© 2010 Hui Kong

NEW STRATEGIES FOR PCB ROUTING

BY

HUI KONG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:

Professor Martin D. F. Wong, Chair
Associate Professor Sanjay Jeram Patel
Assistant Professor Deming Chen
Muhammet Mustafa Ozdal, Ph.D., Intel Corporation

ABSTRACT

As IC technology advances rapidly, the dimensions of packages and PCBs are decreasing while the pin counts and routing layers keep increasing. Today, a high-performance PCB usually contains thousands of pins and more than ten signal layers. Moreover, the manufacturing constraints require all nets to be routed in the planar fashion and the designer requires nets in the same bus to be routed together without any other net. All these factors pose new challenges for the PCB routing problem, making the PCB routing so difficult that no commercial CAD software can provide an automatic solution. Today, all high-end circuit boards are routed manually, in a time-consuming manner. In this dissertation, we present new strategies for automatic PCB routing. In particular, we present novel algorithms for bus sequencing, pin assignment, bus planning, bus escape, and escape routing.

This dissertation is dedicated to my wife, my parents and my parents-in-law and my two lovely kids, for their love and support. I definitely did not have enough time for them due to my study for the Ph.D.

ACKNOWLEDGMENTS

I am heartily thankful to my advisor, Professor Martin D.F. Wong, whose patience, encouragement, guidance and support throughout my study enabled me to complete this research work.

I would also like to thank the members of my dissertation committee, Professor Sanjay Jeram Patel, Professor Deming Chen and Doctor Muhammet Mustafa Ozdal, for their constructive comments.

In addition, I would like to thank all members of the VLSI CAD group at the University of Illinois at Urbana-Champaign. In particular, many thanks to Liang Deng, Lei Cheng, Tan Yan, Lijuan Luo, Hongbo Zhang, Qiang Ma and Pei-Ci Wu, for their support and help during my study and research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A typical printed circuit board (PCB) contains several components such as MCMs (multichip modules), memory, and I/O modules, as shown in Figure 1.1. These components are either mounted on or plugged into the board such that each component pin is accessible from every layer of the board. As IC technology advances rapidly, the dimensions of packages and PCBs are decreasing while the pin counts and routing layers keep increasing. Today, a high-end PCB usually contains thousands of pins and more than ten signal layers [1–3]. For instance, we have worked on several state-of-the-art industrial boards, which have been fully routed in manual efforts. One contains more than 7000 nets and 12 signal layers and another contains more than 10000 nets and 14 signal layers. Therefore, the current high-end PCBs have the following characteristics. On one hand, due to the high-density fine-pitch packages used, the available routing resources in the areas beneath components are extremely limited. Furthermore, there are large numbers of nets that need to escape from dense pin arrays to the corresponding component boundaries. For example, an MCM of the IBM z9 enterprise class server [4] (introduced in 2007) consists of sixteen chips, including eight dual-core CPUs, four 10-MB cache chips, one system control chip, two memory storage controllers and one clock chip. This MCM is only 95 mm $\times$ 95 mm in area, but there are 2970 signal I/O pins on its bottom. On the other hand, there are relatively more routing resources outside components since there are few blockages between components. According to these characteristics, the PCB routing is decomposed into two subproblems: the *escape routing*, which is to route nets from pins to their corresponding component boundaries, and the *area routing*, which is to route nets between component boundaries. The zoomed-in part of Figure 1.2 demonstrates the escape routing solution and the area routing solution of nets in a bus.

Due to manufacturing constraints, buried vias are not allowed in some board designs.

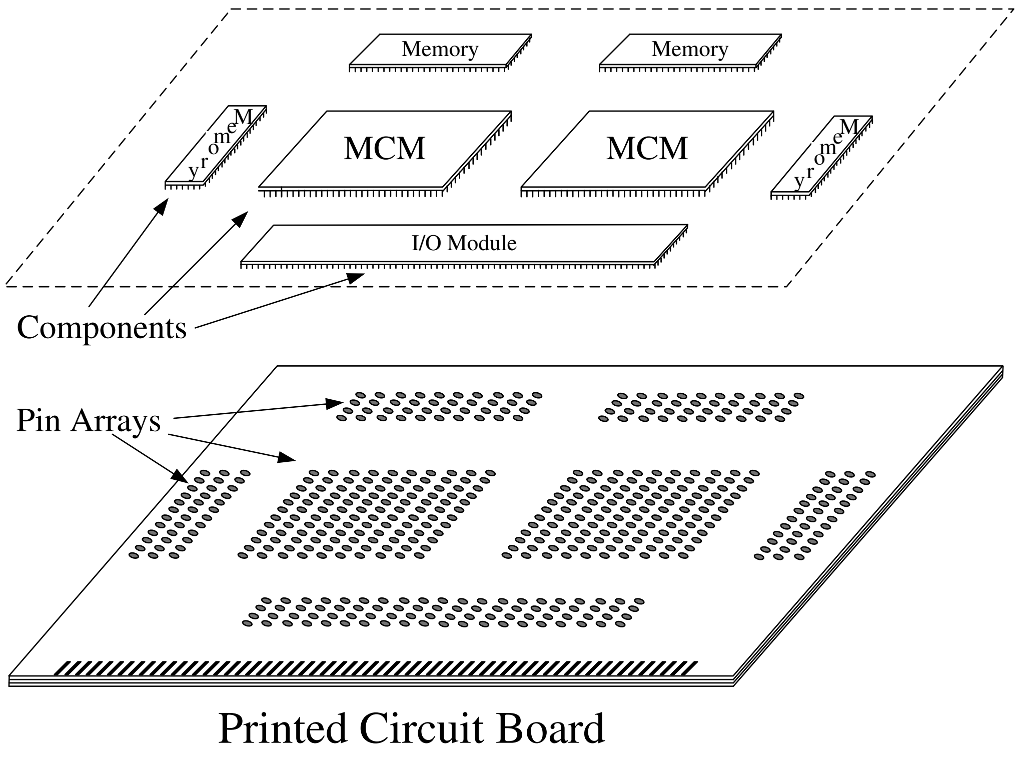


Figure 1.1: A PCB with five components. Each component corresponds to a pin array on the PCB.

In some other designs, buried vias are permitted, but they need to be kept at minimum, since vias have adverse effects on performance and signal integrity characteristics of nets, and they lower the manufacturing yields. Moreover, the triplate layer structure (i.e only one signal layer between power/GND layers) used in the recent IBM server boards [5, 6] eliminates the vertical coupling problems among signal wires, and allows full utilization of routing resources on each signal layer. In other words, we do not need to route signal nets on x-y layer pairs anymore. So, routing nets in a planar fashion on every layer becomes very important, both in terms of minimization (or complete avoidance) of buried vias, and effective utilization of routing resources on every layer. However, planar routing is a different problem, especially when all source-destination pin pairs are fixed. Furthermore, the scarcity of routing resources inside dense components, and the large number of nets (on the order of tens of thousands) make this routing problem even harder. In addition, several other design constraints need to be enforced during routing due to performance and manufacturing related issues (e.g. min/max length constraints, adjacency/separation constraints, differential pairs,

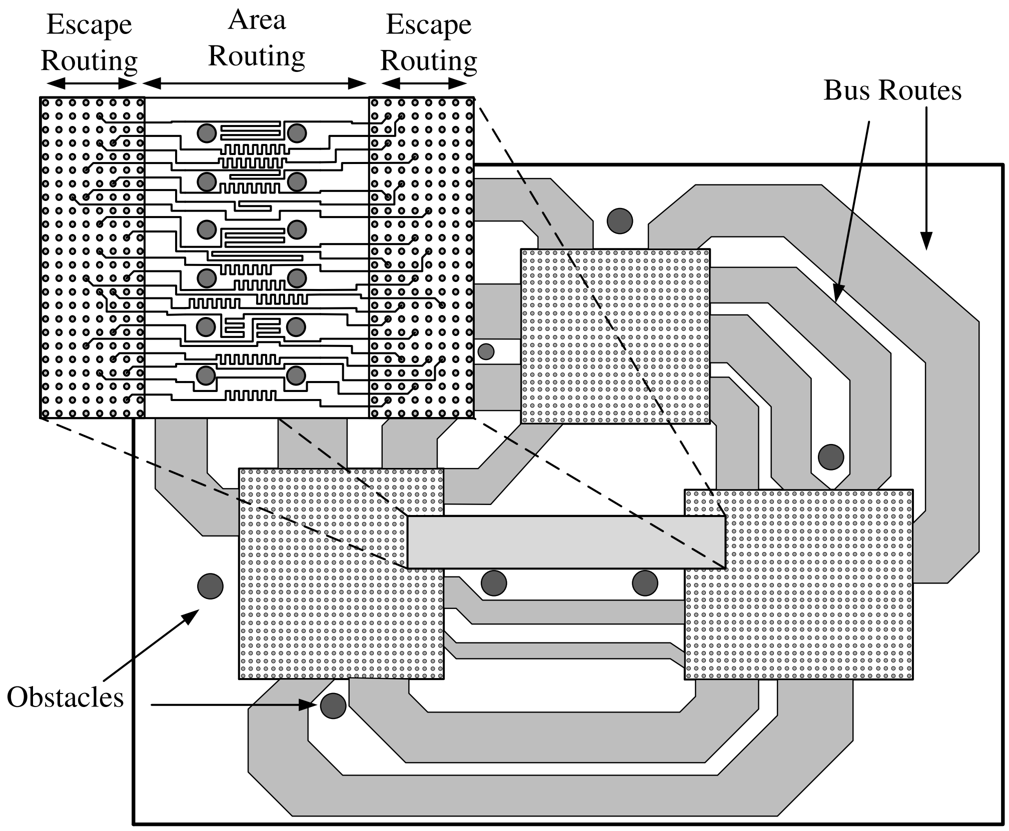


Figure 1.2: A PCB routing solution with bus structures. The zoom-in part shows the escape routing solution and the area routing solution of nets in a bus.

etc.). Whenever the design constraints permit, designers prefer the nets in a PCB to be bundled together as buses. All the nets in a bus are expected to be routed together. In some high-end boards, more than 90% of signal nets belong to buses. Figure 1.2 gives a designer-preferred PCB routing solution, where the bus structure is honored.

All the above factors pose new challenges for the board routing problem, making the PCB routing so difficult that no commercial CAD software can provide an automatic solution. Today, all high-end circuit boards are routed manually, in a time-consuming manner. In this dissertation, we will propose some new routing strategies to handle these challenges.

In Chapter 2, we will propose an optimal bus sequencing algorithm for escape routing. As mentioned above, nets have bus structures and nets in a bus are preferred to be routed together without foreigners. But now all existing algorithms for escape routing are net-centric and directly applying these algorithms will result in mixing nets of different buses together. Thus the bus-centric escape routing problem can be naturally divided into two subproblems: (1) finding a subset of buses that can be routed on the same layer without net mixings and crossings, which we refer to as the *bus sequencing problem*, and (2) finding the escape routing solutions for each chosen bus, which can be solved by a net-centric escape router. In this chapter, we solve the bus sequencing problem. We introduce a new optimization problem called the Longest Common Interval Sequence (LCIS) problem and model the bus sequencing problem as an LCIS problem. By using dynamic programming and balanced search tree data structure, we present an LCIS algorithm which can find an optimal solution in $O(n \log n)$ time. We also show that $O(n \log n)$ is a lower-bound for this problem and thus the time complexity of our algorithm is also the best possible.

In Chapter 3, we will propose some simultaneous pin assignment and escape routing approaches. In PCB designs, pin positions greatly affect routability of the design. State-of-the-art pin assignment algorithms are guided by simple (heuristic) metrics to estimate routability and thus have no guarantee to obtain a routable solution. In this chapter, we present a novel approach to obtain a pin assignment solution that guarantees routability. We show that the problem of simultaneous pin assignment and escape routing can be solved optimally in polynomial time. We then focus on the pin assignment and escape routing for the terminals in a bus, and present algorithmic enhancements as well as discuss the trade-offs between single-layer and multi-layer implementations. We tested our approach on a state-of-the-art industrial board with 80 buses (over 7000 nets). The pin assignment and escape routing solutions for all the 80 buses are successfully obtained in less than 5 minutes of CPU time.

In Chapter 4, we will propose an automatic bus planner. Bus planning is one of the most time-consuming steps of PCB routing. It consists of assigning buses to multiple layers of the PCB and routing them in a planar fashion on each layer. Routing congestion between on-board components and the min-max length bounds of the buses must also be considered during routing. In this chapter, we present the first automatic bus planner. We tested our system on a state-of-the-art industrial circuit board with over 7000 nets and 12 signal layers. All the nets on this board were already manually routed. Our bus planner is able to achieve 100% routing completion using the layer assignment extracted from manual design. For simultaneous layer assignment and bus routing, we are able to successfully route 98.5% of the nets. The remaining 1.5% can be routed either manually or by using vias. The runtime of our bus planner is less than 3 hours on a 3 GHz workstation.

In Chapter 5, we will propose an optimal *maximum disjoint subset* (MDS) algorithm to escape the maximum number of buses from one component. The MDS of rectangles is a subset of non-overlapping rectangles with the maximum total weight. The problem of finding the MDS of general rectangles has been proved to be NP-complete in [7]. In this chapter, we focus on the problem of finding the MDS of boundary rectangles, which is an open problem and is closely related to some difficult problems in PCB routing. We propose a polynomial time algorithm to optimally solve the MDS problem of boundary rectangles. Then we show this algorithm can be applied to find the optimal solution of the bus escape routing problem.

In Chapter 6, we will propose a wire packing based escape routing algorithm, which can provide a simultaneous L-shape escape routing solution in a very short time. All existing escape routing algorithms which can provide comparable escape routing solutions are slower than this algorithm. In this chapter, we will also introduce the applications of this L-shape escape routing algorithm in the whole PCB routing process.

CHAPTER 2

BUS SEQUENCING

2.1 Introduction

As was pointed out in Chapter 1, today the printed circuit board (PCB) routing problem becomes more and more challenging such that it cannot be solved fully automatically. It is decomposed into two separate problems: (1) routing nets from pin terminals to component (MCM, memory, etc.) boundaries, which is called *escape routing* (see the solid line in Figure 2.1), and (2) routing nets between component boundaries, which is called *area routing* (see the dashed line in Figure 2.1). In this chapter, we only discuss the escape routing problem for a single layer.

Previous escape routing algorithms [8, 9] are net-centric. However, in practice, as shown in industrial manual routing solutions, nets are usually organized in bus structures, and nets in a bus are expected to be routed together without foreign wires in between. Obviously, directly applying the net-centric algorithms in [8, 9] to all the buses will result in mixing nets of different buses together as shown in Figure 2.1. Thus, the escape routing problem is bus-centric and can be defined as finding the maximum number of nets, such that (1) the routed nets in a bus are bundled together without foreign wire (i.e., the buses are not mixed up), and (2) the nets escaped from the components do not lead to any crossing between component boundaries. Therefore, the bus-centric escape routing problem can naturally be divided into two subproblems: (1) finding a subset of buses that can be routed on the same layer without net mixings and crossings, which we refer to as the *bus sequencing problem*, and (2) finding the escape routing solution for each chosen bus, which can be solved by

[1]This work has been published as "Optimal Bus Sequencing for Escape Routing in Dense PCBs," pages 390-395, Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2007).

existing net-centric escape routing algorithms.

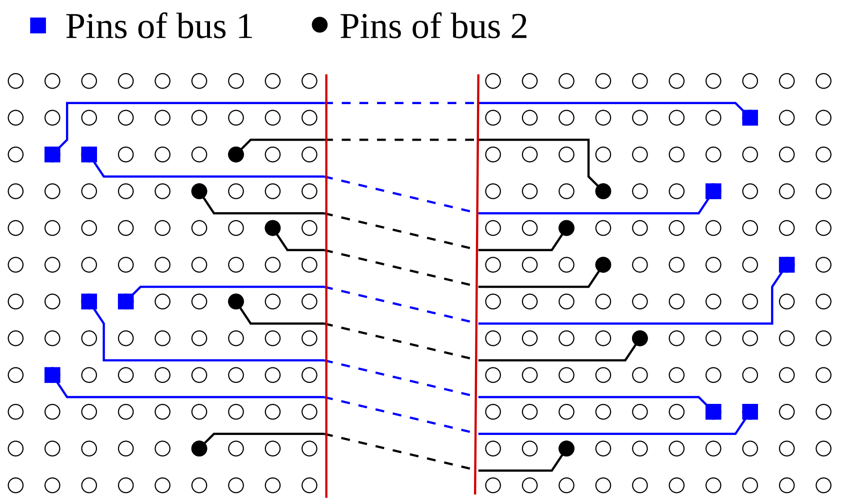


Figure 2.1: A sample net-centric escape routing solution for a problem with two buses. Nets of these two buses are mixed up.

In this chapter, we solve the bus sequencing problem. We introduce a new optimization problem called the Longest Common Interval Sequence (LCIS) problem and formulate the bus sequencing problem as an LCIS problem. The LCIS problem is a generalized version of the well-known longest common subsequence (LCS) problem. But traditional LCS algorithms cannot be directly applied to solve it. We propose an LCIS algorithm based on dynamic programming and balanced binary search tree (BST) data structure. Our algorithm guarantees finding an optimal solution in $O(n \log n)$ time. We also show that $O(n \log n)$ is a lower bound and thus the time complexity of our algorithm is also the best possible.

The rest of this chapter is organized as follows: in Section 2.2, we formulate the bus sequencing problem as the LCIS problem. Then we present our algorithm and its time complexity analysis in Section 2.3. Section 2.4 demonstrates some experimental results to compare the escape routing results with and without using the LCIS algorithm. Finally, concluding remarks are given in Section 2.5.

2.2 Problem Formulation

A typical circuit board contains several chip components such as MCM, memory, and I/O module. These components are mounted on or plugged into the board, making a number of dense pin arrays on the board. Let a component be defined as a 2-D array of pins. The input to the escape routing problem is assumed to contain two components separated by a channel. A bus is a group of 2-pin nets, and it has a pin cluster in each component. The escape route for a given net is defined as the route from its terminal pins (within components) to the respective component boundaries. Then the bus sequencing problem for the escape routing is to find a subset of buses, such that: (1) the sum of the number of nets in the chosen buses is maximum, (2) the net escape routes belonging to different buses will not be mixed up, and (3) the net escape routes of different buses will not have crossings in the intermediate channel. For simplicity of presentation, we focus on a horizontal problem, where one component is to the right of the other. It is straightforward to extend the algorithm to a vertical problem.

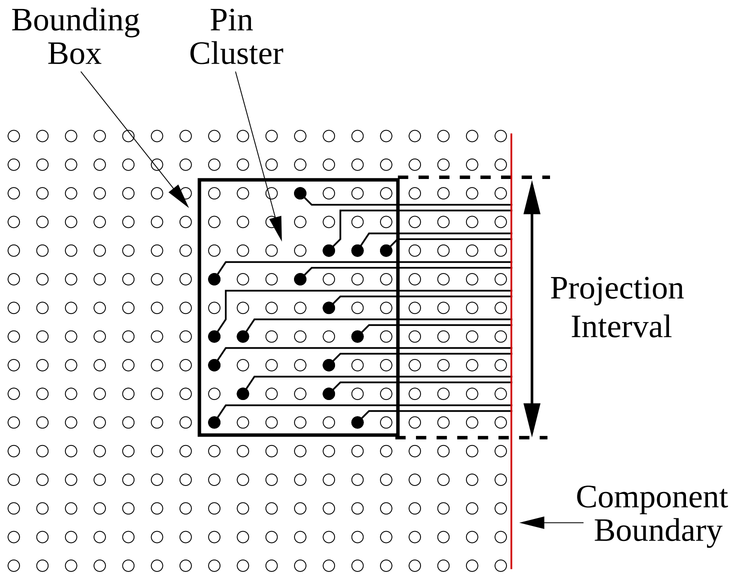


Figure 2.2: Illustration of the bus projection interval.

For each bus, its projection interval on a component can be obtained by projecting the bounding box of its pin cluster onto the component boundary, as shown in Figure 2.2. From industrial manual routing solutions, we observe that the nets of a bus typically escape a component from its projection interval. Based on this observation, each bus can be represented by two intervals located in different components.

Naturally, if two bus intervals overlap in a component, the net escape routes of the corresponding buses are mixed up. Therefore, for the buses chosen to be routed together, their intervals in each component should have no overlapping. If we define the ordering of a group of intervals in a component without overlapping by counting them from top to bottom, then the intervals of the chosen buses correspond to two sequences of intervals, one in each component. For example, in Figure 2.3, intervals of buses B, C, and D overlap in component 1, so they do not form a sequence, but intervals of buses B, D, and E do form a sequence in component 1. Furthermore, these two sequences of intervals should have the same ordering; otherwise their nets have crossings in the intermediate channel. In Figure 2.3, the intervals of buses A, B, and D in component 1 correspond to interval sequence $\langle A, B, D\rangle$, but those in component 2 correspond to a different interval sequence $\langle B, A, D\rangle$. Therefore, a common interval sequence is a bus interval sequence existing on both components, such as $\langle A, D, E\rangle$ in Figure 2.3. In addition, we define the weight of a bus as the number of its nets. Then the bus sequencing problem looks for an optimal common interval sequence, which is a common interval sequence with maximum sum of weights (i.e. $\langle A, D, E\rangle$ in Figure 2.3). Thus the bus sequencing problem is equivalent to the LCIS problem defined as follows:

Definition 1 *Given a bus set $B = \{b_1, b_2, \ldots, b_n\}$, its corresponding intervals on the left side are $L = \{l_1, l_2, \ldots, l_n\}$ and the intervals on the right side are $R = \{r_1, r_2, \ldots, r_n\}$. Each bus b_i also has a weight w_i. The LCIS problem is to find a common interval sequence of L and R such that the total weight of the corresponding buses is maximized. This total weight is denoted as $LCIS(L, R)$.*

A related problem in the literature is the LCS problem [10–13], in which each sequence defines a linear ordering on its elements. But in our LCIS problem, due to the interval overlappings, the intervals in a component only have a partial ordering. Thus, the LCIS problem is a generalization of the LCS problem since if the intervals do not overlap, it is reduced to the LCS problem. No LCS algorithm can be applied to solve it directly. In the

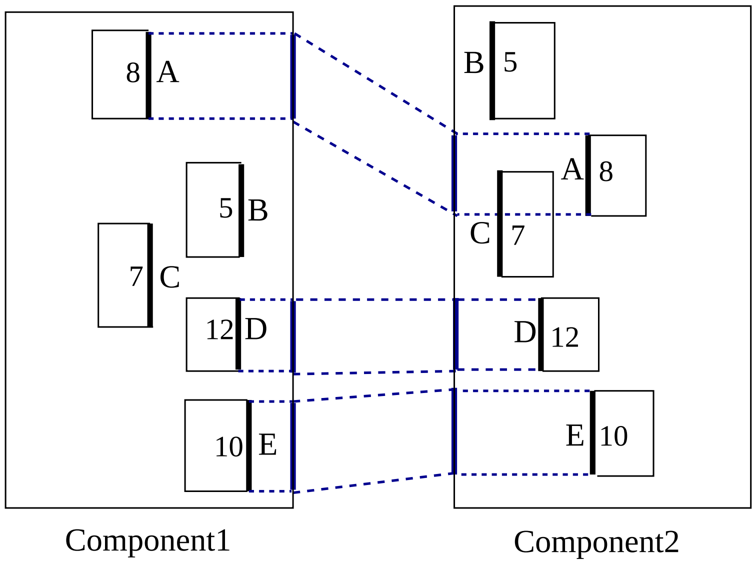


Figure 2.3: A sample bus sequencing problem with five buses. Each bus is represented by the bounding boxes of their pin clusters. The thick edge of each bounding box should be projected to the boundary to get the projection interval. The number inside the box represents the bus weight. The dashed lines show the solution.

next section, we propose an LCIS algorithm which can find an optimal solution in $O(n \log n)$ time.

In practice, the bus sequencing algorithm can be applied to the bus-centric escape routing problem followed by the net-centric escape routing algorithm in [8,9]. Here, the net-centric algorithm is applied to the buses chosen by the bus sequencing algorithm one by one, but not to all the buses directly. For the example in Figure 2.3, $\langle A, D, E\rangle$ is the longest common interval sequence, so buses A, D, and E are chosen to be routed on the same layer. Figure 2.4 gives a net-centric escape routing solution for bus D in the example in Figure 2.3. It is easy to see that all the nets in bus D escape the components from its projection intervals, so the net escape routes of bus D cannot mix or cross with those belonging to other buses. Similarly, apply the net-centric escape routing algorithm to bus A and E; we can then get the escape routing solution for this bus-centric escape routing problem, where 30 nets are routed.

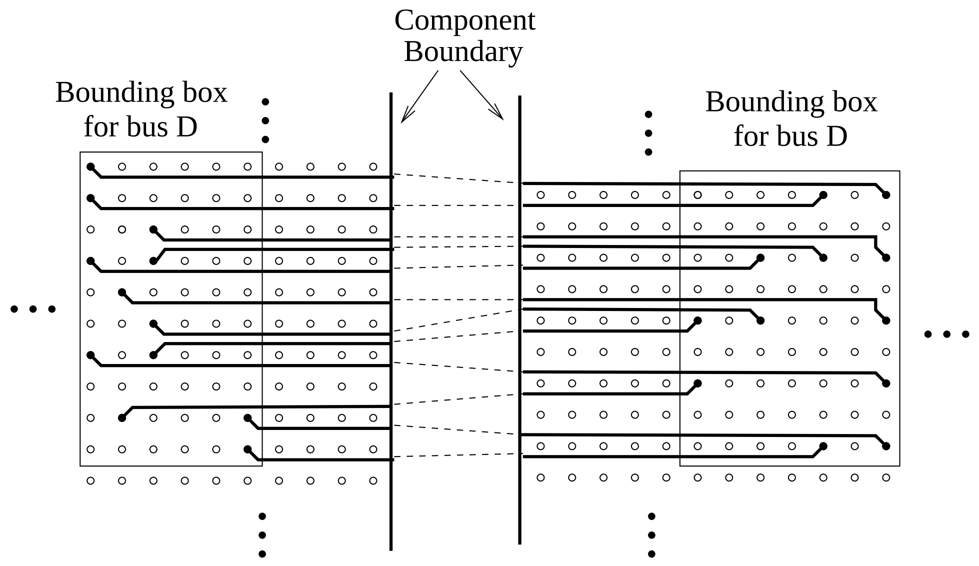


Figure 2.4: The net-centric escape routing solution for bus D in the example shown in Figure 2.3. The solid lines show the escape routes inside the components and the dashed lines show the connections between component boundaries.

2.3 Optimal LCIS Algorithm

Before we present our algorithm, we first introduce some terminology and some definitions. For bus set B with left intervals L and R, assume all intervals are parallel with the y axis, where the y coordinates increase from top to bottom. Then each interval $l_i = [l_i^l, l_i^u]$ in L is specified by its lower endpoint l_i^l and upper endpoint l_i^u, with $l_i^l > l_i^u$. Similarly, in R, r_i^l and r_i^u are the lower and upper endpoints of interval r_i, with $r_i^l > r_i^u$.

Definition 2 *For a bus b_i, we define the set of buses above its lower endpoints on both sides as its* above set *A_i:*

$$A_i = \{b_j | l_j^l < l_i^l \text{ and } r_j^l < r_i^l\}$$

We also define the set of buses above its upper endpoints on both sides as its strictly above set *SA_i:*

$$SA_i = \{b_j | l_j^l < l_i^u \text{ and } r_j^l < r_i^u\}$$

See Figure 2.5 for an example. For bus b_2, the buses that are above the solid line in both sides compose A_2 ($A_2 = \{b_1, b_3, b_6\}$), and the buses that are above the dashed line in both sides compose SA_2 ($SA_2 = \{b_1, b_6\}$).

Definition 3 *For a bus b_i, we also define the longest common interval sequence length $LCIS(b_i)$ as the longest length of the common interval sequence that is (inclusively) above the lower endpoint of b_i.*

In other words, if we draw a line between the lower endpoints of the two intervals of b_i, $LCIS(b_i)$ gives the length of the longest common interval sequence problem above that line. See Figure 2.5 for an example. $LCIS(b_2)$ defines a problem of the region above the solid line.

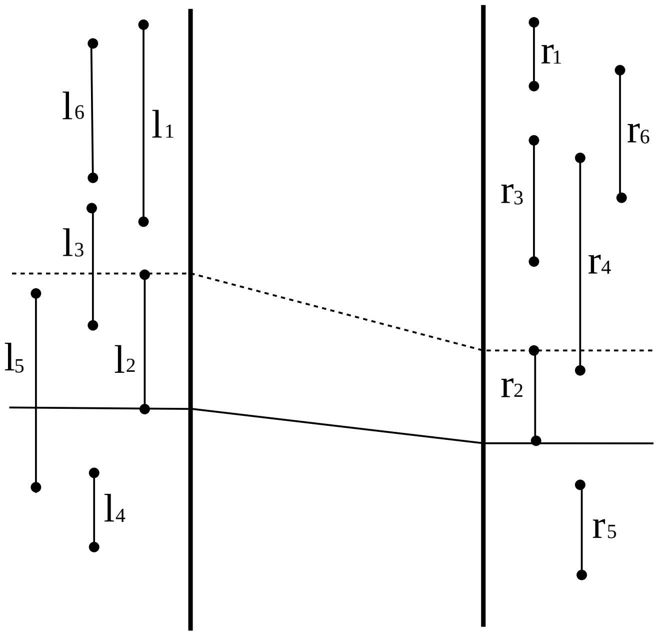


Figure 2.5: Illustration for the definitions and Lemma 1.

2.3.1 The Algorithm

For bus b_i, there are two cases: either bus b_i is included in $LCIS(b_i)$ or it is not included. If it is included, then $LCIS(b_i)$ is the maximum of all $LCIS(b_j)$, in which b_j is in the strict above set of b_i, plus the weight of b_i. Otherwise, $LCIS(b_i)$ is the maximum of all $LCIS(b_k)$, in which b_k is in the above set of b_i. We can formally describe this fact in Lemma 1:

Lemma 1 $LCIS(b_i) = \max(\max_{b_j \in SA_i} LCIS(b_j) + w_i, \max_{b_k \in A_i} LCIS(b_k))$. *If such* $SA_i = \emptyset$ *or* $A_k = \emptyset$, *then* $LCIS(b_j)$ *or* $LCIS(b_k)$ *is zero.*

We can see that this lemma gives a great hint: one LCIS problem could be solved by combining the solutions of two LCIS subproblems. Figure 2.5 gives an illustration. The LCIS length of bus b_2 is either the maximum of the LCIS of all the buses above the dashed line (the maximum of $LCIS(b_1)$ and $LCIS(b_6)$) plus the weight of b_2, or the maximum of the LCIS length of all the buses above the solid line (the maximum of $LCIS(b_1)$,$LCIS(b_6)$, and $LCIS(b_3)$). It should be the larger one of the two.

It is trivial that:

Lemma 2 *The solution of the LCIS problem is the maximum of the LCIS length of each bus:*

$$LCIS(L, R) = \max_{b_i \in B}(LCIS(b_i))$$

The two lemmas naturally lead to a dynamic programming approach. Our basic idea is as follows: we scan through the end points on the left side from top to bottom. When we meet an upper end of a bus b_i, we find $\max_{b_j \in SA_i} LCIS(b_j)$ as in Lemma 1 and associate it with this upper endpoint. When we meet a lower endpoint of an interval b_i, we find $\max_{b_k \in A_i} LCIS(b_k)$ and compare it with the $LCIS$ value associated with the upper endpoint of this interval plus the weight of b_i. The larger value becomes the LCIS value of this bus. Finally, we find the largest LCIS value among all the buses. The algorithm is shown in Algorithm 1. In the algorithm, $LCISset$ is a sorted array that collects the lower endpoint of each interval on the right side and the LCIS length of its corresponding bus. The array is then sorted by the

Algorithm 1 LCIS length computation

```
1: $LCISset = \emptyset$
2: $UPPER[i] = 0$ for all $i$
3: sort the upper and lower ends on the left side
4: **for** $i = 1$ to $2n$ **do**
5:     $value = 0$
6:     $e =$ the $i^{th}$ end on the left side
7:     **if** $e$ is an upper end $l_p^u$ of bus $b_p$ **then**
8:         By binary search, find $(endpoint, value)$ in $LCISset$ satisfying
           $(endpoint, value) = \max_u\{(u, v)|(u, v) \in LCISset$ and $u < r_p^u\}$
9:         $UPPER[p] = value$
10:    **else**
11:        $//e$ is a lower end $l_q^l$ of bus $b_q$
12:        By binary search, find $(endpoint, value)$ in $LCISset$ satisfying
           $(endpoint, value) = \max_u\{(u, v)|(u, v) \in LCISset$ and $u < r_q^l\}$
13:        $lower_q = \max(UPPER[q] + w_q, value)$
14:        add $(r_q^l, lower_q)$ to $LCISset$
15:        remove all $(u, v)$ from $LCISset$ satisfying
           $u > r_q^l$ and $v < lower_q$
16:    **end if**
17: **end for**
18: find $(endpoint, value)$ in $LCISset$ with the largest $value$
19: **return** $value$
```

right lower endpoints of the intervals. Each of its elements is in the form of $(r_i^l, LCIS(b_i))$. The $UPPER$ array records all the LCIS values associated with the upper endpoints on the right.

We show the correctness of this algorithm by comparing it with Lemma 1. Lines 8 and 9 calculate $\max_{b_j \in SA_i} LCIS(b_j)$, line 12 calculates $\max_{b_k \in A_i} LCIS(b_k)$, then line 13 compares $\max_{b_j \in SA_i} LCIS(b_j) + w_i$ with $\max_{b_k \in A_i} LCIS(b_k)$ and $LCIS(b_i)$ is equal to the larger of the two. Since the endpoints on the left side are sorted and since we scan from top to bottom, we process a lower or upper endpoint on the left side only when all the lower endpoints above it are processed. On the right side, whenever we query for the LCIS, we enforce that the endpoint we query is above the upper endpoint or lower endpoint of the bus ($u < r_p^u$ in line 8 and $u < r_q^l$ in line 12). These two properties guarantee $b_j \in SA_i$ and $b_k \in A_i$ in the lemma. (Notice that if nothing can be found in line 8 and line 12, $value = 0$.) To find the maximal LCIS for all b_j and b_k as in the lemma, we use the $LCISset$ to keep track of the LCIS value for all the buses that are processed. $LCISset$ records the lower endpoints in the right side and the $LCIS$ values of their corresponding buses ($LCISset = \{(r_i^l, LCIS(b_i))\}$). Whenever we obtained the $LCIS(b_i)$ for bus b_i, we store its lower endpoint on the right side together with $LCIS(b_i)$ into $LCISset$ (line 14). After that, we should remove all the entries in $LCISset$ that have the lower endpoint below the newly added one but have an LCIS value smaller than the newly added one. Such an entry cannot lead to an optimal solution. This removing step guarantees that the $LCIS(b_i)$ value in $LCISset$ is increasing as the lower endpoint coordinate r_i^l increases. Then, whenever we need to find the maximal LCIS above an endpoint a on the right side, we just use a as the key to query in $LCISset$ and find the lowest lower-endpoint that is above a. The $LCIS$ value associated with a must be the largest among all the $LCIS$ values above a. This also explains why, in lines 8 and 12, we find the lowest endpoint (u) instead of largest LCIS value. The two are consistent: the lowest endpoint in $LCISset$ above the endpoint a means the largest LCIS length above it. In a later example, we will show how this guarantees that the maximal LCIS for all possible j and k are selected. Finally, we take the largest LCIS length from the LCISset, which reflects Lemma 2. From the discussion above, we can see that:

Theorem 1 *Algorithm 1 gives the correct LCIS length.*

By this algorithm, we can obtain only the LCIS length. However, it is easy to obtain the LCIS sequence by adding a postprocess to back track $LCISset$.

2.3.2 An Example

Now we give a simple example to illustrate how our algorithm works. Figure 2.6 gives the problem configuration: the weights of each bus and the locations of each interval.

Now we simulate the process of our algorithm. Please refer to Figure 2.7 as we go through the whole process. We scan from top to bottom on the left side. The first two endpoints we encounter are l_1^u (the upper endpoint of l_1) and then l_3^u; since nothing has been added to $LCISset$ till now, we cannot find anything by line 8. Therefore, elements in $UPPER$ array

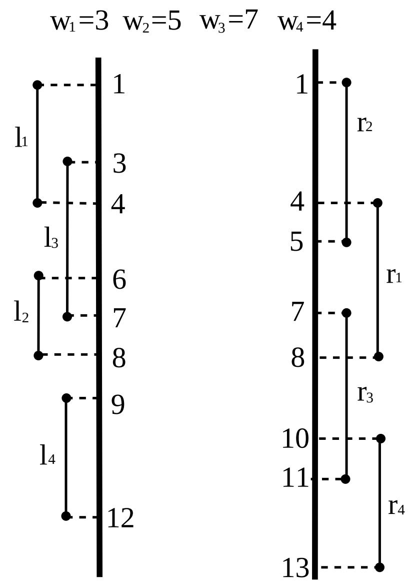


Figure 2.6: Example problem configuration.

are still 0 (Figure 2.7 (a) and (b)). We keep going and meet l_1^l. At this time, $UPPER[1] = 0$ and $LCISset = \emptyset$. Therefore, $lower_q = w_1 = 3$, and we add $(8, 3)$ into $LCISset$, where $8 = r_1^l$ (Figure 2.7(c)). We then meet l_2^u. We try to find the largest endpoint smaller than $r_2^u = 1$ in $LCISset$, but cannot find anything. Therefore, $UPPER[2] = 0$ (Figure 2.7(d)). For l_3^l, we use $r_3^l = 11$ to query in $LCISset$ and get $8 < 11$. We then compare 3 with $w_3 + UPPER[3] = 7$ and choose $lower_q = 7$. After this, $(11, 7)$ is added into $LCISset$.

$UPPER = [\mathbf{0}, 0, 0, 0]$ $LCISset = \emptyset$ (a) Processing l_1^u	$UPPER = [0, 0, \mathbf{0}, 0]$ $LCISset = \emptyset$ (b) Processing l_3^u
$UPPER = [0, 0, 0, 0]$ $LCISset = \{\mathbf{(8,3)}\}$ (c) Processing l_1^l	$UPPER = [0, \mathbf{0}, 0, 0]$ $LCISset = \{(8,3)\}$ (d) Processing l_2^u
$UPPER = [0, 0, 0, 0]$ $LCISset = \{(8,3), \mathbf{(11,7)}\}$ (e) Processing l_3^l	$UPPER = [0, 0, 0, 0]$ $LCISset = \{\mathbf{(5,5)}, \underline{(8, 3)}, (11,7)\}$ (f) Processing l_2^l
$UPPER = [0, 0, 0, \mathbf{5}]$ $LCISset = \{(5,5), (11,7)\}$ (g) Processing l_4^u	$UPPER = [0, 0, 0, 0]$ $LCISset = \{(5,5), (11,7), (13,9)\}$ (h) Processing l_4^l

Figure 2.7: Execution of our algorithm. Bold means newly modified in $UPPER$ or added into $LCISset$. Underlined item means newly removed from $LCISset$.

Then we process l_2^l; r_2^l is 5 and the $lower_q$ we obtain is also 5. We insert $(5, 5)$ into $LCISset$.

Notice that we should now remove $(8, 3)$ because $8 > 5$ and $3 < 5$. Otherwise, if we keep $(8, 3)$ in $LCISset$, we will have problems when processing l_4^u. In that case, we will use $r_4^u = 10$ to query in $LCISset$, and the largest endpoint that is smaller than 10 is 8. We would think that the LCIS length above r_4^u is 3 (bus 1 is selected). However, the best choice should be bus 2, and the LCIS should be 5. Therefore, we can see that if we do not remove $(8, 3)$ from $LCISset$, we cannot guarantee that the best choice is made every time we query $LCISset$, and we may lose the optimal solution. The rest of the simulation is similar to the previous steps. Finally, the longest LCIS length in this case is 9 (by choosing bus 2 and 4), which is optimal.

2.3.3 Time Complexity

Assuming we have n buses, sorting the endpoints on the left takes $O(n \log n)$ time. With the $LCISset$ implemented by a height-balanced BST using the endpoint as key and its corresponding LCIS as value, finding the endpoint in $LCISset$ as in lines 8 and 12 takes $O(n \log n)$ time. Adding (line 14) one entry into the BST can also be done in $O(\log n)$. So the total time for one iteration in the **for** loop takes $O(\log n)$, except for line 15 in which there might be multiple entries to be removed in one iteration. We will discuss the time complexity of line 15 by amortized analysis. Removing one entry takes $O(\log n)$. Since we will add at most n lower endpoints into the BST, we have at most n entries to remove. Therefore, line 15 takes $O(n \log n)$ time for all iterations. Line 18 takes $O(n)$. Summing them up, we get $O(n \log n)$ time complexity for this algorithm.

Theorem 2 *$O(n \log n)$ is the lower bound time complexity for the LCIS problem.*

The proof of this theorem is similar to the proof of the $O(n \log n)$ lower bound for comparison-based sorting. Here we give a proof sketch. Since all the decisions can be made only by comparisons (of the interval endpoints and of the weights), we can model any algorithm as a decision tree with the leaves as solution and internal nodes as comparison operation.

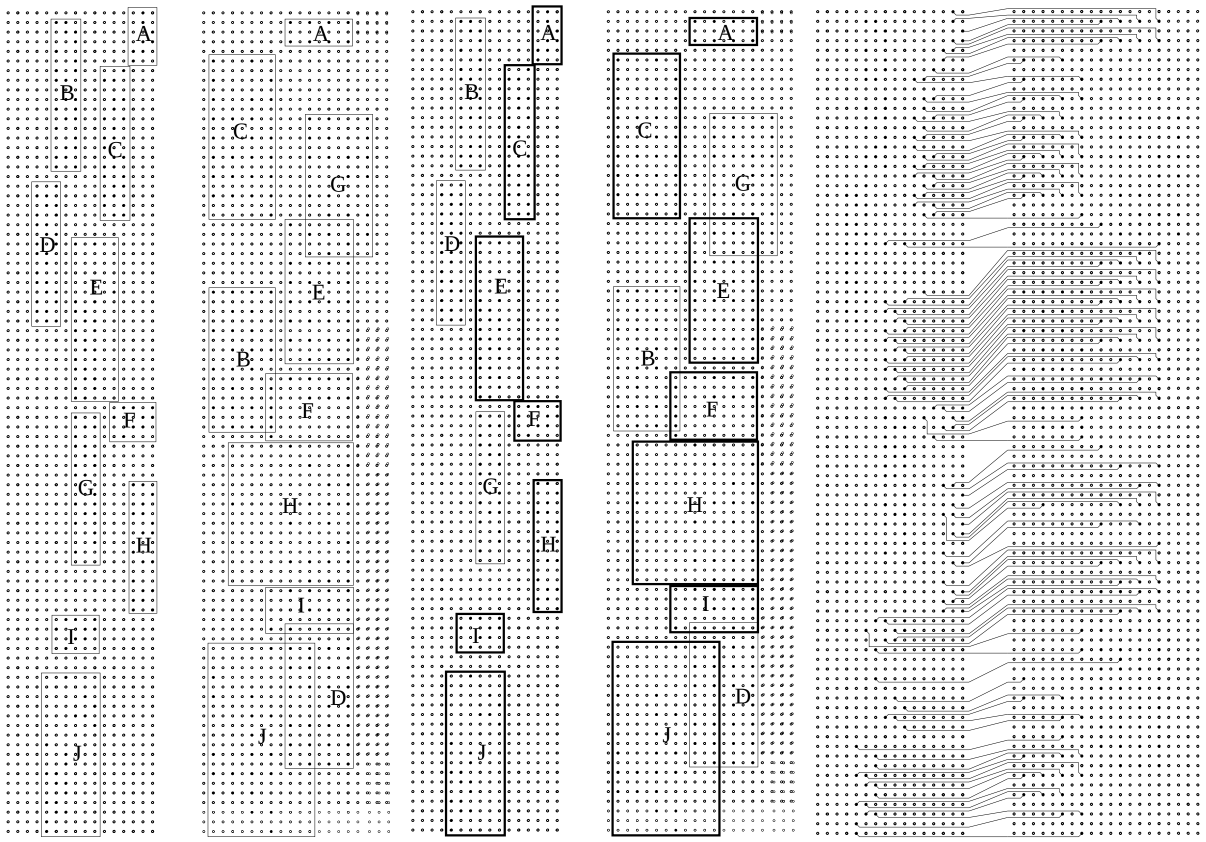
Since we have more than $n!$ possible solutions for this problem, the tree height must be over $O(n \log n)$. This indicates that we must do at least $O(n \log n)$ comparisons before we can reach a solution. The details of the proof are omitted.

2.4 Experimental Results

In this section, we derive some two-component experiments from industrial data to compare the escape routing results between the escape router with and without using the LCIS algorithm. Table 2.1 gives the experimental results. From Table 2.1, it is easy to see that the number of nets routed by the router with LCIS algorithm is much more than the number of nets routed by the original escape router [8] after eliminating all mixed-up nets. Through this comparison, we can conclude that the LCIS algorithm significantly improves the bus-centric escape router. For all our experiments, the LCIS algorithm is very efficient and it can be done in 1 second or less. We also illustrate a one-layer bus-centric escape routing problem and its solution in Figure 2.8. In this example, seven buses are chosen by the LCIS algorithm and 111 nets are routed finally.

Table 2.1: Comparison of the escape routing results between the escape router with and without using the $LCIS$ algorithm.

Data	# of routed nets	
	Escape without $LCIS$	Escape with $LCIS$
Ex1	42	54
Ex2	47	56
Ex3	45	65
Ex4	62	80
Ex5	87	111
Ex6	91	128



(a) Original problem (b) Bus sequencing solution (c) Final escape routing solution

Figure 2.8: A bus-centric escape routing problem with 10 buses from A to J and its solution. Here, $w_A = 8$, $w_B = 24$, $w_C = 26$, $w_D = 27$, $w_E = 25$, $w_F = 6$, $w_G = 21$, $w_H = 19$, $w_I = 7$, and $w_J = 20$. (a) The original bus-centric escape routing problem. The bounding boxes of all buses are shown. (b) The solution of the bus sequencing problem. Buses A, C, E, F, H, I and J, whose bounding boxes are highlighted, are chosen to be routed on the same layer. (c) The final escape routing solution of this problem by using the net-centric escape routing algorithm for buses A, C, E, F, H, I and J, respectively. In total 111 nets are routed.

2.5 Conclusions

In this chapter, we gave the bus sequencing problem, which is a subproblem of the bus-centric escape routing problem, and then solved it. We introduced a new optimization problem called the Longest Common Interval Sequence (LCIS) problem and formulated the bus sequencing problem as an LCIS problem. We proposed an LCIS algorithm that can find an optimal solution in $O(n \log n)$ time. We also showed that $O(n \log n)$ is a lower-bound and thus the time complexity is also the best possible. Experimental results show that our algorithm performs well.

CHAPTER 3

PIN ASSIGNMENT

3.1 Introduction

In PCB designs, pin assignment is to assign signals to pins in components on the board (a component is usually a pin array). Pins that are assigned the same signal are expected to be connected in a later routing stage. Therefore, pin assignment can also be viewed as a correspondence between pins from different components. It is an important step after placement because it greatly affects the later routing solution. Some assignments lead to very simple and clean routing while others may result in very complex or even unsolvable routing. Figure 3.1 gives such an example, whose 6 pins in a pin array are expected to be connected to 6 pins in the other array. We need to establish a pin-to-pin correspondence between them. It can be seen that the assignment in (a) leads to a very simple routing with short wirelength while the assignment in (b) results in back detours and thus longer wirelength.

In the past few years, as circuit designs become more and more dense, boards and packages are decreasing in size while the pin count is increasing. Today high-density fine pitch packages contain thousands of pins while they only occupy minimal board space. Such a large-scale problem makes early works on pin assignment for PCBs [14–17] inapplicable because those works focus on fairly small problems with around 20 pins [18]. Recently, Meister et al. [18] presented several pin assignment algorithms for large-scale PCB designs, but they are guided by some heuristic metrics to estimate routability and have no guarantee to obtain a routable solution. In this chapter, we present a novel pin assignment algorithm that guarantees

[2]This work has been published as "Optimal Simultaneous Pin Assignment and Escape Routing for Dense PCBs," pages 275-280, Proceedings of the 2010 Asia and South Pacific Design Automation Conference (ASP-DAC 2010).

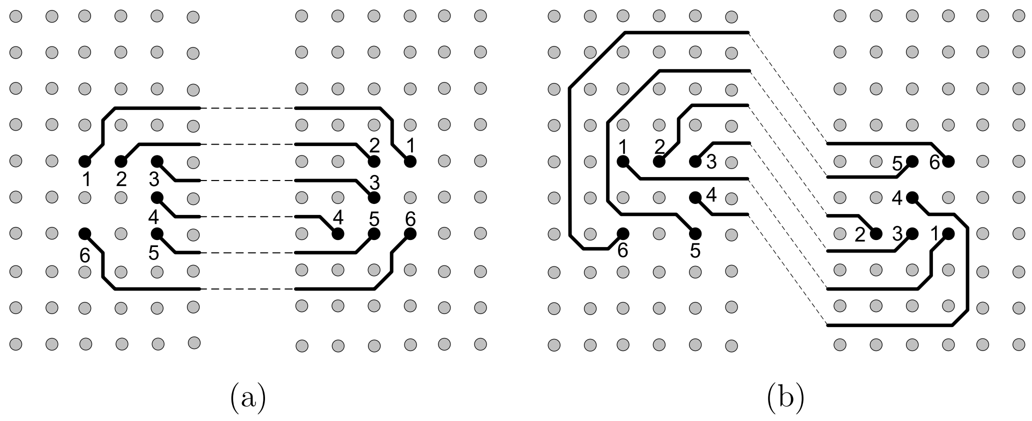


Figure 3.1: Pin assignment has great influence on later routing. Some assignments like (a) lead to simple and short routes while others like (b) lead to complex routes.

routability. We show that the problem of simultaneous pin assignment and escape routing can be solved optimally in polynomial time.

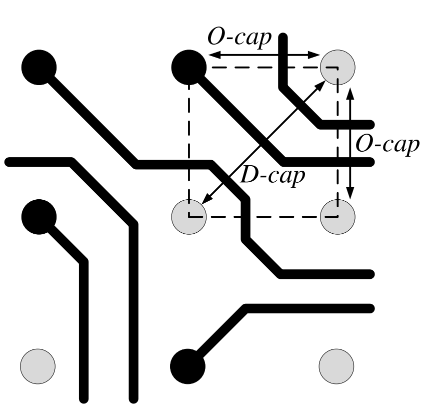
In high-end PCBs, nets are usually grouped into buses. In this work, we also discuss the advantages of using multiple layers to escape a bus, rather than a single layer, and we present an effective strategy for multi-layer escape routing of buses. We show that our routing strategy usually leads to the minimum wire length and occupies the minimum routing space. We test our approach on a state-of-the-art industrial board with 80 buses (over 7000 nets). The pin assignment and escape routing solutions for all the 80 buses are successfully obtained in less than 5 minutes of CPU time.

The rest of this chapter is organized as follows: Section 3.2 formulates the multi-layer simultaneous pin assignment and escape routing problem and then presents optimal algorithms to solve it. Section 3.3 focuses on pin assignment and escape routing for pins in buses, and proposes an effective bus escape strategy. Section 3.4 and Section 3.5 give the experimental results and the concluding remarks.

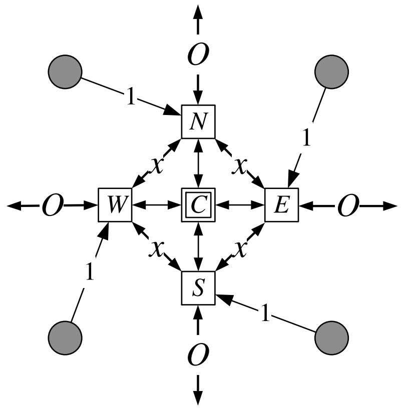
3.2 Optimal Algorithms

In [19], a flow-based algorithm is proposed, which can escape the maxinum number of pins from one component on one layer. This work will extend its flow model to escape pins from two components on multiple layers. In this section, we will briefly introduce the basic flow model in [19], and then propose an optimal multi-layer simultaneous pin assignment and escape routing algorithm and a fast optimal single-layer algorithm.

First of all, let us give some terminology. A component is defined as a 2-D pin array. A tile is a square formed by 4 adjacent pins in a pin array and the capacities of its four sides are equal to the orthogonal capacity constraint *O_cap*, which gives the maximum number of nets between two orthogonally adjacent pins, and the capacities of its two diagonals are equal to the diagonal capacity constraint *D_cap*, which gives the maximum number of nets between two diagnonal adjacent pins, as shown in Figure 3.2(a). An escape routing solution is *feasible* if all nets are routed to component boundaries in a planar fashion and the orthogonal and diagonal capacity constraints are satisfied everywhere inside components.



(a) Capacity constraints



(b) Basic flow model in a tile

Figure 3.2: Capacity constraints and the basic flow model inside a tile. (a) The orthogonal capacity constraint *O_cap* and the diagonal capacity constraint *D_cap*. The dashed box represents a tile. (b) Basic flow model inside a tile. The numbers and letters marked on edges represent their capacities, where $O = O\_cap$ and $x = \lfloor O\_cap/2 \rfloor$. Node C has capacity $D\_cap - 2 \cdot \lfloor O\_cap/2 \rfloor$.

3.2.1 Basic Network Flow Model

Given a pin grid with orthogonal capacity *O_cap* and diagonal capacity *D_cap*, the basic flow model M in [19] consists of two kinds of nodes:

- *Pin Nodes*: Pin nodes represent pins. In Figure 3.2(b), 4 circles represent 4 pin nodes on the corners of the tile.

- *Tile Nodes*: In each tile, there are 5 tile nodes: 4 *peripheral nodes* E, S, W and N represent 4 tile sides and have infinite capacity and one *central node* C has capacity $D\_cap - 2 \cdot \lfloor O\_cap/2 \rfloor$.

and two kinds of directed edges:

- *Intra-tile edges*: The directed edges between peripheral nodes have capacity $\lfloor O\_cap/2 \rfloor$. The directed edges between 4 peripheral nodes and the central node have infinite capacities. The directed edge from a pin node on the corner to a peripheral tile node has capacity 1 if the pin node corresponds to a to-be-escaped pin; otherwise it has capacity 0.
- *Inter-tile edges*: All inter-tile edges have capacity O_cap.

In the basic flow model, the peripheral nodes corresponding to the tile sides on the component boundaries are called the *boundary tile nodes*, and the pin nodes corresponding to the pins on the component boundaries are called the *boundary pin node*. Obviously, a pin escape route corresponds to a path from a pin node to a boundary node in the basic flow model.

[19] shows that if the max-flow of this basic network is equal to the number of to-be-escaped pins, there exists a feasible escape routing solution; otherwise there is no feasible solution. The max-flow solution can be converted into a planar topology of the escape routing by splitting nodes and edges whose flows are larger than 1. By using the algorithms in [20, 21], the escape routing topology can then be converted into detailed routing.

3.2.2 Multi-Layer Algorithm

The k-layer simultaneous pin assignment and escape routing problem can be defined as follows: Given k ($k \geq 1$) routing layers, two components C_1, C_2 with n to-be-escaped pins in each and n nets, the k-layer simultaneous pin assignment and escape routing problem is to find a pin assignment such that each net must have exactly one pin in C_1 and C_2 respectively, and the corresponding feasible k-layer escape routing solution. Here C_1 is a $m_1 \times n_1$ pin array with orthogonal capacity O_cap_1 and diagonal capacity D_cap_1, C_2 is a $m_2 \times n_2$ pin array with orthogonal capacity O_cap_2 and diagonal capacity D_cap_2, and $p_1, \ldots, p_n$ and

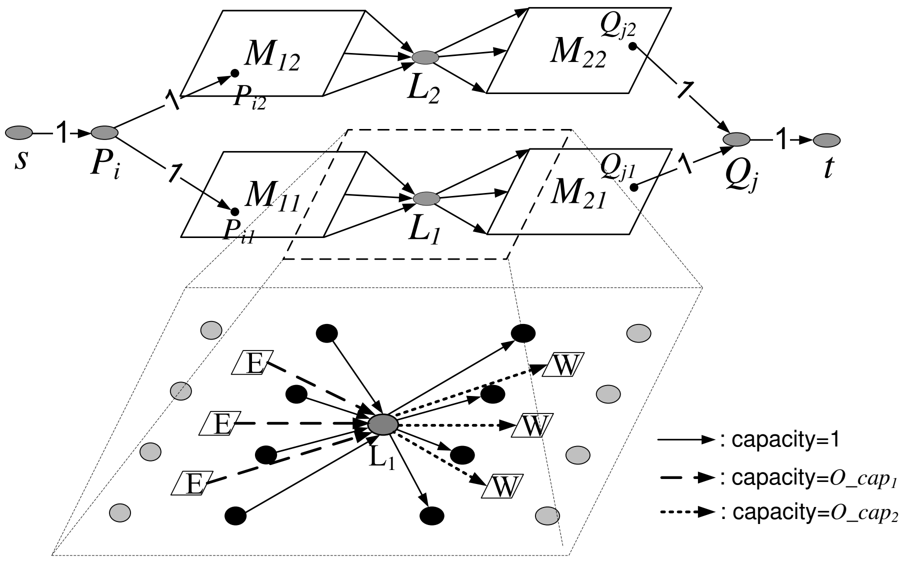


Figure 3.3: A two-layer flow model. In the zoomed-in view, the black circles are the boundary pin nodes in M_{11} and M_{21}, and the boxes labeled with E and W are the boundary tile nodes of M_{11} and M_{21} respectively.

$q_1, \ldots, q_n$ denote the specified pins in C_1 and C_2 respectively.

To solve this problem, we extend the network model in [19] to multiple layers as follows (see Figure 3.3) : Firstly, we construct a single-layer network for each layer l ($1 \leq l \leq k$) to model two component pin grids and their connections. We create the basic flow models M_{1l} and M_{2l} for the pin arrays of C_1 and C_2 on each layer l respectively. Then we add a node L_l for each layer l and create directed edges from all boundary nodes of M_{1l} to L_l and from L_l to all boundary nodes of M_{2l}. As shown in the zoomed-in view of Figure 3.3, the directed edges between L_l and the boundary pin nodes of M_{1l} and M_{2l} have capacity 1. The directed edges from the boundary tile nodes of M_{1l} to L_1 have capacity O_cap_1 and the directed edges from L_l to the boundary tile nodes of M_{2l} have capacity O_cap_2. Secondly, we connect pin nodes corresponding to the same to-be-escaped pin together. We create a node P_i for each pin p_i and connect directed edges from P_i to P_{il}, where P_{il} is the pin node corresponding to p_i in each layer l and $1 \leq i \leq n$ and $1 \leq l \leq k$. Similarly, we create a node Q_j for each pin q_j and connect directed edges from Q_{jl} to Q_j, where Q_{jl} is the pin node corresponding to q_j in M_{2l} and $1 \leq j \leq n$ and $1 \leq l \leq k$. Finally, we introduce a super source node s and a super sink node t. We create directed edges from s to P_i and directed edges from Q_j to t, where $1 \leq i, j \leq n$. All directed edges created in the last two steps have capacity 1. It is easy to verify that our model can correctly model the multi-layer simultaneous pin assignment and escape routing problem if the flow model in [19] can correctly model the escape routing problem in one component on one layer.

The complexity of this algorithm is $O(k^3(m_1n_1+m_2n_2)^3)$. In order to minimize the escape routing wirelength, we can assign cost 1 to all inter-tile edges of M_{1l} and M_{2l} $(1 \leq l \leq k)$ and assign cost 0 to all other edges, then compute the min-cost max-flow of the network.

3.2.3 Faster Single-Layer Algorithm

If the problem contains only 1 layer (i.e. $k = 1$), we can directly run the flow-based algorithm in [19] to escape all pins from C_1 and C_2 independently and then assign pins to nets by sweeping pin escape positions along boundaries of C_1 and C_2 in reverse orderings.

The time complexity of this algorithm is $O((m_1n_1)^3 + (m_2n_2)^3)$, which is faster than the algorithm in Section 3.2.2, whose running time is $O((m_1n_1 + m_2n_2)^3)$ when $k = 1$.

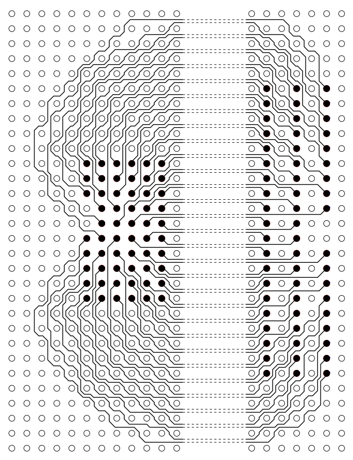
3.3 An Effective Escape Strategy

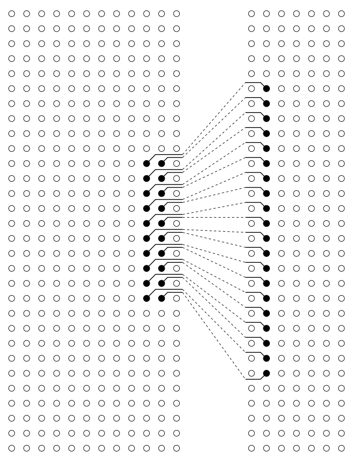
Today nets in high-end PCBs are always organized in bus structures. Nets in a bus have similar pin locations, which leads to dense routing inside components. In this section, we focus on pin assignment and escape routing for pins in buses. We will discuss the advantages of multi-layer solutions compared to single-layer solutions, then we will introduce projection-style escape routing and propose a cut-like algorithm to find a projection-style solution with minimal routing area inside components.

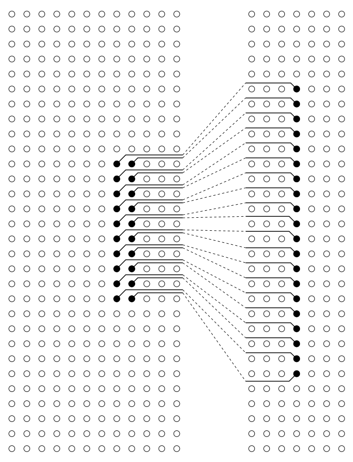
3.3.1 Single-Layer vs. Multi-Layer

When studying the simultaneous pin assignment and escape routing for buses, we discover that multi-layer solutions have some advantages over single-layer solutions. Here we use an example (Figure 3.4) to illustrate such advantages.

- *Shorter wirelength:* As shown in Figure 3.4(a), wires in the single-layer solution are


(a) Single-layer solution


(b) 3-layer solution
(layer 1)


(c) 3-layer solution
(layer 2)

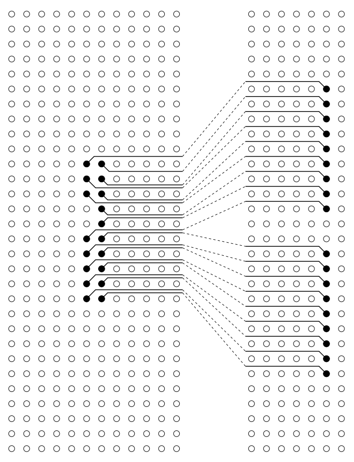

(d) 3-layer solution
(layer 3)

Figure 3.4: Single-layer solution vs. multi-layer solution.

very complicated. We can see a lot of wires with back detours in the left component. On the other hand, as shown in Figure 3.4 (b), (c) and (d), the 3-layer solution has very clean and simple escape routing; almost all pins escape components by the shortest route.

- *Less routing space:* In the single-layer solution (Figure 3.4(a)), pins escape components from a very wide range on the component boundaries, and the back detour wires on the left component take a lot of extra routing space on the left of the pin cluster. Due to this issue, the single-layer solution may force other nets to be routed on different layers, potentially leading to the routing layer increase. But the 3-layer solution (Figure 3.4 (b), (c) and (d)) does not cause this problem since pins escape the left component from a small interval on the component boundary and no extra routing space is needed.

- *Smaller wirelength variation:* If we take a close look at the wires on the left component in Figure 3.4(a), we can see that the escape routes of pins on the leftmost pin column are back detoured a lot, so they are very long. But the escape routes of pin on the rightmost pin column are very short. So the escape wirelength difference is large in the single-layer solution. In high frequency designs, signals in a bus are often expected to arrive almost at the same time, so their routes should have the same length. In order to find an equal-length routing solution, the escape wirelength variations need to be compensated outside the components, which is handled by length-matching routers [22, 23]. If the length difference of the net escape routing is large, it would be very difficult for length-matching routers to find a feasible solution. Therefore, the single-layer solution increases the difficulty of length-matching routing. However, the escape wirelength difference in the 3-layer solution is very small because almost all pin escape routes have been minimized.

3.3.2 Projection-Style Escapes

Section 3.3.1 shows that compared to single-layer solutions, multi-layer simultaneous pin assignment and escape routing solutions have some benefits, such as short escape wirelength. To obtain a multi-layer solution, we can limit the number of nets routed on the same layer. Since nets in a bus usually have the same escape directions, their escape wirelength is minimized when all nets escape components from bus projection intervals, which can be obtained by projecting the bus pin cluster bounding boxes to the component boundaries along the escape directions, as shown in Figure 3.5. A bus escape routing is in projection-style if all its nets escape components from bus projection intervals. Projection-style bus escape routing is also discussed in [24]. Figure 3.4 (b), (c) and (d) show a 3-layer projection-style escape routing for a bus.

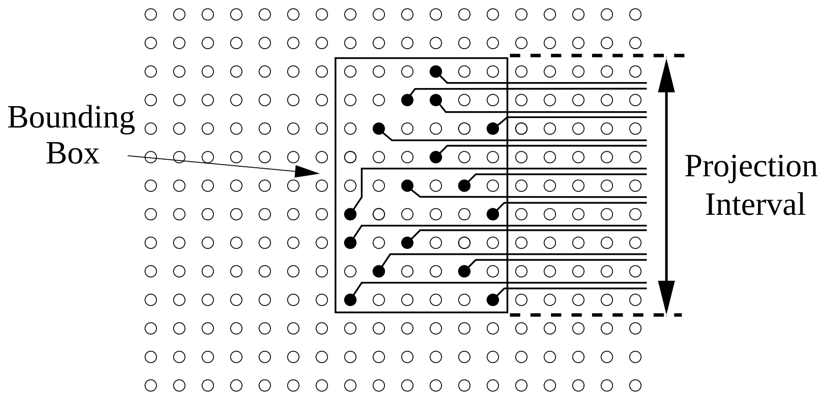


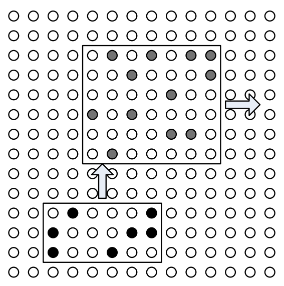
Figure 3.5: Bus projection interval on a component.

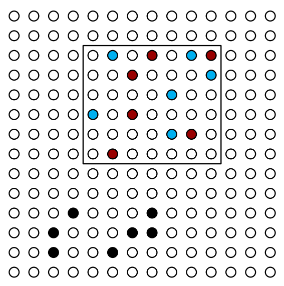
3.3.3 Cut-Like Layer Assignment

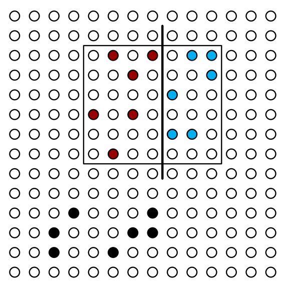
When the pins in a bus are dense, it is not possible to escape all pins on one layer using the projection style. In this case, we need to assign pins to different layers. However, there exist many layer assignments which lead to different routing resource consumptions. Some assignments may be preferable to others. Here we propose a cut-like layer assignment.

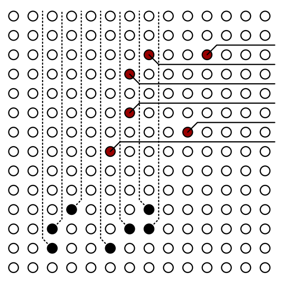
A cut-like layer assignment is to cut the bounding box of the pins into several rectangular pieces and assign pins in each piece onto an individual layer. We observe that this cut-like layer assignment usually consumes less routing space and can therefore lead to better solution. Figure 3.6 gives an example. Given two buses A (represented by gray pins in (a)) and B (represented by black pins in (a)), we expect to escape A through the left boundary and B through the top boundary. Since pins in A are very dense, we need to escape them on two layers using projection style. Pins in B can be escaped on one layer. If we randomly choose pins in A to be escaped in one layer and escape the rest in the other layer like shown in (b), we will end up with the escape routing of B conflicting with the escape routing of A in both layers, as shown in (d) and (e). Therefore, we have to use a third layer to escape B. However, if we cut the pins into two rectangular pieces and escape them on different layers as shown in (c), the escape routing of B can then be routed with one rectangular piece of pins without conflict as shown in (g). As a result, we can escape both buses using only two layers.

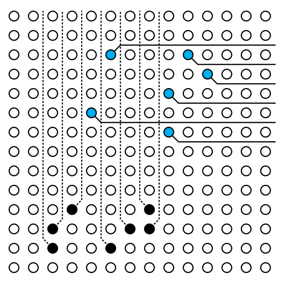
The cut-like layer assignment can be obtained by repeatedly applying the fast single-layer algorithm in Section 3.2.3. However, we need to make some minor changes to the flow model

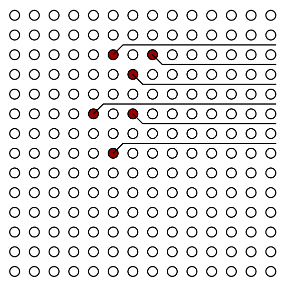

(a) A two bus problem


(b) Random assignment


(c) Cut-like assignment


(d) Layer 1 of (b)


(e) Layer 2 of (b)


(f) Layer 1 of (c)

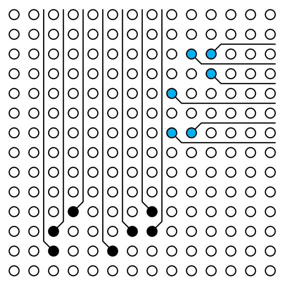

(g) Layer 2 of (c)

Figure 3.6: Random layer assignment vs. cut-like layer assignment.

in order to obtain the cut-like effect. First, we need to compute the number of wires that can be escaped through the projection interval, which we denote as p. This number limits the number of pins that can be assigned on one layer in our cut-like assignment. Next we compute the distance from each pin P_i to the boundary of the component, which we denote as d_i. Then we change the weight of the edge from super source s to P_i to $d_i \times M$, in which M is a very large number. Now by applying the fast single-layer algorithm in Section 3.2.3 to compute the min cost p-flow of the network, we can obtain p pins with smallest distances to the boundary. We assign them to one layer and delete these pins from our network. We then repeat this procedure to assign pins to new layers until all pins are assigned. Because pins are assigned according to their distances to the boundary, pins with similar distances are assigned to the same layer, which leads to the cut-like effect we want.

If the algorithm produces a k layer routing, the runtime would be $O(k(m_1n_1)^3+(m_2n_2)^3)$, which is much faster than the k-layer algorithm in Section 3.2.2, whose complexity is $O(k^3(m_1n_1+m_2n_2)^3)$.

3.4 Experimental Results

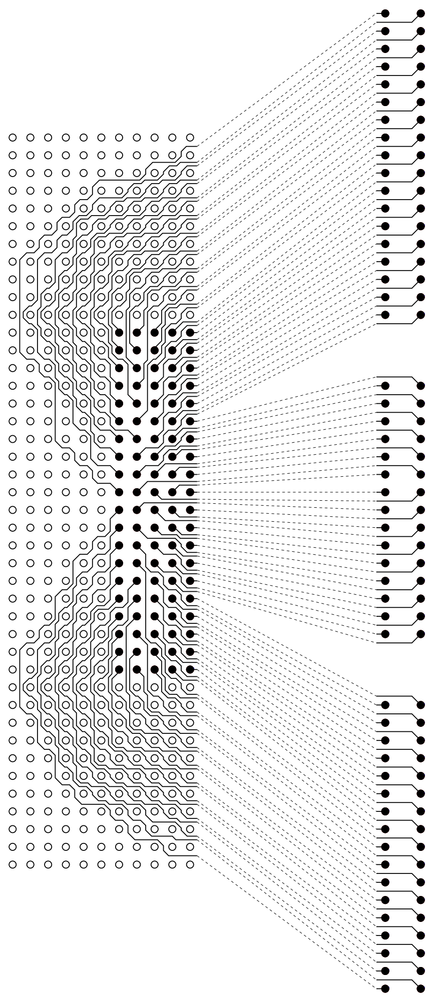
We implement our simultanous pin assignment and escape routing algorithms in C++ and use the min-cost flow solver CS2 [25] to obtain the flow solutions of our networks. We performed three experiments to test our simultaneous pin assginment and escape routing approaches, and all of them are run on a workstation with two 3.0 GHz Intel Xeon CPUs and 4 GB memory.

A) GPU Example

We tested our simultaneous pin assignment and escape routing algorithms on an industrial example in [18], which is to perform pin assignment for a 100-signal graphics processor unit (GPU). [18] only gives a possible pin assignment without routablity guarantee, but our algorithms produced two routable pin assignments and their corresponding escape routing solutions. In this test, first we set the orthogonal and diagonal capacity constraints for the left and right components to be 2, 3 and 1, 2, respectively, and we obtained a single-layer pin assignment and escape routing solution, as shown in Figure 3.7(a). Then we changed the orthogonal and diagonal capacity constraints of the left component to be 3 and 4, and we got another single-layer solution, as shown in Figure 3.7(b).

B) Single-Layer vs. Multi-Layer

We performed an experiment to compare the pin-to-pin wirelengths of a single-layer and a cut-like two-layer pin assignment and escape routing solutions for a bus with 62 nets. In this experiment, we applied our single-layer and cut-like multi-layer algorithms to assign


(a)

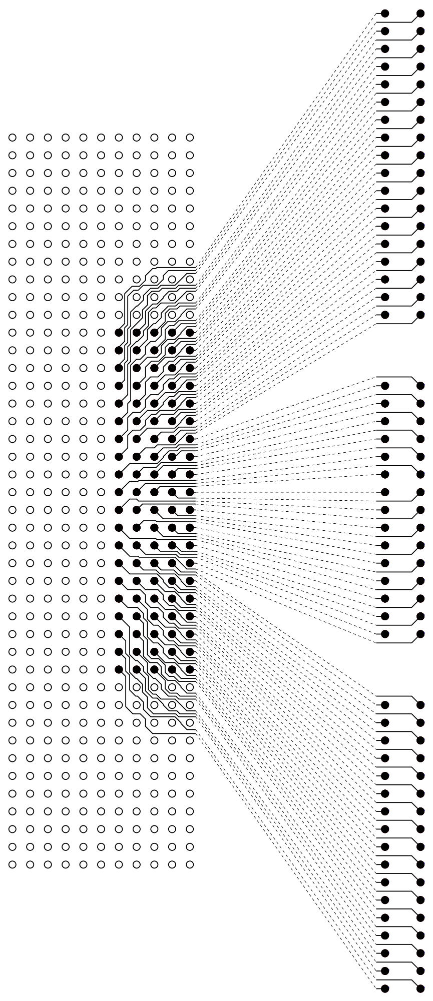

(b)

Figure 3.7: Two single-layer pin assignment and escape routing solutions for a GPU with 100 signals. (a) Single-layer solution with orthogonal capacity 2 and diagonal capacity 3 in the left component. (b) Another single-layer solution but the orthogonal and diagonal capacities of the left component are changed to 3 and 4. In both (a) and (b), the right component has orthogonal capacity 1 and diagonal capacity 2.

pins to nets and route nets inside components, and applied the length-matching router in [22] to route nets between components. Figure 3.8 and Figure 3.9 display the single-layer and two-layer solutions respectively, where all net routes have the same length in each solution. It is clear that the single-layer solution in Figure 3.8 has much longer escape routes inside components and has much more length extensions outside components than the two-layer solution in Figure 3.9. In this experiment, we set the distance between centers of two orthogonally adjacent pins to 1 and the distance between two components to 43. The total wirelength of the single-layer solution is 5792.11, which is 37.15% longer than the total wirelength of the two-layer solution, 4223.05. The escape wirelength and the detailed wirelength of the single-layer solution are 1695.21 and 4096.90, increases of 21.36% and 44.96% over the two-layer solution respectively.

C) Industrial Board

Finally, we tested our cut-like simultaneous pin assignment and escape routing algorithm on a state-of-the-art industrial PCB, which has 80 buses (over 7000 nets) and has been manually routed (usually it takes about two months to manually route a PCB of this scale). The largest bus has 338 nets which are routed in 6 layers in the manual solution. In this experiment, we extract bus escape directions from the manual solution. Finally all 80 buses are successfully assigned and routed in less than 5 minutes of CPU time, and for each bus, the number of layers it utilized is no more than that in the manual routing solution.

3.5 Conclusions

In this chapter, we first presented a novel algorithm to obtain a pin assignment solution that guarantees routability. We showed that the problem of simultaneous pin assignment and escape routing can be solved optimally in polynomial time. We then focused on the pin assignment and escape routing for the terminals in a bus; we discussed advantages of multi-layer solutions over single-layer solutions and then proposed an effective strategy for multi-layer pin assignment and escape routing for buses. We showed that our strategy usually

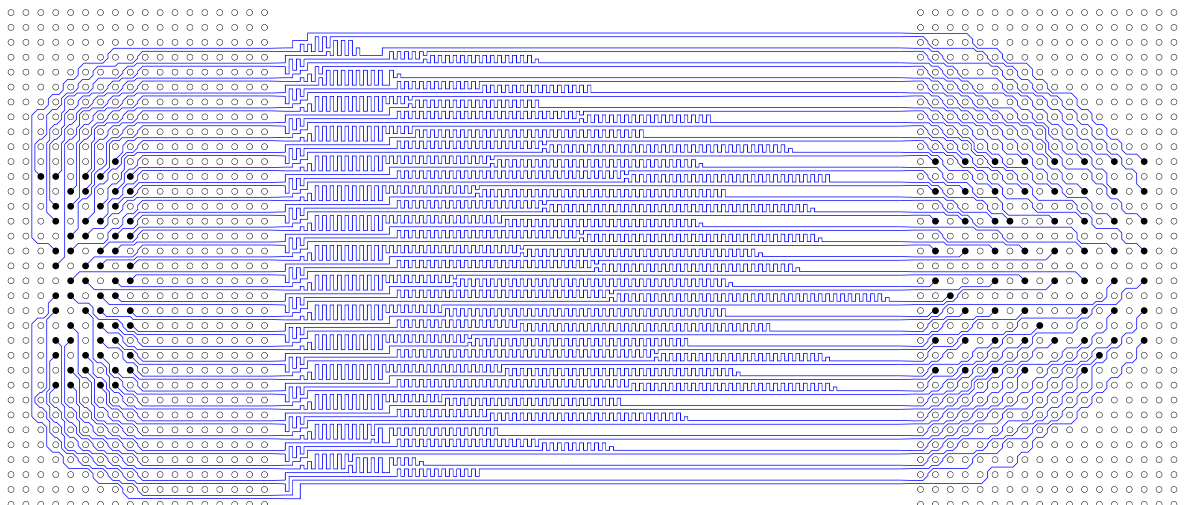
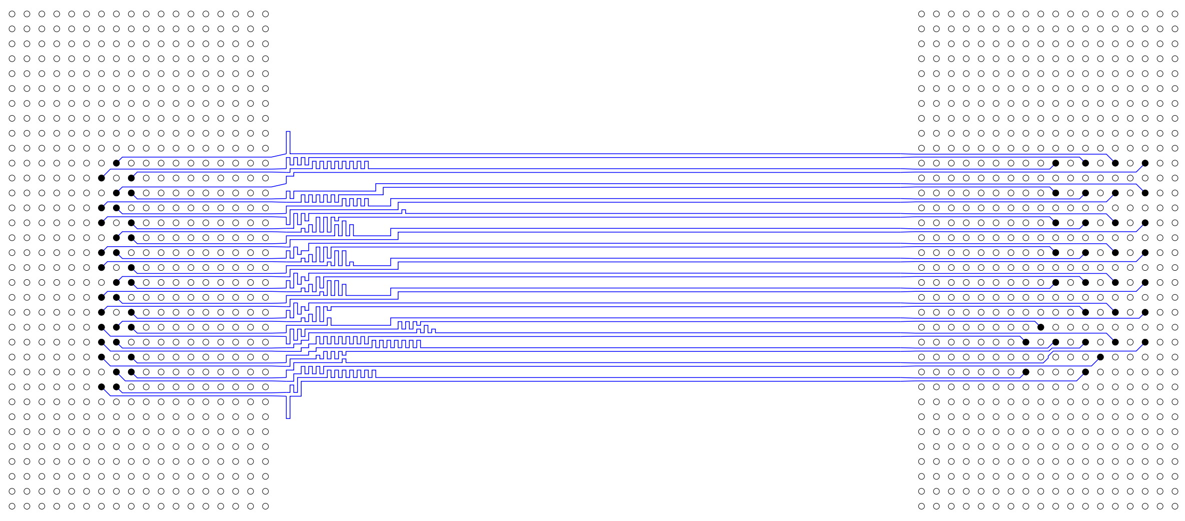
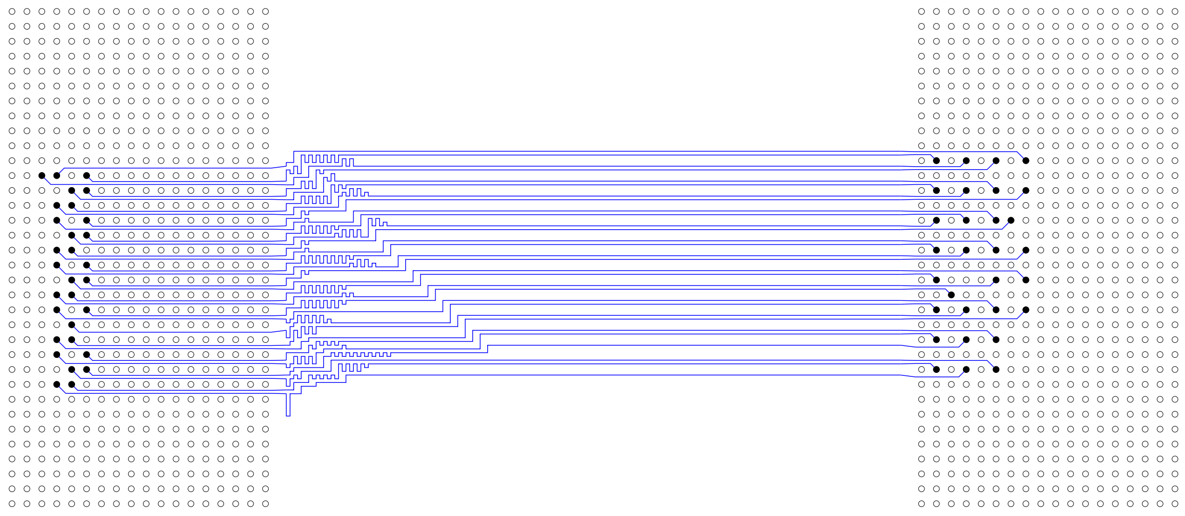


Figure 3.8: An equal-length single-layer simultaneous pin assignment and escape routing solution of a bus.

leads to multi-layer solutions having minimum wirelength and occupying minimum routing space. We tested our approach on a state-of-the-art industrial board with 80 buses (over 7000 nets). The pin assignment and escape routing solutions for all 80 buses are successfully obtainted in less than 5 minutes of CPU time.



(a)



(b)

Figure 3.9: An equal-length cut-like two-layer simultaneous pin assignment and escape routing solution of a bus.

CHAPTER 4

BUS PLANNING

4.1 Introduction

As introduced in Chapater 1, the PCB routing problem is so difficult that current auto-routers are not able to produce acceptable solutions. Today many industrial designs are still completed by manual efforts. In the future, PCB designs will be substantially more complex, making manual routing approaches infeasible.

Designers always prefer the nets in a PCB to be bundled together as buses. All the nets in a bus are expected to be routed together. Planning the buses, which consists of assigning the buses to the routing layers and topologically routing them on each layer in a planar fashion, is one of the most time-consuming steps in PCB routing. Figure 4.1 shows an example of the bus planning result on one layer. Practical boards may contain many such layers. During the planning, we have to consider crossings of the bus topologies and the routing congestion between the on-board components as well as the min-max length bounds of the nets. As far as we know, existing commercial tools that help bus planning are interactive tools [26–28].

Notice that the bus planning problem is different from the multi-layer global routing problem for ICs [29–31] because the X-Y style for IC routing breaks a net into horizontal and vertical segments on different layers, while in PCBs, all the nets in a bus are expected to be routed on a single layer because any via introduced to the net can reduce the reliability and performance of circuits, damage signal integrity and increase the manufacturing cost. This fundamental difference makes IC routers incapable of performing bus planning.

We are developing a complete PCB auto-routing system, and this chapter reports the bus

[3]This work has been published as "Automatic Bus Planner for Dense PCBs," pages 326-331, Proceedings of the 46th Annual Design Automation Conference (DAC 2009).

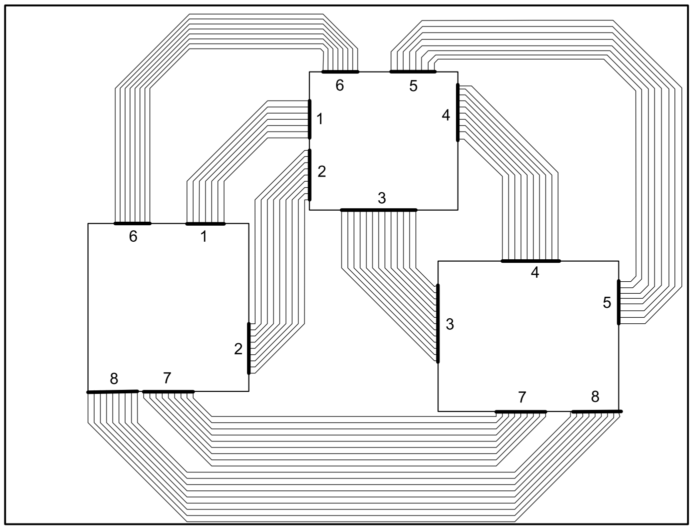


Figure 4.1: One layer of bus planning solution.

planning part of the system. To the best of our knowledge, this is the first automatic bus planner for complex PCB design. Our bus planner consists of three main stages:

1. Global routing, which routes all buses on a single layer with all routing resources mapped onto one layer.
2. Layer assignment of the routed bus.
3. Iterative improvement by reassignment and rerouting.

Our bus planner has the following features:

- We introduce a novel dynamic routing graph to guide the search of topological bus routes. The dynamic structure of the graph enables us to find planar routes for the buses more efficiently.
- We introduce a bin-packing-based congestion estimation. It provides a more accurate estimation than traditional congestion models. Owing to its accuracy, our planner effectively avoids violations of routing capacity constraints.
- We introduce an iterative improvement technique that resembles the bubble-sort. It helps further improve the routing solution.

We tested our bus planner on a state-of-the-art industrial circuit board with over 7000 nets and 12 signal layers. It has been completely routed by manual efforts. We perform two experiments. First, we take the layer assignment of the buses from the manual solution and apply our bus planner to route each layer. We are able to achieve 100% routing completion while satisfying the length constraints within 40 minutes. Then, we use our bus planner to perform simultaneous layer assignment and topological bus routing. We are able to successfully route 98.5% of the nets while satisfying length constraints. The remaining 1.5% can be routed either manually or by using vias. The runtime of our planner is less than 3 hours.

The rest of this chapter is organized as follows: Section 4.2 introduces the bus planning problem. The three stages of our planner—global routing, layer assignment and iterative improvement—are introduced in Section 4.3, Section 4.4 and Section 4.5 respectively. Experimental results are given in Section 4.6, and Section 4.7 concludes the chapter.

4.2 Bus Planning Problem

A typical high-end PCB contains a number of components, such as MCM, memory, and I/O modules, and a large number of bus structures between these components. Each bus is a group of two-pin nets between components. Inside the component, the nets of a bus will be escaped to the component boundaries to generate bus escape intervals, which are the starting points of our bus planner. Each bus can be represented by a pair of escape intervals on the component boundaries. We note that for any pair of buses, their internal escape routes may conflict with each other, as displayed in Figure 4.2, which means they cannot be assigned to the same layer.

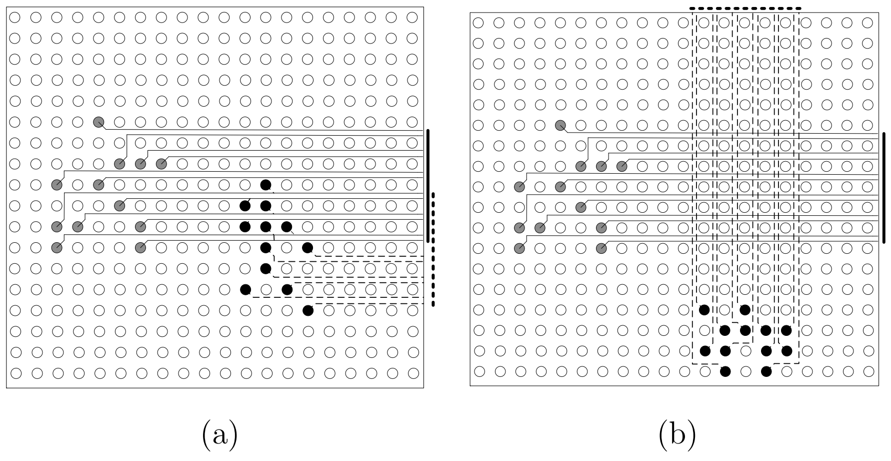


Figure 4.2: Internal route conflicts.

The input of our bus planning problem is the number of routing layers and a set of bus intervals as well as a bus internal conflict graph, whose vertices correspond to buses and whose edges correspond to internal route conflicts between buses. The expected output is the layer assignment of the topological routes of buses, where each bus is routed in planar fashion considering crossings, routing congestion and min-max length bounds. Figure 4.3 gives an example of a bus planning problem and its two-layer planning solution. In this paper, we define the bus net count as its number of nets and the bus route width as the width of the space its net routes need to occupy, which is proportional to the bus net count.

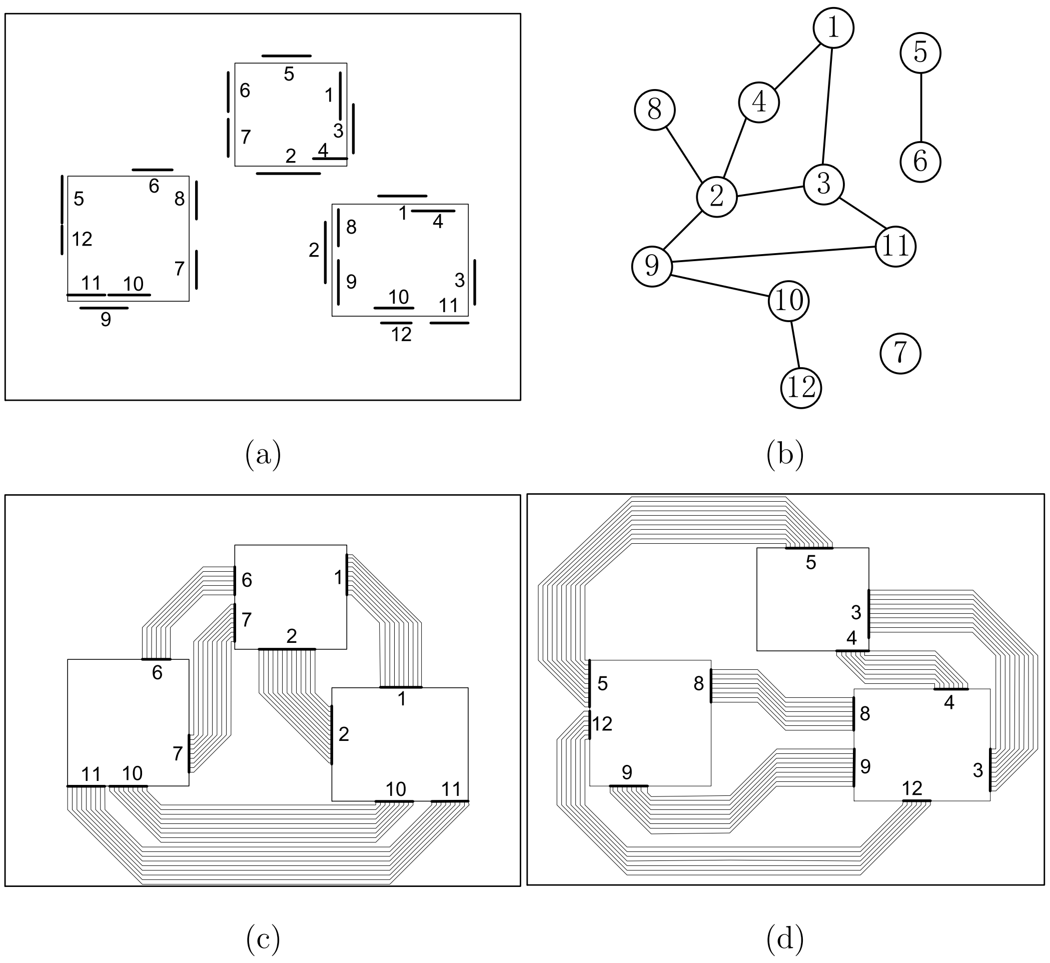


Figure 4.3: Given bus escape intervals (a) and bus internal conflict graph (b) as input, the expected output of planning buses to two layers is (c) and (d).

Our bus planning consists of 3 stages: (1) bus global routing, (2) bus layer assignment and (3) iterative improvement by rerouting and reassigning. We will explain them in detail one by one.

4.3 Global Routing

Global routing is the first stage of our bus planner. It generates the initial topological bus routes by routing buses on a single layer which includes the routing resources of all routing layers. A negotiated-congestion router is applied in this stage. The generated initial routes are then assigned to given routing layers in the layer assignment stage and then the planning solution will be improved in the iterative improvement stage.

In this section, we first introduce the Hanan grid and the dynamic routing graph data structure; then we present the bin-packing-based congestion estimation approach and finally describe the negotiated-congestion router.

4.3.1 Hanan Grid

The routing graph of our global routing is based on the Hanan grid, which is constructed from the components by extending their boundaries until they touch other components or the board boundaries. The board and its components are all represented by their bounding boxes. Figure 4.4 shows a Hanan grid for three components. The grid cells and edges are refined to Hanan grid cells and Hanan grid edges, respectively.

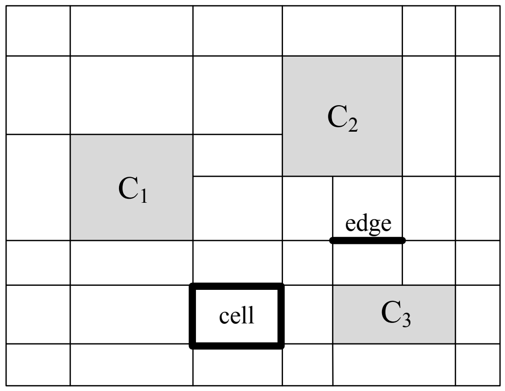


Figure 4.4: Hanan grid.

4.3.2 Dynamic Routing Graph

In the global routing stage, we route all buses on a single layer in which the routing resources of all routing layers are lumped together. On one hand, we want to avoid crossings between buses without internal conflicts since we do not know if they will be assigned to the same layer in later stages. On the other hand, we allow crossings between buses with internal conflicts since we know they will not be assigned to the same layer. Thus we need a flexible data structure to handle both of these situations.

For this purpose, we introduce a *dynamic routing graph*, whose vertices are the middle points of all Hanan grid edges and whose edges are connections of vertices in the same cell. When an edge is occupied by a bus, the cell is split into two parts and new vertices and new edges are created. Figure 4.5 gives an example. Figure 4.5(a) shows the initial routing graph in a Hanan grid cell, and Figure 4.5(b) shows the new routing graph after a bus b passes through this cell, where two vertices (l and r) representing the left and right Hanan grid edges are eliminated and four vertices (l_1, l_2, r_1 and r_2) representing four smaller grid edges are added. In Figure 4.5(b), the thick solid lines represent the route of bus b, which cuts the left and right Hanan grid edges into two smaller ones. The thin solid lines represent the new routing graph edges that do not cross with bus b. The dashed lines represent the new routing graph edges that cross b. The edges represented by dashed lines will be given high penalties to avoid bus crossings with b. Figure 4.6 gives a global view of how the routing graph change happens after a bus is routed.

By using this dynamic routing graph, our global router can handle all possible bus topologies, and each topological bus route is a path on this graph. Figure 4.7 gives an example, where three buses b_1, b_2 and b_3 pass through a Hanan grid cell consecutively and b_1 and b_3 have internal conflict. In Figure 4.7, to simplify the demonstration, we only show the graph vertices. Figure 4.7(a) gives the initial routing graph. As shown in Figure 4.7(b), when b_1 is routed, four new graph vertices are added and two old vertices are deleted. When routing b_2, the routing graph edges crossing with b_1 are given high penalties. As a result, b_2 passes through this cell without crossing b_1 and the routing graph changes again, as shown in Figure 4.7(c). When routing b_3, since b_3 has internal conflict with b_1, the two nets will

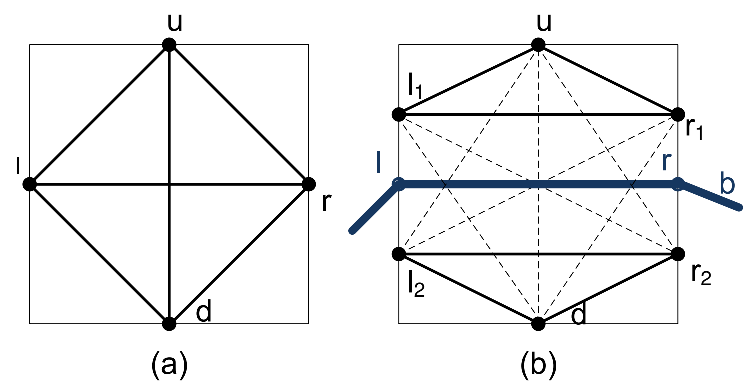


Figure 4.5: The dynamic routing graphs in a Hanan grid cell. (a) and (b) The routing graphs before and after a bus is routed. The thin dashed lines in (b) represent edges intersecting with the routed bus and they may be given high penalties.

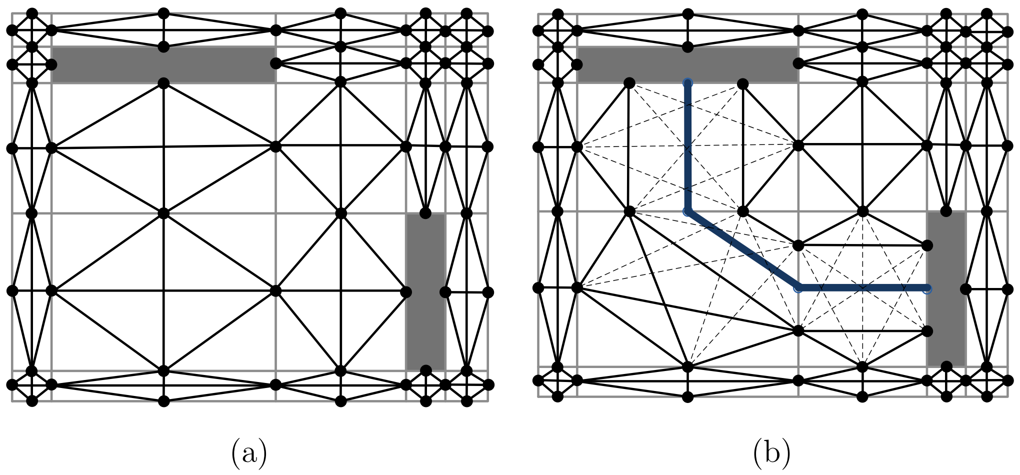


Figure 4.6: The dynamic routing graphs of a PCB before (a) and after (b) a bus is routed. The thin dashed lines in (b) represent routing graph edges that are given high penalties.

be routed on different layers. Therefore, we do not need to assign a high penalty to the edges crossing with b_1. To avoid bus crossing between b_1 and b_2, edges crossing with b_2 are given high penalties. Finally, as shown in Figure 4.7(d), b_3 is routed to pass through this cell intersecting b_1 but not intersecting b_2.

Figure 4.8 gives an example of all the routing paths on one layer. It can be seen that each routing path goes through either the middle points of a Hanan edge or newly created middle points after splitting a Hanan cell.

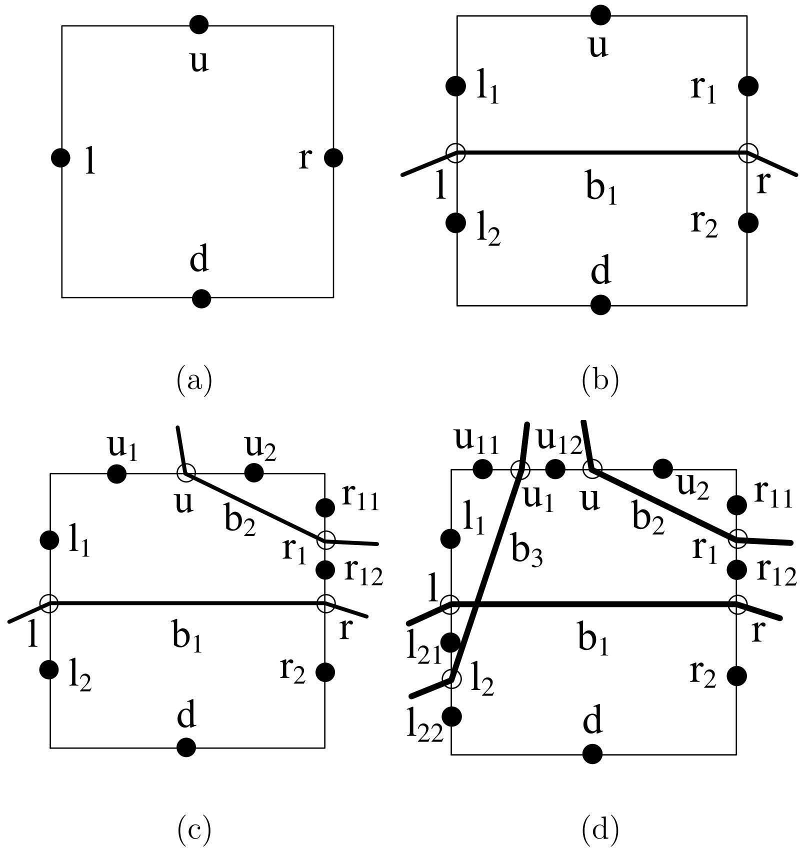


Figure 4.7: The dynamic routing graph in a Hanan grid cell as three buses b_1, b_2 and b_3 pass through it consecutively. Here b_1 has internal conflict with b_3. (a) Initial routing graph. (b), (c) and (d) Routing graphs when b_1, b_2 and b_3 are routed consecutively.

4.3.3 Congestion Estimation

In the global routing stage, to capture the routing space outside the components, we introduce the *critical cuts*, which are the line segments connecting each component corner to all its visible component corners or component or board boundaries. A corner is visible to another corner if the line segment between the two corners does not intersect any other component. Similarly, for a component corner, if the line segment that starts from this corner and is normal to a component or board boundary does not intersect any component, then the component or board boundary is visible to the corner. The thick lines in Figure 4.9 gives all critical cuts starting from the component corner v.

Since the critical cuts capture the routing space outside the components, congestion may happen on them and our global router should be able to estimate the congestion. All critical

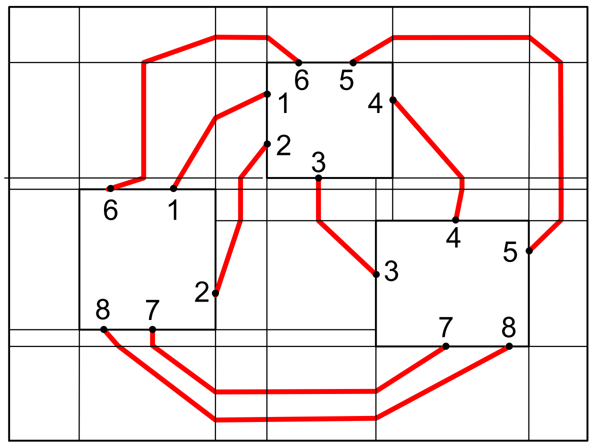


Figure 4.8: An example of all the routing paths on one layer.

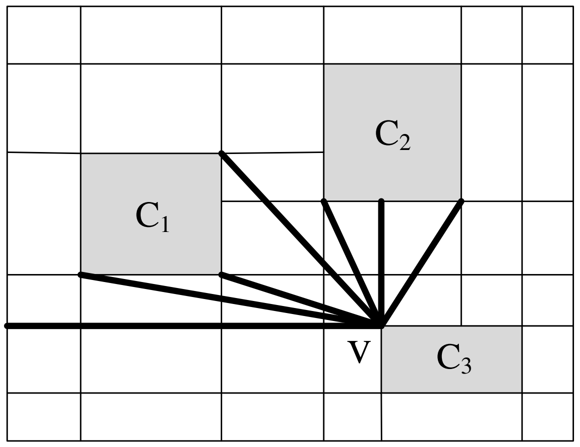


Figure 4.9: The critical cuts from corner v.

cuts can be captured by our routing graph, so their congestion can be estimated. As shown in Figure 4.9, each orthogonal critical cut consists of a sequence of consecutive Hanan grid edges, so it can be captured by routing graph vertices on these edges. Each diagonal critical cut intersects some Hanan grid cells, so it can be captured by routing graph edges crossing this cut inside these cells. Since our routing graph is dynamic, whenever the graph changes, the graph vertices and edges capturing critical cuts will be updated.

Given a critical cut t and L routing layers, the traditional approach estimates its congestion by comparing the total number of nets passing through it and the maximum number of nets it can accommodate, as listed in Equation 4.1. In the equation, $c(t)$ is the maximum number of nets t can accommodate on a single layer, b_i $(1 \leq i \leq m)$ are buses passing through t and $s(b_i)$ is the net count of b_i.

$$Congestion(t) = max(0, \sum_{i=1}^{m} s(b_i) - L \cdot c(t)) \tag{4.1}$$

But this traditional approach estimates congestion at the net level instead of at the bus level, so it is not accurate. Figure 4.10 gives an example, where there are five buses b_1, b_2, b_3, b_4 and b_5 with 20, 20, 20, 20 and 15 nets respectively passing through a critical cut with single-layer capacity 50 in two layers. By using capacity-based estimation, this cut is not congested, since the total net count, 95, is less than its overall capacity, 100. But as shown in Figure 4.10, we can not assign all five buses onto two layers.

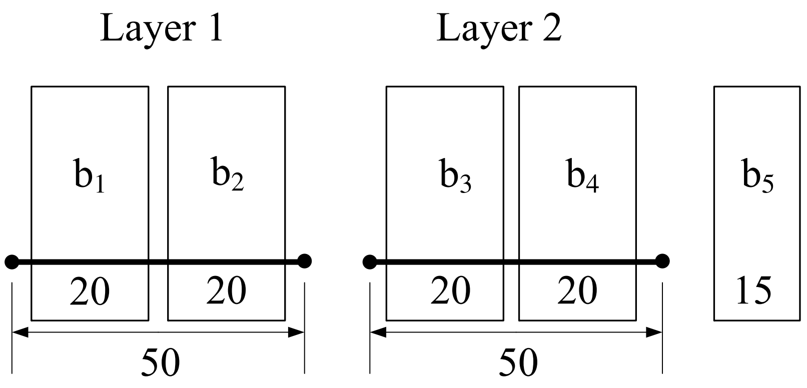


Figure 4.10: An example of bin-packing-based congestion estimation.

To estimate congestion at the bus level, we propose the bin-packing-based estimation. Here we regard critical cut t as a bin with size $c(t)$ and regard a bus b_i $(1 \leq i \leq m)$ as an item with size $s(b_i)$. The bin number b is the minimum number of bins to accommodate all these m items. The congestion it estimates is

$$Congestion(t) = max(0, b - L) \tag{4.2}$$

For the example shown in Figure 4.10, if we apply the bin-packing-based estimation to it, the bin number is 3, but the layer number is 2, so the critical cut must be congested on some layer.

Although the bin packing problem is NP-hard [32], there exist many effective and fast heuristic algorithms [33].

4.3.4 Negotiated-Congestion Routing

In this section, we will present our negotiated-congestion router. Our negotiated-congestion algorithm is similar to the Pathfinder negotiated-congestion algorithm [34–36], which was originally proposed for FPGA routing. It rips up and reroutes buses iteratively. In each iteration, each bus is ripped up and rerouted once by following the same ordering. When routing a bus b, the router is to find a minimum cost path on the routing graph. The cost function consists of four terms: length cost, crossing cost, congestion cost and length-bound violation cost. The length cost is the primary objective and it is proportional to the route length of b. The other three terms are penalty costs. The crossing cost is proportional to total net count of buses b crosses. The congestion cost is proportional to the total congestion of congested critical cuts b passes through. The length-bound violation cost is proportional to the amount by which b's route length exceeds its maximum length bound. Note that the minimum length bound of nets can be satisfied by extending net routes.

4.4 Layer Assignment

Simulated annealing (SA) [37] is used to assign the global routes of the buses obtained from the previous stage onto the given layers. The initial assignment is a random assignment of all the buses. In each step of the annealing process, we attempt to move one bus to another layer and evaluate the cost of the new assignment. The cost of an assignment is a weighted sum of two parts: (1) intersection cost, which is the total number of nets that have crossings with each other, and (2) congestion cost, which is the total congestion cost of all critical cuts by using bin-packing estimation.

The output of this SA procedure is a layer assignment of all the topological bus routes. Later, we will iteratively improve this layer assignment as well as the individual routes by using a bubble-sort-like bus recycling technique.

4.5 Iterative Improvement

After the initial bus topologies are assigned to multiple layers, there might still be conflicts between the bus routes on some layers, such as bus intersections and routing congestion. This is because when we perform global routing, we did not know the subsequent layer assignment result and thus may have produced suboptimal routes for some buses. To address this issue, we perform another negotiated-congestion routing to the existing buses on each layer after the layer assignment. In this routing step, we assign very high cost to bus intersections so that they are minimized.

If routing conflicts still exist, we perform a bubble-sort-like iterative improvement. First we collect the conflicting buses on all layers together, reassign them to the first layer and reroute all buses on this layer. Then we choose and fix the maximum set of non-conflicting buses on the first layer. The remaining conflicting buses are then moved to the second layer and rerouted together with the buses already assigned on it. The maximum set of non-conflicting buses are then selected to be fixed on the second layer and the remaining conflicting ones are moved to the third layer. Such procedure is performed iteratively until all buses are routed without conflicts or the result cannot be further improved.

Figure 4.11 gives an example of bubble-sort-like iterative improvement, where the numbers in the dark shaded boxes and in the light shaded boxes represent the number of non-conflicting and conflicting nets assigned to the layers, respectively. Figure 4.11(a) gives the initial layer assignment result after rerouting nets layer by layer, resulting in 40, 30 and 30 conflicting nets on the three layers. All these 100 conflicting nets need to be recycled. Figure 4.11(b) shows that all 100 conflicting nets are reassigned to layer 1. In Figure 4.11(c), 20 of 100 conflicting nets are added to layer 1 without conflicts after rerouting all nets on layer 1; then the remaining 80 conflicting nets are reassigned to layer 2. In Figure 4.11(d), 50 of 80 conflicting nets are added to layer 2 after net rerouting and the remaining 30 nets are reassigned to layer 3. Finally, these 30 nets are all successfully added to layer 3, as shown in Figure 4.11(e).

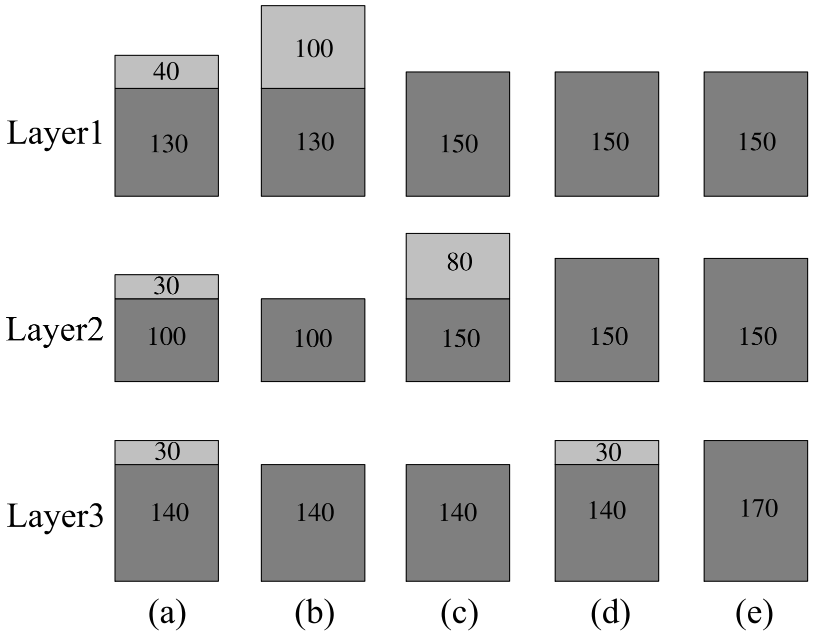


Figure 4.11: An example of bubble-sort-like improvement.

4.6 Experimental Results

We test our bus planner on a state-of-the-art PCB from industry that has 7000+ nets and 12 routing layers and has been manually routed. The manual solution is able to route all buses without any intersection or congestion. We run our experiments on a workstation with two 3.0 GHz Intel Xeon CPUs and 4 GB memory.

We did two kinds of experiments. The first one was to test the effectiveness of our single-layer bus router. We directly used the layer assignment extracted from the manual solution and ran our negotiated-congestion router on each layer to obtain the topological routing of each bus. We were able to route *all the buses* with no conflicts or congestion within 40 minutes. The length constraints of all the buses were also satisfied. We compared our routing with the manual routing and found that most of our routing is the same as the manual solution. Figure 4.12 shows the distribution of the nets among the 12 layers in the manual solution and how many of the nets have the same routing as our solution. It can be seen that on the first four layers, the bus routes are exactly the same while on the other layers, the majority of the routes are the same.

We performed another experiment to test our bus planner including layer assignment and

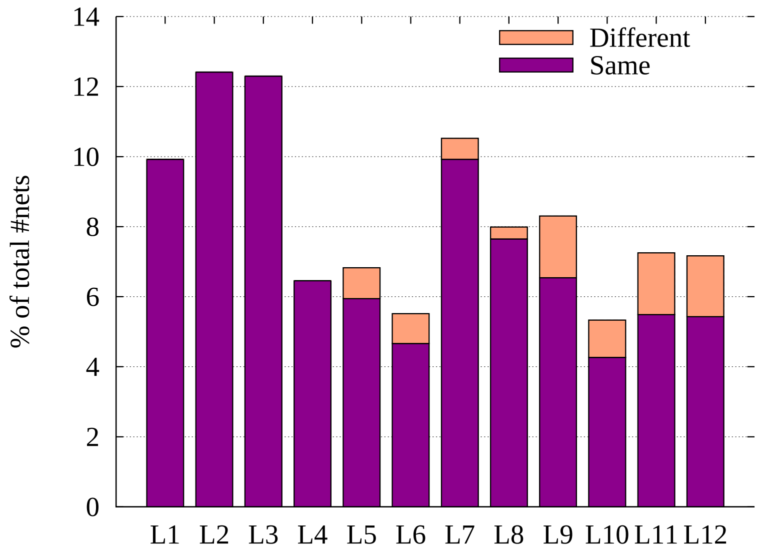


Figure 4.12: Layer-by-layer comparison between our routing solution and manual routing solution. y-axis is the percentage over the total number of nets in the design.

topological routing. In this experiment, we were only given the intervals of all the buses and output the layer assignment as well as topological routing. After the layer assignment stage, 95.9% of the nets were successfully assigned and routed. The iterative improvement procedure increased the completion rate to 98.5%. This completion rate is very close to 100%. Only 1.5% of the nets are left unrouted. These nets can be routed by manual effort or by using vias. The total CPU time was less than 3 hours.

Table 4.1 compares our layer assignment result with the manual layer assignment. The first two columns give the layer as well as the number of nets assigned to each layer by our planner. The next 12 columns show the number of nets that are assigned to each of the 12 layers in the manual solution. For example, among the 573 nets that our planner assigned to layer 1, only 81 nets are still assigned to layer 1 in the manual routing. It can be concluded from this table that the correlation between our layer assignment and the manual assignment is very small, which means our SA-based approach obtained a very different solution.

Table 4.1: Comparison between our layer assignment and the manual layer assignment.

Layer	net#	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12
1	573	81	183	269	0	0	26	0	0	0	0	0	14
2	486	0	0	0	0	0	91	0	95	103	60	13	124
3	698	56	0	82	84	31	0	117	14	128	0	94	92
4	611	78	14	25	0	31	31	0	0	0	0	280	152
5	741	0	243	42	84	0	0	84	124	41	0	62	61
6	646	173	82	0	28	0	0	84	155	43	81	0	0
7	498	0	81	140	170	62	31	0	0	0	14	0	0
8	507	0	60	0	60	59	113	83	118	0	14	0	0
9	622	78	0	0	0	81	0	155	0	84	102	61	61
10	437	0	159	0	0	185	0	0	0	93	0	0	0
11	557	143	0	225	14	31	62	82	0	0	0	0	0
12	549	89	51	82	14	0	34	81	30	92	76	0	0

4.7 Conclusions

In this chapter, we reported an automatic bus planner, which is one of the key parts of a complete PCB auto-routing system we are developing. In this bus planner, we use a dynamic routing graph to guide the efficient search of planar topologies of the bus routes and a bin-packing-based congestion estimation to effectively avoid violations of routing capacity constraints. We also use a bubble-sort-like iterative improvement to further increase the number of routed nets. We tested our bus planner on a state-of-the-art industrial circuit board with over 7000 nets and 12 signal layers. Our bus planner is able to achieve 100% routing completion using the layer assignment extracted from manual design. For simultaneous layer assignment and bus routing, we are able to successfully route 98.5% of the nets while satisfying length bounds. The remaining 1.5% can be routed either manually or by using vias. The runtime of our bus planner is less than 3 hours.

CHAPTER 5

BUS ESCAPE

5.1 Introduction

For a bus, its projection interval on a component can be obtained by projecting the bounding box of its pin cluster onto the component boundary and its boundary rectangle can be obtained by extending its pin cluster bounding box to the component boundary along its escape direction, as shown in Figure 5.1. From industrial manual routing solutions, we observe that the nets of a bus typically escape a component from its projection interval and the escape routes of all its nets are within its boundary rectangle. Since a bus can escape the component from any component boundary, each bus can be represented by four mutually overlapping boundary rectangles, which are attached to four different component boundaries, respectively. Here, the weight of a boundary rectangle is assigned to be the number of nets in its corresponding bus. Obviously, a feasible bus escape routing solution must correspond to the a disjoint subset of the bus boundary rectangles, and vice versa. Thus, the optimal solution of the bus escape routing problem must correspond to the *maximum disjoint subset (MDS)* of all bus boundary rectangles. In this chapter, we will formulate the bus escape problem as an MDS problem for boundary rectangles and then solve it optimally.

The MDS of a group of rectangles is defined to be a subset of non-overlapping rectangles with the maximum total weight. The general MDS problem, which is finding the MDS of general rectangles, has been proved to be NP-complete in [7]. One special case of this problem is to find the MDS of a set of boundary rectangles, which still remains as an open problem and is closely related to some difficult routing problems in PCB designs. In this

[4]This work has been published as "An Optimal Algorithm for Finding Disjoint Rectangles and Its Application to PCB Routing," pages 212-217, Proceedings of the 47th Annual Design Automation Conference (DAC 2010).

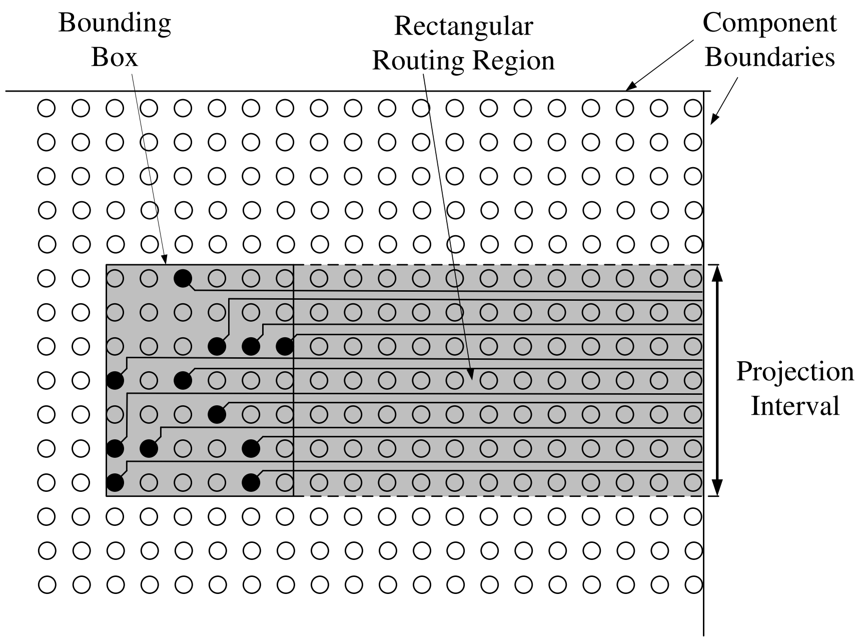


Figure 5.1: Bus projection interval and bus boundary rectangle (the shaded rectangle).

paper, we study this open problem.

Given a rectangular region R, a rectangle r completely lying within R is said to be a *boundary rectangle* if r has at least one boundary attached to the boundary of the rectangular region R. Figure 5.2(a) shows a set of boundary rectangles of the same rectangular region and Figure 5.2(b) gives the corresponding MDS of these boundary rectangles. To the best of our knowledge, the MDS problem of boundary rectangles is still open. In this paper, we will propose a polynomial time optimal algorithm to solve it.

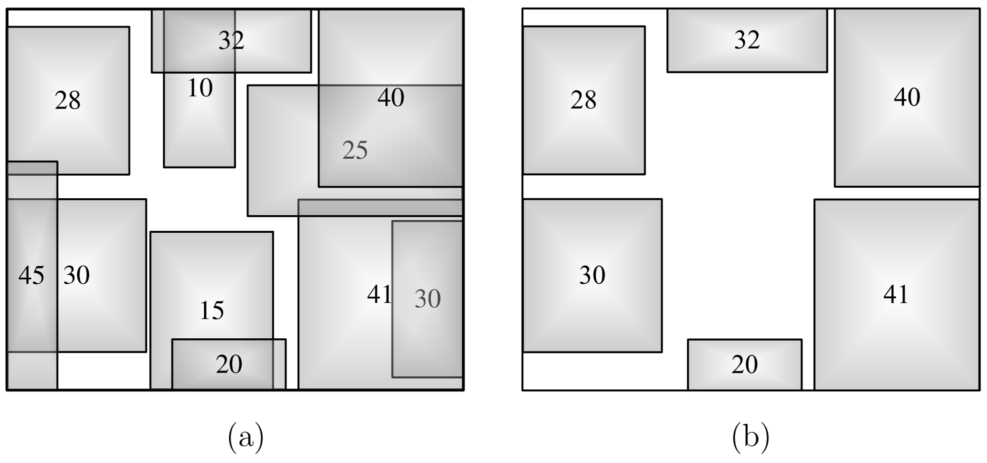


Figure 5.2: A set of boundary rectangles (a) and its maximum disjoint subset (b).

The study of finding the MDS of boundary rectangles is motivated from solving the escape routing problem on the bus level. In the past few years, as the dimensions of packages and PCBs keep decreasing and the pin counts and routing layers keep increasing, the *escape routing problem*, which is to route nets from their pins to the corresponding component boundaries, becomes more and more critical. Although a lot of escape routing algorithms [19, 38–42] have been proposed, all of them are net-centric because they ignore the bus structures. Directly applying these algorithms may mix up nets from different buses, as seen in Figure 5.3(a). However, designers always prefer nets in the same bus to be routed together without foreigners, as in Figure 5.3(b). Therefore, before escaping nets from components, it is necessary to find a subset of buses without net mixings, which is defined as the *bus escape routing problem*. Although some new approaches [24, 43, 44] about bus routing for PCBs have been presented recently, none of them can simultaneously escape buses from four component boundaries. In this paper, we will show that the optimal MDS algorithm can be applied to find the optimal solution of the bus escape routing problem.

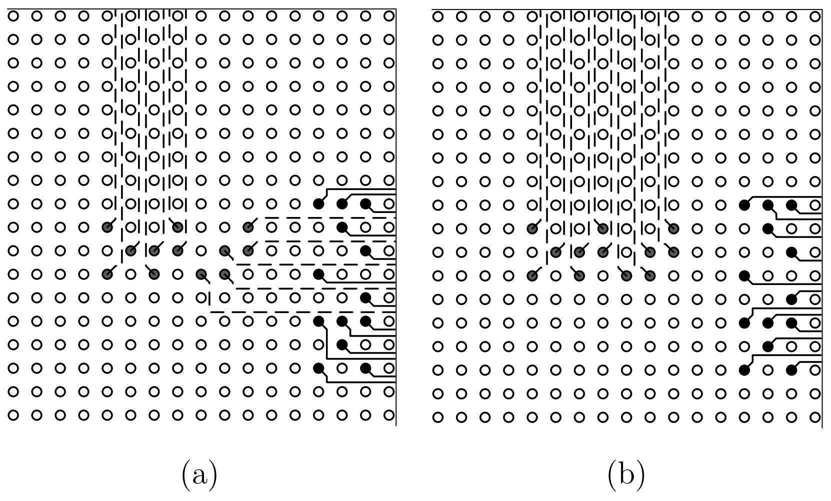


Figure 5.3: Bus structure needs to be considered in solving the maximum escape routing problem. (a) and (b) The maximum escape routing solutions of two buses with and without considering the bus structure of nets, respectively.

The rest of this chapter is organized as follows: Section 5.2 will propose an optimal algorithm to solve the MDS problem of boundary rectangles. Section 5.3 and Section 5.4 will demonstrate the experimental results and the conclusions respectively.

5.2 Optimal Algorithms

Given a rectangular region R and a set S of boundary rectangles $\{1, 2, \ldots, n\}$ lying within R, where each boundary rectangle i is associated with a weight w_i, the MDS problem of S is to find a subset of S such that chosen boundary rectangles do not overlap with each other while their total weight is maximized.

We now introduce some extra notations in order to facilitate the presentation of our algorithms. Given an MDS problem instance as defined above, we use $M(R)$ to represent the set of boundary rectangles in the optimal solution, and we use $w(M(R))$ to denote the total weight of its rectangles. Moreover, we divide all the boundary rectangles in S into four subsets according to their specified attaching boundaries, namely, S_l, S_r, S_t and S_b, and they correspond to the subsets whose boundary rectangles' attaching boundaries are, respectively, the left, right, top and bottom boundaries of the rectangular region R; thus we have $S_l \cup S_r \cup S_t \cup S_b = S$.

The most general case of the MDS problem is that all the four subsets S_l, S_r, S_t and S_b are nonempty, and we call it the 4-boundary case. There are several special or degenerate cases of the MDS problem, where some of the subsets S_l, S_r, S_t and S_b is (are) empty; thus we also have 1-boundary, 2-boundary and 3-boundary cases, in addition to the general 4-boundary case. The 1-boundary case can be easily solved by straightforward dynamic programming, so we will omit its discussion in our presentation due to the space limitation. The 2-boundary case can be subdivided into two cases: the corner case (e.g., S_l and S_b are nonempty) and the opposite case (e.g., S_l and S_r are nonempty). In this section, we first propose two polynomial time algorithms that optimally solve the corner case and opposite case, respectively. Then we show that the 3-boundary case can be decomposed into two corner case instances and one opposite case instance, so the optimal solution can be found by enumerating all possible decompositions. Finally, we solve the general 4-boundary case by decomposing it into previously solved cases.

5.2.1 The Corner 2-Boundary Case

There are four corners of the rectangular region R, so there are four different corner cases. For the ease of presentation, here we only discuss the bottom-left corner case, i.e., S_t and S_r are empty (other corner cases can be handled by a simple rotation).

We first construct a Hanan grid of on the rectangular region R as follows (an example is shown in Figure 5.4(a)):

- Extend the top and bottom boundaries of all the rectangles (including rectangle R) to get a set of horizontal cutlines (only one cutline is kept if multiple cutlines overlap). The horizontal cutlines are indexed with $1, 2, \ldots, p$ from bottom to top.

- Similarly, extend the left and right boundaries of all the rectangles (including rectangle R) to get a set of vertical cutlines. The vertical cutlines are indexed with $1, 2, \ldots, q$ from left to right.

We then construct a direct acyclic graph (DAG) based on the Hanan grid:

- We assign a vertex v_{ij} to the intersection of horizontal cutline i and vertical cutline j on the Hanan grid, $\forall i = 1, 2, \ldots, p$, $\forall j = 1, 2, \ldots, q$.

- Connectivity edges are added. For each vertex v_{ij} where $i < p$ and $j < q$, we add a directed edge from v_{ij} to the vertex immediately on its top (namely, $(v_{ij} \rightarrow v_{i+1,j})$), and add a directed edge from v_{ij} to the the vertex immediately to its right (namely, $(v_{ij}) \rightarrow v_{i,j+1})$. The weights of these connectivity edges are set to 0.

- Rectangle edges are added. For each rectangle in S_l, let horizontal cutlines t and b be its top boundary and bottom boundary, and let vertical cutline r be its right boundary; we add the set of edges $(v_{bj} \rightarrow v_{tj})$, $\forall j \in \{r, r+1, \ldots, q\}$. The weights of this set of edges are set to be the weight of this rectangle. Similarly, for each rectangle in S_b, let vertical cutlines l and r be its left boundary and right boundary, and let t be its top boundary; we add the set of edges $(v_{il} \rightarrow v_{ir})$, $\forall i \in \{t, t+1, \ldots, p\}$. The weights of this set of edges are set to be the weight of this rectangle.

An example DAG constructed for four rectangles is shown in Figure 5.4(a), where the solid edges are the connectivity edges and the dashed edges are the rectangle edges. The longest

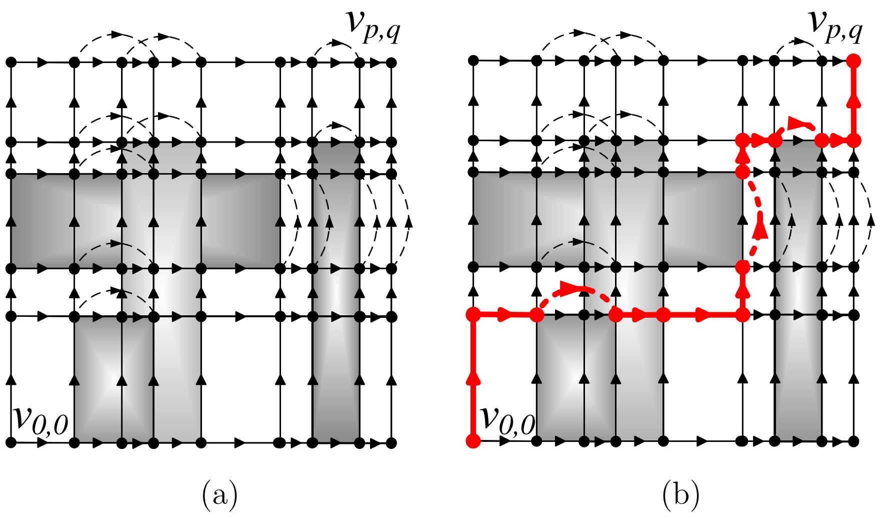


Figure 5.4: The DAG constructed on the Hanan grid of four boundary rectangles is displayed in (a). The longest path from $v_{0,0}$ to $v_{p,q}$ is highlighted in (b). The solid edges are connectivity edges and the dashed edges are the rectangle edges.

path from the bottom-left vertex $v_{0,0}$ to the top-right vertex $v_{p,q}$ on the DAG is highlighted in Figure 5.4(b). We claim that the longest path from $v_{0,0}$ to $v_{p,q}$ corresponds to the MDS of this instance; thus we have the following lemma:

Lemma 3 *Given a corner case MDS problem instance, the length of the longest path from $v_{0,0}$ to $v_{p,q}$ on the constructed DAG equals the total weight of the boundary rectangles in the optimal solution, and the optimal solution is exactly the set of rectangles corresponding to the set of rectangle edges appearing in the longest path.*

PROOF. Intuitively, any path from $v_{0,0}$ to $v_{p,q}$ can contain at most one edge among all the rectangle edges corresponding to the same rectangle. Also, it is easy to observe that any two edges corresponding to two rectangles can both appear in a path from $v_{0,0}$ to $v_{p,q}$ if and only if the two rectangles do not overlap. Thus, we can see the correspondence between a path from $v_{0,0}$ to $v_{p,q}$ and a feasible subset of rectangles: each path from $v_{0,0}$ to $v_{p,q}$ corresponds to a subset of disjoint rectangles, and a subset of disjoint rectangles corresponds to a path from $v_{0,0}$ to $v_{p,q}$ (or multiple paths of equal length). Moreover, the total weight of this set of disjoint rectangles equals the length of its corresponding path, since the length of a path is just the total weight of its rectangle edges. Therefore, the longest path from $v_{0,0}$ to $v_{p,q}$ corresponds to the optimal solution of this corner case MDS problem instance. □

Algorithm 2 MDS-Corner(R, S_l, S_b)

```
1: Construct the Hanan grid.
2: Construct the DAG from the Hanan grid
3: Find the longest path $p$ from $v_{0,0}$ to $v_{p,q}$
4: Let $M(R)$ be the set of rectangles corresponding to the rectangle edges on longest path $p$
5: return $M(R)$
```

The detailed algorithm is in Algorithm 2.

Both the Hanan grid and the DAG can be constructed in $O(n^2)$ time, where n is the total number of rectangles, as there are $O(n^2)$ vertices and $O(n^2)$ edges in the DAG. The longest path can be found in $O(n^2)$ time by using topological sorting. Thus the complexity of Algorithm 2 is $O(n^2)$ and we have the following theorem.

Theorem 3 *Algorithm 2 correctly returns the optimal solution of a corner case MDS instance in $O(n^2)$, where n is the total number of boundary rectangles.*

5.2.2 The Opposite 2-Boundary Case

There are two situations for the opposite 2-boundary case: (1) S_t and S_b are empty; (2) S_l and S_r are empty. In the following presentation, we assume, without loss the generality, situation (1) is being considered, i.e., S_t and S_b are empty, so $S = S_l \cup S_r$.

Before presenting our algorithm, we first introduce some notations and definitions. For each boundary rectangle $i \in S$, we use i^t, i^b, i^l and i^r to denote the top, bottom, left and right boundaries of rectangle i. We now define two types of contours, namely, flat contours and Z-shape contours, based on the top and bottom boundaries of the rectangles in S. The flat contours consist of $c[i^t]$ and $c[i^b]$, $\forall i \in S$, where $c[i^t](c[i^b])$ is a horizontal cutline obtained by extending the $i^t(i^b)$; and we use $R(c[i^t])(R(c[i^b]))$ to denote the subregion of R that is below $c[i^t](c[i^b])$. Figure 5.5(a) shows the $c[i^t]$ and $c[i^b]$ obtained from a boundary rectangle i. The Z-shape contours contain $c[i^t, j^b]$, $c[i^b, j^t]$ and $c[i^b, j^b]$, $\forall i \in S_l, j \in S_r$. Each Z-shape contour is obtained from a rectangle $i \in S_l$ and a rectangle $j \in S_r$, and consists of two horizontal segments and one vertical segment: one horizontal segment is obtained by extending the higher boundary until it reaches the other rectangle, the other horizontal segment is just the lower boundary, and the vertical segment simply connects the two horizontal ones. Similarly, we use $R(c[i^t, j^b])$ $(R(c[i^b, j^t]), R(c[i^b, j^b]))$ to denote the subregion of R that is below $c[i^t, j^b](c[i^b, j^t], c[i^b, j^b])$. Figure 5.5(b) shows a Z-shape contour $c[i^b, j^b]$ obtained from a rectangle $i \in S_l$ and a rectangle $j \in S_r$. Particularly, if the two rectangles overlap or if the extension of the higher boundary can never touch the other rectangle, we say this contour is invalid, e.g., $c[i^t, j^b]$ in Figure 5.5 is an invalid contour.

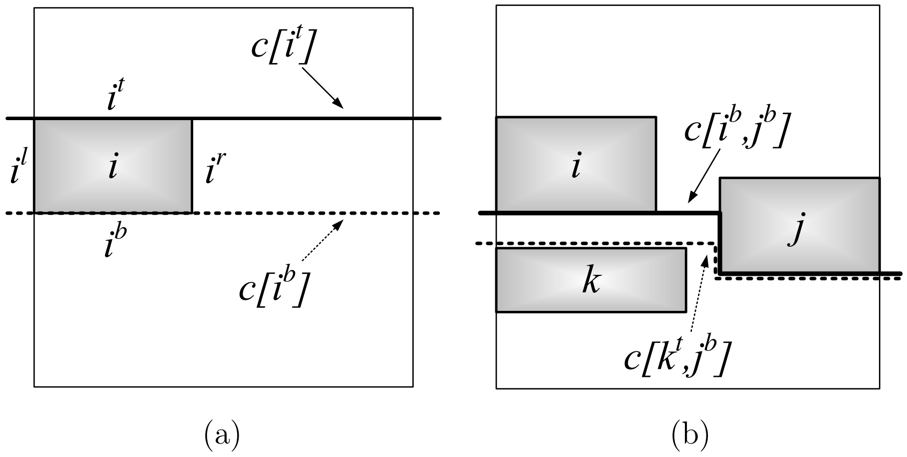
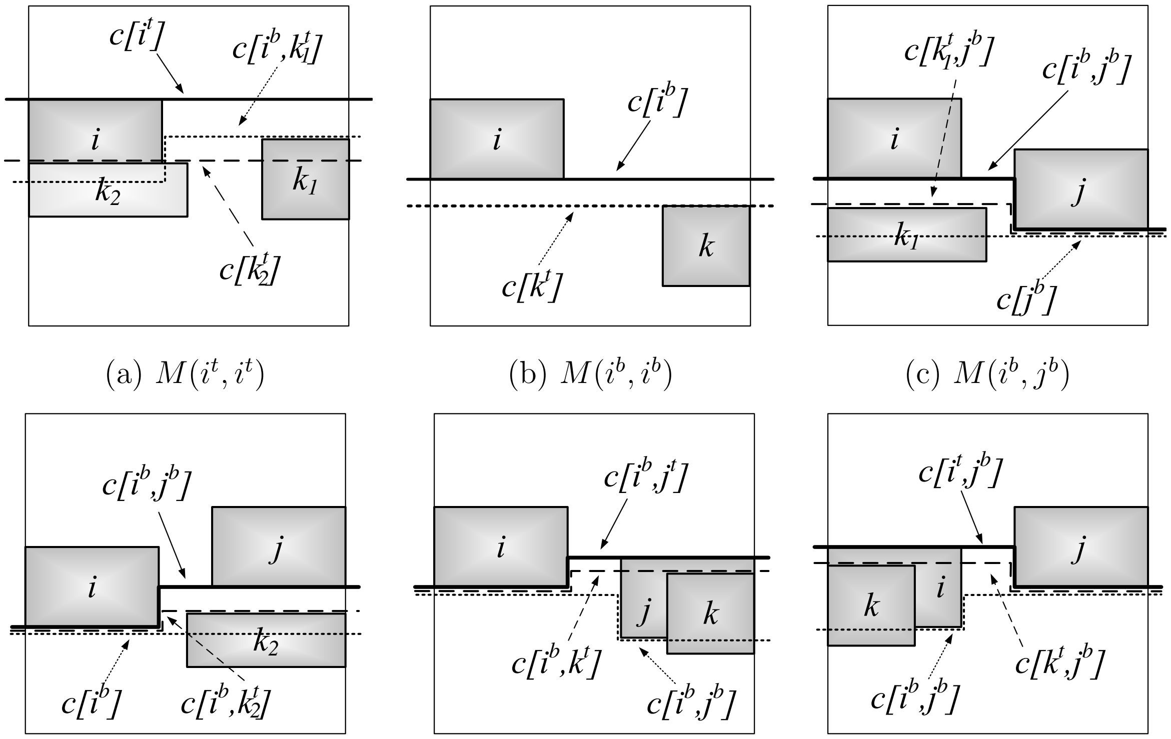


Figure 5.5: (a) Flat contours, (b) Z-shape contours.

Given a contour c and the subregion $R(c)$ defined by c, we use $M(R(c))$ to denote the optimal solution to the MDS subproblem within the subregion $R(c)$. For any two contours c_1 and c_2, we say $c_1 \leq c_2$ if and only if the region $R(c_1)$ lies completely within $R(c_2)$. Obviously, $w(M(R(c_1))) \leq w(M(R(c_2)))$ if $c_1 \leq c_2$. Based on this observation, it is easy to figure out that $M(R(c))$ can be easily computed if $M(R(c'))$ is already obtained $\forall c' \leq c$. We list below how to recursively compute $M(R(c))$ for each specific kind of contour c. In the following, we slightly abuse our notation by using $M(R(c))$ to denote both the set of rectangles in the optimal solution to the subproblem as well as the total weight of these rectangles.

(1) $M(R(c[i^t]))$ (Figure 5.6(a)): w.l.o.g., we assume $i \in S_l$. There are two cases: i is either included or not included in $M(R(c[i^t]))$. If i is included, $M(R(c[i^t])) = max\{M(R(c[i^b])) \cup \{i\}, M(R(c[i^b, k_1^t])) \cup \{i\}\}$, where k_1 is a rectangle in S_r such that $c[i^b, k_1^t]$ is the largest valid contour, if such a contour exists. If i is not included, $M(R(c[i^t]) = M(R(c[k_2^t]))$, where $c[k_2^t]$ is the largest contour that is smaller than $c[i^t]$.



(a) $M(i^t, i^t)$ (b) $M(i^b, i^b)$ (c) $M(i^b, j^b)$

(d) $M(i^b, j^b)$ (e) $M(i^b, j^t)$ (f) $M(i^t, j^b)$

Figure 5.6: Illustration on how to recursively compute the $M(R(c))$ for each kind of contour c.

(2) $M(R(c[i^b]))$ (Figure 5.6(b)): w.l.o.g., we assume $i \in S_l$. $M(R(c[i^b])) = M(R(c[k^t]))$, where $c[k^t]$ is the largest contour that is smaller than $c[i^b]$.

(3) $M(R(c[i^b, j^b]))$ (Figure 5.6(c) and (d)): If i^b is higher than j^b (Figure 5.6(c)), $M(R(c[i^b, j^b])) = \max\{M(R(c[j^b, j^b])), M(R(c[k_1^t, j^b]))\}$, where k_1 is a rectangle in S_l such that $c[k_1^t, j^b]$ is the largest valid contour that is smaller than $c[i^b, j^b]$. Things are similar if j^b is higher than i^b (Figure 5.6(d)).

(4) $M(R(c[i^b, j^t]))$ (Figure 5.6(e)): There are two cases: j is either included or not included. If j is included, $M(R(c[i^b, j^t])) = M(R(c[i^b, j^b])) \cup \{j\}$. Otherwise, $M(R(c[i^b, j^t])) = \max\{M(R(c[i^b])), M(R(c[i^b, k^t]))\}$, where k is a rectangle in S_r such that $c[i^b, k^t]$ is the largest valid contour that is smaller than $c[i^b, j^t]$.

(5) $M(R(c[i^t, j^b]))$ (Figure 5.6(f)): $M(R(c[i^t, j^b]))$ can be computed in a similar way as that of 4, since these two situations are symmetric with each other.

Naturally, the above formulas lead to a dynamic programming procedure to find the MDS of an opposite case instance. Let $i \in S$ be the rectangle with highest top boundary; then computing $M(R(c[i^t]))$ gives the optimal solution of the problem, i.e., $M(R) = M(R(c[i^t]))$. By using the standard memorization technique in dynamic programming, $M(R(c))$ is computed at most once for each contour c. Each $M(R(c))$ can be computed in $O(n)$ time, and there are $O(n^2)$ contours, so the time complexity of this procedure is $O(n^3)$. Thus we have the following theorem.

Theorem 4 *Algorithm 3 correctly returns the MDS of an opposite case instance in $O(n^3)$ time, where n is the total number of boundary rectangles.*

Algorithm 3 MDS-Opposite(R, S_l, S_r)

```
1: Let $i$ be the rectangle with highest top boundary
2: Compute $M(R(c[i^t]))$ using dynamic programming according to the formulas in (1)-(5)
3: return $M(R(c[i^t]))$
```

5.2.3 The 3-Boundary Case

Obviously, there are four situations of 3-boundary case. Without loss of generality, we assume that S_b is empty, so $S_l \cup S_t \cup S_r = S$.

The 3-boundary case can be solved by decomposing it into opposite case and corner case subproblems. There are several situations to consider.

1. If the optimal solution $M(R)$ does not contain any rectangle in S_t, this instance is degenerated to an opposite case as discussed in subsection 5.2.2. So,

$$M(R) = \mathit{Opposite}(R, S_l, S_r).$$

2. Otherwise, $M(R)$ contains at least one rectangle from S_t. Let i be the rectangle such that it has the lowest bottom boundary among all the rectangles in $M(R)$ from S_t, so the region R can be divided into three subregions R_1, R_2 and R_3 by extending i^b, as displayed in Figure 5.7(a). $M(R)$ can contain at most one rectangle from S_l and one

rectangle from S_r that overlap with the extension line of i^b, as shown in Figure 5.7(b). So we need to consider the following cases:

- No rectangle in $M(R)$ overlaps with the extension line of i^b. So, R_1 is the region above i^b and to the left of i^l, R_2 is the region above i^b and to the right of i^r, and R_3 is the region below i^b. Thus,

$$M(R) = M(R_1) \cup M(R_2) \cup M(R_3) \cup \{i\}.$$

- $M(R)$ contains only one rectangle $j \in S_l$ that overlaps with the extension line of i^b. In this case, R_1 is the region above j^t and to the left of i^l, R_2 is the region above i^b and to the right of i^r, and R_3 is the region below the contour $c[j^b, i^b]$. So,

$$M(R) = M(R_1) \cup M(R_2) \cup M(R_3) \cup \{i, j\}.$$

- $M(R)$ contains only one rectangle $k \in S_r$ that overlaps with extension line of i^b. In this case, R_1 is the region above i^b and to the left of i^l, R_2 is the region above k^t and to the right of i^r, and R_3 is the region below the contour $c[i^b, k^b]$. So,

$$M(R) = M(R_1) \cup M(R_2) \cup M(R_3) \cup \{i, k\}.$$

- $M(R)$ contains both a rectangle $j \in S_l$ and a rectangle $k \in S_r$ that overlap with the extension line of i^b. In this case, R_1 is the region above j^t and to the left of i^l, R_2 is the region above k^t and to the right of i^r, and R_3 is the region below the contour $c[j^b, k^b]$. So,

$$M(R) = M(R_1) \cup M(R_2) \cup M(R_3) \cup \{i, j, k\}.$$

Our algorithm for the 3-boundary case basically enumerates all the $O(n^3)$ possible decompositions and returns the best found solution. Before the enumeration step starts, the solutions to the corner case and opposite case subproblems are pre-computed and stored into

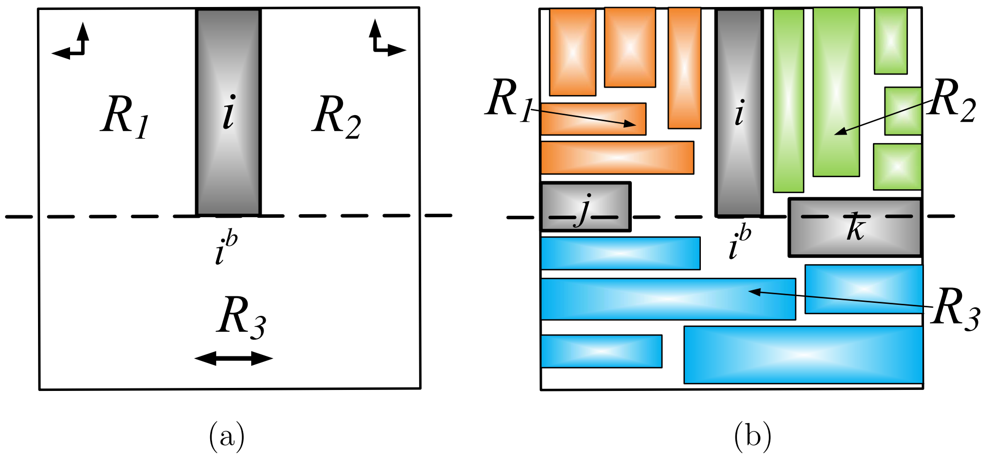


Figure 5.7: R is divided into three regions: R_1, R_2 and R_3, where R_1 and R_2 are two corner case subproblems, R_3 is an opposite case subproblem.

a hashing table, so that evaluating each decomposition solution can be done in $O(1)$ time[5]. The pseudocode of the algorithm for the 3-boundary case is listed below. Theorem 5 can be concluded based on the above analysis.

Algorithm 4 MDS-3side(R, S_l, S_t, S_r)

```
1: Solve the top-left corner case subproblem
2: Solve the top-right corner case subproblem
3: Solve the left-right opposite case subproblem
4: Enumerate all possible decompositions
5: return The best found solution
```

Theorem 5 *Algorithm 4 correctly returns the MDS of a 3-boundary case in $O(n^3)$ time, where n is the total number of boundary rectangles.*

5.2.4 The 4-boundary Case

We first analyze all the possible structures of the optimal solution $M(R)$. Let i_l, i_t, i_r and i_b be the tallest boundary rectangles in $M(R)$ standing on the left, top, right and bottom boundaries of the rectangular region of R, respectively. There are several situations to take into account.

[5]The opposite case subproblem here is slightly different from the basic version we have discussed, as here we need to add some extra contours for each $i \in S_t$ (e.g., $c[j^b, i^b]$, where $j \in S_l$), in order to pre-compute the solutions to all the opposite case subproblems in region R_3.

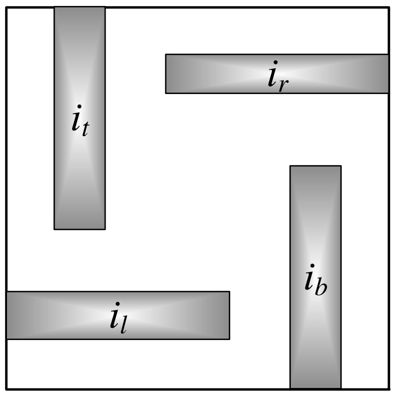
1. **Degeneracy** - If any of i_l, i_t, i_r and i_b does not exist, then the 4-boundary instance degenerates to a 3-boundary case and we can find its optimal solution by applying the 3-boundary case solver in Section 5.2.3.

2. **Non-wheel structure** - We say $M(R)$ contains a non-wheel structure if either the bottom boundary of i_t is higher than the top boundary of i_b ($i_t^b \geq i_b^t$), or the left boundary of i_r is to the right of the right boundary of i_l ($i_r^l \geq i_l^r$). We assume, without loss of generality, $i_t^b \geq i_b^t$. We can then divide R into three subregions R_1, R_2 and R_3 by extending the bottom boundary of i_t, which is very similar to the situation shown in Figure 5.7. The only difference is that the subproblem in subregion R_3 is a 3-boundary case, which can be solved by calling the 3-boundary case solver.

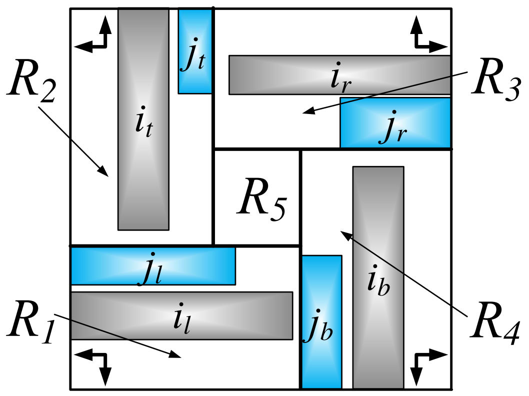
3. **Wheel structure** - When $i_l^r > i_r^l$ and $i_d^t > i_u^b$, $M(R)$ is a wheel structure, as shown in Figure 5.8(a). The wheel structure has two possible orientations: (1) clockwise, where i_l is above i_r and i_t is to the left of i_b; and (2) counter-clockwise, where i_l is below i_r and i_t is to the right of i_b. Here we only discuss the counter-clockwise orientation as shown in Figure 5.8(b). Let $j_b \in S_b$ be the leftmost rectangle that is to the right of i_l in $M(R)$ (note that j_b can be i_b). Let $j_l \in S_l$ be the highest rectangle that is below i_t in $M(R)$. Similarly, we define rectangles j_t and j_r, as shown in Figure 5.8. We then extend j_b^l until it reaches j_r^b, extend j_r^b until it reaches j_t^r, extend j_t^r until it reaches j_l^t, and extend j_l^t until it reaches j_b^l. Now the whole region R is divided into five subregions R_1 to R_5, where R_1, R_2, R_3 and R_4 correspond to four corner case subproblems, and R_5 is dead space in the center. Therefore, if $M(R)$ contains a wheel structure, the instance can be decomposed into four corner case subproblems, so

$$M(R) = M(R_1) \cup M(R_2) \cup M(R_3) \cup M(R_4).$$

In summary, the MDS of the 4-boundary case can be computed by enumerating all the possible degenerate situations, and all the non-wheel and wheel structures. The pseudocode of algorithm MDS-4side is listed below. There are $O(n^4)$ possible wheel structures to enumerate, where each enumerated wheel structure needs to compute four corner case subproblems



(a)



(b)

Figure 5.8: (a) a counter-clockwise wheel structure of $M(R)$; (b) the whole region R is divided into 5 subregions.

in $O(n^2)$ time. There are $O(n^3)$ non-wheel structures to consider, each of which needs to compute one 3-boundary case subproblem in $O(n^3)$ time and two corner case subproblems in $O(n^2)$ time. Therefore, the time complexity of algorithm MDS-4side is $O(n^6)$. Note that for this algorithm, preprocessing steps cannot help to reduce the order of magnitude of its time complexity, since the 3-boundary case subproblems cannot be computed incrementally like the 2-boundary cases. The following theorem can be easily concluded based on the above analysis.

Algorithm 5 MDS-4side (R, S_l, S_r, S_t, S_b)

```
1: Compute all the degenerated 3-boundary cases
2: Enumerate and evaluate all possible non-wheel structures
3: Enumerate and evaluate all possible wheel structures
4: return The best found solution
```

Theorem 6 *Algorithm 5 correctly returns the MDS of a 4-boundary case in $O(n^6)$ time, where n is the total number of boundary rectangles.*

5.3 Experimental Results

We implemented our optimal MDS algorithm in C++, and applied it to eight bus escape routing cases from the real industrial data. All the experiments are performed on a Linux

Table 5.1: Experimental results.

Test Cases	Input		Output			
	Bus #	Rectangle #	Bus #	Rectangle #	Net #	Time (s)
Ex1	8	32	8	8	272	0.1
Ex2	8	32	8	8	680	0.1
Ex3	11	44	10	10	629	1.2
Ex4	13	52	3	3	66	1.3
Ex5	14	56	6	6	276	1.3
Ex6	43	172	15	15	655	159
Ex7	44	176	15	15	659	184
Ex8	64	256	22	22	288	1148

workstation with two 3.0 GHz Intel Xeon CPUs and 4 GB memory. Table 5.1 shows experimental results. In our experiments, four rectangles are created for each bus, whose escaping directions are respectively left, right, top and bottom. For each rectangle from a specific bus, its weight is set to be the maximum number of nets that can escape along its direction. In Table 5.1, the first two columns give the number of input buses as well as the number of input boundary rectangles. The third and the fourth columns show the number of selected rectangles in the maximum disjoint subset as well as the number of different buses corresponding to these rectangles. The sixth column gives the total weight of selected boundary rectangles, which is also the total number of nets in chosen buses. The last column gives running times of these test cases. From Table 5.1, it is clear that although the complexity of our MDS algorithm is $O(n^6)$, where n is the number of bus boundary rectangles, because there are not so many buses in one component, the real running time of our algorithm is still acceptable.

5.4 Conclusions

In this chapter, we proposed a polynomial time algorithm to optimally solve an open problem, the maximum disjoint subset problem of boundary rectangles. We then showed this algorithm can be applied to find the optimal solution of the bus escape routing problem.

CHAPTER 6

ESCAPE ROUTING

6.1 Introduction

As was pointed out in previous chapters, the PCB routing problem can be decomposed into two subproblems: the escape routing, which is to route nets from pin terminals to component boundaries, and the area routing, which is to route nets between component boundaries. In this chapter, we will propose a fast escape routing algorithm.

In high-end PCBs nets are always organized in bus structures. Nets in a bus have similar pin locations and they are expected to be routed together without foreigners in between. From industrial manual routing solutions we observe that nets in a bus always escape a component from its projection interval, which can be obtained by projecting its pin cluster bounding box onto the component boundary along its escape direction, as shown in Figure 6.1.

Moreover, within components, nets are typically routed in an L-shape manner: nets always escape components monotonically and each net bends at most once inside each component, as shown in Figure 6.2.

Based on these observations, we propose a wire packing based escape routing algorithm, which only takes a very short time to route nets from their terminals to component boundaries in an L-shape manner. We tested it on a state-of-the-art industrial board, which contains 7000+ nets and 146 buses. It successfully routed 98.2% nets in less than 3 minute CPU time. All other escape routers [8,9,42] which can provide comparable routing solutions have much longer running time than ours.

In the rest of this chapter, Section 6.2 will introduce the L-shape escape routing algorithm. Section 6.3 will introduce the applications of this L-shape routing algorithm in the whole

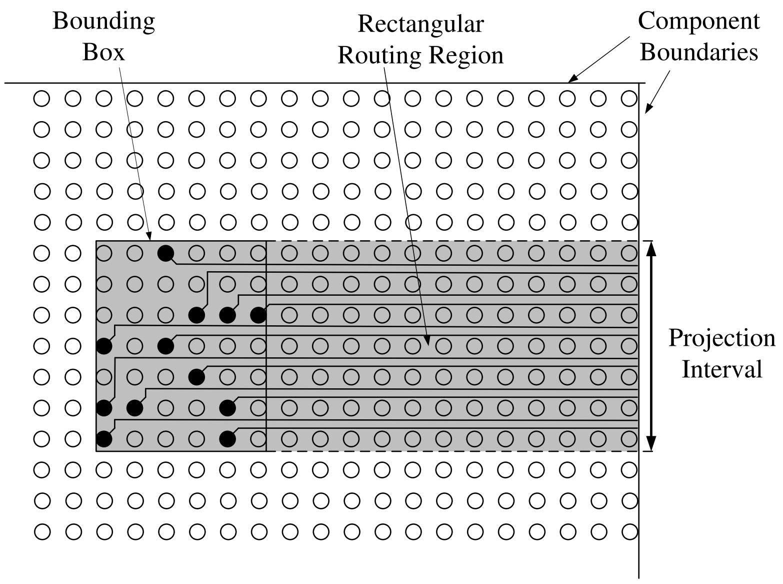


Figure 6.1: Bus projection interval.

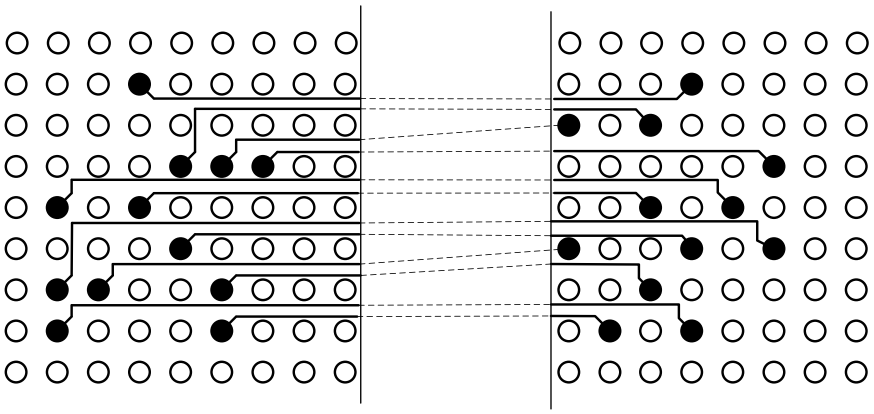
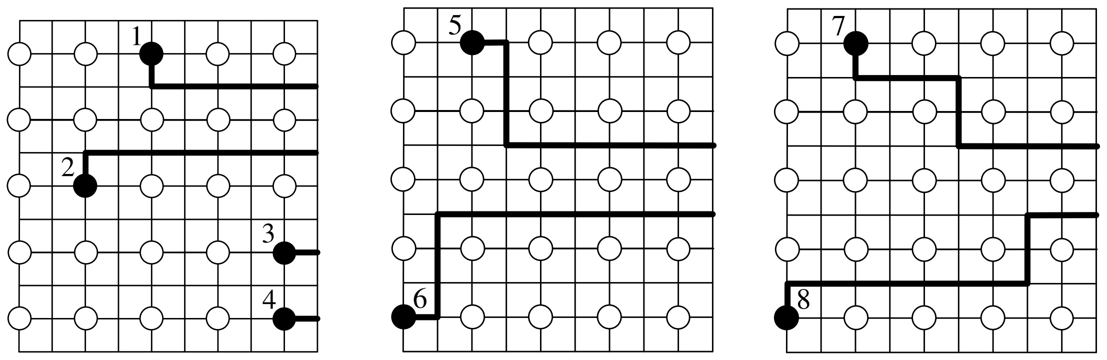


Figure 6.2: L-shape escape routing.

PCB routing process. Section 6.4 will show some experimental results and Section 6.5 will give some concluding remarks.

6.2 Algorithm

As shown in Figure 6.1, nets in a bus occupy a rectangular region to escape a component, which can be easily obtained by extending the bus pin cluster bounding box to the component boundary along the bus escape direction. We assume that each bus escape routing region covers a regular pin grid and it involves an underlying routing grid as follows: each horizonal line is either a pin row or a horizontal routing track and each vertical line is either a pin column or a vertical routing track, where each horizontal or vertical routing track allows only one net to pass. Except for nets whose pins are located on the component boundary, all other nets can only escape components through routing tracks. Figure 6.3 demonstrates a routing grid of an escape routing region for a bus whose escape direction is right. In this example, the routing region contains a 5×5 pin grid and there is only one routing track between any two neighboring pin rows or pin columns. According to the bus escape direction, the right boundary of the routing region is also the component boundary. Therefore all nets, except nets 3 and 4 whose pins are on the component boundary, can only escape components from horizonal routing tracks.



(a) No Bending (b) One Bending (c) Two Bending

Figure 6.3: L-shape and non-L-shape escape routes.

The escape route of a net in the routing grid is its path escaping the routing region. If a net escape route only passes at most one horizontal routing track and at most one vertical routing track, it is defined as an L-shape escape route. Figure 6.3(a) and (b) demonstrate some L-shape escape routes and Figure 6.3(c) demonstrates some escape routes which are in L-shape. In Figure 6.3(a), net 1 as well as net 2 only passes one horizontal track and net 3 and net 4 only pass their pin rows. In Figure 6.3(b), net 5 passes only one horizontal track and only one vertical track, as well as net 6. In Figure 6.3(c), both nets 7 and 8 pass two horizontal tracks.

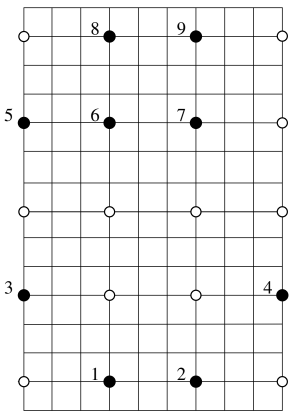
In PCB designs, each bus is between two components and all their nets must be routed in a planar fashion. Therefore, each bus must escape two components simultaneously. In the rest of this section, we will first introduce an algorithm which escapes as many nets as possible in an L-shape manner from one component. Then we will extend it to another algorithm which escapes nets from two components simultaneously.

6.2.1 One Component L-Shape Algorithm

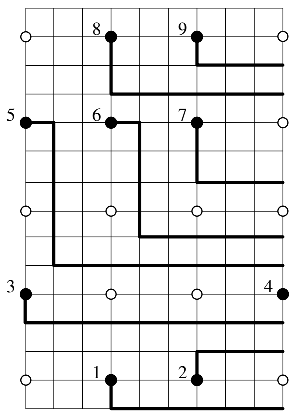
Given a bus with K nets $\{1, 2, \cdots, K\}$ and a rectangular routing region C with an $M \times N$ routing grid (M rows and N columns), let the coordinate of the bottom-left corner of the routing grid be (1,1), the coordinate of the upper-right corner be (N, M) and the pin coordinate of net i be (x_i, y_i) $(1 \leq i \leq K)$. Without loss of generality, we assume the escape direction is "right," which means that nets are only allowed to escape C from its right boundary. For a bus with other escape direction we can rotate its routing region to make its escape direction to be "right." Figure 6.4(a) demonstrates an example which is composed of a 9-net bus and a 15×10 routing grid.

We perform a two-step wire packing based algorithm to find an L-shape routing solution for the given K nets as follows: We first sort all nets according to their pin locations (the net sorting step), and then route nets in sequence (the net routing step). In the net sorting step, we sort nets into two orderings: the "up" ordering and the "down" ordering. The "down" ordering can be obtained by sweeping the pin grid row by row from bottom to top and counting net pins from left to right in each row. It is denoted as $\{D(1), D(2), \ldots, D(K)\}$, where $1 \leq D(j) \leq K$. For the example shown in Figure 6.4(a), its "down" net ordering is $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Similarly, the "up" ordering can be obtained by sweeping the pin grid row by row from top to bottom and counting net pins from left to right in each row. It is denoted as $\{U(1), U(2), \ldots, U(K)\}$, where $1 \leq U(j) \leq K$. The "up" net ordering for the example in Figure 6.4(a) is $\{8, 9, 5, 6, 7, 3, 4, 1, 2\}$.

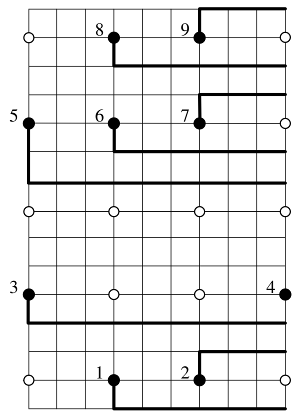
In the net routing step, we route given nets twice: one follows the "up" ordering and the other follows the "down" ordering; and we keep the routing solution which contains more routed nets. When net $D(i)$ is processed during routing nets along the "down" ordering, if its pin is on the right boundary of the routing grid, as net 4 in Figure 6.4(b), it escapes C through its pin row directly and does not pass any horizontal or vertical routing track. If



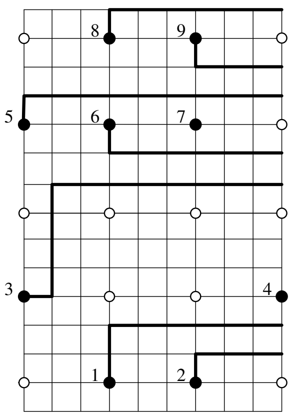
(a) Example



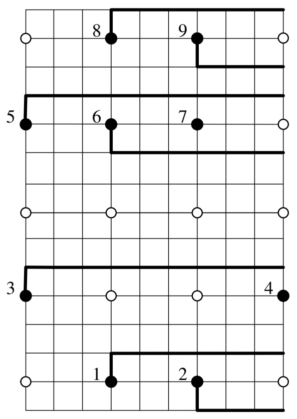
(b) "down" routing



(c) "down" compaction



(d) "up" routing



(e) "up" compaction

Figure 6.4: A single component L-shape escape routing example with nine nets is displayed in (a) and its corresponding solutions by applying the one component escape routing algorithm in "down" and "up" net orderings are demonstrated in (b) and (d). (c) and (e) The compacted routing solutions corresponding to (b) and (d).

$D(i)$'s pin is not on the right boundary of C, it must pass a horizontal track if it is routable. We define a feasible escape route of $D(i)$ as an L-shape escape route which satisfies that $\forall$ $j \in [1, i-1]$, if $D(j)$ is routed, then:

- It does not intersect with $D(j)$'s escape route.
- The horizontal track it passes is above the horizontal track $D(j)$'s route passes.

If there is no feasible escape route, $D(i)$ cannot be routed; otherwise, we keep the feasible escape route whose horizontal routing track has the maximum vertical coordinate. Figure 6.4(b) gives the L-shape escape routing solution of the example in Figure 6.4(a) along the "down" net ordering, where all nine nets are routed. Similarly, when we process $U(i)$ during routing nets along "up" ordering, it escapes C though its pin row directly if its pin is on the right boundary of the routing grid; otherwise it escapes C along its feasible escape route whose horizontal routing track has the minimum vertical coordinate. Figure 6.4(d) gives the L-shape escape routing solution of the example in Figure 6.4(a) along the "up" net ordering, where only 8 of 9 nets are routed. For this example, we keep the escape routing solution along "down" net ordering because it contains more routed nets.

It is easy to observe from Figure 6.4(b) that the total route length can be diminished if some nets, such as net 7, escape C from higher horizontal routing tracks. Similarly, as shown in Figure 6.4(d), the total route length can be minimized if some nets, such as net 3, escape C from lower horizontal routing tracks. Therefore, we perform a compaction process at the end of the net routing step to minimize the total length of the L-shape escape routing solution.

When compacting the routing solution obtained along the "down" net ordering, we process nets from $D(K)$ to $D(1)$. When we process $D(i)$, if its pin is on the right boundary of the pin grid, its escape route cannot be shortened. Otherwise if it is routed, we reroute it along its feasible escape route which passes the horizontal routing track closest to its pin row. Figure 6.4(c) compacts the escape routing solution in Figure 6.4(b). Similarly, we process nets from $U(K)$ to $U(1)$ when compacting the escape routing solution obtained along the "up" net ordering. If $U(i)$ is routed and does not escape the routing grid through its pin row, we also reroute it along its feasible escape route which passes the horizontal routing track closest to its pin row. Figure 6.4(e) compacts the escape routing solution in Figure 6.4(d). For the example in Figure 6.4(a), Figure 6.4(c) displays its final escape routing solution.

6.2.2 Two Component L-Shape Algorithm

We can extend the one component L-shape escape algorithm to escape nets from two components simultaneously. Suppose that a bus with K nets $\{1, 2, \cdots, K\}$ is between two components and its routing regions within these two components are C_1 and C_2. Without loss of generality, we assume that C_1 is on the left of C_2 and nets escape C_1 and C_2 along the right and left directions respectively. Let the coordinate of the lower left corner of the routing grid in C_1 be $(1, 1)$ and the coordinate of its upper right corner be (M_1, N_1). Similarly, let the coordinate of the lower left and upper right corners of the routing grid in C_2 be $(1, 1)$ and (M_1, N_1), respectively. Figure 6.5 gives an example containing an 11-net bus. For a bus whose routing region positions or escape directions do not satisfy the above assumptions, we can relocate and rotate its routing regions to make assumptions satisfied. The routability of a bus remains the same before and after relocating and rotating its routing regions.

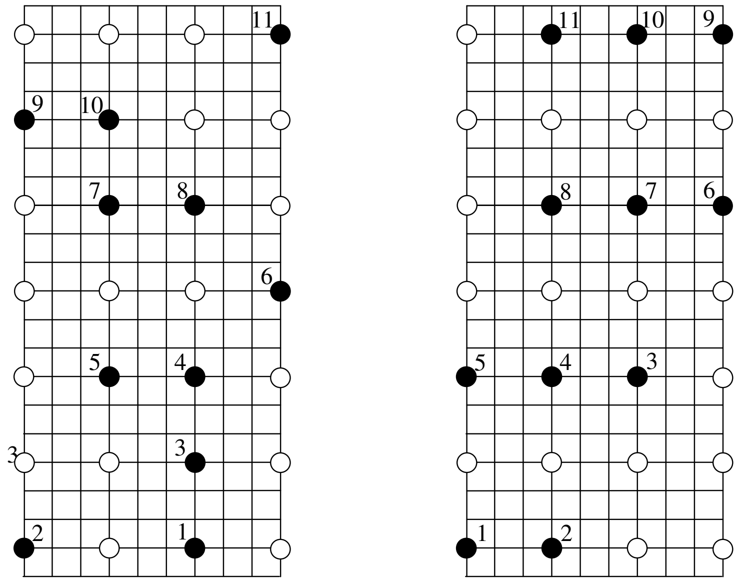


Figure 6.5: A two component escape routing example.

During the net sorting step, nets are sorted into four orderings according to their pin locations in C_1 and C_2. From C_1 nets are sorted into two orderings, the left-up (LU) ordering and left-down (LD) ordering:

- Left-up (LU) ordering

  The LU ordering can be obtained by sweeping the pin grid of C_1 row by row from top to bottom and couting net pins from left to right in each row, denoted by $\{LU(1), LU(2), \cdots, LU(K)\}$, where $1 \leq LU(i) \leq K$. The LU ordering of nets in Figure 6.5 is $\{11, 9, 10, 7, 8, 6, 5, 4, 3, 2, 1\}$.

- Left-down (LD) ordering

  The LD ordering can be obtained by sweeping the pin grid of C_1 row by row from bottom to top and counting net pins from left to right in each row, denoted by $\{LD(1), LD(2), \cdots, LD(K)\}$, where $1 \leq LD(i) \leq K$. The LD ordering of nets in Figure 6.5 is $\{2, 1, 3, 5, 4, 6, 7, 8, 9, 10, 11\}$.

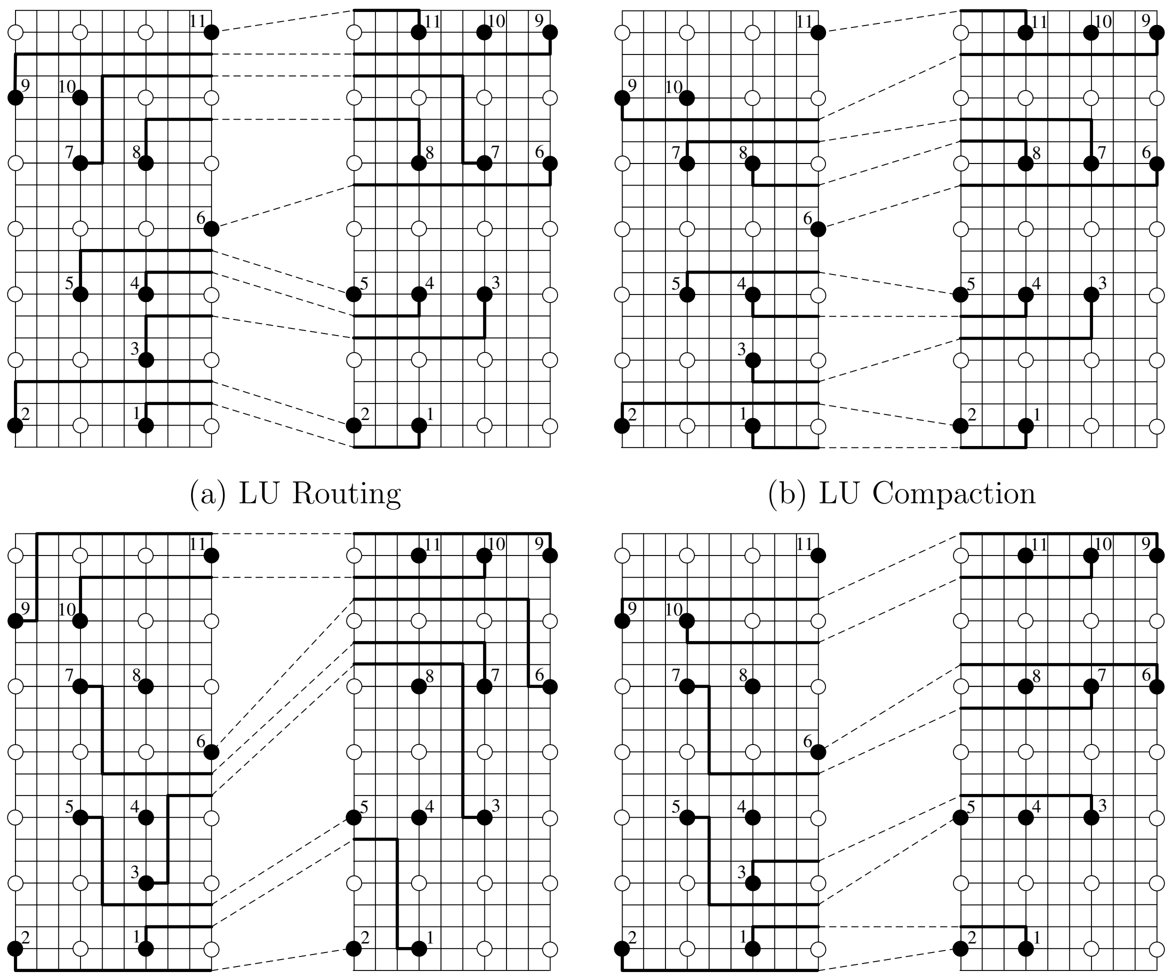
By applying the similar approach, from C_2 nets are sorted into the right-up (RU) ordering and right-down (RD) ordering:

- Right-up (RU) ordering
  The RU ordering can be obtained by sweeping the pin grid of C_2 row by row from top to bottom and counting net pins from right to left in each row, denoted by $\{RU(1), RU(2), \cdots, RU(K)\}$, where $1 \leq RU(i) \leq K$. The RU ordering of nets in Figure 6.5 is $\{9, 10, 11, 6, 7, 8, 3, 4, 5, 1, 2\}$.

- Right-down (RD) ordering
  The RD ordering can be obtained by sweeping the pin grid of C_2 row by row from bottom to top and counting net pins from right to left in each row, denoted by $\{RD(1), RD(2), \cdots, RD(K)\}$, where $1 \leq RD(i) \leq K$. The RD ordering of nets in Figure 6.5 is $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$.

In the net routing step we route all given nets once along each of four routing orderings and keep the escape routing solution containing the maximum number of routed nets. When we route net $LU(i)$ along the LU ordering, we regard the LU ordering as the "up" ordering in each region and escape $LU(i)$ from C_1 and C_2 respectively. If $LU(i)$ cannot escape either C_1 or C_2, it cannot be routed. After all nets are processed we can improve the escape routing solution by compacting net routes in C_1 and C_2 respectively. The process of routing nets along the RU ordering is identical to the process of routing nets along the LU ordering. For the example in Figure 6.5, 10 of 11 nets are routed along the LU ordering, as displayed in Figure 6.6(a), and 8 of 11 nets are routed along the RU ordering, as displayed in Figure 6.6(c). Figure 6.6(b) and (d) display the compacted solutions in Figure 6.6(a) and (c), respectively.

When we process net $LD(i)$ along the LD ordering, we regard the LD ordering as the "down" ordering in each region and escape $LD(i)$ from C_1 and C_2 respectively. Only when both escape routes can be obtained, can $LD(i)$ be routed. After processing all nets the escape



(a) LU Routing

(b) LU Compaction

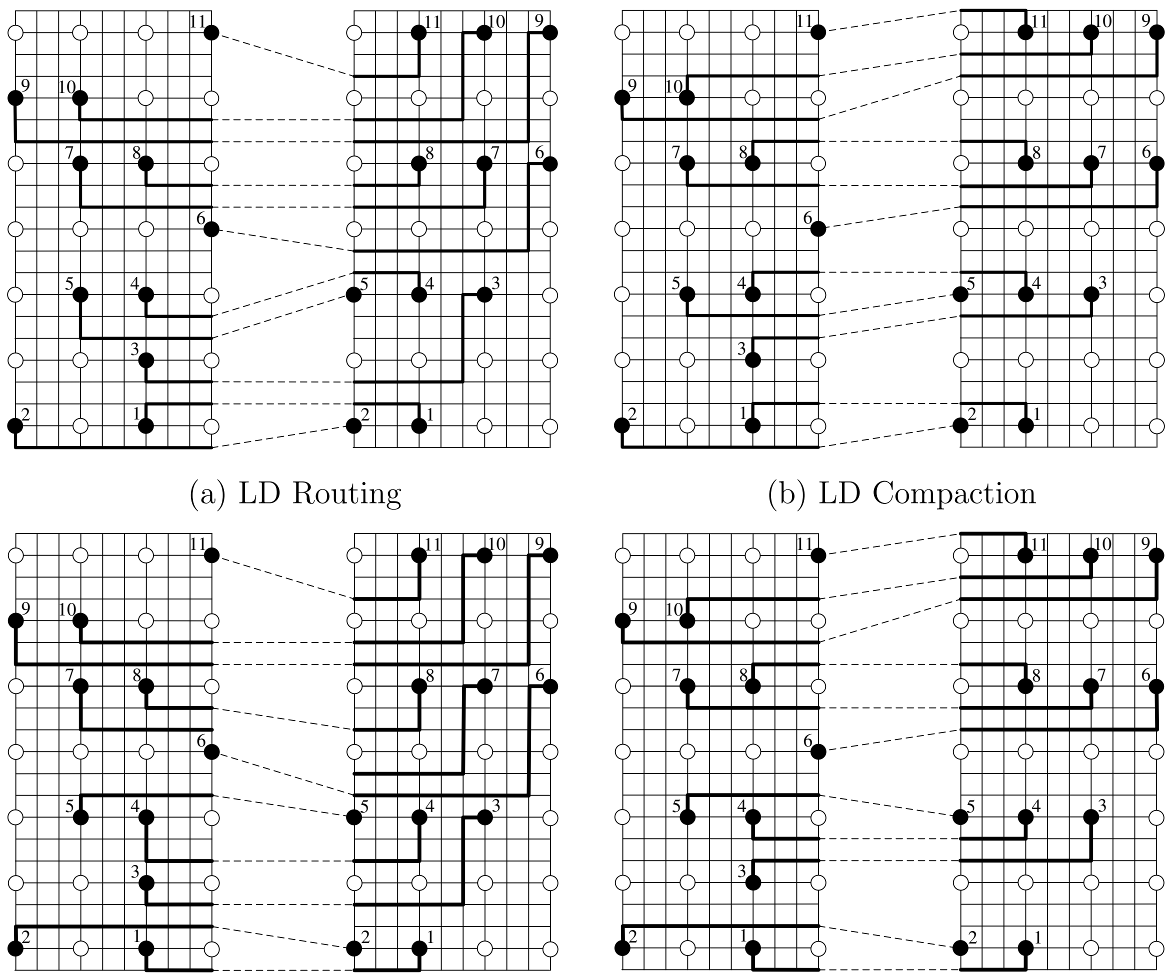
(c) RU Routing

(d) RU Compaction

Figure 6.6: Escape routing and compaction along LU and RU orderings.

routing solution can be improved by compacting net routes in C_1 and C_2 respectively. We can follow the same procedure to route nets along the RD ordering. For the example in Figure 6.5, all 11 nets are routed along the LD and RD orderings, as displayed in Figure 6.7(a) and (c). Figure 6.7(b) and (d) display the compacted solutions in Figure 6.7(a) and (c), respectively.

Therefore, for the example in Figure 6.5 we can select the escape routing solution obtained either along the LD ordering or along the RD ordering to be its final solution.


(a) LD Routing

(b) LD Compaction

(c) RD Routing

(d) RD Compaction

Figure 6.7: Escape routing and compaction along LD and RD orderings.

6.2.3 Escape Ordering Adjustments

In practice, an high-end PCB always contains a lot of differential pairs. A differential pair is a pair of nets which transfer a pair of complementary signals. Within components two nets in a differential pair are required to escape components from neighboring routing tracks, and outside components they are required to be routed together all the time. Figure 6.8 demonstrates two differential pairs. Directly applying our L-shape escape routing algorithms may violate the routing constraints of differential pairs. In this section, we will discuss how to adjust routing orderings in the net routing step.

In addition, in the net routing step we observe that more nets can be routed if we adjust the net ordering. For example, in Figure 6.6(a) if we process net 10 before net 9, all 11

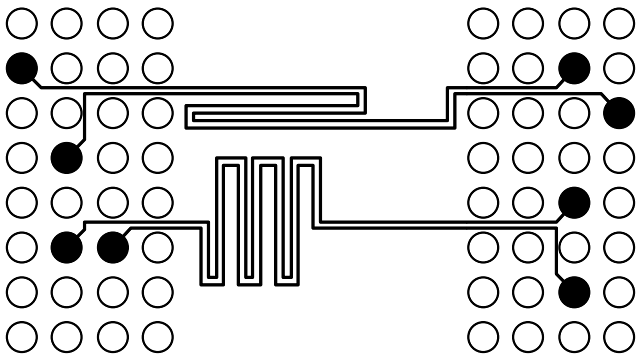


Figure 6.8: Examples of differential pairs.

nets can be routed. In this section we will discuss the net ordering adjustments under some specific circumstances.

Differential Pair

Let n_1 and n_2 represent two nets in a differential pair, where $1 \leq n_1, n_2 \leq K$. Then when we process n_1 in the net routing step, if n_2 has not been processed, we must adjust the escape ordering to make n_2 become the next to-be-processed net after n_1, and in both C_1 and C_2 we escape n_1 through a horizontal track whose neighboring track has not been taken. If n_2 has been routed, n_1 must escape C_1 and C_2 though horizontal routing tracks neighboring to n_2's track. If either n_1 or n_2 is not routed, both of them fail escape routing.

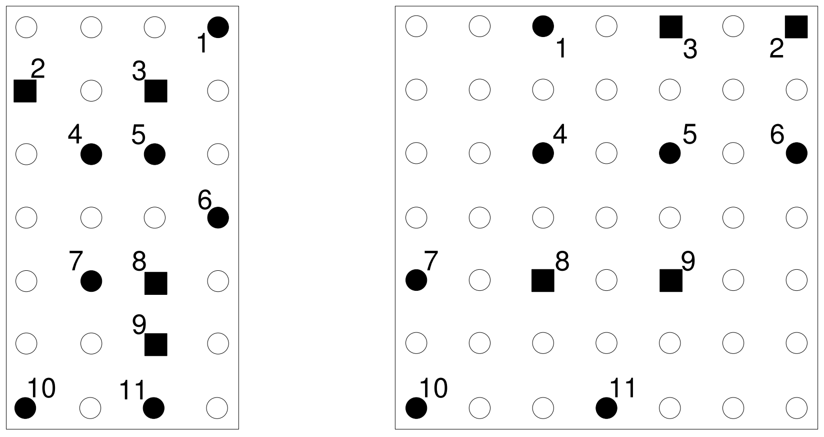


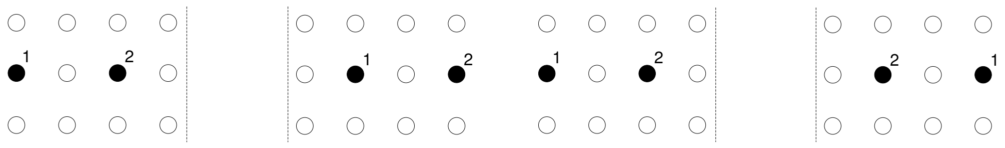
Figure 6.9: An escape routing example with differential pairs.

Figure 6.9 demonstrates a bus with seven single nets and two differential pairs, where nets 2 and 3 belong to one differential pair and nets 8 and 9 belong to another one. In

Figure 6.9 rectangles represent pins of differential pairs. The LD ordering of this example is 10,11,9,7,8,6,4,5,2,3,1, where nets 8 and 9 are separated. Therefore we adjust the LD ordering to be 10,11,9,8,7,6,4,5,2,3,1.

Same-Row Net Pair

For any two given nets n_1 and n_2, where $1 \leq n_1, n_2 \leq K$, if their pins are located in the same pin row in both C_1 and C_2, they are named a same-row net pair. The are two kinds of same-row net pairs: the same-ordering net pair as shown in Figure 6.10(a), where n_1's pins are always on the left of n_2's pins or vise versa in both routing regions, and the reverse-ordering net pair as shown in Figure 6.10(b), where n_1's pins are on different sides of n_2's pins in C_1 and C_2.

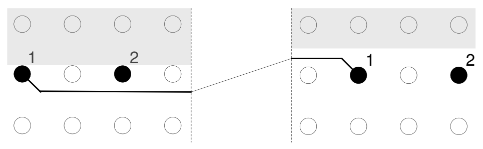


(a) Same-Ordering (b) Reverse-Ordering

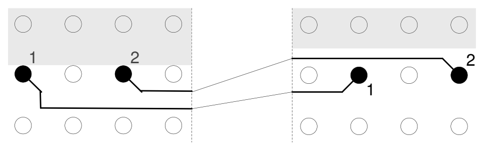
Figure 6.10: Same-ordering same-row net pair and the reverse-ordering same-row net pair.

Without loss of generality, we assume that n_1's pin is always located on the left side of n_2's pin in C_1. If n_1 and n_2 are a same-ordering net pair, n_1 is ahead of n_2 in both LU and LD net orderings. When n_1 is to be routed along the LU ordering, if the horizontal routing track which is next to and above n_1's pin row is occupied in C_1 and it is available in C_2, then n_2 is unable to escape C_1, as shown in Figure 6.11(a). In this case, both n_1 and n_2 can be routed if we swap their routing ordering as shown in Figure 6.11(b). Similarly, when n_1 is to be routed along the LD ordering, if the horizontal routing track which is next to and below n_1's pin row is occupied in C_1 but it is still available in C_2, then swapping their routing ordering makes both of them routed, as shown in Figure 6.11(c) and (d). Under similar circumstances when we route nets along RU and RD orderings, swapping n_1 and n_2's ordering enables both of them to be routed.

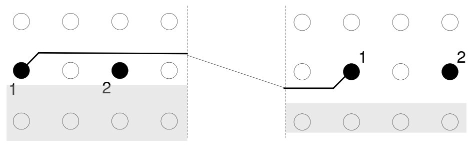
If n_1 and n_2 is a reverse-ordering net pair, n_1 is ahead of n_2 in LU and LD net orderings. When n_1 is to be routed along the LU ordering, if the horizontal routing track which is next



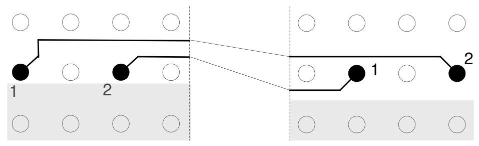
(a) LU solution before swap



(b) LU solution after swap

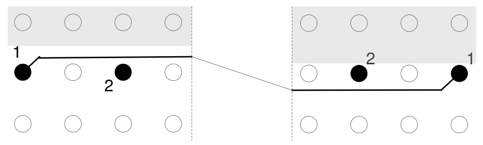


(c) LD solution before swap

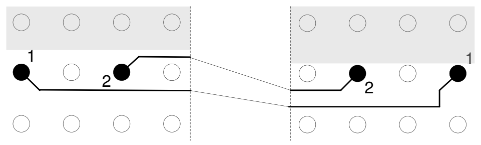


(d) LD solution after swap

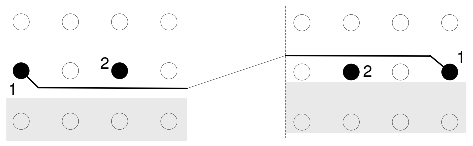
Figure 6.11: Routing ordering swap in LU and LD orderings for a same-ordering net pair.



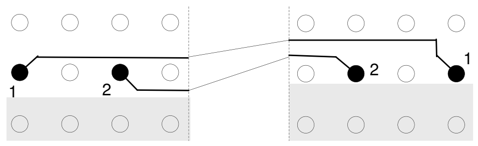
(a) LU solution before swap



(b) LU solution after swap



(c) LD solution before swap

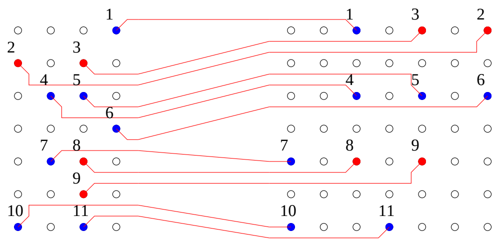


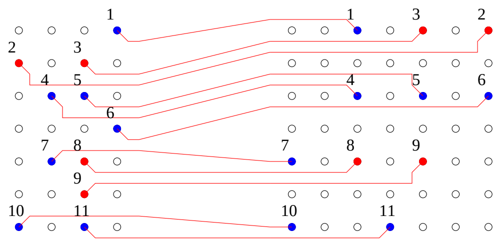
(d) LD solution after swap

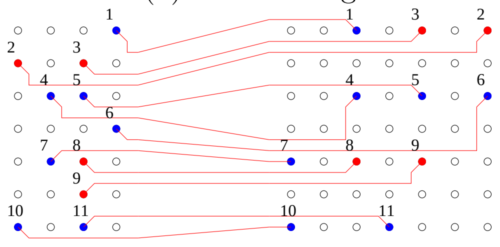
Figure 6.12: Routing ordering swap in LU and LD orderings for a reverse-ordering net pair.

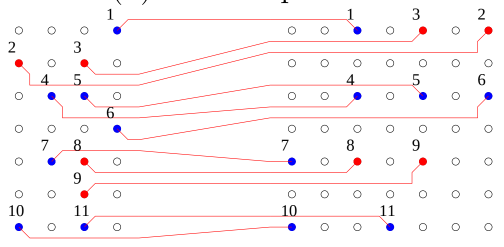
to and above n_1's pin row in C_2 is occupied and it is not taken in C_1, then n_2 fails escaping C_2 as shown in Figure 6.12(a). In this case, swapping n_1 and n_2's ordering can make both of them routed as shown in Figure 6.12(b). Similarly, when n_1 is to be routed along the LD ordering, if the horizontal routing track which is next to and below n_1's pin row is occupied in C_2 and it is available in C_1, then n_2 is unable to escape C_1 as shown in Figure 6.12(c). If we swap the ordering of n_1 and n_2, both of them can be routed as shown in Figure 6.12(d). Under similar circumstances when routing nets along RU and RD orderings, swapping n_1 and n_2's ordering enables both of them to be routed successfully.

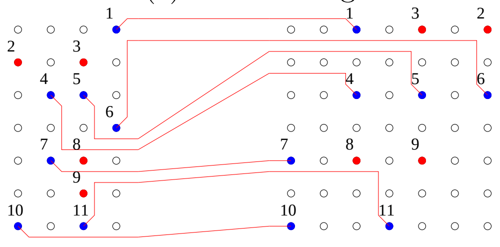
Figure 6.13 gives the LU, LD, RU and RD escape routing solutions for the case in Figure 6.9. All 11 nets are routed along LU and LD orderings, and only 7 and 9 nets are routed along RU and RD ordering respectively. From Figure 6.13(a), it can be observed that nets 4 and 5 are a same-ordering same-row net pair and their routing orderings are swapped.

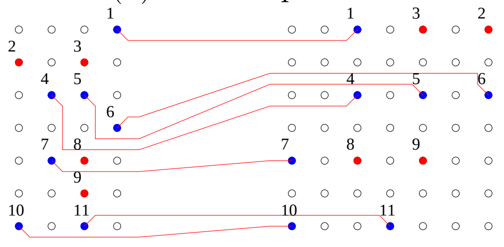

(a) LU Routing

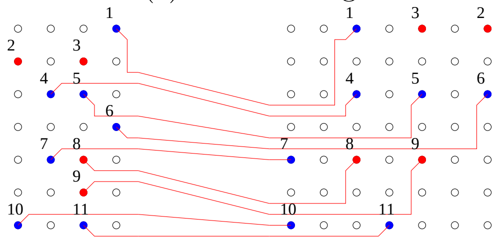

(b) LU Compression


(c) LD Routing


(d) LD Compression


(e) RU Routing


(f) RU Compression


(g) RD Routing

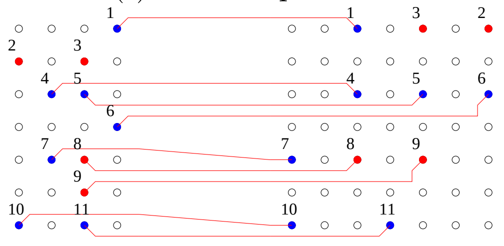

(h) RD Compression

Figure 6.13: The L-shape escape routing and compacted solutions for the case in Figure 6.9. (a) and (b) The LU routing and compaction solutions. (c) and (d), (e) and (f), and (g) and (h) The LD, RU and RD routing and compaction solutions, respectively. Red circles represent pins of a differential pair.

6.3 Applications in the PCB Routing System

All our above discussions are based on the assumption that the bus escape ordering is known before escape routing. In practice, this assumption is not always true. In high-end PCBs routing resources are very limited inside components; therefore the bus escape direction determination is quite critical to the whole PCB routing system. Appropriate bus escape directions can significantly diminish bus routing region overlaps inside components, which may lead to a successful whole board routing solution.

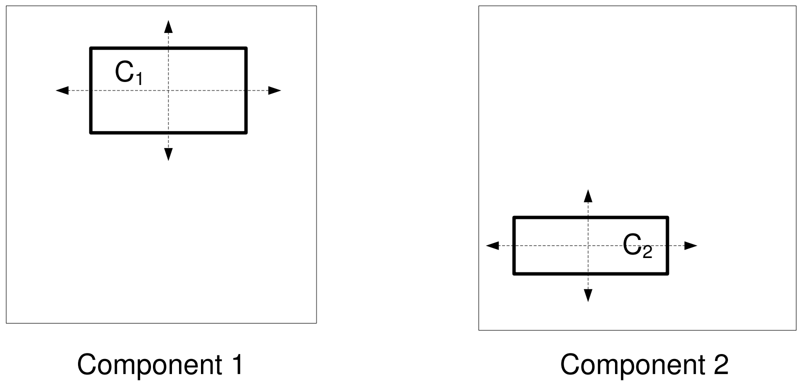


Figure 6.14: All possible escape directions for a bus.

As shown in Figure 6.14, each bus is between two components and in each component it has four optional escape directions: up, down, left and right. Therefore, each bus has 16 ways to escape both components simultaneously. A high-end board contains hundreds of buses and in order to utilize the routing resources within components effectively, we should take into account all buses globally rather than processing them individually. Therefore, for each bus we may need to find its escape routing solutions for all escape direction combinations. Our router is suitable to be applied for this purpose.

For each bus, we can perform the L-shape escape routing algorithm to find a routing solution for each of 16 possible escape direction combinations in Figure 6.14. For one escape direction combination, if the L-shape escape router cannot escape all nets in one layer, it will be performed iteratively to find a multi-layer escape solution. A two-layer escape routing solution is displayed in Figure 6.15.

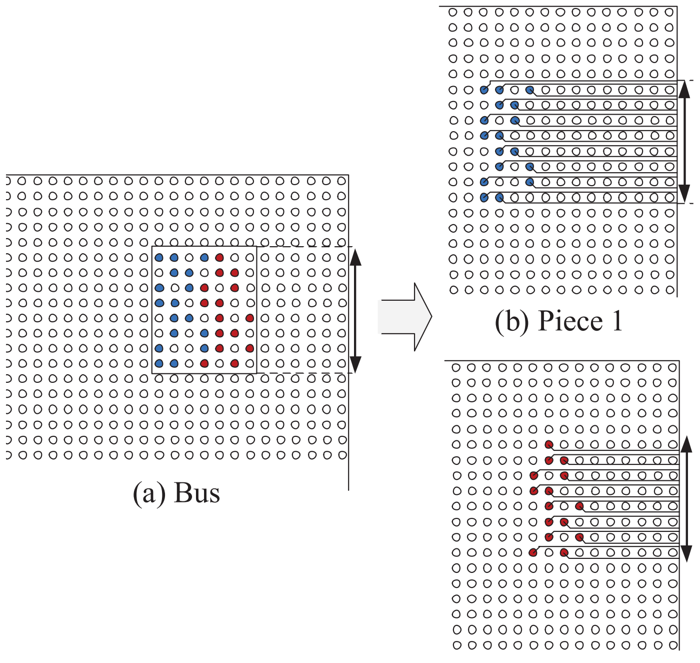


Figure 6.15: A two-layer escape routing solution of a bus in (a).

6.4 Experimental Results

We tested our bus L-shape escape algorithm on a state-of-the-art industrial board which contains 7000+ nets and 12 routing layers and has been manually routed. We performed our experiments on a workstation with two 3.0 GHz Intel Xeon CPUs and 4 GB memory.

We extracted bus definitions and bus escape directions from the manual solution, where all nets in a bus can be fully routed in a single layer along the escape direction. We obtained 146 buses from the manual routing solution and each bus contains at most 82 nets. We applied our L-shape escape routing algorithm to all these buses. Without allowing the net ordering adjustments, 95.9% nets are successfully routed in less than 1 minute CPU time. After applying the net ordering adjustments, totally 98.2% nets are routed in less than 3 minute CPU time. We demonstrate the detailed routing results on each routing layer in Table 6.1. In this table, the second column gives the number of buses we extracted from the manual solution on each layer. The third and fourth columns show the total number of nets and the total number of routed nets on each layer. The last column gives the percentages of routed nets.

Table 6.1: Escape routing statistics.

layerID	# of buses	# of nets	# of routed nets	% of routed nets
1	10	698	683	97.9%
2	14	873	860	98.5%
3	14	865	854	98.7%
4	10	454	450	99.1%
5	10	480	472	98.3%
6	9	388	382	98.4%
7	19	740	728	98.4%
8	18	562	552	98.2%
9	12	584	573	98.1%
10	10	325	320	98.5%
11	10	510	495	97.1%
12	10	504	488	96.8%
total	146	7033	6907	98.2%

6.5 Conclusions

In this chapter, we first proposed a wire packing based escape routing algorithm, which can find an L-shape escape routing solution in a very short time. This algorithm is much faster than all existing escape routing algorithms and is capable of providing comparable escape routing results. Then we introduced how to utilize this algorithm to explore the escape routing solutions in the real PCB routing process.

REFERENCES

[1] D. Wiens, “Printed circuit board routing at the threshold, ” white paper, Mentor Graphics, 2000.

[2] J. C. Whitaker, *The Electronics Handbook*, 2nd ed. Boca Raton, FL: CRC Press, 2005.

[3] M. M. Ozdal, “Routing algorithms for high-performance VLSI packaging,” Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2005.

[4] H. Harrer, D. M. Dreps, T.-M. Winkel, W. Scholz, B. G. Truong, A. Huber, T. Zhou, K. L. Christian, and G. F. Goth, “High-speed interconnect and packaging design of the ibm system z9 processor cage,” *IBM J. Res. Dev.*, vol. 51, no. 1/2, pp. 37–52, 2007.

[5] H. Harrer, H. Pross, T.-M. Winkel, W. D. Becker, H. Stoller, M. Yamamoto, S. Abe, B. J. Chamberlin, and G. A. Katopis., “First- and second-level packaging of the z990 processor cage,” *IBM J. Res. Dev.*, vol. 46, no. 4-5, pp. 397–420, 2002.

[6] T.-M. Winkel, W. D. Becker, H. Harrer, H. Pross, D. Kaller, B. Garben, B. J. Chamberlin, and S. A. Kuppinger, “First- and second-level packaging of the z990 processor cage,” *IBM J. Res. Dev.*, vol. 48, no. 3-4, pp. 379–394, 2004.

[7] C. S. Rim and K. Nakajima, “On rectangle intersection and overlap graphs,” *IEEE Transactions on Circuits and Systems-1: Fundamental Theory and Applications*, vol. 42, no. 9, pp. 549–553, 1995.

[8] M. M. Ozdal and M. D. F. Wong, “Simultaneous escape routing and layer assignment for dense PCBs,” in *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2004, pp. 822–829.

[9] M. M. Ozdal, M. D. F. Wong, and P. S. Honsinger, “An escape routing framework for dense boards with high-speed design constraints,” in *Proceedings of the 2005 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2005, pp. 759–766.

[10] X. Tang, R. Tian, and D. F. Wong, “Fast evaluation of sequence pair in block placement by longest common subsequence computation,” in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2000, pp. 106–111.

[11] X. Tang and D. F. Wong, “Fast-sp: A fast algorithm for block placement based on sequence pair,” in *Proceedings of the 2001 Conference on Asia South Pacific Design Automation (ASP-DAC)*, 2001, pp. 521–526.

[12] J. W. Hunt and T. G. Szymanski, “A fast algorithm for computing longest common subsequences,” *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[13] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* Boston, MA: MIT Press, 1992.

[14] N. L. Koren, “Pin assignment in automated printed circuit board design,” in *DAC ’72: Proceedings of the 9th Design Automation Workshop*, 1972, pp. 72–79.

[15] L. Mory-Rauch, “Pin assignment on a printed circuit board,” in *DAC ’78: Proceedings of the 15th Conference on Design Automation*, 1978, pp. 70–73.

[16] H. Brady, “An approach to topological pin assignment,” *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 3, no. 3, pp. 250–255, July 1984.

[17] T. D. Am, M. Tanaka, and Y. Nakagiri, “An approach to pin assignment in printed circuit board design,” *SIGDA Newsl.*, vol. 10, no. 2, pp. 21–33, 1980.

[18] T. Meister, J. Lienig, and G. Thomke, “Novel pin assignment algorithms for components with very high pin counts,” in *DATE ’08: Proceedings of the Conference on Design, Automation and Test in Europe*, 2008, pp. 837–842.

[19] T. Yan and M. D. F. Wong, “A correct network flow model for escape routing,” in *DAC ’09: Proceedings of the 46th Annual Design Automation Conference*, 2009, pp. 332–335.

[20] D. Staepelaere, J. Jue, T. Dayan, and W. Dai, “Surf: Rubber-band routing system for multichip modules,” in *IEEE Design and Test of Computers*, 1993, pp. 18–26.

[21] D. J. Staepelaere, “Geometric transformations for a rubber-band sketch,” M.S. thesis, University of California at Santa Cruz, Santa Cruz, CA, USA, September 1992.

[22] M. M. Ozdal and M. D. F. Wong, “Length-matching routing for high-speed printed circuit boards,” in *ICCAD’03: Proceedings of the 2003 IEEE/ACM International Conference on Computer-Aided Design*, 2003, p. 394.

[23] T. Yan and M. D. F. Wong, “BSG-route: A length-matching router for general topology,” in *ICCAD ’08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, 2008, pp. 499–505.

[24] H. Kong, T. Yan, M. D. F. Wong, and M. M. Ozdal, “Optimal bus sequencing for escape routing in dense PCBs,” in *ICCAD ’07: Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 390–395.

[25] “CS2:min-cost flow solver,” 2007, [Online]. Available: http:/www.igsystems.com/cs2/index.html.

[26] "Cadence OrCAD PCB Designer," 2007, [Online]. Available: http://www.cadence.com/products/pcb/pcbdesigner.

[27] "Mentor graphics topological router," 2007, [Online]. Available: http://www.mentor.com/products/pcb-system-design/layout-routing/topology-router.

[28] "Cadence Allegro PCB Design," 2007, [Online] Available: http://www.cadence.com/products/pcb/pcbdesign.

[29] M. Cho, K. Lu, K. Yuan, and D. Z. Pan, "BoxRouter 2.0: Architecture and implementation of a hybrid and robust global router," in *ICCAD '07: Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 503–508.

[30] M. Pan and C. Chu, "FastRoute 2.0: A high-quality and efficient global router," in *ASP-DAC '07: Proceedings of the 2007 Conference on Asia South Pacific Design Automation*, 2007, pp. 250–255.

[31] M. M. Ozdal and M. D. F. Wong, "Archer: A history-driven global routing algorithm," in *ICCAD '07: Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 488–495.

[32] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York, NY: W. H. Freeman & Co., 1990.

[33] D. S. Hochbaum, *Approximation Algorithms for NP-Hard Problems.* Florence, KY: Course Technology, 1996.

[34] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *FPGA '95: Proceedings of the Third International ACM Symposium on Field-Programmable Gate Arrays*, 1995, pp. 111–117.

[35] C. Ebeling, L. McMurchie, S. A. Hauck, and S. Burns, "Placement and routing tools for the Triptych FPGA," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 3, no. 4, pp. 473–482, 1995.

[36] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, 1997, pp. 213–222.

[37] D. F. Wong, H. W. Leong, and C. L. Liu, *Simulated Annealing for VLSI Design.* Norwell, MA: Kluwer Academic Publishers, 1988.

[38] J.-W. Fang, I.-J. Lin, Y.-W. Chang, and J.-H. Wang, "A network-flow-based RDL routing algorithm for flip-chip design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 8, pp. 1417–1429, Aug. 2007.

[39] R. Wang, R. Shi, and C.-K. Cheng, "Layer minimization of escape routing in area array packaging," in *ICCAD '06: Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*, 2006, pp. 815–819.

[40] W.-T. Chan, F. Y. L. Chin, and H.-F. Ting, “A faster algorithm for finding disjoint paths in grids,” in *ISAAC ’99: Proceedings of the 10th International Symposium on Algorithms and Computation*, 1999, pp. 393–402.

[41] M.-F. Yu and W. W.-M. Dai, “Single-layer fanout routing and routability analysis for ball grid arrays,” in *ICCAD ’95: Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*, 1995, pp. 581–586.

[42] L. Luo and M. D. F. Wong, “Ordered escape routing based on Boolean satisfiability,” in *ASP-DAC ’08: Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, 2008, pp. 244–249.

[43] T. Yan, H. Kong, and M. D. F. Wong, “Optimal layer assignment for escape routing of buses,” in *ICCAD ’09: Proceedings of the 2009 IEEE/ACM International Conference on Computer-Aided Design*, 2009, pp. 275–280.

[44] H. Kong, T. Yan, and M. D. F. Wong, “Automatic bus planner for dense PCBs,” in *DAC ’09: Proceedings of the 46th Annual Design Automation Conference*, 2009, pp. 326–331.