VLSI Routing: Seeing Nano Tree in Giga Forest

CHEN, Gengjie

A Thesis Submitted in Partial Fulfilment

of the Requirements for the Degree of

Doctor of Philosophy

in

Computer Science and Engineering

The Chinese University of Hong Kong

June 2019

Abstract

We are using nanometer-sized transistors and gigahertz clock frequency in very large scale integration (VLSI). Under such extreme conditions, timing, power, manufacturability and reliability are all crucial issues in VLSI design. Tree structure is the major topology used in VLSI routing. Optimizing the tree and forest on a chip is essential for a successful design flow. However, the problems are in general difficult in two aspects, single-net routing and multiple-net routing. The thesis will discuss both of them.

For many single-net routing problems, finding an optimal tree from a huge candidate forest is already NP-hard. Two fundamental multi-objective problems are studied here. In signal net, shallowness (path length) implies wire delay, while lightness (wirelength) implies routing resource usage, power, cell delay and wire delay. Shallowness and lightness are both needed but they contradict with each other. We propose an effective algorithm to build Steiner shallow-light tree and trade off between the two. Regarding clock net, skew is the maximum difference in signal arrival time among sinks, which should be minimized. Zero-skew tree (ZST) is too expensive in wirelength and also unnecessary, but can help to build bounded-skew tree (BST). We prove the equivalence between the wirelength minimization of ZST and the diameter sum minimization of hierarchical clustering. Based on this insight, better algorithms for both ZST and BST are proposed.

Regarding multiple-net routing, a large number of trees need to be built on chip by sharing resources and need to be well coordinated for avoiding conflicts. Three problems are solved in this aspect in this thesis. First, for detailed routing, a set of two-level sparse data structures is designed to store the enormous 3D grid graph with runtime and memory efficiency. An effective path search algorithm is also proposed to capture the minimum-area constraint directly. Second, in the bus routing problem, the topologies of all the bits in a bus need to be consistent. We propose to resolve it by routing all the bits concurrently. Meanwhile, efficiency is achieved by a hierarchical scheme. Third, in order to tackle the power issue in 3D ICs, tree-like liquid cooling network is optimized for a better trade-off between thermal profile and pumping power.

摘要

我們在超大規模集成（VLSI）中使用納米級晶體管和千兆赫茲時鐘頻率。 在如此極端的條件下，時序，功耗，可製造性和可靠性都是VLSI設計中的關鍵問題。 樹結構是VLSI佈線中使用的主要拓撲。優化芯片上的樹和森林對於一個成功的設計流程至關重要。 但是，在單網路由和多網路由兩個方面，問題一般都很困難。 論文將討論它們。

對於許多單網佈線問題，從巨大的候選森林中尋找最佳樹已經是NP難的。 在信號網中，淺度（路徑長度）意味著線路延遲，而輕度（線路長度）意味著佈線資源佔用，功耗，信號延遲和線路延遲。 淺度和輕度都是必需的，但它們相互矛盾。 我們提出了一種有效的算法來構建斯坦納淺輕樹並在兩者之間進行權衡。 關於時鐘網，時鐘偏斜是信號到達時間的最大差異，應該最小化。 零偏斜樹（ZST）的線長太貴而且也是不必要的，但可以幫助構建有界偏斜樹（BST）。 我們證明了ZST的線長最小化與層次聚類的直徑和最小化之間的等價性。 基於這種見解，提出了更好的ZST和BST算法。

對於多網佈線，需要通過共享資源在芯片上構建大量樹，並且需要很好地協調以避免衝突。 本論文解決了這方面的三個問題。 首先，對於詳細佈線，設計了一組兩級稀疏數據結構以時間和存儲高效的方式來存儲巨大三維網格圖。 還提出了一種有效的路徑搜索算法來直接捕獲最小面積約束。 其次，在總線佈線問題中，總線中所有位的拓撲需要保持一致。 我們提出通過同時佈所有位的線來解決它。 同時，通過層次方案實現效率。 第三，為了解決三維芯片中的功率問題，樹狀液體冷卻網絡經過優化，可以更好地在熱分佈和泵功率之間進行權衡。

Acknowledgments

First of all, I sincerely thank my advisor, Prof. Evangeline F. Y. Young, for her guidance and support over the years. Throughout my Ph.D. study, she not only encourages me to explore challenging research problems that combine the beauty of algorithms with practical needs, but also shares with me the precious experience and knowledge. This thesis would be impossible without her warm advice and continuous encouragement.

I also want to express my grateful thanks to the rest of my doctoral committee, Prof. Qiang Xu, Prof. Bei Yu, and Prof. Wai-Kei Mak, for their insightful comments and constructive suggestions.

My sincere thanks also go to my fellow labmates: Dr. Jian Kuang, Dr. William Wing-Kai Chow, Dr. Peishan Tu, Ka-Chun Lam, Jordan Chak-Wa Pui, Hang Zhang, Haocheng Li, Sirius Chong Wing Cheung, Jingsong Chen, Bentian Jiang, Jinwei Liu, Xiaopeng Zhang, and Dan Zheng. Thanks for all the helps from you and all the fun we had in the last four years.

Last but not least, I am sincerely thankful to my parents, Jinhui Chen and Sufang Chen, for their unconditional love. I am also deeply grateful for my wife, Yixuan Shang, for everything. Thank you for being there for me from the very beginning.

This work is partially supported by the Hong Kong Ph.D. Fellowship Scheme.

Contents

List of Figures

List of Tables

Part I

Introduction and Background

Chapter 1

Introduction

1.1 VLSI Design

The *very large scale integration* (VLSI) technology is the core of today's electronic equipment and the foundation of our digital world. Servers, personal computers, mobile phones, wearable devices, and other computing equipment, which are indispensable parts of modern life, are all impossible without the explosive development of the VLSI technology during the past decades. Moreover, cars, airplanes, home appliances, cameras, industrial pipelines, and many other traditional industries, have been revolutionized by the continuous digitalization enabled by VLSI technology.

Nowadays, we are using nanometer-sized transistors and gigahertz clock frequency in the VLSI design. Under such extreme conditions, timing, power, manufacturability and reliability are all crucial issues in VLSI design. For example, 50% – 80% of gates in the high-performance *integrated circuit* (IC) today are repeaters, which do not perform useful computation but work for timing closure [1]; over 50% of the chip at around 7nm will be powered off and cannot be utilized due to the power constraint [2]; the power-sensitive applications in mobile and the Internet of Things become ubiquitous [3].

The process of designing and optimizing a VLSI circuit is multi-objective and complicated. From architectural design all the way to signoff, it can be roughly separated into several steps [4], as Figure 1.1 shows.

- **Architectural and register-transfer level design** converts system specification to *hardware description language* (HDL) like Verilog or VHDL.
- **Logic synthesis** maps HDL to a gate-level netlist.
- **Placement** determines the spatial locations of all the instantiated design components (e.g., gates, transistors, macros).
- **Routing** assigns routing resources, usually the wires and vias across a stack of metal layers, to connect all the nets.

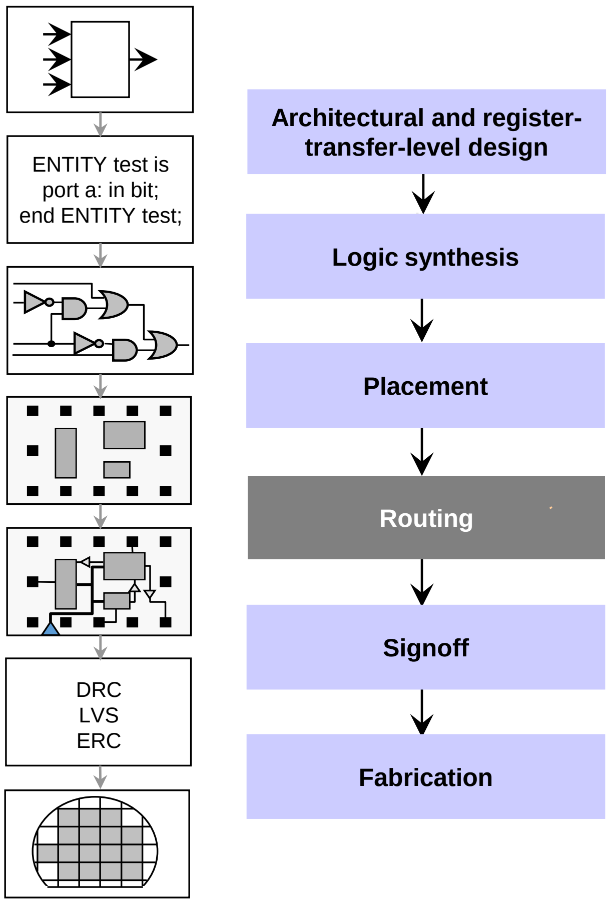


Figure 1.1: The VLSI design flow.

- **Signoff** is to fully verify the electrical and logical functionality of the design and ensure its correctness.

The process of placement and routing is usually called the *physical design*, or back-end design, as a whole. Modern VLSI design has become so complex that it largely relies on *electronic design automation* (EDA) tools, which provide the algorithms and software for assisting or even automating the design process. The algorithms for VLSI routing are the focus of this thesis.

1.2 VLSI Routing Problem

Tree structure is the major topology used in VLSI routing. Optimizing the tree and the forest is essential for a successful design flow [5]. It directly determines various eventual design metrics such as the power profile, timing closure, manufacturability and reliability. However, VLSI routing problems are in general challenging. First, even for many *single-net routing* problems, finding an optimal tree from a huge candidate forest is already NP-hard. Second, for *multiple-net routing*, a large number of trees need to be built on the chip by sharing resources and to be well coordinated for avoiding conflicts.

Solving the subproblem of routing a single net is the foundation of routing all the nets of a chip. However, even though we ignore the resource competition among all the nets here, the problem is still very challenging. Many basic NP-hard problems have been studied by researchers for decades. Moreover, the problems are typically multi-objective in nature even under a simplified model.

On top of the difficulties for the single-net routing, multiple-net routing needs to resolve the congestion issue, i.e., the resource competition among the huge number of nets. Because of its enormous computational complexity, multiple-net routing is usually performed in two stages, global and detailed. In the *global routing* stage, the routing space is split into an array of regular cells, where a coarse-grained routing solution is generated. It optimizes wirelength, via count, routability, timing and other metrics with a global view. *Detailed routing*, on the other hand, realizes the global routing solution by considering exact metal shapes and positions. It takes care of many complicated detailed design rules (e.g., parallel-run spacing, end-of-line spacing, cut spacing, minimum area, etc).

1.3 Overview of this Thesis

The major limitations of the many previous works on VLSI routing are in three folds. First, many works heavily rely on heuristics without any theoretical guarantee, which means inferior average-case quality and unpredictable worst-case performance. Second, step-after-step post-processing may be effective in some situations, but it is insufficient for a comprehensive routing problem where many factors (e.g., power, timing, manufacturability) need to be simultaneously considered. Third, some methods cannot be scaled to today's chip with up to billions of transistors. The thesis proposes a set of solutions by trying to overcome the limitations.

In Chapter 2 of this part, we will review the literature on VLSI routing, especially those closely related to our contributions. The thesis is then organized into two major parts.

In Part II, we study two fundamental problems for single-net routing with mathematical rigorousness. In Chapter 3, regarding the trade-off between wirelength and path length in signal net, our proposed Steiner shallow-light tree algorithm achieves a best bound of $(1+\epsilon, 2+\lceil\log\frac{2}{\epsilon}\rceil)$ [6, 7]. In Chapter 4, we prove the equivalence between zero-skew tree problem and hierarchical clustering problem for the clock tree. The new insight leads us to simple yet more effective algorithms pursuing wirelength and skew [8].

In Part III, we propose two fast and high-quality routers for two multiple-net routing problems beyond the relatively well-studied global routing problem. Special design rules are satisfied together with other design rules in a correct-by-construction manner. Chapter 5 describes a scalable framework with two-level sparse data structures and path search capturing the nontrivial minimum area constraint is designed [9]. In order to meet the topology consistency constraint in bus routing, an efficient maze router under a concurrent and hierarchical scheme is proposed in Chapter 6 [10]. Moreover, Chapter 7 introduces our method for optimizing the tree-like liquid cooling network to better trade-off between the thermal profile and power consumption of 3D ICs [11].

Chapter 2

Literature Review

This chapter reviews the literature about VLSI routing in both single-net routing and multiple-net routing.

2.1 Single-Net Routing

Among all the objectives optimized in the single-net routing problems, *wirelength* is the most important one. It implies many things including routing resource usage (routability), power consumption, cell delay and wire delay [12]. Meanwhile, the *path lengths* from the source to sinks of the net have a huge impact on wire delay. For signal nets, both wirelength and path length are important but they are usually contradicted with each other, which should be properly balanced. Clock signal is distributed to synchronous elements in a VLSI circuit. In a clock net, *skew*, the maximum difference in signal arrival time among all sinks, should be small to ensure timing correctness. Skew should also be optimized with a controlled wirelength overhead.

2.1.1 Wirelength Minimization

For spanning trees, the *minimum spanning tree* (MST) can be obtained by various classical algorithms like Prim's and Kruskal's algorithms in $O(m + n \log n)$ time [13]. For rectilinear Steiner trees, the one with minimum tree weight is called a *rectilinear Steiner minimum tree* (RSMT). The RSMT construction can be achieved by using Steiner nodes on Hanan grid [14] and is NP-hard [15]. Besides the exponential-time exact algorithms (e.g., GeoSteiner [16]), there are, however, many efficient heuristics achieving good or even near optimal quality. First of all, the *rectilinear MST* (RMST) actually provides a 1.5-approximation [17] and can be constructed in $O(n \log n)$ time [18]. It also seeds many RSMT heuristics. Ho et al. [19] give a linear-time dynamic programming to optimally overlap the edges and reduce the wirelength for separable MSTs. The iterated 1-Steiner (I1S) heuristic due to Kahng and Robins [20] iteratively inserts Steiner points with the

Table 2.1: Spanning and Steiner Shallow-Light Tree

	Shallowest	Lightest	Shallow light
Spanning	Spanning SPT ($O(m + n \log n)$)	MST ($O(m + n \log n)$)	Spanning SLT
Steiner	Steiner SPT (NP hard)	SMT (NP hard)	Steiner SLT
Rectilinear Steiner	RSMA (NP hard)	RSMT (NP hard)	Rectilinear Steiner SLT

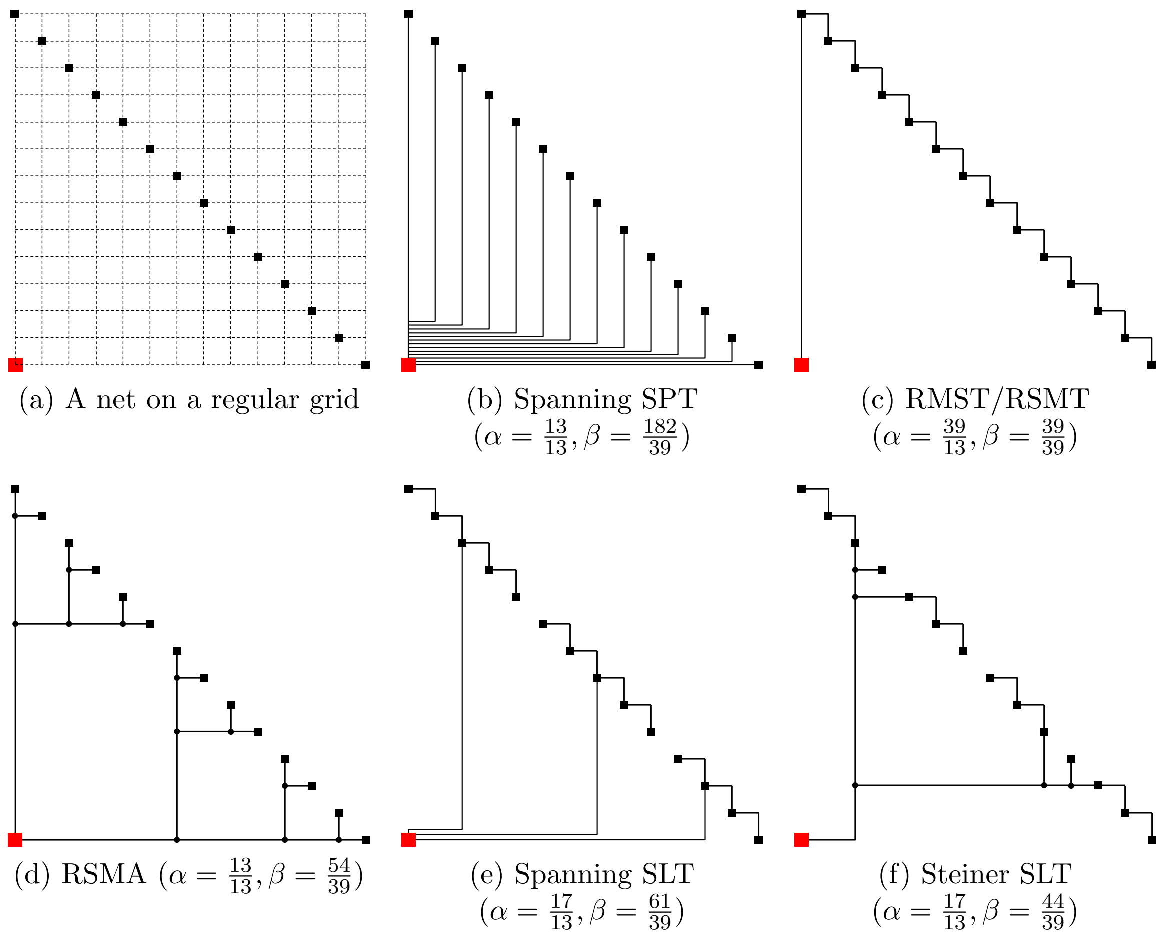
largest wirelength saving. Griffith et at. [21] proposed a faster $O(n^3)$-time implementation of I1S based on dynamic MST maintenance. The BOI algorithm proposed by Borah et al. [22] repeatedly connects a vertex to a nearby edge and removes the longest edge of the loop formed. Zhou [23] uses spanning graph [18] to help finding good candidates for the edge substitution in BOI and thus improves the runtime to $O(n \log n)$ with small hidden constants. The $O(n \log^2 n)$-time batch greedy algorithm (BGA) by Kahng et al. [24] is a batched version of the greedy triple contraction. Chu et al. present FLUTE, which efficiently constructs RSMTs based on pre-computed look-up tables and net breaking in $O(n \log n)$ time and is optimal for nets up to 9 pins [25].

2.1.2 Trade-off Between Wirelength and Path Length

For spanning trees, the *shortest-path tree* (SPT) can be constructed by Dijkstra's algorithm in $O(m + n \log n)$ time [26]. For rectilinear Steiner trees, the lightest one with all paths from root being shortest is a *rectilinear Steiner minimum arborescence* (RSMA). The RSMA construction is also NP-hard [27]. Approaches for optimal RSMAs include integer programming [28] and dynamic programming [29]. An $O(n \log n)$-time 2-approximation is first proposed by Rao et at. [30] and later generalized to all four quadrants by Córdova and Lee [31]. There are also several other fast algorithms. Cong et al. [32] propose the A-tree algorithm with safe moves and heuristic moves towards RSMA. Alexander and Robins [33] adapt the iterative greedy insertion scheme of I1S to the RSMA problem. Pan et al. [34] apply the table look-up and net breaking techniques in FLUTE to RSMA construction.

The spanning/Steiner *shallow-light tree* (SLT) combines the objectives of shallow path length and light tree weight together, as Table 2.1 and Figure 2.1 show. In a spanning/Steiner tree with *shallowness* α and *lightness* β, each path length is at most α times the shortest-path distance, while the tree weight is β times the minimum tree weight. In an $(\bar{\alpha}, \bar{\beta})$-SLT, $\alpha \leq \bar{\alpha}$ and $\beta \leq \bar{\beta}$.

A spanning SLT approximates SPT and MST simultaneously, where the trade-off is in the order of $(1 + \epsilon, O(\frac{1}{\epsilon}))$. The ABP algorithm by Awerbuch et al. [35] and the BRBC algorithm by Cong et at. [36] are in fact identical and provide a bound of $(1 + 2\epsilon, 1 + \frac{2}{\epsilon})$.


(a) A net on a regular grid

(b) Spanning SPT ($\alpha = \frac{13}{13}, \beta = \frac{182}{39}$)

(c) RMST/RSMT ($\alpha = \frac{39}{13}, \beta = \frac{39}{39}$)

(d) RSMA ($\alpha = \frac{13}{13}, \beta = \frac{54}{39}$)

(e) Spanning SLT ($\alpha = \frac{17}{13}, \beta = \frac{61}{39}$)

(f) Steiner SLT ($\alpha = \frac{17}{13}, \beta = \frac{44}{39}$)

Figure 2.1: Routing topologies with varied shallowness α and lightness β (root is marked by red).

The algorithm first constructs an MST and a Hamiltonian path according to the depth-first tour of the MST. It then accumulates the distance along the Hamiltonian path and identify a breakpoint whenever the accumulated distance becomes too long. On the graph that unions the MST and the edges connecting all the breakpoints to the root, the SPT is obtained and outputted. In this way, the distance between a breakpoint and the root becomes the shortest-path distance. For other vertices, the path length is bounded.

After the ABP/BRBC algorithm, Khuller et al. [37] propose a beter algorithm, the KRY algorithm. It also modifies an initial light topology towards a SLT. However, it directly uses the MST itself as the initial topology and identifies breakpoints during a depth-first tour on the MST. The KRY algorithm has a bound of $(1+\epsilon, 1+\frac{2}{\epsilon})$, which is also proved to be the best possible one for spanning trees. It also provides a smooth trade-off between SPT and MST controlled by ϵ, while ABP does not (e.g., a MST is not implied when $\epsilon = +\infty$).

Besides, the PD algorithm due to Alpert et al. [38] smoothly trades off between SPT and MST by a direct combination of Dijkstra's and Prim's algorithms. Note that both Dijkstra's and Prim's algorithms start with the root vertex and iteratively add an unvisited vertex until visiting all vertices. In Prim's algorithm, an unvisited vertex the *closest to a visited vertex* is chosen, while Dijkstra's algorithm prefers an unvisited vertex the *closest to the root*. The PD algorithm achieves the trade-off by setting the preference to a weighted sum of the objectives of the two algorithms. The approach has been widely used in industry [3], but the resulted tree is not guaranteed to be a SLT.

Recently, Steiner SLTs are proved to be exponentially lighter than spanning ones by Elkin and Solomon [39, 40]. The ES algorithm extends the ABP algorithm for spanning SLTs to Steiner ones. Its main idea is still to accumulate the distance along a Hamiltonian path and identify breakpoints. But breakpoints are then connected to the root by a Steiner SPT instead of individual edges. It generates a Steiner $(1+\epsilon, O(\log\frac{1}{\epsilon}))$-SLT with a time complexity of $O(n^2)$. The constants in the shallowness-lightness bound of $(1+2\epsilon, 4+2\lceil\log\frac{2}{\epsilon}\rceil)$ are, however, quite large (log denotes $\log_2$ in this thesis). In the work [41], Held and Rotter study the problem of Steiner SLT with vertex delays (measured by the number of bifurcations). When vertex delays are not taken into account, they tighten the bound of ES to $(1+\epsilon, 2+\lceil\log\frac{2}{\epsilon}\rceil)$ in 2D Manhattan space.

2.1.3 Trade-off Between Wirelength and Skew

There are in general two base formulations for clock tree construction, *zero-skew tree* (ZST) and *bounded-skew tree* (BST).

Many methods have been proposed for building ZSTs, e.g., [42–48]. Among them, deferred-merge embedding (ZST/DME) is a dynamic programming approach by Chao et al. [44]. For a given topology, it outputs the locations of Steiner points achieving zero skew and optimal wirelength under the linear or Elmore delay model. In ZST/DME, the *merging segment* of a vertex is a set of possible placements of the vertex. The merging segment of a sink is simply itself. The merging segment of a Steiner node (i.e., between two merging segments) is a Manhattan arc (line segment with slope +1 or -1). There are two phases in ZST/DME. In the first bottom-up phase, a tree of merging segments is computed recursively. In the top-down phase, the merging point achieving the minimum wirelength is picked according to the location of its parent Steiner node. For determining the ZST topology, Greedy-DME, the DME-based greedy heuristics proposed by Edahiro [46], is regarded as the best algorithm in practice (see [12, 48–50]). The key idea of the method is simple. It iteratively merges the two merging segments with the smallest distance between them until only a single merging segment remains. Several techniques have been also proposed to accelerate the process and refine the result.

Nowadays, ZST is, however, not a good choice in practice due to two reasons. First, ZST is too expensive in wirelength, which implies excessive power usage. Note that the clock net can consume 40% of the overall chip power due to its high frequency and large coverage [51]. Besides, longer wirelength usually comes with larger path divergence and suffers from more on-chip variations. Second, ZST topology is not necessary, considering the large tolerance (due to buffer insertion and sizing [52,53]) and the widely-used useful-skew optimization techniques [54,55]. Nonetheless, ZST is still useful as it can serve as the backbone of a BST [47,48].

Regarding BST, Cong et al. extend ZST/DME to BST/DME by generalizing the merging segments to regions [50]. Due to the more complicated shapes of the merging regions, BST/DME prunes many possibilities (e.g., restrict the merging point to the boundary of a merging region or sample the interior points) and is not optimal for a given topology. Besides, skew variation is allocated over all levels relatively by chance.

With theoretical interest, approximation algorithms are also proposed [47,48]. The approaches of Zelikovsky and Mandoiu [48] give the best approximation ratios – three and nine for ZST and BST respectively. The algorithm is not as good as the best heuristics in practice, but provides several inspirations for this work. Besides, an integer linear programming (ILP) method is recently proposed for constructing optimal BST [56].

2.2 Multiple-Net Routing

We will first introduce the two major paradigms for resolving the resource competition issue in routing multiple nets. Then, we will review the previous works on global routing and detailed routing respectively. After that, the problem of bus routing will be introduced. In the end, we will provide a background on the liquid cooling network for 3D IC, where multiple tree-like cooling networks will help to cool down the 3D IC.

2.2.1 Sequential and Concurrent Routing

There are two major paradigms for routing multiple nets: sequential routing and concurrent routing [57].

In *sequential routing*, nets are routed one after another, where previously routed nets are regarded as obstacles for later nets. A basic tool for routing a net is *maze routing* [58], which finds a shortest path between two points on a regular grid graph with possibly some obstacles. It is performed like a breath-first search. When the grid graph has weighted routing edges or even irregular structures, a priority queue can help the efficient expansion by the idea of Dijkstra's algorithm. There are in general two approaches to route a net with more than two pins by maze routing. The first is to decompose the net into multiple two-pin nets by Steiner tree method and then perform maze routing for each two-pin net. The second is to start from a source pin for searching other sink pins. Whenever a sink pin is reached, the whole path from the source to this sink pin is also regarded as the source. Moreover, after routing all the nets sequentially with possible violations, several rounds of negotiation-based *rip-up and reroute* (RRR) help to clean them up [59].

The sequential routing algorithms are commonly used mainly because of their simplicity, scalability and flexbility. However, its quality heavily relies on the net order. To avoid the issue, *concurrent routing* methods adopt the formulations such as multi-commodity flow and integer linear programming (ILP), but they usually suffer from poor scalability.

2.2.2 Global Routing

During the past two decades, many approaches were proposed to complete fast and high-quality global routing with a sustaining progress.

Most of the methods adopt the scheme of sequential routing. Labyrinth [60] proposes the pattern routing, which can greatly accelerate the maze routing by constraining the number of bends in the path. FGR [61] applies discrete Lagrange multipliers and extends A* search to build up a high-performance global router. FastRoute [62–64] proposes a congestion-driven Steiner tree topology generation technique and monotonic routing to avoid invoking maze routing too frequently. Besides, a multi-source multi-sink maze routing is proposed to overcome the limitations of decomposing the multi-pin net into two-pin nets. Moreover, it adopts via-aware Steiner tree generation, 3-bend routing and layer assignment with careful ordering to reduce via count. BoxRouter [65,66] uses ILP in gradually expanded boxes for congested regions together with an adaptive maze routing. It also applies robust negotiation-based A* search for routing stability, and topology-aware RRR for flexibility. Ancher [67] proposes a history-based cost metric, a framework smoothly trading off between overflow and wirelength, and a Lagrangian relaxation-based bounded-length min-cost topology improvement algorithm. NTHU-route [68] explores the history based cost function and ordering methods for the global routing. NCTU-GR [69]) proposes a bounded-length maze routing algorithm and a RSMT-aware routing scheme together with a collision-aware multithreading to better balance the quality and runtime.

There are also some exploration on the concurrent routing method. Albrecht et al. [70] formulates the global routing as a multi-commodity flow problem, which can be solved by an approximation algorithm for fractional flow together with the randomized rounding. GRIP [71] solves the ILP formulation for the entire global routing problem by the column generation method. It overcomes the scalability issue of ILP by decomposing the chip into rectangular subregions. BonnRoute [72] models the global routing by the min-max resource sharing problem, a generalization of the multi-commodity flow problem, where a path composition Steiner tree algorithm is used as an oracle function. RRR is also performed to resolve the violations due to the randomized rounding. The scheme has also been extended to take timing into account recently [73].

2.2.3 Detailed Routing

The solution quality of detailed routing directly influences various eventual design metrics such as timing, signal integrity, and chip yield [74]. Meanwhile, its solution space, a 3D grid graph, is significantly larger than that of global routing. In advanced technology nodes, detailed routing becomes the most complicated and time-consuming stage [75].

However, there is insufficient effort for exploring efficient and effective detailed routers compared with global routing in academia. RegularRoute [74] encourages regular routing patterns and exploits a maximum independent set formulation for better design rule satisfaction. MANA [76] considers end-of-line spacing and minimum length of a wire segment in maze routing. The work in [77] presents the data structures and algorithms for detailed routing used in BonnRoute. Besides, several specific issues in detailed routing have been discussed. For example, methods for the pin access optimization are proposed in [78–80]. For others, the impact of various manufacturing technologies have been dealt with, including triple patterning [81–83], self-aligned doubling patterning [84], and directed self-assembly [85].

Recently, the ISPD 2018 Initial Detailed Routing Contest [75] stimulates several works on detailed routing. Kahng et al. [86] propose TritonRoute, a detailed router with integer linear programming (ILP) based intra-layer parallel routing. Sun et al. [87] present a detailed routing algorithm with a multi-stage RRR scheme. Their approaches suffer from the weakness in both design rule satisfaction and runtime scalability.

As the feature size scales down, not only the problem size but also the complexity of design rules of detailed routing becomes increasingly challenging. Moreover, many detailed routers heavily rely on post processing for fixing design rule violations. Design rule dimensions, however, do not scale well with feature miniaturization (e.g., feature size decreases much faster than minimum area values) and require relatively more spaces for fixing. In this way, a post processing step fails more frequently [77]. Therefore, a detailed routing framework that is scalable in runtime as well as memory usage and provides more correct-by-construction design rule satisfaction is in need.

2.2.4 Bus Routing

The continuous development of modern VLSI technology has brought new challenges for on-chip interconnections. In modern designs, there are buses with long wires that can introduce long wire delay. To maintain signal integrity, some post-routing optimizations such as buffer insertions are needed. However, if the bits in the same bus are routed in different topologies, it is very difficult to find places to insert buffers for different bits of the same bus in a regular manner. To resolve this problem, it is preferred to have the same routing topology among all the bits of a bus, which is different from classic net-by-net routing. In spite of reducing the size of the solution space of the routing problem to some extent, this topology constraint also makes it more difficult to efficiently allocate appropriate routing resources to each bus on multiple metal layers. Meanwhile, similar to classic net-by-net routing, solution qualities such as wirelength and via count are also important metrics to optimize for bus routing. An effective bus router should provide a solution with high routing quality while maintaining topology consistency in a bus for the benefits of synchronizing signals.

The techniques of net-by-net routing can hardly be straightforwardly applied in bus routing due to the difficulty of maintaining topology consistency. There are some previous works handling issues related to escape routing on printed circuit board (PCB) designs, e.g. pin assignment guaranteeing routability [88], layer assignment to minimize the number of used layers [89], and an ILP-based solution [90] to solve the entire bus planning problem. However, for typical escape routing on PCB designs, having the same topology among different bits of the same bus is not required although the bus bits are typically routed together.

To observe the topology constraint, Streak [91] uses a representative bit to generate a set of topology candidates and then applies an ILP to select a good one. All the other bits in the bus try to follow the selected one. However, the selected topology may not be achievable due to the lack of routing resources. To handle this issue, there is a post-refinement stage in Streak where the original bus will be divided into several sub-buses and different sub-buses will have different topologies. Therefore, the techniques in Streak will not be suitable if the bus structure cannot be changed freely. Besides Streak, there is very few previous work aiming at routing buses with topology constraint.

2.2.5 3D IC Liquid Cooling Network

With not only significant saving in delay, power and area but also possibility of yield increase and heterogeneous integration, through-silicon-via (TSV) based 3D ICs are envisioned as one of the most promising solutions to continue the performance increase of computer systems [92]. However, 3D integration increases both heat dissipation density and thermal resistance from junction to ambient, aggravating the existing thermal problem.

To resolve the huge thermal challenge in chip design and especially in 3D ICs, microchannel-based single-phase liquid cooling has been proposed with immense potential for high-performance servers [93]. Single-phase fluid, such as water, is injected into micro-scale channels (a.k.a. microchannels) etched between two consecutive vertical tiers to carry the heat out from the 3D stack, as shown in Figure 2.2(a). It is much more effective than both conventional air cooling and back-side liquid cold plate [94]. Moreover, with this aggressive cooling mechanism, some high-performance technologies limited by thermal constraints will become possible and leakage power consumption can also be reduced [95].

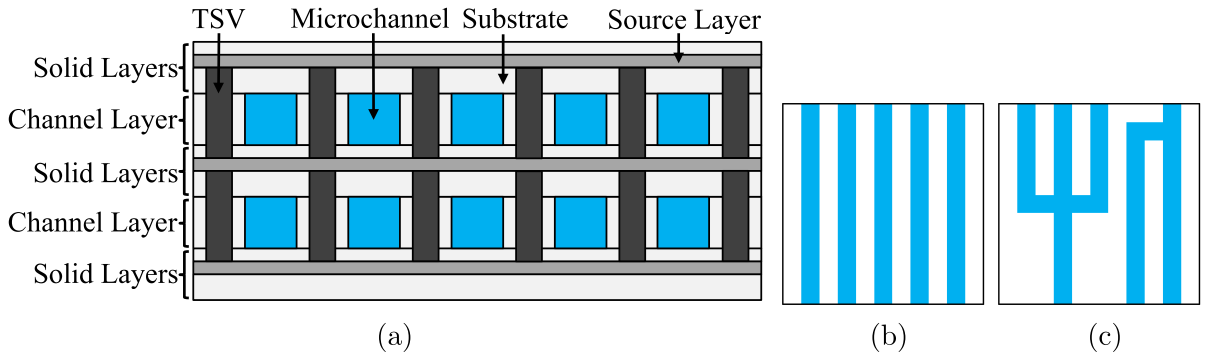


Figure 2.2: (a) 3D IC using microchannel-based liquid cooling. (b) Straight microchannels. (c) Cooling network with bends and branches.

Not only is liquid cooling effective and beneficial, the fabrication of microchannels is also compatible with current CMOS process. Prototypes of 3D ICs with microchannel-based liquid cooling system have been built by various research groups showing promising results [96–98].

However, liquid cooling brings new challenges including large thermal gradient [99] and high pumping requirement [100]. In liquid-cooled chips, coolant absorbs heat along the microchannels as it flows from inlets to outlets, making temperatures in downstream regions tend to be much higher than those in upstream regions. The deduced large thermal gradient may lead to reliability issues and timing errors. Also, due to the limited diameter of the microchannels, the energy required to inject the coolant can be a significant overhead to the whole system.

There have been many studies on the challenges of liquid-cooled 3D ICs. Qian et al. [101] propose to divide straight microchannels into clusters and apply appropriate flow rates separately to avoid pumping power waste. Shi et al. [102] combine microchannel liquid cooling and thermal TSVs to improve power efficiency. Shi et al. [103] also develop a heuristic to allocate straight microchannels, which is further co-optimized with channel sizing and flow rates to minimize the cooling power consumption. The main disadvantage of these proposals is that they all primarily target the improvement of energy efficiency and neglect the thermal gradient. Therefore, the thermal gradient can only be improved by limited extent by chance. Apart from these design-time exploration, the existing run-time thermal management approaches (e.g. [104]) also have this deficiency. In [99], Sabry et al.. use channel width modulation to optimize the cooling energy under thermal gradient and peak temperature constraints. However, the optimization is based on an one dimensional model which ignores heat transfer between regions cooled by different channels and is thus inaccurate on the full-chip scale. In addition, they all consider straight channels only and do not utilize the flexibility of CMOS process to design liquid cooling networks, as Figures 2.2(b) and 2.2(c) show. In [105], Van Oevelen et al.. begin to adopt the topological design in order to minimize heat transfer, but the assumption of constant temperature heat source is far from realistic chip design.

Part II

Single-Net Routing

Chapter 3

Trade-off Between Wirelength and Path Length

In this chapter, we propose an efficient algorithm called SALT for constructing a Steiner SLT and apply it to routing topology construction. Our contributions are summarized as follows.

1. We propose SALT for the Steiner SLT on general graphs, whose shallowness-lightness bound is $(1+\epsilon, 2+\lceil\log\frac{2}{\epsilon}\rceil)$. To the best of our knowledge, the bound is tighter than all the previous methods for constructing general-graph spanning/Steiner SLTs (See Table 3.1).

2. We simplify SALT and reduce the runtime from $O(n^2)$ to $O(n\log n)$, when applying it to the Manhattan space for VLSI routing. We further decrease path lengths and tree weight in the Manhattan space by integrating SALT with the classical RSMA [30] and RSMT [25] algorithms. The method (rectilinear SALT) provides a smooth trade-off between RSMA and RSMT controlled by ϵ[1].

[1]It was found after the preliminary version [6] of this work was published that our rectilinear SALT achieves the same bound as the approach in [41] in the Manhattan space. A minor difference is that rectilinear SALT incorporates a classical RSMA method [31] directly, leading to better Steiner trees in practice. Another small difference is that we break a tie (line 22 of Algorithm 3.2) to reduce the wirelength.

Table 3.1: Historical Progress of Shallow-Light Trees

Algorithm	Shallowness-lightness bound	Metric
ABP/BRBC [35,36]	$(1+2\epsilon, 1+\frac{2}{\epsilon})$	General
KRY [37]	$(1+\epsilon, 1+\frac{2}{\epsilon})$	General
ES [39,40]	$(1+2\epsilon, 4+2\lceil\log\frac{2}{\epsilon}\rceil)$	General
HR [41]	$(1+\epsilon, 2+\lceil\log\frac{2}{\epsilon}\rceil)$	Manhattan
SALT	$(1+\epsilon, 2+\lceil\log\frac{2}{\epsilon}\rceil)$	General

3. We apply several effective safe refinement techniques to improve the wirelength and path lengths of the tree output by rectilinear SALT.

4. As another post-processing step, we design an edge substitution algorithm to further minimize the wirelength, where slight path length degradation is allowed but is controlled under the shallowness constraint.

Note that we follow the definition in ES [40] for the general-metric Steiner tree. There are, however, some limitations on the generality (e.g., cannot be embedded into a Euclidean metric). The definition will be introduced in detail in Section 3.1.1.1.

As a geometric approach for VLSI routing, our method directly targets wirelength and path lengths instead of a highly accurate timing model. However, this is desirable due to three reasons. First, SALT provides a bounded trade-off and has a strong global view. It can generate high-quality initial solutions for later stage optimization. Second, the linear delay model is reasonable due to buffering [106, 107], wire sizing and layer assignment [108], compared to the Elmore delay model. Third, in the experiment, SALT is also comparable in terms of Elmore delay with the state-of-the-art Steiner tree construction method targeting Elmore delay directly [73, 109, 110].

Last but not least, we want to highlight that even though the bound analysis of SALT is complicated, it can be easily implemented with hundreds of lines of codes. The source code of SALT implementation is also publicly available at `https://github.com/chengengjie/salt`.

The remainder of this chapter is organized as follows. The Steiner SLT algorithm on general graph (SALT) is presented in Section 3.1. Its adaption to the Manhattan space (rectlinear SALT) is detailed in Section 3.2. The post-processing techniques for further improving constructed trees are illustrated by Section 3.3 and Section 3.4. In the end, Section 3.5 shows and analyzes the experimental results.

3.1 Steiner Shallow-light Tree Algorithm

The exact problem formulation and the ES algorithm [40] for the Steiner SLT are first briefly introduced as preliminaries. The framework as well as the light Steiner SPT construction of SALT is then described, followed by the bound analysis.

3.1.1 Preliminaries

3.1.1.1 Problem Formulation

Our Steiner SLT algorithm on general graphs is under the same problem formulation used in [40]. A spanning/Steiner tree T of a weighted undirected n-vertex graph $G = (V, E, w)$ with respect to a root vertex r is called an SLT if (i) it approximates all shortest-path

Table 3.2: Notations Used in ES

$MST(G)$	Minimum spanning tree on graph G
$d_G(u, v)$	Distance between vertices u and v in graph G
P_i	i-th vertex on path P

Algorithm 3.1 ES

```
Require: Graph $G = (V, E, w)$, root $r$, trade-off parameter $\epsilon$;
Ensure: Steiner SLT $T = (V', E', w')$ with $V' \supseteq V$ that dominates $G$;
 1: $T_M \leftarrow MST(G)$;
 2: $P \leftarrow$ Hamiltonian path based on $T_M$ starting from $r$;
 3: Breakpoint set $B \leftarrow \emptyset$;
 4: Breakpoint $b \leftarrow r$;
 5: for $v \leftarrow P_1$ to $P_n$ do
 6:     if $d_P(b, v) > \epsilon \cdot d_G(r, v)$ then
 7:         $b \leftarrow v$;
 8:         $B \leftarrow B \cup \{b\}$;
 9:     end if
10: end for
11: $T_B \leftarrow$ Steiner SPT on $G[B \cup \{r\}]$ rooted at $r$;
12: $T \leftarrow$ spanning SPT on graph $T_M \cup T_B$;
```

distances $d_G(r, v)$ from r to $v \in V$, and (ii) its weight $w(T)$ is bounded by that of MST $w(MST(G))$. For a $(\bar{\alpha}, \bar{\beta})$-SLT, (i) the shallowness $\alpha = \max\{\frac{d_T(r,v)}{d_G(r,v)} | v \in V \backslash \{r\}\} \leq \bar{\alpha}$, and (ii) lightness $\beta = \frac{w(T)}{w(MST(G))} \leq \bar{\beta}$. Note that on a graph that is metric (i.e., with edge weights satisfying triangle inequality), a lightness bound with respect to MST infers one with respect to SMT because $w(MST(G)) \leq 2 \cdot w(SMT(G))$ (i.e., $w(T) \leq \bar{\beta} \cdot w(MST(G)) \leq 2 \cdot \bar{\beta} \cdot w(SMT(G))$). For rectilinear Steiner trees, the gap is smaller with $w(RMST(G)) \leq 1.5 \cdot w(RSMT(G))$.

Considering the general metric scenario, a Steiner tree for a graph $G = (V, E, w)$ is defined as a tree $T = (V', E', w')$ with $V' \supseteq V$ and $w' : E' \to \mathbb{R}^+$ that *dominates* the metric M_G induced by G, i.e., $\forall u, v \in V, d_T(u, v) \geq d_G(u, v)$.

Even though such Steiner SLT cannot be embedded into many metric spaces (e.g., Euclidean space[2] or finite graph metrics), it is applicable to the Manhattan space, which will be shown in Section 3.2. For simplicity of illustration, we henceforth assume all the input graph G is complete and metric. Indeed, any weighted undirected graph G^* defines a metric space and thus implies a graph G that is complete and metric.

3.1.1.2 ES Algorithm

The ES algorithm extends the ABP algorithm for spanning SLTs to Steiner ones. The key steps are shown in Algorithm 3.1 and Figure 3.1 with notations summarized in Table 3.2.

[2]In the Euclidean plane, a bound of $(1 + \epsilon, O(\sqrt{\frac{1}{\epsilon}}))$ is achievable and tight for Steiner SLTs [111, 112].

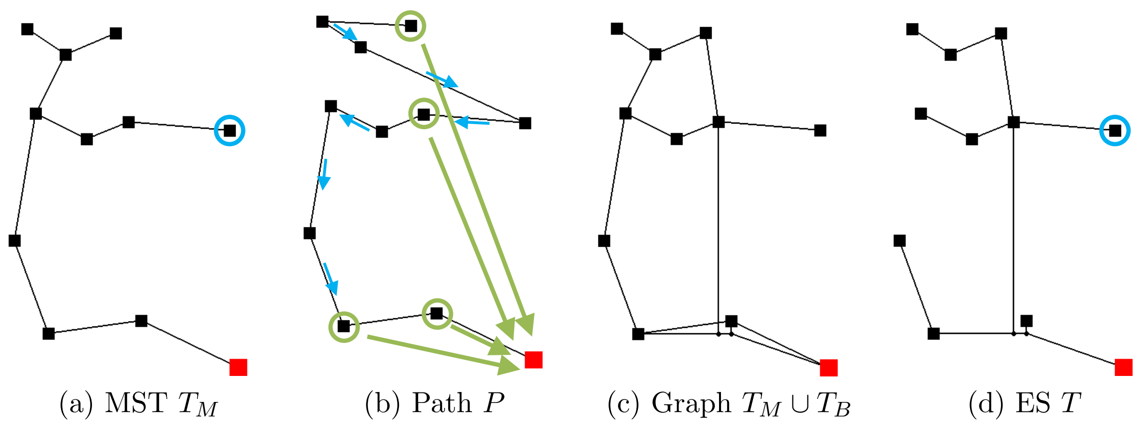


Figure 3.1: Sample run of ES ($\epsilon = 1$). (a) Construct MST T_M (shallowness $\alpha = 3.14$, lightness $\beta = 1$). (b) Identify breakpoints B (circled by green) on the Hamiltonian path P, where each blue arrow points from a non-breakpoint v to its previous vertex for accumulating distance $d_P(b, v)$. (c) Obtain the Steiner SPT T_B on $G[B \cup \{r\}]$, and get graph $T_M \cup T_B$. (d) Construct the spanning SPT on $T_M \cup T_B$, which is the desired Steiner SLT T ($\alpha = 1.90, \beta = 1.06$).

Its main idea is to accumulate the distance along a Hamiltonian path P and identify a breakpoint b whenever the accumulated distance becomes too long. Breakpoints are then connected to the root r directly by a Steiner SPT (line 11). In this way, the distance $d_T(r, b)$ between a breakpoint b and r in the tree T becomes the shortest-path distance $d_G(r, b)$. For other vertices, the path length is bounded.

The Steiner SPT for connecting breaking points is a dedicated design (refer to Section 2 of [40] for details). Applying it to a graph G' leads to the lightness bound $\bar{\beta} = 1+2\lceil \log n \rceil$. The algorithm starts by building a skeleton of a full balanced binary tree, of which the leaves are the original vertices and the inner nodes are Steiner points. From bottom to top, the edge weights are assigned carefully, to make sure the tree will be a SPT that dominates G.

ES is not complicated, but surprisingly, it leads to an exponentially lighter SLT than ABP. Besides, it is reasonably fast. The exact bounds are shown by Theorem 3.1.

Theorem 3.1. *The ES algorithm generates a Steiner* $(1 + 2\epsilon, 4 + 2\lceil \log \frac{2}{\epsilon} \rceil)$*-SLT in* $O(n^2)$ *time.*

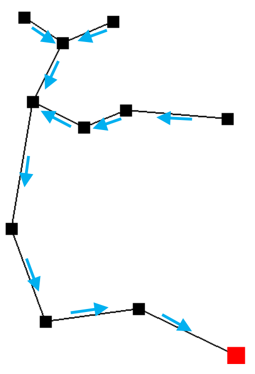
Proof. See Lemmas 3.4, 3.5 and 3.6 of [40]. □

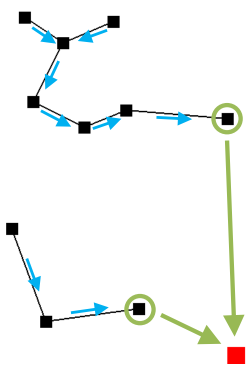
3.1.2 Framework

SALT first identifies some breakpoints on an initial topology and then connect them to the root by a Steiner SPT, which is similar to ES. Inspired by the KRY algorithm [37], we propose to use (i) a tighter criterion for identifying breakpoints and (ii) a better initial topology (i.e., an MST instead of a Hamiltonian path) in the Steiner SLT construction.

Table 3.3: Additional Notations Used in SALT

$p[v]$	Parent of vertex v
$d[v]$	Current distance estimate from r to vertex v


(a) MST T_M


(b) Forest F

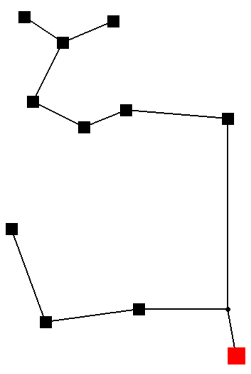

(c) SALT T

Figure 3.2: Sample run of SALT ($\epsilon = 1$). (a) Construct MST T_M, where each blue arrow points from a vertex v to its parent $p[v]$. (b) Update $p[v]$ and identify breakpoints B (circled by green) during the DFS on T_M, which results to a forest F with tree roots being B. (c) Obtain the Steiner SPT T_B on $G[B \cup \{r\}]$, and $T = F \cup T_B$ is the final Steiner SLT (shallowness $\alpha = 1.43$, lightness $\beta = 1.05$).

The framework with the two effective techniques is illustrated by Algorithm 3.2 and Figure 3.2. As a subroutine, the light Steiner SPT construction method will be described by Algorithm 3.3 in the next subsection.

In SALT, the solution is initialized to an MST and gradually modified towards a Steiner SLT. The major routine is based on a depth-first search on the MST (function `DFS`). During DFS, if the shallowness constraint is violated at a vertex, the vertex will become a breakpoint (line 9). In the end, breakpoints will be connected to r via a SPT, so its distance estimate $d[v]$ is set to the shortest-path distance $d_G(r, v)$ for relaxing the distance estimates of the other vertices (line 10). Two relaxations are conducted on each edge, from parent to child and from child to parent (lines 13 and 15). After DFS, edges $(v, p[v])$ for non-breakpoints v define a forest F, with tree roots being breakpoints. In the end, breakpoints are connected to r by a Steiner SPT T_B.

The relaxation (function `Relax`) from vertex u to v means updating distance estimate $d[v]$ if the path from r via u to v is shorter (line 19). Different from KRY, we also update the parent $p[v]$ of v even if $d[v]$ can not be shortened but its edge to the parent can becomes shorter (line 22). The latter situation actually frequently happens in Manhattan space and benefits the tree weight.

The two techniques mentioned above are detailed here. First, breakpoints are identified by checking distance estimate $d[v]$ instead of the accumulated distance $d_P(b, v)$ on the Hamiltonian cycle (in Algorithm 3.1 line 6). As a straight-forward modification, $d[v]$ can

Algorithm 3.2 SALT

```
Require: Graph $G = (V, E, w)$, root $r$, trade-off parameter $\epsilon$;
Ensure: Steiner SLT $T = (V', E', w')$ with $V' \supseteq V$ that dominates $G$;
 1: Initialize ($B \leftarrow \emptyset, d[r] = 0, \forall v \in V, d[v] = +\infty, p[v] = null$);
 2: $T_M \leftarrow MST(G)$;
 3: DFS($r, T_M$);
 4: Forest $F \leftarrow \{(v, p[v]) | v \in V \backslash (B \cup \{r\})\}$;
 5: $T_B \leftarrow$ Steiner SPT rooted at $r$ for $G[B \cup \{r\}]$ by Algorithm 3.3;
 6: $T \leftarrow F \cup T_B$;
 7: function DFS($v, T_M$)
 8:     if $d[v] > (1 + \epsilon) \cdot d_G(r, v)$ then
 9:         $B \leftarrow B \cup \{v\}$;
10:         $d[v] \leftarrow d_G(r, v)$;
11:     end if
12:     for each child $u$ of $v$ in $T_M$ do
13:         Relax($v, u$);
14:         DFS($u, T_M$);
15:         Relax($u, v$);
16:     end for
17: end function
18: function Relax($u, v$)
19:     if $d[v] > d[u] + w(uv)$ then
20:         $d[v] \leftarrow d[u] + w(uv)$;
21:         $p[v] \leftarrow u$;
22:     else if $d[v] = d[u] + w(uv)$ and $w(p[v]v) < w(uv)$ then
23:         $p[v] \leftarrow u$;
24:     end if
25: end function
```

be the sum of the shortest-path length $d_G(r, b)$ (from r to the previous breakpoint b) and the path length $d_P(b, v)$ (from b to v), which is an upper bound on $d_T(r, v)$ in the final T. More specifically, we can change the condition $d_P(b, v) > \epsilon \cdot d_G(r, v)$ to $d_G(r, b) + d_P(b, v) > (1 + \epsilon) \cdot d_G(r, v)$. Note that the value of $d[v]$ in Algorithm 3.2 is computed correctly by the relaxation steps before and after each recursive call (lines 13 and 15). Second, the initial topology is an MST instead of a Hamiltonian path. In this way, the distance estimate $d[v]$ is according to the MST, which is tighter than $d[v] = d_G(r, b) + d_P(b, v)$ based on the Hamiltonian path and can trigger fewer breakpoints. Note that in extreme cases, the second technique brings no benefit (e.g., MST is also a Hamiltonian path), but it does help in most practical cases.

3.1.3 Light Steiner Shortest-Path Tree

A light Steiner SPT can be constructed by Algorithm 3.3, which has smaller tree weight than that in the ES algorithm. Notations used are in Table 3.4 and Figure 3.3.

Same as the Steiner SPT in ES, our SPT is also a full balanced binary tree, with leaves being the given vertices and inner nodes being Steiner vertices. Initially, the vertex

Table 3.4: Additional Notations Used in Light Steiner SPT

$T(z)$	Subtree rooted at vertex z
$Leaves(z)$	Set of leaf vertices in T_z
$t(z)$	Distance from root r to vertex z in the SPT
$b(z_l, z_r)$	Disbalance between vertices z_l and z_r
$s(z_l, z_r)$	Distance surplus between vertices z_l and z_r
$c(z_l, z_r)$	Edge cost between vertices z_l and z_r
L	Vertex sequence
L_i	i-th vertex of L
$\|L\|$	Vertex number in $\|L\|$
v_i	i-th vertex along the traveling salesman circle
$f(z)$	First index of $Leaves(z) = \{v_{f(z)}, v_{f(z)+1}, ..., v_{l(z)}\}$
$l(z)$	Last index of $Leaves(z) = \{v_{f(z)}, v_{f(z)+1}, ..., v_{l(z)}\}$
$W(i,j)$	Length of path $(v_i, v_{i+1}, ..., v_j)$: $\sum_{k=i}^{j-1} d_G(v_k, v_{k+1})$
W_i'	Total weight of edges added in the i-th iteration

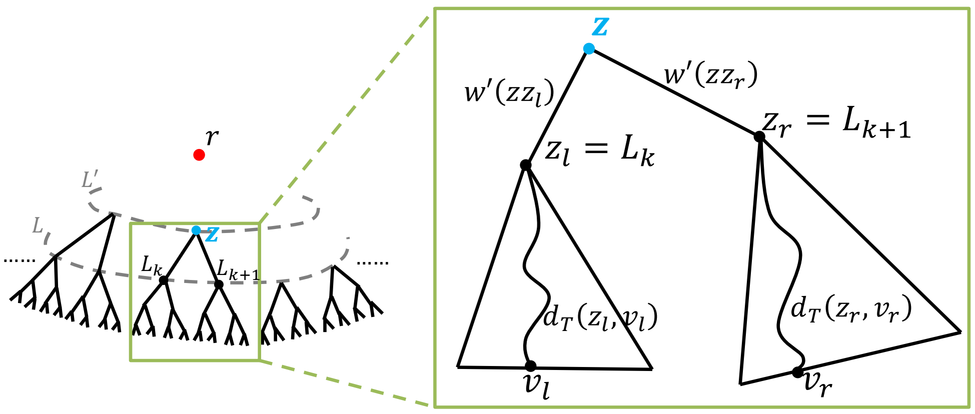


Figure 3.3: During Steiner SPT construction, neighboring vertices in L are merged pair by pair into Steiner vertices in L'. To be more specific, vertices L_k (i.e., z_l) and L_{k+1} (i.e., z_r) are merged to a Steiner vertex z in L'.

sequence L contains all the given vertices. In each iteration of the main loop (line 4–16), neighboring vertices are merged (i.e., connected to a parent Steiner vertex) pair by pair to form the vertex sequence L' for the next iteration. Note that the vertex number is reduced by half in each iteration and eventually becomes one.

When a Steiner vertex z is inserted as the parent for vertices z_l and z_r, the edge weights are assigned under the consideration of disbalance b and distance surplus s:

$$b(z_l, z_r) = t(z_l) - t(z_r), \tag{3.1}$$

$$\begin{aligned} s(z_l, z_r) = \; & \max\{d_G(v_l, v_r) - d_T(z_l, v_l) - d_T(z_r, v_r) \\ & |v_l \in Leaves(z_l), v_r \in Leaves(z_r)\}, \end{aligned} \tag{3.2}$$

where $t(z)$ is the distance from root r to vertex z in the final SPT. It is obvious that

Algorithm 3.3 Light Steiner SPT

```
Require: Graph $G = (V, E, w)$, root $r$;
Ensure: Steiner SPT $T = (V', E', w')$ with $V' \supseteq V$ that dominates $G$;
 1: Initialize $(V' \leftarrow V, E' \leftarrow \emptyset, \forall v \in V, t(v) \leftarrow d_G(r, v))$;
 2: $L \leftarrow$ Hamiltonian circle based on $MST(G)$ $(L_{n+1} = L_1)$;
 3: while $|L| > 1$ do
 4:     for $k = 1$ to $n$ do
 5:         Calculate $b(L_k, L_{k+1}), s(L_k, L_{k+1})$ by (3.1) (3.2);
 6:         $c(L_k, L_{k+1}) \leftarrow \max\{s(L_k, L_{k+1}), |b(L_k, L_{k+1})|\}$;
 7:     end for
 8:     $M_L \leftarrow$ a light perfect (or near perfect) matching on the circle defined by $L$ and $c$;
 9:     $L' \leftarrow$ empty vertex sequence;
10:     for $L_k L_{k+1} \in M_L$ do
11:         AddSteiner$(L_k, L_{k+1})$;
12:     end for
13:     if $|L|$ is odd then
14:         Append the unmatched vertex to $L'$;
15:     end if
16:     $L \leftarrow L'$;
17: end while
18: function AddSteiner$(z_l, z_r)$
19:     Add a Steiner vertex $z$ into $V'$;
20:     Add edges $zz_l$ and $zz_r$ into $E'$;
21:     if $|b(z_l, z_r)| \le s(z_l, z_r)$ then
22:         $w'(zz_l) \leftarrow \frac{s(z_l,z_r)+b(z_l,z_r)}{2}$;
23:         $w'(zz_r) \leftarrow \frac{s(z_l,z_r)-b(z_l,z_r)}{2}$;
24:     else
25:         $w'(zz_l) \leftarrow \max\{b(z_l, z_r), 0\}$;
26:         $w'(zz_r) \leftarrow \max\{-b(z_l, z_r), 0\}$;
27:     end if
28:     $t(z) \leftarrow d_G(r, v) - d_T(z, v)$ for an arbitrary $v \in Leaves(z)$;
29:     Append $z$ to $L'$;
30: end function
```

$t(z) = d_G(r, z)$ if z is a leaf. t and b help maintain T to be a SPT and require the choice of edge weights $w'(zz_l)$ and $w'(zz_r)$ to satisfy:

$$w'(zz_l) - w'(zz_r) = b(z_l, z_r). \tag{3.3}$$

In this way, $t(z_l) = t(z) + w'(zz_l)$ and $t(z_r) = t(z) + w'(zz_r)$ can be true at the same time. For distance surplus s, $w'(zz_l)$ and $w'(zz_r)$ should satisfy:

$$w'(zz_l) + w'(zz_r) \ge s(z_l, z_r). \tag{3.4}$$

This guarantees $d_G(v_l, v_r) \le d_T(z_l, v_l) + w'(zz_l) + w'(zz_r) + d_T(z_r, v_r) = d_T(v_l, v_r)$ (i.e., T dominates G). Algorithmic details are in function `AddSteiner`. Note that in line 28, arbitrary $v \in Leaves(z)$ can be picked to calculate $t(z)$ due to the following lemma.

Lemma 3.1. *In Algorithm 3.3, for any vertex z in T and any vertex $v \in$ Leaves(z), $d_G(r,v) - d_T(z,v)$ is a constant.*

Proof. See Lemma 2.2 of [40]. □

Unlike the ES algorithm, which first determines the full-tree topology based on a Hamiltonian path and then assigns weight to the edges, our algorithm calculates the edge cost $c(L_k, L_{k+1})$ along L at each level and selects a good matching M_L to add Steiner vertices. According to the function `AddSteiner`, if a Steiner point z is inserted, the sum $c(z_l, z_r)$ of the weights of the two edges added will be

$$c(z_l, z_r) = w'(zz_l) + w'(zz_r) = \max\{|b(z_l, z_r)|, s(z_l, z_r)\}. \tag{3.5}$$

Since a cycle of even (resp. odd) number of edges can be decomposed into two perfect (resp. near perfect) matching, the weight of the lighter one will be no more than half of the cycle weight. In this way, the sum of the weights of the added edges is bounded.

Another technique that we use is to include the root r into the initial Hamiltonian circle. In this way, an edge between the final Steiner point and r is avoided and saved.

The resulted tree T is a SPT, of which the proof is simple and is similar to that in [40].

3.1.4 Bound Analysis

We first analyze the lightness β of the Steiner SPT generated by Algorithm 3.3.

Lemma 3.2. *In Algorithm 3.3, for any vertex z in T, there exist $v_i, v_j \in$ Leaves(z), such that $d_T(z, v_i) + d_T(z, v_j) = d_G(v_i, v_j)$.*

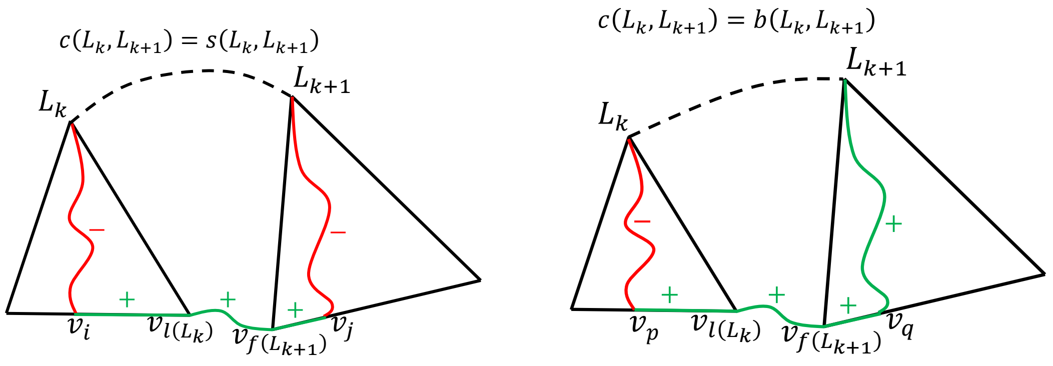
Proof. The proof is by induction. If z is a leaf, it is trivial by making $v_i = v_j = z$. We then assume that the statement holds for the two children z_l and z_r of z, and prove it for z.

Suppose first that $|b(z_l, z_r)| \leq s(z_l, z_r)$, i.e., $w'(zz_l) + w'(zz_r) = s(z_l, z_r)$. Let $v_i \in$ *Leaves*(z_l) and $v_j \in$ *Leaves*(z_r) be two vertices that achieve $s(z_l, z_r) = d_G(v_i, v_j) - (d_T(z_l, v_i) + d_T(z_r, v_j))$. Therefore,

$$\begin{aligned} & d_T(z, v_i) + d_T(z, v_j) \\ = \; & w'(zz_l) + d_T(z_l, v_i) + w'(zz_r) + d_T(z_r, v_j) \\ = \; & s(z_l, z_r) + d_T(z_l, v_i) + d_T(z_r, v_j) \\ = \; & d_G(v_i, v_j). \end{aligned} \tag{3.6}$$

Otherwise, $|b(z_l, z_r)| > s(z_l, z_r)$. Suppose w.l.o.g. that $w'(zz_l) = 0$. By the induction hypothesis, there are $v_i, v_j \in$ *Leaves*(z_l) such that $d_T(z_l, v_i) + d_T(z_l, v_j) = d_G(v_i, v_j)$. Hence,

$$d_T(z, v_i) + d_T(z, v_j) = d_T(z_l, v_i) + d_T(z_l, v_j) = d_G(v_i, v_j). \tag{3.7}$$



(a) Balanced case (b) Unbalanced case

Figure 3.4: Decomposed edge cost $c(L_k, L_{k+1})$.

Note that $v_i, v_j \in Leaves(z_l) \subset Leaves(z)$. □

The next lemma is the key to our weight analysis, which shows that the weight of the circle defined by L and c is bounded by the weight of the initial Hamiltonian cycle $W(1, n+1) = \sum_{k=1}^{n} d_G(v_k, v_{k+1})$.

Lemma 3.3. *For the vertex sequence L in any iteration of Algorithm 3.3, $\sum_{k=1}^{|L|-1} c(L_k, L_{k+1}) \leq W(1, n+1)$.*

Proof. We start by decomposing $c(L_k, L_{k+1})$. There are two cases, as Figure 3.4 shows. First, suppose $c(L_k, L_{k+1}) = s(L_k, L_{k+1})$. Let $v_i \in Leaves(L_k)$ and $v_j \in Leaves(L_{k+1})$ be two vertices that achieve $s(L_k, L_{k+1})$. Therefore,

$$\begin{aligned}
& c(L_k, L_{k+1}) = s(L_k, L_{k+1}) \\
= \ & d_G(v_i, v_j) - (d_T(L_k, v_i) + d_T(L_{k+1}, v_j)) \\
\leq \ & \underbrace{d_G(v_i, v_{l(L_k)}) - d_T(L_k, v_i)}_{within\ T(L_k)} + \underbrace{W(l(L_k), f(L_{k+1}))}_{between\ T(L_k),\ T(L_{k+1})} \\
& + \underbrace{d_G(v_{f(L_{k+1})}, v_j) - d_T(L_{k+1}, v_j)}_{within\ T(L_{k+1})},
\end{aligned} \tag{3.8}$$

where the last inequality holds due to triangle inequality.

Second, $c(L_k, L_{k+1}) = |b(L_k, L_{k+1})|$. If $b(L_k, L_{k+1}) \geq 0$, by Lemma 3.1, $\forall v_p \in$

$Leaves(L_k), \forall v_q \in Leaves(L_{k+1})$,

$$\begin{aligned} & c(L_k, L_{k+1}) = b(L_k, L_{k+1}) = t(L_k) - t(L_{k+1}) \\ = \;& (d_G(r, v_p) - d_T(L_k, v_p)) - (d_G(r, v_q) - d_T(L_{k+1}, v_q)) \\ \le \;& d_G(v_p, v_q) - d_T(L_k, v_p) + d_T(L_{k+1}, v_q) \\ \le \;& \underbrace{d_G(v_p, v_{l(L_k)}) - d_T(L_k, v_p)}_{within\ T(L_k)} + \underbrace{W(l(L_k), f(L_{k+1}))}_{between\ T(L_k),\ T(L_{k+1})} \\ & + \underbrace{d_G(v_{f(L_{k+1})}, v_q) + d_T(L_{k+1}, v_q)}_{within\ T(L_{k+1})}. \end{aligned} \tag{3.9}$$

If $b(L_k, L_{k+1}) < 0$, the result is symmetric. Therefore, the part decomposed from $c(L_k, L_{k+1})$ into $T(L_k)$ is

$$C_r(L_k) = \begin{cases} d_G(v_i, v_{l(L_k)}) - d_T(L_k, v_i), & c(L_k, L_{k+1}) = s(L_k, L_{k+1}), \\ d_G(v_p, v_{l(L_k)}) - d_T(L_k, v_p), & c(L_k, L_{k+1}) = b(L_k, L_{k+1}), \\ d_G(v_q, v_{l(L_k)}) + d_T(L_k, v_q), & c(L_k, L_{k+1}) = -b(L_k, L_{k+1}), \end{cases} \tag{3.10}$$

where indices i is fixed while p, q are flexible. Meanwhile, there is $C_l(L_k)$, which is decomposed from $c(L_{k-1}, L_k)$ and can be calculated similarly. The weight sum within $T(L_k)$ is then $C(L_k) = C_l(L_k) + C_r(L_k)$.

We will prove $C(L_k) \le W(f(L_k), l(L_k))$, which has three cases.

Case 1: $C_l(L_k)$ and $C_r(L_k)$ both contain minus. Then $C(L_k) = d_G(v_{f(L_k)}, v_j) - d_T(L_k, v_j) + d_G(v_i, v_{l(L_k)}) - d_T(L_k, v_i)$. When $j \le i$, it is obvious. Otherwise, since $d_T(L_k, v_j) + d_T(L_k, v_i) \ge d_T(v_i, v_j) \ge d_G(v_i, v_j)$,

$$\begin{aligned} C(L_k) &\le d_G(v_{f(L_k)}, v_j) + d_G(v_i, v_{l(L_k)}) - d_G(v_i, v_j) \\ &\le d_G(v_{f(L_k)}, v_i) + d_G(v_i, v_j)) + d_G(v_i, v_{l(L_k)}) \\ &\le W(f(L_k), l(L_k)). \end{aligned} \tag{3.11}$$

Case 2: only one of $C_l(L_k)$ and $C_r(L_k)$ contains minus. Suppose w.l.o.g. that $C_l(L_k)$ does, then $C(L_k) = d_G(v_{f(L_k)}, v_q) + d_T(L_k, v_q) + d_G(v_i, v_{l(L_k)}) - d_T(L_k, v_i)$. By setting $q = i$,

$$C(L_k) = d_G(v_{f(L_k)}, v_i) + d_G(v_i, v_{l(L_k)}) \le W(f(L_k), l(L_k)). \tag{3.12}$$

Case 3: neither of $C_l(L_k)$ and $C_r(L_k)$ contains minus. That is, $C(L_k) = d_G(v_{f(L_k)}, v_q) + d_T(L_k, v_q) + d_G(v_p, v_{l(L_k)}) + d_T(L_k, v_p)$. By Lemma 3.2, there exist $f(L_k) \le q \le p \le l(L_k)$

such that

$$\begin{aligned} C(L_k) &= d_G(v_{f(L_k)}, v_q) + d_G(v_q, v_p) + d_G(v_p, v_{l(L_k)}) \\ &\leq W(f(L_k), l(L_k)). \end{aligned} \tag{3.13}$$

By (3.8), (3.9) and $C(L_k) \leq W(f(L_k), l(L_k))$, the proof is done. □

Lemma 3.4. *In the i-th iteration of Algorithm 3.3, the total weight of added edges $W'_i \leq w(MST(G))$.*

Proof. Due to the perfect (or near perfect) matching used and Lemma 3.3, $W'_i \leq \frac{1}{2} \cdot \sum_{k=1}^{|L|-1} c(L_k, L_{k+1}) \leq \frac{1}{2} \cdot W(1, n+1)$. Because of triangle inequality, $W(1, n+1) \leq 2 \cdot w(MST(G))$. By combining them, $W'_i \leq w(MST(G))$. □

With the help of Lamma 3.4, the lightness bound of Algorithm 3.3 can be easily proved to be $\bar{\beta} = \lceil \log n \rceil$. Note that the Steiner SPT in ES has $\bar{\beta} = 1 + 2\lceil \log n \rceil$, which is more than twice of ours.

Theorem 3.2. *The Steiner SPT T generated by Algorithm 3.3 has lightness bound $\bar{\beta} = \lceil \log n \rceil$.*

Proof. With $\lceil \log n \rceil$ iterations, $|L|$ can be reduced from n to 1. Therefore, $w(T) = \sum_{i=1}^{\lceil \log n \rceil} W'_i \leq \lceil \log n \rceil \cdot w(MST(G))$. □

We then analyze the bounds on shallowness α and lightness β of SALT. Two lemmas are first needed.

Lemma 3.5. *In Algorithm 3.3, if $\sum_{v \in V \setminus \{r\}} d_G(r, v) \leq \theta \cdot \eta$ $(\theta \geq 1, \eta > 0)$, then $w(T) \leq \lceil \log \theta \rceil \cdot w(MST(G)) + \eta$.*

Proof. First, n is assumed to be the power of 2. Indeed, we can duplicate r into $2^{\lceil \log n \rceil} - n$ new vertices if it is not. Besides, if $\lceil \log \theta \rceil \geq \log n$, it is trivial by Theorem 3.2. Hence, we assume that $\lceil \log \theta \rceil < \log n$.

Let $E'_i \subseteq E'$ denote the set of edges added during the i-th iteration $(1 \leq i \leq \log n)$, W'_i denote $\sum_{e \in E'_i} w'(e)$, *Leaves*(e) denote the set of leaf vertices in the downstream from an edge e. Since T is SPT and $|Leaves(e)| = 2^{i-1}$ for $e \in E'_i$,

$$\begin{aligned} &\sum_{v \in V \setminus \{r\}} d_G(r, v) = \sum_{v \in V \setminus \{r\}} d_T(r, v) = \sum_{i=1}^{\log n} \sum_{e \in E_i} |Leaves(e)| w'(e) \\ =&\sum_{i=1}^{\log n} 2^{i-1} \cdot W'_i \geq \sum_{i=\lceil \log \theta \rceil + 1}^{\log n} 2^{i-1} \cdot W'_i \geq \theta \sum_{i=\lceil \log \theta \rceil + 1}^{\log n} W'_i. \end{aligned} \tag{3.14}$$

Therefore, $\sum_{i=\lceil \log \theta \rceil + 1}^{\log n} W_i' \leq \eta$ since $\sum_{v \in V \setminus \{r\}} d_G(r, v) \leq \theta \cdot \eta$. Together with Lemma 3.4,

$$\begin{aligned} w(T) = \sum_{i=1}^{\log n} W_i' = \sum_{i=1}^{\lceil \log \theta \rceil} W_i' + \sum_{i=\lceil \log \theta \rceil + 1}^{\log n} W_i' \\ \leq \lceil \log \theta \rceil \cdot w(MST(G)) + \eta. \end{aligned} \tag{3.15}$$

□

Lemma 3.6. *In SALT,* $\sum_{v \in B} d_G(r, v) \leq \frac{2}{\epsilon} \cdot w(MST(G))$.

Proof. See Lemma 3.2 of [37]. □

According to Lemma 3.6, KRY, which connects breakpoints to root r by edges directly, leads to a spanning $(1+\epsilon, 1+\frac{2}{\epsilon})$-SLT. Introducing Steiner points by Algorithm 3.3 makes the bound tighter.

Theorem 3.3. *SALT generates a Steiner* $(1+\epsilon, 2+\lceil \log \frac{2}{\epsilon} \rceil)$*-SLT.*

Proof. Whenever $d[v]$ of a vertex v exceeds $(1+\epsilon)$ times its shortest-path length $d_G(r, v)$, $d[v]$ is set to $d_G(r, v)$ and fixed. Therefore, we have shallowness $\alpha \leq 1+\epsilon$.

Since T_B is a Steiner SPT on graph $G[B \cup \{r\}]$, substituting $\theta = \frac{2}{\epsilon}$ and $\eta = w(MST(G))$ (by Lemma 3.6) into Lemma 3.5 makes $w(T_B) \leq (1 + \lceil \log \frac{2}{\epsilon} \rceil) \cdot w(MST(G))$. Besides, $w(F) \leq w(MST(G))$ because $F \subset MST(G)$. Hence, $w(T) = w(T_B) + w(MST(G)) \leq (2 + \lceil \log \frac{2}{\epsilon} \rceil) \cdot w(MST(G))$. □

3.2 Rectilinear Steiner Shallow-light Tree Algorithm

SALT, which generates a Steiner $(1+\epsilon, 2+\lceil \log \frac{2}{\epsilon} \rceil)$-SLT for a general graph, can be directly applied in the Manhattan space. However, it can be enhanced with the help of some special properties as well as classical algorithms. The resulted algorithm, rectilinear SALT, is shown by Algorithm 3.4 and Figure 3.5. W.l.o.g., we assume that the root r is at the origin of the space.

First of all, to build a rectilinear Steiner SPT, adding a Steiner point to merge two vertices (function `AddSteiner` in Algorithm 3.3) becomes easier on Manhattan plane. In the following discussion, we focus on the two-dimensional situation, but it can be extended to higher dimensions. For two vertices $z_l = (x_{z_l}, y_{z_l})$ and $z_r = (x_{z_r}, y_{z_r})$, the **x** coordinate of their parent Steiner point z is

$$x_z = \begin{cases} \min\{x_{z_l}, x_{z_r}\}, & x_{z_l}, x_{z_r} \geq 0, \\ \max\{x_{z_l}, x_{z_r}\}, & x_{z_l}, x_{z_r} \leq 0, \\ 0, & x_{z_l} \cdot x_{z_r} < 0. \end{cases} \tag{3.16}$$

Algorithm 3.4 Rectilinear SALT

```
Require: Points $V$ on Manhattan plane, root $r$;
Ensure: Rectilinear Steiner SLT $T = (V', E')$ with $V' \supseteq V$;
 1: Initialize $(B \leftarrow \emptyset, d[r] = 0, \forall v \in V, d[v] = +\infty, p[v] = null)$;
 2: $T_M \leftarrow$ RSMT on $V$ by FLUTE ;
 3: DFS$(r, T_M)$;
 4: Forest $F \leftarrow \{(v, p[v]) | v \in V \backslash (B \cup \{r\})\}$;
 5: $T_B \leftarrow$ RSMA rooted at $r$ on $B \cup \{r\}$ by CL ;
 6: $T \leftarrow F \cup T_B$;
 7: function DFS$(v, T_M)$
 8:     if $v \in V$ and $d[v] > (1 + \epsilon) \cdot d_G(r, v)$ then
 9:         $B \leftarrow B \cup \{v\}$;
10:         $d[v] \leftarrow d_G(r, v)$;
11:     end if
12:     for each child $u$ of $v$ in $T_M$ do
13:         Relax$(v, u)$;
14:         DFS$(u, T_M)$;
15:         Relax$(u, v)$;
16:     end for
17: end function
```

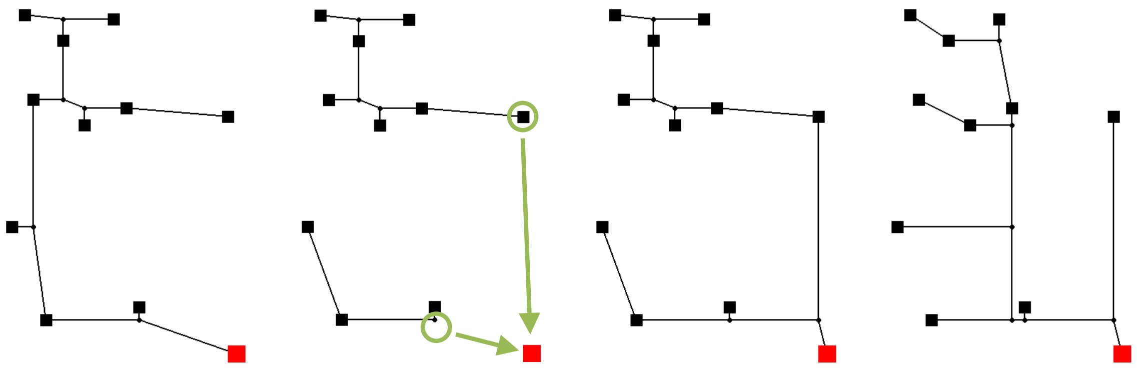

(a) RSMT T_M by FLUTE (b) Forest F (c) Rectilinear SALT T (d) RSMA by CL

Figure 3.5: Sample run of rectilinear SALT ($\epsilon = 1$). (a) Construct RSMT T_M by FLUTE (shallowness $\alpha = 2.66$, lightness $\beta = 0.91$). (b) Get breakpoints B (circled by green) and forest F. (c) Obtain the RSMA T_B on $G[B \cup \{r\}]$ by CL, and $T = F \cup T_B$ is the rectilinear Steiner SLT ($\alpha = 1.22, \beta = 1.01$). (d) RSMA by CL on the net ($\alpha = 1, \beta = 1.11$).

y_z is computed similarly. This location assignment of z is determined by distances $w'(zz_l)$, $w'(zz_r)$ and $t(z)$. Note that the case $|b(z_l, z_r)| > s(z_l, z_r)$ (Algorithm 3.3 lines 24–26) never happens now. Intuitively, such Steiner point z maximizes the overlapping of the two shortest paths from r to vertices z_l and z_r. In this way, the coordinate of z can be directly obtained from locations of z_l and z_r, which avoids the checking of all leaves of z_l and z_r in (3.2). Therefore, the time complexity is now bounded by obtaining the MST and is improved to $O(n \log n)$.

Second, the Steiner SPT problem in Manhattan space is exactly the classical RSMA problem [27,30]. The CL heuristics [31] is an approximation algorithm produces a tree of weight at most twice the optimal. In practice, it is mostly optimal or near optimal, and is very efficient with a time complexity of $O(n \log n)$. On the other hand, our light Steiner SPT algorithm with lightness $\beta \leq \lceil \log \frac{2}{\epsilon} \rceil$ may be far away from the optimal SPT in worst cases. For example, when all vertices locate on a straight line, the optimal SPT is a path and also the MST (i.e., $\beta = 1$). Hence, we use CL to construct the Steiner SPT to further reduce the tree weight in practice (Algorithm 3.4 line 5). Note that this modification maintain the proved complexity for both the quality (shallowness α and lightness β) and time of Algorithm 3.2. While the constant in the shallowness bound ($\bar{\alpha} = 1 + \epsilon$) is also maintained, the constant in the lightness bound ($\bar{\beta} = 2 + \lceil \log \frac{2}{\epsilon} \rceil$) may be slightly worsened in some corner cases but is better or much better in most cases.

Third, instead of starting from an MST in Algorithm 3.2, an initial tree with lighter weight is achievable by allowing Steiner points. In Manhattan space, RSMT is a well-investigated problem, and FLUTE [25] is adopted in our implementation (Algorithm 3.4 line 2). In this way, the bound on the tree weight $w(T)$ actually becomes tighter. There is still $w(T) \leq (2 + \lceil \log \frac{2}{\epsilon} \rceil) \cdot w(T_M)$, where T_M is MST in Theorem 3.3 but now becomes RSMT. Note that different from Algorithm 3.2, the Steiner vertices in the RSMT do not need to be checked during the DFS (Algorithm 3.4 line 8).

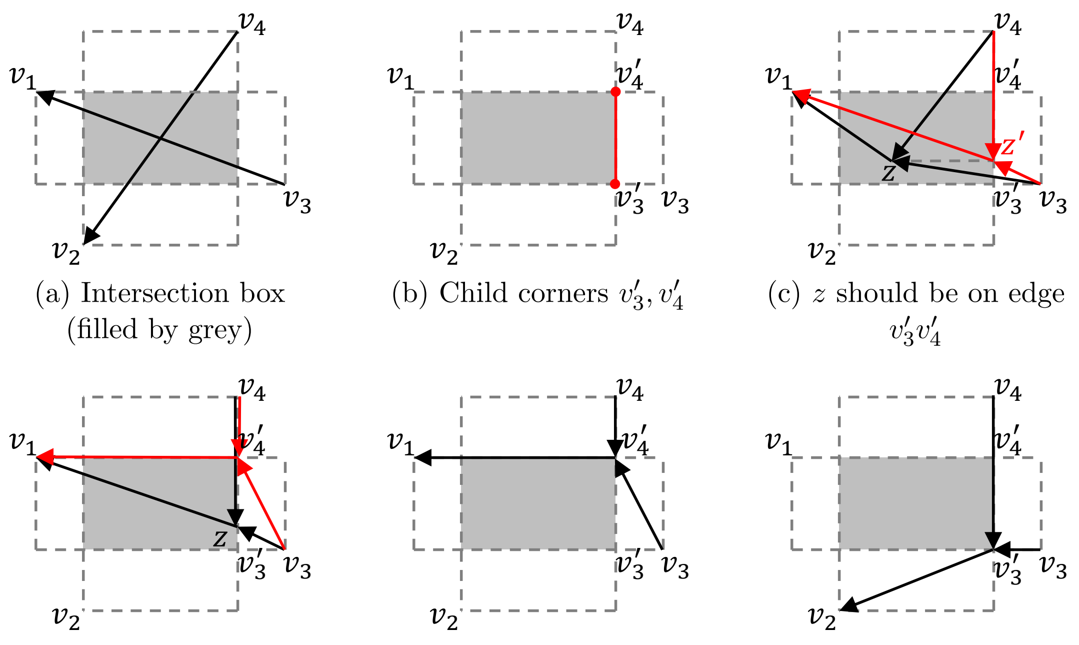
By the above modifications, we reduce the lightness β of the Steiner SLT constructed and improve the time complexity to $O(n \log n)$. From another viewpoint, rectilinear SALT is a smooth trade-off between RSMA and RSMT. The smaller the ϵ, the closer the rectilinear SALT is to a RSMA; the larger the ϵ, the closer it is to a RSMT. It is almost a CL RSMA when $\epsilon = 0$ and an FLUTE RSMT when $\epsilon = +\infty$. In the middle, it is a bounded trade-off between them. To a certain extent, Figure 3.5 illustrates the situation. The RSMT in Figure 3.5(a) is the lightest but has some long paths, while the RSMA in Figure 3.5(d) is the shallowest but is of a large tree weight. Combining the strengths of the both, the rectilinear SALT in Figure 3.5(c) is not only light but also shallow.

3.3 Safe Refinement

Three effective safe refinement techniques are adopted to further improve rectilinear SALT, including intersected edge canceling, L-shape edge flipping, and U-shape edge shifting. They are safe as they improve wirelength or path length or both without worsening any of them. For simplicity, rectilinear SALT will be referred as SALT hereafter.

3.3.1 Intersected Edge Canceling

In SALT, edges in RSMA T_B may intersect with edges in forest F, since T_B and F are constructed separately. Here, the *intersection* between two edges in the Manhattan space



(a) Intersection box (filled by grey)

(b) Child corners v_3', v_4'

(c) z should be on edge $v_3'v_4'$

(d) z should be either v_3' or v_4'

(e) First solution

(f) Second solution

Figure 3.6: Intersected edge canceling (arrows point to parents).

means that their bounding boxes intersect, which is illustrated by Figure 3.6(a). For intersected edges v_3v_1 and v_4v_2, we can add a Steiner vertex z within the intersection box, connect *child vertices* v_3 and v_4 to it, and then connect it to either v_1 or v_2. By choosing the shorter path between $(z, v_1, ..., r)$ and $(z, v_2, ..., r)$, both path lengths and wirelength can be reduced. The question is where the best location for the Steiner vertex z is, and it can be answered by Theorem 3.4. Among the four corners of an intersection box, a *child corner* is the closest to a child vertex (e.g., v_3' and v_4' in Figure 3.6(b)).

Theorem 3.4. *For intersected edges, the optimal Steiner vertex z is a child corner of the intersection box.*

Proof. First, z should be on a *child edge* (i.e., the edge between the two child corners). If not, its projected point z' on the child edge can improve the wirelength without impacting path lengths (Figure 3.6(c)). Supposing z is connected to v_1, there is $w(v_3z) + w(v_4z) + w(zv_1) = (w(v_3z') + w(z'z)) + (w(v_4z') + w(z'z)) + (w(z'v_1) - w(z'z)) \geq w(v_3z') + w(v_4z') + w(z'v_1)$.

When z is on the child edge but not a child corner, it can be improved by moving to a child corner (Figure 3.6(d)). Assume z is still connected to v_1. For wirelength, there is $w(v_3z) + w(v_4z) + w(zv_1) \geq w(v_3v_4') + w(v_4v_4') + w(v_4'v_1)$; for path lengths, there is $w(v_4z) + w(zv_1) \geq w(v_4v_4') + w(v_4'v_1)$, while $w(v_3z) + w(zv_1) = w(v_3v_4') + w(v_4'v_1)$.

The argument is similar if z is connected to v_2. In short, the optimal solution is a child corner (either v_3' or v_4') shown by Figures 3.6(e) and 3.6(f). Note that, in some cases, the

Algorithm 3.5 Intersected Edge Canceling

```
Require: Tree $T$;
Ensure: Tree $T'$ without intersected edges;
 1: Queue $Q \leftarrow$ bounding boxes of all edges in $T$;
 2: R-tree $R \leftarrow \emptyset$;
 3: while $Q$ is not empty do
 4:     Box $r \leftarrow$ dequeue $Q$;
 5:     Search for a box $r'$ in $R$ that intersects with $r$;
 6:     if there is such $r'$ then
 7:         Delete $r'$ from $R$;
 8:         Cancel the intersection between $r$ and $r'$;
 9:         Enqueue newly-generated edges to $Q$;
10:     else
11:         Insert $r$ to $R$;
12:     end if
13: end while
```

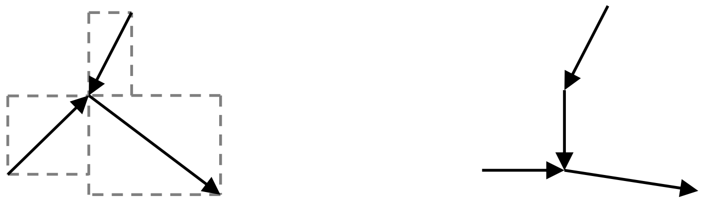

(a) Input (b) Output

Figure 3.7: L-shape edge flipping.

two child corners may merge into one, or the intersection box may even degenerates to a segment, but our discussion is generic. □

For a Steiner tree, we propose an iterative scheme for identifying and canceling all the intersected edges (Algorithm 3.5) based on R-tree [113]. Throughout the process, the major invariant is that the boxes in R-tree R do not intersect with each other. By iteratively examining boxes (lines 3–13), a new box r will be broken or shrank (due to the intersection canceling in Figure 3.6) until all the intersections caused by it has been resolved. Regarding the running time of Algorithm 3.5, it is $O(n \log n)$ thanks to the $O(\log n)$-time query of R-tree and the $O(n)$ edges in total.

3.3.2 L-/Z-Shape Edge Flipping

Edges may be overlapped with each other by flipping (in L or Z shape) and thus improves wirelength and path lengths, as Figs. 3.7 and 3.8 show. Ho et al. [19] propose a dynamic programming for edge flipping. The method is linear-time if the vertex degree is bounded, and generates optimal wirelength if only edge overlapping around a vertex is counted. We apply this techique. In SALT, the maximum vertex degree is the sum of that in FLUTE (four, according [14]) and CL (four, considering the root), as a SALT T is the union

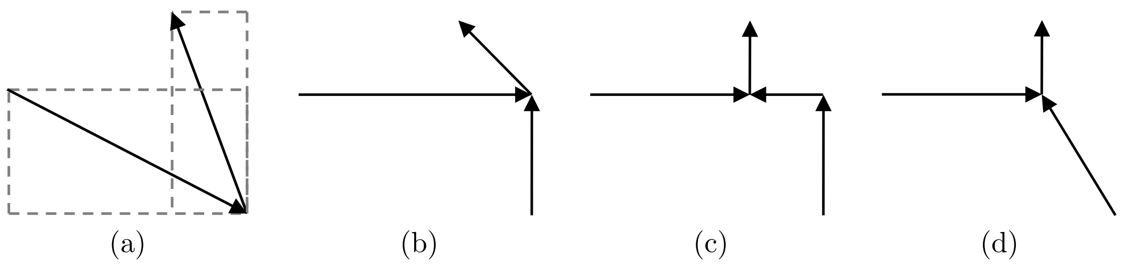


Figure 3.8: Z-shape edge flipping by iterative L-shape flipping. (a) Input. (b) First L-shape flipping. (c) Second L-shape flipping (i.e., a Z-shape flipping). (d) Removing redundant Steiner vertex.

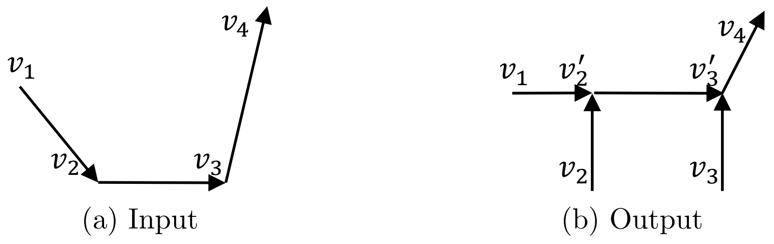


Figure 3.9: (Canonical) U-shape edge shifting.

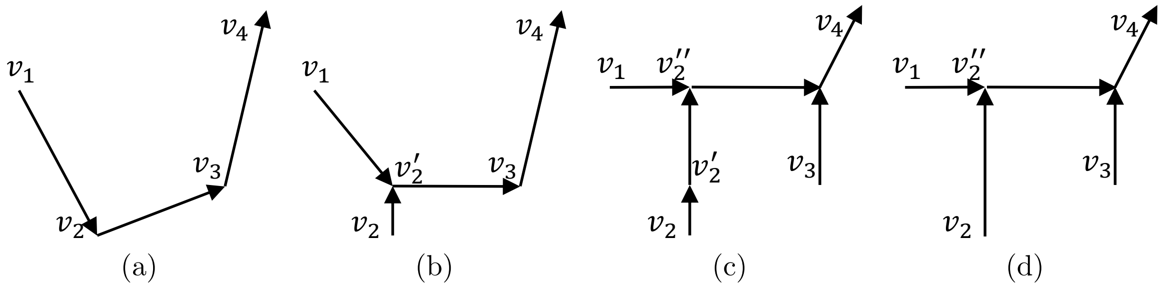


Figure 3.10: General U-shape edge shifting. (a) Input. (b) L-/Z-shape edge flipping. (c) Canonical U-shape edge shifting. (d) Removing redundant Steiner vertex.

of a FLUTE forest F and a CL RSMA T_B. Therefore, the vertex degree is bounded ($\leq 4+4=8$) and guarantees the $O(n)$ time. In our implementation, the optimal L-shape flipping is adopted, since the constant in the time complexity of the optimal Z-shape flipping is quite large. The Z-shape flipping can be achieved by iterative L-shape flipping, which is demonstrated by Figure 3.8.

3.3.3 U-Shape Edge Shifting

The U-shape edge shifting is proposed by Boese et al. [114]. It is beneficial not only to wirelength and path lengths but also to Elmore delay. An example is in Figure 3.9, where the edge v_2v_3 is shifted to $v_2'v_3'$. The U-shape shifting can be performed during a tree traversal. It takes $O(n)$ time due to the bounded vertex degree in SALT.

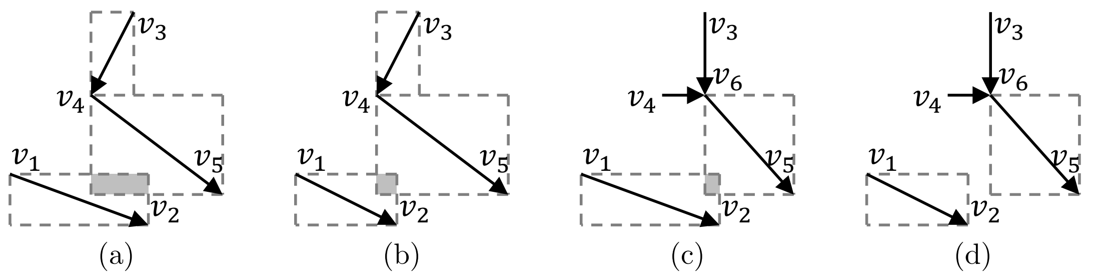


Figure 3.11: L-/Z-shape edge flipping may make the edge intersection shrank or even disappeared. (a) A case where edge v_1v_2 intersects with edge v_3v_4. (b) Another case where edge v_1v_2 intersects with edge v_3v_4. (c) After L-shape edge flipping on (a), the edge intersection shrinks. (d) After L-shape edge flipping on (b), the edge intersection disappears.

3.3.4 Order of Safe Refinement Techniques

In our implementation, the three safe refinement techniques are performed in the following order: (i) intersected edge canceling (IEC), (ii) L-/Z-shape edge flipping (LEF), and (iii) U-shape edge shifting (UES). It is based on two considerations.

First, among the three safe refinement methods, IEC changes topologies more globally and significantly, while the other two methods work on topologically neighbored edges only. Meanwhile, the other two methods may influence the solution space of IEC. For example in Figure 3.11, after LEF, the previous edge intersection may shrink (Figure 3.11(b)) or disappear (Figure 3.11(d)), which means less or even missed improvement.

Second, a general UES can be decomposed into LEF and a canonical UES (Figure 3.10). In a canonical U-shape path, the middle edge (e.g., edge v_2v_3 of path $v_1v_2v_3v_4$ in Figure 3.9) is strictly horizontal or vertical. Therefore, conducting LEF first avoids handling the many corner cases and thus eases the implementation of general UES.

3.4 Shallowness-Constrained Edge Substitution

Compared with safe refinement, shallowness-constrained edge substitution (SCES) is more aggressive in wirelength minimization. It allows slight path length degradation but make it under control by constraining the shallowness.

Edge substitution is an effective technique for constructing RSMT [22, 23], where a *target vertex* (e.g., v_i in Figure 3.12) is considered for connecting to a nearby *candidate edge* ($v_jv'_j$ in Figure 3.12). The idea can be brought to shallow-light trees, but the consideration in shallowness besides lightness poses a great limit on the solution space. To minimize wirelength in RSMT construction, it simply requires removing the longest edge along the circle formed by the new edge. But this could incur huge influence on the path lengths. Because not only the path lengths of many vertices can be degraded (if a vertex has longer path to the root, all its descendants suffer), but also the directions of edges may

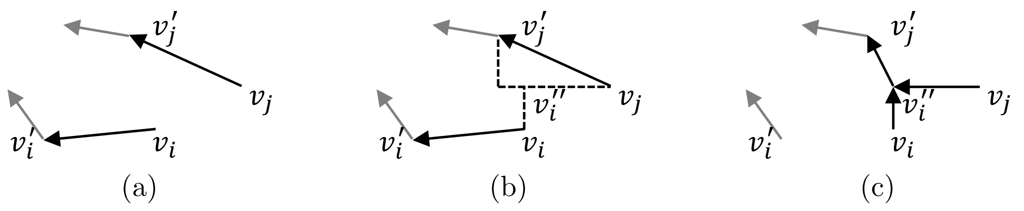


Figure 3.12: Shallowness-constrained edge substitution (SCES). (a) Before SCES, v_i is the target vertex with v_i' being its parent, while v_jv_j' is the candidate edge. (b) v_i'' is the closest point to v_i within the bounding box of edge v_jv_j'. (c) After SCES, edge v_iv_i' is substituted by edges v_iv_i'', v_jv_i'', and $v_i''v_j'$.

Algorithm 3.6 Shallowness-Constrained Edge Substitution

```
Require: Tree $T(V, E)$;
Ensure: Refined tree $T'$;
 1: Compute $slack(T(v))$ for $v \in V$ (by two tree traversals);
 2: Query candidate edges for $V$ (by nearest neighbors or R-tree);
 3: for $v_i \in V$ do
 4:     Best edge index $k \leftarrow null$
 5:     Best wirelength change $\Delta WL^* \leftarrow 0$
 6:     for each candidate edge $v_jv_j'$ of $v_i$ do
 7:         Continue if $v_j \in T(v_i)$;
 8:         $v_i'' \leftarrow$ closest point to $v_i$ in the bounding box of $v_jv_j'$;
 9:         Wirelength change $\Delta WL \leftarrow d_G(v_i, v_i'') - d_G(v_i, v_i')$;
10:         Path length change $\Delta PL \leftarrow d_T(r, v_j') + d_G(v_j', v_i) - d_T(r, v_j')$;
11:         if $\Delta WL < \Delta WL^*$ and $\Delta PL < slack(T(v_i))$ then
12:             $\Delta WL^* \leftarrow \Delta WL$;
13:             $k \leftarrow j$;
14:         end if
15:     end for
16:     if $\Delta WL^* < 0$ then
17:         Disconnect $v_iv_i'$, connect $v_iv_i''$, $v_kv_i''$ and $v_i''v_k'$;
18:         Update $slack(T(u))$ for vertex $u$ in $T(v_i)$ and path to $r$;
19:     end if
20: end for
```

be reversed. SCES is thus proposed. Here, for a tree resulted by running SALT with ϵ, its shallowness α after SCES will be still under $1 + \epsilon$.

In SCES, the *substituted edge* is restricted to be the parent edge of the target vertex only (e.g., edge v_iv_i' in Figures 3.12(b) and 3.12(c)) due to two reasons. First, reversing edges tends to cause detour and thus shallowness violation. Second, for each of the $O(n)$ possible substituted edges along the circle, $O(n)$ vertices may have path lengths affected, leading to high computation cost for a single pair of target vertex and candidate edge.

Algorithm 3.6 shows the details of the proposed SCES. In order to efficiently check whether an edge substitution violates shallowness constraint, path length $d_T(r, v)$ for each vertex v is pre-computed by a pre-order traversal. Slack $slack(v)$ of each vertex v and $slack(T(v))$ of the subtree $T(v)$ rooted at v are then computed by a post-order traversal

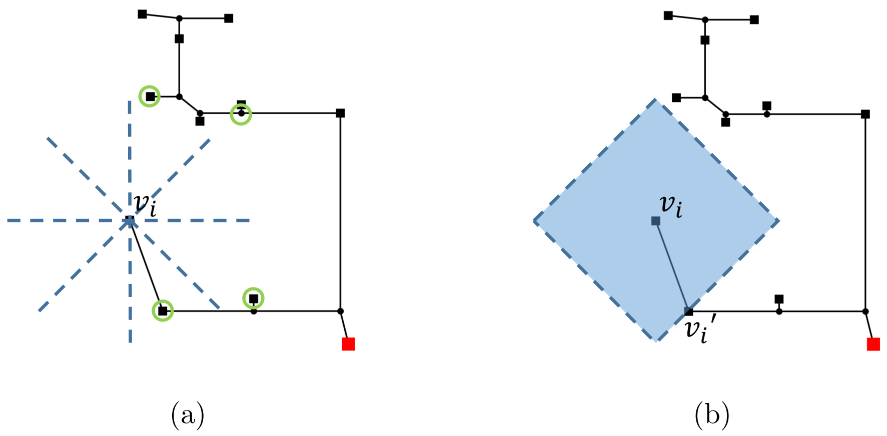


Figure 3.13: Two ways to find the candidate edges for SCES: (a) nearest neighbor in each octant (marked by green), and (b) R-tree query by an Manhattan circle.

followed (line 1):

$$slack(v) = (1+\epsilon)\cdot d_G(r,v) - d_T(r,v) \tag{3.17}$$

$$slack(T(v)) = \min_{u\in T(v)} slack(u). \tag{3.18}$$

In this way, for a target vertex v_i, if a candidate substitution increases its path length by ΔPL, its legality means $\Delta PL < slack(T(v_i))$ (line 11). Among all the candidate edges of a vertex v_i, the one that legally saves most wirelength will be connected to v_i by a Steiner point (v_i'' in Figure 3.12(c)). Note that a legal candidate edge v_jv_j' cannot be in $T(v_i)$ (line 7), which will otherwise make the tree disconnected. For a good order of visiting vertices (line 3), Algorithm 3.6 can be run twice in different modes. The first run calculates the wirelength improvement under the input topology T, while the second processes the edge substitutions in the order of descending improvement and commits those that are still legal.

Now the only problem left is how to efficiently identify candidate edges (line 2). The first way is to exploit the geometrical proximity information embedded in the *spanning graph* [18], similar to what Zhou [23] does for RSMT. That is, consider the edges connected to the *nearest neighbor* vertex of the target vertex in each octant (Figure 3.13(a)). Therefore, candidate edges are in a total number of $O(n)$ and can be obtained in $O(n \log n)$ time [23].

The second way adopts an R-tree [113], which stores the bounding boxes of all the edges. For a target vertex v_i with parent v_i', we query candidate edges by the Manhattan circle centered at v_i and with a radius of $d_G(v_i, v_i')$ (Figure 3.13(b)). Here, an edge outside the Manhattan circle is unable to save wirelength. Compared with using nearest neighbors, querying by R-tree has two strengths. First, it never misses any candidate that can reduce wirelength. Meanwhile, a "good" candidate edge may be blocked by other vertices

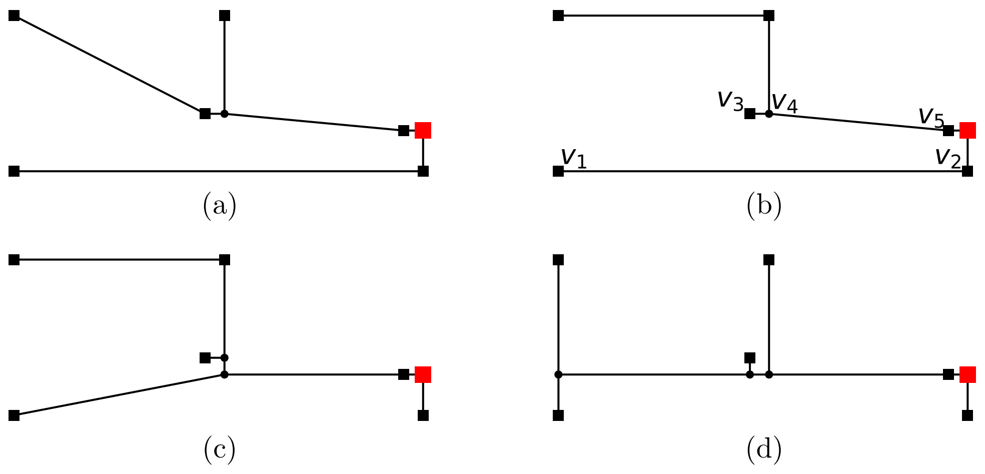


Figure 3.14: Insufficiency of SCES based on nearest neighbors. Assume that a small ϵ (e.g., 0.05) is used. (a) The input tree. (b) The only SCES that can be achieved by considering nearest neighbors. (c) The SCES achieved by using R-tree. (d) The final result of iterative SCES.

in the spanning graph. For example, for the tree in Figure 3.14(b), SCES that connects target vertex v_1 to candidate edge v_4v_5 can lead to an improved tree (Figure 3.14(c)). Note that connecting v_1 to edge v_3v_4 also reduces wirelength, but it causes detour and may violate the shallowness constraint for the path from v_1 to the root. Here, the method of nearest neighbors cannot identify edge v_4v_5 as a candidate for v_1 because v_4 is blocked by v_3 in the spanning graph. With the help of R-tree query, the candidate edge v_4v_5, however, can be easily obtained. Second, R-tree usually results fewer candidate edges and saves runtime, especially for vertex with shorter parent edge (recall Figure 3.13). Therefore, R-tree-based SCES is used in our default flow. Besides, it can be iterated to accumulate wirelength improvement (Figure 3.14(d)).

Besides [22,23], SCES also recalls the detour-aware Steinerization (DAS) in PD-II [3]. DAS also restricts the substituted edge to the parent edge of the target vertex. However, SCES and DAS have two-fold differences. First, DAS uses nearest neighbors to identify candidate edges. Even though the nearest neighbor graph in DAS is not exactly the spanning graph, it also suffers from the two aforementioned problems. Second, it only constrains the path length degradation on the target vertex v_i, without considering its impact to the downstream vertices (i.e., $d_T(r, v_i) \leq 0.5 \cdot max_{u \in V} d_T(r, u)$ instead of our $\Delta PL \leq slack(T(v_i))$). That is, the path length of a vertex may be degraded several times due to its ancestors without constraint on such accumulation.

SCES is performed after safe refinement due to two reasons. First, safe refinement mostly reduces path length and never degrades path length, which slackens the shallowness constraint for SCES. Second, SCES by R-tree is a generalization of intersected edge canceling and L-/Z-shape edge substitution. It explores a larger solution space but also requires more runtime. Conducting safe refinement first will trigger SCES fewer times

Table 3.5: ICCAD 2015 Benchmark Statistics

Design	# cells	# nets classified by pin number ($\times 10^3$)						
	($\times 10^3$)	2	3	4-7	8–15	16–31	≥ 32	≥ 4
`superblue1`	1932	893	146	121	30	19	5.6	176
`superblue3`	1876	952	113	87	38	28	6.0	160
`superblue4`	796	610	88	63	20	18	3.3	104
`superblue5`	982	824	119	115	20	15	4.5	154
`superblue7`	768	1493	184	134	59	53	10.4	257
`superblue10`	1087	1457	238	129	40	26	9.0	204
`superblue16`	1213	756	99	103	23	13	4.5	144
`superblue18`	1210	575	101	47	22	22	4.8	96
Total	9863	7559	1087	800	253	194	48	**1295**

and save the total runtime.

3.5 Experimental Results

We implement SALT as well as ES [40], CL [31], ABP/BRBC [35,36], KRY [37], PD [38], and Bonn [110] algorithms in C++, while the source code of FLUTE [25] is obtained from the authors. For a low-degree net, the idea of FLUTE has been extended to generate all the RSMTs instead of just one [115]. Among all the RMSTs, the shallowest one can be selected to serve as a better reference. We obtain the look-up table files from the authors. Moreover, the results of PD-II [3][3] are provided by the authors.

Benchmarks of ICCAD 2015 Contest [116] are used for a comprehensive evaluation and comparison. The benchmark statistics are shown in Table 3.5. By ignoring 2-pin and 3-pin nets, which are trivial, the batch test covers around 1.3 million nets in total. Experiments are performed on a 64-bit Linux workstation with Intel Xeon 3.4 GHz CPU and 32 GB memory. A single thread is used for simplicity, in spite that different nets can be routed with SALT in parallel.

In the batch test, ϵ is set to 20 values ranging from 0 to 73.895 (mainly a geometric sequence 0.05×1.5^i) to cover the variation of different methods. The lightness metric is changed to $\beta' = \frac{w(T)}{w(FLUTE)}$ (instead of $\beta = \frac{w(T)}{w(MST)}$), where FLUTE serves as a tighter baseline than MST. Besides, a normalized Elmore delay metric γ, which assumes uniform unit-length capacitance and resistance, is also used. For each routing tree, *delay* γ is the longest Elmore delay among all source-sink paths, which is then normalized by a delay lower bound using the method in [110]. For each method and each ϵ, we average the scores over all the nets.

[3]Here PD-II denotes the complete flow in [3]. It is the PD construction followed by the spanning tree refinement, Steinerization, the Steiner tree refinement, and a meta-heuristic. The meta-heuristic runs FLUTE in parallel. If FLUTE is better in both wirelength and path lengths, it is output. In the original paper, PD-II stands for PD with spanning tree refinement.

3.5.1 Effectivenss of Post Processing

Table 3.6 and Figure 3.15 show the effectiveness of our post-processing techniques, safe refinement (SR) and shallowness-constrained edge substitution (SCES). The contributions of the three SR techniques, intersected edge canceling (IEC), L-/Z-shape edge flipping (LEF), and U-shape edge shifting (UES), are all shown. Performance of both implementation of SCES, by nearest neighbor (NN) and by R-tree, is also presented.

As Table 3.6 shows, SR simultaneously improves α, β' and γ for every ϵ. Meanwhile, for a given tree, SCES gives large improvement on lightness by possibly slightly sacrificing shallowness (and delay). For example, when $\epsilon = 0.05$, "SR + SCES by R-tree" reduces the lightness of SR by 4.8% (from 1.0897 to 1.0376), with shallowness only increased by 0.14% (from 1.0062 to 1.0076). In general, it is obvious from Figure 3.15 that the Pareto frontiers are pushed towards the origin by both SR and SCES. Regarding the two kinds of implementation of SCES, R-tree achieves larger wirelength savings than NN due to its more comprehensive scope.

For nets with various pin numbers, the shallowness and lightness gaps between RSMA and RSMT are enlarged as net scales increase (see Figures 3.15(a) to 3.15(d)). SALT with "SR + SCES by R-tree", however, can always deliver a smooth trade-off between RSMA and RSMT. For low-degree nets, the shallowest RSMT achieves much better α than FLUTE by enumerating all RSMTs (Figure 3.15(a)). However, after post processing, SALT obtains almost the same α even when β' is the minimum. Note that this is achieved without the time-consuming enumeration.

Figures 3.15(e) and 3.15(f) summarize the shallowness-lightness and delay-lightness trade-off for all nets. Note that as lightness β increases, delay γ first decreases and then slightly goes up. The reason is that larger β causes higher load capacitance for the driving cell and thus more cell delay.

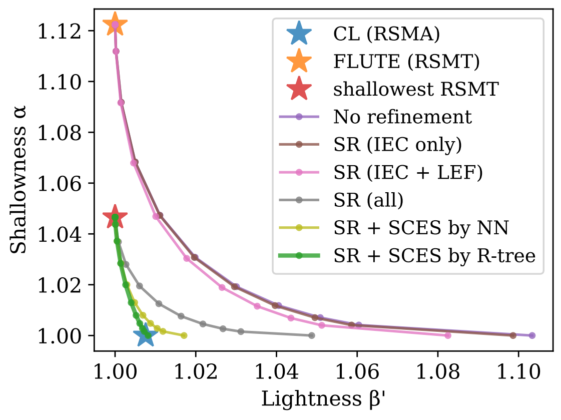
SALT is also very efficient. The runtime breakdown is shown in Figure 3.16. An $O(n \log n)$ runtime growth (w.r.t. net scales) can be observed. Moreover, for the 1.3 million nets in the ICCAD 2015 benchmark, SALT with post processing spends 0.0654 ms for each net on average. That is, it finishes routing all the eight benchmarks in 1.42 minutes with a single thread under an ϵ.

3.5.2 Superiority over Other Methods

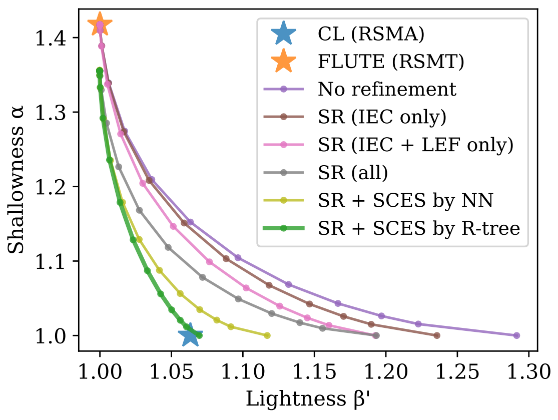
First of all, to give the readers some understanding of other routing tree construction methods, sample runs on the example net are shown in Figure 3.17. Trade-off parameter ϵ is set to 1. Recall that it implies a shallowness-lightness bound of $(1 + 2\epsilon, 1 + \frac{2}{\epsilon})$ for ABP and $(1 + \epsilon, 1 + \frac{2}{\epsilon})$ for KRY. In PD, it means shallowness $\alpha \leq 1 + \epsilon$. In Bonn, which targets the Elmore delay, the total tree capacitance is at most $1 + \frac{2}{\epsilon}$ times the minimum (i.e., lightness bound $\bar{\beta} = 1 + \frac{2}{\epsilon}$ if pin capacitances are ignorable), while wire delay is at

Table 3.6: Effectiveness of Post Processing

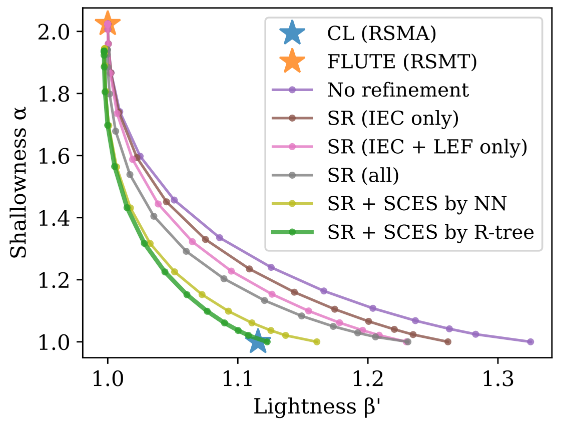
Trade-off	No refinement			SR (IEC only)			SR (IEC + LEF only)			SR (all)			SR + SCES by NNs			SR + SCES by R-tree		
parameter	Light	Shallow	Delay	Light	Shallow	Delay	Light	Shallow	Delay	Light	Shallow	Delay	Light	Shallow	Delay	Light	Shallow	Delay
ϵ	-ness β'	-ness α	γ	-ness β'	-ness α	γ	-ness β'	-ness α	γ	-ness β'	-ness α	γ	-ness β'	-ness α	γ	-ness β'	-ness α	γ
0.000	1.1834	1.0000	1.4302	1.1574	1.0000	1.4309	1.1330	1.0000	1.4262	1.1125	1.0000	1.4206	1.0647	1.0000	1.4114	**1.0429**	**1.0000**	**1.4110**
0.050	1.1358	1.0105	1.4073	1.1188	1.0102	1.4131	1.1035	1.0095	1.4119	1.0897	**1.0062**	1.4070	1.0519	1.0076	**1.4047**	**1.0376**	1.0076	1.4088
0.075	1.1211	1.0178	**1.4036**	1.1067	1.0174	1.4099	1.0934	1.0163	1.4090	1.0812	**1.0110**	1.4039	1.0469	1.0132	1.4043	**1.0349**	1.0131	1.4091
0.113	1.1038	1.0291	1.4025	1.0922	1.0286	1.4087	1.0810	1.0271	1.4080	1.0707	**1.0187**	**1.4024**	1.0407	1.0220	1.4058	**1.0312**	1.0219	1.4111
0.169	1.0844	1.0463	1.4058	1.0754	1.0455	1.4115	1.0665	1.0436	1.4108	1.0582	**1.0309**	**1.4040**	1.0333	1.0354	1.4109	**1.0265**	1.0353	1.4160
0.253	1.0638	1.0711	1.4159	1.0574	1.0701	1.4204	1.0507	1.0678	1.4196	1.0444	**1.0492**	**1.4109**	1.0252	1.0549	1.4209	**1.0208**	1.0547	1.4253
0.380	1.0441	1.1053	1.4340	1.0397	1.1040	1.4372	1.0349	1.1016	1.4363	1.0307	**1.0748**	**1.4246**	1.0172	1.0814	1.4371	**1.0147**	1.0812	1.4402
0.570	1.0272	1.1478	1.4602	1.0246	1.1464	1.4621	1.0215	1.1441	1.4612	1.0190	**1.1068**	**1.4456**	1.0103	1.1140	1.4593	**1.0090**	1.1141	1.4611
0.854	1.0145	1.1985	1.4927	1.0131	1.1971	1.4937	1.0114	1.1951	1.4928	1.0101	**1.1456**	**1.4729**	1.0052	1.1526	1.4855	**1.0045**	1.1530	1.4868
1.281	1.0064	1.2509	1.5269	1.0058	1.2499	1.5272	1.0050	1.2485	1.5266	1.0044	**1.1873**	**1.5028**	1.0019	1.1934	1.5121	**1.0015**	1.1938	1.5131
1.922	1.0022	1.2952	1.5556	1.0020	1.2945	1.5556	1.0017	1.2938	1.5553	1.0014	**1.2252**	**1.5283**	1.0002	1.2295	1.5334	**1.0000**	1.2298	1.5340
2.883	1.0006	1.3244	1.5735	1.0006	1.3242	1.5735	1.0005	1.3239	1.5734	1.0003	**1.2530**	**1.5445**	0.9996	1.2549	1.5460	**0.9995**	1.2545	1.5459
4.325	1.0001	1.3448	1.5834	1.0001	1.3448	1.5833	1.0001	1.3447	1.5833	1.0000	**1.2721**	1.5532	0.9995	1.2731	1.5534	**0.9994**	1.2722	**1.5530**
6.487	1.0000	1.3554	1.5873	1.0000	1.3554	1.5873	1.0000	1.3553	1.5872	0.9999	1.2815	1.5566	0.9994	1.2822	1.5565	**0.9994**	**1.2808**	**1.5559**
9.731	1.0000	1.3592	1.5884	1.0000	1.3591	1.5884	1.0000	1.3591	1.5883	0.9999	1.2848	1.5575	0.9994	1.2854	1.5573	**0.9994**	**1.2837**	**1.5568**
14.597	1.0000	1.3598	1.5885	1.0000	1.3598	1.5885	1.0000	1.3598	1.5885	0.9999	1.2853	1.5576	0.9994	1.2858	1.5574	**0.9994**	**1.2841**	**1.5569**
...	1.0000	1.3598	1.5885	1.0000	1.3598	1.5885	1.0000	1.3598	1.5885	0.9999	1.2853	1.5577	0.9994	1.2858	1.5574	**0.9994**	**1.2841**	**1.5569**



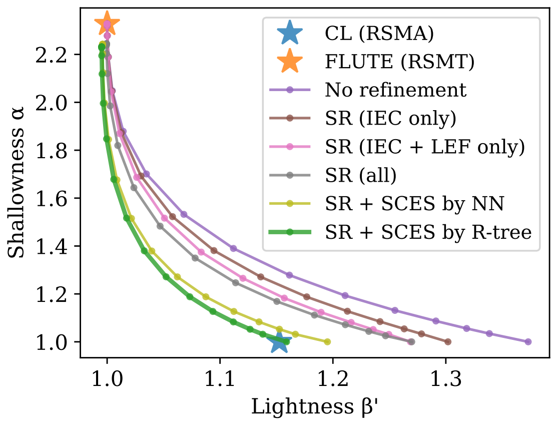
(a) Shallowness-lightness on nets with 4–7 pins



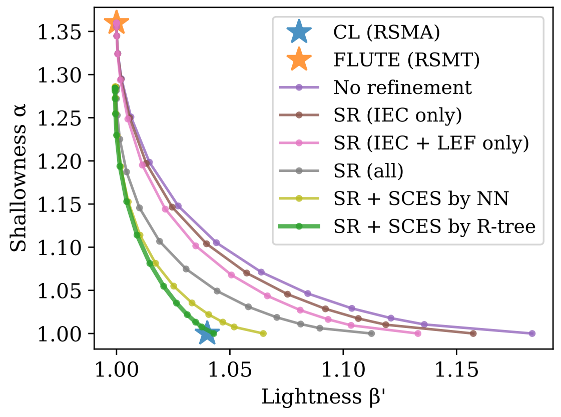
(b) Shallowness-lightness on nets with 8–15 pins



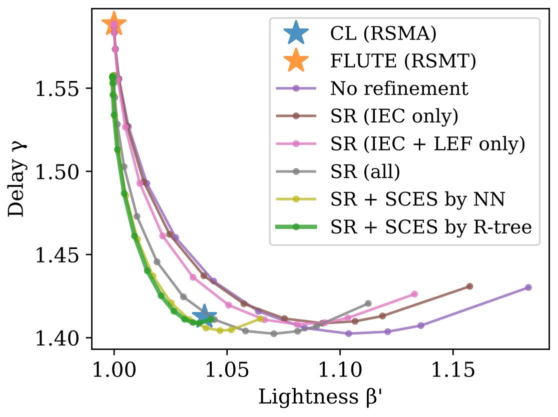
(c) Shallowness-lightness on nets with 16–31 pins



(d) Shallowness-lightness on nets with 32+ pins



(e) Shallowness-lightness on all nets



(f) Delay-lightness on all nets

Figure 3.15: Effectiveness of post processing shown by shallowness-lightness and delay-lightness trade-off on nets of various scales.

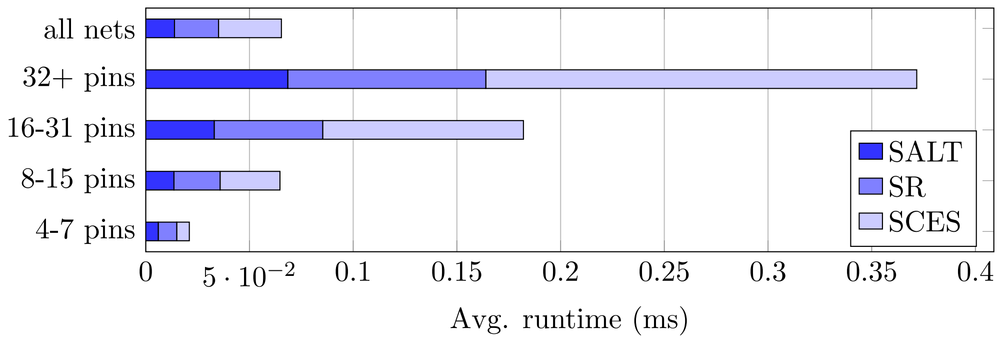


Figure 3.16: Runtime breakdown of SALT with post processing.

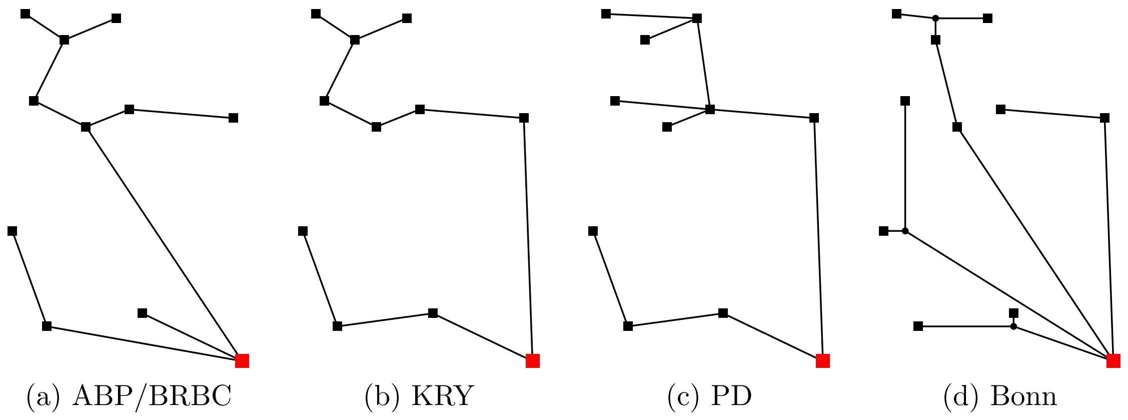


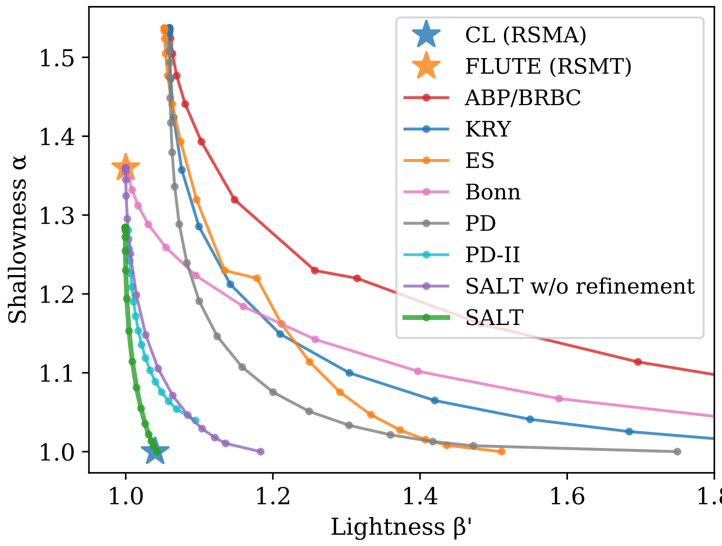
Figure 3.17: Sample runs of various algorithms ($\epsilon = 1$). (a) ABP/BRBC (shallowness $\alpha = 2.24$, lightness $\beta' = 1.51$). (b) KRY ($\alpha = 1.43, \beta' = 1.22$). (c) PD ($\alpha = 1.11, \beta' = 1.30$) (d) Bonn ($\alpha = 1.22, \beta' = 1.87$).

most a factor of $(1 + \epsilon)^2$ compared to a lower bound.

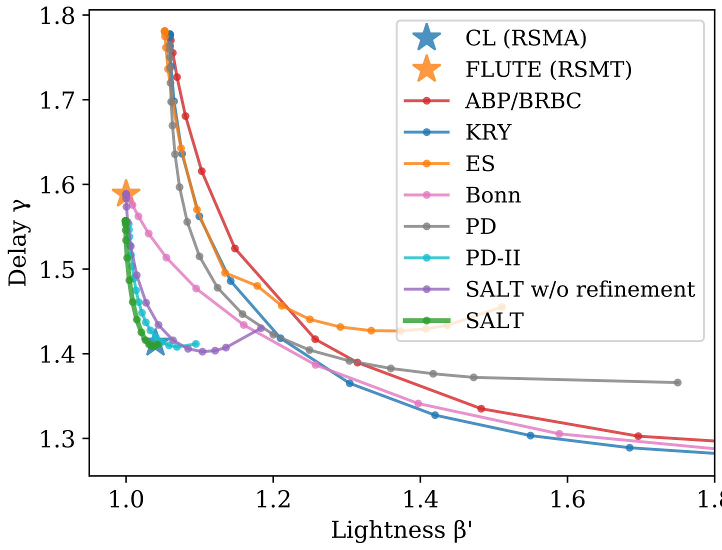
Compared with all the other methods (including ABP, KRY, ES, Bonn, PD, and PD-II), SALT shows superior performance, which mostly leads to both smaller wirelength and shorter path lengths. The average situation is illustrated by Figure 3.18(a). It can be clearly observed that our method has the best Pareto frontier between RSMT and RSMA.

Figure 3.18(b) illustrates the delay and lightness of different methods, where SALT still achieves a good trade-off. Though KRY may obtain a slightly smaller delay, the wirelength cost is actually significant. Besides, the smaller Elmore delay there is usually achieved by unnecessary long edges, which is much less preferable than assigning the edge to a higher metal layer or buffering. For example in Figure 3.19, the longest path of SALT (measured by Elmore delay) is the path from root r to pin v_1. Comparing with that in KRY, $r - v_1$ path in SALT has the same path length but drives more capacitance load before vertex v_2. KRY reduces the delay by connecting v_2 to r directly at the cost of wirelength. However, appropriate layer assignment or buffering on $r - v_2$ path will be more economical in practice.

Lastly, we conduct a detailed comparison between SALT and PD-II, the closest com-

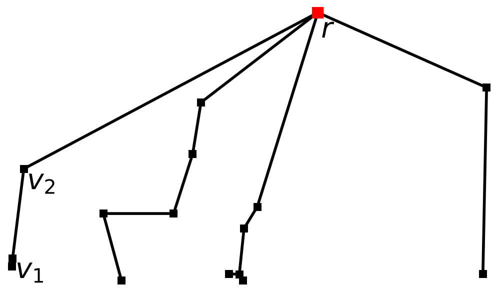


(a) Trade-off between shallowness and lightness

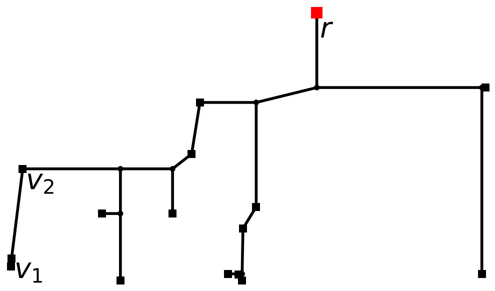


(b) Trade-off between delay and lightness

Figure 3.18: Comparing SALT with other routing tree construction methods.

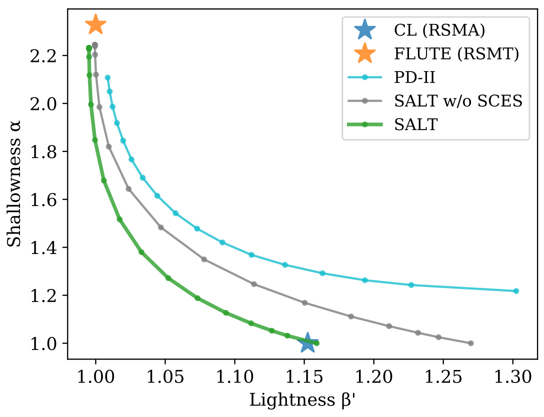


(a) KRY ($\beta' = 1.572, \gamma = 1.292$)

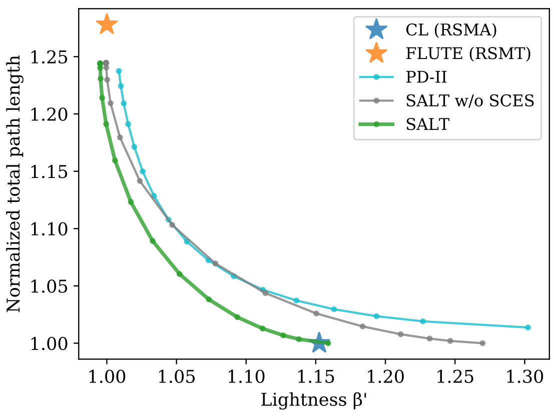


(b) SALT ($\beta' = 1.098, \gamma = 1.994$)

Figure 3.19: Comparing SALT with KRY on a 16-pin net of `superblue1`. The trade-off parameter ϵ leading to the smallest delay γ is picked.



(a) Trade-off between shallowness and lightness



(b) Trade-off between total path length and lightness

Figure 3.20: Comparing SALT with PD-II on high-pin nets (# pins $\geq$ 32).

petitor among all other methods. In [3], the authors compare PD-II with the preliminary version of SALT [6], which is without SCES. They use two metrics to measure the path lengths. The first is our shallowness metric $\alpha = \max\{\frac{d_T(r,v)}{d_G(r,v)} | v \in V\}$; the second is the total path length normalized by the total shortest-path distance, i.e., $\frac{\sum_{v\in V} d_T(r,v)}{\sum_{v\in V} d_G(r,v)}$. There, PD-II wins SALT in some cases, especially for nets with 32+ pins. The experimental results of PD-II and SALT (with and without SCES) on nets with 32+ pins are shown in Figure 3.20. As we can see, if without SCES, SALT loses out to PD-II in total path length (when lightness β' is around 1.05–1.10). However, with the help of SCES, SALT dominates PD-II. Regarding the shallowness-lightness trade-off, SALT is always superior to PD-II, even without SCES.

Chapter 4

Trade-off Between Wirelength and Skew

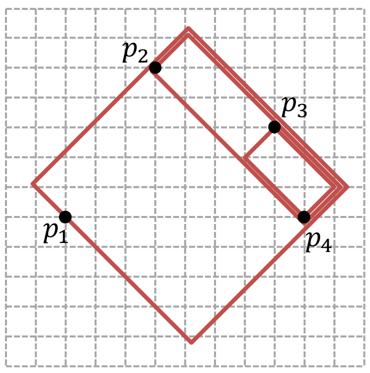
A problem similar to ZST is the *hierarchical clustering* (HC) [117]. In HC, each point starts as a cluster by itself, and pairs of clusters are merged when moving up the hierarchy. By retrospecting the classical DME algorithm, we found the equivalence between ZST and HC. To be more specific, the wirelength of a ZST is a linear function of the sum of diameters of its corresponding HC (see Figure 4.1 and Theorem 4.1 for a glance). With the help of the new insight as well as the "BST by ZST" idea from [47,48], better algorithms for both ZST and BST construction are devised. Our contributions are summarized as follows.

- We proved the strict correspondence between the wirelength of ZST and the diameter sum of HC.
- With this new insight, we designed an effective $O(n \log n)$-time $O(1)$-approximation algorithm and an optimal dynamic programming for ZST construction.
- We proposed a linear-time optimal tree decomposition algorithm. Together with the black-box ZST construction, it leads to an improved BST generation method.

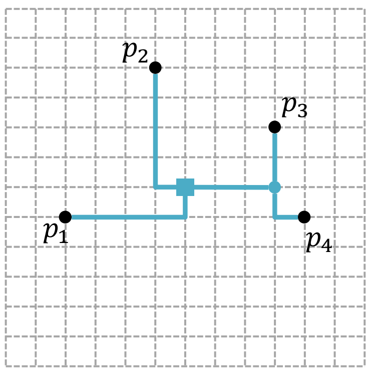
4.1 Zero-Skew Tree Properties

VLSI routing is in Manhattan space. Before formal illustration, we would like to define the notations for distance. For two points p_1 and p_2, $dist(p_1, p_2)$ represents their Manhattan (l_1) distance. For two sets of points P_a and P_b, their distance $dist(P_a, P_b)$ is defined as the smallest distance between a point in P_a and a point in P_b. That is, $dist(P_a, P_b) = \min_{p_1 \in P_a, p_2 \in P_b} dist(p_1, p_2)$. We use linear delay model in this chapter.

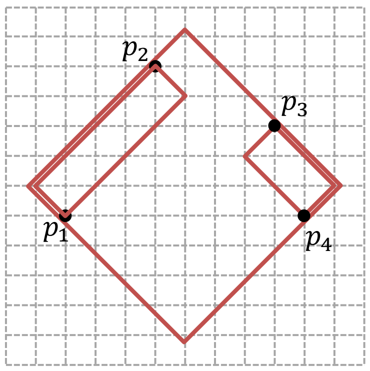
Manhattan arc, the line segment with slope +1 or -1, is widely needed in ZST construction. For convenience and efficiency, the computation in the l_1 space can be converted



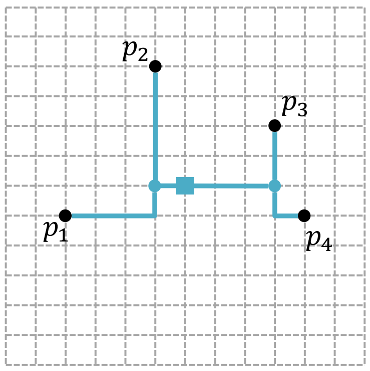
(a) HC with diameter sum $= d(\{p_3, p_4\}) + d(\{p_2, p_3, p_4\}) + d(P) = 4 + 10 + 10 = 24$.



(b) ZST corresponding to (a) with wirelength $= \frac{1}{2}(4 + 10) + 10 = 17$.



(c) HC with diameter sum $= d(\{p_3, p_4\}) + d(\{p_1, p_2\}) + d(P) = 4 + 8 + 10 = 22$.



(d) ZST corresponding to (c) with wirelength $= \frac{1}{2}(4 + 8) + 10 = 16$.

Figure 4.1: Equivalence between zero-skew tree (ZST) and hierarchical clustering (HC) on points $P = \{p_1, p_2, p_3, p_4\}$.

to that in the l_∞ space. More specifically, suppose the l_∞ distance between p_1 and p_2 in the coordinate system tilted by 45° is $dist'(p1, p2)$. The distance in the l_∞ space can be obtained by scaling $dist'(p1, p2) = \frac{\sqrt{2}}{2} dist(p1, p2)$, and the Manhattan arc will become axis-aligned.

4.1.1 Manhattan Circle

Manhattan circle $C(o, r)$ is a circle in the Manhattan space with *center* o and *radius* r. To be more specific, $C(o, r)$ is the set of all points that are at a given Manhattan distance r from a given point o (i.e., $C(o, r) = \{p | dist(p, o) = r\}$). *Diameter* d of a circle C is the largest distance between any two points on the circle (i.e. $d = \max_{p_1, p_2 \in C} dist(p_1, p_2)$). Note that $d = 2r$.

Circle $C(o, r)$ is an *internally tangent circle* of circle $C'(o', r')$ if (i) C intersects C' (i.e., $C \cap C' \neq \emptyset$), and C is inside C' (i.e., $\forall p \in C, dist(p, o') \leq r'$). This is illustrated by Figure 4.2. Regarding the internally tangent circle, we have Lemma 4.1, which is similar

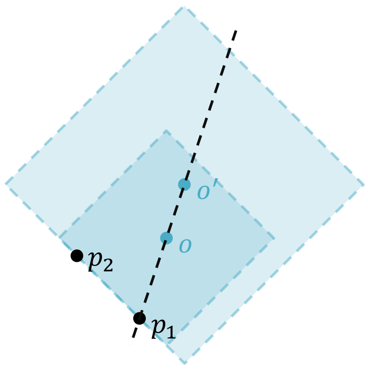


Figure 4.2: Manhattan circle $C(o, r)$ is an internally tangent circle of $C'(o', r')$.

for circles in the Euclidean space.

Lemma 4.1. *If circle $C(o, r)$ is an internally tangent circle of circle $C'(o', r')$, then $dist(o, o') = r' - r$.*

Proof. It can be proved by construction that there exists a point $p_1 \in C$ with $dist(p_1, o') = r + dist(o, o')$. To be more specific, there are two points at distance r from o on line oo' (see Figure 4.2). The one further from o' between the two can be such a p_1. Obviously, $dist(p_1, o') = dist(p_1, o) + dist(o, o')$. Besides, there is $dist(p_1, o') \le r'$ since C is inside C'. Therefore,

$$dist(o, o') = dist(p_1, o') - dist(p_1, o) \le r' - r. \tag{4.1}$$

Let point $p_2 \in C \cap C'$. By triangle inequality,

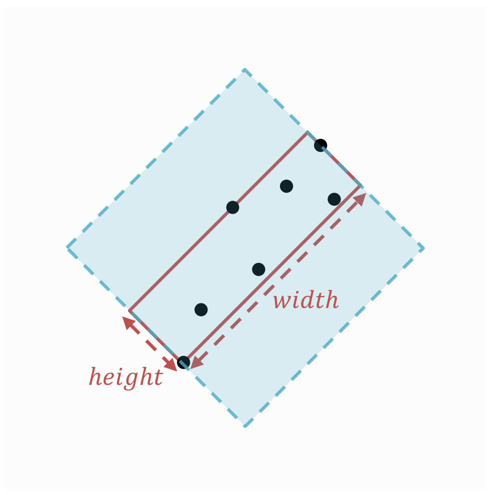
$$dist(o, o') \ge dist(p_2, o') - dist(p_2, o) = r' - r. \tag{4.2}$$

Combining (4.1) and (4.2) leads to $dist(o, o') = r' - r$. □

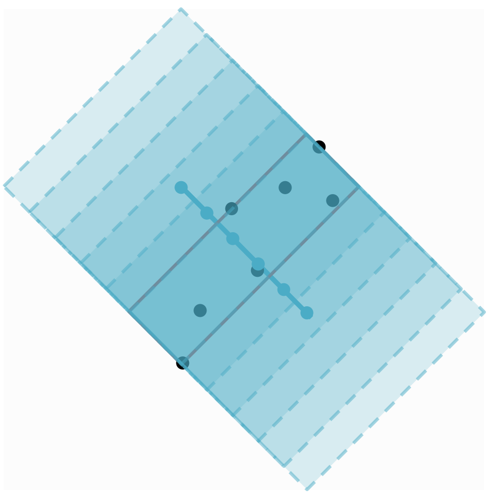
4.1.2 Manhattan Bounding Circle

For a set of points P, its *Manhattan bounding circle (MBC)* is a (minimum) Manhattan circle covering all the points (see Figure 4.3(a)). Its *radius* $r(P)$ (resp. *diameter* $d(P)$) is the radius (resp. diameter) of the MBC. Note that for a point set P, there may be many MBCs, but the radius as well as the diameter of the MBCs are all the same (Figure 4.3(b)).

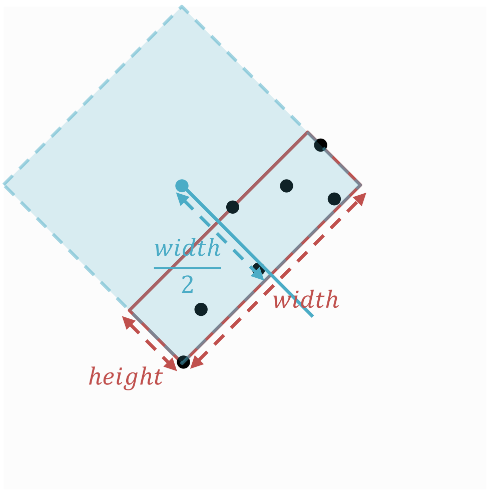
For computational convenience, we can analyze and compute MBC in the 45°-tilted coordinate system. Refer the (minimum) bounding box for P in this tilted coordinate system as *tilted bounding box (TBB)* $tbb(P)$. A Manhattan circle is an MBC of points P iff it is an MBC of $tbb(P)$. In this way, it is easy to see that the centers of all possible MBCs of P form an Manhattan arc parallel to the shorter side of $tbb(P)$, which will be called the *center segment* $cs(P)$ hereafter. We refer the length of the longer (resp. shorter) side of $tbb(P)$ as the *width* (resp. *height*) of $tbb(P)$. There is thus $d(P) = 2r(P) = \sqrt{2}width$.



(a) The TBB (solid red) and an MBC (dash blue) for P



(b) Centers of all possible MBCs form center segment (solid blue)



(c) Relationship between TBB and center segment

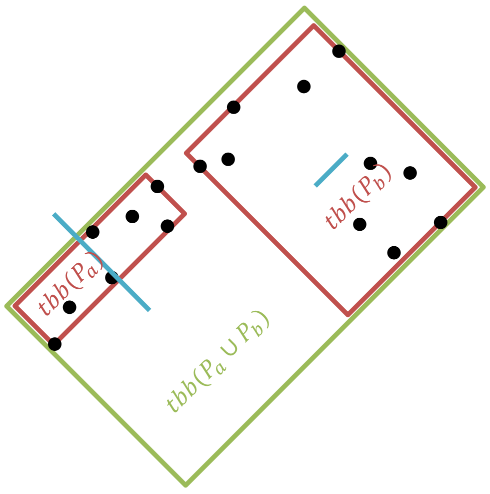
Figure 4.3: Manhattan bounding circle (MBC), tilted bounding box (TBB), and center segment for a set of points P.

Besides, the distance from an end of $cs(P)$ to the further side of $tbb(P)$ is $\frac{width}{2}$ (see Figure 4.3(c)).

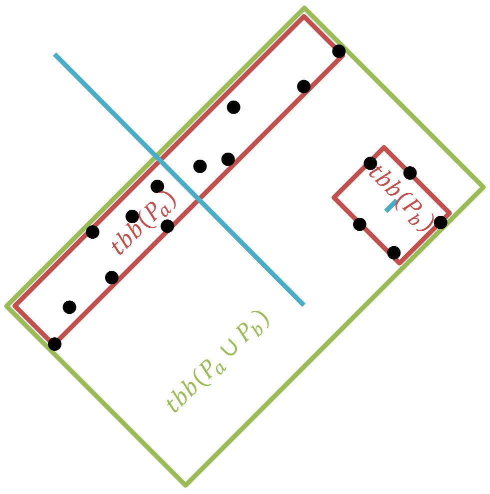
Regarding diameter $d(P)$, Lemma 4.2 is obvious (same as Property 11 of [118]), which provides another view to $d(P)$. Essentially, there exists at least a point in P on each of the two shorter sides of $tbb(P)$ such that they determine the diameter $d(P)$ of any MBC of P as well as the maximum distance within P. Note that Lemma 4.2 does not hold for the Euclidean metric.

Lemma 4.2. *For a point set P, $d(P) = \max_{p_1,p_2 \in P} dist(p_1, p_2)$.*

For two point sets P_a and P_b, we say P_a *dominates* P_b if and only if $r(P_a) = r(P_a \cup P_b)$. The two kinds of relationships are illustrated by Figure 4.4. We have Lemma 4.3 regarding



(a) No domination



(b) P_a dominates P_b

Figure 4.4: Relationship between two point sets P_a and P_b.

domination. The proof is omitted due to the space limit.

Lemma 4.3. *If two point sets P_a and P_b do not dominate each other, then $dist(cs(P_a), cs(P_b)) = 2r(P_a \cup P_b) - r(P_a) - r(P_b)$. If P_a dominates P_b, then $dist(cs(P_a), cs(P_b)) \leq r(P_a) - r(P_b)$.*

Proof. Let $P_{ab} = P_a \cup P_b$.

Case 1: P_a and P_b do not dominate each other (Figure 4.4(a)).

Since P_a and P_b do not dominate each other, the largest distance $d(P_{ab})$ must be achieved by one point from each set. Suppose they are $p_a^* \in P_a$ and $p_b^* \in P_b$. Therefore, $\forall o_a \in cs(P_a), \forall o_b \in cs(P_b)$,

$$\begin{aligned} dist(o_a, o_b) &\geq 2r(P_{ab}) - dist(p_a^*, o_a) - dist(o_b, p_b^*) \\ &\geq 2r(P_{ab}) - r(P_a) - r(P_b), \end{aligned} \tag{4.3}$$

where the first inequality is due to triangle inequality.

For arbitrary $o_{ab} \in cs(P_{ab})$, there exist $o_a^* \in cs(P_a)$ and $o_b^* \in cs(P_b)$ such that $C(o_a^*, r(P_a)), C(o_b^*, r(P_b))$ are inside $C(o_{ab}, r(P_{ab}))$. We claim that such o_a^* and o_b^* can achieve the lower bound in (4.3). That is, $dist(cs(P_a), cs(P_b)) = dist(o_a^*, o_b^*) = 2r(P_{ab}) - r(P_a) - r(P_b)$. First, it is easy to see that $p_a^* \in C(o_a^*, r(P_a)) \cap C(o_{ab}, r(P_{ab}))$. Combining with the fact that $C(o_a^*, r(P_a))$ is inside $C(o_{ab}, r(P_{ab}))$, it implies $C(o_a^*, r(P_a))$ is the internally tangent circle of $C(o_{ab}, r(P_{ab}))$. So is $C(o_b^*, r(P_b))$. According to triangle inequality and Lemma 4.1, there is

$$\begin{aligned} dist(o_a^*, o_b^*) &\leq dist(o_a^*, o_{ab}) + dist(o_b^*, o_{ab}) \\ &= (r(P_{ab}) - r(P_a)) + (r(P_{ab}) - r(P_b)) \\ &= 2r(P_{ab}) - r(P_a) - r(P_b). \end{aligned} \tag{4.4}$$

Combining (4.3) and (4.4) together achieves $dist(o_a^*, o_b^*) = 2r(P_{ab}) - r(P_a) - r(P_b)$.

Case 2: P_a dominates P_b (Figure 4.4(b)).

Suppose $o_{ab}^* \in cs(P_{ab})$ and $o_b^* \in cs(P_b)$ realize $dist(cs(P_{ab}), cs(P_b))$. There exists $p_b^* \in P_b$ with $dist(p_b^*, o_b^*) = r(P_b)$. Besides, $dist(p_b^*, o_{ab}^*) \leq r(P_{ab}) = r(P_a)$. By triangle inequality,

$$\begin{aligned} r(P_a) &\geq dist(p_b^*, o_{ab}^*) \\ &\geq dist(p_b^*, o_b^*) + dist(o_b^*, o_{ab}^*) \\ &= r(P_b) + dist(cs(P_{ab}), cs(P_b)). \end{aligned} \tag{4.5}$$

Note that an MBC for P_{ab} is always an MBC for P_a, so $cs(P_{ab}) \subseteq cs(P_a)$. Therefore, $dist(cs(P_{ab}), cs(P_b)) \geq dist(cs(P_a), cs(P_b))$. Together with (4.5), the proof is completed. $\square$

4.1.3 ZST/DME by Manhattan Bounding Circle

ZST/DME algorithm [44] can find the optimal Steiner node placement for a given tree topology on a set of points. Note that in a ZST T, the given points/sinks can never be Steiner nodes due to the zero skew constraint. For a tree T, we denote its vertex v (either leaf or Steiner node) by $v \in T$ and denote the leaves of the subtree rooted at v as $leaves(v)$. Path lengths from v to $leaves(v)$ are all the same and referred as $p(v)$. In ZST/DME, the *merging segment* $ms(v)$ of vertex $v \in T$ is a set of possible placements of v. There are two phases in ZST/DME. In the first bottom-up phase, a tree of merging segments is computed recursively. In the top-down phase, the merging point achieving the minimum wirelength is picked according to the location of its parent Steiner node.

The reader may refer to [44] for algorithmic details, but in the bottom-up phase, $ms(v)$ for a vertex v with two children v_a and v_b is essentially computed as follows. Denote the nonnegative edge cost (length) from v to v_a as e_a and similarly for vertex v_b. The objective is to minimize $e_a + e_b$, which will be a part of the wirelength of the final ZST[1]. Suppose w.l.o.g. that $p(v_a) \leq p(v_b)$. Zero skew among $leaves(v) = leaves(v_a) \cup leaves(v_b)$ requires $e_a + p(v_a) = e_b + p(v_b)$. That is,

$$e_a - e_b = p(v_b) - p(v_a). \tag{4.6}$$

Meanwhile, by triangle inequality,

$$e_a + e_b \geq dist(ms(v_a), ms(v_b)), \tag{4.7}$$

[1]It can be proved that the greedy proposal here is sufficient for obtaining the minimum wirelength of the whole ZST [44].

Here, if $p(v_b) - p(v_a) \leq dist(ms(v_a), ms(v_b))$, then $e_a + e_b = dist(ms(v_a), ms(v_b))$. Otherwise, $e_a + e_b = p(v_b) - p(v_a)$, where $e_a = p(v_b) - p(v_a)$ and $e_b = 0$.

Based on Lemma 4.3, we can prove an intrinsic correspondence between DME and MBC (Lemma 4.4). It says that for a vertex v, the path length $p(v)$ equals the radius $r(leaves(v))$ of its leaves, while its merging segment $ms(v)$ is exactly the center segment $cs(leaves(v))$ of its leaves. For example, in Figure 4.1, the path length from the ZST root is always $r(P) = 5$, regardless of the topology. Similarly, the merging segment of the ZST root is uniquely determined by $tbb(P)$ (since $width = height$ in $tbb(P)$ for this example, $cs(P)$ degenerates to a point). The key point of the proof is that whether $p(v_b) - p(v_a) \leq dist(ms(v_a), ms(v_b))$ depends on the domination relationship between $leaves(v_a)$ and $leaves(v_b)$.

Lemma 4.4 (DME-MBC Correspondence)**.** *In ZST/DME, for a vertex v, (i) $p(v) = r(leaves(v))$; (ii) $ms(v) = cs(leaves(v))$.*

Proof. By induction. Denote $r(leaves(v))$ as $r(v)$ and $cs(leaves(v))$ as $cs(v)$ for short. It is definitely true for a leaf v, since (i) $p(v) = r(v) = 0$, and (ii) $ms(v) = cs(v)$ degenerates from a segment to a fixed point. Suppose it is true for children v_a and v_b of vertex v.

(a) We will first prove the inductive part of $p(v) = r(v)$.

Assume w.l.o.g that $r(v_a) \geq r(v_b)$. Note that $e_b - e_a = p(v_a) - p(v_b) = r(v_a) - r(v_b) \geq 0$. There are two cases.

Case 1: No domination, i.e., $r(v) > r(v_a) \geq r(v_b)$. According to Lemma 4.3, $dist(ms(v_a), ms(v_b)) = dist(cs(v_a), cs(v_b)) = 2r(v) - r(v_a) - r(v_b)$. Consider,

$$\begin{aligned} e_b - e_a &= r(v_a) - r(v_b) \\ &< 2(r(v) - r(v_a)) + r(v_a) - r(v_b) \\ &= 2r(v) - r(v_a) - r(v_b) \\ &= dist(ms(v_a), ms(v_b)), \end{aligned} \tag{4.8}$$

where the DME algorithm will let $e_b + e_a$ achieve its lower bound $dist(ms(v_a), ms(v_b))$. Then, $p(v) = \frac{1}{2}(p(v_a) + p(v_b) + e_a + e_b) = \frac{1}{2}(r(v_a) + r(v_b) + dist(ms(v_a), ms(v_b))) = \frac{1}{2}(2r(v)) = r(v)$.

Case 2: Having domination, i.e., $r(v) = r(v_a) \geq r(v_b)$. According to Lemma 4.3,

$$\begin{aligned} e_b - e_a &= r(v_a) - r(v_b) \\ &\geq dist(cs(v_a), cs(v_b)) \\ &= dist(ms(v_a), ms(v_b)). \end{aligned} \tag{4.9}$$

In this situation, DME lets $e_a = 0$ and $e_b = dist(ms(v_a), ms(v_b))$. Therefore, $p(v) = p(v_a) + e_a = p(v_a) = r(v_a) = r(v)$.

(b) For proving the induction on $ms(v) = cs(v)$, we will first show that $ms(v) \subseteq cs(v)$. For arbitrary $p \in v_a$, there is $dist(p, v) = dist(p, v_a) + dist(v, v_a) \leq r(v_a) + p(v) - p(v_a) = r(v)$. Similar for $p \in v_b$. That is, for $p \in v$, there is $dist(p, v) \leq r(v)$. Therefore, v should be an MBC center of v.

We then prove that $ms(v) \supseteq cs(v)$. That is, for arbitrary $o \in cs(v)$, $dist(o, ms(v_a)) = p(v) - p(v_a)$ and $dist(o, ms(v_b)) = p(v) - p(v_b)$. By $v_a \subset r(v)$ and Lemma 4.3, we have $dist(o, cs(v_a)) = dist(o, ms(v_a)) = r(v) - r(v_a) = p(v) - p(v_a)$. Similarly $dist(o, ms(v_b)) = p(v) - p(v_b)$ also holds and completes the proof. □

In this way, the merging segments in DME can be more efficiently computed. Moreover, the new insight unlocks several better clock tree construction methods, which will be introduced later.

4.1.4 ZST by Hierachical Clustering

On a set of point P, a *hierachical clustering* (HC) [117] is a hierachy of clusters with each cluster composed of two sub-clusters or points. There are generally two approaches for creating it. In the agglomerative approach, each point starts as a cluster by itself, and pairs of clusters are merged and move up the hierarchy. By the divisive one, all points start in one cluster, and splits are performed recursively as one moves down the hierarchy.

W.l.o.g, assume that any ZST is binary (i.e., each Steiner node has two children). If a Steiner node has three (or more) children, overlapped Steiner node(s) can be inserted to make it binary without affecting the actual structure. In this way, a ZST topology T implies a HC, where $leaves(v)$ of every $v \in T$ is treated as a cluster, and vice versa (see Figure 4.1 for two ZST/HC instances on the same set of points). Furthermore, we prove Theorem 4.1, which shows the equivalence between ZST and HC. To be more specific, the wirelength of a ZST $length(T)$ is a linear function of the sum of diameters $\sum_{v \in T} d(leaves(v))$ of its corresponding HC (note that $d(P)$ is a constant). Besides, note that Theorem 4.1 is stronger than Lemma 2.1 of [48] and implies the later.

Theorem 4.1 (ZST-HC Equivalence). *For points/sinks P in the Manhattan space, a ZST T has a wirelength cost $length(T)$ equivalent to the sum of diameters $\sum_{v \in T} d(leaves(v))$ of its corresponding HC: $length(T) = \frac{1}{2}(\sum_{v \in T} d(leaves(v)) + d(P))$.*

Proof. Simplify notation $r(leaves(v))$ as $r(v)$. We will prove $length(T) = \frac{1}{2}(\sum_{v \in T} d(leaves(v)) + d(P)) = \sum_{v \in T} r(v) + r(P)$ by induction.

Denote the subtree rooted at v as T_v. The claim then becomes $length(T_v) = \sum_{u \in T_v} r(u) + r(v)$. It is trivial for a leaf v, since (i) $length(T_v) = 0$, and (ii) $v = \{v\}$.

Suppose it is true for children v_a and v_b of vertex v. Note that $T_v = T_{v_a} + T_{v_b} + \{v\}$. Then it is sufficient to prove that $length(T) - length(T_{v_a}) - length(T_{v_b}) = \sum_{u \in \{v\}} r(u) + r(v) - r(v_a) - r(v_b) = 2r(v) - r(v_a) - r(v_b)$. Meanwhile, according to Lemma 4.4, there

Algorithm 4.1 ZST by Iterative Merging

```
Require: Sinks $P$
Ensure: ZST $T$
1: Initialize clusters with each sink being a cluster by itself
2: for $i = 1, 2, ..., |P| - 1$ do
3:     Merge the two clusters with the smallest diameter of their union among all pairs
4: end for
5: Run DME top-down phase on the HC to realize the ZST $T$
```

is $length(T) - length(T_{v_a}) - length(T_{v_b}) = e_a + e_b = (p(v) - p(v_a)) + (p(v) - p(v_b)) = 2p(v) - p(v_a) - p(v_b) = 2r(v) - r(v_a) - r(v_b)$.

4.1.5 Proof of Theorem 4.3

Let P'_B be a trie node in the sub-trie rooted at P_B. Due to the enforced order of points in the trie and $\pi_i < \sigma_j$, we have $P'_B \cap P_A = \emptyset$. That is, $P_A \subseteq (P - P'_B)$ and there is $cost(T^*(P_A)) \leq cost(T^*(P - P'_B))$

The lemma essentially says that the optimal sub-solution $T^*(P'_B)$ cannot be part of the optimal solution $T^*(P)$. Suppose it is false and there is an optimal ZST/HC T'' on P with $T^*(P'_B)$ being a sub-solution. For T'', we can remove $T^*(P'_B)$ and get another ZST/HC T'''. After the removal, the original ancestors of $T^*(P'_B)$ on T'' may have smaller diameters, while the sibling subtree of $T^*(P'_B)$ can move up. Therefore, $cost(T'') \geq cost(T''') + cost(T^*(P'_B)) \geq cost(T^*(P - P'_B)) + cost(T^*(P'_B)) \geq cost(T^*(P_A)) + cost(T^*(P'_B)) > cost(T') \geq cost(T^*(P))$, which is a contradiction. □

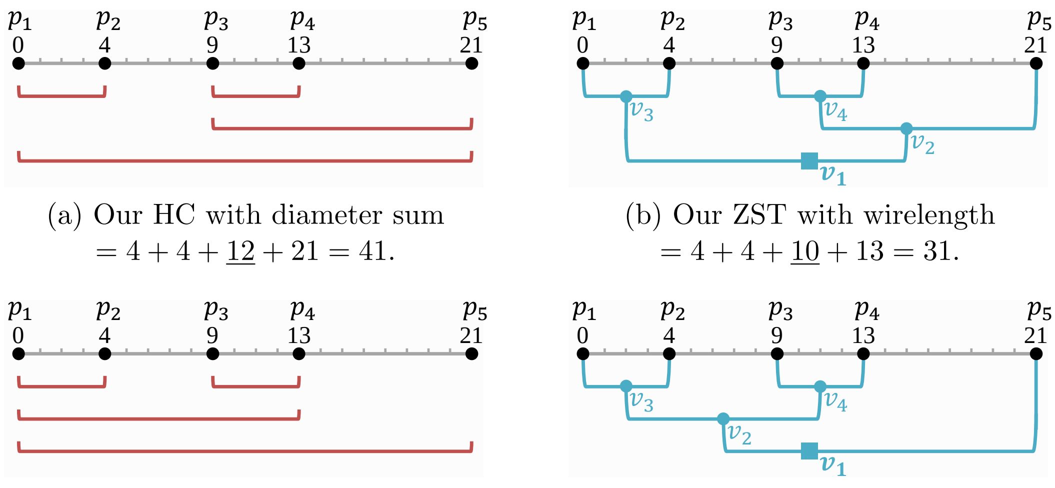
4.2 Zero-Skew Tree Construction

The zero-skew tree (ZST) problem has been converted to a hierarchical clustering (HC) problem with the diameter sum as objective, then how good can HC be solved? Two algorithms are proposed in this section.

4.2.1 Efficient and Effective Iterative Merging

With Theorem 4.1, a simple algorithm that comes up immediately is to iteratively merge two clusters with the smallest diameter resulted (Algorithm 4.1). *Complete-linkage clustering (c-link)* [117] is a popular agglomerative HC method where the merging in each iteration minimizes the maximum intra-cluster distance. It is noticeable that our algorithm is actually c-link under the Manhattan metric since the diameter of a cluster equals the the maximum intra-cluster distance in the Manhattan metric (Lemma 4.2).

Algorithm 4.1 is similar to Greedy-DME [46] except that the preference in selecting pairs is changed. The preference is changed from the distance between two merging



(a) Our HC with diameter sum $= 4 + 4 + \underline{12} + 21 = 41$.

(b) Our ZST with wirelength $= 4 + 4 + \underline{10} + 13 = 31$.

(c) Greedy-DME HC with diameter sum $= 4 + 4 + \underline{13} + 21 = 42$.

(d) Greedy-DME ZST with wirelength $= 4 + 4 + \underline{9} + 14.5 = 31.5$.

Figure 4.5: 1D example showing the impact of different preferences in iterative merging. Ours prefers small diameter of merged cluster, while Greedy-DME prefers small distance between two merging segments.

segments to the diameter of the merged cluster. The diameter objective in our iterative merging has more global view and tends to generate better results. In Section 4.4, the experimental results (Table 4.1 and Figure 4.8) will evidence the difference. Here, an 1D example in Figure 4.5 shows the idea. Note that in 1D, a "merging segment" degenerates to a point. In the first two iterations, both methods get clusters $\{p1, p2\}$ and $\{p3, p4\}$. In the third iteration, there are two choices (ignoring the bad choice of merging $\{p1, p2\}$ and $\{p_5\}$). The first is merging $\{p3, p4\}$ with $\{p_5\}$, which gives a merged diameter of 12 and a merging segment distance of 10. The second choice is merging $\{p1, p2\}$ and $\{p3, p4\}$, which gives a merged diameter of 13 and a distance of 9. It turns out that the first choice with a smaller diameter is better with an eventual wirelength of $31 < 31.5$. The detailed structural insight is in the proof of Theorem 4.1.

Moreover, Algorithm 4.1 is an $O(1)$-approximation for ZST/HC. Before showing it, we first introduce the non-hierarchical clustering problem, which serves for the HC lower bound. The *diameter k-clustering* [119–121] on points P is to partition P into k clusters and minimize the maximum diameter of the k clusters. The problem is NP-hard [119] and has a simple 2-approximation algorithm [120], where the approximation ratio is tight for l_1 distance [121]. A lower bound of HC on P is thus the diameter sum of a series of optimal k-clustering on P with $k = 1, 2, ..., |P| - 1$. Essentially, in ZST/HC, every Steiner node v can correspond a unique k-clustering with the maximum diameter being $d(leaves(v))$. The mapping can be obtained as follows. First, sort the Steiner nodes by $d(leaves(v))$ in descending order into $v_1, v_2, ..., v_{|P|-1}$. To get k clusters, we split at $v_1, v_2, ..., v_{k-1}$. Take the ZST/HC in Figures 4.5(a) and 4.5(b) as an example. Splitting at v_1 and v_2 leads to the 3-clustering of $\{\{p_1, p_2\}, \{p_3, p_4\}, \{p_5\}\}$, where the maximum diameter is $d(leaves(v_3))$.

According to Theorem 24 of [122], in the l_p space, any k-clustering induced by c-link is an $O(1)$-approximation of the optimal k-clustering. Together with Theorem 4.1 and the above lower bound for ZST/HC, we have the following theorem.

Theorem 4.2. *Algorithm 4.1 is an $O(1)$-approximation for ZST.* [2]

Regarding the efficiency of Algorithm 4.1, it can be done in $O(n \log n)$ time and $O(n)$ space [124] with the help of the data structure for dynamic closest pair queries [125].

4.2.2 Other Approximation Algorithms

Based on the 2-approximation algorithm [120] for k-clustering, Dasgupta proposes a deterministic algorithm and a randomized algorithm for constructing HC under any metric space [126]. The induced k-clustering at every level of the HC is at most 8 times and $2e \approx 5.44$ times the optimal k-clustering respectively. This directly leads to an 8-approximation and a 5.44-approximation for ZST based on the same argument for Theorem 4.2. However, the method is for arbitrary metric. In Manhattan space, it does not perform better than our iterative merging.

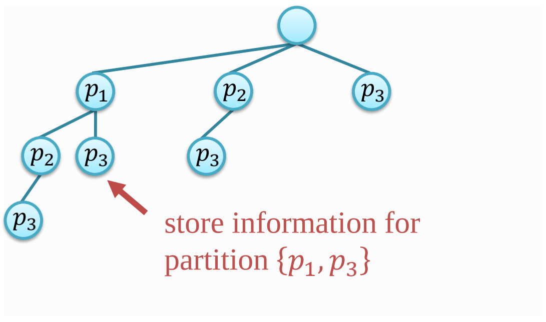
4.2.3 Optimal Dynamic Programming

In the ZST/HC problem, it is easy to show that an optimal ZST consists of two optimal sub-ZSTs (the two subtrees at the root), which enables a dynamic programming approach. The general idea is to expand the optimal solutions sink after sink, where the optimal ZST on a set P' is obtained by checking every bipartition (P_A, P_B) of P' and looking up the optimal ZSTs on subsets P_A and P_B.

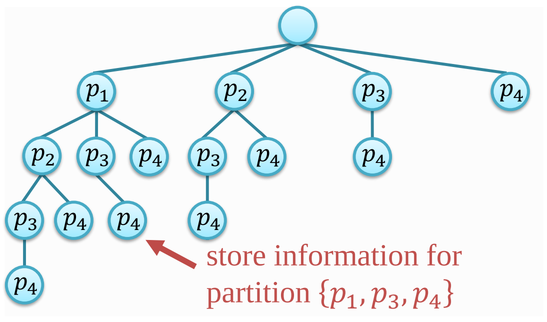
The major obstacle of such approach is the large number ($O(2^{|P'|})$) of possible bipartitions. By regarding ZST as HC, this dynamic programming approach however becomes more practical. First, it is more efficient to compute the diameter cost. Moreover, many inferior partitions can be identified and thus pruned (e.g., by the upper bound condition).

Trie (prefix tree) [127] is used for efficiently storing and retrieving the optimal sub-ZST/HCs (see Figure 4.6). Each node in the trie represents a partition/subset $P' \subseteq P$ of the input sinks P. Suppose $P' = \{p_{\pi_1}, p_{\pi_2}, ..., p_{\pi_i}\}$ and $\pi_1 < \pi_2 < ... < \pi_i$. The optimal cost and bipartition of P' will be stored at the corresponding trie node, which is at the end of the path $(p_{\pi_1}, p_{\pi_2}, ..., p_{\pi_i})$ (starting from the root of the trie). For example in Figure 4.6(a), the optimal cost of partition $\{p_1, p_3\}$ is stored at the end of path (p_1, p_3).

[2]The hidden constant here is forwarded from Theorem 24 of [122] and is not small. We conjecture that a much tighter approximation ratio exists. In the experiment on realistic and random cases, the gap between Algorithm 4.1 and the optimal solution observed almost never exceeds 10%. The largest gap encountered on artificial cases is 22% (Fig. 11 of [123]).



(a) Trie for three sinks $\{p_1, p_2, p_3\}$



(b) Trie for four sinks $\{p_1, p_2, p_3, p_4\}$

Figure 4.6: Trie for storing and retrieving the optimal sub-ZST/HC.

In such a trie data structure, a partition P' takes $O(1)$ memory only. Retrieving it takes $O(|P'|)$ time, which can be amortized to $O(1)$ if conducted incrementally.

The optimal trie-based dynamic programming (Algorithm 4.2) is as follows. Sinks are added one after another (line 4). For each new sink p_k, every existing trie node (i.e., partition) P' is expanded with it (line 5). To obtain the optimal ZST/HC on the expanded partition $P'' = P' \cup \{p_k\}$, all the bipartitions of P'' are enumerated (line 9). For each of the bipartitions, the costs of the two partitions need to be retrieved from the trie in order to compute the cost after merging (line 10). Note that one of the two partitions can have its cost retrieved incrementally by following a traversal on the trie to save runtime. An important issue for the dynamic programming is to order the generation of the trie nodes properly. The order of expanding the trie nodes should guarantee that a partition will have all of its sub-partitions expanded before its own expansion and with their costs available for looking up. The pre-order traversal that iterates children with larger indexes first will suffice.

There is an effective pruning technique in Algorithm 4.2, which is by an upper bound of the ZST/HC cost. Refer the optimal ZST/HC on points P as $T^*(P)$ and the cost (diameter sum) of a ZST/HC T as $cost(T)$. At the beginning of the dynamic programming, Algorithm 4.1 is invoked to generate a sub-optimal ZST/HC T' on input sinks P (line 1). Its diameter sum $cost(T')$ can then serve as an upper bound for pruning according to Theorem 4.3. Note that the order of adding sinks influences the effectiveness of the pruning. If early sinks tend to form clusters with larger diameters, the upper bound condition will be triggered more frequently. We adopt the order of the furthest-first traversal [120], which maximizes the minimum pairwise distances of the first k points during the traversal.

Theorem 4.3 (Upper Bound Condition)**.** *In the dynamic programming on input points P, suppose a bipartition of a node/partition P' is $P_A = \{p_{\pi_1}, ..., p_{\pi_i}\}$ with $P_B = \{p_{\sigma_1}, ..., p_{\sigma_j}\}$ ($P = P_A \cup P_B$ and $\pi_i < \sigma_j$). If the sum of their optimal cost exceeds the upper bound, i.e., $cost(T^*(P_A)) + cost(T^*(P_B)) > cost(T')$, then the sub-trie rooted at node P_B can be pruned.*

Algorithm 4.2 Optimal ZST by Dynamic Programming

```
Require: Sinks $P$ with $|P| = n$
Ensure: Optimal ZST $T$
 1: Compute ZST/HC $T'$ on $P$ by Algorithm 4.1 as an upper bound
 2: Get the furthest-first traversal $(p_1, p_2, ..., p_n)$ of $P$
 3: Initialize an empty trie
 4: for all $k = 1, 2, ..., n$ do
 5:     for all existing trie node $P'$ do                ▷ By pre-order traversal
 6:         Skip if $P'$ is marked as inferior
 7:         Initialize new trie node $P'' = P' \cup \{p_k\}$
 8:         Mark & skip if $P''$ violates convex hull condition
 9:         for all bipartition of $P''$ do
10:             Retrieve sub-partition cost from the trie
11:             Mark if violating upper bound condition
12:             Update the optimal cost & solution of $P''$
13:         end for
14:         Add $P''$ to be a child of $P'$
15:     end for
16: end for
17: Obtain the optimal HC by backtracking from trie node $P$
18: Run DME top-down phase on the HC to realize the ZST $T$
```

Algorithm 4.3 ZST Refinement by Dynamic Programming

```
Require: Size of enumeration $k$, ZST vertex $v$ ($|leaves(v)| \geq k$)
Ensure: Refined ZST beneath $v$
 1: Vertices $U \leftarrow v.children$
 2: while $|U| < k$ do
 3:     $u^* \leftarrow \arg\min_{u \in U} d(leaves(u))$
 4:     $U \leftarrow U - u^* + u^*.children$
 5: end while
 6: Get the optimal ZST/HC above $U$ by Algorithm 4.2
```

Proof. See Section 4.1.5. □

Besides constructing an optimal ZST, Algorithm 4.2 can be generalized for improving a ZST/HC T (Algorithm 4.3). For a vertex $v \in T$, the ZST/HC beneath it can be refined by local reconstruction. Essentially, the hierarchy below v is first recursively broken to obtain k descendants U with their corresponding clusters $\{leaves(u)|u \in U\}$ having small diameters (lines 1–5). The optimal local HC above U is then obtained by Algorithm 4.3. The original Algorithm 4.3 runs on points/sinks, but it can be trivially extended for working on several clusters. Note that the local change here has no impact to the topologies either beneath U or above v according to Lemma 4.4. By setting k as a constant (eight in our implementation) and running Algorithm 4.3 on every $v \in T$ (with $|leaves(v)| \geq k$ to make sure k descendants exist), the refinement takes $O(n)$ time.

Algorithm 4.4 BST by Combining Light Topology with ZST

```
Require: Sinks $P$, skew bound $b$
Ensure: BST $T_B$
1: Construct light topology $T_M$ on $P$ (e.g., by RSMT heuristics)
2: Forests $F_M \leftarrow$ decomposition of $T_M$ with maximum distance being $2b$ by Algorithm 4.5
3: Taps $P_M \leftarrow$ centers of $F_M$
4: $T \leftarrow$ ZST on $P_M$ by Algorithm 4.1
5: return $T_B \leftarrow T \cup F_M$
```

4.3 Bounded-Skew Tree Construction

Inspired by [47,48], our bounded-skew tree (BST) construction is to graft light topologies on a backbone of ZST. Compared with BST/DME [50], there are two significant differences. First, the flexibility of skew tolerance is put on lower levels (closer to leaves) instead of allocating over all levels by chance. Second, the algorithm is a simple combination of the efficient black-box constructions for light topologies and ZSTs, which means ease of implementation and runtime efficiency.

The algorithmic flow is described by Algorithm 4.4. The routing topology on the input sinks P is initialized to be a very light one T_M, e.g., generated by the rectilinear Steiner minimal tree (RMST) heuristics like FLUTE [25] (line 1). In order to satisfy the skew bound b, T_M is decomposed into several subtrees, where the distance between any two vertices in the same subtree along tree edges does not exceed $d_{max} = 2b$ (line 2). For each of the obtained subtrees, there thus exists a *center* that can reach all the vertices of the subtree within a distance of b. The centers P_M are then used as taps connecting to the clock source by a ZST T (line 4). In this way, distances from the source to sinks P are between $p(T)$ and $p(T) + b$, where $p(T)$ is the path length of T (from the clock source to P_M). A BST is thus resulted.

The core algorithm that bridges RSMT and ZST is the optimal tree decomposition (Algorithm 4.5). It decomposes an input tree T_M into a minimum possible number of subtrees under the distance constraint d_{max}. A post-order traversal on T_M is run after a *root* is specified (line 3). The information that a vertex u passes to its parent v is $u.depth$, which is the longest path from v to the current subtree of u (after possible decomposition of its original subtree). There may be multiple optimal decompositions for the original subtree of u. During the algorithm, $u.depth$ is made as small as possible among them. Lemma 4.5 explains how the invariant on $u.depth$ is kept by the greedy condition (line 12). The proof is omitted. By induction, we can prove Theorem 4.4 on the optimality of the whole algorithm.

Lemma 4.5. *In Algorithm 4.5, for children u_i and u_j of vertex v, if $u_i.depth + u_j.depth > d_{max}$, at least one of them should be decomposed from v.*

Algorithm 4.5 Optimal Tree Decomposition

```
Require: Tree $T_M(V_M, E_M, w_M)$, maximum distance $d_{max}$ allowed within each subtree
Ensure: Forest $F_M$ with number of trees minimized
 1: $F_M \leftarrow T_M$
 2: $v.depth \leftarrow 0, \forall v \in V_M$
 3: Make $T_M$ rooted by specifying an arbitrary vertex as $root$
 4: PostOrderTraversal($root$)
 5: function PostOrderTraversal($v$)
 6:     $v.depth \leftarrow w_M(v, v.parent)$ and return if $v$ is a leaf
 7:     for all $u \in v.children$ do
 8:         PostOrderTraversal($u$)
 9:     end for
10:     Sort $v.children$ by descending $depth$ to $u_1, u_2, ...$
11:     for $i = 1, 2, ..., |v.children| - 1$ do
12:         if $u_i.depth + u_{i+1}.depth > d_{max}$ then
13:             Remove edge $(u_i, v)$ from $F_M$
14:             $u_i.depth \leftarrow 0$
15:         end if
16:     end for
17:     $v.depth \leftarrow \max_{u \in v.children} u.depth + w_M(v, v.parent)$
18: end function
```

Theorem 4.4. *For a tree T, Algorithm 4.5 generates a decomposition with the minimum number of subtrees.*

The time of Algorithm 4.5 is $O(n)$, because each vertex is processed once and in $O(1)$ time. Note that the sorting in line 10 is in constant time due to the bounded degree of a vertex in RSMT. Moreover, it can be avoided by scanning *v.children* a few passes. Essentially, all children with $depth \leq \frac{1}{2}d_{max}$ can always be kept, while all children with $depth > \frac{1}{2}d_{max}$ should be split except at most one.

4.4 Experimental Results

We implement our ZST and BST construction methods by C++ on a 3.8 GHz Linux machine. There are two kinds of benchmarks tested. The first is the realistic benchmark (prim1-2 from [42], r1-r5 from [45], and ISPD 2019 and 2010 contest benchmarks [128, 129]). The second is a large number of random nets with different number of sinks. We generate 100 nets for each number of sinks under a uniform distribution.

For ZST construction on realistic cases (Table 4.1), our iterative merging (Algorithm 4.1, Dim Sum) is compared with other topology generation methods including MMM [42], Rooted Kruskal [48], geometric merging algorithm (GMA) [43], balance bipartition (BB) [44], and Greedy-DME [46]. The results of GMA and BB are cited from [44]. The others are implemented by ourselves. The only implementation difference between Greedy-DME and Dim Sum is the score of merging a pair (the distance between two

Table 4.1: Wirelength Comparison of ZST Methods on Realistic Benchmarks (unit: μm)

Benchmarks	[42]		[45]					Avg.
	p1	p2	r1	r2	r3	r4	r5	ratio
# sinks	269	594	267	598	862	1903	3101	
MMM + DME	149	373	1601	3240	4165	8218	12274	1.248
Rooted Kruskal + DME	142	341	1461	2837	3711	7656	11052	1.136
GMA + DME	140	350	1497	3013	3902	7782	11665	1.173
BB + DME	141	361	1500	3010	3908	8000	11757	1.185
Greedy DME	133	314	1313	2566	3339	6707	9943	1.028
Dim Sum w/o refinement	131	309	1297	2568	3285	6631	9801	1.015
Dim Sum w/ refinement	**128**	**306**	**1272**	**2506**	**3248**	**6545**	**9711**	**1.000**

Benchmarks	ISPD 2009 [128]											ISPD 2010 [129]								Avg.
	f11	f12	f21	f22	f31	f32	f33	f34	f35	fnb1	fnb2	1	2	3	4	5	6	7	8	ratio
# sinks	121	117	117	91	273	190	209	157	193	330	440	1107	2249	1200	1845	1016	981	1915	1134	
MMM + DME	186	171	205	113	439	321	315	270	324	45.2	120	355	636	57.8	126	66.3	49.6	93.3	61.9	1.333
Rooted Kruskal + DME	190	171	202	116	420	314	317	257	290	41.5	110	306	564	31.2	99.8	45.6	38.5	71.1	44.2	1.138
Greedy DME	174	156	192	105	380	291	292	241	273	36.6	96.6	277	510	28.2	87.4	40.4	34.0	62.6	40.2	1.032
Dim Sum w/o refinement	176	157	184	106	375	283	287	238	268	35.0	95.4	273	504	27.8	86.6	40.0	33.3	61.5	39.2	1.016
Dim Sum w/ refinement	**171**	**153**	**182**	**103**	**369**	**281**	**282**	**236**	**264**	**34.7**	**93.6**	**269**	**496**	**27.6**	**85.5**	**39.0**	**32.7**	**60.6**	**38.8**	**1.000**

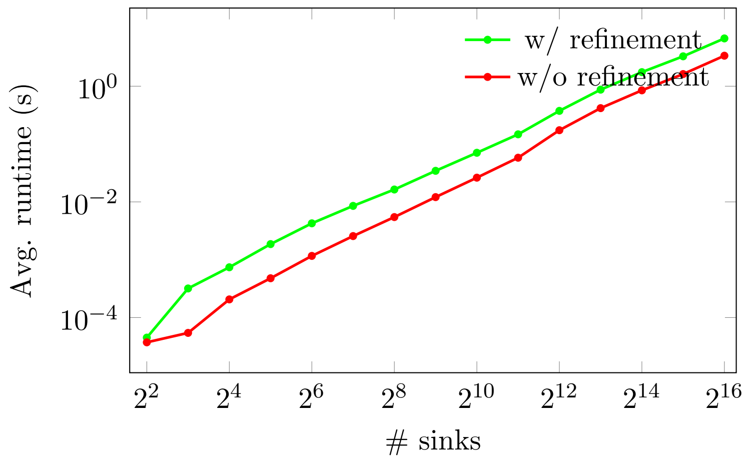
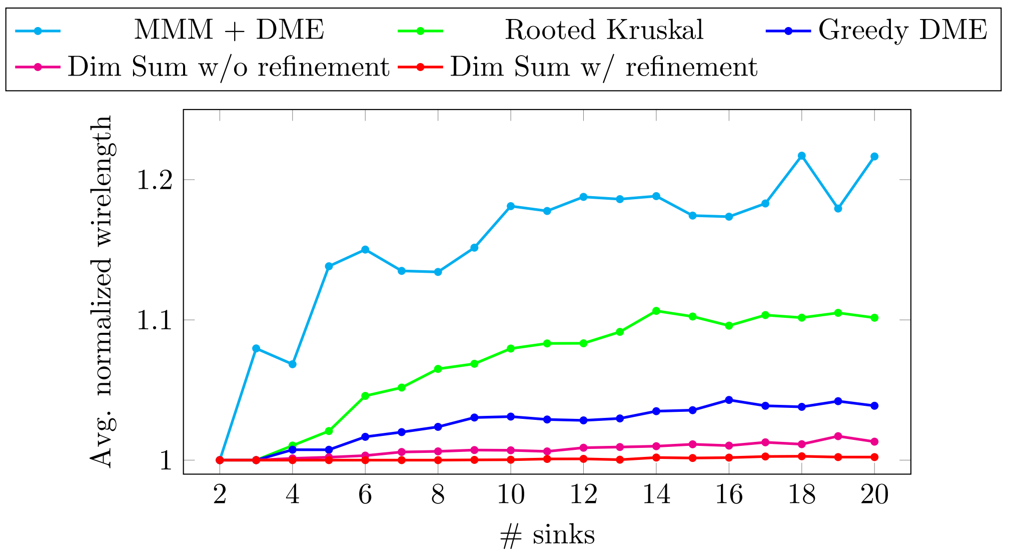


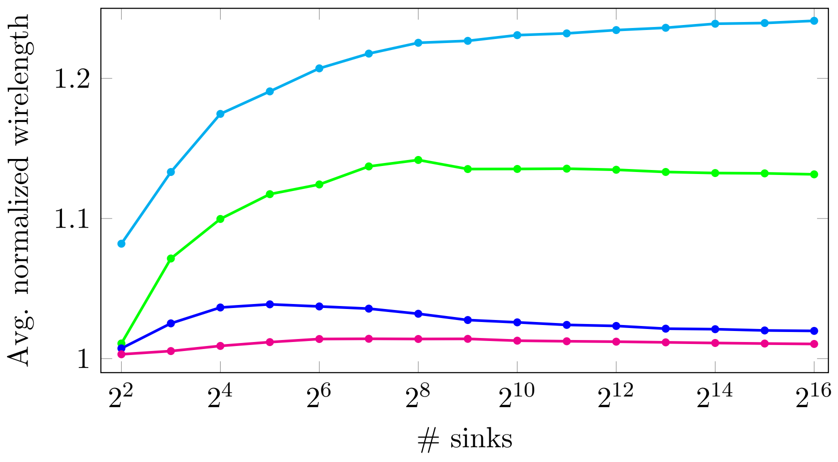
Figure 4.7: Runtime of Dim Sum.

merging segments v.s. the diameter of the merged cluster), where the simple change leads to 1.5% improvement on average. Together with the dynamic-programming-based refinement, the wirelength is furthered improved by 1.6%.

To systematically know the optimality of the methods, a batch test is conducted with random nets. A solid reference is the optimal ZST generated by the dynamic programming (Algorithm 4.2, Optimal Dim Sum). But it does not scale and is thus used on the small nets with up to 20 sinks. Even though Optimal Dim Sum is not polynomial-time, it is significantly more efficient than the ILP in [56]. The optimal BST by this ILP needs more time as skew bound decreases; for a relative skew bound of 0.3 and on 16 sinks, it already takes 3477.73 s [56]. Meanwhile, Optimal Dim Sum takes only 1.08 s for 16 sinks. In Figure 4.8(a), each data point represents the average of the normalized wirelength on 100 nets. As the number of sinks increases, the average gap from Dim Sum to Optimal Dim Sum becomes larger. However, for 20 sinks, Dim Sum (with refinement) still gets a small



(a) On small nets. Wirelength normalized by Optimal Dim Sum.

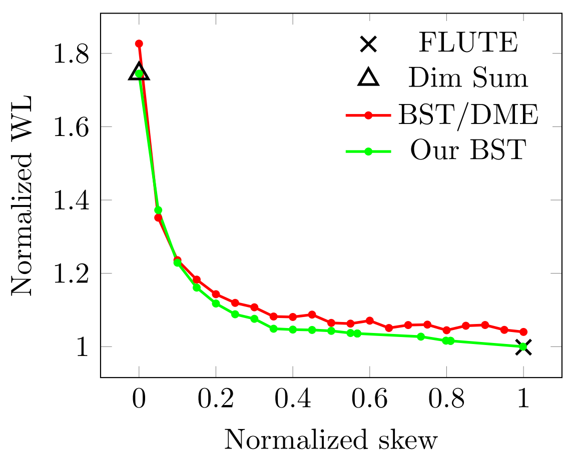


(b) On large nets. Wirelength normalized by Dim Sum w/ refinement.

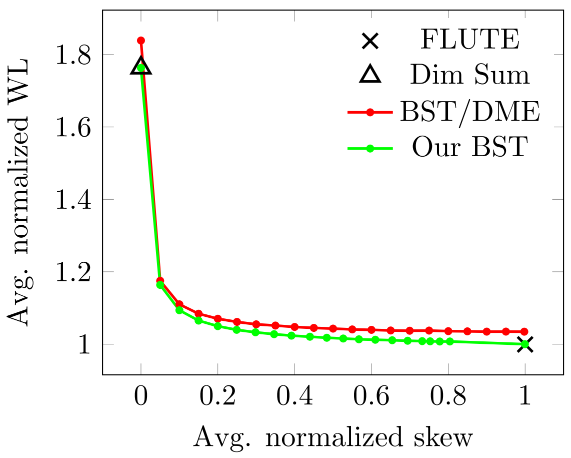
Figure 4.8: Wirelength comparison of ZST methods on random nets.

gap of 0.22%, while that of Greedy-DME is as large as 3.88%. Besides, note that Optimal Dim Sum on average takes 24.6 s with the upper-bound pruning and 149.7 s without pruning, but Dim Sum takes 0.7 ms. With more sinks (up to 65536 in Figure 4.8(b)), Dim Sum with refinement is used as the reference for normalization. The improvement over Greedy-DME reaches the peak of around 4% at 32 sinks. Figure 4.7 shows the scalablity of Dim Sum. For 65536 sinks, even with refinement, it still needs 6.71 s only.

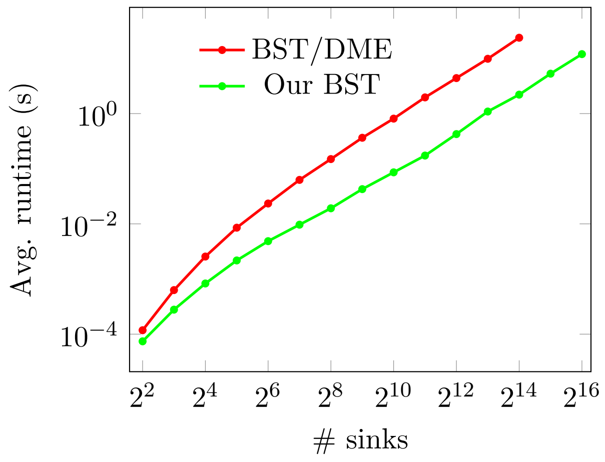
We implement our BST method (Algorithm 4.4) with RSMT heuristics FLUTE [25] for initial topologies. Figure 4.9 shows the comparison with BST/DME [50,130]. The implementation of BST/DME is obtained from its authors [131]. We use the wirelength and skew of FLUTE to normalize the result on each net. The input skew bound is set to 0, 0.05, 0.1, ..., 1 of the skew of FLUTE. Due to the space limit, we only show two skew-wirelength trade-off curves. One is for r5, the largest net among those publicly-available cases (Fig-



(a) Wirelength on case `r5` with 3101 sinks.



(b) Wirelength on 100 random nets with 16384 sinks.



(c) Runtime on random nets with various sizes.

Figure 4.9: Comparison between BST/DME and our BST construction method.

ure 4.9(a)). The other is for the 100 random nets with 16384 sinks (Figure 4.9(b)). Here, our method shows better Pareto frontiers compared with BST/DME. In general, as the skew bound becomes larger, a smooth decrease of wirelength from ZST to RSMT can also be observed. Besides, our method is significantly more efficient than BST/DME. For each method and each number of sinks, Figure 4.9(c) shows an average runtime of all 100 nets and all 21 skew bounds. For relatively large nets, a 10× speed-up stably exists.

Part III

Multiple-Net Routing

Chapter 5

Detailed Routing

In this chapter, we proposes Dr. CU, a detailed routing framework that is superiorly scalable in runtime as well as memory usage and provides more correct-by-construction design rule satisfaction. Our contributions can be summarized as follows.

- We design a set of two-level sparse data structures for a 3D detailed routing grid graph of enormous size.
- We develop an optimal correct-by-construction path search that captures the minimum-area constraint.
- We also propose an efficient bulk synchronous parallel scheme to further reduce the turn-around time of the detailed routing process.

The source code of Dr. CU is also publicly available at `https://github.com/cuhk-eda/dr-cu`.

The remainder of this chapter is organized as follows. Section 5.1 introduces the formulation of the VLSI detailed routing problem. Section 5.2 and Section 5.3 provide the details of our data structures and algorithms respectively. Section 5.4 describes the parallel scheme. In the end, Section 5.5 shows the experimental results.

5.1 Preliminaries

Before illustrating the details of our data structures and algorithms, the problem formulation of detailed routing is introduced in this section.

5.1.1 Routing Space

VLSI Routing is on a stack of *metal layers*. A *wire* segment on a layer runs either horizontally or vertically. Each layer has a *preferred direction* for routing, which benefits manufacturability [80], routability and design rule checking [74]. The preferred directions

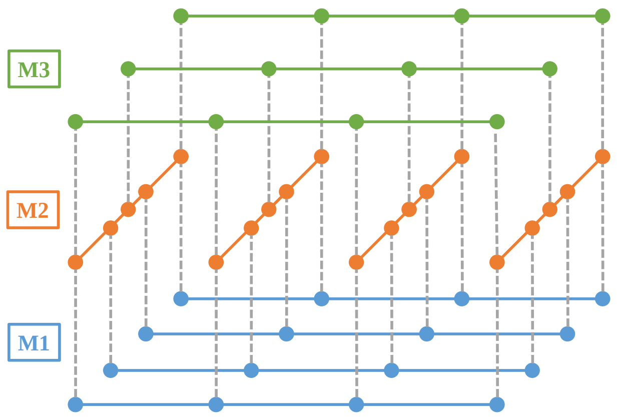


Figure 5.1: An example 3D detailed routing grid graph. Here, preferred directions of metal 1 (M1) and M3 layers are both horizontal, while that of M2 is vertical.

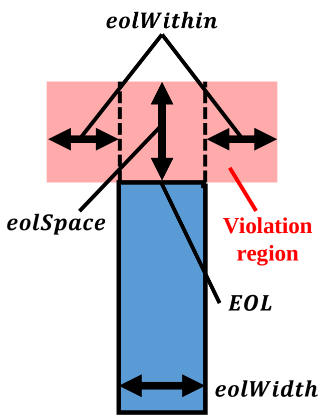
of adjacent layers are perpendicular to each other in common design practice. Besides, regularly-spaced *tracks*, where the majority of wires are routed on, can be predefined according to the wire width and parallel-run spacing constraint. In this work, wrong-way and off-track wires are considered only for short connections (especially to pins).

Wires on adjacent metal layers can be electrically connected by *vias*. A via is across a *cut layer*, which is between the two metal layers. Note that for vias across a specific cut layer, there may be several via types to be selected from. Different via types have varied metal shapes (usually rectangles with various widths and heights) on the two metal layers. The flexibility provides a way for resolving the spacing violations between vias and obstacles.

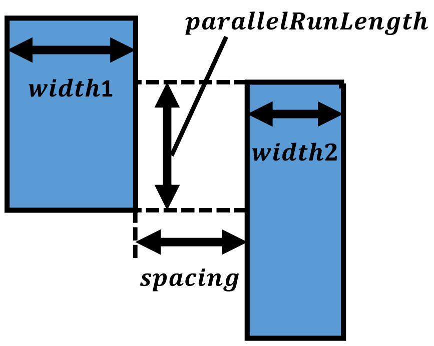
The tracks on all metal layers define a 3D grid graph for detailed routing, as Figure 5.1 shows. On each track, there is a series of vertices. Note that a vertex is therefore uniquely defined by a 3D index, which is a tuple of layer index, track index (in the non-preferred direction), and relative index along the track (in the preferred direction). A vertex connects downwards to the lower layer, upwards to the upper layer, or both. Adjacent vertices along a track are also connected. In this way, a same-layer edge represents a possible on-track wire segment, while a cross-layer edge represents a possible via. In this grid graph, an *edge* represents either a wire segment or a via.

Over the chip, there are some routing *obstacles* that vias and wire segments should avoid to prevent short and spacing violations. In detailed routing, the relatively small obstacles within standard cells (e.g., pins and intra-cell wires) should also be handled.

Assuming that a global routing result is already well optimized for certain metrics (e.g., timing, routability, power), a detailed router needs to honor the global routing result as much as possible. The optimized metrics are thus kept with detailed design rules handled. In this work, the 3D global routing result is referred as *routing guide*, and out-of-guide routing (either wire or via) is penalized.



(a) End-of-line (EOL) spacing



(b) Parallel-run spacing

Figure 5.2: Examples of spacing violations.

5.1.2 Design Rules

The most fundamental and representative design rules handled by detailed routing are as follows [75].

- *Short.* A via metal or wire metal cannot overlap with another metal object like via metal, wire metal, obstacle, or pin, except when the two metal objects belong to the same net.
- *End-of-line (EOL) spacing.* A metal end is an EOL if its width is shorter than *eolWidth*. EOL is required to preserve a spacing greater than or equal to *eolSpace* beyond the EOL anywhere less than the *eolWithin* distance, as Figure 5.2(a) shows.
- *Parallel-run spacing.* For two metal objects with *parallelRunLength* (i.e., the projection length between them), there is a spacing requirement, as Figure 5.2(b) shows. The value of parallel-run spacing rule depends on the widths of the two metal rectangles.
- *Cut spacing.* For vias across the same cut layer, their cut shapes in the cut layer should be sufficiently far away from each other.
- *Minimum area.* The area of a metal polygon is required to be above a threshold.

5.1.3 Problem Formulation

The detailed routing problem can be formally defined as follows. Given a placed netlist, routing guides, routing tracks, and design rules, route all the nets and minimize a weighted sum of

- Total wirelength,

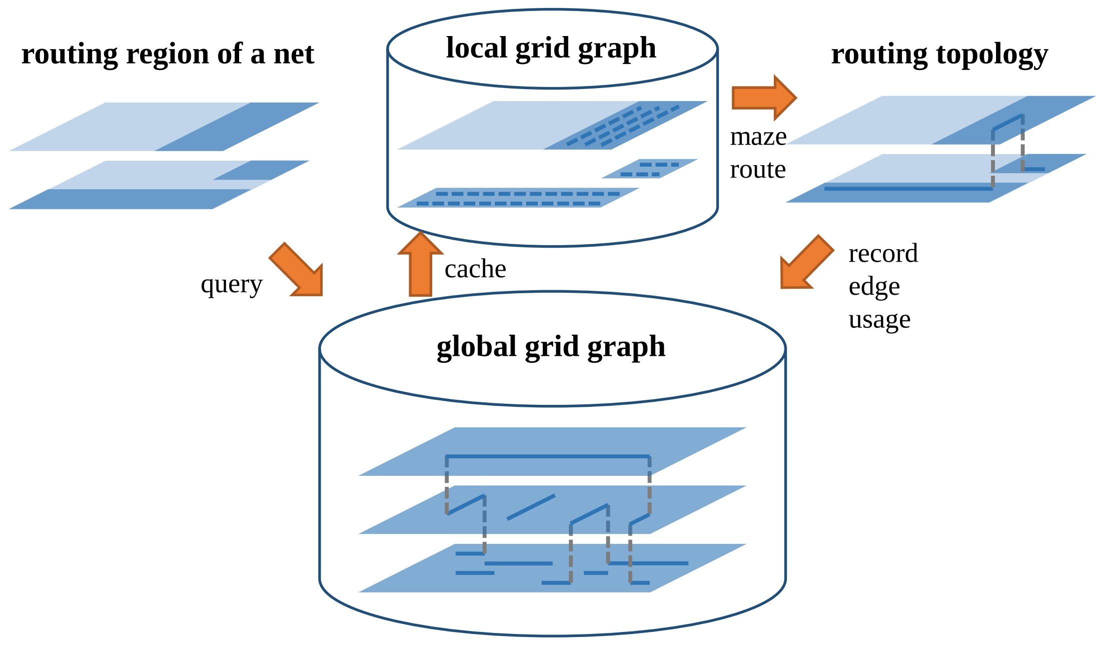


Figure 5.3: Overview of the two-level grid graph data structures.

- Total via count,
- Non-preferred usage (including out-of-guide and off-track wires/vias, and wrong-way wires), and
- Design rule violations (including short, spacing and minimum-area violations).

Note that design rule violations are highly discouraged and suffer much more significant penalty than others.

5.2 Two-Level Sparse Data Structures

The grid graph for detailed routing is similar to that for global routing in structure, but is significantly more fine-grained and thus has much a larger scale. To support the detailed routing algorithms with both economic memory usage and efficient query, we design a set of two-level data structures for the routing grid graph.

There are a global grid graph and local ones, as Figure 5.3 shows. The *global grid graph* data structure stores the graph implicitly without instantiating all vertices. Here, the information of routed edges are stored sparsely by balanced binary search trees (BSTs) and intervals. The *local grid graph*, a local cache of the global one, is created for routing a net. It is a sparse subgraph of the full-chip 3D grid graph on the routing region of a net, where edge costs are readily available for conducting maze routing.

5.2.1 Sparse Global Grid Graph

Edges of routed nets are called *routed edges*. Note that the an edge can be either a via or a wire segment. The global grid graph stores routed edges in the sparse data structure based on BSTs and intervals.

5.2.1.1 BST and Interval Based Storage

It is very expensive to use a full-chip 3D direct-address table for storing routed edges. First, its size will be unaffordable (10^9 vertices for just 10 metal layers and 10^4 tracks on each layer) [72]. Besides the time-consuming memory allocation and initialization, some queries are also inefficient if using this data structure. For example, to record, query or remove the usage of a wire segment (e.g., spanning 10^3 vertices), we need to change or check all the 10^3 vertices on it.

Instead of a 3D direct-address table, we use a 2D table for the dimension of layers and the dimension of tracks (i.e., the non-preferred direction), and use BST and intervals in the third dimension (i.e., the preferred direction) for sparsity. For a track, there are three balanced BSTs, two for storing routed vias and one for storing routed wires. For vias, normal BSTs with indexes in the preferred direction being keys are used. Each via is stored twice, one on the lower track and the other one on the upper track. The duplication benefits the range searches that are needed on both the lower and upper tracks. This will be illustrated in detail later. For wire segments, a BST with nodes representing non-overlapping intervals is employed. In this way, the memory used is linear to the number of wire segments instead of the number of vertices involved.

5.2.1.2 Conflicts with Obstacles and Pins

For obstacles and pins with irregular shapes, the vias and wires that may cause short or spacing violations with them are marked in advance in batch. Since obstacles and pins cannot be ripped up, the marking is a one-time effort. Note that a conflict with a pin is net-dependent, because a via or wire is allowed to be close to a pin of the same net. Therefore, some conflicts should be associated with some possibly *excepted net(s)*.

Figure 5.4 shows an example of marking wires conflicted with obstacles and pins. For each obstacle or pin, there are several vertices in the grid graph that will cause short or spacing violations if a wire segment is routed through it. For an obstacle, the conflict applies to all nets (red crosses in Figure 5.4(b) indicate conflicts without exception); for a pin, the conflict applies to all nets but the net of the pin (yellow crosses in Figure 5.4(b) indicate conflicts with exception). However, the conflicts between a wire and the pins of different nets cannot be excepted. To save memory usage, we use an interval based storage here as well. Only conflicted vertices are stored, while violation-free vertices are

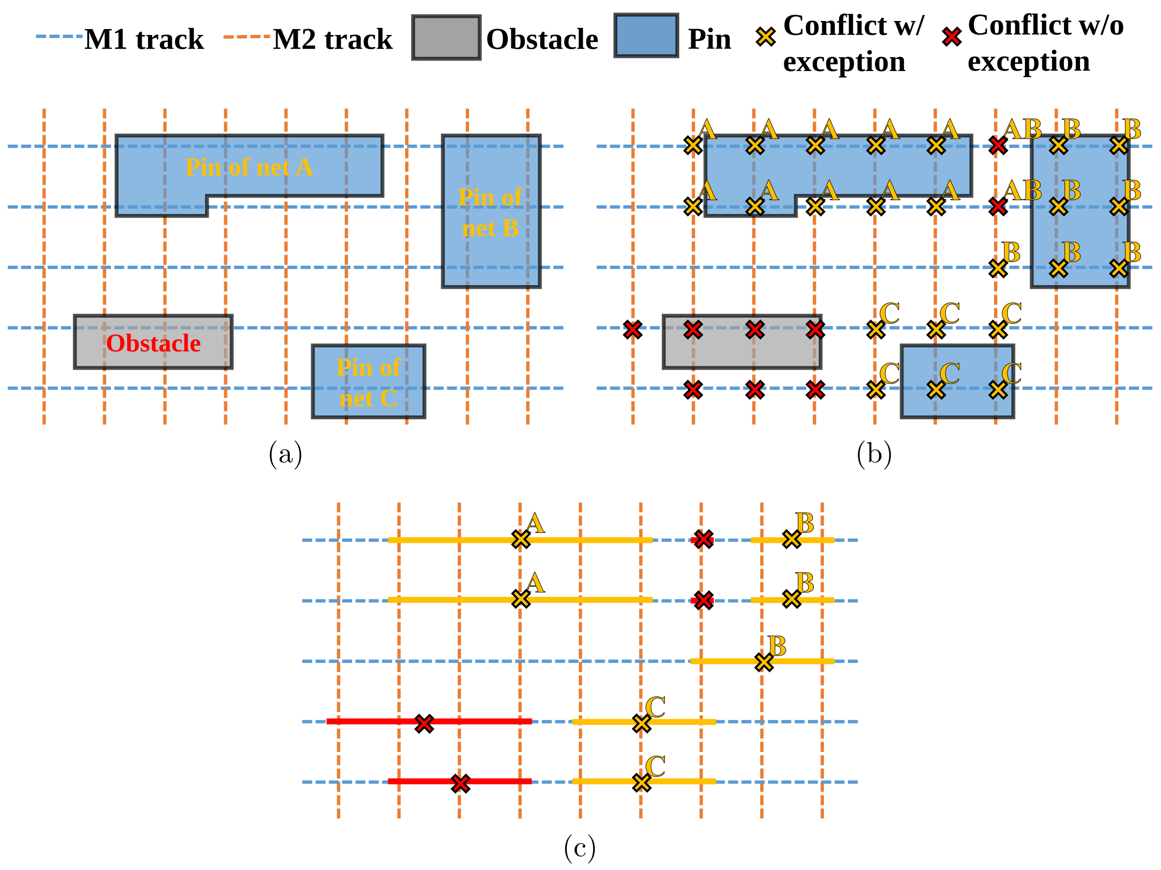


Figure 5.4: Wire-obstacle and wire-pin conflicts stored in global grid graph. (a) A region with an obstacle and three pins. (b) Wires conflicted with obstacles/pins, where a wire-pin conflict is excepted for the net of the pin, but wires conflicted with pins of different nets have no exception. (c) Interval based storage.

implied. For continuous vertices with the same-type of conflict along a track, they will be stored as an interval, as Figure 5.4(c) shows.

Via-obstacle and via-pin violations are more difficult to capture than wire-obstacle and wire-pin violations, because there are several types of vias that can be chosen from. Essentially, all via types need to be attempted. A via location should be penalized if and only if all via types fail to satisfy the spacing requirement with its neighboring obstacles or pins. Note that a via-pin conflict may be excepted for multiple nets due to the via type selection.

When routing a net, the vias that will be considered for using are referred as *candidate vias*. In the preliminary version [9] of this work, we simply store all the obstacles and pins in R-trees [113] and later query the via-obstacle and via-pin violations from the R-trees. For each candidate via of a net, its neighboring obstacles and pins are queried from the R-trees and checked for possible violations. There is a big drawback with this approach. A via may be treated as a candidate by many nets, resulting in repeated queries and checking processes for a single via. The aforementioned pre-computation scheme for conflicted vias can save runtime significantly, which will also be evidenced by

Table 5.1: Statistics of Via-Obstacle and Via-Pin Conflicts on `ispd18_test10`

Layer	Metal layer information			Cut layer information								
	# obtacle/pin rectangles			Pre-compute?	# via locations						# conflicted intervals	# conflicted intervals / # vias
	Obtacle	Pin	Total		Conflicted (w/o excepted nets)	Conflicted (w/ an excepted net)	Conflicted (w/ multiple excepted nets)	Conflicted (total)	Total	Conflicted / total		
1	839912	2107724	2947636	Yes	41559185	11818934	9608	53387727	189000000	28.247%	16075707	**8.506%**
2	763422	0	763422	Yes	38375423	0	0	38375423	189000000	20.304%	585913	**0.310%**
3	24092	0	24092	Yes	11665650	0	0	11665650	189000000	6.172%	11248534	**5.952%**
4	580772	0	580772	Yes	7537450	0	0	7537450	189000000	3.988%	16040	**0.008%**
5	0	0	0	No	-	-	-	-	-	-	-	-
6	0	0	0	No	-	-	-	-	-	-	-	-
7	0	332	332	No	-	-	-	-	-	-	-	-
8	0	879	879	No	-	-	-	-	-	-	-	-
9	0	0	0	-	-	-	-	-	-	-	-	-

the experiments in Section 5.5.

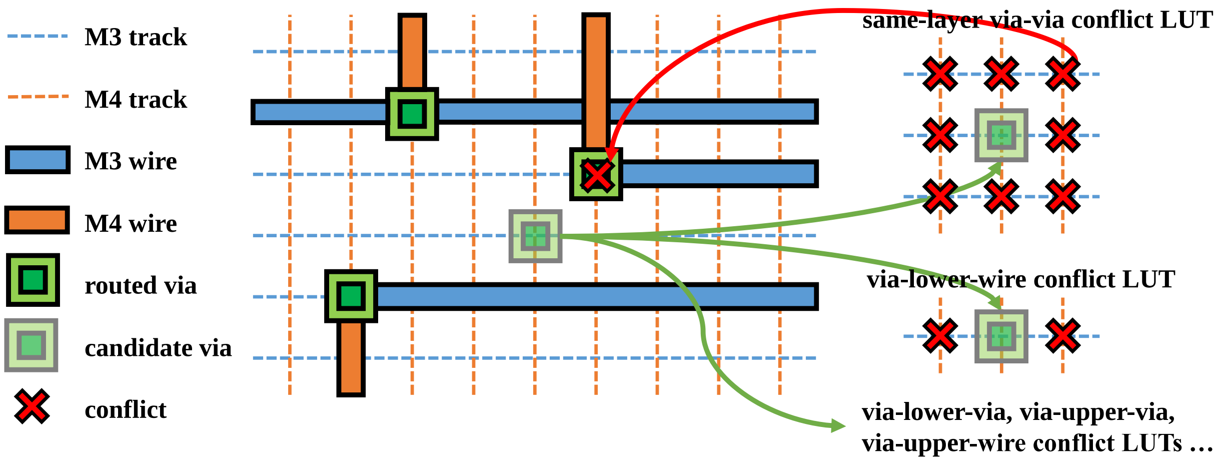
Three techniques are crucial for enabling such speed-up. First, we only perform the pre-computation for metal layers with huge numbers of obstacles and pins[1]. In our implementation, we set a lower bound threshold on the number of obstacle/pin metal rectangles to 10^5. For many designs, it means a pre-computation for one or two layers. Second, we store conflicted vias only, while violations-free vias are implied. The third technique is the usage of BST and interval based storage scheme. The statistics in Table 5.1 provides some evidence on the advantages of using these techniques. On `ispd18_test10`, metal layers 1, 2, and 4 have large numbers of obstacles and pins. Therefore, cut layers 1, 2, 3, and 4 need the pre-computation of via conflicts (cut layer i connects metal layers i and $i+1$). If storing the conflict situation for all via locations with direct-address tables, it means GB scale memory usage for a single layer (note that we need to store the information of excepted nets). Storing conflicted vias reduces the memory usage to 28.247% for cut layer 1. Using intervals further reduces the usage to 8.506%. For some layers, the reduction can be even much larger (to 0.008%).

5.2.2 Global Grid Graph Query by Look-up Table

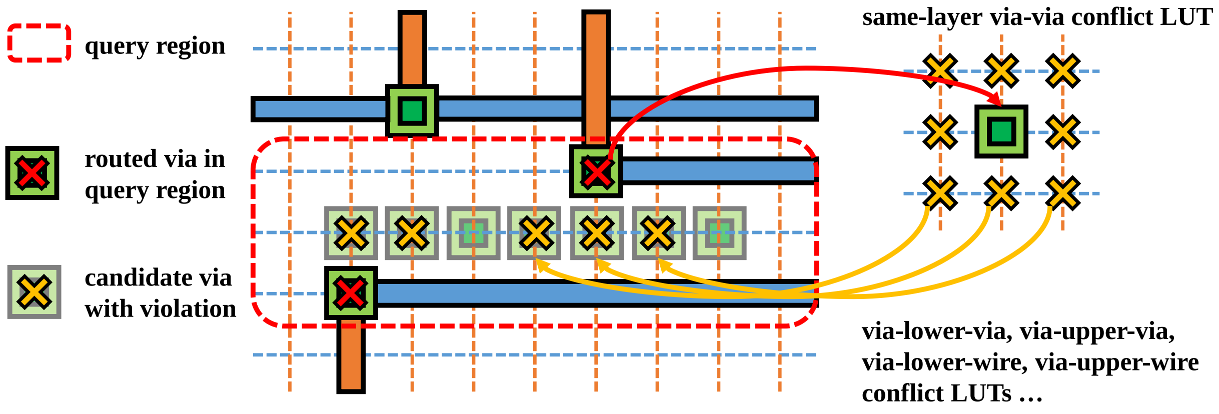
When routing a net, the edges that will be considered for using are referred as *candidate edges*. Their costs (possibly penalized by the short/spacing violations) will be queried from the global grid graph before running maze routing on a net.

Different from the conflict with obstacles, the conflict with routed edges will change during the routing process and cannot be marked in advance. Considering various design rules and a significant number of candidate edges, a proper scheme that can efficiently query their costs is in need. We build several via/wire conflict LUTs to achieve that.

[1]We focus the discussion on metal layers for simplicity. In ISPD 2018 benchmarks, which we use for the experiments, there is also no obstacle in cut layers. However, our method is generic and can be easily extended for considering violations in cut layers.


M3 track
M4 track
M3 wire
M4 wire
routed via
candidate via
conflict
same-layer via-via conflict LUT
via-lower-wire conflict LUT
via-lower-via, via-upper-via,
via-upper-wire conflict LUTs ...

(a) Query a single candidate via


query region
routed via in query region
candidate via with violation
same-layer via-via conflict LUT
via-lower-via, via-upper-via,
via-lower-wire, via-upper-wire
conflict LUTs ...

(b) Query a set of neighboring candidate vias

Figure 5.5: Query the violations on candidate vias due to the previously routed edges in global grid graph.

5.2.2.1 Via/Wire Conflict Look-up Table

For routing a net, the metal short with routed edges can be trivially detected as interval overlapping. For the following spacing violation conflicts, their identification is less straight-forward:

- Via-via conflicts: for a specific via, it may conflict not only with vias on the same cut layer (*same-layer vias*) but also with vias on the adjacent cut layers. The conflict between same-layer vias may be due to spacing rules on either cut layer, metal layers, or both. The conflict between different-layer vias is caused by metal spacing requirement.

- Via-wire conflicts: a via may have spacing violations with wires on the lower and the upper metal layers that it connects.

- Wire-wire conflicts: two wires may be too close to each other at their ends and violate the spacing constraint.

The above violations can be detected during routing. However, these detection operations are extremely frequent and on-the-fly detections are too time-consuming. Since we are working on a relatively regular grid graph, some light-weight LUTs can accelerate the process. Conceptually, *via/wire conflict LUTs* immediately tells what neighboring edges will conflict with a given edge. There are several types of them: when the given edge is a via e_i, a *via-lower-wire conflict LUT* tells what neighboring wire segments on the lower metal layer of e_i cause conflicts with e_i; similarly, given a wire segment e_j, a *wire-upper-via conflict LUT* tells what vias connecting to the layer above e_j may be conflicted with e_j; so on and so forth. Two conflict LUTs are called the *inverse LUT* to each other if the types of the given edge and the neighboring edges are swapped. For example, the inverse of a via-lower-wire LUT is a wire-upper-via LUT.

Regarding the indexing and sizes of conflict LUTs, we explain the via-via one as an example. For two same-layer vias, their distance is unique for specific track index differences in the lower metal layer and the upper metal layer, because of the equal spacing of the tracks. Therefore, only one LUT is needed for each layer. Such an LUT itself is 2D and is indexed by the track index differences. For two different-layer vias, three consecutive metal layers are involved. Using their corresponding vertices on the middle metal layer for indexing, their distance in the non-preferred direction is solely determined by the difference in track indexes. However, in the preferred direction, vertices along a track may have irregular spacing (e.g., M2 in Figure 5.1). As a result, a layer needs a series of 2D LUTs, where each LUT serves for vertices with a specific index in the preferred direction.

5.2.2.2 Single Edge Query

The cost of a candidate edge consists of a unit edge cost and some possible penalty caused by two types of violations. The first type is violations with obstacles and pins, which has been introduced in Section 5.2.1.2. The second type is violations with routed edges. The via/wire conflict LUTs tell the neighboring edge positions that will have conflict with the candidate edge. The only thing to do is to check whether the positions are occupied. An example is shown by Figure 5.5(a). For the candidate via, a same-layer via-via conflict is detected with the help of the corresponding LUT. Meanwhile, there is no via-lower-wire conflict because no routed wire exists at the two potentially conflicting positions specified by the LUT.

5.2.2.3 Batch/Long Edge Query

Usually, a set of neighboring edges (either vias or wire segments) along a track are all candidate edges for routing a net. If querying them individually, $O(k \log n)$ time is needed with k being the number of candidate edges and n being the BST size[2]. A range search on BST can improve the efficiency. Given a set of candidate edges along a track and the corresponding LUTs, a *query region* where routed edges may have conflicts with can be identified. By the range search on BSTs according to this query region and referring to the inverse LUTs, the conflicted candidate edges can be found. An example on detecting same-layer via-via conflict is illustrated by Figure 5.5(b). First, the query region and two routed vias within it are identified. Starting from the two routed vias, the inverse LUT (the same-layer via-via conflict LUT) finds five conflicted candidate vias.

Suppose the number of routed edges within the query region is m. The range search on a BST takes $O(m + \log n)$ time, which can be conducted by finding the first and last tree nodes within the range. Besides, $m = O(k)$. Note that m can be significantly smaller than k because a long routed wire segment is stored as a long interval instead of a bunch of short edges in a BST. Therefore, the time for retrieving the routing cost of the k candidate edges is $O(k) + O(m + \log n) = O(k + \log n)$ instead of $O(k \log n)$. Moreover, the cost of a long wire segment may be queried as a whole, then the time is further improved to $O(m + \log n)$.

In the batch query along a track, routed vias to both lower and upper layers should be considered. As mentioned in Section 5.2.1.1, a via is stored twice on both its lower and upper tracks. In this way, efficient BST range search along either track is enabled.

5.2.3 Sparse Local Grid Graph

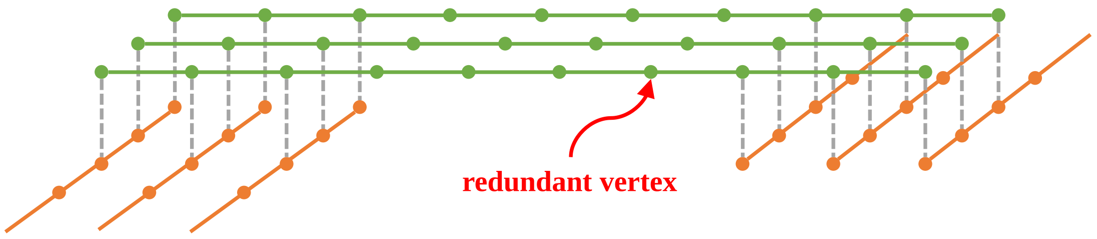
The local grid graph of a net is the subgraph of the full-chip 3D grid graph within its *routing region* (the routing guide with possibly minor expansion). In terms of data structures, it caches the graph structure and all edge costs of the subgraph by direct-address tables, supporting the maze routing.

Its sparsity is in two aspects. First, only the routing region is considered, which is substantially smaller than the full-chip region. Second, many vertices become redundant in this subgraph and are removed.

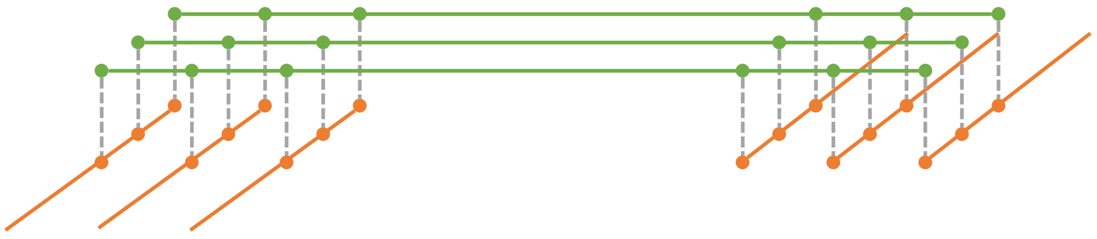
5.2.3.1 Routing Region

When routing a net, only the region around its routing guide is considered due to two reasons. First, detailed routing should honor global routing solution, i.e., routing guides, because many objectives (e.g., timing, routability) are optimized in global routing. For some local congestions, global routing may not be able to model and resolve, so minor out-of-guide routes may be necessary. However, such disturbance should be minimized. Second, maze routing on the full-chip 3D grid graph will suffer from prohibitive runtime

[2]To be more rigorous, since multiple BSTs (for vias or wires, for different layers) may all need to be queried, n represents the largest size of all BSTs.



(a) Before removing



(b) After removing

Figure 5.6: Long edges by removing redundant vertices.

due to its enormous scale. In our implementation, the routing region of a net is expanded by a small margin from its routing guide. All out-of-guide edges are penalized. For difficult-to-route nets, the expansion margin may be increased.

5.2.3.2 Long Edge

Conceptually, the local grid graph is simply a subgraph induced by vertices within the routing region. However, many vertices in the subgraph have only two neighbors remained and become redundant, as Figure 5.6(a) shows. In this snippet of the subgraph, many vertices originally have neighbors on adjacent layers that are out of the routing region now. They have thus only two neighbors left on the track. In this way, as long as such a vertex does not belong to a pin, it can be safely removed with the two connected edges merging into one. This compressing step cuts down the problem size without affecting the final results. Both memory usage and runtime can be reduced.

5.2.3.3 Wrong-Way Edge

Wrong-way edges are discouraged due to three reasons. First, more regular designs with fewer wrong-way usage is beneficial to manufacturability [80]. Second, a long wrong way edge will block many tracks, which hurts routabiltiy. Third, for routing a single net, heavy usage of wrong-way edges leads to a significantly larger solution space and thus runtime overhead.

But it should be allowed. In the preliminary version [9] of this work, we only try using wrong-way edges in some post processing steps. However, it turns out that adding some wrong-way edges in the local grid graph can greatly benefit escaping congested tracks. In our implementation, we add wrong-way edges densely in the small regions around pins.

Besides, along two neighboring tracks, a wrong way edge is added for every ten vertices. The improvement due to the wrong-way consideration will be shown in Section 5.5.

5.2.3.4 Explicit Storage

In the global grid graph, vertices are implied by 3D indexes but are not instantiated. To support efficient vertex-wise operation in maze routing (e.g., recording the prefix and cost, propagating to neighbors), the local graph instantiates all its vertices and edges. To be more specific, vertices are assigned with continuous indexes staring from zero, and adjacency lists are also created. In this way, any vertex/edge information can be efficiently stored and retrieved by direct-address tables (instead of hash tables or BSTs).

5.3 Routing Algorithm

In routing (especially detailed routing), sequential maze routing is widely adopted due to its scalability (compared with concurrent methods like [70, 71]) and flexibility (for capturing various objectives and violations). Recall from Figure 5.3 that our local grid graph is sparse because of the routing guide and long edges, which enhances the efficiency of our maze routing. We follow the convention of sequential maze routing. Essentially, nets are routed one after another, where previously routed nets are treated as blockages. After routing all nets with possible violations, several rounds of RRR help to clean them up.

5.3.1 Edge Cost in Local Grid Graph

The cost $w(e)$ of each edge e in the local grid graph $G(V, E, w)$ is a weighted sum of

- Basic wire cost (by length),
- Basic via cost (by count),
- Out-of-guide penalty, and
- Short/spacing violation penalty.

In this way, a path search (like Dijkstra's algorithm [26]) running on the grid graph will optimize these objectives automatically. The basic via/wire cost together with the short/spacing violation penalties are queried from the sparse global grid graph in batch. The out-of-guide penalty is charged according to the routing guide after the query.

Note that it is not determined by a single edge whether the minimum-area rule is violated or not. The minimum-area violation thus cannot be reflected as expensive edges like short/spacing violations and can only be captured by the path search algorithm.

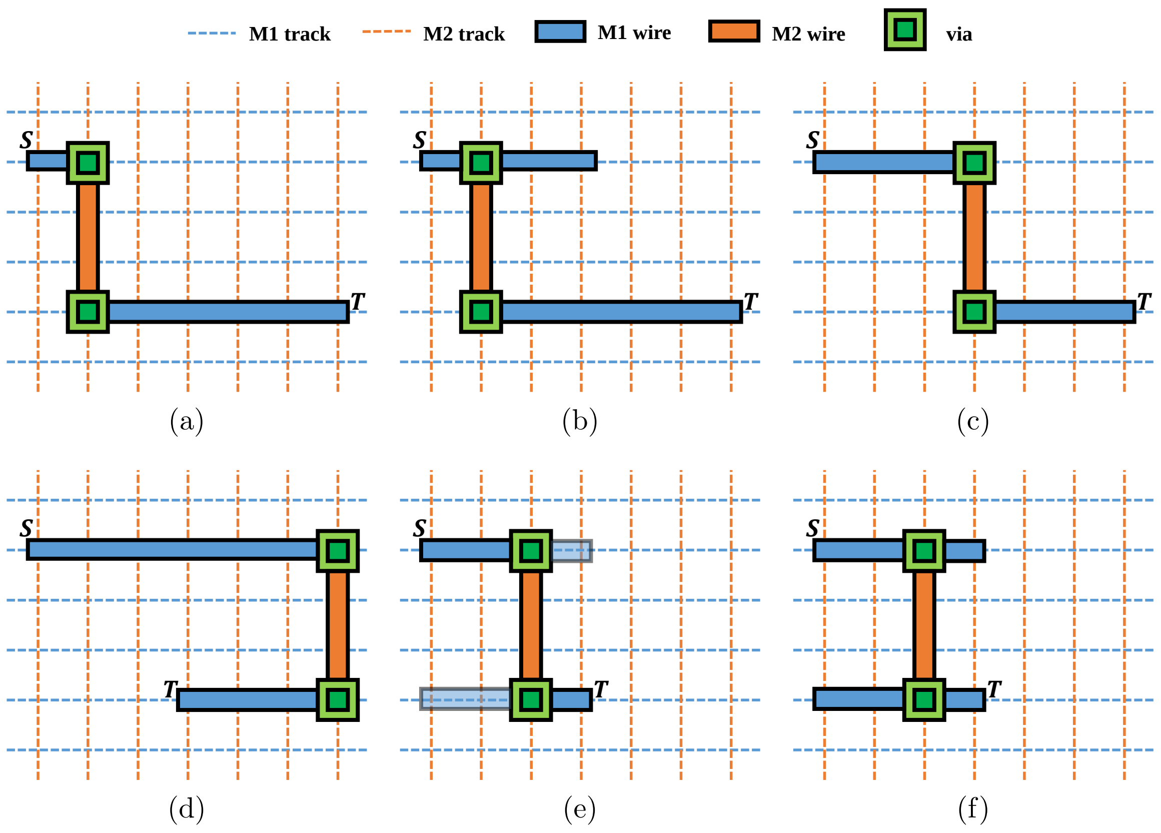


Figure 5.7: Capture minimum area cost in path search. Suppose the minimum area implies a length of three pitches. A path from source S to sink T is needed. (a) A normal path search without considering minimum-area violation. (b) Post fixing by extending wire. (c) Forcing the minimum length of wire segment in path search. (d) Detour due to the forcing. (e) & (f) Path search with wire extension considered.

5.3.2 Minimum-Area-Captured Path Search

For wires with a specific width, a minimum area implies a minimum-length constraint l_{min}. A straight-forward idea for fixing the violation after maze routing is to extend the wire segments that are not long enough. Such a greedy method may suffer from excessive wirelength (e.g., Figure 5.7(b) compared with Figure 5.7(c)) and even insufficient spare space for extension. Another method, multi-label path search [77], forces the minimum length for every wire segment without considering the possibility of extension. In this way, significant but unnecessary detour may be paid (Figure 5.7(d)). By capturing the minimum-area violation and its possible fixing during the path search, a better solution can be obtained (Figure 5.7(e)).

We extend the conventional Dijkstra's algorithm [26] to comprehensively handle the minimum-area rule. In Dijkstra's algorithm, the cost/distance of a path can be directly incremented. That is, the cost of a path from vertex v_1 via v_2 to v_3 is simply the sum of the cost of the two partial paths:

$$cost(v_1 \rightsquigarrow v_2 \rightsquigarrow v_3) = cost(v_1 \rightsquigarrow v_2) + cost(v_2 \rightsquigarrow v_3).$$

The challenge for considering the minimum area constraint is an uncertain cost of a partial path, which is unknown until the path turns or stops. At vertex v_2, it is unknown whether a minimum-area overhead (either wire extension or violation penalty) is needed, which depends on the future propagation of the path. However, for a path up to a certain wire segment, bounds on its cost can be calculated as follows.

- Lower bound cost: sum of edge costs plus the minimum-area overhead on all the previous wire segments.
- Upper bound cost: lower bound cost plus the potential minimum-area overhead on the current wire segment.

Our path search is detailed by Algorithm 5.1. The process is still based on a priority queue Q, but the operation domain is generalized from vertices to paths, because each vertex may have several candidate paths now. The information stored for a partial path P' includes:

- Prefix path $P'.prefix$ and current vertex $P'.vertex$. Note that such incremental storage requires $O(1)$ memory only for each propagated path, instead of $O(|P'|)$ with $|P'|$ being the number of vertices in P'.
- The lower bound $P'.costLB$ and upper bound $P'.costUB$ of the path cost.
- Length of the current wire segment $P'.length$. It is needed for calculating the minimum-area overhead.

The information stored at each vertex v is the smallest upper bound cost $v.costUB$ among all the paths reaching it.

In each iteration, the path P' with the smallest lower bound cost in the priority queue Q is popped out (line 6). It will be considered for propagating to the neighbors of $P'.vertex$. For an extended path P'' to a neighbor $v \in P'.vertex.neighbors$, satisfying $P''.costLB < v.costUB$ means that P'' is a potentially optimal path and should be considered for further propagation (line 25). If $P''.costLB \geq v.costUB$, P'' can be pruned. The algorithm stops when a sink vertex is reached (line 7). Note that for a sink vertex, the pin metal is sufficiently large and thus can guarantee that $P'.costLB$ is achievable (i.e., no minimum-area overhead charged).

The overhead due to the minimum-area rule depends on the length of the current wire segment $P''.length$, whether vertex v has sufficient spare space for wire extension

Algorithm 5.1 Optimal Minimum-Area-Captured Path Search

```
Require: A local grid graph $G(V, E, w)$, source and sink vertices $s$ and $t$, minimum length $l_{min}$
    of wire segment (implied by the minimum-area constraint).
Ensure: $s - t$ path $P$.
 1: $Q \leftarrow$ an empty priority queue for storing paths
 2: $v.costUB \leftarrow \infty, \forall v \in V$
 3: Initialize path $P'$ at $s$ ($P'.prefix \leftarrow null$, $P'.vertex \leftarrow s$, $P'.costLB \leftarrow 0$, $P'.costUB \leftarrow 0$,
    $P'.length \leftarrow l_{min}$)
 4: Push $P'$ into $Q$
 5: while $Q$ is not empty do
 6:     Pop the path $P'$ with smallest $P'.costLB$ from $Q$
 7:     if $P'.vertex = t$ then
 8:         return $P'$
 9:     end if
10:     for $v \in P'.vertex.neighbors$ do
11:         Relax($P', v$)
12:     end for
13: end while
14: function Relax($P', v$)                                  ▷ Extend path $P'$ to $v$
15:     $P''.prefix \leftarrow P'$
16:     $P''.vertex \leftarrow v$
17:     if $P'.vertex.layer \neq v.layer$ then
18:         $P''.costLB \leftarrow P'.costUB + w(P'.vertex, v)$
19:         $P''.length \leftarrow 0$
20:     else
21:         $P''.costLB \leftarrow P'.costLB + w(P'.vertex, v)$
22:         $P''.length \leftarrow P'.length + dist(P'.vertex, v)$
23:     end if
24:     $P''.costUB \leftarrow P''.costLB+$
    [2] MinAreaOverhead($P''.length, v.hasSpace$)
25:     if $P''.costLB < v.costUB$ then
26:         Push $P''$ into $Q$
27:         if $P''.costUB < v.costUB$ then
28:             $v.costUB \leftarrow P''.costUB$
29:         end if
30:     end if
31: end function
```

($v.hasSpace$), and the minimum length requirement l_{min} (line 24). To be more specific,

$$\textsc{MinAreaOverhead}(P''.length, v.hasSpace) =$$
$$\begin{cases} 0, & \text{if } P''.length \geq l_{min}, \\ w_{wire} \cdot (l_{min} - P''.length), & \text{if } P''.length < l_{min} \text{ and } v.hasSpace, \\ w_{minArea}, & \text{otherwise,} \end{cases}$$

where w_{wire} is the unit-length basic cost for wires, and $w_{minArea}$ is the penalty for each minimum-area violation. Note that the flag $v.hasSpace$ for all the vertices in the local grid graph can be queried from the global grid graph in batch. The flags are then stored explicitly in the direct-address table mentioned in Section 5.2.3.4.

Theorem 5.1 states the optimality of Algorithm 5.1. The proof is similar to that of the original Dijkstra's Algorithm (see [13]).

Theorem 5.1. *For a given local grid graph $G(V, E, w)$, Algorithm 5.1 gives an optimal $s - t$ path P satisfying the minimum length constraint l_{min}.*

The path search algorithm in MANA [76] also captures the minimum length constraint in a similar manner. The strengths of our approach over MANA are two folds. First, our framework allows minimum-area violations to exist in earlier RRR iterations. The minimum-area penalty serves as Lagrange multiplier [61] and helps to build a smooth RRR optimization process. It avoids satisfying minimum-area constraint at a huge price of sacrificing other metrics (e.g., wirelength) in early iterations but still leads to almost zero minimum-area violation eventually. Second, we query the flag *v.hasSpace* in batch from our global grid graph, which is more efficient.

For a multiple-pin net, path search starts from a source pin s. When reaching the first other pin, all vertices on the path are regarded as source for searching a next pin, until all pins are reached [57].

5.3.3 Rip-up and Reroute

Our rip-up and reroute (RRR) strategy is similar to those widely used in global routing (e.g., NCTU-GR [69]) with two major differences. First, only nets with violations are ripped up to save runtime, considering that detailed routing is more time-consuming. Second, for ripped-up nets, their routing regions will be slightly expanded for attempting a larger solution space in the next iteration.

For the wires and vias with design rule violations in the current RRR iteration, a history cost will be recorded. Note that for a wire segment with violations, history cost is charged only for the intervals with violations on it. In this way, the actual situation of resource competition can be reflected. Such a negotiation-based RRR results in a better and faster convergence, as Section 5.5 will show.

5.4 Parallelism

Detailed routing is time-consuming in general. There are many jobs during the whole process that can be easily parallelized. For example, the initialization of conflicted wires and vias in the global grid graph can be conducted in parallel for different layers and different regions of a chip. However, the major runtime bottleneck of Dr. CU is to construct the local grid graph, run maze routing, and update the global grid graph for each net.

The turn-around time of detailed routing can be further shortened by routing different nets in parallel. The challenge here is that the routing regions of different nets may

Algorithm 5.2 Scheduling for Parallel Routing

```
Require: Nets
Ensure: $batchList$
 1: Sort all nets in decreasing size of routing region
 2: $batchList \leftarrow \emptyset$
 3: for each net $n_i$ do
 4:     for each batch $b_j$ in $batchList$ do
 5:         if $n_i$ has no conflict with $b_j$ then                    ▷ By R-trees
 6:             Add $n$ into $b_j$
 7:             Break
 8:         end if
 9:     end for
10:     if $n_i$ has not been assigned to any batch then
11:         Append a single-net batch with $n_i$ to $batchList$
12:     end if
13: end for
14: Reverse the order of batches in $batchList$
15: for each batch $b_j$ in $batchList$ do
16:     Sort nets in $b_j$ by decreasing size of routing region
17: end for
```

overlap. We design an efficient bulk synchronous parallel scheme [132]. It routes batches of independent nets one after another. Note that such independence, together with a deterministic scheduling of batches, can ensure deterministic routing results.

For nets in the same batch, their routing regions do not overlap. Here a *safety margin* is also considered, which captures spacing rules and possible wire extension for minimum-area compliance. There are two phases for each batch. The *routing phase* queries nets from the global grid graph, constructs the local grid graphs, and runs maze routing; the *committing phase* records routed edges into the global grid graph (see Figure 5.3), which can be regarded as a data synchronization needed by later batches. The parallelism for the independent jobs in either the routing or committing phase is trivial: each thread keeps consuming a net from a pool of unprocessed nets until the pool becomes empty. With runtime dominated by the routing phase, the reason for having a separate committing phase is to avoid a heavy usage of mutual exclusion (mutex) [133] among threads. Routed edges in the global grid graph are stored by BSTs. A BST cannot be accessed when it is being modified by another thread, even if the ranges of access and modification do not overlap. One solution is to set up locks. Its drawback is that reading BSTs is significantly more frequent than writing. Note that for a net, reading BSTs is performed on its routing region, while writing is only performed for the solution paths, which comprises just a small part of the whole routing region. By separating the committing phase, the BST read in the routing phase becomes lock-free and thus can be performed faster.

A scheduling of all the batches will be performed in the beginning of an RRR iteration by Algorithm 5.2. Nets are assigned one after another by trying to join an existing batch (lines 4–9) and thus minimizing the number of batches. R-trees are used to detect the conflict between a net and a candidate batch. For a batch of nets, there are several R-trees storing their rectangular routing regions, one for each layer. In this way, the scheduling is very efficient and empirically only takes 1.02–2.07% of the total running time. Figure 5.8(a) shows the runtime profile of all the batches on a test case. Note that in a batch, different threads may finish their last jobs at different time and thus have various durations. The maximum duration of all the threads is the time that a batch needs, while the average duration is the runtime lower bound that can be achieved by an ideal scheduling. Their small difference shown in Figure 5.8 justifies the good quality of our scheduling.

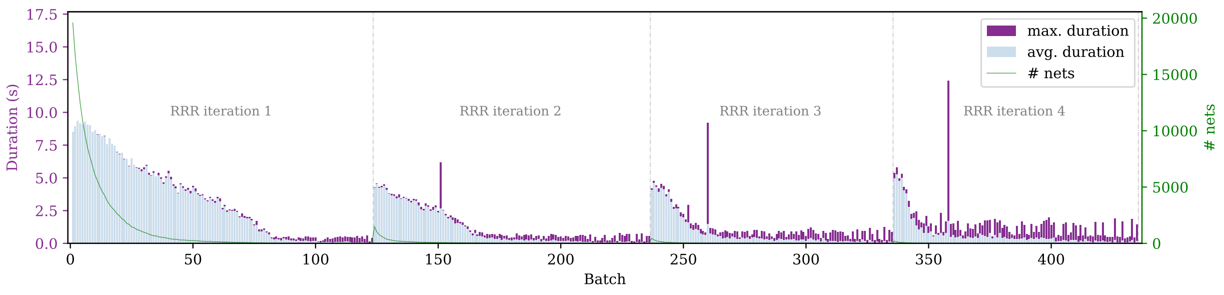
Moreover, we apply three techniques to further improve the effectiveness of scheduling. The first two techniques are to enhance the load balancing.

- *Within-batch balancing* (WBB, Algorithm 5.2 lines 15–17). The workload of different threads in a batch can be more balanced by processing larger nets first. The improvement is evidenced by the smaller gaps between the maximum and the average durations of each batch in Figure 5.8(b).

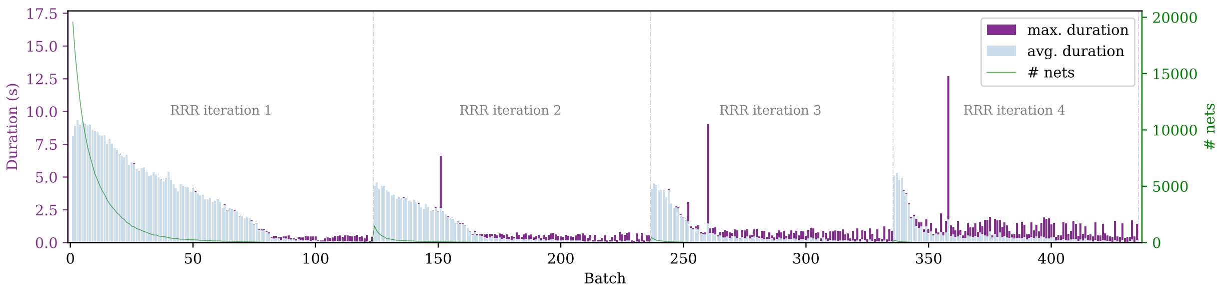
- *Inter-batch balancing* (IBB, line 1). Attempting larger nets first during the scheduling can improve the parallelism, as Figure 5.8(c) shows. The benefits are in three folds. First, larger nets are more likely to have overlap with the existing nets in a candidate batch. Therefore, IBB can help to reduce the number of batches by increasing the success rate of larger nets (e.g., reduced from 123 to 96 for the first RRR iteration on `ispd18_test9`). Second, our scheduling algorithm tends to make later batches with fewer nets and thus worse load balancing among threads. IBB remedies the problem by making later batches have fewer nets and by making nets in later batches smaller. Third, some nets may be very huge and need a long time to be routed. If the other nets in its batch do not take a sufficiently long time in total, there will be a single thread routing the huge net with other threads idle (seen by the long "pulse" in Figure 5.8(b)). IBB can gives more load to the batch of huge nets (usually the first several batches) and avoid such an issue.

The third technique is *batch with small nets first* (BSF, line 14). IBB also lets large nets be routed earlier. The problem is that small nets are less flexible in maze routing than large nets due to their smaller solution space. Routing large nets first makes later small nets even more difficult to be routed. BSF reverses the order of all batches and avoids the problem. In terms of runtime, it leads to fewer nets with violations in a RRR iteration, reroutes fewer nets in the next RRR iteration, and thus saves the runtime, which can be seen from Figure 5.8(d).

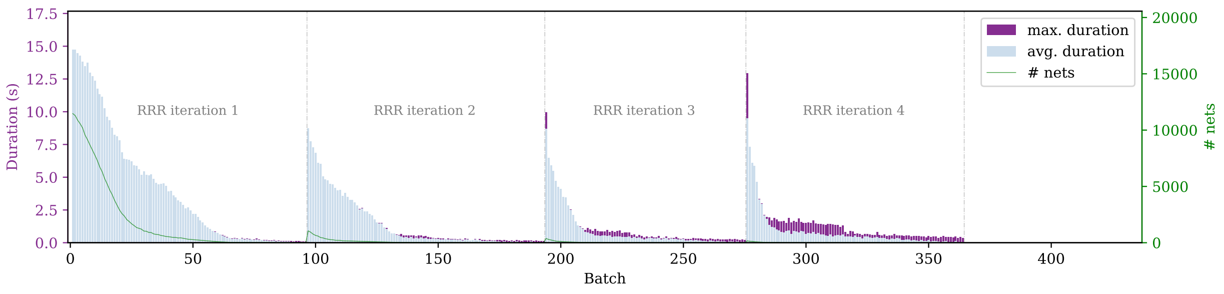
The eventual runtime benefits of the three techniques will be shown in Section 5.5.



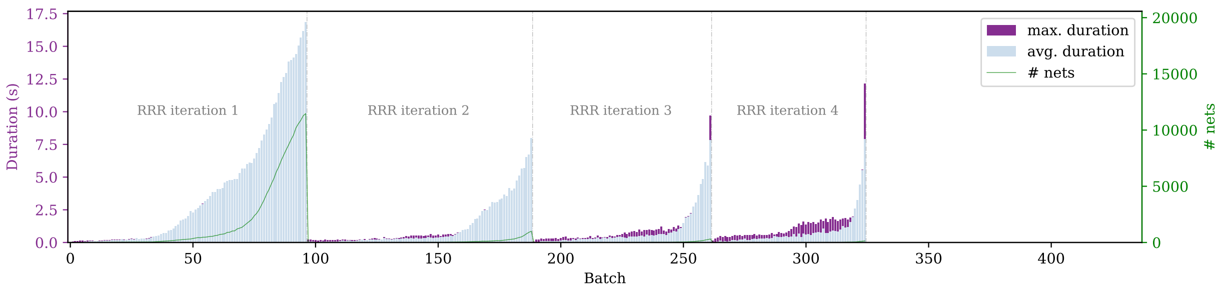
(a) None (routing phase 937s)



(b) WBB (routing phase 906s)



(c) WBB + IBB (routing phase 827s)



(d) WBB + IBB + BSF (routing phase 749s)

Figure 5.8: Better parallelism by within-batch balancing (WBB), inter-batch balancing (IBB), and batch with small nets first (BSF). The result is on `ispd18_test9` and across four RRR iterations.

Table 5.2: Metric Weights in ISPD 2018 Contest Benchmarks

Metric		**Weight**
Basic cost	wirelength	0.5
	# vias	2
Non-preferred usage	out-of-guide wirelength	1
	# out-of-guide vias	1
	off-track wirelength	0.5
	# off-track vias	1
	wrong-way wirelength	1
Design rule violations	short metal area	500
	# spacing violations	500
	# min-area violations	500

Table 5.3: ISPD 2018 Contest Benchmark Characteristics

Benchmark	# std. cells	# block macros	# nets	# pins	# IO pins	# layers	M2 # tracks	M2 pitch (μm)	Die size (mm^2)
test1	8879	0	3153	17203	0	9	977	0.2	0.20×0.19
test2	35913	0	36834	159201	1211	9	3254	0.2	0.65×0.57
test3	35973	4	36700	159703	1211	9	4943	0.2	0.99×0.70
test4	72094	0	72401	318245	1211	9	8886	0.1	0.89×0.61
test5	71954	0	72394	318195	1211	9	9800	0.1	0.93×0.92
test6	107919	0	107701	475541	1211	9	5312	0.1	0.86×0.53
test7	179865	16	179863	793289	1211	9	13500	0.1	1.36×1.33
test8	191987	16	179863	793289	1211	9	13500	0.1	1.36×1.33
test9	192911	0	178857	791761	1211	9	13500	0.1	0.91×0.78
test10	290386	0	182000	811761	1211	9	13500	0.1	0.91×0.87

5.5 Experimental Results

Dr. CU is implemented in C++ with the boost geometry library [134] for R-tree query and Rsyn [135] as parser. Experiments are performed on a 64-bit Linux workstation with Intel Xeon Silver 4114 CPU (2.20GHz, 40 cores) and 256GB memory. Benchmarks are from the ISPD 2018 Initial Detailed Routing Contest [75]. The metric weights for the total quality score and the benchmark characteristics are shown by Table 5.2 and Table 5.3 respectively. Consistent with the contest, eight threads are used by default. The result reporting is conducted by Cadence Innovus 17.1 [136] and the official evaluation script [137].

The result statistics of Dr. CU is illustrated by Table 5.4. Figure 5.9 shows a GUI view of the solution on `ispd18_test10`.

5.5.1 Effectiveness of Quality Enhancement

Figure 5.10 shows the score breakdown of Dr. CU on `ispd18_test9` and `ispd18_test10` across the four RRR iterations. The score is calculated under the metric of ISPD 2018 Contest and divided into three categories – basic cost, non-preferred usage, and design

Table 5.4: Comparison with State-of-the-Art Academic Detailed Routers on ISPD 2018 Contest Benchmarks

		Basic cost		Non-preferred usage					Design rule violations					**ISPD'18**		
		WL[a]	# vias	Out-of-guide		Off-track		Wrong-way	# short	Short	# min	#	**Total**	**quality**	Mem	**Time**
				WL[a]	# vias	WL[a]	# vias	WL[a]		area[a]	area	spacing	**#**	**score**	(GB)	**(s)**
Dr. CU	test1	433254	32031	1706	446	393	0	4749	4	0.4	0	17	**21**	**296504**	0.33	**11**
	test2	7806294	317160	34194	5948	4937	0	44495	12	1.3	0	73	**85**	**4661740**	1.70	**85**
	test3	8683731	307545	52408	5499	5714	0	45541	346	372.5	0	161	**507**	**5330014**	1.75	**113**
	test4	26033480	658644	132938	16103	9190	0	59579	463	436.8	6	1071	**1540**	**15304156**	3.94	**320**
	test5	27729394	916715	92872	16686	1588	0	44680	406	77.4	10	496	**912**	**16144832**	5.42	**426**
	test6	35595790	1403634	142595	25939	8735	0	69829	168	92.7	21	587	**776**	**21198243**	6.48	**527**
	test7	64994186	2271738	235497	36269	16459	0	106884	772	230.8	38	325	**1135**	**37724327**	10.77	**969**
	test8	65289434	2281513	290418	38596	17082	0	111173	861	249.5	20	399	**1280**	**37990696**	11.73	**1034**
	test9	54602832	2282226	284645	42078	12746	0	108324	297	162.7	28	379	**704**	**32592136**	11.20	**906**
	test10	67907614	2439531	1137257	64535	30527	0	197840	14605	11370.4	44	3910	**18559**	**47909940**	11.95	**1299**
	Avg. ratio	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	**1.00**	**1.00**	1.00	**1.00**
[9]	test1	434914	34443	4352	859	276	0	2363	127	15.3	0	122	249	362725	0.33	22
	test2	7817285	339055	104720	11784	4353	0	22023	1005	1329.9	0	1949	2954	6366885	1.48	114
	test3	8707641	331958	176736	10731	4344	0	22187	2444	1982.1	0	2419	4863	7430091	1.57	128
	test4	26042785	701994	769265	31444	41791	0	89537	6914	26328.8	0	11224	18138	34112927	3.50	443
	test5	27852167	942588	649224	43071	13390	0	63397	5466	4722.2	0	7742	13208	22805759	4.48	692
	test6	35813473	1446807	976672	68656	20357	0	95811	7959	12891.0	0	11023	18982	33908650	6.35	1054
	test7	65360688	2349580	2187794	101866	33105	0	170316	23141	33040.9	0	14880	38021	63816461	10.54	1848
	test8	65668468	2360231	2288159	102982	33373	0	170583	20641	22352.8	0	14384	35025	58501486	10.62	1867
	test9	54993356	2358857	1604576	115465	29620	0	168722	18830	17315.6	0	14470	33300	50010786	10.43	1804
	test10	68282001	2532666	2826908	140343	32865	0	180586	26688	150704.9	0	20837	47525	128141528	11.10	1909
	Avg. ratio	1.00	1.05	5.39	2.34	2.50	-	1.14	31.75	165.37	0.00	21.91	21.76	1.67	0.92	1.66
[86]	test1	464503	39199	5659	1301	64	114	17	4364	1.1	0	120	4484	378304	4.43	101
	test2	8097032	385111	63976	12746	2474	1241	172	29845	36.3	1	1419	31265	5626235	29.19	897
	test3	9013950	389718	42336	691	19905	1117	205	34753	1507.2	0	1755	36508	6971806	38.45	1395
	test4	27165618	847643	267804	50040	180535	1675	1016	42024	17088.6	54	3130	45208	25825193	52.44	6164
	test5	29206112	1142635	278618	54613	20047	9905	1174	145826	1795.2	118	7438	153382	21918275	34.75	2317
	test6	37905264	1768984	408860	80328	31165	16368	2765	152194	1937.0	188	11630	164012	29892011	35.89	3807
	test7	68655629	2866477	677287	131394	93328	23387	4378	243375	9187.7	270	12896	256541	52120721	44.53	6561
	test8	68988139	2879647	696911	135073	88394	23598	4506	238519	8386.2	240	12744	251503	51842741	44.98	6136
	test9	58255989	2872574	620805	127167	51854	23438	4449	264230	2531.7	260	12581	277071	43361290	45.04	5737
	test10	71637851	3055779	957064	138008	232529	27502	5610	340647	12162.8	259	16164	357070	57467840	47.21	12614
	Avg. ratio	1.05	1.25	2.22	2.69	6.25	-	0.02	653.96	20.71	9.15	18.41	189.76	1.35	9.39	9.25
[87][b]	test1	487842	42565	1107	502	167	0	1724	95	4.4	0	773	868	721170	0.29	67
	test2	8322145	403543	34068	5689	2783	0	17897	1218	172.0	0	6803	8021	8514694	1.93	1680
	test3	9212616	398144	16211	5987	1920	0	16672	3919	1365.2	0	8477	12396	10363523	2.09	2194
	test4	27699631	822662	56624	28601	4369	0	56093	5969	8273.2	126	45661	51756	42668733	4.78	7201
	test5	29493060	1072812	130131	13385	15748	0	107889	7037	7681.5	37	96575	103649	69298177	5.46	10017
	test6	38123660	1641879	190819	21811	30645	0	183006	10375	12213.7	48	113048	123471	85411399	7.75	16345
	test7	69033346	2656802	453807	39365	52255	0	302204	19802	20961.3	108	179190	199100	140781447	13.11	51768
	test8	69039670	2621050	468925	39302	53192	0	307267	20944	22601.7	103	182494	203541	143203359	13.53	51328
	test9	58299612	2621857	336589	39504	46044	0	301241	19246	17949.0	74	185270	204590	136740361	13.39	45024
	test10	71636304	2791064	523896	47663	60925	0	358457	35218	221528.4	55	218478	253751	262391463	14.21	71552
	Avg. ratio	1.07	1.21	1.03	1.04	2.71	-	1.73	34.57	70.45	5.55	217.47	110.52	3.45	1.14	33.00
1st place of ISPD 2018	test1	472032	41641	6246	1385	3528	116	3509	4223	0.7	0	107	4330	386190	5.66	100
	test2	8150588	409551	71685	13451	20402	1362	18214	36601	94.9	1	1158	37760	5636272	29.91	831
	test3	9086139	427410	69182	2450	33470	1216	18882	46966	4891.4	0	1387	48353	8645535	41.44	1408
	test4	27514053	858224	240226	8841	150961	1011	224715	349597	52947.1	6	50957	400560	67978775	43.93	4374
	test5	29415618	1158945	342675	31391	46870	10514	194054	431909	28428.7	28	66742	498679	65227110	23.02	1794
	test6	38191983	1800286	471017	42714	151178	17549	281027	628776	31227.5	15	100196	728987	89303688	28.36	2969
	test7	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail
	test8	69559382	2929578	1006247	82478	375236	22294	455824	1058138	76790.0	48	161229	1219415	161426825	40.78	5030
	test9	58803453	2920259	813750	67367	331766	22915	446432	1051112	56580.8	40	158305	1209457	144221468	40.16	4481
	test10	72244024	3110163	1414338	81831	625291	27392	476670	1289359	120966.0	33	177426	1466818	193867712	43.42	5271
	Avg. ratio	1.06	1.30	2.61	1.66	16.74	-	2.70	1628.84	175.34	1.52	138.97	582.42	3.28	9.90	7.60

[a] Unit of length is M2 pitch; unit of area is the square of M2 pitch.

[b] Two versions, with and without spacing-to-short conversion, are reported in [87]. The version without spacing-to-short conversion is shown here because it is more practically meaningful.

rule violations. During the RRR process, even though the non-preferred usage (especially out-of-guide wirelength) may slightly increase, the design rule violations can be significantly reduced. This demonstrates the effectiveness of our RRR scheme.

Figure 5.11 shows the enhancement due to three other techniques. First, adding some wrong-way edges in the local grid graph (Section 5.2.3.3) helps to enlarge the solution space and thus alleviate the congestion problem, which brings 7.7% score improvement on average. Second, the minimum-area-captured path search (Section 5.3.2) provides more correct-by-construction design rule satisfaction. To be more specific, it reduces the number of minimum area violations by 82.6% and the total score by 0.93% on average. Third, using history cost in RRR (Section 5.3.3) improves the quality score by 1.02% eventually. Meanwhile, it also results in a faster convergence, reducing the total runtime

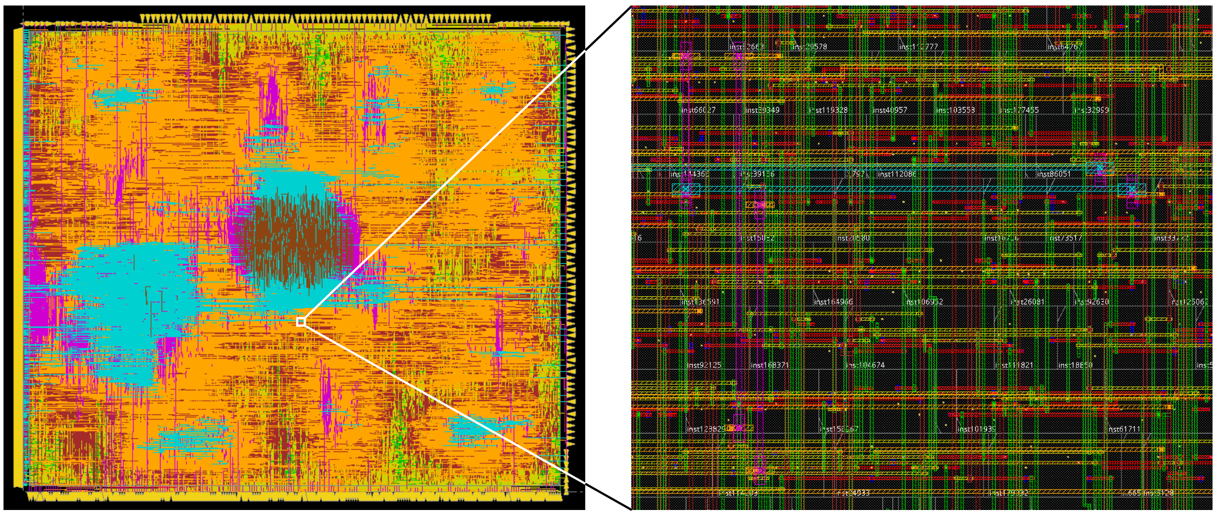
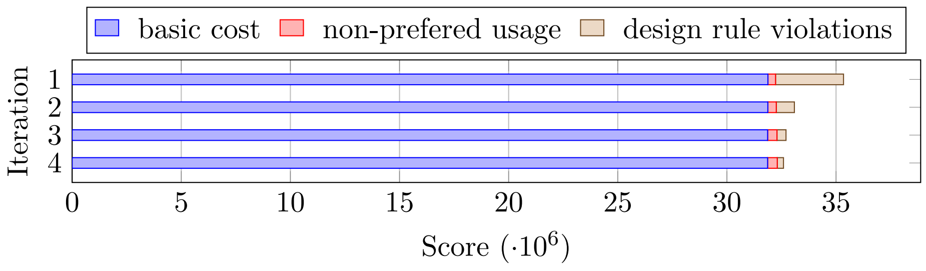
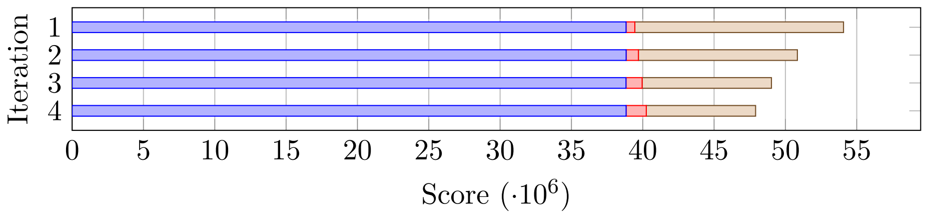


Figure 5.9: Solution of Dr. CU on ispd18_test10.



(a) on ispd18_test9



(b) on ispd18_test10

Figure 5.10: Improving routing quality by RRR.

by 6.8% on average.

5.5.2 Effectiveness of Runtime Reduction

Figure 5.12 shows the speed-up due to pre-computing via-obstacle and via-pin conflicts. Here, the turn-around time of the whole detailed routing process is saved by 32%–63%.

The acceleration achieved by our parallelism is shown in Figure 5.13. Eight threads give around five to six times speed-up compared with single-thread routing. Here, within-batch balancing (WBB), inter-batch balancing (IBB), and batch with small nets first (BSF) contribute 3.63%, 9.20%, and 6.65% improvement on average respectively. In total, "WBB+IBB+BSF" reduces runtime by 18.5% on average.

Fig. 5.14 shows the runtime breakdown of Dr. CU on ispd18_test10. Before

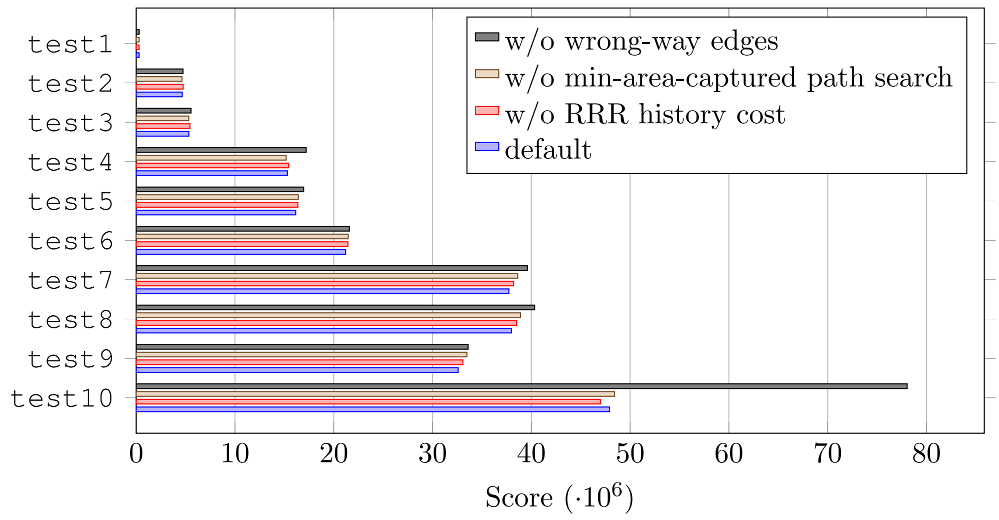


Figure 5.11: Improving routing quality by using wrong-way edges, minimum-area-captured path search, and history cost.

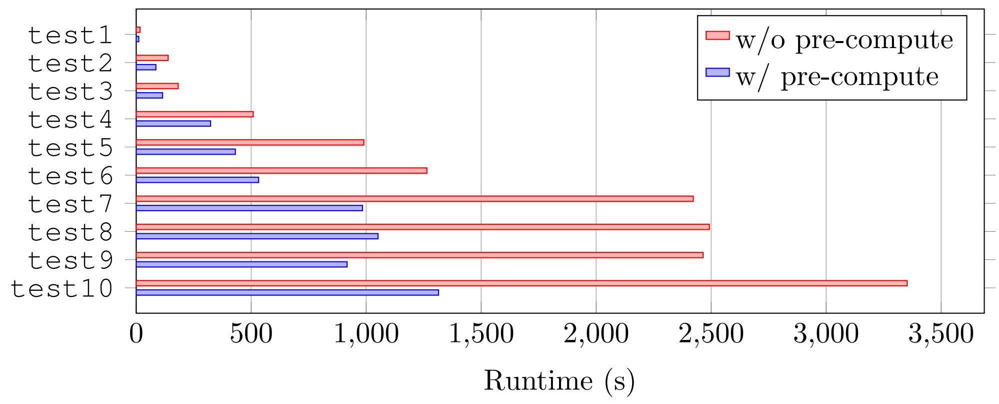


Figure 5.12: Speed-up by pre-computing via-obstacle and via-pin conflicts.

routing, the global grid graph and conflict LUTs are initialized, which takes 4.9% of the total runtime. We define the process of caching (including querying the global grid graph and constructing local grid graphs) and maze routing as *core routing*, which is the major consumer of runtime (38.4%+37.3%+3.7% = 79.4%). In each RRR iteration, core routing is performed under our bulk synchronous parallel scheme, where there is a parallel loss[3]. The miscellaneous jobs for routing including the committing phase mentioned in Section 5.4, ripping up violated nets, updating history cost, etc. They take 12.9%. After routing, we write the routing solution to the output file.

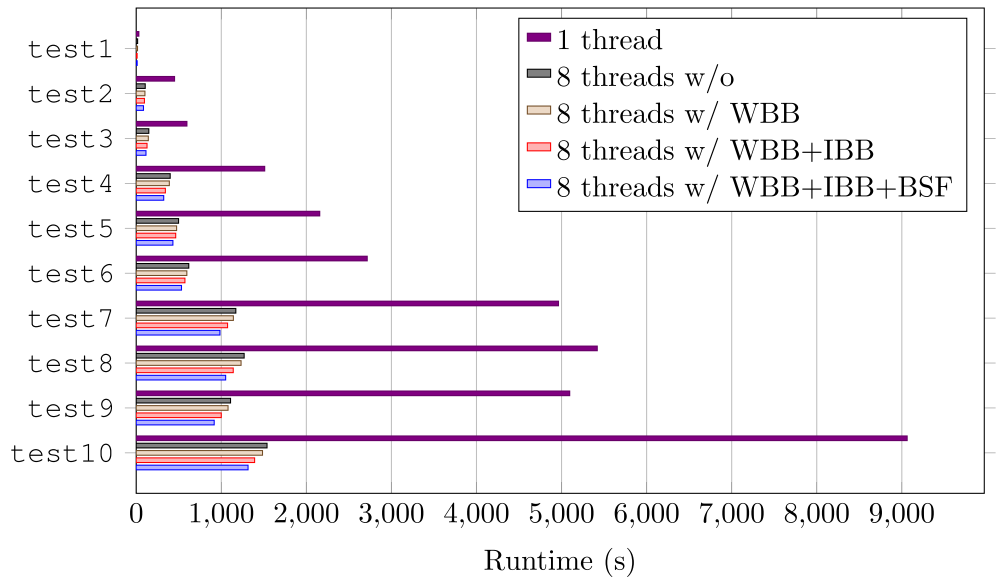


Figure 5.13: Speed-up by parallelism.

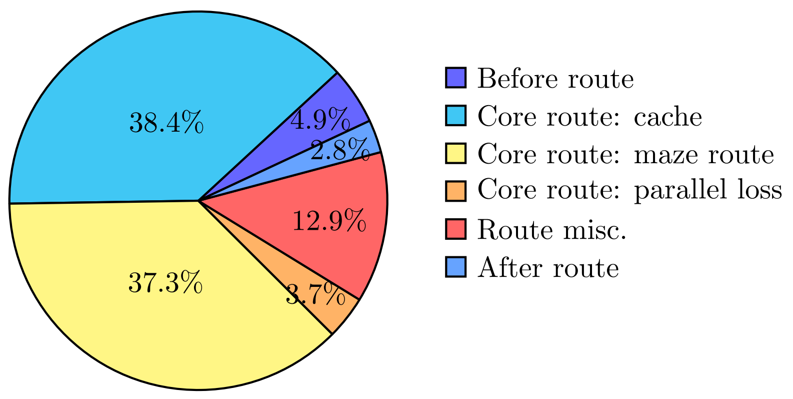


Figure 5.14: Runtime breakdown on `ispd18_test10`.

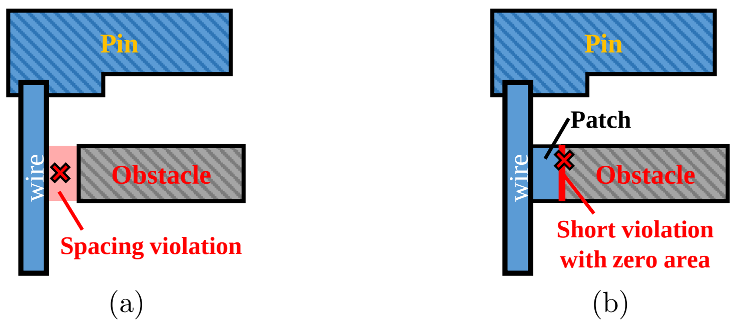


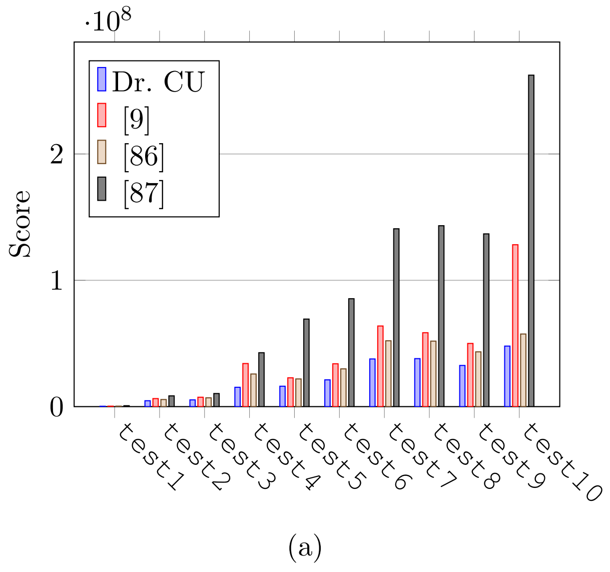
Figure 5.15: Spacing-to-short conversion done by some other detailed routers. (a) A spacing violation between a wire segment and an obstacle. (b) A metal patch that converts the spacing violation to a short violation with zero area.

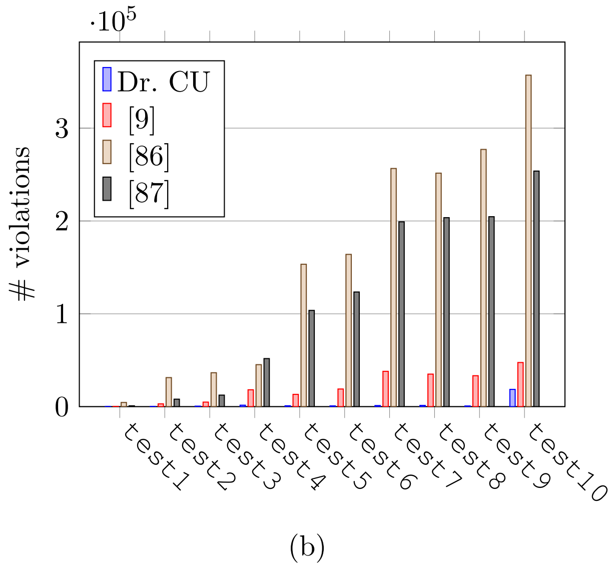
5.5.3 Comparison with State-of-the-Art Detailed Routers

We also compare Dr. CU with TritonRoute [86], the work [87], and the first place in ISPD 2018 Contest (Table 5.4). For all the detailed routers, We run the binaries provided by the authors on our machine with eight threads. Besides the ISPD 2018 Contest metric, we also report the number of short violations. This is to avoid the misleading due to the abusing of the contest metric. In the design rule verification of Innovus, a spacing violation (e.g., Figure 5.15(a)) can be removed by inserting a metal patch between the two violating objects (e.g., Figure 5.15(b)). The patch generates a short violation with zero area, which improves the score under the contest metric but is not beneficial to the real design need.

Regarding the routing quality, Dr. CU shows significantly better scores in many aspects (including wirelength, via count, out-of-guide usage, off-track usage, and design rule violations) in most cases. According to the metric of ISPD 2018 Contest, our routing quality wins all the other state-of-the-art detailed routers in all test cases, as Figure 5.16(a) summarizes. Regarding the number of design rule violations, our strength is even more obvious (better by one or two orders of magnitude), as Figure 5.16(b) shows. At the same time, the runtime of Dr. CU is also tremendously better than the others (Figure 5.16(c)).

[3]We divide the total CPU time for caching by the number of threads to get *the equivalent wall time for caching.* Similarly, there is the *equivalent wall time for maze routing.* The parallel loss of core routing is therefore the real wall time of while core routing process minus the equivalent wall time for caching and maze routing.





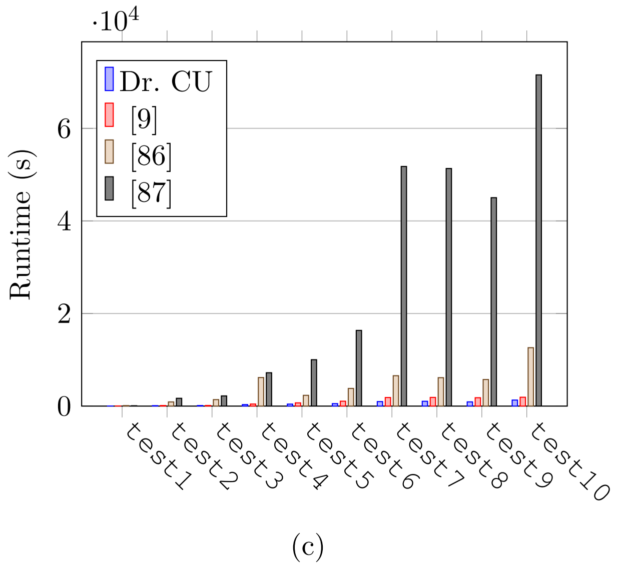


Figure 5.16: Comparison with state-of-the-art detailed routers on (a) quality score under the metric of ISPD 2018 Contest, (b) total number of design rule violations, and (c) runtime.

Chapter 6

Bus Routing

The major challenge of bus routing is to maintain topology consistency. If processing bit by bit (e.g. route bit 1, 2 and 3 sequentially as in Figure 6.1 (b)), the latter bits may lack available track segments to be routed on especially when the routing track configuration is non-uniform and complex. In the worst case, much effort of trial and error is needed until finding a feasible topology.

In this chapter, we present an effective bus routing method named MARCH which can efficiently solve this important problem handling practical issues like minimum spacing and minimum wire width on metal layers with irregular track structures. The objective is to finish routing all the buses, maintaining the same topology for different bits of a bus while optimizing metrics like wirelength, wire segment number and compactness of the buses. Our main contributions can be summarized as follows.

- We propose MARCH, which routes all the bits in a bus concurrently, instead of processing bit after bit. Such concurrency directly captures topology consistency constraint together with other objectives (e.g. wirelength) and constraints (e.g. spacing) in a correct-by-construction manner.
- A hierarchical framework is designed for the efficiency of MARCH, consisting of a Topology-Aware Path planning (TAP) and a Track Assignment for Bits (TAB). TAP is efficient as it works on a coarse-grained solution space (see Figure 6.1 (c)). TAB generates fine-grained routing solution, but it also gains efficiency by searching on the regions provided by TAB only (see Figure 6.1 (d)).
- We present an effective rip-up and reroute scheme to further improve the routing solution quality.

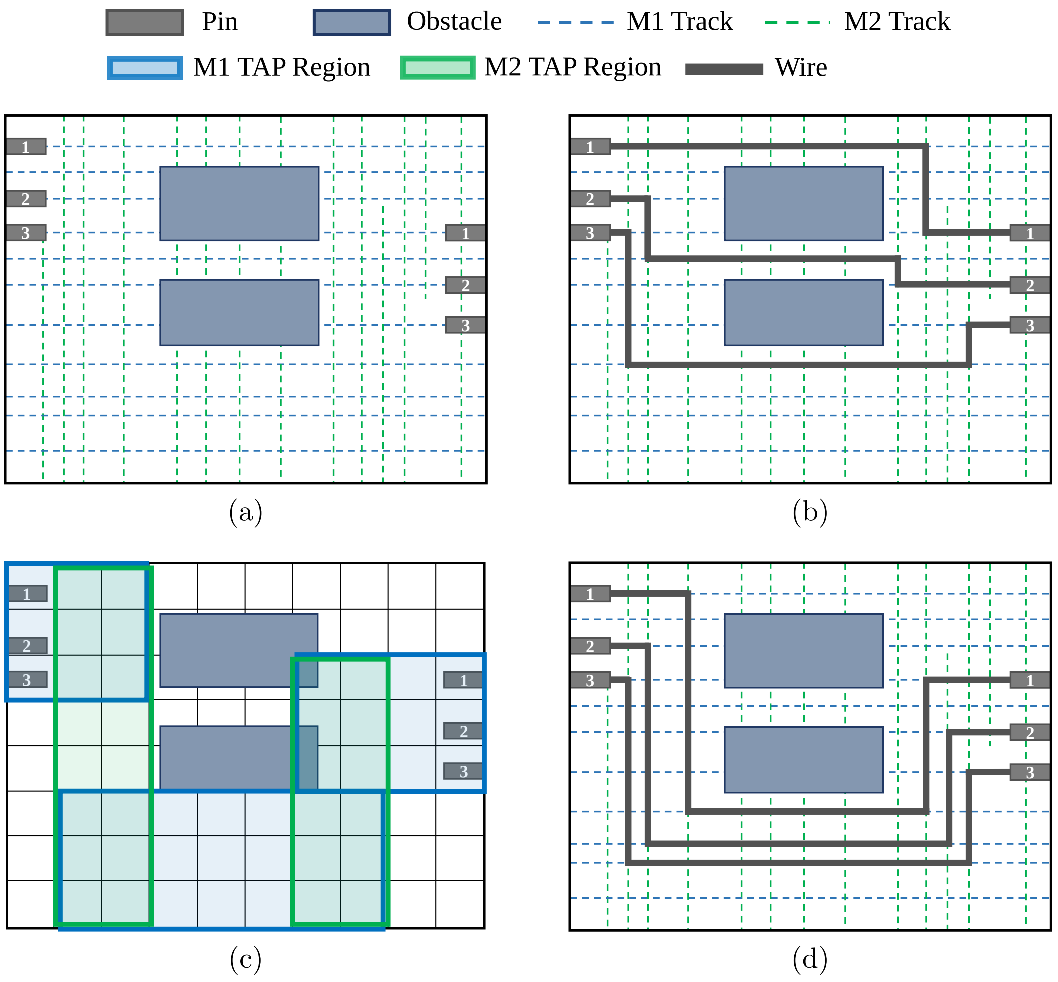


Figure 6.1: (a) A toy bus with two pins and three bits to be routed. (b) The bits routed one by one. (c) The Topology-Aware Path planning (TAP) result. (d) The Track Assignment for Bits (TAB) result.

6.1 Preliminaries

In the bus routing problem, buses may have multiple pins and have different wire width constraints on different metal layers. In each layer, there are routing obstacles and non-uniform routing tracks with different lengths and wire width constraints. An example is shown in Figure 6.2.

6.1.1 Evaluation Metrics

The total cost C_{total} of a bus routing solution is the summation of the failure penalty cost C_{fail}, the spacing penalty cost C_{space}, and the routing cost C_{route} of all the buses.

$$C_{total} = C_{fail} + C_{space} + C_{route} \tag{6.1}$$

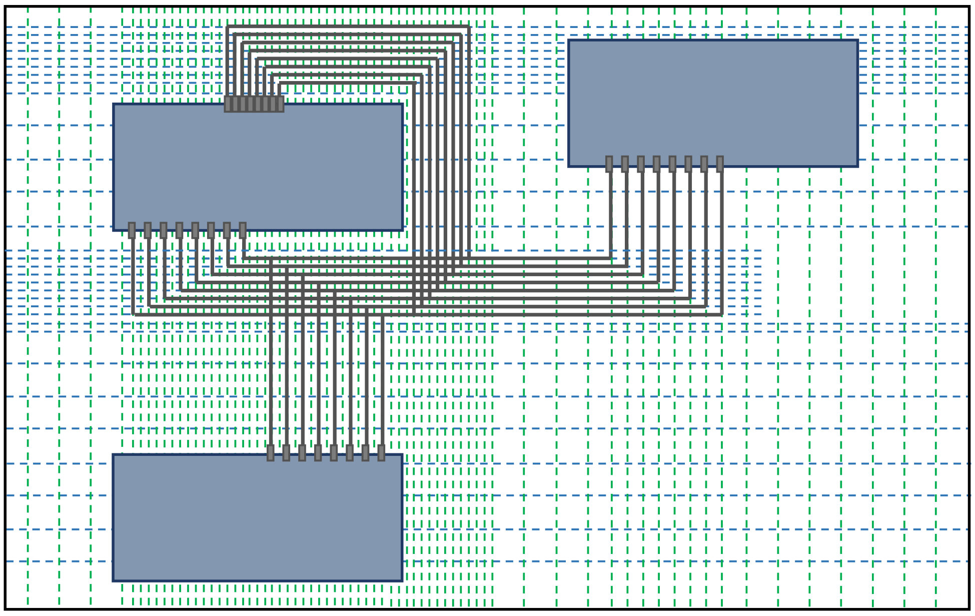


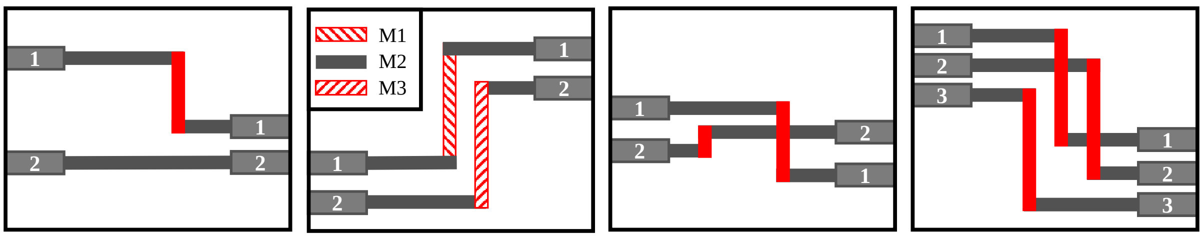
Figure 6.2: A bus with four pins and eight bits

Failure Penalty Cost A successfully routed bus has to satisfy the following requirements. The routing tree of a bit needs to connect all the pins of the bit. All wires should be on-track and do not violate the width constraint of the track. More importantly, all bits are routed with the same topology satisfying the following requirements.

1. All bits should have the same number of wire segments. In Figure 6.3 (a), bit 1 has three wire segments, while bit 2 has only one.
2. Wires of different bits should go through the same sequence of layers. In Figure 6.3 (b), bit 1 goes through M2, M1, and M2, while bit 2 goes through M2, M3, and M2.
3. Wires of different bits should be routed towards the same directions. In Figure 6.3 (c), bit 1 is routed right, down, and right, while bit 2 is routed right, up, and right.
4. Within each segment of the topology, the bit order should either be the same as or in reversed order of the bits at the pin locations. Note that on the layer with horizontal tracks, the bit order is the order of the bits from bottom to top. It is similar for the layer with vertical tracks. In Figure 6.3 (d), the bit order of the middle wire segments is neither the same as nor in reversed order of the bits at the pin locations.

Let N_{fail} denote the number of buses failed to be routed, then $C_{fail} = w_{fail} \cdot N_{fail}$.

Spacing Penalty Cost Any pair of objects (e.g. wires of different bits, obstacles, chip boundary, etc.) on the same layer should not violate their corresponding spacing constraints. Let N_{space} denote the number of spacing violations, then $C_{space} = w_{space} \cdot N_{space}$.



(a) (b) (c) (d)

Figure 6.3: Four types of topology failures.

Routing Cost For successfully routed buses, the routing cost C_{route} can be computed based on three normalized costs: wirelength cost C^b_{wire}, segment cost C^b_{seg}, and compactness cost C^b_{com}:

$$C_{route} = \sum_{bus\ b} w_{wire} \cdot C^b_{wire} + w_{seg} \cdot C^b_{seg} + w_{com} \cdot C^b_{com} \tag{6.2}$$

where C^b_{wire} is the average (among all the bits) of the total wirelength of a bit divided by the half parameter wirelength of the bit, and C^b_{seg} is the segment number of the topology divided by a lower bound [138]. C^b_{com} is the average (among all the wire segments) of the segment width divided by a lower bound [138], where segment width is the distance between the two outermost bits of the segment. A good routing solution should have short wirelength, less segments in the topology, and more compacted width in each segment.

6.1.2 Problem Formulation

Problem 6.1 (**Bus Routing**). *Given the pin information and width constraints of the buses, the tracks and their width constraints on each layer, and the obstacles, connect all the pins of each bit for all the buses and minimize the cost C_{total}.*

6.2 Algorithms

The framework of MARCH consists of two levels of loops. The inner loop routes all the buses (see the green box in Figure 6.4), and the outer loop is a Rip-up and Reroute (RR) scheme that tries to find a better solution. During initialization, with the loaded information of buses, tracks, and obstacles, a Bus-based Grid Graph (BGG) data structure, which will be used during the whole procedure, will first be constructed.

In the inner loop, each bus will go through fours steps: Bus-based Grid Graph (BGG) update, Topology-Aware Path planning (TAP), Track Assignment for Bits (TAB), and track occupancy update. First, BGG will be updated according to the bus to be routed so that it can provide accurate information of the routing resources meeting the width

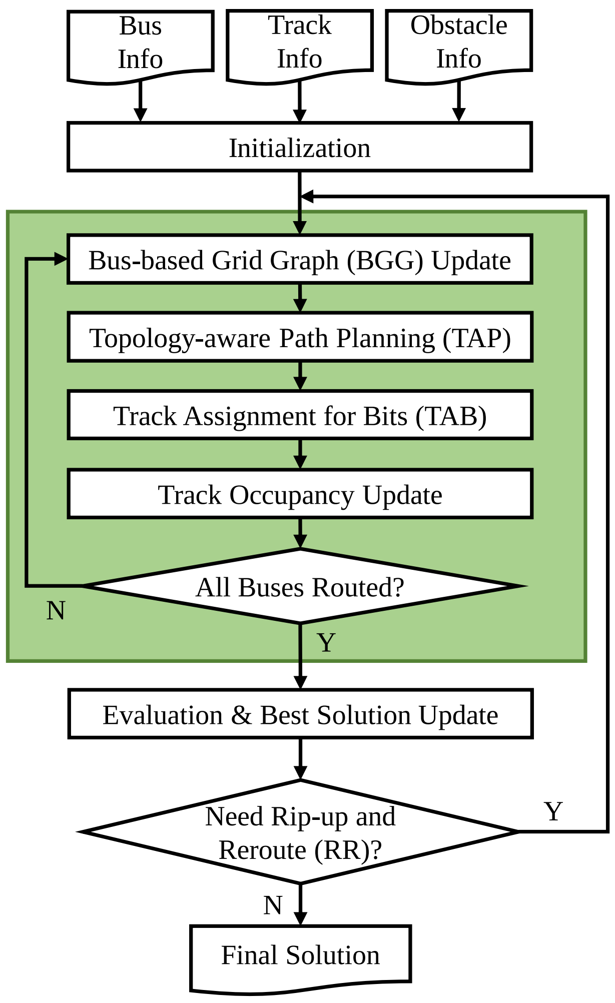

Bus Info
Track Info
Obstacle Info
Initialization
Bus-based Grid Graph (BGG) Update
Topology-aware Path Planning (TAP)
Track Assignment for Bits (TAB)
Track Occupancy Update
All Buses Routed?
N
Y
Evaluation & Best Solution Update
Need Rip-up and Reroute (RR)?
Y
N
Final Solution

Figure 6.4: Overall flow of MARCH.

constraint of the bus. The pins of the bus will be marked on the BGG. The bits will then be routed concurrently on BGG during TAP. In TAP, a row of grid graph cells (G-cells), named *frontline*, will propagate from the source pins to the sink pins, generating a routing path consisting of a set of rectangular regions (e.g. Figure 6.1 (c)) called TAP regions. The TAP regions will be used to guide TAB later on.

To check spacing violations, each track maintains its *track occupancy* which records the positions of the segments on the track that cannot be used because of the spacing violation with some neighboring obstacles or routed wires. When one tries to use a certain part of a track, an accurate number of spacing violations incurred can be obtained by checking the track occupancy. The track occupancy is maintained by a binary search tree (BST) for efficiency. After TAB, the track occupancies of all the tracks will be updated according to the track segments used by the previously routed bus.

After routing all the buses, a routing solution will be generated. An evaluator then computes C_{total} according to the metrics in Section 6.1.1. The best solution will be updated if a lower C_{total} is found. The evaluator results will also determine whether a RR process is needed. If so, history costs computed according to the detected spacing

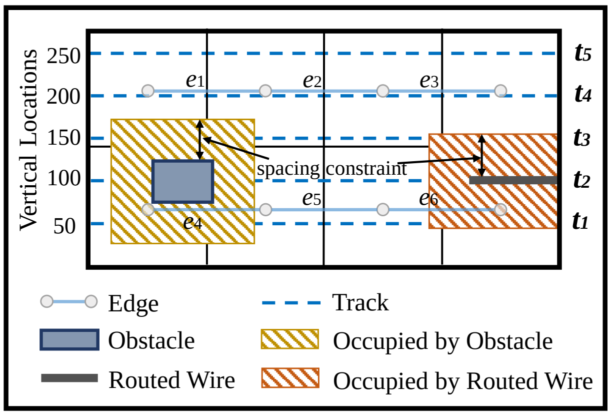


Figure 6.5: An example of computing edge capacity.

violations will be added to BGG, and the routing procedure will be restarted.

6.2.1 Bus-based Grid Graph (BGG)

BGG plays an important role in our algorithm since it provides the necessary information for cost estimation during TAP. BGG is a multi-layer grid graph with uniform G-cells. In each layer of BGG, there is an edge connecting adjacent G-cells along the layer's routing direction. The bits can be routed along the edge or switched to a neighboring G-cell on adjacent layers by via.

Each edge stores its own edge capacity and history cost. The edge capacity is computed during BGG update. It approximates the maximum number of tracks that meets the width constraint of the bus to be routed and can be used concurrently without causing any spacing violation. For instance in Figure 6.5, suppose the bus width is 10 and the spacing constraint is 50. To obey the spacing constraint among different bits of the same bus, neighboring tracks cannot be used concurrently for this example. Meanwhile, since the track with the lowest index will be checked first, the tracks t_1, t_3, and t_5 will be used to compute the edge capacity of the BGG in this example. By checking the track occupancies, the available segments on these tracks can be obtained (shown as dashed lines Figure 6.5). The edge capacity of an edge in the BGG is then computed as the number of available track segments with enough length to connect the two adjacent G-cells. In Figure 6.5, for the edge e_1, its edge capacity is 1 since only one available segment from t_5 has enough length to connect its two neighboring G-cells. The edge capacities of edges (e_1, e_2, ..., e_6) are (1, 2, 1, 0, 1, 0) respectively. For history cost, it can reflect the degree of routing congestion in the edge and will be accumulated when the rip-up and reroute process is performed.

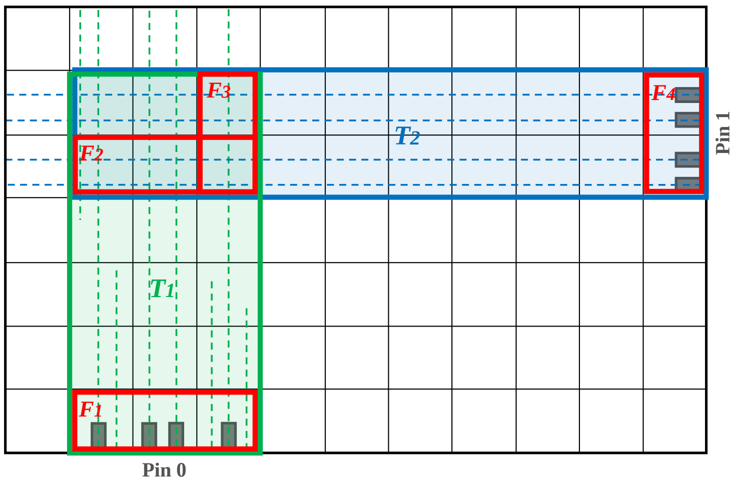


Figure 6.6: Result of topology-aware path planning (TAP).

6.2.2 Topology-Aware Path Planning (TAP)

In order to obtain a routable topology that all the bits can follow, TAP is needed to route the bits concurrently. In TAP, the frontline, which is a row of G-cells, will concurrently propagate to find a good path for all the bits. During initialization, the frontline sizes, i.e. the number of G-cells contained, are determined. Note that each bus has different frontline sizes on different layers because the track spacing on different layers varies. The frontline size for a layer considers both the bit number of the bus and the average edge capacity of that layer. The frontline sizes will all be computed at the beginning and can only be changed during rip-up and reroute.

6.2.2.1 A Toy Example

In Figure 6.6, the bus has two pins and four bits. The BGG has two layers where the frontline sizes of the bus are 3 and 2 respectively. The aim of TAP is to generate a ”path” to connect Pin 0 marked at F_1 and Pin 1 marked at F_4. The path consists of a set of TAP regions. It can be generated as follows. Starting from position F_1, the frontline will go up along the routing direction of the metal layer until reaching F_2. At F_2, the frontline will switch to the adjacent layer and reach F_3. Finally, the frontline will go right to reach the destination F_4. This path planner result consists of two connected TAP regions T_1 and T_2, formed by propagating the frontline on the same layer.

6.2.2.2 Same Layer Propagation

When propagating the frontline on the same layer, it is necessary to know the number of tracks that can be used currently. Thus, the frontline will maintain a set of values, called *running capacity*, where each value corresponds to a G-cell in the frontline. When propagating on the same layer, the values in the running capacity of the frontline will decrease if the capacities of edges the frontline goes through are smaller. It is because some tracks are broken midway, while the new ones will not be counted since the track needs to run from the beginning to the end. For instance, at F_1 of Figure 6.6, the running capacity (from the left G-cell to the right) of the frontline is (2, 2, 3). Reaching F_2, the running capacity becomes (1, 2, 1). The feasibility of the propagation can be determined by comparing the summation of the running capacity values with the bit number of the bus. The running capacity of the last frontline in a TAP region is considered as the running capacity of the TAP region. After switching layers, the running capacity will be reinitialized.

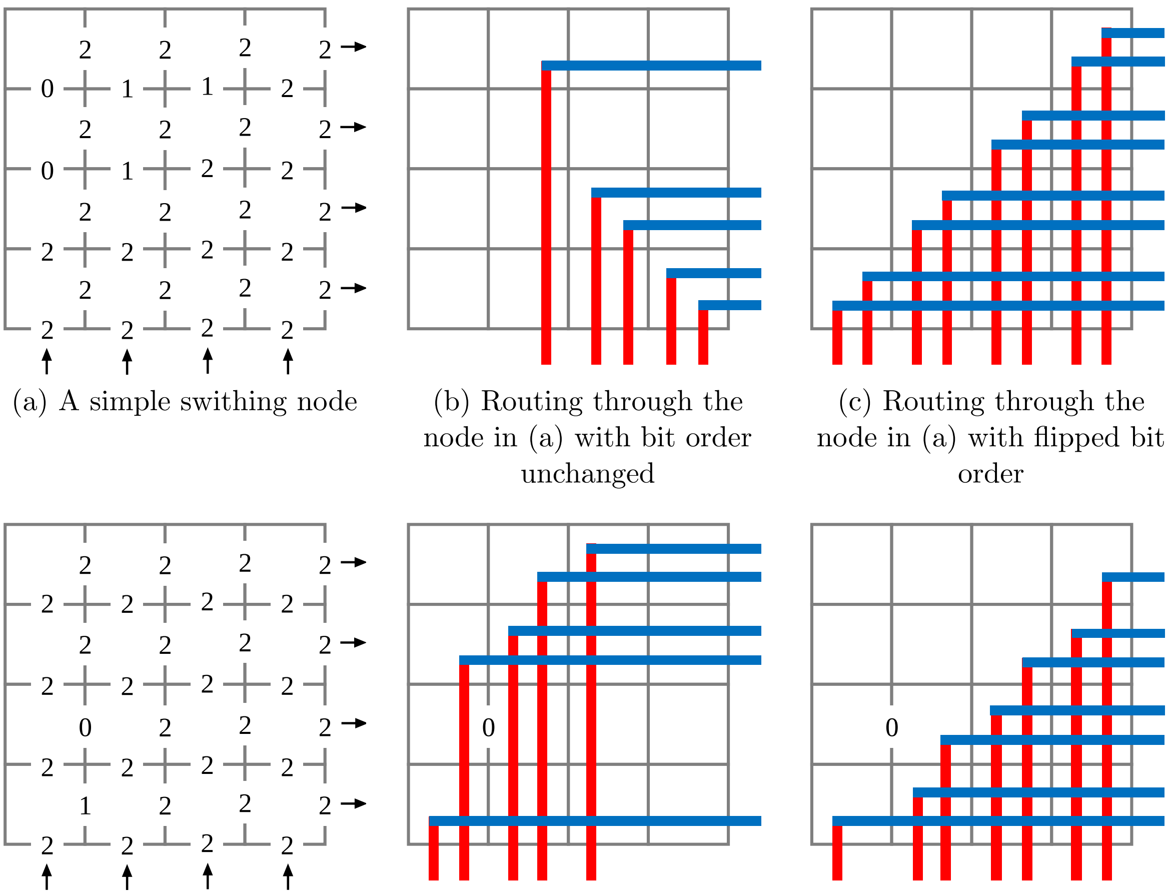
6.2.2.3 Layer Switching

Switching layers is the process that the frontline goes from one layer to its upper layer or lower layer which is usually of different routing direction. The main difficulty of switching layer is to decide whether it is safe or not to switch, or in another word, whether it will have enough routing resources without causing any spacing violation. Figure 6.7 (a) denotes a simple switching node which is comprised of 4×4 G-cells. In the following we will always assume that the wires enter the node from below and leave from the right without loss of generality, because other situations can be handled similarly. The number on each edge denotes the edge capacity. Moreover, there are two ways of passing through a switching node, one is passing with the bit order unchanged (Figure 6.7 (b)), and the other is passing with the bit order flipped (Figure 6.7 (c)).

To decide whether passing through a switching node is safe or not, we have to compute the maximum number of bits that can pass through a given node with bit order either flipped or not. Take the node in Figure 6.7(a) as an example, the number of bits that can pass through the node are very different for the flipping and not flipping cases. Keeping the bit order unchanged, the node allows at most 5 bits to pass through (Figure 6.7 (b)), whereas by flipping the bit order, up to 8 bits can be routed through the node (Figure 6.7 (c)).

We propose an efficient algorithm to compute the maximum bit number that can pass through a given switching node for the two cases. For the case of keeping the bit order unchanged, we start by routing the bits from the rightmost column, and continue to the left one by one. Every time, we will use up all the resources on a lower row before using the resources on an upper row. In this way, the maximum number of bits that can be routed can be counted.

For the case of flipping the bits, the situation is a lot more complicated, since a bit routed earlier may sometimes block the path of the bits to be routed later due to the topology constraints. Therefore, the greedy approach used in the former case is no longer applicable. For example, if we try to route through the node in Figure 6.7 (d) greedily from the leftmost to the rightmost column, only 5 bits can be routed (Figure 6.7 (e)).



(a) A simple swithing node

(b) Routing through the node in (a) with bit order unchanged

(c) Routing through the node in (a) with flipped bit order

(d) A switching node with a bottleneck edge

(e) Routing through the node in (d) with flipped bit order greedily

(f) Routing through the node in (d) with flipped bit order by giving up one bit

Figure 6.7: A 4 × 4 switching nodes.

However, if we give up 1 bit in the first column, we can end up routing 7 bits through (Figure 6.7 (f)). Unfortunately, it is usually unknown sacrificing which bits would give us better result before all the bits are routed, so both will be tried, and the better result will be adopted.

Algorithm 6.1 demonstrates our methodology to compute the maximum switching node capacity with flipped bit order. Assume that the columns are indexed $1, 2, ..., n$ from left to right, and the rows are indexed $1, 2, ..., m$ from bottom to top. The function NodeCapacityFlip(i, j) returns the maximum number of bits that can pass through the node with flipped bit order and under the constraint that the bits can only enter the columns with indexes larger than i, and leave from the rows with indexes larger than j. Hence, NodeCapacityFlip$(0, 0)$ will make use of the whole node, and returns the desired result of the maximum layer switching capacity with flipped bit order. The time complexity of the algorithm in the worst case is exponential. However, it can be finished very efficiently for most of the cases, because the runtime is linear for a congestion-free node, and will at most double when one bottleneck edge (Figure 6.7 (d)) exists in the

Algorithm 6.1 Compute $m \times n$ switching node capacity with flipped bit order

```
1: function GetFlippedCap($i$, $j$)
2:     if $i \geq n$ or $j \geq m$ then
3:         return 0
4:     end if
5:     $capacity \leftarrow$ max number of bits that can be routed from column $i$ to row $j$
6:     Record capacity change
7:     if no more bits can enter column $i$ then
8:         return $capacity$ + GetFlippedCap($i+1, j$)
9:     end if
10:    if no more bits can exit row $j$ then
11:        return $capacity$ + GetFlippedCap($i, j+1$)
12:    end if
13:    return $capacity$ + max(GetFlippedCap($i+1, j$), GetFlippedCap($i, j+1$))
14: end function
```

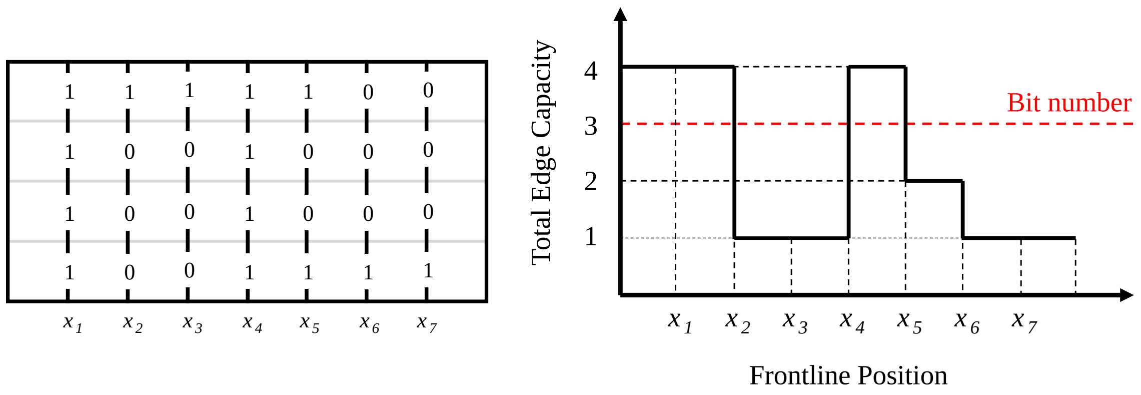
node. Empirically, very few nodes (less than 5%) contain such bottleneck edges. The actual percentage depends highly on the sufficiency of the routing resources as indicated by the BGG. Besides, the algorithm improves the overall runtime of TAP, because unsafe nodes are pruned in an early stage and no longer explored.

After exiting the switching nodes, the running capacity of each G-cell in the frontline will be reinitialized as the maximum number of bits that can exit from the G-cell for subsequent propagation.

6.2.2.4 Cost Estimation

During TAP, the actual cost that will be induced is not known yet, but it is essential to estimate the cost accurately in order to find a better path having potentially lower actual cost. The estimated cost is the summation of three values: wirelength cost, segment count cost and spacing violation cost. Note that the compactness cost is not taken into consideration, because our algorithm will always propagate with the most compact frontline and will enlarge its size only when necessary.

The first two costs are fairly easy to estimate. Therefore, in the following we will mainly focus on estimating the violation cost. We estimate the violation count based on the changes of the total edge capacity of the edges that the frontline passes through. More specifically, when the total edge capacity drops below the bit number of the bus, the amount of the drop below the bit number will be accumulated as violation count. For instance, consider a frontline with 4 G-cells moving from x_1 to x_7 as in Figure 6.8 (a), the capacity change in the frontline is illustrated in Figure 6.8 (b). If the bit number of the bus is 3, the total capacity drops below the bit number at x_2, x_5 and x_6 respectively. Consequently the estimated violation cost is calculated as $(3-1)+(3-2)+(2-1)=4$. This approach can well avoid counting the same violation multiple times. Additionally,



(a) BGG edge capacities in a TAP region

(b) Total edge capacity change when propagating from left to right

Figure 6.8: Spacing violation cost estimation.

the history cost will be added into the total cost at last.

6.2.3 Track Assignment for Bits (TAB)

TAB selects a track and also determines the exact positions on the track (called track segment range) to be used for each bit. However, this is a chicken-and-egg problem. On one hand, in a TAP region, the track segment range of a bit determines which track can be selected. On the other hand, the track selections also determines the exact positions on the tracks where the bits can be routed. For instance, the track selections of T_1 and T_2 determine the track segment ranges of each other in Figure 6.9. To handle this problem, TAB is conducted in four steps: rough track selection, track segment range estimation, exact track selection, and exact track segment range assignment.

Rough Track Selection First, the track for each bit is roughly selected by determining the column/row of G-cells where the bit will be routed. To perform this rough track selections for all the bits, a simple greedy method based on the running capacity of the TAP region is adopted. Take T_3 in Figure 6.9 as an example, assume that its running capacity is (3, 3) (as explained in Section 6.2.2.2). The bits will be roughly routed, following the bit order. Therefore, bits 1$\sim$3 will be routed in the leftmost column of G-cells, while bit 4 will be routed in the next column on the right. The rough track selections in other TAP regions (e.g. T_1 and T_2) will be performed in the same way.

Track Segment Range Estimation For a TAP region, its estimated track selection determines the track segment range estimations of its neighboring TAP regions (e.g. T_1 and T_3 affects T_2). For example, for bit 2 in T_2 of Figure 6.9, it needs to reach the middle column of G-cells in T_1 and the leftmost column in T_3. Thus, its track segment range in T_2 can be estimated conservatively as shown by the red solid line in Figure 6.9.

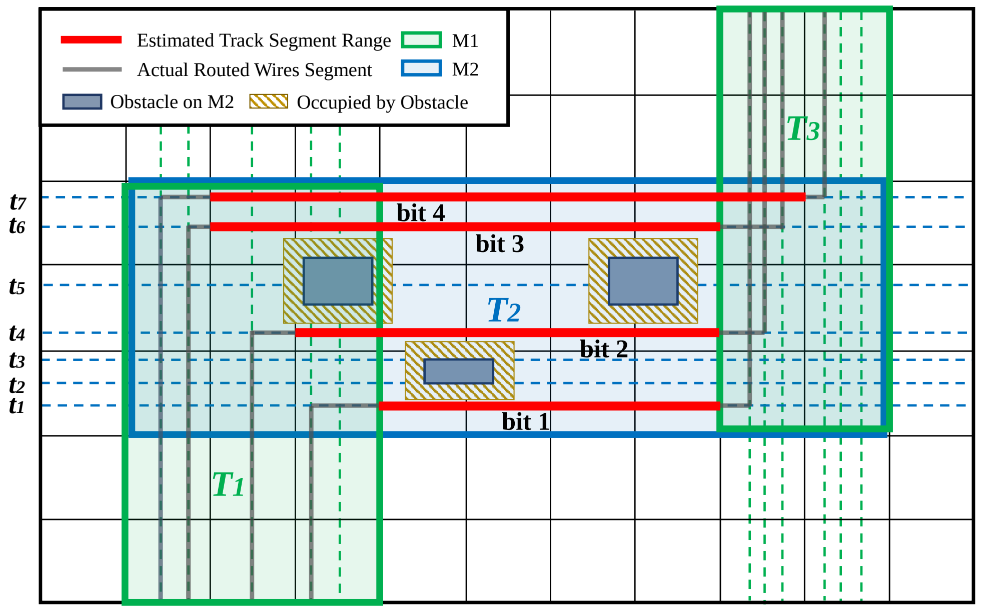


Figure 6.9: An example of track assignment for bits (TAB).

Exact Track Selection In each TAP region, with the estimated track segment ranges, the track for each bit can be exactly selected. The track meeting the following three requirements will be selected for the bit: (1) satisfying the width constraint of the bus, (2) with long enough segment, and (3) without spacing violation.

The last two requirements are checked with the information of the estimated track segment ranges. If a track cannot meet all the requirements for a bit, the next track will be attempted. In a horizontal (vertical) TAP region, the tracks will be attempted from bottom (left) to top (right). The track selection will follow the bit order. That is, the track selection for a bit will start from the track next to the track selected for the previous bit. For T_2 in Figure 6.9, the track t_1 will first be selected for bit 1 because it is long enough and violation-free. For bit 2, t_4 will be selected, instead of t_2 and t_3 because of spacing violation. The other bits will then processed in the same way. Suppose the spacing between each pair of tracks in T_2 does not violate spacing constraint, the exact track selection is shown in Figure 6.9.

Sometimes, there are not enough violation-free tracks in a TAP region, a violation threshold will be set which is an upper bound on the number of spacing violations caused by selecting a track. For each track, its track occupancy can give an accurate number of spacing violations incurred. This violation threshold will be incremented gradually from zero until finding an enough number of tracks.

Exact Track Segment Range Assignment After finishing exact track selections in all the TAP regions, the track segment range for each bit can be decided. One can see the actual routed wires segments in Figure 6.9.

Table 6.1: ICCAD 2018 Benchmark Statistics

	Characteristics				Metric Weights				
	# buses	# nets	# layers	# tracks	w_{wire}	w_{seg}	w_{com}	w_{space}	w_{fail}
`beta1`	34	1260	3	49209	5	1	5	8	2000
`beta2`	26	1262	3	49209	5	1	5	8	2000
`beta3`	60	665	3	22732	12	1	4	8	2000
`beta4`	62	698	3	22732	12	1	4	8	2000
`beta5`	6	1964	4	54150	8	1	5	8	2000
`final1`	18	1032	3	81226	10	1	5	10	2000
`final2`	70	1285	3	14209	10	1	5	10	2000
`final3`	47	852	4	21379	10	1	5	10	2000

6.2.4 Rip-up and Reroute Scheme

Rip-up and Reroute (RRR) is widely used in classic routing problems to negotiate between different nets routed in congested regions. The RRR scheme in MARCH will do two things: (1) Add history cost to the edge of BGG; (2) Enlarge the frontline size when violation-free routing resources are not sufficient.

After the evaluation, the wire segments violating spacing constraints will be known. The corresponding edges of BGG covered by these segments will be added a history cost. The history cost of an edge is accumulated by this equation: $h_{new} = \alpha \cdot n_{space} + \beta \cdot h_{old}$ where n_{space} is the number of spacing violations on this edge, and α and β are weights. When more iterations of RRR are executed, the congested regions on the BGG can be eliminated gradually.

Recall that each bus has different frontline sizes for different layers. For a bus, when the number of spacing violations in a TAP region of one layer is more than its bit number, this layer will be marked. In the next RRR, if the same problem occurs, the frontline size of the bus on that layer will be increased by one.

6.3 Experimental Results

We implement MARCH in C++. Experiments are performed on a 64-bit Linux workstation with Intel Xeon 3.4 GHz CPU and 32 GB memory. Benchmarks are from ICCAD 2018 Bus Routing Contest [138], the statistics of which is shown in Table 6.1. The runtime limit of each case is 1 hour.

The detailed scores of MARCH are shown in Table 6.2. It can be observed that the penalty cost $C_{fail} + C_{space}$ can be reduced to a level relatively smaller than the routing cost C_{route}.

Table 6.3 shows the comparison with the winners of ICCAD 2018 Contest[1]. Compared

[1]The scores of top 3 teams of ICCAD 2018 Contest are provided by the contest organizer. A binary is also obtained from the first place to get its runtime information.

Table 6.2: Detailed Results of MARCH

	C_{wire}	C_{seg}	C_{com}	C_{route}	N_{space}	N_{fail}	C_{total}	Time (s)
beta1	34	34	112	765	0	0	765	50
beta2	26	26	85	578	0	0	578	9
beta3	72	62	253	1942	0	0	1942	72
beta4	76	71	294	2165	0	0	2165	39
beta5	6	6	13	118	231	0	1966	12
final1	18	22	30	356	84	0	1196	352
final2	70	81	259	2071	148	0	3551	199
final3	47	51	558	3313	15	0	3463	133

* $C_{wire} = \sum_{bus\ b} C^b_{wire}$, $C_{seg} = \sum_{bus\ b} C^b_{seg}$, and $C_{com} = \sum_{bus\ b} C^b_{com}$

with them, MARCH not only reduces spacing violations greatly but also gets rid of all routing failures, achieving the best total cost C_{total} in seven out of eight cases. On average, C_{total} of MARCH is 2.130, 3.731, and 7.832 times better than the first, second and third places. At the same time, MARCH runs tremendously faster than the first place (with 105× speed-up on average). This indicates the effectiveness and efficiency of the concurrent and hierarchical scheme of MARCH.

Table 6.3: Comparison with Winners of ICCAD 2018 Contest

		C_{route}	C_{space}	C_{fail}	C_{total}	Time (s)
1st place	`beta1`	689	280	0	969	3600
	`beta2`	515	760	0	1275	3600
	`beta3`	1936	0	0	**1936**	**71**
	`beta4`	2192	0	0	2192	64
	`beta5`	119	1848	0	1967	3600
	`final1`	327	830	2000	3157	3317
	`final2`	1824	4500	8000	14324	3600
	`final3`	2966	490	10000	13456	3600
	Avg. Ratio				2.130	105.45
2nd place	`beta1`	701	5096	0	5797	-
	`beta2`	563	4904	0	5467	-
	`beta3`	2024	0	0	2024	-
	`beta4`	2271	0	0	2271	-
	`beta5`	95	616	2000	2711	-
	`final1`	367	2750	2000	5117	-
	`final2`	1890	2990	8000	12880	-
	`final3`	2678	300	2000	4978	-
	Avg. Ratio				3.731	
3rd place	`beta1`	641	8744	4000	13385	-
	`beta2`	484	9472	2000	11956	-
	`beta3`	1999	1928	0	3927	-
	`beta4`	2250	1048	0	3298	-
	`beta5`	98	1216	2000	3314	-
	`final1`	252	0	10000	10252	-
	`final2`	1976	6910	0	8886	-
	`final3`	4238	20	24000	28258	-
	Avg. Ratio				7.832	
MARCH	`beta1`	765	0	0	**765**	**50**
	`beta2`	578	0	0	**578**	**9**
	`beta3`	1942	0	0	1942	72
	`beta4`	2165	0	0	**2165**	**39**
	`beta5`	118	1848	0	**1966**	**12**
	`final1`	356	840	0	**1196**	**352**
	`final2`	2071	1480	0	**3551**	**199**
	`final3`	3313	150	0	**3463**	**133**
	Avg. Ratio				**1.000**	**1.000**

Chapter 7

3D IC Liquid Cooling Network

We propose novel thermal modeling and design optimization methodologies for liquid cooling networks in realistic 3D ICs, in order to achieve better trade-offs among energy efficiency, thermal gradient and peak temperature. Our major contributions are as follows.

- We develop a fast and accurate thermal modeling method for cooling networks.
- We propose some design guidelines and also a hierarchical tree-like cooling network structure based on an extensive experimental exploration. We calculate the limit of energy efficiency improvement under certain design constraints. This provides a guidance as well as an evaluation for the design optimization.
- We develop a novel search scheme to obtain a desirable configuration for the tree-like structure, under two problem formulations which minimizes pumping power and thermal gradient respectively. The result of our method outperforms the first place in the ICCAD 2015 Contest [139].

The rest of the chapter is organized as follows. Section 7.1 introduces the thermal modeling methods for liquid cooling network in 3D ICs. Section 7.2 provides the detailed problem formulations. Section 7.3 and Section 7.4 presents optimization methodology of designing a cooling system. The experimental results are detailed in Section 7.5.

7.1 Thermal Modeling

While thermal modeling for convectional air cooling has been well investigated [140], thermal modeling for liquid-cooled 3D ICs has recently received much attention [141–143], which however all assume unidirectional straight channels. The latest work 3D-ICE [143] is quite accurate, which has been validated by commercial computational fluid dynamics simulator and a real liquid-cooled 3D IC. The ICCAD 2015 Contest [139] extends it for flexible topology. Nevertheless, the extension only considers a 4-register model (4RM) and is slow. Therefore, we construct a fast thermal simulator for cooling network based

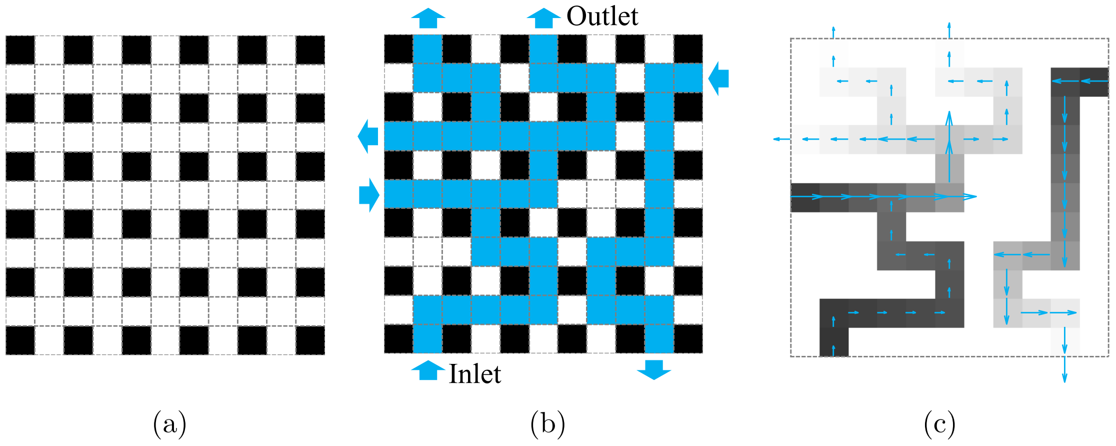


Figure 7.1: (a) Discretized channel layer where a basic cell either is TSV (black) or may be used for microchannels (white). (b) A cooling network. (c) Pressure and flow rate distribution where longer arrows represent larger flow rates and darker liquid cells have higher pressures.

on a 2-register model (2RM), which enables simulation in the inner loops of the design flow.

Before discussing details of our 2RM method, some preliminaries and 4RM method are briefly introduced.

7.1.1 Preliminaries

For a liquid *cooling system*, there are two variables: (1) *cooling network* $\boldsymbol{N}$, including its topology and positions of inlets and outlets; (2) *system pressure drop* P_{sys} across inlets and outlets.

To represent $\boldsymbol{N}$, we divide the channel layer into discrete *basic cells* with a 2D rectangular grid, assign solid/liquid properties to each basic cell, and designate the boundary liquid cells as inlets/outlets. An *inlet* or *outlet* is defined as the surface where the coolant flows into or out of the corresponding liquid cell. Besides, some basic cells are reserved for TSVs and thus not allowed to be liquid, as Figures 7.1(a) and 7.1(b) show.

7.1.2 Pressure and Flow Rate

To calculate the heat transfer caused by the flowing coolant, local flow rates should be known. Their relationship with P_{sys} here is not as trivial as that in straight-channel design. The following calculation is among liquid cells (with a number of n), inlets and outlets.

For fully developed laminar flow, the volumetric flow rate $Q_{i,j}$ from liquid cell i to its neighbor j is [144]:

$$Q_{i,j} = g_{fluid,i,j} \cdot (P_i - P_j), \tag{7.1}$$

where P_i and P_j are pressures at i and j respectively, and $g_{fluid,i,j}$ is the fluid conductance computed as $g_{fluid,i,j} = (D_h^2 A_c)/(32 l_{i,j} \mu)$. Here, $l_{i,j}$ is the distance between centers of i and j, μ is the coolant dynamic viscosity, A_c is the cross-sectional area, and D_h is the hydraulic diameter computed as $D_h = 2w_c h_c/(w_c + h_c)$ with w_c and h_c being the width and height of the channel. Besides, the flow rate at an inlet/outlet of cell i is calculated similarly with a smaller fluid conductance $g_{fluid,i,edge}$.

By assuming constant water density, there is volume conservation for cell i:

$$\sum_{j \in N_i} Q_{i,j} = 0, \tag{7.2}$$

where N_i is the set of neighboring cells and possible neighboring inlet/outlet of i.

For convenience, the pressure at the outlet P_{out} is put as zero, so pressure value at the inlet P_{in} is P_{sys}. Then, substituting (7.1) into (7.2) creates the following system of linear equations:

$$\boldsymbol{G} \cdot \boldsymbol{P} = \boldsymbol{Q}_{in}, \tag{7.3}$$

where $\boldsymbol{P} \in \mathbb{R}^n$ is the vector of all liquid cell pressures, $\boldsymbol{Q}_{in} \in \mathbb{R}^n$ is the vector about flow rates at inlets ($(\boldsymbol{Q}_{in})_i = g_{fluid,i,edge} P_{in}$, if cell i neighbors an inlet; otherwise, $(\boldsymbol{Q}_{in})_i = 0$), and $\boldsymbol{G} \in \mathbb{R}^{n \times n}$ is the conductance matrix:

$$\boldsymbol{G}_{i,j} = \begin{cases} \sum_{k \in N_i} g_{fluid,i,k}, & j = i, \\ -g_{fluid,i,j}, & j \in N_i, \\ 0, & \text{otherwise.} \end{cases} \tag{7.4}$$

Since $\boldsymbol{G}$ and $\boldsymbol{Q}_{in}$ are known, the pressure vector $\boldsymbol{P}$ can be solved. Local flow rates are then attained by (7.1). An example is in Figure 7.1(c).

7.1.3 4RM-Based Thermal Modeling

To model liquid cooling, heat transfer inside microchannels is incorporated into a lumped thermal resistance network. 4-register-model (4RM) based modeling [143] follows the microchannel geometry, where *thermal cells* are formed according to both the 2D grid defining basic cells and the stack layer division. Each thermal cell is then represented by its center as a node. There are totally three kinds of heat transfer: between solid and solid, between solid and liquid, and between liquid and liquid (shown as Figure 7.2). Calculation of solid-solid and solid-liquid heat transfer can be derived from 3D-ICE directly. For the liquid-liquid one, a liquid cell now may couple with two or more liquid cells due to branches, requiring a new modeling technique.

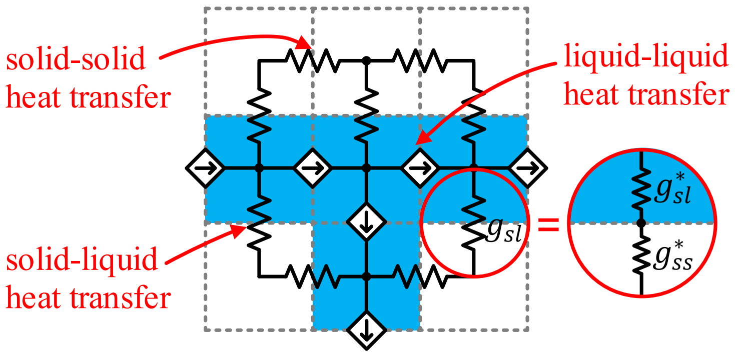

solid-solid heat transfer
liquid-liquid heat transfer
solid-liquid heat transfer
g_{sl}
g_{sl}^*
g_{ss}^*

Figure 7.2: 4RM model with three kinds of heat transfer.

The thermal conductance between two neighboring solid nodes i and j is:

$$g_{ss} = \frac{q_{i,j}}{T_i - T_j} = \frac{k_{solid} \cdot A_{i,j}}{l_{i,j}}, \tag{7.5}$$

where $q_{i,j}$ is the heat transfer from i to j, T_i and T_j are their temperatures, k_{solid} is the thermal conductivity of the solid material, and $A_{i,j}$ is the cross-sectional area, and $l_{i,j}$ is the distance between their centers.

There are two parts for the thermal conductance g_{sl} between a solid node i and its liquid neighbor j, the conductance g_{ss}^* from i to the channel wall, and the conductance g_{sl}^* from the channel wall to j:

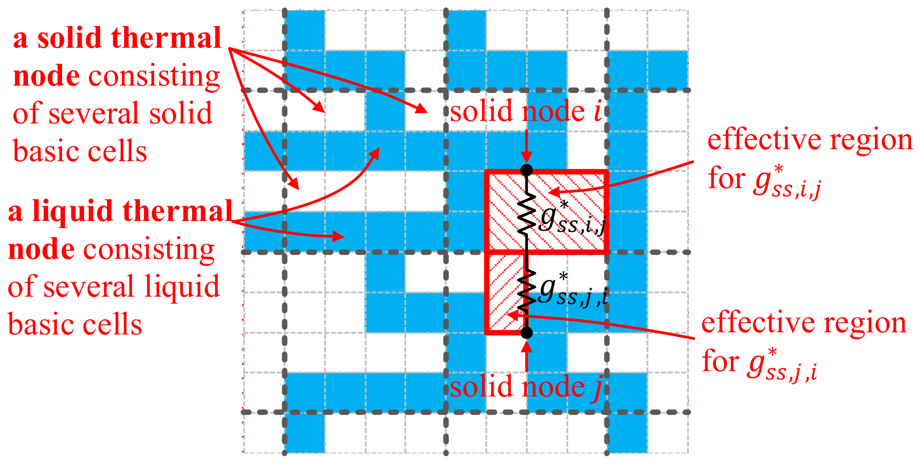
$$g_{sl} = \frac{q_{i,j}}{T_i - T_j} = g_{sl}^* \parallel g_{ss}^* = \frac{g_{sl}^* \cdot g_{ss}^*}{g_{sl}^* + g_{ss}^*}, \tag{7.6}$$

where g_{ss}^* is still calculated from (7.5), while g_{sl}^* is derived by $g_{sl}^* = (k_{liquid} A_{i,j} Nu)/D_h$ with k_{liquid} and Nu being the coolant thermal conductivity and Nusselt number [145] obtained by correlations.

For liquid-liquid heat transfer, a liquid cell receives thermal energy from upstream cells and send some energy to downstream cells. The net energy received by cell i is $q_{ll} = C_v \cdot \sum_{j \in N_i} (Q_{j,i} \cdot T_{j,i}^*)$, where C_v is the volumetric specific heat of the coolant, and $T_{j,i}^*$ represents the temperature at the corresponding boundary. The boundary is either inlet/outlet or the interface between two liquid cells. The temperature at inlet $T_{in,i}^* (= T_{in})$ is constant; that at outlet $T_{out,i}^*$ can be approximated by T_i; that at the interface between two cells $T_{j,i}^* = (T_j + T_i)/2$ under the central differencing scheme. Then together with (7.2), there is

$$q_{ll} = \frac{C_v}{2} \cdot \sum_{j \in N_i} (Q_{j,i} \cdot T_j). \tag{7.7}$$

Combining energy conservation for each cell, (7.5), (7.6) and (7.7), a system of linear equations similar to (7.3) can be created. Temperatures of all thermal cells are then solved from it.



a solid thermal node consisting of several solid basic cells
a liquid thermal node consisting of several liquid basic cells
solid node i
solid node j
effective region for $g^*_{ss,i,j}$
effective region for $g^*_{ss,j,i}$
$g^*_{ss,i,j}$
$g^*_{ss,j,i}$

Figure 7.3: 2RM model with discretization of 4×4 basic cells.

7.1.4 Faster 2RM-Based Thermal Modeling

As Section 7.5 will show, 4RM simulation is quite slow. For a three-die stack, it takes as much as 16 seconds to finish a simulation. This may be acceptable for final evaluation, but is forbidding for simulation inside the design flow, where the simulator usually needs to be invoked repeatedly. Therefore, we propose a simulation method in this section, which can be tremendously faster with limited accuracy loss.

Actually, it is not difficult to understand why 4RM simulation is slow, since it requires thermal cells to conform to the microchannel geometry. Freed from this constraint in the horizontal 2D discretization, thermal cells can be larger and thus fewer, accelerating the simulation. In [98, 143], the porous-medium approach (a.k.a. 2-register-model (2RM) based modeling) has applied the idea to straight microchannels. It is also applicable to general cooling network. The resulted speedup is necessary because the simulator usually needs to be invoked many times for designing a cooling system.

In 2RM, the horizontal 2D discretization is therefore coarser than basic cells. Figure 7.3 shows an example with a grid size of $m \times m$ ($m = 4$) basic cells. In the channel layer, basic cells in each grid are treated as two thermal nodes, a solid one and a liquid one, because of their diverse thermal properties and temperatures. In solid layers, a thermal node represents exactly $m \times m$ basic cells.

For the solid-solid heat transfer, the core calculation is still (7.5), but the corresponding geometry is no longer a simple cuboid. Take solid thermal nodes i and j in Figure 7.3 as an example. If node j represents a pure solid region, its effective region for calculating the thermal conductance with node i will be the upper half of the 4×4 grid. Now among the distributed solid basic cells, only complete conducting paths are taken into account. In this way, $g^*_{ss,j,i}$, the thermal conductance between node j and the interface, is obtained. Similarly, there is $g^*_{ss,i,j}$. The total conductance between nodes i and j is then computed as:

$$g_{ss,i,j} = g^*_{ss,i,j} \parallel g^*_{ss,j,i} = \frac{g^*_{ss,i,j} \cdot g^*_{ss,j,i}}{g^*_{ss,i,j} + g^*_{ss,j,i}}. \tag{7.8}$$

The above example is for a neighboring solid node pair in the same layer, but the approach is also applicable to a cross-layer pair (a solid node in the channel layer and its top/bottom node in solid layers).

For the solid-liquid heat transfer, both the vertical heat transfer (from top/bottom walls to the coolant) and the horizontal one (from side walls to the coolant) are considered only in the vertical direction [143]. That is, the thermal conductance between a liquid node and its side wall $g^*_{sl,side} = 0$. The side wall area A_{side} is added into the calculation of vertical heat transfer. The thermal conductance between a liquid node and its top/bottom wall is thus:

$$g^*_{sl,top/bottom} = h_{conv} \cdot (A_{top/bottom} + A_{side}/2), \tag{7.9}$$

where $A_{top/bottom}$ is its top/bottom wall area. The total solid-liquid conductance is then derived by (7.6).

The liquid-liquid heat transfer depends on flow rates between liquid thermal nodes. With possible multiple microchannel connections, total heat transfer is determined by the net flow rate and (7.7).

Similar to 4RM, we then obtain temperatures from a system of linear equations. In general, an $m \times m$ discretization reduces the problem size to $\frac{1}{m^2}$ of the 4RM one, and thus accelerates more than m^2 times (note that the exact value depends on the linear algebra (LA) solver used). Note that though only steady thermal analysis is discussed above, it can be easily extended to transient one.

7.2 Problem Formulations

By Bernoulli's equation, *pumping power* $W_{pump} = P_{sys}Q_{sys}/\eta$ with P_{sys} being the system pressure drop and Q_{sys} being the system flow rate. There is an efficiency term η because energy loss across components such as tubes, heat exchangers and pumps. However, η depends on the parts outside the cooling network and has no impact on the optimization procedure, so it will be removed from the upcoming calculation of W_{pump}. *Thermal gradient* is defined as $\Delta T = \max_i(\Delta T_i)$ with ΔT_i being the range of node temperatures in the i-th source layer [139]. *Peak temperature* T_{max} is the maximum of the thermal node temperatures. Note that T_{max} can only occur in the source layer due to energy conservation.

Besides, the following design rules are used to make the problem concrete and realistic:

- TSV positions are assumed to be at alternating basic cells in both dimensions, like Figure 7.1(b).

- Inlets and outlets can only occur at the edges of the channel layer.
- To reduce the complexity of packaging, there can be at most one "continuous" inlet and outlet on each side.

In fact, without the last rule, straight channels with alternating directions can compensate temperature rise from inlets to outlets of each other very well. However, it is unpractical for packaging.

Based on the above design rules, we present two problem formulations for trade-off among W_{pump}, ΔT and T_{max}. The first problem formulation is from ICCAD 2015 Contest [139], where W_{pump} should be minimized:

Problem 7.1 (Pumping Power Minimization)**.** *Given the heat dissipation of a 3D IC and some design rules, decide the cooling network and the system pressure drop of the cooling system, such that the pumping power is minimized, while the constraints on peak temperature and thermal gradient are satisfied.*

Liquid cooling causes large ΔT, bringing reliability issues and timing errors. Therefore, a second formulation treating it as the objective will also be discussed:

Problem 7.2 (Thermal Gradient Minimization)**.** *Given the heat dissipation of a 3D IC and some design rules, decide the cooling network and the system pressure drop of the cooling system, such that the thermal gradient is minimized, while the constraints on peak temperature and pumping power are satisfied.*

7.2.1 General Considerations

Among the three targets (i.e., W_{pump}, ΔT and T_{max}), we should take more care of ΔT. It is due to the following two reasons. First, for a specific cooling network $\boldsymbol{N}$, increasing W_{pump} (i.e. flow rates) will lower T_{max}, and vice versa, which is a simple trade-off between them. However, increasing W_{pump} will not necessarily lead to lower ΔT. Second, with liquid cooling, T_{max} is decreased already, while ΔT is higher when comparing with convectional heat sinks as mentioned in Section 2.2.5.

For ΔT, there are three inducing factors:

1. With a practical flow rate, temperature rise of the coolant will create uneven heat-sinking from inlet to outlet.
2. The power source distribution in active layers is probably non-uniform, making temperatures in different regions tend to differ.
3. For non-uniform channel distribution, some regions have less contact area with the coolant or are even far from the channel, which also creates uneven heat-sinking

Algorithm 7.1 Optimization Flow for Pumping Power Minimization

```
Require: $\boldsymbol{N}_{init}$, $\Delta T^*$, $T^*_{max}$, stack description and floorplan files.
Ensure: $\boldsymbol{N}$, $P_{sys}$.
1: $\boldsymbol{N} \leftarrow \boldsymbol{N}_{init}$
2: while # iteration is within the limit do
3:     Obtain neighboring network solution $\boldsymbol{N}'$;
4:     Get the score $W'_{pump}$ of $\boldsymbol{N}'$;                    ▷ Algorithm 7.2
5:     $\boldsymbol{N} \leftarrow \boldsymbol{N}'$ or not according to SA mechanism;
6:     if $W'_{pump}$ converges then return $\boldsymbol{N}$ and $P_{sys}$;
7:     end if
8: end while
```

Among them, the first two are unavoidable, but factor 3 can be used to compensate for them. For example, in a region with higher power source (factor 2), more channels can be assigned to achieve stronger heat-sinking (factor 3).

7.3 Minimizing Pumping Power

The ideas and technical details for solving pumping power minimization are introduced in this section. The extension to thermal gradient minimization will be explained in the next section.

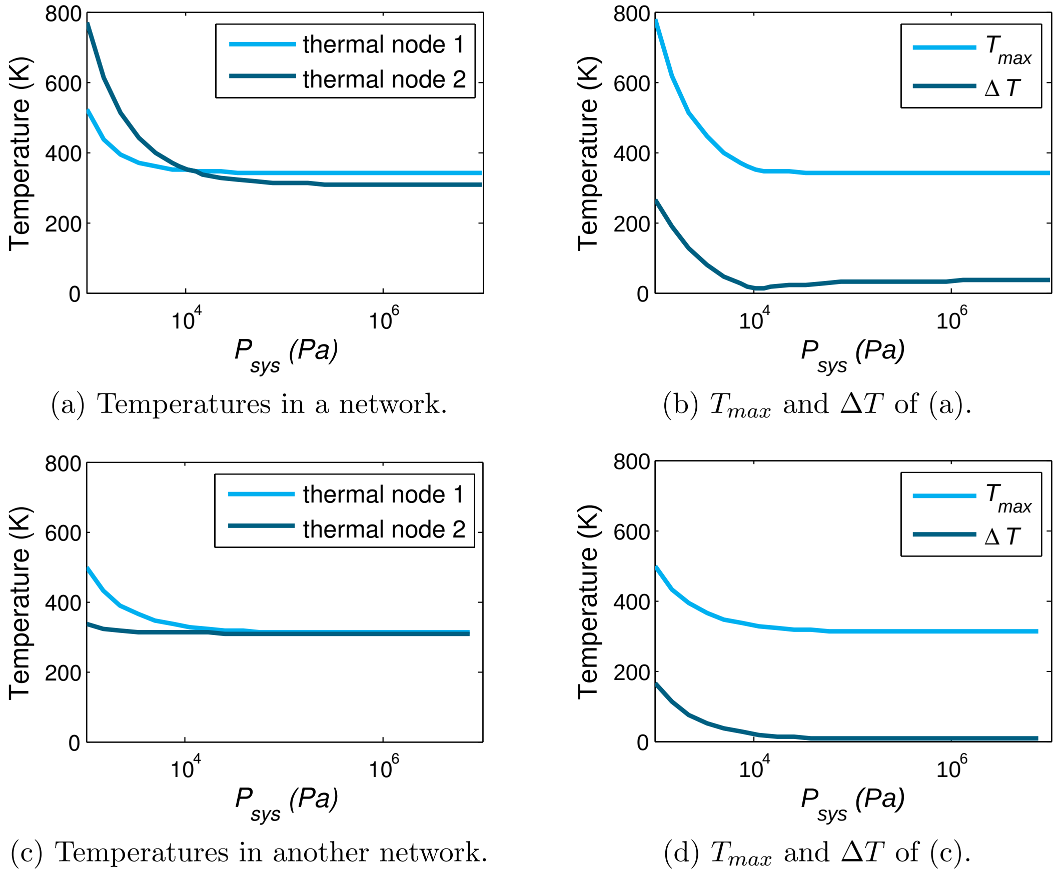
First of all, the mathematical formulation for Problem 7.1 can be written as follows:

$$\begin{aligned} \min \quad & W_{pump}, \\ \text{s.t.} \quad & P_{sys} \in \mathbb{R}^+, \ \boldsymbol{N} \in \mathcal{N}, \ T_{max} \le T^*_{max}, \ \Delta T \le \Delta T^*, \end{aligned} \tag{7.10}$$

where $\mathcal{N}$ is the set of all legal cooling networks, ΔT^* and T^*_{max} are the corresponding constraints.

The overall two-level optimization framework for (7.10) is shown in Algorithm 7.1. As mentioned in Section 7.1.1, there are two variables in each solution, i.e., the cooling network $\boldsymbol{N}$ and system pressure drop P_{sys}. For each solution, not only the cost function W_{pump} but also the constraints on ΔT and T_{max} need to be checked. We thus handle the problem in two levels according . In the inner level, P_{sys} is varied to minimize W_{pump} for a specific $\boldsymbol{N}$, which evaluates $\boldsymbol{N}$ by its *lowest feasible pumping power* W'_{pump} (line 4). In the outer level, simulated annealing (SA) searches for a good $\boldsymbol{N}$ solution according to W'_{pump}.

Before introducing the inner level (Section 7.3.2) and the outer level (Section 7.3.4), the relationship between P_{sys} and thermal profile is introduce first.



(a) Temperatures in a network. (b) T_{max} and ΔT of (a).

(c) Temperatures in another network. (d) T_{max} and ΔT of (c).

Figure 7.4: Relation between temperatures and P_{sys} in a network.

7.3.1 Relationship Between Pressure and Temperature

In general, as P_{sys} increases, temperatures of all thermal cells will decrease. When P_{sys} becomes sufficiently large, coolant temperature will be very close to T_{in} and approximately constant, making the temperatures of its neighboring solid cells also almost constant. Prior to this sufficiently large P_{sys}, temperature decrease is gradually smaller and becomes almost zero. We call it a *turning point*, as Figures 7.4(a) and 7.4(c) show. For different cells, turning points are different. More specifically, the temperatures of the upstream liquid cells are closer to T_{in} than those of the downstream ones. Hence, upstream regions reach turning points earlier.

Suppose $T_{max} = h(P_{sys})$ and $\Delta T = f(P_{sys})$. Since T_{max} is the maximum among node temperatures, h also decreases monotonically and finally becomes approximately constant, as P_{sys} increases. For ΔT, if thermal cells with later turning points become cooler than those with earlier turning points (e.g., Figure 7.4(a)), ΔT will begin rising at certain P_{sys}, as Figure 7.4(b) shows. Otherwise (e.g., Figure 7.4(c)), ΔT will keep dropping, as Figure 7.4(d) shows. Note that, in both cases, when P_{sys} is very large and all the nodes reach their turning points, ΔT becomes approximately constant. In short, f is either uni-modal (with minimum) or monotonically decreasing. The correctness of

Algorithm 7.2 Network Evaluation for Pumping Power Minimization

```
Require: $\boldsymbol{N}$, $\Delta T^*$, $T^*_{max}$.
Ensure: $W'_{pump}$.
1: Solve (7.12);                                        ▷ Algorithm 7.3
2: if $\Delta T > \Delta T^*$ then return $+\infty$;
3: else if $T_{max} > T^*_{max}$ then
4:     Minimize $P_{sys}$, s.t. $T_{max} \leq T^*_{max}$;
5:     if $\Delta T > \Delta T^*$ or $T_{max} > T^*_{max}$ then return $+\infty$;
6:     end if
7: end if
8: return $W'_{pump}$ corresponding to $P_{sys}$ ;
```

this analysis has been verified by extensive experiments.

7.3.2 Network Evaluation

A network $\boldsymbol{N}$ is evaluated by a lowest feasible pumping power W'_{pump}. For a specific $\boldsymbol{N}$, the relationship between W_{pump} and P_{sys} is monotonic:

$$W_{pump} = P_{sys} \cdot Q_{sys} = P^2_{sys}/R_{sys}, \tag{7.11}$$

where, R_{sys} is the system fluid resistance determined by $\boldsymbol{N}$. In this way, optimizing W_{pump} is equivalent to optimizing P_{sys}.

Based on the knowledge in Section 7.3.1, P_{sys} is minimized for a specific $\boldsymbol{N}$ in two steps (Algorithm 7.2). In the first step, the problem without constraint ΔT^* on ΔT is solved. By replacing W_{pump} with P_{sys}, ignoring T^*_{max} temporarily and substituting $\Delta T = f(P_{sys})$ in (7.10), the mathematical formulation is single-variable:

$$\begin{aligned} \min \quad & P_{sys}, \\ \text{s.t.} \quad & P_{sys} \in \mathbb{R}^+, \; f(P_{sys}) \leq \Delta T^*. \end{aligned} \tag{7.12}$$

Solving (7.12) is still difficult because: (1) f comes from numerical simulation so analytical method is not suitable; (2) f may not be monotonic; (3) probing f once is time-consuming. Thus, Algorithm 7.3 is carefully designed to achieve accuracy and speed, based on the analysis about f in Section 7.3.1. If a feasible P_{sys} exists (e.g., $\Delta T^* = \Delta T^*_1$ in Figure 7.5), it returns the optimal P_{sys} (i.e., P^*_1); otherwise (e.g., $\Delta T^* = \Delta T^*_2$), it returns the P_{sys} for minimum f (i.e., P^*_2), which in fact shows the nonexistence of a feasible P_{sys}.

The general idea of Algorithm 7.3 is moving three probing points of P_{sys} to search for the smaller P_{sys} for $f(P_{sys}) = \Delta T^*$ (line 19) or minimum f (line 10 and line 15). The initialization step (lines 1–6) makes sure that $f(P_0) > \Delta T^*$ and $f(P_0) > f(P_1)$, where P_{init} is the initial pressure and r_{init} is the initial step ratio.

In the second step (Algorithm 7.2 line 4), if T^*_{max} is violated, another binary search is

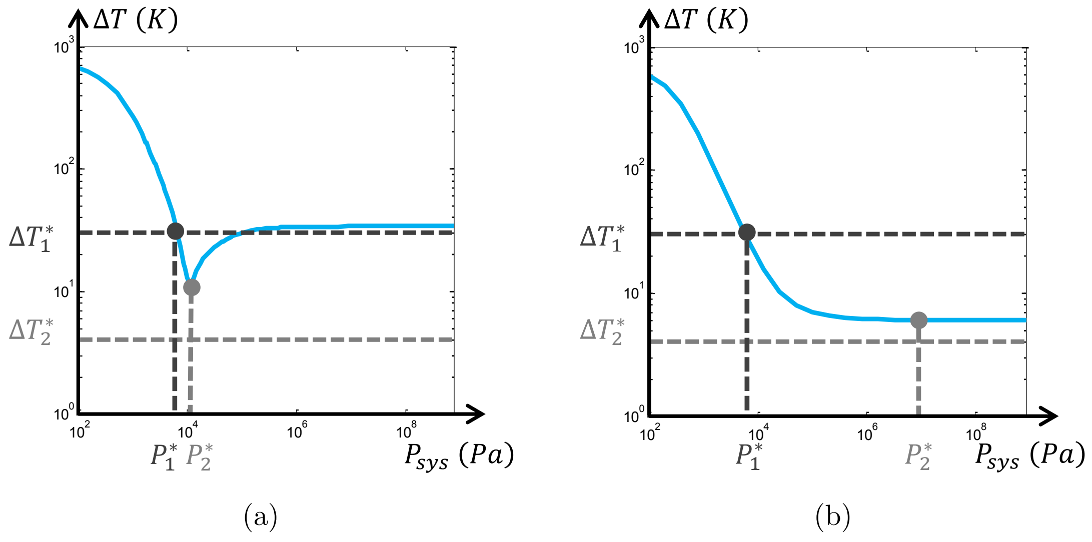


Figure 7.5: Cooling network with (a) uni-modal f and (b) monotonically decreasing f.

applied directly due to the monotonic h. Note that Algorithm 7.2 is optimal according to the properties of h and f stated in Section 7.3.1. The proof is easy and omitted.

7.3.3 Hierarchical Tree-like Cooling Network

In our early exploration, many cooling networks were designed manually with various styles. Among our observations, the most important one is that the thermal coupling between different regions in a chip is strong. For example, if the upstream region of a channel becomes slightly hotter, the temperature in the downstream region will be increased immediately. Therefore, global consideration is more significant than the subtle design in a local region.

Among the general structures attempted, the hierarchical tree-like structure is found to be simple (with a controllable number of parameters) and good (with respect to improving W_{pump} and ΔT under constraints). It includes several "trees" in which the coolant flows from roots to leafs, as Figure 7.6 shows. This structure also conforms to the general considerations in Section 7.2 and can: (1) make cooling in upstream and downstream regions more even by having different surface areas of the microchannel walls (i.e., compensate for factor 1); (2) make cooling of different trees more even by differing fluid resistance and thus flow rates (i.e., compensate for factor 2).

7.3.4 Network Topology Optimization

In the proposed tree-like network structure, each "tree" has two parameters to be configured, the positions for the first and the second branches. There are thus around 20 parameters for the whole cooling network (with the size of 101×101 basic cells). We design an SA-based algorithm to search for a good configuration of those parameters.

Algorithm 7.3 Algorithm Solving (7.12)

```
Require: $\boldsymbol{N}$, $\Delta T^*$.
Ensure: $P_{sys}$.
 1: $P_0 \leftarrow P_{init}$;
 2: while $f(P_0) < \Delta T^*$ do  $P_0 \leftarrow P_0/2$;
 3: end while
 4: $S \leftarrow P_0 \cdot r_{init}$, $P_1 \leftarrow P_0 + S$;
 5: if $f(P_0) < f(P_1)$ then $P_0 \leftarrow P_0/2$ and go to 2;
 6: end if
 7: while $f(P_1) > \Delta T^*$ do
 8:     $S \leftarrow 2 \cdot S$, $P_2 \leftarrow P_1 + S$;
 9:     while $f(P_1) < f(P_2)$ do
10:         if $|1 - \frac{P_0}{P_1}|$ and $|1 - \frac{P_2}{P_1}|$ are small enough then return $P_1$;
11:         end if
12:         $P_2 \leftarrow P_1$, $P_1 \leftarrow (P_0 + P_2)/2$, $S \leftarrow P_2 - P_1$;
13:     end while
14:     $P_0 \leftarrow P_1$, $P_1 \leftarrow P_2$;
15:     if keep moving right with small $|1 - \frac{f(P_0)}{f(P_1)}|$ then return $P_1$;
16:     end if
17: end while
18: Use binary search to find $P_{sys} \in [P_0, P_1]$ so that $f(P_{sys}) = \Delta T^*$;
19: return $P_{sys}$;
```

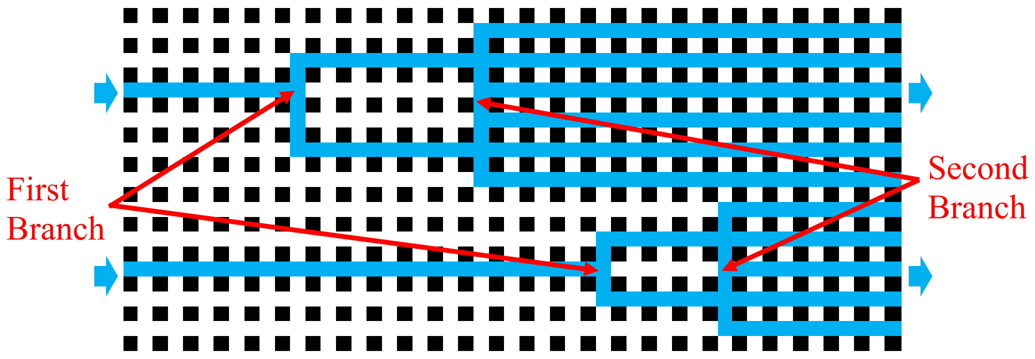


Figure 7.6: A tree-like cooling network on 23 × 51 basic cells.

Before searching, $\boldsymbol{N}$ is initialized with uniform tree parameters (i.e., same position for all first branches and same position for all second branches). There are totally four stages, each of which corresponds to a complete SA process described in Algorithm 7.1.

1. In each iteration, every tree parameter may be changed by a large step size or remains unchanged (with equal possibility). The acceptance of neighboring solutions are determined by SA. ΔT under a fixed P_{sys} is the cost function at this stage, which only needs simulating once. Note that a single simulation can also generate a result on W_{pump}, but under a fixed P_{sys}, W_{pump} reveals nothing about the die power (purely determined by $\boldsymbol{N}$) and thus is not eligible for the cost function. Besides, 2RM simulator is used for quick searching.

Table 7.1: Four-stage Optimization for Pumping Power Minimization

Stage	Step Size	Objective	Simulator	Runtime for an Iteration
1	10	ΔT	2RM	short
2	10	W'_{pump}	2RM	medium
3	2	W'_{pump}	2RM	medium
4	2	W'_{pump}	4RM	long

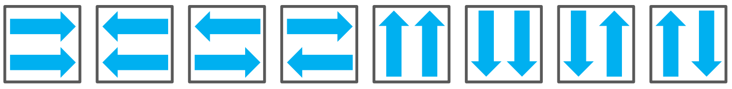


Figure 7.7: Eight types of global flow directions.

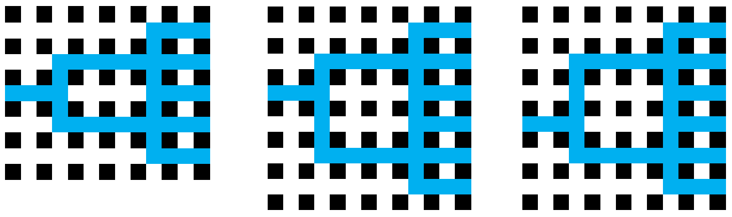


Figure 7.8: Three types of branches.

2. Stage 2 adopts the same move and simulator as stage 1 except that the neighboring solution is evaluated by the lowest feasible pumping power W'_{pump}. If there is no feasible P_{sys}, W'_{pump} is $+\infty$. This evaluating scheme needs invoking the simulator several times and takes longer runtime.

3. It is similar to stage 2, but a smaller step size is used.

4. It is similar to stage 3 except that the more accurate 4RM simulator is applied.

Settings for the four stages are summarized in Table 7.1. In general, earlier stages are rougher and much quicker. Therefore, with small runtime overhead, more rounds can be afforded to fully explore the solution space. In different rounds of a stage, all settings are the same except the random seed. After finishing a stage, the best solution in each round is re-evaluated by the metric in the next stage (if the metric is different). The re-evaluated best solution among all rounds is then selected as the output of the stage.

For the global flow directions (see Figure 7.7), all configurations are attempted and the best is chosen. Besides, there are three types of suitable branches (see Figure 7.8). They are assigned manually to fit the chip size.

7.4 Minimizing Thermal Gradient

Our method for minimizing thermal gradient (Problem 7.2) is still under the flow in Algorithm 7.1, but adaption is necessary for validity and quality.

Table 7.2: Three-stage Optimization for Thermal Gradient Minimization

Stage	Step Size	Objective	Simulator	Runtime for an Iteration
1	10	$\Delta T'$	2RM	short
2	10	$\Delta T'$	4RM	medium
3	2	$\Delta T'$	4RM	medium

First, its mathematical formulation in general becomes:

$$\begin{aligned} \min \quad & \Delta T, \\ \text{s.t.} \quad & P_{sys} \in \mathbb{R}^+,\ \boldsymbol{N} \in \mathcal{N}, \\ & T_{max} \le T^*_{max},\ W_{pump} \le W^*_{pump}, \end{aligned} \tag{7.13}$$

where W^*_{pump} is the constraint on W_{pump}.

Second, compared to the network evaluation process in Section 7.3.2, two modifications are needed. (1) Its simplified form corresponding to (7.12) becomes:

$$\begin{aligned} \min \quad & f(P_{sys}), \\ \text{s.t.} \quad & P_{sys} \in \mathbb{R}^+,\ P_{sys} \le P^*_{sys}, \end{aligned} \tag{7.14}$$

where P^*_{sys} is the constraint on P_{sys} computed by (7.11) and W^*_{pump}. (2) Solving (7.14) is simpler. If P^*_{sys} locates on the falling side of f, it is the optimal solution directly; otherwise, golden section search is adopted to find the minimum f.

Third, the network optimization process is similar to that in Section 7.3.4 with the following changes. (1) Objective function is changed from W_{pump} to ΔT. (2) Several consecutive iterations are grouped together, the first of which is evaluated normally by the complete network evaluation process. The other iterations are then evaluated through one simulation under a fixed P_{sys} (the optimal P_{sys} obtained in the first iteration). In this way, runtime is reduced significantly. The later evaluations may be pessimistic but inaccuracy is small because the optimal P_{sys} of neighboring $\boldsymbol{N}$ is close. (3) The original stage 1 is no longer needed due to the speed-up obtained from above technique 2. (4) The speed-up also makes 4RM affordable in the original stage 3. The summary of three stages for Problem 7.2 is in Table 7.2.

7.5 Experimental Results

Our thermal modeling and design optimization methods were implemented in C++ and with LA library Eigen [146]. Experiments were performed on an 80-core 2.20 GHz Linux server and with ICCAD 2015 Contest benchmarks [139]. In the benchmarks, the die is as large as $10.1mm \times 10.1mm$ and divided into 101×101 basic cells. Channel width $w_c = 100\mu m$ and inlet temperature $T_{in} = 300K$. More details are listed in Table 7.3,

Table 7.3: ICCAD 2015 Benchmark Statistics

#	Die Num	h_c (μm)	Die Power (W)	ΔT^* (K)	T^*_{max} (K)	Other Constraint
1	2	200	42.038	15	358.15	-
2	2	400	37.038	10	358.15	-
3	2	400	43.038	15	358.15	no channel in a restricted area
4	3	200	43.438	10	358.15	matched inlets/outlets across layers
5	2	400	148.174	10	338.15	-

where h_c is the channel height, T^*_{max} and ΔT^* are constraints on T_{max} and ΔT.

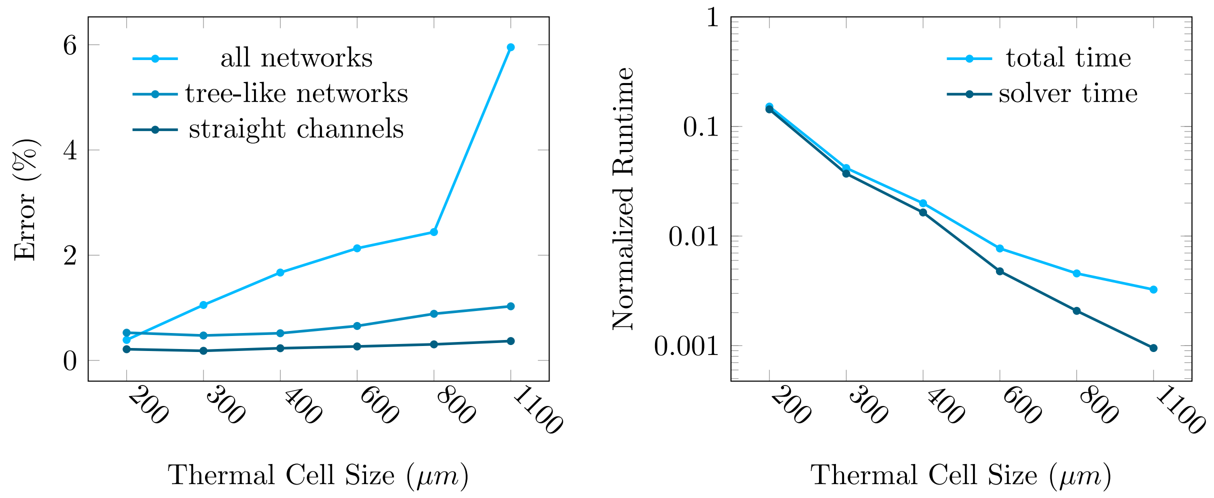
7.5.1 Effectiveness of 2RM Thermal Modeling

The 2RM method reduces the problem size and thus accelerates simulation significantly, which however may lose some accuracy. To examine whether the accuracy loss is limited, an experiment with 5 benchmarks, 40 network samples, 6 thermal cell sizes and 13 pressures is conducted. The network samples cover straight-channel networks, the proposed tree-like networks, and many styles of manual designs generated during our early exploration. They are also of diverse global flow directions. The pressures used is from 10^3 to $10^5 Pa$.

Amoung the $5 \times 40 \times 6 \times 13 = 15600$ 2RM simulations, the error of each is evaluated by its average relative error of thermal nodes in the source layers (compared with 4RM simulation). Errors of all networks with different benchmarks, different pressures and the same thermal cell size are then averaged. The same computation is also conducted for all tree-like networks and all straight-channel networks. The result in Figure 7.9(a) shows that accuracy decreases as the thermal cell size increases. Error is also affected by the network structure with straight-channel networks having the smallest. Besides, accuracy is also related to the benchmark and the pressure (details are omitted due to space limitation). Nevertheless, errors of 2RM simulation with small thermal cell sizes are all very small.

The runtime of 2RM depends on the thermal cell size. Figure 7.9(b) shows the runtime speed-up over 4RM model. Here, a 4RM simulation takes 3.37 s for test cases 1, 2, 3 and 5 (with two dies), and 15.62s for case 4 (with three dies).

In general, when the thermal cell size is small, the runtime saving by enlarging thermal cells is significant while accuracy loss is small. For example, simulating with tree-like networks and $400\mu m$ thermal cells results in only 0.517% errors compared with 4RM, but the runtime is reduced from 3.37s to 0.07s, which is a 50 times speed up. However, when thermal cell becomes very large, little runtime is consumed by the LA solver and the overhead dominates. The speed-up is thus increasingly less, while the accuracy still keeps worsening.



(a) Accuracy (b) Runtime

Figure 7.9: Accuracy and runtime of 2RM compared to 4RM.

7.5.2 Effectiveness of Design Optimization

As a good trade-off between accuracy and runtime, $400\mu m$ thermal cell is adopted for the 2RM simulations in solving pumping power minimization (Problem 7.1) and thermal gradient minimization (Problem 7.2). With the multi-core computer, 64 neighboring $\boldsymbol{N}$ solutions are evaluated simultaneously in each iteration.

As mentioned in Section 2.2.5, nearly all previous works about liquid-cooled 3D-ICs assume straight microchannels. We thus use regular straight-channel networks as baselines. For each test case, straight channels of diverse global directions are evaluated by the network evaluation process and the best is the baseline. For Problem 7.1, there is no feasible baseline solution on case 5, due to the high and highly varied die power, and tight T^*_{max}. The baseline experiments also provide a suitable P_{sys} for stage 1 of solving Problem 7.1. In case 3, there is a region forbidding microchannels. To satisfy the requirement, that region is filled by solid cells and surrounded by liquid cells, in both baseline networks and our tree-like network designs.

For solving Problem 7.1, the four stages consist of 60, 40, 40 and 30 iterations, and 8, 4, 2 and 1 round(s), respectively. The whole SA searching takes about 40 min for cases 1-3 and about 240 min for case 4. In the difficult case 5, SA cannot find a feasible solution with tree-like structure, so the cooling system is designed manually.

Because Problem 7.1 is exactly the formulation in ICCAD 2015 Contest, the contest benchmarks are used directly. The result is shown in Table 7.4. Compared with the baseline, our method achieves up to 84.03% improvement on W_{pump}. The W_{pump} of our approach also outperforms the first place in the ICCAD 2015 Contest[1] by 16.35% on

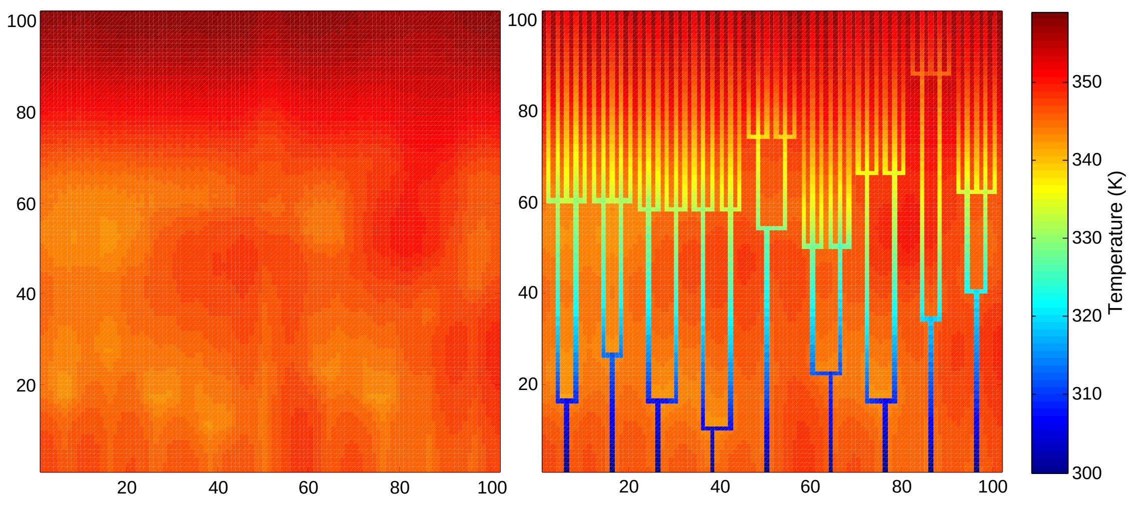
[1]Only the result of the first place is listed because the final score of the first is 29× and 2596× better than the second and third respectively, as reported by the contest organizer.

Table 7.4: Result for Pumping Power Minimization (Problem 7.1)

Case #		1	2	3	4	5
Baseline (straight channels)	P_{sys} (kPa)	12.98	6.23	7.85	9.71	N/A
	T_{max} (K)	322	314	321	314	N/A
	ΔT (K)	15.0	10.0	15.0	10.0	N/A
	$\boldsymbol{W_{pump}}$ $\boldsymbol{(mW)}$	**10.41**	**6.91**	**8.34**	**11.65**	**N/A**
Manual (1st place in ICCAD Contest)	P_{sys} (kPa)	8.86	5.54	6.98	9.45	40.1
	T_{max} (K)	357	336	328	336	338
	ΔT (K)	15.0	10.0	15.0	10.0	10.0
	$\boldsymbol{W_{pump}}$ $\boldsymbol{(mW)}$	**1.72**	**1.51**	**3.36**	**2.96**	**113.96**
Ours	P_{sys} (kPa)	8.72	5.13	5.81	8.27	40.10
	P_{system} (kPa)	358	336	337	335	338
	ΔT (K)	15.00	10.0	15.0	10.00	10.00
	$\boldsymbol{W_{pump}}$ $\boldsymbol{(mW)}$	**1.66**	**1.37**	**1.90**	**2.68**	**113.96**

Table 7.5: Result for Thermal Gradient Minimization (Problem 7.2)

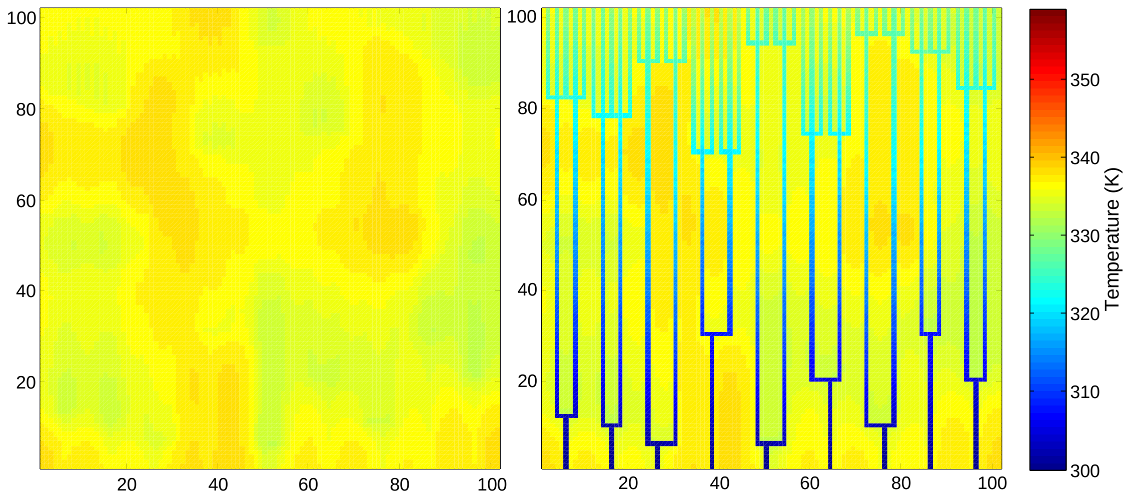
Case #		1	2	3	4	5
Baseline (straight channels)	P_{sys} (kPa)	26.08	14.43	17.82	26.51	45.81
	T_{max} (K)	316	309	316	308	338
	W_{pump} (mW)	42.0	37.0	43.0	43.4	148.2
	$\boldsymbol{\Delta T}$ $\boldsymbol{(K)}$	**8.75**	**5.42**	**11.42**	**4.76**	**26.48**
Ours	P_{sys} (kPa)	16.51	8.96	11.46	13.80	40.06
	T_{max} (K)	338	319	327	321	338
	W_{pump} (mW)	5.67	5.66	6.56	4.16	113.80
	$\boldsymbol{\Delta T}$ $\boldsymbol{(K)}$	**5.54**	**3.81**	**7.12**	**3.87**	**9.64**



(a) Bottom source layer (b) Channel layer

Figure 7.10: Temperature result of case 1 for pumping power minimization.

average. To the best of our knowledge, the network designs of the first place rely heavily on manual search, while our results are generated by automatic SA searching except for case 5.



(a) Bottom source layer (b) Channel layer

Figure 7.11: Temperature result of case 1 for thermal gradient minimization.

For solving Problem 7.2, the three stages consist of 80, 20 and 20 iterations, and 8, 2 and 1 round(s), respectively. The whole searching takes about 180 min for case 4, and 30 min for the others.

To evaluate our algorithm for Problem 7.2, all settings in the ICCAD 2015 benchmark are kept except that the objective is changed from W_{pump} to ΔT and ΔT^* is replaced by the constraint W^*_{pump} on W_{pump}. Table 7.5 shows the result when W^*_{pump} is set to 0.1% of the die power. For cases 1-4, our SA-based approach achieves as much as 37.65% improvement on ΔT compared to the baseline, with even smaller W_{pump}. Due to the difficulty of case 5, manual design is used, where the cooling network with flexible topology is still much better than the straight-channel network.

Figures 7.10 and 7.11 show the resulted temperature maps for case 1. Here, the map of pumping power minimization is hotter in general and implies smaller W_{pump}, but its ΔT is more significant. In the contrary, result of minimizing thermal gradient has much smaller ΔT with larger W_{pump}. In practice, the problem formulation can be chosen according to preference between W_{pump} and ΔT.

Conclusion

In this thesis, we propose a set of algorithms tackling the three aspects of challenges, single-net routing, multiple-net routing, and early-stage routability optimization, with the considerations of both practical VLSI design needs and mathematical rigorousness.

For single-net routing, two important basic multi-objective optimization Steiner tree construction problems are studied. First, we describe a novel Steiner SLT construction method called SALT, which is efficient and has the tightest bound over all the state-of-the-art general-graph SLT algorithms. Applying SALT to Manhattan space leads a smooth trade-off between RSMT and RSMA for VLSI routing. Cooperating with some post-processing techniques, it achieves superior trade-off between path length (or delay) and wirelength, compared to both classical and recent routing tree construction algorithms. A promising further work may be to integrate SALT into a complete routing optimization flow. Another line of research is to consider congestion when building the tree. Second, based on the intrinsic equivalence between ZST and HC, an efficient $O(1)$-approximation algorithm, Dim Sum, and an optimal dynamic programming, Optimal Dim Sum, are proposed for ZST construction. Besides, the optimal tree decomposition method enables a simple yet effective integration between ZST and RSMT. The directions of future work may include proving better approximation ratio for Dim Sum or designing even better algorithms based on the understanding of ZST-HC equivalence. For "BST by ZST", other clustering methods instead of the tree decomposition may worth trying. Besides, another line of research is to extend the idea to consider more practical factors (e.g, layer assignment, Elmore delay, etc).

Considering the cooperation and competition of different routing trees, we provide solutions to three multiple-net routing problems. First, we propose Dr. CU, an efficient and effective detailed router, to tackle the challenges in detailed routing. A set of two-level sparse data structures is designed for the routing grid graph of enormous size. An optimal path search algorithm is proposed to handle the minimum-area constraint. Besides, an efficient bulk synchronous parallel scheme is adopted to further reduce the runtime usage. Compared with state-of-the-art detailed routers, Dr. CU shows superior routing quality, runtime, and memory usage. Second, we propose MARCH for bus routing. Compared with classic net-by-net routing methods, MARCH routes all the bits of a bus concurrently for topology consistency. MARCH also has an efficient hierarchical framework, consisting of a coarse-grained TAP and a fine-grained TAB. To reduce the routing congestion, a RRR scheme is used. The experiments show that compared with the top contest teams, MARCH greatly reduces spacing violations and avoids any routing failure with competitive routing costs in a much shorter runtime. Third, we investigate liquid cooling networks for better trade-offs between energy efficiency and thermal profile. Specifically, we develop a fast and accurate 4RM-based thermal modeling method for liquid cooling networks. Design optimization methodologies which minimize pumping power and thermal gradient respectively are then proposed. Future work includes combining cooling networks with run-time thermal management techniques (e.g., DVFS and adjustable flow rates) to handle dynamic die power. Moreover, since channel layers are shared by TSVs and microchannels, another line of research is co-optimization between them for a better global benefit.

Bibliography

[1] Igor L Markov. Limits on fundamental limits to computation. *Nature*, 512(7513):147–154, 2014.

[2] Hadi Esmaeilzadeh, Emily Blem, Renée St Amant, Karthikeyan Sankaralingam, and Doug Burger. Power challenges may end the multicore era. *Communications of the ACM*, 56(2):93–102, 2013.

[3] 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 *ACM International Symposium on Physical Design*, pages 10–17, 2018.

[4] Andrew B Kahng, Jens Lienig, Igor L Markov, and Jin Hu. *VLSI physical design: from graph partitioning to timing closure*. Springer, 2011.

[5] Jason Cong. An interconnect-centric design flow for nanometer technologies. *Proceedings of the IEEE*, 89(4):505–528, 2001.

[6] Gengjie Chen, Peishan Tu, and Evangeline FY Young. SALT: provably good routing topology by a novel Steiner shallow-light tree algorithm. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 569–576, 2017.

[7] Gengjie Chen and Evangeline FY Young. SALT: provably good routing topology by a novel steiner shallow-light tree algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[8] Gengjie Chen and Evangeline FY Young. Dim Sum: Light clock tree by small diameter sum. In *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe*, 2019.

[9] Gengjie Chen, Chak-Wa Pui, Haocheng Li, Jinsong Chen, Bentian Jiang, and Evangeline FY Young. Detailed routing by sparse grid graph and minimum-area-captured path search. In *IEEE/ACM Asia and South Pacific Design Automation Conference*, pages 754–760, 2019.

[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 *ACM/IEEE Design Automation Conference*, 2019.

[11] Gengjie Chen, Jian Kuang, Zhiliang Zeng, Hang Zhang, Evangeline FY Young, and Bei Yu. Minimizing thermal gradient and pumping power in 3D IC liquid cooling network design. In *ACM/IEEE Design Automation Conference*, page 70, 2017.

[12] Jason Cong, Lei He, Cheng-Kok Koh, and Patrick H Madden. Performance optimization of VLSI interconnect layout. *Integration, the VLSI Journal*, 21(1-2):1–94, 1996.

[13] Thomas H Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[14] Maurice Hanan. On Steiner's problem with rectilinear distance. *SIAM Journal on Applied Mathematics*, 14(2):255–265, 1966.

[15] Michael R Garey and David S. Johnson. The rectilinear Steiner tree problem is NP-complete. *SIAM Journal on Applied Mathematics*, 32(4):826–834, 1977.

[16] David M Warme, Pawel Winter, and Martin Zachariasen. Exact algorithms for plane Steiner tree problems: A computational study. In *Advances in Steiner trees*, pages 81–116. Springer, 2000.

[17] Frank K Hwang. On Steiner minimal trees with rectilinear distance. *SIAM Journal on Applied Mathematics*, 30(1):104–114, 1976.

[18] Hai Zhou, Narendra Shenoy, and William Nicholls. Efficient minimum spanning tree construction without delaunay triangulation. *Information Processing Letters*, 81(5):271–276, 2002.

[19] J-M Ho, Gopalakrishnan Vijayan, and Chak-Kuen Wong. New algorithms for the rectilinear Steiner tree problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(2):185–193, 1990.

[20] Andrew B Kahng and Gabriel Robins. A new class of iterative Steiner tree heuristics with good performance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(7):893–902, 1992.

[21] Jeff Griffith, Gabriel Robins, Jeffrey S Salowe, and Tongtong Zhang. Closing the gap: Near-optimal Steiner trees in polynomial time. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(11):1351–1365, 1994.

[22] Manjit Borah, Robert Michael Owens, and Mary Jane Irwin. An edge-based heuristic for Steiner routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(12):1563–1568, 1994.

[23] Hai Zhou. Efficient Steiner tree construction based on spanning graphs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(5):704–710, 2004.

[24] Andrew B Kahng, Ion I Măndoiu, and Alexander Z Zelikovsky. Highly scalable algorithms for rectilinear and octilinear Steiner trees. In *IEEE/ACM Asia and South Pacific Design Automation Conference*, pages 827–833, 2003.

[25] Chris Chu and Yiu-Chung Wong. FLUTE: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(1):70–83, 2008.

[26] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[27] Weiping Shi and Chen Su. The rectilinear Steiner arborescence problem is NP-complete. *SIAM Journal on Computing*, 35(3):729–740, 2005.

[28] L Nastansky, SM Selkow, and NF Stewart. Cost-minimal trees in directed acyclic graphs. *Mathematical Methods of Operations Research*, 18(1):59–67, 1974.

[29] Jason Cong, Andrew B Kahng, and Kwok-Shing Leung. Efficient algorithms for the minimum shortest path Steiner arborescence problem with applications to VLSI physical design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(1):24–39, 1998.

[30] Sailesh K Rao, P Sadayappan, Frank K Hwang, and Peter W Shor. The rectilinear Steiner arborescence problem. *Algorithmica*, 7(1-6):277–288, 1992.

[31] Javier Córdova and Yann-Hang Lee. A heuristic algorithm for the rectilinear Steiner arborescence problem. Technical report, University of Florida, 1994.

[32] Jason Cong, Kwok-Shing Leung, and Dian Zhou. Performance-driven interconnect design based on distributed RC delay model. In *ACM/IEEE Design Automation Conference*, pages 606–611, 1993.

[33] Michael J Alexander and Gabriel Robins. New performance-driven fpga routing algorithms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1505–1517, 1996.

[34] Min Pan, Chris Chu, and Priyadarshan Patra. A novel performance-driven topology design algorithm. In *IEEE/ACM Asia and South Pacific Design Automation Conference*, pages 244–249, 2007.

[35] Baruch Awerbuch, Alan Baratz, and David Peleg. Effcient broadcast and lightweight spanners. Technical report, Weizmann Institute of Science, 1992.

[36] Jason Cong, Andrew B Kahng, Gabriel Robins, Majid Sarrafzadeh, and Chak-Kuen Wong. Provably good performance-driven global routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(6):739–752, 1992.

[37] Samir Khuller, Balaji Raghavachari, and Neal Young. Balancing minimum spanning trees and shortest-path trees. *Algorithmica*, 14(4):305–321, 1995.

[38] Charles J Alpert, TC Hu, JH Huang, Andrew B Kahng, and D Karger. Prim-Dijkstra tradeoffs for improved performance-driven routing tree design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(7):890–896, 1995.

[39] Michael Elkin and Shay Solomon. Steiner shallow-light trees are exponentially lighter than spanning ones. In *IEEE Symposium on Foundations of Computer Science*, pages 373–382, 2011.

[40] Michael Elkin and Shay Solomon. Steiner shallow-light trees are exponentially lighter than spanning ones. *SIAM Journal on Computing*, 44(4):996–1025, 2015.

[41] Stephan Held and Daniel Rotter. Shallow-light Steiner arborescences with vertex delays. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 229–241, 2013.

[42] Michael AB Jackson, Arvind Srinivasan, and Ernest S Kuh. Clock routing for high-performance ICs. In *ACM/IEEE Design Automation Conference*, pages 573–579, 1990.

[43] Jason Cong, Andrew B Kahng, and Gabriel Robins. Matching-based methods for high-performance clock routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(8):1157–1169, 1993.

[44] Ting-Hai Chao, Yu-Chin Hsu, Jan-Ming Ho, and AB Kahng. Zero skew clock routing with minimum wirelength. *IEEE Transactions on Circuits and Systems II*, 39(11):799–814, 1992.

[45] Ren-Song Tsay. An exact zero-skew clock routing algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(2):242–249, 1993.

[46] Masato Edahiro. A clustering-based optimization algorithm in zero-skew routings. In *ACM/IEEE Design Automation Conference*, pages 612–616, 1993.

[47] Moses Charikar, Jon Kleinberg, Ravi Kumar, Sridhar Rajagopalan, Amit Sahai, and Andrew Tomkins. Minimizing wirelength in zero and bounded skew clock trees. *SIAM Journal on Discrete Mathematics*, 17(4):582–595, 2004.

[48] Alexander Z Zelikovsky and Ion I Mandoiu. Practical approximation algorithms for zero-and bounded-skew trees. *SIAM Journal on Discrete Mathematics*, 15(1):97–111, 2002.

[49] Andrew B Kahng and Gabriel Robins. *On optimal interconnections for VLSI*, volume 301. Springer Science & Business Media, 1994.

[50] Jason Cong, Andrew B Kahng, Cheng-Kok Koh, and C-W Albert Tsao. Bounded-skew clock and Steiner routing. *ACM Transactions on Design Automation of Electronic Systems*, 3(3):341–388, 1998.

[51] David Papa, Charles Alpert, Cliff Sze, Zhuo Li, Natarajan Viswanathan, Gi-Joon Nam, and Igor Markov. Physical synthesis with clock-network optimization for large systems on chips. *IEEE/ACM International Symposium on Microarchitecture*, 31(4):51–62, 2011.

[52] Charles J Alpert, Gopal Gandham, Milos Hrkic, Jiang Hu, Andrew B Kahng, John Lillis, Bao Liu, Stephen T Quay, Sachin S Sapatnekar, and AJ Sullivan. Buffered steiner trees for difficult instances. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(1):3–14, 2002.

[53] Muhammet Mustafa Ozdal, Steven Burns, and Jiang Hu. Gate sizing and device technology selection algorithms for high-performance industrial designs. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 724–731, 2011.

[54] Stephan Held, Bernhard Korte, Jens Maβberg, Matthias Ringe, and Jens Vygen. Clock scheduling and clocktree construction for high performance ASICs. In *IEEE/ACM International Conference on Computer-Aided Design*, page 232, 2003.

[55] Chuan Yean Tan, Rickard Ewetz, and Cheng-Kok Koh. Clustering of flip-flops for useful-skew clock tree synthesis. In *IEEE/ACM Asia and South Pacific Design Automation Conference*, pages 507–512, 2018.

[56] Kwangsoo Han, Andrew B Kahng, Christopher Moyes, and Alex Zelikovsky. A study of optimal cost-skew tradeoff and remaining suboptimality in interconnect

tree constructions. In *ACM Workshop on System Level Interconnect Prediction*, page 2, 2018.

[57] Charles J Alpert, Dinesh P Mehta, and Sachin S Sapatnekar, editors. *Handbook of algorithms for physical design automation*. CRC press, 2008.

[58] Chin Yang Lee. An algorithm for path connections and its applications. *IRE transactions on electronic computers*, 10(3):346–365, 1961.

[59] Larry McMurchie and Carl Ebeling. Pathfinder: a negotiation-based performance-driven router for fpgas. In *ACM Symposium on FPGAs*, pages 111–117. ACM, 1995.

[60] Ryan Kastner, Elaheh Bozorgzadeh, and Majid Sarrafzadeh. Pattern routing: use and theory for increasing predictability and avoiding coupling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(7):777–790, 2002.

[61] Jarrod A Roy and Igor L Markov. High-performance routing at the nanometer scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(6):1066–1077, 2008.

[62] Min Pan and Chris Chu. FastRoute: A step to integrate global routing into placement. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 464–471, 2006.

[63] Min Pan and Chris Chu. Fastroute 2.0: A high-quality and efficient global router. In *IEEE/ACM Asia and South Pacific Design Automation Conference*, pages 250–255, 2007.

[64] Yue Xu, Yanheng Zhang, and Chris Chu. FastRoute 4.0: global router with efficient via minimization. In *IEEE/ACM Asia and South Pacific Design Automation Conference*, pages 576–581, 2009.

[65] Minsik Cho and David Z Pan. Boxrouter: a new global router based on box expansion and progressive ilp. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(12):2130–2143, 2007.

[66] Minsik Cho, Katrina Lu, Kun Yuan, and David Z Pan. BoxRouter 2.0: A hybrid and robust global router with layer assignment for routability. *ACM Transactions on Design Automation of Electronic Systems*, 14(2):32, 2009.

[67] Muhammet Mustafa Ozdal and Martin DF Wong. Archer: A history-based global routing algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(4):528–540, 2009.

[68] Yen-Jung Chang, Yu-Ting Lee, Jhih-Rong Gao, Pei-Ci Wu, and Ting-Chi Wang. NTHU-Route 2.0: a robust global router for modern designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(12):1931–1944, 2010.

[69] 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. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(5):709–722, 2013.

[70] Christoph Albrecht. Global routing by new approximation algorithms for multicommodity flow. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(5):622–632, 2001.

[71] Tai-Hsuan Wu, Azadeh Davoodi, and Jeffrey T Linderoth. GRIP: Global routing via integer programming. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(1):72–84, 2011.

[72] Michael Gester, Dirk Müller, Tim Nieberg, Christian Panten, Christian Schulte, and Jens Vygen. BonnRoute: Algorithms and data structures for fast and good vlsi routing. *ACM Transactions on Design Automation of Electronic Systems*, 18(2):32, 2013.

[73] Stephan Held, Dirk Muller, Daniel Rotter, Rudolf Scheifele, Vera Traub, and Jens Vygen. Global routing with timing constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(2):406–419, 2018.

[74] Yanheng Zhang and Chris Chu. RegularRoute: An efficient detailed router applying regular routing patterns. *IEEE Transactions on Very Large Scale Integration Systems*, 21(9):1655–1668, 2013.

[75] Stefanus Mantik, Gracieli Posser, Wing-Kai Chow, Yixiao Ding, and Wen-Hao Liu. ISPD 2018 initial detailed routing contest and benchmarks. In *ACM International Symposium on Physical Design*, pages 140–143, 2018.

[76] 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. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(10):1557–1568, 2013.

[77] Markus Ahrens, Michael Gester, Niko Klewinghaus, Dirk Müller, Sven Peyer, Christian Schulte, and Gustavo Tellez. Detailed routing algorithms for advanced technology nodes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(4):563–576, 2015.

[78] Muhammet Mustafa Ozdal. Detailed-routing algorithms for dense pin clusters in integrated circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(3):340–349, 2009.

[79] Tim Nieberg. Gridless pin access in detailed routing. In *ACM/IEEE Design Automation Conference*, pages 170–175, 2011.

[80] Xiaoqing Xu, Yibo Lin, Vinicius Livramento, and David Z Pan. Concurrent pin access optimization for unidirectional routing. In *ACM/IEEE Design Automation Conference*, page 20, 2017.

[81] Qiang Ma, Hongbo Zhang, and Martin DF Wong. Triple patterning aware routing and its comparison with double patterning aware routing in 14nm technology. In *ACM/IEEE Design Automation Conference*, pages 591–596, 2012.

[82] Yen-Hung Lin, Bei Yu, David Z Pan, and Yih-Lang Li. Triad: A triple patterning lithography aware detailed router. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 123–129, 2012.

[83] Zhiqing Liu, Chuangwen Liu, and Evangeline F. Y. Young. An effective triple patterning aware grid-based detailed routing approach. In *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe*, pages 1641–1646, 2015.

[84] Yixiao Ding, Chris Chu, and Wai-Kei Mak. Self-aligned double patterning-aware detailed routing with double via insertion and via manufacturability consideration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(3):657–668, 2018.

[85] Yu-Hsuan Su and Yao-Wen Chang. VCR: Simultaneous via-template and cut-template-aware routing for directed self-assembly technology. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 49:1–49:8, 2016.

[86] Andrew B Kahng, Lutong Wang, and Bangqi Xu. TritonRoute: an initial detailed router for advanced vlsi technologies. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 81:1–81:8, 2018.

[87] 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 *IEEE/ACM International Conference on Computer-Aided Design*, pages 82:1–82:7, 2018.

[88] Hui Kong, Tan Yan, and Martin D. F. Wong. Optimal simultaneous pin assignment and escape routing for dense PCBs. In *IEEE/ACM Asia and South Pacific Design Automation Conference*, pages 275–280, 2010.

[89] Qiang Ma, Evangeline F. Y. Young, and Martin D. F. Wong. An optimal algorithm for layer assignment of bus escape routing on PCBs. In *ACM/IEEE Design Automation Conference*, pages 176–181, 2011.

[90] Pei-Ci Wu, Qiang Ma, and Martin D. F. Wong. An ILP-based automatic bus planner for dense PCBs. In *IEEE/ACM Asia and South Pacific Design Automation Conference*, pages 181–186, 2013.

[91] Derong Liu, Bei Yu, Vinicius Livramento, Salim Chowdhury, Duo Ding, Huy Vo, Akshay Sharma, and David Z Pan. Synergistic topology generation and route synthesis for on-chip performance-critical signal groups. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, page 18, 2018.

[92] Shekhar Borkar. 3D integration for energy efficient system design. In *ACM/IEEE Design Automation Conference*, pages 214–219, 2011.

[93] C. Serafy et al. Unlocking the true potential of 3-D CPUs with microfluidic cooling. *IEEE Transactions on Very Large Scale Integration Systems*, 24(4):1515–1523, 2016.

[94] Thomas Brunschwiler, B. Michel, Hugo Rothuizen, U. Kloter, B. Wunderle, H. Oppermann, and H. Reichl. Forced convective interlayer cooling in vertically integrated packages. In *IEEE Intersociety Thermal Conference (ITherm)*, pages 1114–1125, 2008.

[95] Caleb Serafy, Bing Shi, Anurag Srivastava, and Donald Yeung. High performance 3D stacked DRAM processor architectures with micro-fluidic cooling. In *IEEE International 3D Systems Integration Conference*, pages 1–8, 2013.

[96] Bing Dang, Muhannad S. Bakir, Deepak Chandra Sekar, Calvin R. King Jr, and James D. Meindl. Integrated microfluidic cooling and interconnects for 2D and 3D chips. *IEEE Transactions on Advanced Packaging*, 33(1):79–87, 2010.

[97] Calvin R. King Jr, Jesal Zaveri, Muhannad S. Bakir, and James D. Meindl. Electrical and fluidic C4 interconnections for inter-layer liquid cooling of 3D ICs. In *IEEE Electronic Components and Technology Conference*, pages 1674–1681, 2010.

[98] T. Brunschwiler, S. Paredes, U. Drechsler, B. Michel, W. Cesar, Y. Leblebici, B. Wunderle, and H. Reichl. Heat-removal performance scaling of interlayer cooled chip stacks. In *IEEE Intersociety Thermal Conference (ITherm)*, pages 1–12, 2010.

[99] Mohamed M. Sabry, Arvind Sridhar, Jie Meng, Ayse K. Coskun, and David Atienza. GreenCool: An energy-efficient liquid cooling design technique for 3-D mpsocs via channel width modulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(4):524–537, 2013.

[100] Suresh V. Garimella, Vishal Singhal, and Dong Liu. On-chip thermal management with microchannel heat sinks and integrated micropumps. *Proceedings of the IEEE*, 94(8):1534–1548, 2006.

[101] Hanhua Qian, Chip-Hong Chang, and Hao Yu. An efficient channel clustering and flow rate allocation algorithm for non-uniform microfluidic cooling of 3D integrated circuits. *Integration, the VLSI Journal*, 46(1):57–68, 2013.

[102] Bing Shi, Ankur Srivastava, and Avram Bar-Cohen. Hybrid 3D-IC cooling system using micro-fluidic cooling and thermal TSVs. In *IEEE Annual Symposium on VLSI*, pages 33–38, 2012.

[103] Bing Shi and Anurag Srivastava. Optimized micro-channel design for stacked 3-D-ICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(1):90–100, 2014.

[104] Mohamed M. Sabry, Ayse K. Coskun, David Atienza, Tajana Šimunić Rosing, and Thomas Brunschwiler. Energy-efficient multiobjective thermal control for liquid-cooled 3-D stacked architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(12):1883–1896, 2011.

[105] Tijs Van Oevelen and Martine Baelmans. Numerical topology optimization of heat sinks. In *International Heat Transfer Conference*, pages 10–15, 2014.

[106] Christoph Bartoschek, Stephan Held, Jens Maßberg, Dieter Rautenbach, and Jens Vygen. The repeater tree construction problem. *Information Processing Letters*, 110(24):1079–1083, 2010.

[107] Stephan Held and Benjamin Rockel. Exact algorithms for delay-bounded Steiner arborescences. In *ACM/IEEE Design Automation Conference*, pages 44:1–44:6, 2018.

[108] Zhuo Li, Charles J Alpert, Shiyan Hu, Tuhin Muhmud, Stephen T Quay, and Paul G Villarrubia. Fast interconnect synthesis with layer assignment. In *ACM International Symposium on Physical Design*, pages 71–77, 2008.

[109] Rudolf Scheifele. RC-aware global routing. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 21:1–21:8, 2016.

[110] Rudolf Scheifele. Steiner trees with bounded RC-delay. *Algorithmica*, 78(1):86–109, 2017.

[111] Shay Solomon. Euclidean steiner shallow-light trees. In *International Symposium on Computational Geometry*, page 454, 2014.

[112] Shay Solomon. Euclidean steiner shallow-light trees. *Journal of Computational Geometry*, 6(2):113–139, 2015.

[113] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD Conference*, pages 47–57, 1984.

[114] Kenneth D Boese, Andrew B Kahng, and Gabriel Robins. High-performance routing trees with identified critical sinks. In *ACM/IEEE Design Automation Conference*, pages 182–187, 1993.

[115] Sheng-En David Lin and Dae Hyun Kim. Construction of all rectilinear Steiner minimum trees on the hanan grid. In *ACM International Symposium on Physical Design*, pages 18–25, 2018.

[116] Myung-Chul Kim, Jin Hu, Jiajia Li, and Natarajan Viswanathan. ICCAD-2015 CAD Contest in incremental timing-driven placement and benchmark suite. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 921–926, 2015.

[117] Rui Xu and Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005.

[118] Masato Edahiro. Equispreading tree in Manhattan distance. *Algorithmica*, 16(3):316–338, 1996.

[119] Robert J Fowler, Michael S Paterson, and Steven L Tanimoto. Optimal packing and covering in the plane are NP-complete. *Information Processing Letters*, 12(3):133–137, 1981.

[120] Teofilo F Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[121] Tomás Feder and Daniel Greene. Optimal algorithms for approximate clustering. In *ACM Symposium on Theory of computing*, pages 434–444, 1988.

[122] Anna Großwendt and Heiko Röglin. Improved analysis of complete-linkage clustering. *Algorithmica*, 78(4):1131–1150, 2017.

[123] Marcel R Ackermann, Johannes Blömer, Daniel Kuntze, and Christian Sohler. Analysis of agglomerative clustering. *Algorithmica*, 69(1):184–215, 2014.

[124] Drago Krznaric and Christos Levcopoulos. Optimal algorithms for complete linkage clustering in d dimensions. *Theoretical Computer Science*, 286(1):139–149, 2002.

[125] Sergei N Bespamyatnikh. An optimal algorithm for closest-pair maintenance. *Discrete & Computational Geometry*, 19(2):175–195, 1998.

[126] Sanjoy Dasgupta and Philip M Long. Performance guarantees for hierarchical clustering. *Journal of Computer and System Sciences*, 70(4):555–569, 2005.

[127] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[128] Cliff N Sze, Phillip Restle, Gi-Joon Nam, and Charles Alpert. ISPD 2009 clock network synthesis contest. In *ACM International Symposium on Physical Design*, pages 149–150, 2009.

[129] Cliff N Sze. ISPD 2010 high performance clock network synthesis contest: benchmark suite and results. In *ACM International Symposium on Physical Design*, pages 143–143, 2010.

[130] Dennis JH Huang, Andrew B Kahng, and Chung-Wen Albert Tsao. On the bounded-skew clock and Steiner routing problems. In *ACM/IEEE Design Automation Conference*, pages 508–513, 1995.

[131] VLSI CAD software bookshelf: Bounded-skew clock tree routing. `http://vlsicad.ucsd.edu/GSRC/bookshelf/Slots/BST/`.

[132] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[133] Edsger W Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[134] Boost geometry library. `http://www.boost.org/doc/libs/1_67_0/libs/geometry/doc/html/`.

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

[136] Cadence Innovus Implementation System. `http://www.cadence.com`.

[137] ISPD 2018 Contest. `http://www.ispd.cc/contests/18/`.

[138] ICCAD 2018 Contest. `http://iccad-contest.org/2018/`.

[139] Arvind Sridhar, Mohamed M Sabry, and David Atienza. ICCAD 2015 Contest in 3D interlayer cooling optimized network. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 912–915, 2015.

[140] Wei Huang, Shougata Ghosh, Siva Velusamy, Karthik Sankaranarayanan, Kevin Skadron, and Mircea R. Stan. HotSpot: A compact thermal modeling methodology for early-stage VLSI design. *IEEE Transactions on Very Large Scale Integration Systems*, 14(5):501–513, 2006.

[141] Yoon Jo Kim, Yogendra K. Joshi, Andrei G. Fedorov, Young-Joon Lee, and Sung-Kyu Lim. Thermal characterization of interlayer microfluidic cooling of three-dimensional integrated circuits with nonuniform heat flux. *Journal of Heat Transfer*, 132(4):041009, 2010.

[142] Hitoshi Mizunuma, Yi-Chang Lu, and Chia-Lin Yang. Thermal modeling and analysis for 3-D ICs with integrated microchannel cooling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(9):1293–1306, 2011.

[143] Arvind Sridhar, Alessandro Vincenzi, David Atienza, and Thomas Brunschwiler. 3D-ICE: A compact thermal model for early-stage design of liquid-cooled ICs. *IEEE Transactions on Computers*, 63(10):2576–2589, 2014.

[144] T. L. Bergman et al. *Fundamentals of Heat and Mass Transfer*. John Wiley & Sons, 2011.

[145] R. K. Shah et al. *Laminar Flow Forced Convection in Ducts*. Academic Press, 1978.

[146] Eigen. `http://www.eigen.tuxfamily.org/`.

Publication List

[1] Gengjie Chen, Chak-Wa Pui, Haocheng Li and Evangeline F. Y. Young, "Dr. CU: Detailed Routing by Sparse Grid Graph and Minimum-Area-Captured Path Search", accepted by *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[2] Gengjie Chen and Evangeline F. Y. Young, "SALT: Provably Good Routing Topology by a Novel Steiner Shallow-Light Tree Algorithm", accepted by *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[3] Gengjie Chen, Chak-Wa Pui, Wing-Kai Chow, Ka-Chun Lam, Jian Kuang, Evangeline F. Y. Young and Bei Yu, "RippleFPGA: Routability-Driven Simultaneous Packing and Placement for Modern FPGAs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 10, pp. 2022—2035, 2018.

[4] Haocheng Li, Gengjie Chen, Bentian Jiang, Jingsong Chen and Evangeline F. Y. Young, "Dr. CU 2.0: A Scalable Detailed Routing Framework with Correct-by-Construction Design Rule Satisfaction", *IEEE/ACM International Conference on Computer-Aided Design*, Nov, 2019.

[5] Jingsong Chen, Jinwei Liu, Gengjie Chen, Dan Zheng and Evangeline F. Y. Young, "MARCH: Maze Routing Under a Concurrent and Hierarchical Scheme for Buses", *ACM/IEEE Design Automation Conference*, June, 2019.

[6] Bentian Jiang, Xiaopeng Zhang, Ran Chen, Gengjie Chen, Peishan Tu, Wei Li, Evangeline F. Y. Young and Bei Yu, "FIT: Fill Insertion Considering Timing", *ACM/IEEE Design Automation Conference*, June, 2019.

[7] Gengjie Chen and Evangeline F. Y. Young, "Dim Sum: Light Clock Tree by Small Diameter Sum", *IEEE/ACM Design, Automation and Test in Europe*, Mar, 2019.

[8] Gengjie Chen, Chak-Wa Pui, Haocheng Li, Jingsong Chen, Bentian Jiang and Evangeline F. Y. Young, "Detailed Routing by Sparse Grid Graph and Minimum-Area-Captured Path Search", *IEEE/ACM Asia and South Pacific Design Automation Conference*, Jan, 2019.

[9] Haocheng Li, Wing-Kai Chow, Gengjie Chen, Evangeline F. Y. Young and Bei Yu, "Routability-Driven and Fence-Aware Legalization for Mixed-Cell-Height Circuits", *ACM/IEEE Design Automation Conference*, June, 2018.

[10] Chak-Wa Pui, Peishan Tu, Haocheng Li, Gengjie Chen and Evangeline F. Y. Young, "A Two-Step Search Engine For Large Scale Boolean Matching Under NP3 Equivalence", *IEEE/ACM Asia and South Pacific Design Automation Conference*, Jan, 2018.

[11] Gengjie Chen, Peishan Tu and Evangeline F. Y. Young, "SALT: Provably Good Routing Topology by a Novel Steiner Shallow-Light Tree Algorithm", *IEEE/ACM International Conference on Computer-Aided Design*, Nov, 2017. (**Best Paper Award**)

[12] Chak-Wa Pui, Gengjie Chen, Yuzhe Ma, Evangeline F. Y. Young and Bei Yu, "Clock-Aware UltraScale FPGA Placement with Machine Learning Routability Prediction", *IEEE/ACM International Conference on Computer-Aided Design*, Nov, 2017.

[13] Gengjie Chen, Jian Kuang, Zhiliang Zeng, Hang Zhang, Evangeline F. Y. Young and Bei Yu, "Minimizing Thermal Gradient and Pumping Power in 3D IC Liquid Cooling Network Design", *IEEE/ACM Design Automation Conference*, Jun, 2017.

[14] Chak-Wa Pui, Gengjie Chen, Wing-Kai Chow, Jian Kuang, Ka-Chun Lam, Peishan Tu, Hang Zhang, Evangeline F. Y. Young and Bei Yu, "RippleFPGA: A Routability-Driven Placement for Large-Scale Heterogeneous FPGAs", *IEEE/ACM International Conference on Computer-Aided Design*, Nov, 2016.