國立臺灣科技大學

資 訊 工 程 系

碩士學位論文

學號：M10815047

以拓撲逃脫繞線應用於封裝基板設計規劃

Topological Escape Routing for Package Substrate Design Planning

研 究 生：謝岳璋

指導教授：劉一宇 博士

中華民國 111 年 4 月

M10815047

碩士學位論文指導教授推薦書

Master's Thesis Recommendation Form

系所：
Department/Graduate Institute

資訊工程系
Department of Computer Science and Information Engineering

姓名：
Name

謝岳璋
SHIEH YUEH-CHANG

論文題目：
(Thesis Title)

以拓樸逃脫繞線應用於封裝基板設計規劃
Topological Escape Routing for Package Substrate Design Planning

係由本人指導撰述，同意提付審查。

This is to certify that the thesis submitted by the student named above, has been written under my supervision. I hereby approve this thesis to be applied for examination.

指導教授簽章：
Advisor's Signature

劉一宇

共同指導教授簽章（如有）：
Co-advisor's Signature (if any)

日期：
Date(yyyy/mm/dd)

111 / 4 / 8

M10815047

碩士學位考試委員審定書

Qualification Form by Master's Degree Examination Committee

系所： Department/Graduate Institute	資訊工程系 Department of Computer Science and Information Engineering
姓名： Name	謝岳璋 SHIEH YUEH-CHANG
論文題目： (Thesis Title)	以拓樸逃脫繞線應用於封裝基板設計規劃 Topological Escape Routing for Package Substrate Design Planning

經本委員會審定通過，特此證明。

This is to certify that the thesis submitted by the student named above, is qualified and approved by the Examination Committee.

學位考試委員會

Degree Examination Committee

委員簽章：

Member's Signatures

指導教授簽章：

Advisor's Signature

劉一宇

共同指導教授簽章（如有）：

Co-advisor's Signature (if any)

系所（學程）主任（所長）簽章：

Department/Study Program/Graduate Institute Chair's Signature

日期：

Date(yyyy/mm/dd)

111 / 4 / 8

以拓撲逃脫繞線應用於封裝基板設計規劃

研究生：謝岳璋　　指導教授：劉一宇 博士

國立臺灣科技大學資訊工程學研究所

摘要

廣義逃脫繞線問題在基板設計規劃中是很棘手的問題。使用幾何角度的繞線方法已經研究及發展了多年，然而繞線成率依然是無法突破的瓶頸。廣義逃脫繞線問題，例如：由導孔繞線到手指、由凸塊繞線到邊界、由手指繞線到邊界，即便在現代封裝設計中，依然由人力完成。本論文以拓撲的角度解決廣義逃脫繞線問題。相較於使用幾何角度的繞線方法，使用拓撲角度的繞線方法在解決繞線問題的初期專注於繞線的拓撲關係，這有助於避免繞線初期階段決定過多細節，降低計算複雜度。

首先，將環形拓墣表示法的抽象介面建立地完整，我們的研究指出，若要發展基於環形拓墣表示法的繞線演算法，關鍵在於要加入更多資訊到抽象介面，才能得到更好的繞線結果。原論文提出的**基本環形拓撲繞線演算法**理論上保證拓墣百分百的繞線成功率，但實際應用上產生的結果會有過度纏繞的問題，以至於無法在下個階段轉成幾何繞線。為了解決這個問題，我們提出將嵌入樹林的資訊納入環形拓墣表示法的抽象介面，並使用**三鏈結**表來說明我們提出的新的環形拓墣表示法的抽象介面以及三鏈結資料結構。接著，我們提出**拓撲跳脫繞線演算法**，搭配**啟發式目標偏差值的最小成本樹林**，此搭配可以大幅減少纏繞問題，且可以在幾秒內就得到結果。依據我們的研究結果，我們展示了廣義拓撲繞線問題可在繞線設計規劃初期，以拓撲繞線的角度有效率地解決。

關鍵詞：封裝、基板繞線、拓撲繞線、跳脫繞線、環形拓撲表示法

TOPOLOGICAL ESCAPE ROUTING FOR PACKAGE SUBSTRATE DESIGN PLANNING

Yueh-chang Shieh **Advisor: Yi-Yu Liu**

Computer Science and Information Engineering
National Taiwan University of Science and Technology

Abstract

General escape routing problem is a challenging problem in substrate design. Although geometrical routing methodology has been researched and developed for years, the routing completion rate has been a bottleneck. In modern package design, the escape routing problems, such as wire connections from vias to finger rows, from bump-balls to peripheral boundary, and from fingers to escape boundary, can only be solved by a substantial amount of manual effort. In this work, we tackle the problem from a topological routing perspective. Compared to geometrical routings, topological routings focus on the topology of paths, which greatly helps to avoid making unnecessarily detailed decisions in early design planning stage and to reduce computation complexity. Firstly, we indicate that the circular frame interface should include more information to develop a circular-frame-based algorithm that generates a quality solution. The basic circular frame routing algorithm theoretically guarantees 100% completion rate. However, the solution may suffer from path tangling problem and is unrealistic to transform into a homotopic geometrical sketch. To tackle this issue, we develop the circular frame interface including the information of the forest style of the embedding frame, and use the **triple-lists table** which is a diagram illustrating the circular frame interface and its underly-

ing data structure to represent a topology class. We apply the **topology escape routing algorithm** on the **minimum spanning forest of heuristic target net offset**, which generates a solution that effectively alleviates the path tangling problem in seconds. Based on the experimental results, we demonstrate that the escape routing problem could be efficiently and effectively resolved via topological routing perspective in early design planning stage.

Keywords: Circular Frame, Escape Routing, Packaging, Substrate Routing, Topology Routing

TABLE OF CONTENTS

ABSTRACT vi

List of Tables x

List of Figures xii

CHAPTER 1. Introduction 1
1.1 Package substrate design 1
1.2 Geometrical and Topological Routing 1

CHAPTER 2. Preliminaries 3
2.1 Terminology 3
2.2 Representation of Topology Class 4
2.2.1 Triangulation Crossing Sketch 4
2.2.2 Rubber Band Sketch 5
2.2.3 Circular Frame 5
2.3 Problem Formulation 8

CHAPTER 3. Triple-lists and Circular Frame Interface 9
3.1 Introduce To Triple-lists Table 10
3.2 Make-slice and Free-slice Operation 15
3.3 Implementation The Basic Circular Frame Routing Algorithm Using Triple List Data Structure 25

CHAPTER 4. Topology Escape Routing 28
4.1 Topology Escape Routing Algorithm 29
4.2 Heuristic Method of Finding a Forest for Circular Frame abstraction 49

CHAPTER 5. Experimental Results 60
5.1 Experimental Setup 60
5.2 Wire Length and Run Time Comparisons 60
5.3 Case Inspections 63

CHAPTER 6. Conclusion **83**

Bibliography **84**

List of Tables

3.1 **The Mapping List.** 14

3.2 **The Slice List.** 16

3.3 **The Frame List.** 17

3.4 **The Content of the Mapping List after the Operation of** $make\_slice(u_{16}, u_9)$**.** 22

3.5 **The Content of the Slice List after the Operation of** $make\_slice(u_{16}, u_9)$**.** 23

3.6 **The Content of the Frame List after the Operation of** $make\_slice(u_{16}, u_9)$**.** 24

5.1 Industrial design specifications 61

5.2 Industrial 1 Wire Length Experimental Results of All Forest Style and Routing Algorithm Combinations 64

5.3 Industrial 1 Run Time (microsecond) Experimental Results of All Forest Style and Routing Algorithm Combinations 64

5.4 Industrial 2 Wire Length Experimental Results of All Forest Style and Routing Algorithm Combinations 65

5.5 Industrial 2 Run Time (microsecond) Experimental Results of All Forest Style and Routing Algorithm Combinations 66

5.6 Industrial 3 Wire Length Experimental Results of All Forest Style and Routing Algorithm Combinations 66

5.7 Industrial 3 Run Time (microsecond) Experimental Results of All Forest Style and Routing Algorithm Combinations 67

5.8 Industrial 4 Wire Length Experimental Results of All Forest Style and Routing Algorithm Combinations 68

5.9 Industrial 4 Run Time (microsecond) Experimental Results of All Forest Style and Routing Algorithm Combinations 68

5.10 Industrial 5 Wire Length Experimental Results of All Forest Style and Routing Algorithm Combinations 69

5.11 Industrial 5 Run Time (microsecond) Experimental Results of All Forest Style and Routing Algorithm Combinations 70
5.12 Industrial 6 Wire Length Experimental Results of All Forest Style and Routing Algorithm Combinations 70
5.13 Industrial 6 Run Time (microsecond) Experimental Results of All Forest Style and Routing Algorithm Combinations 71

List of Figures

2.1 **Illustration of the Circular Frame Representation.** (a) A boundary and terminals are given. (b) A random embedding frame is built from the given terminals. (c) The corresponding circular frame is transformed from the embedding frame and there is initially one slice annotated as σ_1 in the frame. (d) Two topology paths are annotated in the Figure. (e) According to the constructed embedding frame, the blue topology path passes through the edge (b_4, t_4), and the purple topology path passes through the edges (b_4, t_5) and (b_4, t_5). And the original slice σ_1 is divided by the two topology paths into 6 slices annotated as σ_1 to σ_6. (f) The two topology paths are mapped to their corresponding straight line in the circular frame annotated in the same color respectively. 7

3.1 **The Triple-lists Table.** 11

3.2 **The Representation of the Triple-lists Table after the Operation of $u_{16} = C.make\_slice(u_{21}, u_{18})$.** 18

3.3 **The Representation of the Triple-lists Table after the Operation of $u_9 = make\_slice(u_{16}, u_9)$.** 20

3.4 **The complete topology sketch represented by the triple-list table.** 21

4.1 **Topology Escape Routing Flow.** The flow shows how we generate a geometrical sketch from a given boundary, terminals, and nets. The flow is generic and based on the circular frame interface. We can apply a different policy to acquiring a spanning forest and the routing process. 30

4.2 **Topology Escape Routing Flow.** The flow shows how we generate a geometrical sketch from a given boundary, terminals, and nets. The flow is generic and based on the circular frame interface. We can apply a different policy to acquiring a spanning forest and the routing process. 31

4.3 **The Triple-lists Table of Example One (a).** 35

4.4 **The Triple-lists Table of Example One (b).** 36
4.5 **The Triple-lists Table of Example One (c).** 37
4.6 **The Triple-lists Table of Example One (d).** 38
4.7 **The Topology Sketch of Example Two.** Terminals associated with the same net are annotated in the same color. (a) The solution is acquired by the simple-make-slice-issue tangling avoidance algorithm. (b) The solution is acquired by the root-issue tangling avoidance algorithm. 39
4.8 **The Triple-lists Table of Example Two (a).** 41
4.9 **The Triple-lists Table of Example Two (b).** 42
4.10 **The Topology Sketch of Example Three.** Terminals associated with the same net are annotated in the same color. (a) The solution is acquired by the root-issue tangling avoidance algorithm. (b) The solution is acquired by the detour-aware tangling avoidance algorithm. 43
4.11 **The Triple-lists Table of Example Three (a).** 43
4.12 **The Triple-lists Table of Example Three (b).** 46
4.13 **The Topology Sketch of Example Four.** Terminals associated with the same net are annotated in the same color. (a) The solution is acquired by the detour-aware tangling avoidance algorithm. (b) The solution is acquired by the topology escape routing algorithm. . . 47
4.14 **The Triple-lists Table of Example Four (a).** 47
4.15 **The Triple-lists Table of Example Four (b).** 49
4.16 **The Triple-lists Table of Example Four (c).** 50
4.17 **The Triple-lists Table of Example Four (d).** 51
4.18 **The Illustration of the Minimum Spanning Forest of A* heuristic Euclidean Distance.** The heuristic edge weight $W_2(v_4, v_{29}) = E(v_4, v_{29}) + H(v_4, v_{29}) = \alpha + \beta$, where α is the Euclidean distance between vertex v_3 and v_4, and β is the Euclidean distance between vertex v_4 and v_{16}. 53

4.19 **The Illustration of the Minimum Level Spanning Forest.** The comparison between a random spanning forest and a minimum level spanning forest. The edges of the forests are annotated in red. (a) All the terminals in the boundary belong to the tree with the root v_3. (b) Terminals are evenly distributed to different roots to acquire a spanning forest with an overall minimum level. 54

4.20 **The Illustration of the Minimum Spanning Forest of Heuristic Target Net Offset.** (a) The weights of each edge are annotated respectively. (b) The corresponding minimum spanning forest is annotated in red. 56

4.21 **The Illustration of the Minimum Spanning Forest of Target Net Offset from the Root.** (a) The weights of each edge are computed dynamically and annotated respectively. (b) The first ten edges chosen for the spanning forest are annotated in red. As an edge is chosen, the weights of its adjacent edges are computed. 58

4.22 **The Illustration of the Minimum Spanning Forest of Target Net Offset from the Root.** (a) The edge $\{v_{18}, v_{20}\}$, whose edge weight is 1, has the current minimum weight. As a result, it is the eleventh chosen edge. (b) At last, the edge $\{v_{20}, v_{24}\}$ and $\{v_{20}, v_{24}\}$ are chosen. The vertex v_24 belongs to the tree of root v_4 instead of the tree of root v_1. 59

5.1 **The Illustration of Fingers Projected to the Margin of the Substrate.** 61

5.2 Industrial 1 Wire Length Experimental Result Summary 64

5.3 Industrial 1 Geometrical Visualization Results of All Forest Style and Routing Algorithm Combinations 65

5.4 Industrial 2 Wire Length Experimental Result Summary 66

5.5 Industrial 2 Geometrical Visualization Results of All Forest Style and Routing Algorithm Combinations 67

5.6 Industrial 3 Wire Length Experimental Result Summary 68

5.7 Industrial 3 Geometrical Visualization Results of All Forest Style and Routing Algorithm Combinations 69

5.8 Industrial 4 Wire Length Experimental Result Summary 70

5.9 Industrial 4 Geometrical Visualization Results of All Forest Style and Routing Algorithm Combinations 71

5.10 Industrial 5 Wire Length Experimental Result Summary 72

5.11 Industrial 5 Geometrical Visualization Results of All Forest Style and Routing Algorithm Combinations 73

5.12 Industrial 6 Wire Length Experimental Result Summary 74

5.13 Industrial 6 Geometrical Visualization Results of All Forest Style and Routing Algorithm Combinations 75

5.14 **The Industrial 1 Partial Geometrical Visualization Using the Minimum Spanning Forest of Heuristic Target Net Offset and the Basic Circular Frame Routing Algorithm.** 76

5.15 **The Industrial 1 Partial Geometrical Visualization Using the Minimum Spanning Forest of Heuristic Target Net Offset and the Topology Escape Routing Algorithm Version One.** . . 77

5.16 **The Industrial 1 Partial Geometrical Visualization Using the Minimum Spanning Forest of Heuristic Target Net Offset and the Topology Escape Routing Algorithm Version Two.** . 78

5.17 **The Industrial 1 Partial Geometrical Visualization Using the Minimum Spanning Forest of Heuristic Target Net Offset and the Topology Escape Routing Algorithm Version Three.** 79

5.18 **The Industrial 1 Complete Geometrical Visualization Using the Minimum Spanning Forest of Heuristic Target Net Offset and the Topology Escape Routing Algorithm Version Four.** . 80

5.19 **The Industrial 2 Complete Geometrical Visualization Using the Minimum Spanning Forest of Heuristic Target Net Offset and the Topology Escape Routing Algorithm Version Four.** . 81

5.20 **The Industrial 3 Complete Geometrical Visualization Using the Minimum Spanning Forest of Heuristic Target Net Offset and the Topology Escape Routing Algorithm Version Four.** . 82

CHAPTER 1

Introduction

1.1 Package substrate design

The integrated circuit (IC) packaging is the last stage in the production of semiconductor devices. As the CMOS scaling slows down, IC packaging industry has gradually grown its vital importance in the development of next generation low-power and cost-effective design integration. The IC package design practice has evolved over the years. The advancement in packaging technology and high-density integration arouse tremendous demands on substrate design automation. In particular, the substrate routing problem, which is considered more challenging than the routing problem within a chip, owing to complex design rules and less routing resource. Additionally, the allowed orientations for routing traces are flexible , which incurs solution space boosting as well as design complexity explosion.

1.2 Geometrical and Topological Routing

The methodology of current existing routing tools falls into the categories of the geometrical router and topological router. Geometric routers determine the exact geometrical features of wire connection; while topological routers determine the topology of the path instead of the exact features. Geometry routers are rather mature and well-researched. However, modern geometry routers have encountered

the bottleneck of reaching 100% routability. One of the main failing reasons is the puzzle problem, which argues that the optimal solution is unobtainable by any sequential routing strategy owing to the clearance problem, and that geometrical routers make too detailed decisions in the early stage even with the traditional two-stage, global and detailed routing, approach [7]. Researches about topological routing are few in comparison to that of geometrical routing. However, topology routers have several advantages over geometrical routers. For examples, topological routing occupies fewer computational resources, consumes fewer computational runtime, and is immune to the clearance problem. Therefore, topological routing is promising to break through the predicament and capable of providing a global guide to the detailed routing for geometrical routers.

Although the geometrical representation is intuitive, it is hard to argue the best topological representation. As far as we are aware, current topological representations are classified into Triangulation Crossing Sketch (TCS), Rubber-band Sketch (RBS), and the newly proposed Circular Frame [1–3, 5, 7, 8]. Triangulation Crossing Sketch is introduced to compare with Rubber-band Sketch [8]. SURF, a rubber-band-based router, is one of the few advanced topological routers, which is well known for its capability of transforming topological routing solution to a homotopic geometrical routing solution [3–6]. However, the rubber-band representation for topological routing paths is complicated and hard to develop algorithms based on the representation. There is still room to improve its global routing methodology. The innovative circular frame representation is simple to manipulate and develop algorithms based on the representation. The basic circular frame is proposed as a demonstration and is guaranteed to have 100% routability topologically [1, 2]. Nevertheless, despite of the simple representation and the theoretical routability, we find that the solution quality acquired by the algorithm may not be acceptable.

CHAPTER 2

Preliminaries

In this chapter, we introduce the terms commonly used in the literature about topology routing and briefly present an overview of existing topology representations.

2.1 Terminology

A **sketch** $S = (B, T, P)$ represents a routing solution of a given net N. T is a set of **terminals** that are 2-dimensional points within the interior of **boundary** B, a simple polygon specified by a sequence of points. N is a set of nets. For all net $n \in N$, $n = \{p | p \in B \cup T\}$ is a set of terminals in B and T, representing the interconnection between terminals. We refer to a **2-pin net** as a set with exactly two points, and a **multi-pin net** as a set containing more than two points. P is a set of **paths**, which are categorized into geometrical and topological. We refer to a sketch as a **geometrical sketch** if the paths are geometrical and a sketch as a **topological sketch** if the paths are topological.

A **geometrical sketch** is rather simple in terms of its representation. A primitive sequence of terminals served as a line string to represent a geometrical path is sufficient.

On the other hand, a **topological sketch** is well defined mathematically.

A topological path is defined as a parametric and continuous line on a plane $P : [0...1] \rightarrow R^2$, such that $P(0)$ and $P(1)$ are the starting terminal and ending terminal, respectively; however it is hard to argue the most efficient computational representation.

A **topology class** is a collection of topological sketches that have the same topology. To have an intuitive grasp of topology class, we describe a topological sketch as several elastic treads on a paper. Terminals are specific points where the starting or ending point of a thread is pinned on. Any threads on the paper, i.e. topological path, are allowed to stretch and move to anywhere while their starting and ending terminals are still pinned on the paper. And all threads have to contact the paper at all times. We called that the resulting new topological sketch is **homotopic** to the original sketch, and the two sketches belong to the same **topology class**. In terms of topology routing, we are more interested in the representation of topology class instead of topology sketch. In the following section, we briefly explain all topology class representation as far as we are aware.

2.2 Representation of Topology Class

2.2.1 Triangulation Crossing Sketch

Triangulation Crossing Sketch (TCS) indirectly specifies a topology class by representing a homotopic subset of topology sketches. The methodology requires an arbitrary triangulation built from terminals within the boundary before a topology path can be specified in terms of a sequence of triangles. Each of the adjacent pairs of triangles in the sequence must be adjacent to each other in the triangulation as well and thus has an exact one terminal that differs. Hence a sequence of triangles can be efficiently represented by a sequence of terminals. However, the representations of a topology class in terms of TCS are not unique. As a consequence,

Triangulation Normalized Crossing Sketch (TCNS) is proposed to use a minimal number of triangles to specify each topology path of a topology sketch.

2.2.2 Rubber Band Sketch

For all homotopic topological paths, there exists a topological path that has a conceptually minimal wire length. Using metaphor, a topological path is a thread made of rubber-band material and thus the tensest form of itself has the minimal wire length concerning upholding the same topology of its class. Therefore, a rubber-band sketch (RBS) represents a topology class with a corresponding triangulation instead of an arbitrary one [3]. Each of the topological paths with the minimum wire length representing its class is specified by a sequence of edges in the triangulation. Moving a vertex or changing the topology entails dynamic updating of the triangulation. However, RBS is considered to have more geometrical information than TCNS and has the advantage of facilitating progressive refinement of the sketch and the geometrical transformation process [4].

2.2.3 Circular Frame

The circular frame is a newly proposed topological representation inspired by the **polygonal schema** introduced in **Algebraic topology**, a branch in the science of mathematics [1]. The greatest strength of the circular frame is to represent a topology class directly.

We explain the circular frame representation with an example shown in Figure 2.1. Given a boundary B and a set of terminals T, the first step of representing a topology sketch is to set up **the embedding frame schema** [2]. As illustrated in Figure 2.1 (b). We need to make several straight cut lines on the paper using a scissor. Each cut line starts from the boundary or a dot and ends at a dot. All dots

in T have to be reached by at least a cut line. And the paper must remain in one piece.

The second step is to construct the **circular frame schema** [1]. As shown in Figure 2.1 (c), as walking along the edge of the scissored paper, corresponding vertices are sequentially added to the circular frame at the event of encountering a boundary point, a terminal, or each side of the cut line. The enclosing area annotated as σ is called **slice** [2].

It has been proven that there exists a $1-1$ mapping relation between any topology class and its non-intersecting chords starting from and ending at a vertex on the circular frame.

In Figure 2.1 (d) (f), there is a topological sketch with two topological paths. After the construction of the embedding frame schema, each of the two topological paths is mapped to a corresponding sequence of straight lines. Note that a slice is divided into two while a path passes through cut lines in the embedding frame. As annotated in Figure 2.1 (e), (f), there are five slices in both the embedding frame and in the circular frame. The order of multiple paths entering into a cut line must be consistent with the order of existing from the other side of the cut line. Take Figure 2.1 (f) as an example. The order of two topological paths entering vertex r'_4 on the circular frame must be consistent with the order of existing from vertex r_4. The order is called **slice ordering** [2].

Since there exists a $1-1$ mapping relation between the circular frame and an embedding frame. The **basic circular frame routing algorithm** is proposed to find a routing solution in the circular frame schema and to transform the solution back to a corresponding embedding frame. The **basic circular frame routing algorithm** guarantees a 100% routed topology class [2].

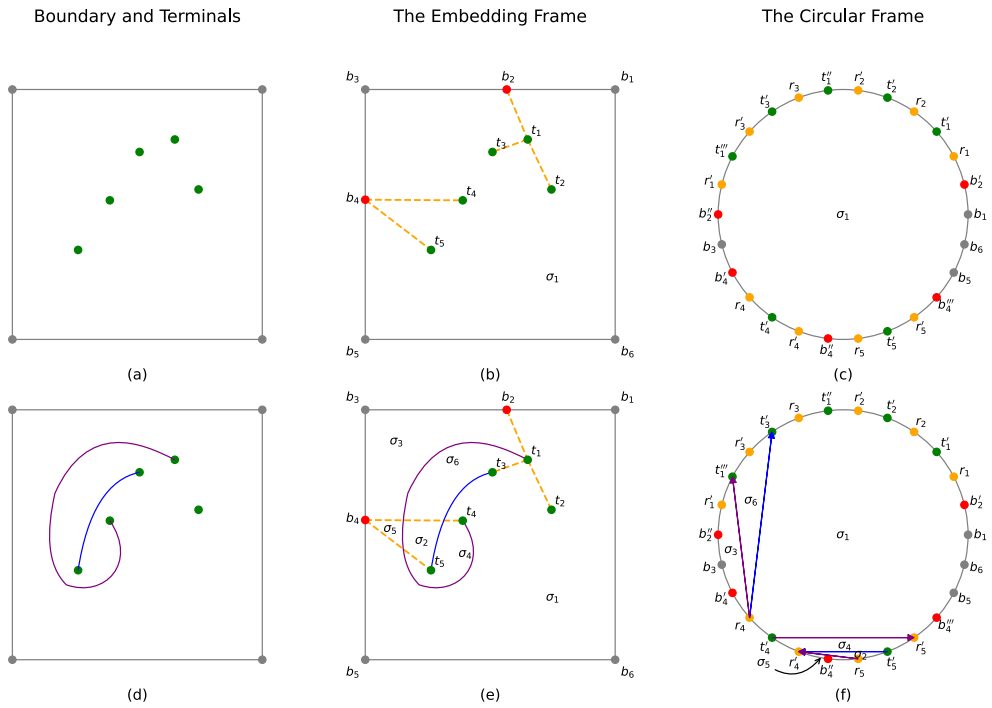

Figure 2.1: **Illustration of the Circular Frame Representation.** (a) A boundary and terminals are given. (b) A random embedding frame is built from the given terminals. (c) The corresponding circular frame is transformed from the embedding frame and there is initially one slice annotated as σ_1 in the frame. (d) Two topology paths are annotated in the Figure. (e) According to the constructed embedding frame, the blue topology path passes through the edge (b_4, t_4), and the purple topology path passes through the edges (b_4, t_5) and (b_4, t_5). And the original slice σ_1 is divided by the two topology paths into 6 slices annotated as σ_1 to σ_6. (f) The two topology paths are mapped to their corresponding straight line in the circular frame annotated in the same color respectively.

2.3 Problem Formulation

The formulation for a topology escape routing problem is as follows:

Input:

1. A boundary $B =< b_1, b_2, ..., b_n >$ which is a sequence of terminals specifying a simple polygon.
2. A set of terminals $T = \{t_1, t_2, ..., t_m\}$, and for all terminal $t \in T$ are located in the area enclosed by boundary B.
3. A set of 2-pin net N. And for all nets $n \in N$ such that $n = \{b, t\}$, where $b \in B$ and $t \in T$.

And the output should be a topology sketch (B, T, P), where P is a set of topology paths for each net.

CHAPTER 3

Triple-lists and Circular Frame Interface

We consider the circular frame as an interface for topology representation that allows algorithms to be written independently of any particular data structure. However, there are still a few that needed to be established:

1. The basic circular routing algorithm [2] has merely proven that a 100% routing completion rate is guaranteed from any topology routing problem. The topology class acquired by the algorithm is usually heavily tangled and thus impractical to be transformed into a homotopic geometrical sketch. Our research shows that the circular frame interface has to enrich the information for an algorithm to be able to acquire a solution of better quality.

2. Since the circular frame interface is still opaque at the moment, we need a concrete collection of concepts that specifies what information the interface could provide and how efficient a topology class can be inspected and manipulated under the interface. By concepts, we mean a set of operations as well as semantic requirements, and time and space complexity guarantees.

3. In addition, using the circular frame schema as an illustration of a topology class suffers from a scalability problem. It is difficult to illustrate two topological paths using the circular frame schema since chords are too compact

for visual observations. It is even harder to illustrate a case of industrial scalability.

In this chapter, we intend to help establish the circular frame interface. Furthermore, we provide a model, called **triple-lists**, as an implementation of circular frame abstraction. We analyze the traversal concept and efficiency of operations provided by this model. In addition, we propose a diagram, called the **triple-lists diagram**, to replace the circular frame schema. The diagram implies the underlying triple-lists data structure. And it can show topology paths with a scale as large as it can be. The diagram only consists of ANSII characters, thus can be stored in a text file and be easily inspected with a text editor.

3.1 Introduce To Triple-lists Table

The circular frame abstraction represents a topology class based on the embedding frame which we consider as a refinement of graph concept in the graph theory. The information of forest should be provided by the interface and be made used by an algorithm, which is an essential key to acquire a solution of quality.

We propose to illustrate the circular frame interface that provides the information of the embedding frame via the **triple-lists table**. Take the embedding frame $F = (V, E)$ shown in Figure 2.1 (b) as an example, where $V =< b_1, ..., b_6, t_1, ..., t_5 >$. Figure 3.1 shows the corresponding triple-list table of F.

The triple-list table consists of three columns. As an example shown in Figure 3.1, note that there are two rows of thirteen equal signs on the top and at the bottom; we refer to the first column as the space between the first 4 equal signs, the second column as the space between the sixth to the eleventh equal signs, and the third column as the space between the thirteen equal sign.

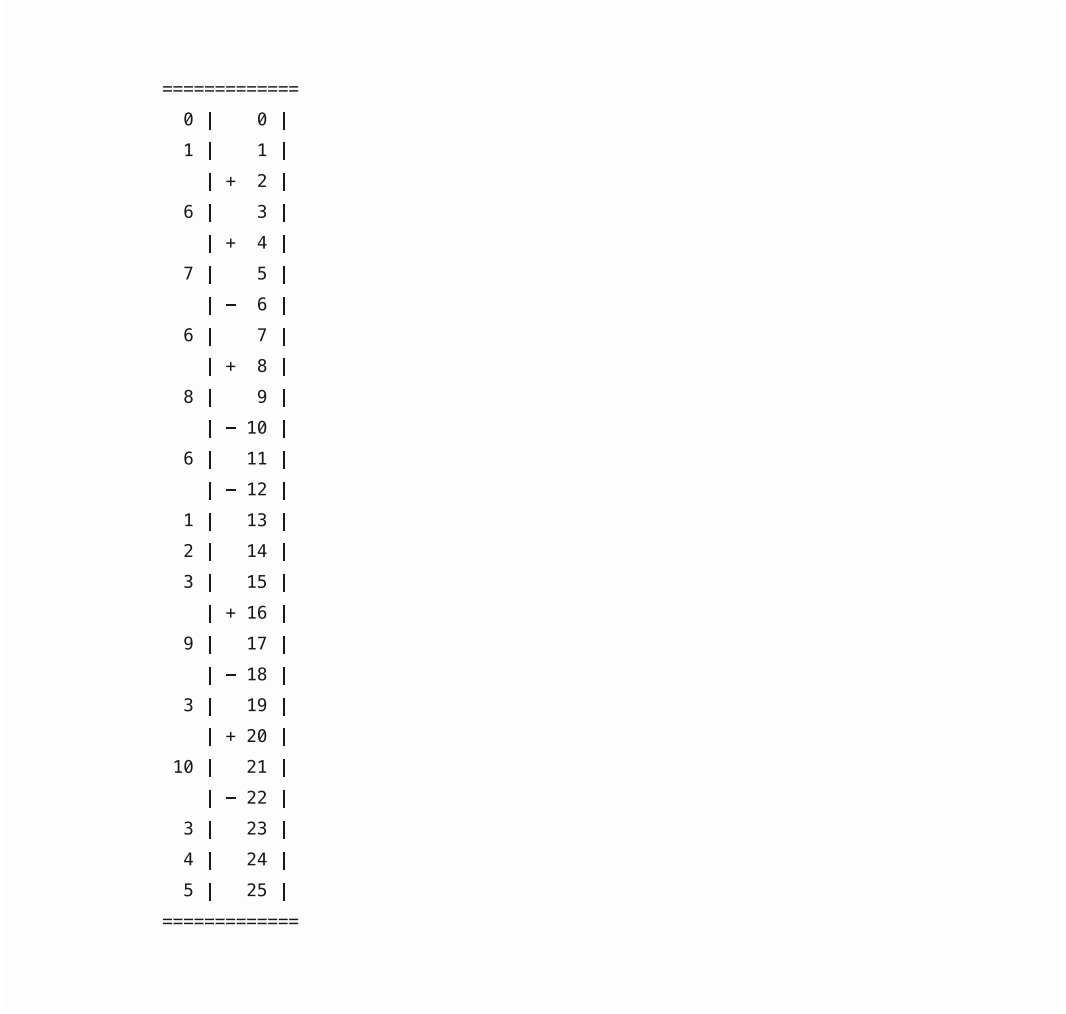

Figure 3.1: **The Triple-lists Table.**

Firstly, we introduce two terms. We refer to the vertices in the embedding frame schema as **graph vertices** and the vertices in the circular frame schema as **topology vertices**. In the following content in this literature, we refer to a graph vertex with an index of i as v_i, and a topology vertex with an index of j as u_j.

Each of the numbers in the first column specifies the index of the graph vertices in accord with their order in V. The sequence of graph vertex indexes in the column specifies a depth-first-search order of the embedding frame, and a blank row in the column denotes an edge between the two graph vertices. For example, b_1 has an index of 0, b_6 has an index of 5, t_1 has an index of 6, and t_5 has an index of 10. And the depth-first-search order of the embedding frame is $< b_1, b_2, t_1, t_2, t_1, t_3, t_1, b_2, b_3, b_4, t_4, b_4, t_5, b_4, b_5, b_6 >$. The second column shows the corresponding index of topology vertices. Note the plus signs and the minus signs. A topology vertex with a plus sign is said to have attribution of **POSITIVE_EDGE**, which specifies that it corresponds to the left-hand side of a graph edge. A topology vertex with a minus sign is said to have attribution of **NEGATIVE_EDGE**, which specifies that it corresponds to a right-hand side of a graph edge. And a topology vertex without any prefixed sign is said to have attribution of **VERTEX**, which specifies that it corresponds to a graph vertex. For example, the topology vertex u_2 corresponds to the left-hand side of the graph edge $\{v_1, v_6\}$, The topology vertex u_{12} corresponds to the right-hand side of the forest edge $\{v_1, v_6\}$. We call u_2 and u_{12} an **ordering pair**. Conceptually, a topology vertex has a semantic meaning of a specific part of a topology path or a location identified in a topology sketch.

The third column specifies a list of **slices**. In the circular frame interface we propose, slice concept represents a portion of topology paths starting from a topology vertex s and ending at a topology vertex t, which "slice off" a topology plane $P = \{u_1, u_2, ..., u_k\}$ from an origin topology plane, where $u_1, u_2, ..., u_k$ are

topology vertices. In addition, all topology vertices belong to exactly one slice. By topology plane P, we mean an area that allows making a new topology path from u_i to u_j, which does not cross any existing topology paths, where $u_i, u_k \in P$. Initially, the column has a width of one equal sign, which indicates that there is one slice at the start which starts from and ends at no topology vertex and consists of all the topology vertices. We refer to the slice as the **original slice**. Note that a topology path can represent using a sequence of slices.

The triple-list table also implies the underlying data structure, which consists of three closely interlinked lists: **mapping list**, **slice list**, and **frame list**.

Firstly, We define the structure, called **mapping list**, to implement the mapping relationship from graph vertices to topology vertices. Since every graph vertices of a forest have exactly one mother vertex, we define the **mapping list** as an array of lists of arrays of **topology vertices**. The index of the array of lists corresponds to the index of the graph vertices sequence. The first array of topology vertices in each of the lists corresponds to a sequence of topology vertices that is mapped from a graph vertex. And the following array of topology vertices in the lists corresponds to a pair of topology vertices that is mapped from the edge of a corresponding vertex and its mother vertex. All topology vertices in the arrays are placed in counterclockwise order.

Given the same embedding frame $F = (V, E)$ shown in Figure 2.1 (b), we illustrate a corresponding mapping list in Table 3.1. Let M be the mapping list. The '#' character prefixed to M indicates that $M[v]$ is a list, where v corresponds to an index of sequence V. We refer to the ith iterator of list type $M[v]$ as @M:i, where $v \in V$. Likewise, @C:j is referred to as the jth iterator of a list type C. For example, the graph vertex v_6 has a mother vertex v_1. Then we can see that $M[6]$ is a list consisting of 2 arrays of topology vertices. And the three topology

v	$\#M$
0	→ [(0, @C:0)]
1	→ [(1, @C:1), (13, @C:13)]
2	→ [(14, @C:14)]
3	→ [(15, @C:15), (19, @C:19), (23, @C:23)]
4	→ [(24, @C:24)]
5	→ [(25, @C:25)]
6	→ [(3, @C:3), (7, @C:7), (11, @C:11)] → [(2, @C:2), (12, @C:12)]
7	→ [(5, @C:5)] → [(4, @C:4), (6, @C:6)]
8	→ [(9, @C:9)] → [(8, @C:8), (10, @C:10)]
9	→ [(17, @C:17)] → [(16, @C:16), (18, @C:18)]
10	→ [(21, @C:21)] → [(20, @C:20), (22, @C:22)]

Table 3.1: **The Mapping List.**

vertices u_3, u_7, u_{11} in the first array corresponds to the graph vertex t_1. The two topology vertices in the second array correspond to the edge of $\{v_1, v_6\}$. The first topology vertex in the second array corresponds to the left-hand side of $\{v_1, v_6\}$. And the second topology vertex in the second array corresponds to the left-hand side of $\{v_1, v_6\}$.

A topology vertex is a tuple (i, p), where i indicates the index of the topology vertex, and p is an iterator of a list type C. We'll discuss C in the following content.

Secondly, we define **slice** as a circular list of 3-tuple (t_1, t_2, t_3), where t_1 indicates the index of a graph vertex, t_2 is an iterator of $M[t_1]$, and t_3 is the index of an array of **topology vertices** pointed by t_2. Furthermore, we define the **topology list** as a list of slices. The iterator of a topology list is called **slice descriptor**. The iterator of a slice is called the **topology descriptor**. Conceptually, we handle each slice by its slice descriptor, and we iterate each slice with its topology descriptors. Both slice descriptor and topology descriptor should be provided by the circular frame interface. Every topology descriptor s is allowed to connect to another topology descriptor t in the same slice. The connection results in a new slice starting

from s and ending at t. Note that connecting two topology descriptors in the same slice at all times ensures that there is no crossing of any topology paths.

As illustrated in Table 3.2, there is initially one slice in the slice list S, starting from and ending at no topology descriptor. And the slice consists of 26 tuples. We refer to the slice descriptor of the first slice in S as @S:0, and the topology descriptor of ith member in a slice as @T:i. For example, accessed by the third topology descriptor @T:2, the tuple (6, @M:1, 0) refers to a topology vertex placed as the first element of the second array in the list $M[6]$, which is a topology vertex v_2.

Finally, as illustrated in Table 3.3, we define a data structure **Frame list** to be a circular list of 3-tuple (c_1, c_2, c_3), where c_1 and c_3 are slice descriptors and c_2 is a topology descriptor. c_1 and c_2 specify where the corresponding slice and position the topology vertex is at. And c_3 specifies whether the topology vertex is a starting or ending terminal of a slice. We call the iterator of the frame list **the circular descriptor**. Conceptually, iterating through the circular descriptor allows us to know the next topology vertex on the frame.

3.2 Make-slice and Free-slice Operation

How to make a topology path in a circular frame interface is the primary issue. Let C be the model of the circular frame interface illustrated in Figure 3.1. To make a topology path in C, we need to connect a topology vertex s to another topology vertex t in the same slice, which slices off a new topology plane represented by a new slice; The new slice represents the portion of the topology path from s to t. The following illustrates how to make the topology path from v_{10} to v_7, i.e. t_5 to t_3 shown in Figure 2.1 (e). We call this operation make-slice.

$\#S$	:0
$\#T$	(0, @M:0, 0) (1, @M:0, 0) (6, @M:1, 0) (6, @M:0, 0) (7, @M:1, 0) (7, @M:0, 0) (7, @M:1, 1) (6, @M:0, 1) (8, @M:1, 0) (8, @M:0, 0) (8, @M:1, 1) (6, @M:0, 2) (6, @M:1, 1) (1, @M:0, 1) (2, @M:0, 0) (3, @M:0, 0) (9, @M:1, 0) (9, @M:0, 0) (9, @M:1, 1) (3, @M:0, 1) (10, @M:1, 0) (10, @M:0, 0) (10, @M:1, 1) (3, @M:0, 2) (4, @M:0, 0) (5, @M:0, 0)
start	@T:nullptr
end	@T:nullptr

Table 3.2: **The Slice List.**

$\#C$	(c_1, c_2, c_3)
	(@S:0, @T:0, @S:nullptr)
	(@S:0, @T:1, @S:nullptr)
	(@S:0, @T:2, @S:nullptr)
	(@S:0, @T:3, @S:nullptr)
	(@S:0, @T:4, @S:nullptr)
	(@S:0, @T:5, @S:nullptr)
	(@S:0, @T:6, @S:nullptr)
	(@S:0, @T:7, @S:nullptr)
	(@S:0, @T:8, @S:nullptr)
	(@S:0, @T:9, @S:nullptr)
	(@S:0, @T:10, @S:nullptr)
	(@S:0, @T:11, @S:nullptr)
	(@S:0, @T:12, @S:nullptr)
	(@S:0, @T:13, @S:nullptr)
	(@S:0, @T:14, @S:nullptr)
	(@S:0, @T:15, @S:nullptr)
	(@S:0, @T:16, @S:nullptr)
	(@S:0, @T:17, @S:nullptr)
	(@S:0, @T:18, @S:nullptr)
	(@S:0, @T:19, @S:nullptr)
	(@S:0, @T:20, @S:nullptr)
	(@S:0, @T:21, @S:nullptr)
	(@S:0, @T:22, @S:nullptr)
	(@S:0, @T:23, @S:nullptr)
	(@S:0, @T:24, @S:nullptr)
	(@S:0, @T:25, @S:nullptr)

Table 3.3: **The Frame List.**

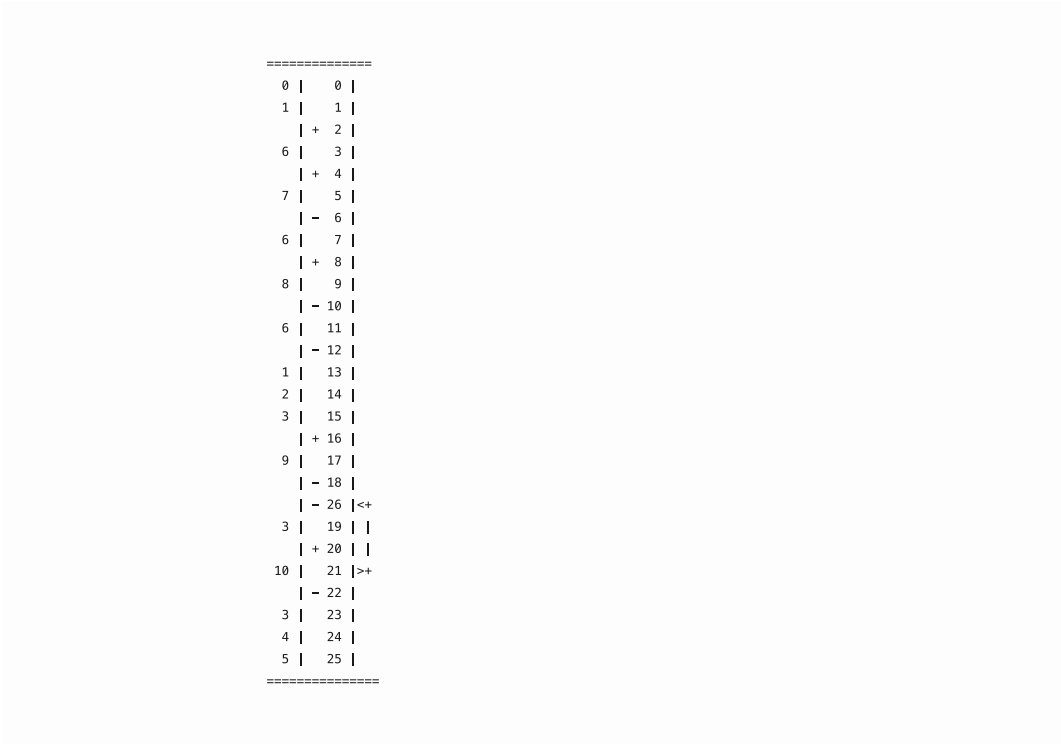

Figure 3.2: **The Representation of the Triple-lists Table after the Operation of** $u_{16} = C.make\_slice(u_{21}, u_{18})$**.**

Given any graph vertices or edges, we are able to get corresponding topology vertices in C. In figure 3.2, The graph vertex v_{10} has a corresponding topology vertices u_{21}, which is in the first slice. Making a topology path from the topology vertex u_{21} to u_{18}, which corresponds to the right-hand side of the edge (v_3, v_9), Conceptually, The operation $C.make\_slice(u_{21}, u_{18})$ return a topology vertex u_{16}, which is the ordering pair of u_{18}. We show the result in Figure 3.2.

Note that, instead of u_{18}, u_{21} is actually connected to a newly added topology vertex u_{26}, which is the next vertex of u_{18} in the topology vertices sequence. At the moment, the third column in the triple-list table has a width of two equal signs, which indicates that there are two slices after the connection from u_{21} to u_{26}.The original slice consists of $< u_{26}, u_{19}, u_{20}, u_{21} >$. The second slice starts at u_{21}, ends at

u_{26}, and consists of all the other topology vertices $< u_0, u_1, ..., u_{18}, u_{22}, u_{23}..., u_{25} >$. The purpose of this mechanism is to have u_{26} and u_{18} both correspond to the right-hand side of the edge (v_3, v_9), where u_{26} is in the first slice and u_{18} is in the second slice. In this way, edge (v_3, v_9) is allowed to be passed through by a topology path from both slices.

Then we continue to make-slice from u_{16} to u_9, which both now resides in the second slice. The result is shown in Figure 3.3. At the moment, there are three slices. The first slice consists of $< u_{26}, u_{19}, u_{20} >$. The second slice consists of $< u_9, u_{10}, ..., u_{15}, u_{27} >$. And the third slice consist of $\{u_0, ..., u_8, u_{16}, u_{17}, u_{18}, u_{21}, ..., u_{25}\}$. According to the slice ordering introduced in [1], u_{27}, the ordering pair of u_{26}, should be added as a previous topology vertex of u_{16} in the counterclockwise sequence. And the topology path from v_{10} to v_7 is represented by a sequence of slice $< s_1, s_2 >$, where s_1 is the second slice starting at u_{21} and ending at u_{26}, and $s2$ is the third slice starting at u_{27} and ending at u_9.

Furthermore, The free-slice operation should be provided to redo the make-slice operation. For example, operation $C.free\_slice(u_5)$ should return u_{16} and move the third slice back to the second slice. Operation $C.free\_slice(u_{16})$ should return u_{21} and move the second slice back to the origin slice.

At last, the example of the topology sketch shown in Figure 2.1 (e) represented by the triple list table is shown in Figure 3.4. There are six slices $< s_0, s_1, ...s_5 >$ in the topology plane. The topology path from the graph vertex t_5 to the graph vertex t_3 is specified by the slice sequence $< s_2, s_4 >$. The topology path from the graph vertex t_4 to the graph vertex t_1 is specified by the slice sequence $< s_5, s_1, s_3 >$.

Let's analyze the complexity of the operation. With the triple-lists data

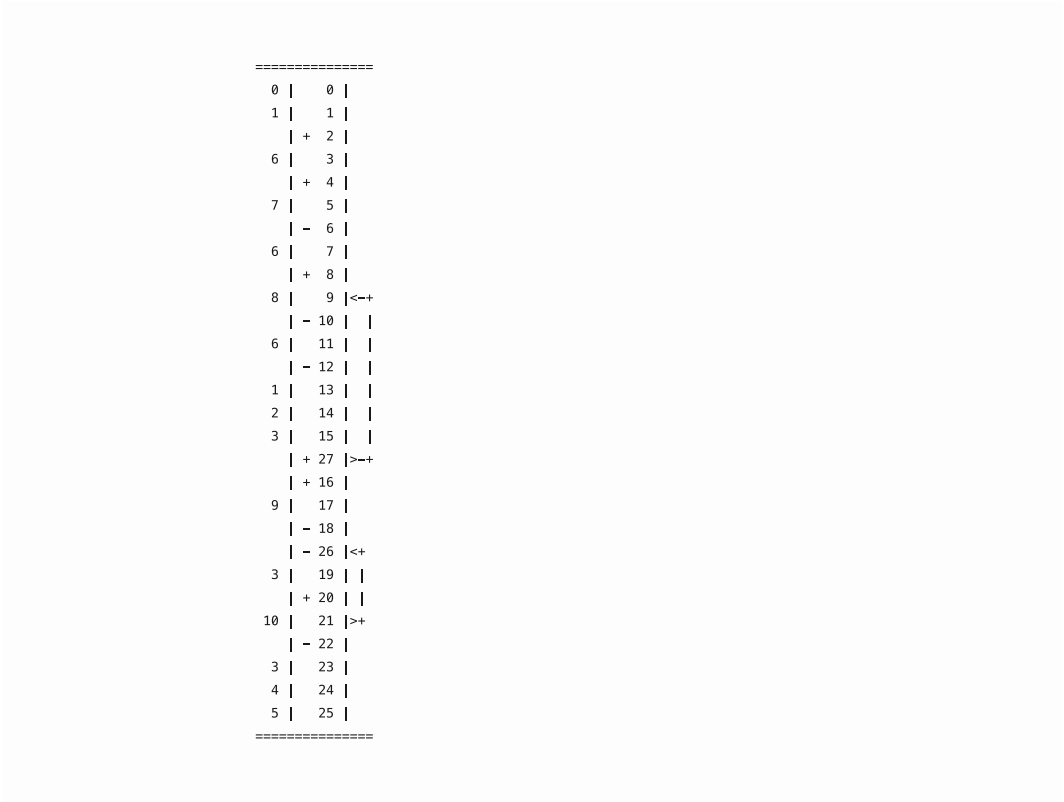

Figure 3.3: **The Representation of the Triple-lists Table after the Operation of $u_9 = make\_slice(u_{16}, u_9)$.**

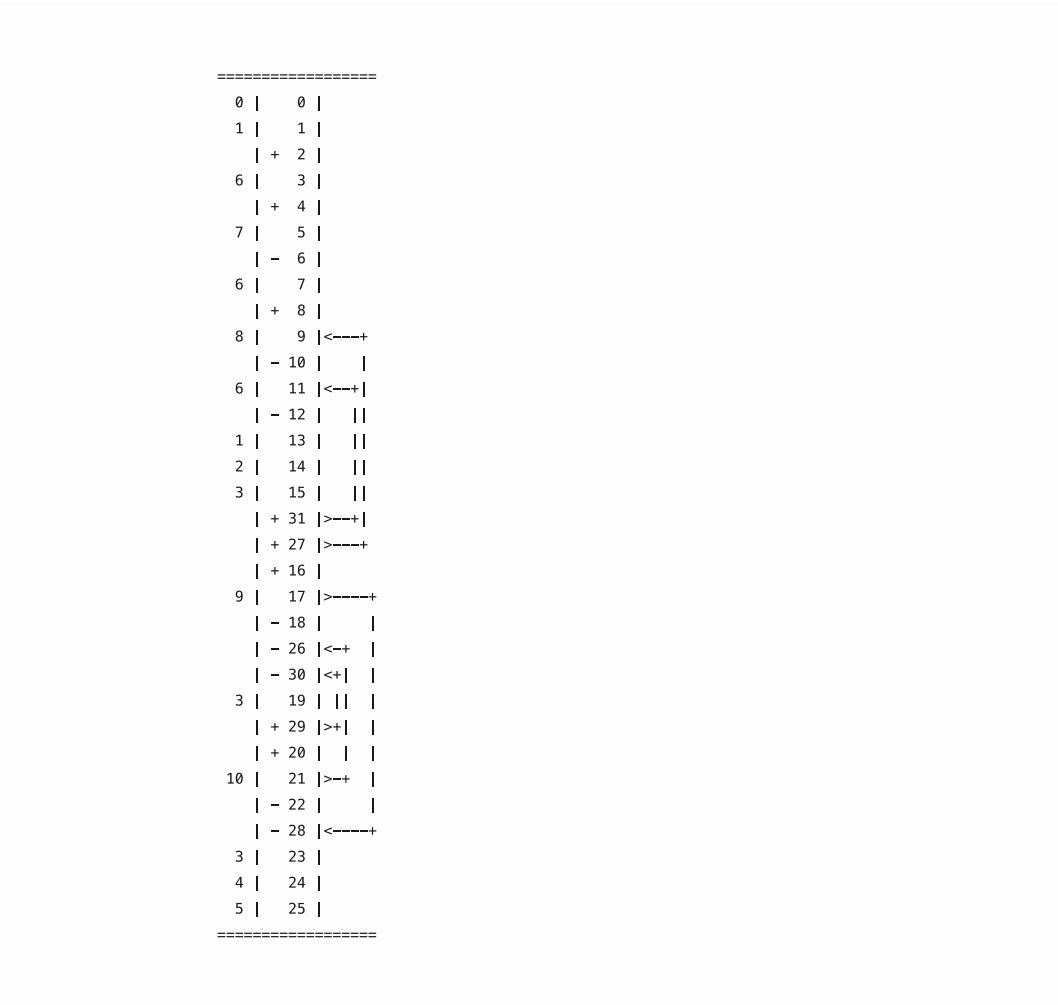

Figure 3.4: **The complete topology sketch represented by the triple-list table.**

v	$\#M$
0	→ [(0, @C:0)]
1	→ [(1, @C:1), (13, @C:13)]
2	→ [(14, @C:14)]
3	→ [(15, @C:15), (19, @C:20), (23, @C:24)]
4	→ [(24, @C:25)]
5	→ [(25, @C:26)]
6	→ [(3, @C:3), (7, @C:7), (11, @C:11)] → [(2, @C:2), (12, @C:12)]
7	→ [(5, @C:5)] → [(4, @C:4), (6, @C:6)]
8	→ [(9, @C:9)] → [(8, @C:8), (10, @C:10)]
9	→ [(17, @C:17)] → [(27, @C:16), (26, @C:19)] → [(16, @C:16), (18, @C:18)]
10	→ [(21, @C:22)] → [(20, @C:21), (22, @C:23)]

Table 3.4: **The Content of the Mapping List after the Operation of** $make\_slice(u_{16}, u_9)$**.**

structure we propose, the complexity of *make_slice* and $free\_slice$ operation is guaranteed to have a complexity of $O(1)$.

Starting at graph vertex v_{10}, consider the mapping list illustrated in Table 3.1. Making a topology path from the topology vertex u_{21} to the right-hand side of the edge (v_3, v_9), which has the corresponding topology vertex u_{18}, we show the resulting triple-lists diagram in Table 3.4, 3.5, and 3.6. Consider the mapping list. $M[10]$ shows that the graph vertex v_{10} has only one corresponding topology vertex which is u_{21}. By following the link @C:21, we locate the topology vertex, which is in the slice handled by a slice descriptor @S:0 and at the position pointed by a topology descriptor @T:21. With the slice descriptor @S:0, we are able to get the rest of the topology vertex in the slice by iterating through its topology descriptor.

Consider the slice list shown in Table 3.5 after the operation of $make\_slice(u_{16}, u_9)$. A slice, which is newly added to the next of the original slice, moves away from all the topology vertices other than $< u_{26}, u_{19}, u_{20}, u_{21} >$. Notice that slices are implemented as circular lists. So the movement of vertices requires changing a constant

$\#S$	:0	:1
$\#T$		(0, M:0, 0) (1, M:0, 0) (6, M:1, 0) (6, M:0, 0) (7, M:1, 0) (7, M:0, 0) (7, M:1, 1) (6, M:0, 1) (8, M:1, 0) (8, M:0, 0) (8, M:1, 1) (6, M:0, 2) (6, M:1, 1) (1, M:0, 1) (2, M:0, 0) (3, M:0, 0) (9, M:2, 0) (9, M:0, 0) (9, M:2, 1)
	(9, M:1, 1) (3, M:0, 1) (10, M:1, 0) (10, M:0, 0)	
		(10, M:1, 1) (3, M:0, 2) (4, M:0, 0) (5, M:0, 0)
start	T:nullptr	T:22
end	T:nullptr	T:19

Table 3.5: **The Content of the Slice List after the Operation of** $make\_slice(u_{16}, u_9)$**.**

$\#C$	(c_1, c_2, c_3)
	(S:1, T:0, S:nullptr)
	(S:1, T:1, S:nullptr)
	(S:1, T:2, S:nullptr)
	(S:1, T:3, S:nullptr)
	(S:1, T:4, S:nullptr)
	(S:1, T:5, S:nullptr)
	(S:1, T:6, S:nullptr)
	(S:1, T:7, S:nullptr)
	(S:1, T:8, S:nullptr)
	(S:1, T:9, S:nullptr)
	(S:1, T:10, S:nullptr)
	(S:1, T:11, S:nullptr)
	(S:1, T:12, S:nullptr)
	(S:1, T:13, S:nullptr)
	(S:1, T:14, S:nullptr)
	(S:1, T:15, S:nullptr)
	(S:1, T:16, S:nullptr)
	(S:1, T:17, S:nullptr)
	(S:1, T:18, S:nullptr)
	(S:0, T:26, S:1)
	(S:0, T:19, S:nullptr)
	(S:0, T:20, S:nullptr)
	(S:0, T:21, S:1)
	(S:1, T:22, S:nullptr)
	(S:1, T:23, S:nullptr)
	(S:1, T:24, S:nullptr)
	(S:1, T:25, S:nullptr)

Table 3.6: **The Content of the Frame List after the Operation of** $make\_slice(u_{16}, u_9)$**.**

number of pointers. In fact, the mapping list and the frame list are closely interlinked to the slice list. Adding a topology vertex requires only a few insertions. As a result, the complexity of the operation of make-slice is guaranteed to be $O(1)$. The empirical execution speed is extremely fast as well. Similarly, the complexity guarantee and empirical speed of the free slice operation, which is merely a reverse process of make-slice operation, are comparable.

3.3 Implementation The Basic Circular Frame Routing Algorithm Using Triple List Data Structure

Algorithm 1 and 2 shows how the basic circular frame routing algorithm [2] is implemented on the refined circular frame interface. Given a constructed model of circular frame abstraction C and a set of net N which specifies the source and the target vertices. In lines 2 and 3, firstly acquire the corresponding topology vertices of the source graph vertex and the target graph vertex. In lines 5 to 9, choose a random topology vertex and perform the DEPTH-FIRST-ROUTE sub-function. Basically, the DEPTH-FIRST-ROUTE sub-function iterates all the topology vertex in the current slice. If a topology vertex has its corresponding graph vertex that matches the target graph vertex, perform the make-slice operation. However, if there is no such topology vertex in the current slice, push all the topology vertex with attribution of **POSITIVE_EDGE** or **NEGATIVE_EDGE** into a stack in which all the topology vertex is later chosen as a candidate entrance to the other slice. Therefore, the basic circular frame routing algorithm finds a deterministic solution by a depth-first-search in slices.

The algorithm proved that the circular frame interface perfectly solves the clearance problem encountered in geometrical routing algorithms and that the routing completion rate is guaranteed to be 100%. However, the quality of the routing

solution is extremely tangled. The transformed geometry sketches in a seriously tangled topology class could result in longer total wire length. Moreover, it is almost impossible to transform the topology class into a geometry routing result with no design rule violation. The fatal reason is that the algorithm directly performing make-slice operation to a topology vertex in the same slice which causes a topology path that detours the whole forest.

Since the lack of consideration of net ordering is a fatal issue, we suggest that the information of the forest can be used as a guide for the algorithm. Specifically, the attribution of the topology vertex is the key, which is either **POSITIVE_EDGE**, **NEGATIVE_EDGE**, or **VERTEX**.

Algorithm 1 the basic circular frame routing algorithm

```
1: procedure THE BASIC CIRCULAR FRAME ROUTING ALGORITHM(C,N)
2:     for all n ∈ N do
3:         S ← C.topo_vertices( source( n ) );
4:         T ← C.topo_vertices( target( n ) );
5:         Let h ∈ S;
6:         C.reset_slice_indexes();
7:         if ¬ DEPTH_FIRST_ROUTE(C,h,T) then
8:             return FAILURE;
9:         end if
10:    end for
11:    return SUCCESS;
12: end procedure
```

Algorithm 2 depth first route

```
1: procedure DEPTH FIRST ROUTE(C,h,T)
2:     m ← C.slice(h);
3:     if ∃t ∈ T, C.slice(t) = m then
4:         C.make_slice(h,t);
5:         return TRUE;
6:     end if
7:     (∗m).index ← 1;
8:     E ← stack of topology_descriptor;
9:     p ← C.next(h);
10:    while p ≠ h do
11:        if C.attribution( p ) ≠ VERTEX then
12:            q ← C.ordering_pair(p);
13:            s ← C.slice(q);
14:            if (∗s).index = 0 then
15:                E.push( p );
16:            end if
17:        end if
18:        p ← C.next(p);
19:    end while
20:    while E is not empty do
21:        w ← E.pop();
22:        h ← C.make_slice( h, w );
23:        if DEPTH_FIRST_ROUTE(C,h,T) then
24:            return TRUE;
25:        end if
26:    end while
27:    h ← C.free_slice(h);
28:    return FALSE;
29: end procedure
```

CHAPTER 4

Topology Escape Routing

The following is the overall topology routing flow that we propose to solve the topology escape routing problem. Given a set of terminals T within the boundary B and a set of nets N, we construct a Delaunay triangulation before finding an embedding frame from the triangulation. We propose several heuristic methods to acquire an embedding frame that considers the net ordering. Furthermore, we construct a triple-lists model of the circular frame abstraction that includes topology vertex attribute and information of the associated embedding frame. Then we incrementally develop four algorithms that consider the information of the forest to acquire routing solutions with minor path tangling problems. The first algorithm, slice-issue tangling avoidance algorithm, uses ROUTE-THROUTH procedure to alleviate the tangling problem caused by the make-slice operation from a topology vertex to the other topology vertices in the same slice. The second algorithm, root-issue tangling avoidance algorithm, gives ROUTE-THROUTH procedure higher priority and targets the topology vertices in the same slice specifically. The third algorithm, detour-aware tangling avoidance algorithm introduces the FIND-ENDTRANCE-TO-DIFF-SLICE procedure which cooperates with ROUT-THROUGH procedure to alleviate the tangling problem that arises from the problem of detouring successors. The fourth algorithm, topology escape routing algorithm, uses the breathe-first-search order instead of the depth-first-search order, which gives the lower level vertices a

higher priority of net ordering. Finally, the topology sketch is transformed into a homotopic geometrical sketch through a geometrical transformation process.

4.1 Topology Escape Routing Algorithm

An example of an embedding frame $F = (V, E)$ with a set of net N is shown in Figure 4.2 (a) as well as (b), where $V =< v_0, v_1, ..., v_{13} >$. There are nine trees in the forest, where $v_0, v_1, ..., v_8$ are roots. The circular frame concept considers the sequence of roots as a boundary of a topology class. $E =< (v_2, v_9), (v_9, v_{10}), (v_9, v_{12}), (v_{10}, v_{11}), (v_{12}, v_{13}) >$ are the edges of F. And $N = \{\{v_1, v_{11}\}, \{v_2, v_9\}, \{v_3, v_{10}\}, \{v_4, v_{13}, \{v_5, v_{12}\}\}\}$. The same color of the vertices denotes that those vertices belong to the same net. After constructing a circular frame, the basic circular frame routing algorithm finds the topology class illustrated in Figure 4.2 (a). Notice that the senseless decision of net $\{v_3, v_{10}\}$ results in the tangled topology paths for the rest of the nets. This is the main reason that the result acquired from the basic circular frame routing algorithm is not practical. In this case, a sensible and proper decision in terms of a topology class with the less tangled paths is shown in Figure 4.2 (b).

With the observation, we propose the simple-make-slice-issue tangling avoidance algorithm to acquire the solution in Figure 4.2 (b). The pseudo-code is given in Algorithm 3, 4, and 5. Firstly, The algorithm routes each of the topology paths from a terminal within the boundary to a terminal on the boundary. We notice that the ordering of the **target vertex** of a net is extremely important. Taking the net $n_1 = \{v_{10}, v_3\}$ and the net $n_2 = \{v_{13}, v_4\}$ for example, we refer to v_4 as the target vertex for net n_1 if the corresponding topology path is starting from vertex v_{13}; we refer to v_3 as the target vertex for net n_2 if the corresponding topology path is starting from vertex v_{10}. Consider Figure 4.2 (a), if v_3 is on the right of v_4 and v_5,

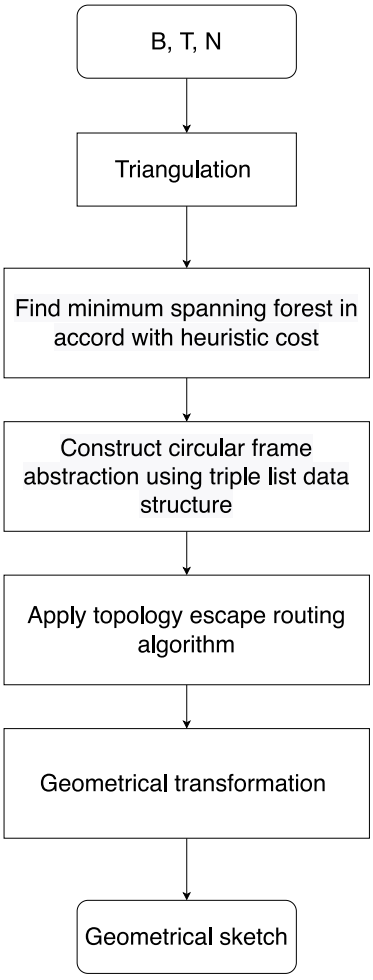

Figure 4.1: **Topology Escape Routing Flow.** The flow shows how we generate a geometrical sketch from a given boundary, terminals, and nets. The flow is generic and based on the circular frame interface. We can apply a different policy to acquiring a spanning forest and the routing process.

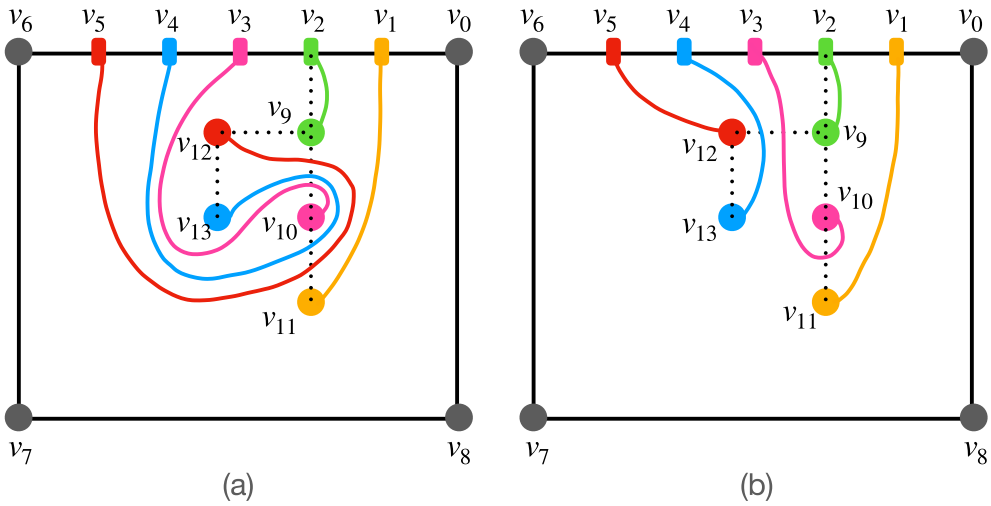

Figure 4.2: **Topology Escape Routing Flow.** The flow shows how we generate a geometrical sketch from a given boundary, terminals, and nets. The flow is generic and based on the circular frame interface. We can apply a different policy to acquiring a spanning forest and the routing process.

it is insensible to make a topology path passing by the left side of v_{12} and v_{13}. Our topology escape routing algorithm intends to behave by this simple logic. In the first line Algorithm 3, all the nets are routed sequentially in a depth-first-search order according to the associated embedding frame. That is, in this case, we will route the nets in the following order: $< \{v_9, v_2\}, \{v_{10}, v_3\}, \{v_{11}, v_1\}, \{v_{12}, v_5\}, \{v_{13}, v_4\} >$. For each net, we get the topology vertices S from the source vertex of the net and the topology vertices T from the target vertex. If there exists a topology vertex $t \in T$ that resides in a different slice, we simply perform the depth-first-route method to find a connection. And if all the topology vertices $t \in T$ are in the same slice as a topology vertices $h \in S$, we do not rush to make-slice from h to t. Instead, we use the **counterclockwise-route-through** or **clockwise-route-through** method specified in Algorithm 4 and 5 respectively.

Algorithm 3 Simple-make-slice-issue tangling avoidance algorithm

```
1: procedure SIMPLE-MAKE-SLICE-ISSUE TANGLING AVOIDANCE ALGORITHM(C)
2:     N ← get_nets_in_dfs_order( C );
3:     for all n ∈ N do
4:         S ← C.topo_vertices( source( n ) );
5:         T ← C.topo_vertices( target( n ) );
6:         Let h ∈ S;
7:         C.reset_slice_indexes();
8:         if ∃t ∈ T, C.slice( t ) ≠ C.slice(h) then
9:             if DEPTH_FIRST_ROUTE(C,h,T) then
10:                return SUCCESS;
11:            else
12:                return FAILURE;
13:            end if
14:        else
15:            Let t ∈ T;
16:            if C.shortest_direction(h,t) = CCW then
17:                h ← CCW_ROUTE_THROUGH(C,h,t);
18:            else
19:                h ← CW_ROUTE_THROUGH(C,h,t);
20:            end if
21:        end if
22:    end for
23: end procedure
```

Algorithm 4 CCW route through

```
1: procedure CCW ROUTE THROUGH(C,h,t)
2:     while TRUE do
3:         p ← C.next(h);
4:         while p ≠ h do
5:             if p = t then
6:                 return h ← C.make_slice(h,p);
7:             end if
8:             if C.attribution( p ) = POSITIVE_EDGE then
9:                 q ← C.ordering_pair(p);
10:                if C.slice(q) = C.slice(h) then
11:                    h ← C.make_slice(h,p);
12:                    break;
13:                end if
14:            end if
15:            p ← C.next(h)
16:        end while
17:    end while
18: end procedure
```

Algorithm 5 CW route through

```
1: procedure CW ROUTE THROUGH(C,h,t)
2:     while TRUE do
3:         p ← C.prev(h);
4:         while p ≠ h do
5:             if p = t then
6:                 return h ← C.make_slice(h,p);
7:             end if
8:             if C.attribution( p ) = NEGATIVE_EDGE then
9:                 q ← C.ordering_pair(p);
10:                if C.slice(q) = C.slice(h) then
11:                    h ← C.make_slice(h,p);
12:                    break;
13:                end if
14:            end if
15:            p ← C.prev(h)
16:        end while
17:    end while
18: end procedure
```

Let us explain the algorithm in this case with the triple-list tables shown in the following. For the first net $\{v_9, v_2\}$, we acquire $h = u_4$ as the starting topology vertex of the topology path for this net. We can see that the corresponding topology vertex $t = u_2$ of the graph vertex v_2 is in the same slice with h. And in line 16 in Algorithm 3, by the shortest direction we mean the direction of the least number of iterations, which in this case is clockwise. As a result, the $CW\_ROUTE\_TROUGH(C, u_4, u_2)$ function should be executed. And thus, we clockwise iterate the topology vertices in the current slice, encountering u_2, and perform the make-slice operation to it before encountering any topology vertices with the attribution of NEGATIVE_EDGE as shown in line 8 in Algorithm 4. The results are shown in Figure 4.3.

For the second net $\{v_{10}, v_3\}$, we acquire $h = u_6$ as the starting topology vertex. Since the target topology vertex $t = u_{23}$ is in the same slice with h and the direction of the least number of iterations is counterclockwise, the CCW_ROUTE_TROUGH (C, u_6, u_{23}) function is executed. As iterating the topology vertices counterclockwise, we encounter u_7 which has the attribution of POSITIVE_EDGE and whose ordering pair u_9 is in the same slice as well. Therefore, we perform the make-slice operation to u_7 and continue to iterate counterclockwise. Then, for the same reason, the make-slice operation to the topology vertex u_{13} is performed. Note that the two newly added ordering pairs (u_{29}, u_{30}) and (u_{31}, u_{32}) are actually created and connected as part of the topology path for this net. Please look into chapter three for this mechanism if you are confused. At last, The iteration goes on and the make-slice operation to topology vertex u_{23} is performed. The result is shown in Figure 4.4.

The following two net $\{v_{11}, v_1\}$ and $\{v_{12}, v_5\}$ is routed as expected as the first two nets. The result is shown in Figure 4.5. For the last net $\{v_{13}, v_4\}$, we acquire $h = u_{16}$ as the starting topology vertex. Notice that the target topology

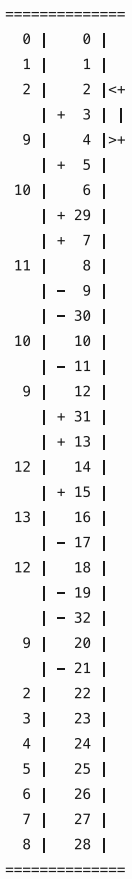

Figure 4.3: **The Triple-lists Table of Example One (a).**

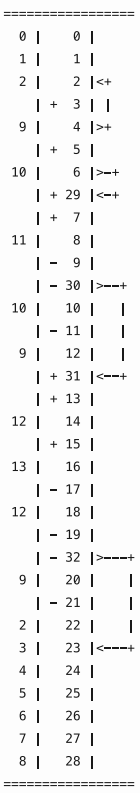

Figure 4.4: **The Triple-lists Table of Example One (b).**

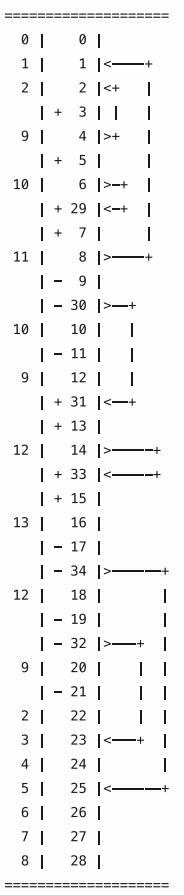

Figure 4.5: **The Triple-lists Table of Example One (c).**

vertex $t = u_{24}$ which corresponds to the graph vertex v_4 is in a different slice from h. As a result, a simple depth-first-route will eventually find the topology vertex u_{13} as the currently only entrance from the eighth slice to the seventh slice where u_{24} currently resides, make-slice to u_{13}, and then make-slice to u_{24}. The result is shown in Figure 4.6, which is exactly the solution in figure 4.2 (b).

However, we discover an edge case that results in the tangled solution as illustrated in Figure 4.7 (a). The problem only happens when the target vertex is

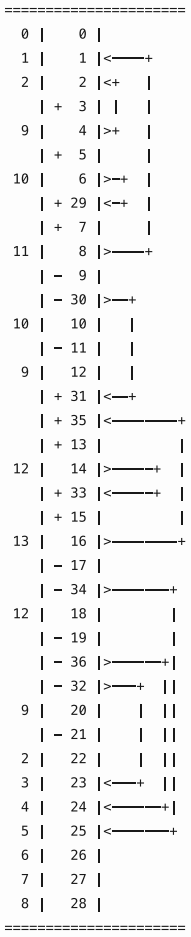

Figure 4.6: **The Triple-lists Table of Example One (d).**

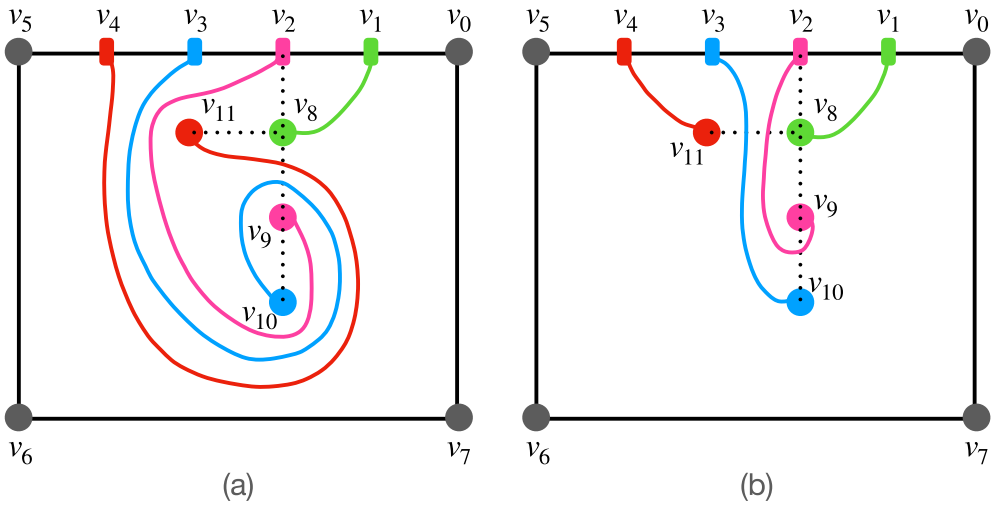

Figure 4.7: **The Topology Sketch of Example Two.** Terminals associated with the same net are annotated in the same color. (a) The solution is acquired by the simple-make-slice-issue tangling avoidance algorithm. (b) The solution is acquired by the root-issue tangling avoidance algorithm.

also a root and a predecessor of the source vertex. We explain it with Figure 4.8. Consider the second net $\{v_9, v_2\}$. For the target vertex v_2, there are two corresponding topology vertices u_{18} and u_2 which currently reside in different slices from each other. According to line 8 in Algorithm 3, since there exists a topology vertex u_2 in a different slice, the depth-first-route method is used to find a path for the net, therefore, resulting in the detour around graph vertex v_{10} and all the following tangled topology paths. The proper topology sketch is shown in Figure 4.7 (b). In order to acquire such a solution, we develop the second version of the topology escape routing algorithm as shown in Algorithm 6.

In Algorithm 6, we modify to check first if there exists a topology vertex $t \in T$ that resides in the same slice as h before using the counterclockwise-route-through or clockwise-route-through method in line 8 to 13, which allows the topology vertex

Algorithm 6 Root-issue tangling avoidance algorithm

```
1: procedure ROOT-ISSUE TANGLING AVOIDANCE ALGORITHM(C)
2:     N ← get_nets_in_dfs_order( C );
3:     for all n ∈ N do
4:         S ← C.topo_vertices( source( n ) );
5:         T ← C.topo_vertices( target( n ) );
6:         Let h ∈ S;
7:         C.reset_slice_indexes();
8:         if ∃t ∈ T, C.slice( t ) = C.slice( h ) then
9:             if C.shortest_direction(h,t) = CCW then
10:                h ← CCW_ROUTE_THROUGH(C,h,t);
11:            else
12:                h ← CW_ROUTE_THROUGH(C,h,t);
13:            end if
14:        else
15:            Let t ∈ T;
16:            if DEPTH_FIRST_ROUTE(C,h,T) then
17:                return SUCCESS;
18:            else
19:                return FAILURE;
20:            end if
21:        end if
22:    end for
23: end procedure
```

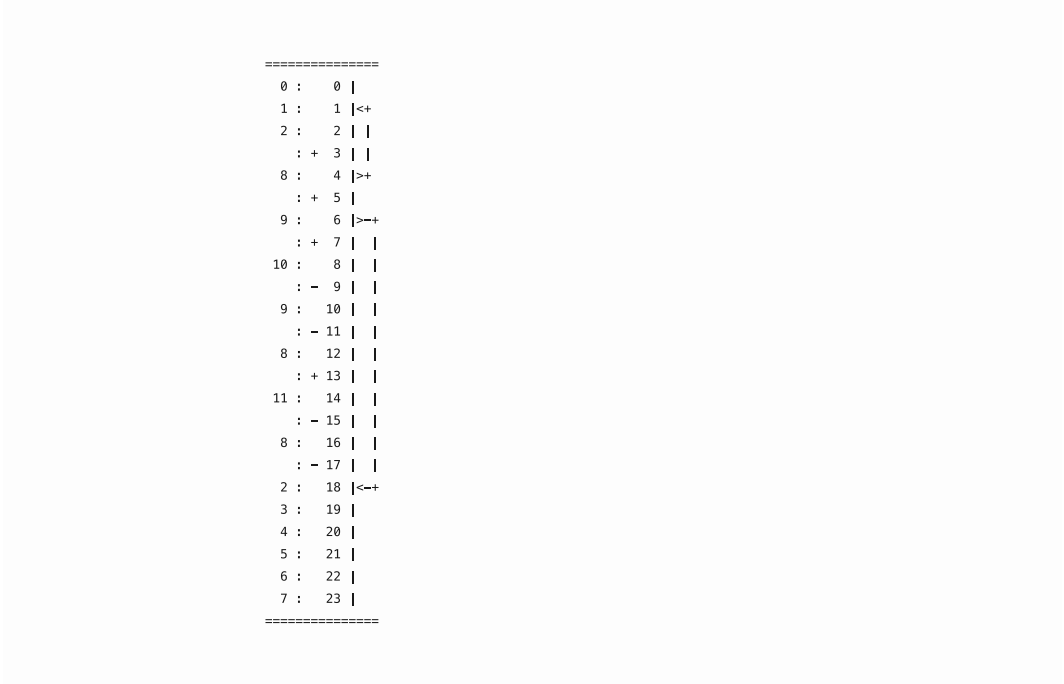

Figure 4.8: **The Triple-lists Table of Example Two (a).**

u_6 to use the **CCW-ROUTE-THROUGH** function and to make-slice to u_7 and then to u_{13} as shown in the figure 4.9. The resulting topology path along with the topology sketch acquired by the version two topology escape routing algorithm is the same as the solution illustrated in the figure4.7 (b).

There is still an edge case for the root-issue tangling avoidance algorithm. Consider the net $\{v_8, v_2\}$ in the case illustrated in Figure 4.10 (a). The topology path for this net still detours around v_9 before it makes it to v_2. The reason is that the graph vertex v_2 is not a root, however, its corresponding topology vertex u_2 is sliced into a different slice by the time topology path is determined for the net $\{v_7, v_1\}$. As a result, the else branch in line 14 in Algorithm 6 is taken and u_7 is made-slice to u_{14} and then to u_2 as shown in Figure 4.11.

Targeting this sort of issue, we develop the detour-aware tangling avoidance algorithm along with the new method **find-entrance-to-diff-slice**. The pseudo-

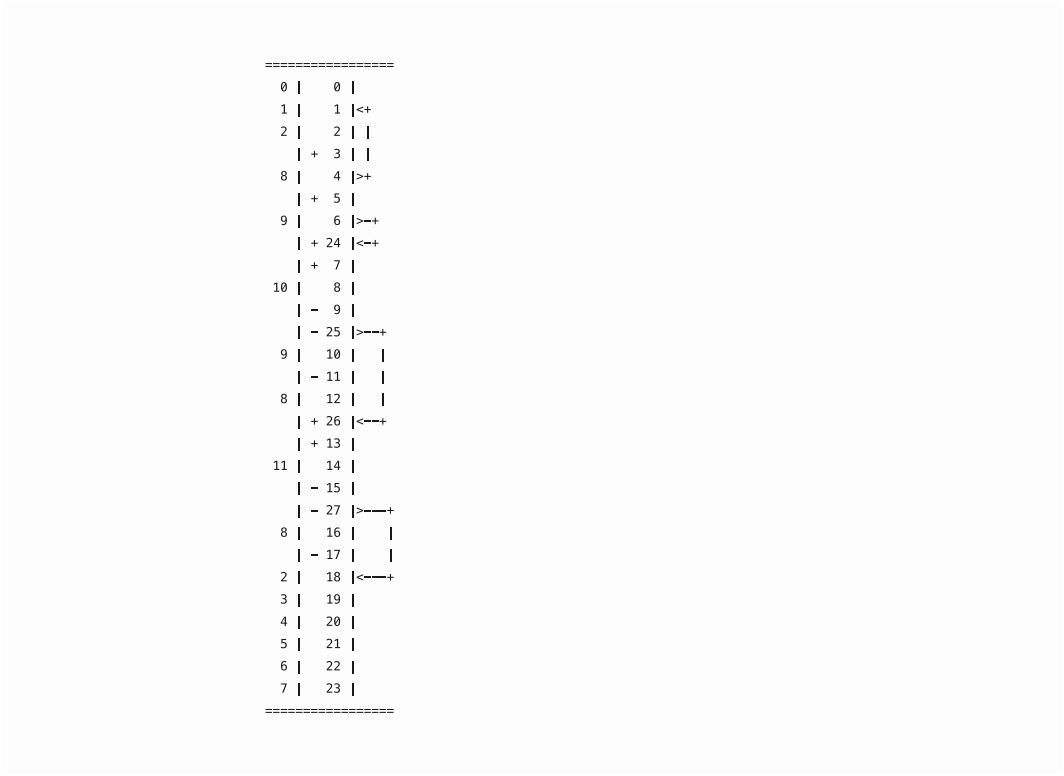

Figure 4.9: **The Triple-lists Table of Example Two (b).**

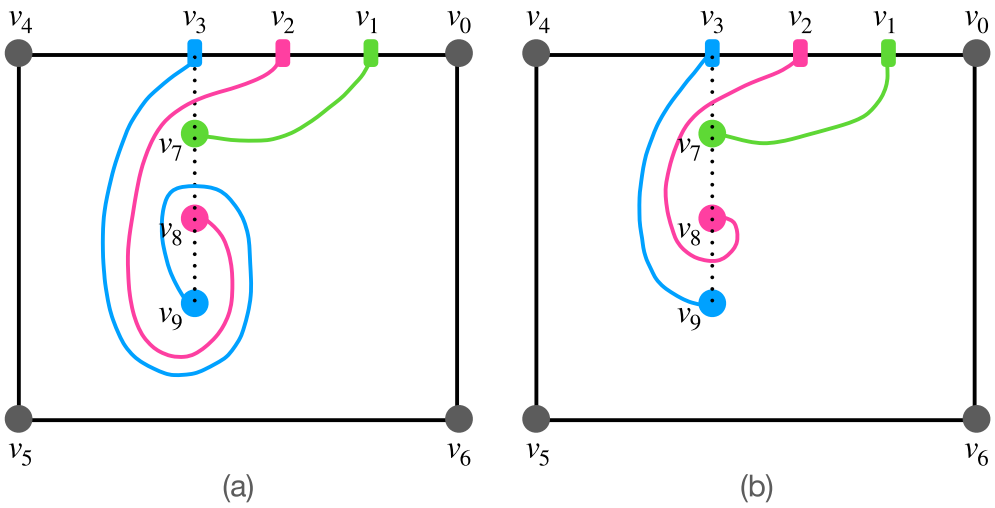

Figure 4.10: **The Topology Sketch of Example Three.** Terminals associated with the same net are annotated in the same color. (a) The solution is acquired by the root-issue tangling avoidance algorithm. (b) The solution is acquired by the detour-aware tangling avoidance algorithm.

```
================
 0 |    0 |
 1 |    1 |<—+
 2 |    2 |<+|
 3 |    3 | ||
   | + 20 |>+|
   | +  4 |  |
 7 |    5 |>—+
   | +  6 |
 8 |    7 |>——+
   | +  8 |    |
 9 |    9 |    |
   | – 10 |    |
 8 |   11 |    |
   | – 12 |    |
 7 |   13 |    |
   | – 14 |    |
   | – 19 |<——+
 3 |   15 |
 4 |   16 |
 5 |   17 |
 6 |   18 |
================
```

Figure 4.11: **The Triple-lists Table of Example Three (a).**

code is shown in Algorithm 7 and 8. The philosophy is that it is necessary to make-slice to u_{14} in some way or another ultimately; however, it is not necessary to detour around the graph vertex v_9. As a result, modification is made in lines 15-29 in Algorithm 7. The **find-entrance-to-diff-slice** method is shown in Algorithm 8, which finds all entrances to a different slice and stores all the candidate entrances. Then, we connect to each entrance e via counterclockwise-route-through or clockwise-route-through methods. After making-slice to the entrance and exit to a different slice, we apply the depth-first-search method shown in lines 22-23 to route to the target vertex. Now consider Figure 4.12, the only entrance topology vertex is u_{14}, and the direction of the shortest iteration from u_7 to u_{14} is counterclockwise. As a result, u_7 is made-slice to u_8 and then to u_{21} before using the depth-first-route to make-slice to u_2, which is illustrated by Figure 4.12. And the final result is shown in Figure 4.10 (b).

Still, there is room for further improvement. Consider the case in Figure 4.13. Notice that the topology path for net $\{v_{10}, v_4\}$, it is determined before the topology path for net $\{v_{11}, v_3\}$ being forced to detour in that the graph vertex v_{10} is traversed before v_{11} according to the depth-first-search order. In other word, the net $\{v_{10}, v_4\}$ has a higher priority than net $\{v_{11}, v_3\}$, which results in the insensible solution in Figure 4.13 (a). Therefore, we propose the fourth version of the topology escape algorithm which modifies the net ordering into a breadth-first-search order, as shown in the line 2 in Algorithm 9. That is, in this case, the net is processed in the order of $< \{v_8, v_1\}, \{v_{11}, v_3\}, \{v_9, v_2\}, \{v_{10}, v_4\} >$. The underlying philosophy is to have a lower-level vertex to possess a higher priority. The process of net-by-net determination according to the topology algorithm of the fourth version is illustrated in Figure 4.14, 4.15, 4.16, and 4.17. And the ultimate result is shown in Figure 4.13 (b).

Algorithm 7 Detour-aware tangling avoidance algorithm

```
1: procedure DETOUR-AWARE TANGLING AVOIDANCE ALGORITHM(C)
2:     N ← get_nets_in_dfs_order( C );
3:     for all n ∈ N do
4:         S ← C.topo_vertices( source( n ) );
5:         T ← C.topo_vertices( target( n ) );
6:         h ∈ S;
7:         C.reset_slice_indexes();
8:         if ∃t ∈ T, C.slice( t ) = C.slice( h ) then
9:             if C.shortest_direction(h,t) = CCW then
10:                h ← CCW_ROUTE_THROUGH(C,h,t);
11:            else
12:                h ← CW_ROUTE_THROUGH(C,h,t);
13:            end if
14:        else
15:            E ← FIND_ENTRANCE_TO_DIFF_SLICE(C,h);
16:            for all e ∈ E do
17:                if C.shortest_direction(h,e) = CCW then
18:                    h ← CCW_ROUTE_THROUGH(C,h,e);
19:                else
20:                    h ← CW_ROUTE_THROUGH(C,h,e);
21:                end if
22:                if DEPTH_FIRST_ROUTE(C,h,T) then
23:                    break;
24:                else
25:                    while C.attribution( h ) ≠ VERTEX do
26:                        h ← C.free_slice( h );
27:                    end while
28:                end if
29:            end for
30:        end if
31:    end for
32: end procedure
```

Algorithm 8 Find entrance to diff slice

```
1: procedure FIND ENTRANCE TO DIFF SLICE(C,h)
2:     let E ← a array of topology descriptor;
3:     p ← C.next(h);
4:     while p ≠ h do
5:         if C.attribution( p ) ≠ VERTEX then
6:             q ← C.ordering_pair(p);
7:             if C.slice(q) ≠ C.slice(h) then
8:                 E.push_back(p);
9:             end if
10:        end if
11:        p ← C.next(h)
12:    end while
13:    return E;
14: end procedure
```

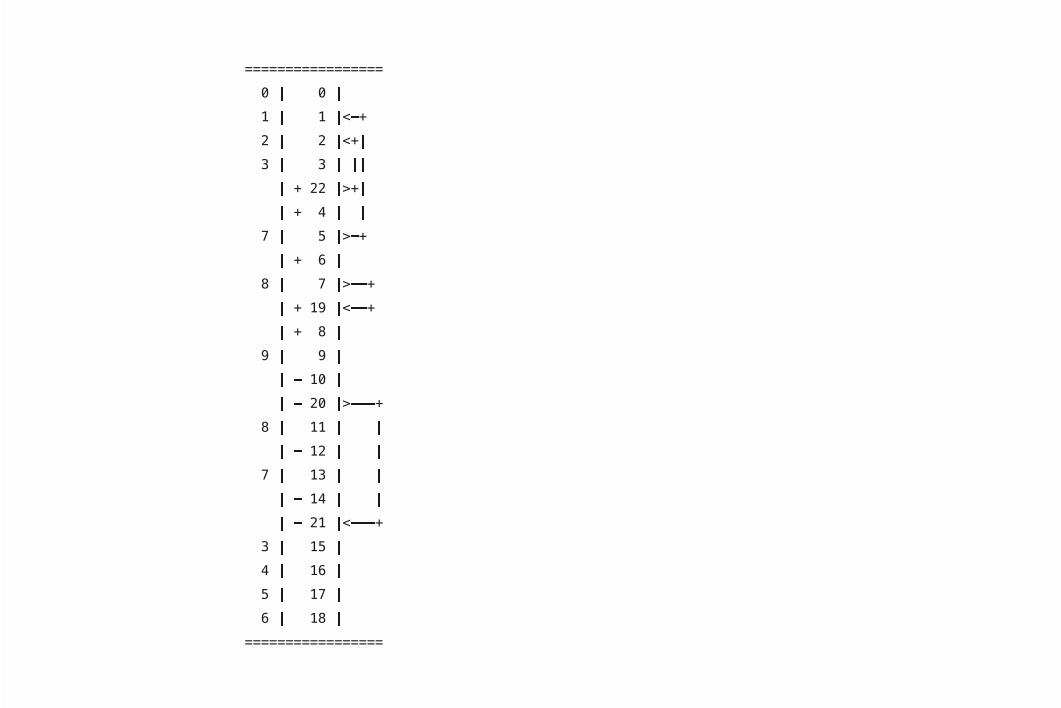

Figure 4.12: **The Triple-lists Table of Example Three (b).**

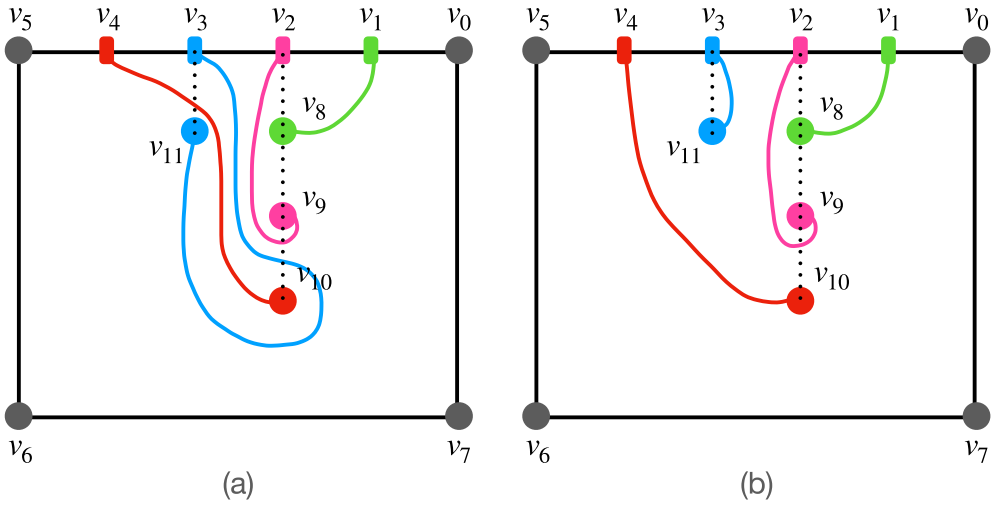

Figure 4.13: **The Topology Sketch of Example Four.** Terminals associated with the same net are annotated in the same color. (a) The solution is acquired by the detour-aware tangling avoidance algorithm. (b) The solution is acquired by the topology escape routing algorithm.

```
==============
 0 |    0 |
 1 |    1 |<+
   | +  2 | |
 8 |    3 |>+
   | +  4 |
 9 |    5 |
   | +  6 |
10 |    7 |
   | -  8 |
 9 |    9 |
   | - 10 |
 8 |   11 |
   | - 12 |
 1 |   13 |
 2 |   14 |
 3 |   15 |
   | + 16 |
11 |   17 |
   | - 18 |
 3 |   19 |
 4 |   20 |
 5 |   21 |
 6 |   22 |
 7 |   23 |
==============
```

Figure 4.14: **The Triple-lists Table of Example Four (a).**

Algorithm 9 Topology escape algorithm

```
1: procedure TOPOLOGY ESCAPE ALGORITHM(C)
2:     N ← get_nets_in_bfs_order( C );
3:     for all n ∈ N do
4:         S ← C.topo_vertices( source( n ) );
5:         T ← C.topo_vertices( target( n ) );
6:         h ∈ S;
7:         C.reset_slice_indexes();
8:         if ∃t ∈ T, C.slice( t ) = C.slice( h ) then
9:             if C.shortest_direction(h,t) = CCW then
10:                h ← CCW_ROUTE_THROUGH(C,h,t);
11:            else
12:                h ← CW_ROUTE_THROUGH(C,h,t);
13:            end if
14:        else
15:            E ← FIND_ENTRANCE_TO_DIFF_SLICE(C,h);
16:            for all e ∈ E do
17:                if C.shortest_direction(h,e) = CCW then
18:                    h ← CCW_ROUTE_THROUGH(C,h,e);
19:                else
20:                    h ← CW_ROUTE_THROUGH(C,h,e);
21:                end if
22:                if DEPTH_FIRST_ROUTE(C,h,T) then
23:                    break;
24:                else
25:                    while C.attribution( h ) ≠ VERTEX do
26:                        h ← C.free_slice( h );
27:                    end while
28:                end if
29:            end for
30:        end if
31:    end for
32: end procedure
```

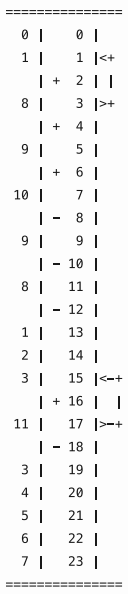

Figure 4.15: **The Triple-lists Table of Example Four (b).**

4.2 Heuristic Method of Finding a Forest for Circular Frame abstraction

The topology escape routing algorithm assumes that the embedding frame has been constructed according to the overall information of the net ordering and determines the routing paths with the guide of the information of the embedding frame. Therefore, the construction of the embedding frame exerts great influence on the quality of the solution.

To acquire an embedding frame, we firstly construct a triangulation before applying Prim's algorithm to find a minimum spanning forest among the triangulation edges. Initially, all the vertices on the boundary are selected as a root of a tree respectively. Then the algorithm builds the forest by selecting one vertex into one of the trees at a time, at each time adding the smallest edge weight from a vertex

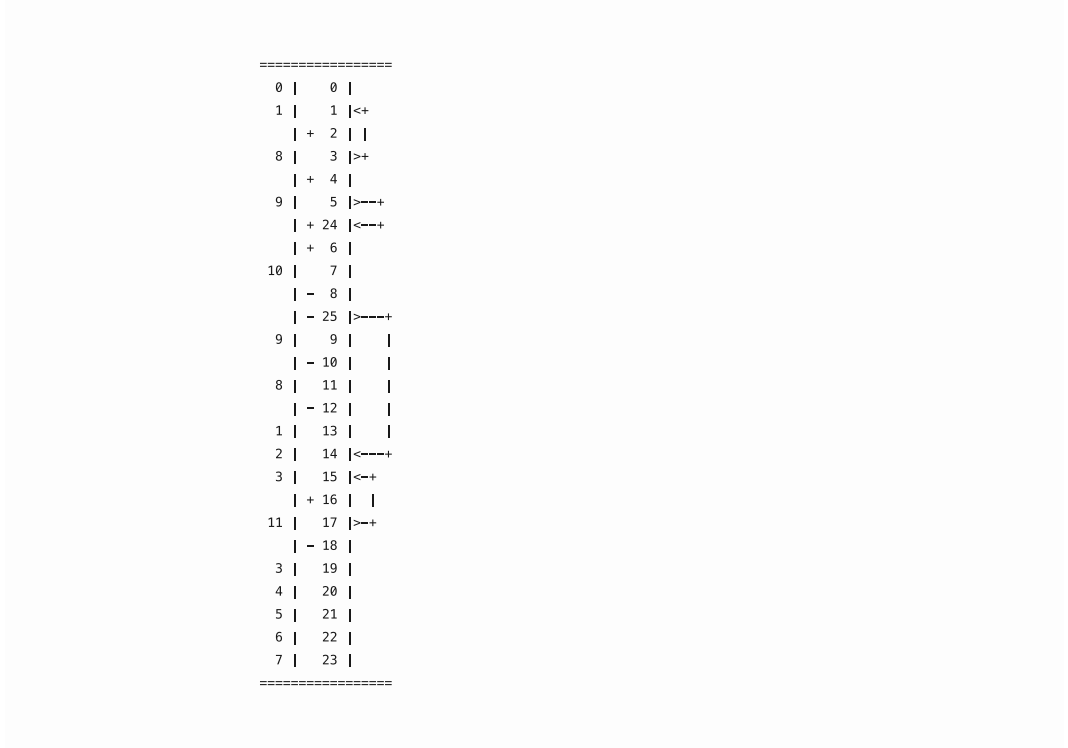

Figure 4.16: **The Triple-lists Table of Example Four (c).**

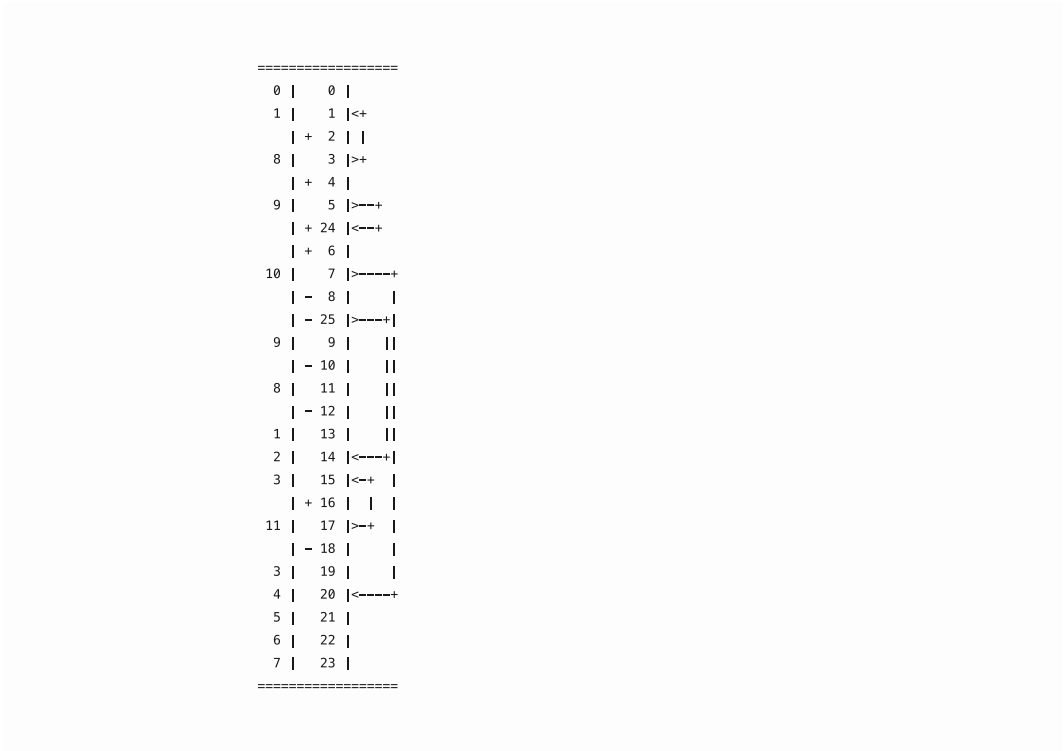

Figure 4.17: **The Triple-lists Table of Example Four (d).**

currently in the tree to another candidate vertex.

We propose five heuristic methods to determine the weight of each edge. The five styles of embedding frames are listed in the following.

- minimum spanning forest of Euclidean distance
- minimum spanning forest of A* heuristic Euclidean distance
- minimum level spanning forest
- minimum spanning forest of heuristic target net offset
- minimum spanning forest of target net offset from the root

The minimum spanning forest of Euclidean distance simply applies the Euclidean distance of two adjacent vertices u and v in the triangulation as the weight of each edges $W_1(u,v) = ((u_x - v_x)^2 + (u_y - v_y)^2)^1/2$, where (u_x, u_y) is the coordinate associated to the vertex u and (v_x, v_y) is the coordinate associated to the vertex v. This method does not take the net ordering into consideration and does not exert the advantage of the topology escape routing algorithm.

The minimum spanning forest of A* heuristic Euclidean distance applies the Prim's algorithm using the heuristic edge weight $W_2(u,v) = E(u,v) + H(u,v)$, where $\{u,v\}$ is an edge of the vertex u currently selected in the tree and the vertex v which is a candidate to be selected; $E(u,v)$ denotes the Euclidean distance between the coordinate associated with the vertices u and v; and $H(u,v)$ denotes the Euclidean distance between the coordinate associated with the vertices u and w, given that the vertex w and v are associated to the same two-pin net. For example, consider the edge $e = \{v_3, v_{16}\}$ in the triangulation shown in Figure 4.18, its edge weight $W_2(v_3, v_{16}) = \alpha + \beta$, where α is the Euclidean distance between vertex v_3 and

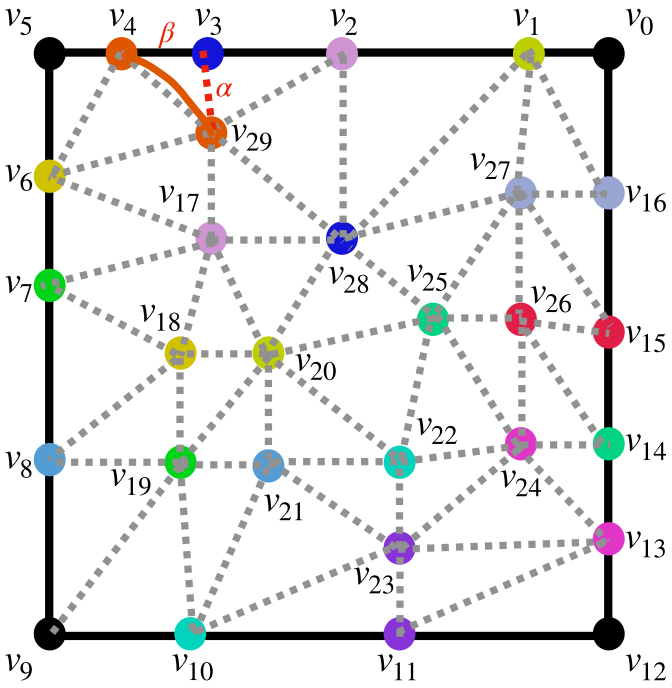

Figure 4.18: **The Illustration of the Minimum Spanning Forest of A* heuristic Euclidean Distance.** The heuristic edge weight $W_2(v_4, v_{29}) = E(v_4, v_{29}) + H(v_4, v_{29}) = \alpha + \beta$, where α is the Euclidean distance between vertex v_3 and v_4, and β is the Euclidean distance between vertex v_4 and v_{16}.

v_4, and β is the Euclidean distance between vertex v_4 and v_{16}. Suppose that the edge e is selected, the resulting topology path for the net $\{v_4, v_{16}\}$ is illustrated in Figure 4.18. As you can see, β is the heuristic cost since the determination of this topology path for this net conceptually intends to route to the left side of the vertex v_3 and eventually to the target vertex v_4. Unfortunately, despite some improvement over the first style of embedding frame, the A* heuristic cost part of the edge weight account for the improvement mostly and the Euclidean distance part is almost always a disturbance in terms of the quality.

The minimum level spanning forest strives to acquire a forest that the av-

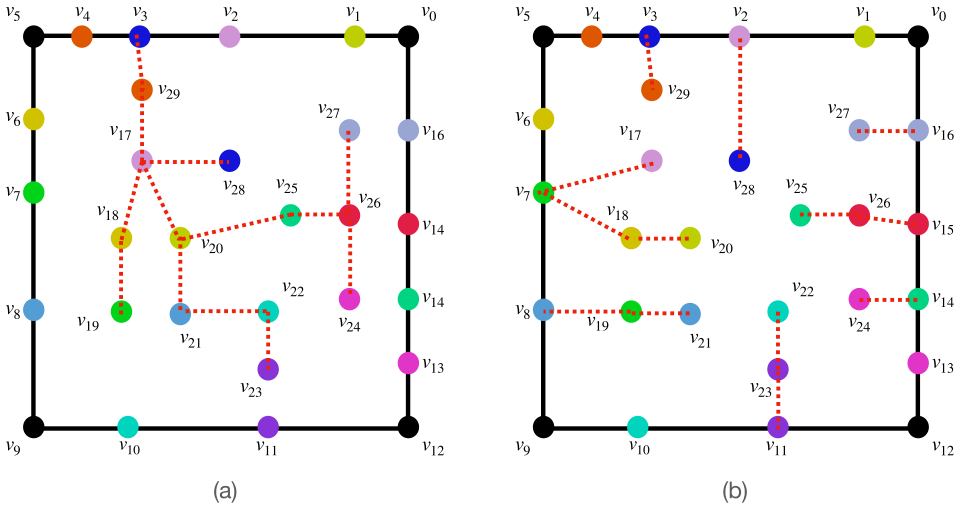

Figure 4.19: **The Illustration of the Minimum Level Spanning Forest.** The comparison between a random spanning forest and a minimum level spanning forest. The edges of the forests are annotated in red. (a) All the terminals in the boundary belong to the tree with the root v_3. (b) Terminals are evenly distributed to different roots to acquire a spanning forest with an overall minimum level.

erage level of the trees is minimized by using the heuristic edge cost $W_3(u, v) = (L(u), D(v, w))$, where $L(u)$ is the level of vertex u and $D(v, w)$ is the Euclidean distance of the vertices v and w which associate to the same net. It is considered an edge with less cost if its $L(u)$ is smaller, and if two edges with the same $L(u)$, the edge cost with the less Euclidean distance part is considered a less one. Consider the illustration in Figure 4.19. The dotted edges colored in red in each triangulation denote that they are selected as part of each forest, respectively. In Figure 4.19 (a) shows a random forest that has all the vertices in the boundary be selected in a tree with the root v_3. In comparison, in Figure 4.19 (b) shows the minimum level spanning forest. This style of forest empirically contributes to a better solution than the previous two styles, suggesting that evenly leveled trees instead of a big whole tree are more appreciated.

The last two styles of the forest completely focus on net ordering. Their goals are to build a tree of vertices with their target vertices conceptually as close to each other as possible. Firstly, we introduce the **heuristic target net offset** $W_3(u, v)$ to the weight of each edge $\{u, v\}$ in the triangulation. Using the heuristic target net offset as the weight function of each edge, the weight of each edge is annotated in Figure 4.20 (a). The computation of the weight function is described in the following. If both u, v resides on the boundary, the edge is illegal to be selected as part of the forest; therefore, it is given a maximum value to the cost. If one of u, v resides on the boundary and the other is not, without loss of generality, say, u is on the boundary and v is in the boundary, $W_3(u, v)$ is given a value of the shortest distance from u to the target vertex of v. Take $W_3(v_3, v_{16})$ in Figure 4.20 (a) as an example, the shortest offset from v_3 to the target vertex of v_{16} which is v_4 is to move counterclockwise by one offset. That is, $W_3(v_3, v_{16}) = 1$. Again, take $W_3(v_9, v_{21})$ in Figure 4.20 (a) as an example, the shortest distance from v_9 to the target vertex of v_{21} which is v_8 is to move clockwise by two offsets. That is, $W_3(v_9, v_{21}) = 2$. And if both of u, v reside in the boundary, $W_3(u, v)$ is given a value of the shortest offset from the target vertex of u to the target vertex of v. Take $W_3(v_{24}, v_{22})$ as an example, the target vertex of v_{24} is v_{12}, and the target vertex of v_{22} is v_9. The shortest offset from v_{12} to v_9 is one clockwise movement, that is, $W_3(v_{24}, v_{22}) = 3$.

With the edge weight computed by the heuristic target net offset, we acquire the minimum spanning forest illustrated in Figure 4.20 (b). Consider the edge $\{v_{16}, v_4\}$, $\{v_{15}, v_{27}\}$, $\{v_{14}, v_{26}\}$, $\{v_{12}, v_{24}\}$ and $\{v_{23}, v_{10}\}$. Their weight is given the minimum value in that both vertices of each edge are associated with the same net. Again, consider the vertices v_1, v_2, and v_3. They are relatively close to each other in terms of the offset, which results in the vertices v_{20}, v_{17}, and v_{28} being selected in the tree with the root v_2. This style of forest empirically has even trees, additionally,

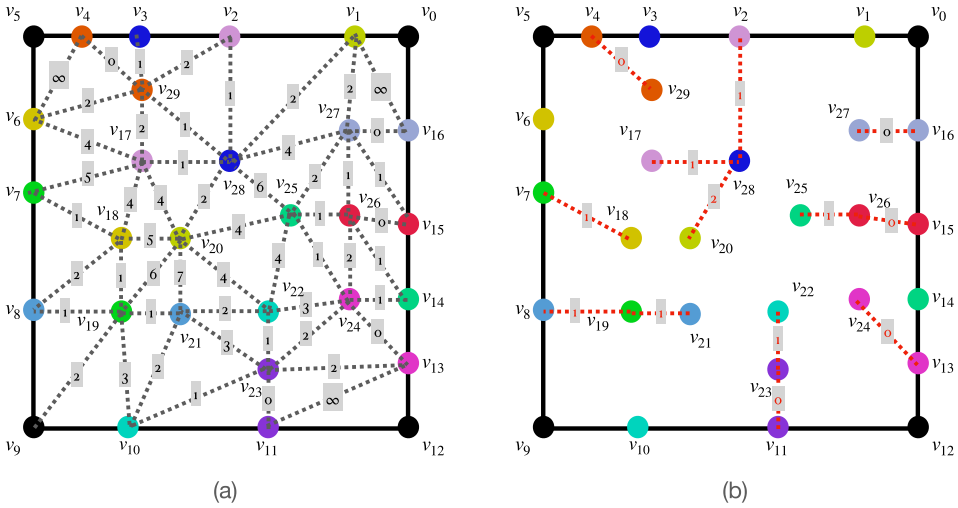

Figure 4.20: **The Illustration of the Minimum Spanning Forest of Heuristic Target Net Offset.** (a) The weights of each edge are annotated respectively. (b) The corresponding minimum spanning forest is annotated in red.

with the consideration of the target net offset, the solution acquired by the topology escape routing algorithm applying this heuristic forest outperforms all the other styles of the forest.

At last, we try to apply an enhanced version of the heuristic target net offset, which dynamically computes the net offset from the current root instead of their target vertices. Initially, each of the vertices on the boundary is selected as a root of its tree. For any vertex that is connected to those roots with an edge, the edge weight is given to the edge as annotated in Figure 4.21 (a). After each of those edges with the least cost, zero weight, in this case, is selected by a root, each of the edge weights of new adjacent edges is computed and added to the priority queue. For example, suppose that v_{23} is selected by v_1, the edge $\{v_{23}, v_{22}\}$ of weight 2, the edge $\{v_{23}, v_{25}\}$ of weight 3, the edge $\{v_{23}, v_{24}\}$ of weight 2, and the edge $\{v_{23}, v_{21}\}$ of weight 5 are added to the priority queue. After all the vertices with zero weight edge

adjacent to a root are selected, the current state is illustrated in Figure 4.21 (b). As shown in Figure 4.21 (a), at the moment, v_{20} is to be selected by v_{18} for the reason that the edge weight of $\{v_{18}, v_{20}\}$ is 1 and is the least one. After v_{20} being selected, the edge $\{v_{20}, v_{21}\}$ of weight 2, and the edge $\{v_{20}, v_{24}\}$ of weight 1 are added to the priority queue. Now we are to explain the **target net offset from the root**. Take the edge $\{v_{20}, v_{24}\}$ for example, v_{20} is currently selected in a tree with the root v_4. And the target vertex of v_{21} is v_3. Then, the weight of the edge is given by the net offset from v_4 to v_3, which is 1. The philosophy for this mechanism is to have a vertex to be selected by the closest tree. That is, consider v_{24} in this case. v_{24} is to be selected by v_{20} instead of v_{23}, which is great since its target vertex v_3 is closer to the root v_4 instead of the root v_1. The final forest determined by this method would be Figure 4.21 (b). We assume that this method acquires a better solution. However, The empirical result shows that it is just as good as the heuristic target net offset method.

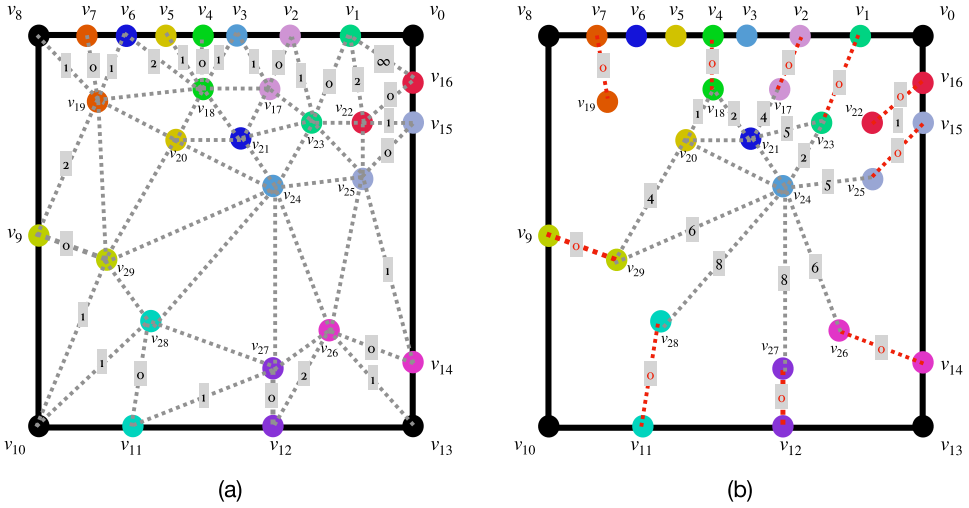

Figure 4.21: **The Illustration of the Minimum Spanning Forest of Target Net Offset from the Root.** (a) The weights of each edge are computed dynamically and annotated respectively. (b) The first ten edges chosen for the spanning forest are annotated in red. As an edge is chosen, the weights of its adjacent edges are computed.

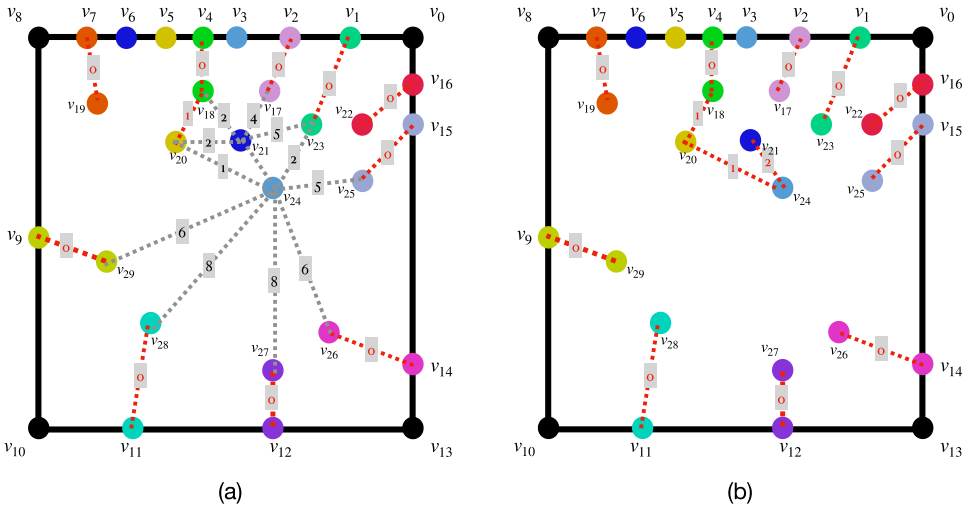

Figure 4.22: **The Illustration of the Minimum Spanning Forest of Target Net Offset from the Root.** (a) The edge $\{v_{18}, v_{20}\}$, whose edge weight is 1, has the current minimum weight. As a result, it is the eleventh chosen edge. (b) At last, the edge $\{v_{20}, v_{24}\}$ and $\{v_{20}, v_{24}\}$ are chosen. The vertex $v_2 4$ belongs to the tree of root v_4 instead of the tree of root v_1.

CHAPTER 5

Experimental Results

5.1 Experimental Setup

We implement our framework in C++ and test in an environment of a Linux-based workstation with a 2.2GHz Intel Gold 5120 processor and 128G memory. We use six modified industrial wire-bond substrate designs as our benchmark. The specification of each design is listed in Table 5.1.

At the moment, we have to pre-process those designs in order to model our current problem formulation. Assuming via-in-pad condition, to have all vias located in the area surrounded by all the fingers, we project each of the fingers to the margin of the design from the centroid of all finger positions, as illustrated in Figure 5.1. The points in orange are finger positions in the design, and the point in red is the calculated centroid of all the finger positions. We project each of the fingers from the centroid to the margin of the design annotated by the dotted line in pink. Although fingers are moved from their original positions, they still possess the original relative position ordering in a way.

5.2 Wire Length and Run Time Comparisons

We have experimented on all combinations of five styles of forest and five algorithms based on the circular frame interface. All topology classes are transformed

Table 5.1: Industrial design specifications

Design	Nets	Ball Array	Substrate
Industrial 1	59	8×8	4,500×4,500
Industrial 2	95	10×10	7,000×7,000
Industrial 3	255	16×16	17,000×17,000
Industrial 4	301	20×20	11,000×11,000
Industrial 5	394	22×22	15,000×15,000
Industrial 6	466	23×23	19,000×19,000

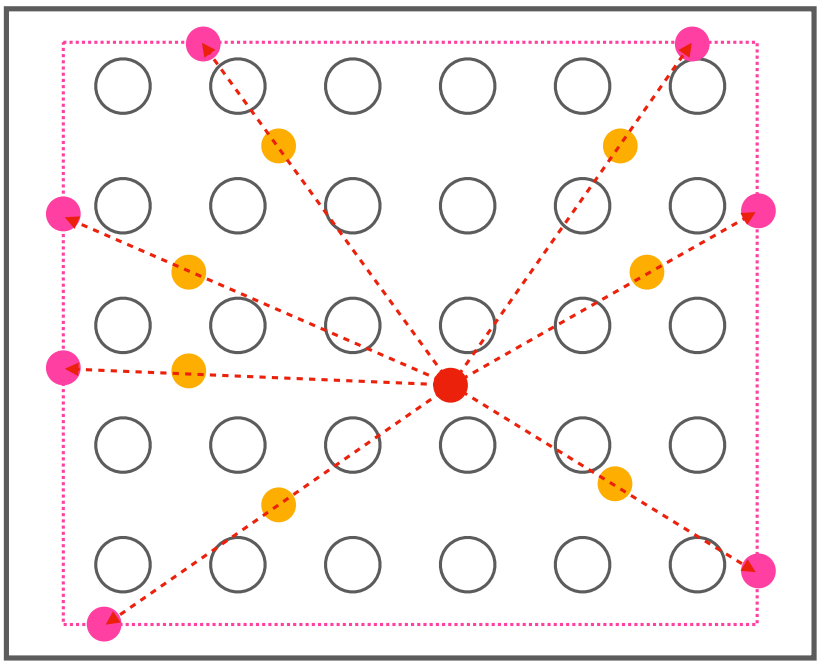

Figure 5.1: **The Illustration of Fingers Projected to the Margin of the Substrate.**

to a geometrical sketch of its class using the Triangulation Crossing Normalized Sketch methodology. Then we use the average wire length of the geometrical sketch as an evaluation criterion. We show the wire-length result for each of the cases in Table 5.2, 5.4, 5.6, 5.8, 5.10, and 5.12. NaN denotes that the topology class is too heavily tangled and thus unable to show a result. And the run-time result for each of the cases in Table 5.3, 5.5, 5.7, 5.9, 5.11, and 5.13. The first column lists all the forest styles:

- D_{Euc}: minimum spanning forest of Euclidean distance
- A* D_{Euc}: minimum spanning forest of A* heuristic Euclidean distance
- MinLevel: minimum level spanning forest
- tno: minimum spanning forest of heuristic target net offset
- tnor: minimum spanning forest of target net offset from the root

And the first row lists all the algorithms based on the circular frame interface:

- Basic: The basic circular frame routing algorithm
- TE1: Topology escape routing algorithm version 1
- TE2: Topology escape routing algorithm version 2
- TE3: Topology escape routing algorithm version 3
- TE4: Topology escape routing algorithm version 4

Although the basic circular frame routing algorithm is claimed to have a 100% topology routing completion rate, The result is too heavily tangled, and thus has the longest average wire length or even fails to show a result.

Comparison between algorithms based on minimum spanning forest of Euclidean distance shows that the topology escape routing algorithm has better results over the basic circular frame routing algorithm, even if the forest has not yet considered net ordering. And the comparison between all styles of forest used on basic circular frame routing algorithm shows that minimum spanning forest of heuristic target net offset results in better solution quality, even if the algorithm has not taken any forest topology into consideration. In the fifth row of each table, we notice that the best result is always based on the minimum spanning forest of heuristic target net offset, and the result gets better by using the algorithm from the second column to six column. Overall the topology escape routing algorithm version 4 based on a minimum spanning forest of heuristic target net offset has the best result.

We show the run-time in Table 5.3, 5.5, 5.7, 5.9, 5.11, and 5.13. The run time is extremely short. Obtaining a good result takes seconds. And notice the run time is even reducing with the increase of the resulting quality.

Finally, we show all the 25 combination experimental results in Figure 5.3, 5.5, 5.7, 5.9, 5.11, and 5.13. Notice that the two below-right sketches in columns 5 and rows 4 and 5 show the best solution quality of all the other solution quality.

5.3 Case Inspections

The essential drawback of the basic circular frame routing algorithm is that it connects topology vertices in the same slice blindly, which means a detour from all the trees. For example, the first four topology paths determined by the basic

Table 5.2: Industrial 1 Wire Length Experimental Results of All Forest Style and Routing Algorithm Combinations

	Basic	TE1	TE2	TE3	TE4
D_{Euc}	739417.77	199145.80	199145.80	45991.14	27793.72
A* D_{Euc}	501771.97	133144.45	132138.91	57305.43	22264.81
MinLevel	133085.50	34311.04	34311.04	33130.99	22176.00
Δ_t	51054.00	18697.06	17017.40	17737.94	11590.30
$\Delta_{t,r}$	104249.19	21002.05	19940.79	20496.75	11300.39

Table 5.3: Industrial 1 Run Time (microsecond) Experimental Results of All Forest Style and Routing Algorithm Combinations

	Basic	TE1	TE2	TE3	TE4
D_{Euc}	30366.50	1225.25	1163.75	778.50	648.50
A* D_{Euc}	4717.50	944.00	991.75	1454.00	704.00
MinLevel	953.00	543.75	56	888.25	1082.75
Δ_t	363.50	178.50	183.50	320.00	237.25
$\Delta_{t,r}$	430.50	265.00	269.75	408.50	234.25

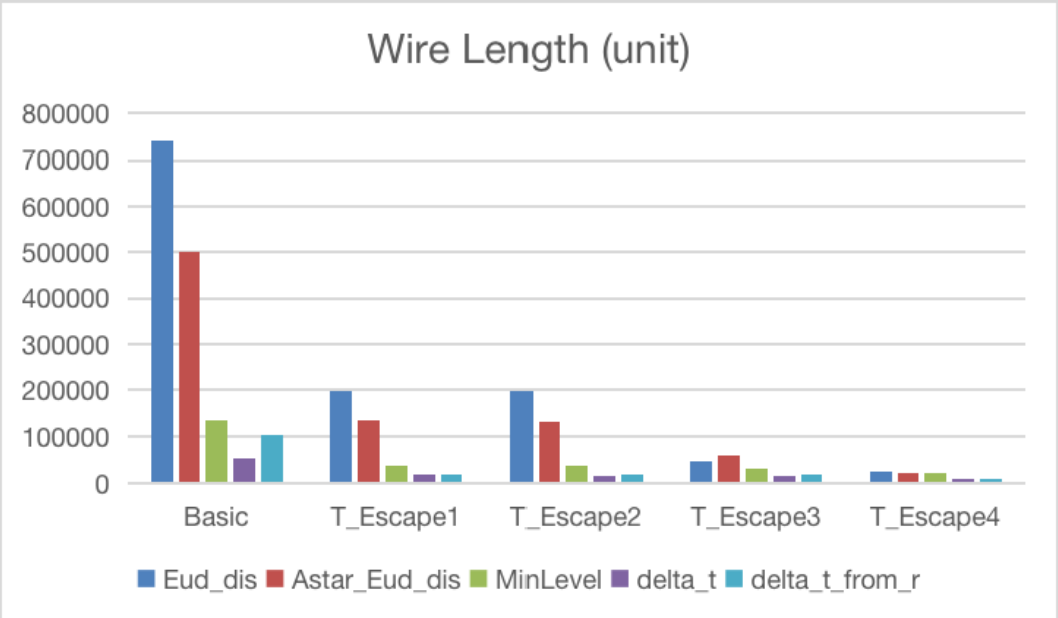

Figure 5.2: Industrial 1 Wire Length Experimental Result Summary

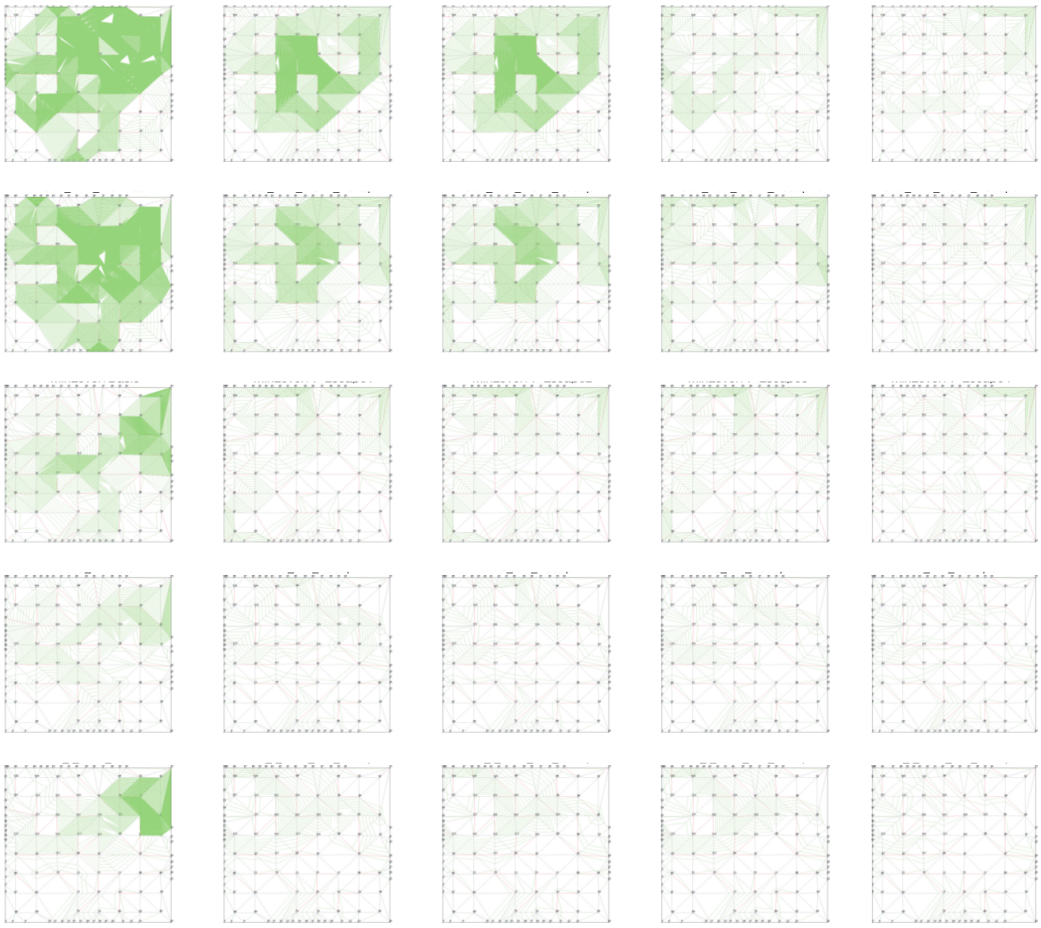

Figure 5.3: Industrial 1 Geometrical Visualization Results of All Forest Style and Routing Algorithm Combinations

Table 5.4: Industrial 2 Wire Length Experimental Results of All Forest Style and Routing Algorithm Combinations

	Basic	TE1	TE2	TE3	TE4
D_{Euc}	NaN	378502.68	378502.68	208648.85	58830.31
A* D_{Euc}	NaN	416614.35	416614.35	171103.54	46263.63
MinLevel	9924347.59	115294.64	115294.64	109767.58	34361.22
Δ_t	38794.93	17821.41	17821.41	17453.27	16296.64
$\Delta_{t,r}$	47503.16	20226.72	20226.72	15889.71	15889.71

Table 5.5: Industrial 2 Run Time (microsecond) Experimental Results of All Forest Style and Routing Algorithm Combinations

	Basic	TE1	TE2	TE3	TE4
D_{Euc}	NaN	18860.00	18931.25	17352.25	2113.75
A* D_{Euc}	NaN	25319.75	26073.25	7691.00	2786.50
MinLevel	18660884.75	6344.25	6543.00	9975.00	3385.75
Δ_t	453.75	333.50	345.75	469.50	486.50
$\Delta_{t,r}$	783.50	363.25	391.00	536.50	727.25

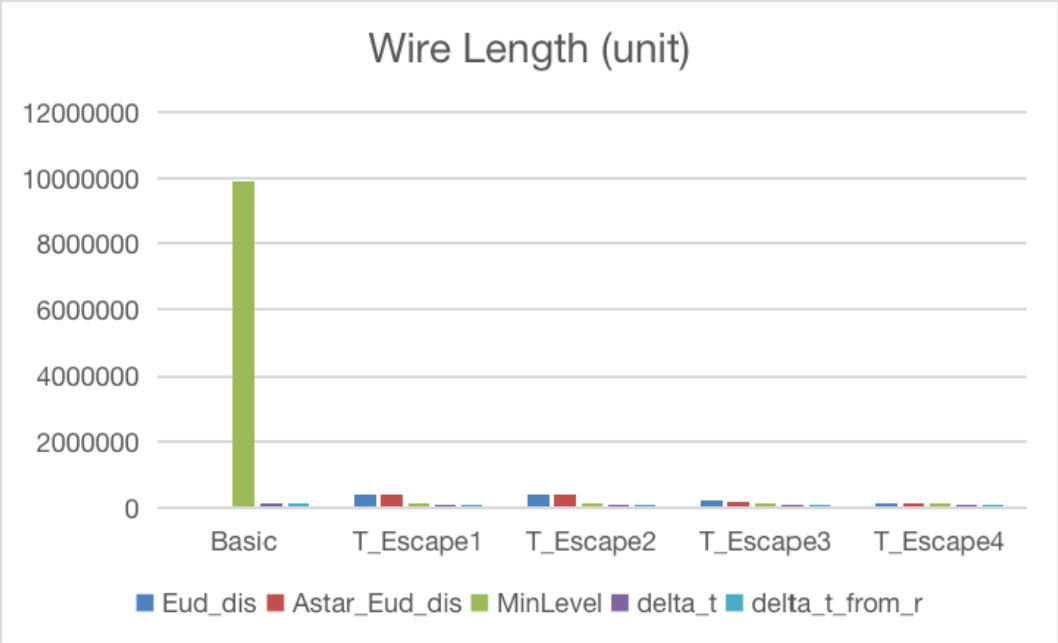

Figure 5.4: Industrial 2 Wire Length Experimental Result Summary

Table 5.6: Industrial 3 Wire Length Experimental Results of All Forest Style and Routing Algorithm Combinations

	Basic	TE1	TE2	TE3	TE4
D_{Euc}	NaN	11902782.84	11902782.84	134159.57	162215.38
A* D_{Euc}	NaN	3507856.62	3500811.24	168003.20	45905.65
MinLevel	NaN	954969.97	922070.04	353137.98	43475.56
Δ_t	859098.12	65623.41	65623.41	56301.04	49409.00
$\Delta_{t,r}$	1318357.51	116108.26	115825.40	56932.38	50164.11

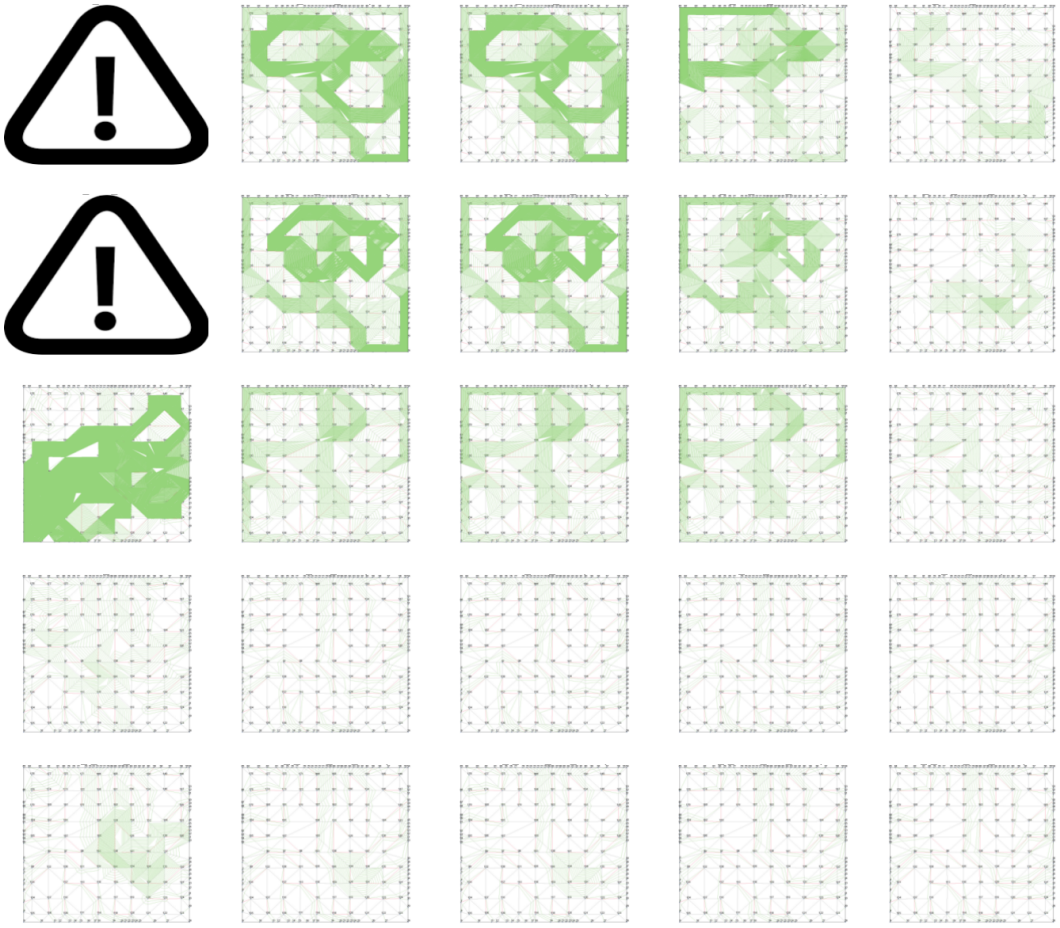

Figure 5.5: Industrial 2 Geometrical Visualization Results of All Forest Style and Routing Algorithm Combinations

Table 5.7: Industrial 3 Run Time (microsecond) Experimental Results of All Forest Style and Routing Algorithm Combinations

	Basic	TE1	TE2	TE3	TE4
D_{Euc}	NaN	20657118.50	66201933.25	77475.75	71347.50
A* D_{Euc}	NaN	3625310.50	3657535.50	301442.25	207506.25
MinLevel	NaN	861078.50	817206.75	717682.25	138122.75
Δ_t	104565.25	4924.00	5100.75	17744.50	30606.25
$\Delta_{t,r}$	626278.75	5889.75	6080.00	18250.00	27757.00

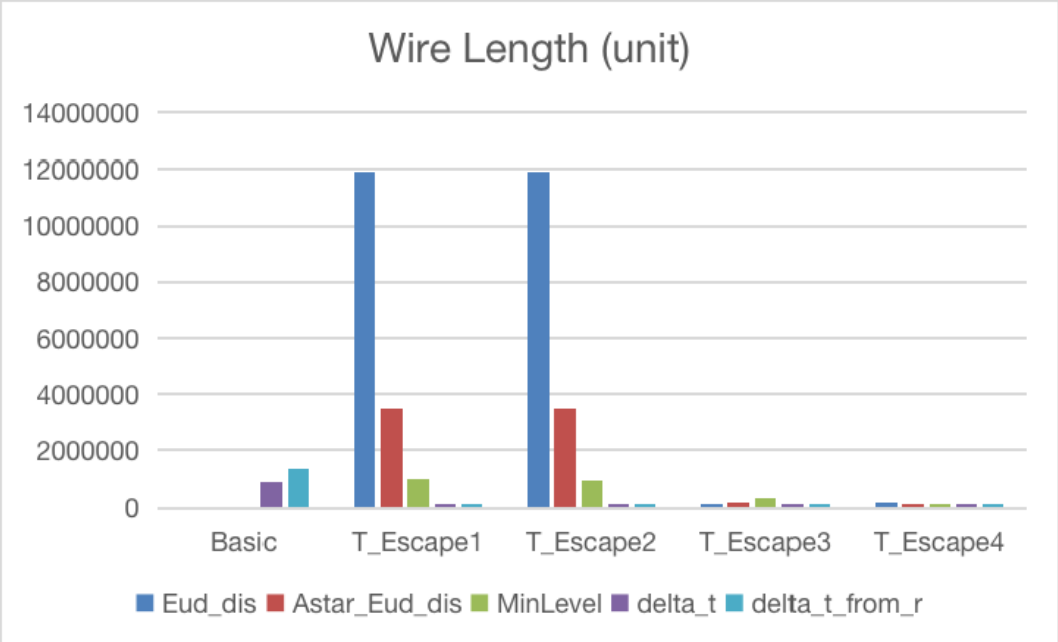

Figure 5.6: Industrial 3 Wire Length Experimental Result Summary

Table 5.8: Industrial 4 Wire Length Experimental Results of All Forest Style and Routing Algorithm Combinations

	Basic	TE1	TE2	TE3	TE4
D_{Euc}	168907.04	35320.69	34902.54	17800.36	12227.14
A* D_{Euc}	83375.94	28681.46	22948.38	15820.75	11853.18
MinLevel	483751.69	24570.81	23928.93	23693.42	13287.21
Δ_t	31477.15	13328.44	13060.13	13060.13	11908.37
$\Delta_{t,r}$	24386.14	16899.74	16141.02	14525.62	11981.26

Table 5.9: Industrial 4 Run Time (microsecond) Experimental Results of All Forest Style and Routing Algorithm Combinations

	Basic	TE1	TE2	TE3	TE4
D_{Euc}	18780.75	1848.00	1860.00	6712.25	15285.50
A* D_{Euc}	6055.25	1962.75	1910.75	6408.25	14052.25
MinLevel	96750.50	1953.25	1875.00	10902.75	25502.25
Δ_t	1261.25	892.50	934.50	1427.75	1656.75
$\Delta_{t,r}$	1128.25	1098.75	1193.00	1702.75	2558.50

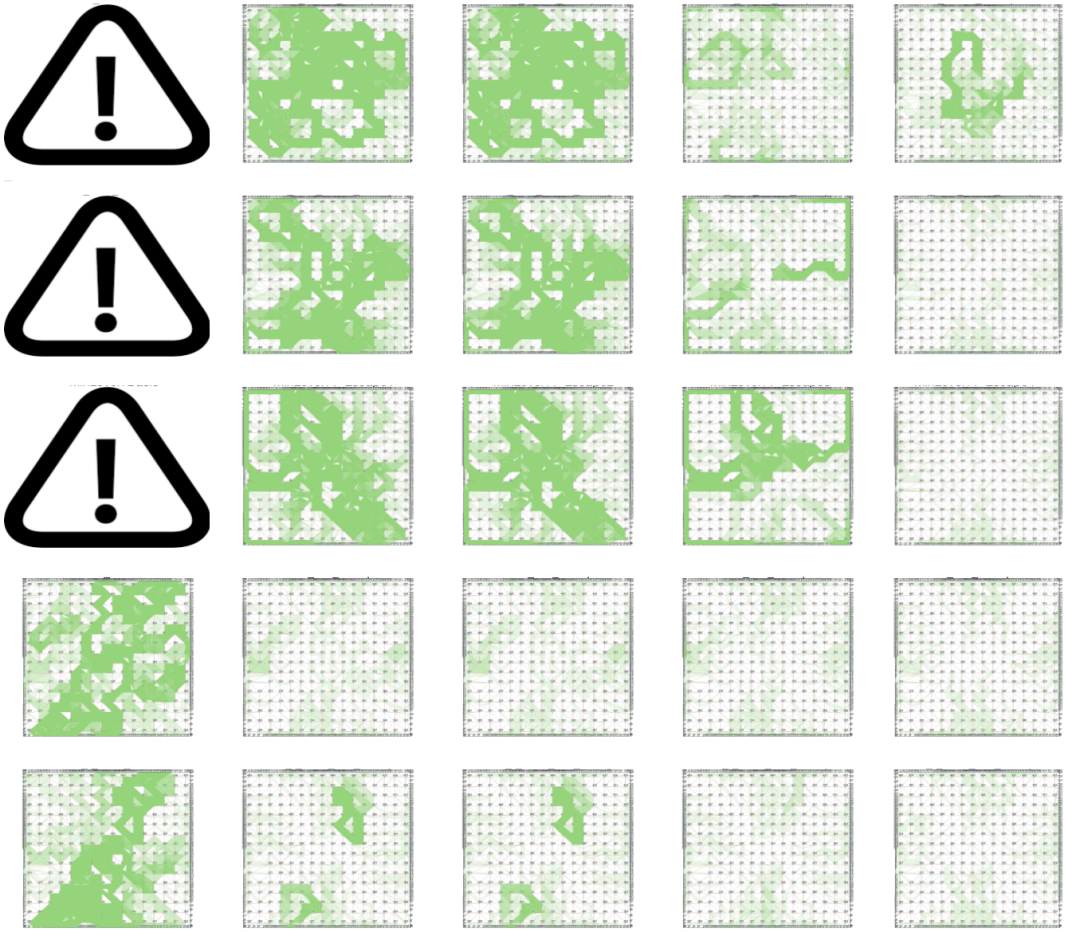

Figure 5.7: Industrial 3 Geometrical Visualization Results of All Forest Style and Routing Algorithm Combinations

Table 5.10: Industrial 5 Wire Length Experimental Results of All Forest Style and Routing Algorithm Combinations

	Basic	TE1	TE2	TE3	TE4
D_{Euc}	NaN	1093698.94	1090897.96	85843.31	62185.62
A* D_{Euc}	NaN	1212234.94	1212234.94	81422.46	69451.07
MinLevel	NaN	659571.26	652508.48	93610.61	56625.23
Δ_t	5338905.68	842853.64	840155.72	90927.20	51407.28
$\Delta_{t,r}$	699564.80	432404.26	415443.89	62212.91	55244.36

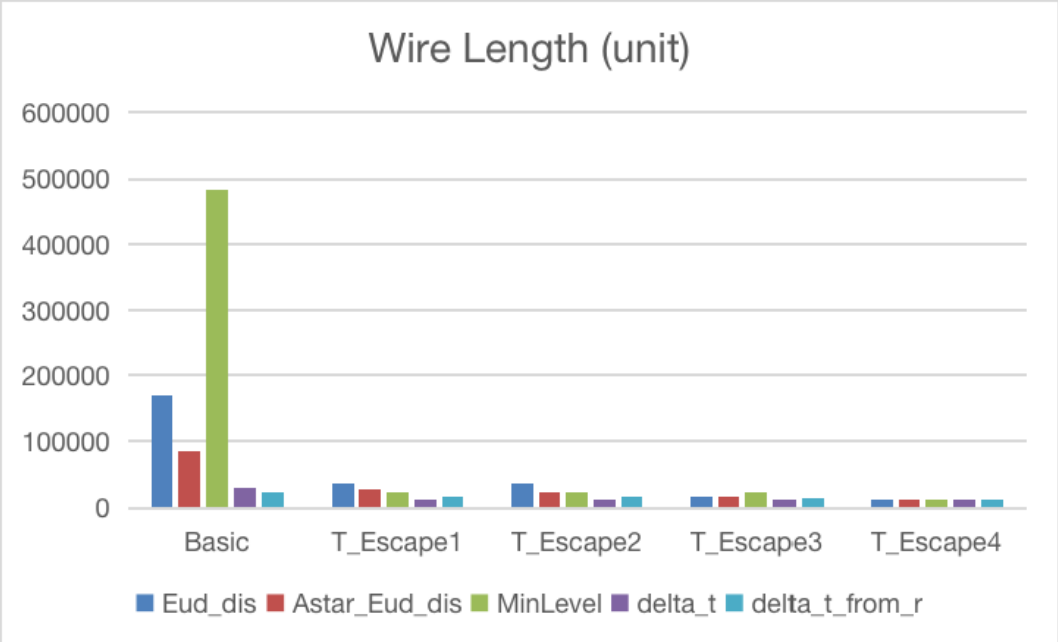

Figure 5.8: Industrial 4 Wire Length Experimental Result Summary

Table 5.11: Industrial 5 Run Time (microsecond) Experimental Results of All Forest Style and Routing Algorithm Combinations

	Basic	TE1	TE2	TE3	TE4
D_{Euc}	NaN	315878.75	317202.00	161746.75	181231.00
A* D_{Euc}	NaN	313275.25	314220.00	186059.75	201104.50
MinLevel	NaN	332656.00	328019.25	244388.50	360938.00
Δ_t	20235207.00	84655.50	86864.25	43843.75	49474.25
$\Delta_{t,r}$	457103.75	54220.75	54186.00	43798.75	62671.75

Table 5.12: Industrial 6 Wire Length Experimental Results of All Forest Style and Routing Algorithm Combinations

	Basic	TE1	TE2	TE3	TE4
D_{Euc}	NaN	NaN	NaN	281280.51	250427.52
A* D_{Euc}	NaN	NaN	NaN	230606.46	609509.98
MinLevel	NaN	NaN	NaN	345383.79	249112.62
Δ_t	NaN	1309387.75	1309387.75	257466.42	214843.95
$\Delta_{t,r}$	NaN	895106.68	889400.47	181413.17	249906.39

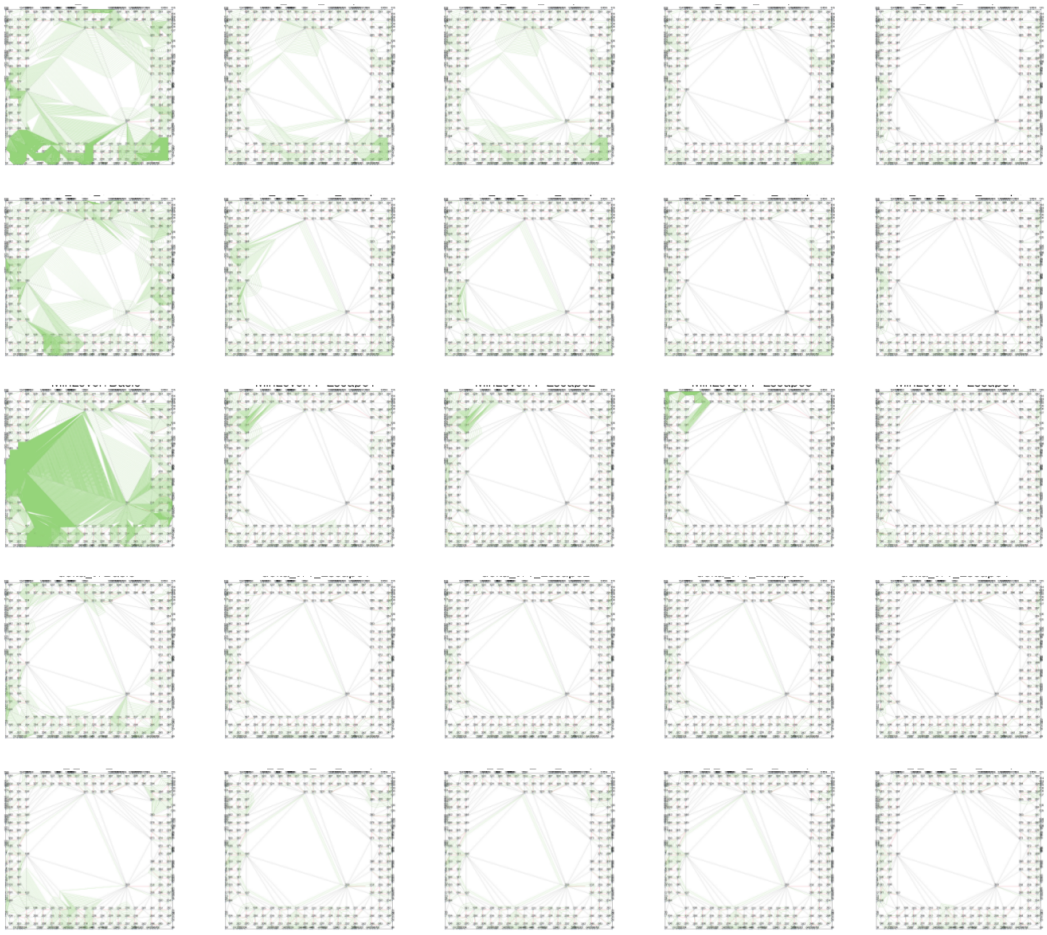

Figure 5.9: Industrial 4 Geometrical Visualization Results of All Forest Style and Routing Algorithm Combinations

Table 5.13: Industrial 6 Run Time (microsecond) Experimental Results of All Forest Style and Routing Algorithm Combinations

	Basic	TE1	TE2	TE3	TE4
D_{Euc}	NaN	NaN	NaN	304853.00	793113.00
A* D_{Euc}	NaN	NaN	NaN	353490.75	10012669.50
MinLevel	NaN	NaN	NaN	1210946.25	3407917.75
Δ_t	NaN	489633.25	485607.25	133152.75	389724.50
$\Delta_{t,r}$	NaN	548395.00	564750.75	125310.25	377942.00

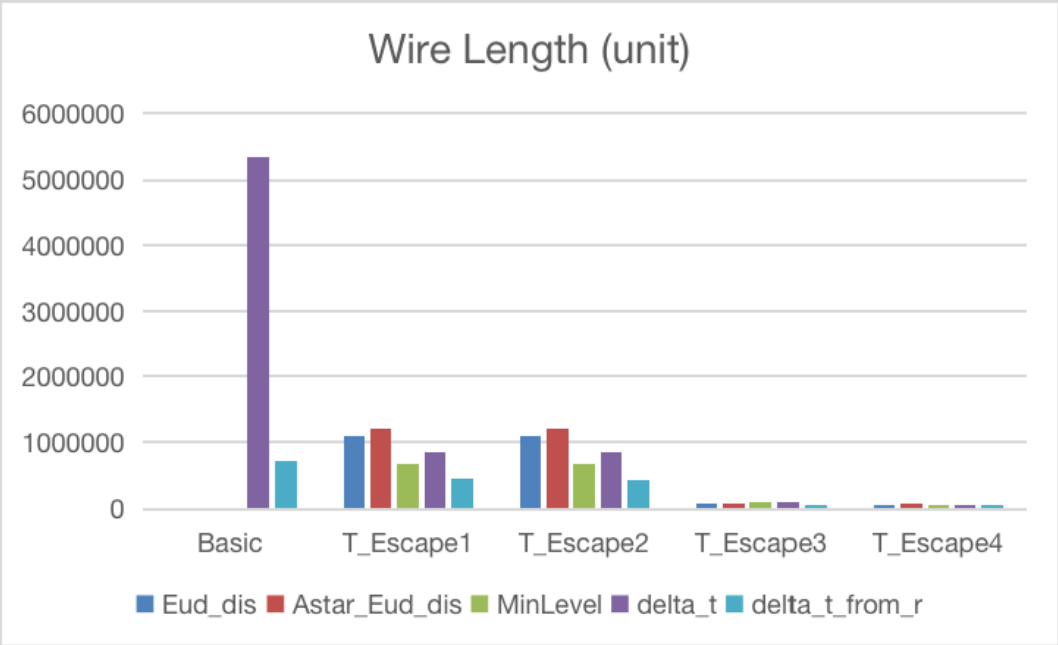

Figure 5.10: Industrial 5 Wire Length Experimental Result Summary

circular frame routing algorithm of industrial one are shown in Figure 5.14. The first path starts from graph vertex v_{61} to graph vertex v_0, the second path starts from graph vertex v_{62} to graph vertex v_1, the third path starts from graph vertex v_{63} to graph vertex v_2, and the fourth path starts from graph vertex v_{64} to graph vertex v_3 results in the tangle problem. The first path separates the graph vertex v_3, into a different slice from the graph vertex v_{64}. Once v_{64} has chosen to pass through the graph edge (v_{54}, v_{111}), the path takes a detour to avoid passing any additional graph edge.

In comparison, Figure 5.15 shows the first twenty-eight paths obtained from the first version of the topology escape routing algorithm. We can see that no detour occurs until the path made from graph vertex v_{83} to v_{29}, which alleviates the tangle problem greatly. However, we can see that the path made from graph vertex v_{83} to v_{29} also makes the next path from graph vertex v_{89} to v_{30} forced to detour.

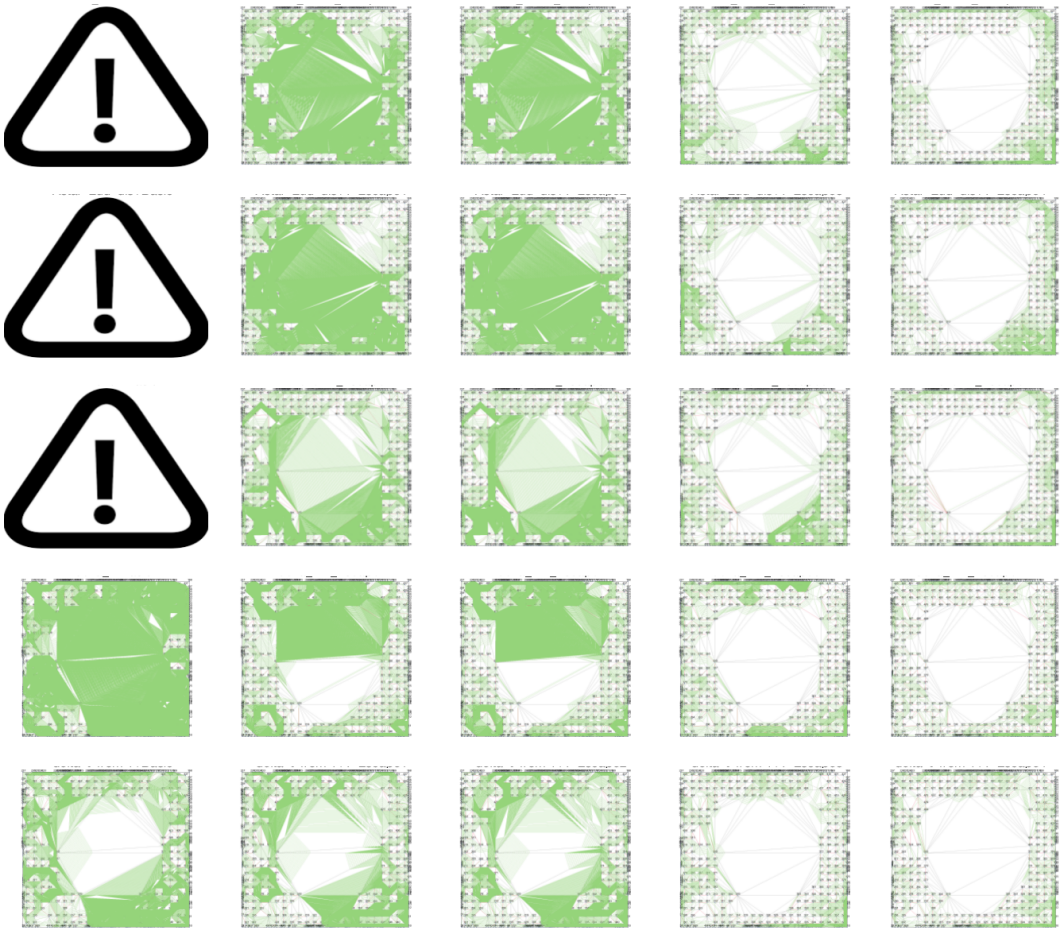

Figure 5.11: Industrial 5 Geometrical Visualization Results of All Forest Style and Routing Algorithm Combinations

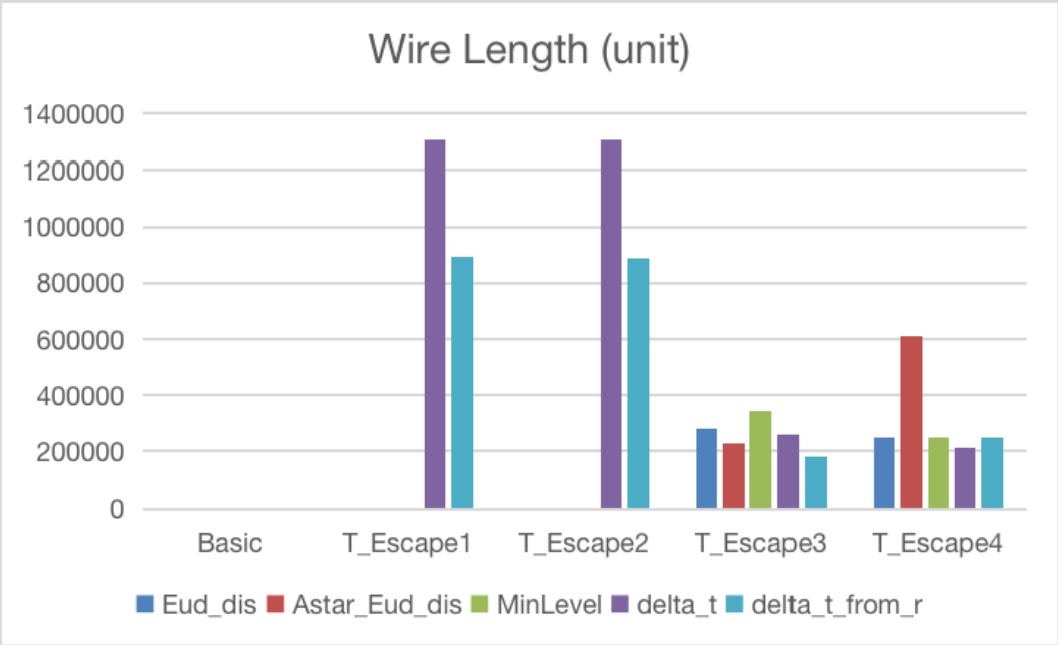

Figure 5.12: Industrial 6 Wire Length Experimental Result Summary

The second version of the topology escape routing algorithm aims to avoid such problems. The result is shown in Figure 5.16. We can see that the choice of the two paths becomes much more reasonable. Still, notice that the path from graph vertex v_{61} to v_0 should have passed through v_{64} on the different sides, which results in the detour of the path from v_{64} to v_3.

The third version of the topology escape routing algorithm fixed this blind spot. The result is shown in Figure 5.17. We can see that the graph vertex v_{61} pass through v_{64} on the different slide; however, graph vertex v_{61} and v_{64} both pass through the edge (v_{115}, v_{58}), causing a detour. This problem is then alleviated by the fourth version.

Figure 5.18, 5.19, and 5.20 shows the results of the fourth version of the topology escape routing algorithm based on the minimum spanning forest of heuristic target net offset. We can see that almost every choice is reasonable, thus pro-

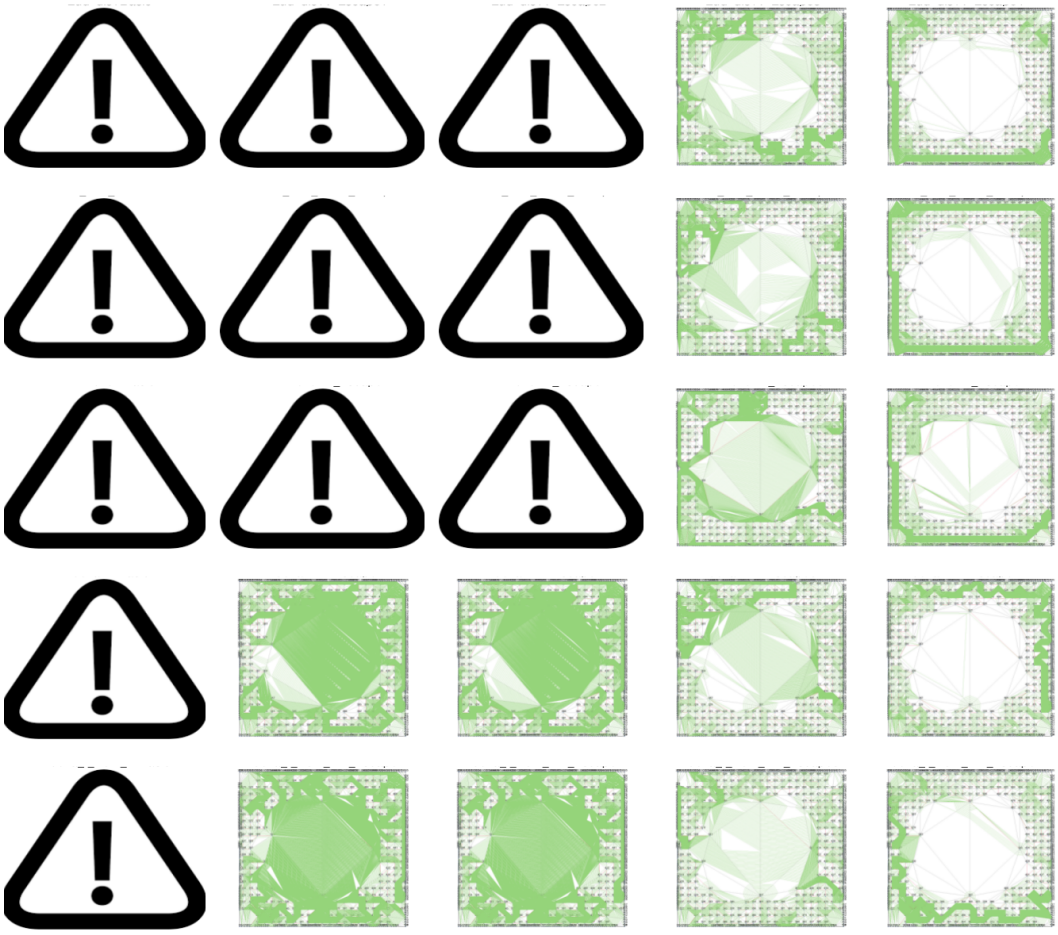

Figure 5.13: Industrial 6 Geometrical Visualization Results of All Forest Style and Routing Algorithm Combinations

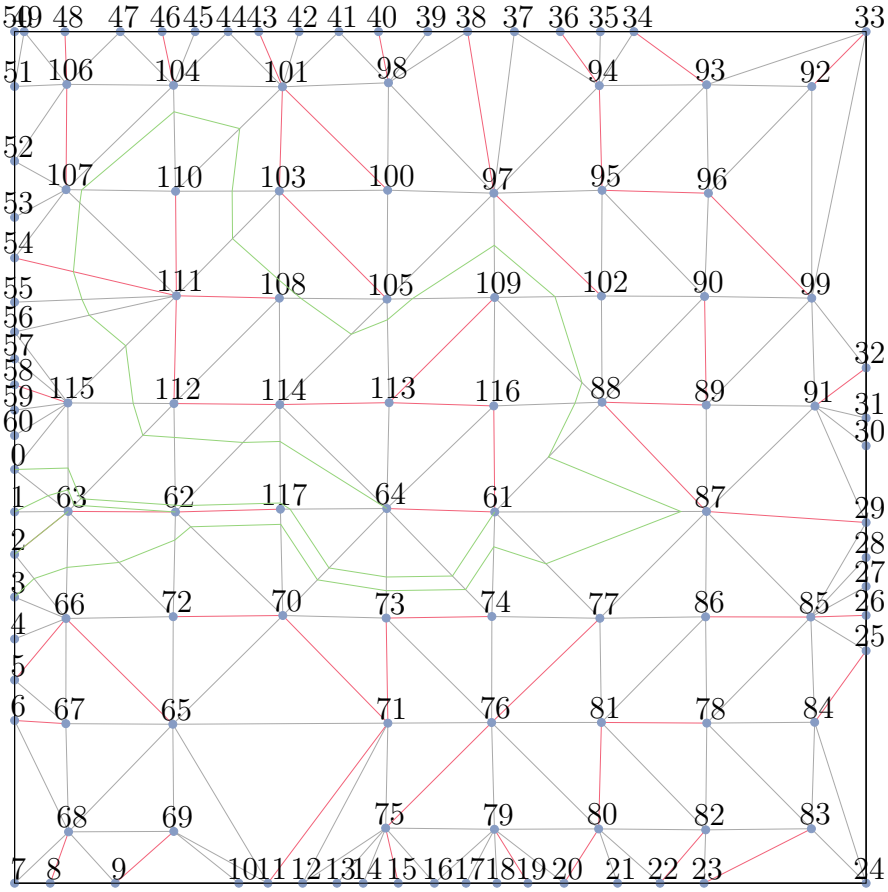

Figure 5.14: **The Industrial 1 Partial Geometrical Visualization Using the Minimum Spanning Forest of Heuristic Target Net Offset and the Basic Circular Frame Routing Algorithm.**

viding a persuasive guide for the geometrical router.

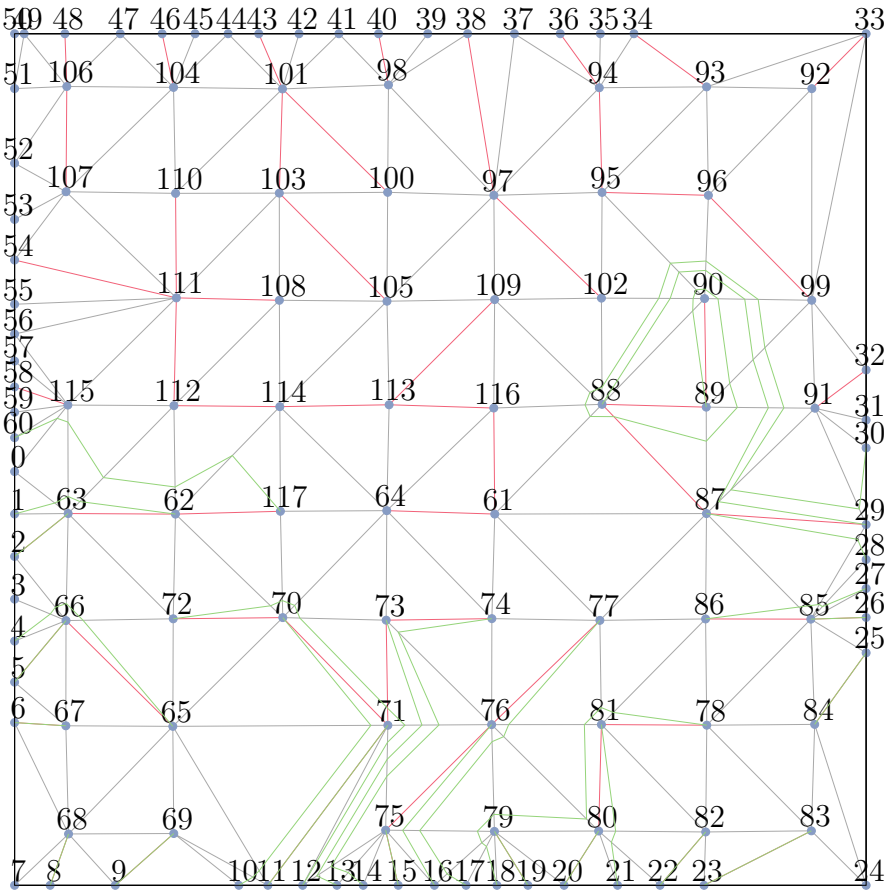

Figure 5.15: **The Industrial 1 Partial Geometrical Visualization Using the Minimum Spanning Forest of Heuristic Target Net Offset and the Topology Escape Routing Algorithm Version One.**

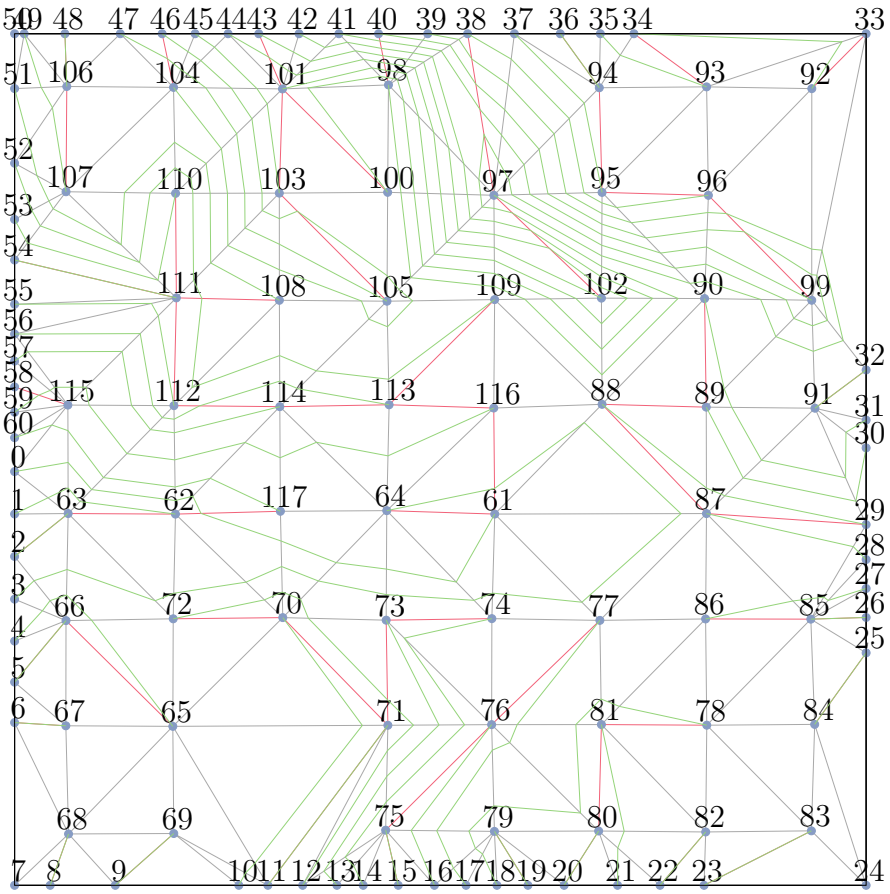

Figure 5.16: **The Industrial 1 Partial Geometrical Visualization Using the Minimum Spanning Forest of Heuristic Target Net Offset and the Topology Escape Routing Algorithm Version Two.**

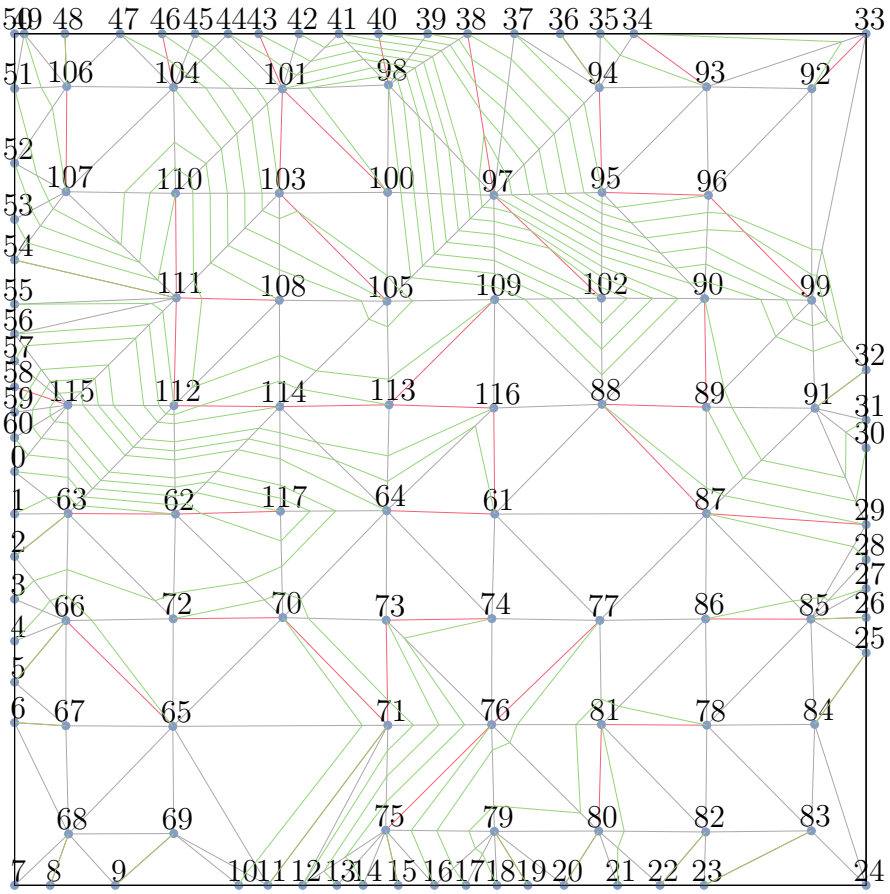

Figure 5.17: **The Industrial 1 Partial Geometrical Visualization Using the Minimum Spanning Forest of Heuristic Target Net Offset and the Topology Escape Routing Algorithm Version Three.**

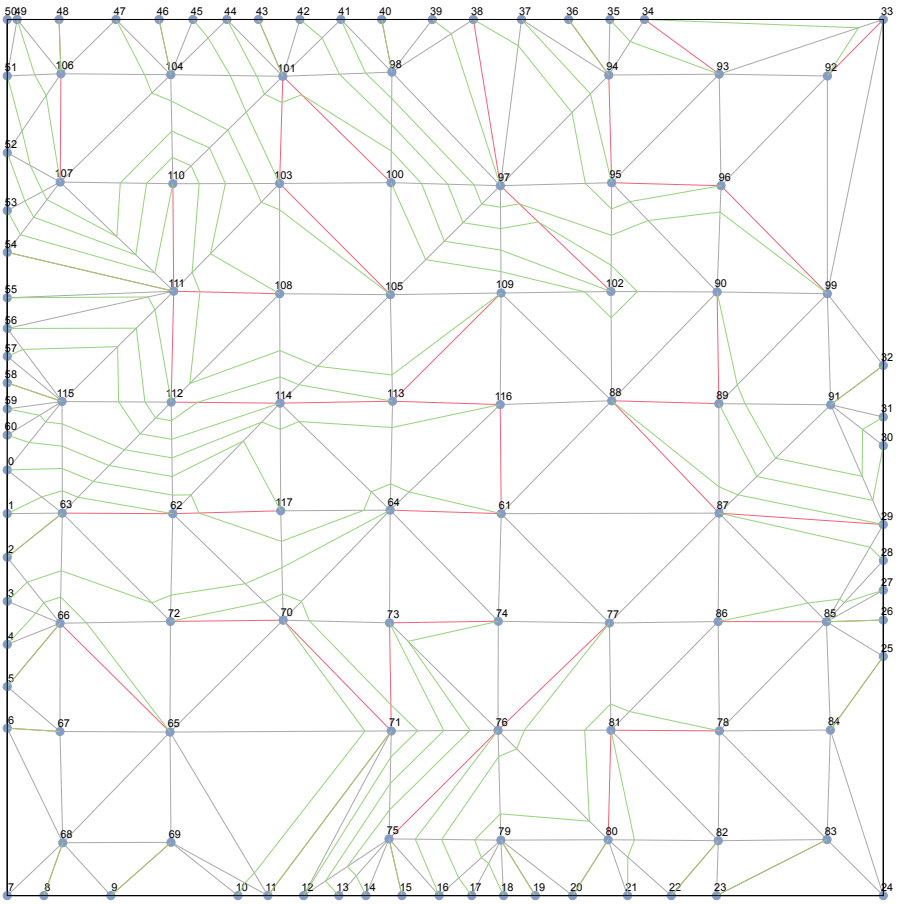

Figure 5.18: **The Industrial 1 Complete Geometrical Visualization Using the Minimum Spanning Forest of Heuristic Target Net Offset and the Topology Escape Routing Algorithm Version Four.**

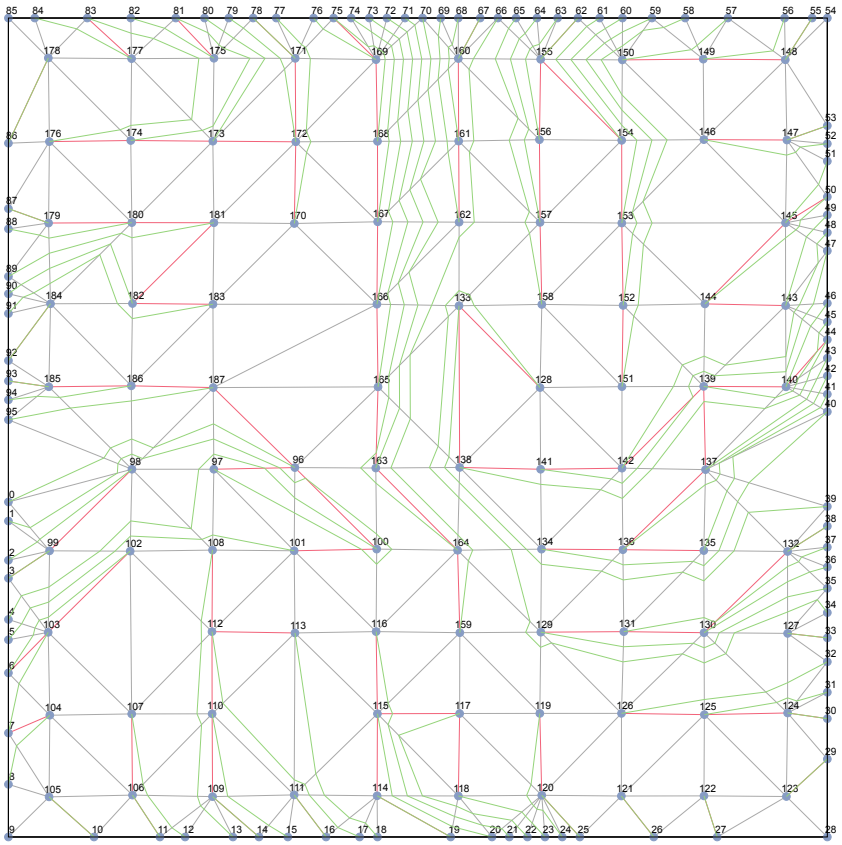

Figure 5.19: **The Industrial 2 Complete Geometrical Visualization Using the Minimum Spanning Forest of Heuristic Target Net Offset and the Topology Escape Routing Algorithm Version Four.**

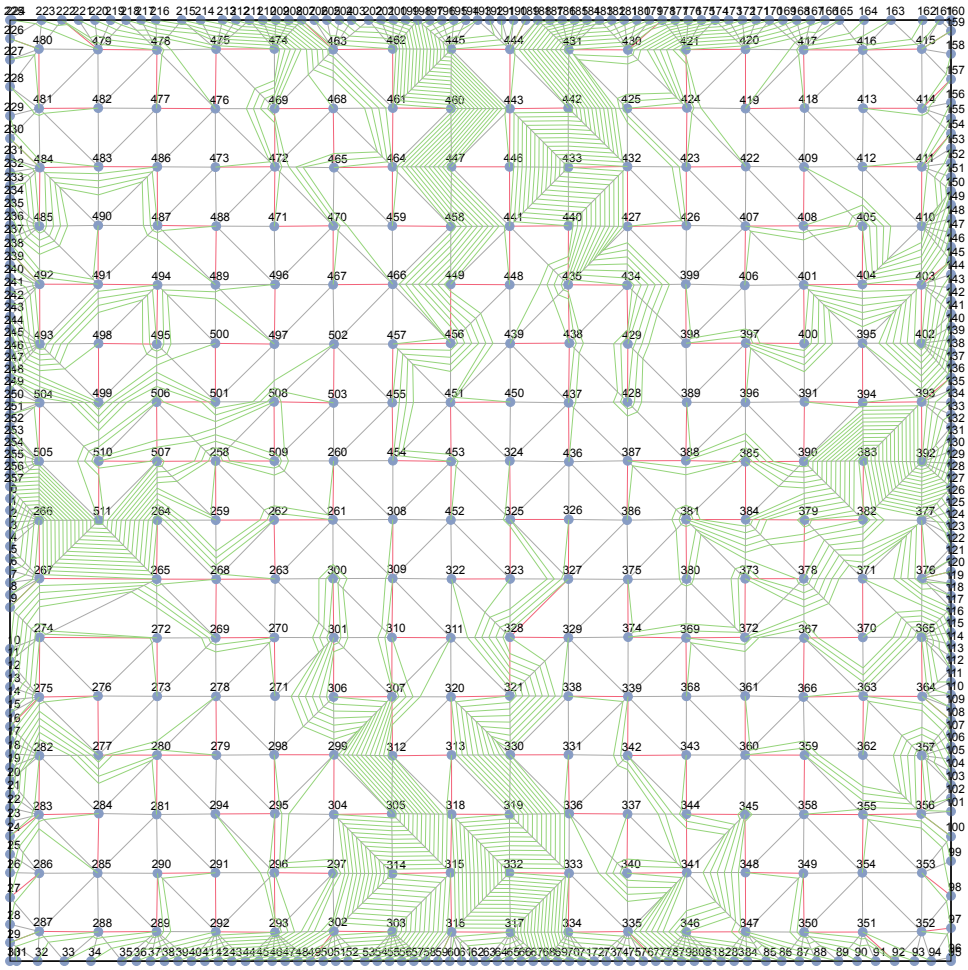

Figure 5.20: **The Industrial 3 Complete Geometrical Visualization Using the Minimum Spanning Forest of Heuristic Target Net Offset and the Topology Escape Routing Algorithm Version Four.**

CHAPTER 6

Conclusion

The basic circular frame routing algorithm guarantees the 100% routing completion rate topologically. However, it is impractical to transform such topological sketches into homotopic geometrical solutions due to the path tangling problem. To acquire a better solution quality from an algorithm based on the circular frame interface, it is necessary to take the associated embedding frame into consideration. The style of minimum spanning forest using the edge weight of heuristic target net offset cooperates greatly with the topology escape routing algorithm and generates great solution quality. Furthermore, the run-time is extremely fast thanks to the proposed triple-list data structure, which reduces the complexity of the two most used operations, make-slice and free slice operations, to $O(1)$. As a result, our methodology efficiently and effectively resolves the general escape routing problem from the topological perspective and shows great potential for practical application in substrate design planning.

Bibliography

[1] Rak-Kyeong Seong, Chanho Mina, Sang-Hoon Hanr, Jaeho Yanga, Seungwoo Nama, Kyusam Oha, "Topology and Routing Problems: The Circular Frame," *arXiv*, arXiv:2105.03386, 2021.

[2] Rak-Kyeong Seong, Jaeho Yanga, Sang-Hoon Han, "Topology for Substrate Routing in Semiconductor Package Design," *arXiv*, arXiv:2105.07892, 2021.

[3] Wayne Wei-Ming Dai, Raymond, and Jeffrey Jue, Masao Sato, "Rubber Band Routing and Dynamic Data Representation," *International Conference on Computer-Aided Design*, 1990.

[4] David Joseph Staepelaere, "Geometric Transformations for a Rubber-band Sketch," M. S. thesis, University of California Santa Cruz, 1992.

[5] Tal Dayan Wayne, "Rubber-band Based Topological Router," M. S. thesis, University of California Santa Cruz, 1992.

[6] David Staepelaere, Jeffrey Jue, Tal Dayan, WayneDai, "SURF: Rubber-band Routing System for Multichip Modules," *IEEE Design & Test of Computers*, P.18-26, 1993.

[7] K. Kawamura, T. Shindo, T. Shibuya, H. Miwatari, and others, "Touch and cross router," *IEEE International Conference on Computer-Aided Design*, 1990.

[8] H. Murata, and Y. Kajitani, "Interactive terminal sliding algorithm for hybrid IC planar layout," Transactions of Information Processing Society of Japan, Vol.35, No.23, pp.2806-2815, 1994.