less than $\frac{3}{2}$. Kahng and Robins [2] showed that the $\frac{3}{2}$ bound is tight for a large number of RMST-based methods. Motivated by this fact, Kahng and Robins [3] proposed a heuristic called iterative 1-Steiner that does not, implicitly or explicitly, make use of a RMST. The algorithm is based on the answer to the following question. If at most one Steiner point is allowed, what is the optimal Steiner tree and where should the Steiner point be placed? This is called the 1-Steiner problem.

In the Euclidean plane, Georgakopoulos and Papadimitriou [19] are the first to give an $O(n^2)$ algorithm to solve the 1-Steiner problem. Kahng and Robins adapted this method for the rectilinear plane. The algorithm makes use of the concept of nearest neighbor for the construction of RMST to partition the plane into $O(n^2)$ isodendral regions. An important property of isodendral regions is that introducing any point in a given region will result in a constant RMST topology. Therefore,

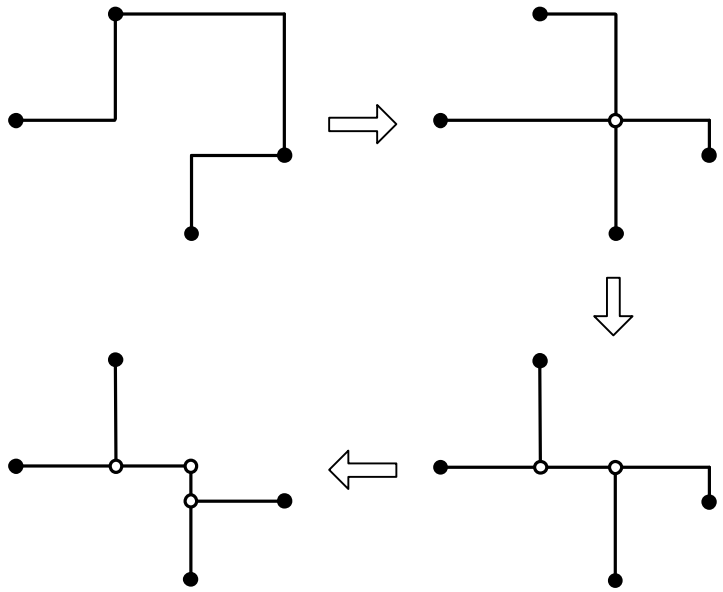


Figure 2.2: An example of the iterative 1-Steiner algorithm.

after an $O(n^2)$ preprocessing step, updating the RMST to include a new point requires only constant time. Moreover, the optimal Steiner point in each region can also be determined in constant time. As a result, the 1-Steiner problem can be solved in $O(n^2)$ time by iterating through the isodendral regions and selecting the point with the lowest cost.

The iterative 1-Steiner heuristic works by iteratively calculating optimal 1-Steiner points and include them into the point set. Accepted Steiner points are deleted if they become useless, i.e., if their degree becomes 1 or 2 in the tree. The algorithm terminates when no improvement can be achieved by adding new Steiner points or the maximum number of iterations has been reached. An example of the iterative 1-Steiner heuristic is shown in Fig. 2.2. In [3], the maximum number of iterations is set to be the number of terminals n. Therefore, the overall time complexity of iterative 1-Steiner is $O(n^3)$.

2.1.1.3 Batched iterated 1-Steiner

Kahng and Robins [3] proposed several variants to the iterative 1-Steiner. Among those variants, the most promising one make use of a batched way to include Steiner points. Instead of adding one Steiner point per iteration, a maximal independent set of Steiner points are included.

The heuristic starts by evaluating every candidate Steiner points in the Hanan grid. By preprocessing the $O(n^2)$ isodendral regions as a planar subdivision, the planar region in which a given point lies can be determined in $O(\log n)$ time. This preprocessing requires $O(n^2 \log n)$ time. Since the MST of a planar weighted graph can be maintained using $O(\log n)$ time per addition of a point, the RMST cost savings for all the candidate Steiner point can be calculated in $O(n^2 \log n)$ time. Then, the Steiner point candidates are sorted according to their gains on cost savings in decreasing order. Next, all of the candidates are processed in order. Each candidate with a positive gain are added, as long as it is independent of all the Steiner points previously added during the round. The criterion for independence is that no candidate is allowed to reduce the potential MST cost saving of any other candidate in the added set. This process iterates until no Steiner point can be included. The total time required for one iteration is $O(n^2 \log n)$. Since Steiner point candidates are added in a batched way, the number of iterations required grows much more slowly than the number of Steiner points considered. Empirical study showed that batched iterated 1-Steiner performs close to iterated 1-Steiner, but the computational cost is much lower.

Although batched iterated 1-Steiner can be implemented to run in $O(n^2 \log n)$ per iteration, the computational geometric methods have a large hidden constant and are also difficult to code. Therefore, an $O(n^4 \log n)$ implementation is used in [3]. A more efficient $O(n^3)$ implementation is later presented by Griffith *et*

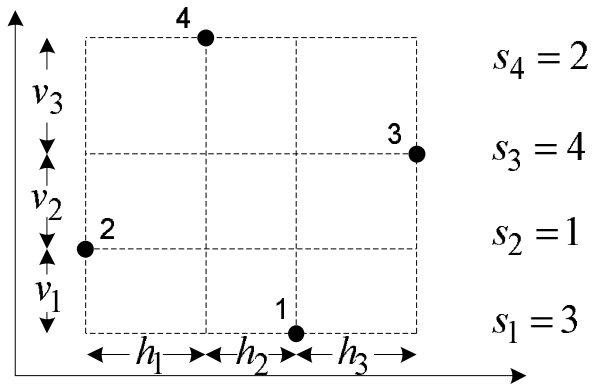


Figure 2.3: An example of the position sequence of a net.

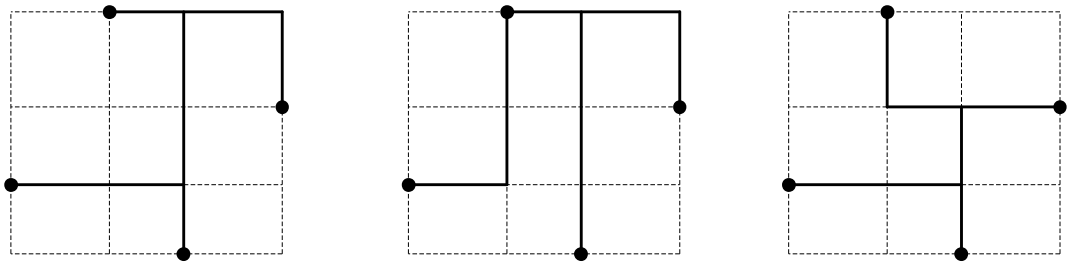


Figure 2.4: An example of different Steiner trees for a net.

al. [29]. Experimental results showed that a speedup factor of three orders of magnitude over previous implementation can be achieved.

2.1.1.4 FLUTE

The RSMT problem has many applications in very large scale integration (VLSI) design. In VLSI circuits, many nets have just a small number of terminals. Therefore, it is more important for RSMT algorithms to be simple and efficient for small problems. Based on this observation, Chu and Wong [4] proposed a RSMT algorithm called fast lookup table estimation (FLUTE).

Given a set of n terminals, the Hanan grid can be built by drawing horizontal and vertical lines through each terminal. Let x_i be the x-coordinates of the vertical grid lines such that $x_1 \leq x_2 \leq \ldots \leq x_n$, and y_i be the y-coordinates of the vertical

grid lines such that $y_1 \leq y_2 \leq \ldots \leq y_n$. Label the terminal in ascending order of the y-coordinates and let s_i be the rank of terminal i in ascending order of the x-coordinates. The sequence $s_1 s_2 \ldots s_n$ is called the position sequence. An example is shown in Fig. 2.3 where the position sequence of the net is 3142. Let $v_i = y_{i+1} - y_i$ and $h_i = x_{i+1} - x_i$ be the distance between adjacent Hanan grid lines. Since a Steiner tree in the Hanan grid is a union of Hanan grid edges, the length of any Steiner tree can always be written as a linear combination of edge lengths in which every coefficient is a positive integer. For example, the length of the three Steiner trees as shown in Fig. 2.4 can be expressed by $h_1 + 2h_2 + h_3 + v_1 + v_2 + 3v_3$, $h_1 + h_2 + h_3 + v_1 + 2v_2 + 3v_3$, and $h_1 + 2h_2 + h_3 + v_1 + v_2 + v_3$. Therefore, a lookup table can be used to store the lengths of all possible Steiner trees as linear combinations of h_i and v_i. For simplicity, only the vectors of the coefficients are stored, e.g. (1, 2, 1, 1, 1, 3), (1, 1, 1, 1, 2, 3), and (1, 2, 1, 1, 1, 1). It is also easy to find that some vectors are suboptimal, e.g. the length induced by (1, 2, 1, 1, 1, 3) cannot be shorter than that of (1, 2, 1, 1, 1, 1). A vector that can potentially produce the optimal length is called a POWV. For each POWV, a set of corresponding RSMTs called POST are also stored. A key observation is that, if two nets have the same position sequence, then every Steiner tree of one net is topologically equivalent to a Steiner tree of the other net. This means that nets with the same position sequence can be grouped together to share the set of POWVs and the following theorem can be stated.

Theorem 2.2. *The set of all nets with n terminals can be divided into $n!$ groups according to the position sequence such that all nets in each group share the same set of POWVs.*

FLUTE makes use of precomputed lookup table of POWVs and POSTs. Given a net, its position sequence is firstly determined and the corresponding POWVs are extracted from the table. The tree length of each POWV is computed according

to the values of h_i and v_i and the POWV with minimum length is selected. The corresponding POSTs are the RSMTs for the net.

The precomputation of the lookup table for small nets can be done by enumerating all possible Steiner trees in the Hanan grid. For larger nets, a boundary-compaction technique is proposed to efficiently generate all possible POWVs and POSTs. Some reductions are also applied to reduce the size of the lookup table. It is reported that the total table size is only 9.00 MB for all nets with up to 9 terminals.

FLUTE is able to generate optimal RSMTs for small nets (e.g. with up to 9 terminals) by using the lookup table. However, for large nets, the lookup table approach is impractical because of the high cost in both space and time. Therefore, a large net is divided into small nets with only the breaking terminals in common by using a net breaking heuristic. Each small net is then solved by using the lookup table and the resulting RSMTs are combined to form a RSMT for the original net. Finally, some refinement schemes are applied to eliminate overlapping segments or further reduce the length of the tree.

The total run time complexity of FLUTE is $O(n\log n)$. Empirical results on VLSI design showed that FLUTE is more accurate than the batched 1-Steiner heuristic and is almost as fast as a very efficient implementation of Prim's RMST algorithm.

2.1.2 Exact algorithms

In previous sections, we mentioned that at least one RSMT can be found in the Hanan grid graph. Therefore, exact algorithms for the Steiner problem in networks [16] can also be used to solve the RSMT problem. However, these approaches are considered to be less effective for the RSMT problem because they

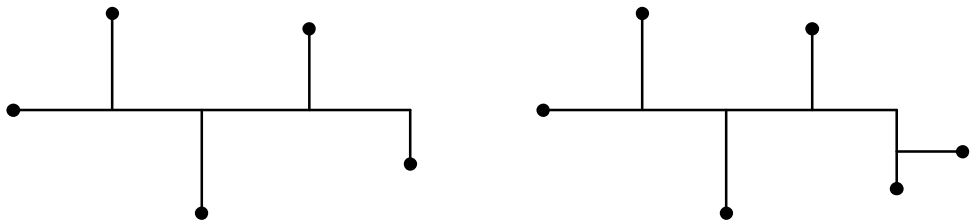


Figure 2.5: Two generic forms for a FST when $n > 4$.

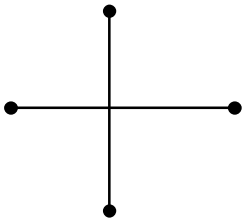


Figure 2.6: The only exception to Theorem 2.3.

do not exploit the geometric of the problem. Therefore, in this section, we will focus on the geometric approaches.

Let V' be a set of points in the plane, and T be a SMT spanning V'. T is said to have a *full topology* if every point in V' is a leaf node in T. A terminal set V' is a *full set* if every SMT for V' has a full topology. A full Steiner tree (FST) is a SMT that spans a full set of terminals. It can be easily verified that any SMT can be uniquely decomposed into a set of edge-disjoint FSTs by splitting at the terminals with degree[2] more than one. In the rectilinear plane, Hwang [26] first characterized the structures of FSTs. A series of lemmas are developed to reach the following Theorem.

Theorem 2.3. *For a full set of $n > 4$ terminals in the rectilinear plane, there exists a corresponding FST that either consists of a single line with $n-1$ alternating incident segments, or a corner with $n-3$ alternating segments incident to one leg and a single segment incident to the other leg.*

[2]The degree of a terminal is the number of edges connecting it.

The two FST structures described in Theorem 2.3 are shown in Fig. 2.5. Hwang also showed that Theorem 2.3 holds for $n = 2$, 3, or 4. The only exception is when $n = 4$ and the four terminals are the endpoints of a cross as shown in Fig. 2.6. We call these FST topologies, i.e., Fig. 2.5 and Fig. 2.6, Hwang's topology. Since any RSMT can be uniquely decomposed into a set of FSTs and FSTs are much simpler to construct than RSMTs, a straightforward strategy to construct RSMTs is to use a two-phase approach. The first phase is to generate a set of FSTs such that there is at least one RSMT composed of the FSTs in the set only. This phase is called the FST generation phase. In the second phase, a subset of FSTs with the minimum total length are selected and combined such that all terminals are connected. This phase is called the FST concatenation phase.

2.1.2.1 FST generation

Salowe and Warme [45] gave the first rectilinear FST generation algorithm. The algorithm generates FST by considering all pairs (a,b) of terminals as *backbone* in Hwang's topology. The backbone is the complete corner in Hwang's topology connecting the first terminal from the left and the last or the second last terminal as described in Theorem 2.3. In the corner, the leg with alternating incident segments is called the long leg, and the other is called the short leg. For each pair (a,b), all candidates terminals that can be attached to the backbone are found. Then, the candidate terminals are tried recursively to be attached to the backbone and the resulting structure is tested to check if a FST can be formed. Some screening tests are developed to eliminate those FSTs that cannot be in any RSMT. The algorithm is able to generate FSTs for 100 terminals in a short time. However, it is impractical for larger instances because of the high computational cost. Later, Warme [49] improved this algorithm to handle 1000-terminal instance in hours.

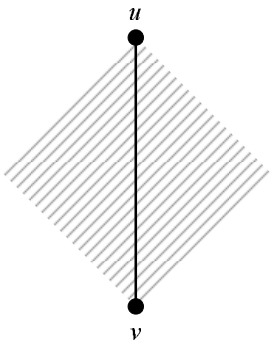


Figure 2.7: Empty diamond.

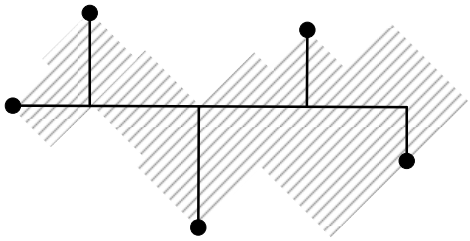


Figure 2.8: Empty diamond regions with respect to a FST.

The state-of-the-art rectilinear FST generation algorithm is presented by Zachariasen [61]. Let the root of a FST be the terminal incident by the long leg. For a given root z, the algorithm works by growing the long legs in four possible directions. For a given direction, the algorithm recursively try to attach terminals to the long leg. A series of necessary conditions are used to prune away useless FSTs.

The *empty diamond property* states that no other points of the RSMT can lie in $\mathscr{L}(u,v)$, where uv is a (horizontal or vertical) segment and $\mathscr{L}(u,v)$ is an area on the plane such that all the points in this area are closer to both u and v than u and v are to each other. The empty diamond region of a segment is shown in Fig. 2.7. If there is a terminal w inside the empty region of segment uv, we can delete uv and connect either uw or vw to reduce the length of the tree, a contradiction. The empty diamond regions with respect to a FST are shown in Fig. 2.8.

Let uw and vw denote two perpendicular segments sharing a common endpoint

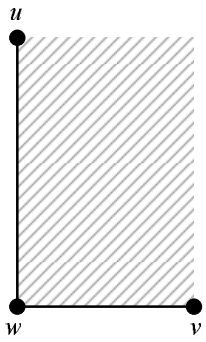


Figure 2.9: Empty corner rectangle.

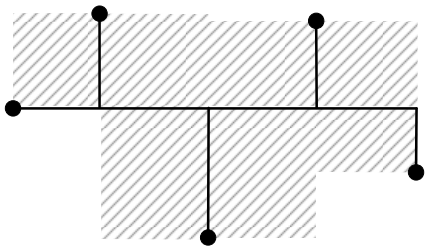


Figure 2.10: Empty corner rectangle regions with respect to a FST.

w. The *empty corner rectangle property* states that no other points of the RSMT can lie in the interior of the smallest axis-aligned rectangle containing *u* and *v*. The empty corner rectangle region is shown in Fig. 2.9. Assume that there is a terminal *x* inside the empty rectangle region. The unique path *P* from *x* to *w* in the RSMT visits either *u* or *v* first, or none of them, before reaching *w*. If *P* visits *u* (*v*) first, we can delete *uw* (*vw*) and add a vertical (horizontal) segment from x to a point on *vw* (*uw*), forming a tree with shorter length. If *P* reaches neither *u* nor *v* before reaching *w*, we can delete *uw* or *vw* and add *ux* or *vx* depending on the location of *x* to obtain a shorter tree, a contradiction. The empty corner rectangle regions with respect to a FST are shown in Fig. 2.10.

The *empty inner rectangle property* can be used to prune away useless FSTs. A FST can be transformed to its corner-flipped version by shifting segments and flipping corners as shown in Fig. 2.11. The empty inner rectangle property states

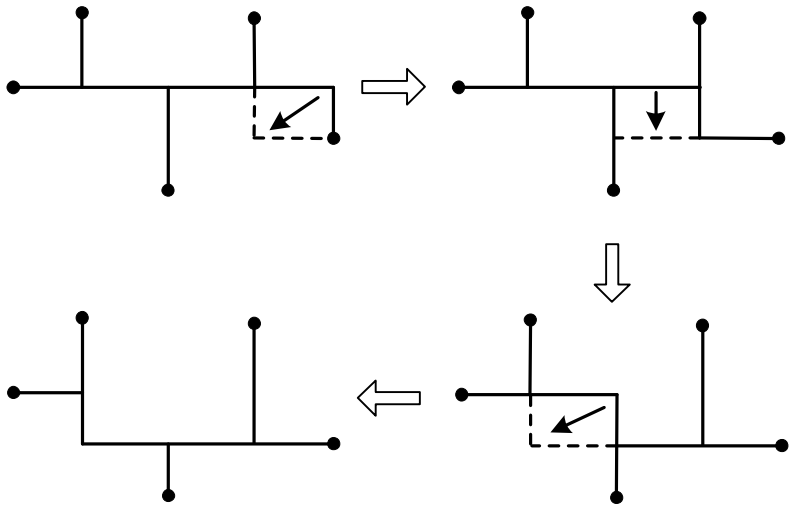


Figure 2.11: Transformation of a FST to its corner-flipped version.

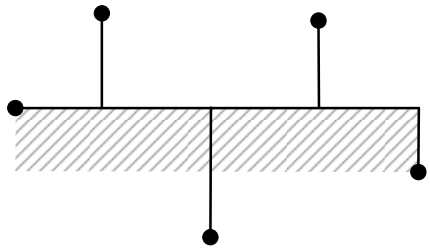


Figure 2.12: Empty inner rectangle in a FST.

that no terminal can be in between the backbone of the origin topology and that of the corner-flipped topology. The empty region with respect to a FST is shown in Fig. 2.12. Assume that there is a terminal inside the empty inner rectangle region. We can shift some segments and flip some corners to align with the terminal such that splitting at this terminal will result in two smaller FSTs.

The *bottleneck Steiner distance*, which is analogous to that of the Steiner tree problem in networks, can also be used to eliminate useless rectilinear FSTs. Let $Tr(V)$ be a tree spanning a terminal set V. We use $\delta_{Tr(v_i v_j)}$ to denote the length of the longest edge on the unique path between v_i and v_j in $Tr(V)$. Let RMST(V) be a RMST of the terminal set V, then the bottleneck Steiner distance is equal to $\delta_{RMST(v_i v_j)}$. It can be proved that if RMST(V) and RSMT(V) are respectively a minimum spanning tree and a Steiner minimal tree on a set of vertices V, then $\delta_{RMST(v_i v_j)} \geq \delta_{RSMT(v_i v_j)}$ for any $v_i, v_j \in V$. Therefore, for a FST to be part of a RSMT, we require that $\delta_{RMST(v_i v_j)} \geq \delta_{FST(v_i v_j)}$ for any $v_i, v_j \in V$.

The above conditions are used to prune away those FSTs that cannot be part of any RSMT. Empirical study showed that most of the FSTs can be pruned away by one of these tests and the number of resulting FSTs grows almost linear with respect to the number of terminals. The algorithm is able to generate FSTs for 1000 terminals in less than a minute.

2.1.2.2 FST concatenation

Let $F = \{f_1, f_2, \ldots, f_m\}$ be the set of FSTs generated in the first phase. The second phase is to select a subset such that all terminals are spanned. Different from the FST generation phase, the FST concatenation phase is purely combinatorial and metric-independent. Therefore, early FST concatenation algorithms proposed for the Euclidean Steiner minimum tree (ESMT) problem can also be applied for the

rectilinear case. These approaches include backtrack search, dynamic programming, and integer linear programming.

Backtrack search

A straightforward way to combine FSTs is to use backtrack search. Starting from a single FST, recursively add new FSTs into the solution until the solution spans all terminals or it can be verified that the solution cannot be optimal. In these cases, the search backtracks to try to add some other FSTs.

Winter [50] proposed the first FST concatenation algorithm by backtrack search for the ESMT problem. Simple tests such as length tests, degree tests, and cycle tests are employed during the search. The algorithm is able to solve, in a reasonable amount of time, problems with less than or equal to 15 terminals. Experimental results showed that, for the instances with more than 15 terminals, the computation time of the concatenation phase dominates that of the generation phase. Cockayne and Hewgill [11, 12] presented an improved version of Winter's algorithm. Problem decomposition is applied to divide the initial concatenation problem into several sub-problems. If the set of all FSTs can be divided into biconnected components, then each biconnected component corresponds to a subproblem on which concatenation can be done separately. They also proposed to use an incompatibility matrix to speedup the search. Two FSTs are incompatible if they cannot appear simultaneously in any of the SMTs (e.g. if they have more than one terminal in common, a cycle will be formed). This information is pre-computed and stored in a matrix. The incompatibility matrix can be used to guide the backtrack search. For example, only the FSTs that are compatible with every FST in the current solution can be added. This can significantly reduce the solution space with almost no computational overhead. In comparison with the savings in searching, the time required for computation of the incompatibility matrix is negligible. They reported

a solvable range of 32 terminals. Salowe and Warme [45] proposed to select and add "the most promising" FST during the search. They also gave a more powerful graph decomposition theorem to decompose the problem. More recently, Winter and Zachariasen [51] improved FST compatibility and FST pruning substantially and report solutions for 140-terminal instances in the Euclidean space.

Dynamic programming

Ganley and Cohoon [30] presented a dynamic programming approach to combine FSTs. From Theorem 2.3, it is clear that any RSMT for any set of terminals is either a FST itself or it can be divided into two smaller RSMTs joining at a terminal. Therefore, dynamic programming is applicable. Subsets of terminals are processed in increasing order of their cardinality. For subsets of more than two terminals, the algorithm first tries to construct a FST according to Theorem 2.3. Then, several trees are produced by joining the RSMTs of every pair of disjoint subsets having exactly one terminal in common. Since the subsets are enumerated in increasing order of cardinality, the RSMTs of the smaller subsets are already computed and stored. Among all the generated trees, the one with minimum length is remembered in a lookup table. The time complexity of this algorithm is $O(n3^n)$. By proving that the number of candidate FSTs for a set of n terminals is at most $O(n1.62^n)$, Ganley and Cohoon improved the time complexity of the algorithm to $O(n^2 2.62^n)$. Based on this dynamic programming algorithm, Fößmeier and Kaufmann [17] make use of the empty region properties to reduce the number of candidate FSTs. An $O(n1.38^n)$ bound is derived which lead to an algorithm with $O(n^2 2.38^n)$ time complexity.

Although dynamic programming algorithms can provide the best theoretical worst-case time bound, their practical performance are inferior to the backtrack search.

Integer linear programming

Despite the substantial efforts made to improve the performance, backtrack search and dynamic programming algorithms can only handle problems with around 100 terminals. A breakthrough in the concatenation algorithm is achieved by Warme [49, 14] who observed that the FST concatenation problem is equivalent to find a minimum spanning tree in hypergraph and formulated the problem as an integer linear programming (ILP).

Let V be the set of terminals to be connected and n be the number of terminals in the set. Let m be the number of FSTs generated in the first phase, i.e. the number of FSTs in F. Each FST $f_i \in F$ is associated with a binary variable x_i indicating whether f_i is taken as a part of the RSMT. We use $|f_i|$ to denote the size of f_i, i.e., the number of terminals connected by f_i, and use l_i to denote the length of f_i. In the following, $(A:B)$ means $\{f_i \in F : f_i \cap A \neq \varnothing \wedge f_i \cap B \neq \varnothing\}$. The ILP formulation is as follows.

Minimize:

$$\sum_{i=1}^{m} l_i \times x_i. \tag{2.2}$$

Subject to:

$$\sum_{i=1}^{m} x_i \times (|f_i| - 1) = n - 1, \tag{2.3}$$

$$\sum_{i:\, f_i \in (X:V-X)} x_i \geq 1 \quad \forall X \subset V, \tag{2.4}$$

$$\sum_{i:\, f_i \cap X \neq \varnothing} x_i \times (|f_i \cap X| - 1) \leq |X| - 1 \quad \forall X \subset V \wedge |X| \geq 2. \tag{2.5}$$

In the ILP, the objective function (2.2) is to minimize the total length of selected FSTs. Constraint (2.3) is the *total degree constraint* that requires the right number of FSTs in order to span V. Constraints (2.4) are the *cutset constraints*.

The constraints ensure that for any cut $(X : V - X)$ of the terminal set, there should be at least one selected FST to connect them. Constraints (2.5) are the *subtour elimination constraints* that eliminate any cycle in the solution. Since there is an exponential number of cutset constraints and subtour elimination constraints, they are considered in an incremental way and the ILP is solved by a branch-and-cut algorithm with the lower bound provided by linear programming (LP) relaxation, i.e., by relaxing integrality of variable x_i to $0 \leq x_i \leq 1$. At the beginning of the algorithm, only some simple constraints are considered. Other constraints are added by separation methods. The separation problems can be solved in polynomial time by finding minimum cuts in some graphs. It is shown in [14] that Warme's FST concatenation algorithm combined with Zachariasen's FST generation algorithm can solve instacnes with as many as 2000 terminals in a reasonable amount of time.

More recently, Polzin and Daneshmand [41] presented a efficient alternative for the concatenation phase. The set of FSTs are further decomposed into a set of edges. An algorithm which is originally designed for general graphs can then be applied to construct a RSMT. Polzin and Daneshmand showed that their algorithm, in most cases, is faster than Warme's algorithm. They claimed that the superiority is due to the sophisticated reduction techniques they developed to reduce the size of the problem instance.

2.2 OARSMT algorithms

2.2.1 Heuristics

Since the OARSMT problem is NP-complete, most of the previous works have been focused on the development of heuristics. These heuristics can be generally classified into three categories, namely sequential approach, maze-routing based

approach, and connection graph based approach.

2.2.1.1 Sequential approach

The sequential approach, also called the construction-by-correction approach, consists of two steps. In the first step, a RSMT is constructed without considering any of the obstacles. This step can be done by using any of the aforementioned RSMT algorithms. In the second step, edges that overlap with obstacles are found and replaced by edges going around the obstacles. Generally, a simple line sweep technique can be applied. Yang *et al.* [56] proposed a complicated 4-step heuristics to remove the overlaps in the second step. The sequential approach is popular in industry due to its simplicity and efficiency. However, this approach usually cannot provide solution with good quality because it lacks a global view of the obstacles.

2.2.1.2 Maze-routing based approach

The maze-routing approach is originally proposed by Lee [10] for making connection between two points. Since then, several multi-terminal variants have been proposed. Despite early works that will incur unsatisfiable solution quality, recent developments on maze-routing demonstrate its effectiveness on the OARSMT problem. Hentschke *et al.* [42] presented AMAZE, a fast maze-routing based algorithm to build Steiner trees. The algorithm starts from a particular terminal and grow the tree by connecting one terminal at a time by using A* search. Li and Young [35] proposed another maze-routing based approach for the OARSMT problem. Similar to Hentschke's algorithm, during the construction of the tree, terminals are added one by one to the existing tree. The key difference is that, in the work by Li and Young, instead of adding only one path between terminals, multiple paths will be kept and the path selection is delayed until all the terminals are reached. During

this process, a number of candidate Steiner points can be generated. A MST is then constructed to connect all the Steiner points and the terminals. By deleting dangling Steiner points, an OARSMT can be obtained. Although this approach can provide solutions with high quality, the space and time complexities are relatively high which limit its applications to large scale problems. Recently, Liu *et al.* [6] extended Li's work by using a linear-space rectilinear graph. They showed that the proposed graph contains satisfactory Steiner point candidates and is also much simpler than the extended Hanan grid. The experimental results demonstrated a very competitive performance of the algorithm in both solution quality and run time.

2.2.1.3 Connection graph based approach

Most of the recent approaches on the OARSMT problem are graph based algorithms where an OARSMT is built based on a connection graph (not necessary rectilinear) that captures the global blockage information. Shi *et al.* [55] proposed to use the global routing graph which contains the escape graph as its subgraph as the connection graph. They developed a circuit simulation- based technique to build the OARSMTs. Feng *et al.* [59] proposed an $O(n\log n)$ algorithm to construct OARSMTs in a graph called obstacle-avoiding constrained Delaunay triangulation. Shen *et al.* [60] proposed to use the obstacle-avoiding spanning graph. The obstacle-avoiding spanning graph can be formed by making connections between terminals and obstacle corners. The authors showed that the graph contains only $O(n)$ edges and is much simpler than the escape graph. A MST in the graph can be easily found in $O(n\log n)$ time. The OARSMT can then be generated by rectilinearizing the MST. They showed that the proposed spanning graph can always produce a RSMT with good quality. The worst case time complexity of the algorithm is $\Omega(n^2\log n)$. Lin *et al.* [9] extended Shen's approach by identifying

Figure 2.13: An example of FST in the presence of an obstacle.

many “essential” edges that can lead to more desirable solutions in the construction of the obstacle-avoiding spanning graph. They proved that their algorithm guarantees to find optimal OARSMT for any 2-pin nets. For higher-pin net, their algorithm is able to find solutions with better quality. However, the number of edges, in the worst case, is increased to $O(n^2)$. Therefore, the time complexity of their algorithm is $O(n^3)$. Long *et al.* [33] presented an efficient $O(n\log n)$ four-step algorithm to construct an OARSMT. They proposed a more sparse graph and efficient local and global refinements were used to improve the solution quality. Liu *et al.* [7] proposed another $O(n\log n)$ algorithm based on the generation of critical paths. Recently, Ajwani *et al.* [18] presented the FOARS, a FLUTE-based top down approach for the OARSMT problem. They apply the obstacle avoiding spanning graph to partition the problem and construct the OARSMT by using the obstacle-aware version of FLUTE. The time complexity of their algorithm is also $O(n\log n)$.

2.2.2 Exact algorithms

In comparison with heuristics, there has been relatively less research on exact algoriths for the OARSMT problem. Maze-routing [10] can give optimal solutions to two-terminal instances. Along with the escape graph, Ganley and Cohoon [31] presented a topology enumeration scheme to construct optimal three-terminal and four-terminal OARSMTs.

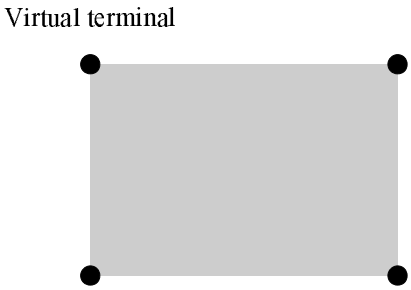


Figure 2.14: Locations of virtual terminals of an obstacle.

For multi-terminal instances, a natural idea is to make use of the two-phase exact algorithm (i.e. generate FSTs in the first phase and then concatenate them in the second phase) which is originally proposed for the RSMT problem. However, this algorithm cannot be directly applied when obstacles exist in the plane. An example is shown in Fig. 2.13. In the absence of obstacles, a FST has a topology, as characterized by Hwang, that consists of a backbone and alternating incident segments connecting the terminals. In contrast, the structures of FSTs in the presence of obstacles can be very different. Therefore, the construction of FSTs in the presence of obstacles can itself be a difficult problem that limits the application of the two-phase algorithm for the OARSMT problem.

Li *et al.* [36, 48] presented a pioneer work to extend the two-phase approach to solve the OARSMT problem. The key observation is that, by adding the so-called *virtual terminals*, the structures of FSTs can be greatly simplified. For each obstacle, four virtual terminals are added to its four corners as shown in Fig. 2.14. We use T to denote the set of virtual terminals added. The direct impact of adding virtual terminals is that FSTs can be further decomposed into smaller FSTs by splitting at these virtual terminals. In Fig. 2.15, the FST can be decomposed into a set of five smaller FSTs each of which is of simple structure. These smaller FSTs are called FSTs with blockages.

Let t be a rectilinear Steiner tree. A tree t' is equivalent to t if and only if t'

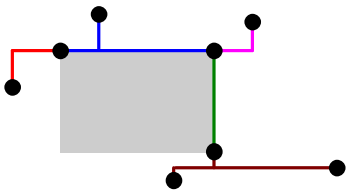


Figure 2.15: Decomposition of a FST.

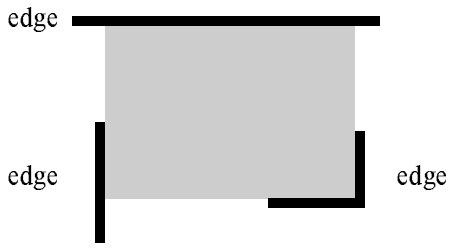


Figure 2.16: Forbidden edges in a FST with blockages.

can be obtained from t by shifting or flipping some edges which have no nodes on them. With the concept of equivalent trees, a FST with blockage f over a set of terminals $T_f \subseteq (V + T)$ can be defined as follows:

1. f is an OARSMT over T_f;
2. every terminal in T_f has degree one in f and in all its equivalent trees;
3. all the equivalent trees of f cannot contain forbidden edges as shown in Figure 2.16. (Otherwise, splitting can be done to further decompose the FST.)

With the definition, it can be easily verified that an OARSMT is a union of FSTs with blockages. An important theoretical result is that the structures of FSTs with blockages are the same as those of FSTs in the absence of obstacles. This indicates that, by adding virtual terminals, we can use the two-phase approach to construct an OARSMT efficiently. In the first phase, we generate a sufficient set of FSTs with blockages. In the second phase, we identify and combine a subset of FSTs with minimum total length such that all real terminals are interconnected. For simplicity, we will use FSTs to denote FSTs with blockages in the following.

To generate FSTs with more than two terminals, a modified version of the Zachariasen's algorithm [61] is used. To generate FSTs with exactly two terminals, a more efficient way is proposed. For the FST concatenation phase, it can be formulated as an ILP. Warme's branch-and-cut algorithm [49] is extended to solve the ILP. Experimental results showed that the proposed method is able to handle problems with hundreds of terminals in the presence of multiple obstacles, generating optimal solution in a reasonable amount of time. However, the performance is severely affected by the number of obstacles and all the solvable test cases contain less than one hundred obstacles. Moreover, the algorithm can only handle rectangular obstacles.

CHAPTER 3

ObSteiner - an exact OARSMT algorithm

Contents

3.1 Introduction

In this chapter, we study the OARSMT problem. In recent years, many heuristics have been proposed for the OARSMT problem. On the other hand, only few exact algorithms have been proposed. The state-of-the-art presented in [36] and [48] extended GeoSteiner [14] to an obstacle-aware version. Their algorithms are able to generate optimal OARSMTs for multi-terminal nets in the presence of rectangular obstacles. However, these approaches cannot be applied when there are complex rectilinear obstacles in the routing region, as is often the case in the routing problem. Moreover, their algorithms can only handle benchmarks with less than one hundred obstacles, while modern VLSI design may contain over one thousand obstacles. To the best of our knowledge, no previous algorithm can generate optimal solutions to the OARSMT problem with a large number of terminals among complex rectilinear obstacles. Although the escape graph model can transform the OARSMT problem into a graph problem which can be solved optimally by using some graph based algorithms [41, 15], these approaches are believed to be less efficient than the geometric approaches for solving the geometric Steiner tree problem[1]. An example is GeoSteiner [14] that remains to be the most efficient approach to solve the RSMT problem when no obstacle exists. Therefore, it is necessary to develop an efficient exact algorithm that allows the presence of complex obstacles. The aim of this chapter is to propose an algorithm called ObSteiner to construct OARSMTs among rectilinear obstacles of both convex and concave shapes. To generate OARSMTs, we first study the full Steiner trees (FSTs) among com-

[1]Standard benchmarks for the Steiner tree problem in graphs also include rectilinear graphs which correspond to the RSMT problems. When solving these problems, most of the algorithms [41, 15] will preprocess them by using the first phase of GeoSteiner [14] to reduce the problem size. Otherwise, the problem will be much more difficult to solve. This is mainly because the algorithms for Steiner tree problem in graphs cannot exploit the geometric of the RSMT problem.

plex obstacles and verify how their structures can be simplified by adding virtual terminals. We then propose an iterative two-phase approach to construct optimal OARSMTs based on GeoSteiner.

The rest of this chapter is organized as follows. In Section 3.2, we give preliminaries on the OARSMT problem and an exact algorithm for the RSMT problem. In Section 3.3, we study the structures of FSTs among complex obstacles. Section 3.4 and 3.5 describe the proposed exact algorithm in detail. Finally, experiment results are presented in Section 3.6.

3.2 Preliminaries

3.2.1 OARSMT problem formulation

In this problem, we are given a set V of terminals and a set O of obstacles. An obstacle is a rectilinear polygon. All edges of an obstacle are either horizontal or vertical. Rectilinear polygons can be classified into two types: convex polygons and concave polygons. A rectilinear polygon is a *convex rectilinear polygon* if any two points in the polygon have a shortest Manhattan path lying inside the polygon. Otherwise, it is called a *concave rectilinear polygon*.

As shown in Figure 3.1, a *corner* of an obstacle is the meeting point of two neighboring edges. If the two neighboring edges of a corner form a 90 degree angle inside the polygon, the corner is called a *convex corner*. Otherwise, if the two neighboring edges of a corner form a 270 degree angle inside the polygon, the corner is called a *concave corner*. If both end points of an edge are convex corners, this edge is called an *essential edge* (e.g. Fig. 3.1). Note that the essential edge defined in this chapter is also known as extreme edge in [53, 5]. However, the way we make use of essential edges is very different.

A terminal cannot be located inside an obstacle, but it can be at the corner or on the edge of an obstacle. The OARSMT problem asks for a rectilinear Steiner tree with minimum total length that connects all terminals. No edge in the tree can intersect with any obstacle, but it can be point-touched at a corner or line-touched on an edge of an obstacle. This tree is known as an OARSMT.

In the following figures, we use a solid circle to denote a terminal and an empty circle to denote a Steiner point.

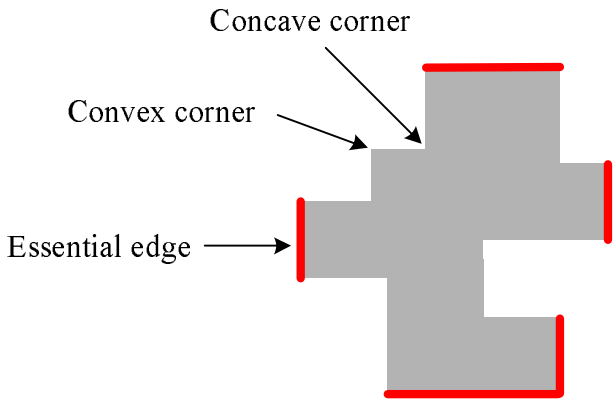


Figure 3.1: Corners and essential edges of an obstacle.

3.2.2 An exact RSMT algorithm

The RSMT problem in the absence of obstacles has been studied excessively over years [16]. Among various approaches, GeoSteiner [14] is the most efficient exact algorithm in practice. The algorithm is developed based on the construction of full Steiner trees (FSTs). A FST is a rectilinear Steiner minimum tree in which every terminal is a leaf node (i.e. of degree one). In the absence of obstacles, it is proved in [26] that a FST has one of the two generic forms as shown in Fig. 3.2, consisting of a backbone and alternating incident legs connecting the terminals. A folk theorem states that any RSMT can be decomposed into a set of edge-disjoint FSTs by splitting at terminals with degree more than one. Since FSTs are much

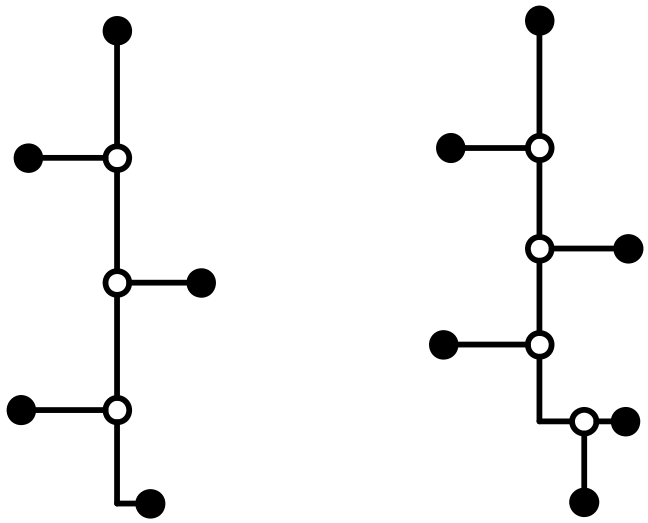


Figure 3.2: FST structures in the absence of obstacles. (a) Type I structure. (b) Type II structure.

easier to construct than RSMTs, most of the exact algorithms for the construction of a RSMT will first generate its FST components. GeoSteiner makes use of a two-phase approach, consisting of a FST generation phase and a FST concatenation phase, to construct a RSMT. In the first phase, a set of FSTs are generated such that there is at least one RSMT composed of the FSTs in the set only. In the second phase, a subset of FSTs are selected and combined to form a RSMT. The key observation is that the FST concatenation problem is equivalent to the spanning tree in hypergraph problem and can be formulated as an integer linear programming. On the rectilinear plane, GeoSteiner remains the fastest exact algorithm for the RSMT problem, but it cannot be applied when obstacles exist in the routing plane.

3.3 OARSMT decomposition

The exact algorithm for the RSMT problem indicates the importance of studying the structures of FSTs when there are obstacles in the routing region. However, the structures of these FSTs can be complicated due to the existence of rectilinear obstacles. We will show in this section how we can simplify the FST structures in the presence of complex obstacles by adding the so called virtual terminals. In addition, we will propose a new simple graph model that contains at least one optimal solution for the OARSMT problem. This section gives the theoretical foundations for the exact algorithm for the OARSMT problem.

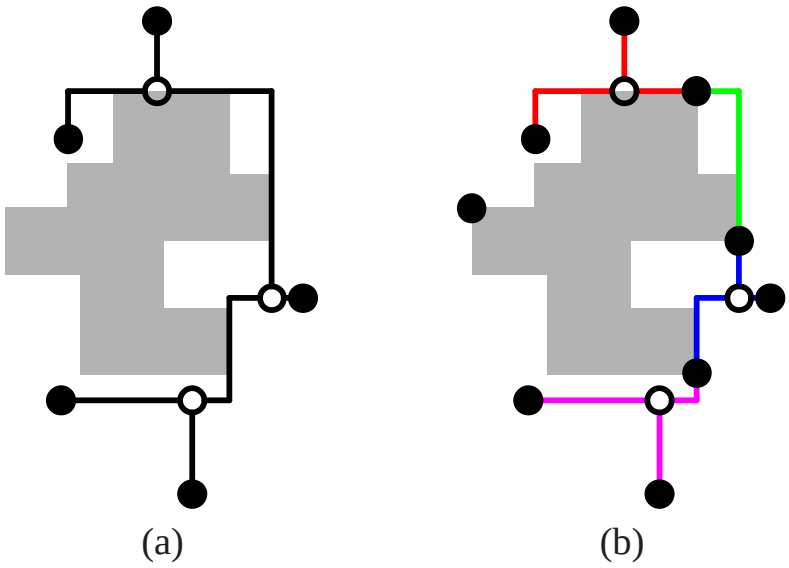


Figure 3.3: (a) A FST structure in the presence of obsatcles. (b) Decomposition of FST after adding virtual terminals.

3.3.1 Full Steiner trees among complex obstacles

To construct OARSMTs among complex obstacles, we first study the FSTs in the presence of complex obstacles. An example of such a FST in the presence of one

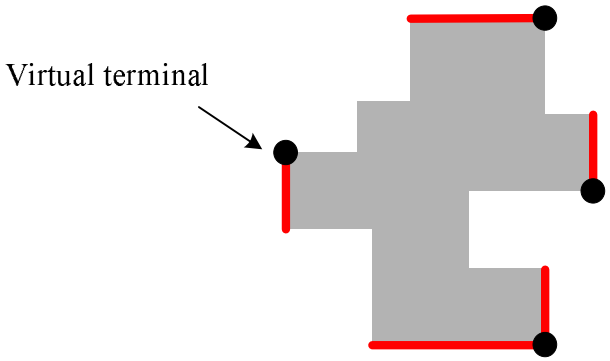
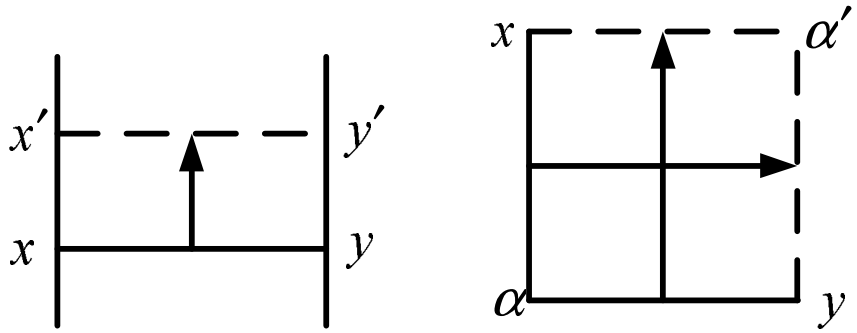


Figure 3.4: An example of adding virtual terminals.

obstacle is shown in Fig. 3.3(a). As we can observe from the figure, the structure of FSTs in the presence of obstacles can be very complicated. In such a case, the construction of FSTs can itself be a hard problem which limits the application of the two-phase approach to the OARSMT problem. Therefore, virtual terminals are added to simplify their structures in our approach. These terminals are called virtual because they can be connected by the OARSMT or not. It should be noted that although virtual terminals are also used in [36, 48], there are critical differences when dealing with rectilinear obstacles. In this work, the virtual terminals are added in such a way that there is at least one virtual terminal on every essential edge of all the obstacles. This is a simplified but sufficient way of adding virtual terminals in comparison with those in [36, 48]. Note that the location of a virtual terminal on an essential edge is not restricted. It will not affect the optimality of the solution. For simplicities, in the following proofs, we assume that the virtual terminal on an essential edge is located at one of its end points. An example is shown in Fig. 3.4. Note that for two essential edges sharing a common endpoint at a corner, we only need to add one virtual terminal at that corner. We use U to denote the set of virtual terminals we added. The direct impact of adding virtual terminal is that we can further decompose the complicated FSTs (e.g. Fig. 3.3(a))

into smaller and simpler FSTs by splitting at the virtual terminals (e.g. Fig. 3.3(b)). We call these smaller trees *FSTs among complex obstacles*. In the following, we will give a formal definition to the FSTs among complex obstacles and prove that they will follow some very simple structures.



(a) (b)

Figure 3.5: Two operations on a rectilinear Steiner tree. (a) Shifting and (b) Flipping.

To define FSTs among complex obstacles, we introduce two operations as follows. As defined in [26], there are two basic operations on a tree that will not change the total length: *shiftings* and *flippings*, as shown in Fig. 3.5. Shifting a line means moving a line between two parallel lines to a new position. Flipping a corner means moving the two perpendicular lines of the corner so as to move the corner to the opposite position diagonally. A rectilinear Steiner tree t is equivalent to another tree t' if and only if t can be obtained from t' by flipping and shifting some lines that have no node on them. With these two operations, a FST among complex obstacles can be defined as follows.

Definition 3.1. *A FST f over a set $V_f \subseteq V + U$ of terminals is an OARSMT of V_f such that every terminal $v \in V_f$ is a leaf node in f and in all its equivalent trees. Moreover, all the equivalent trees of f cannot contain forbidden edges. A forbidden edge is an edge that passes through a virtual terminal. If a FST f or its equivalent*

trees contain forbidden edges, we can split this FST into smaller FSTs at this virtual terminal.

Note that the definition of FST in this chapter is similar to the definitions in [36, 48]. However, the obstacles considered in this work are rectilinear polygons, which are more general and complicated than the rectangles considered in [36, 48]. In the following, we use FSTs to refer FSTs among complex obstacles for simplicities.

To derive the structures of a FST, we mainly follow the steps as described in [26] and [48]. The main difference is that there can be rectilinear obstacles in the routing region. For the two operations (i.e. shiftings and flippings) used in the proofs, it is possible that some of the operations cannot be performed due to obstacles. We will show in the following how this problem can be solved by adding virtual terminals.

The notations we are going to use in this section are the same as in [26]. A vertex can be a *node* (real terminal or virtual terminal) or a *Steiner point*. An edge between two vertices is a sequence of alternating vertical and horizontal lines and each turning point is a *corner*. A line has only one direction but may contain a number of vertices on it. V_{xu} (V_{xd}) denotes the maximal vertical line at point x which is above (below) x excluding x itself. Similarly H_{xr} (H_{xl}) denotes the maximal horizontal line at point x which is on the right (left) of x excluding x itself. If a line ends at a node and contains no other vertices, we call it a *node line*. If it ends at a corner and contains no vertices, we call it a *corner line*. In the following figures for the proofs, we use an empty circle to represent a Steiner point and an solid circle to represent a node.

Lemma 3.1. *All Steiner points in a FST either have degree three or degree four.*

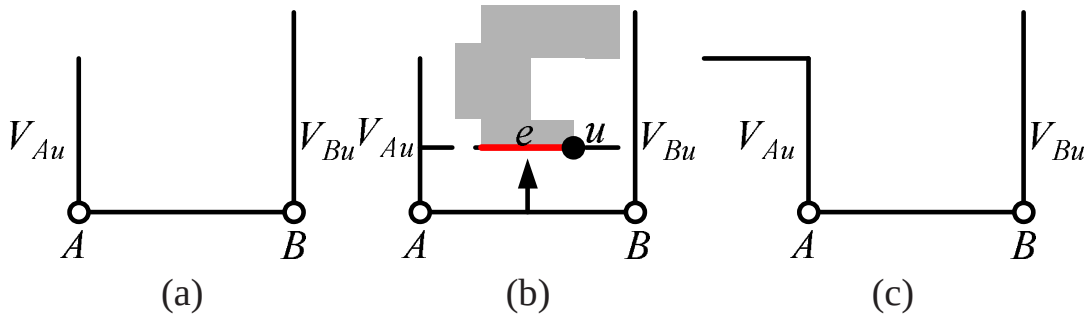


Figure 3.6: A structure of two neighboring Steiner points when both V_{Au} and V_{Bu} exist and $|V_{Bu}| \geq |V_{Au}|$. (a) In the absence of obstacles. (b) In the presence of an obstacle. (c) The resulting structure of Lemma 3.2.

Lemma 3.2. *Let A and B be two adjacent Steiner points in a FST. Suppose that AB is a horizontal line and both V_{Au}, V_{Bu} exist. Then $|V_{Bu}| \geq |V_{Au}|$ implies that V_{Au} is a line that ends at a corner turning away from V_{Bu}.*

Proof. See Fig. 3.6(a). Suppose *A* is to the left of *B*.

(i) V_{Au} contains no terminal at its end point, for otherwise we can shift *AB* to that terminal and obtain an equivalent tree in which a terminal has degree more than one. If the line *AB* cannot be shifted due to some obstacles as shown in Fig. 3.6(b), we can shift *AB* up until it overlaps with an edge *e* of the obstacle. According to the definition, since the two endpoints of *e* are convex corners, *e* is an essential edge. Let *u* be the virtual terminal added on *e*. As a result, *AB* will pass through *u* and thus is a forbidden edge, which is a contradiction to the definition of FSTs.

(ii) No Steiner points on V_{Au} can have a line going right, for otherwise we can replace *AB* by extending that line to meet V_{Bu} and reduce the total length. If the line cannot be extended due to obstacles, we can repeat the operation described in the previous step and result in a contradiction.

(iii) Therefore, the upper endpoint of V_{Au} cannot be a Steiner point since it has no lines going right or upward, hence it must be a corner turning left, as shown in

Fig. 3.6 (c).

(iv) V_{Au} can contain no Steiner point, for let C be such a Steiner point which is nearest to the corner point. Since H_{Cr} does not exist, H_{Cl} must exist. We can then shift the line between point C and the corner point to the left to reduce the total length, a absurdity. If the line cannot be shifted due to some obstacles (this line overlaps with an edge of the obstacle and this edge must be a essential edge), the line will pass through a virtual terminal, an absurdity. □

Corollary: *Suppose V_{Bu} contains a vertex, then V_{Au} is a corner line that ends at a corner turning away from V_{Bu} and $|V_{Au}| < |V_{Bu}|$.*

Proof. By Lemma3.2, if $|V_{Au}| \geq |V_{Bu}|$, V_{Bu} must be a corner line and can have no vertex on it. Therefore, $|V_{Au}| < |V_{Bu}|$. Again from Lemma3.2, V_{Au} is a corner line that ends at a corner turning away from V_{Bu}. □

Lemma 3.3. *Suppose V_{xu} (where x is a vertex) is a corner line ends at a corner turning left (right), then H_{xl} (H_{xr}) does not exist.*

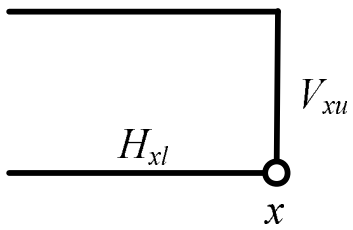


Figure 3.7: The structure when V_{xu} is a corner line ended at a left-turn corner and H_{xl} exists.

Proof. See Fig. 3.7. If H_{xl} exists, we can shift the line V_{xu} to the left and reduce the total length. If the line cannot be shifted due to some obstacles, the tree will contain a forbidden edge that passes through a virtual terminal, a violation of the FST definition. □

Lemma 3.4. *No Steiner point can have more than one corner line.*

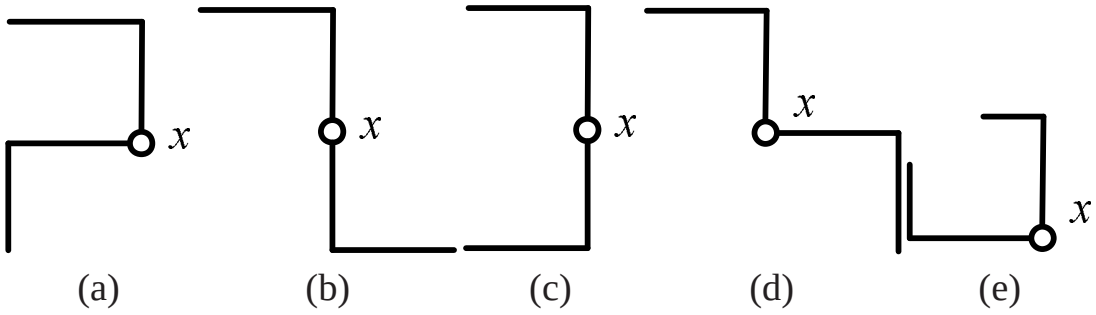


Figure 3.8: Five possible structures when a Steiner point has two corner lines.

Proof. Consider a Steiner point x with two corner lines. Without loss of generality, we assume V_{xu} exists and ends at a corner turning left. The second corner line can be V_{xd}, H_{xl}, or H_{xr} and ends at a corner turning to two different directions. The case when H_{xr} exists and ends at a corner turning up is equivalent to the case when H_{xl} exists and ends at a corner turning down, and thus can be removed. Therefore, there are totally five possible cases as shown in Fig. 3.8. Fig. 3.8(a) and (e) cannot exist according to Lemma 3.3. Fig. 3.8(b) and (d) is impossible because the third line at the Steiner point cannot exist by Lemma 3.3. Considering Fig. 3.8(c), by Lemma 3.3 H_{xl} cannot exist, and therefore H_{xr} must exist. We can shift the horizontal line containing x to the left to reduce the wire length, an absurdity. If the line cannot be shifted due to some obstacles, the tree will contain a forbidden edge that passes through a virtual terminal, a violation of the FST definition. □

Lemma 3.5. *If f is a FST, the Steiner points in f form a chain.*

Proof. First of all, if f is a FST, the Steiner points in f are connected, for otherwise some Steiner points have to be connected by terminals of degree two or more. Therefore we only need to prove that no Steiner point in f is adjacent to more than

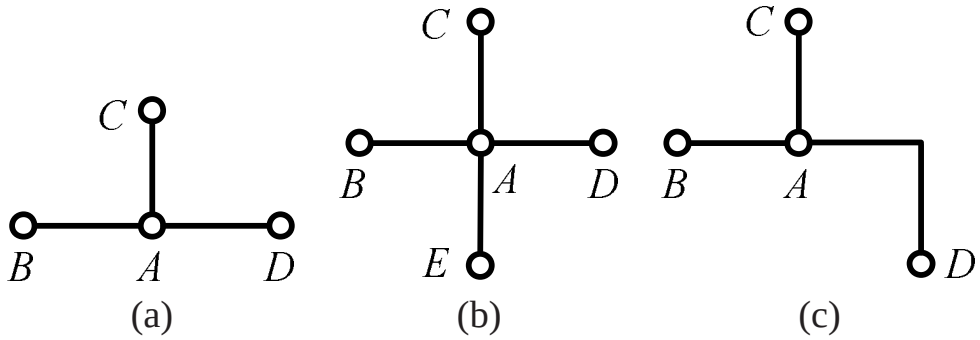


Figure 3.9: Three possible structures when a Steiner point is adjacent to more than two other Steiner points.

two other Steiner points. Suppose the contrary, and let A be such a Steiner point. Then from Lemma 3.3 and Lemma 3.4, the connection between A and its adjacent Steiner points must be in one of the three forms as shown in Fig. 3.9.

First, consider Fig. 3.9(a) and Fig. 3.9(b). Suppose H_{Cl} exists. Then, from the corollary of Lemma 3.2, H_{Cl} must be a corner line that ends at a corner turning up. Similarly, if H_{Cr} exists, it is also a corner line ends at a corner turning up. Since C is a Steiner point, at least two of the three lines H_{Cl}, H_{Cr} and V_{Cu} must exist. However, regardless of which two (or all three) exist, we end up a contradiction to either Lemma 3.3 or Lemma 3.4.

Next, consider Fig. 3.9(c). The argument on H_{Cl} is the same as that in Fig. 3.9(a) and Fig. 3.9(b). If H_{Cr} exists and $|H_{Cr}| \leq |H_{Ar}|$, the argument on H_{Cr} is again the same. Thus, we only need to discuss the case that $|H_{Cr}| > |H_{Ar}|$ (see Fig. 3.10).

Let α be the corner on the edge connecting A and D. Shift AC to α and let the new line meet H_{Cr} at β. Now, the tree contains a Steiner point α that is adjacent to three other Steiner points β, B and D in the form of Fig. 3.9(a), which has already been shown to be impossible. If AC cannot be shifted due to some obstacles, we can shift AC to the boundary of the obstacle and achieve an equivalent tree with

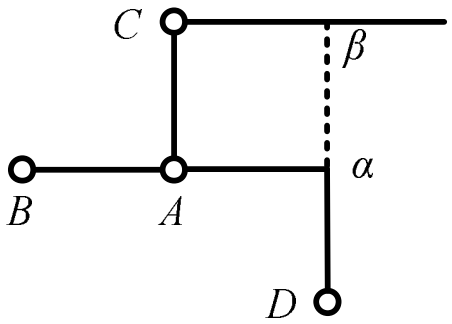


Figure 3.10: Special structure of one Steiner point with more than two neighboring Steiner points.

forbidden edges, a contradiction to the assumption that f is a FST. □

We call the chain of Steiner points the *Steiner chain*.

Lemma 3.6. *Suppose f is a FST. Then its Steiner chain cannot contain the subgraph shown in Fig. 3.11.*

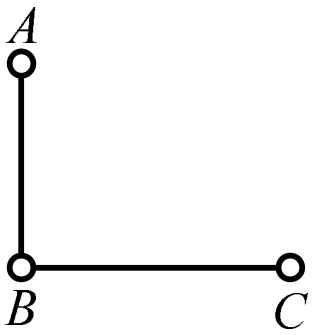


Figure 3.11: An impossible Steiner chain structure in a FST.

Proof. Suppose H_{Ar} exists. Then from the corollary of Lemma 3.2, H_{Ar} is a corner line. Since H_{Ar} and V_{Au} cannot both exist by Lemma 3.3, H_{Al} must exists for A is a Steiner point. If H_{Ar} exists, we can simply shift AB to $\alpha\beta$, as shown in Fig. 3.12(b), and obtain a similar structure as Fig. 3.12(a). Therefore, in the following, we can just consider the case when V_{Au} exists.

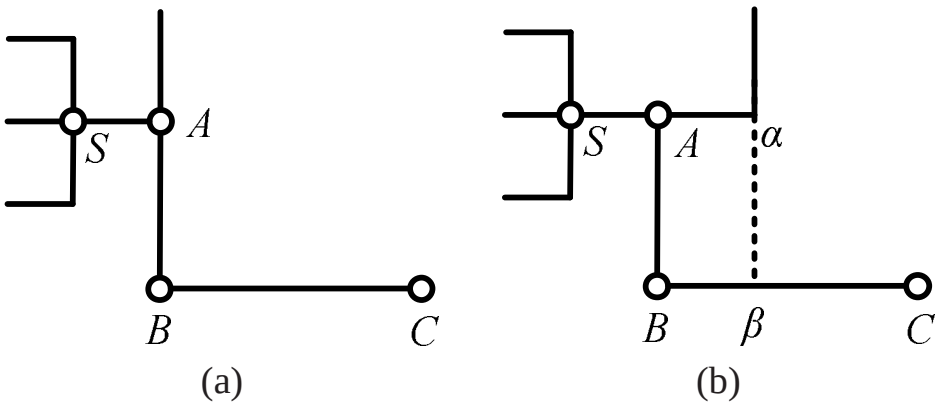


Figure 3.12: Two possible Steiner chain structures.

From Lemma 3.3, H_{Al} cannot be a corner line. Besides, H_{Al} cannot contain any Steiner points. If H_{Al} contains a Steiner point S, V_{Au} cannot contain any Steiner points by Lemma 3.5. If V_{Au} is a corner line, it must be a corner turning right by Lemma 3.3. We can shift the line between the corner and B to the right and obtain a similar structure as Fig. 3.12(a). Therefore, we can assume without loss of generality that V_{Au} is a node line. Since S is a Steiner point, two of the lines H_{Sl}, V_{Su} and V_{Sd} must exist. By the corollary of Lemma 3.2, V_{Su} and V_{Sd} must be corner lines and the corners must turn away from AB. As a result, by Lemma 3.3 and Lemma 3.4, H_{Al} cannot contain any Steiner point. Moreover, H_{Al} cannot contain more than one node for the tree is a FST. Therefore, H_{Al} is a node line. By symmetry, V_{Cd} exists and is a node line. Since B is a Steiner point, at least one of the lines H_{Bl} or V_{Bd} exists. We first assume that V_{Bd} exists and V_{Bl} does not exist. Since V_{Cd} contains a vertex (see Fig. 3.13), by the corollary of Lemma 3.2, V_{Bd} must be a corner line that ends at a corner (denoted by β) turning left and connects to a node b by Lemma 3.5. But this is impossible, for otherwise we can shift BC to $\beta\alpha$ to obtain a tree in which both H_{Al} and $H_{\beta l}$ are node lines, a contradiction to Lemma 3.2. If the line cannot be shifted due to some obstacles, the tree or its equivalent will

contain forbidden edges, an absurdity. Similarly, H_{Bl} cannot exist. As a result, B cannot be a Steiner point which is contradictory to the assumption. □

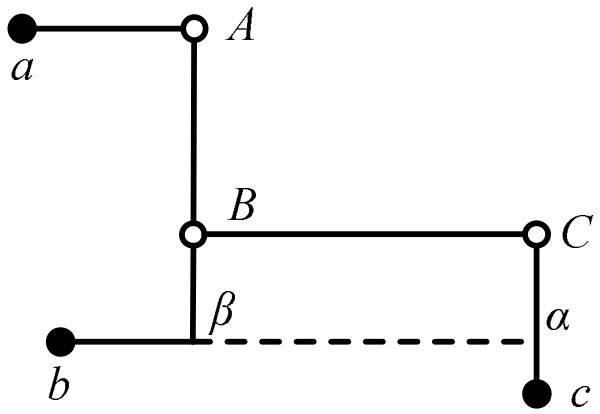


Figure 3.13: The topology when V_{Bd} exists.

Define a staircase to be a continuous path of alternating vertical lines and horizontal lines such that their projections on the vertical and horizontal axes have no overlaps.

Lemma 3.7. *Suppose f is a FST. The Steiner chain of f is then a staircase.*

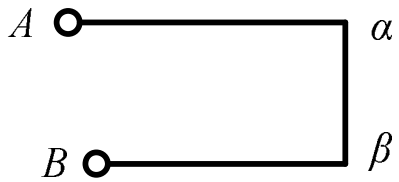


Figure 3.14: A structure of the Steiner chain when it bends back.

Proof. Suppose that the Steiner chain bends back as shown in Fig. 3.14, where A and B are Steiner points that are closest to the turning points α and β. There must be at least two Steiner points on αβ, for otherwise we can shift αβ to the left and reduce the total length. If the line cannot be shifted due to some obstacles, the tree will contain forbidden edges. From Lemma 3.6, neither α nor β can be a Steiner

point. From Lemma 3.3 and Lemma 3.5, the horizontal line of any Steiner point on $\alpha\beta$ must be a node line and the first one below $A\alpha$ must be a line going right. From the corollary of Lemma 3.2, the adjacent Steiner points on $\alpha\beta$ cannot have horizontal lines going in the same direction. Therefore, $\alpha\beta$ must have more left lines (including $A\alpha$ and $B\beta$) than right lines, which implies that we can shift $\alpha\beta$ to the left and reduce the total length, an absurdity. □

Lemma 3.8. *Suppose f is a FST. The Steiner chain of f cannot contain a corner with more than one Steiner points on the two neighboring lines.*

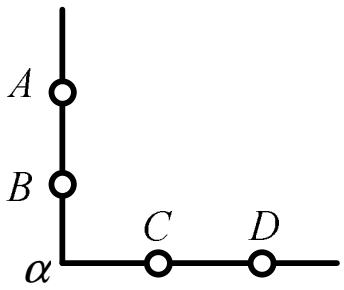


Figure 3.15: A corner with more than one Steiner point on each line.

Proof. Suppose to the contrary that f contains the subgraph shown in Fig. 3.15.

From Lemma 3.3, V_{Cu} does not exist. Thus, V_{Cd} exists and is a node line by Lemma 3.3 and Lemma 3.5. Suppose V_{Dd} exists. Then from the corollary of Lemma 3.2, V_{Dd} is a corner line. As a result, H_{Dr} does not exist by Lemma 3.3. Therefore V_{Dd} and H_{Dr} cannot both exist, and hence V_{Du} must exist and is a node line by Lemma 3.3 and Lemma 3.6. If $|V_{Du}| \leq |B\alpha|$, we can shift $D\alpha$ to the node on V_{Du} and obtain an tree in which a node has degree two. If the line cannot be shifted, the tree will contain forbidden edges. If $|V_{Du}| > |B\alpha|$, we can shift $D\alpha$ to B moving the Steiner point C to C'. But the induced subgraph between ABC' cannot exist by Lemma 3.6. Again, the line can be shifted, or else it cannot be a part of a FST. □

Lemma 3.9. *Suppose f is a FST. If the number of Steiner points is greater than two, either every vertical line on the Steiner chain contains more than one Steiner points (except possibly the first and the last vertical lines) and every horizontal line on the Steiner chain contains no Steiner point except at the end point, or vice versa.*

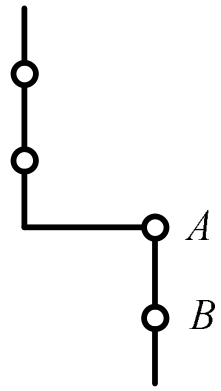


Figure 3.16: A possible structure of the Steiner chain.

Proof. By Lemma 3.4, a Steiner point cannot have two corner lines. Hence at least two of the first three Steiner points (counting from either end) are collinear. Without loss of generality, suppose the first collinearity occurs on a vertical line. Let A be the first Steiner point (if any) not on the vertical line. Then A is connected to its preceding Steiner point through a corner as shown in Fig. 3.16 by Lemma 3.6. Let B be the Steiner point (if any) succeeding A. Then A and B must be on the same vertical line, for otherwise either Lemma 3.8 is contradicted (if A and B are on the same horizontal line), or Lemma 3.4 is contradicted for A has two corner lines (if A and B are connected through a corner). If there are more Steiner points after B, we repeat the above argument to prove Lemma 3.9. □

Note that the structure of a FST is not affected by ninety degree rotation. In the following lemmas and theorems, we assume that if f is a FST, the corresponding Steiner chain will consist of a set of vertical lines and adjacent vertical lines are

connected by corners. We label the i^{th} Steiner point on the chain counting from above by A_i.

Lemma 3.10. *Suppose f is a FST. Every Steiner point on f must have a horizontal node line and the node lines alternate in the left-right direction.*

Proof. Note that a horizontal line not on the Steiner chain cannot contain any Steiner points, nor can it contain more than one node since f is a FST. Therefore it suffices to show that there exists a horizontal line and it cannot be a corner line.

(i) If the Steiner point is connected to its preceding Steiner point through a corner (A in Fig. 3.16), the third line of the Steiner point cannot be a vertical line according to Lemma 3.3. Therefore, the third line must be horizontal and it cannot be a corner line by Lemma 3.4.

(ii) If the Steiner point is on the vertical line of the Steiner chain (B in Fig. 3.16), the third line must be horizontal and it cannot be a corner line by Lemma 3.3.

By the corollary of Lemma 3.2, two adjacent Steiner points on the same line cannot have node lines on the same side. For the Steiner points connected through corner, it is also easy to prove this (Lemma 3.3). Hence, if f is a FST, the node line on the Steiner chain must alternate in the left-right direction. □

The proofs of the above lemmas are similar to those in [48], except that of Lemma 3.11 in which flippings are required. In this chapter, a different lemma is proposed.

Lemma 3.11. *Let A_i be the i^{th} Steiner point on the Steiner chain of a FST. A corner connecting A_i and A_{i+1} can be transferred to either one connecting A_{i-2} and A_{i-1}, or one connecting A_{i+2} and A_{i+3}, regardless of whether the place it transfers to has*

a corner of not. If the corner cannot be transferred due to obstacles, A_{i+3} is the last Steiner point or A_i is the first Steiner point on the chain (if A_{i+3} or A_i exist).

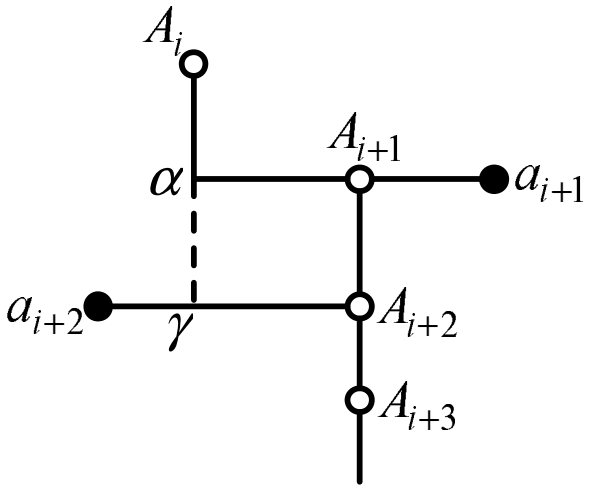


Figure 3.17: A structure of Steiner chain when A_i and A_{i+1} are connected by a corner.

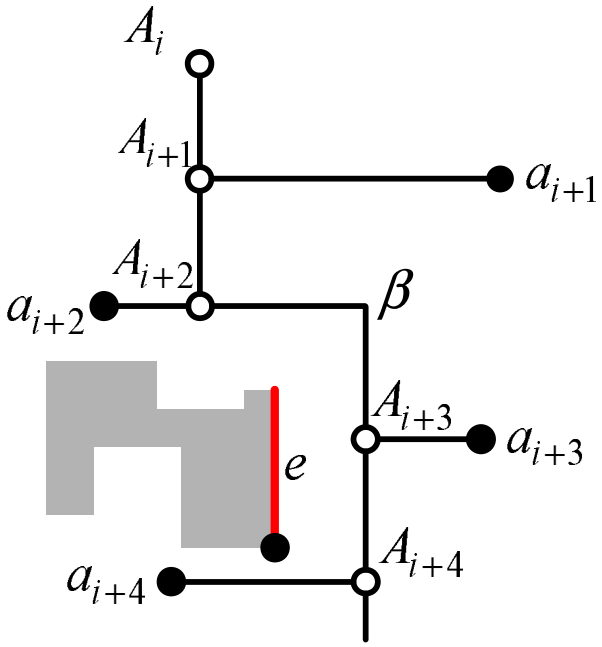


Figure 3.18: A structure of Steiner chain when the corner between A_{i+2} and A_{i+3} can not be flipped due to obstacles.

Proof. From Lemma 3.9 and Lemma 3.10, when A_i and A_{i+1} are connected by a corner, the graph must be the one given in Fig. 3.17. We use a_i to denote the node connected by A_i.

Necessarily $|a_{i+2}A_{i+2}| > |\alpha A_{i+1}|$, for otherwise we can shift $A_{i+1}A_{i+2}$ to a_{i+2} and obtain an equivalent tree in which a_{i+2} has degree two. Now shift $A_{i+1}A_{i+2}$ to α and suppose this line meets $a_{i+2}A_{i+2}$ at γ. Flip the corner A_{i+2} between γ and A_{i+3}. The corner connecting A_i and A_{i+1} is then transferred to one connecting A_{i+2} and A_{i+3}.

If the corner A_{i+2} cannot be flipped due to obstacles and A_{i+4} exists, the graph must be the one given in Fig. 3.18, in which there are obstacles inside the bounded rectangular region defined by A_{i+2} and A_{i+3}. We use β to denote the corner connecting A_{i+2} and A_{i+3}. We can shift βA_{i+4} to the left until it meets an edge e on one of the obstacle inside the rectangular region. Similar to the proof for Lemma 3.1, e is an essential edge and has a virtual terminal on it. Therefore, the FST has an equivalent tree that passes through a virtual terminal, a contradiction. If shifting βA_{i+4} meets a_{i+4} first, the FST has an equivalent tree in which a_{i+4} has degree two, again a contradiction. As a result, if the corner cannot be transferred due to obstacles, A_{i+3} is the last Steiner point on the chain if it exists. Similarly, we can transfer the corner to one connecting A_{i-2} and A_{i-1}. If the corner cannot be transferred, A_i is the first Steiner point on the chain if it exists.

Note that if A_{i+3} does not exist, the above operation eliminates the corner between A_i and A_{i+1}. □

Lemma 3.12. *Suppose f is a FST and let m be the number of Steiner points on f. There exists a f' equivalent to f such that*

(i) if m is odd, the Steiner chain of f' is a straight line.

(ii) if m is even, all the Steiner points are on a straight line except possibly the last one.

Proof. By pushing the corners along the direction according to Lemma 3.11, there will be at most one corner connecting the last two Steiner points. If m is odd, the corner can be eliminated. □

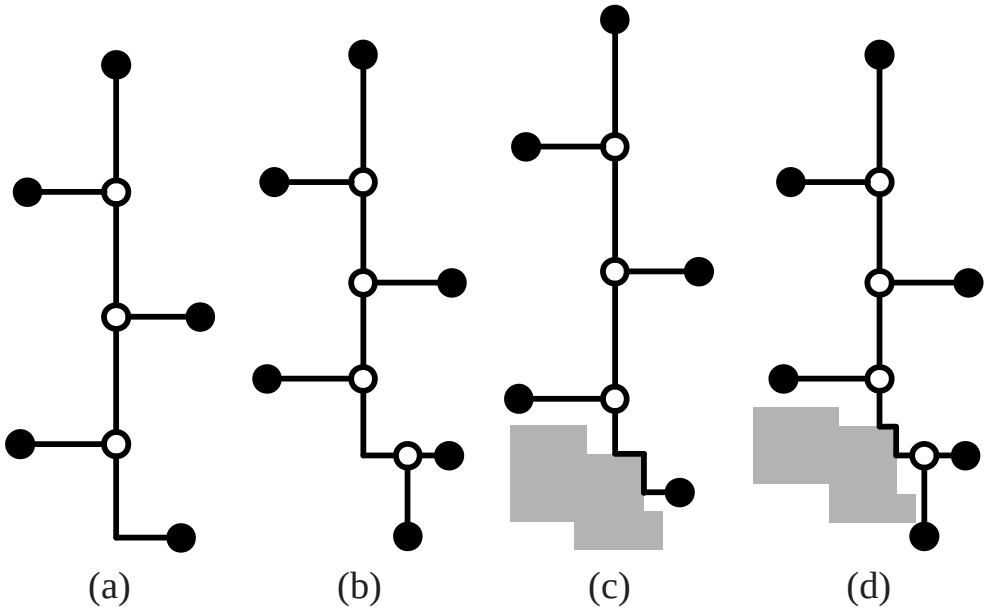


Figure 3.19: Possible structures of a FST among complex obstacles. (a) Type I structure. (b) Type II structure. (c) Type III structure. (d) Type IV structure.

To summarize, if the Steiner chain is a straight line, the horizontal node line linked to the sequence of Steiner points must alternate in the left-right directions. Hence, each Steiner point has exactly one horizontal node line except the first and the last one. Similar as in [48], by putting all of the above lemmas together, we can have the following conclusion:

Theorem 3.1. *The structures of a FST among complex obstacles must be in one of the four forms as shown in Fig. 3.19.*

As we can observe from the figures, the structures of FSTs in the presence of rectilinear obstacles are very similar to those in [26] and [48]. The first two structures are exactly the same as those in [26] and [48]. However, in the presence of

complex obstacles, the FSTs have two additional structures. A main characteristic of these two additional structures is that the last corner connecting two Steiner points or one Steiner point and one terminal is blocked by some obstacles. The similarities indicate that we can use the same method to construct the FSTs defined in this chapter efficiently, making it possible to use the two-phase approach to solve an OARSMT problem in the presence of complex rectilinear obstacles.

3.3.2 More Theoretical results

We mentioned in the previous section that the OARSMT problem can be transformed into a graph problem by using the escape graph. The escape graph is known to be the simplest graph model that contains at least one optimal solution to the OARSMT problem. In this section, we will introduce a new graph called *virtual graph* that is simpler than the escape graph. Based on the theorem we presented in the previous section, we will show that the virtual graph is a strong connection graph that contains at least one optimal solution.

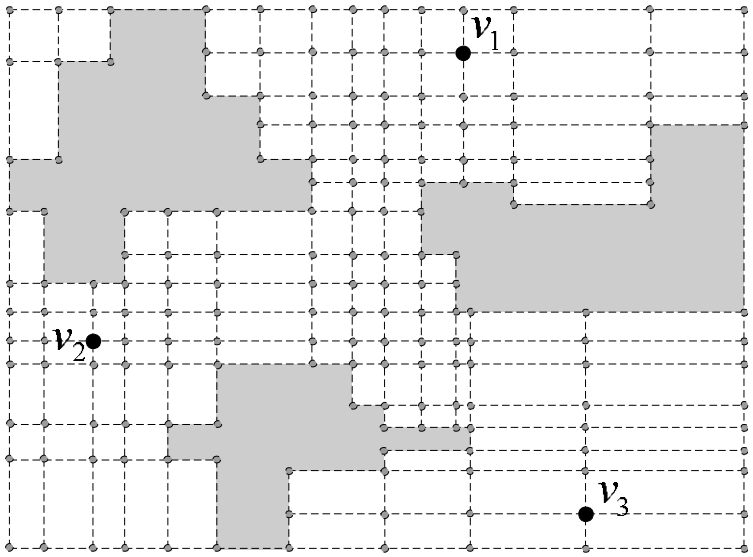


Figure 3.20: Escape graph.

The escape graph consists of two types of segments. The first type is the segments that extend from the terminals in the vertical and horizontal directions, and end at an obstacle boundary or the boundary of the whole routing region. The other type of segments is obtained by extending boundary segments of each obstacle until an obstacle boundary or the boundary of the whole routing region is met. The vertices of the graph are the terminals and the segment intersection points. An example is given in Fig. 3.20 where there are three terminals in the presence of three rectilinear obstacles. Therefore, the size of the escape graph is $O((m+b)^2)$, where m is the number of terminals and b is the number of obstacle boundary segments. It is proven in [30] that for any OARSMT problem, there is at least one optimal solution composed only of the escape segments in the escape graph. The importance of the escape graph is that, with this model, one can transform the geometric OARSMT problem into a graph problem. As a result, some graph based searching algorithms [41, 15] can also be applied to this problem. The introduction of escape graph has also led to a set of heuristics [54, 55] for the OARSMT problem.

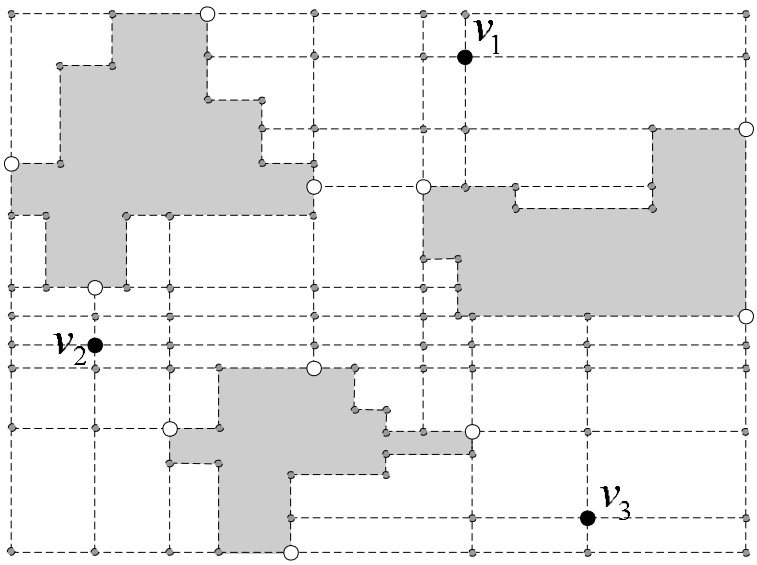


Figure 3.21: Virtual graph.

In the following, we will introduce a new graph called virtual graph based on the virtual terminals we added to the problem. The virtual graph is composed of two types of segments. The first type is the segments that extend from the terminals and virtual terminals in the vertical and horizontal directions, and end at an obstacle boundary or the boundary of the whole routing region. The second type is the obstacle boundary segments. The vertices of the graph are the terminals, virtual terminals and the segment intersection points. An example is shown in Fig.3.21.

Theorem 3.2. *For any OARSMT problem, there is at least one optimal solution contained in the virtual graph.*

Proof. Any optimal OARSMT can be decomposed into a set of FSTs among complex obstacles. By Theorem 3.1, there are only two types of segments in the FSTs. The first type is the segments that extend from either a terminal or virtual terminal horizontally or vertically. The second type is the segments that go around obstacles. By the definition of the virtual graph, it can be easily verified that all FSTs can be further decomposed into segments in the virtual graph. Therefore, there is at least one optimal solution contained in the virtual graph. □

By Theorem 3.2, we can see that virtual graph is also a strong connection graph. The size of the graph is $O((m+e)^2+b)$, where e is the number of essential edges. In comparison with the escape graph, the size of the virtual graph is smaller. In the particular examples shown in Fig. 3.20 and 3.21, the escape graph consists of 184 nodes and 319 edges while the virtual graph only consists of 104 nodes and 158 edges. The simplicity of virtual graph also benefits from the flexibility in choosing the positions of the virtual terminals. Note that we only require one virtual terminal on each essential edge. As shown in Fig. 3.21, three virtual terminals are chosen to be internal points of essential edges to align with real terminals or other virtual

terminals. This can further reduce the size of the graph.

Proposing a simple graph model is of vital importance for the OARSMT problem. Since the problem is NP-complete, a simpler graph can lead to a dramatic reduction of the solution space. Moreover, this graph model is also promising for the graph-based heuristics to improve their performance.

3.4 OARSMT construction

An OARSMT can be partitioned into a set of FSTs by splitting at real terminals or virtual terminals of degree more than one. Therefore, any OARSMT is a union of FSTs. As we can observe, FSTs are much easier to generate than OARSMTs. Therefore, one possible way to construct an OARSMT is to first construct its FSTs components and then combine a subset of them.

Similar to [48], we adopt a two-phase approach to construct an OARSMT. The first phase is to generate a set of FSTs. The second phase is to combine a subset of FSTs generated in the first phase to construct an OARSMT. In our early experiments, we found that the FST concatenation phase usually dominates the total run time. Therefore, we propose a pruning algorithm to further eliminate useless FSTs resulting from the FST generation phase. This can reduce the number of FSTs that needs to be considered in the second phase leading to a significant improvement on the total run time.

3.4.1 FST generation

To grow FSTs of a RSMT, Zachariasen [61] proposed an efficient algorithm in which some pre-processing information is applied to prune away those FSTs that are not required in any RSMTs. In this chapter, we modify this algorithm for the

generation of FSTs with blockages. Our FST generation algorithm differs from the previous one in the following aspects. First, we extend the screening tests to handle virtual terminals and blockages. Second, we develop an efficient approach to construct two-terminal FSTs when virtual terminals exist.

3.4.1.1 Generation of FSTs with three or more terminals

The structures described in Theorem 3.1 will be used to identify FSTs. To reduce the number of resulting FSTs, we identify some necessary conditions for a FST to be a part of an OARSMT as in [61]. Most of the conditions in [61] are applicable to the proposed FSTs after some modifications. In the following, we will focus on the modifications made when obstacles and virtual terminals exist.

The *bottleneck Steiner distance* can be used to eliminate useless FSTs when obstacles exist. Let OARMST(V) be an obstacle avoiding rectilinear minimum spanning tree of the point set V and v_i, $v_j \in V$ be a pair of vertices. The bottleneck Steiner distance $\delta_{OARMST(v_iv_j)}$ between v_i and v_j is equal to the length of the longest edge on the unique path between v_i and v_j in OARMST(V). Salowe *et al.* [45] proposed a theorem stating that if MST and SMT are respectively a minimum spanning tree and a Steiner minimal tree on a set of vertices V, then $\delta_{MST(v_iv_j)} \geq \delta_{SMT(v_iv_j)}$ for any $v_i, v_j \in V$. It can be easily verified that the property also holds for OARMST(V) and OARSMT(V). For a FST f to be part of an OARSMT, we require that $\delta_{MST(v_iv_j)} \geq \delta_{s(v_iv_j)}$ for any $v_i, v_j \in f \cap V$.

The *empty diamond property* proposed in [61] states that no other points of the RSMT can lie in $\mathscr{L}(u,v)$, where uv is a (horizontal or vertical) segment and $\mathscr{L}(u,v)$ is an area on the plane such that all the points in this area are closer to both u and v than u and v are to each other. However, when there are obstacles and virtual terminals, the points which cannot lie in $\mathscr{L}(u,v)$ are the real terminals in V

only.

The *empty corner rectangle property* is also proposed in [61]. Let *uw* and *vw* denote two perpendicular segments sharing a common endpoint *w*. Then, no other points of the RSMT can lie in the interior of the smallest axis-aligned rectangle containing *u* and *v*. However, when there are obstacles and virtual terminals in the routing region, we only need to consider real terminals which can be projected on *uw* and *vw* without intersecting with any obstacles.

We also make use of the *empty inner rectangle property* proposed in [61]. A FST can be transformed to its corner-flipped version by shifting segments and flipping corners. The empty inner rectangle property states that no terminal (real or virtual) should be located between the backbone of the origin topology and that of the corner-flip topology.

Based on the above properties, we can generate all the required FSTs by growing them recursively as in [61].

3.4.1.2 Generation of FSTs with two terminals

For those FSTs with exactly two terminals, we will construct them by the following method. First of all, these FSTs can be divided into two types. The first type has its two end points both in V. The second type has at least one of its end points in U.

For the first type, we can construct them according to the following lemma which is proposed by Fößmeier *et al.* [17].

Lemma 3.13. *Let $G = (W, E)$ be a graph with edges assigned mutually distinct weights and let W' be a subset of W. Let L be an MST of G and L' be an MST of $G[W']$, the subgraph of G induced by W'. Then every edge (u, w) in L where both u and w are in W' will also appear in L'.*

This lemma indicates that every two-terminal FST, in the OARSMT and with its two end points both in V, will also appear in the OARMST of V. In order to generate all possible type one two-terminal FSTs, we only need to construct an OARMST of V and include all the edges in it as candidates. In order to handle the requirement of mutually distinct weights, we arrange the edges with the same length by comparing their positions in the edge array. The one that has a smaller index is assumed to be "shorter". Note that this will not affect the optimality of the generated OARSMT.

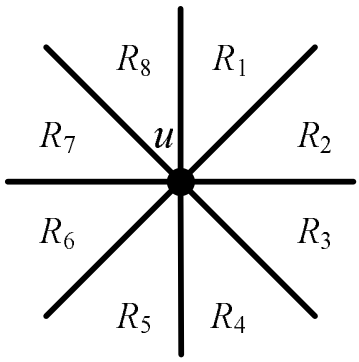


Figure 3.22: The eight regions of a terminal.

For the second type, we will make use of a lemma proposed by Yao [58]. We know that at least one of the two end points of the FST under construction is in U and the rectangular area covered by the two end points is obstacle free (otherwise we can flip the edge to the boundary of the obstacle to obtain an equivalent tree with forbidden structures). For each virtual terminal $u \in U$, we divide its surrounding area into eight regions R_i for $i = 1, \ldots, 8$, as shown in Fig. 3.22. In every region R_i, we find the point $v \in V$ that has the shortest manhattan distance (d_{uv}) from u and the rectangular area covered by u and v has no obstacle. Then, the edge connecting u and v is a two-terminal FST candidate. In this region R_i, we also find those points $w \in U$ with distance $d_{uw} \leq d_{uv}$ and the rectangular area covered by u and w

is obstacle free. Then, the edge connecting u and w will also be included as a FST candidate. To verify the correctness of this approach, we assume on the contrary that there exist a two-terminal FST in the OARSMT, but not in our candidate set. Without loss of generality, we use $u \in U$ and $w \in V+U$ to denote the two end points of the FST. Assume that w is in the R_k region of u. Since the FST is not in our candidate set, there exist a terminal $v \in V$ in R_k such that $d_{uv} < d_{uw}$. According to [58], we have $d_{wv} < d_{uw}$, but this is impossible for otherwise we can delete (u,w) and connect either (u,v) or (w,v) to build a shorter tree. Therefore, the FST cannot exist which proves the correctness of our approach.

Based on the above methods, we can find all necessary two-terminal FSTs. Since the number of two-terminal FSTs is very large, some techniques are adopted to remove redundancies. Firstly, the empty diamond property is tested for every two-terminal FST and those that fail to satisfy the condition will be eliminated. Secondly, according to the definition of FSTs, we will remove an edge if the rectangular area covered by the end points is not obstacle free. Finally, the empty inner rectangle property is checked. If the rectangular area covered by the end points is obstacle free but contains some terminals in U, we will also remove the FST. This technique has also been adopted by Zachariasen in [61].

3.4.2 Pruning of FSTs

We propose an efficient pruning procedure to reduce the number of FSTs needed to construct an OARSMT. Although some pruning is also done in the FST generation phase, these tests consider only one FST at a time. To further eliminate useless FSTs, a set of FSTs should be considered simultaneously. The proposed pruning algorithm works by growing a FST f to larger trees and test if these larger trees can exist in the optimal solution. We know that virtual terminals in an OARSMT

must have degree two, three or four. Therefore, it is possible to grow a tree at a leaf node which is a virtual terminal. The growing is done by combining FSTs at virtual terminals. The rationale behind is that a FST f can be eliminated if no tree grew from f can exist in an OARSMT. The pseudocode of the pruning algorithm is shown in Fig. 3.23.

The input of the function PRUNE(f) is a FST f. The function returns a value *true* or $false$ to indicate whether f can be eliminated or not. A queue Q is used to store all the trees we can grow from f during the test. Initially, Q contains f only. The algorithm repeatedly removes a tree T from Q and tests if T can be a part of any OARSMT. The function PASS_TEST(T) returns *true* if T passes all the tests used to eliminate useless trees. In this case, the function LEAST_DEGREE(T) is used to select a virtual terminal u that is also a leaf node of T. If there are more than one such virtual terminals, the function returns the one that is connected by the least number of FSTs. The algorithm then tries to grow T by connecting T with combinations of FSTs at u. All such expansions are added to the queue. If PASS_TEST(T) returns $false$, T can be eliminated and no more expansion is needed. The algorithm stops when Q is empty which means that no tree grew from f can be in an OARSMT. We can then safely eliminate f. If at some point Q is full or all leaf nodes of T are real terminals, the algorithm terminates and returns $false$.

Four tests are used in the function PASS_TEST(T) to eliminate useless trees. In the following, we let $V_T \subseteq V+U$ be the set of terminals connected by T.

The first test tries to construct a shorter tree that spans the same set of terminals.

Lemma 3.14. *T cannot be a part of any OARSMT over V, if the length of T is larger than the length of an OARSMT over V_T.*

Proof. If T is part of a Steiner minimum tree, we can replace T with the OARSMT over V_T, yielding a tree with shorter length, an absurdity. □

Algorithm 1: PRUNE(f)
Input: f
Output: $true$ or $false$

1: initialize a queue Q
2: push $f \rightarrow Q$
3: **while** Q is not empty **do**
4: pop $T \leftarrow Q$
5: **if** PASS_TEST(T) **then**
6: **if** ALL_REAL(T) **then**
7: **return** $false$
8: **end if**
9: u = LEAST_DEGREE(T)
10: $S = \{ f_i : (f_i \in F) \wedge (u \in f_i) \wedge (f_i \notin T) \}$
11: **for all** $T' \in \begin{pmatrix} S \\ 1 \end{pmatrix} \cup \begin{pmatrix} S \\ 2 \end{pmatrix} \cup \begin{pmatrix} S \\ 3 \end{pmatrix}$ **do**
12: push $T \cup T' \rightarrow Q$
13: **if** Q is full **then**
14: **return** $false$
15: **end if**
16: **end for**
17: **end if**
18: **end while**RETURN $true$

Figure 3.23: Pseudocode of the pruning algorithm

To compute an OARSMT over V_T, we will include all FSTs that span exactly a subset of V_T and pass them to the FST concatenation phase. Since the computation of OARSMT over V_T can be expensive, this test is performed only when the number of terminals in T is less than a predefined number (this number is set to 30 in our implementation).

The second test makes use of the bottleneck Steiner distance. Let $(V+U, E, c)$ be the distance graph[2] of $V+U$, with E being the set of edges between every pair of terminals in $V+U$ and $c : E \rightarrow \Re^+$ being a positive length function on E. A path P in the distance graph is an elementary path if both of its two endpoints are in V.

[2] A distance graph is a graph formed from a collection of points in the plane by connecting every two points by an edge, and the edge weight equals to the distance between the two points.

The Steiner distance of P is the length of the longest elementary path in P. The bottleneck Steiner distance $s_{u,v}$ between u and v is the minimum Steiner distance over all the paths from u to v in $(V+U,E,c)$. Such a path is known as a bottleneck Steiner path.

Lemma 3.15. *T cannot be a part of any OARSMT over V, if the length of the tree $c(T)$ is larger than the length of the minimum spanning tree over V_T in $(V+U,E,s)$ (the graph that uses distances $s_{u,v}$ as a measure of the edge weight for every pair of terminals).*

Proof. It is proven in [15, 41] that if $c(T)$ is larger than the length of the minimum spanning tree over V_T in $(V+U,E,s)$, a tree shorter than $c(T)$ spanning V_T exists in $(V+U,E,c)$. As a result, in such a case, T cannot be a part of any OARSMT. □

The third test compares the tree distance and the bottleneck Steiner distance between two terminals in T.

Lemma 3.16. *Let u and v be two terminals in T and $t_{u,v}$ be the length of the longest edge on the path between u and v in T. T cannot be a part of any OARSMT over V, if $t_{u,v} > s_{u,v}$.*

Proof. Assume the contrary that T is in a Steiner minimum tree. Remove the longest edge on the path between u and v in T and the Steiner minimum tree is divided into two components. Along the bottleneck Steiner path between u and v, let P' be an elementary path such that its two endpoints are in different components. Note that the length of P' should be no larger than $s_{u,v}$. Therefore, we can reconnect the two components by P' yielding a shorter tree, a contradiction. □

Note that the second and third tests both make use of the bottleneck Steiner distance between pairs of vertices. The bottleneck Steiner distance between any

pair of vertices u and v in the graph $(V+U,E,c)$ can be obtained by determining the Steiner distance on the path between these two vertices in the spanning tree over V.

The fourth test exploits the lower and upper bounds on the length of a Steiner minimum tree. To obtain the lower bound on the length of an OARSMT, one way is to solve the linear programming relaxation of the FST concatenation problem formulation as described in [48]. However, this approach is not practical due to its high computational cost. An alternative is the dual ascent heuristic proposed in [52], which is a fast heuristic that provides a lower bound for the Steiner arborescence problem in a directed graph. To apply this method, we first construct a directed graph $(V+U+S,E_F,d)$. S is the set of all Steiner points in all FST. E_F is the set of directed edges which is generated by transferring each edge in a FST to its two directed versions. $d: E_F \rightarrow \Re^+$ is the edge length function. It can be easily verified that the FST concatenation problem is equivalent to finding a shortest arborescence tree in $(V+U+S,E_F,d)$ that rooted at a terminal z and spans all the other terminals in V. As a result, we can use the dual ascent heuristic to compute the lower bound and the associated reduced cost for each edge. To compute an upper bound, the maze routing based heuristic proposed in [35] is used. In the following, let *lower* be the lower bound, *upper* be the upper bound, $r: E_F \rightarrow \Re^+$ be the reduce cost[3] function on E_F, and $r(u,v)$ be the reduced cost distance between u and v in the graph. Let $l_1, l_2, \ldots, l_k$ be the leaves of T and $\overrightarrow{T}_i$ be the directed version of T rooted at l_i. We use $r(T_i)$ to denote the reduced cost of $\overrightarrow{T}_i$.

Lemma 3.17. *T cannot be a part of any OARSMT over V, if $lower + min_i\{r(z,l_i) +$*

[3]Finding a shortest arborescence tree in a graph can be formulated as an integer linear programming. In linear programming, the reduced cost value indicates how much the objective function coefficient on the corresponding variable must be improved before the value of the variable will be positive in the optimal solution.

$r(T_i) + \sum_{j \neq i} min_{v \in V - \{z\}} r(l_j, v)\} > upper$ *in which z is the root node.*

Proof. It is proven in [41] that $lower_{constrained} = lower + min_i\{r(z, l_i) + r(T_i) + \sum_{j \neq i} min_{v \in V - \{z\}} r(l_j, v)\}$ is a lower bound for the length of any Steiner tree with the additional constraint that it contains T. Therefore, if $lower_{constrained} > upper$, T cannot be a part of any OARSMT. □

If T fails any of these four tests, PASS_TEST(T) returns $false$, and T can be eliminated.

3.4.3 FST concatenation

The second phase of the algorithm is to use the FSTs generated in the first phase to construct an OARSMT spanning all real terminals with the minimum total length. In the construction of RSMTs, Warme [49] found that the FST concatenation problem is equivalent to the minimum spanning tree problem in hypergraph and formulated it as an integer linear program (ILP). A branch-and-cut algorithm is used to solve this problem. In this section, we will show that the FST concatenation problem in this chapter can also be formulated as an ILP and solved by using the branch-and-cut search. Generally, our FSTs concatenation phase differs from the previous one in the following aspects. We modify the ILP formulation for FST concatenation and the separation algorithm in [49] to handle virtual terminals. New features are introduced to accommodate the presence of virtual terminals. We also provide a theoretical proof to verify the correctness of the new separation algorithm.

3.4.3.1 ILP formulation

In the following, let F be the set of all FSTs found. Let V be the set of all real terminals and U be the set of all virtual terminals. Let $|V|$ be the number of real

terminals, $|F|$ be the number of FSTs in F and $|U|$ be the number of virtual terminals. Each FST $f_i \in F$ is associated with a binary variable x_i indicating whether f_i is taken as a part of the OARSMT. Besides, there are binary variables y_i for $i = 1 \ldots |U|$ indicating whether virtual terminal $u_i \in U$ is connected in the OARSMT. We use $|f_i|$ to denote the size of f_i, i.e., the number of terminals (including virtual ones) connected by f_i, and use l_i to denote the length of f_i. The ILP formulation is as follows.

Minimize:

$$\sum_{i=1}^{|F|} l_i \times x_i. \tag{3.1}$$

Subject to:

$$\sum_{i=1}^{|F|} x_i(|f_i| - 1) = |V| - 1 + \sum_{i=1}^{|U|} y_i, \tag{3.2}$$

$$2y_j \leq \sum_{i:u_j \in f_i} x_i \quad \forall u_j \in U, \tag{3.3}$$

$$4y_j \geq \sum_{i:u_j \in f_i} x_i \quad \forall u_j \in U, \tag{3.4}$$

$$\sum_{i:f_i \in (X:V+U-X)} x_i \geq 1$$

$$\forall X \subseteq V+U \text{ and } V \nsubseteq X \text{ and } X \cap V \neq \varnothing, \tag{3.5}$$

$$\sum_{i:f_i \cap X \neq \varnothing} x_i(|f_i \cap X| - 1) \leq |X \cap V| + \sum_{i:u_i \in X} y_i - 1$$

$$\forall X \subset V+U \text{ and } X \cap V \neq \varnothing \text{ and } |X| \geq 2, \tag{3.6}$$

$$\sum_{i:f_i \cap X \neq \varnothing} x_i(|f_i \cap X| - 1) \leq \sum_{i:u_i \in X} y_i - \max_{i:u_i \in X}(y_i)$$

$$\forall X \subseteq U \text{ and } |X| \geq 2. \tag{3.7}$$

The notation $(X : V+U-X)$ in (3.5) means $\{ f_i \in F : f_i \cap X \neq \varnothing \wedge f_i \cap (V+U-X) \neq$

$\varnothing\}$. Constraint (3.2) is the *total degree constraint*. It requires the right amount of FSTs to construct an OARSMT. Each selected FST f_i contributes $|f_i|-1$ edges for the tree. Since we do not know the exact number of terminals in the tree, $\sum_{i=1}^{|U|} y_i$ is added to indicate the number of selected virtual terminals. Constraints (3.3) and (3.4) bound the degree of any selected virtual terminal to be two, three, or four. Constraints (3.5) are the *cutset constraints*. The constraints require that a solution should be connected, that is, for any cut with partitions X and $V+U-X$, there must be at least one selected FST to connect them. We require $X \cap V \neq \varnothing$ and $V \nsubseteq X$, because we do not need to ensure the connectivity of the virtual terminals. Constraints (3.6) and (3.7) are the *subtour elimination constraints* that are used to eliminate cycles. In (3.6), we consider those sets $X \cap V \neq \varnothing$. Since y_i tells whether u_i is selected, $|X \cap V| + \sum_{i:u_i \in X} y_i$ gives the exact number of selected terminals including virtual ones in X. In (3.7), we use $\sum_{i:u_i \in X} y_i$ to indicate the number of selected terminals in X. Since it is possible that the number of selected terminals in X is equal to zero, we do not simply subtract one from the right hand side of the inequality. Instead, the term $\max_{i:u_i \in X}(y_i)$ is used to ensure that the inequality is not binding when the number of selected terminals in X is zero.

3.4.3.2 Branch-and-cut

The ILP described in the above section is solved via a branch-and-cut framework using lower bounds provided by the linear programming (LP) relaxation. We adopt the algorithm proposed by Warme [49] and extend it for solving the ILP formulation of our FST concatenation problem. The pseudocode of the algorithm is shown in Fig. 3.24. In the following, we will give a brief overview of the algorithm, including initialization, node processing, and branching, and point out the differences in the separation algorithm in order to deal with our formulation. The readers may

refer to [49] for more details.

```
Algorithm branch-and-cut(F)
Input: F // The set of all FSTs
Output: OARSMT
 1: initialization
 2: add the first node to the node list
 3: while node list is not empty do
 4:    select a node from the node list
 5:    repeat
 6:       node processing
 7:       if LP feasible and objective value < best known objec-
          tive value then
 8:          if the LP solution is integral and connected then
 9:             save it as the best known integral solution
10:          end if
11:          separation
12:       end if
13:    until 1: LP infeasible, or
             2: objective value ≥ best known objective value,
              or
             3: separation found no violation
14:    if case 1 or case 2 then
15:       delete the current node
16:    end if
17:    if case 3 then
18:       if the solution is fractional then
19:          branching
20:       end if
21:       if the solution is integral then
22:          delete the current node
23:       end if
24:    end if
25: end while
26: return the best integral solution // OARSMT
```

Figure 3.24: Pseudocode of the branch-and-cut algorithm.

Initialization

Since there are an exponential number of constraints according to the problem formulation, we handle them incrementally by using some separation methods.

A constraint pool is used to keep all the currently processing constraints of the ILP. At the beginning of the algorithm, the constraint pool is initialized with the total degree constraint (3.2), constraints for virtual terminals (3.3) and (3.4), all one-terminal cutset constraints ((3.5) with $|X| = 1$), and all two-terminal subtour elimination constraints ((3.6) and (3.7) with $|X| = 2$). Besides, an LP tableaux is constructed to store the constraints (which is a subset of the constraints retained in the constraint pool) being handled by the LP solver. The initial LP tableaux consists of all the constraints in the constraint pool except the two-terminal subtour elimination constraints.

Node processing

The objective of the node processing procedure is to compute an optimal LP solution over the current constraint pool. The process begins with solving a linear relaxation with the constraints in the LP tableaux. If a solution exists, we will scan the constraint pool and check for violations. All violated constraints found will be added to the LP tableaux which is solved again in the next iteration. This operation terminates when the LP solution satisfies all the constraints in the pool (LP feasible) or a feasible solution does not exist (LP infeasible). If the result is LP infeasible or the objective value of the LP solution exceeds the objective value of the best known integral solution, the processing of this node ends and the node will be deleted. If the objective value of the LP solution is better than that of the best known integral solution, the integrality and connectivity of the LP solution is checked. If the solution is both integral and connected, it is saved as the best known integral solution. A separation procedure will then be invoked. Note that after obtaining an optimum over the current pool, slack constraints[4] will be deleted from the LP tableaux (but are retained in the constraint pool).

[4]Slack values of linear constraints are available to be queried from the LP solver.

Separation

The objective of the separation procedure is to find a set of constraints that is not present in the constraint pool, but is violated by the current solution. These constraints will be added to the constraint pool. There are mainly two sets of constraints to be considered, namely the cutset constraints (3.5) and the subtour elimination constraints (3.6) and (3.7).

The first step in the separation procedure is to find the cuts $(X : V + U - X)$ with $\sum_{i: f_i \in (X:V+U-X)} x_i = 0$ that violate the cutset constraints (3.5). We first compute the connected components $D_1, D_2, D_3, \ldots, D_k$ of the solution. Since we do not need to ensure the connectivity of the virtual terminals, we require that $D_i \cap V \neq \emptyset$, $\forall 1 \leq i \leq k$. If $k > 1$, there exist cutsets of zero weight. If $k < 10$, we generate cutsets constraints for all the cuts induced by the connected components. If $k \geq 10$, we only generate the cutset constraints $(D_i : V + U - D_i)$ for $1 \leq i \leq k$. Notice that we do not consider those cuts with $0 < \sum_{i: f_i \in (X:V+U-X)} x_i < 1$ because they are too expensive to be identified while little improvement in the objective value can be made.

The second step is to find violations of the subtour elimination constraints (3.6) and (3.7). We define the following function

$$f(X) = |X \cap V| + \sum_{i: u_i \in X} y_i - \sum_{i: f_i \cap X \neq \emptyset} x_i(|f_i \cap X| - 1). \tag{3.8}$$

Then finding violations of constraints (3.6) is equivalent to finding an $X \subset V + U$ such that $X \neq \emptyset$ and $f(X) < 1$. Before exactly solving this problem, we first apply problem reductions to speedup the process. In [49], the "congestion level" of a real terminal b_{v_j} is defined as

$$b_{v_j} = \sum_{i: v_j \in f_i} x_i. \tag{3.9}$$

A real terminal v_j is uncongested if $b_{v_j} \leq 1$. In this chapter, we define the "congestion level" of a virtual terminal as

$$b_{u_j} = \sum_{i:u_j \in f_i} x_i. \tag{3.10}$$

We say that a virtual terminal u_j is uncongested if $b_{u_j} \leq y_j$. By the definition of "congestion level" we can have the following lemma.

Lemma 3.18. *If a terminal w is uncongested and $f(X \cup \{w\}) < 1$, then $f(X) \leq f(X \cup \{w\}) < 1$.*

Proof. Let

$$A = \{ f_i \in F : |f_i \cap X| \geq 1 \wedge w \in f_i \} \text{ and } B = \{ f_i \in F : |f_i \cap X| \geq 1 \wedge w \notin f_i \}.$$

Then

$$\begin{aligned}
&f(X \cup \{w\}) - f(X) \\
=&|(X \cup \{w\}) \cap V| + \sum_{i:u_i \in X \cup \{w\}} y_i - \sum_{i:f_i \in A} |f_i \cap X| x_i - \sum_{i:f_i \in B} (|f_i \cap X| - 1) x_i \\
&- |X \cap V| - \sum_{i:u_i \in X} y_i + \sum_{i:f_i \in A} (|f_i \cap X| - 1) x_i + \sum_{i:f_i \in B} (|f_i \cap X| - 1) x_i.
\end{aligned}$$

If w is a real terminal, then

$$\begin{aligned}
&f(X \cup \{w\}) - f(X) \\
&= |X \cap V| + 1 - \sum_{i:f_i \in A} |f_i \cap X| x_i - |X \cap V| + \sum_{i:f_i \in A} |f_i \cap X| x_i - \sum_{i:f_i \in A} x_i \\
&= 1 - \sum_{i:f_i \in A} x_i \geq 1 - b_w \geq 0.
\end{aligned}$$

If w is a virtual terminal and its index is k, then

$$\begin{aligned} & f(X \cup \{w\}) - f(X) \\ &= |X \cap V| + y_k - \sum_{i: f_i \in A} |f_i \cap X| x_i - |X \cap V| + \sum_{i: f_i \in A} |f_i \cap X| x_i - \sum_{i: f_i \in A} x_i \\ &= y_k - \sum_{i: f_i \in A} x_i \geq y_k - b_w \geq 0. \end{aligned}$$

In conclusion, we have $f(X) \leq f(X \cup \{w\}) < 1$. □

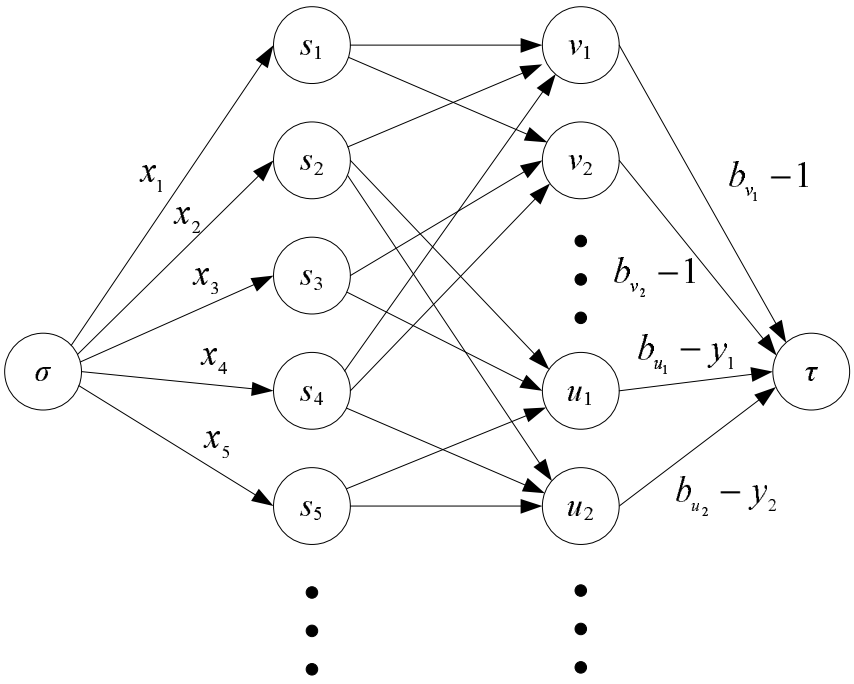


Figure 3.25: The flow network formulation.

According to Lemma 3.18, we can eliminate all uncongested terminals while looking for violations of the subtour elimination constraints. Since subtour elimination constraints are used to eliminate cycles, we can further confine our search to within several biconnected components. We use $C_1, C_2, C_3, \ldots, C_k$ to denote the biconnected components in which every terminal is congested. Now, the problem is reduced to identifying violations within $C_1, C_2, C_3, \ldots, C_k$. For each component

C_i with less than 10 terminals, we will enumerate all subsets X of C_i checking for violations of (3.6) and (3.7). For each remaining component C_i, we use a deterministic network flow method to find violations of (3.6) and (3.7). The deterministic flow network $G = (N, A)$ is defined as follows. Let $N = \{\sigma\} \cup Y \cup Z \cup \{\tau\}$ be the set of nodes in the graph, where $Y = \{f_i : f_i \cap C_i \neq \emptyset\}$, and $Z = \{v_j : v_j \in C_i\} \cup \{u_j : u_j \in C_i\}$ Let the arcs in the graph be $A = A_1 \cup A_2 \cup A_3$, where $A_1 = \{(\sigma, f_i)\}$, $A_2 = \{(f_i, v_j) : v_j \in f_i\} \cup \{(f_i, u_j) : u_j \in f_i\}$, and $A_3 = \{(v_j, \tau) : v_j \in C_i\} \cup \{(u_j, \tau) : u_j \in C_i\}$. Let the arcs A_1 have capacity x_i. Let arcs A_2 have infinite capacity. Let arcs $(v_j, \tau) \in A_3$ have capacity $b_{v_j} - 1$, and $(u_j, \tau) \in A_3$ have capacity $b_{u_j} - y_j$. The flow network is shown in Fig. 3.25. Note that different from the flow network formulation in [49], there are nodes that represent virtual terminals in our formulation.

We define a source to terminal cut $(W : N - W)$ of G such that $\sigma \in W$ and $\tau \in (N - W)$. The capacity of the cut $c(W)$ is the sum of the capacity of all arcs $(a, b) \in A$ such that $a \in W$ and $b \in (N - W)$. We have the following theorem.

Theorem 3.3. *Let $(W : N - W)$ be a source to terminal cut of G that minimize $c(W)$. Let $X_m = \{w : w \in V + U \wedge w \in N - W\}$. Then X_m minimizes $f(X)$.*

Proof. Let $W = \{\sigma\} \cup I \cup J$ be such a minimum cut of G, where $I \subseteq Y$ and $J \subseteq Z$. According to [49], I is completely determined by J.

Let $w_{v_j} = 1$ if $v_j \in W$ and $w_{v_j} = 0$ otherwise. Let $w_{u_j} = 1$ if $u_j \in W$ and $w_{u_j} = 0$ otherwise. Then $c(W)$ can be written as

$$\begin{aligned} c(W) &= \sum_{i: f_i \in F} \left(1 - \prod_{j: v_j \in f_i} w_{v_j} \prod_{j: u_j \in f_i} w_{u_j}\right) x_i + \sum_{j: v_j \in V} (b_{v_j} - 1) w_{v_j} \\ &\quad + \sum_{j: u_j \in U} (b_{u_j} - y_j) w_{u_j} \\ &= \sum_{i: f_i \in F} \left(-x_i \prod_{j: v_j \in f_i} w_{v_j} \prod_{j: u_j \in f_i} w_{u_j}\right) + \sum_{j: v_j \in V} (b_{v_j} - 1) w_{v_j} \end{aligned}$$

$$+ \sum_{j:u_j \in U} (b_{u_j} - y_j) w_{u_j} + \sum_{i:f_i \in F} x_i.$$

Note that the last term in the equation does not depend on w_{v_j} or w_{u_j}, and therefore is a constant. Now consider the function $f(X)$. Let $z_{v_j} = 1$ if $v_j \in X$ and $z_{v_j} = 0$ otherwise. Let $z_{u_j} = 1$ if $u_j \in X$ and $z_{u_j} = 0$ otherwise. Let $\bar{z}_{v_j} = 1 - z_{v_j}$ and $\bar{z}_{u_j} = 1 - z_{u_j}$ be the complementary variables. We can rewrite $f(X)$ as

$$\begin{aligned}
& f(X) \\
&= |X \cap V| + \sum_{i:u_i \in X} y_i - \sum_{i:f_i \cap X \neq \varnothing} x_i (|f_i \cap X| - 1) \\
&= \sum_{j:v_j \in V} z_{v_j} + \sum_{j:u_j \in U} z_{u_j} y_j \\
&\quad - \sum_{i:f_i \in F} \left[\left(\sum_{j:v_j \in f_i} z_{v_j} + \sum_{j:u_j \in f_i} z_{u_j} \right) - 1 + \prod_{j:v_j \in f_i} (1 - z_{v_j}) \prod_{j:u_j \in f_i} (1 - z_{u_j}) \right] x_i \\
&= \sum_{j:v_j \in V} (1 - \bar{z}_{v_j}) + \sum_{j:u_j \in U} (1 - \bar{z}_{u_j}) y_j \\
&\quad - \sum_{i:f_i \in F} \left[\left(\sum_{j:v_j \in f_i} (1 - \bar{z}_{v_j}) + \sum_{j:u_j \in f_i} (1 - \bar{z}_{u_j}) \right) - 1 + \prod_{j:v_j \in f_i} \bar{z}_{v_j} \prod_{j:u_j \in f_i} \bar{z}_{u_j} \right] x_i \\
&= |V| - \sum_{j:v_j \in V} \bar{z}_{v_j} + \sum_{j:u_j \in U} y_j - \sum_{j:u_j \in U} \bar{z}_{u_j} y_j \\
&\quad - \sum_{i:f_i \in F} \left(|f_i| - \sum_{j:v_j \in f_i} \bar{z}_{v_j} - \sum_{j:u_j \in f_i} \bar{z}_{u_j} - 1 + \prod_{j:v_j \in f_i} \bar{z}_{v_j} \prod_{j:u_j \in f_i} \bar{z}_{u_j} \right) x_i \\
&= |V| - \sum_{j:v_j \in V} \bar{z}_{v_j} + \sum_{j:u_j \in U} y_j - \sum_{j:u_j \in U} \bar{z}_{u_j} y_j - \sum_{i:f_i \in F} (|f_i| - 1) x_i \\
&\quad + \sum_{i:f_i \in F} \left(x_i \sum_{j:v_j \in f_i} \bar{z}_{v_j} \right) + \sum_{i:f_i \in F} \left(x_i \sum_{j:u_j \in f_i} \bar{z}_{u_j} \right) \\
&\quad - \sum_{i:f_i \in F} \left(x_i \prod_{j:v_j \in f_i} \bar{z}_{v_j} \prod_{j:u_j \in f_i} \bar{z}_{u_j} \right) \\
&= |V| - \sum_{j:v_j \in V} \bar{z}_{v_j} + \sum_{j:u_j \in U} y_j - \sum_{j:u_j \in U} \bar{z}_{u_j} y_j - \sum_{i:f_i \in F} (|f_i| - 1) x_i
\end{aligned}$$

$$
\begin{aligned}
&+\sum_{j:v_j\in V}\left(\bar{z}_{v_j}\sum_{i:v_j\in f_i}x_i\right)+\sum_{j:u_j\in U}\left(\bar{z}_{u_j}\sum_{i:u_j\in f_i}x_i\right)-\sum_{i:f_i\in F}\left(x_i\prod_{j:v_j\in f_i}\bar{z}_{v_j}\prod_{j:u_j\in f_i}\bar{z}_{u_j}\right)\\
=&|V|-\sum_{j:v_j\in V}\bar{z}_{v_j}+\sum_{j:u_j\in U}y_j-\sum_{j:u_j\in U}\bar{z}_{u_j}y_j-\sum_{i:f_i\in F}(|f_i|-1)x_i\\
&+\sum_{j:v_j\in V}\bar{z}_{v_j}b_{v_j}+\sum_{j:u_j\in U}\bar{z}_{u_j}b_{u_j}-\sum_{i:f_i\in F}\left(x_i\prod_{j:v_j\in f_i}\bar{z}_{v_j}\prod_{j:u_j\in f_i}\bar{z}_{u_j}\right)\\
=&\sum_{i:f_i\in F}\left(-x_i\prod_{j:v_j\in f_i}\bar{z}_{v_j}\prod_{j:u_j\in f_i}\bar{z}_{u_j}\right)+\sum_{j:v_j\in V}\bar{z}_{v_j}\left(b_{v_j}-1\right)\\
&+\sum_{j:u_j\in U}\bar{z}_{u_j}\left(b_{u_j}-y_j\right)-\sum_{i:f_i\in F}(|f_i|-1)x_i+\sum_{j:u_j\in U}y_j+|V|.
\end{aligned}
$$

The last three terms do not depend on $\bar{z}_{v_j}$ or $\bar{z}_{u_j}$, and therefore are constants. By setting $\bar{z}_{v_j}=w_{v_j}$ and $\bar{z}_{u_j}=w_{u_j}$, we can see that $c(W)$ and $f(X)$ differ only by a constant. Therefore, minimizing $c(W)$ is equivalent to minimizing $f(X)$. Let $(W:N-W)$ be a source to terminal cut of G such that $c(W)$ is minimized, then $X_m=\{w:w\in V+U\wedge w\in N-W\}$ is a minimum of $f(X)$. □

This theorem states that finding an X of C_i that violates (3.6) can be reduced to finding a minimum cut on the flow network G. This problem can be solved in polynomial time. Note that although the above procedure is not exact in finding violations of constraints (3.7), it can still provide good estimations.

Branching

If no violation can be found by separation and the node processing terminates with a fractional solution, branching on the current node occurs. A branch variable x_i (y_j) with non-integral value is selected. Two new nodes are generated by appending the constraints $x_i=0$ or $x_i=1$ ($y_j=0$ or $y_j=1$) to the current node. The processing of the current node terminates. New nodes are selected for processing until there is no node left in the node list.

Algorithm 2: ObSteiner(V, O)
Input: V, O
Output: OARSMT

```
 1: initialize the obstacle list OL to ∅
 2: while true do
 3:    FST generation
 4:    FST pruning
 5:    FST concatenation
 6:    for all FSTs in the current solution do
 7:       for all line segments in the FST do
 8:          check if the line segment intersects with any obsta-
             cles
 9:          if it intersects with obstacles then
10:             add the dominating obstacle to OL
11:          end if
12:       end for
13:    end for
14:    if no overlapping obstacle exists then
15:       goto line 18
16:    end if
17: end while
18: return the OARSMT
```

Figure 3.26: Pseudocode of ObSteiner.

3.5 Incremental construction

By using the two-phase approach, we can solve the OARSMT problem optimally. However, considering all obstacles together may result in a large number of virtual terminals. In our early experiments, we found that adding all obstacles simultaneously would result in an explosion of FSTs. A more efficient way is to consider an obstacle only when it is necessary. Therefore, we adopt an incremental approach to construct an OARSMT. An obstacle list is maintained during the generation of the OARSMT. The list is responsible for keeping track of the obstacles we need to avoid during the construction. Initially, the OARSMT problem with an empty list of obstacles is solved resulting in an RSMT. We then check for obstacles that

overlap with the solution. For each FST used to build the current solution, we decompose it into line segments. For each line segment, we will check whether it intersects with any obstacles. Among all overlapping obstacles we will choose the dominating one. For example, for a vertical segment, we will choose an obstacle that has the largest width. All chosen obstacles are added to the obstacle list. A new iteration then starts again by solving the OARSMT problem with the obstacles in the renewed list. This procedure repeats until no overlapping obstacle can be found. This approach is effective as in most cases only a fraction of the obstacles will affect the final OARSMT. The pseudocode of this OARSMT construction framework is shown in Fig. 3.26.

3.6 Experiments

We implemented ObSteiner in C based on GeoSteiner-3.1 [1]. The experiments are conducted on a Sun Blade 2500 workstation with two 1.6GHz processors and 2GB memory. Our program runs sequentially on a single processor. There are 21 benchmark circuits which are commonly used as test cases for the OARSMT problem. IND1-IND5 are industrial test cases from Synopsys. RC01-RC11 are benchmarks used in [59]. RT1-RT5 are randomly generated circuits used in [9]. Note that there are overlapping obstacles in these benchmarks. We regard overlapping obstacles as one rectilinear obstacle.

Table 3.1 shows the results obtained by ObSteiner. Column "m" provides the number of terminals in each benchmark. Column "n" provides the number of obstacles in each benchmark. Column "t_{total}" provides the total run time of the algorithm. Column "t_{prune}" provides the run time of the pruning procedure. Column "$|OL|$" provides the number of obstacles considered in the algorithm. We can see

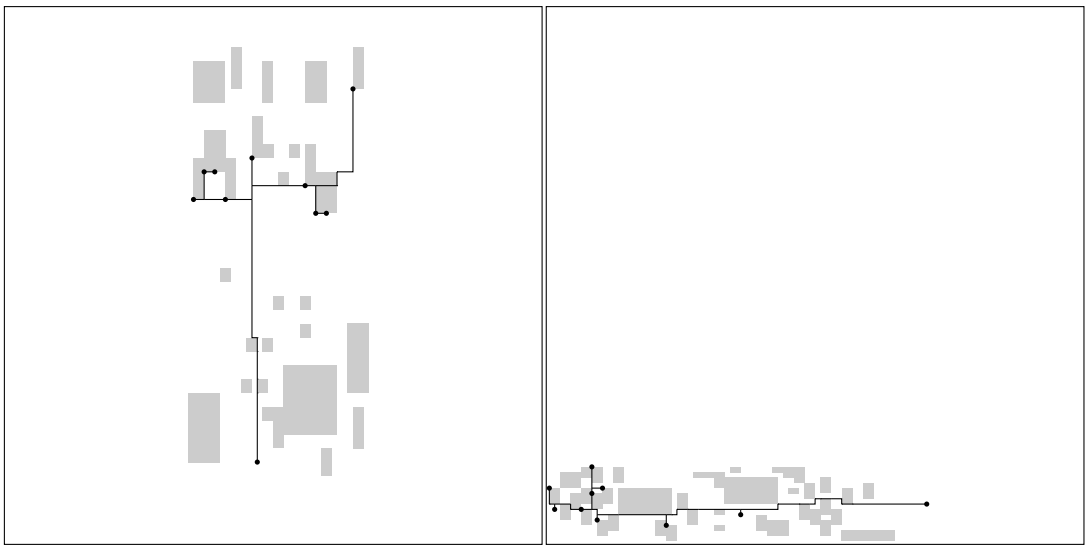
that all benchmarks are solved to optimal in a reasonable amount of time. For small benchmarks (RC01-RC05, IND1-IND5), it takes only seconds to obtain the optimal solution. For the benchmarks with less than or equal to 500 obstacles, the required time is in minutes. We can also observe that the total run time is closely related to the number of obstacles, and more obstacles usually lead to more iterations of the algorithm. In the table, we also list the average FST reduction achieved by the FST pruning procedure and the run time over all iterations. For all benchmarks, around 60% of the FSTs can be eliminated and the run time of the pruning procedure in most cases is less than half of the total time. This can greatly reduce the search space of the branch-and-cut algorithm, and therefore leads to a significant improvement in performance. The computational overhead cause by the pruning procedure is small compared to the savings in the concatenation phase. We can also observe from the table that the incremental construction is very effective. On average, only 23.1% obstacles needs to be considered. This can greatly reduce the number of additional virtual terminals and the resulting FSTs.

In order to clearly show the effectiveness of the pruning procedure and the incremental approach, we compare the run time of ObSteiner with and without these two techniques. Results are listed in Table 3.2. Considering the incremental approach, we can see that, without using this technique, RC06-RC11 and RT1-RT5 will not be solvable within the run time limit. Although for the small benchmarks with 10 obstacles, the incremental approach may worsen the run time, the speedup on large benchmarks is tremendous. Considering the pruning procedure, although it is not as effective as the incremental approach, a considerable speedup can still be achieved. Without using the technique, RT5 cannot be solved within time limit. For small benchmarks, the benefit of using pruning procedure is limited. However, the technique can be very useful for difficult cases. This is because the parameters

in the pruning procedure (e.g. when to stop pruning) are set according to the large benchmarks, which may not be necessary for small cases.

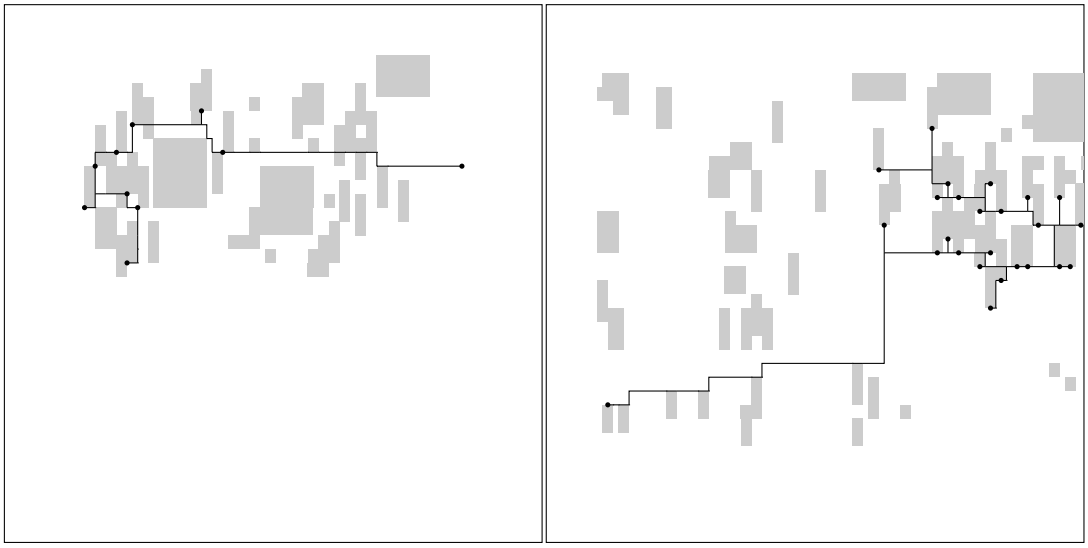
To show the efficiency of ObSteiner, we compare our method with the approach in [48]. The results are tabulated in Table 3.3. We execute the algorithm in [48] on our platform. Since [48] can only handle rectangular obstacles, we change the benchmarks by dissecting rectilinear obstacles into several rectangular obstacles. For completeness, we also tabulate the results of twenty additional test cases which are used in [48]. These test cases can be divided into two categories. The test cases in the first category are generated by taking the first few obstacles in the corresponding benchmarks. We use "benchmark_number" to denote them, in which "benchmark" is the original benchmark and "number" is the number of obstacles taken. The test cases in the second category are generated by taking the obstacles randomly. We use "benchmark_rand_number" to denote them. We run each test case for 96 hours at most. In the table, "-" means that the solution cannot be achieved within the run time limit. As can be observed from the table, the run time required for the OARSMT construction has been improved a lot by our algorithm. Comparing with the approach in [48], ObSteiner can solve problems with up to two thousand obstacles, while the approach in [48] can only deal with cases with less than one hundred obstacles. For small solvable cases, our approach is 31 times faster than the approach in [48] on average.

Table 3.4 compares the performance of some recently published heuristics [6, 35, 7, 18] based on the optimal solutions provided by the proposed exact algorithm. The results are quoted from the corresponding papers. We can see that all four heuristics works better on small problems, obtaining optimal solutions in several cases. The performance gradually decreases with the increasing number of obstacles.



(a) (b)

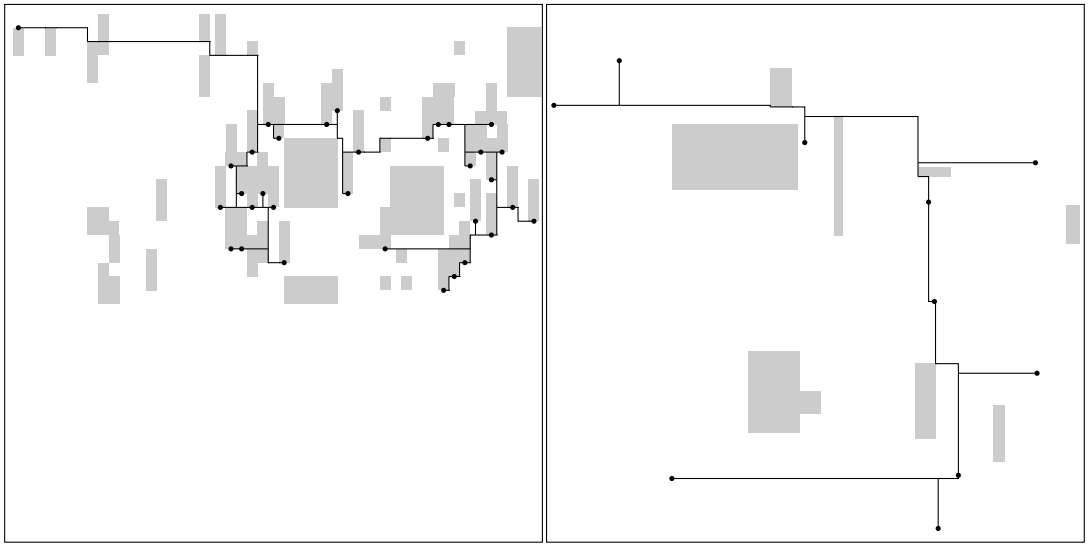
Figure 3.27: The OARSMTs of (a) IND1 (b) IND2.



(a) (b)

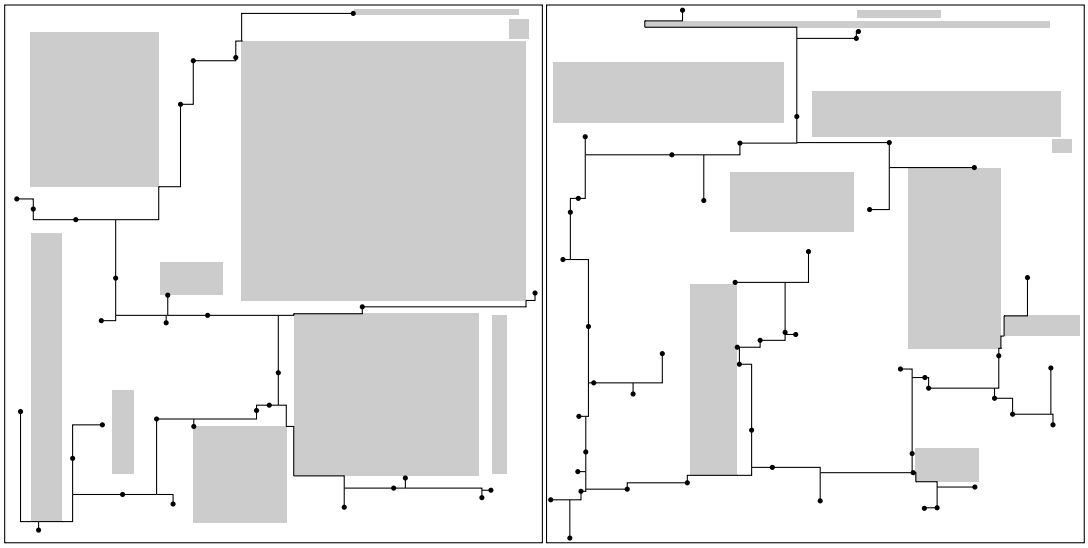
Figure 3.28: The OARSMTs of (a) IND3 (b) IND4.

Fig. 3.27-3.37 shows the resulting OARSMTs generated by ObSteiner for all the benchmarks.



(a) (b)

Figure 3.29: The OARSMTs of (a) IND5 (b) RC01.



(a) (b)

Figure 3.30: The OARSMTs of (a) RC02 (b) RC03.

Table 3.1: Detailed results of ObSteiner.

Benchmark	m	n	OARSMT length	t_{total} (s)	FST reduction (%)	t_{prune} (s)	$\frac{t_{prune}}{t_{total}}$ (%)	Number of iterations	$\|OL\|$	$\frac{\|OL\|}{n}$(%)
RC01	10	10	25980	0.16	76.1	0.02	12.5	2	3	30.0
RC02	30	10	41350	0.52	63.8	0.18	34.6	2	3	30.0
RC03	50	10	54160	0.68	59.6	0.21	30.9	3	6	60.0
RC04	70	10	59070	0.95	72.5	0.37	38.9	2	5	50.0
RC05	100	10	74070	1.31	63.7	0.51	38.9	2	6	60.0
RC06	100	500	79714	335	60.3	180	53.7	6	89	36.0
RC07	200	500	108740	541	62.6	324	59.9	7	100	20.0
RC08	200	800	112564	24170	67.1	4549	18.8	7	161	20.1
RC09	200	1000	111005	14174	72.8	5026	35.5	7	192	19.2
RC10	500	100	164150	176	63.7	90	51.1	5	28	28.0
RC11	1000	100	230837	706	66.4	345	48.9	3	18	18.0
RT1	10	500	2146	25	72.0	10	40.0	6	33	6.6
RT2	50	500	45852	31	61.3	23	74.2	5	42	8.4
RT3	100	500	7964	840	71.6	794	94.5	5	61	12.2
RT4	100	1000	9693	34521	63.7	7939	23.0	11	197	19.7
RT5	200	2000	51313	276621	64.4	26772	9.7	13	388	19.4
IND1	10	32	604	0.11	63.3	0.02	18.2	1	0	0
IND2	10	43	9500	0.25	61.4	0.05	20.0	3	5	11.6
IND3	10	59	600	0.19	68.5	0.04	21.1	2	2	3.4
IND4	25	79	1086	0.87	55.7	0.25	28.7	4	11	13.9
IND5	33	71	1341	1.03	43.9	0.27	26.2	4	14	19.7
Average					64.5		37.1			23.1

Table 3.2: Run time of ObSteiner with and without the pruning procedure and the incremental approach.

Benchmark	ObSteiner w/o PN & IN	ObSteiner w/o IN	ObSteiner w/o PN	ObSteiner
RC01	0.38	0.23	0.17	0.16
RC02	0.21	0.19	0.65	0.52
RC03	0.18	0.20	0.78	0.68
RC04	0.50	0.32	0.96	0.95
RC05	0.70	0.52	1.63	1.31
RC06	-	-	876	335
RC07	-	-	1796	541
RC08	-	-	61005	24170
RC09	-	-	40150	14174
RC10	-	-	855	176
RC11	-	-	21242	706
RT1	-	-	81	25
RT2	-	-	32	31
RT3	-	-	478	840
RT4	-	-	120552	34521
RT5	-	-	-	276621
IND1	29.88	20.78	0.13	0.11
IND2	23.25	18.92	0.27	0.25
IND3	8.78	6.07	0.18	0.19
IND4	133852	1089	0.96	0.87
IND5	43.59	4.24	1.20	1.03
Average	15431×	156×	3.29×	1.00×

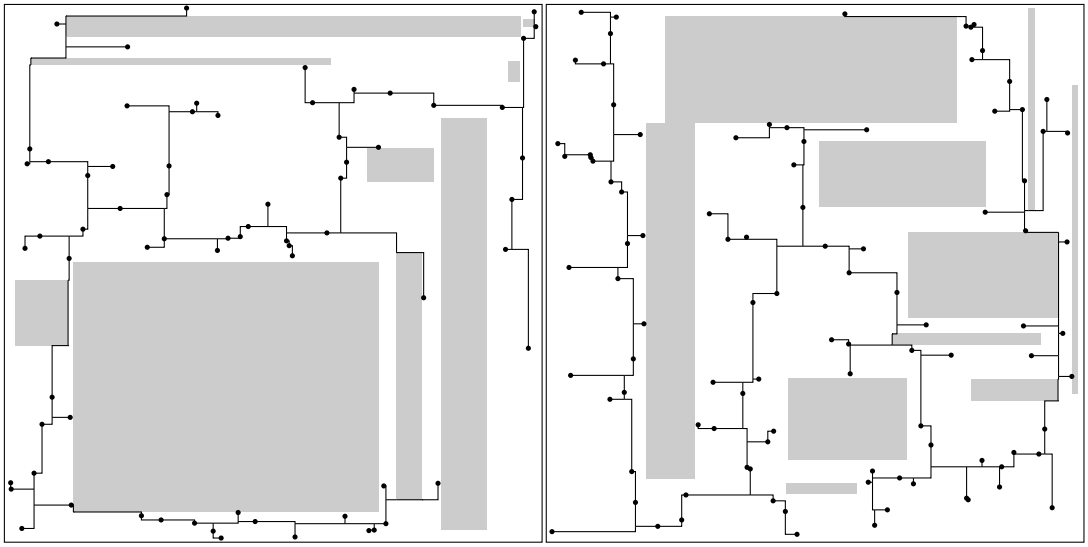
Table 3.3: Results of ObSteiner in comparison with the approach in [48].

Benchmark	ObSteiner		Huang [48]		$\frac{t_2}{t_1}$
	L_1	t_1	L_2	t_2	
RC1	25980	0.16	25980	0.58	3.63×
RC2	41350	0.52	41350	0.55	1.06×
RC3	54160	0.68	54160	0.58	0.85×
RC4	59070	0.95	59070	1.10	1.16×
RC5	74070	1.31	74070	2.09	1.60×
RC6	79714	335	-	-	-
RC7	108740	541	-	-	-
RC8	112564	24170	-	-	-
RC9	111005	14174	-	-	-
RC10	164150	176	-	-	-
RC11	230837	706	-	-	-
RT1	2146	25	-	-	-
RT2	45852	31	-	-	-
RT3	7964	840	-	-	-
RT4	9693	34521	-	-	-
RT5	51313	276621	-	-	-
IND1	604	0.11	604	0.46	4.18×
IND2	9500	0.25	9500	3.44	13.76×
IND3	600	0.19	600	1.31	6.89×
IND4	1086	0.87	1086	3.15	3.62×
IND5	1341	1.03	1341	24.73	24.01×

Benchmark	ObSteiner		Huang [48]		$\frac{t_2}{t_1}$
	L_1	t_1	L_2	t_2	
RC6_40	76946	3.20	76946	264	82.5×
RC7_40	105956	20	105956	179	8.95×
RC8_30	107833	17	107833	495	29.12×
RC9_30	106139	5.89	106139	174	29.54×
RC10_30	163050	48	163050	1463	30.48×
RT1_40	1872	0.16	1872	1.11	6.94×
RT2_30	44294	0.50	44294	45	90.00×
RT3_30	7580	1.02	7580	179	179.49×
RT4_30	7825	6.05	7825	63	10.41×
RT5_30	42879	10	42879	40	4.00×
RC6_rand_40	76840	3.03	76840	538	177.56×
RC7_rand_40	105358	14	105358	154	11.00×
RC8_rand_30	107811	5.55	107811	385	69.37×
RC9_rand_30	105875	4.44	105875	84	18.92×
RC10_rand_30	162470	147	162470	733	4.99×
RT1_rand_40	1817	0.14	1817	2.02	14.43×
RT2_rand_30	44358	0.54	44358	23	42.59×
RT3_rand_30	7595	1.04	7595	33	31.73×
RT4_rand_30	7681	4.23	7681	64	15.13×
RT5_rand_30	42821	5.26	42821	97	18.44×
Average					31.08×

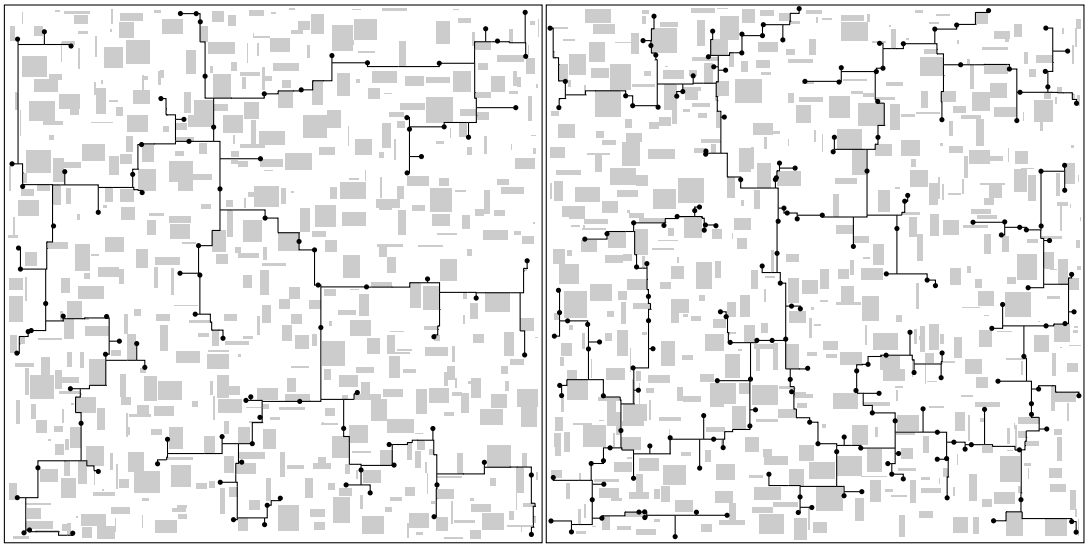
Table 3.4: Comparison of heuristics based on the OARSMT length.

Benchmark	OARSMT length (L)	Wirelength				$\frac{(X-L)}{X}$ (%)			
		Liu [6] (A)	Li [35] (B)	Ajwani [18] (C)	Liu [7] (D)	$X = A$	$X = B$	$X = C$	$X = D$
RC01	25980	26040	25980	25980	26740	0.23	0.00	0.00	2.84
RC02	41350	41570	42010	42110	42070	0.53	1.57	1.80	1.71
RC03	54160	54620	54390	56030	54550	0.84	0.42	3.34	0.71
RC04	59070	59860	59740	59720	59390	1.32	1.12	1.09	0.54
RC05	74070	74770	74650	75000	75430	0.94	0.78	1.24	1.80
RC06	79714	81854	81607	81229	81903	2.61	2.32	1.87	2.67
RC07	108740	110851	111542	110764	111752	1.90	2.51	1.83	2.70
RC08	112564	115516	115931	116047	118349	2.56	2.90	3.00	4.89
RC09	111005	113254	113460	115593	114928	1.99	2.16	3.97	3.41
RC10	164150	166970	167620	168280	167540	1.69	2.07	2.45	2.02
RC11	230837	234875	235283	234416	234097	1.72	1.89	1.53	1.39
RT1	2146	2193	2231	2191	2259	2.14	3.81	2.05	5.00
RT2	45852	47488	47297	48156	48684	3.45	3.06	4.78	5.82
RT3	7964	8231	8187	8282	8347	3.24	2.72	3.84	4.59
RT4	9693	9893	9914	10330	10221	2.02	2.23	6.17	5.17
RT5	51313	52509	52473	54598	53745	2.28	2.21	6.02	4.53
IND1	604	604	619	604	626	0.00	2.42	0.00	3.51
IND2	9500	9600	9500	9500	9700	1.04	0.00	0.00	2.06
IND3	600	600	600	600	600	0.00	0.00	0.00	0.00
IND4	1086	1092	1096	1129	1095	0.55	0.91	3.81	0.82
IND5	1341	1374	1360	1364	1364	2.40	1.40	1.69	1.69
Average						1.59	1.74	2.40	2.76



(a) (b)

Figure 3.31: The OARSMTs of (a) RC04 (b) RC05.

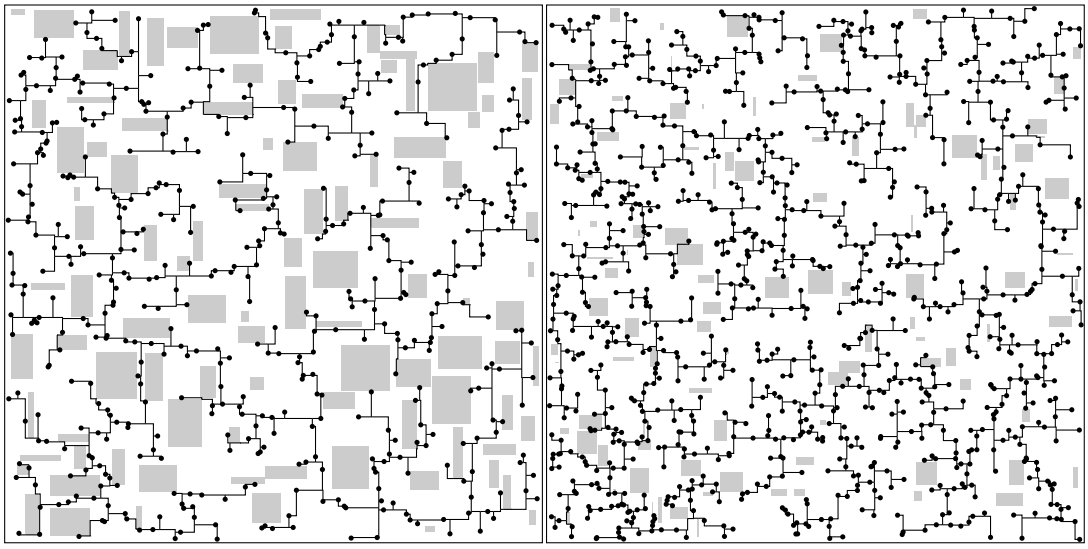


(a) (b)

Figure 3.32: The OARSMTs of (a) RC06 (b) RC07.

(a) (b)

Figure 3.33: The OARSMTs of (a) RC08 (b) RC09.

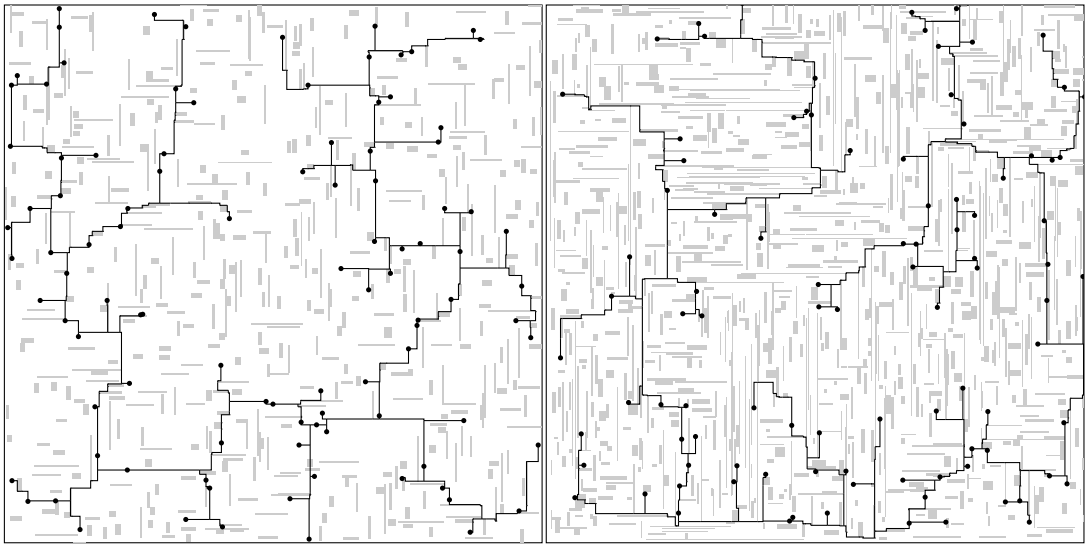


(a) (b)

Figure 3.34: The OARSMTs of (a) RC10 (b) RC11.

(a) (b)

Figure 3.35: The OARSMTs of (a) RT1 (b) RT2.



(a) (b)

Figure 3.36: The OARSMTs of (a) RT3 (b) RT4.

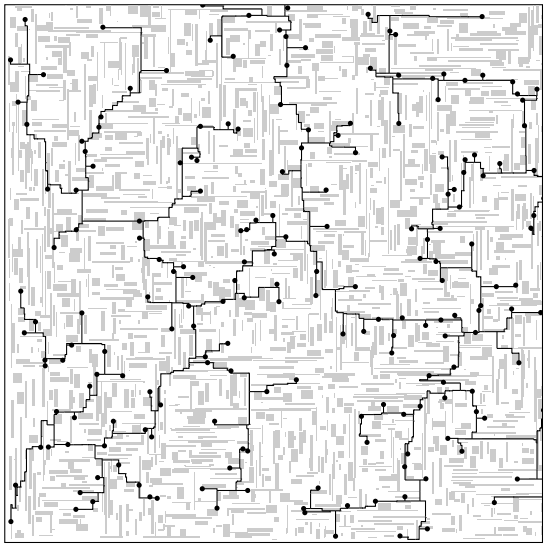


Figure 3.37: The OARSMTs of RT5.

CHAPTER 4

ObSteiner with slew constraints

Contents

4.1 Introduction

In this chapter, we study a variant of the OARSMT problem. In modern VLSI designs, obstacles usually occupy a fraction of the metal layers. Therefore, routing wires on top of obsatcles is possible. However, since buffers cannot be placed on

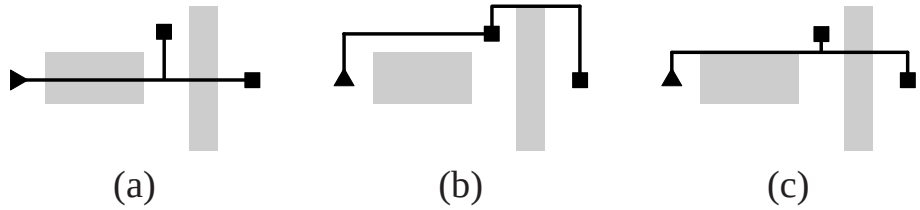


Figure 4.1: The routes of a net with a source and two sinks in the presence of obstacles.

top of any obstacle, one should be aware of the signal integrity issue and avoid routing long wires on top of obstacles that may lead to complicated post-routing electrical fixups. One way to tackle this problem is to construct an OARSMT [18, 7, 35, 36, 46, 47, 48]. However, avoiding all obstacles may result in an unnecessary resource overhead. A smarter router should be able to avoid some of the obstacles that cause problems, while allowing wires to cross the others.

Consider a problem of finding a rectilinear Steiner tree to connect a net with a source and two sinks in the presence of two obstacles, as shown in Fig. 4.1. One way is to use a RSMT as shown in Fig. 4.1(a) which is the shortest possible connection. However, there is a long wire crossing the left obstacle and this may cause signal integrity problems because no buffer can be placed on top of the obstacle. An alternative is to find an OARSMT as shown in Fig. 4.1(b). Since the tree avoids routing over any obstacle, it may take more routing resources than necessary. In comparison with these two solutions, a better way is to avoid one of the obstacles that cause problem while allow wires to cross the other, as shown in Fig. 4.1(c). This solution achieves better performance with less resource overhead.

This chapter aims at solving the RSMT problem in the presence of obstacles. In order to keep circuit performance, we impose slew constraints on the interconnects that are routed over obstacles. This is because slew is one of the most important factors in electrical correctness. Violations to the slew constraints may result in a

misleading timing analysis, and therefore degrade the performance and yield of the design. Moreover, slew constraints are more prevalent than timing constraints in the buffer insertion step. According to [44], for the majority of the nets in a design (around 90-95%), if the net's slew constraint is met, the timing constraint can be satisfied as well. Therefore, it is more important to restrict the routing on top of an obstacle to meet the slew constraints. This problem is called the OARSMT problem with slew constraints over obstacles. Since slew constraints are related to both wire length and delay, this problem is more complicated than the traditional OARSMT problem that does not consider timing. The solutions to this problem can guarantee the interconnect performance and avoid post-routing electrical fixups due to slew violations. Comparing with OARSMT, the solutions to this problem can reduce the routing resource overhead. In this thesis, we propose an exact algorithm, called ObSteiner with slew constraints, that is able to find an optimal solution embedded in the extended Hanan grid. Experimental results show that the proposed algorithm is able to reduce nearly 5% routing resources on average in comparison with the OARSMT algorithm and is also very much faster.

The rest of this chapter is organized as follows. In Section 4.2, we give a formal formulation of the problem. In Section 4.3, we present an overview of our approach. In Section 4.4, we study the structures of the trees inside obstacles in an optimal solution. Section 4.5 describes the algorithm to find the optimal solution embedded in the extended Hanan grid. Finally, experimental results are provided in Section 4.6.

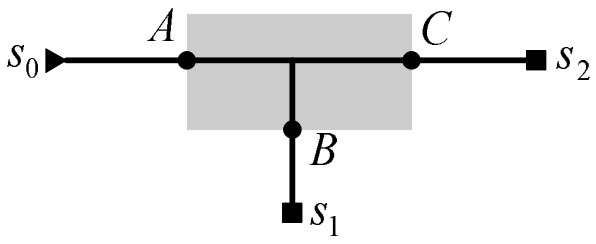


Figure 4.2: Boundary terminals on a rectilinear Steiner tree.

4.2 Problem Formulation

Given a source s_0, a set of sinks S, and a set of rectangular obstacles O, a rectilinear Steiner tree T is a tree that connects all nodes in $V = \{s_0\} \cup S$. We define a new type of nodes in T called *the boundary terminals*. A boundary terminal is a node that is on the boundary of an obstacle and has at least one of its incident lines lying over the obstacle. An example is shown in Fig. 4.2 where A, B, and C are three boundary terminals. Note that a line going along the boundary of an obstacle is considered to be outside the obstacle.[1] By splitting at the boundary terminals, a tree T can be uniquely decomposed into two sets of smaller trees either lying completely inside an obstacle or lying completely outside all obstacles. We call them internal trees and external trees and use TI and TO to denote these two sets, respectively. For example, the tree in Fig. 4.2 can be decomposed into four smaller trees in which the tree connecting A, B and C is completely inside the obstacle and the rest three trees are completely outside the obstacle. We assume that buffers can be inserted outside an obstacle. Therefore, a buffer can be inserted on a tree $t_o \in TO$ but it cannot be inserted on a tree $t_i \in TI$ except right at the leaf nodes (boundary terminals). To ensure signal integrity along the wires routed over obstacles, we impose slew constraints to the internal trees in TI.

[1] For abutted obstacles, we consider the the boundary between them as outside obstacles.

The slew rate of a signal refers to the rising or falling time of a signal. In this chapter, the slew rate is defined as the time it takes for a waveform to cross the 10% point and the 90% point. The slew model proposed in [44] is employed to compute the slew rate. We first briefly introduce this slew model. Let u_i be an upstream node, u_j be a downstream node in a tree and p be the path between them. Assume a buffer b at u_i but no buffer on p. The slew value at u_j is given by

$$S(u_j) = \sqrt{S_{b,out}(u_i)^2 + S_w(p)^2}. \tag{4.1}$$

$S_w(p)$ is the slew degradation along path p given by

$$S_w(p) = ln9 \cdot D(p), \tag{4.2}$$

where $D(p)$ is the Elmore delay from u_i to u_j. $S_{b,out}(u_i)$ is the output slew of buffer b given by

$$S_{b,out}(u_i) = R_b \cdot C(u_i) + K_b, \tag{4.3}$$

where $C(u_i)$ is the downstream capacitance at u_i, R_b is the slew resistance of b and K_b is the intrinsic slew of b. Slew constrained buffer insertion problem is to insert buffers on a routing tree such that the input slew at each buffer or sink is no greater than a constant α. In our current problem, instead of assuming a given tree, we will construct a slew-aware but length-optimal tree in the presence of obstacles.

Given an internal tree $t_i \in TI$ in a rectilinear Steiner tree T, we use u_0 to denote the source of t_i (i.e. a terminal that is closest to s_0 in T) and U_i to denote the set of sinks on t_i (i.e. the remaining terminals). Without loss of generality, in the computation of the best possible slew of t_i, we assume that a buffer will be inserted at u_0 and at each node $u \in U_i$. Note that we are not really inserting buffers there,

but just assuming the best possible buffer locations to see if violation to the slew constraint will still be caused. Therefore, the slew rate $S(u)$ at each sink $u \in U_i$ can be computed by (4.1). We define the slew of an internal tree t_i to be

$$S_{in}(t_i) = \max_{u \in U_i}\{S(u)\}. \tag{4.4}$$

As a result, the slew of an internal tree is defined as the maximum slew taking over the slew rates at all the sinks, and according to (4.1), this is related to the tree capacitance $C(u_0)$ (i.e. tree length) and the delay from the source to the sinks $D(p)$.

Based on this definition, the slew of a general rectilinear Steiner tree is defined as

$$S_{tree}(T) = \max_{t_i \in TI}\{S_{in}(t_i)\}. \tag{4.5}$$

where TI is the set of internal trees after breaking T at the boundary terminals.

At this stage, we want to focus on the routing problem to reduce the required routing resource as much as possible, while keeping the slew constraints in mind to avoid complicated post-routing electrical fixups. Therefore, the OARSMT problem with slew constraints over obstacles is formulated as follows. Given a source s_0, a set of sinks S, and a set of rectangular obstacles O, construct a rectilinear Steiner tree T that

$$\text{minimize}: \quad len(T), \tag{4.6}$$

$$\text{subject to}: \quad S_{tree}(T) \leq \alpha. \tag{4.7}$$

where $len(T)$ is the length of T and α is the slew limit [2].

[2] It should be noted that, in our implementation according to equation (4.1), we assume a uniform unit wire resistance and capacitance. Although different layer assignment can lead to different unit wire resistances and capacitances, it is acceptable to assume uniform

4.3 Overview of our approach

From the problem formulation, we can see that any optimal solution to the OARSMT problem with slew constraints over obstacles can be uniquely decomposed into a set of external trees TO and a set of internal trees TI. Therefore, one way to construct an optimal solution is to first construct its external tree candidates and internal tree candidates. This fact brings out the importance of studying the structures of the trees in TO and TI. We will show that,in an optimal solution, the trees in TI with slew constraints will follow some very simple forms. By applying existing lemmas, we can show that the trees in TO will also be very simple. Therefore, we can use a two-phase algorithm to generate an optimal solution. In the first phase, we generate a set of candidate trees in TI and a set of candidate trees in TO. In the second phase, we select and combine a subset of these trees to give an optimal solution.

4.4 Internal tree structures in an optimal solution

We have shown in the previous section that a tree T can be uniquely decomposed into two sets of smaller trees TI and TO either inside an obstacle or outside all obstacles. For the trees in TO, we only need to concern about minimizing the total wire length, since buffers can be inserted flexibly and the interconnect performance can be guaranteed. We will impose slew constraints on the trees in TI. For the trees in TI, we not only will consider the length of the tree but also handle carefully the timing, because the slew constraint is closely related to both the tree length and

unit resistance and capacitance values by taking the worst case values. This can guarantee the correctness of a solution no matter how layer assignment is done. Moreover, since obstacles usually block lower metal layers and the upper remaining layers will have similar parasitics, this assumption will not lead to a significant degradation of solution quality.

delay. This critical requirement makes previous approaches incapable of handling this new problem. Note that our internal trees are different from the Steiner trees that consider source-to-sink delay [34, 22], as we only need to consider slew constraints for the parts that overlap with obstacles. Therefore, it is possible to change the internal tree structure to move a part of the tree out of an obstacle to reduce the slew. In this section, we are interested in the possible structures of the trees in TI. We will show that, in an optimal solution, the trees in TI will follow some very simple forms. In the figures of this section, we use a solid circle to denote a boundary terminal and an empty circle to denote a Steiner point. For simplicity, we will use the term terminal instead of boundary terminal in this section.

We first make the following observations about the properties of an internal tree $t_i \in TI$ in an optimal solution.

1. t_i connects a set of boundary terminals on an obstacle and all the connected terminals have degree one in the tree.
2. One of the connected terminals is the source of t_i and all the other terminals are sinks.
3. The slew constraint is satisfied, i.e., $S_{tree}(t_i) \leq \alpha$.
4. t_i is length-optimal over all the trees connecting the same set of terminals subject to the slew constraint.

The first property is true because the set of trees TI is obtained by splitting at the boundary terminals. If there is a terminal of degree more than one, we will split the tree into two smaller trees with at most one tree in TI. The second and third properties are obviously true according to the problem formulation. The fourth property is true because if t_i is not length-optimal, we can replace t_i with a shorter tree that satisfies the slew constraint, a contradiction to the fact that t_i is in an

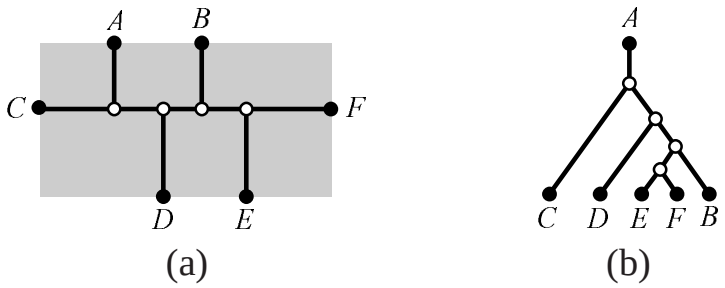


Figure 4.3: A SCIFST and its corresponding binary tree.

optimal solution.

Since there can be several length-optimal trees that satisfy the slew constraint, in order to construct a tree with better timing, we further require that t_i should have the smallest slew rate $S_{tree}(t_i)$ over all length-optimal trees connecting the same set of terminals subject to the slew constraint. That is, among all the length-optimal trees, we always prefer the one with the smallest slew rate. This is a reasonable requirement because it provides more flexibilities for the later buffer insertion step. We call the internal trees satisfying the above properties slew constrained internal full Steiner trees (SCIFSTs).

In the following, we will show that the SCIFSTs will follow some very simple structures. The proof begins with the observation that the topology of any SCIFST can be represented by a binary tree with the source as the root, all sinks as leaf nodes and all Steiner points as internal nodes. An example is shown in Fig. 4.3 where A is the source. Without loss of generality, we allow edges of zero length in the binary tree so that Steiner points with degree more than three can also be represented. As we can see, any subtree in the binary tree corresponds to a subtree in the SCIFST. Since any subtree in the binary tree is a combination of its left subtree and right subtree, we can view a subtree in a SCIFST as a combination of two smaller subtrees in the SCIFST. We will start with the smallest subtree in a SCIFST,

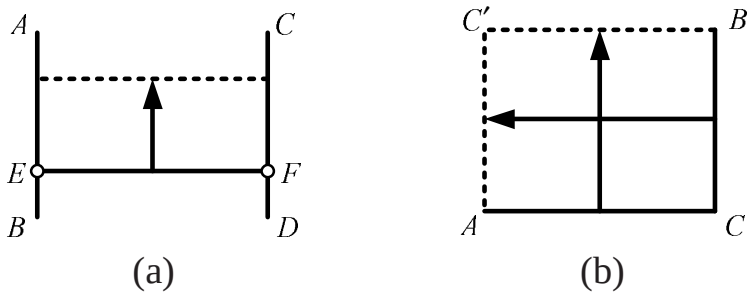


Figure 4.4: (a) Shifting (b) Flipping.

and show that these subtrees will just have some very limited structures. We then consider larger subtrees as combinations of these small subtrees and show that all subtrees in a SCIFST will be very simple, and thus leading to simple structures of the resulting SCIFSTs. In the following figures, we use an empty square to denote the root node of a subtree.

Before the proof, we introduce two operations: *shifting* and *flipping* on a tree, as shown in Fig. 4.4. Shifting a line means moving a line between two parallel lines to a new position. Flipping an edge with two perpendicular lines meeting at a corner means moving these two lines to flip the corner to the opposite side diagonally. Note that these two operations will not change the length of the tree. In the following, shifting a line towards the source in a tree means shifting the line to a position that is closer to the source by counting the distance in the tree (not geometric distance).

Lemma 4.1. *Shifting a line towards the source in a tree t will not increase the slew rate at any sink of t.*

Proof. Consider Fig. 4.4(a) and assume without loss of generality that A is closest to the source of the tree. Let l_1 be the length of AB and CD, l_2 be the length of EF, d be the distance between A and E, and $c(B)$, $c(C)$, $c(D)$ be the downstream capacitance of B, C, D, respectively. Note that the slew rates at the sinks are related

to both the tree length and delay (equation (4.1)). Since shifting will not change the length of a tree, we focus on the delay. The delay from A to B, C and D can be given as

$$\begin{aligned}
&D(A \to B) = dr_0[(2l_1 + l_2 - d)c_0 + c(B) + c(C) + c(D)] \\
&+ 0.5d^2c_0r_0 + (l_1 - d)r_0c(B) + 0.5(l_1 - d)^2c_0r_0 \\
&= dr_0[(l_1 + l_2)c_0 + c(C) + c(D)] + l_1r_0c(B) + 0.5l_1^2c_0r_0, \qquad (4.8)\\
&D(A \to D) = dr_0[(2l_1 + l_2 - d)c_0 + c(B) + c(C) + c(D)] \\
&+ 0.5d^2c_0r_0 + (l_1 - d)r_0c(D) + 0.5(l_1 - d)^2c_0r_0 \\
&+ l_2r_0(l_1c_0 + c(C) + c(D)) + 0.5r_0c_0l_2^2 \\
&= dr_0[(l_1 + l_2)c_0 + c(B) + c(C)] + l_1r_0c(D) + 0.5l_1^2c_0r_0 \\
&+ l_2r_0(l_1c_0 + c(C) + c(D)) + 0.5r_0c_0l_2^2, \qquad (4.9)\\
&D(A \to C) = dr_0[(2l_1 + l_2 - d)c_0 + c(B) + c(C) + c(D)] \\
&+ 0.5d^2c_0r_0 + l_2r_0(l_1c_0 + c(C) + c(D)) + 0.5r_0c_0l_2^2 \\
&+ dr_0c(C) + 0.5d^2c_0r_0 \\
&= dr_0[(2l_1 + l_2)c_0 + c(B) + c(C) + c(D)] \\
&+ l_2r_0(l_1c_0 + c(C) + c(D)) + 0.5r_0c_0l_2^2 + dr_0c(C), \qquad (4.10)
\end{aligned}$$

where c_0 and r_0 are the unit wire capacitance and resistance, respectively. As we can see, (4.8), (4.9), and (4.10) are all strictly increasing function with respect to d. If we shift EF up (i.e. towards the source), d will decrease and thus the delays from A to B, C and D will all decrease. Therefore, the delays of all downstream sinks of A will also decrease. Since the tree length is not changed, the delays of the sinks that are not downstream of A will be the same. As a result, the slew rates at all downstream sinks of A will be reduced and the slew rates of all the other sinks

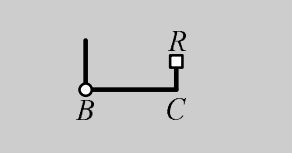


Figure 4.5: An invalid structure in a SCIFST.

that are not downstream of A will remain unchanged. □

According to Lemma 4.1, shifting a line in a tree towards the source will reduce the slew rates at some sinks while keeping the slew rates of the remaining sinks unchanged. Note that the slew of an internal tree is defined as the maximum slew taking over the slew rates at all its sinks (equation (4.5)). Therefore, shifting a line towards the source may or may not reduce the slew of the tree. Without loss of generality, we further require that all lines (that can be shifted) have been shifted to a position that is closest to the source in a SCIFST. Note that this will not change the optimality of the resulting solution.

Lemma 4.2. *A subtree in a SCIFST will not contain the structure as shown in Fig. 4.5 where B is a Steiner point and R is the root of the subtree.*

Proof. Since the root node is to be connected to the source (i.e. the root node is a point in the subtree that is closest to the source of the SCIFST), by Lemma 4.1, we can shift BC up (i.e. towards the source) to reduce the slew rates at downstream sinks, an absurdity. As a result, the structure cannot exist. □

Lemma 4.3. *In a SCIFST, a terminal must be connected to a Steiner point or another terminal by a straight line that is perpendicular to the boundary on which the terminal is located.*

Proof. Assume the contrary that in a SCIFST t, a terminal A is connected to a Steiner point B (or another terminal) through a corner C, as shown in Fig. 4.6. We

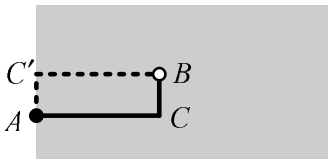


Figure 4.6: The structure when a terminal connected to a Steiner point through a corner.

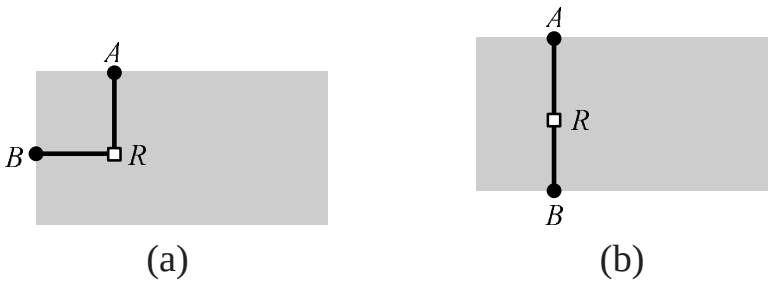


Figure 4.7: Possible structures of a subtree of two terminals in a SCIFST.

can flip AB and move the corner from C to C'. Now t becomes a new tree t' that consists of an external tree (AC') and an internal tree t_1. Note that C' now is also a boundary terminal and we can insert a buffer there. According to equation (4.5), slew $S_{tree}(t')$ of t' is equal to $S_{in}(t_1)$ which is smaller than the slew $S_{tree}(t)$ of t (assuming that buffers will be inserted at the boundary terminals). This violates the last property of SCIFST that the slew rate is the minimum possible one. As a result, the statement is true. □

Lemma 4.4. *In a SCIFST, a subtree of two terminals must be one of the trees as shown in Fig. 4.7.*

Proof. Note that the root node of a subtree in a SCIFST t must be an internal node inside the blockage to be connected to the source. By Lemma 4.3, if the two terminals are both located on a horizontal or a vertical boundary of the obstacle, they must have the same x-coordinate or y-coordinate and the tree structure must be the one as shown in Fig. 4.7(b). If the two terminals are located on a horizontal

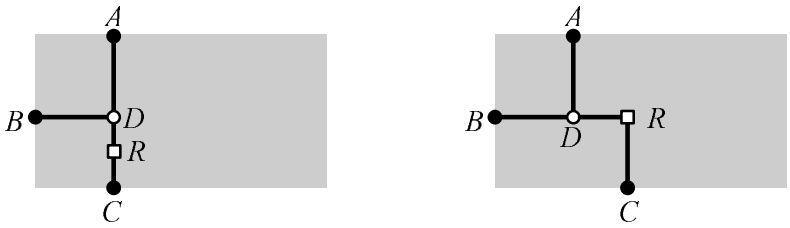


Figure 4.8: Possible structures of a subtree of three terminals in a SCIFST.

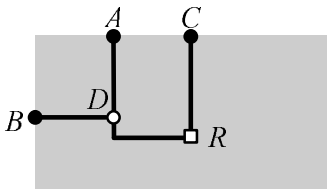


Figure 4.9: Invalid structures when Fig. 4.7(a) is combined with a terminal.

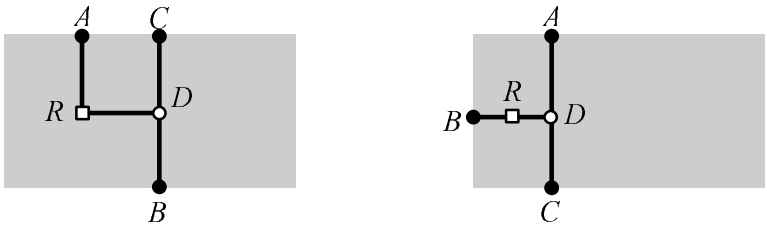
and a vertical boundary of the obstacle, according to Lemma 4.3, they must be connected by the root node of the subtree as shown in Fig. 4.7(a). □

Corollary 4.1. *In a SCIFST, any subtree must be connecting terminals located on at least two different boundaries of the obstacle.*

Proof. Consider a subtree connecting two terminals. By Lemma 4.4, it must be one of the trees as shown in Fig. 4.7. Therefore, it connects two terminals located on two different boundaries of the obstacle. Since any subtree of more than two terminals must contain at least one subtree of two terminals, the statement is true. □

Lemma 4.5. *In a SCIFST, a subtree of three terminals must be one of the trees as shown in Fig. 4.8.*

Proof. A subtree of three terminals must be a combination of a subtree of two terminals (as shown in Fig. 4.7) with another terminal.



(a) (b)

Figure 4.10: Invalid structures when Fig. 4.7(b) is combined with a terminal.

(1) Consider the case when Fig. 4.7(a) is combined with a terminal. By Lemma 4.2 and Lemma 4.3, the combined tree must be one of the subtrees as shown in Fig. 4.9 or Fig. 4.8. For Fig. 4.9, we can delete DR, connect AC and flip AB to the boundary of the obstacle. The SCIFST t becomes another tree t' that consists of one smaller internal t_1 and two external trees connecting AC and AB respectively. First of all, the length of t' will not be longer than the original SCIFST. Besides, the slew of t' is smaller than the slew of t, an absurdity. Therefore, the only possible structures are shown in Fig. 4.8. Note that in Fig. 4.8(a), the root node can be anywhere on AC except at A and C, and in Fig. 4.8(b) the root can be anywhere on RC except at C.

(2) Consider the case when Fig. 4.7(b) is combined with a terminal. By Lemma 4.2 and Lemma 4.3, the combined tree must be one of the trees as shown in Fig. 4.10. We first consider Fig. 4.10(a). If AR is longer than RD, we can remove AR and connect AC to obtain a shorter tree with smaller slew, an absurdity. If AR is not longer than RD, we can remove RD and connect AC. The original SCIFST t becomes a new tree t' that consists of two smaller internal trees t_1, t_2 that are connected by an external tree AC. The total wire length of the new tree will remain unchanged. In t', for the internal tree t_1 connecting sink A, we can easily verify that the slew rates at all sinks will be reduced. For the internal tree t_2 connecting

C and B, C will become the source of t_2. Since AR is not longer than RD, CD will not be longer than RD implying that the delay from C to B will be smaller than the delay from R to B in t. Therefore, the slew rate at sink B will also be reduced. As a result, $S_{tree}(t') = \max\{S_{in}(t_1), S_{in}(t_2)\}$ will be smaller than $S_{tree}(t)$ and Fig. 4.10(a) is not a valid subtree in a SCIFST. Consider Fig. 4.10(b). R cannot be connected to the source through a line going up, or otherwise we can shift RD up (i.e. towards to source) to reduce the slew rates at downstream sinks. R cannot be connected to the source through a line going down either, or otherwise we can shift RD down to reduce the slew rates at downstream sinks. Therefore, Fig. 4.10(b) is also not a valid subtree in a SCIFST. Note that in Fig. 4.10(b), if the root node is right at D, we can consider the tree as a combination of Fig. 4.7(a) with another terminal instead and all such redundant cases will not be discussed in the proofs.

As a result, a subtree of three terminals can only be one of the trees as shown in Fig. 4.8. □

Corollary 4.2. *In a SCIFST, a subtree of three terminals must be connecting three terminals on three different boundaries of the obstacle.*

Lemma 4.6. *In a SCIFST, a subtree of four terminals must be one of the trees as shown in Fig. 4.11.*

Proof. A subtree of four terminals can be a combination of two subtrees of two terminals or a combination of a subtree of three terminals with another terminal.

(1) Considering a subtree of four terminals as a combination of two subtrees of two terminals, by Lemma 4.2 and Lemma 4.4, the combined tree must be one of the trees as shown in Fig. 4.12 or Fig. 4.11(a)-(c). Consider Fig. 4.12(a). R cannot be connected to the source through a line going down, or otherwise we can shift RF down (i.e. towards to source) to reduce the slew, an absurdity. Similarly, R

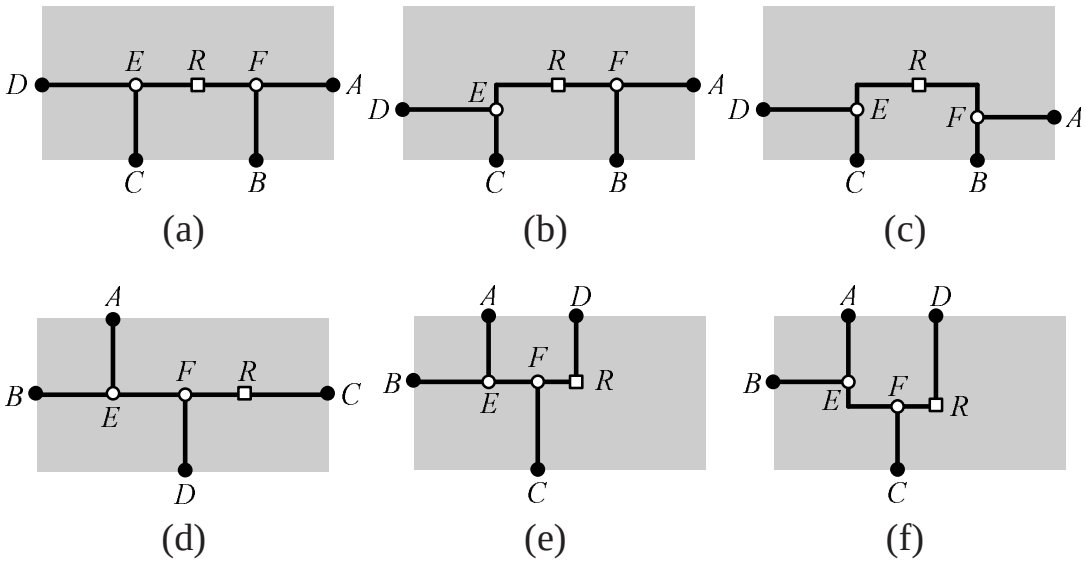


Figure 4.11: Possible structures of a subtree of four terminals in a SCIFST.

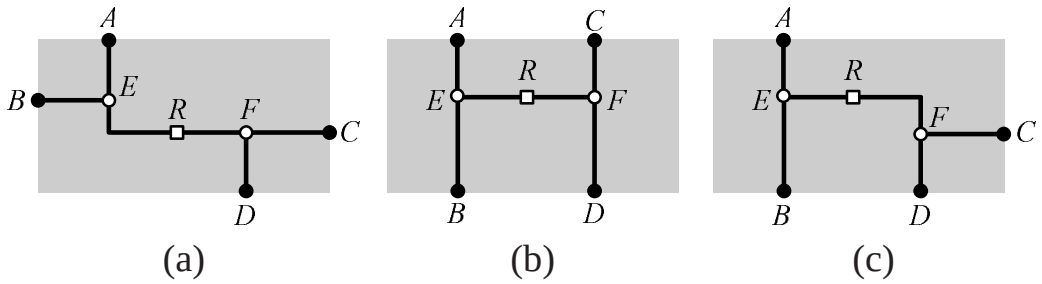


Figure 4.12: Invalid structures when two subtrees of two terminals are combined.

cannot be connected to the source through a line going up or left (if we flip *RE*). Therefore, Fig. 4.12(a) is invalid. For similar reason, in Fig. 4.12(b), *R* cannot be connected to the source through a line going up or down, and thus Fig. 4.12(b) is invalid. In Fig. 4.12(c), *R* cannot be connected to the source through a line going up, down, or left (if we flip *RF*), and thus Fig. 4.12(c) is invalid. As a result, if a subtree of four terminals is a combination of two subtrees of two terminals, it must be in the form as shown in Fig. 4.11(a)-(c). Note that in Fig. 4.11(a), the root node can be anywhere on EF except right at point E and F, or otherwise we can delete EF and connect BC to reduce the slew. For similar reason, in Fig. 4.11(a)-(b), the

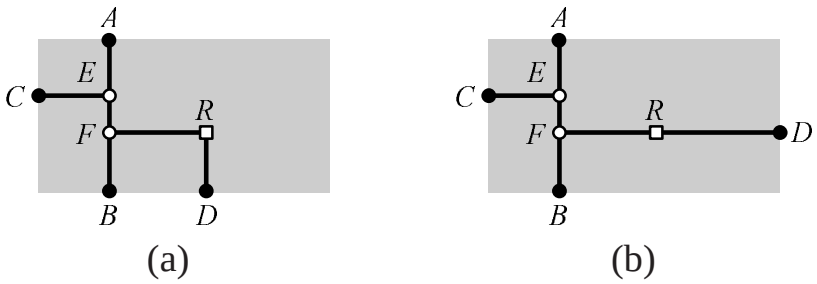


Figure 4.13: The subtree structures when Fig. 4.8(a) is combined with a terminal.

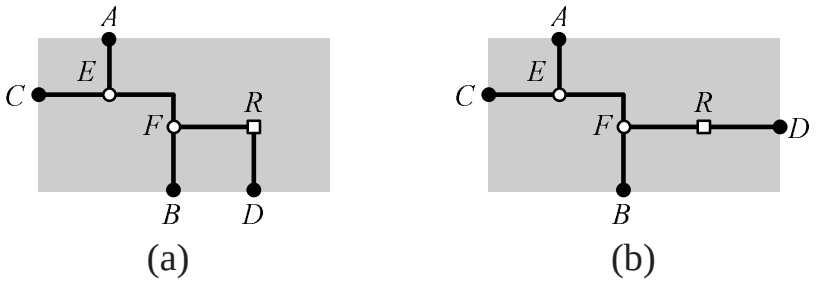


Figure 4.14: Invalid subtree structures when Fig. 4.8(b) is combined with a terminal.

root can be anywhere on EF except the points that have the same x-coordinate as E or F.

(2) Consider a subtree of four terminals as a combination of a subtree of three terminals with another terminal.

Firstly, we consider the subtree of three terminals as shown in Fig. 4.8(a). R cannot be connected to the source through a line going left, or otherwise we can shift RD left to reduce the slew. Therefore, by Lemma 4.2 and Lemma 4.3, the combined tree must be in the form as shown in Fig. 4.13. Fig. 4.13(a) is invalid and the reason is the same as why Fig. 4.10(a) is invalid. Fig. 4.13(b) is invalid because R cannot be connected to the source through a line going up or down. Therefore, Fig. 4.8(a) cannot be combined to form a subtree of four terminals in a SCIFST.

Secondly, we consider the subtree of three terminals as shown in Fig. 4.8(b). R

cannot be connected to the source through a line going left (similar to Fig. 4.8(a)) or up (or we can shift RE up), and therefore, by Lemma 4.2 and Lemma 4.3 the combined tree must be as shown in Fig. 4.14 or Fig. 4.11(d)-(f). For the same reasons as Fig. 4.13(a) and Fig. 4.13(b), Fig. 4.14(a) and Fig. 4.14(b) are both invalid. As a result, if a subtree of four terminals is a combination of a subtree of three terminals with another terminal, it must be in the form as shown in Fig. 4.11(d)-(f). Note that in Fig. 4.11(d), the root node can be anywhere on FC except at C and F. In Fig. 4.11(e)-(f), the root can only be at R, or otherwise we can flip RF and shift EF up (i.e. towards the source). □

Corollary 4.3. *In a SCIFST, a subtree of four terminals must be connecting four terminals on at least three different boundaries of the obstacle.*

Corollary 4.4. *In a SCIFST, a subtree of more than two terminals must be connecting terminals located on at least three different boundaries of the obstacle.*

Proof. By Corollary 4.2 and Corollary 4.3, a subtree of three or four terminals must be connecting terminals located on at least three different boundaries of the obstacle. Moreover, a subtree of five terminals must be connecting five terminals located on at least three different boundaries, since it must contain at least one subtree of three or four terminals. We assume that up to a subtree of n terminals, the statement is still true. Since a subtree of $n+1$ terminals must contain at least one subtree of more than two terminals, its terminals must located on at least three different boundaries of the obstacle. Therefore, by induction, the statement is true. □

Lemma 4.7. *In a SCIFST, a subtree of five terminals must be one of the trees as shown in Fig. 4.15.*

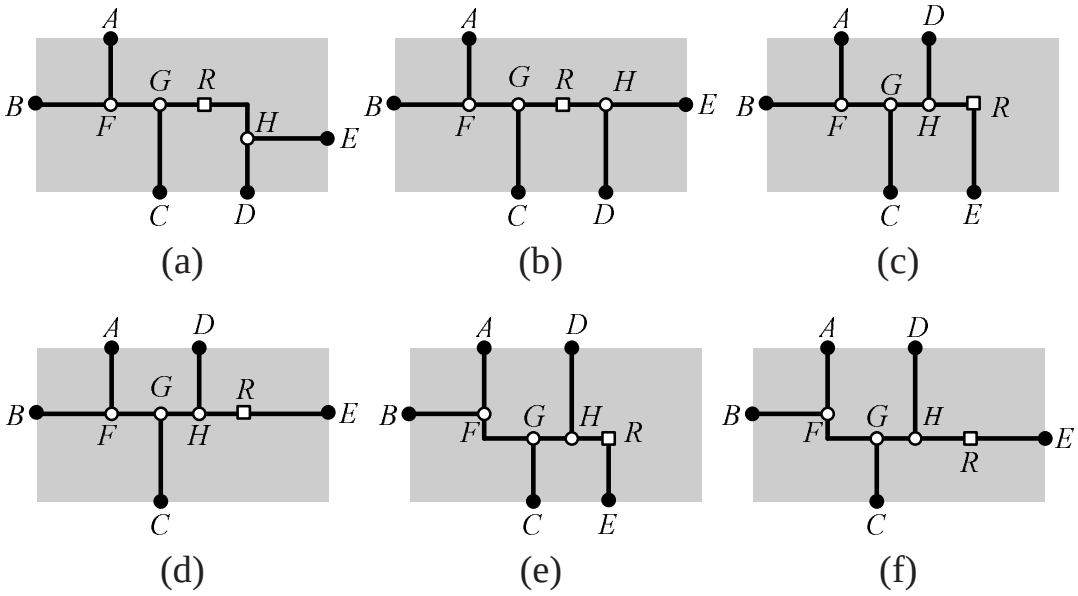


Figure 4.15: Possible structures of a subtree of five terminals in a SCIFST.

Proof. A subtree of five terminals can be a combination of a subtree of two terminals with a subtree of three terminals or a combination of a subtree of four terminals with another terminal.

(1) Consider a subtree of five terminals as a combination of a subtree of three terminals as shown in Fig. 4.8 with a subtree of two terminals. Fig. 4.8(a) cannot be combined with any subtree of two terminals as shown in Fig. 4.7. The reason is the same as why Fig. 4.7(b) cannot be combined with any subtree of two terminals. Fig. 4.8(b) cannot be combined with Fig. 4.7(b) to form a subtree of five terminals, and the reason is the same as why Fig. 4.7(a) cannot be combined with Fig. 4.7(b). Therefore, the only possible case is when Fig. 4.8(b) is combined with Fig. 4.7(a). By Lemma 4.2, the combined tree must be in the form as shown in Fig. 4.17 or Fig. 4.15(a)-(b). Fig. 4.17(a) is invalid because *R* cannot be connected to the source through a line going up, down, or right (if we flip *RG*). Similarly, Fig. 4.17(b) is invalid as well. As a result, if a subtree of five terminals is a combination of a

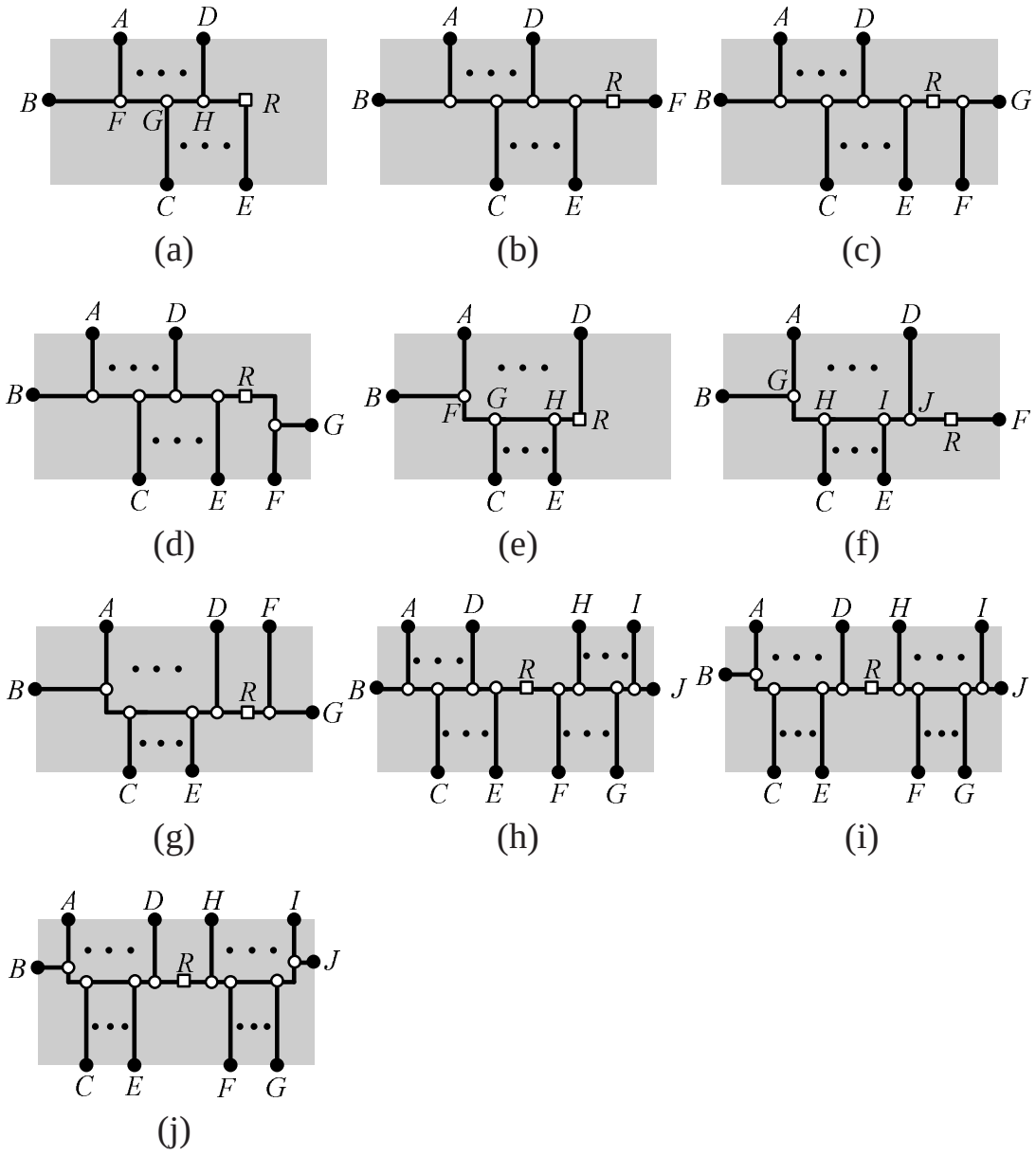


Figure 4.16: Possible structures of a subtree of more than five terminals in a SCIFST.

subtree of three terminals with a subtree of two terminals, the combined tree must be in the form as shown in Fig. 4.15(a)-(b). Note that in Fig. 4.15(a)-(b), the root node can be anywhere on *GH* except the points that have the same *x*-coordinate as *G* or *H*, or otherwise we can delete either *GR* or *RH* and connect *CD* to reduce the

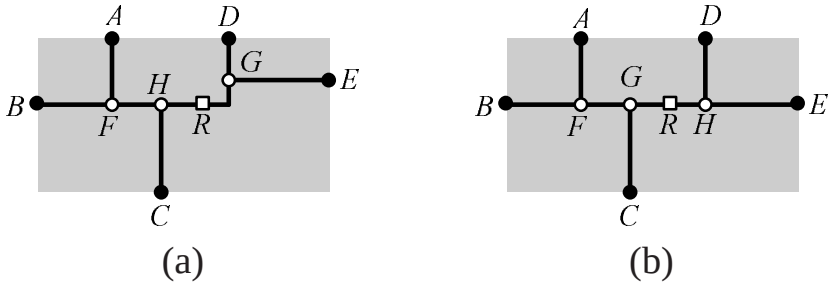


Figure 4.17: Invalid subtree structures when Fig. 4.8(b) is combined with Fig. 4.7(a).

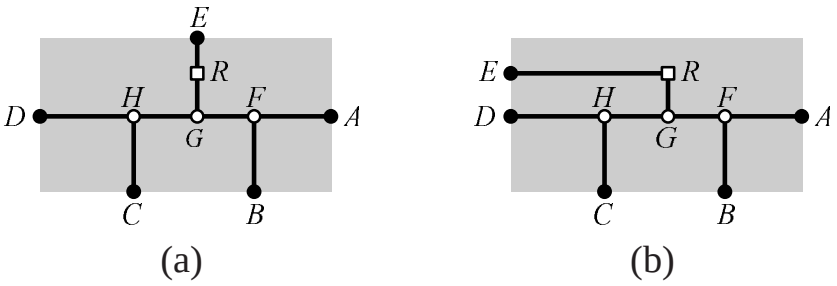


Figure 4.18: Invalid subtree structures when Fig. 4.11(a) is combined with a terminal.

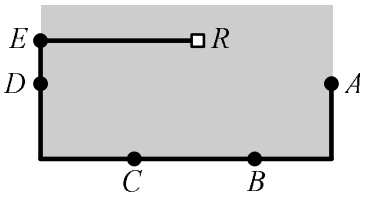


Figure 4.19: A subtree structure that can be obtained from Fig. 4.18(b).

slew.

(2) Consider a subtree of five terminals as a combination of a subtree of four terminals as shown in Fig. 4.11 with another terminal.

Firstly, we consider Fig. 4.11(a). Since R cannot be connected to the source by a line going down, by Lemma 4.2, the combined tree must be in the form as shown in Fig. 4.18. Fig. 4.18(a) is invalid because R cannot be connected to the source through a line going left or right. For Fig. 4.18(b), we can delete RG, HG, FG, and connect ED, BC to change the subtree to Fig. 4.19 with equal length and reduce the

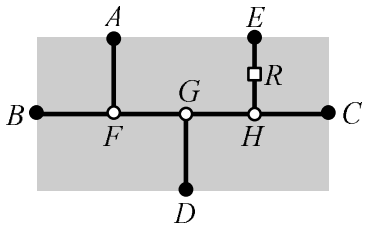


Figure 4.20: The subtree structure when Fig. 4.11(d) is combined with a terminal.

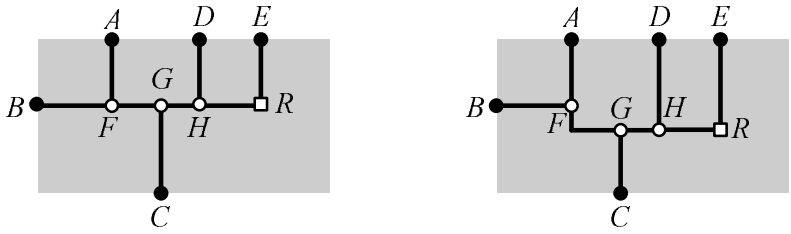


Figure 4.21: Invalid subtree structures when Fig. 4.11(e) or Fig. 4.11(f) is combined with a terminal.

slew of the tree, an absurdity. Therefore, Fig. 4.11(a) cannot be combined to form a subtree of five terminals, and neither do Fig. 4.11(b) and Fig. 4.11(c) for similar reasons.

Secondly, we consider when Fig. 4.11(d) is combined with a terminal. R cannot be connected to the source through a line going down, and therefore, by Lemma 4.2, the combined tree must be in the form as shown in Fig. 4.20. However, the combined tree is invalid because R cannot be connected to the source through a line going left or right.

Thirdly, we consider when Fig. 4.11(e) is combined with a terminal. This case is similar to the case when Fig. 4.8(b) is combined with a terminal. Since R cannot be connected to the source through a line going down, by Lemma 4.2, the combined tree must be in the form as shown in Fig. 4.21(a) or Fig. 4.15(c)-(d). However, Fig. 4.21(a) is invalid for we can delete HR and connect DE (similar to Fig 4.14(a)). Therefore, the only possible cases are Fig. 4.15(c)-(d). Note that in Fig. 4.15(c) the

root node can only be at R, and in Fig. 4.15(d) the root node can be anywhere on HE expect at H and E.

Finally, we consider when Fig. 4.11(f) is combined with a terminal. This case is similar to the case we discussed above. The combined tree must be in the form as shown in Fig. 4.21(b) or Fig. 4.15(e)-(f). However, Fig. 4.21(b) is invalid. Therefore, the only possible cases are Fig. 4.15(e)-(f). Note that in Fig. 4.15(e) the root node can only be at R and in Fig. 4.15(f), the root node can be anywhere on HE expect at H and E. □

Lemma 4.8. *In a SCIFST, a subtree of more than five terminals must be one of the trees as shown in Fig. 4.22.*

Proof. Without loss of generality, we can generalize Fig. 4.8(b), Fig. 4.11(e), and Fig. 4.15(c) as Fig. 4.22(a) that consists of a single line and alternating incident segments connecting to the terminals. We call this line a Steiner chain. We can also generalize Fig. 4.11(f) and Fig. 4.15(e) as Fig. 4.22(e). The only difference between Fig. 4.22(a) and (b) is that in the Steiner chain in Fig. 4.22(b), the first two Steiner points are connected by a corner.

To prove this lemma, we first prove that some of the subtrees cannot be grown to larger subtrees. Consider Fig. 4.11(d). We have already shown that Fig. 4.11(d) cannot be combined with a terminal. By Corollary 4.4, Fig. 4.11(d) can only be combined with a subtree of two terminals as shown in Fig 4.7(a), or otherwise there will be intersection. The combined tree must be in the form as shown in Fig. 4.23 which is invalid for R cannot be connected to the source by a line going left, right, or up. Therefore, Fig. 4.11(d) cannot be combined to form a larger subtree. For similar reason, Fig. 4.15(a), Fig. 4.15(b), Fig. 4.15(d) and Fig. 4.15(f) cannot be grown to a larger subtree either.

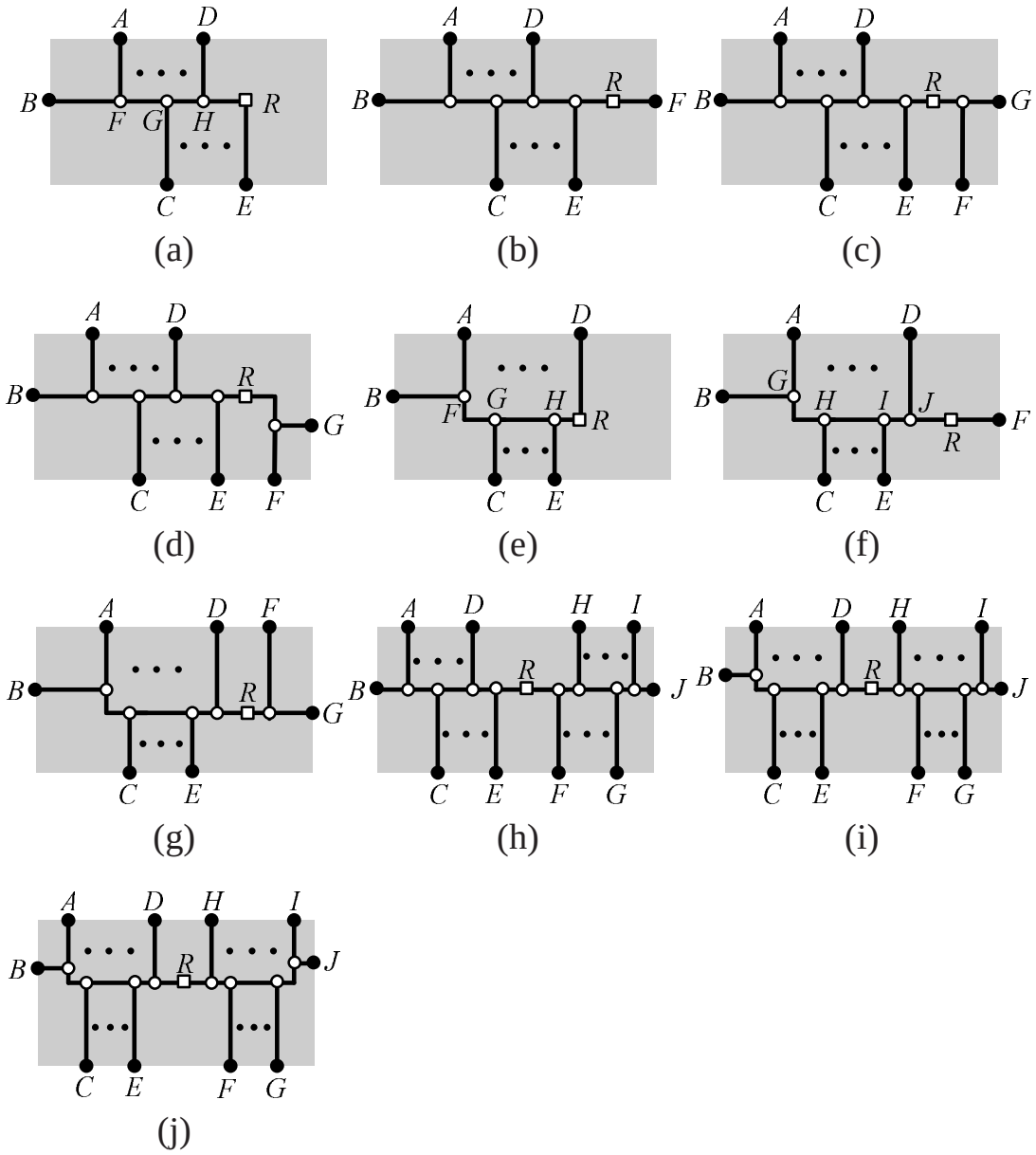


Figure 4.22: Possible structures of a subtree of more than five terminals in a SCIFST.

Secondly, we eliminate some impossible combinations. Consider Fig. 4.11(c). If Fig. 4.11(c) is combined with a subtree of two terminals, by Lemma 4.2, the combined tree must be in the form as shown in Fig. 4.24(a) or (b). Fig. 4.24(a) is invalid for R cannot be connected to the source through a line going left or

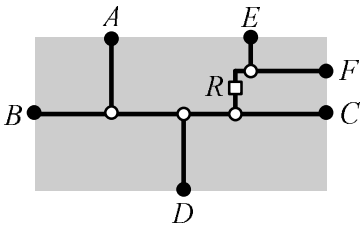


Figure 4.23: The subtree structure when Fig. 4.11(d) is combined with Fig 4.7(a).

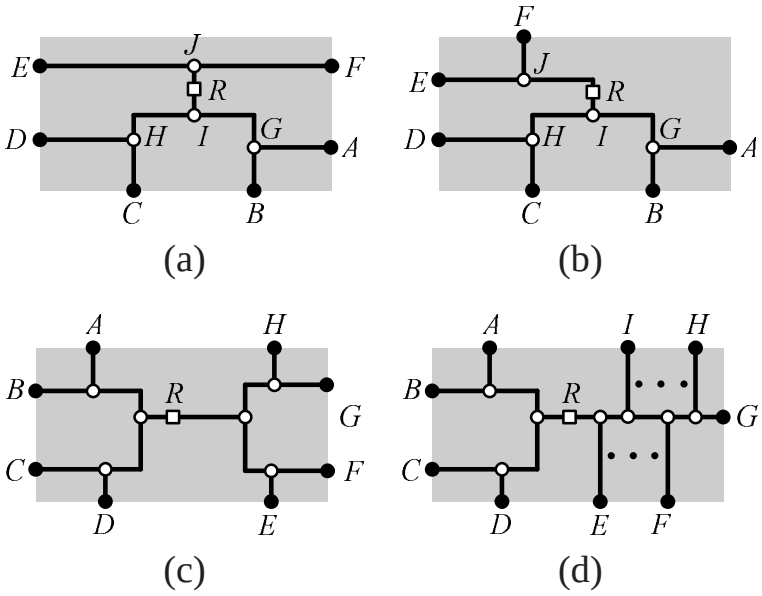


Figure 4.24: The subtree structures when Fig. 4.11(c) is combined with another subtree.

right. Fig. 4.24(b) is invalid for *R* cannot be connected to the source through a line going left, right, or up. For similar reason Fig. 4.11(c) cannot be combined with Fig. 4.8(a) either. If two subtrees as shown in Fig. 4.11(c) are combined together, the combined tree will be in the form as shown in Fig. 4.24(c). The combined tree is invalid for *R* cannot be connected to the source through a line going left or right. For similar reason, Fig. 4.11(c) cannot be combined with Fig. 4.11(a) or Fig. 4.11(b). Consider a combination of Fig. 4.11(c) with Fig. 4.22(a). The combined tree must be as shown in Fig 4.24(d). However, this subtree is invalid for *R* cannot be connected to the source through a line going up or down. For the same

reason, Fig. 4.11(c) cannot be combined with Fig. 4.22(e). Therefore, Fig. 4.11(c) cannot be combined with all the subtrees we have enumerated before. Fig. 4.11(a) and Fig. 4.11(b) can be discussed in the same way and they cannot be combined with all the subtrees we have enumerated either.

Thirdly, we will show that the possible combinations will lead to subtrees as shown in Fig. 4.22 among which only Fig. 4.22(a) and Fig. 4.22(e) can be combined to form larger subtrees. Note that the remaining ways to form a subtree of more than five terminals are: (1) to combine Fig. 4.22(a) or Fig. 4.22(e) with Fig. 4.7(a), Fig. 4.7(b), Fig. 4.8(a), or a terminal, (2) to combined two Fig. 4.22(a), two Fig. 4.22(b), or Fig. 4.22(a) with Fig. 4.22(e).

The case when Fig. 4.22(a) is combined with Fig. 4.7(a), Fig. 4.7(b), Fig. 4.8(a), or a terminal is similar to the case when Fig. 4.8(b) is combined with one of these subtrees. For the same reasons as discussed in Lemma 4.6 and Lemma 4.7, the combined tree must be in the form as shown in Fig. 4.22(a)-(d). Note that if Fig. 4.22(a) is combined with a terminal, one of the resulting subtrees can be generalized as Fig. 4.22(a) itself. Moreover, for the same reason as Fig. 4.11(d), Fig. 4.22(b)-(d) cannot be combined to form a larger subtree. The case when Fig. 4.22(e) is combined with Fig. 4.7(a), Fig. 4.7(b), or Fig. 4.8(a) can be discussed similarly. The combined tree must be in the form as shown in Fig. 4.22(e)-(g), among which only Fig. 4.22(e) can be combined to form a larger subtree.

Consider when two subtrees as shown in Fig. 4.22(a) are combined together. The combined tree must be in the form as shown in Fig. 4.25(a) or Fig. 4.22(h). Fig. 4.25(a) is invalid for R cannot be connected to the source through a line going up or down. Therefore, the only possible subtree is Fig. 4.22(h). We then prove that Fig. 4.22(h) cannot be combined to form a larger subtree. By Corollary 4.1, it can only be combined with a terminal. Since R cannot be connected to

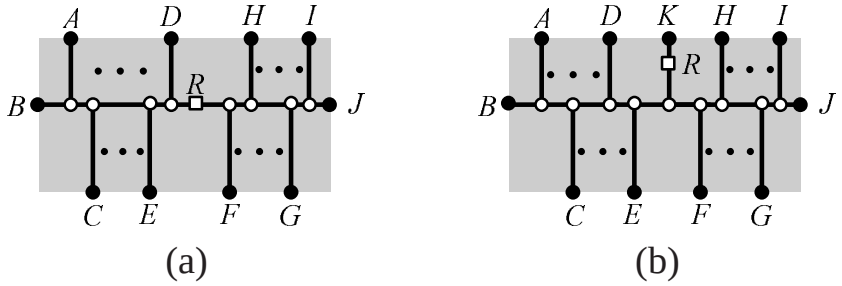


Figure 4.25: The case when two subtrees as shown in Fig. 4.22(a) are combined together.

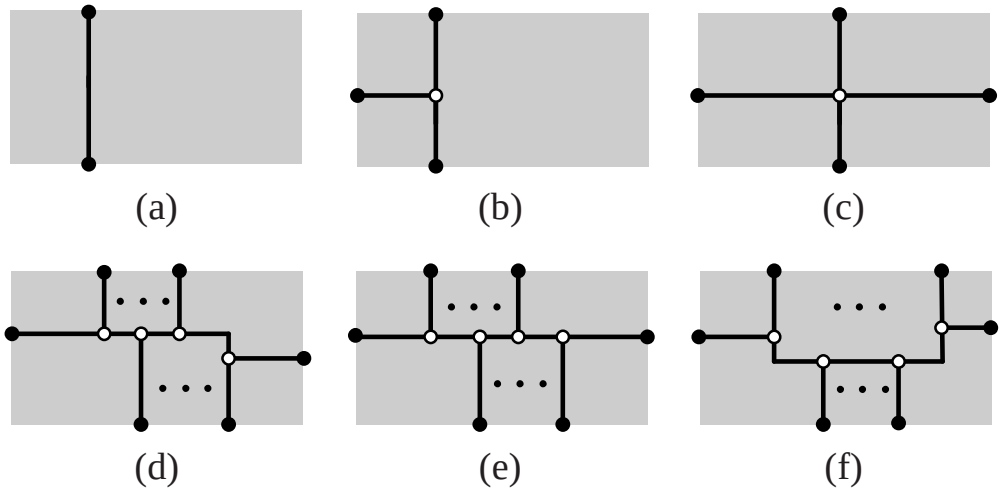


Figure 4.26: Possible structures of SCIFSTs.

the source through a line going down, the combined tree must be in the form as shown in Fig. 4.25(b). However, this tree is invalid for R cannot be connected to the source. The case when two Fig. 4.22(e) are combined together and the case when Fig. 4.22(a) are combined with Fig. 4.22(e) can be discussed similarly. The resulting trees will be in the form as shown in Fig. 4.22(i)-(j). Moreover, both trees cannot be grown to larger subtrees.

Finally, since all possible combinations of subtrees, that can be grown to larger subtree, can all be generalized as Fig. 4.22(a) or Fig. 4.22(e), we can conclude that a subtree of more than five terminals must be in the form as shown in Fig. 4.22. □

Theorem 4.1. *A SCIFST must have one of the structures as shown in Fig. 4.26.*

Proof. Firstly, consider a SCIFST connecting the source with one sink only. By Lemma 4.3, the SCIFST must be in the form as shown in Fig. 4.26(a). For the rest SCIFSTs, they can be constructed by connecting a subtree as deduced by the above lemmas to the source. By Lemma 4.3, to form a SCIFST, a subtree should be connected to the source directly by a straight line. Consider a SCIFST connecting the source with two sinks. By Lemma 4.4, the final SCIFST (connecting the root of a 2-terminal subtree to the source) must be in the form as shown in Fig. 4.26(b). Consider a SCIFST connecting the source with three sinks. If Fig. 4.8(a) is connected to the source and the root node is at D, the resulting SCIFST will be in the form as shown in Fig. 4.26(c). If the root node is not at D, since the root cannot be connected to the source through a line going left, the subtree can only be connected to the source on the right boundary. The resulting SCIFST can be generalized as the form shown in Fig. 4.26(e). Similarly, if Fig. 4.8(b) is connected to the source, the resulting SCIFSTs can either be generalized as Fig. 4.26(d) or Fig. 4.26(e). Finally, we can follow the same way to analyze the rest cases and find that a complete SCIFST must be in one of the structures as shown in Fig 4.26. For example, subtrees Fig. 4.11(a)(d)(e), Fig. 4.15(b)(c)(d), and Fig. 4.22(a)(b)(c)(h) will all lead to the SCIFST as shown in 4.26(e). □

Theorem 1 shows that the SCIFSTs (internal trees in an optimal solution) will follow some simple structures. This result leads to a two-phase algorithm presented in the next section.

4.5 Algorithm

We have shown in the previous section that, in an optimal solution of the OARSMT problem with slew constraints over obstacles, the trees in TI will follow some very simple structures. Now, we consider the external trees in TO in an optimal solution. We can further divide the trees in TO into smaller trees by splitting at sinks and the source with degree more than one. Then, a tree $t_o \in TO$ will have the following properties.

1. t_o connects a set of nodes in V and boundary terminals, and all the connected nodes have degree one in the tree.
2. One of the connected nodes in the tree is a source and all the other nodes are sinks.
3. t_o is length-optimal over all the trees connecting the same set of nodes.

By applying the lemmas proposed in [48], it can be shown that the trees with the above properties will also follow some simple structures as shown in Fig. 4.27. In this chapter, we call these trees external full Steiner trees (EFSTs). Therefore, one way to construct an optimal OARSMT with slew constraints over obstacles is to first construct a set of candidate SCIFSTs in TI according to Fig. 4.26 and a set of candidate EFSTs in TO according to Fig. 4.27, and then select and combine a subset of them.

However, this process is still difficult to realize, because the locations of the boundary terminals are not fixed. Therefore, in this chapter, we aim at providing an optimal solution that is embedded in the extended Hanan grid. Considering a set of nodes $V = S \cup \{s_0\}$ and a set of rectangular obstacles O, the extended Hanan grid is a grid graph formed by constructing vertical and horizontal lines through each node in V and each corner of the obstacles. By restricting the solution to the

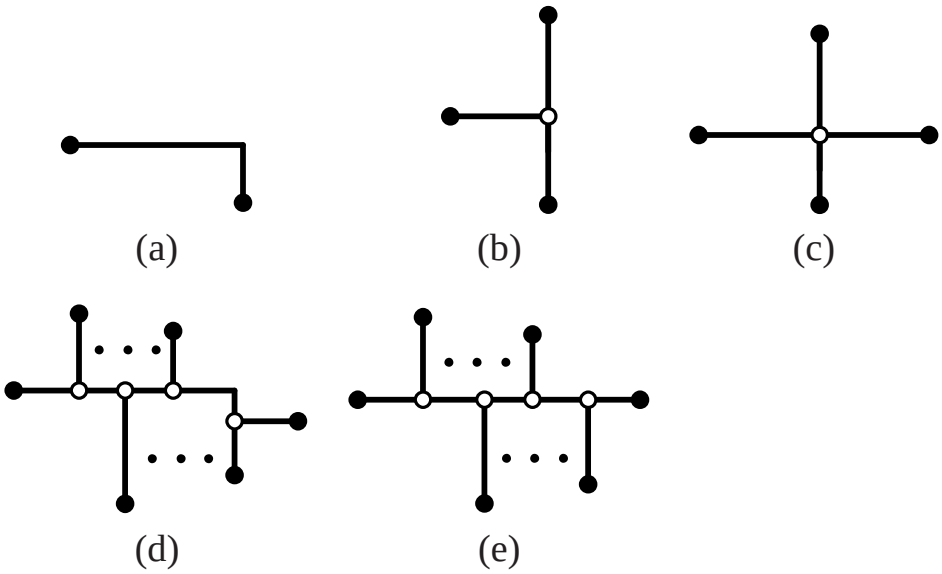


Figure 4.27: Possible structures of EFSTs.

extended Hanan grid, the boundary terminals are the grid intersection points on the boundaries of the obstacles. We use a set B to denote these boundary terminals. In this way, we can realize a two-phase algorithm to construct an optimal solution as follows.

4.5.1 EFST and SCIFST generation

The first phase is to generate a set of EFSTs and a set of SCIFSTs.

We first consider the construction of EFSTs. Note that EFSTs are very similar to the full Steiner trees (FSTs) defined in the RSMT problem [61]. However, there are two critical differences. Firstly, EFSTs are trees that connect the nodes in $V \cup B$, while FSTs are trees that connect the nodes in V only. Secondly, EFSTs are directed, while FSTs are not. The reason we need direction is that, in the computation of slew rate, we need to calculate the delay of a tree and we must have the source and sink information. A feasible internal tree (over obstacle) with

a specific terminal as the source may fail to meet the slew constraint if the source is changed to another terminal. Therefore, in order to ensure a feasible solution, we need to keep the source/sink information and the signal flow directions in both EFSTs and SCIFSTs. In general, we can modify the algorithm described in [61] to generate EFSTs. However, we need to apply different screening tests to prune useless trees taking into consideration the direction information.

We then consider the construction of SCIFSTs. Since each SCIFST is completely within an obstacle, for each obstacle in O, we will generate a set of SCIFSTs that connect its boundary terminals. It can be observed that the structures of SCIFSTs are very similar to the structures of EFSTs. The only different structure is Fig. 4.26(f). Therefore, we can make use the algorithm that generates EFSTs to construct the SCIFSTs as shown in Fig. 4.26(a)-(e). For each of the generated trees, we will check if the slew constraint can be met. All SCIFSTs that satisfy the slew constraint will be save as candidates in TI. We can also see that a tree with structure Fig. 4.26(f) can actually be obtained from another tree with structure Fig. 4.26(d) or Fig. 4.26(e), by moving a part of the Steiner chain towards the source. Note that this operation will increase the tree length but may reduce the slew of the tree. Therefore, for each of the generated SCIFSTs with structure Fig. 4.26(d) or Fig. 4.26(e), if the slew constraint cannot be satisfied and the tree structure can be changed to that in Fig. 4.26(f), we will try to move the Steiner chain towards the source to meet the constraint. Note that in this operation, we only need to consider the Hanan grid lines, and thus it can be done efficiently. Finally, all the internal trees that fail to satisfy the slew constraint will be discarded.

It should be noted that during the construction of external and internal trees, the algorithm will try all combinations of terminals to generate all possible candidates. However, we adopt some very efficient pruning techniques to eliminate useless

trees. Therefore, the run time in this stage is not significant and the resulting set of candidate trees are kept in a reasonable size.

Moreover, the proposed algorithm can be easily extended to handle routing obstacles that blocks all routing resources. For each routing obstacle we can simply eliminate all associated SCIFSTs forcing the algorithm to avoid the obstacle.

4.5.2 Concatenation

Let $E = \{e_0, e_1, e_2, \ldots\}$ be the set of directed trees we generated in the first phase. The second phase of the algorithm is to select a subset of E to form an optimal solution to the problem. That is, to find a set of directed trees with minimum total length such that there is a path from the source s_0 to every sink $s \in S$. We use a binary variable x_i to indicate whether a tree $e_i \in E$ is selected as a part of the solution and a binary variable y_i to indicate whether a boundary terminal $b_i \in B$ is selected as a part of the solution. Let $W \subset V \cup B$ be a set of nodes. We define $\delta^-(W)$ to be the set of trees in E that have their source in $\overline{W}$ and at least one sink in W. Similarly, $\delta^+(W)$ is defined as the set of trees in E that have their source in W and at least one sink in $\overline{W}$. Then, the EFST an SCIFST concatenation problem can be formulated as an integer linear program (ILP) as follows.

Minimize:

$$\sum_{i:e_i \in E} len(e_i) \times x_i. \tag{4.11}$$

Subject to:

$$\sum_{i:e_i \in \delta^-(\{s\})} x_i = 1 \quad \forall s \in S, \tag{4.12}$$

$$\sum_{i:e_i \in \delta^-(\{b_j\})} x_i \geq x_k \quad \forall b_j \in B \; \forall e_k \in \delta^+(\{b_j\}), \tag{4.13}$$

$$y_j \geq x_i \quad \forall b_j \in B \quad \forall e_i \in E \text{ s.t. } b_j \in e_i \tag{4.14}$$

$$\sum_{i:e_i \in \delta^-(W)} x_i \geq 1$$

$$\forall W \subset V \cup B \wedge s_0 \in \overline{W} \wedge W \cap V \neq \varnothing \tag{4.15}$$

$$\sum_{i:e_i \cap X \neq \varnothing} x_i(|e_i \cap X| - 1) \leq |X \cap V| + \sum_{i:b_i \in X} y_i - 1$$

$$\forall X \subset V \cup B \wedge X \cap V \neq \varnothing \wedge |X| \geq 2, \tag{4.16}$$

$$\sum_{i:e_i \cap X \neq \varnothing} x_i(|e_i \cap X| - 1) \leq \sum_{i:b_i \in X} y_i - \max_{i:v_i \in X}(y_i)$$

$$\forall X \subseteq B \wedge |X| \geq 2. \tag{4.17}$$

Constraints (4.12) require that the flow in of a sink must be one. Constraints (4.13) ensure that there is no boundary terminal that only has flow out but no flow in. Constraints (4.14) ensure that if a tree is selected, all the boundary terminals it connects are selected as well. Constraints (4.15) are the cutset constraints that guarantee, for any partition W and $\overline{W}$ with the source s_0 in $\overline{W}$ and at least one sink in W, there must be at least one selected tree crossing them with the right direction. Constraints (4.16) and (4.17) are the subtour elimination constraints that eliminate cycles.

This ILP can be solved by a branch-and-bound framework. We use the algorithm proposed in [49], which is the FST concatenation algorithm for the RSMT problem, and extend it to solve the ILP formulated in this chapter.

4.5.3 Incremental construction

Given a source, a set of sinks, and a set of rectangular obstacles, if we include all the obstacles in the algorithm, an optimal solution can be obtained by running the two-phase algorithm once. However, this is usually inefficient for two reasons. Firstly, among all the obstacles, only a fraction of them will overlap with the routing tree. Secondly, among all the obstacles that overlap with the tree, only a fraction of them may cause slew problems. Therefore, we adopt an iterative approach. In the first iteration, we construct a solution without considering any of the obstacles. Then, we check if there is a part of the tree that is over an obstacle and the slew constraint is violated. If the constraint is violated, all the corresponding obstacles will be included in the algorithm and a new iteration will be launched. This process iterates until no slew violation is found.

4.6 Experiments

We implemented ObSteiner with slew constraints based on the Geosteiner-3.1 [1] and all the tests are conducted on a Sun Blade 2500 workstation with two 1.6GHz processors and 2GB memory. Note that although a dual processor machine is used, our algorithm runs sequentially on only one processor. We employ a set of 21 test cases, RC1-RC11, RT1-RT5, IND1-IND5, which are commonly used for the OARSMT problem. The technology parameters are set according to those used in [44]. For the slew constraint α, we set it according to the size of the routing region of each benchmark. We let $\alpha = 0.3ns$ for the larger benchmarks (i.e. IND2, RC01-RC11). and $\alpha = 0.2ns$ for the remaining smaller benchmarks. For comparison, we run the executable of an optimal algorithm for the OARSMT [46] problem on our platform. We choose [46] for comparison because it provides op-

Table 4.1: Results of our approach in comparison with the approach in [46].

Bench mark	m	k	ObSteiner			[46]		$\frac{L_2-L_1}{L_2}$ (%)	$\frac{t_2}{t_1}$ (x)
			$\|E\|$	L_1	t_1(s)	L_2	t_2(s)		
IND1	10	32	61	604	1	604	1	0	1
IND2	10	43	31	9100	1	9500	1	4.21	1
IND3	10	50	37	587	1	600	1	2.17	1
IND4	25	79	315	1078	1	1086	1	0.74	1
IND5	33	71	231	1295	1	1341	1	3.43	1
RC1	10	10	43	25290	1	25980	1	2.66	1
RC2	30	10	357	41060	1	41350	1	0.70	1
RC3	50	10	492	52540	1	54160	1	2.99	1
RC4	70	10	800	56570	2	59070	1	4.23	0.5
RC5	100	10	991	72090	1	74070	1	2.67	1
RC6	100	500	1686	76680	3	79714	369	3.81	123
RC7	200	500	5573	105290	109	108740	629	3.17	5.8
RC8	200	800	4716	107846	66	112564	25027	4.19	379.2
RC9	200	1000	3632	105911	87	111005	18849	4.59	216.7
RC10	500	100	7892	161920	107	164150	149	1.36	1.4
RC11	1000	100	15309	229971	2011	230837	778	0.38	0.4
RT1	10	500	33	1817	1	2146	22	15.33	22
RT2	50	500	649	44217	2	45852	35	3.57	17.5
RT3	100	500	1230	7579	1	7964	774	4.83	774
RT4	100	1000	1582	7634	3	9693	42418	21.24	14139.3
RT5	200	2000	3686	42706	105	51313	289363	16.77	2755.8
Avg								4.91	878.3

timal OARSMTs that give the lower bounds of the wire lengths we can achieve by avoiding all obstacles. In this way, we can clearly see the benefits of allowing some wires to be routed over obstacles. Moreover, since both algorithms aim at achieving the optimal solutions, it is reasonable to compare the run time of them.

The results of the experiments are illustrated in Table 4.1. Column "m" provides the number of sinks and the source in the benchmark. Column "k" provides the number of obstacles in the benchmark. Column "$|E|$" provides the number of candidate trees generated in the first phase. Columns "L_1" and "L_2" provide the wire lengths of the solution. Columns "t_1" and "t_2" provide the run times of the

two algorithms in seconds, respectively.

We can observe from the table that by using our algorithm, the resulting OARSMTs with slew constraints over obstacles can save nearly 5% routing resources on average in comparison with the optimal OARSMT generated by [46]. In particular, our algorithm is more efficient for the benchmarks that contain a smaller number of terminals but a larger number of obstacles. For those benchmarks, our solutions can save more than 10% of the routing resources. Since the majority of the nets in a design will not have a large number of terminals, the solutions provided by us will thus be very applicable in practice. We also observe that our algorithm runs much faster in most of the cases. On average, our algorithm can achieve over 800 times speedup. When there are only a few obstacles in the routing region, the running time of the two algorithms are similar. However, as the number of obstacles increases, our algorithm will be more and more efficient than [46]. The main reason is that when there are a large number of obstacles, an OARSMT algorithm will try to avoid every obstacle even if it does not cause problems, while our algorithm will only focus on the problematic ones which may be a small fraction. It should also be mentioned that the second phase (concatenation phase) of our algorithm dominates the total run time. On average, over 90% of the run time is spent in the second phase.

CHAPTER 5

Conclusion

In this thesis, we study the RSMT problem in the presence of obstacles. The RSMT problem has been of both theoretical and practical interests for nearly half a century. Substantial efforts have been made to develop efficient algorithms, prove performance bound of approximations, and solve the problem exactly. Being a premier application of the RSMT problem, the increasing demand on the design automation of VLSI has greatly promoted the research development of the problem.

In modern VLSI designs, there can be obstacles such as macro cells, IP blocks, and pre-routed nets. How to adapt to these obstacles is becoming a new challenge of the RSMT problem. Previous research works on this problem have been focused heuristic methods. The state-of-the-art exact algorithm can only handle less than one hundred rectangular obstacles. However, the hard IP count per chip can easily be thousands in the recent future. In order to deal with these new requirements, we present efficient exact algorithms for the RSMT problem in the presence of obstacles. For the obstacles that block all routing layers, an exact OARSMT algorithm is developed. For the obstacles that block a fraction of the routing layers, we propose the OARSMT with slew constraints over obstacles and solve it optimally. A combination of these researches provides a powerful tool for solving the RSMT problem in the presence of obstacles. With our optimal methods, we can easily compare the performance of different approaches and see how far a heuristic solution is away

from the optimum. The works presented in this dissertation give key insights into this difficult problem.

As the process technology advances, the number of nets in a design can easily be tens of millions and is still growing. Highly efficient RSMT algorithms are still in great demand. Besides minimizing the wire length, future research on RSMT should also be adapted to the new requirements of VLSI design, such as timing constraints, signal integrity, and the manufacturability issues.

Bibliography

[1] GeoSteiner – Software for Computing Steiner Trees. http://www.diku.dk/geosteiner/.

[2] A. Kahng and G. Robins. On Performance Bounds for Two Rectilinear Steiner Tree Heuristics in Arbitrary Dimension. *IEEE Trans. on Compouter-Aided Design*, 11:1462–1465, 1992.

[3] A. Kahng and G. Robins. A New Class of Iterative Steiner Tree Heuristics with Good Performance. *IEEE Trans. on Compouter-Aided Design*, 11:893–902, 1994.

[4] C. Chu and Y. C. Wong. FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design. *IEEE Trans. on Comput.-Aided Des. Integr. Circuits Syst.*, 27:70–83, 2008.

[5] C. D. Yang, D. T. Lee, and C. K. Wong. Rectilinear Path Problems among Rectilinear Obstacles Revisited. *IEEE Transaction on Computer-Aided Design*, 24(3):457–472, 1995.

[6] C. H. Liu, S. Y. Kuo, D. T. Lee, C. S. Lin, J. H. Weng, and S. Y. Yuan . Obstacle-Avoiding Rectilinear Steiner Tree Construction: A Steiner-Point-Based Algorithm. *IEEE Trans. on Comput.-Aided Des. Integr. Circuits Syst.*, 31:1050–1060, 2012.

[7] C. H. Liu, S. Y. Yuan, S. Y. Kuo, and Y. H. Chou. An $O(n \log n)$ Path-based Obstacle-avoiding Algorithm for Rectilinear Steiner Tree Construction. In *Proc. Design Automation Conf.*, pages 314–319, 2009.

[8] C. J. Alpert, A. Devgan, and S. T. Quay. Buffer Insertion for Noise and Delay Optimization. *IEEE Trans. on Comput.-Aided Des. Integr. Circuits Syst.*, 18(11):1633–1645, 1999.

[9] C. W. Lin, S. Y. Chen, C. F. Li, Y. W. Chang, and C. L. Yang. Efficient Obstacle-avoiding Rectilinear Steiner Tree Construction. In *Proc. Int. Symp. Phys. Des.*, pages 380–385, 2007.

[10] C. Y. Lee. An Algorithm for Connections and Its Application. *IRE Trans. on Electronic Computer*, page 346¨C365, 1961.

[11] E. J. Cockayne and D. E. Hewgill. Exact Computation of Steiner Minimal Trees in the Plane. *Inf. Process. Lett.*, 22:151–156, 1986.

[12] E. J. Cockayne and D. E. Hewgill. Improved Computation of Plane Steiner Minimal Trees. *Algorithmica*, 7:219–229, 1992.

[13] R. Courant and H. Robbins. *What is Mathematics?* Oxford Univ., New York, 1941.

[14] D. M. Warme, P. Winter, and M. Zachariasen. Exact algorithms for plane steiner tree problems: A computational study. In D. Z. Du, J. M. Smith, and J. H. Rubinstein, editors, *Advances in Steiner Trees*, pages 81–116. Kluwer Academic Publishers, Boston, 2000.

[15] C. Duin. Preprocessing the steiner problem in graph. In J. M. Smith D. Z. Du and J. H. Rubinstein, editors, *Advances in Steiner Trees*, pages 81–116. Kluwer Academic Publishers, Boston, 2000.

[16] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*. Number 53. Elsevier, Amsterdam, Netherlands, 1992.

[17] U. Fößmeier and M. Kaufmann. On Exact Solutions for the Rectilinear Steiner Tree Problem Part I: Theorectical Results. *Algorithmica*, 26:68–99, 2000.

[18] G. Ajwani, C. Chu, and W. K. Mak. FOARS: FLUTE Based Obstacle-Avoiding Rectilinear Steiner Tree Construction. In *Proc. Int. Symp. Phys. Des.*, pages 27–34, 2010.

[19] G. Georgakopoulos and C. H. Papadimitriou. The 1-Steiner Tree Problem. *J. Algorithms*, 8:122–130, 1987.

[20] L. J. Guibas and J. Stolfi. On Computing All North-east Nearest Neighbor in the L_1 Metric. *Inf. Process. Let.*, 17:219–223, 1983.

[21] H. Chen, C. Qiao, F. Zhou, and C. K. Cheng. Refined Single Trunk Tree: A Rectilinear Steiner Tree Generator For Interconnect Prediction. In *Proc. international workshop on System-level interconnect prediction*, pages 85–89, 2002.

[22] H. Hou, J. Hu, and S. S. Sapatnekar. Non-Hanan Routing. *IEEE Trans. on Comput.-Aided Des. Integr. Circuits Syst.*, 18(4):436–444, 1999.

[23] H. Zhou. Efficient Steiner Tree Construction Based on Spanning Graphs. In *Proc. Int. Symp. Phys. Des.*, page 152¨C157, 2003.

[24] H. Zhou, N. Shenoy, and W. Nicholls. Efficient Spanning Tree Construction Without Delaunay Triangulation. *Inf. Process. Lett.*, 81:271¨C276, 2002.

[25] M. Hanan. On Steiner Minimal Trees with Rectilinear Distance. *J. SIAM Appl. Math.*, 14:225–265, 1966.

[26] F. K. Hwang. On Steiner Minimal Trees with Rectilinear Distance. *SIAM J. Appl. Math.*, 30:104–114, 1976.

[27] F. K. Hwang. An $O(nlogn)$ Algorithm for Rectilinear Minimal Spanning Trees. *J. Assoc. Comput. Mach.*, 26:177–182, 1979.

[28] J. A. Roy and I. L. Markov. Seeing the Forest and the Trees: Steiner Wirelength Optimization in Placement. *IEEE Trans. on Comput.-Aided Des. Integr. Circuits Syst.*, 26(4):632–644, 2007.

[29] J. Griffith, G. Robins, J. S. Salowe, and T. Zhang. Closing the Gap: Near-optimal Steiner Trees in Polynomial Time. *IEEE Trans. on Compouter-Aided Design*, 13:1351–1365, 1994.

[30] J. L. Ganley and J. P. Cohoon. Optimal Rectilinear Steiner Minimal Trees in $O(n^2 2.62^n)Time$. In *Proc. Canad. Conf. on Computational Geometry*, page 308¨C313, 1994.

[31] 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*, pages 113–116, 1994.

[32] G. Vijayan J. M. Ho and C. K. Wong. New Algorithms for the Rectilinear Steiner Tree Problem. *IEEE Trans. on Compouter-Aided Design*, 9:185–193, 1990.

[33] J. Y. Long, H. Zhou, and S. O. Memik. EBOARST: An Efficient Edge-Based Obstacle-Avoiding Rectilinear Steiner Tree Construction Algorithm. *IEEE Trans. on Comput.-Aided Des. Integr. Circuits Syst.*, 27:2169–2182, 2008.

[34] K. D. Boese, A. B. Kahng, B. A. McCoy, and G. Robins. Rectilinear Steiner Trees with Minimum Elmore Delay. In *Proc. Design Automation Conf.*, pages 381–386, 1994.

[35] L. Li and Evangeline F. Y. Young. Obstacle-avoiding Rectilinear Steiner Tree Construction. In *Proc. Int. Conf. Comput.-Aided Des.*, pages 523–528, 2008.

[36] L. Li, Z. Qian, and Evangeline F. Y. Young. Generation of Optimal Obstacle-avoiding Rectilinear Steiner Minimum Tree. In *Proc. Int. Conf. Comput.-Aided Des.*, pages 21–25, 2009.

[37] M. Borah, R. M. Owens, and M. J. Irwin. An Edge-Based Heuristic for Steiner Routing. *IEEE Trans. on Compouter-Aided Design*, 13:1563–1568, 1994.

[38] M. D. Moffitt, J. A. Roy and I. L. Markov. The Coming of Age of (Academic) Global Routing. In *Proc. Int. Symp. Phys. Des.*, pages 148–155, 2008.

[39] M. Garey and D. Johnson. The Rectilinear Steiner Tree Problem is NP-Complete. *SIAM Journal of Applied Mathematics*, 32:826–834, 1977.

[40] M. Pan and C. Chu. FastRoute: A Step to Integrate Global Routing into Placement. In *Proc. Int. Conf. Comput.-Aided Des.*, pages 464–471, 2006.

[41] T. Polzin and S. V. Daneshmand. On Steiner Trees and Minimum Spanning Trees in Hypergraphs. *Operations Research Letters*, 31:12¨C20, 2003.

[42] R. Hentschke, J. Narasimham, M. Johann, and R. Reis. Maze Routing Steiner Trees with Effective Critical Sink Optimization. In *Proc. Int. Symp. Phys. Des.*, pages 135–142, 2007.

[43] D. S. Richards. On the effectiveness of greed heuristics for the rectilinear steiner tree problem. Technical report, Univ. of Virginia, 1991.

[44] S. Hu, C. J. Alpert, J. Hu, S. Karandikar, Z. Li, W. Shi, and C. N. Sze. Fast Algorithm for Slew Constrained Minimum Cost Buffering. In *Proc. Design Automation Conf.*, pages 308–313, 2006.

[45] J. S. Salowe and D. M. Warme. Thirty-five-point Rectilinear Steiner Minimal Trees in a Day. *Networks*, 25:69–87, 1995.

[46] T. Huang and Evangeline F. Y. Young. Obstacle-avoiding Rectilinear Steiner Minimum Tree Construction: An Optimal Approach. In *Proc. Int. Conf. Comput.-Aided Des.*, pages 610–613, 2010.

[47] T. Huang and Evangeline F. Y. Young. An Exact Algorithm for the Construction of Rectilinear Steiner Minimum Trees Among Complex Obstacles. In *Proc. Design Automation Conf.*, pages 164–169, 2011.

[48] T. Huang, L. Li, and Evangeline F. Y. Young. On the Construction of Optimal Obstacle-avoiding Rectilinear Steiner Minimum Trees. *IEEE Trans. on Comput.-Aided Des. Integr. Circuits Syst.*, 30:718–731, 2011.

[49] D. M. Warme. A new exact algorithm for rectilinear steiner minimal trees. Technical report, System Simulation Solutions, Inc., Alexandria, VA, 1997.

[50] P. Winter. An Algorithm for the Steiner Problem in the Euclidean Plane. *Networks*, 15:323–345, 1985.

[51] P. Winter and M. Zachariasen. Euclidean Steiner Minimum Trees: An Improved Exact Algorithm. *Networks*, 30:149–166, 1997.

[52] R. T. Wong. A Dual Ascent Approach for Steiner Tree Problems on a Directed Graph. *Mathematical Programming*, 28:271–287, 1984.

[53] 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 Transaction on Computer-Aided Design*, 36(3):321–331, 1987.

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

[55] Y. Shi, P. Mesa, H. Yao, and L. He. Circuit Simulation Based Obstacle-aware Steiner Routing. In *Proc. Asia South Pacific Des. Automat. Conf.*, pages 385–388, 2006.

[56] Y. Yang, Q. Zhu, T. Jing, X. Hong, and Y. Wang. Rectilinear Steiner Minimal Tree among Obstacles. In *Proc. Intl. Conf. on ASIC*, pages 348–351, 2003.

[57] Y. Zhang and C. Chu. RegularRoute: An Efficient Detailed Router with Regular Routing Patterns. In *Proc. Int. Symp. Phys. Des.*, pages 45–52, 2011.

[58] A. C. C. Yao. On Constructing Minimal Spanning Trees in k-dimensional Spaces and Related Problems. *SIAM J. Comput.*, 11:721–736, 1982.

[59] Z. Feng, Y. Hu, T. Jing, X. Hong, X. Hu, and G. Yan. An $O(n \log n)$ Algorithm for Obstacle-avoiding Routing Tree Construction in the λ-geometry Plane. In *Proc. Int. Symp. Phys. Des.*, pages 48–55, 2006.

[60] Z. Shen, C. Chu, and Y. Li. Efficient Rectilinear Steiner Tree Construction with Rectilinear Blockages. In *Proceedings ICCD*, pages 38–44, 2005.

[61] M. Zachariasen. Rectilinear Full Steiner Tree Generation. *Networks*, 33:125–143, 1999.