Towards Quality and Efficient VLSI Routing

LIU, Jinwei

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

The Chinese University of Hong Kong
September 2022

Thesis Assessment Committee

Professor YU Bei (Chair)

Professor YOUNG Fung Yu (Thesis Supervisor)

Professor SHAO Zili (Committee Member)

Professor LI Yih-Lang (External Examiner)

Abstract

Routing is the one of the most time-consuming processes in the design of very large scale integrated circuit (VLSI), and an indispensable step in realizing the physical layout of an integrated circuit. Ever since the invention of integrated circuits, the number of transistors keep increasing and the feature size keeps shrinking. In advanced technology nodes, Due to the scaling effects, interconnect delay has almost dominated the total delay, which makes routing even more important and challenging. Design quality and routing efficiency are two main objectives in designing routing algorithms. In this thesis, we focus on routing algorithms that can produce high-quality design with high efficiency.

Routing tree construction studies the optimal topology for a single net and is the backbone of many sophisticated routing algorithms. Wirelength is one of the most important optimizing objectives for routing tree construction. A routing tree of shorter wirelength not only occupies fewer routing resources, but often also implies lower power consumption and delay. In this thesis, we propose a neural network framework to learn to construct rectilinear Steiner minimum trees (RSMT), i.e., the routing tree having the shortest wirelength, via reinforcement learning. Our proposed framework is especially effective and efficient for small to median nets which are most commonly seen in real VLSI designs.

Single net routing is already NP-hard, and routing a whole netlist is even much harder. In netlist routing, not only the topology of each net must be optimized, but the topologies of different nets also need to be well coordinated so as to route all the nets correctly at the same time. Due to the complexity of netlist routing, the problem is usually divided into two phases, global routing and detailed routing, and solved progressively. An effective and

efficient global router is very much needed, in the sense that it plays an important role in optimizing design objectives and evaluating wirelength and routability at early stages of VLSI design. In this thesis, we propose a detailed-routability-driven global router that is able to generate high-quality design after detailed routing. We discuss several techniques that can effectively improve detailed-routability so as to improve design quality.

Moreover, many routers rely heavily on time-consuming path searching algorithms like maze routing to resolve overflows. As a result, it takes longer time to finish routing and premature detours can lead to serious routability issues. Thus, we propose a DAG-based routing algorithm that explores only a few promising paths and create detours only when necessary. We demonstrate the effectiveness and efficiency of our algorithms in our new global router. Enhanced with other techniques, our global router can perform routing with superior efficiency and produce higher quality of result compared to other routers.

摘要

佈線是超大規模集成電路（VLSI）設計過程中最耗時的環節之一，也是實現集成電路的物理佈局無可或缺的步驟。 自集成電路被發明以來，電路上的晶體管的數量不斷增多，特徵尺寸不斷縮小。 在先進製程中，尺寸縮小導致線路延遲幾乎主導總延遲，這讓佈線變得更加重要及具挑戰性。 在本論文中，我們將著重討論可以高質量以及高效實現佈線的算法。

佈線樹構造研究單個網的最優拓撲結構，是很多複雜佈線算法的骨幹。 線長是佈線樹構造中最重要的優化目標之一。 線長短的佈線樹不僅佔用更少的佈線資源，往往也意味著低功耗和延遲。 在本論文裡，我們提出了一個用强化學習來學習構建直角斯坦納最小樹（RSMT），也就是線長最短的佈線樹，的神經網絡框架。 我們提出的框架在處理VLSI設計中最常見的小型到中型網時尤為有用且高效。

單網的佈線已經是NP難問題， 而為一整個網表佈線更加困難。 在網表佈線中，不止單個網的拓撲結構需要優化， 佈線器還需要有效地協調不同網的拓撲結構使得所有網能同時正確佈線。 因為佈線的複雜性，該問題往往會被分為全局佈線和詳細佈線兩個階段逐步解決。 有效且高效的全局佈線器是非常有需要的， 考慮到它不僅在優化設計目標中起到重要作用， 也能用來在VLSI設計的早期階段預估線長和可佈線性。 在本篇論文中，我們提出了一個高可詳細佈線性的全局佈線器， 它的結果在詳細佈線後可以生成高質量的設計。 我們對數個可以有效提高詳細佈線性並以此提高設計質量的技術進行了討論。

此外，許多佈線器高度依賴耗時的路徑搜索算法。 結果就是它們需要花更多時間完成佈線，而且過早的繞行會導致嚴重的可佈線性問題。 因此我們提出一個基於有向無環圖（DAG）的佈線算法，它只探索少數有潛力的路徑，並且只在必要的時候創造繞行。 我們將它用在一個全新的全局佈線器中來體現它的有效性和高效性。 配合其它技術，我們的佈線器相比其它全局佈線器可以更加高效地完成佈線並且獲得更好的結果。

Contents

List of Tables

List of Figures

Acknowledgments

First of all, I would like to express my gratitude to Prof. Evangeline F.Y. Young. It is very lucky for me to have her as my supervisor. I really appreciate every meaningful discussion we have had about research, life, religion and so on. This thesis wouldn't be possible without her patient guidance, generous sharing of knowledge and continuous support.

I would also like to thank my other thesis committee members, Prof. Bei Yu, Prof. Zili Shao and Prof. Yih-Lang Li, for their valuable comments and constructive suggestions.

Special thanks to my fellow students: Dr. Peishan Tu, Dr. Gengjie Chen, Dr. Jordan Chak-Wa Pui, Dr. Christina Huan Shi, Dr. Haocheng Li, Dr. Yuzhe Ma, Dr. Haoyu Yang, Dr. Jingsong Chen, Dr. Bentian Jiang, Xiaopeng Zhang, Dan Zheng, Fangzhou Wang, Lixin Liu, Xinshi Zang, Shiju Lin, Tianji Liu, Bangqi Fu and Qijing Wang. I really enjoy working with them at the Chinese University of Hong Kong.

Last but not least, I am deeply grateful to my parents Fukui Liu and Tengxue Lai. Their love has always been unconditional and they always try to provide me with the best thing they can provide. Without them, I should have been soldering circuit boards in an electronics factory in Shenzhen for the moment.

Chapter 1

Introduction

1.1 Electronic Design Automation

Electronic devices are now ubiquitous in modern society, from personal computers to satellites in outer space. Thanks to the invention of integrated circuits (IC) in 1958 by Jack Kilby [1], we can now make chips out of a single chunk of semiconductor material. Until the late 1960s, electronic systems and circuits are still mainly designed by hand [78], e.g., doing paper designs and cutting rubylith plastic for mask creation. People have witnessed the rapid growth of IC industry and related technologies. IC design soon became an impossible task to be done by hand. Every year, the number of transistors in a circuit keeps increasing and the feature size keeps shrinking [84]. It takes no more than 15 years for integrated circuits to evolve from the so called small-scale integration (SSI) having no more than 10 transistors to the very large scale integration (VLSI) that integrates over 1,000,000 transistors. The design processes of VLSI circuits are gradually automated by various specialized electronic design automation (EDA) softwares and algorithms. In the nano-era, designing effective EDA algorithms remains a challenging problem, as people are constantly trying to accommodate more transistors in smaller area while maintaining consistently functional, reliable, and yielding design with even lower cost and shorter time-to-market [60, 104].

The main EDA flow can be summarized by Figure 1.1 [52, 102]. The journey starts from

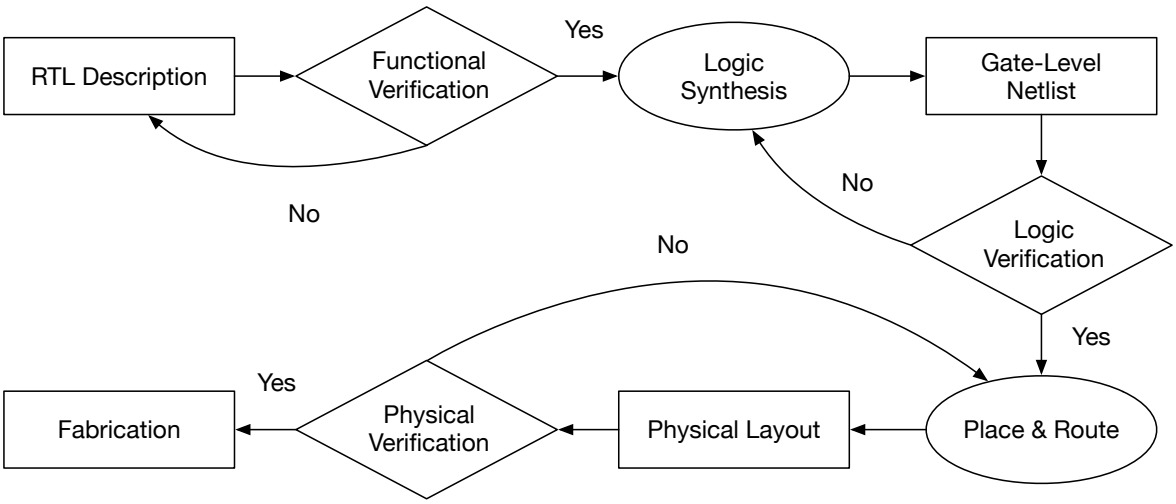
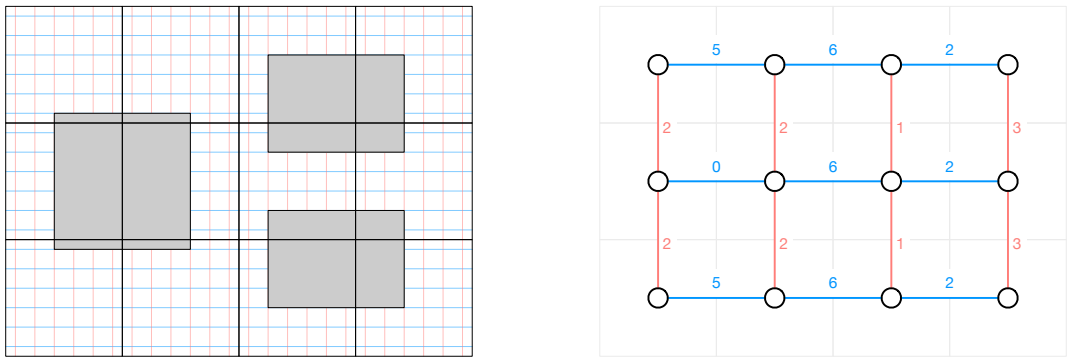


Figure 1.1: *Main Electronic Design Automation (EDA) Flow.*

a register-transfer level (RTL) design generated according to the system specification by using a hardware description language (HDL) like Verilog or VHDL. Logic synthesis will convert the RTL description to a gate-level netlist. Placement and routing, on the other hand, will assign a location for all design components, e.g., macros, cells, gates, and transistors, and realize the connections between them on metal layers. The intermediate results after each step must be fully verified before entering the next step so as to ensure correctness.

The process of placement and routing are also referred to as physical design, as they jointly determine the physical layout of a circuit. Physical design is a rather complex and important process and has direct impact on circuit performance, power, area, reliability and manufacturability [52]. Moreover, like many other processes in VLSI design, physical design is iterative in nature [93] and many steps like global routing and channel routing are repeated several times to obtain a better layout. Besides, placement can sometimes produce unroutable layouts and makes it necessary for the design to be re-placed and re-routed. This iterative nature makes physical design extremely time-consuming. In this thesis, we will especially focus on the algorithms and solutions for routing problems in physical design.



(a) (b)

Figure 1.2: *Global Routing Grid Graph. (a) Routing Space Partitioned by Evenly Spaced Grid Lines. Grey rectangles indicate obstacles; blue and red lines indicate horizontal and vertical tracks respectively. (b) Global Routing Grid Graph Obtained from the Partitioning. The value on each edge indicates its capacity.*

1.2 VLSI Routing

In fact, VLSI routing is a very complicated problem itself and has multiple optimizing objectives. On one hand, a router must complete all connections while minimizing the wire-length, meeting the timing requirements and avoiding design rule check (DRC) violations. On the other hand, a modern IC can contain over billions of transistors and billions nets to be routed. The routing topologies of different nets must be coordinated carefully so as to route all nets correctly at the same time. Due to this complexity, the problem of VLSI routing has long been divided into two phases, global routing and detailed routing, and solved progressively.

In global routing, a circuit is partitioned into several rectangular regions that are called global cells and a global router will assign a collection of interconnected global cells to each net so as to construct a rough routing solution. The most common way of partitioning is to use evenly spaced grid lines to partition the routing space into cells of the same size as shown in Figure 1.2a. A grid graph, as shown in Figure 1.2b, can be constructed by converting each global cell to a vertex and adding an edge for every two neighboring cells. Global routing is usually performed on a grid graph instead of the original routing space.

Due to abstraction of routing models and hiding of details in the stage of global routing,

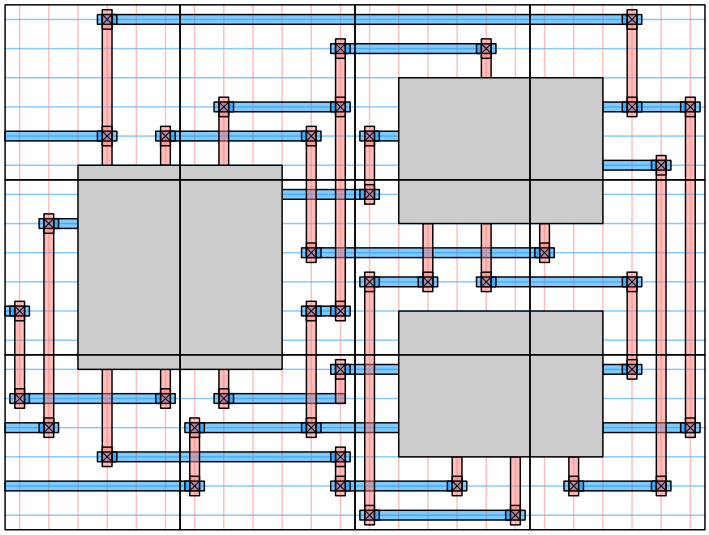


Figure 1.3: *Detailed Routing.*

it is often still possible to find close to optimal solutions for a single net w.r.t. some simple objectives by constructing a routing tree. For example, the optimal wirelength can be found by rectilinear Steiner minimum tree (RSMT) and the optimal path length to all sinks can be achieved by rectilinear Steiner minimum arborescence (RSMA) tree. Both problems are NP-complete [32, 95] and it takes patience, i.e., exponential time, to find the optimal solution. Heuristics are often used instead of exact algorithms for pragmatic reasons.

The routing of a whole netlist is an even harder problem. Not only the topology of each individual net needs to be optimized, the traffic from different nets also needs to be carefully coordinated so as to avoid congestion. Even if all the nets are simple two-pin nets, routing a whole netlist is still NP-complete [65]. Despite its difficulty, global routing is a crucial step in physical design and it plays an important role in both optimizing design objectives and evaluating wirelength and routability at an early stage of placement [42, 91, 62].

In detailed routing (Figure 1.3), the connection of each net is realized by using actual metal wires and vias in the routing resources assigned to it, while avoiding layouts that could lead to design rule check (DRC) violations, e.g., insufficient area or insufficient spacing. Since detailed routing is often the most time-consuming step in physical design, starting from a well-optimized global routing solution can effectively improve detailed-routability

and therefore reduce the time needed to finish detailed routing.

1.3 Machine Learning for EDA

Machine learning studies algorithms that learn from experience [82], i.e., improve their performance based on the data they see. In recent years, machine learning, especially deep learning, has gain unprecedented popularity due to its success in computer vision, natural language processing, robotics, etc. Traditionally, developing solutions for many problems in EDA takes a lot of time and extensive domain knowledge. Machine learning-based approaches can often reduce the amount of human efforts needed so as to shorten the development cycle. Therefore, they have also attracted attention from EDA researchers and practitioners.

Some deep learning models and algorithms also find applications in EDA problems. Convolutional neural networks (CNN) make use of shared-weight convolutional filters, that move all over the input, to extract shift-invariant features in a 2D image. By treating the circuit layout as a 2D image, CNN becomes a natural fit for tasks like prediction of routing congestion [19] or DRC hotspots [110]. Graph neural networks (GNN) generates graph representations that capture the dependence of a graph via message passing between the nodes of a graph [120]. They can be used to learn representations for structures like logic networks or netlist to facilitate downstream tasks like logic optimization [36] or pre-placement wirelength prediction [111]. Reinforcement learning aims at training agents that are able to learn through trying what to do, i.e., how to map situations to actions, so as to maximize a numerical reward signal [97]. Such algorithms can be used to discover strategies for many optimization tasks in EDA, such as parameter tuning [103], placement and routing [81, 71], etc.

The survey [46] classifies the application of machine learning in EDA into four categories: decision making in traditional methods, performance prediction, black-box optimization and automated design, ordered by decreasing manual efforts and expert experiences in the design process. Designing machine learning frameworks with high level of automation is especially

challenging. Moreover, issues like overfitting, non-scalability, and over-simplification must be addressed properly before machine learning-based approaches can be practically useful in realistic scenarios.

1.4 Thesis Overview

When designing routing algorithms, quality of result and routing efficiency are often two conflicting objectives. Iterations of rip-up and reroute are often needed to improve the quality of result. It is a rather time-consuming process and reckless simplification of the process will lead to degradation of results. In this thesis, we propose routing algorithms that achieve a great balance between the two objectives. The global router proposed in the end, wielding Occam's razor, has a minimalist code base and can realize the state-of-the-art quality of result while still having superior efficiency.

In Chapter 2, we will review the literature on various routing algorithms, including routing tree construction, global routing and detailed routing.

In Chapter 3, we explore using reinforcement learning for RSMT construction. RSMT construction algorithms are widely used in global routers to generate initial routing topologies. Traditionally, similar problems are solved repeatedly by hand-engineered heuristics which take a lot of time and extensive domain knowledge to develop. Our proposed rectilinear edge sequence (RES) makes it possible to use sequence-generating deep learning models for RSMT construction. Based on RES, our proposed network can construct RSMT for small to median nets with shorter wirelength and faster, compared to the state-of-the-art heuristics.

In Chapter 4, we introduce our global router CUGR that incorporates multiple techniques to improve detailed-routability. Detailed-routability is an effective metric to evaluate the performance of a global router. Higher detailed-routability often implies shorter detailed routing time and better design quality. For the purpose of improving detailed-routability, we propose several effective routing techniques, such as three-dimensional pattern routing and two-level maze routing, and guide patching.

In Chapter 5, we reformulate the routing problem as a dynamic programming (DP) problem based on directed acyclic graph (DAG). By constructing a routing DAG, our proposed DP algorithm can simultaneously find the optimal two-dimensional topology and layer assignment of each edge segment within the DAG. Moreover, the DAG can be easily augmented to support edge shifting or Steiner point movement. Along with a sparse graph maze routing algorithm, we propose our second generation global router CUGR 2.0, that produces better quality of result with superior efficiency.

Chapter 2

Literature Review

In this chapter, we will review the literature on several routing related topics, including routing tree construction, global routing and detailed routing.

2.1 Routing Tree Construction

Routing tree construction is a fundamental problem in VLSI routing and is the building block of many routing algorithms. For example, it is often used in global routing to generate initial routing topologies for multi-pin nets. The most important optimization objective of routing tree construction is wirelength. A routing topology having a shorter wirelength not only occupies fewer routing resources, but often also implies lower power consumption and delay [24]. The optimization of wirelength is therefore one of the main focuses of this thesis. Apart from wirelength, the construction of shallow-light tree [5, 4, 17, 18] are studied for the purpose of wirelength and timing co-optimization. For clock routing, the construction of zero skew tree [14, 98, 99] or bounded skew tree [25, 113] is studied extensively.

2.1.1 Rectilinear Steiner Minimum Tree

In the scenario of VLSI design, the routing tree with the shortest wirelength is called rectilinear Steiner minimum tree (RSMT). In an RSMT, additional points known as Steiner points

are often introduced so as to facilitate shorter connections. The problem of constructing the optimal RSMT is NP-complete [32]. In practice, it is common to use rectilinear minimum spanning tree (R-MST) to approximate RSMT, since R-MST can be efficiently constructed in $O(n \log n)$ time [48]. It is also proven that the length of an R-MST is at most $1.5\times$ of the optimal RSMT length [49].

a) R-MST-Based Heuristics: In fact, R-MST is the base of many RSMT heuristics. Ho *et al.* [44] proposed a dynamic programming algorithm to find the best topologies for the edges in an R-MST in linear time, such that the overlaps of edges are maximized and the wirelength gets minimized. Later, Kahng and Robins proposed the iterated 1-Steiner method [54], the main idea of which is to iteratively find the best Steiner point to add to the R-MST until convergence. It is worth mentioning that the number of such Steiner point candidates are limited, due to Hanan's discovery [39], i.e., an optimal RSMT solution is always possible by only considering the Steiner points on the Hanan grid. Griffith *et al.* [35] came up with a better implementation of the iterated 1-Steiner approach and improved its complexity from $O(n^4 \log n)$ to $O(n^3)$ by a dynamic R-MST update scheme. Borah *et al.* [10] proposed an edge-based heuristic that incrementally connects vertices to nearby edges and removes the longest edges in the loops formed. The edge-based algorithm improved the complexity to $O(n^2)$. The batched greedy algorithm (BGA) [53] proposed by Kahng *et al.* also starts from an R-MST. The wirelength of the tree is minimized by iteratively constructing optimal Steiner structures for three-terminal triples. The greedy triple selection is relaxed by the batched method, i.e., the gain of triples are not re-computed for multiple selection. The resulting heuristic has a complexity of $O(n \log^2 n)$. Zhou [118] improved the edge-based heuristic by utilizing the spanning graph [119] to find better edges to connect to, in a more efficient way. The complexity is also improved to $O(n \log n)$.

b) Look-Up Table-Based Heuristics: FLUTE [108], on the other hand, adopts a look-up table-based approach, and is the most efficient heuristic so far. Chu and Wong, the authors of FLUTE, suggested that the relative positions of the pins of a net can be characterized

by a position sequence. For each position sequence, there are only a limited number of potentially optimal Steiner trees (POST). The number of POSTs for small nets is especially small. Therefore, FLUTE pre-computes a minimal set of POSTs for all possible position sequences for small nets with $degree \leq 9$ and stores them in a look-up table. In actual use, low-degree nets are solved directly by computing the position sequence and retrieving the optimal Steiner tree from the look-up table. High-degree nets are recursively broken into small nets that can be handled by the look-up table. The solutions of the small nets are then merged back to form that of the original nets. Regarding time complexity, though looking-up in the table takes merely constant time, both net breaking and merging takes $O(n \log n)$ time to complete.

c) Others: For exact algorithm, GeoSteiner [105] is an efficient optimal algorithm that enumerates all possible full Steiner tree to form an RSMT. It is proven that an optimal RSMT can always be found by combining full Steiner trees only, which are Steiner trees with a special structure. The running time of GeoSteiner inevitably goes to exponential.

Furthermore, some works also study RSMT construction in the presence of obstacles. The works of [2, 70, 72] study the construction of obstacle-avoiding RSMT which optimize wirelength in the presence of impassable obstacles. The works of [43, 117, 47] construct RSMT with restricted length over obstacles in order to facilitate buffer insertion.

2.1.2 Learning Heuristics for Combinatorial Optimization

The traditional approaches for solving combinatorial optimization problems like RSMT construction require hand-engineered heuristics by human experts. Such heuristics take very long time to develop even though similar problems have been solved repeatedly. On the other hand, machine learning-based approaches can effectively reduce the reliance on expert experience and therefore accelerate the solving of these problems. Therefore, we also review some recent development of machine learning approaches for solving combinatorial optimization problems, as they inspire our deep reinforcement learning approach for RSMT

construction which will be discussed in this thesis. More specifically, we will follow the development of contemporary machine learning approaches for the the travelling salesman problem (TSP), as it has a similar objective to wirelength minimization. The travelling salesman problem studies the shortest way to visit each vertex in a graph exactly once and return to the start.

a) Pointer Networks: Vinyals *et al.* [101] proposed the pointer network that is able to learn the conditional probability of an output sequence, the elements of which are selected from the input sequence. The pointer network is mainly comprised of two components, an encoder and a decoder, both of which are implemented using a recurrent neural network (RNN). For TSP solving, the encoder processes the input vertices sequentially to generate a code vector. The decoder then decodes the code vector to generate a visiting order for a TSP instance stochastically, one vertex at a time.

b) Learning Heuristics for TSP: Unlike the supervised approach taken by the pointer network, Bello *et al.* [8] trained the pointer network using model-free policy-based reinforcement learning. An auxiliary network called critic was introduced to learn the expected TSP tour length given an input sequence so as to help with convergence. Dai *et al.* [61], used the graph embedding network called structure2vec [26] to featurize the vertices in a graph. The graph embedding network encodes the input vertices as an order-invariant set rather than an ordered sequence, thus better suits the problem. They also used a value-based reinforcement learning approach instead of a policy-based one. The main advantage of value-based approaches is that reward is generated for every vertex being selected, whereas for policy-based approach, reward is generated only once when the whole tour is completed. Deudon *et al.* [28] used policy-based reinforcement learning and the multi-head attention encoder [100] for vertex encoding, which has a similar advantage as structure2vec. Kool *et al.* [64] adopted the same encoder but replaced the critic network with an efficient deterministic greedy rollout.

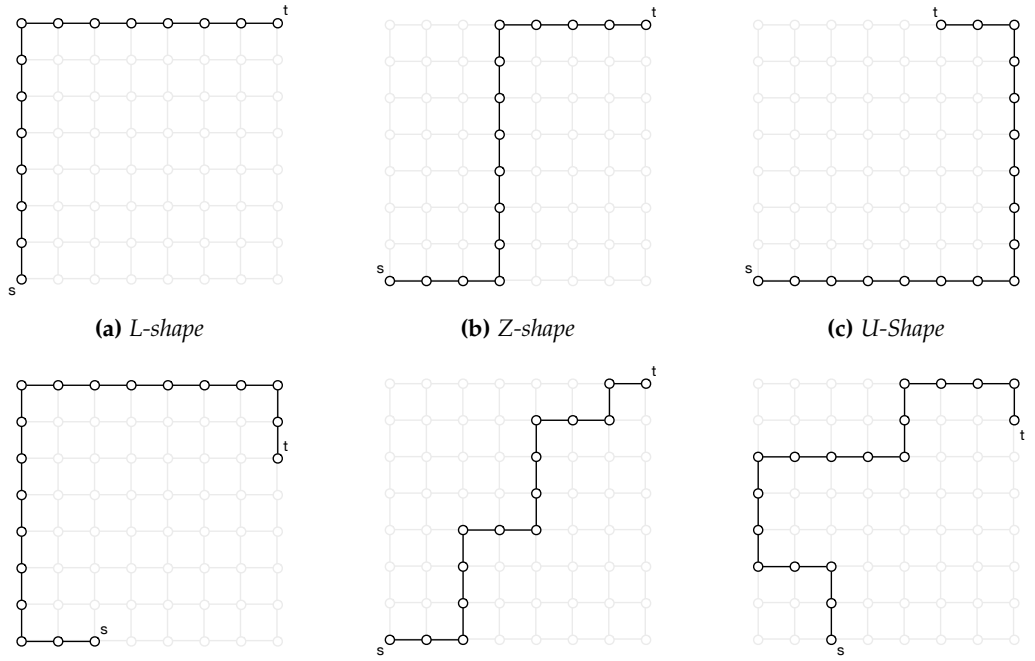
Though we have witnessed great progresses in machine learning approaches for com-

binatorial optimization problems like TSP, it is worth noting that RSMT construction is actually a much harder problem. For example, the solution to a TSP problem can be easily represented by a permutation of the input vertices, however, it is not possible to do the same for an RSMT solution. The size of the solution space of RSMT construction is in fact of a much higher magnitude.

c) Learning Heuristics for Combinatorial Optimization Problems in EDA: In addition to RSMT construction, there are many other combinatorial optimization problems in EDA. Haaswijk *et al.* [36] applied deep reinforcement learning for logic optimization by casting the process as a deterministic Markov decision process (MDP) where each move transforms the logic network to a new state. Mirhoseini *et al.* [81] trained a reinforcement learning agent for macro placement and achieved superhuman results w.r.t. power, performance, and area (PPA). Liao *et al.* [71] proposed a reinforcement learning algorithm for global routing and outperformed A* search on small made-up designs. Generally speaking, machine learning approaches directly tackling combinatorial optimization problems that arise in EDA are still relatively rare to see.

2.2 Global Routing

The academic research of global routing is often stimulated by some contests, at which a number of industrial benchmarks are released, e.g., ISPD 2007-2008 benchmarks [85] and ICCAD 2019 benchmarks [29]. There has been over 10 years between the last two major global routing contests. In ISPD 2007-2008 contests, global routing was treated as a relatively independent routing problem, and the main objectives were to minimize the total overflow and routed wirelength. In ICCAD 2019 contest, global routing was put back to the global-detailed routing paradigm. It required participating global routers to directly take LEF/DEF format [68] as input and extract a grid graph model themselves from the information of layers, standard cells, tracks, etc. Moreover, the global routed solutions were evaluated by running an actual detailed router rather than counting overflows in the grid



(a) *L-shape* **(b)** *Z-shape* **(c)** *U-Shape*

(d) *Three-Bend Routing* **(e)** *Monotonic Routing* **(f)** *Maze Routing*

Figure 2.1: *Different Routing Styles for Two-Pin Nets.*

graphs. The realistic formulation poses new challenges to the research of global routing.

In this section, we will review some routing techniques adopted by different global routers. It is worth mentioning that a successful global router often incorporates more than one routing techniques. We roughly divide the techniques into three categories - initial routing, rip-up and reroute, and concurrent routing. For initial routing, academic global routers often devise different techniques to limit routing topology so as to generate an initial routing solution for all nets efficiently. For rip-up and reroute, more flexible techniques like maze routing are employed to resolve remaining overflows. For concurrent routing, multiple nets are routed concurrently to achieve better quality or routing acceleration.

2.2.1 Initial Routing

a) Routing with Restrained Topology: For initial routing, a multi-pin net is often broken into multiple two-pin nets by constructing RSMT and treating each edge as a two-pin net.

To ensure efficiency, each two-pin net is routed separately by considering limited topologies. Labyrinth [59] was the first to describe the pattern routing technique, the basic idea of which is to connect each two-pin net using only topologies conforming to a specific pattern, such as L-shape (Figure 2.1a) or Z-shape (Figure 2.1b). Apart from these two shapes, Archer [88] also considered U-shape (Figure 2.1c) in early stage routing, which allows bounded increase of wirelength. The three-bend routing proposed by FastRoute 4.0 [112] searches any routing topology having three or fewer bending points. An earlier version of FastRoute, FastRoute 2.0 [89], proposed monotonic routing (Figure 2.1e) that allows any routing direction that monotonically reduce the distance to the sink. DpRouter [12] independently proposed dynamic pattern routing which had the same idea. Monotonic routing is much more flexible compared to the techniques mentioned above but still way more efficient than maze routing (Figure 2.1f).

b) RSMT Adaptation: Some global routers also develop techniques to adapt the RSMT so as to better deal with congestion. FastRoute 1.0 [90], as an example, came up with a simple way to construct congestion-driven Steiner tree. It will stretch the congested rows or columns in a Hanan grid before calling FLUTE [23], a pure wirelength-driven RSMT construction algorithm, such that the edges in the congested rows or columns will become more expensive to use. This technique works well when all edges in a row or column of the Hanan grid are congested. However, it can not differentiate local congestion that does not span a whole row or column, because all edges in a row or column must be stretched together and share the same length. The congestion-driven Steiner tree construction is complemented with an edge shifting technique. It allows some congested edge segments to be translated to elsewhere if such translations result in no increase of wirelength.

Moreover, Archer [88] proposed a Lagrangian relaxation-based edge replacement method to optimize the topology of a Steiner tree under bounded wirelength. MaizeRouter [83] proposed extreme edge shifting that allows a congested edge to be translated to elsewhere even if wirelength increase is needed. Extreme edge shifting can sometimes lead to superfluous wires which are remedied by an edge retraction technique.

2.2.2 Rip-up and Reroute

Rip-up and reroute is the stage when overflown nets are ripped up and rerouted. Though only a small portion of the netlist need to be ripped up, Rip-up and Reroute is often the most time-consuming stage in global routing.

a) Maze Routing: Most rip-up and reroute techniques are based on the maze routing algorithm [66] that explores virtually all kinds of topologies by breadth first search to find the path giving the minimum cost. FastRoute 2.0 [89] proposed multi-source multi-sink maze routing to find a better connection between two partially routed sub-trees after an overflown edge is removed. The main idea is to treat all grid points in one sub-tree as the sources and all grid points in the other sub-tree as the sinks. Then, maze routing are performed to find the best path from the sources to the sinks to form a tree. FGR [92] re-designed maze routing cost using discrete Lagrange multipliers which provided a natural way to handle net weights and timing objectives. NCTU-GR 2.0 [77] proposed a bounded-length maze routing algorithm to speed up maze routing and reduce redundant wires. In addition, an RSMT-aware routing scheme was proposed to build routing topologies with shorter wirelength. It was achieved by applying lower routing cost to edges covered by an RSMT skeleton.

b) Negotiation-Based Rip-Up and Reroute: PathFinder [80] proposed a negotiation-based rip-up and reroute technique that is extensively used in many global routers proposed thereafter. The main idea of negotiation-based rip-up and reroute is to introduce a history term in the cost of using an edge. The corresponding history term will be increased if an edge is overflown so that early routed nets will have extra incentive to find alternative paths.

BoxRouter 2.0 [21] proposed a negotiation-based A* search algorithm with adjusted congestion penalty to achieve stable improvement of routing result. NTHU-Route [31] adopts a history-based cost scheme and reroutes nets in more congested regions with higher priority to resolve overflows more effectively. FastRoute 3.0 [116] introduced virtual capacity

to reduce the capacity of congested edges so as to push away extra demands and provided an alternative option for negotiation-based schemes.

Moreover, in the original formulation of the negotiation-based cost scheme, the history term will increase monotonically as rip-up and reroute progresses, even when some previously congested regions become un-congested. Such a scheme can cause unnecessary detours. Thus, NCTU-GR 1.0 [27] re-designed the cost function such that the history term for the regions, where congestion is resolved, will gradually wear off as the number of iterations increases. Besides, it also adopted a simulated evolution-based rip-up and reroute scheme to rip-up and reroute each net stochastically. Nets routed with poorer wirelength or more overflows will have higher chances to be rerouted.

2.2.3 Concurrent Routing

There are two different kinds of philosophy when it comes to concurrent routing - simultaneous optimization and parallelization.

a) Simultaneous Optimization: For simultaneous optimization, the topology of multiple nets are determined simultaneously in order to improve routing quality. Take BoxRouter [22] as an example. It first decomposes multi-pin nets into two-pin wires using RSMT. Flat wires, i.e. wires that can be connected without a bend, are pre-routed to estimate routing congestion. A box will be initiated to cover the most congested region and expanded in each iteration. During each expansion, all wires between two successive boxes are routed simultaneously as many as possible using integer linear programming (ILP). The topology of each wire is chosen from two L-shape routing candidates. Failed wires are routed by maze routing instead. The process will stop until the box covers the whole circuit. ILP-based method can find the optimal solution w.r.t. the formulation, but the process of solving it is NP-hard. The expanding box scheme ensure the sub-problem in each iteration is tractable, because only the wires between successive boxes will be routed. BoxRouter 2.0 [21, 20] extended the ILP formulation to solve layer assignment after 2D routing. It is also

solved progressively using the expanding box scheme, i.e., newly covered wires during each expansion are simultaneously assigned layers.

Routers having similar philosophy all develop different techniques to ensure scalability. SideWinder [45] used ILP to optimize the whole netlist in the entire routing space by gradually introducing more candidate routing topologies for each net. GRIP [109] made ILP applicable to large-scale designs by decomposing the circuit into rectangular regions. The nets in the same region are routed simultaneously. Albrecht [3] formulated global routing as a multi-commodity flow problem. By allowing fractional flows, the problem can be solved in linear time by linear programming and more efficiently by an approximation algorithm. After solving, a Steiner tree is chosen for each net from a set of candidates stochastically with a probability equal to the corresponding flow. BonnRoute [33] reduced global routing to a min-max resource sharing problem, which is a more general form of the multi-commodity flow problem. The candidate Steiner trees are generated by an oracle function which is basically a Steiner tree algorithm rather than given explicitly. For most routers mentioned above, rip-up and reroute is still needed to resolve overflows caused by relaxation.

b) Parallelization: Some other works develop parallel routing techniques solely for the purpose of acceleration. Many academic papers have discussed ways of accelerating global routing through net-level parallelization on multi-core central precessing units (CPU) [77, 37, 40]. Due to the recent advance of graphical processing units (GPU), it becomes possible to run tens of thousands of threads concurrently on a single GPU and paves the way for higher degree of parallelization. Han *et al.* [38] accelerated two-pin net routing by a GPU implementation of the A* method. FastGR [76] parallelized 3D pattern routing and distributed the workload on both CPU and GPU for balance and efficiency. Gamer [73] parallelized maze routing by updating horizontal and vertical routing costs alternatively.

2.3 Detailed Routing

After global routing, detailed routing will be responsible to realize the actual connections while avoiding various design rule check (DRC) violations.

2.3.1 Pin Access

The first technical issue faced by detailed routing is pin access. Pin access planning can be a hard problem due to several reasons. Firstly, pins can have complex geometric shapes formed by multiple rectangles and routing segments must be placed carefully to avoid DRC violations. Sometimes, off-grid wires and vias are inevitable. Secondly, routing segments for nearby pins are likely to conflict each other. Ozdal [87] proposed a multicommodity flow model to find escape routes for pins in dense clusters. Nieberg [86] discussed a method to create offgrid pin access paths. Their method first generates a set of design rule conforming paths, and then searches for conflict-free path sets by branch-and-bound. The method proposed by Kahng *et al.* [55] assumes only adjacent pins could have conflicts and constructs DRC-clean pin access scheme by dynamic programming.

2.3.2 Track Assignment

Track assignment refers to the step that wire segments in the same panel, i.e., a row or a column of global cells, are assigned to tracks simultaneously. It provides not only a good starting point for detailed routing, but also a fast way to estimate local congestion during global routing. Batterywala *et al.* [7] formulated the track assignment problem by weighted bipartite matching and solved it progressively using the shortest augmenting path algorithm. Wong *et al.* [107] proposed a negotiation-based track assignment method. In their method, overlapped wire segments are ripped-up and re-assigned to eliminate as many overlaps as possible. TraPL [94] connects local nets by single-trunk tree and determines via locations to better estimate partial track utilization. Local nets refer to the nets having all pins inside the same global cell and are ignored by many previous methods. RDTA [74] adopted the rip-up

and re-assign method to reduce overlapping and proposed an efficient way to connect wire segments and pins so as to get a fully connected solution.

2.3.3 Detailed Routers Proposed in the Early 2010s

Detailed routers proposed in the early 2010s have developed different techniques to handle DRC and scalability issues. RegularRoute [115, 114] formulates the problem of segment assignment inside each panel on a layer as a maximum weighted independent set problem so as to minimize design rule violations. After routing, routing shapes are adjusted locally to further reduce DRC violations. BonnRoute [33] models the routing space using an efficient two-level data structure, enabling both fast and accurate design rule violation checking. MANA [13] proposed a design rule-aware maze routing algorithm. The proposed algorithm avoids end-of-line spacing violations by both infeasible grid point marking and post-processing. Besides, potential wire extension needed for satisfying the minimum length is captured in the routing cost. Partial paths unable to be extended to avoid minimum length violation are pruned during routing.

2.3.4 Detailed Routers Emerged Recently

The recent ISPD 2018 and ISPD 2019 detailed routing contest [79, 51] released a detailed routing benchmark suite containing industrial designs in 65, 45 and 32 nm nodes and stimulated many new researches in detailed routing [57, 96, 15, 34]. Dr. CU [16] and TriotonRoute [57] are two most prominent detailed routers emerged from the contest.

One of the most important features of Dr. CU [16] is correct-by-construction, i.e., correct DRC violations during routing rather than wait till post-processing. Dr. CU performs detailed routing mainly by a proposed path searching algorithm that captures the minimum-area constraint. The path searching algorithm accurately predicts any potential cost for wire extension or minimum-area violation. Compared with MANA [13], Dr. CU treats the minimum-area penalty as a Lagrange multiplier rather than a hard constraint and does not try to fix minimum-area violations at all cost. Furthermore, the safe spaces for

wire extension are pre-computed and queried efficiently rather than verified during search. Besides, Dr. CU also proposed a set of two-level grid graph data structures, consisting of global and local grid graph, to achieve both economic memory usage and efficient query. The global grid graph stores routed edges by balanced binary search trees (BST) and intervals without instantiating all vertices. A local grid graph is created for each net and is a sparse subgraph of the whole grid graph having all edge costs ready-to-use. Dr. CU 2.0 [69] extends its design rule-aware maze routing to handle end-of-line spacing with parallel edges by pessimistic estimation. Moreover, Dr. CU 2.0 computes a lookup table for each via type so as to efficiently estimate violations caused by vias.

The first version of TritonRoute [57] proposed a MILP-based panel routing scheme to perform track assignment. In an updated version [58], the MILP-based method was replaced by a simplified greedy track assignment method due to prohibitive running time. After track assignment, multiple iterations of rip-up and reroute are performed locally to fix violations. In each iteration, the design is partitioned into 7×7 global cell blocks such that the overflown partial nets in different blocks can be routed efficiently in parallel. TritonRoute-WXL [56] discussed ways to check various DRC violations and proposed a relatively complete DRC engine. After routing, the DRC engine will be invoked to generate design rule violation markers so as to discourage certain routing topologies in the future iterations. Moreover, a queue-based ripup and reroute flow is adopted to reroute nets in a circular order to improve DRC convergence.

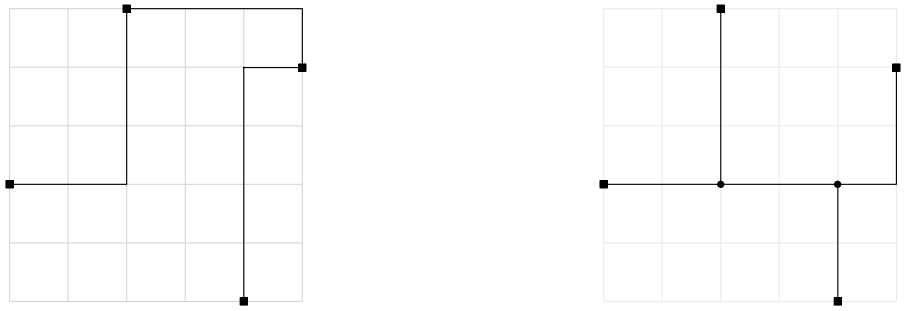
Chapter 3

RSMT Construction by RL

In this chapter, we propose to use the idea of machine learning for combinatorial optimization to construct rectilinear Steiner minimum tree (RSMT). We propose Rectilinear Edge Sequence (RES) to represent an RSMT, thus we call our method REST. We also design an actor-critic neural network model to produce RES, and train it using reinforcement learning. The negative length of the RSMT constructed is used as reward to encourage the model to find shorter solutions. The reward can also be redesigned to cater other needs. The main contributions of this work are as follows:

- To the best of our knowledge, this work is the first successful attempt to solve RSMT construction using a machine learning based method.
- Rectilinear edge sequence (RES) is proposed to encode an RSMT solution to bridge the gap between machine learning output and RSMT structure. We show in Section 3.1.3 that it has some nice properties, making it suitable for reinforcement learning.
- An actor-critic model is designed for RSMT construction. After training, it yields solution with $\leq 0.36\%$ error for nets with ≤ 50 pins in ≤ 1.9 ms on average, which is much faster than traditional heuristics of similar quality.

The rest of the chapter is organized as follows. In Section 3.1, we give a brief introduction of the problem, and discuss the rectilinear edge sequence (RES). In Section 3.2, the architecture



(a) *R-MST* **(b)** *RSMT*

Figure 3.1: *Rectilinear Minimum Spanning Tree vs. RSMT.*

of our actor-critic networks is introduced in detail. We discuss the experiments and results in Section 3.3.

3.1 Preliminary

3.1.1 RSMT the problem

The problem of constructing RSMT or rectilinear Steiner minimum tree can be defined as follows. Given a set of points V, construct a rectilinear tree $T(U)$ connecting all points in U of minimum L_1 length, such that U is a superset of V ($V \subseteq U$). The points newly introduced in U are called *Steiner points*. The *Steiner points* help to reuse some edges and reduce the total tree length.

Take Figure 3.1 as an example. Given the 4 points in square, the shortest way to interconnect them without introducing extra *Steiner points* is by a rectilinear minimum spanning tree (R-MST). The shortest tree length in L_1 norm will be 14 as shown in Figure 3.1a. However, by introducing extra *Steiner points*, some edges can be reused, and a rectilinear Steiner minimum tree (RSMT) with even shorter total length can be constructed. Figure 3.1b illustrates such an RSMT of length 12. The two solid dots are the *Steiner points* introduced in order to construct the RSMT.

The problem of constructing RSMT for a random set of points is known to be NP-

complete, but Hanan [39] has proven that an optimal RSMT can always be constructed on the Hanan grid (Figure 3.2b). The Hanan grid of a set of points is formed by drawing a horizontal line and a vertical line over each point in the set.

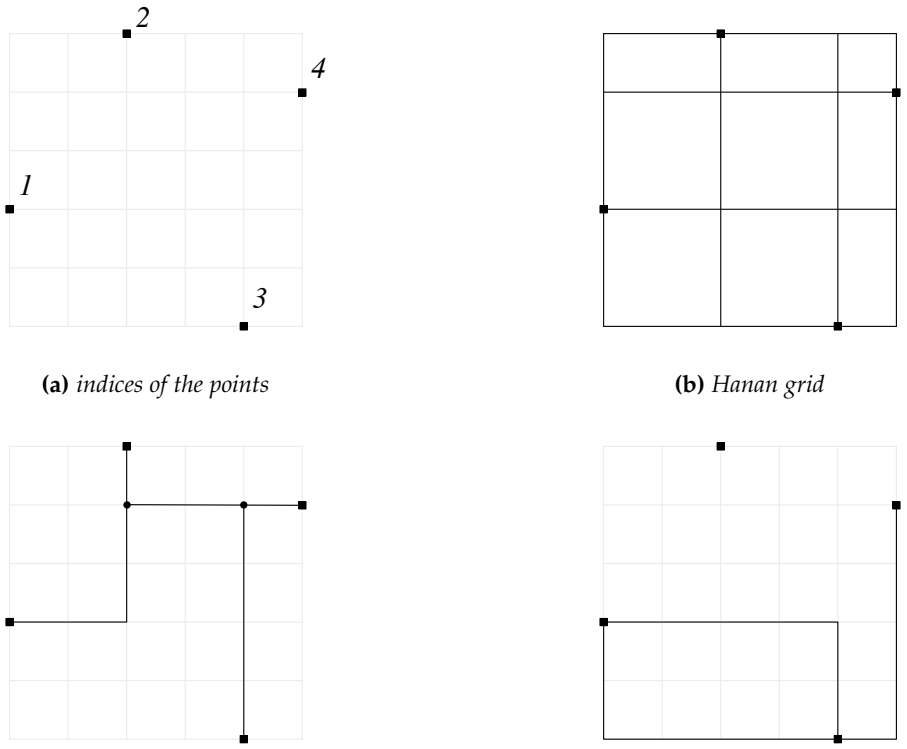
3.1.2 Sequence to RSMT

Neural networks have always being good at generating sequences, no matter it is generating a sentence or a visiting order for the travelling salesman problem (TSP). In this work, we will also use neural networks to produce sequences that can be converted to RSMT. We name this kind of sequences rectilinear edge sequence (RES).

For a given set of points $V = \{(x_1, y_1), ..., (x_n, y_n)\}$, an RES for V is a sequence of $n-1$ index pairs $((v_1, h_1), ..., (v_{n-1}, h_{n-1}))$ where $v_i, h_i \in \{1, ..., n\}$ for $i = 1, ..., n-1$. Each pair in the RES will decide a rectilinear edge that connects two points. The i^{th} pair (v_i, h_i) will connect the v_i^{th} point and the h_i^{th} point by drawing a vertical line segment over the v_i^{th} point, and a horizontal line segment over the h_i^{th} point. For example, suppose a given point set is $V = \{(0,2), (2,5), (4,0), (5,4)\}$, which is the one shown in Figure 3.1. The positions of the points and their indices are shown in Figure 3.2a. One possible RES for V will be $res_1 = ((3,1), (2,1), (4,1))$, which will give us the optimal RSMT in Figure 3.1b. Besides, there can be multiple optimal RES for the same set of points, for example $res_2 = ((2,1), (2,4), (3,4))$ representing the RSMT in Figure 3.2c is also optimal. Note that the overlapping edges indicated by an RES are merged automatically, with *Steiner points* created.

It is worth mentioning that an RES is not always optimal and valid. Consider $res_3 = ((3,1), (1,3), (4,3))$, it suggests the solution in Figure 3.2d. It is obviously not a valid solution to interconnect all points. Therefore, we enforce two requirements in Theorem 1 to guarantee the validity of an RES.

Theorem 1. *Suppose* $res = ((v_1, h_1), ...(v_{n-1}, h_{n-1}))$ *is an RES for a point set* $V = \{(x_1, y_1), ..., (x_n, y_n)\}$*, if the following 2 requirements are satisfied, res is guaranteed to be valid, i.e. the solution defined by res is guaranteed to connect all points in* V *: (1)* $v_1 \neq h_1$*. (2) Exactly one of* v_i*,* h_i *is*



(a) *indices of the points*

(b) *Hanan grid*

(c) *Another Solution*

(d) *An Invalid RES*

Figure 3.2: *Rectilinear Edge Sequence (RES) Explained.*

visited before, for $i = 2, ..., n-1$*, i.e., exactly one of* v_i*,* h_i *appeared in* $\{v_1, h_1, ..., v_{i-1}, h_{i-1}\}$*.*

Proof. Let's consider the sub-tree formed by the visited points. In the first pair, we build a sub-tree with 2 distinct points connected by a rectilinear edge. In each of the following $n-2$ pairs, we will attach a new point to the sub-tree by a rectilinear edge. Thus, eventually all n points will be visited and interconnected by the RES satisfying the two requirements. □

Theorem 2. *For any point set* $V = \{(x_1, y_1), ..., (x_n, y_n)\}$*, we can always find an RES such that the tree it represents is an optimal RSMT for* V*.*

Theorem 2 ensures that an optimal RES always exists. This is actually implied by the Hanan theory [39], which states that an optimal RSMT can always be constructed on the Hanan grid (Figure 3.2b). Given such an RSMT, we can break it into $n-1$ rectilinear edges on the Hanan grid, each connecting two points in V. By picking an appropriate order, the edges can be written as the $n-1$ pairs of an RES.

3.1.3 Good Properties of RES

RES is not the only way to encode an RSMT solution, but there are several good properties that make RES a good choice for RSMT construction via reinforcement learning. The major advantages of RES are as follows.

a) Fixed Length Sequence: Determining the number of pairs to output is non-trivial for a neural network model. Fortunately, this will not be a problem with RES, since the length of the RES for any set of n points is always $n-1$.

b) Linear Time Evaluation: In order to learn from experiences, we need to evaluate the solutions obtained by the model on the fly to generate reward. The reward in our case is the negative of the total length. The evaluation process is often the bottleneck of reinforcement learning, as it usually requires lots of computations or even simulations. An RES can be evaluated in linear time by summing up the length of the horizontal and vertical segments over each point. For each point, we can compute the length of these segments by keeping

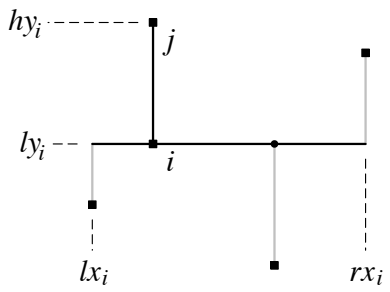


Figure 3.3: *The horizontal and vertical segments over point i. The left end x of its horizontal segment is lx_i, and the right end is rx_i. The lower end y of its vertical segment is ly_i, and the higher end is hy_i.*

track of the their two ends as illustrated in Figure 3.3. Point j in Figure 3.3 is regarded as being connected by a horizontal segment of zero length to avoid adding the length of the same segment twice. The algorithm for RES evaluation is summarized in Algorithm 1.

Algorithm 1 RES Evaluation

Input: $V = \{(x_1, y_1), ..., (x_n, y_n)\}$, $res = ((v_1, h_1), ..., (v_{n-1}, h_{n-1}))$
Output: $length$

$lx_i \leftarrow x_i$, $rx_i \leftarrow x_i$, for $i = 1, ..., n$
$ly_i \leftarrow y_i$, $hy_i \leftarrow y_i$, for $i = 1, ..., n$
for $i = 1$ to $n - 1$ **do**
    $(v, h) \leftarrow (v_i, h_i)$
    $lx_h \leftarrow \min(lx_h, x_v)$, $rx_h \leftarrow \max(rx_h, x_v)$
    $ly_v \leftarrow \min(ly_v, y_h)$, $hy_v \leftarrow \max(hy_v, y_h)$
$length \leftarrow \sum_{i=1}^{n}((hy_i - ly_i) + (rx_i - lx_i))$
return $length$

3.2 Neural Network Model

In this section we will introduce the neural network model we designed for RSMT construction. Figure 3.4a summarizes our model in a simplified fashion. The model we use mainly has two components - actor and critic. Both of them take a set of n point coordinates V ($V \in \mathbb{R}^{n \times 2}$) as input. The actor will take several actions to determine the elements in the RES according to a stochastic policy, i.e., each action is chosen by different probabilities

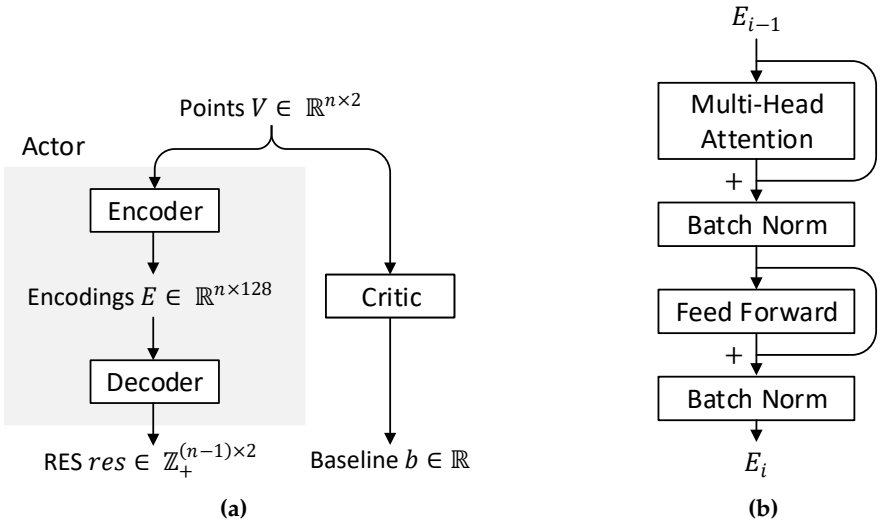


Figure 3.4: *(a) Simplified Structure of Our Neural Network Model; (b) The i^{th} Encoding Process.*

produced by the model. The critic will set a baseline by predicting the expected length of the RSMT found by the current policy, so that it encourages the actor to constantly improve its performance. The general goal is to increase the probability of generating good RES(s), and decrease the probability of generating bad ones.

3.2.1 Encoder

As shown in Figure 3.4a, our encoder will take the points as input, and produce a fixed-length encoding for each point. The dimension of each encoding is $d = 128$ in our implementation. The encodings for n points, $e_1, ..., e_n \in \mathbb{R}^d$, can be packed in an encoding matrix $E \in \mathbb{R}^{n\times d}$ ($E = [e_1, ..., e_n]^T$). The reason to encode the points before RES generation is to create better representations for the points. Before encoding, the representation of each point is just its 2D coordinates. During the encoding, the dimension is expanded to d by linear transformation. The expanded representations of different points will then exchange information for multiple rounds. The final encoding of a point will therefore contain extra information about its neighbors or the environment.

In practice, we implemented a modified multi-head attention encoder [100] to serve as our encoder. The encoder takes the 2D coordinates of the points $V \in \mathbb{R}^{n\times 2}$ (n is not

fixed) as input. It first expands the dimension of the points to get the initial encodings by $E_0 = \text{BatchNorm}(VW_{emb})$, where $W_{emb} \in \mathbb{R}^{2\times d}$ is a projection matrix that can be trained. BatchNorm, a.k.a. batch normalization [50], will normalize each value in the vectors across a mini-batch. Empirically, it makes the training more stable and faster to converge.

This initial encoding is then processed by $N = 3$ identical encoding processes, each of which is illustrated in Figure 3.4b. The output of the i^{th} process is E_i, and $E = E_N$ will be the final output of the encoder. Each of the processes includes two sub-layers - a multi-head attention layer and a feed forward layer. The merging arrows with $+$ marks in Figure 3.4b indicate residual connections [41], i.e. point-wise addition of two inputs. Batch normalization is again used to stabilize the training.

To explain multi-head attention layer, let's first look at the single head attention,

$$\text{SingleHead}(Q, K, M) = \text{softmax}(\frac{QK^T}{\sqrt{d_s}})M$$

where $Q, K, M \in \mathbb{R}^{n\times d_s}$ ($d_s = 16$), and softmax is a function that normalize each row of its input into a probability distribution that sum up to 1. The single head attention is essentially a message passing process. We can imagine that the i^{th} row of M is the message held by the i^{th} point. Assume the result of this SingleHead is $S \in \mathbb{R}^{n\times d_s}$, the i^{th} row of S is no more than a weighted sum of M's rows. That means the information of different points are gathered to form a new representation for point i. The weights are decided by Q and K, and can be learned during training.

The multi-head attention layer can then be defined as

$$\text{MultiHead}(E_i) = \text{Concat}(S_1, ..., S_h)W_m$$
$$\text{where } S_j = \text{SingleHead}(E_iW_{Q,j}, E_iW_{K,j}, E_iW_{M,j})$$

where $h = 16$ is the number of heads, S_j is the output of the j^{th} single head, and the Concat function concatenates h $n \times d_s$ matrices to form one $n \times hd_s$ matrix. $W_m \in \mathbb{R}^{hd_s\times d}$ and $W_{Q,j}, W_{K,j}, W_{M,j} \in \mathbb{R}^{d\times d_s}$ are trainable parameters.

In addition, the feed forward layer applies the following function to each row of the

input matrix.

$$\text{FeedForward}(x^T) = \max(0, x^T W_1 + b_1^T)W_2 + b_2^T$$

where x^T ($x \in \mathbb{R}^d$) is the input row, and $W_1 \in \mathbb{R}^{d \times d_h}$ ($d_h = 512$), $b_1 \in \mathbb{R}^{d_h}$, $W_2 \in \mathbb{R}^{d_h \times d}$ and $b_2 \in \mathbb{R}^d$ are all parameters that can be trained.

3.2.2 Decoder

The decoder takes the encodings E as inputs, and will generate an RES satisfying the two requirements in Theorem 1. The decoder basically will decide the pairs in the RES recurrently, i.e., the previously generated pairs will serve as additional inputs for computing the next pair. The decoder achieves this by producing a stochastic policy, or in another word, computing the probability of generating different pairs given the current state. The state consists of the point set V and the existing partial RES.

More specifically, before generating any pairs, the decoder will first select a point as the starting point u_0 and mark it as visited. In the following $n-1$ steps, a new pair of the RES will be generated at each step. At time step t, rather than generating a pair (v_t, h_t) directly, the decoder will first select an *unvisited* point u_t, and then determine a *visited* point w_t and a boolean s_t simultaneously. If $s_t = 0$, $(v_t, h_t) = (u_t, w_t)$; if $s_t = 1$, the positions of u_t and w_t are swapped, and $(v_t, h_t) = (w_t, u_t)$. This intermediate step is introduced to guarantee that exactly one of h_t and v_t is visited (as required by Theorem 1). Algorithm 2 provides the pseudo-code for our RES generation algorithm.

In order to decide u_0, u_t, w_t and s_t for $t = 1, ..., n-1$, we make use of a pointing mechanism first proposed in [101]. For example, when we want to select a single *unvisited* point, the pointing mechanism takes the encodings $E = [e_1, ..., e_n]^T$ and a query vector $q \in \mathbb{R}^{d_q}$ ($d_q = 360$) as inputs, and output the probability $p \in \mathbb{R}^n$ of selecting each point. The query vector is used to differentiate between different states. The computation behind the

Algorithm 2 RES Generation

Input: $E = [e_1, ..., e_n]^T$
Output: *res*
1: $res = ()$
2: Select a starting point u_0 and mark as visited
3: **for** $t = 1$ to $n - 1$ **do**
4: Select an *unvisited* point u_t
5: Select a *visited* point w_t and a boolean s_t
6: Mark u_t as *visited*
7: **if** $s_t = 0$ **then** $(v_t, h_t) = (u_t, w_t)$
8: **if** $s_t = 1$ **then** $(v_t, h_t) = (w_t, u_t)$
9: Append (v_t, h_t) to *res*
10: **return** *res*

pointing mechanism is as follows.

$$p = \text{PTM}(E, q; \phi) = \text{softmax}(C \times \tanh(l))$$

$$\text{where } l = [l_1, l_2, ..., l_{n-1}, l_n]^T,$$

$$l_i = \begin{cases} -\infty & \text{if point } i \text{ is } \textit{visited} \\ g_\phi^T \tanh(W_\phi^3 e_i + W_\phi^4 q) & \text{otherwise} \end{cases}$$

where $\phi = \{g_\phi \in \mathbb{R}^{d_q}, W_\phi^3 \in \mathbb{R}^{d_q \times d}, W_\phi^4 \in \mathbb{R}^{d_q \times d_q}\}$ is a set of trainable parameters. The vector $l \in \mathbb{R}^n$ are the logits for selecting different points. The larger l_i is, the more likely point i will be selected. The logits for *visited* points are reset to $-\infty$ by a boolean mask to avoid being selected. The logit values are next clipped by $C \times \tanh(l)$ where $C = 10$, so that it will not become so biased towards selecting some certain points and get stuck in local minima easily. Lastly, the softmax function converts the logits into a probability distribution.

Using this mechanism, the decoder will first compute the probabilities of selecting each point as the starting point u_0 (given point set V) by

$$p(u_0|V) = PTM(E, 0; \phi_1) \tag{3.1}$$

which means vector zero is used as the query vector, and the parameter set ϕ_1 is used for computation.

After having the starting point, the decoder will decide (v_t, h_t) by deciding (u_t, w_t, s_t) at time step $t = 1, ..., n-1$. The decoder will first compute the probabilities of selecting each point as u_t (given the partial RES before t, or $res(< t)$) by

$$p(u_t|V, res(< t)) = PTM(E, q_t; \phi_2) \tag{3.2}$$

where a different parameter set ϕ_2 is used. The query vector q_t encoding the current state $res(< t)$, is computed recurrently as follows,

$$\begin{aligned} edge_t &= W_u e_{u_t} + W_w e_{w_t} + W_v e_{v_t} + W_h e_{h_t} \\ subtree_t &= \max(subtree_{t-1}, W_{edge} edge_t) \\ q_t &= \max(0, edge_{t-1} + subtree_{t-1}) \end{aligned} \tag{3.3}$$

where $W_u, W_w, W_v, W_h \in \mathbb{R}^{d_q \times d}$ and $W_{edge} \in \mathbb{R}^{d_q \times d_q}$ are trainable parameters. Vector $edge_t$ is a representation of the t^{th} generated edge (pair). Note that it is not redundant to have both (v_t, h_t) and (u_t, w_t) in the computation, as this can actually help the model to differentiate the role of each participating point. The representations of all edges until time step t are aggregated by point-wise maximum to form a representation of the existing sub-tree $subtree_t$ at time step t.

Next, a *visited* point w_t and a boolean s_t are selected simultaneously based on the existing partial RES and the u_t just selected by an extended pointing mechanism

$$\begin{aligned} &p' = \text{EPTM}(E, q'; \phi_3, \phi_4) = \text{softmax}(C \times \tanh(l')) \\ &\text{where } l' = [l'_{1,0}, ..., l'_{n,0}, l'_{1,1}, ..., l'_{n,1}]^T, \\ &l'_{i,s} = \begin{cases} -\infty & \text{if point } i \text{ is } \textit{unvisited} \\ g_{\phi_3}^T \tanh(W_{\phi_3}^3 e_i + W_{\phi_3}^4 q') & \text{else if } s = 0 \\ g_{\phi_4}^T \tanh(W_{\phi_4}^3 e_i + W_{\phi_4}^4 q') & \text{else if } s = 1 \end{cases} \end{aligned}$$

The major difference is that $EPTM$ will produce $2 \times n$ probabilities instead of n. Different parameter sets are used to compute the logits for the case when $s = 0$ and $s = 1$. We

compute the probabilities of choosing each pair of w_t and s_t given V and $res(< t)$ by

$$p(w_t, s_t | V, res(< t), u_t) = EPTM(E, q'_t; \phi_3, \phi_4) \tag{3.4}$$

The query vector for this purpose is obtained by adding the knowledge of the u_t just selected to Equation (3.3):

$$q'_t = \max(0, edge_{t-1} + subtree_{t-1} + W_5 e_{u_t})$$

where $W_5 \in \mathbb{R}^{d_q \times d}$ is learned.

Note that we can use equation 3.1, 3.2 and 3.4 to compute the probability of generating a specific RES given a point set V and the actor parameter set θ by

$$p_\theta(res|V) = p_\theta(u_0|V) \prod_{t=1}^{n-1} p_\theta(u_t|V) p_\theta(w_t, s_t|u_t, V)$$

This will be useful when updating the parameters, as we want to increase the probability of generating good RES(s).

3.2.3 Critic and Parameters Update

The critic network is used to assist the learning process. The critic will try to set a baseline by predicting the length of the RSMT found by the actor. Assume that the actual length of the RSMT constructed by the actor is $L(V, res)$ (obtained by Algorithm 1) and the critic prediction is $b(V)$, we will judge the performance of the actor for this specific point set using the advantage value computed as $a(V, res) = b(V) - L(V, res)$. This tells us how much the actor performs better than the expectation.

The critic adopts the same encoder as the actor but with different parameters, and will first encode the points into the critic encodings $E' \in \mathbf{R}^{n \times d}$. The baseline $b(V)$ will next be computed as

$$\text{glimpse}(E') = \text{softmax}(\tanh(E')g')^T E'$$

$$b(V) = \max(0, \text{glimpse}(E')W_6 + b_6^T)W_7 + b_7$$

where $g' \in \mathbb{R}^d$, $W_6 \in \mathbb{R}^{d \times d_c}$ ($d_c = 256$), $b_6 \in \mathbb{R}^{d_c}$, $W_7 \in \mathbb{R}^{d_c \times 1}$ and $b_7 \in \mathbb{R}$ are all learnable

parameters. The glimpse function will compute a weighted sum of the encodings according to the weights $\text{softmax}(\tanh(E')g')$. It is followed by two fully connected layers to yield the baseline scalar.

So far we have introduced all the architectures of our neural network model. In the following, we will discuss our method to update the parameters of the actor θ and the parameters of the critic ψ.

For a specific point set V, the expected advantage of the RSMT generated by the actor is

$$J(\theta|V) = \sum_{r \in R} (b(V) - L(V, r)) p_\theta(r|V) \tag{3.5}$$

where R is the set of all valid RES. We use the REINFORCE algorithm [106] to compute the gradient of Equation (3.5) as

$$\begin{aligned} \nabla_\theta J(\theta|V) &= \sum_{r \in R} (b(V) - L(V, r)) \nabla_\theta p_\theta(r|V) \\ &= \sum_{r \in R} (b(V) - L(V, r)) (\nabla_\theta \log p_\theta(r|V)) p_\theta(r|V) \end{aligned} \tag{3.6}$$

In practice, we will draw B point sets $V_1, ..., V_B$ and sample a single RES for each set, i.e. $r_i \sim p_\theta(res|V_i)$ for $i = 1, ..., B$. We approximate the gradient in Equation (3.6) by Monte Carlo sampling, i.e., using a single trial for each point set to approximate the expectation, and take average to get the overall gradient,

$$\nabla_\theta J(\theta) \approx \frac{1}{B} \sum_{i=1}^{B} (b(V_i) - L(V_i, r_i)) \nabla_\theta \log p_\theta(r_i|V_i) \tag{3.7}$$

We update the parameters of the actor using stochastic gradient ascent.

On the other hand, we train the critic to predict the actual length of the RSMT found by the actor, and minimize the mean square error as follows.

$$Loss(\psi) = \frac{1}{B} \sum_{i=1}^{B} ||b_\psi(V_i) - L(V_i, res_i)||_2^2$$

We use stochastic gradient descent to update the parameters of the critic.

3.3 Experiment and Results

We implement and train REST with PyTorch and run our experiments on a 64-bit Linux machine with Intel Xeon 2.2GHz CPUs and an Nvidia Titan V GPU.

We train the model with random point sets from degree 3 to 50, and keep a set of parameters, a.k.a. checkpoint, for each degree. The training for degree $n+1$ will start after that of n, and will kick start using the parameters learned for degree n. This will speed up the training process, since the model trained with degree n nets is already a fairly good solver for degree $n+1$. For each degree, we will train the model for $40k$ iterations, and a mini-batch of B point sets are fed into the model for each iteration. The batch size varies for different degrees. We use a batch size of $B = 4096$ for degree 3, and will halve that at degree 10, 20 and 40 respectively. We use Adam optimizer [63] to train our model with an initial learning rate of 2.5×10^{-4}. The learning rate will decay by 0.96 after the training of each degree.

3.3.1 Results on Randomly Generated Nets

When testing, we will greedily pick the action assigned with the largest probability at each step to produce one RES for each net. We use a batch size of $100k/degree$ for each degree during testing. Besides, due to the stochastic nature of machine learning based methods, it is possible to further improve the quality by introducing variations. We found 8 transformations that can provide such variations without changing the RSMT solutions. The 8 transformations include rotating the point set by 0, 90, 180 and 270 degrees, with or without the x and y coordinates swapped. We utilize this feature by feeding T ($1 \leq T \leq 8$) transformed versions into the model, and pick the best solution.

We compare REST with GeoSteiner (optimal), an efficient implementation of R-MST [6], BGA [53], FLUTE [108] with default setting ($A = 3$) and the most accurate setting ($A = 18$) as mentioned in their work. We test the algorithms on $10k$ randomly generated point sets for each degree. We show the results for every 5 degree in Table 3.1, and their runnings in

Table 3.1: *Average Percentage Errors (%) for RSMT Construction*

Deg	GeoSt	R-MST	BGA	FLUTE (A = 3)	REST (T = 1)	FLUTE (A = 18)	REST (T = 8)
5	0.00	10.91	0.23	**0.00**	0.02	**0.00**	**0.00**
10	0.00	11.96	0.48	0.12	0.23	0.04	**0.01**
15	0.00	12.19	0.53	0.55	0.45	0.06	**0.03**
20	0.00	12.41	0.57	1.03	0.56	0.11	**0.07**
25	0.00	12.47	0.58	1.44	0.69	0.18	**0.12**
30	0.00	12.56	0.60	1.83	0.77	0.23	**0.16**
35	0.00	12.63	0.62	2.13	0.84	0.26	**0.21**
40	0.00	12.65	0.63	1.05	0.86	0.29	**0.25**
45	0.00	12.67	0.63	1.07	0.98	**0.30**	0.32
50	0.00	12.72	0.64	1.12	1.01	**0.29**	0.36

Table 3.2: *Time to Construct* 10*k RSMT for Each Degree (s)*

Deg	GeoSt	R-MST	BGA	FLUTE (A=3)	REST (T=1)	FLUTE (A = 18)	REST (T = 8)
5	0.25	0.03	0.41	0.01	0.26	0.01	1.84
10	3.88	0.13	1.63	0.05	0.38	0.10	2.67
15	8.32	0.33	3.33	0.13	0.55	2.02	4.20
20	15.39	0.60	5.36	0.19	0.64	7.87	5.56
25	23.24	0.93	7.04	0.24	0.97	16.68	8.02
30	31.93	1.02	10.11	0.34	1.20	21.67	9.39
35	43.41	1.18	11.92	0.40	1.52	26.17	12.00
40	55.68	1.39	14.71	1.22	1.76	33.31	13.60
45	71.69	1.63	17.44	1.44	2.30	42.12	17.02
50	87.45	1.87	20.63	1.64	2.66	52.76	19.09

Table 3.2. REST is almost as fast as FLUTE ($A = 3$) when choosing $T = 1$, and outperforms both BGA and FLUTE ($A = 18$) w.r.t. both quality and speed by choosing $T = 8$.

3.3.2 Results on Real Nets

Besides, we also test our model on the benchmarks of ICCAD 2019 global routing contest [29] (ispd18_test{1-10}, ispd19_test{1-10}). We test 1.72 million nets having degree 3 to 100. We use the corresponding checkpoints for nets of degree 3 to 50, and use the degree 50 checkpoint for degree 51 to 100. It takes our algorithm ($T = 1$) 66.60s to construct all RSMTs, and the total length is just 1.008$\times$ optimal length.

Note that, despite the result, it is not very beneficial to use REST for global routing on these academic benchmarks yet. On the one hand, over 90% percent of the nets in the ICCAD 2019 benchmarks have a degree below 10, for which REST has no particular advantage over FLUTE. On the other hand, for the moment, REST still cannot well handle larger nets and produce results as good as other heuristics. Nevertheless, REST explores a new direction for solving hard combinatorial optimization problems in EDA like RSMT construction, especially considering the state-of-the-art heuristic was proposed more than ten years ago and hasn't been challenged ever since.

3.3.3 Inference Examples

To better illustrate the inference algorithm. The process that REST builds an RSMT for a degree 10 net from scratch is shown in Figures 3.5 to 3.7. The green outlines show the probability of selecting each point or each edge in a step. Thicker outline indicates higher probability. The red points or edges are the selected point or edge in each step. In step 0, a starting point is selected. In step x.1, a point outside the tree is selected, whereas in step x.2, an edge is selected to connect the point to the tree. The process will stop until all points are connected.

We show example solutions for a degree 20 net and a degree 40 net in Figure 3.8 and Figure 3.9 respectively. It can be seen that REST can often find good enough alternatives

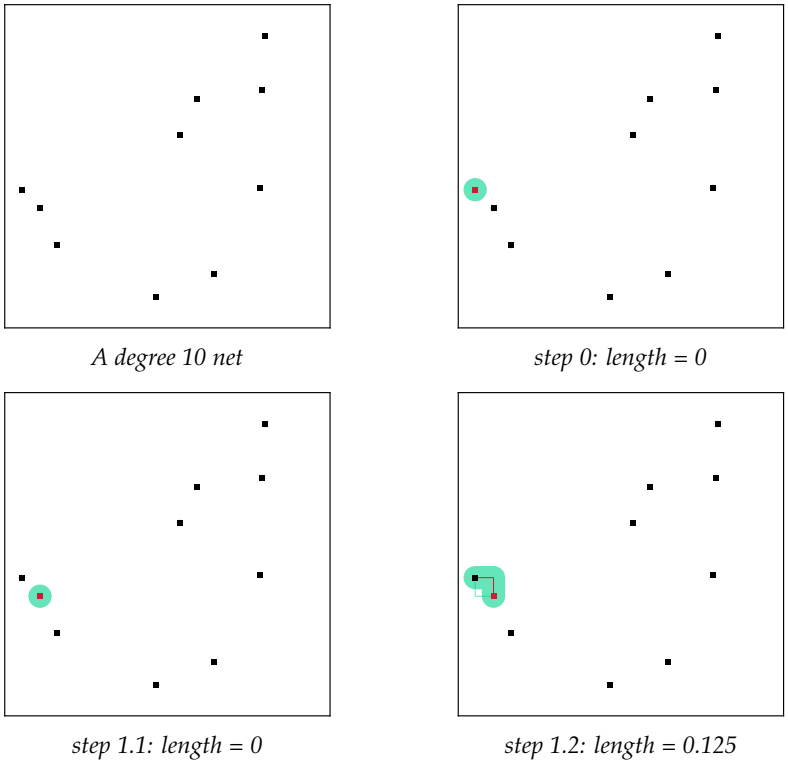
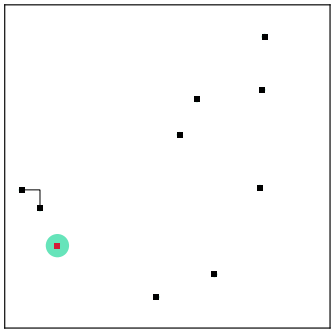
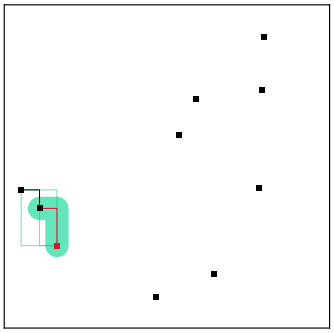


Figure 3.5: *Building an RSMT for a Degree* 10 *Net by REST (step 0 to 3).*

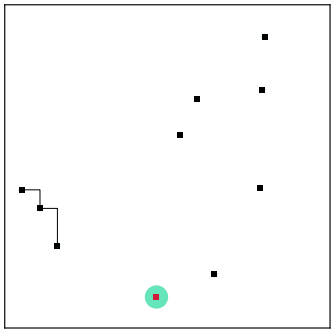
even if it does not find the optimal RSMTs.



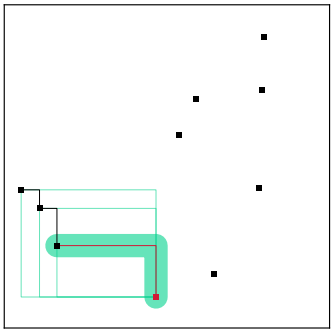
step 2.1: length = 0.125



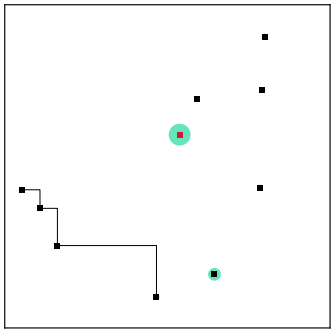
step 2.2: length = 0.311



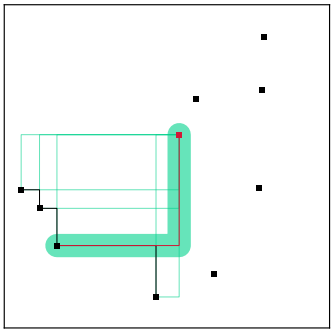
step 3.1: length = 0.311



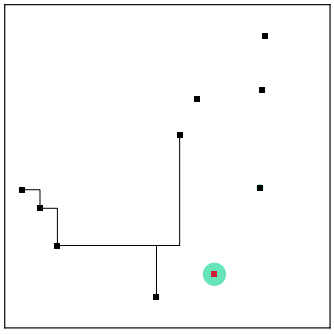
step 3.2: length = 0.821



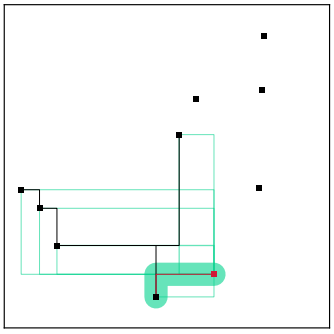
step 4.1: length = 0.821



step 4.2: length = 1.276

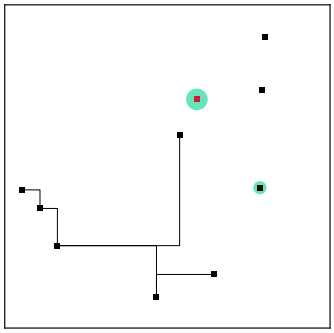


step 5.1: length = 1.276

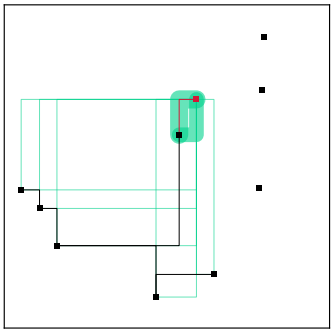


step 5.2: length = 1.472

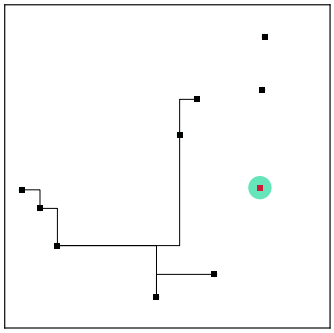
Figure 3.6: *Building an RSMT for a Degree* 10 *Net by REST (step 4 to 7).*



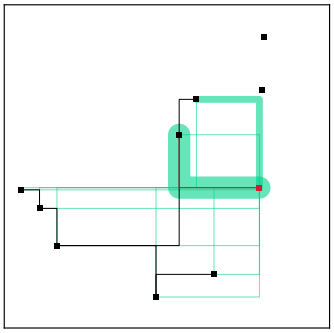
step 6.1: length = 1.472



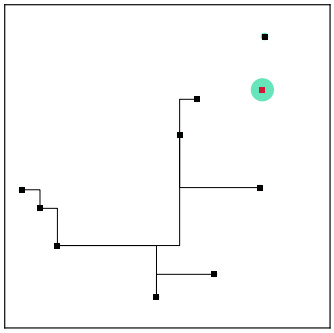
step 6.2: length = 1.651



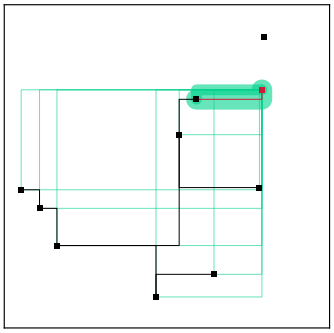
step 7.1: length = 1.651



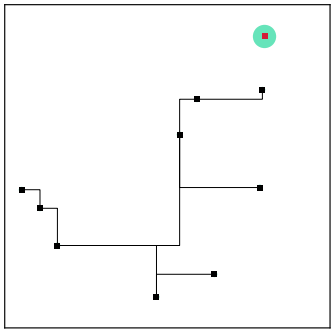
step 7.2: length = 1.923



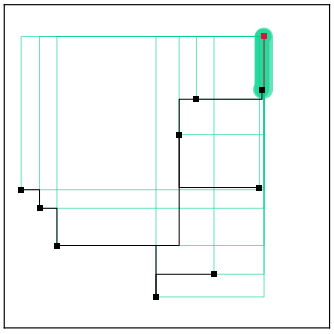
step 8.1: length = 1.923



step 8.2: length = 2.176

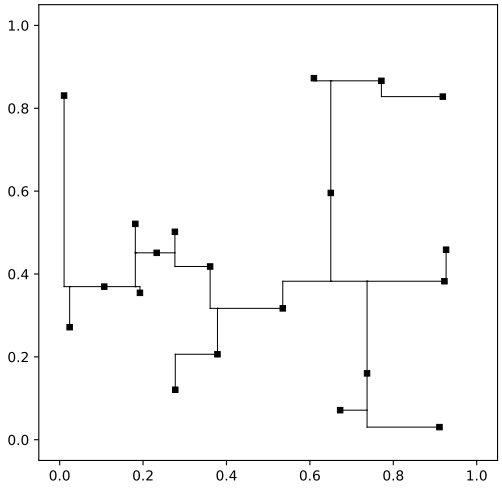


step 9.1: length = 2.176

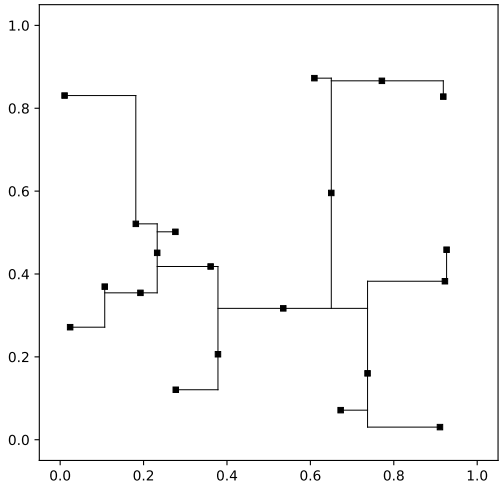


step 9.2: length = 2.365

Figure 3.7: *Building an RSMT for a Degree* 10 *Net by REST (step 8 to 9).*

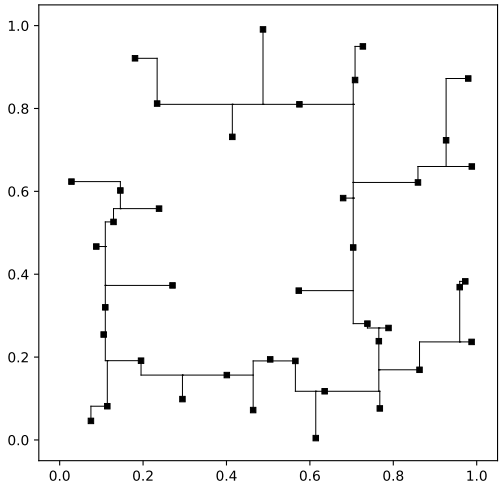


GeoSteiner (optimal): length = 3.704

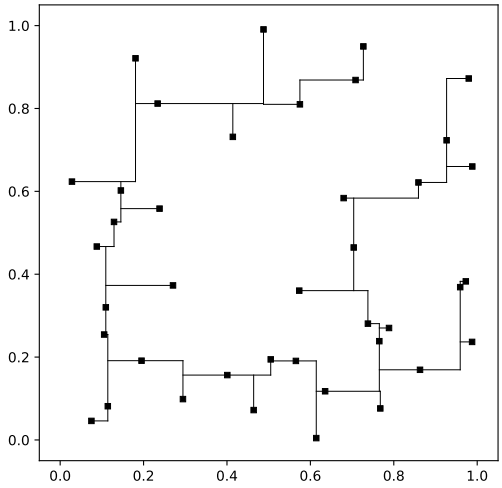


REST: length = 3.710

Figure 3.8: *Example Solution for a Degree* 20 *Net.*



GeoSteiner (optimal): length = 5.177



REST: length = 5.212

Figure 3.9: *Example Solution for a Degree* 40 *Net.*

Chapter 4

Global Routing for Detailed Routability

In this chapter, we propose a detailed-routability-driven 3D global router with probabilistic resource model. The proposed router utilizes a probability-based cost scheme, and routes the nets directly in the 3D space with two main phases, 3D pattern routing and two-level 3D maze routing. In 3D pattern routing, the nets are broken down into two-pin nets, and a dynamic programming based algorithm will route the two pin nets sequentially using L-shape patterns and stacking vias at the turns. In the two-level 3D maze routing phase, the grid graph is coarsened to shrink the routing space, and maze routing is first performed in the coarsened space with an objective to find a routing region with the highest routability. A fine-grained maze routing will then search for a lowest cost path within the region. Besides, we also propose a new technique called patching to add some stand-alone route guides or patches onto the routed path to further improve the detailed routability. Note that, the source code of this work is released on Github[1].

Our main contributions in this work include:

1. A sophisticated probability-based cost scheme minimizing the possibility of overflow after detailed routing.

[1] https://github.com/cuhk-eda/cu-gr

2. An optimal 3D pattern routing technique that combines 2D pattern routing and layer assignment, and is able to route a majority of the nets without overflow, even if only L-shapes are used.

3. A two-level maze routing utilizes two levels of routing - a coarsened level that searches for a region with the best routability, and a finer level that searches for a lowest cost solution within the region.

4. A patching technique that adds useful route guides to further improve the detailed routability.

The rest of the chapter is organized as follows: Section 4.1 introduces the problem formulation, evaluation metrics and terminologies. Our proposed algorithm will be discussed in Section 4.2. Section 4.3 presents the experiments and results.

4.1 Preliminary

4.1.1 Problem Formulation

In the traditional global routing problem, the 3D routing region is represented by a set of global cells obtained by a set of evenly distributed horizontal and vertical grid lines. A grid graph $G(V, E)$ can then be defined by treating each global cell as a vertex ($v \in V$) and creating an edge ($e \in E$) between every two adjacent global cells. We call the edge between two global cells on the same layer *wire edge*, and the capacity of a wire edge is equal to the number of tracks that can go through the edge. The edge between two global cells with the same 2D coordinates but on different layers is called *via edge*, and its capacity is put as infinity by many 2D routers (but not in this work).

Traditional global routing problem can be defined as, given a grid graph and a set of nets to be routed, find a path for each net such that all the pins of a net are connected without overflow and the overall wirelength is minimized. However, in the ICCAD'19 global routing contest, the problem is formulated in a slightly different way. Rather than merely

path finding, the global routers are required to produce a set of connected rectangle guides, each containing an integral number of global cells, so that the detailed router can find a path within the guides to connect all pins of each net and minimize a give cost function.

4.1.2 Evaluation Metrics

In the past, the quality of a global routing algorithm is usually measured by the total wirelength, the number of vias used and the number of overflows on the global routing edges. Although this evaluation method is fast and straightforward, it cannot reflect the detailed routability of the solution accurately. The discrepancy between the global routing solution evaluation and the detailed routing performance is large in many cases. In this work, we adopted the same evaluation method in the ICCAD'19 global routing contest. The output of the global router is a set of connected rectangular route guides which will be fed into an academic detailed router called Dr.CU, and the score is calculated based purely on the quality of the detailed routing solution according to the metrics in Table 4.1.

Metric	Weight
length of wire	0.5
num of vias	4
length of wrong-way wire	1
num of off-track vias	1
length of off-track wires	0.5
length of out-of-guide wires	1
num of out-of-guide vias	1
num of min-area violations	500
num of spacing violations	500
num of short violations	500
short metal area / metal2 pitch	500

Table 4.1: *Evaluation Metrics for Detailed Routed Solution.*

4.1.3 Terminologies

Our algorithm is designed based on three major concepts - capacity, demand and resource, all of which can be used to describe a wire edge or a global cell. For example, we have both the capacity of an edge and the capacity of a global cell. Since all the following discussion will revolve around these three concepts, a brief explanation for each is given below:

Capacity: The capacity of a wire edge $e(u, v)$ denoted by $c(u, v)$ is the number of tracks going through that edge, which is also roughly the maximum number of nets that can utilize the edge. The capacity of a global cell u, $c(u)$, is then defined as the average capacity of its two abutting wire edges.

Demand: The demand of an edge, $d(u, v)$, is the part of the capacity that is already used by the routed nets. It is the sum of two parts - demand by wires and demand by vias. The demand by wires is simply the number of wires going through the edge, while the demand by vias is an estimated number of tracks affected by the vias in the edge region. For instance, the demand of edge $e(u, v)$ is calculated using the the first equation in Equation (4.1a),

$$d(u, v) = wires(u, v) + 1.5 \times \sqrt{\frac{vias(u) + vias(v)}{2}}, \tag{4.1a}$$

$$d(v) = \frac{wires(u, v) + wires(v, w)}{2} + 1.5 \times \sqrt{vias(v)}, \tag{4.1b}$$

where $wires(u, v)$ is the number of wires going through the edge and $vias(u)$ is the number of vias that have metals inside global cell u. Note that a stacking via that enters a global cell from below (above) and exit from above (below) is counted as two.

The demand of a global cell, $d(u)$, is defined similarly. It is also the sum of the demand by wires and the demand by vias. However, the demand by wires is replaced by the average demand by wires of its two abutting edges, and the demand by vias only considers the vias in the global cell, as shown in Equation (4.1b). Note that u and w are the two global cells adjacent to v in the preferred routing direction.

Resource: The resource of an edge or a global cell is the part of the capacity that can still be utilized by the nets to be routed. Its relationship with the capacity and demand is shown in Equation (4.2), in which the resource of an edge $e(u,v)$ is denoted as $r(u,v)$ and the resource of a global cell u is denoted as $r(u)$.

$$r(u,v) = c(u,v) - d(u,v) \tag{4.2a}$$

$$r(u) = c(u) - d(u) \tag{4.2b}$$

Neither capacity nor demand can well describe the degree of congestion, but the remaining resource will be a good indicator of congestion and how big the capacity margin is.

4.2 Proposed Algorithm

4.2.1 Overview

Figure 4.1 presents the overall flow of the proposed algorithm, which can be divided into three parts: initial routing, two-level maze routing and route guide generation. Note that, both intial routing and mutli-level maze routing operate in the 3D space.

4.2.2 Cost Scheme

Our carefully crafted cost scheme, taking wirelength, possibility of overflow and ability of detailed router to solve overflow into consideration, is capable of minimizing wirelength and via number, while avoiding overflow and reserving enough capacity for later use. To discuss the proposed cost scheme, note that a path of a net is formed by a set P of wire edges and vias, and the total cost of the path will be the sum of the costs of all those edges as computed by Equation (4.3).

$$cost(P) = \sum_{e(u,v)\in P} cost_w(u,v) + \sum_{via(u,u')\in P} cost_v(u,u'), \tag{4.3}$$

where $cost_w(u,v)$ is the cost of a wire edge and $cost_v(u,u')$ is the cost of a via edge. The wire edge cost is comprised of two parts - wirelength cost and congestion cost, and is computed

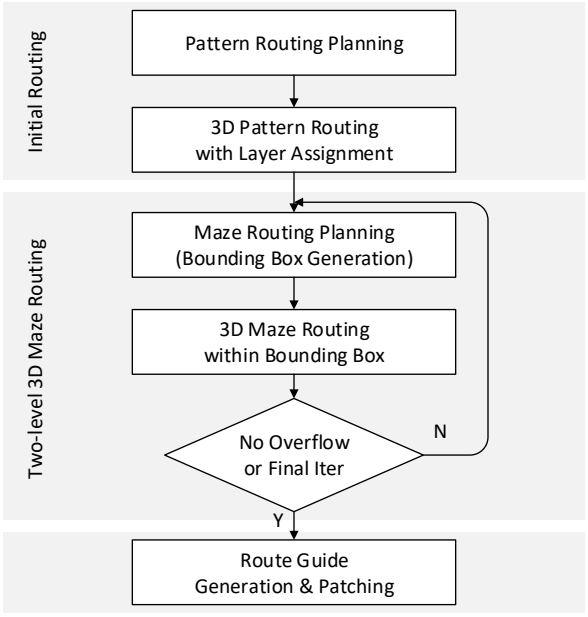


Figure 4.1: *Overall Flow of the Proposed Algorithm.*

using the formulae in Equation (4.4),

$$cost_w(u,v) = wl(u,v) + eo(u,v) \times lg(u,v), \tag{4.4a}$$

$$eo(u,v) = wl(u,v) \times \frac{d(u,v)}{c(u,v)} \times uoc, \tag{4.4b}$$

$$lg(u,v) = (1.0 + exp(slope \times r(u,v)))^{-1}. \tag{4.4c}$$

$wl(u,v)$ is the wirelength of the wire edge (which is also its wirelength cost, since the coefficient for wirelength cost is one). The product of $eo(u,v)$ and $lg(u,v)$ forms the congestion cost, where $eo(u,v)$ is the expected overflow cost and $lg(u,v)$ is a logistic function of $r(u,v)$. The expected overflow cost is calculated by multiplying the wirelength of the edge being considered by the possibility of overflow, $d(u,v)/c(u,v)$, and the unit length overflow cost uoc (a given constant). Note that the possibility of overflow, computed as the ratio between demand and capacity, is accurate if the detailed router adopts the simplest strategy of picking a track randomly to route. However, most well designed detailed routers will do much better than random selection. To handle this, we use a logistic function to model the ability of a detailed router to avoid overflow. The variable $slope$ of the

logistic function is an adjustable parameter that determines the global router's sensitivity to overflow. The formation of the congestion cost of a wire edge is illustrated in Figure 4.2. When the resources are abundant, there is almost no congestion cost, but the cost will increase rapidly as the resources are being used up and will keep increasing almost linearly after all the resources are used. This is because when the resources are used up, the expected overflow cost will dominate the congestion cost.

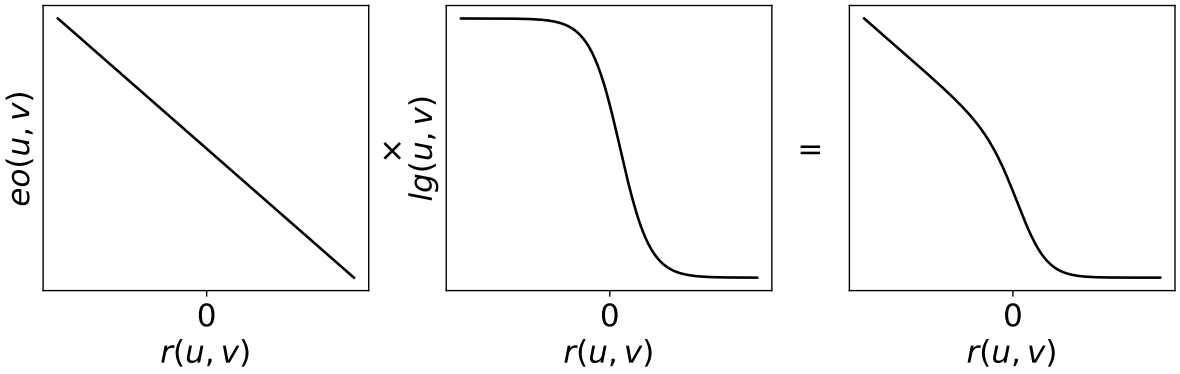


Figure 4.2: *Congestion Cost Composition.*

Many global routers ignore the impact of vias before the layer assignment stage, thanks to our 3D pattern routing strategy, a via cost scheme can be embedded to reflect the impact. Suppose u and u' are the lower and upper global cells connected by a via, the via cost of $via(u, u')$ can be computed using Equation (4.5), where uvc represents the given unit via cost. The logistic function with the same $slope$ value is similar to that for wire edge, and can model the ability of the detailed router to avoid overflow in either global cell u or u'. Note that for simplicity, no expected overflow cost for vias is included.

$$cost_v(u, u') = uvc \times (1 + lg(u) + lg(u')), \tag{4.5a}$$

$$lg(u) = (1.0 + exp(slope \times r(u)))^{-1}. \tag{4.5b}$$

4.2.3 Initial Routing / 3D Pattern Routing

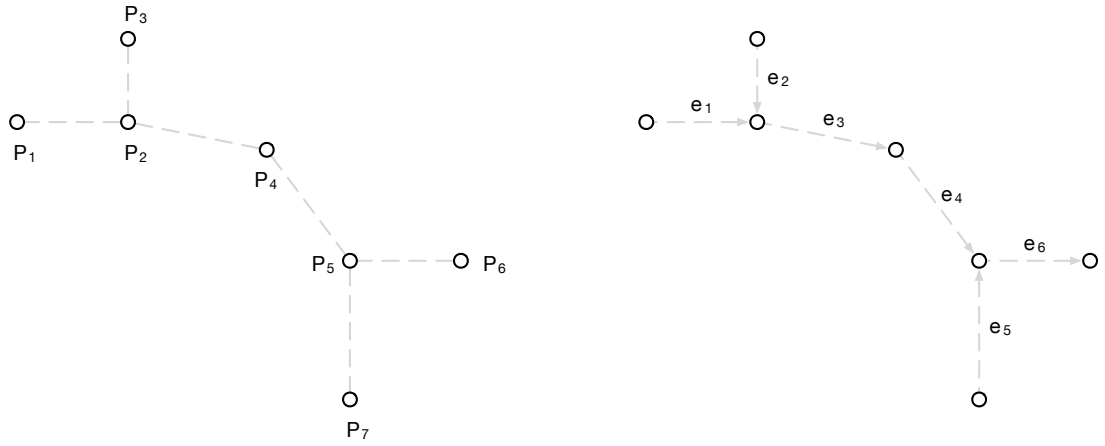
Traditional pattern routing generates 2D topologies only, while our proposed 3D pattern routing directly generates 3D topologies without the need of an extra layer assignment

stage. Take L-shape routing as an example, traditional pattern routing will only choose between 2 possible paths for each two-pin net, and some of the paths may turn out to be impossible to be routed after layer assignment even if they did not cause any overflow in the 2D grid graph. However, the proposed 3D pattern routing combines 2D pattern routing and layer assignment. It chooses a path among $2 \times L \times L$ possible options for every two-pin net, where L is the number of layers. With the help of dynamic programming, an optimal solution with respect to our cost scheme on wirelength and congestion is guaranteed to be found if it exists.

In order to perform 3D pattern routing, each multi-pin net will first be broken down into a set of two-pin nets in a step called pattern routing planning. In the same step, the order in which they are routed will also be determined. A dynamic programming algorithm is then conducted to pattern route the two-pin nets and minimize the overall cost.

a) Pattern Routing Planning: During the planning stage, we will first utilize FLUTE[23] to generate a rectilinear Steiner minimum tree (RSMT) for each multi-pin net so as to guide the pattern routing to produce shorter wirelength. FLUTE is an RSMT construction algorithm adopting a look-up table approach, which is both fast and optimal for low-degree nets. However, FLUTE is unaware of routing congestion. We use a technique called edge shifting described in [90] to alleviate the problem. The multi-pin net will then be broken down into a set of two-pin nets in such a way that every two points sharing an edge in the RSMT will form a two-pin net.

Next, we will determine the order in which the two-pin nets will be routed. A degree-one node in the RSMT will be randomly picked as the root, and a depth first search (DFS) traversal will be performed to visit all the other nodes. The two-pin nets will be routed in the reverse order in which they are visited in this traversal. For example, suppose we pick P_6 of the RSMT in Figure 4.3a as the root, and the DFS traversal visits the nodes in the order of P_6, P_5, P_7, P_4, P_2, P_3, and P_1. We then label the two-pin nets in the reverse order as e_1, e_2, e_3, e_4, e_5 and e_6 as shown in Figure 4.3b, which is also the order in which they will be routed. Note that each two-pin net will be routed from destination to source, and the arrows of the



(a) *Two-Pin Nets Obtained from RSMT* **(b)** *Two-Pin Net Routing Order*

Figure 4.3: *Two-Pin Nets Ordering.*

edges indicate the directions that they will be routed. We call the point P_i pointed to by an out-going edge from P_j the parent of P_j, and we say P_j is a child of P_i or $P_j \in ch(P_i)$.

b) Dynamic Programming: Having the nets been broken down into two-pin nets and their order decided, a dynamic programming algorithm is designed to perform pattern routing and layer assignment simultaneously. Our dynamic programming algorithm is similar to that introduced in [67]. However, their algorithm only performs layer assignment after a net is pattern routed, while our algorithm combines the two steps so as to avoid loss of accuracy caused by compressing 3D grid graph to 2D.

First, we define $S(P_i)$ as the sub-tree including P_i and all its descendants, and $S(P_i, P_j)$ as the sub-tree including P_i, P_i's child P_j and all the descendants of P_j. Take the net in Figure 4.3 as an example, $S(P_2)$ refers to the sub-tree comprised of $\{P_2, P_1, P_3\}$ and $S(P_4, P_2)$ refers to the sub-tree comprised of $\{P_4, P_2, P_1, P_3\}$. Most importantly, as the building blocks of our dynamic programming algorithm, we define minimum sub-tree cost, $msc(P_i, l)$, as the minimum cost of routing the sub-tree $S(P_i)$ rooted at $P_{i,l}$, the global cell at position P_i and on the l^{th} layer. Similarly, we define $msc(P_i, P_j, l)$ as the minimum cost of routing the sub-tree $S(P_i, P_j)$ rooted at $P_{i,l}$.

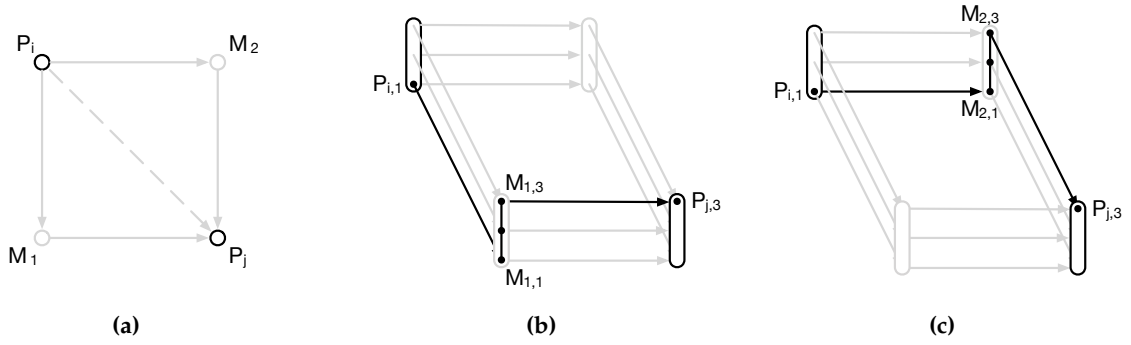


Figure 4.4: *Two-Pin Net 3D L-Shape Routing.*

For each two-pin net $P_i \rightarrow P_j$, we'll first calculate $msc(P_i, l)$ for $l = 1, ..., L$ using Equation (4.6), assuming there are totally L layers, and P_i has k children $\{P_{c(1)}, ..., P_{c(k)}\}$.

$$msc(P_i, l) = \min_{\substack{1 \le l_1 \le L \\ \dots \\ 1 \le l_k \le L}} (cost(V(P_i, l_1, ..., l_k, l)) + \sum_{j=1}^{j \le k} msc(P_i, P_{c(j)}, l_j)). \tag{4.6}$$

Note that in Equation (4.6), $V(P_i, l_1, ..., l_k, l)$ is the set of vias needed to connect all the k children sub-trees and pins to $P_{i,l}$ if they exist, and the cost function is simply the one described in Equation (4.3). For example, if a node P_i has k children and the pin itself is on the l_p^{th} layer, to connect them all to $P_{i,l}$ will need vias at P_i from the $min(l_1, ..., l_k, l_p, l)^{th}$ layer to the $max(l_1, ..., l_k, l_p, l)^{th}$ layer. If a node P_i has no children, $msc(P_i, l)$ will only consist of the via cost. After that, all possible 3D L-shapes to connect $P_i \rightarrow P_j$ will be considered to update $msc(P_j, P_i, l)$ for $l = 1, ..., L$ by,

$$msc(P_j, P_i, l) = \min_{\substack{1 \le l_i \le L \\ x=1,2}} (cost(Path(P_{i,l_i}, M_{x,l_i}, M_{x,l}, P_{j,l})) + msc(P_i, l_i)). \tag{4.7}$$

$Path(P_{i,l_i}, M_{x,l_i}, M_{x,l}, P_{j,l})$ is an L-shape path with two wire segments, $P_{i,l_i} \rightarrow M_{x,l_i}$ and $M_{x,l} \rightarrow P_{j,l}$, and a stack of vias, $M_{x,l_i} \rightarrow M_{x,l}$. The variable x specifies the two possible options of the L-shape route, and the turning position of the L-shape path is denoted by $M_{x,-}$. Figure 4.4a shows the 2D structure of a two-pin net. Suppose there are three metal layers without prefered direction, Figure 4.4b and Figure 4.4c show the 3D structure of the two possible paths to connect $P_{i,1}$ and $P_{j,3}$. In Figure 4.4b, M_1 is chosen as the turning

position. There are wires connecting $P_{i,1}$ and $M_{1,1}$ on the first layer, and wires connecting $M_{1,3}$ and $P_{j,3}$ on the third layer. The two layers are connected by two vias from $M_{1,1}$ to $M_{1,3}$. Figure 4.4c shows another path for which M_2 is chosen as the turning position.

After the minimum costs of all sub-trees are computed by Equations (4.6) and (4.7), the minimum sub-tree cost of the root is the final cost ($\min_{0 \leq l \leq L} msc(P_6, l)$ in our example) and we can then trace back the solution to find the path achieving the minimum cost.

4.2.4 Two-level 3D Maze Routing

After initial routing, those nets with violations will be ripped up and may go through multiple iterations of rip-up and reroute (RRR) by maze routing. However, maze routing on the whole 3D grid graph G will be very time consuming due to the large search space. One alternative is to restrict the searching area to the bounding box of the net. Although this approach makes rerouting much faster, the routing quality will be compromised. Considering the above pros and cons, we propose a two-level 3D maze routing to enable whole-graph searching, while achieving a good trade-off between runtime and routing quality. The proposed routing technique has two levels, maze route planning and fine-grained maze routing within guides, each of which serves a different purpose. Maze route planning aims at finding a smaller but highly routable search space, while fine-grained maze routing seeks to find a path with minimum actual cost within the search space.

First, we introduce an idea called grid graph coarsening, which means to compress a block of global cells (5x5 in our implementation) into a coarsened cell, like what is shown in Figure 4.5. The resource $R(A)$ of a coarsened cell A is computed as the average resource of the global cells in A. We then use the following formula to model the edge cost between two adjacent coarsened cells.

$$C_W(A, B) = s \times \left(\frac{1}{\max(R(A), 0.1)} + \frac{1}{\max(R(B), 0.1)}\right), \tag{4.8a}$$

$$C_V(A, B) = \frac{1}{\max(R(A), 0.1)} + \frac{1}{\max(R(B), 0.1)}. \tag{4.8b}$$

In Equation (4.8), $C_W(A, B)$ represents the edge cost between coarsened cells A and B when

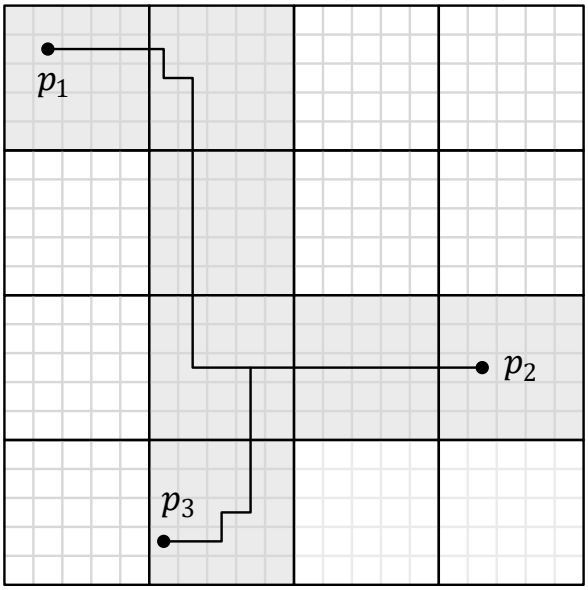


Figure 4.5: *Two-level 3D Maze Routing.*

they are on the same layer, while $C_V(A, B)$ represents the edge cost when the two cells are on adjacent layers. The parameter s refers to the coarsening scale, i.e., the number of global cells in each row (column) of the coarsened cell. In our implementation, s is set to be 5. We multiply the cost by the parameter s because moving from the center of a coarsened cell A to that of its neighboring coarsened cell B on the same layer, the wire will pass through s fine-grained edges. However, moving from the center of A to that of B on an adjacent layer, only one via is needed. This cost scheme for maze routing planning is highly sensitive to resource changes, and the cost will double when the resource is reduced by half.

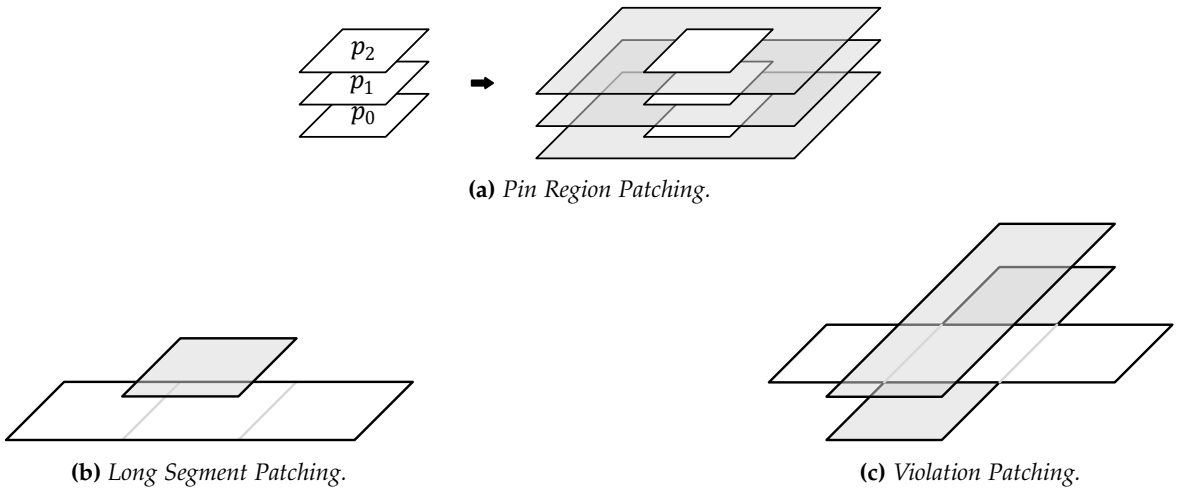
In this way, a coarsened grid graph G_c with a much smaller size than the original graph G can be constructed, with a different cost scheme to enable the global router to navigate in G_c. The grey area in Figure 4.5 shows an example solution of this maze route planning step. The solution is then transformed into bounding boxes and fed into the fine-grained maze router. The fine-grained maze router will then perform another level of maze routing with the cost scheme discussed in Section 4.2.2 within these bounding boxes, aiming at finding a minimum cost path. An example solution of the fine-grained maze routing step is shown as the dark black path in Figure 4.5.

4.2.5 Postprocessing / Guide Patching

Since the output of the global router is connected rectangular route guides, not restricted to paths of single global cell width, we can add new guides to improve detailed routability, so long as the guides still forms a connected path. For this reason, we develop a technique called patching - adding new stand-alone guides to alleviate routing hot spots. We propose 3 kinds of patching to address hot spots that occur for different reasons, namely pin region patching, long segment patching and violation patching.

a) Pin Region Patching: Pin region patching is the most effective patching among the three. It is needed due to the fact that pin accessibility is crucial for a net to be successfully routed, while it also involves too many details for global routing to handle. Therefore, the ideal way of improving pin accessibility is to identify those hard-to-access pins and assign more resources to them so that the detailed router can have the flexibility to find a way to access the pins. Our global router will check the upper (or lower) two layers of a pin, which are vital for accessing the pin. If the resource on either of them is below a specified threshold T, a larger guides will be patched. Take Figure 4.6a as an example, global cell p_0 contains a pin, so the resources of p_1 and p_2 will be checked. If either $r(p_1) < T$ or $r(p_2) < T$, three 3×3 patching guides will be added to all the three layers.

b) Long Segment Patching: A completely vacant long track is often hard to find, so a longer routing segment often means more wrong way wires and causing more congestion. We will thus identify the bottlenecks for long segments and add patches above or below to help with track switching, so as to reduce wrong ways. If a guide is longer than a specified length I, we'll consider long segment patching. Starting from one end of the segment and walking towards the other, if a global cell with resource below a threshold T is encountered, a single global cell route guide will be patched above or below it, depending on which of them has sufficient resource as shown in Figure 4.6b. Besides, after a global cell is patched we will skip next I global cells before we patch another one.



(a) *Pin Region Patching.*

(b) *Long Segment Patching.*

(c) *Violation Patching.*

Figure 4.6: *Different Kinds of Patching.*

c) Violation Patching: For global cell with inevitable violations, patching will be used again to enable the detailed router to search with more flexibility. Figure 4.6c shows an example of violation patching, in which the global cells on the two sides of the global cell with violation, along with the three global cells above are patched.

4.3 Experimental Results

We use C++ with the boost geometry library[9] to implement our global router, and we use Rsyn[30] to parse LEF/DEF formats. All our experiments are conducted on a 64-bit Linux workstation with Intel Xeon 3.4 GHz CPUs and 32 GB memory.

4.3.1 Results on ICCAD'19 Benchmarks

The benchmarks are all from ICCAD'19 global routing contest[29]. We global route all the benchmarks using 8 threads in accordance with the contest, and Dr. CU 2.0[69] is used to generate the detailed routing solution. Lastly, the final results are reported by Cadence Innovus 18.1[11].

Table 4.2 first shows the detailed decomposition of the overall detailed routing scores. Our global router produces solutions with similar wirelength and via score with the first

Table 4.2: *Scores Decompositions Compared with the First and Second Places in ICCAD'19 Global Routing Contest.*

Design	Wirelength & Via			Non-Preferred Usage			Short			Min-Area and Spacing		
	1st	2nd	ours	1st	2nd	ours	1st	2nd	ours	1st	2nd	ours
ispd18_test5	15614801	15728314	15613663	155898	457247	166994	318500	505340	330425	271000	381500	288500
ispd18_test8	37434586	37385956	37441058	261785	592927	269993	212445	333670	209470	143500	255500	144000
ispd18_test10	39037528	39291497	39061258	891219	1488538	882371	671755	1705070	669965	483000	948000	471000
ispd19_test7	77294941	78534037	77286072	1428490	1916508	1428396	9800300	13820375	9680620	6987000	9213000	6883000
ispd19_test8	119200121	120443016	119199593	1339678	1789897	1338449	7826500	9472060	7780220	6053000	6525000	6103000
ispd19_test9	184218174	186350610	184246497	2185321	2955375	2181774	14783160	18772785	14765850	10865500	11769000	10847000
ispd18_test5_metal5	15810136	15426511	15807997	126947	213345	135293	273220	270765	261150	232000	222500	224000
ispd18_test8_metal5	36743453	35912143	36746610	339927	575688	336768	210580	408540	194510	132000	184500	129500
ispd18_test10_metal5	40233544	40209785	40246090	1433956	2957377	1413120	4633110	71940390	4021620	705500	4058000	685500
ispd19_test7_metal5	70845164	70994659	70848996	1528759	2643684	1535876	9735870	18154405	9943260	6634000	8349500	6686000
ispd19_test8_metal5	116107932	117059329	116062781	1466713	2194550	1493314	8815065	11847405	8561400	6158500	6496500	6089000
ispd19_test9_metal5	179227079	180119959	179242111	2334544	3350841	2323850	16289715	21250500	16020280	11124500	12105500	10948000
Avg.	1.00	1.01	1.00	1.00	1.56	1.00	1.02	2.33	1.00	1.01	1.22	1.00

Table 4.3: *Runtime and Scores Compared with the First and Second Places in ICCAD'19 Global Routing Contest.*

Design	GR Runtime (s)			DR Score		
	1st	2nd	ours	1st	2nd	ours
ispd18_test5	83	56	68	16111082	16690901	16089196
ispd18_test8	260	229	236	37920522	38312552	37908815
ispd18_test10	349	510	334	40613594	42485106	40600501
ispd19_test7	564	1297	506	88453086	94332917	88577731
ispd19_test8	562	966	365	128356260	131762471	128412302
ispd19_test9	896	1207	528	201262621	208180772	201270655
ispd18_test5_metal5	80	403	85	16204442	15910624	16210303
ispd18_test8_metal5	240	621	300	37277889	36896370	37293962
ispd18_test10_metal5	350	21119	373	45680831	115107553	46300610
ispd19_test7_metal5	335	1780	377	82380132	91849749	82169293
ispd19_test8_metal5	499	32257	588	126172995	131164782	126429212
ispd19_test9_metal5	593	6755	658	197686238	204815803	197937335
Avg.	1.09	15.21	1.00	1.00	1.11	1.00

place router, but with $1-2\%$ fewer shorts and spacing violations. We then compare both our scores and runtime with the first and second places in the ICCAD'19 contest, which are shown in Table 4.3. Note that the average runtimes and scores are scaled by those of ours. Our router is around 9% faster than the first place and 15 times faster than the second place. The guides generated by our global router takes 5753 seconds on average for the detailed router to route, which is similar to that of the first place ($<$ 1% difference) and 9% slower than the second place. However, our router produces detailed routing result comparable to the first place and around 11% better than the second place. Besides, our algorithm's peak memory is close to the first place and is 1.83 times of that of the second place on average (ours is 8.22 GB on average and is 19.8 GB for the biggest design).

4.3.2 Effectiveness of Guide Patching

Among the three types of patching techniques, pin region patching is the most effective one. Taking away pin region patching from the current flow will result in 1.03% score increase on average. Long segment patching and violation patching are not as useful, but they still help to improve the detailed routability of some specific cases. Take ispd19_test4, an open benchmark not compared in Section 4.3.1, as an example, removing long segment or violation patching will result in 1.37% or 1.24% score increase respectively.

Chapter 5

DAG-Based Efficient Routing

Though CUGR 1.0 has achieved the state-of-the-art detailed routability, its algorithm is far from being efficient. The dynamic programming-based algorithm in CUGR 1.0 for 3D pattern routing induces $O(L^4|V|)$ runtime penalty, given L layers and $|V|$ vertices. It easily becomes too slow when the number of layers increases. Besides, once a net fails to be successfully routed by pattern routing, CUGR 1.0 will immediately invoke the maze router to fix the problem, which causes huge runtime overhead. The two-level maze routing technique can reduce the search space, and therefore reduce the runtime to some extent, but it also requires a net to be routed twice - first in a coarse-grained grid graph and then in a fine-grained grid graph. The problem is that the two levels of maze routing do not have a uniform cost scheme and both levels are time-consuming.

In this chapter, we will further develop some ideas proposed in CUGR 1.0 and introduce some additional techniques that are more efficient and parallelization friendly. We will propose our second generation global router - CUGR 2.0 that not only can achieve better detailed routability but also runs 7.8x faster using a single thread compared to CUGR 1.0 using 8 threads. Our major contributions are summarized as follows:

- We propose a directed acyclic graph (DAG) based generalized pattern routing algorithm. We will demonstrate that most commonly used routing patterns can be formulated by a routing DAG to realize pattern routing in a multi-layer routing space.

- We propose a new dynamic programming-based algorithm to calculate the routing cost on a given routing DAG. The proposed new vertex cost update algorithm can improve the complexity of 3D pattern routing from $O(L^4|V|)$ to $O(L^2|V|)$. The same algorithm can also be applied to the layer assignment for 2D routing to achieve the same improvement.

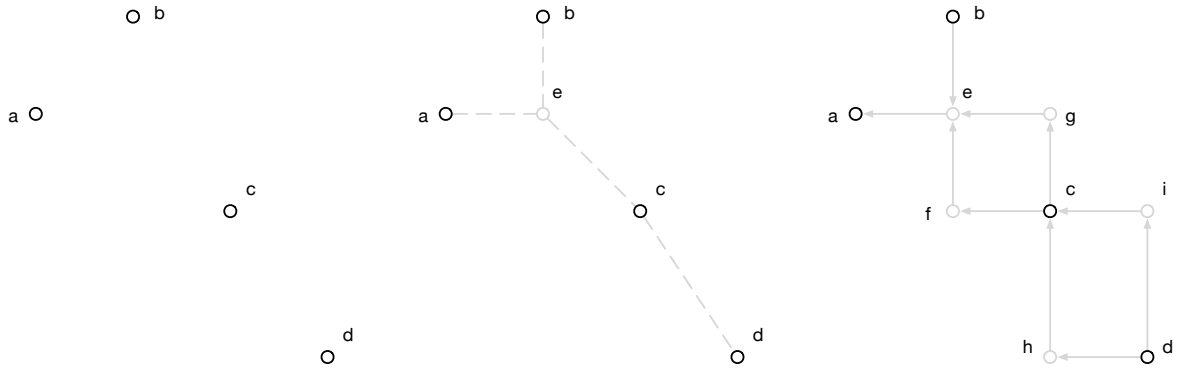
- We propose a DAG augmentation algorithm that enables the creation of alternative paths in a routing DAG. Not limited to simple edge shifting, the proposed DAG augmentation algorithm can even shift or create Steiner points. With this technique, over 99% nets can be successfully routed without the need of maze routing.

- We also propose a new sparse graph maze routing algorithm. The proposed method will sparsify the grid graph before routing and significantly reduce the runtime overhead without quality drop.

5.1 DAG-Based Generalized Pattern Routing

The general idea of our proposed DAG based generalized pattern routing is as follows. In order to route a multi-pin net, we will first break it into a set of two-pin nets by a rectilinear Steiner minimum tree (RSMT) algorithm. Each pin or Steiner point will become a vertex, and one of the vertices will be selected as the root. We will construct a DAG by using depth-first search to traverse the edges of the RSMT. For each edge (two-pin net), we will connect the later visited end point to the earlier visited end point using directed edges conforming to the patterns allowed. Then, we can compute the minimum routing cost of the vertices recursively using a dynamic programming-based algorithm. The dynamic programming-based algorithm not only can find the best 2D topology within the DAG, but also assign each edge to the best layer at the same time.

In the following sub-sections, we will first discuss how to construct routing DAGs to realize routing with commonly used patterns and formulate it as a dynamic programming problem. Next, we will explain our improved vertex cost update algorithm that can compute



(a) *Multi-pin Net.* **(b)** *Two-Pin Nets Obtained from RSMT.* **(c)** *Routing DAG (L-shape).*

Figure 5.1: *Process to Construct a Routing DAG.*

the cost of $|V|$ vertices for L layers in $O(L^2|V|)$ time, which is significantly faster compared to CUGR 1.0's $O(L^4|V|)$ implementation. Lastly, we will show that the routing DAG can actually be augmented easily to allow more flexible patterns in congested areas. By adding alternative paths to the DAG, the proposed algorithm can easily realize edge shifting or even Steiner point movement during routing to avoid congestion.

5.1.1 DAG Formulation for Pattern Routing

To begin with, we will first introduce how to construct the aforementioned routing DAG for a multi-pin net and formulate pattern routing as a dynamic programming problem. In our discussion, pattern routing always refers to 3D pattern routing that simultaneously determines the best 2D topology and layer assignment (of the wires). Given a multi-pin net, in order to construct the routing DAG, we will first construct an RSMT. Take the four-pin net in Figure 5.1a as an example, we will first construct an RSMT and break it into four two-pin nets as shown in Figure 5.1b. The grey vertex e in Figure 5.1b represents the Steiner point constructed in the process, and each dashed edge represents a two-pin net. Next, we will select a random vertex as the root and traverse the vertices using depth-first search (DFS). When the second end point of each two-pin net is visited, we will construct directed paths that conform to the available patterns to connect the end point to the first end point. In our

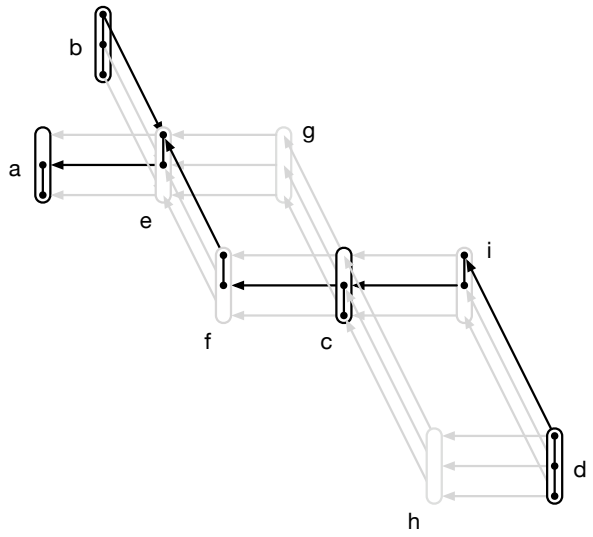
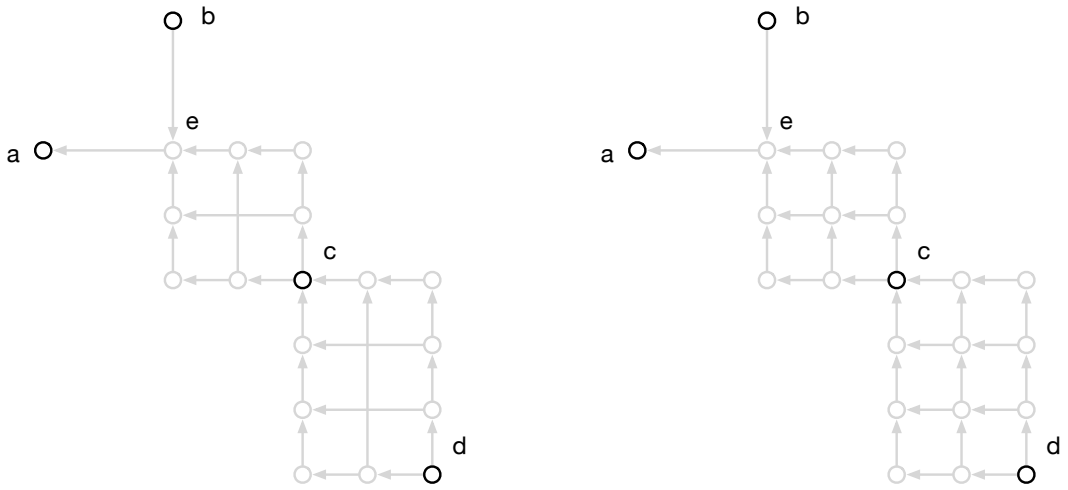


Figure 5.2: *Example Routing Solution.*

example, we select vertex a in Figure 5.1b as the root and use L-shapes as the only available patterns. We can eventually construct the routing DAG in Figure 5.1c. Two-pin net (e, a) and (b, e) are connected by a single directed edge, as it is the only way that conforms to L-shape. However, two-pin net (c, e) and (d, c) will each be connected by two directed paths as in Figure 5.1c, because the turning point of the L-shape can locate at either side of the two-pin net. Each of them consists of a new vertex and two directed edges. A routing solution on the DAG should connect all the pins to the root with each wire being assigned to a specific layer. Suppose the number of layers is three, one possible routing solution is shown in Figure 5.2. It is worth mentioning that vias are needed at each vertex to interconnect wires on different layers and pins if any. In our example, we assume that the pins are located on the lowest layer at vertex a, b, c and d respectively.

Different costs are associated with different wires and vias. The objective of DAG-based pattern routing is to find the solution with the minimum total routing cost. Besides, without modifying the routing algorithm, we can construct different routing DAGs to easily realize routing with many other commonly used patterns. For instance, Figure 5.3a shows the routing DAG for Z-shape routing and Figure 5.3b shows the routing DAG for monotonic routing, i.e., routing without detours. Notice that all the paths in the routing DAGs conform



(a) *Z-shape Routing.* **(b)** *Monotonic Routing.*

Figure 5.3: *Routing DAG for More Patterns.*

to the available patterns. In Section 5.1.3, we will also discuss how to purposefully modify the routing DAG to realize congestion aware routing with detours.

5.1.2 Dynamic Programming-based Vertex Cost Update Algorithm

Before diving into the routing algorithm, we will first define some useful terms and concepts. Suppose the DAG being constructed is $G(V, E)$ where V is the set of vertices and E is the set of directed edges. We say vertex u is a *preceding* vertex of v if $e(u \rightarrow v) \in E$. The set of all preceding vertices of v is $P(v)$. Vertex u is an *upstream* vertex of v if there exists a path from u to v in G. We will divide the set of preceding vertices of v into *streams* $S_1, ..., S_m$ such that the vertices in the same *stream* will have the same *upstream* pin vertices. Take the DAG in Figure 5.1c as an example, vertex e has three *preceding* vertices namely $P(e) = \{b, f, g\}$. Vertices in $P(e)$ can be divided into two *streams*, $S_1 = \{b\}$, $S_2 = \{f, g\}$. Vertex f and g are in the same stream because they have the same *upstream* pin vertices namely c and d. It should be obvious that we only need to connect one vertex in each *stream* so as to connect all pins. Next, we will define the minimum vertex cost $mvc(v, k)$ as the minimum routing cost to connect all *upstream* pins to vertex v on layer k. Suppose the number of layers is

L and the root vertex is r, the minimum routing cost for the whole net is calculated as $\min_{1 \leq k \leq L} mvc(r,k)$. Assume the wire cost needed to connect vertex u to vertex v on layer k is $wire(u \rightarrow v, k)$ and the via cost needed to connect the k^{th} and $(k+1)^{th}$ layer at vertex v is $via(v,k)$, the minimum vertex cost $mvc(v,k)$ $(k = 1, ..., L)$ can be updated recursively using Equation (5.1). The outer minimum expression selects a vertex u_i in each *stream* S_i while the inner minimum expression decides a wire layer k_i to connect the selected vertex u_i to vertex v. The equation assumes there is no pin located at vertex v, so only the via cost to connect the lowest and highest layers being used is needed.

$$mvc(v,k) = \min_{\substack{u_1 \in S_1 \\ \cdots \\ u_m \in S_m}} \min_{\substack{1 \leq k_1 \leq L \\ \cdots \\ 1 \leq k_m \leq L}} \sum_{1 \leq i \leq m} (mvc(u_i, k_i) + wire(u_i \rightarrow v, k_i)) + \sum_{\substack{\min(k_1,...,k_m,k) \leq k_v \\ < \max(k_1,...,k_m,k)}} via(v, k_v) \tag{5.1}$$

The most straightforward way of updating the minimum vertex costs is to enumerate all the decision variables. The layer assignment problem can be regarded as a special form of our DAG-based routing problem where each preceding vertex of a vertex forms a *stream* by itself. The dynamic programming-based layer assignment algorithm proposed by Lee et al. [67] adopts this brute-force method to update the minimum vertex costs. Since the maximum number of *preceding* vertices of any vertex is bounded by four (one from each direction), the number of streams is also bounded by $m \leq 4$. Enumerating the vertices in each *stream* will take constant time, however, enumerating the layer combinations for four *streams* will take $O(L^4)$ time. The routing cost for a set of decision variables can be calculated in constant time by using a pre-computed look-up table. Therefore, calculating $mvc(v,k)$ for $k = 1, ..., L$ will need $O(L^5)$ time. Updating the minimum vertex cost for all vertices will take $O(L^5|V|)$ time. CUGR 1.0 reduce the complexity to $O(L^4|V|)$ by selecting the root carefully so that each vertex will have at most three preceding vertices instead of four. In the following, we will discuss a new implementation that can further reduce the complexity to $O(L^2|V|)$.

The general idea is that, instead of enumerating the layers for each *stream*, we will

enumerate the lowest layer k_{min} and the highest layer k_{max} to use. The overall algorithm is summarized in Algorithm 3. To update the minimum vertex costs of vertex v, we should first recursively update the costs of all *preceding* vertices (line 2). Once the costs for all preceding vertices are ready, we will first enumerate the lowest layer k_{min}. For each k_{min}, we will first initialize the sum of via cost to zero and minimum stream cost to inf for all *streams* (line 4-6). Then, we will progressively increase the highest layer k_{max}. Each time k_{max} is increased, we will first add the via cost for connecting layer k_{max} if needed (line 8-9). Next, for each stream S_i, we will update the minimum stream cost for S_i if we can find a cheaper solution for the *stream* (line 10-14). We will update $mvc(v, k_{max})$ if we find a cheaper total cost (line 15-17). Lastly, after enumerating all k_{max}, we still need to propagate the minimum vertex cost to lower layers till layer k_{min} (line 18-20). This is because a solution found for a larger k_{max} is also a solution for a lower k_{max} for the same k_{min}.

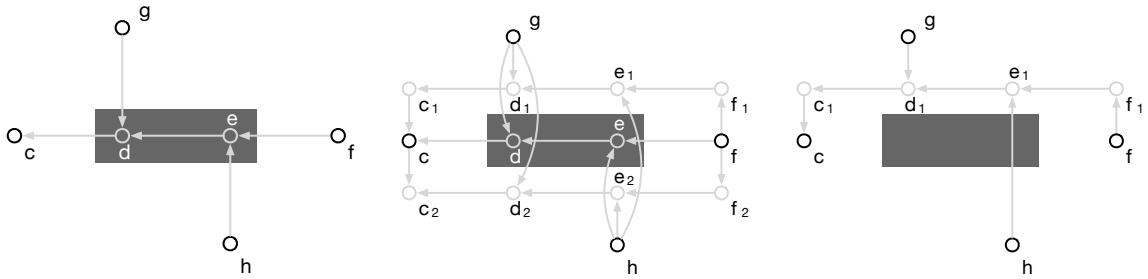
Algorithm 3 Minimum Vertex Cost Update

```
1:  function UPDATE(v)
2:      for u ∈ P(v) do UPDATE(u)
3:      for k_min = 1, ..., L do
4:          sum_via_cost ← 0
5:          for i = 1, ..., m do
6:              min_stream_cost(S_i) ← inf
7:          for k_max = k_min, ..., L do
8:              if k_max > k_min then
9:                  sum_via_cost ← sum_via_cost + via_cost(v, k_max − 1)
10:             for i = 1, ..., m do
11:                 for u ∈ S_i do
12:                     stream_cost ← mvc(u, k_max) + edge_cost(u → v, k_max)
13:                     if stream_cost < min_stream_cost(S_i) then
14:                         min_stream_cost(S_i) ← stream_cost
15:             cost ← sum_via_cost + Σ_{1≤i≤m} min_stream_cost(S_i)
16:             if cost < mvc(v, k_max) then
17:                 mvc(v, k_max) ← cost
18:         for k = L − 1, ..., k_min do
19:             if mvc(v, k + 1) < mvc(v, k) then
20:                 mvc(v, k) ← mvc(v, k + 1)
```

For complexity, since both the number of *streams* m and the size of each *stream* $|S_i|$ (for

(a) *Congested Edge.* **(b)** *Alternative Paths.*

Figure 5.4: *Alternative Paths for Edge Shifting.*



(a) *Steiner Points in Horizontally Congested Region.* **(b)** *Alternative Paths.* **(c)** *One Possible 2D Topology.*

Figure 5.5: *Alternative Paths for Steiner Point Movement.*

$i = 1, ..., m$), are bounded by four, the computation for each k_{max} can finish in $O(1)$ time. The computation for all possible values k_{min} and k_{max} should take $O(L^2)$ time. Therefore, we are able to solve the routing problem for all vertices in $O(L^2|V|)$ time.

5.1.3 DAG Augmentation for Congestion Aware Routing

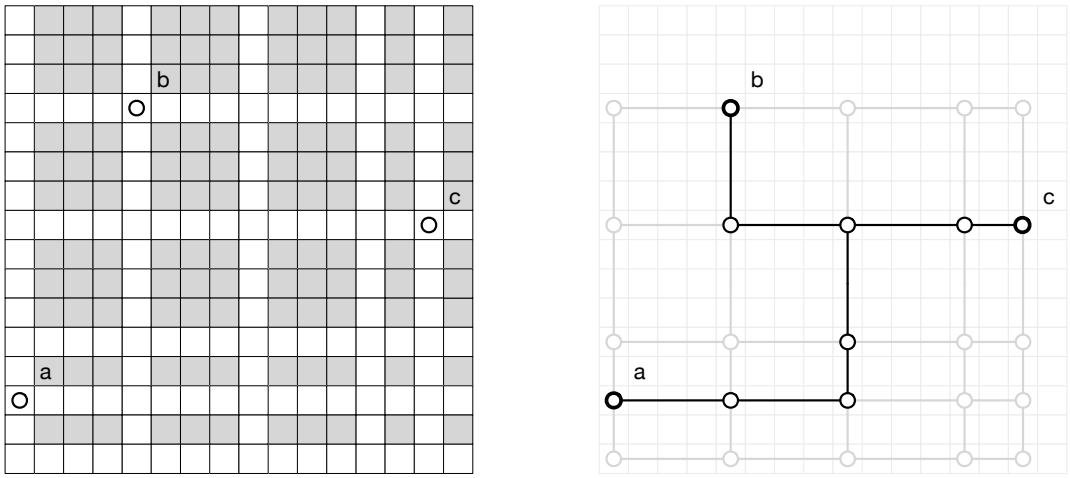
One of the many advantages of formulating the pattern routing problem using DAG is that the routing DAG can be easily augmented or modified with extremely high flexibility. For example, we can use monotonic routing for some hard-to-route two-pin nets and use basic L-shape for the rest. Or we can customize the directed paths for the areas with congestion while keeping the rest of the DAG unchanged. One example in shown in Figure 5.4a. Suppose edge $e(b \rightarrow a)$ is one edge of the DAG that crosses a congested region. Most traditional pattern routing can do little about this situation and will await maze routing to solve the problem. However, with DAG formulation, we can easily create alternative paths

for the edge on its two sides as shown in Figure 5.4b. A worse situation is that some Steiner points are themselves inside the congested region, making it almost impossible for pattern routing. Fortunately, with the flexibility of DAG formulation, we can also realize some capability of Steiner point movement. Take the situation in Figure 5.5a as an example, two Steiner points d and e in the DAG are located in a horizontally congested region. We can construct alternative paths for $path(f \rightarrow e \rightarrow d \rightarrow c)$ in a similar way with moved Steiner points, and re-build the connections between the moved Steiner points and other vertices as shown in Figure 5.5b. Given the augmented DAG, our dynamic programming-based routing algorithm will automatically find the best 2D topology and layer assignment within the DAG. Figure 5.5c shows one possible 2D topology for the DAG in Figure 5.5b.

The proposed pattern routing with DAG augmentation technique induces very little runtime penalty compared to ordinary pattern routing. Yet, it can resolve moderate to medium congestion much faster than maze routing and produce even shorter wirelength for many nets. The increase of wirelength can be precisely controlled.

5.2 CUGR 2.0 - Global Routing with DAG Formulation

In this section, we will briefly introduce our new global router CUGR 2.0 based on the techniques discussed in Section 5.1. CUGR 2.0 is mainly comprised of three phases - DAG-based pattern routing, DAG-based pattern routing with augmentation and sparse graph maze routing. Nets failed to be routed in a phase will be ripped-up and re-routed in the following phase. For the first phase, we will perform DAG-based (3D) pattern routing using L-shapes only. The effect of our DAG-based pattern routing is similar to the initial routing in CUGR 1.0, but it is much faster. For the second phase, we will perform DAG-based pattern routing with augmentation. We will re-use the DAG constructed in the first phase and augment the DAG to make it more resilient to congestion. In reality, we will identify congested paths in the DAG and create multiple alternative paths for each of them. We will limit the length of an alternative path to at most $A\%$ longer than the original path. Our dynamic programming-based routing algorithm are then invoked to re-route the nets and



(a) *Global Cell Grid.* **(b)** *Sparse Grid Graph.*

Figure 5.6: *Maze Routing on Sparse Grid Graph.*

find the best 2D topology and layer assignment simultaneously.

Though the first two-phases are enough for most nets, we still provide a maze routing phase to handle the really-hard-to-route nets. In order to search the whole design efficiently, we propose sparse graph maze routing to trade-off between searching completeness and search space size. More specifically, we will construct a sparse grid graph using part of the rows and columns of the global cell grid. The set of rows being used, for example, will be $\{r \mid 1 \leq r \leq R, r \mod X = x \text{ or row } r \text{ contains pins}\}$, where R is the number of rows and X is a parameter to control the sparsity of the grid graph, and x is an offset variable. The columns are also selected in the same way. We will update x by $x \leftarrow (x+1) \mod X$ after each net is routed to make sure the global cell grid will be evenly utilized. For instance, Figure 5.6a shows a three-pin net in a global cell grid. Suppose $X = 4$ and $x = 1$, the white rows and columns in Figure 5.6a will be used to construct the sparse grid graph shown in Figure 5.6b. The sparsification can reduce the size of the grid graph (the number of vertices) by almost X^2 times, and greatly reduce the time needed for maze routing. Note that though CUGR 1.0 also constructs a compressed grid graph, the philosophies of the two approaches are quite different. In CUGR 1.0, every $X \times X$ global cells are compressed into one coarse-grained cell by aggregating their resources. After maze routing in the compressed

grid graph, fine-grained maze routing is still needed to map the routing solution to the actual global cell grid. However, in CUGR 2.0, we will simply not consider the resources in cells outside the selected rows and columns. Each vertex in the sparse grid graph represents an actual global cell. Therefore, not only the time for compression is saved, the fine-grained maze routing is also no longer needed. Experimental results show that our proposed sparse graph maze routing runs much faster with comparable routing quality.

5.3 Experimental Results

We implement CUGR 2.0 using C++. All the experiments are conducted on a 64-bit Linux workstation with Intel Xeon Silver 2.20 GHz CPUs and 256 GB memory. The benchmarks we use to test the algorithms are from ICCAD'19 global routing contest [29]. Dr. CU 2.0 [69] are used to generate the detailed routed solutions, which is consistent with the contest settings. Cadence Innovus 18.1 [11] is used to evaluate the detailed routed solution and report the final scores.

5.3.1 Comparison with Other Global Routers

We first compare CUGR 2.0 with other two recently proposed global routers, namely CUGR 1.0 [75] and SPRoute 2.0 [40]. We use Dr. CU 2.0 to generate detailed routed solution rather than evaluate the global routed solution directly. We list the final score reported by Innovus and the total running time of different global routers in Table 5.1. For scores, smaller values indicate better solutions. Experimental results show that the solution quality of CUGR 2.0 is the best among all three global routers and it achieves over 1.4% improvement compared with the other two. For running time, CUGR 2.0 is around 683% faster than CUGR 1.0 and around 26% faster than SPRoute 2.0. It is worth mentioning that the other two routers use 8 threads by default, however, we use only one thread for CUGR 2.0 to demonstrate its superior efficiency. Such a significant speed-up is possible using a single thread for many reasons. Firstly, we reformulated 3D pattern routing and reduced its complexity. Moreover, by DAG augmentation, our router can resolve a lot more overflows efficiently before maze

Table 5.1: *Score and Running Time Comparison with Other Global Routers*

Benchmarks	Detailed Routed Score			Total Running Time (s)		
	CUGR 1.0	SPRoute 2.0	CUGR 2.0	CUGR 1.0 (8-threaded)	SPRoute 2.0 (8-threaded)	CUGR 2.0 (single-threaded)
ispd18_test5	16111387	16266324	16168206	74.112	10.487	9.756
ispd18_test8	37923375	38303840	37552269	263.774	28.045	25.581
ispd18_test10	40567853	42865920	40184353	359.057	35.680	29.367
ispd19_test7	88569644	87406430	85936253	582.001	49.963	55.874
ispd19_test8	128252583	127723613	127085092	403.778	70.100	66.381
ispd19_test9	201133173	199900738	198680565	595.969	108.092	103.188
ispd18_test5_metal5	16197187	16101826	15814649	91.902	11.859	10.919
ispd18_test8_metal5	37304768	39604236	36864567	308.500	48.342	30.293
ispd18_test10_metal5	45618888	47267127	46554472	388.044	106.169	38.488
ispd19_test7_metal5	82447080	81104527	79727008	428.850	55.105	57.633
ispd19_test8_metal5	126412903	126389940	125038407	652.924	130.786	78.376
ispd19_test9_metal5	197713563	197033254	194838869	737.866	133.064	118.385
Average	84854367	84997315	83703726	407.231	65.641	52.020
Ratio	1.014	1.015	1.000	7.828	1.262	1.000

routing. Lastly, our proposed sparse graph maze routing algorithm greatly reduced the search space and shrunk the time needed for rip-up and reroute.

The detailed decomposition of the score is listed in Table 5.2. It shows that CUGR 2.0's solution uses the least amount of wires and vias and has significantly lower amount of non-preferred usage, 36% and 10% lower than CUGR 1.0 and SPRoute respectively. When it comes to design rule violations, CUGR 1.0 has the worst performance and causes around 12% more violations. SPRoute 2.0 does much better but still causes 1% extra violations compared to CUGR 2.0.

5.3.2 Effectiveness and Efficiency of Different Routing Phases

To demonstrate the effectiveness of all three phases of our global routing algorithm, we show the number of nets being routed and the time spent in each phase in Table 5.3. The last row of Table 5.3 shows the average percentage of nets being routed against the whole netlist and the average percentage of time used in each phase. Basically, phase 1 or pattern routing routes all the nets using 47.33% of the time on average. The first phase is the fastest

Table 5.2: *Detailed Routed Score Decomposition*

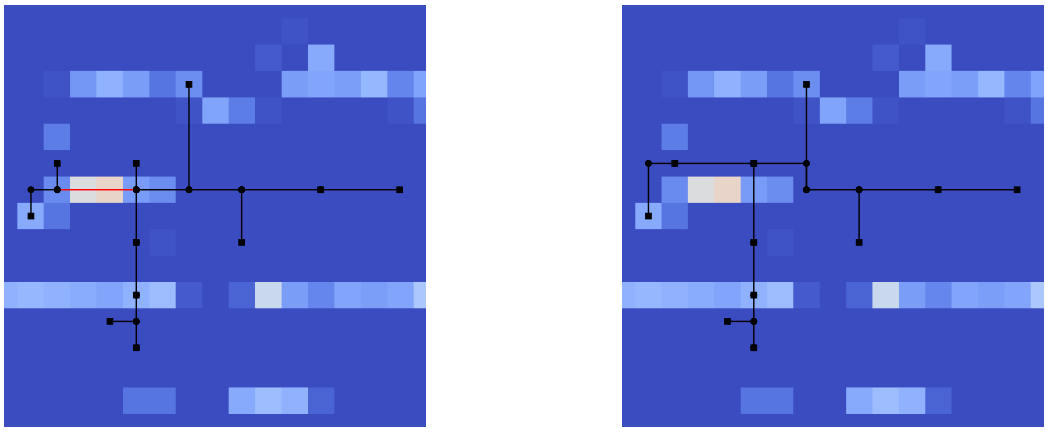
Benchmarks	Wires			Vias			Non-preferred Usage			Design Rule Violations		
	CUGR 1.0	SPRoute 2.0	CUGR 2.0	CUGR 1.0	SPRoute 2.0	CUGR 2.0	CUGR 1.0	SPRoute 2.0	CUGR 2.0	CUGR 1.0	SPRoute 2.0	CUGR 2.0
ispd18_test5	13763143	13802030	13699907	1854438	1876508	1849918	152307	251915	219273	341499	335871	399109
ispd18_test8	32745567	32737048	32422365	4692606	4726586	4648996	269892	607792	261796	215310	232414	219113
ispd18_test10	34053084	36133244	33826371	4993464	5077966	4988012	877900	1103160	767729	643405	551550	602240
ispd19_test7	61002588	60697883	60663314	16278664	16374484	16202912	1422735	753028	875068	9865657	9581035	8194960
ispd19_test8	93406342	93453444	93172400	25800680	25487900	25315884	1339314	984397	970780	7706247	7797872	7626027
ispd19_test9	141244656	141412123	140895973	42978536	42428964	42148080	2177444	1637129	1587265	14732538	14422523	14049248
ispd18_test5_metal5	13978880	13663435	13636161	1831332	1899368	1828950	129120	250649	96188	257856	288374	253350
ispd18_test8_metal5	32237984	33872837	31767069	4491928	4774254	4568180	337558	722922	299218	237297	234222	230100
ispd18_test10_metal5	35375467	40504890	38167335	4864564	5276492	5034728	1422217	1144235	1250505	3956640	341510	2101905
ispd19_test7_metal5	54620903	55321488	54368026	16237016	16581648	16239620	1533612	793259	868979	10055550	8408132	8250383
ispd19_test8_metal5	90510087	91348525	90641938	25557372	25817712	25293780	1489444	1059280	1039826	8856000	8164423	8062862
ispd19_test9_metal5	136661914	136735909	135702014	42559400	42851540	42030432	2326264	1738312	1707903	16165985	15707492	15398520
Average	61633385	62473571	61580239	16011667	16097785	15845791	1123150	920506	828711	6086165	5505452	5448985
Ratio	1.001	1.015	1.000	1.010	1.016	1.000	1.355	1.111	1.000	1.117	1.010	1.000

Table 5.3: *Number of Nets Routed and Time Spent in Each Phase*

Benchmarks	Phase 1		Phase 2		Phase 3	
	# Nets	Time (s)	# Nets	Time (s)	# Nets	Time (s)
ispd18_test5	72394	3.273	583	1.392	0	0.000
ispd18_test8	179863	8.083	1348	4.151	23	0.597
ispd18_test10	182000	8.269	3633	7.069	2054	2.143
ispd19_test7	358720	16.154	7330	16.592	38	1.237
ispd19_test8	537577	26.043	1597	6.496	19	1.633
ispd19_test9	895252	40.297	2751	8.095	14	2.613
ispd18_test5_metal5	72394	2.182	5058	3.305	624	0.975
ispd18_test8_metal5	179863	5.347	13864	8.371	2579	5.020
ispd18_test10_metal5	182000	5.818	14796	10.354	7585	10.521
ispd19_test7_metal5	358720	10.120	16974	11.864	5494	15.671
ispd19_test8_metal5	537577	16.711	15654	13.104	3466	18.901
ispd19_test9_metal5	895252	26.167	20449	17.237	3478	22.910
Average Percentage	100.00%	47.33%	3.25%	33.78%	0.85%	18.89%

phase and can route the majority of the nets very efficiently without overflow. Phase 2 or pattern routing with DAG augmentation re-routes 3.25% of the nets using 33.78% of the time on average. The second phase can resolve most congestion while still being more efficient than maze routing. The DAG augmentation technique in the second phase not only resolves congestion faster but also finds a relatively good 2D topology with bounded wirelength increase. Figure 5.7 shows the 2D topology of a net before and after DAG augmentation. It demonstrates how DAG augmentation "smartly" moves some edges and Steiner points to resolve congestion while maintaining a relatively good topology. The last phase is responsible for the rest 0.85% really-hard-to-route nets and uses 18.89% of the time.

Figure 5.8 shows the resource heatmaps of ispd18_test10 on metal3 after each of the three phases. It is obvious to see that after phase 2, most congestion can be resolved. After phase 3, the routing wires in the hot spot are further spread and more evenly distributed.



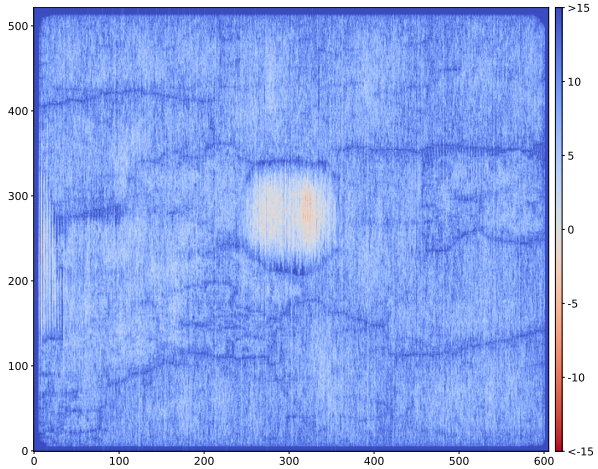
(a) *Before* **(b)** *After*

Figure 5.7: *Topology of a Net (net349500 in ispd19_test9_metal5) Before and After DAG Augmentation.*

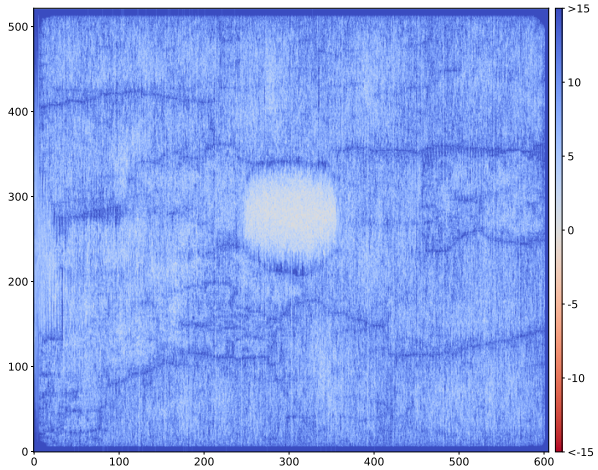
5.3.3 Comparison with GPU-Accelerated Routing Algorithms

To further demonstrate the efficiency of our proposed routing algorithms, we also compare the runtime of our algorithms with GPU-accelerated routing algorithms. Modern GPUs can run tens of thousands of threads in parallel, while CPUs are usually only capable of running tens or hundreds of threads at the same time. Therefore, they have the potential to achieve tremendous speed-up compared to algorithms running on CPUs. We compare our algorithm with two GPU-accelerated routing algorithms. FastGR [76] accelerated the 3D pattern routing algorithm of CUGR 1.0 by both net-level and path-level parallelization on GPU. GAMER [73], on the other hand, accelerated the coarse-level maze routing of CUGR 1.0 by updating vertical and horizontal routing costs alternatively.

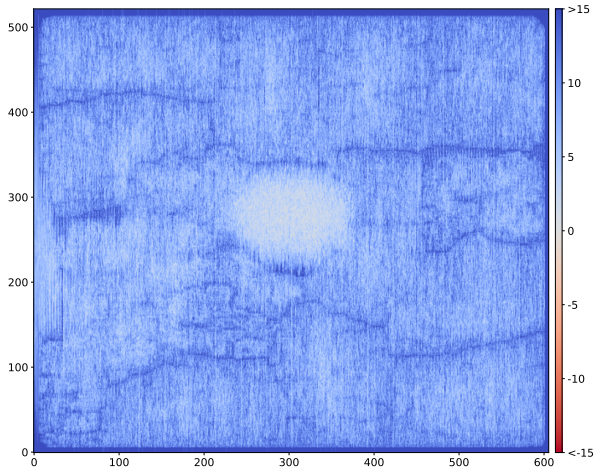
We show the runtime of initial routing of CUGR 1.0, FastGR and CUGR 2.0 in Table 5.4. All three algorithms perform 3D pattern routing for initial routing. It shows that, FastGR is over 4× as fast as CUGR 1.0, and CUGR 2.0 runing with a single thread is even faster. We show the runtime of rip-up and reroute of CUGR 1.0, GAMER and CUGR 2.0 in Table 5.5. For rip-up and reroute, CUGR 1.0 performs multiple iterations of two-level maze routing. GAMER accelerated the coarse-level of the two-level maze routing by GPU parallelization. CUGR 2.0 performs one iteration of pattern routing with DAG augmentation (phase 2) and



(a) *Metal3 Resource Heatmap after Phase 1.*



(b) *Metal3 Resource Heatmap after Phase 2.*



(c) *Metal3 Resource Heatmap after Phase 3.*

Figure 5.8: *Metal3 Resource Heatmap after Each Phase for ispd_test10.*

Table 5.4: *Initial Routing Runtime (s) Comparison*

Benchmarks	CUGR 1.0 (8-threaded)	FastGR (with GPU)	CUGR 2.0 (single-threaded)
ispd18_test5	29.334	6.683	3.273
ispd18_test8	53.537	12.076	8.083
ispd18_test10	60.142	12.705	8.269
ispd19_test7	88.112	19.872	16.154
ispd19_test8	119.647	23.278	26.043
ispd19_test9	173.454	32.294	40.297
ispd18_test5_metal5	8.394	4.099	2.182
ispd18_test8_metal5	17.080	6.989	5.347
ispd18_test10_metal5	18.415	8.590	5.818
ispd19_test7_metal5	27.102	10.876	10.120
ispd19_test8_metal5	41.244	15.158	16.711
ispd19_test9_metal5	62.837	22.331	26.167
Average	58.275	14.579	14.039
Ratio	4.151	1.039	1.000

one iteration of sparse graph maze routing (phase 3). The experimental results show that GAMER is over 3.5× as fast as CUGR 1.0. On the other hand, CUGR 2.0 is over 5× faster than GAMER and almost 19× as fast as CUGR 1.0.

Table 5.5: *Rip-Up and Reroute Runtime (s) Comparison*

Benchmarks	CUGR 1.0 (8-threaded)	GAMER (with GPU)	CUGR 2.0 (single-threaded)
ispd18_test5	34.887	7.968	1.392
ispd18_test8	189.048	30.851	4.748
ispd18_test10	278.780	41.213	9.212
ispd19_test7	444.162	41.021	17.829
ispd19_test8	208.768	47.008	8.129
ispd19_test9	304.816	50.755	10.708
ispd18_test5_metal5	74.699	40.298	4.280
ispd18_test8_metal5	271.836	120.343	13.391
ispd18_test10_metal5	350.603	154.515	20.875
ispd19_test7_metal5	355.763	78.871	27.535
ispd19_test8_metal5	535.467	171.955	32.005
ispd19_test9_metal5	555.640	197.294	40.147
Average	300.372	81.841	15.854
Ratio	18.946	5.162	1.000

Conclusion

In this thesis, we propose a set of routing algorithms for the purpose of generating high-quality design and finishing routing with high efficiency.

Rectilinear Steiner minimum tree (RSMT) is the backbone of many routing algorithms. In this thesis, we propose a neural network framework called REST to learn to construct RSMT via reinforcement learning. In order to make use of the sequence generation models for RSMT generation, we propose the rectilinear edge sequence (RES) for RSMT representation. The RSMT represented by an RES can be evaluated in linear time, and it is guaranteed that there always exists an RES representing an optimal RSMT. These characteristics make RES ideal for reinforcement learning. Besides, we also propose an actor-critic model for RES generation and train it with REINFORCE algorithm. After training, compared to traditional heuristics that take long time to develop, REST can produce RSMT of higher quality for small to median nets and is even faster.

In addition, we propose a detailed-routability-driven global router, CUGR. We propose several techniques to improve detailed-routability. Firstly, we propose 3D pattern routing to find the best 2D topology and layer assignment simultaneously. Secondly, we propose two-level maze routing to explore the routing space thoroughly and efficiently. Lastly, we propose a guide patching technique to provide detailed routers with extra flexibility in handling issues like pin access and overflow. We evaluate CUGR by an actual detailed router rather than counting number of overflows. Experimental results show that the routing solutions produced by CUGR lead to much fewer shorts and design rule violations compared to other global routers.

Last but not least, we propose a directed acyclic graph (DAG)-based efficient routing algorithm. By constructing a routing DAG, our proposed algorithm is able to find the best 3D topology to connect all the pins efficiently by dynamic programming. Various routing algorithms, e.g., L-shape routing, Z-shape routing, and monotonic routing, can be performed by constructing the corresponding routing DAG without re-implementation of the algorithm. We also propose a DAG augmentation technique that allows edge shifting or Steiner point movement during routing. We demonstrate the effectiveness and efficiency of our DAG-based routing algorithm in CUGR 2.0. Apart from DAG-based routing, CUGR 2.0 also includes a sparse graph maze routing algorithm that pre-assigns some routing resources and performs maze routing in an extremely sparse grid graph for each net. Experimental results show that CUGR 2.0 is much faster than its predecessor and is able to generate designs of even higher quality after detailed routing.

As routing becomes ever more important and challenging than before in VLSI design, the works presented in this thesis explore new directions and methodologies for quality and efficient routing, including routing tree construction by RL, detailed routability-driven global routing, efficient exploration of routing space by DAG-based routing, etc.

References

[1] Nobel Media AB 2014. *The History of the Integrated Circuit*. https://web.archive.org/web/20180629102838/https://www.nobelprize.org/educational/physics/integrated_circuit/history/.

[2] Gaurav Ajwani, Chris Chu, and Wai-Kei Mak. "FOARS: FLUTE based obstacle-avoiding rectilinear Steiner tree construction". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.2 (2011), pp. 194–204.

[3] Christoph Albrecht. "Global routing by new approximation algorithms for multicommodity flow". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.5 (2001), pp. 622–632.

[4] Charles J Alpert, Wing-Kai Chow, Kwangsoo Han, Andrew B Kahng, Zhuo Li, Derong Liu, and Sriram Venkatesh. "Prim-Dijkstra revisited: Achieving superior timing-driven routing trees". In: *Proceedings of the 2018 International Symposium on Physical Design*. 2018, pp. 10–17.

[5] Charles J Alpert, Te C Hu, Jen-Hsin Huang, Andrew B Kahng, and David Karger. "Prim-Dijkstra tradeoffs for improved performance-driven routing tree design". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 14.7 (1995), pp. 890–896.

[6] *An R-MST implementation*. https://github.com/shininglion/rectilinear_spanning_graph.

[7] Shabbir Batterywala, Narendra Shenoy, William Nicholls, and Hai Zhou. "Track assignment: A desirable intermediate step between global routing and detailed routing". In: *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*. 2002, pp. 59–66.

[8] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. "Neural combinatorial optimization with reinforcement learning". In: *arXiv preprint arXiv:1611.09940* (2016).

[9] *Boost geometry library*. https://www.boost.org/doc/.

[10] Manjit Borah, Robert Michael Owens, and Mary Jane Irwin. "An edge-based heuristic for Steiner routing". In: *IEEE transactions on computer-aided design of integrated circuits and systems* 13.12 (1994), pp. 1563–1568.

[11] *Cadence Innovus Implementation System*. https://www.cadence.com.

[12] Zhen Cao, Tong Jing, Jinjun Xiong, Yu Hu, Lei He, and Xianlong Hong. "DpRouter: A fast and accurate dynamic-pattern-based global routing algorithm". In: *2007 Asia and South Pacific Design Automation Conference*. IEEE. 2007, pp. 256–261.

[13] Fong-Yuan Chang, Ren-Song Tsay, Wai-Kei Mak, and Sheng-Hsiung Chen. "MANA: A shortest path maze algorithm under separation and minimum length nanometer rules". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.10 (2013), pp. 1557–1568.

[14] Ting-Hai Chao, Yu-Chin Hsu, Jan-Ming Ho, and AB Kahng. "Zero skew clock routing with minimum wirelength". In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 39.11 (1992), pp. 799–814.

[15] Gengjie Chen, Chak-Wa Pui, Haocheng Li, and Evangeline FY Young. "Dr. CU: Detailed Routing by Sparse Grid Graph and Minimum-Area-Captured Path Search". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019), pp. 1–1. DOI: 10.1109/TCAD.2019.2927542.

[16] Gengjie Chen, Chak-Wa Pui, Haocheng Li, and Evangeline FY Young. "Dr. cu: Detailed routing by sparse grid graph and minimum-area-captured path search". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.9 (2019), pp. 1902–1915.

[17] Gengjie Chen, Peishan Tu, and Evangeline FY Young. "SALT: Provably good routing topology by a novel steiner shallow-light tree algorithm". In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2017, pp. 569–576.

[18] Gengjie Chen and Evangeline FY Young. "Salt: provably good routing topology by a novel steiner shallow-light tree algorithm". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.6 (2019), pp. 1217–1230.

[19] Jingsong Chen, Jian Kuang, Guowei Zhao, Dennis J-H Huang, and Evangeline FY Young. "PROS: A plug-in for routability optimization applied in the state-of-the-art commercial EDA tool using deep learning". In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE. 2020, pp. 1–8.

[20] Minsik Cho, Katrina Lu, Kun Yuan, and David Z Pan. "BoxRouter 2.0: A hybrid and robust global router with layer assignment for routability". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 14.2 (2009), pp. 1–21.

[21] Minsik Cho, Katrina Lu, Kun Yuan, and David Z Pan. "BoxRouter 2.0: Architecture and implementation of a hybrid and robust global router". In: *2007 IEEE/ACM International Conference on Computer-Aided Design*. IEEE. 2007, pp. 503–508.

[22] Minsik Cho and David Z Pan. "BoxRouter: A new global router based on box expansion and progressive ILP". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.12 (2007), pp. 2130–2143.

[23] Chris Chu and Yiu-Chung Wong. "FLUTE: Fast lookup table based rectilinear steiner minimal tree algorithm for VLSI design". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.1 (2007), pp. 70–83.

[24] Jason Cong, Lei He, Cheng-Kok Koh, and Patrick H Madden. "Performance optimization of VLSI interconnect layout". In: *Integration* 21.1-2 (1996), pp. 1–94.

[25] Jason Cong, Andrew B Kahng, Cheng-Kok Koh, and C-W Albert Tsao. "Bounded-skew clock and Steiner routing". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 3.3 (1998), pp. 341–388.

[26] Hanjun Dai, Bo Dai, and Le Song. "Discriminative embeddings of latent variable models for structured data". In: *International conference on machine learning*. PMLR. 2016, pp. 2702–2711.

[27] Ke-Ren Dai, Wen-Hao Liu, and Yih-Lang Li. "NCTU-GR: Efficient simulated evolution-based rerouting and congestion-relaxed layer assignment on 3-D global routing". In: *IEEE Transactions on very large scale integration (VLSI) systems* 20.3 (2011), pp. 459–472.

[28] Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. "Learning heuristics for the tsp by policy gradient". In: *International conference on the integration of constraint programming, artificial intelligence, and operations research*. Springer. 2018, pp. 170–181.

[29] Sergei Dolgov, Alexander Volkov, Lutong Wang, and Bangqi Xu. "2019 cad contest: Lef/def based global routing". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2019, pp. 1–4.

[30] Guilherme Flach, Mateus Fogaça, Jucemar Monteiro, Marcelo Johann, and Ricardo Reis. "Rsyn: An extensible physical synthesis framework". In: *Proceedings of the 2017 ACM on International Symposium on Physical Design*. ACM. 2017, pp. 33–40.

[31] Jhih-Rong Gao, Pei-Ci Wu, and Ting-Chi Wang. "A new global router for modern designs". In: *2008 Asia and South Pacific Design Automation Conference*. IEEE. 2008, pp. 232–237.

[32] Michael R Garey and David S. Johnson. "The rectilinear Steiner tree problem is NP-complete". In: *SIAM Journal on Applied Mathematics* 32.4 (1977), pp. 826–834.

[33] Michael Gester, Dirk Müller, Tim Nieberg, Christian Panten, Christian Schulte, and Jens Vygen. "BonnRoute: Algorithms and data structures for fast and good VLSI routing". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 18.2 (2013), pp. 1–24.

[34] Stèphano MM Gonçalves, Leomar S da Rosa Jr, and Felipe S Marques. "SmartDR: Algorithms and techniques for fast detailed routing with good design rule handling". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 26.2 (2020), pp. 1–38.

[35] Jeff Griffith, Gabriel Robins, Jeffrey S Salowe, and Tongtong Zhang. "Closing the gap: Near-optimal Steiner trees in polynomial time". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.11 (1994), pp. 1351–1365.

[36] Winston Haaswijk, Edo Collins, Benoit Seguin, Mathias Soeken, Frédéric Kaplan, Sabine Süsstrunk, and Giovanni De Micheli. "Deep learning for logic optimization algorithms". In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2018, pp. 1–4.

[37] Yiding Han, Dean Michael Ancajas, Koushik Chakraborty, and Sanghamitra Roy. "Exploring high-throughput computing paradigm for global routing". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.1 (2013), pp. 155–167.

[38] Yiding Han, Koushik Chakraborty, and Sanghamitra Roy. "A global router on GPU architecture". In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE. 2013, pp. 78–84.

[39] Maurice Hanan. "On Steiner's problem with rectilinear distance". In: *SIAM Journal on Applied mathematics* 14.2 (1966), pp. 255–265.

[40] Jiayuan He, Udit Agarwal, Yihang Yang, Rajit Manohar, and Keshav Pingali. "SPRoute 2.0: A detailed-routability-driven deterministic parallel global router with soft capacity". In: *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2022, pp. 586–591.

[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[42] Xu He, Tao Huang, Wing-Kai Chow, Jian Kuang, Ka-Chun Lam, Wenzan Cai, and Evangeline FY Young. "Ripple 2.0: High quality routability-driven placement via global router integration". In: *Proceedings of the 50th Annual Design Automation Conference*. 2013, pp. 1–6.

[43] Stephan Held and Sophie Theresa Spirkl. "A fast algorithm for rectilinear Steiner trees with length restrictions on obstacles". In: *Proceedings of the 2014 on International symposium on physical design*. 2014, pp. 37–44.

[44] J-M Ho, Gopalakrishnan Vijayan, and Chak-Kuen Wong. "New algorithms for the rectilinear Steiner tree problem". In: *IEEE transactions on computer-aided design of integrated circuits and systems* 9.2 (1990), pp. 185–193.

[45] Jin Hu, Jarrod A Roy, and Igor L Markov. "Sidewinder: a scalable ILP-based router". In: *Proceedings of the 2008 international workshop on System level interconnect prediction*. 2008, pp. 73–80.

[46] Guyue Huang, Jingbo Hu, Yifan He, Jialong Liu, Mingyuan Ma, Zhaoyang Shen, Juejian Wu, Yuanfan Xu, Hengrui Zhang, Kai Zhong, et al. "Machine learning for electronic design automation: A survey". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 26.5 (2021), pp. 1–46.

[47] Tao Huang and Evangeline FY Young. "Construction of rectilinear Steiner minimum trees with slew constraints over obstacles". In: *Proceedings of the International Conference on Computer-Aided Design*. 2012, pp. 144–151.

[48] Frank K Hwang. "An O (n log n) algorithm for rectilinear minimal spanning trees". In: *Journal of the ACM (JACM)* 26.2 (1979), pp. 177–182.

[49] Frank K Hwang. "On Steiner minimal trees with rectilinear distance". In: *SIAM journal on Applied Mathematics* 30.1 (1976), pp. 104–114.

[50] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[51] *ISPD 2019 Detailed Routing Contest*. http://www.ispd.cc/contests/19/. Accessed: 2019-10-31.

[52] Andrew B Kahng, Jens Lienig, Igor L Markov, and Jin Hu. *VLSI physical design: from graph partitioning to timing closure*. Springer Nature, 2022.

[53] Andrew B Kahng, Ion I Măndoiu, and Alexander Z Zelikovsky. “Highly scalable algorithms for rectilinear and octilinear Steiner trees”. In: *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*. 2003, pp. 827–833.

[54] Andrew B Kahng and Gabriel Robins. “A new class of iterative Steiner tree heuristics with good performance”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11.7 (1992), pp. 893–902.

[55] Andrew B Kahng, Lutong Wang, and Bangqi Xu. “The tao of PAO: Anatomy of a pin access oracle for detailed routing”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2020, pp. 1–6.

[56] Andrew B Kahng, Lutong Wang, and Bangqi Xu. “TritonRoute-WXL: The Open-Source Router With Integrated DRC Engine”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.4 (2021), pp. 1076–1089.

[57] Andrew B Kahng, Lutong Wang, and Bangqi Xu. “Tritonroute: An initial detailed router for advanced vlsi technologies”. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.

[58] Andrew B Kahng, Lutong Wang, and Bangqi Xu. “Tritonroute: The open-source detailed router”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.3 (2020), pp. 547–559.

[59] Ryan Kastner, Elaheh Bozorgzadeh, and Majid Sarrafzadeh. "Pattern routing: Use and theory for increasing predictability and avoiding coupling". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21.7 (2002), pp. 777–790.

[60] Jamil Kawa, Charles Chiang, and Raul Camposano. "EDA challenges in nano-scale technology". In: *IEEE Custom Integrated Circuits Conference 2006*. IEEE. 2006, pp. 845–851.

[61] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. "Learning combinatorial optimization algorithms over graphs". In: *Advances in neural information processing systems* 30 (2017).

[62] Myung-Chul Kim, Jin Hu, Dong-Jin Lee, and Igor L Markov. "A SimPLR method for routability-driven placement". In: *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2011, pp. 67–73.

[63] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[64] Wouter Kool, Herke Van Hoof, and Max Welling. "Attention, learn to solve routing problems!" In: *arXiv preprint arXiv:1803.08475* (2018).

[65] Mark R Kramer. "The complexity of wirerouting and finding minimum area layouts for arbitrary VLSI circuits". In: *Advances in computing research* 2 (1984), pp. 129–146.

[66] Chin Yang Lee. "An algorithm for path connections and its applications". In: *IRE transactions on electronic computers* 3 (1961), pp. 346–365.

[67] Tsung-Hsien Lee and Ting-Chi Wang. "Congestion-constrained layer assignment for via minimization in global routing". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.9 (2008), pp. 1643–1656.

[68] *LEF/DEF Language Reference*. https://www.ispd.cc/contests/18/lefdefref.pdf.

[69] Haocheng Li, Gengjie Chen, Bentian Jiang, Jingsong Chen, and Evangeline FY Young. "Dr. CU 2.0: A Scalable Detailed Routing Framework with Correct-by-Construction Design Rule Satisfaction". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2019, pp. 1–7.

[70] Liang Li and Evangeline FY Young. "Obstacle-avoiding rectilinear Steiner tree construction". In: *2008 IEEE/ACM International Conference on Computer-Aided Design*. IEEE. 2008, pp. 523–528.

[71] Haiguang Liao, Wentai Zhang, Xuliang Dong, Barnabas Poczos, Kenji Shimada, and Levent Burak Kara. "A deep reinforcement learning approach for global routing". In: *Journal of Mechanical Design* 142.6 (2020).

[72] Chung-Wei Lin, Szu-Yu Chen, Chi-Feng Li, Yao-Wen Chang, and Chia-Lin Yang. "Obstacle-avoiding rectilinear Steiner tree construction based on spanning graphs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.4 (2008), pp. 643–653.

[73] Shiju Lin, Jinwei Liu, Evangeline F.Y. Young, and Martin D.F. Wong. "GAMER: GPU Accelerated Maze Routing". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022), pp. 1–1. DOI: 10.1109/TCAD.2022.3184281.

[74] Genggeng Liu, Zhen Zhuang, Wenzhong Guo, and Ting-Chi Wang. "Rdta: an efficient routability-driven track assignment algorithm". In: *Proceedings of the 2019 on Great Lakes Symposium on VLSI*. 2019, pp. 315–318.

[75] Jinwei Liu, Chak-Wa Pui, Fangzhou Wang, and Evangeline FY Young. "Cugr: Detailed-routability-driven 3d global routing with probabilistic resource model". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2020, pp. 1–6.

[76] Siting Liu, Peiyu Liao, Rui Zhang, Zhitang Chen, Wenlong Lv, Yibo Lin, and Bei Yu. "FastGR: global routing on CPU-GPU with heterogeneous task graph scheduler". In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2022, pp. 760–765.

[77] Wen-Hao Liu, Wei-Chun Kao, Yih-Lang Li, and Kai-Yuan Chao. "NCTU-GR 2.0: Multithreaded collision-aware global routing with bounded-length maze routing". In: *IEEE Transactions on computer-aided design of integrated circuits and systems* 32.5 (2013), pp. 709–722.

[78] Don MacMillen, Raul Camposano, D Hill, and Thomas W Williams. "An industrial view of electronic design automation". In: *IEEE transactions on computer-aided design of integrated circuits and systems* 19.12 (2000), pp. 1428–1448.

[79] Stefanus Mantik, Gracieli Posser, Wing-Kai Chow, Yixiao Ding, and Wen-Hao Liu. "ISPD 2018 initial detailed routing contest and benchmarks". In: *Proceedings of the 2018 International Symposium on Physical Design*. 2018, pp. 140–143.

[80] Larry McMurchie and Carl Ebeling. "PathFinder: A negotiation-based performance-driven router for FPGAs". In: *Reconfigurable Computing*. Elsevier, 2008, pp. 365–381.

[81] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Sungmin Bae, et al. "Chip placement with deep reinforcement learning". In: *arXiv preprint arXiv:2004.10746* (2020).

[82] Tom M Mitchell and Tom M Mitchell. *Machine learning*. Vol. 1. 9. McGraw-hill New York, 1997.

[83] Michael D Moffitt. "MaizeRouter: Engineering an effective global router". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.11 (2008), pp. 2017–2026.

[84] Gordon E Moore et al. "Progress in digital integrated electronics". In: *Electron devices meeting*. Vol. 21. Washington, DC. 1975, pp. 11–13.

[85] Gi-Joon Nam, Cliff Sze, and Mehmet Yildiz. "The ISPD global routing benchmark suite". In: *Proceedings of the 2008 international symposium on Physical design*. 2008, pp. 156–159.

[86] Tim Nieberg. "Gridless pin access in detailed routing". In: *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2011, pp. 170–175.

[87] Muhammet Mustafa Ozdal. "Detailed-routing algorithms for dense pin clusters in integrated circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.3 (2009), pp. 340–349.

[88] Muhammet Mustafa Ozdal and Martin DF Wong. "Archer: a history-driven global routing algorithm". In: *2007 IEEE/ACM International Conference on Computer-Aided Design*. IEEE. 2007, pp. 488–495.

[89] Min Pan and Chris Chu. "FastRoute 2.0: A high-quality and efficient global router". In: *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*. IEEE Computer Society. 2007, pp. 250–255.

[90] Min Pan and Chris Chu. "FastRoute: A step to integrate global routing into placement". In: *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*. ACM. 2006, pp. 464–471.

[91] Chak-Wa Pui, Gengjie Chen, Wing-Kai Chow, Ka-Chun Lam, Jian Kuang, Peishan Tu, Hang Zhang, Evangeline FY Young, and Bei Yu. "RippleFPGA: A routability-driven placement for large-scale heterogeneous FPGAs". In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2016, pp. 1–8.

[92] Jarrod A Roy and Igor L Markov. "High-performance routing at the nanometer scale". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.6 (2008), pp. 1066–1077.

[93] Naveed A Sherwani. *Algorithms for VLSI physical design automation*. Springer Science & Business Media, 2012.

[94] Daohang Shi and Azadeh Davoodi. "TraPL: Track planning of local congestion for global routing". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. 2017, pp. 1–6.

[95] Weiping Shi and Chen Su. "The rectilinear Steiner arborescence problem is NP-complete". In: *SIAM Journal on Computing* 35.3 (2005), pp. 729–740.

[96] Fan-Keng Sun, Hao Chen, Ching-Yu Chen, Chen-Hao Hsu, and Yao-Wen Chang. "A multithreaded initial detailed routing algorithm considering global routing guides". In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–7.

[97] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[98] R-S Tsay. "An exact zero-skew clock routing algorithm". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 12.2 (1993), pp. 242–249.

[99] Ren-Song Tsay. "Exact zero skew". In: *The Best of ICCAD*. Springer, 2003, pp. 509–520.

[100] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[101] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. "Pointer networks". In: *Advances in neural information processing systems* 28 (2015).

[102] Team VLSI. *ASIC Design Flow - An Overview*. https://teamvlsi.com/2020/05/asic-design-flow-overview-v1.html.

[103] Hanrui Wang, Jiacheng Yang, Hae-Seung Lee, and Song Han. "Learning to design circuits". In: *arXiv preprint arXiv:1812.02734* (2018).

[104] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Tim Cheng. *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[105] David Michael Warme and Jeffrey S Salowe. *Spanning trees in hypergraphs with applications to Steiner trees*. University of Virginia, 1998.

[106] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3 (1992), pp. 229–256.

[107] Man-Pan Wong, Wen-Hao Liu, and Ting-Chi Wang. "Negotiation-based track assignment considering local nets". In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2016, pp. 378–383.

[108] Yiu-Chung Wong and Chris Chu. "A scalable and accurate rectilinear Steiner minimal tree algorithm". In: *2008 IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE. 2008, pp. 29–34.

[109] Tai-Hsuan Wu, Azadeh Davoodi, and Jeffrey T Linderoth. "GRIP: scalable 3D global routing using integer programming". In: *Proceedings of the 46th Annual Design Automation Conference*. ACM. 2009, pp. 320–325.

[110] Zhiyao Xie, Yu-Hung Huang, Guan-Qi Fang, Haoxing Ren, Shao-Yun Fang, Yiran Chen, and Jiang Hu. "RouteNet: Routability prediction for mixed-size designs using convolutional neural network". In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.

[111] Zhiyao Xie, Rongjian Liang, Xiaoqing Xu, Jiang Hu, Yixiao Duan, and Yiran Chen. "Net 2: A graph attention network method customized for pre-placement net length estimation". In: *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2021, pp. 671–677.

[112] Yue Xu, Yanheng Zhang, and Chris Chu. "FastRoute 4.0: global router with efficient via minimization". In: *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*. IEEE Press. 2009, pp. 576–581.

[113] Alexander Z Zelikovsky and Ion I Mandoiu. "Practical approximation algorithms for zero-and bounded-skew trees". In: *SIAM Journal on Discrete Mathematics* 15.1 (2001), pp. 97–111.

[114] Yanheng Zhang and Chris Chu. "RegularRoute: An efficient detailed router applying regular routing patterns". In: *IEEE transactions on very large scale integration (VLSI) systems* 21.9 (2012), pp. 1655–1668.

[115] Yanheng Zhang and Chris Chu. "RegularRoute: An efficient detailed router with regular routing patterns". In: *Proceedings of the 2011 international symposium on Physical design*. 2011, pp. 45–52.

[116] Yanheng Zhang, Yue Xu, and Chris Chu. "FastRoute3. 0: a fast and high quality global router based on virtual capacity". In: *2008 IEEE/ACM International Conference on Computer-Aided Design*. IEEE. 2008, pp. 344–349.

[117] Yilin Zhang, Ashutosh Chakraborty, Salim Chowdhury, and David Z Pan. "Reclaiming over-the-IP-block routing resources with buffering-aware rectilinear Steiner minimum tree construction". In: *2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2012, pp. 137–143.

[118] Hai Zhou. "Efficient Steiner tree construction based on spanning graphs". In: *Proceedings of the 2003 international symposium on Physical design*. 2003, pp. 152–157.

[119] Hai Zhou, Narendra Shenoy, and William Nicholls. "Efficient minimum spanning tree construction without Delaunay triangulation". In: *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*. 2001, pp. 192–197.

[120] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. "Graph neural networks: A review of methods and applications". In: *AI Open* 1 (2020), pp. 57–81.

List of Publications

[1] Shiju Lin, Jinwei Liu, Evangeline F.Y. Young, and Martin D.F. Wong. "GAMER: GPU Accelerated Maze Routing". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022), pp. 1–1. DOI: 10.1109/TCAD.2022.3184281.

[2] Jinwei Liu, Xiaopeng Zhang, Shiju Lin, Xinshi Zang, Jingsong Chen, Bentian Jiang, Martin DF Wong, and Evangeline FY Young. "Partition and Place Finite Element Model on Wafer Scale Engine". In: *2022 59th ACM/IEEE Design Automation Conference (DAC)*.

[3] Shiju Lin, Jinwei Liu, Tianji Liu, and Martin DF Wong. "NovelRewrite: Node-Level Parallel AIG Rewriting". In: *2022 59th ACM/IEEE Design Automation Conference (DAC)*.

[4] Jinwei Liu, Gengjie Chen, and Evangeline FY Young. "REST: Constructing Rectilinear Steiner Minimum Tree via Reinforcement Learning". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2021, pp. 1135–1140.

[5] Shiju Lin, Jinwei Liu, and Martin DF Wong. "GAMER: GPU Accelerated Maze Routing". In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE. 2021, pp. 1–8.

[6] Fangzhou Wang, Lixin Liu, Jingsong Chen, Jinwei Liu, Xinshi Zang, and Martin DF Wong. "Starfish: An Efficient P&R Co-Optimization Engine with A*-based Partial Rerouting". In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE. 2021, pp. 1–9.

[7] Bentian Jiang, Jingsong Chen, Jinwei Liu, Lixin Liu, Fangzhou Wang, Xiaopeng Zhang, and Evangeline FY Young. "CU. POKer: Placing DNNs on WSE With Optimal Kernel Sizing and Efficient Protocol Optimization". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.6 (2021), pp. 1888–1901.

[8] Bentian Jiang, Jingsong Chen, Jinwei Liu, Lixin Liu, Fangzhou Wang, Xiaopeng Zhang, and Evangeline FY Young. "CU. POKer: placing DNNs on wafer-scale AI accelerator with optimal kernel sizing". In: *Proceedings of the 39th International Conference on Computer-Aided Design*. 2020, pp. 1–9.

[9] Jinwei Liu, Chak-Wa Pui, Fangzhou Wang, and Evangeline FY Young. "Cugr: Detailed-routability-driven 3d global routing with probabilistic resource model". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2020, pp. 1–6.

[10] Jingsong Chen, Jinwei Liu, Gengjie Chen, Dan Zheng, and Evangeline FY Young. "March: Maze routing under a concurrent and hierarchical scheme for buses". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–6.