ALGORITHMIC STUDIES ON PCB ROUTING

BY

TAN YAN

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 and Director of Research
Assistant Professor Deming Chen
Associate Professor Steven S. Lumetta
Professor Robin A. Rutenbar

ABSTRACT

As IC technology advances, the package size keeps shrinking while the pin count of a package keeps increasing. A modern IC package can have a pin count of thousands. As a result, a complex printed circuit board (PCB) can host more than ten thousand signal nets. Such a huge pin count and net count make manual design of packages and PCBs an extremely time-consuming and error-prone task. On the other hand, increasing clock frequency imposes various physical constraints on PCB routing. These constraints make traditional IC and PCB routers not applicable to modern PCB routing. To the best of our knowledge, there is no mature commercial or academic automated router that handles these constraints well. Therefore, automated PCB routers that are tuned to handle such constraints become a necessity in modern design. In this dissertation, we propose novel algorithms for three major aspects of PCB routing: escape routing, area routing and layer assignment.

Escape routing for packages and PCBs has been studied extensively in the past. Network flow is pervasively used to model this problem. However, previous studies are incomplete in two senses. First, none of the previous works correctly model the diagonal capacity, which is essential for 45° routing in most packages and PCBs. As a result, existing algorithms may either produce routing solutions that violate the diagonal capacity or fail to output a legal routing even though one exists. Second, few works discuss the escape routing problem of differential pairs. In high-performance PCBs, many critical nets use differential pairs to transmit signals. How to escape differential pairs from a pin array is an important issue that has received too little attention in the literature.

In this dissertation, we propose a new network flow model that guarantees the correctness when diagonal capacity is taken into consideration. This model leads to the *first optimal algorithm* for escape routing. We also extend our model to handle missing pins. We then propose two algorithms for the

differential pair escape routing problem. The first one computes the *optimal routing* for a single differential pair while the second one is able to simultaneously route multiple differential pairs considering both routability and wire length. We then propose a two-stage routing scheme based on the two algorithms. In our routing scheme, the second algorithm is used to generate initial routing and the first algorithm is used to perform rip-up and reroute.

Length-constrained routing is another very important problem for PCB routing. Previous length-constrained routers all have assumptions on the routing topology. We propose a routing scheme that is free of any restriction on the routing topology. The novelty of our proposed routing scheme is that we view the length-constrained routing problem as an *area assignment problem* and use a *placement* structure to help transform the area assignment problem into a mathematical programming problem. Experimental results show that our routing scheme can handle practical designs that previous routers cannot handle. For designs that they could handle, our router runs much faster.

Length-constrained routing requires the escaped nets to have matching ordering along the boundaries of the pin arrays. However, in some practical designs, the net ordering might be mismatched. To address this issue, we propose a preprocessing step to untangle such twisted nets. We also introduce a practical routing style, which we call *single-detour routing*, to simplify the untangling problem. We discover a *necessary and sufficient condition* for the existence of single-detour routing solutions and present a dynamic programming based algorithm that *optimally* solves the problem. By integrating our algorithm into the bus router in a length-constrained router, we show that many routing problems that cannot be solved previously can now be solved with insignificant increase in runtime.

The nets on a PCB are usually grouped into buses. Because of the high pin density of the packages, the buses need to be assigned into multiple routing layers. We propose a layer assignment algorithm to assign a set of buses into multiple layers without causing any conflict. Our algorithm guarantees to produce a layer assignment with minimum number of layers. The key idea is to transform the layer assignment problem into a bipartite matching problem. This research result is an improvement over a previous work, which is optimal for only one layer.

To my parents, my wife and my son

ACKNOWLEDGMENTS

I owe my deepest gratitude to my adviser, Prof. Martin D. F. Wong. He has been constantly helping me in many aspects: from establishing research topics to presenting research results. This dissertation would not have been possible without his patient guidance, inspiring discussions and abundant encouragement.

Besides my adviser, I would like to thank the rest of my doctoral committee, Prof. Deming Chen, Prof. Steven S. Lumetta and Prof. Robin A. Rutenbar, for their insightful comments and constructive suggestions. Their invaluable opinions have significantly improved the quality of this dissertation

I also want to express my grateful thanks to Fujitsu Lab for funding and supporting my PCB research. In particular, I would like to thank Mr. Toshiyuki Shibuya, Mr. Takao Yamaguchi and Mr. Ikuo Ohtsuka for sharing their precious design experience and preparing design data for my experiments. I would also like to thank Mr. Philip Honsinger of IBM Corp. for the valuable discussions on PCB design.

The members of Prof. Wong's research team made my life at UIUC very enjoyable. I would like to thank Hui Kong for the heated discussions. It has been a great pleasure to work with him. I would also like to thank Lijuan Luo, Qiang Ma, Peggi Wu and Hongbo Zhang for all the stimulating discussions and the seamless collaborations we had. I would like to thank Leslie Hwang for making the office a fun place and for helping me with my ECE444 study. I would also like to thank Dr. Lei Cheng, Dr. Liang Deng and Dr. Yu Zhong for the tips and help in both study and life.

Last but not least, I would like to thank my family for their endless love and support. I want to thank my parents for raising me up and putting all their efforts in educating me. I cannot thank my wife, Ning Fu, enough for her love, support, and encouragement throughout my Ph.D. study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

BGA	Ball Grid Array
BSG	Bounded-Sliceline Grid
CPU	Central Processing Unit
DIP	Dual In-line Package
DPER	Differential Pair Escape Routing
LP	Linear Programming
NE	Northeast
NW	Northwest
OTT	Oct-Touched Tile
PCB	Printed Circuit Board
SDU	Single-Detour Untangling
SE	Southeast
SW	Southwest

CHAPTER 1

INTRODUCTION

As IC technology advances, the package size keeps shrinking while the pin count of a package keeps increasing. Nowadays, a dense package can have as many as 2000 pins [1]. Such a huge pin count makes manual design of packages and printed circuit boards (PCBs) an extremely time-consuming and error-prone task. On the other hand, increasing clock frequency imposes special physical constraints such as length-constrained routing, pairwise routing, planar routing, etc., on high performance printed circuit boards [2–5]. These constraints make traditional IC and PCB routers not applicable to modern PCB routing. To the best of our knowledge, there is no mature commercial or academic automated router that handles these constraints well. Therefore, automated PCB routers that are tuned to handle such constraints become a necessity in modern design.

1.1 Printed Circuit Board (PCB) Routing

A modern PCB usually hosts several chip packages whose footprints on board are arrays of pins (see Figure 1.1).[1] Such pins on the board are expected to be connected by non-crossing wires. Not all connections can be routed on one layer, so we may need multiple layers to accommodate all the wire connections. However, introducing vias at the middle of a route would introduce reflection and ringing effects which can cause serious signal integrity issues [4, 5]. Therefore, it is highly preferred that no vias are inserted in the middle of the routing (vias are allowed at the two ends to connect to the package pin/ball). This requires the routing of a net to be *planar*, without switching layers. This planar routing style distinguishes PCB routing

[1] Denser packages usually use a ball grid array (BGA), which is mounted on the surface of the board. The footprint of a BGA is usually an offset array of through vias, which looks like an array of pins

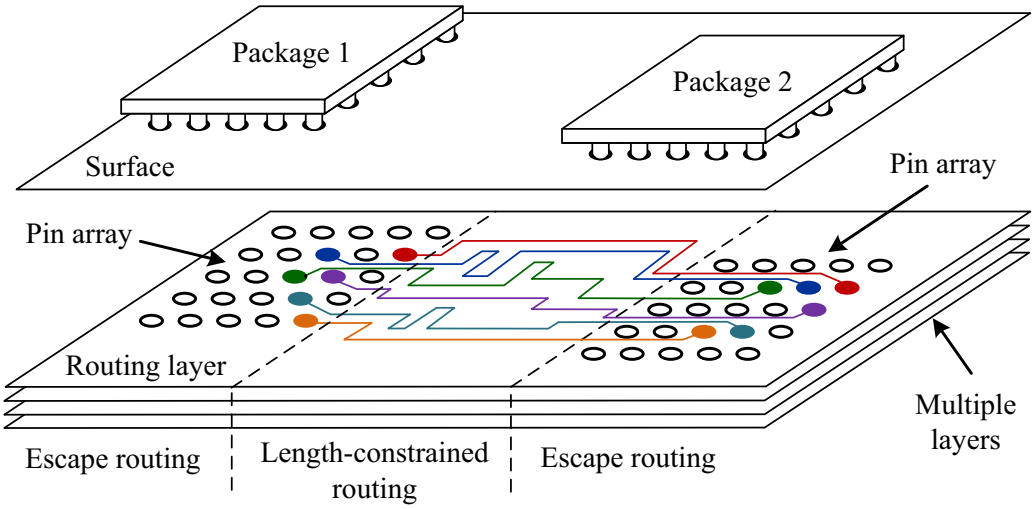


Figure 1.1: An illustration of PCB routing.

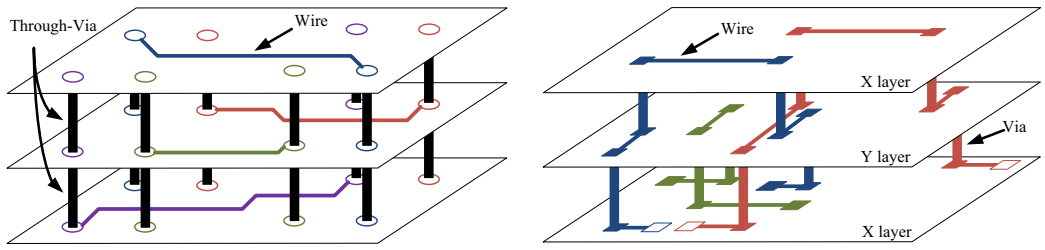


Figure 1.2: PCB routing vs. IC routing. (a) PCB routing: a net is preferred to be routed on one single layer; vias are added only at the two ends of the route. (b) IC routing: each routing layer has a preferred routing direction; the routing path needs to switch layers (thus needs vias) to change routing direction.

from the conventional XY routing of ICs, in which each routing layer has a preferred routing direction. In IC routing, if a routing path needs to change direction, vias are inserted into the path to make the switch to the layer with the desired direction. Figure 1.2 illustrates the differences between PCB and IC routing. For this reason, conventional IC routing algorithms cannot be applied to solve the PCB routing problem.

PCB routing is usually decomposed into two subproblems:

Escape routing : routing from pins inside the arrays to the boundary of the arrays (help the pins "escape" the array).

Length-constrained routing : routing between the pin arrays.

Escape routing and length-constrained routing have different tasks. Since escape routing usually dominates the total number of layers, the major task of escape routing is to escape as many pins from the array as possible or to escape a set of specified pins using as few layers as possible. Sometimes it may also need to provide matching net orderings along the boundaries of the two arrays in order to provide a planar topology for later length-constrained routing. The focus of length-constrained routing, on the other hand, is to carefully detour the wires to meet the length bounds while maintaining the planar topology inherited from escape routing.

Nets are usually grouped as buses on PCBs. Nets belonging to one bus usually have similar timing and other constraints and are thus expected to be routed close to each other on the same layer. Due to the huge pin count and high density of the pin array, it usually requires multiple layers to route the buses without any conflict. In fact, modern PCBs may contain as many as 20 layers [6]. Therefore, how to assign the routing of buses to different layers also becomes an important issue.

1.2 Overview of this Dissertation

In this dissertation, we present our research results on escape routing, length-constrained routing and layer assignment [7–13].

In Chapter 2, we discuss the escape routing problem. Escape routing for packages and PCBs has been studied extensively in the past. Network flow is pervasively used to model this problem. However, none of the previous works correctly models the *diagonal capacity*, which is essential for 45° routing in most packages and PCBs. As a result, existing algorithms may either produce routing solutions that violate the diagonal capacity or fail to output a legal routing even though one exists. In Chapter 2, we propose a new network flow model that guarantees the correctness when diagonal capacity is taken into consideration. This model leads to the *first optimal algorithm* for escape routing. We also extend our model to handle *missing pins*.

Although the escape routing problem is extensively studied, few works discuss the escape routing of differential pairs. In Chapter 3, we study the *differential pair escape routing problem* and propose two algorithms. The first one computes the *optimal routing* for a single differential pair while

the second one is able to simultaneously route multiple differential pairs considering both routability and wire length. We then propose a two-stage routing scheme based on the two algorithms. Experimental results show that our routing scheme is very effective in solving the differential pair escape routing problem.

Length-constrained routing is another very important issue for PCB routing. Previous length-constrained routers all have assumptions on the routing topology, whereas practical designs may be free of any topological constraint. In Chapter 4, we propose a routing scheme that deals with *general topology*. Unlike previous works, our approach does not impose any restriction on the routing topology. Moreover, our routing scheme is *gridless*. Its performance does not depend on the routing grid size of the input while previous length-constrained routers do. This is a big advantage because modern PCB routing configurations usually imply huge routing grids. The novelty of our approach is that we view the length-constrained routing problem as an *area assignment problem* and use a *placement* structure, bounded-sliceline grid (BSG) [14], to help transform the area assignment problem into a mathematical programming problem. We then use an iterative approach to solve the mathematical programming problem. Experimental results show that our routing scheme can handle practical designs that previous routers cannot handle. For designs that they could handle, our router runs much faster. For example, in one of our results, we obtain the solution in 88 s while a previous router [15] takes more than one day.

In order to perform the length-constrained routing in a planar fashion, the orderings of escaped nets along the boundaries of the pin arrays are required to be matching. However, in some practical designs, the net ordering might be mismatched and the nets become twisted. In Chapter 5, we propose a preprocessing step to untangle such twisted nets. We also introduce a practical routing style, which we call *single-detour routing*, to simplify the untangling problem. We then present a *necessary and sufficient condition* for the existence of single-detour routing solutions. Furthermore, we present a dynamic programming based algorithm to solve the single-detour untangling problem with consideration of wire capacity between adjacent pins. Our algorithm produces an optimal single-detour routing solution that rematches the net ordering. By integrating our algorithm into the bus router in a previous length-constrained router, we show that many routing problems

that could not be solved previously can now be solved with insignificant increase in runtime.

The nets on a PCB are usually grouped into buses. Because of the high pin density of the packages, it is impossible to escape all the buses in one layer. How to assign the buses into multiple layers without causing any conflict is an important issue in PCB design. In Chapter 6, we propose an optimal layer assignment algorithm to assign a set of buses into multiple layers. Our algorithm guarantees to produce a layer assignment with minimum number of layers. This is an improvement over a previous work [16], which is optimal for only one layer.

CHAPTER 2

ESCAPE ROUTING WITH DIAGONAL CAPACITY

2.1 Introduction

Escape routing is an important problem in package and PCB design. Its purpose is to route from specified pins in a pin array to the boundary of the array (see Figure 2.1). It can be further classified into three categories:

Unordered escape is to route from pins inside one pin array to the boundary of the array without considering the pin ordering along the boundary. Previous works on this topic include [17–26].

Ordered escape also considers only one pin array. However, it requires the escape routing to conform to specified ordering along the boundary of the pin array. Previous works on this topic include [27–31].

Simultaneous escape considers escape routing of two pin arrays. The orderings of the escaped nets along the boundaries of the two pin arrays are required to match each other in order to provide a planar topology for later length-constrained routing. Previous works on this topic include [32–35].

The three types of escape routing problems have different applications in package and PCB routing. In this dissertation, we focus on the unordered escape problem.

In the pin array, the design rules limit the number of wires between two orthogonally or diagonally adjacent pins. We call such constraints *orthogonal capacity* and *diagonal capacity* respectively, or *O-cap* and *D-cap* for short (see Figure 2.1). If we consider a *tile* of the pin array, which is the square formed by four adjacent pins, we can see that *O-cap* limits the number of wires that go through its four sides while *D-cap* limits the number of wires that go through its two diagonals.

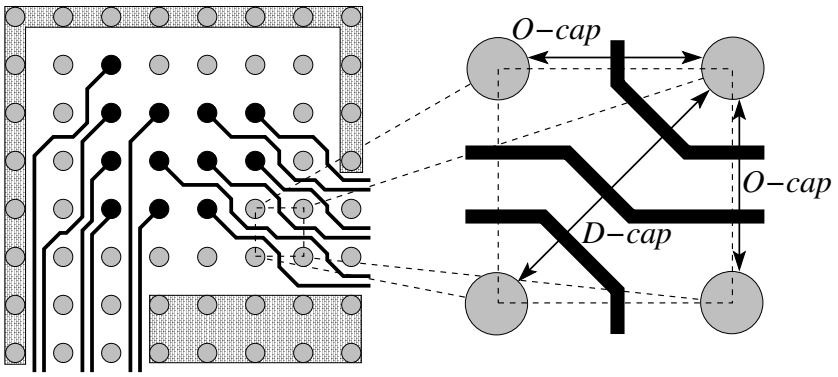


Figure 2.1: An example of the escape routing problem (left) and the enlarged view of a tile (right). Shaded areas denote blockages. Specified pins (black) are escaped to the boundary of the pin grid. In this example, *O-cap* = 2 and *D-cap* = 3.

Network flow is pervasively used to model this problem [17, 18, 21, 23–25]. The idea is to view each routing path as a unit flow from the pin to the boundary. Since no ordering is specified, a flow solution always corresponds to some non-crossing routing. However, none of the previous network flow models correctly capture the diagonal capacity, which is essential for 45° routing in most packages and PCBs. The models in [23,25] simply ignore the diagonal capacity and may therefore result in design rule violations. On the other hand, the models [17,18] set a limit on the number of wires inside a tile. Since tile capacity does not correctly reflect diagonal capacity, their models may miss the optimal solution. Detailed discussion about these models can be found in the next section. The other two models are not correct either. [24] uses triangulation, which captures *only one* of the two diagonals in a tile. Therefore, its solution may still contain capacity violations on the other diagonal. [21]'s network only gives an upper bound estimation on the number of routable nets. As we can see from the above discussion, none of the previous network flow models is able to correctly model the diagonal capacity.

Non-flow solutions were also proposed in some early works [22, 26]. Those works assume that all the pins in the array must be escaped on a single layer and the routing is symmetrical. This symmetry assumption is the foundation of their algorithms. However, due to the high pin density in modern packages and PCB designs, not all pins in the array can be escaped on one layer and the escape routing is very likely to be asymmetrical. Therefore, their algorithms are not applicable to the more general escape routing problem we discuss

here (we do not require all the pins to be escaped on one layer or the routing to be symmetrical).

In this chapter, we propose a network flow model that correctly models the diagonal capacity. Our model guarantees to give a legal routing if one exists. We then build an algorithm based on this model. As far as we know, this is the *first algorithm that guarantees optimality*. We also extend our model to handle missing pins, which are unused pins removed to increase the routing resource. Experimental results show that our algorithm has very short runtime.

The rest of this chapter is organized as follows: Section 2.2 introduces some background information. Section 2.3 presents our network flow model and escape routing algorithm, and Section 2.4 extends our model to deal with missing pins. Finally, experimental results are given in Section 2.5 and concluding remarks are given in Section 2.6.

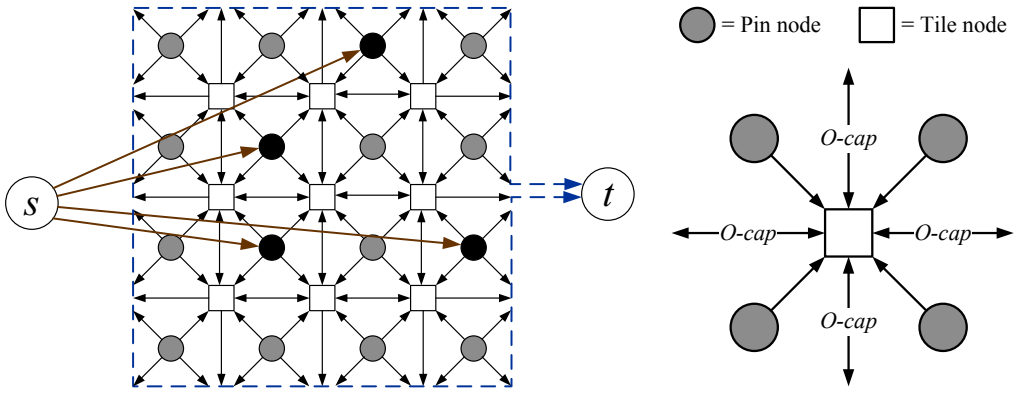
2.2 Background

In this section, we will first formulate the escape routing problem and then introduce the traditional network flow model used by some previous works.

2.2.1 Problem Formulation

The input to the escape routing problem is an $m \times n$ pin array with p pins specified as to-be-escaped pins. Certain areas of the pin array are marked as blockages, which do not allow any routing inside. Such blockages are used to model pre-routed nets or to guide the routing to escape through certain preferred boundaries. *O-cap* and *D-cap* are also given to specify the orthogonal capacity and diagonal capacity in the tile. We can safely assume that *O-cap* $\leq$ *D-cap* $\leq 2 \cdot$ *O-cap* for all our inputs due to the following two facts:

- The diagonal of a square tile is longer than the side of the square. Therefore, *D-cap* $\geq$ *O-cap*.
- The *O-cap* constraint already implies that at most $2 \cdot$ *O-cap* wires can pass the diagonal. So setting *D-cap* to be larger than $2 \cdot$ *O-cap* is



(a) Flow model for a pin grid (b) Inside a tile

Figure 2.2: The traditional network flow model used in previous works. Black pins denote the to-be-escaped pins and there are unit capacity edges from the super source s to them. Edges extending outside the boundary of the pin array are all incident to the super sink t.

meaningless.

The expected output of the problem is an octilinear planar routing from the to-be-escaped pins to the boundary of the pin array satisfying the capacity constraints and avoiding the blockages. We would also like the total length of the routing to be minimized.

2.2.2 Previous Works

In the traditional network flow model used by [17, 18, 23, 25],[1] each pin is represented by a pin node and each tile is represented by a tile node. Edges are added between horizontally and vertically adjacent tile nodes and from pin nodes to their adjacent tile nodes (see Figure 2.2). Edges extending out of the pin grid boundary are connected to a super sink. There are also edges from a super source to the pin nodes that are expected to be escaped. This model works fine if we consider only the orthogonal capacity *O-cap* because we can add capacity constraints on the orthogonal edges in the network to realize *O-cap*. However, diagonal capacity *D-cap* is not captured by this model. Consider the case where *O-cap* = 2 and *D-cap* = 3, which is very

[1]Some of these works assume monotonic and/or symmetric routing. Therefore, some orthogonal edges are unidirectional in their models. However, the network structure is the same as what we show here.

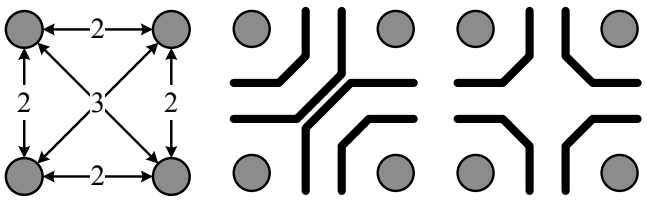


Figure 2.3: Previous network flow models cannot handle *D-cap* correctly. For the case *O-cap* = 2, *D-cap* = 3 in (a), models in [23, 25] may produce illegal routing in (b) and models in [17, 18] cannot capture the valid routing in (c).

common for PCB routing. Since the model has no control over the number of wires passing through the diagonals of the tile, it may produce routing like Figure 2.3 (b) which violates *D-cap*. In [17, 18], the number of wires inside a tile is also limited by adding node capacities to tile nodes. However, such node capacity does not correctly reflect the diagonal capacity. No matter how we set the tile node capacity, there are always counter-examples:

- Tile node capacity ≥ 4: This is the same as having no tile capacity because *O-cap* $= 2$ already implies at most 4 wires in a cell. The network may produce the routing in Figure 2.3 (b), which violates *D-cap*.
- Tile node capacity ≤ 3: The network cannot model the routing in Figure 2.3 (c) because there are 4 wires inside the tile. However, the routing itself is legal because only 2 wires pass each diagonal. As a result, the network model may fail to capture a legal routing solution even when one exists.

From the discussion above, we can see that such a simple network cannot model diagonal capacity correctly. A more sophisticated network model is needed to capture the diagonal capacity.

2.3 Our Network Flow Model

In our proposed network flow model, each tile contains five nodes, namely N-node on the north, E-node on the east, S-node on the south, W-node on

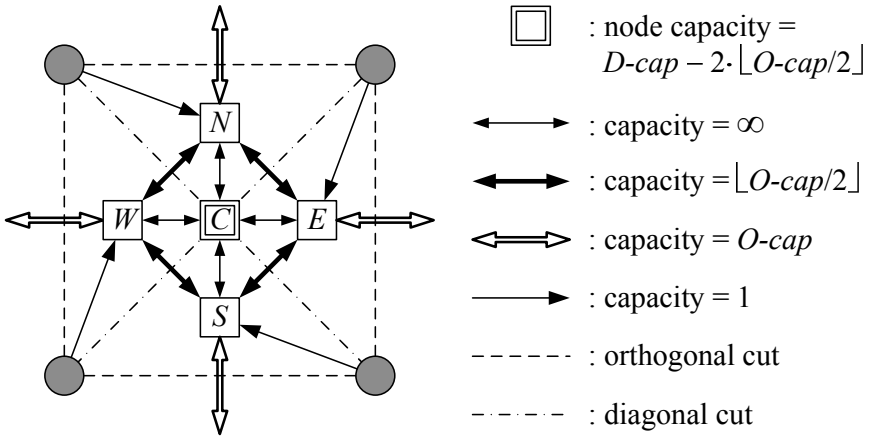


Figure 2.4: Our network flow model inside a tile.

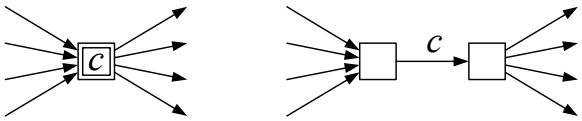


Figure 2.5: Turning a node with capacity c (left) into an edge with the same capacity (right).

the west and C-node in the center (see Figure 2.4). The first four nodes are called *peripheral nodes* and the last node is called the *center node*. We give the center node a capacity of $D\text{-}cap - 2 \cdot \lfloor O\text{-}cap/2 \rfloor$. Node capacity can be realized by splitting the node into two and adding an edge with corresponding capacity between them (see Figure 2.5).

We create bidirectional edges (which are realized by two directed edges: a forward edge and a backward edge) between every peripheral node and the center node and give these edges infinite capacity. We also introduce bidirectional edges between peripheral node pairs (N, E), (E, S), (S, W) and (W, N). We call such edges *peripheral edges* and give each of them capacity $\lfloor O\text{-}cap/2 \rfloor$. The five nodes and the edges between them compose an *intra-tile network*. Connections between tiles are also necessary. We use bidirectional edges to connect the N-node of a tile to the S-node of the tile above it as well as the E-node of a tile to the W-node of the tile to the right of it. Such edges are called *inter-tile edges* and have capacity $O\text{-}cap$. In order to escape the pins, we also create a pin node for each pin and create unidirectional edges from the four pins at the corners of a tile to the four peripheral nodes in the tile. The edges are from the pin at the NW corner to N-node, from

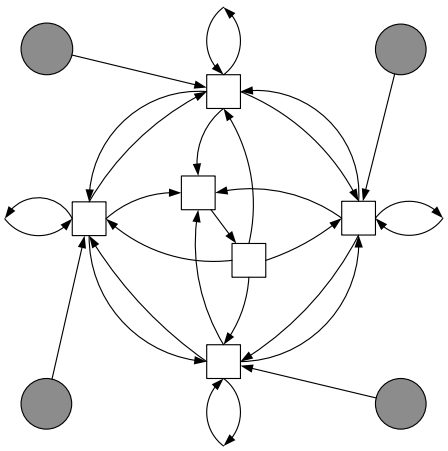


Figure 2.6: The actual implementation of our model.

the pin at the NE corner to E-node, from the pin at the SE corner to S-node and from the pin at the SW corner to W-node. We call these edges *pin edges*. All pin edges have capacity 1. Of course, if any nodes or edges lie in the blockage, we will not create them. In our implementation, we realize the bidirectional edges and node capacities through directed edges. Hence, the actual network looks like Figure 2.6.

Similar to the traditional model, we also introduce a super source s and a super sink t. All edges from the boundary tiles to the outside of the pin array are connected to t. Finally, we add edges with capacity 1 from s to the pin nodes of all the to-be-escaped pins.

The intuition behind this network model is that we need shortcut edges (the edges between peripheral nodes) to model the routing in Figure 2.3 (d). Such shortcut edges have capacity $\lfloor O\text{-}cap/2 \rfloor$. Then, in order to ensure the diagonal capacity constraint, we give C-node a capacity of $D\text{-}cap - 2 \cdot \lfloor O\text{-}cap/2 \rfloor$ so that the diagonal cuts have capacity of exactly $D\text{-}cap$.

2.3.1 Correctness of the Model

We call a flow of the network *legal* if the flow through every edge is an integer that does not exceed the edge capacity. The total flow from source s to sink t is called the *value* of the flow. We call a routing *legal* if it satisfies $O\text{-}cap$ and $D\text{-}cap$ constraints in all tiles.

Any legal flow of the network can be decomposed into a collection of unit

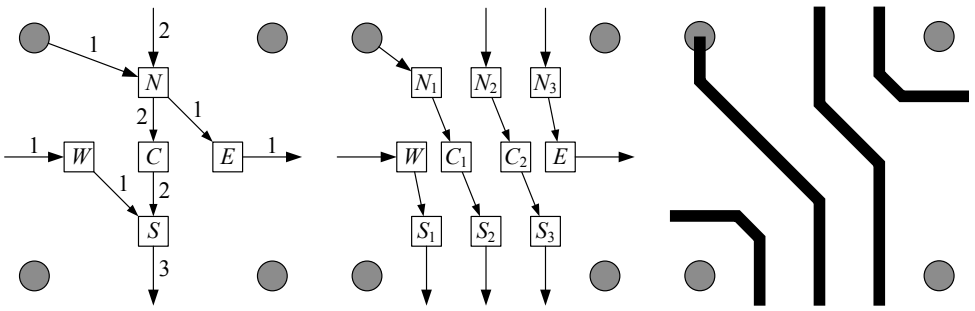


Figure 2.7: Split the nodes and edges of a flow (left) to obtain a planar topological routing (right) and then apply geometric transformation to obtain detail routing (c).

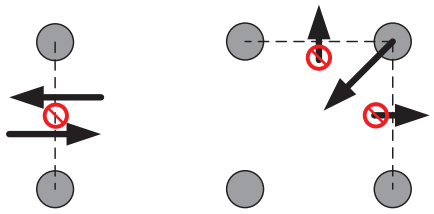
flow which corresponds to routing paths. Therefore, there is a correspondence between a flow solution and a routing solution:

Lemma 1. *Any legal flow of value k corresponds to a legal escape routing of k pins.*

Proof. We can transform the flow into a routing topology by node and edge splitting (see Figure 2.7): for a node v, if the flow through it $flow(v) > 1$, we split the node into $flow(v)$ copies. Similarly, we also split each edge with flow larger than 1. The split nodes and edges are connected in a planar fashion.

After node and edge splitting, each node and edge has flow at most 1. This flow gives us a planar topology of the escape routing. Notice that the value of the flow is k, so the escape routing contains k wires, each from a unique pin because the edge from s to each to-be-escaped pin has capacity 1. So the escape routing is from k pins.

It can be seen in Figure 2.4 that any orthogonal cut cuts only one inter-tile edge with capacity exactly *O-cap*. Any diagonal cut cuts two peripheral edges plus the center node. The total capacity of this cut is $2 \cdot \lfloor \textit{O-cap}/2 \rfloor + \textit{D-cap} - 2 \cdot \lfloor \textit{O-cap}/2 \rfloor = \textit{D-cap}$. Therefore, if the flow is legal, then the routing topology we have also satisfies all the *O-cap* and *D-cap* constraints. Past researches show that if a topology satisfies all the capacity constraints, then there always exists a corresponding legal detail routing [36, 37]. Algorithms are also proposed in [38,39] to transform the topological routes in to octilinear detail routes. □



Property 1 Property 2

Figure 2.8: The two properties of a directed routing R with minimum number of crossings with all orthogonal cuts.

On the other hand, if there is a legal escape routing, then there exists a legal flow:

Lemma 2. *If there exists a legal escape routing of k pins, then our model has a legal flow of value k.*

Proof. The proof is by construction. We show that at least one legal routing of k pins can be converted into a legal flow of value k.

Among all legal routing of k pins, we pick the one with minimum number of crossings with all the orthogonal cuts (orthogonal cut is the cut between orthogonally adjacent pins, see the dashed line segments in Figure 2.8). We assign a direction to each wire, which is from the pin to the array boundary. Such a directed routing R has the following properties (see also Figure 2.8):

1. No two wires of opposite directions can pass the same orthogonal cut.

2. If a wire is routed from a pin into one tile, then no wire can exit the tile from the two orthogonal cuts incident to that pin.

Let us show why these two properties hold. If we have two wires of opposite directions passing one orthogonal cut, we can reconnect the two wires as shown in Figure 2.9 and reduce the number of crossings with the orthogonal cut. Notice that the reconnection is local. Therefore, the routing after such reconnection is still legal. This contradicts our assumption that R already has minimum number of crossings with all the orthogonal cuts. Therefore, Property 1 is true.

For Property 2, let us assume that there is one wire exiting the tile from an orthogonal cut incident to the pin. There are two situations: (1) the wire is from that pin; (2) the wire is from some other pin. For the first situation,

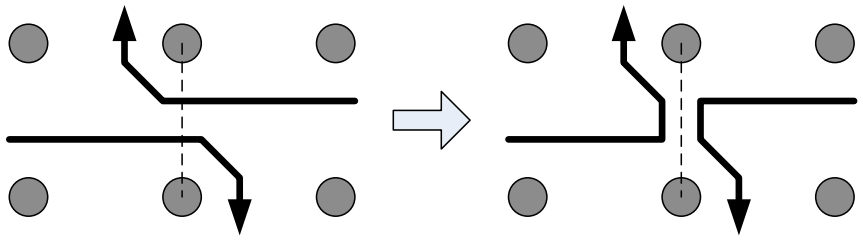


Figure 2.9: By reconnecting the wires, we can reduce the number of crossings between the wires and the orthogonal cuts without affecting the legality of the routing.

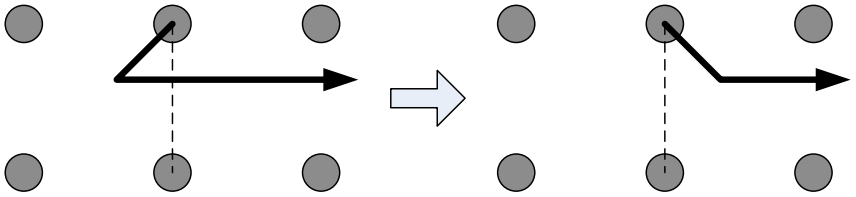


Figure 2.10: By shifting the wire from the pin to the neighboring tile, we can reduce the number of crossings between the wires and the orthogonal cuts without affecting the legality of the routing.

we can simply shift the wire to the neighboring tile to reduce the number of crossings between the wires and the orthogonal cuts (see Figure 2.10). Note that the change is local and the resultant routing is still legal. This contradicts our assumption that R already has the minimum number of crossings with all the orthogonal cuts. For the second situation, we reconnect the wires as shown in Figure 2.11 and reduce it to the first situation. Therefore, Property 2 is true.

Now we construct a legal flow solution from the legal routing R in two steps:

Step 1 : We construct the flow on inter-tile edges and pin edges.

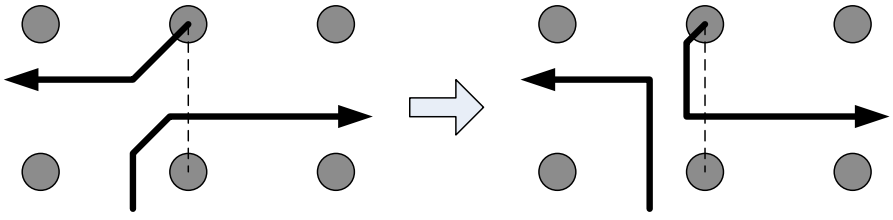


Figure 2.11: By reconnecting the wires, we can reduce the second situation to the first situation.

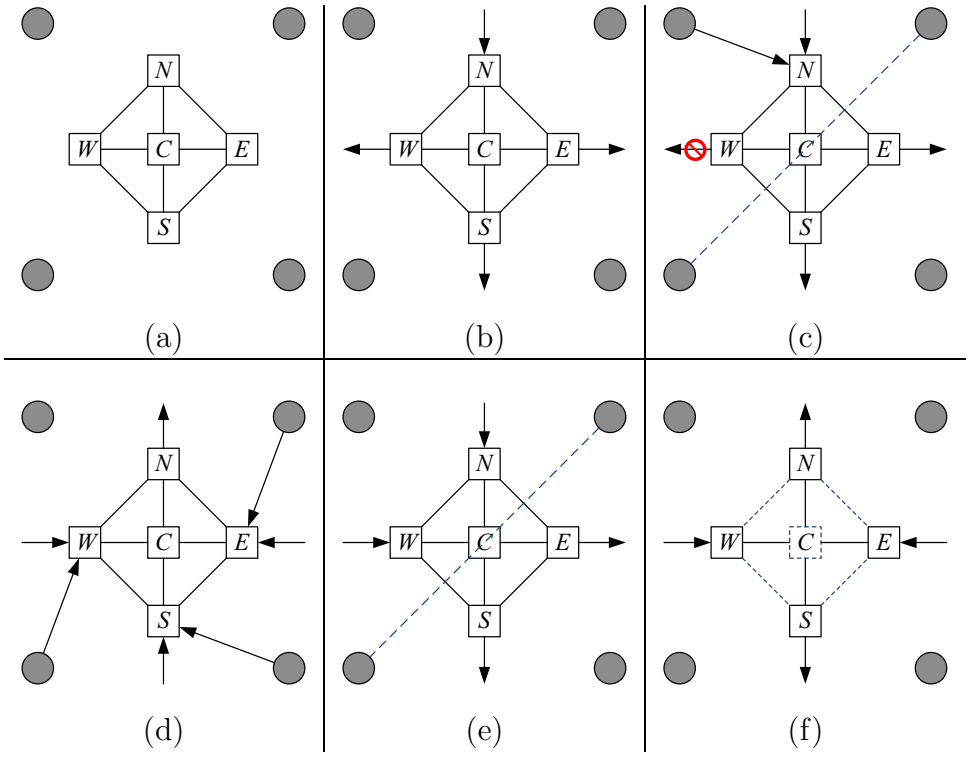


Figure 2.12: Constructing a flow solution of the intra-tile network. (a) The intra-tile network. (b)–(f) Analysis of the possible flow configurations of the intra-tile network.

Step 2 : We construct the flow in the intra-tile network (recall that an intra-tile network consists of the five tile nodes and the edges between them, see also Figure 2.12 (a)).

If we view each directed wire in R as a unit flow from a pin to the array boundary, we can obtain the flow crossing each orthogonal cut. We then assign the same flow to the inter-tile edges corresponding to the orthogonal cuts. If there is a wire from a pin to a tile, we assign a unit flow to the corresponding pin edge. Notice that such a flow assignment would not violate the capacity constraint because in a legal routing, there can be at most *O-cap* wires crossing an orthogonal cut and at most one wire from each pin.

Now with all the flow on inter-tile edges and pin edges determined, we know the flow going in and coming out of any intra-tile network. We also know that *total incoming flow* = *total outgoing flow* for an intra-tile network because of the continuity of the routing. Therefore, we can apply the flow algorithm to obtain a flow solution for the intra-tile network. Here, we show that for any incoming flow and outgoing flow configurations obtained from

Step 1, we can always obtain a legal flow solution for the intra-tile network.

We can classify the possible configurations of the incoming/outgoing flow of the intra-tile network into three categories:

1. The flow comes into the intra-tile network at only one peripheral node and out at one or more peripheral nodes. Without loss of generality, we assume that the flow comes in at N-node. There are two cases: either the incoming flow contains only flow from the inter-tile edge (Figure 2.12 (b)) or the incoming flow also includes a flow from the pin (Figure 2.12 (c)). In the first case, the total incoming flow is bounded by *O-cap* because at most *O-cap* wires can pass an orthogonal cut in a legal routing. In the second case, we know that no flow can come out of the tile from W-node due to Property 2. Correspondingly, in the routing R, the wires come into the tile from the top boundary of the tile and from the pin. These wires must exit the tile at the bottom and/or left boundary. As a result, all wires must cross the diagonal cut shown in Figure 2.12 (c). Therefore, the total number of wires in the tile cannot exceed *D-cap*, meaning that the total incoming flow cannot exceed *D-cap*.

2. The flow comes into the intra-tile network at one or more peripheral nodes but exits the network at only one peripheral node (Figure 2.12 (d)). In this case, the total outgoing flow is bounded by *O-cap* because at most *O-cap* wires can pass an orthogonal cut in a legal routing.

3. The flow comes into the intra-tile network at exactly two peripheral nodes and exits the network at exactly two peripheral nodes. Note that there cannot be any flow from a pin to the peripheral nodes due to Property 2. There are two cases: either the two incoming nodes are adjacent to each other (Figure 2.12 (e)) or the two incoming nodes are not adjacent (Figure 2.12 (f)). For the first case, we assume that the flow comes in at N, W nodes without loss of generality. Then all the wires in the tile must cross the diagonal cut shown in (Figure 2.12 (e)). Therefore, the total number of wires in the tile cannot exceed *D-cap*, meaning that the incoming flow is bounded by *D-cap*. We will discuss the second case (Figure 2.12 (f)) later.

From the above discussion, we can see that *total incoming flow* $\leq$ *D-cap* for all the cases except the one in Figure 2.12 (f). On the other hand, we can verify by enumeration that any cut of the intra-tile network (a cut of a network is a set of edges that separate the network into two disconnected components) has a capacity of at least *D-cap*. According to the max-flow min-cut theorem [40], we know that there must exist a legal flow solution of the intra-tile network.

Now let us discuss the situation in Figure 2.12 (f), in which the flow comes in at two non-adjacent peripheral nodes and exits at the two other peripheral nodes. Without loss of generality, we assume the flow comes in at W and E. The total incoming flow is bounded by $2 \cdot$ *O-cap* because each orthogonal cut allows at most *O-cap* wires. The min-cut between the incoming nodes W, E and the outgoing nodes N, S includes the four peripheral edges and the center node C. (The dashed edges and node in Figure 2.12 (f) illustrate the cut.) Therefore, the capacity of the min-cut is

$$4 \cdot \lfloor \textit{O-cap}/2 \rfloor + \textit{D-cap} - 2 \cdot \lfloor \textit{O-cap}/2 \rfloor = 2 \cdot \lfloor \textit{O-cap}/2 \rfloor + \textit{D-cap}$$

If *O-cap* is even, then $2 \cdot \lfloor \textit{O-cap}/2 \rfloor = \textit{O-cap}$. Therefore,

$$\text{min-cut} = 2 \cdot \lfloor \textit{O-cap}/2 \rfloor + \textit{D-cap} = \textit{O-cap} + \textit{D-cap} \geq 2 \cdot \textit{O-cap}$$

Since the total incoming flow is bounded by $2 \cdot$ *O-cap*, we know that there must exist a legal flow through the max-flow min-cut theorem.

If *O-cap* is odd, then $2 \cdot \lfloor \textit{O-cap}/2 \rfloor = \textit{O-cap} - 1$. Therefore,

$$\text{min-cut} = 2 \cdot \lfloor \textit{O-cap}/2 \rfloor + \textit{D-cap} = \textit{O-cap} + \textit{D-cap} - 1 \geq 2 \cdot \textit{O-cap} - 1$$

So if the total incoming flow does not exceed $2 \cdot$ *O-cap* $- 1$, there must exist a legal flow solution of the intra-tile network. If the total incoming flow is exactly $2 \cdot$ *O-cap*, then in the routing R, there are exactly *O-cap* wires coming in from the left side of the tile and exactly *O-cap* wires coming in from the right side of the tile (see Figure 2.13). Suppose x wires go from left to top and y wires go from left to bottom ($x + y =$ *O-cap*). In order to satisfy the orthogonal capacity, y wires from the right must go to the top side and x wires from the right must go to the bottom. As a result, we have $2x$ wires crossing one diagonal cut and $2y$ wires crossing another diagonal cut. Notice

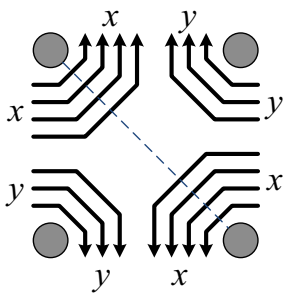


Figure 2.13: If *O-cap* is odd and there are $2 \cdot$ *O-cap* wires passing the tile, then *D-cap* $\geq 2x \geq$ *O-cap* $+ 1$.

that *O-cap* is odd, $x \neq y$. Without loss of generality, let us assume $x > y$. Then $2x > x + y =$ *O-cap*. Since x is an integer, we know $2x \geq$ *O-cap* $+ 1$. Because R is a legal routing, we have *D-cap* $\geq 2x \geq$ *O-cap* $+ 1$. Therefore,

$$\text{min-cut} = \textit{O-cap} + \textit{D-cap} - 1 \geq 2 \cdot \textit{O-cap}$$

So min-cut $\geq 2 \cdot$ *O-cap* but total incoming flow is bounded by $2 \cdot$ *O-cap*. By the max-flow min-cut theorem, we know that there must exist a legal flow solution of the intra-tile network. □

Lemma 1 shows that the flow solution of our model can always be turned into a legal detail routing solution while Lemma 2 shows that if there exists a legal detail routing solution, then our model will always capture one. As a result, we have the following theorem:

Theorem 1. *A given escape routing problem with k to-be-escaped pins has a legal routing solution* **iff** *our network model has a legal flow of value k. Furthermore, the legal flow of value k can be transformed into a legal routing solution.*

In order to minimize the wire length, we can assign cost 1 to the inter-tile edges (the hollow edges in Figure 2.4) and zero cost to all other edges. If we compute the min-cost max-flow of the network, we can minimize the number of tiles each wire traverses and thus the total wire length can be minimized. The min-cost max-flow solution can then be converted into detail routing through the transformation in the proof of Lemma 1.

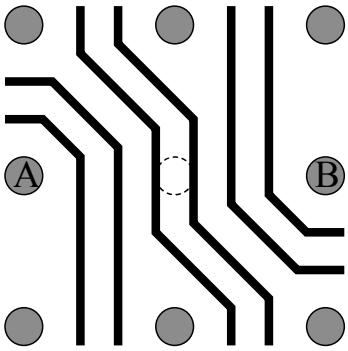


Figure 2.14: Missing pins increase the routing resource.

2.4 Modeling the Missing Pins

In practical PCB designs, the designer may remove some unused pins in the array to increase the routing resource. Figure 2.14 gives an example in which *O-cap* = 2 and *D-cap* = 4. It can be seen that by removing the pin at the center, the maximum number of wires allowed between A and B increases from 4 to 6. The difference, 2, is called the *extra horizontal capacity* of the missing pin. Similarly, we can define *extra vertical capacity* and *extra diagonal capacity*. Since usually the pin is round, the three types of extra capacities are the same. So we do not distinguish them but call them *extra capacity* and denote it as Δ. This extra capacity depends on the design rules and is usually given as input.

To model this extra capacity, we use a resource node to replace the pin node (see Figure 2.15). The resource node has node capacity the same as Δ. The unidirectional edges from the pin node to the peripheral nodes are now replaced with bidirectional edges between the resource node and the peripheral nodes. We give such edges infinite capacity.

The resource node essentially increases the horizontal and vertical capacity of the network by Δ. So it is able to capture the extra capacity introduced by the missing pin.

2.5 Experimental Results

We implement our network flow-based escape routing algorithm in C++ and test it on several industrial data sets. We use the min-cost flow solver CS2 [41]

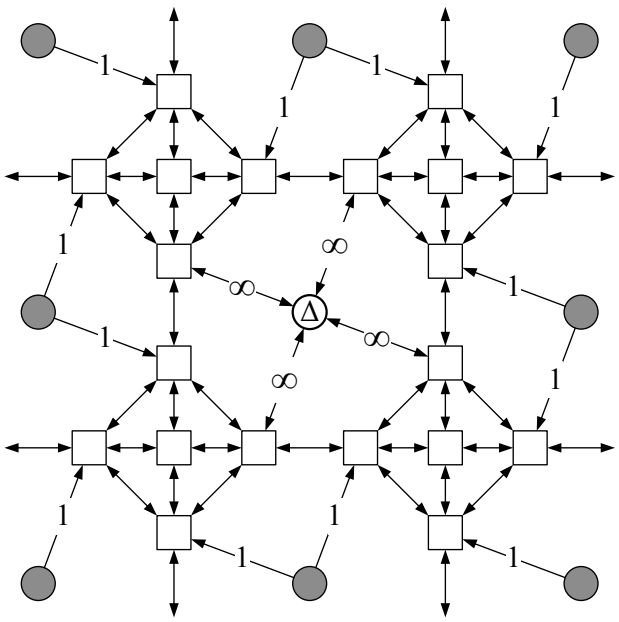


Figure 2.15: Modeling missing pins in our network.

to obtain the min-cost flow solution of our network model. All experiments are performed on a workstation with two 3.0 GHz Intel Xeon processors and 4 GB memory. The operating system is RedHat Linux 2.6.9.

We test our router on eight data sets and the result is reported in Table 2.1. Among the eight data sets, *industrial_1* to *industrial_7* are actual industrial data and *modified_8* is derived from industrial data with some modification. The left five columns of the table give the information on the data including the name, the pin array size, the number of to-be-escaped pins, the number of missing pins, and the capacity rules (*O-cap*, *D-cap* and extra capacity Δ). The next two columns show the number of *D-cap* constraint violations in our result as well as the runtime of our router. The last two columns show the number of *D-cap* violations and the runtime of the traditional model used in [23, 25].

It can be seen that our model gives *zero D-cap* violations while the traditional model leads to as many as 53 *D-cap* violations because it ignores the diagonal capacity. Since the total runtime is only one second or less, the runtime difference of the two methods is insignificant.

Figure 2.16 shows our routing solution of *modified_8*. We can see that there are several missing pins on the north, west and east side of the array (highlighted by the shaded zones), and their spaces are fully utilized in our

Table 2.1: Experimental results of our proposed network flow model.

	array	escape	missing	capacities			our model		traditional model	
	$w \times h$	pin no.	pin no.	O	D	Δ	*D-cap* vio.	runtime	*D-cap* vio.	runtime
industrial_1	14×16	78	13	2	3	3	0	0.16 s	0	0.14 s
industrial_2	29×11	66	20	2	3	3	0	0.22 s	10	0.17 s
industrial_3	33×14	120	46	2	3	3	0	0.33 s	6	0.28 s
industrial_4	35×17	160	30	2	3	3	0	0.61 s	9	0.28 s
industrial_5	35×35	108	105	2	3	3	0	0.86 s	1	0.68 s
industrial_6	35×17	143	38	2	3	3	0	0.39 s	0	0.30 s
industrial_7	35×35	220	106	2	3	3	0	1.01 s	53	0.79 s
modified_8	35×35	225	33	2	3	3	0	0.99 s	53	0.79 s

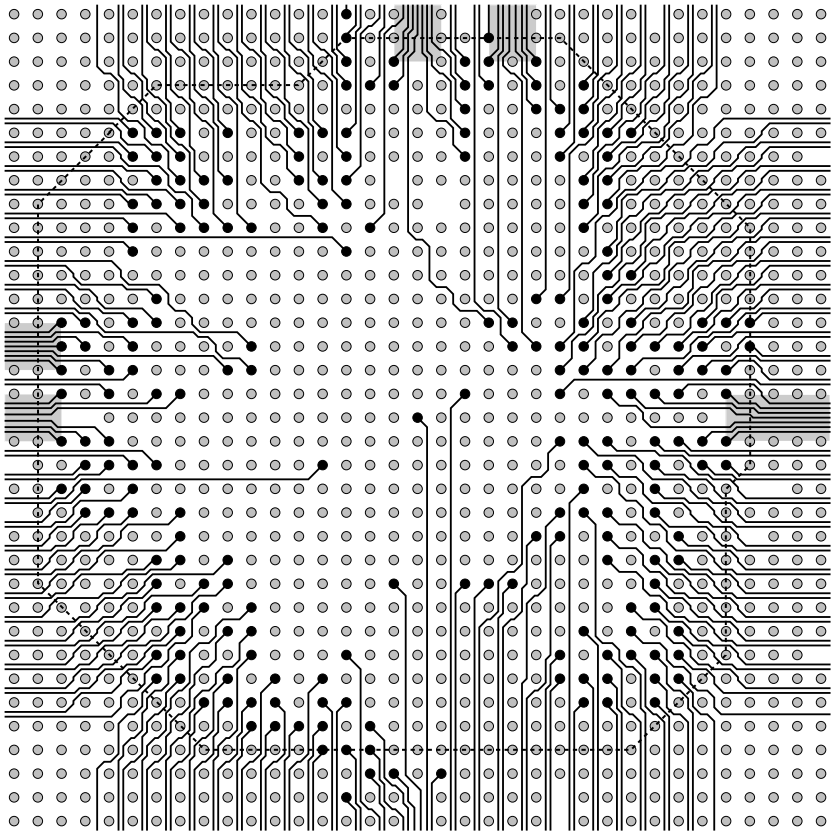


Figure 2.16: Routing solution of *modified_8*. The shaded zones highlight the spaces of the missing pins that are fully utilized by our router. The dashed polygon is drawn on top of the result to show that the routing uses up almost all routing resources.

result. To show that our router can handle dense designs, we draw a dashed polygon on the routing result. It can be seen that almost all the *O-cap* and *D-cap* along the polygon are used up by our routing, which indicates that the routing is very dense.

2.6 Conclusion

In this chapter, we presented a novel network-flow model that correctly models the diagonal capacity which is essential for 45° routing in packages and PCBs. We proved the correctness of our model. We also showed how to extend our model to handle missing pins, which appear frequently in practical designs.

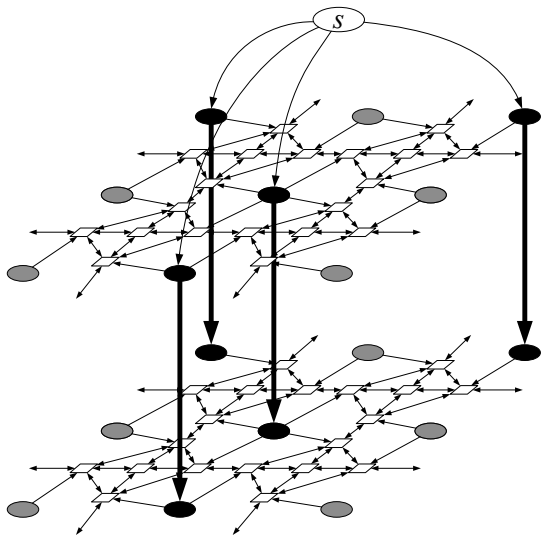


Figure 2.17: Multi-layer network. Black pins are to be escaped and thick edges are inter-layer edges.

Notice that if the max-flow solution of our network has a value less than the number of to-be-escaped pins, then not all these pins can be escaped on a single layer. We have to use multiple layers to escape all of them. In this case, we can extend our model to find out the minimum number of layers to escape all the pins. Supposing we are given k layers, we can build k copies of our network. Then inter-layer edges are added between adjacent layers to model the vias. Each inter-layer edge is from a to-be-escaped pin to the corresponding pin on the layer below (see Figure 2.17). The capacity of the edge is 1 and the cost is the cost of a via. The edges from the super source s connect only to the to-be-escaped pin on the top layer.

If the max-flow of this network and the number of to-be-escaped pins are the same, then k layers are enough to escape all the pins. Otherwise, we need more layers. We can use binary search on k to find out the minimum layer number such that all the pins can be escaped.

CHAPTER 3

ESCAPE ROUTING OF DIFFERENTIAL PAIRS

3.1 Introduction

Previous works on escape routing [7,17,18,21–24,26] did not take differential pairs into consideration. A work on chip-package-board co-design by Fang et al. [42] considers differential pairs but assumes that the routing style is monotonic, which might not be the case in many practical designs. In this chapter, we study the unordered escape routing problem of differential pairs.

In PCB designs, high frequency signals are usually transmitted through differential pairs. A *differential pair* is a pair of wires transmitting two complementary signals. The actual signal is obtained by subtracting the two signals at the receiver side (see Figure 3.1). By subtracting the two complementary signals, the noise and interference on the two wires are subtracted away while the actual signal is amplified. Therefore, a differential pair has the advantages of higher tolerance of ground offsets, better noise immunity and higher resistance to electromagnetic interference. However, to achieve these advantages, the differential pair must be carefully routed. They should be routed as close as possible to each other so that they receive the same noise and perturbation from the environment. A typical pin grid on a PCB allows two wiring tracks between adjacent pins. Therefore, we would like the

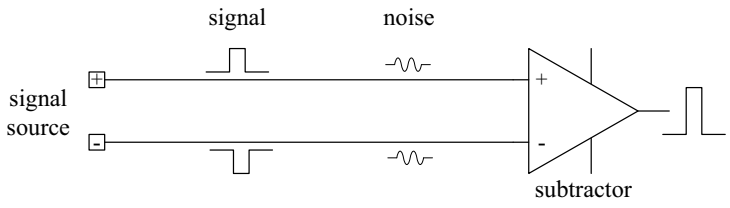


Figure 3.1: A differential pair is a pair of wires transmitting complementary signals. The two signals are subtracted at the receiver side to obtain the actual signal.

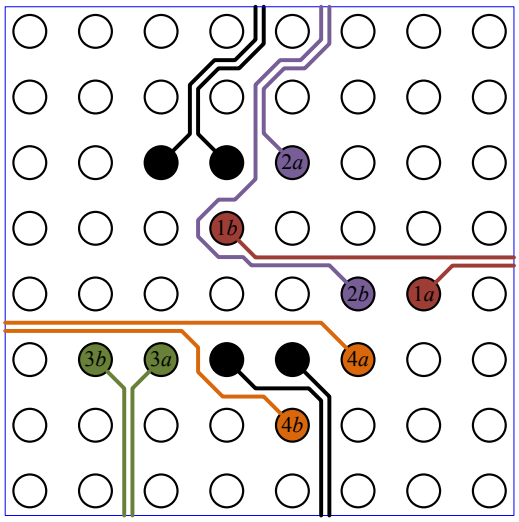


Figure 3.2: An example of differential pair escape routing. Pins with the same prefix (e.g. $1a$ and $1b$) are a differential pair. Routing from black pins are pre-routed pairs and are treated as obstacles.

wires of a differential pair to occupy such adjacent routing tracks as much as possible. Figure 3.2 gives an example of differential pair routing. It can be seen that two wires of a differential pair try to meet each other as soon as possible and then stay on adjacent tracks after they meet.

If the two pins of a differential pair were adjacent, then routing them would be easy. However, practical designs may contain differential pairs with pins far apart from each other. Figure 3.3 shows the histograms of the distance between two pins of a differential pair in two industrial boards we have. It can be seen that even though the majority of the differential pairs have adjacent pins, a large portion of the differential pairs still have significant distances between their two pins. Many of them have Manhattan distance 4 between their two pins. Such a small distance can already imply non-trivial problems. Figure 3.4 shows such a case. This problem is difficult to solve if we use the net-by-net approach. If we route pair $\{2a, 2b\}$ first, then we would route them to the right, blocking pair $\{1a, 1b\}$. However, if we route pair $\{1a, 1b\}$ first, then in order to meet as soon as possible, their route will cut between $2a$ and $2b$. A similar issue will also occur between pair $\{2a, 2b\}$ and $\{3a, 3b\}$. To solve this kind of problem, we need an approach that has the global view of all the differential pairs.

In this chapter, we study the differential pair escape routing problem and

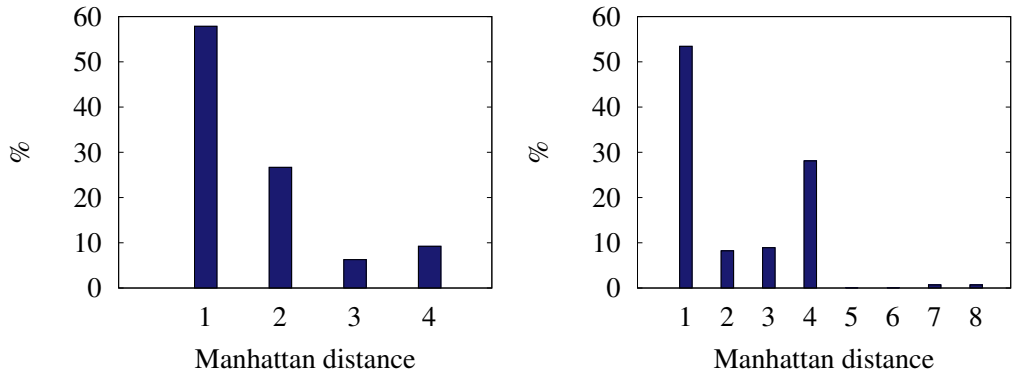


Figure 3.3: Histograms of the distances between the two pins of a differential pair in two industrial boards. The x-axis is the Manhattan distance between the two pins of a differential pair in terms of pin pitch. For example, if two pins are adjacent, then their distance is 1 (pitch). The y-axis is the percentage of the differential pairs that have the corresponding distance. Notice that a design usually has hundreds of differential pairs, so even a small percentage means many nets.

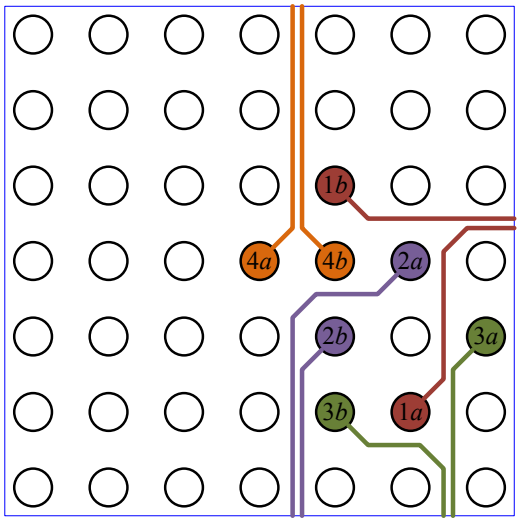


Figure 3.4: A non-trivial case. The Manhattan distance between any pair of pins is no larger than 4.

propose two algorithms. First, we propose an algorithm that computes the *optimal* routes for a single differential pair. This algorithm can be considered as the "maze router" for differential pair escape routing. We then propose another algorithm to simultaneously route multiple differential pairs. It first computes a set of candidate locations for wires of each differential pair to meet and then uses a network-flow approach to choose the meeting point for each differential pair from the candidates and compute the escape paths from the meeting point to the grid boundary. Our min-cost max-flow formulation is able to guarantee that the optimal meeting point is chosen to maximize the routability while minimizing the wire length. However, since the candidate locations computed at the first step might not be ideal, this algorithm may fail to produce good results for complex problems. Therefore, we propose a two-stage routing scheme based on the two algorithms: First, we use the simultaneous routing algorithm to construct initial routing of all the differential pairs. Then, we rip-up and reroute each differential pair using our optimal single pair routing algorithm to further improve the routability and wire length. Experimental results show that our routing scheme is very powerful in solving practical problems.

The rest of this chapter is organized as follows: Section 3.2 briefly introduces the routing constraints for differential pairs and then formulates the differential pair escape routing problem. Our two algorithms are presented in Section 3.3 and Section 3.4. Section 3.5 gives our overall routing scheme based on the two algorithms. Experimental results are then presented in Section 3.6. Finally, Section 3.7 gives the concluding remarks.

3.2 Background

In this section, we will first introduce the routing constraint of differential pairs and then we will formulate the differential pair escape routing problem and discuss its major difficulties.

3.2.1 Differential Pair Routing Constraint

As mentioned before, we need to carefully route the differential pairs in order to take full advantage of its benefits. The most critical requirement for

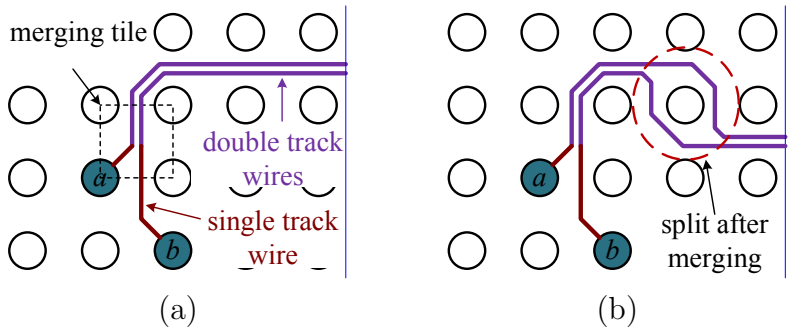


Figure 3.5: Ideal differential pair escape routing (a) can be viewed as two short single track wires from the two pins merging into double track wires. Splitting of the wires after merging (b) is illegal.

differential pair routing is that the two wires need to be routed as close to each other as possible. Typically, a pin grid on a PCB has two wiring tracks between adjacent pins. So in the context of escape routing, we expect the two wires from a differential pair to be routed along such adjacent tracks as much as possible. Therefore, ideal escape routing of a differential pair can be viewed as two *single track wires* from the two pins merging into *double track wires* and staying together till they reach the boundary (see Figure 3.5 (a)). We observe from industrial boards that manual routings all follow this merging style. We call the tile where the two wires merge their *merging tile*. If the escape routing of a differential pair follows the above pattern, we call the routing *legal*. Otherwise, if two wires never merge, or they merge and then split again (see Figure 3.5 (b)), we call the routing *illegal*.

The major objective of differential pair escape routing is to find legal routing such that the single track routing length is minimized. This is because if the two wires of a differential pair are not together, the noise they receive from the environment might be different and this can cause perturbation in the differential signal. On the other hand, we also want the total wire length to be minimized as well. Therefore, the routing cost of a differential pair is a weighted sum of its single track routing length and double track routing length with more weight on the single track length:

$$cost = st\_length + \alpha \cdot dt\_length \tag{3.1}$$

Here, st_length and dt_length are respectively the length of the single track wires and double track wires and α is a parameter to control the priority of

single track wires. For simplicity, length is evaluated as the number of tiles the wire traverses. For example, for the routing in Figure 3.5 (a), we have $st\_length = 1$ (single track wire from a passes no tile and single track wire from b passes one tile before they meet) and $dt\_length = 4$ (the double track wires pass through 4 tiles). Therefore, $cost = 1 + 4\alpha$.

If $\alpha = 0$, only the single track length is considered in the cost. Therefore, by minimizing this cost, we minimize only the single track routing length. If $\alpha = 2$, then we minimize the total wire length. (Remember double track wires contains two wires, so we need to multiply its length by 2 to get the total wire length.) Any value between 0 and 2 is a trade-off between the total wire length and the single track wire length. Usually, α is set to be a small constant to minimize the single track wire length in first priority while keeping the total wire length small.

3.2.2 Differential Pair Escape Routing Problem

Now with this cost function, we can formulate the differential pair escape routing problem as follows:

Problem 1. Given k pairs of pins $\{(1a, 1b), \ldots, (ka, kb)\}$ in an r row by c column pin grid and a set of pre-routed wires as obstacles, the *differential pair escape routing (DPER) problem* is to find legal routing paths from the pins to the boundary of the grid such that their total cost (computed by Equation (3.1)) is minimized.

There are two major difficulties of this routing problem:

1. Where to merge each differential pair is a big issue. The location of the merging tile affects the routability and length of both the double track wires and the single track wires. However, if we are dealing with multiple differential pairs, choosing the best merging location for one differential pair may increase the wire length of another differential pair or even make it unroutable. How to wisely choose the merging tiles so that all the differential pairs can be routed and the total cost can be minimized is the key to solving the DPER problem.

2. Even if we know a good merging tile for a differential pair, it is still difficult to determine how the single track wires and the double track

wires should be routed. If we route the single track wires first, then they become obstacles for the double track wires. Bad routing of single track wires may lead to longer double track wire length or even unroutable cases. The opposite is also true: routing double track wires first may also affect the routing of single track wires. How to route them so that both are routable and their total cost in Equation (3.1) is minimized is another key issue.

In the next two sections, we will present two algorithms. The first one is able to route a single differential pair optimally while the second one uses a network-flow approach to simultaneously route multiple differential pairs. The difficulties above are resolved by the two algorithms.

3.3 Routing One Differential Pair

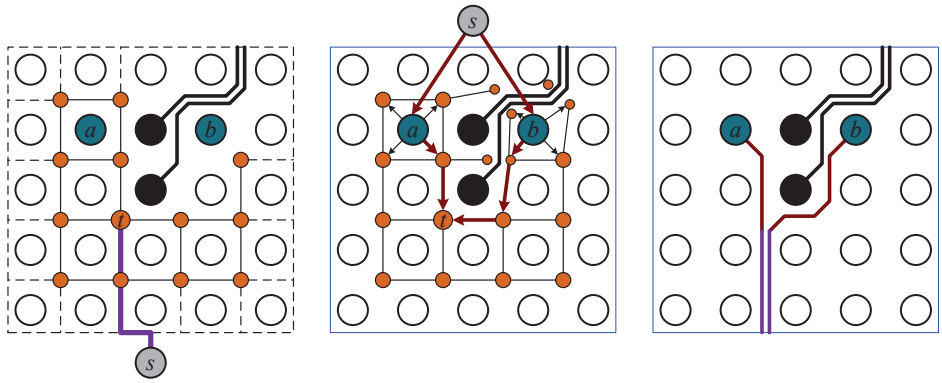
Let us first consider the most basic case of the DPER problem: routing only one differential pair. In this section, we propose an algorithm that finds the optimal routing paths for one differential pair in $O(n^2 \log n)$ time, in which n denotes the total number of tiles in the grid.

For only one differential pair, finding the best merging tile is not a difficult task because we can enumerate all $O(n)$ tiles to find the best one. However, the second difficulty mentioned in the previous section is still there. We need to carefully route the single track wires and double track wires so that the cost in Equation (3.1) is minimized.

Suppose we already know the merging tile for a differential pair (a, b) is t. We can then view the whole routing as two parts: (1) single track wires from a and b to t and (2) double track wires from t to the boundary of the grid. We compute the two parts separately.

To compute the double track routes, we construct an *undirected* routing graph G_D as follows (see Figure 3.6 (a)):

- Each empty tile is assigned a *tile node*.
- Adjacent tile nodes are connected by edges of cost 1 (solid edges in the figure).
- All the nodes of the boundary tiles are connected to a super source s by edges with cost 0 (dashed edges in the figure).



(a) (b) (c)

Figure 3.6: Routing one differential pair. (a) Routing graph G_D for double track wires; thick path shows the shortest path between s and t. (b) Network graph G_S for single track wires; thick arrows indicate the flow result. (c) Routing result by combining the results of (a) and (b).

By computing the shortest path from s to t in G_D, we can obtain the routing path for the double track wires.

We then construct a network graph G_S for single track routes. The graph is constructed as follows (see Figure 3.6 (b)):

- Each empty tile is assigned a *tile node*. Each tile with obstacle wires are partitioned into regions by those wires. Each region is assigned a *region node* (the smallest nodes in the figure).
- *Undirected* edges are added between adjacent tile/region nodes if there exists an available wiring track between the two nodes. All the edges have capacity 1 and cost 1. Notice that in a flow-network, an undirected edge allows flow in both directions. Such an edge a — b can be implemented by two directed edges $a \rightarrow b$ and $b \rightarrow a$, both with capacity 1 and cost 1.
- To prevent the single track wires from merging before they reach t, we enforce capacity 1 on all tile and region nodes.
- Two *pin nodes* are assigned to the two pins of this differential pair. *Directed edges* are added from each pin node to its four adjacent tile/region nodes. These edges have capacity 1 and cost 1.
- Finally, we create a super source s and add *directed* edges from it to the two pin nodes. Each edge has capacity 1 and cost 0.

By computing the min-cost 2-flow from s to t in this network, we can obtain the routing paths from the two pins to the merging tile t with the shortest total length.

Finally, we can combine the shortest path result of G_D (which represents the double track wires) and the flow result of G_S (which represents the single track wires) to compose the routing solution. In Figure 3.6, (c) is the resultant routing solution by combining the results of (a) and (b). Notice that this is the optimal solution assuming tile t is the merging tile. To obtain the global optimal, we need to enumerate all tiles in the grid and choose the best one.

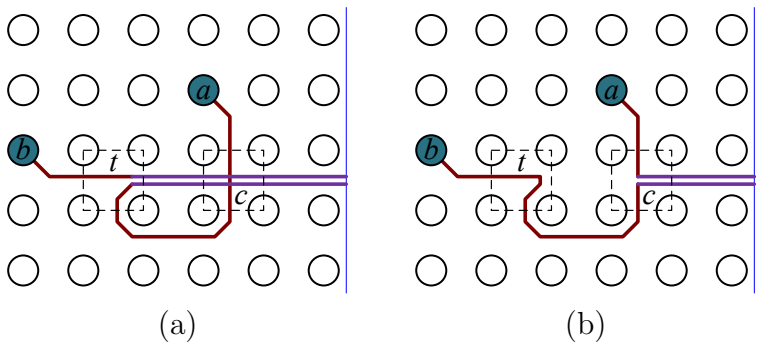
Now let us analyze the time complexity of this algorithm. Let n be the total number of tiles in the grid. We can compute the shortest path lengths from s to all tile nodes in G_D by Dijkstra's algorithm [40] in $O(n \log n)$ time because G_D contains $O(n)$ nodes and $O(n)$ edges. For each possible merging tile t, we can obtain the the min-cost 2-flow of G_S by computing the shortest path in the residual graph of G_S twice [40]. Again, since the residual graph of G_S contains $O(n)$ nodes and edges, this can be done in $O(n \log n)$ time. We have to compute the min-cost 2-flow for all $O(n)$ possible merging tiles. So the total time complexity on the flow computation is $O(n^2 \log n)$. Finally, we need to compare all $O(n)$ choices. As a result, the total time complexity of our algorithm is $O(n \log n + n^2 \log n + n) = O(n^2 \log n)$.

Notice that in our algorithm, we construct the single track wires and double track wires independently without considering each other. One natural question is whether the solution produced by our algorithm will have crossings between the single track wires and the double track wires. The following lemma shows that this cannot happen.

Lemma 3. *The optimal routing solution obtained by our algorithm does not have wire crossings.*

Proof. First of all, double track wires will not cross themselves because any shortest path in G_D (which has non-negative edge costs) does not have self-intersections. Since we enforce unit capacity on the tile/region nodes in G_S, single track wires cannot have crossings with themselves either. Therefore, crossing can only happen between single track wires and double track wires.

Suppose we have such a crossing in our solution. Without loss of generality, we assume the single track wires from the two pins a and b merge at tile t



(a) (b)

Figure 3.7: A crossing between the double track wires and single track wires (a) can be resolved, resulting in even shorter wire length (b).

and the single track wire from a crosses the double track wires at tile c as shown in Figure 3.7 (a). By changing the two single track wires to $a \rightarrow c$ and $b \rightarrow t \rightarrow c$, we move the merging tile to c and resolve the crossing (see Figure 3.7 (b)). The new routing is crossing-free and has the same single track wire length as the original routing. Moreover, the double track wire length is reduced. This means c is a better merging tile than t. In this case, our algorithm will not choose t as the optimal solution and will choose c instead. Therefore, in the optimal routing produced by our algorithm, there exist no wire crossings. □

With the above lemma and the optimality of the shortest path in G_D and the optimality of the min-cost 2-flow in G_S, we have the following claim:

Theorem 2. *Our algorithm computes the legal routing solution with minimum cost for one differential pair.*

In practice, we do not want the single track routes to be too long. Therefore, we can define a small constant λ as the maximum tolerable distance from the pins to the merging tile. When we enumerate the merging tiles, we only consider those with Manhattan distance less than λ to both pins. By doing this, we can reduce the time complexity down to $O(n \log n)$.

The algorithm we just presented can be regarded as the "maze router" for differential pairs. It can be used as a subroutine for more complicated routing algorithms.

3.4 Routing Multiple Pairs

For multiple differential pairs, we cannot afford to enumerate all possible combinations of the merging tiles because there are $\binom{n}{k}$ of them (n is the number of tiles and k is the number of differential pairs). In this section, we present a network-flow based algorithm that simultaneously determines the merging tiles for all differential pairs.

Recall that the primary objective is to minimize the single track wire length and notice that the single track wires of a differential pair can be viewed as a path from one pin to the other pin of the same differential pair. (See Figure 3.5 (a) for an example. Its single track wires can be viewed as a path from a to b via the merging tile.) Therefore, to minimize the single track wire length, planning the merging tiles along the shortest paths between the two pins is a plausible choice. However, for multiple differential pairs, we cannot simply use their shortest paths because they may have intersections. This essentially becomes a general routing problem: find disjoint paths connecting pairs of pins in a grid and minimize the total length of the paths. This problem has been studied extensively and popular solutions included maze router [43, 44] and negotiated congestion based router [45].

In our scheme, we first construct an *undirected* routing graph G_S^* as follows (see Figure 3.8 (a)):

- A *tile node* is assigned to each tile that contains no obstacles.
- Two tile nodes are connected by an edge if their tiles are adjacent.
- Each differential pair pin is assigned a *pin node*.
- A pin node is connected to its four adjacent tile nodes by four diagonal edges (could be less than four if adjacent tiles have obstacles).
- All the edges in G_S^* have capacity 1 and cost 1.

We then apply a negotiated congestion based router [45] on G_S^* to find disjoint paths connecting the pin pairs with minimum length.

In the routing solution of the previous step, each path corresponds to the single track wires of a differential pair and every tile node along the path is its candidate merging tile. Our task now is to choose one merging tile from the candidates for each differential pair and route double track wires from the

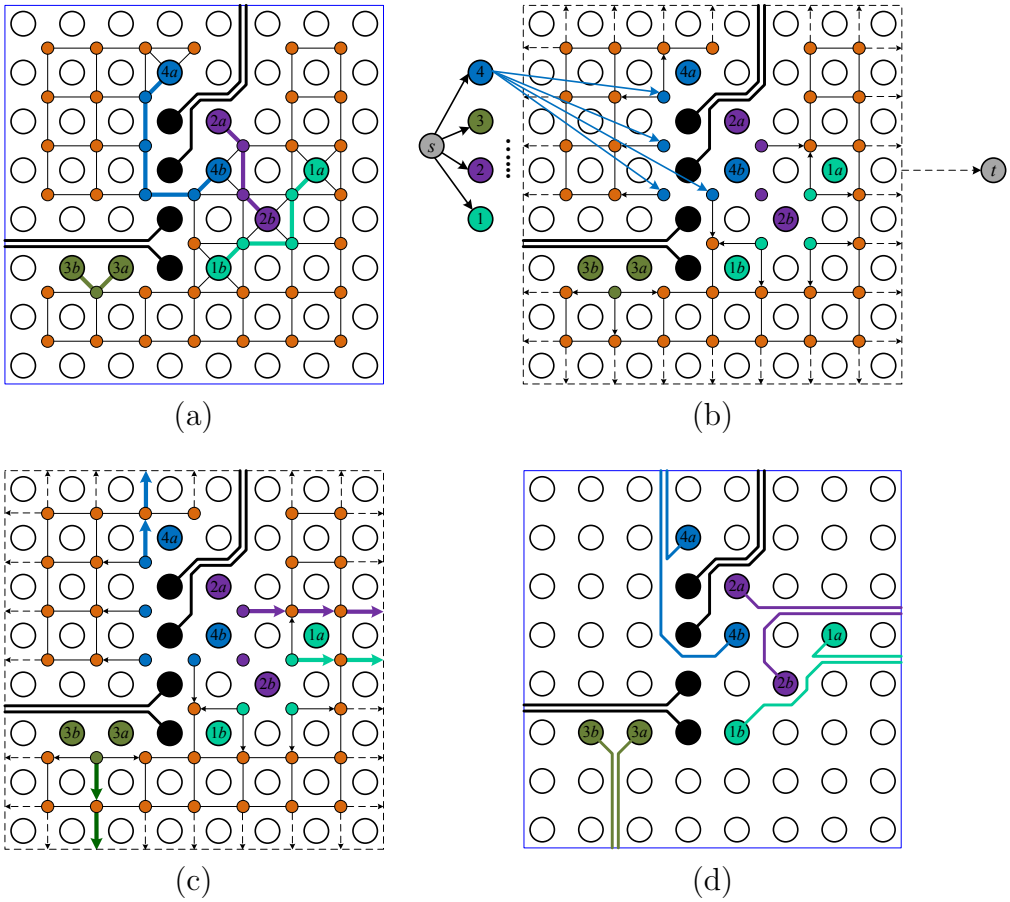


Figure 3.8: Routing multiple nets. (a) Routing graph G_S^* for single track wires; thick paths show the single wire routing paths for each differential pair. (b) Network graph G_D^* for double track wires; some edges are omitted to simplify the illustration. (c) Flow solution of (b). (d) Routing result by combining the result of (a) and (c).

chosen merging tiles to the boundary. Notice that the choices of the merging tile and the routing of double track wires affect each other as discussed in Section 3.2.2. To obtain the global optimal, we use a network-flow approach to simultaneously choose the merging tile and route the double track wires. We construct the network graph G_D^* by the following modification to the routing graph G_S^* constructed in the previous step (see Figure 3.8 (b)):

- All the undirected edges between tile nodes remain in G_D^*. These edges have capacity 1 and cost 1. (Recall that an undirected edge allows flow in both directions.)

- The diagonal edges between pin nodes and their neighboring tile nodes

are removed.

- For every tile node along the disjoint paths computed in the previous step (which are candidate merging tiles), the edges incident to it are oriented such that flow can only flow out of it. That is, the undirected edge becomes *directed* from the candidate merging tile node to other nodes. If both ends of an edge are candidate merging tile nodes, we remove the edge. The purpose is to prevent the flow, which represents the double track wires from intersecting with the routed single track wires.

- We introduce a super source s as well as k *representative nodes* representing the k differential pairs. We add *directed* edges with capacity 1 and cost 0 from s to each representative node and also from each representative node to the candidate merging tile nodes of the corresponding differential pair.

- Finally, we add a directed edge from every boundary tile node to a super sink t for each opening between two adjacent pins along the boundary (see the dashed edges in the figure). Such edges have capacity 1 and cost 0. These edges collect all the flows from s that escape out of the pin array and send them to t.

By computing the min-cost max-flow [40] of this network, we can essentially choose the optimal merging tiles so that the routability of the double track wires is maximized and their wire length is minimized. This resolves the first difficulty mentioned in Section 3.2.2. In Figure 3.8, (c) shows the flow solution to the network in (b). Notice that since the flow solution is for double track wires, each flow represents two wires occupying adjacent tracks.

Now we can combine the routing solution in G_S^* that represents single track wires and the flow solution in G_D^* that represents double track wires to compose the complete routing solution. We do so by tracing the disjoint paths computed in G_S^* from the two pins until they meet at the merging tile selected by the flow algorithm. Then we merge the two wires and follow the flow solution until they reach the grid boundary. In Figure 3.8, (d) shows the routing result by combining the path solution in (a) and the flow solution in (c).

3.5 Overall Routing Scheme

3.5.1 Two-Stage Routing Scheme

One possible issue of the flow-based algorithm in the previous section is that the single track wires generated by the negotiated congestion based router might not be ideal for the double track wires. We may have produced single track routing that blocks later double track wires. To overcome this issue, we propose a two stage routing scheme for the DPER problem:

1. An initial solution for all differential pairs is constructed by the simultaneous routing algorithm presented in Section 3.4.

2. Then, the single differential pair algorithm in Section 3.3 is called to rip-up and reroute each differential pair to improve the routability and the routing cost.

This routing scheme has the following advantages:

1. The first stage is based on min-cost max-flow. It guarantees optimality of the double track routing when the single track routing is fixed.

2. The rip-up and reroute algorithm is able to find the optimal paths for one differential pair (considering other differential pairs as obstacles). Therefore, if the initial routing stage fails to route all the differential pairs in a complex design, the rip-up and reroute engine is able to efficiently find optimal paths for unrouted differential pairs and resolve the issue. Moreover, it is also able to reduce the wire length of the initial routing produced by the first stage.

Our routing scheme is shown by experiments to be very effective and efficient.

3.5.2 Single Net Consideration

The test benchmarks we obtained from industry consist of only differential pair nets. However, one might want to route differential pairs together with single nets on some occasions. In this case, we propose to solve the problem using a negotiated congestion strategy based on our two-stage routing scheme. First, we route the differential pairs using our proposed two-stage

Table 3.1: Experimental results of our two-stage differential pair routing scheme.

test cases	diff pair #	pin grid #row×#col	avg. len. ST	avg. len. DT	runtime (s)
ex1	10	11×8	0.4	2.9	1
ex2	18	22×7	0.3	4.6	3
ex3	18	18×12	0.5	4.3	7
ex4	8	11×3	0	2.9	1
ex5	11	14×3	0	2.9	1
ex6	11	17×6	0.1	2.8	2
ex7	18	17×6	0.8	1.8	35
ex8	20	9×16	0.4	3.4	5
ex9	20	8×15	0.4	3.2	3
ex10	60	35×35	1.6	8.6	453

routing scheme. We then obtain the congestion information from the routing solution. Second, we construct a flow network whose edge costs reflect the congestion information obtained from the differential pair routing result. Higher routing congestion leads to higher edge cost in the network. We then apply the network-flow algorithm to obtain the routing result for single nets. Again, we can obtain the congestion information of the single net routing. We can then repeat the first step, using such information to update the edge cost. By repeating the differential pair step and the single net step, the routing scheme will eventually converge to a solution that is reasonable for both differential pairs and single nets.

3.6 Experimental Results

We implement our routing scheme in C++ and test it on 10 test cases derived from industrial data. The experiments are performed on a Linux workstation with a 3 GHz Intel Xeon CPU and 4 GB memory. Detailed information about the test cases as well as the experimental results is shown in Table 3.1. Notice that the "avg. len." column shows the average number of tiles traversed by single track wires (ST) and double track wires (DT) of a differential pair. The last column gives the runtime of our algorithm.

From the table, it can be seen that our router can achieve very short single track wire length (less than 1) for almost all the data, which is ideal for

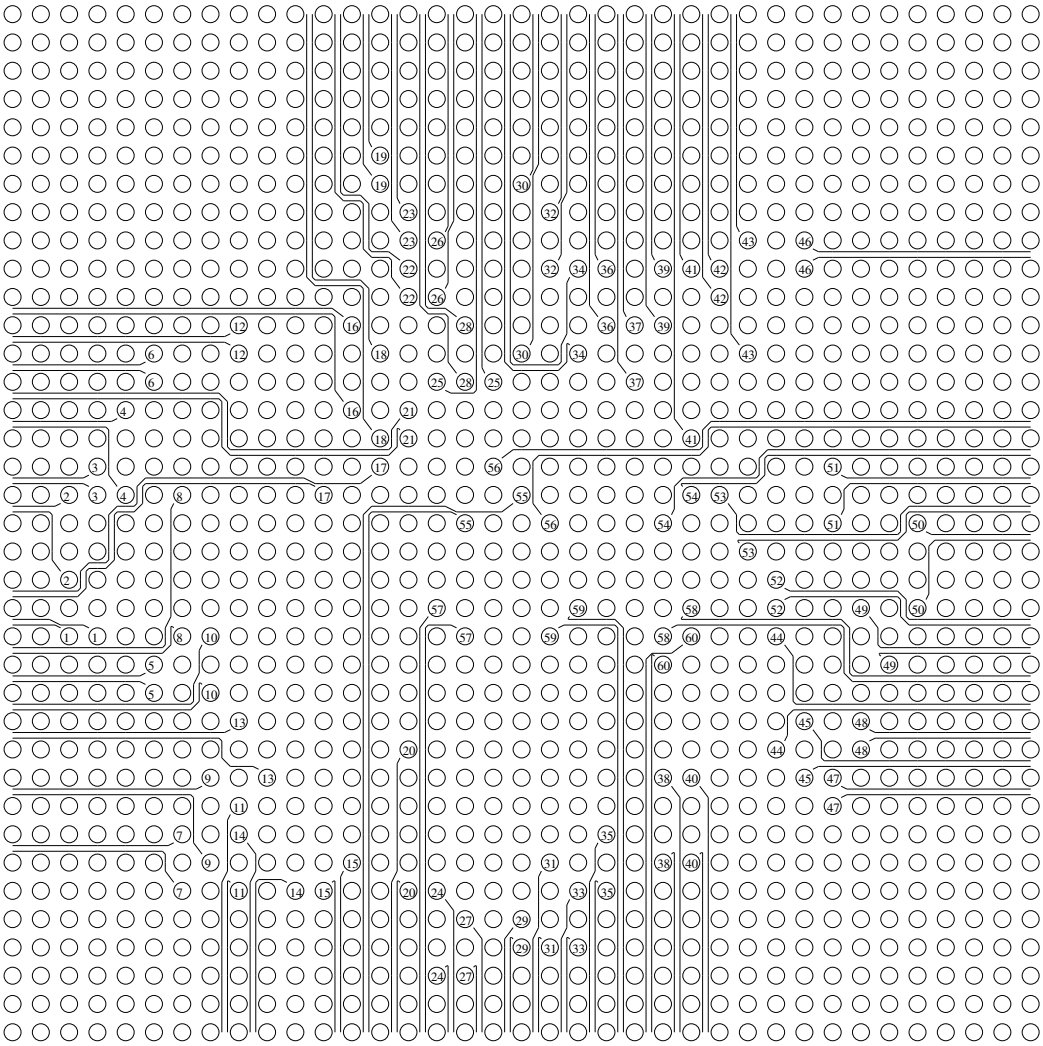


Figure 3.9: Routing result of ex10.

differential pairs. The double track wire length is also short, meaning that the total wire length is minimized too. The overall runtime for the test cases is very short (a few seconds) except for two test cases ex7 and ex10. In ex7, the pins of a differential pair are not located closely, resulting in longer single track wire length and more complicated routing patterns. Therefore, our router spent more time on it. The pin array of ex10 is actually very large and so is the number of differential pairs to be routed. One would expect such a test case to be among the largest in real designs. The routing result of ex10 is shown in Figure 3.9. It can be seen that the routing is very dense, indicating that our router is good at handling difficult designs.

3.7 Conclusion

In this chapter, we studied the differential pair escape routing (DPER) problem and proposed two algorithms. The first algorithm is essentially the differential pair version of the maze routing algorithm [43, 44]. It computes the routes for a differential pair with minimum routing cost. The second algorithm is a network-flow based algorithm that simultaneously routes multiple differential pairs. Because of its min-cost max-flow formulation, it is able to achieve maximum routability and minimum wire length of the double track wires. The two algorithms are then combined into a routing scheme that can effectively and efficiently solve the DPER problem.

We also proposed a negotiated congestion based approach to simultaneously route both differential pairs and single nets. However, due to lack of data, we were not able to verify the effectiveness of our approach.

CHAPTER 4

LENGTH-CONSTRAINED ROUTING

4.1 Introduction

Due to the high clock frequencies on today's high-performance PCBs, the nets in a bus are required to satisfy very stringent min-max length bounds [2]. Because the routing resources are very limited inside the pin arrays, the space between the pin arrays is used to detour the wires to meet the length bounds. The escape routing inside the pin arrays is not of interest in this chapter. Therefore, we view each pin array as a rectangular block which we call a *component* and regard the ends of the escape routing around the component boundaries as the pins for the length-constrained routing problem (see Figure 4.1).

Such a length-constrained routing problem is not easy to solve, even for simple routing topologies. The reason is that different nets with different length constraints are competing for the routing resources. We must carefully distribute the limited routing resources to these nets while keeping the planarity and connectivity.

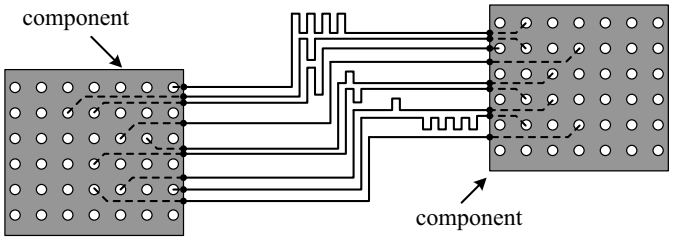
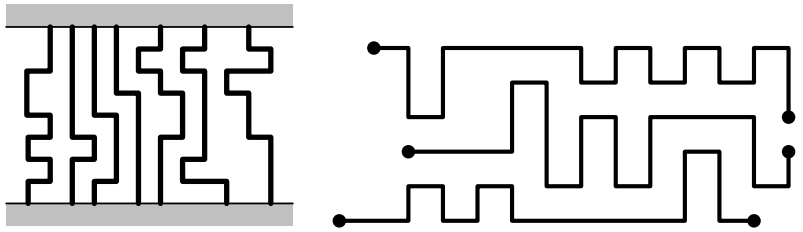


Figure 4.1: Length-constrained routing between pin arrays (solid lines). Each pin array is viewed as a rectangular block and the ends of the escape routing are regarded as the pins of the length-constrained routing problem (black dots). Escape routing inside the components is ignored (dashed lines).



(a) (b)

Figure 4.2: Topological restrictions on previous routers. The routers in [46, 47] can only solve the channel routing problem shown in (a) and the router in [15] routes wires monotonically, as in (b).

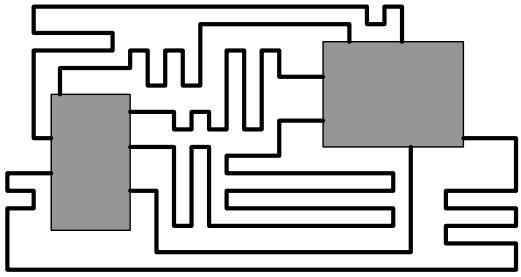


Figure 4.3: A length-constrained problem with general topology.

This length-constrained routing problem has been studied in [15, 46, 47]. However, the routing topologies those works can handle are limited: [46, 47] can only be applied when the routing region between the two components forms a channel and the pins are located along the two opposite sides of the routing channel (see Figure 4.2 (a)) and [15] assumes that each wire must be routed monotonically in one direction, say, from left to right (see Figure 4.2 (b)). Such restrictions greatly limit the application of these routers. For example, none of the three routers can be applied to the problem in Figure 4.3 because the problem is not a channel routing problem and it involves both horizontal and vertical detour.

Another disadvantage of previous routers is that they are all gridded. Modern PCBs usually have very fine wiring pitch and a long distance between components, making the size of the routing grid very large for gridded routers even though the solution may look very simple.

In this chapter, we propose a length-constrained routing scheme that is capable of handling any given topology. The novelty in our approach is that we regard the length-constrained routing problem as an area assignment prob-

lem and use a placement structure, bounded-sliceline grid (BSG) [14], to help formulate the area assignment problem into a mathematical programming problem. We also propose an iterative LP optimization procedure that can efficiently solve the formulated mathematical programming problem.

Our routing scheme has the following virtues:

1. It is the first length-constrained router that is free of any topological restrictions. It handles general topology.

2. It produces a gridless routing. Its performance is not sensitive to the routing grid size of the input while previous routers' performances are. Therefore, our approach is much faster in practice because practical designs usually have very large routing grids.

3. It is area-focused. Experimental results show that it can efficiently utilize the routing area for wire extension to meet the length bounds.

4. It can be extended to handle length-matching routing, multi-layer routing and different separation rules for different nets.

A similar work worth mentioning is the Oct-Touched Tile (OTT) structure proposed by Fu et al. [48] for routing analog circuits. In order to route wires of different widths, they embed the topological routing onto OTT, a structure similar to BSG, and then apply a longest path algorithm on the horizontal and vertical constraint graph to assign the area and obtain the detail routing. Our approach differs from theirs in the following aspects:

1. We focus on a different problem. We are focusing on controlling the length of each wire through area assignment while [48] focuses on controlling the width of each wire.

2. To meet the length constraint, we formulate the area assignment problem into a mathematical programming problem while the area assignment in [48] does not need to meet any length constraint and is therefore formulated into a much simpler longest path problem.

3. After area assignment, we still need to perform non-trivial detail routing, while in [48] detail routing and area assignment are the same thing.

The rest of this chapter is organized as follows: Section 4.2 gives some necessary background. Section 4.3 describes our length-constrained routing scheme for general topology. Section 4.4 presents some extensions of our router. Experimental results are presented in Section 4.5, and Section 4.6 concludes this chapter.

4.2 Background

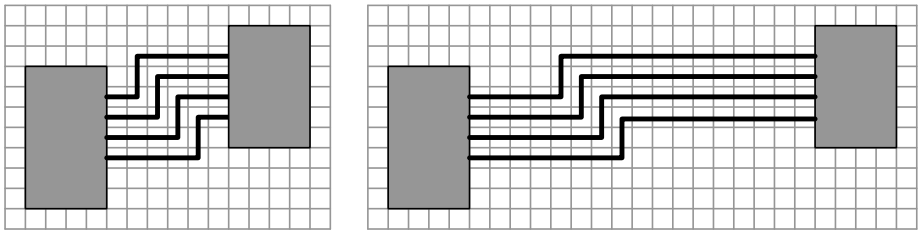
In this section, we will first introduce the length-constrained routing problem and then give a brief review of the bounded-sliceline grid (BSG) structure.

4.2.1 The Length-Constrained Routing Problem

Bus routing for a PCB is usually divided into two phases: escape routing and area routing (see Figure 1.1). Escape routing is to route from the pin array inside a component to the boundary of the component. Area routing is to complete the connections between the boundaries. Escape routing and area routing have different tasks. Escape routing must guarantee that the net orderings on both boundaries are matched in order to provide a planar topology for area routing. The focus of area routing, on the other hand, is to carefully detour the wires to meet the length bounds while maintaining the planar topology inherited from escape routing. That is why we also call area routing "length-constrained routing."

Due to the effort of the escape router, the input to the length-constrained router is usually regarded as planar. However, previous routers also make assumptions on the routing topology: [46] and [47] assume that the two boundaries to be connected are facing each other as in Figure 4.2 (a), and [15] assumes that the wires detour in only one direction, i.e., the routing can detour either horizontally or vertically but not both (see Figure 4.2 (b)). As a result, none of them can solve the routing problem in Figure 4.3. One remedy for this issue is to first route from both boundaries to some virtual boundaries facing each other and then perform length-constrained routing between the virtual boundaries [11]. However, this means that the area around the components cannot be fully utilized for wire extension.

Furthermore, previous routers are all gridded. The ratio of the area of the



(a) (b)

Figure 4.4: Different distances between the two components can make the problem size very different for a gridded router.

input routing domain to the minimum wiring pitch defines the size of the routing grid. We call this grid size *input routing grid size* because both the routing domain and the wiring pitch are input-dependent. The performance of a gridded router is very sensitive to the input routing grid size. For example, the two problems in Figure 4.4 make a huge difference to the router due to the very different grid size, although the two problems look quite similar in human designers' eyes. Moreover, modern PCBs usually have very small wiring pitch and the components they host might be located far apart. This means that the input routing grid size can be very large, making gridded routers unbearably slow.

Now we formulate the general topology length-constrained routing problem as follows:

Input:

- The location and size (width, height) of a set of rectangular components.
- The location of a set of pins. The pins should be located along the boundaries of the components.
- A set of 2-pin nets connecting the given pins.
- A planar topology of the 2-pin nets.
- A rectangular domain for the routing. All the wires are bounded within this domain.
- Design rules: wire width ω and separation rule ε for the wires. Any wire segment must be separated from other wire segments or the components

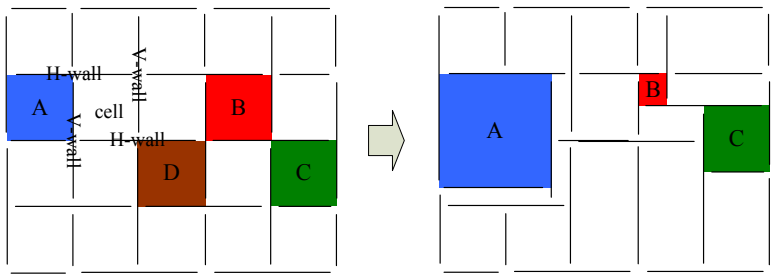


Figure 4.5: The BSG structure and cell sizing.

by a distance of ε. The wiring pitch λ, which is the minimum distance between the center lines of wire segments, is then the sum of the wire width and separation rule: $\lambda = \omega + \varepsilon$.

- Length bounds (l_i, u_i) for each net i. l_i is the lower bound and u_i is the upper bound. $l_i \leq u_i$.

Output:

- A rectilinear routing following the given topology and satisfying the design rules. The length of each net i satisfies $l_i \leq length_i \leq u_i$.

4.2.2 Bounded-Sliceline Grid

Bounded-sliceline grid (BSG) [14] was invented to handle the placement of function modules in ICs. It uses a set of vertical and horizontal segments to partition the whole plane into rectangular cells. Each cell in the grid is bounded by four segments, two vertical (which we call *V-walls*) and two horizontal (which we call *H-walls*). Each wall spans two cells. If we let all the cells be unit size (1×1), then every wall will have a length of 2 and the BSG will form a uniform grid. See the left side of Figure 4.5 for an illustration.

The size and location of a cell are determined purely by the positions of the four walls surrounding it. By moving the walls, we can enlarge a cell (cell A in Figure 4.5), shrink a cell (cell B), move a cell (cell C) or make a cell vanish (cell D).

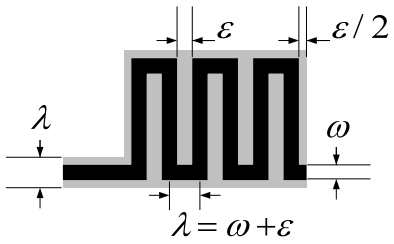


Figure 4.6: The length of a wire and the area it occupies (black wire plus gray margin) are related by the wiring pitch λ.

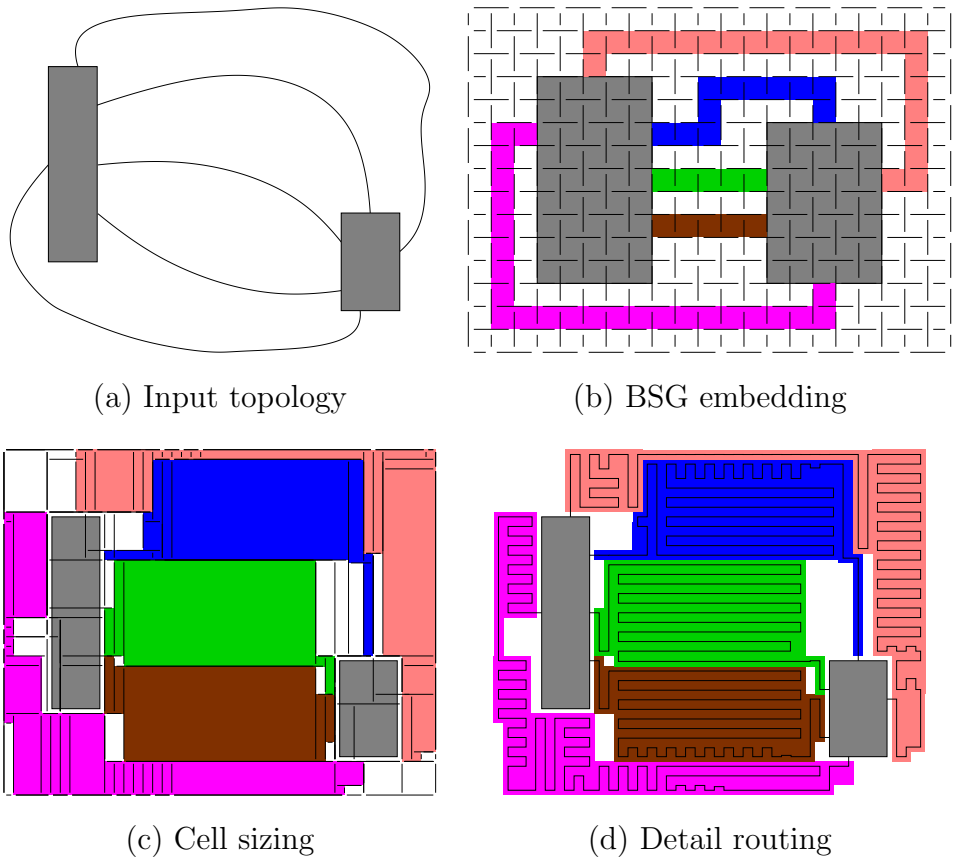
4.3 Our BSG-Route

In this section, we will present our BSG-based length-constrained routing scheme. We will first explain the general idea and then discuss the details.

4.3.1 The Idea

The key issue of length-constrained routing is how to control the length of the wires. An interesting observation is that the area a wire of length l occupies is exactly $l \times \lambda$ (recall that $\lambda = \omega + \varepsilon$ is the wiring pitch). This is because a wire will not only occupy a space of its width ω but also two margins of width $\varepsilon/2$ on its two sides to guarantee the ε separation rule (see Figure 4.6). Therefore, we can regard the wire as a fat wire of width λ and assume zero separation rule. The relationship between the length and area of a wire gives us an alternative to control the length. Instead of thinking about how to detour the wires to meet the length bounds, we can control the length by assigning proper area to the nets. This idea leads to our area-based routing scheme: BSG-route.

Our routing scheme is as follows (see Figure 4.7): First, we embed the given topology onto a BSG. Then, we size the cells in the BSG so that the total area of the cells occupied by a net satisfies its length bounds. Finally, we perform detail routing inside each cell to turn the assigned area into expected length. In the rest of this section, we will discuss each step in detail.



(a) Input topology (b) BSG embedding

(c) Cell sizing (d) Detail routing

Figure 4.7: An illustration of our idea.

4.3.2 BSG Embedding

First, we need to map the components, pins and nets onto a BSG. This can be done either manually or by heuristic algorithms. There may exist multiple embeddings for the same topology. Which embedding we choose has little impact on the final routing result as long as we follow the guidelines below:

1. BSG is a structure that represents *left-right*, *above-below* relations between objects. Therefore, the topological relation between objects should be kept in the embedding. If one component is located to the left of the other, then it should still be on the left in the embedding. Similarly, if a pin is above another pin, its cell should also be above the other's cell.

2. The number of BSG cells we use does NOT depend on the input routing grid size because we can later size the cells to match the routing domain. Instead, the BSG size depends on how complex the routing is.

If we want to represent more complicated routing, we should use more cells for the embedding (that is, add more BSG cells between the two components). However, we should not use an excessively large BSG because this will enlarge the scale of the later cell sizing problem. Our experience is that a 200×200 BSG would provide enough complexity for a routing problem with around 100 nets. Since the number of connections between two components on one layer of a high-performance PCB is usually less than 100, a 200×200 BSG should be more than enough for practical cases. This guideline also explains why our router is insensitive to the input routing grid size.

3. We need to allow at least one empty cell between two nets. If two adjacent cells are occupied by two different nets, moving the wall between them affects the areas of both nets. This means we lose the flexibility of controlling their areas independently.

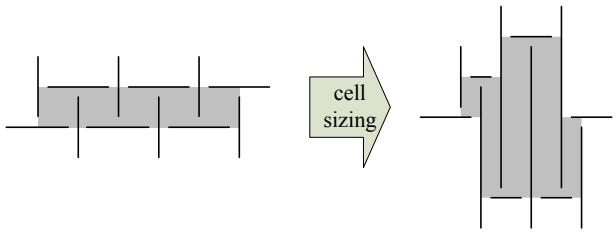
4. A component should be mapped into multiple cells forming a rectangular area. The number of cells it occupies depends NOT on its physical size but on the number of pins around its boundary. A component with more pins requires more BSG cells because each pin takes up an individual cell on the boundary. Notice that we also need to plan at least one empty cell between adjacent pins according to guideline 3.

As we claimed before, the selection of embeddings has little impact on the final result as long as we have enough BSG cells. This is because even with different BSG embeddings, we are able to obtain the same area assignment through careful cell sizing. In the example shown in Figure 4.8, we can still obtain the same shape and size of assigned area by cell sizing even though the initial embeddings are different.[1] Therefore, the key step of our router is cell sizing.

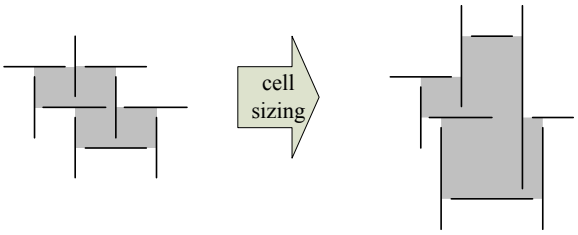
4.3.3 Cell Sizing

After we embed the topology onto a BSG, we size the BSG cells for area assignment. Essentially, the cell sizing problem is to determine the location

[1]However, the embedding in Figure 4.8 (a) does require one more cell than Figure 4.8 (b) to achieve the same area assignment. That is why we need to use enough cells for complicated routing.



(a) One BSG embedding and its cell sizing result



(b) Another BSG embedding and its cell sizing result

Figure 4.8: Different embeddings can lead to the same area assignment.

of the walls. We can formulate it as a mathematical programming problem. For every V-wall v, we use a variable x_v to represent its x coordinate. (Notice that the y coordinates of its two ends are not determined by the V-wall itself. They are determined by the positions of the two H-walls at its two ends.) For every H-wall h, we use a variable y_h to represent its y coordinate. For every cell i, we have four walls surrounding it. We name the variables representing its left, right, bottom and top walls as $x_{i,l}$, $x_{i,r}$, $y_{i,b}$ and $y_{i,t}$, respectively (see Figure 4.9). Notice that these names are only aliases of actual variables. Different names may refer to the same variable. For example, $x_{i,r}$, $x_{j,l}$, $x_{p,r}$ and $x_{q,l}$ all refer to the same variable x_v of wall v in Figure 4.9.

To guarantee that the final routing is legal and the length constraints are satisfied, we need to enforce the following constraints on the variables:

Basic Constraints

Every cell of the BSG must have a nonnegative area. That is, the top wall of a cell cannot be placed below the bottom wall of the cell and the right wall of a cell cannot lie to the left of the left wall of the same cell. Therefore, for each cell i we have the following constraints (assuming a coordinate system

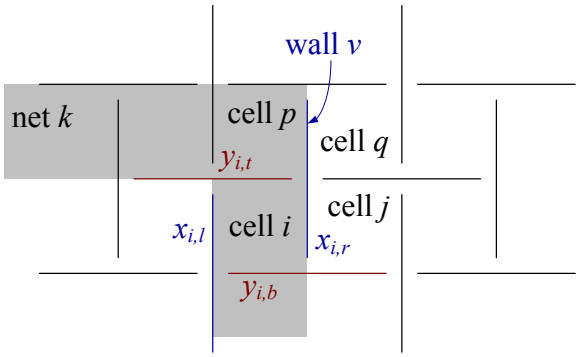


Figure 4.9: Each wall in the BSG is assigned a variable to represent its position.

with x-axis pointing right and y-axis pointing up):

$$x_{i,r} - x_{i,l} \geq 0 \tag{4.1}$$

$$y_{i,t} - y_{i,b} \geq 0 \tag{4.2}$$

Furthermore, if a cell is occupied by a net, we must make sure that the size of the cell allows one wire to pass. Since we regard the wire as a fat wire with width λ, this means that both the width and the height of the cell should be at least λ:

$$x_{i,r} - x_{i,l} \geq \lambda \tag{4.3}$$

$$y_{i,t} - y_{i,b} \geq \lambda \tag{4.4}$$

So the basic constraints for a cell are either (4.1), (4.2) or (4.3), (4.4) depending on whether the cell is empty or occupied.

Location constraints

The locations of the components and the pins are given in the input and the wall locations should conform to them. Therefore, we have to fix the walls on the four boundaries of a component. For a component C with its left-bottom corner located at (x_C, y_C) and width w_C and height h_C, we have

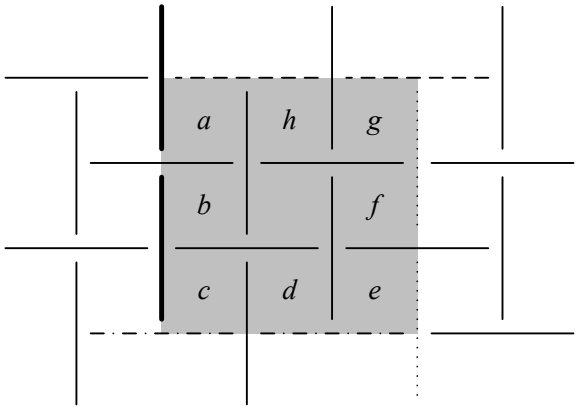


Figure 4.10: Illustration of component location constraints.

the following constraints:

$$x_{i,l} = x_C \qquad \text{for cells } i \text{ on the left boundary} \tag{4.5}$$

$$x_{j,r} = x_C + w_C \qquad \text{for cells } j \text{ on the right boundary} \tag{4.6}$$

$$y_{p,b} = y_C \qquad \text{for cells } p \text{ on the bottom boundary} \tag{4.7}$$

$$y_{q,t} = y_C + h_C \qquad \text{for cells } q \text{ on the top boundary} \tag{4.8}$$

Here we use an example to illustrate these constraints. Suppose a component C occupies 3×3 BSG cells (see Figure 4.10). Then we have the following location constraints:

$$\begin{aligned} x_{a,l} = x_{b,l} = x_{c,l} &= x_C && \text{(thick walls)} \\ x_{g,r} = x_{f,r} = x_{e,r} &= x_C + w_C && \text{(dotted walls)} \\ y_{c,b} = y_{d,b} = y_{e,b} &= y_C && \text{(dash-dot walls)} \\ y_{a,t} = y_{h,t} = y_{g,t} &= y_C + h_C && \text{(dashed walls)} \end{aligned}$$

If a pin p is located on the top boundary of a component (see Figure 4.11), then the cell it occupies is also on the top boundary of the component in the BSG embedding. Therefore, its y-coordinate y_p is already fixed by constraint (4.8). We only need to introduce the following constraints to fix it at its given x-coordinate x_p (suppose its BSG cell is i):

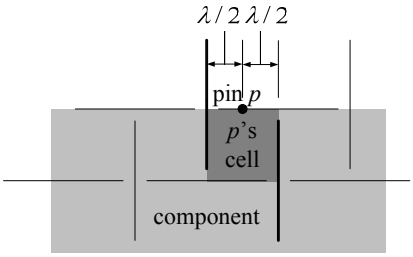


Figure 4.11: Location constraints on pin p. The two thick walls are constrained by equations (4.9) and (4.10).

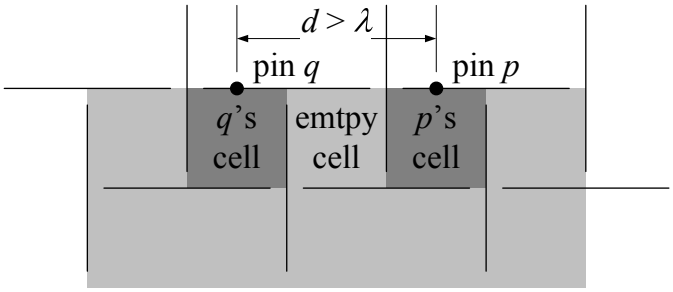


Figure 4.12: The empty cell between the two pins provides the necessary space if the distance between two pins is larger than the wiring pitch.

$$x_{i,l} = x_p - \lambda/2 \tag{4.9}$$

$$x_{i,r} = x_p + \lambda/2 \tag{4.10}$$

Again, since we view wires as fat wires, we leave a $\lambda/2$ margin on both sides. The constraints for pins located on the other three boundaries can be derived in the same way.

Notice that according to guideline 4, we should plan at least one empty cell between two adjacent pins. Therefore, if two adjacent pins have a distance $d > \lambda$, the empty cell in between will provide the extra space so that constraints (4.9) and (4.10) can be satisfied for both pins (see Figure 4.12).

Last, we need to fix the walls on the boundaries of the entire BSG at the boundaries of the input routing domain. This can be done by using constraints (4.5), (4.6), (4.7) and (4.8), imaging that the entire routing domain is a big component.

Length Constraints

We also need to put constraints on the sizes of the cells to satisfy the length bounds. For a net k, we have constraint (4.11) for its upper length bound u_k and constraint (4.12) for its lower length bound l_k (in both constraints, i denotes all the BSG cells occupied by net k):

$$\sum_i((x_{i,r} - x_{i,l}) + (y_{i,t} - y_{i,b}) - \lambda) \quad \leq \quad u_k \tag{4.11}$$

$$\frac{\sum_i((x_{i,r} - x_{i,l})(y_{i,t} - y_{i,b}))}{\lambda} \quad \geq \quad l_k(1+\delta) \tag{4.12}$$

The discussion of how we obtain these constraints and what δ stands for involves the explanation of how we route inside each cell, so we postpone it until Section 4.3.5.

4.3.4 Solve the Cell Sizing Problem

By putting all these constraints together, we have a feasibility problem: find a solution satisfying all the constraints (4.1) – (4.12). Although the problem contains only linear and quadratic constraints, it is not a convex programming problem because the quadratic constraint (4.12) is not convex. Discussions with an expert in mathematical optimization reveal that this problem is fundamentally a non-convex optimization problem [49], and as such is not amenable to many of the classical convex programming formulations known in operations research. Therefore, we use another approach to tackle this problem.

We first transform this feasibility problem into an optimization problem by relaxing the length constraints (4.11) and (4.12) with a slack variable s. The length constraints become

$$\sum_i((x_{i,r} - x_{i,l}) + (y_{i,t} - y_{i,b}) - \lambda) - s \leq u_k \tag{4.13}$$

$$\frac{\sum_i((x_{i,r} - x_{i,l})(y_{i,t} - y_{i,b}))}{\lambda} + s \geq l_k(1+\delta) \tag{4.14}$$

It can be seen that when $s \to +\infty$, both constraints are satisfied for arbitrary x and y (notice that x and y are bounded by the routing domain so they cannot be infinite). When $s \leq 0$, the two constraints become at least as strict

as (4.11) and (4.12), i.e., any x and y satisfying these constraints satisfy (4.11) and (4.12). Therefore, instead of finding a solution to the original feasibility problem, we try to minimize s in the new problem with relaxed length constraints.

Relaxed Problem

minimize:	s
satisfying:	basic constraints (4.1) to (4.4)
	location constraints (4.5) to (4.10)
	relaxed length constraints (4.13) and (4.14)

If we can minimize s to 0 or less, then we find a feasible solution to our original problem.

This optimization problem is still nonlinear because (4.14) is still nonlinear. However, if we fix all x as constants, then (4.14) becomes a linear constraint for y. Similarly, if we fix all y as constants, then (4.14) becomes a linear constraint for x. Therefore, we can solve this nonlinear optimization problem by solving a series of linear programming (LP) problems: we first fix x as constants and solve for y. Since all constraints are linear with respect to y, the relaxed problem becomes an LP problem. Then we do the opposite: we fix the resultant y as constants and solve for x. Again, we can solve this problem via linear programming. We repeat such iteration, each time fixing x (or y) variables obtained from the previous iteration as constants and solving for y (or x). The iterations terminate if s becomes 0 or less, or if s cannot be further reduced (even though it is still positive). If the latter happens, it means we fail to find a feasible solution to our cell sizing problem. There are several possible reasons:

- The given problem is infeasible, possibly because the length bounds are too tight. We should relax the length bounds. We can also enlarge the routing domain if the original length bounds are too long.
- The BSG size is too small to represent the complex routing solution. We should increase the BSG size.
- Our iterative LP approach is stuck at a local optimum. We need a better solver for the cell sizing problem.

Since we need to fix x as constants in the first iteration, we need an initial assignment of x to start with. We can obtain this assignment by solving another LP problem:

Initial Problem

find:	feasible x
satisfying:	basic constraints involving x ((4.1), (4.3))
	location constraints involving x ((4.5), (4.6), (4.9), (4.10))

We take any feasible solution of this problem as the initial x assignment to start our iteration. The complete algorithm of our iterative procedure is shown in Alg. 1.

Algorithm 1 Iterative LP Optimization

```
1: solve Initial Problem to obtain initial x
2: i ← 0
3: repeat
4:    if i is even then
5:       fix x as constants and solve Relaxed Problem for y
6:    else
7:       fix y as constants and solve Relaxed Problem for x
8:    end if
9:    i ← i + 1
10: until s ≤ 0 or s stops decreasing
```

The following theorem guarantees the convergence of our approach.

Theorem 3. *The objective s converges in our iterative LP optimization.*

Proof. In our iterations, the solution of one iteration is automatically a feasible solution to the relaxed problem of the succeeding iteration. Therefore, the objective value s can only decrease or remain the same because the solution of a minimization problem should have the smallest objective value among all feasible solutions. If s remains the same after an iteration, we will terminate. Otherwise s keeps decreasing. As a result, the objective s after each iteration forms a strictly decreasing sequence with a lower bound 0. Therefore, this sequence converges. □

Experiments show that we usually achieve $s \leq 0$ in 2 to 3 iterations.

4.3.5 Detail Routing

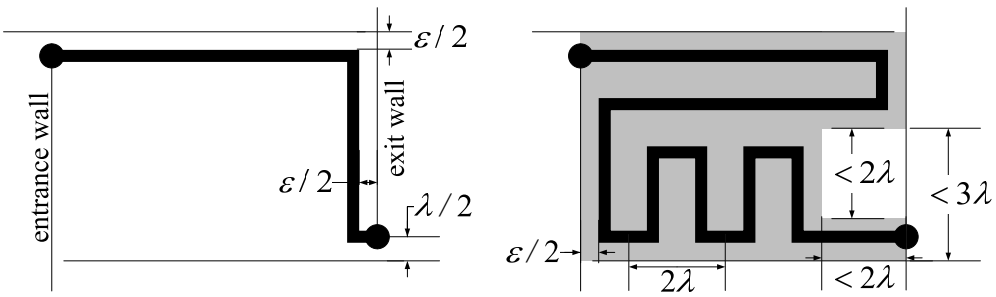
Before we explain how to route inside a cell, we first introduce some definitions. In the BSG embedding, a net must enter a cell by going through a wall and exit the cell by going through another wall. We call the first wall the *entrance wall* of the cell and the second wall the *exit wall* of the cell. For example, in Figure 4.9, the entrance wall of cell i is its top wall and the exit wall is its bottom wall, assuming net k comes from the left and turns to the bottom. If the entrance wall and the exit wall of a cell are of the same type (both H-walls or both V-walls), we call the cell a *straight cell*. Otherwise, we call it a *corner cell*. In Figure 4.9, cell i is a straight cell and cell p is a corner cell.

For any cell occupied by a net, we always route from a point on its entrance wall, which we call the *entrance point*, to a point on its exit wall, which we call the *exit point*. The entrance point is located $\lambda/2$ away from the corner where the entrance wall meets another wall. Similarly, the exit point is located $\lambda/2$ away from the corner where the exit wall meets another wall (see the dots in Figure 4.13). Therefore, the exit point of a cell always coincides with the entrance point of the succeeding cell in a net, making the wire continuous between cells. Notice that a pin is also located $\lambda/2$ away from the corner of its cell according to the pin location constraints, which means it is either an entrance point or an exit point. Therefore, by completing the detail routing from entrance point to exit point inside every occupied cell, we can obtain a continuous pin-to-pin routing for all the nets. The $\lambda/2$ distance guarantees that the wire is separated from the wall by $\varepsilon/2$ so that wires in different cells do not violate the ε separation rule.

For a straight cell i, the minimum routing length from the entrance point to the exit point is their Manhattan distance $w_i + h_i - \lambda$ (w_i and h_i denote the width and height of the cell). Now it becomes clear why we have constraint (4.11). The shortest length we can route for a net is the sum of $w_i + h_i - \lambda$ over all the cells occupied by the net. This length must be shorter than the upper length bound of the net.

To obtain greater length, we need to detour inside the cell. The following theorem shows that we can efficiently use the cell area for length extension:

Theorem 4. *For a straight cell with width $w \geq \lambda$ and height $h \geq \lambda$, there always exists a valid route ("valid" means the design rules are satisfied) from*



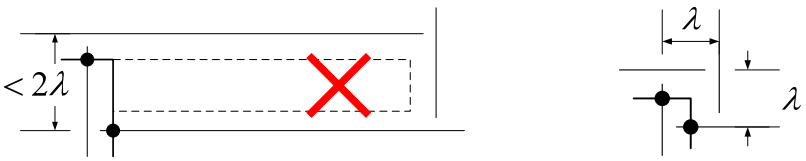
(a) Minimum routing length (b) Maximum routing length

Figure 4.13: The minimum and maximum routing length inside a BSG cell.

the entrance point to the exit point with length $l \geq \frac{wh}{\lambda} - 4\lambda$.

Proof. The proof is by construction (see Figure 4.13 (b)). We first detour horizontally (to the right and back) as much as possible until the space left is less than 3λ tall. Then we detour vertically (up and down) as much as possible and finally exit the cell at the exit point. Because each detour takes only 2λ width, the unused white space is at most 2λ wide because otherwise we will have space to make another detour. Notice that the last segment to the exit point occupies 1λ height of space, the height of the white space can be at most $3\lambda - 1\lambda = 2\lambda$. Therefore, the white space can be at most $2\lambda \times 2\lambda$ large and the area occupied by the routing, A, is at least $wh - 4\lambda^2$. Thus, the routing length $l = A/\lambda \geq \frac{wh}{\lambda} - 4\lambda$. □

While very small compared to the length bounds, the $4\lambda^2$ white space still means that we cannot always fully turn the cell area into routing length. Therefore, we need to assign a little bit more area to a net than its lower length bound requires to compensate for this loss. However, since $4\lambda^2$ is an upper bound of the white space, using it to estimate the area compensation for every cell is too pessimistic. In many cells we are not so unlucky as to hit this upper bound. Therefore, we introduce a parameter δ in constraint (4.12). It enlarges the area requirement in the cell sizing problem so that we are able to achieve the required lower length bound even with some waste of the area. Parameter δ should be large enough to provide the area compensation but not too large for two reasons: (1) large δ makes constraint (4.12) very strict and then the cell sizing problem becomes difficult to solve; (2) $4\lambda^2$ is very small compared to the total length, so we only need a very small δ to compensate for the area loss. In our experiments, we first set δ to a small



(a) Skinny corner cell (b) Corner cell with $\lambda \times \lambda$ size

Figure 4.14: A skinny corner cell does not allow wire extension.

value and increased it when we found the cell sizing result did not provide enough room for detail routing. We found that $\delta = 0.05$ would be enough for all our data.

Unfortunately, Theorem 4 is not necessarily true for corner cells because we may have a very skinny cell with height less than 2λ but very large width (see Figure 4.14 (a)). In this case, even though the area of the cell is large, we cannot utilize it for length extension. To avoid such skinny corner cells, we force the cell to be $\lambda \times \lambda$ large. That is, if a cell i is a corner cell, we use the following constraints instead of (4.3) and (4.4):

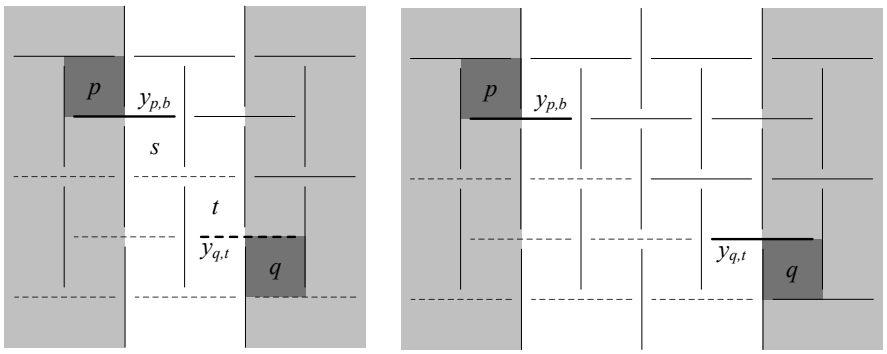
$$x_{i,r} - x_{i,l} = \lambda \tag{4.15}$$

$$y_{i,t} - y_{i,b} = \lambda \tag{4.16}$$

By doing this, we are able to guarantee that all the extra area assigned to a net is assigned to straight cells and can be effectively used for wire extension. Of course, we will lose some flexibility because corner cells can no longer be used for wire extension. However, since corner cells take up only a very small portion of the occupied cells, this has a negligible influence on the capability of our router.

4.3.6 Some Technical Details

Before we end this section, we would like to discuss some technical details about cell sizing. In Section 4.3.4, we mentioned that one of the reasons that the sizing problem is infeasible is that the BSG size is too small. This is because the BSG structure and the basic constraints impose some implicit constraints on the BSG walls. If pins or components are embedded too close to each other, such implicit constraints might lead to conflicts. Figure 4.15



(a) (b)

Figure 4.15: If two components are placed too close to each other as in (a), then there might be a conflict between the two pins p and q. We can resolve the conflict by adding columns of BSG cells in between as in (b).

(a) gives such an example. Suppose pin p is located at $(2.2, 5.5)$ and q is located at $(4.7, 5.3)$ and $\lambda = 1$. Then by the pin location constraint, we have $y_{p,b} = 5.5 - 1/2 = 5$ and $y_{q,t} = 5.3 + 1/2 = 5.8$. On the other hand, we can obtain the following inequalities from the basic constraints:

$$y_{p,b} = y_{s,t} \geq y_{s,b} = y_{t,t} \geq y_{t,b} = y_{q,t}$$

Put them together and we have $5 = y_{p,b} \geq y_{q,t} = 5.8$, which is impossible. The reason for this is that the location constraint on the wall $y_{p,b}$ actually implies a set of constraints on the dashed walls below it due to the structure of BSG: all the dashed walls must be located below $y_{p,b}$. If some other pin location constraint requires a dashed wall to be located above $y_{p,b}$, then we have a conflict. To resolve such a conflict, we can insert extra columns of cells between the two components, as shown in Figure 4.15 (b). In this way, wall $y_{q,t}$ is no longer implicitly constrained by the location of $y_{p,b}$ (it is no longer a dashed wall in the figure) and the conflict is resolved.

A similar situation could also happen inside a component. See Figure 4.16 for an example. If the pin location constraint requires $x_{p,r} = 5$, then implicitly it also requires $x_{q,l} \geq 5$. However, the pin location constraint of q may set $x_{q,l}$ to some value smaller than 5, causing a conflict. Again, we can resolve this by inserting rows of cells between them, but this increases the scale of the cell sizing problem. A better solution is to simply ignore the basic constraints for cells inside a component. We can ignore them because

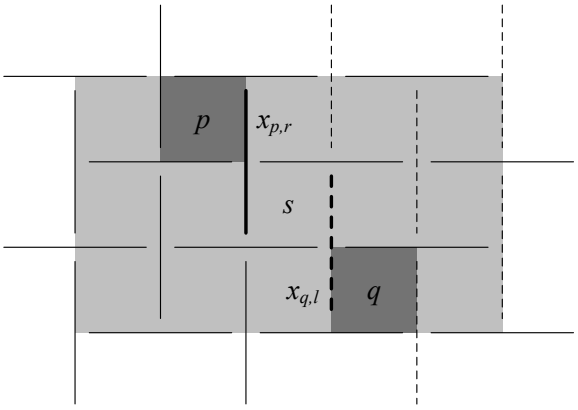
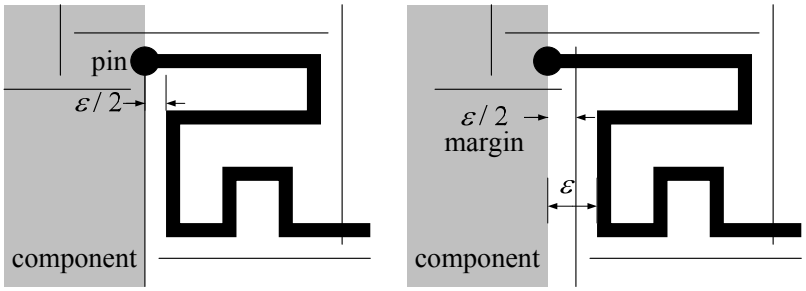


Figure 4.16: Conflict may also occur inside a component.



(a) Without margin (b) With margin

Figure 4.17: Separation rule between the wire and the component is violated in (a). We resolve this by inserting a $\varepsilon/2$ margin around the component as shown in (b).

no cell inside a component is used for routing, so ignoring basic constraints on them will not cause illegal routing. In our implementation, we ignore the basic constraints for all the cells inside a component.

Another thing we want to discuss is the separation between wires and components. The $\varepsilon/2$ separation between the routing and the BSG walls guarantees that the wire segments in different cells satisfy the separation rule. However, the separation rule between the wires and the components is not guaranteed (see Figure 4.17 (a)) because they are separated by only $\varepsilon/2$. To resolve this issue, we allow a $\varepsilon/2$ margin around every component during cell sizing. This $\varepsilon/2$ margin plus the $\varepsilon/2$ wire-to-wall separation give exactly ε wire-to-component separation (see Figure 4.17 (b)). Constraints

(4.5), (4.6), (4.7) and (4.8) should then be rewritten as

$$x_{i,l} = x_C - \varepsilon/2 \tag{4.17}$$

$$x_{j,r} = x_C + w_C + \varepsilon/2 \tag{4.18}$$

$$y_{p,b} = y_C - \varepsilon/2 \tag{4.19}$$

$$y_{q,t} = y_C + h_C + \varepsilon/2 \tag{4.20}$$

Notice that the sizes and locations of the cells are all real numbers. So the completed detail routing does not conform to any routing grid. That is why we call our router gridless. Nevertheless, the design rules are always satisfied.

4.4 Extensions

The routing scheme described so far has the basic features of a length-constrained router. We can further extend it to handle more complex problems.

4.4.1 Length-Matching Routing

In some designs, the length bounds (l_i, u_i) are not given. Instead, the designer expects the lengths of all the nets in a bus to be the same and minimize this length. This is usually called length-matching routing because the lengths of the nets must match each other. To handle this problem, we introduce a new variable l to replace the upper and lower length bound in all the length constraints (4.11) and (4.12):

$$\sum_i ((x_{i,r} - x_{i,l}) + (y_{i,t} - y_{i,b}) - \varepsilon) - s \leq l \tag{4.21}$$

$$\frac{\sum_i ((x_{i,r} - x_{i,l})(y_{i,t} - y_{i,b}))}{\varepsilon} + s \geq l(1 + \delta) \tag{4.22}$$

Since we use this single variable l to represent the length for all nets, their wire lengths are automatically matched.

In order to minimize the wire length, we need to include l into the objective. On the other hand, we still need to make sure $s = 0$ in the result to guarantee

the lengths of the nets are matched. Therefore, we set the objective of our cell sizing problem to be

$$\text{minimize} \quad M \cdot s + l$$

in which M is a very large constant ($M = 10^{10}$ in our experiment). We also include a new constraint into our problem:

$$s \geq 0 \tag{4.23}$$

In this way, s has the first priority to be minimized. However, once it reaches 0, it cannot be further minimized because of constraint (4.23). Then l will be minimized while s is kept at 0. Notice that l is only a linear term in all the constraints. Therefore, we can still use our iterative LP approach to solve the cell sizing problem.

4.4.2 More Complex Routing Topology

Figure 4.7 only shows an example of two components. However, our routing scheme is capable of dealing with three or more components. It is also possible to consider obstacles in our scheme. The only part we need to change is the BSG embedding step. We need a topological router that is able to generate planar routing following our guidelines. A negotiated-congestion based router [45], which is shown to be very powerful for planar routing in [15], can be used to determine the BSG embedding of the topology. The embedding can also be performed manually. It is usually an easy task to embed the topology of less than 100 nets on a BSG of size less than 200×200.

4.4.3 Different Design Rules

It is also possible to consider nets with different wire widths or separation rules. For example, if two adjacent nets have ε_1 and ε_2 as their separation rules, then in the detail routing, the separation between each wire and the BSG walls should be $\max(\varepsilon_1, \varepsilon_2)/2$. This can separate the two wires by $\max(\varepsilon_1, \varepsilon_2)$ (see Figure 4.18). Since the topology is given as input, we know which two wires can be adjacent to each other, so $\max(\varepsilon_1, \varepsilon_2)$ can be

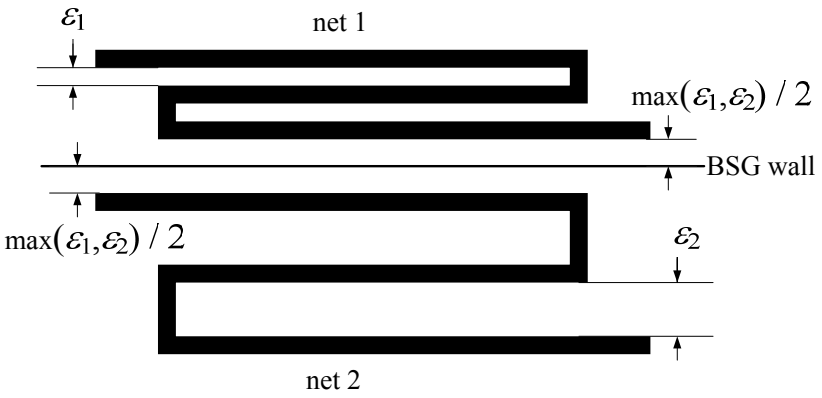


Figure 4.18: Separating the wires from the BSG walls by $\max(\varepsilon_1, \varepsilon_2)/2$ guarantees the satisfaction of wire separation rule.

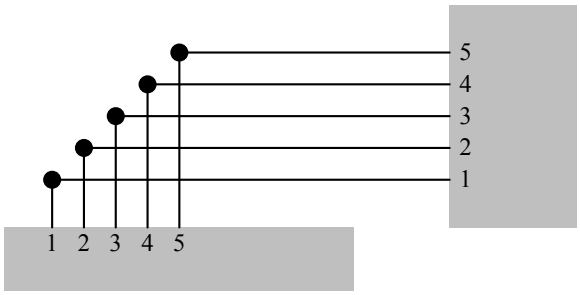


Figure 4.19: A case we observed from industrial data. Vias (black dots) are inserted to resolve the reversed ordering of the pins.

computed beforehand.

4.4.4 Multi-Layer Routing

Although planar routing is preferred in PCB routing, we observe from industrial designs that there are still situations when multiple layers are necessary to route a bus. Figure 4.19 gives such an example. In industrial data, we observed that the pins of a bus form reversed ordering in two components. In the manual solution, vias are inserted and two layers are used to route this bus.

Vias on PCBs can be classified into three categories (see Figure 4.20):

- A *through via* goes through all the layers.
- A *blind via* connects a certain layer in the middle to either the surface or the back of the PCB. It can be further classified as a *surface blind*

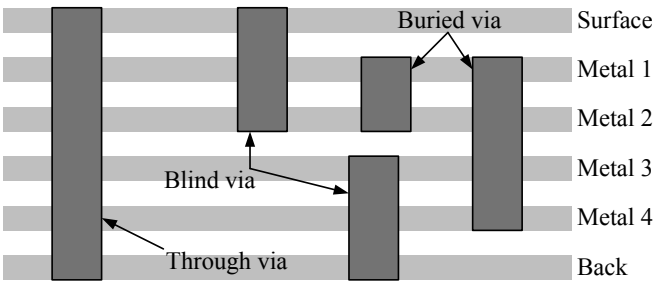


Figure 4.20: Three types of vias: through via, blind via, and buried via.

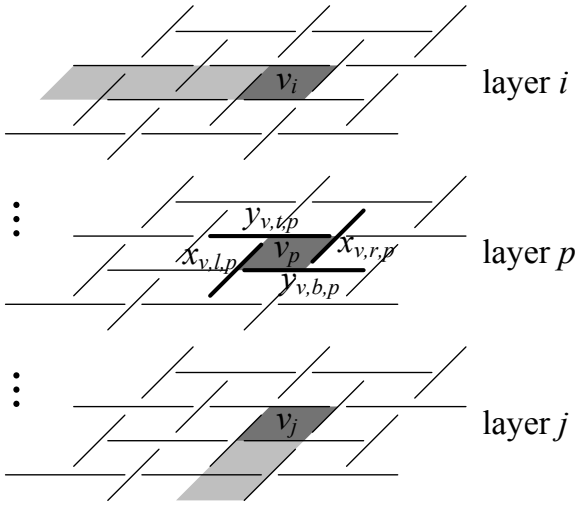


Figure 4.21: The embedding of a net is marked by the gray cells. It changes layers from cell v_i to cell v_j. The darker cells v_i, v_p, v_j indicate via cells.

via or *back blind via* depending on which side it connects to.

- A *buried via* connects two layers in the middle of the PCB.

Our router can be extended to handle multi-layer length-constrained routing with all three types of vias. However, we need to make some modifications to our router.

First of all, we need to use multiple BSGs. To represent the routing resource of a k-layers design, we use k identical copies of the BSG, each representing one layer. These BSGs are stacked vertically just like the routing layers (see Figure 4.21). All the k copies contain the same number of BSG cells and have the same structure. Now a cell v in the original BSG has k copies $v_1, v_2, \ldots, v_k$ that are vertically stacked. Since these k cells belong to different layers, they can be sized independently unless they are occupied by a via, which we will discuss soon.

The second step is to embed the multi-layer topology into the multi-layer BSGs. We must do this very carefully because the embedding must utilize all the layers evenly. Uneven usage of the BSG cells on different layers may lead to under-usage of routing space on some layers and eventually lead to length violations. Because of this even-usage requirement, multi-layer embedding is done manually in our experiment.

In the embedding, if a net switches layers from cell v_i to v_j $(i < j)$, then there is a via from layer i to j. Depending on the via technology (through via, blind via or buried via) we use, the cells occupied by this via are different:

- If it is a through via, then cells on all the layers, which are cells $v_p(1 \leq p \leq k)$, are occupied.
- If it is a surface blind via, then all the cells stacked on top of v_j, which are cells $v_p(1 \leq p \leq j)$, are occupied.
- If it is a back blind via, then all the cells stacked below v_i, which are cells $v_p(i \leq p \leq k)$, are occupied.
- If it is a buried via, then all the cells stacked between v_i and v_j, which are cells $v_p(i \leq p \leq j)$, are occupied.

We call the cells occupied by the via *via cells*.

We need to introduce new constraints to size such via cells. Since we now have multiple BSGs, we make a slight change in our variable notations. For a cell v_p on layer p, we name the variables representing its left, right, bottom and top walls as $x_{v,l,p}$, $x_{v,r,p}$, $y_{v,b,p}$ and $y_{v,t,p}$ respectively (see Figure 4.21).

Suppose a net changes layers from cell v_i to cell v_j $(i < j)$. For the simplicity of later detail routing, we require the size of a via cell to be just enough to accommodate the via (and the separation margin) so that no detour is possible inside a via cell (recall how we treated corner cells). Therefore, we introduce the following constraints on v_i (w_v and h_v are the width and height of the via, and ε is the separation rule):

$$x_{v,r,i} - x_{v,l,i} = w_v + \varepsilon \tag{4.24}$$

$$y_{v,t,i} - y_{v,b,i} = h_v + \varepsilon \tag{4.25}$$

In addition, we need to make sure that the positions of all the via cells are identical so that the via is aligned across the layers. Therefore, we also have

the following constraints:

$$x_{v,r,p} = x_{v,r,i} \tag{4.26}$$

$$x_{v,l,p} = x_{v,l,i} \tag{4.27}$$

$$y_{v,t,p} = x_{v,t,i} \tag{4.28}$$

$$y_{v,b,p} = x_{v,b,i} \tag{4.29}$$

for all p satisfying

$$\begin{cases} 1 \le p \le k & \text{if it is a through via} \\ 1 \le p \le j & \text{if it is a surface blind via} \\ i \le p \le k & \text{if it is a back blind via} \\ i \le p \le j & \text{if it is a buried via} \end{cases}$$

4.5 Experimental Results

Our routing scheme is implemented and compared with the router in [15]. We are not able to compare our router with those in [46] and [47] because those two routers can only be applied to cases when the two components are facing each other, and such cases are very rare in the industrial data we obtained. Our router is implemented in C++ and the linear programming (LP) problems are solved by the open source linear solver *lp_solve* [50]. As for embedding the topology onto the BSG, we use a simple heuristic since our data have only two components. We first route the pins of both components to a channel between them and then use river routing to route inside the channel. Other heuristics such as maze routing or negotiated-congestion router [45] can be employed for more complicated cases, e.g., when more than two components are involved. As mentioned before, using different heuristics to generate the embedding has an insignificant influence on the final routing quality as long as our guidelines are followed. Experiments are performed on a computer with two 2.8 GHz Intel Xeon processors and 4 GB memory. The platform is Redhat Enterprise Linux 4.

We use seven data sets to test our router (see Table 4.1). The *monotonic* data set allows monotonic (left to right) routing topology so the router in [15] can be applied. The topologies in the *general* data set are general and no

Table 4.1: Experimental results of our BSG-route.

				our BSG-route					router in [15]	
data	#net	length slack		BSG size	H-problem	V-problem	#it.	runtime	grid size	runtime
		min.	avg.	$w \times h$	#var./#con.	#var./#con.		(s)	$w \times h$	(s)
monotonic_1	84	1.4%	46.5%	87×175	7829 / 14543	7829 / 14629	2	86	1181×1237	137
monotonic_2	44	0.3%	11.2%	125×95	6093 / 10769	6093 / 10738	2	73	2252×2383	NEM[a]
monotonic_3	83	0.5%	0.6%	67×173	6000 / 10898	6000 / 11002	2	56	1012×899	13859
monotonic_4	45	0.1%	10.0%	119×112	6826 / 12057	6886 / 12050	3	88	2252×680	99491
general_1	36	0.8%	1.7%	105×86	4648 / 8730	4701 / 8664	3	64	N/A	
general_2	28	0.02%	0.02%	62×91	2973 / 5464	2927 / 5456	3	21	N/A	
general_3	36	0.4%	0.9%	109×86	4822 / 9078	4877 / 9008	3	260	N/A	
extend	36	N/A	N/A	(2×)90×90	8410 / 15382	8410 / 15382	3	244	N/A	

[a]NEM: Not enough memory. The required memory to run this data exceeds the memory of the computer (4 GB).

previous routers can be applied. *monotonic_1*, *monotonic_2* and *general_1* are original industrial data and *monotonic_3*, *monotonic_4*, *general_2* and *general_3* are derived from industrial data. The second column of Table 4.1 shows the number of nets for each data set. In order to show how strict the length constraints are for a data set, we calculate the length slack of every net. The slack of a net i is calculated by the following equation (recall that u_i is the upper length bound and l_i is the lower length bound):

$$slack_i = \frac{u_i - l_i}{u_i} \times 100\%$$

The smaller the slack, the more strict the length constraint. The minimum and average length slack of the nets in each data set are shown in the third and fourth columns of Table 4.1.

The experimental results of both routers are reported in Table 4.1. The "BSG size" column gives the size of the embedded BSG. The next two columns give the size (the number of variables and the number of constraints) of the LP problems we formulate for cell sizing. "#.it" gives the number of LP problems we solve before s converges to 0. The runtime of our router includes the runtime of the LP solver. In fact, the majority of the runtime is spent solving the LP problems.

Several observations can be made from the experiments:

1. The BSG size we use is about 10 times smaller than the routing grid size of [15] in both width and height. Moreover, our BSG size is not sensitive to the routing grid size of the problem. For example, the routing grid size of *monotonic_2* is much larger than that of *monotonic_3*. However, our BSG size remains similar.

2. Our router can handle industrial designs that cannot be handled by previous routers.

3. For the data that can be handled by [15], our router runs much faster. The runtime difference can be as huge as 1000x (88 s vs. 99491 s for *monotonic_4*).

4. Only two or three LP problems need to be solved before it converges to a feasible cell sizing solution. This means our approach to solve the cell sizing problem is efficient.

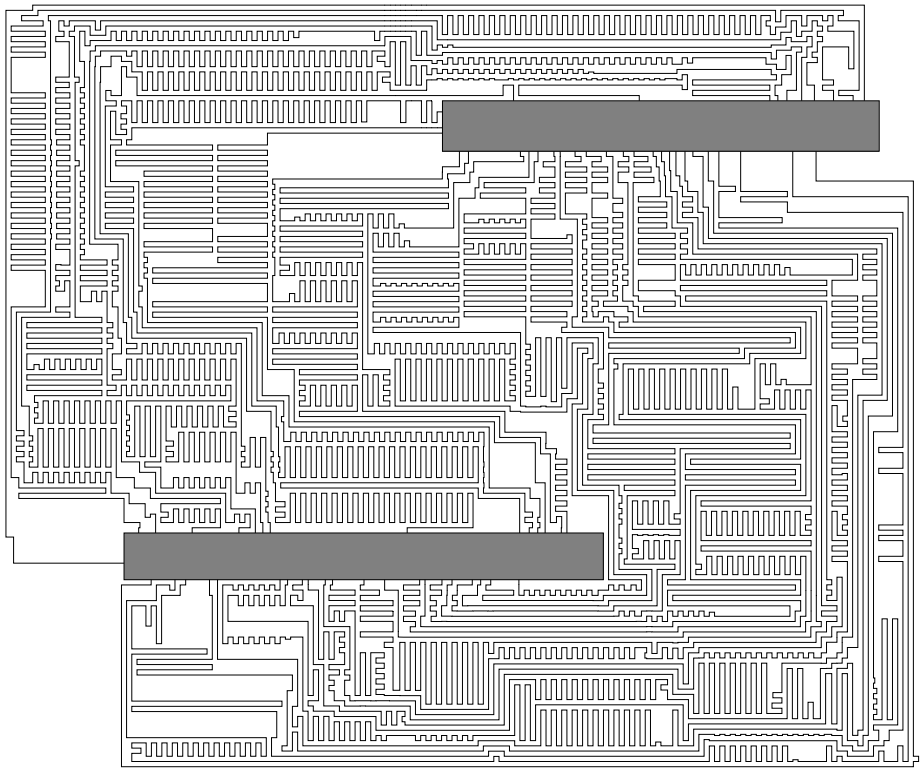
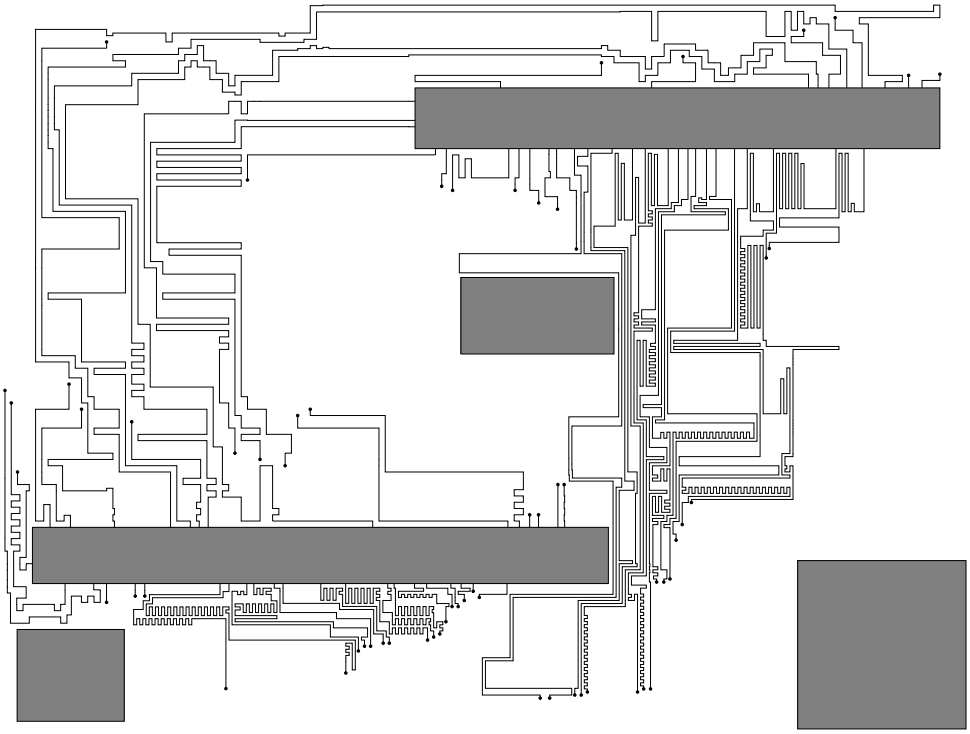


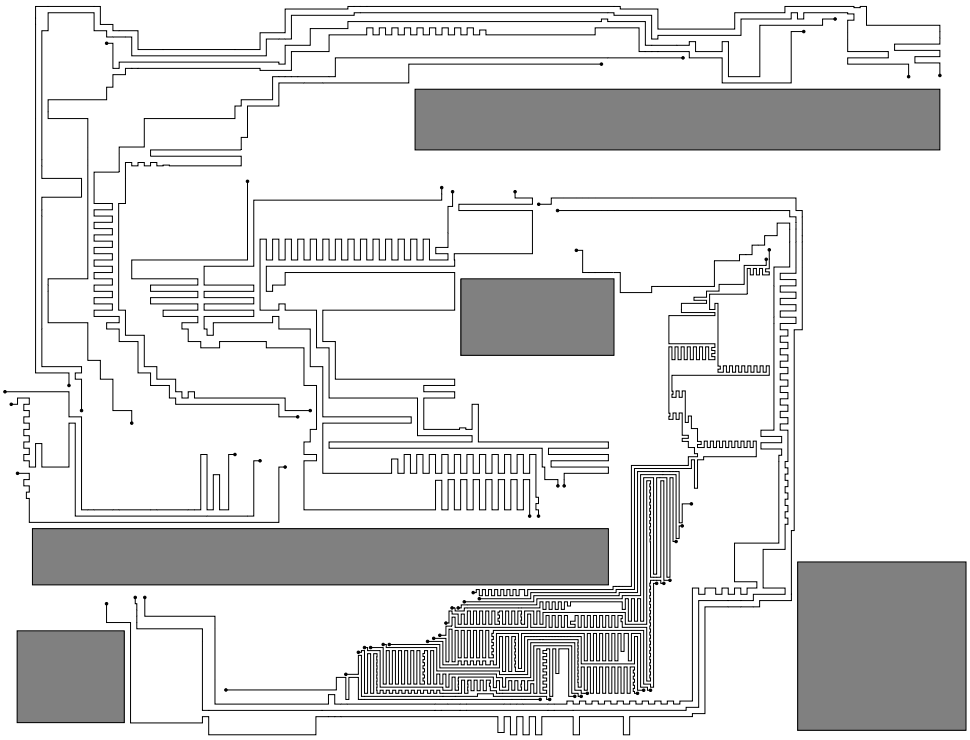
Figure 4.22: Our routing result of *general_3*.

5. The routing result of *general_3* is shown in Figure 4.22. We intentionally made the length bounds of the nets very long in the data in order to show that our router can effectively use the routing area to satisfy the length constraints.

We also extend our router to handle multi-layer routing, length-matching routing, obstacles and different wire separation rules. To test these extended features, we construct a data set *extend* by making some modifications to *general_3*. The data set contains two routing layers and some routing obstacles. All the nets are required to be routed with the same length and the total routing length is expected to be minimized (length-matching routing). Some of the nets have a different wire separation rule from the others. The topology embedding is generated manually. We apply our router on this data and its performance is reported in the last row of Table 4.1. The (2×) in the "BSG size" column indicates that we use two BSGs for the two routing layers. The routing result is shown in Figure 4.23. We can make several observations from the result:



Layer 1



Layer 2

Figure 4.23: Routing result of *extend*.

1. The routing is much sparser than that of Figure 4.22, indicating that the router is trying to minimize the total length while keeping all the wires the same length.

2. Part of the routing (see the bottom left part of the layout) has smaller wire separation than the rest, indicating that the router is able to handle different wire separations.

3. The routing avoids the obstacles.

4. The routing utilizes two layers and the via locations at the two layers align perfectly. This means our router produced a legal two-layer routing.

These observations verify that all the extended features described in Section 4.4 function well in our router.

4.6 Conclusion

In this chapter, we introduced a length-constrained routing scheme that handles general planar topology. It is the first time that the length-constrained routing problem is solved without any restrictions on the routing topology. With the help of the BSG structure, we are able to convert the length-constrained problem into a mathematical programming problem and solve the problem by solving a sequence of linear programming problems. Due to its gridless feature, our router is insensitive to the routing grid size of the input, making it very fast for large PCB designs. We also discussed several extensions of our router including length-matching routing, handling obstacles, using different separation rules for different nets and multi-layer routing. The effectiveness of our router and its extended features is verified by experiments.

CHAPTER 5

UNTANGLING TWISTED BUS

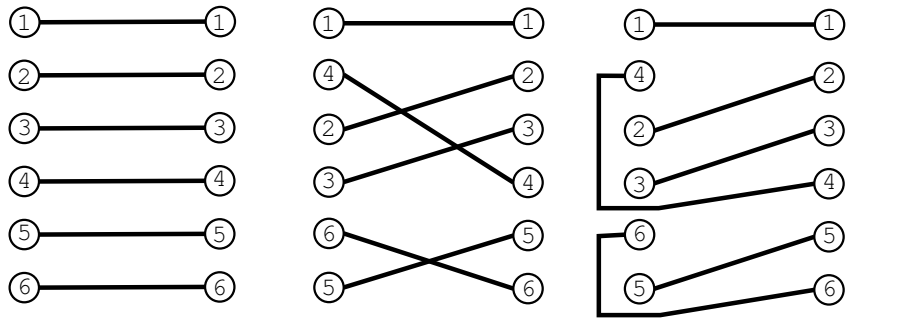
5.1 Introduction

In all the previous works on length-constrained routing [15, 46, 47], the pins on the two sides are assumed to have the same ordering (as in Figure 5.1 (a)). However, such perfectly matched pin ordering might not be available in practical designs (see the next section for detailed explanation). Such mismatched pin ordering causes twisted nets that cannot be resolved by any router in [15,46,47]. In order to obtain a valid routing, we must first untangle such twisted nets before we apply length-constrained routing.

In this chapter, we introduce a preprocessing step to untangle the twisted nets before length-constrained routing. Our contributions lie in the following aspects:

1. This untangling step enables previous length-constrained routers to solve a broader range of problems.

2. We introduce a routing style, *single-detour routing*, to simplify the un-



(a) Matched ordering (b) Twisted nets (c) Untangle the twisted nets

Figure 5.1: Must untangle the twisted nets before length-constrained routing.

tangling problem. This routing style is observed from practical designs.

3. We present a necessary and sufficient condition for the feasibility of single-detour routing problems, which is of theoretical importance.

4. We present a dynamic programming based algorithm to solve the single-detour untangling problem with the consideration of wire capacity between adjacent pins. The optimality is guaranteed by our algorithm.

The rest of this chapter is organized as follows: We introduce the motivation of this work as well as our problem formulation in Section 5.2. In Section 5.3, we introduce a very practical single-detour style that greatly simplifies our problem. We then study this routing style and present a necessary and sufficient condition for the existence of its feasible solutions in the same section. In Section 5.4, we present a dynamic programming based algorithm that gives the optimal solution to the single-detour untangling problem. Experimental results are presented in Section 5.5. We conclude the chapter in Section 5.6.

5.2 Motivation

Previous works on length-constrained routing [15, 46, 47] require that the orderings of the pins on the two sides are matched. However, in some practical designs, this might not be possible. For example (see Figure 5.2), in some high-performance PCB designs, nets in a bus are first routed from one pin array to a column of damping impedance near the pin array and then routed to the pins inside another pin array. The routing from inside the pin arrays to the package boundaries, which is usually called *escape routing*, is expected to produce a matched net ordering on both boundaries. Although researches on escape routing are trying hard to make this possible [16, 51], it is still sometimes impossible to guarantee a matched ordering, especially for complex designs. Failure to provide matching ordering leads to twisted nets, and usually this means that the bus must be split and routed on different layers. Another example is to route from a dual in-line package (DIP) to a pin array. A DIP is a package whose pins form two parallel lines (please refer to [52] for an introduction on DIP). Its footage on the PCB resembles the two lines of pins of the column of damping impedances in Figure 5.2. Sometimes the pin

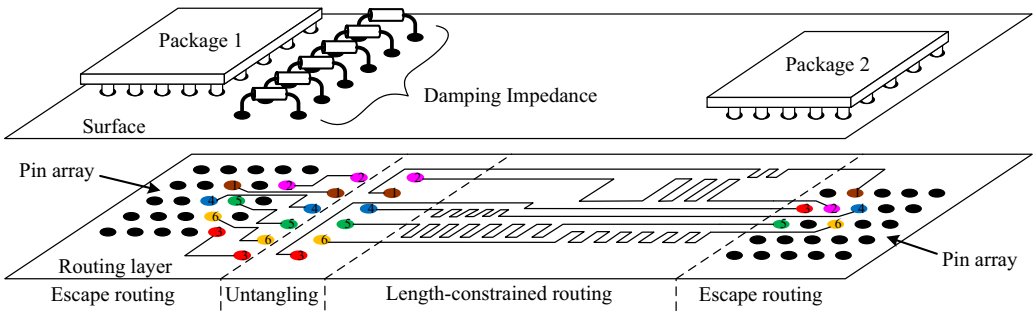


Figure 5.2: Mismatched pin ordering can be resolved by untangling the twisted nets.

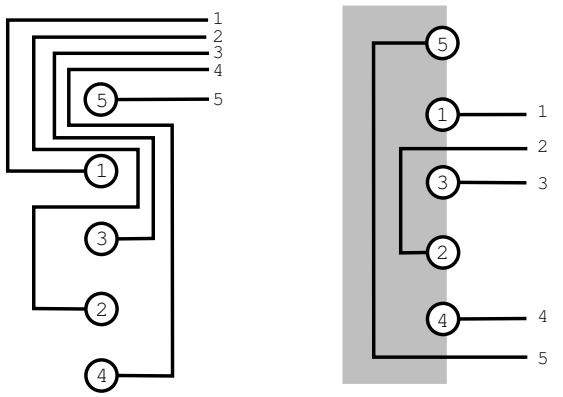
ordering of a DIP may not match with the escape routing of the pin array it connects to and the bus becomes twisted. However, we can use the space under the damping impedances or the DIP to untangle the twisted nets and reduce the number of layers the bus uses.

In this chapter, we introduce an untangling step into the design flow of PCB bus routing as a preprocess so that previous bus routing algorithms can be extended to solve a broader range of problems. Of course, the wire length used to untangle the nets must be subtracted from the length budget in the later length-constrained routing phase. This can be done by small modifications to the length-constrained router.

Without loss of generality, we assume that the escaped nets on the right are labeled $1, 2, \ldots, n$, from top to bottom and the damping impedance in the left is a permutation of $\{1, 2, \ldots, n\}$. We can formulate the problem as follows:

Problem 2. Given a column of n pins whose IDs form a permutation of $\{1, 2, \ldots, n\}$, from top to bottom, the *untangling problem* is to route from the pins to the right such that the routing is planar and the wires to the right follow an increasing order.

This problem has many solutions. One solution can be constructed in a systematic way: we first route the top pin to the right with no detour. Then we route the rest of the pins from top to bottom. When we route a pin, we start from the pin and always let the wire go up. When the wire encounters a pin with ID smaller than the pin we are routing, we let the wire pass this pin on the right. Otherwise, we let the wire pass it on the left. We continue doing this until the wire passes the top pin and then we draw the wire to the



(a) Upward routing (b) Single-detour routing

Figure 5.3: Upward routing vs. single-detour routing.

right. An example of such routing can be found in Figure 5.3 (a).

This routing style guarantees that the routing is planar and the order of the wires to the right is increasing. However, it is not practical due to the following issues:

1. It does not consider the capacity between adjacent pins. Usually, a solution generated in this way has lots of wire congestion between pins.

2. It generates only one solution and thus lacks flexibility.

5.3 Single-Detour Routing

From the above discussion, we can see that "snaking" among the pins usually causes wire congestion, e.g., there are three wires passing between pin 5 and pin 1 in Figure 5.3 (a). To avoid this, we restrain the detouring area to the left of the pin column (the dark region in Figure 5.3 (b)). The right-hand side of the pin column is used only for straight connections. This actually means that each pin is allowed to detour *only once*. We call this routing style *single-detour routing*. Using single-detour routing limits the possibilities of the routing solutions. However, it brings us lots of benefits:

1. The routing pattern is much simpler.

2. It leads to less wire congestion. Notice that the snaking makes a wire pass between pins multiple times while single-detour routing allows at

most one passing per wire.

3. The right-hand side of the pin column can be used completely for length-constrained routing. If multiple detours are allowed, some part of the wires will pose obstacles to later length-constrained routing.

In fact, we observe that the snaking in Figure 5.3 (a) rarely happens in practical designs. Most board routing adopts the routing style in Figure 5.3 (b). Therefore, we focus on this single-detour routing style and put it into our problem formulation:

Problem 3. The *single-detour untangling (SDU) problem* is an untangling problem that allows *only one detour* for each wire.

Here we present a simple *necessary and sufficient* condition to judge whether a given single-detour untangling problem has feasible solutions or not:

Definition 1. A sequence of integer numbers is *reduced* by renaming each number by its order of magnitude in the sequence. That is, the smallest number is renamed as 1, the second smallest as 2, and so on.

For example, $(5, 3, 8, 1)$ is reduced to $(3, 2, 4, 1)$. Notice that some sequences such as $(3, 4, 1, 2)$ are reduced to themselves.

Definition 2. The *pin sequence* of a problem is the sequence of pin IDs obtained by scanning the pin column from top to bottom. A *pattern* is a permutation of $\{1, 2, \ldots, m\}$. The pin sequence of a problem *contains a pattern* if and only if there exists a subsequence of the pin sequence that can be reduced to that pattern.

For example, pin sequence $(6, 4, 3, 5, 1, 2)$ contains the pattern $(2, 3, 1)$ because its subsequence $(3, 5, 2)$ can be reduced to $(2, 3, 1)$.

Theorem 5. *A single-detour untangling problem has at least one feasible solution* if and only if *its pin sequence does not contain pattern* $(3, 4, 1, 2)$, $(2, 4, 1, 3)$ *or* $(3, 1, 4, 2)$.

Proof. See appendix. □

The three patterns $(3, 4, 1, 2)$, $(2, 4, 1, 3)$ and $(3, 1, 4, 2)$ are called *forbidden patterns* for SDU problems.

This theorem is of theoretical importance because it defines whether an SDU problem is solvable or not. If an SDU problem is not solvable, several changes can be made to the design to help resolve this issue:

1. Recall that the order of the pins in the pin sequence is determined by the escape routing of the pin arrays (see Figure 5.2). So we can modify the escape routing to make the pin sequence preferable for single-detour routing. Theorem 5 gives a criterion for what kind of pin sequences are preferable - those which do not contain the forbidden patterns.

2. We can break one unsolvable SDU problem into several smaller solvable SDU problems. This can be done by decomposing the pin sequence into several subsequences which do not contain the forbidden patterns. Those smaller problems can then be routed on different layers.

3. We can increase the routing flexibility by allowing the wire to detour multiple times. Whether optimal algorithms exist for the such multi-detour routing remains an open problem.

5.4 Dynamic Programming Solution

In this section, we will present a dynamic programming based algorithm. We will first show how we define the subproblems of an SDU problem and then show how a subproblem can be recursively built from smaller subproblems. Then we will explain how to turn the recursive construction into dynamic programming. Finally, we will extend our algorithm to handle capacity constraints and also introduce some other extensions of it.

5.4.1 Subproblem Definition

Definition 3. A *subproblem* $P(i, j)$ $(i \leq j)$ is a subsequence of the pin sequence P such that the pin IDs in the subsequence are in range $[i, j]$. If a subproblem is a consecutive subsequence of the original pin sequence, we call it a *valid* subproblem.

For example, for pin sequence $(4, 1, 3, 2, 6, 5)$, we have a valid subproblem $P(1, 4) = (4, 1, 3, 2)$. Notice that not every $P(i, j)$ is valid. For example,

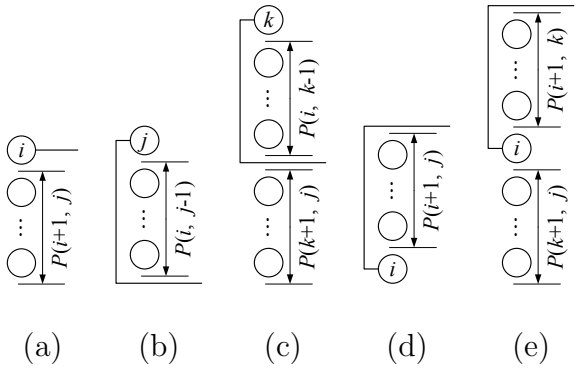


Figure 5.4: The five cases of dividing a subproblem $P(i, j)$.

$P(2, 5)$ is not valid because the subsequence $(4, 3, 2, 5)$ is not consecutive in the original pin sequence.

We are interested only in valid subproblems because invalid subproblems do not have independent solutions. In the example above, pin 1 and pin 6 are located inside subproblem $P(2, 5)$. We cannot route the subproblem without considering how to route those two pins.

For an SDU problem with n pins, we have $1 \leq i \leq j \leq n$. This means there exist at most $O(n^2)$ subproblems.

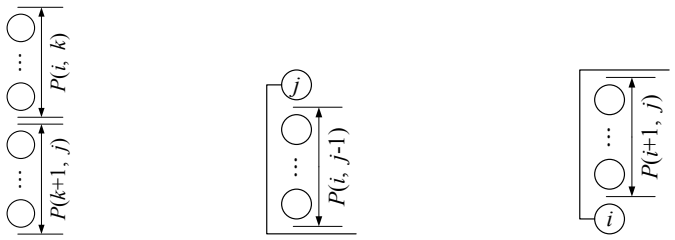
5.4.2 Recursive Nature of the Subproblem

With the subproblem defined, we now show that its solution can be recursively constructed from the solutions of smaller subproblems. We examine the topmost feature (a pin or a wire) of a feasible solution of a subproblem $P(i, j)$. There are totally five cases (see Figure 5.4):

1. The topmost feature is pin i. Then its routing must be straight to the right and the rest of the pins form a smaller subproblem $P(i + 1, j)$. Notice that $P(i + 1, j)$ must be valid if $P(i, j)$ is valid.

2. The topmost feature is pin j. Then its routing must detour over the rest of the pins. Those pins form a smaller subproblem $P(i, j - 1)$. Notice that $P(i, j - 1)$ must be valid if $P(i, j)$ is valid.

3. The topmost feature is a pin k such that $i < k < j$. Then the rest of the pins can be divided into two smaller subproblems $P(i, k - 1)$ and



(a) Combination (b) Top pin detour (c) Bottom pin detour

Figure 5.5: Three ways of decomposing a subproblem.

$P(k+1, j)$. Both subproblems are valid because otherwise there will be intersections on wire k.

4. The topmost feature is a wire from pin i which is located at the bottom of the subproblem. Then all the pins above i form a smaller subproblem $P(i+1, j)$. Notice that $P(i+1, j)$ must be valid if $P(i, j)$ is valid.

5. The topmost feature is a wire from pin i which is located in the middle of the subproblem. Then pins above i form a smaller subproblem $P(i+1, k)$ and pins below i form a smaller subproblem $P(k+1, j)$. Both subproblems are valid because otherwise there will be intersections on wire i.

These five cases can be further generalized into three cases (see Figure 5.5):

1. The subproblem is a combination of two smaller subproblems $P(i, k)$ and $P(k+1, j)$, in which $i \leq k < j$. This covers cases (1), (3) and (5) of Figure 5.4. Notice that for case (1), $k = i$. So the two subproblems are $P(i, i)$ and $P(i+1, j)$.

2. The top pin detours downward over all the other pins. Then the subproblem can be decomposed into a smaller subproblem $P(i, j-1)$ and a detouring route from pin j. This covers case (2) in Figure 5.4.

3. The bottom pin detours upward over all the other pins. Then the subproblem can be decomposed into a smaller subproblem $P(i+1, j)$ and a detouring route from pin i. This covers case (4) in Figure 5.4.

5.4.3 The Algorithm

The above description already gives a recursive algorithm: we could recursively build the solution of the subproblem until we reach the bottom case, a subproblem with only one pin. The solution to this base subproblem is simple: a wire to the right with no detour. Conversely to this top-down recursive fashion, we can also construct all subproblems in a bottom-up way using dynamic programming [53] (see Algorithm 2).

Algorithm 2 Dynamic Programming for SDU

```
1: for 1 ≤ i ≤ n do
2:    construct solution of P(i, i) by routing i to the right with no detour
3: end for
4: for size = 2 to n do
5:    for i = 1 to n − size + 1 do
6:       j = i + size − 1
7:       if solution of P(i, j) has been constructed then
8:          for k = j + 1 to min{j + size, n} do
9:             if solution of P(j + 1, k) has been constructed then
10:               construct solution of P(i, k) from P(i, j) and P(j + 1, k)
11:            end if
12:         end for
13:         for k = i − 1 downto max{i − size, 1} do
14:            if solution of P(k, i − 1) has been constructed then
15:               construct solution of P(k, j) from P(k, i − 1) and P(i, j)
16:            end if
17:         end for
18:         if pin j + 1 is located immediately above P(i, j) then
19:            construct solution of P(i, j + 1) from P(i, j)
20:         end if
21:         if pin i − 1 is located immediately below P(i, j) then
22:            construct solution of P(i − 1, j) from P(i, j)
23:         end if
24:      end if
25:   end for
26: end for
27: return solution of P(1, n)
```

In the algorithm, lines 8 to 17 cover case (1) in Figure 5.5. When a subproblem is a combination of two smaller subproblems, there are two cases: the size of the upper subproblem is larger than or equal to that of the lower subproblem, or the reverse. Lines 8 to 12 cover the first case and lines 13

to 17 cover the later case. Lines 18 to 20 cover case (2) in Figure 5.5 and lines 21 to 23 cover case (3) in Figure 5.5.

Clearly, this algorithm has $O(n^3)$ time complexity because the nested "for" loop has depth 3. We can see that this algorithm covers all possible cases. Therefore, if the problem has a single-detour solution, it will be found by this algorithm. The optimality is guaranteed.

Theorem 6. *Algorithm 2 guarantees to find a feasible solution to an SDU problem if one exists.*

Proof. From the discussion above. □

5.4.4 Capacity Consideration

For practical problems, we have to consider wire capacity as well. *Wire capacity* is the maximum allowable number of wires that could pass between two adjacent pins. Therefore, our problem becomes:

Problem 4. An *SDU problem with wire capacity C* is an SDU problem allowing at most C wires passing between two adjacent pins.

We need to modify Algorithm 2 in order to take wire capacity into consideration. When we combine two subproblems (case (1) of Figure 5.5), we need to check whether the capacity constraints are satisfied between the two subproblems, i.e., whether the number of wires that go below the upper subproblem plus the number of wires that go above the lower subproblem is less than or equal to the capacity limit (see Figure 5.6 (a)). We do not need to check inside the smaller subproblems because the capacity requirements are already satisfied when constructing the smaller subproblems. Similarly, for case (2)/(3), we only need to check if the number of wires that go below/above the subproblem is less than the capacity.

In order to facilitate such capacity check, we need to know the number of wires above the top pin as well as the number of wires below the bottom pin in a subproblem. However, there could be multiple solutions to a subproblem and the solutions may give different numbers of wires above or below the subproblem. Therefore, we have to enumerate all possible cases to guarantee the optimality. For each subproblem $P(i, j)$ we have C^2 subcases (recall that C is the capacity between two adjacent pins). A subcase $P(i, j, a, b)$ means

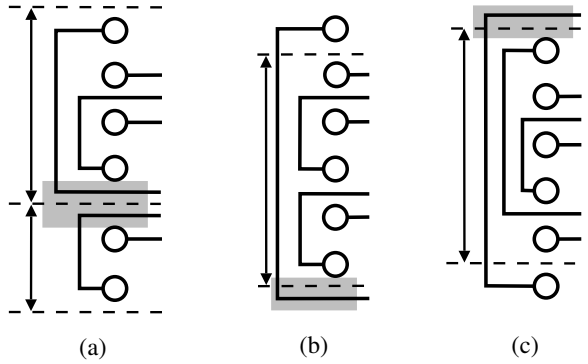


Figure 5.6: Capacity should be checked when constructing bigger subproblems from smaller subproblems. Dark area indicates the region in which the wire capacity should be checked.

that the number of wires above the subproblem is a and the number of wires below the subproblem is b. Now line 10 and line 19 in the previous algorithm should be modified to Algorithm 3 and Algorithm 4. The constructing procedures in line 15 and line 22 should be modified in a similar way.

Algorithm 3 construct solution of $P(i,k)$ from $P(i,j)$ and $P(j+1,k)$

```
1: for a = 0 to C do
2:   for b = 0 to C do
3:     for a' = 0 to C − b do
4:       for b' = 0 to C do
5:         if P(i, j, a, b) and P(j+1, k, a', b') have been constructed then
6:           construct solution of P(i, k, a, b') from P(i, j, a, b) and P(j +
             1, k, a', b')
7:         end if
8:       end for
9:     end for
10:  end for
11: end for
```

The time complexity of the algorithm is $O(n^3C^4)$, in which n is the total number of pins and C is the capacity between two adjacent pins. Usually, C is quite small in practice (≤ 3). So the complexity can be regarded as $O(n^3)$ for practical designs.

Some extensions can be made for our algorithm; all of them require only minor changes to our algorithm:

1. Sometimes we want to optimize some cost while keeping the capacity rule satisfied. For example, sometimes we would like to minimize the

Algorithm 4 construct solution of $P(i, j+1)$ from $P(i, j)$

```
1: for a = 0 to C do
2:   for b = 0 to C − 1 do
3:     for a' = 0 to C do
4:       construct solution of P(i, j + 1, a', b + 1) from P(i, j, a, b)
5:     end for
6:   end for
7: end for
```

number of vertical tracks used for untangling to save space to the left of the pin column, or sometimes we want to minimize the maximum congestion between adjacent pins. Our algorithm is capable of doing it. For each subcase, we not only memorize its solution, but also keep the cost of this solution. We update the cost and the solution whenever we find a solution that has better cost.

2. Although we assume that the wire capacity is uniform for all adjacent pins, it is straightforward to extend the algorithm to handle different capacities between different pins. The only change is that we have different numbers of subcases for each subproblem.

3. The dynamic programming approach gives only one solution. If multiple solutions are demanded to increase the flexibility, we can keep a list of solutions for each subcase and use all the possible solutions in the list to build larger subcases. The complexity increases by a constant factor as long as the length of the list is constrained to constant.

5.5 Experimental Results

We implement our dynamic programming algorithm in C++ and integrate it into the length-constrained router in [15]. The original length-constrained router in [15] gives a solution in which all wires have the same length. By small modifications to it, we can route the wires with specified length difference. We test the integrated router on various test cases with their size ranging from 10 nets to 100 nets. All experiments are performed on a workstation with a 2.4 GHz Intel Xeon CPU and 1 GB memory. The operating system is Red Hat Linux 8.0.

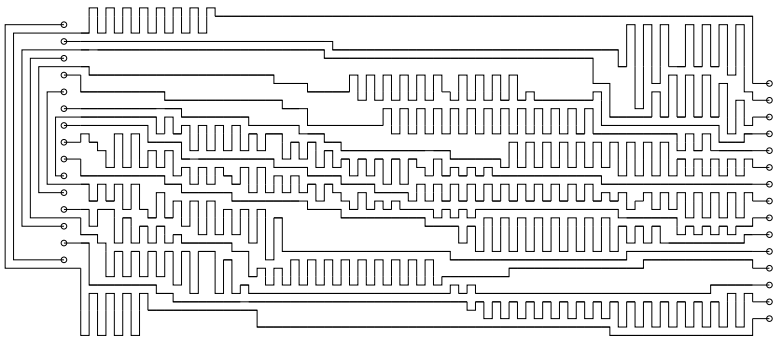


Figure 5.7: Our solution for a test case with 15 nets.

Our router gives valid routing solutions for all the test data. The runtime spent on untangling the nets takes less than 1 s in all cases while length-constrained routing takes minutes. The number of vertical tracks used for detouring is around $4 \sim 6$. It can be seen that our algorithm adds only very minor overhead to the router. The solution of one test case is shown in Figure 5.7.

5.6 Conclusion

We introduced a new step to untangle the twisted nets for bus routing. We also introduced the single-detour constraint to reduce the complexity of the untangling problem. We presented a necessary and sufficient condition for the feasibility of single-detour routing problems. We also presented a dynamic programming based algorithm to solve such single-detour untangling problems. The algorithm is guaranteed to produce an optimal single-detour routing scheme that untangles the nets. The time complexity of the algorithm is $O(n^3)$. Experimental results show that the algorithm takes less than 1% of the total runtime in a bus routing flow and untangles the twisted nets effectively.

CHAPTER 6

LAYER ASSIGNMENT

6.1 Introduction

In PCB routing, nets are usually grouped as buses and the nets from the same bus are usually expected to be routed together without mixing with nets from other buses. Due to the huge pin count and high density of the pin array, it usually requires multiple layers to escape the buses without any conflict. In fact, modern PCBs may contain more than 20 layers of routing [6]. How to assign the escape routing of buses to different layers becomes an important issue.

Recently, Kong et al. [16] proposed a layer assignment algorithm for this problem. The algorithm is optimal for single-layer design in the sense that it determines if a set of buses can be all escaped on one layer. If they cannot, the algorithm is able to select a maximum subset of the buses that can be escaped on one layer. For multi-layer design, this algorithm suggests a heuristic: repeatedly assign a maximum subset of the unassigned buses to a new layer. Such a heuristic may lead to suboptimal results in which the number of layers is not minimal.

In this chapter, we propose an optimal layer assignment algorithm for multi-layer design. We show that the layer assignment problem can be transformed into a bipartite matching problem, which can be solved in $O(n^{2.38})$ time [54]. By finding the maximum matching of a constructed bipartite graph, we can find the optimal bus layer assignment of which the number of layers is guaranteed to be minimal.

The rest of this chapter is organized as follows: Section 6.2 gives the necessary background on the layer assignment problem of bus escape routing; Section 6.3 presents our optimal algorithm for solving this problem; Section 6.4 presents the experimental results; and Section 6.5 gives the conclusion.

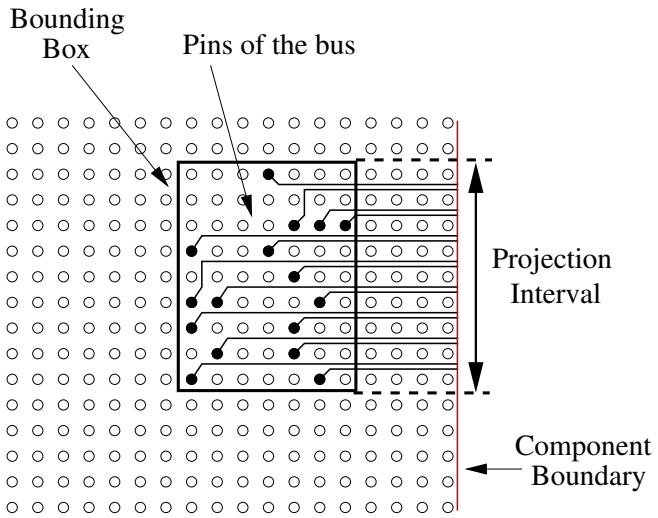


Figure 6.1: Illustration of the projection interval of a bus.

6.2 Background

In practical designs, pins on the PCB are usually grouped into buses and the escape routing of a bus is expected to be grouped together without mixing with routing from other buses. In some industrial manual designs, we observe that the pins of a bus are escaped straight to the boundary of the pin array with minimal detours. In this case, the routing region of a bus is the projection of the bounding box of its pins to the boundary of the pin array [16]. The projection forms an interval along the array boundary, which we call *projection interval* or *interval* for short. Figure 6.1 illustrates the concept of projection interval.

Due to the high density of the pin array, not all the buses can be assigned to one layer. We need to assign all the buses to the minimum number of layers without any conflict between the buses assigned to the same layer. Two buses are called *conflicting* buses and cannot be assigned to the same layer if one of the following two situations happens:

- The intervals of the two buses overlap in either side (see Figure 6.2). If two intervals overlap, then the escape routing of the two buses will have conflicts.

- The intervals of the two buses have different ordering in the two arrays (see Figure 6.3). If the interval of one bus is above the interval of the other bus on one side but the reverse happens on the other side, then

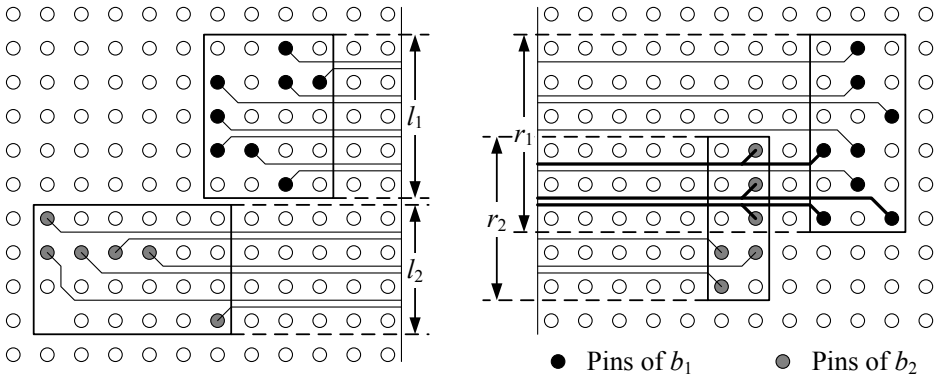


Figure 6.2: Bus b_1 has intervals l_1 and r_1 in the left and right array respectively. Bus b_2 has intervals l_2 and r_2 in the left and right array respectively. The escape routes of the two buses in the left do not have conflicts so l_1 and l_2 do not overlap. Contrarily, their escape routes have conflicts (the thick routing) in the right where r_1 and r_2 overlap.

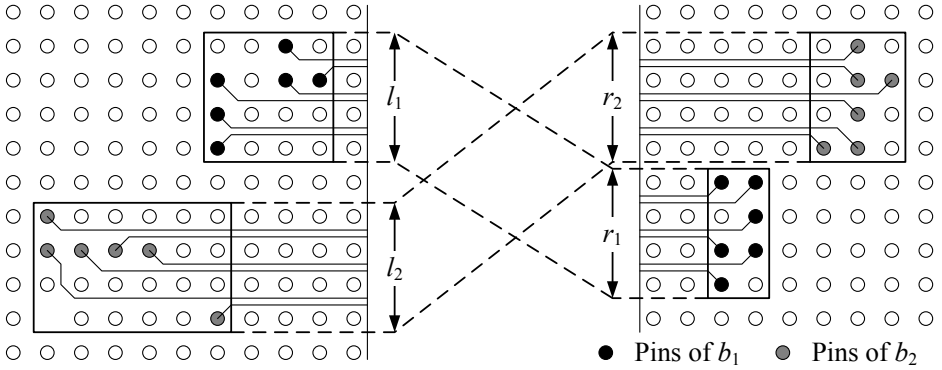


Figure 6.3: The intervals of bus b_1 and b_2 have different ordering on the two sides. This causes intersections between the area routing of the two buses.

the area routing of the two buses will have crossings, which are not desirable.

If two buses are not conflicting, they can be assigned to the same layer without causing any problem. We say such buses are *compatible* with each other.

Now we define the layer assignment problem for escape routing of buses as follows.

Definition 4. Given a set of buses $B = \{b_1, b_2, \ldots, b_n\}$ defined by their intervals $\{l_1, l_2, \ldots, l_n\}$ in the left array and $\{r_1, r_2, \ldots, r_n\}$ in the right array, find out a *valid* layer assignment using the minimum number of layers. A layer

assignment is *valid* if the buses assigned to the same layer are all compatible with each other for all layers.

In [16], Kong et al. proposed an algorithm that determines if a set of buses are all compatible. If not, their algorithm can find the maximum subset[1] of the buses that are compatible with each other. It is an optimal layer assignment algorithm for single-layer design. A heuristic for multi-layer design follows naturally from this algorithm: repeatedly find out the maximum subset of the unassigned buses and assign them to a new layer. However, this heuristic might not minimize the total number of layers. Figure 6.4 (a) gives a sample case of six buses. If we follow the heuristic, we would first assign buses b_1, b_2, b_3, b_4 to one layer because they are the largest set of buses that are compatible with each other. Then we have to assign b_5 and b_6 to different layers because r_5 and r_6 overlap with each other. This layer assignment consumes three layers (see Figure 6.4 (b)). However, if we assign b_1, b_2, b_6 to one layer and b_3, b_4, b_5 to the other, we will use only two layers (see Figure 6.4 (c)). Therefore, we need a layer assignment algorithm that gives us the minimum number of layers for a multi-layer design. In the next section, we will present such an algorithm.

6.3 Our Solution

At first glance, this layer assignment problem looks like a coloring problem: find out the minimum number of colors to color the buses so that incompatible buses are assigned different colors. However, a general coloring problem is NP-hard [55]. In this section, we will show that this layer assignment problem has a very nice property (Lemma 6) that makes it polynomial time solvable. Before we present the polynomial time solution, we will first introduce some terminologies and notations.

A bus b_i has two intervals: l_i in the left array and r_i in the right array. Each interval l_i is defined by its two endpoints: the upper endpoint l_i^u and the lower endpoint l_i^l. Similarly, r_i is defined by r_i^u and r_i^l. We denote the

[1] In [16], each bus has a weight, so the maximum subset means the subset of buses with the maximum total weight. However, the bus weight does not affect the minimal number of layers in a multi-layer problem. Therefore, we assume all the buses have the same weight in our discussion.

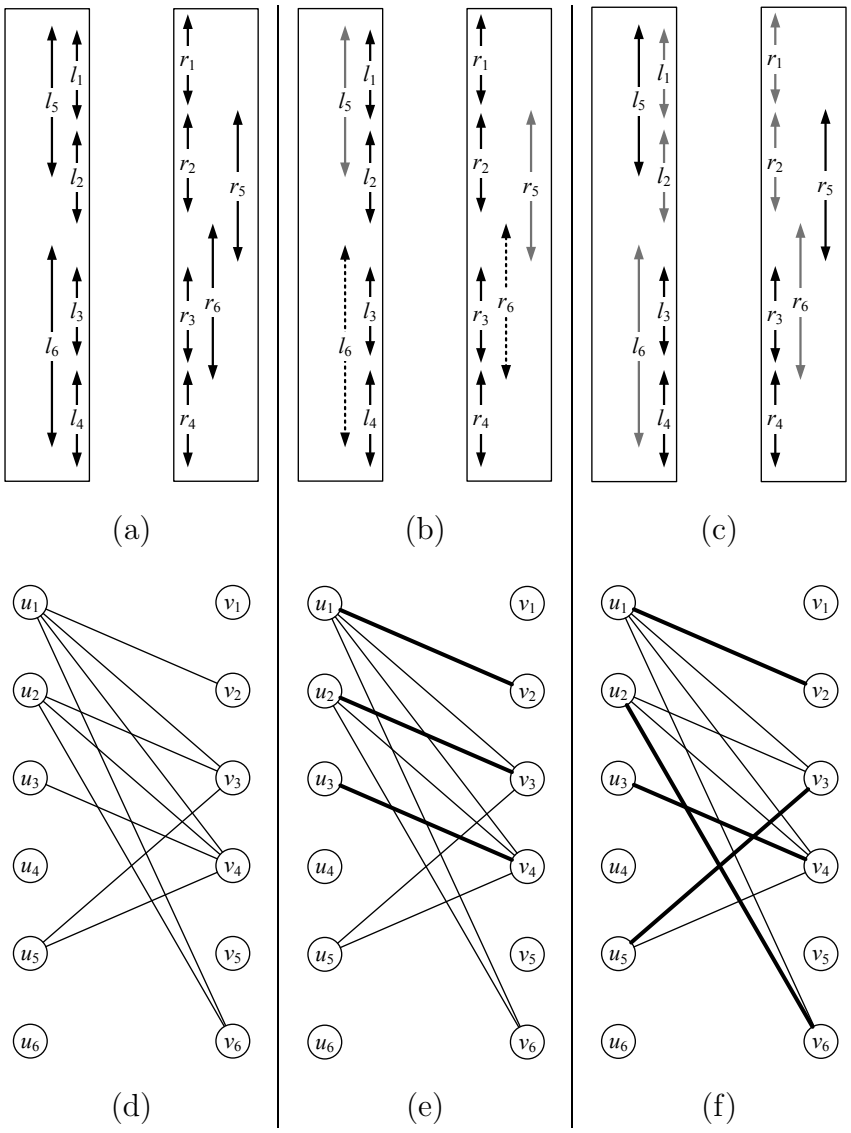


Figure 6.4: An example of the layer assignment problem is given in (a). The heuristic implied by [16] produces a three-layer solution (b) while the optimal layer assignment needs only two layers as in (c). Buses represented by the same line style (solid, dotted, gray) are assigned to the same layer; different line styles indicate different layers. (d) The corresponding bipartite graph G_B for this problem. The matchings indicated by thick edges in (e) and (f) correspond to the layer assignments in (b) and (c) respectively.

"above" relation by the symbol "$\succ$"; that is, if an endpoint a is located *above* another endpoint b, we denote it as $a \succ b$. Naturally, we have $l_i^u \succ l_i^l$ and $r_i^u \succ r_i^l$. We also define the "above" relation between buses as follows.

Definition 5. For two buses b_i and b_j ($b_i \neq b_j$), we say b_i is *above* b_j (denoted as $b_i \succ b_j$) if and only if $l_i^l \succ l_j^u$ and $r_i^l \succ r_j^u$.

Compatibility of buses can be expressed by this "above" relation. The following lemma can be easily verified.

Lemma 4. *Two buses b_i and b_j ($b_i \neq b_j$) are compatible if and only if $b_i \succ b_j$ or $b_j \succ b_i$.*

It can also be verified that this "above" relation is transitive:

Lemma 5. *If $b_i \succ b_j$ and $b_j \succ b_k$, then $b_i \succ b_k$.*

We also define the "immediately above" relation which is more strict than "above." Notice that this relation is defined only for a valid layer assignment.

Definition 6. For two buses b_i and b_j ($b_i \neq b_j$) assigned to the same layer l in a valid layer assignment, we say b_i is *immediately above* bus b_j (denoted as $b_i \succcurlyeq b_j$) if $b_i \succ b_j$ and $\nexists b_k \in$ layer $l, b_i \succ b_k \succ b_j$.

Although the "above" relation is transitive, the "immediately above" relation is not. In fact, one bus can have at most one other bus immediately above it.

Lemma 6. *If two buses $b_i, b_j \in$ layer l and $b_i \succcurlyeq b_j$, then the following two statements are true:*

- $b_k \in$ *layer* $l, b_k \succcurlyeq b_j \Longrightarrow b_k = b_i$
- $b_k \in$ *layer* $l, b_i \succcurlyeq b_k \Longrightarrow b_k = b_j$

Proof. We only prove the first statement; the second one can be proved in the same way. Suppose $b_i \succcurlyeq b_j$ and $b_k \succcurlyeq b_j$ but $b_k \neq b_i$. Since the layer assignment is valid and all three buses are assigned to the same layer, we must have $b_k \succ b_i$ or $b_i \succ b_k$ (Lemma 4). If $b_k \succ b_i$, then we have $b_k \succ b_i \succ b_j$ (Lemma 5), which contradicts $b_k \succcurlyeq b_j$. If $b_i \succ b_k$, then we have $b_i \succ b_k \succ b_j$, which contradicts $b_i \succcurlyeq b_j$. □

With the preceding notations and lemmas, we now present our solution. We solve the layer assignment problem of escape routing of buses by transforming it into a *bipartite matching* problem. Given a set of buses B, we construct its corresponding bipartite graph G_B as follows: for any bus $b_i \in B$, we create two nodes u_i and v_i in G_B. For any pair of buses b_i and b_j, if $b_i \succ b_j$, then we create an edge between node u_i and node v_j in G_B. Since edges are

always created between u nodes and v nodes, G_B is a bipartite graph. In Figure 6.4, (d) gives the corresponding bipartite graph of the problem in (a).

There is a one-to-one correspondence between valid layer assignments of the buses B and matchings of the bipartite graph G_B:

Theorem 7. *A valid layer assignment of a set of n buses B that uses k layers corresponds to a matching of $n-k$ edges in G_B.*

Proof. From a valid layer assignment of B, we can construct a matching M_B of G_B as follows: For any pair of buses b_i and b_j, if $b_i \succcurlyeq b_j$, then include edge $e = (u_i, v_j)$ into edge set M_B. Note that if two edges $e_1 = (u_i, v_j)$ and $e_2 = (u_i, v_k)$ are incident to the same node u_i, then we know $b_i \succcurlyeq b_j$ and $b_i \succcurlyeq b_k$. From Lemma 6, we know that $b_j = b_k$, meaning that e_1 and e_2 are identical. Similarly, no two different edges can be incident to the same v node. Therefore, the resultant M_B is a matching.

The reverse correspondence from matching to valid layer assignment can be constructed by reversing the above procedure: for any edge $e = (u_i, v_j)$ in matching M_B, assign b_i and b_j to the same layer. This "assigned-to-the-same-layer" relation is transitive: if b_i and b_j are assigned to the same layer (because $e_1 = (u_i, v_j) \in M_B$) and b_j and b_k are assigned to the same layer (because $e_2 = (u_j, v_k) \in M_B$), then b_i and b_k are also assigned to the same layer (although edge $e_3 = (u_i, v_k) \notin M_B$ because M_B is a matching and e_1 and e_3 are both incident to u_i). Therefore, two buses b_i and b_j are assigned to the same layer either directly through a matching edge $e = (u_i, v_j) \in M_B$ (or $e = (u_j, v_i) \in M_B$) or through transitivity. If it is the former case, we know b_i and b_j are compatible becauase such edge exists in the graph only when $b_i \succ b_j$ (or $b_j \succ b_i$). If it is the later case, then there must be a sequence of edges $e_1, e_2, \ldots, e_m$ in the matching M_B to bridge the two buses. Notice that no two edges in a matching can be incident to the same node. Therefore, the sequence of edges must be of the form $e_1 = (u_1, v_2), e_2 = (u_2, v_3), e_3 = (u_3, v_4), \ldots, e_m = (u_m, v_{m+1})$ and $u_1 = u_i, v_{m+1} = v_j$ (or symmetrically, $u_1 = u_j, v_{m+1} = v_i$). According to the definition of the graph, we would have $b_i = b_1 \succ b_2 \succ \ldots \succ b_m \succ b_{m+1} = b_j$. Because the "above" relation is transitive, we have $b_i \succ b_j$ and therefore they are compatible. The symmetrical case of $u_1 = u_j, v_{m+1} = v_i$ will lead to $b_j \succ b_i$ for the same reason. As a result, any two buses assigned to the same layer are compatible, meaning that the assignment is valid. Note that the

number of transitive closures of the "assigned-to-the-same-layer" relation is actually the number of layers.

Assume that the number of buses assigned to layer l is n_l. Then for all buses on layer l, we will find exactly $n_l - 1$ pairs of buses with this "immediately above" relation. So we will add exactly $n_l - 1$ edges to M_B. Since the total number of buses $n = \sum_{l=1}^{k} n_l$, the total number of edges in M_B is exactly $\sum_{l=1}^{k}(n_l - 1) = n - k$. □

The following corollary follows directly from Theorem 7.

Corollary 1. *The optimal layer assignment of B corresponds to the maximum matching of G_B.*

In Figure 6.4, (f) gives the maximum matching of (d), and it can be seen that the matching has exactly $6 - 2 = 4$ edges. It can also be seen from (e) that the matching corresponding to the layer assignment of the heuristic implied by [16] has 3 edges. That is the reason the layer assignment uses $6 - 3 = 3$ layers.

Now we analyze the time complexity of our approach. Constructing the graph G_B from the input buses B takes only $O(n^2)$ time in which n is the total number of buses because we need to examine every pair of buses to see if they satisfy the "above" relation. Computing the maximum matching of G_B takes $O(n^{2.38})$ time [54]. Then converting the maximum matching result into layer assignment takes $O(n^2)$ time because for each bus, we need to scan through all the layer-assigned buses to see if it can be assigned to the same layer as such buses. As a result, the total complexity of our method is $O(n^{2.38})$.

6.4 Experimental Results

We implement our layer assignment algorithm in C++ and compare it with the heuristic approach implied by [16]. The maximum matching is computed using the max-flow solver HIPR [56]. We tested the two algorithms on eight test cases derived from industrial data. The experiments are performed on a workstation with a 3.0 GHz Intel Xeon CPU and 4 GB memory.

Table 6.1 shows the results of our experiment. The second column gives the number of buses in each test case. The third and fourth columns present

Table 6.1: Experimental results of our layer assignment algorithm.

	# of Buses	# of layers		
		optimal	heuristic [16]	diff.
Test case 1	110	29	30	1
Test case 2	38	12	13	1
Test case 3	21	12	12	0
Test case 4	16	7	7	0
Test case 5	23	12	12	0
Test case 6	38	12	14	2
Test case 7	14	6	7	1
Test case 8	8	4	4	0

the number of layers in the optimal layer assignment produced by our algorithm and that of the layer assignment produced by the heuristic [16]. Their difference is given in the last column. Since the runtimes of both approaches are negligible, we do not list the runtime in our table.

It can be seen that our optimal algorithm uses fewer layers than the heuristic in four out of the eight test cases. The one or two layer improvement may seem insignificant compared to the total number of layers used. However, the impact on the manufacturing cost is significant, especially for high-density boards. This is because high-density boards usually have higher defective rates. Therefore, by reducing even just one layer of routing, the yield can be increased substantially and the manufacturing cost can be reduced. For example, if we reduce the number of layers in a high-density board from 22 to 18, the manufacturing cost can be cut from \$850 to \$500, a 44% decrease, although the number of layers is decreased by only 18% [57].

6.5 Conclusion

In this chapter, we presented an optimal layer assignment algorithm for escape routing of buses and proved that it guarantees to output a layer assignment with the minimum number of layers. Experimental results also show layer number improvement over a heuristic implied by a previous work [16]. Our algorithm can also be extended. For example, if we prefer two buses to be assigned to the same layer, we can assign a higher weight to the edge connecting the two buses in the bipartite graph G_B. If we compute the max-

imum *weighted* matching for the graph, we can increase the chance that the two buses are assigned to the same layer.

CHAPTER 7

CONCLUSIONS AND FUTURE WORKS

In this dissertation, we have studied modern PCB routing problems. Topics that have been covered in our study include: escape routing, length-constrained routing and layer assignment.

First, we focused on the escape routing problem. In Chapter 2, we proposed a network flow based escape routing algorithm that correctly captures the diagonal capacity. It is the first optimal escape routing algorithm that can handle pin arrays with diagonal capacities. In Chapter 3, we studied the escape routing problem of differential pairs and proposed two algorithms. The objective was to keep the routing paths of a differential pair as close as possible while minimizing the total wire length. The first algorithm we proposed computes the optimal routing for a single differential pair while our second algorithm is able to simultaneously route multiple differential pairs. We also built a two-stage routing scheme based on these two algorithms.

We then studied the length-constrained routing problem and a related net untangling problem. In Chapter 4, we proposed the first length-constrained routing algorithm that has no topological restrictions. Moreover, due to its gridless feature, our algorithm is faster than a previous gridded router [15]. Our key idea was to turn the routing problem into an area assignment problem and use a placement structure to help solve the area assignment problem. In Chapter 5, we studied how to untangle the twisted nets before length-constrained routing. By introducing a single-detour routing style, we greatly simplified the net untangling problem. We then discovered a necessary and sufficient condition for a single-detour routing solution to exist. We have also proposed a dynamic programming based algorithm to solve the single-detour untangling problem. The algorithm was proved to be optimal.

In Chapter 6, we studied the layer assignment problem of buses. We presented a layer assignment algorithm to assign the escape routing regions of buses into multiple layers without any conflict between the buses. The

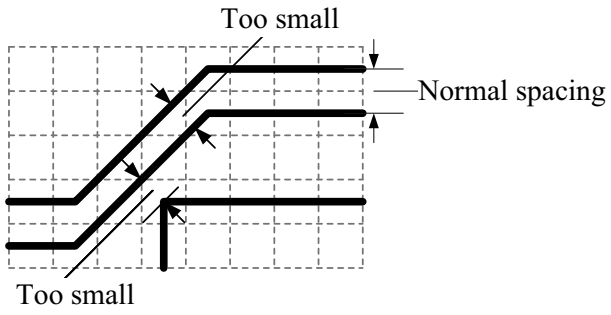


Figure 7.1: Routing diagonal wires on rectangular grid will cause too small wire spacing.

key idea was to transform the layer assignment problem into a bipartite matching problem. Our algorithm was then proved to be optimal, meaning that it guarantees to use the minimum number of layers.

To conclude this dissertation, we would like to point out some future research directions and open problems:

Monotonic routing style has been extensively studied for package escape routing, which is an ordered escape routing problem. Researchers have even discovered necessary and sufficient conditions for feasible monotonic routing to exist in an ordered escape routing problem [22, 29, 30]. However, few studies utilize these research results to solve the simultaneous escape routing problem. Whether the monotonic routing constraint can lead to optimal algorithms or at least good heuristics for the simultaneous escape routing problem is a open problem.

Diagonal routing (45° routing) is pervasively used in PCB routing to shorten the wire length. However, existing studies on the length-constrained routing problem all use only orthogonal routing. How to produce length-constrained routing solution using both orthogonal and diagonal routes is an interesting yet challenging problem. The challenge here is that the rectangular routing grid which we are familiar with does not precisely capture the routing resource occupied by a diagonal wire. Figure 7.1 illustrates this challenge. It can be seen that if we route diagonal wires on the traditional rectangular routing grid, we may produce too small wire spacing. To effectively solve this problem, we need to either find an alternative routing structure or use a gridless approach.

APPENDIX A

PROOF OF THEOREM 5

In this appendix, we prove Theorem 5. We prove the necessary condition (*only if*) by proving its contrapositive: if any of the forbidden patterns appears in the pin sequence, then there exists no feasible solution for the SDU problem. We prove this by showing that the forbidden patterns will lead to either intersection between wires or incorrect ordering of the wires. Proving the sufficient condition (*if*) is more difficult. We prove it by designing an algorithm that constructs a solution to the SDU problem. By carefully designing the algorithm, we are able to show that the solution it produces is infeasible only when a forbidden pattern appears in the input pin sequence.

In the rest of this appendix, we will first introduce some definitions and some helpful lemmas. Then we will prove the necessary condition and sufficient condition in detail.

A.1 Some Definitions and Lemmas

In the following definitions, we assume that a feasible solution to the SDU problem (Problem 3) is already known and all the definitions are based on that solution. Therefore, instead of saying "the pins in a feasible solution to the SDU problem," we just say "the pins." See Figure A.1 for an illustration of our definitions.

Definition 7. All the pins can be classified into two categories: pins whose routing does not detour and pins whose routing does detour. We call the former *straight pins* and the later *detour pins*.

In later discussion, we will need the problem to be bounded by straight pins. For this purpose, we add two virtual pins 0 and $n + 1$ to the solution at the very top and very bottom. They are considered straight pins (see the

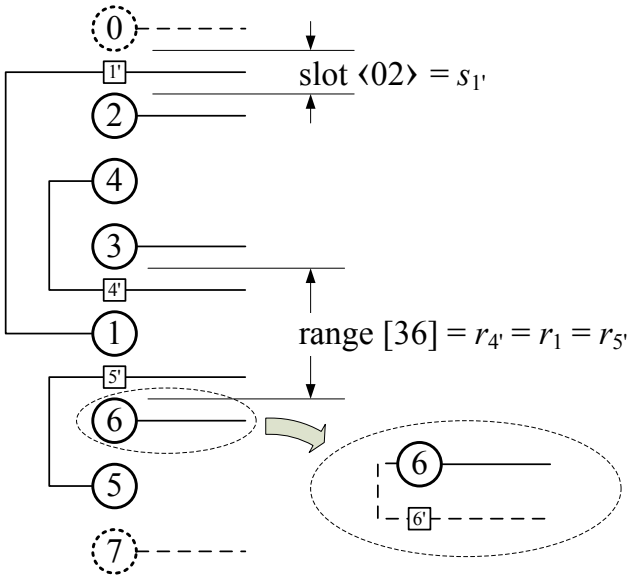


Figure A.1: An SDU problem with pin sequence $(2, 4, 3, 1, 6, 5)$ and its solution. Circles represent pins and squares represent exits.

dashed pins in Figure A.1). In Figure A.1, pins $0, 2, 3, 6$ and 7 are straight pins and the rest are detour pins.

Definition 8. If we draw a vertical line through all the pins, the routing of a detour pin will intersect this line exactly once. We call the intersection the *exit* of this pin. For a straight pin (including virtual pin), we pretend that the routing detours downward a little bit and intersects with the vertical line (see the illustration on the bottom right corner of Figure A.1). Such modification to the routing does not invalidate the solution. We take the intersection as the exit of the straight pin. The exit of a pin i is denoted as i'. We list all pins and exits from top to bottom and obtain a sequence. We call this sequence the *solution sequence*. Since the exit i' of a straight pin i always follows the pin immediately in the sequence, we usually use ii' to denote the pin and the exit together.

For example, the solution sequence of Figure A.1 is $(00', 1', 22', 4, 33', 4', 1, 5', 66', 5, 77')$.

Definition 9. For two objects x and y in the solution sequence (an object is either a pin or an exit), if x appears *before* y in the sequence, then x is *above* y in the solution. We denote it as $x \prec y$. If x's ID is *smaller* than y's

(the ID of an exit is the same as its corresponding pin's ID), we denote it as $x < y$. $x \leq y$ means $x < y$ or $x = y$.

It is obvious that both $<$ and $\prec$ are *asymmetric* ($x \prec y \Rightarrow y \not\prec x$) and *transitive* ($x \prec y, y \prec z \Rightarrow x \prec z$).

Definition 10. For a detour pin i, if $i' \prec i$ we call the pin *up-detouring*; otherwise, we call it *down-detouring*.

Definition 11. A *range* is a pair of straight pins $a \prec b$ such that there are no straight pins between a and b in the solution sequence, i.e., $\nexists$ straight pin p, $a \prec p \prec b$. We denote the range as $[ab]$. If an object x (either a pin or an exit) lies inside a range $[ab]$, i.e., $a \prec x \prec b$, we call this range x's range and denote it as r_x. We also call pin a the *upper straight pin* of x and denote it as u_x and call pin b the *lower straight pin* of x and denote it as l_x.

For example, in Figure A.1, $r_{4'} = r_1 = r_{5'} = [36]$, $u_{4'} = u_1 = u_{5'} = 3$ and $l_{4'} = l_1 = l_{5'} = 6$.

Definition 12. A *slot* is a pair of adjacent pins (either straight or detour) $x \prec y$ in the solution sequence. Between them there are only exits, i.e., $\nexists$ pin $p, x \prec p \prec y$. We denote the slot as $\langle xy \rangle$. The slot inside which an exit i' lies is called the slot of i' and is denoted as $s_{i'}$.

For example, $s_{4'} = \langle 31 \rangle$ in Figure A.1.

The following lemma is straightforward:

Lemma 7. *There is a one-to-one mapping between the topologies of single detour solutions and the solution sequences.*

Therefore, we can talk about feasibility based on the solution sequence instead of the routing topology:

Lemma 8. *A solution is feasible* if and only if *its corresponding solution sequence satisfies the following two requirements:*

1. *For any two pins $i < j$, $i' \prec j'$.*
2. *There exist no two pins $i < j$ such that*
   - *$i \prec j \prec i' \prec j'$ or*

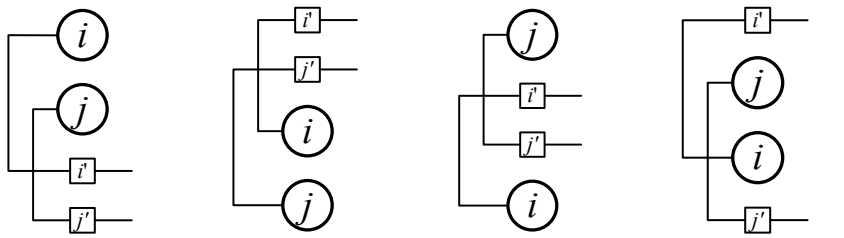


Figure A.2: The four cases when wires have intersections.

- $i' \prec j' \prec i \prec j$ *or*
- $j \prec i' \prec j' \prec i$ *or*
- $i' \prec j \prec i \prec j'$

Proof. The first requirement is for net ordering. The ordering of the exits should be monotonically increasing from top to bottom. The second requirement is to guarantee that the detouring part of the routes (the routing in the dark area in Figure 3.6 (b)) does not have intersections. By enumerating all possible intersections between two wires (see Figure A.2), we can obtain the four cases listed in the second requirement. □

Definition 13. If a solution sequence satisfies the two requirements in Lemma 8, we call the solution sequence a *feasible* solution sequence.

The following lemma gives an equivalent definition on forbidden patterns:

Lemma 9. *The following two statements are equivalent:*

- *"Pin sequence P contains pattern $(3,4,1,2)$, $(2,4,1,3)$ or $(3,1,4,2)$"*
- *"Pin sequence P contains four pins $a < b < c < d$ such that $c \prec d \prec a \prec b$ or $b \prec d \prec a \prec c$ or $c \prec a \prec d \prec b$"*

Proof. Straightforward from the definitions. □

With the help of Lemma 8 and Lemma 9, we can rewrite Theorem 5 as the following equivalent theorem:

Theorem 8. *For an SDU problem, there exists a solution sequence that satisfies the two requirements in Lemma 8* if and only if *the pin sequence of the problem does not contain four pins $a < b < c < d$ such that $c \prec d \prec a \prec b$ or $b \prec d \prec a \prec c$ or $c \prec a \prec d \prec b$.*

In the next two sections, we will prove the necessary condition and the sufficient condition of this equivalent theorem instead of the original Theorem 5. This theorem, together with Lemma 8 and Lemma 9, proves Theorem 5.

A.2 Necessary Condition

Before we prove the necessary condition, we introduce a lemma:

Lemma 10. *For two pins $i < j$ in a feasible solution sequence, we have the following two cases:*

1. *If $i \prec j$, then $i' \prec j$ and $i \prec j'$.*
2. *If $j \prec i$, then either $i' \prec j$ or $i \prec j'$.*

Proof. Because $i < j$, we have $i' \prec j'$ (requirement 1 in Lemma 8). For the first case, suppose $j \prec i'$; we have a full ordering $i \prec j \prec i' \prec j'$, which is forbidden by requirement 2 in Lemma 8. Suppose $j' \prec i$; we have another full ordering $i' \prec j' \prec i \prec j$, which again is forbidden. For the second case, suppose $j \prec i'$ and $j' \prec i$; then we have a full ordering $j \prec i' \prec j' \prec i$, which is forbidden by Lemma 8. □

Now we prove the contrapositive of the necessary condition of Theorem 8: If a pin sequence contains four pins $a < b < c < d$ with $c \prec d \prec a \prec b$ or $b \prec d \prec a \prec c$ or $c \prec a \prec d \prec b$, then there exists no solution sequence that satisfies both requirements of Lemma 8. If this contrapositive is true, then the necessary condition itself is true. We discuss the three cases one by one:

1. $c \prec d \prec a \prec b$: Because $a < b$ and $a \prec b$, we know $a \prec b'$ (case 1 in Lemma 10). Similarly, since $c < d$ and $c \prec d$, we have $c' \prec d$. Therefore, we have $c' \prec d \prec a \prec b'$ ($\prec$ is transitive). Because $b < c$, we have $b' \prec c'$ (requirement 1 in Lemma 8) which contradicts $c' \prec b'$ obtained earlier ($\prec$ is asymmetric).

2. $b \prec d \prec a \prec c$: Because $a < c$ and $a \prec c$, we have $a \prec c'$ (case 1 in Lemma 10). This, together with $d \prec a$, gives us $d \prec c'$. Because $c < d$ and $d \prec c$, we know either $c' \prec d$ or $c \prec d'$ (case 2 in Lemma 10). Since $c' \prec d$ contradicts $d \prec c'$, the only possibility is $c \prec d'$. We can also obtain $a' \prec b$ by similar argument. Then, we have $a' \prec d \prec a \prec d'$ which violates requirement 2 of Lemma 8.

3. $c \prec a \prec d \prec b$: Because $a < b$ and $a \prec b$, we have $a \prec b'$ (case 1 in Lemma 10). This, together with $c \prec a$, gives us $c \prec b'$. From $b < c$ and $c \prec b$, we know that either $b' \prec c$ or $b \prec c'$ (case 2 in Lemma 10). Since $b' \prec c$ contradicts $c \prec b'$, the only possibility is $b \prec c'$. However, we can also obtain $c' \prec d$ because $c < d$ and $c \prec d$ (case 1 in Lemma 10). This, together with $d \prec b$, gives $c' \prec b$ which contradicts $b \prec c'$ obtained earlier.

We have derived contradictions from all three cases. This means that if any of the three cases happens, no feasible solution sequence exists.

A.3 Sufficient Condition

In this section we prove the sufficient condition of Theorem 8: if a pin sequence does not contain four pins $a < b < c < d$ such that $c \prec d \prec a \prec b$ or $b \prec d \prec a \prec c$ or $c \prec a \prec d \prec b$, then there exists a solution sequence that satisfies both requirements of Lemma 8. In order to prove this, we first present an algorithm that constructs a solution sequence for a given pin sequence and then show that the algorithm fails to produce a feasible solution sequence only if the input pin sequence contains one of the three forbidden patterns.

Our construction algorithm takes in a pin sequence and produces a solution sequence. The algorithm consists of five steps (an example of the execution of our algorithm in shown in Figure A.3):

1. Solution sequence = pin sequence + two virtual pins 0 and $n + 1$.

2. Determine the straight pins: Compute the *longest increasing subsequence* [58] of the solution sequence. If there are multiple choices, pick one arbitrarily. Let pins in the subsequence be straight pins. That

$S = \{0, 5, 2, 4, 3, 1, 7, 6, 8\}$

(a) After step 1

$S = \{00', 5, 22', 4, 33', 1, 77', 6, 88'\}$

(b) After step 2

$r_{5'} = r_{4'} = r_{6'} = [37]$, $r_{1'} = [02]$

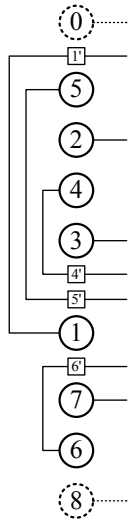
(c) After setp 3

$s_{4'} = s_{5'} = \langle 31 \rangle$, $s_{6'} = \langle 17 \rangle$, $s_{1'} = \langle 05 \rangle$

(d) After step 4

$S = \{00', 1', 5, 22', 4, 33', 4', 5', 1, 6', 77', 6, 88'\}$

(e) After step 5 (final solution sequence)



(f) Corresponding routing

Figure A.3: An example of how our algorithm works on pin sequence $\{5, 2, 4, 3, 1, 7, 6\}$. 0 and 8 are virtual pins. S means solution sequence.

is, for every pin i in the longest increasing subsequence, we insert i' immediately after i in the solution sequence. Notice that the two virtual pins 0 and $n + 1$ are included in the subsequence. Pins not in the subsequence are made detour pins.

3. Determine the ranges of the exits: For every detour pin i, find two adjacent straight pins $a < b$ in the longest increasing subsequence such that $a < i < b$. Let $r_{i'} = [ab]$. We can also determine whether i is up-detouring or down-detouring. If $i \prec a$, then the pin is down-detouring. If $b \prec i$, then it is up-detouring.

4. Determine the slots of the exits: For every detour pin i,

   - If i is up-detouring, then find slot $\langle xy \rangle$ inside $r_{i'}$ such that $((l_i < x) \vee (x \leq u_{i'})) \wedge (l_{i'} \leq y < l_i)$ ($\vee$ means *or* and $\wedge$ means *and*).
   - If i is down-detouring, then find slot $\langle xy \rangle$ inside $r_{i'}$ such that $(l_i < x \leq u_{i'}) \wedge ((l_{i'} \leq y) \vee (y < l_i))$.

   Insert i' into $\langle xy \rangle$ ($s_{i'} = \langle xy \rangle$).

5. Sort inside each slot: Within each slot, we sort the exits according to their pin IDs. The IDs of the resultant exits present an increasing order

inside each slot.

We must first prove that we can always find a unique range and a unique slot for every detour pin. Otherwise, we may get stuck at step 3 or 4 in the algorithm.

For step 3, since we have an increasing subsequence with 0 at its head and $n+1$ at its tail, we can always find adjacent a and b in the subsequence for each detour pin i such that $a < i < b$. Furthermore, the pins inside a range by the algorithm have the following properties (Lemma 11 and Lemma 12):

Lemma 11. *For a detour pin i, $r_i \neq r_{i'}$.*

Proof. Suppose $r_i = r_{i'} = [ab]$. We then have $a < i < b$ and $a \prec i \prec b$, which means that we can make the increasing subsequence even longer by adding i into it. This contradicts the fact that we have already computed the *longest* increasing subsequence in step 2. □

Lemma 12. *The pins in any range $[ab]$ can be divided into two continuous sequences $X \prec Y$, i.e., the pin sequence is in the form $\ldots, a, X, Y, b, \ldots$, such that:*

1. *All pins $x \in X$ have IDs $< a$, all pins $y \in Y$ have IDs $> b$.*
2. *Both sequences are decreasing.*

Notice that both X and Y could be empty.

Proof. Statement 1: According to Lemma 11, for a pin p in range $[ab]$, either $p < a$ or $b < p$. So pins in $[ab]$ form two subsequences X and Y, all pins $x \in X < a$, all pins $y \in Y > b$. Now we only need to show that $X \prec Y$. That is, $\forall x \in X, \forall y \in Y, x \prec y$. Suppose this is not true; then we can find two pins $x < a$ and $b < y$ in range $[ab]$ such that $y \prec x$. In this case, the pin sequence contains four pins $x < a < b < y$ such that $a \prec y \prec x \prec b$. This violates our assumption.

Statement 2: If X is not decreasing, then we can find two pins $p \prec q$ in range $[ab]$ such that $p < q < a$. Pins p and q must be up-detouring because their IDs $< a$. This means that a cannot be the virtual pin 0 and thus there is at least one straight pin $h \prec a$ (h could be an actual pin or the virtual pin 0). We have the following cases:

1. $h < p < q < a < b$ (notice that $h = 0$ is included in this case): Subsequence $\ldots h \ldots p \ldots q \ldots b \ldots$ is an increasing subsequence. Therefore, choosing it instead of $\ldots h \ldots a \ldots b \ldots$ increases length of the increasing subsequence by 1. This contradicts the fact that we have already computed the *longest* increasing subsequence in step 2.

2. $p < h < q < a < b$: The pin sequence contains four pins $p < h < q < a$ such that $h \prec a \prec p \prec q$. This violates our assumption.

3. $p < q < h < a < b$: The pin sequence contains four pins $p < q < h < a$ such that $h \prec a \prec p \prec q$. This violates our assumption.

All the cases lead to contradictions. Therefore, X must be decreasing. We can prove that Y is decreasing by similar argument. □

Because of the property in Lemma 12, it is always possible to find a slot $\langle xy \rangle$ in range $r_{i'}$ for an up-detouring pin i such that $((l_i < x) \vee (x \leq l_{i'})) \wedge (l_{i'} \leq y < l_i)$. Notice that $\nexists$ pin $x \in r_{i'}$ such that $u_{i'} < x < l_{i'}$ (Lemma 11). The above condition is the same as $((l_i < x) \vee (x \leq u_{i'})) \wedge (l_{i'} \leq y < l_i)$. The down-detouring pin case can be proved in a similar way. Therefore, we can always find one slot for a detour pin in step 4 of our algorithm. The other steps in the algorithm involve only standard procedures such as longest increasing subsequence computation (step 2) and sorting (step 5) and thus will not cause any problem. As a result, our algorithm will always terminate normally and produce a solution sequence if the input does not contain the forbidden patterns. Next we prove the feasibility of the solution sequence produced.

Lemma 13. *For a detour pin i in the solution sequence produced by our algorithm,*

- *If i is up-detouring, then for any pin $p \in r_{i'}$:*
  - *If $i' \prec p \prec l_{i'}$, then $i' \prec p' \prec i$.*
  - *If $u_{i'} \prec p \prec i'$, then $(i \prec p') \vee (p' \prec i')$.*
- *If i is down-detouring, then for any pin $p \in r_{i'}$:*
  - *If $i' \prec p \prec l_{i'}$, then $(p' \prec i) \vee (i' \prec p')$.*
  - *If $u_{i'} \prec p \prec i'$, then $i \prec p' \prec i'$.*

Proof. Let $s_{i'} = \langle xy \rangle$. We prove the four cases one by one (the ordering of the four cases are changed for better presentation):

1. i is up-detouring and $u_{i'} \prec p \prec i'$: In step 4 of our algorithm we have $(l_i < x) \vee (x \leq u_{i'})$. According to Lemma 12, we know $(l_i < x \leq p) \vee (p < u_{i'})$. According to Lemma 14, this indicates $(l_i \prec p') \vee (p' \prec u_{i'})$. Since $i \prec l_i$ and $u_{i'} \prec i'$, we have $(i \prec p') \vee (p' \prec i')$.

2. i is down-detouring and $u_{i'} \prec p \prec i'$: In step 4 of our algorithm we have $l_i < x \leq u_{i'}$. According to Lemma 12, we have $l_i < x \leq p < u_{i'}$. According to Lemma 14, this indicates $l_i \prec p' \prec u_{i'}$. Because $i \prec l_i$ and $u_{i'} \prec i'$, we have $i \prec p' \prec i'$.

3. i is down-detouring and $i' \prec p \prec l_{i'}$: In step 4 of our algorithm, we have $(l_{i'} \leq y) \vee (y < l_i)$. According to Lemma 12, we obtain $(l_{i'} < p) \vee (p \leq y < l_i)$. According to Lemma 14, this indicates $(l_{i'} \prec p') \vee (p' \prec l_i)$. Because $i' \prec l_{i'}$, the first term $(l_{i'} \prec p')$ indicates $i' \prec p'$. The second term $(p' \prec l_i)$ incurs two possibilities: $p' \prec i$ and $i \prec p' \prec l_i$. The second possibility indicates that $u_{p'} \prec i \prec p'$. Notice that p is an up-detouring pin $(p' \prec l_i \prec i' \prec p)$, this is exactly case 1 of our proof with p and i swapped. Therefore, we know $(p \prec i') \vee (i' \prec p')$. This contradicts the assumption that $p' \prec l_i \prec i' \prec p$. Therefore, the only possibility for the second term is $p' \prec i$. Putting them together, we have $(p' \prec i) \vee (i' \prec p')$.

4. i is up-detouring and $i' \prec p \prec l_{i'}$: In step 4 of our algorithm we have $l_{i'} \leq y < l_i$. According to Lemma 12, we have $l_{i'} < p \leq y < l_i$. We then have $l_{i'} \prec p' \prec l_i$ by Lemma 14. There are two cases: $l_{i'} \prec p' \prec i$ and $i \prec p' \prec l_i$. The second case indicates that $u_{p'} \prec i \prec p'$. Notice that p is down-detouring $(l_p = l_{i'} \prec p')$; this is exactly case 2 of our proof with p and i swapped. Therefore, we can obtain $p \prec i' \prec p'$. This contradicts the assumption that $i' \prec p$. Therefore, the only possibility is $l_{i'} \prec p' \prec i$. Considering $i' \prec l_{i'}$, this gives us $i' \prec p' \prec i$.

□

The following two lemmas complete our proof by showing that the solution sequence produced by our algorithm satisfies both requirements in Lemma 8 if the input pin sequence does not contain four pins $a < b < c < d$ such

that $c \prec d \prec a \prec b$ or $b \prec d \prec a \prec c$ or $c \prec a \prec d \prec b$. Lemma 14 shows that requirement 1 is satisfied and Lemma 15 shows that requirement 2 is satisfied.

Lemma 14. *For any two pins $i < j$, we have $i' \prec j'$ in our solution sequence.*

Proof. From the algorithm we know that if one of the pins is a straight pin, then the lemma is true. Now suppose we have two detour pins $i < j$ but $j' \prec i'$. We know $s_{i'} \neq s_{j'}$ because we sort the exits in every slot in step 5 of our algorithm. Furthermore, we know $r_{i'} = r_{j'}$ because otherwise the two straight pins $l_{j'}$ and $u_{i'}$ form a decreasing subsequence ($u_{i'} < i < j < l_{j'}$ but $l_{j'} \prec u_{i'}$ because $j' \prec i'$ and $u_{i'} \neq l_{j'}$) which contradicts step 2 of our algorithm. As a result, i' and j' must belong to the same range but different slots. There are four cases:

1. i and j are both down-detouring: In order to have $j' \prec i'$, there must exist one straight pin x such that $l_i < x < l_j$ according to step 4 of our algorithm and Lemma 12. Since straight pins l_i and l_j form increasing subsequence, we have $i \prec l_i \prec j \prec l_j$. Since both i and j are down-detouring, we have $l_j \leq u_{j'} = u_{i'} < i$. Therefore, the pin sequence has four pins $l_i < l_j < i < j$ such that $i \prec l_i \prec j \prec l_j$. This contradicts our assumption.

2. i and j are both up-detouring: This case is symmetrical to the previous case and can be shown to lead to contradictions in a similar way.

3. i is down-detouring and j is up-detouring: $j' \prec i'$ is impossible because step 4 of our algorithm and Lemma 12 guarantees that the exit of a down-detouring pin is always inserted above the exit of an up-detouring pin if the two exits belong to the same range but different slots.

4. i is up-detouring and j is down-detouring: The pin sequence has four pins $u_{i'} < i < j < l_{i'}$ such that $j \prec u_{i'} \prec l_{i'} \prec i$. This contradicts our assumption.

All cases lead to contradictions. Therefore, $i < j \Rightarrow i' \prec j'$ in our solution sequence. □

Lemma 15. *None of the following will happen for two pins $i < j$ in our solution sequence:*

1. $i \prec j \prec i' \prec j'$ *or*
2. $i' \prec j' \prec i \prec j$ *or*
3. $j \prec i' \prec j' \prec i$ *or*
4. $i' \prec j \prec i \prec j'$

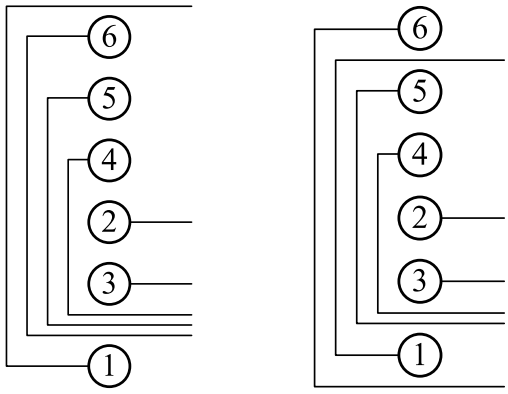
Proof. We discuss the four cases one by one:

1. $i \prec j \prec i' \prec j'$: Suppose there is no straight pin between i and j ($r_i = r_j$). Since both i and j are down-detouring, they both belong to subsequence Y in Lemma 12. Then $i \prec j$ and $i < j$ contradict the statement that Y must be decreasing. Therefore, there is at least one straight pin p such that $i \prec p \prec j$. Suppose there is no straight pin between i' and j. This means $j \in r_{i'}$. According to Lemma 13, we have $i \prec j' \prec i'$ which contradicts $i' \prec j'$. Therefore, there exists at least one straight pin q such that $j \prec q \prec i'$. Then we have $i \prec pp' \prec j \prec qq' \prec i' \prec j'$. Notice the fact that the exits form an increasing sequence (Lemma 14); we have $p < q < i < j$. So the pin sequence contains four pins $p < q < i < j$ such that $i \prec p \prec j \prec q$. This contradicts our assumption.

2. $i' \prec j' \prec i \prec j$: This case is symmetrical to the previous situation and can be shown to have contradiction by similar argument.

3. $j \prec i' \prec j' \prec i$: We know $r_j \neq r_{i'}$ because otherwise we will have $(j' \prec i') \vee (i \prec j')$ (Lemma 13), which contradicts $i' \prec j' \prec i$. Therefore, there is at least one straight pin p such that $j \prec p \prec i'$. By similar argument, we know that there is at least one straight pin q such that $j' \prec q \prec i$. We then have $j \prec pp' \prec i' \prec j' \prec qq' \prec i$. According to Lemma 14, we have $p < i < j < q$. So the pin sequence contains four pins $p < i < j < q$ such that $j \prec p \prec q \prec i$. This contradicts our assumption.

4. $i' \prec j \prec i \prec j'$: By similar argument as above, we know that there is at least one straight pin p such that $i' \prec p \prec j$ and at least one straight



(a) Max #wires = 3 (b) Max #wires = 2

Figure A.4: The algorithm in this proof produces a solution (a) with three wires between 3 and 1 while another solution (b) has at most two wires between them.

pin q such that $i \prec q \prec j'$. Then we have $i' \prec pp' \prec j \prec i \prec qq' \prec j'$. According to Lemma 14, we have $i < p < q < j$. So the pin sequence contains four pins $i < p < q < j$ such that $p \prec j \prec i \prec q$. This contradicts our assumption.

All four cases lead to contradictions. Therefore, none of them should happen in our solution sequence. □

Notice that although the algorithm presented here guarantees to find a feasible topology to an SDU problem if one exists, it has no control over the number of wires passing between adjacent pins. For example, if the pin sequence is $(6, 5, 4, 2, 3, 1)$, then it will construct a solution sequence $(00', 1', 6, 5, 4, 22', 33', 4', 5', 6', 1, 77')$. The corresponding routing is shown in Figure A.4 (a). Notice that virtual pins 0 and 7 are ignored in the figure. It can be seen that there are as many as three wires between 3 and 1. However, another solution, Figure A.4 (b), has at most two wires between any two pins. Therefore, we still need the dynamic programming algorithm if we take capacity and other objectives into account.

REFERENCES

[1] Fujitsu Microelectronics Limited, "IC package," 2009. [Online]. Available: www.fujitsu.com/downloads/MICRO/fma/pdf/a810000113e.pdf

[2] L. W. Ritchey, "Busses: What are they and how do they work?" *Printed Circuit Design Magazine*, Dec. 2000. [Online]. Available: http://www.speedingedge.com/PDF-Files/busses.pdf

[3] L. W. Ritchey and J. Zasio, *Right the First Time, A Practical Handbook on High Speed PCB and System Design*, K. J. Knack, Ed. Glen Ellen, CA: Speeding Edge, 2003.

[4] D. Brooks, *Signal Integrity Issues and Printed Circuit Board Design*. Upper Saddle River, NJ: Prentice Hall, 2003.

[5] K. Mitzner, *Complete PCB Design Using OrCAD Capture and PCB Editor*. Burlington, MA: Newnes, 2009.

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

[7] T. Yan and M. D. F. Wong, "A correct network flow model for escape routing," in *Proc. Design Automation Conf.*, 2009, pp. 332–335.

[8] T. Yan and M. D. F. Wong, "BSG-route: A length-matching router for general topology," in *Proc. Int. Conf. on Computer-Aided Design*, 2008, pp. 499–505.

[9] T. Yan, P.-C. Wu, Q. Ma, and M. D. F. Wong, "On the escape routing of differential pairs," in *Proc. Int. Conf. on Computer-Aided Design*, 2010.

[10] T. Yan and M. D. F. Wong, "BSG-route: A length-constrained routing scheme for general planar topology," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 28, no. 11, pp. 1679–1690, Nov. 2009.

[11] T. Yan and M. D. F. Wong, "Untangling twisted nets for bus routing," in *Proc. Int. Conf. on Computer-Aided Design*, 2007, pp. 396–400.

[12] T. Yan and M. D. F. Wong, "Theories and algorithms on single-detour routing for untangling twisted bus," *ACM Trans. Design Autom. Electr. Syst.*, vol. 14, no. 3, pp. 1–21, 2009.

[13] T. Yan, H. Kong, and M. D. F. Wong, "Optimal layer assignment for escape routing of buses," in *Proc. Int. Conf. on Computer-Aided Design*, 2009, pp. 245–248.

[14] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani, "Module packing based on the BSG-structure and IC layout applications," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 17, no. 6, pp. 519–530, June 1998.

[15] M. M. Ozdal and M. D. F. Wong, "A length-matching routing algorithm for high-performance printed circuit boards," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 25, no. 12, pp. 2784–2794, Dec. 2006.

[16] H. Kong, T. Yan, M. D. F. Wong, and M. M. Ozdal, "Optimal bus sequencing for escape routing in dense PCBs," in *Proc. Int. Conf. on Computer-Aided Design*, 2007, pp. 390–395.

[17] 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 Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 26, no. 8, pp. 1417–1429, Aug. 2007.

[18] J.-W. Fang and Y.-W. Chang, "Area-I/O flip-chip routing for chip-package co-design," in *Proc. Int. Conf. on Computer-Aided Design*, 2008, pp. 518–522.

[19] W.-T. Chan and F. Y. L. Chin, "Efficient algorithms for finding the maximum number of disjoint paths in grids," *J. Algorithms*, vol. 34, no. 2, pp. 337–369, 2000.

[20] J.-W. Fang, I.-J. Lin, P.-H. Yuh, Y.-W. Chang, and J.-H. Wang, "A routing algorithm for flip-chip design," in *Proc. Int. Conf. on Computer-Aided Design*, 2005, pp. 753–758.

[21] R. Wang, R. Shi, and C.-K. Cheng, "Layer minimization of escape routing in area array packaging," in *Proc. Int. Conf. on Computer-Aided Design*, 2006, pp. 815–819.

[22] M.-F. Yu and W. W.-M. Dai, "Single-layer fanout routing and routability analysis for ball grid arrays," in *Proc. Int. Conf. on Computer-Aided Design*, 1995, pp. 581–586.

[23] D. Wang, P. Zhang, C.-K. Cheng, and A. Sen, "A performance-driven I/O pin routing algorithm," in *Proc. Asia and South Pacific Design Automation Conf.*, 1999, pp. 129–132.

[24] M.-F. Yu, J. Darnauer, and W. W.-M. Dai, "Interchangeable pin routing with application to package layout," in *Proc. Int. Conf. on Computer-Aided Design*, 1996, pp. 668–673.

[25] W.-T. Chan, F. Y. L. Chin, and H.-F. Ting, "A faster algorithm for finding disjoint paths in grids," in *Proc. Int. Symp. on Algorithms and Computation*, 1999, pp. 393–402.

[26] M.-F. Yu and W. W.-M. Dai, "Pin assignment and routing on a single-layer pin grid array," in *Proc. Asia and South Pacific Design Automation Conf.*, 1995, pp. 203–208.

[27] J.-W. Fang, C.-H. Hsu, and Y.-W. Chang, "An integer linear programming based routing algorithm for flip-chip design," in *Proc. Design Automation Conf.*, 2007, pp. 606–611.

[28] Y. Kubo and A. Takahashi, "Global routing by iterative improvements for two-layer ball grid array packages," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 25, no. 4, pp. 725–733, Apr. 2006.

[29] Y. Kubo and A. Takahashi, "A global routing method for 2-layer ball grid array packages," in *Proc. Int. Symp. on Physical Design*, 2005, pp. 36–43.

[30] Y. Tomioka and A. Takahashi, "Monotonic parallel and orthogonal routing for single-layer ball grid array packages," in *Proc. Asia and South Pacific Design Automation Conf.*, 2006, pp. 642–647.

[31] L. Luo and M. D. F. Wong, "Ordered escape routing based on Boolean satisfiability," in *Proc. Asia and South Pacific Design Automation Conf.*, 2008, pp. 244–249.

[32] M. M. Ozdal and M. D. F. Wong, "Simultaneous escape routing and layer assignment for dense PCBs," in *Proc. Int. Conf. on Computer-Aided Design*, 2004, pp. 822–829.

[33] M. M. Ozdal, M. D. F. Wong, and P. S. Honsinger, "Simultaneous escape-routing algorithms for via minimization of high-speed boards," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 27, no. 1, pp. 84–95, Jan. 2008.

[34] Q. Ma, T. Yan, and M. D. F. Wong, "A negotiated congestion based router for simultaneous escape routing," in *Proc. Int. Symp. on Qual. Electron. Design*, 2010, pp. 606–610.

[35] L. Luo, T. Yan, Q. Ma, M. D. F. Wong, and T. Shibuya, "B-Escape: A simultaneous escape routing algorithm based on boundary routing," in *Proc. Int. Symp. on Physical Design*, 2010, pp. 19–25.

[36] C. E. Leiserson and F. M. Maley, "Algorithms for routing and testing routability of planar VLSI layouts," in *Proc. Annu. Symp. on Theory of Computing*, 1985, pp. 69–78.

[37] W. W.-M. Dai, R. Kong, and M. Sato, "Routability of a rubber-band sketch," in *Proc. Design Automation Conf.*, 1991, pp. 45–48.

[38] D. J. Staepelaere, "Geometric transformations for a rubber-band sketch," M.S. thesis, University of California at Santa Cruz, Santa Cruz, CA, USA, Sep. 1992.

[39] D. Staepelaere, J. Jue, T. Dayan, and W. W.-M. Dai, "SURF: Rubber-band routing system for multichip modules," *IEEE Des. Test. Comput.*, vol. 10, no. 4, pp. 18–26, Dec. 1993.

[40] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Upper Saddle River, NJ: Prentice Hall, 1993.

[41] "CS2: min-cost flow solver," 1997. [Online]. Available: http://www.igsystems.com/cs2/index.html

[42] J.-W. Fang, K.-H. Ho, and Y.-W. Chang, "Routing for chip-package-board co-design considering differential pairs," in *Proc. Int. Conf. on Computer-Aided Design*, 2008, pp. 512–517.

[43] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Trans. Electron. Comput.*, vol. EC-10, no. 2, pp. 364–365, Sep. 1961.

[44] J. Soukup, "Fast maze router," in *Proc. Design Automation Conf.*, 1978, pp. 100–102.

[45] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for FPGAs," in *Proc. Int. Symp. on Field-Programmable Gate Arrays*, 1995, pp. 111–117.

[46] M. M. Ozdal and M. D. F. Wong, "Algorithmic study of single-layer bus routing for high-speed boards," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 25, no. 3, pp. 490–503, Mar. 2006.

[47] Y. Kubo, H. Miyashita, Y. Kajitani, and K. Takeishi, "Equidistance routing in high-speed VLSI layout design," *Integration, the VLSI Journal*, vol. 38, no. 3, pp. 439–449, Jan. 2005.

[48] N. Fu, S. Nakatake, Y. Takashima, and Y. Kajitani, "The oct-touched tile: A new architecture for shape-based routing," *IEICE Trans. Fundamentals Electron., Commun. and Comput. Sci.*, vol. 89, no. 2, pp. 448–455, 2006.

[49] A. Nedich, private communication, 2008.

[50] "lp_solve: Open source (mixed-integer) linear programming system," 1999. [Online]. Available: http://sourceforge.net/projects/lpsolve

[51] M. M. Ozdal and M. D. F. Wong, "Algorithms for simultaneous escape routing and layer assignment of dense PCBs," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 25, no. 8, pp. 1510–1522, Aug. 2006.

[52] R. R. Tummala, E. J. Rymaszewski, and A. G. Klopfenstein, *Microelectronics Packaging Handbook, Part II: Semiconductor Packaging.* Norwell, MA: Kluwer Academic Publishers, 1997.

[53] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA: The MIT Press, 2001.

[54] M. Mucha and P. Sankowski, "Maximum matchings via gaussian elimination," in *Proc. Annu. Symp. on Foundations of Comput. Sci.*, 2004, pp. 248–255.

[55] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, Eds. New York, NY: Plenum Press, 1972, pp. 85–103.

[56] B. V. Cherkassky and A. V. Goldberg, "On implementing push-relabel method for the maximum flow problem," *Algorithmica*, vol. 19, no. 4, pp. 390–410, 1997.

[57] C. F. Coombs, Ed., *Printed Circuits Handbook*, 6th ed. New York, NY: McGraw-Hill, 2007.

[58] S. S. Skiena, *The Algorithm Design Manual*, 2nd ed. London, United Kingdom: Springer, 2008.

AUTHOR'S BIOGRAPHY

Tan Yan received the B.S. degree in computer science and technology from Fudan University, Shanghai, China, in 2003, and the M.Eng. in information engineering from The University of Kitakyushu, Kitakyushu, Japan, in 2005. He is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering at the University of Illinois. His research interests include combinatorial algorithms with applications in the computer-aided design of integrated circuits. Currently, his major research focus is in printed circuit board (PCB) routing. After the completion of his Ph.D., he will join the detail routing team of Synopsys., Inc.