DIGITAL CMOS VLSI CIRCUIT ROUTING WITH MANUFACTURABILITY AND
YIELD CONSTRAINTS

A DISSERTATION SUBMITTED TO THE GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAI’I AT MĀNOA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

ELECTRICAL ENGINEERING

MAY 2012

By

XIN GAO

Dissertation Committee:

Gary S. Varner, Chairperson
Luca Macchiarulo
Victor M. Lubecke
Galen Sasaki
James Yee
Nancy Reed
We certify that we have read this dissertation and that, in our opinion, it is satisfactory in scope and quality as a dissertation for the degree of Doctor of Philosophy in Electrical Engineering.

DISSERTATION COMMITTEE

[Signatures]

Chairperson
To my parents,

Yuwen Gao and Lihua Xin
Acknowledgments

First, I would like to thank Dr. Luca Macchiarulo for reviewing and giving valuable comments and suggestions to my papers. I also would like to thank him for helping doing the presentations in the conferences. Without his guidance and generous help, the project in the thesis would not be possible.

I would like to thank Dr. Victor M. Lubecke, Dr. Gary Varner, Dr. Galen Sasaki, Dr. James Yee, and Dr. Nacy Reed, for serving on my dissertation committee and helping me modifying this dissertation.

Finally, I wish to thank my parents for their continuous support and encouragement.
Abstract

As the minimum feature size enters nano scale, the IC industry faces great challenges from the yield losses caused by the various imperfections in the manufacturing processes. As the CMOS technology is approaching its fundamental limit, completely relying on the manufacturers to solve the yield problems is not practical. Indeed, the coupling between manufacturing and design at the advanced technology nodes has become so strong that, to achieve a decent yield, the designers must be aware of the impact of the design style on the manufacturing of the circuits.

In this dissertation, the possibilities of optimizing manufacturability and yield in the stage of routing are investigated. The dissertation mainly focuses on problem formulations and algorithms rather than modeling. Since there are so many potential problems involved in the topic, the dissertation is not intended to be complete. Rather, several interesting formulations and algorithms for the optimization of manufacturability and yield are proposed.

In chapter 1, some basic concepts about routing and design for manufacturability and yield are introduced. Then, the research objective is presented in chapter 2.

In chapter 3, a track routing algorithm for the co-optimization of timing and yield is proposed. In the work it is shown that, for a given segment ordering, the optimization problem can be formulated as a mixed linear geometric programming problem, which can be transformed into a general non-linear convex programming problem. Therefore, theoretically, the co-optimization problem can be solved in polynomial time. However, since the practical running time of the optimal solver for the general convex programming problem is too long, a heuristic is proposed to return a decent solution within much shorter running time.

In chapter 4, two techniques are proposed to enhance the double-patterning-friendly detailed routing algorithm. One of the two techniques takes advantage of the idea of delayed decision to make the coloring choices more reasonable. The other technique tries to get rid of the within-path coloring conflicts by recording more information during path searching.

In chapter 5, a jumper insertion algorithm that targets practical antenna rules and is aware of timing constraints is proposed. The problem is first formulated as an integer programming prob-
lem. Then, the technique of Lagrangian relaxation is employed to relax the difficult constraints and convert the original problem into two sub-problems that can be solved efficiently with combinatorial algorithms.

In chapter 6, an algorithm is proposed that combines the wire sizing and layer reassignment techniques together to improve the yield of the circuits. It is shown that, even with its NP-completeness, the problem can be solved optimally by using the technique of tree decomposition. To make the running time practical, a heuristic is proposed to construct the largest subgraphs with bounded treewidth. To take advantage of multi-core CPUs, the dynamic programming algorithm is implemented with multiple threads. Finally, it is pointed out that the proposed algorithms can also be used to solve some problems in multiple patterning technologies.

In the last chapter, the dissertation is concluded and some possible future research directions are presented.
# Table of Contents

Acknowledgments ............................................................... v  
Abstract ........................................................................ vi  
List of Tables ..................................................................... xi  
List of Figures ..................................................................... xiii  
1 Introduction ...................................................................... 1  
  1.1 Design for Manufacturability and Yield ......................... 1  
  1.2 Routing ...................................................................... 3  
  1.3 Routing and DFM&Y .................................................... 4  
    1.3.1 Random Defects ................................................... 5  
    1.3.2 Electroplating (ECP) and Chemical Mechanical Polishing (CMP) ........................................... 6  
    1.3.3 Lithograph .......................................................... 7  
    1.3.4 Via .................................................................... 10  
    1.3.5 Antenna Effect .................................................... 10  
  1.4 Conclusions .................................................................. 11  
2 Research Objective and Previous Work .............................. 12  
  2.1 Research Objective ..................................................... 12  
  2.2 Previous Work ........................................................... 12  
  2.3 Organization of the Rest of the Dissertation ................... 13  
  2.4 Conclusions .................................................................. 14  
3 Track Routing Optimizing Timing and Yield ....................... 15  
  3.1 Introduction .................................................................. 15  
  3.2 Preliminaries ................................................................ 19  
    3.2.1 Notation ............................................................. 19  
    3.2.2 Models .............................................................. 19  
  3.3 Problem Formulation ................................................... 22  
  3.4 Algorithm ................................................................... 24  
    3.4.1 Basic Framework ............................................... 24  
    3.4.2 Wire Segment Ordering ....................................... 25  
    3.4.3 Wire Positioning and Sizing for Yield and Timing ................................................................. 29  
  3.5 Experimental Results .................................................. 36  
  3.6 Conclusions .................................................................. 37  
4 Enhancing Double-Patterning Detailed Routing with Lazy Coloring and Within-Path Conflict Avoidance ................................................ 39  
  4.1 Introduction .................................................................. 39  
  4.2 Background .................................................................. 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3 Lazy Color Decision</td>
<td>45</td>
</tr>
<tr>
<td>4.4 Last Conflict Segment Recording</td>
<td>50</td>
</tr>
<tr>
<td>4.5 Experimental Results</td>
<td>52</td>
</tr>
<tr>
<td>4.6 Conclusions and Future Work</td>
<td>53</td>
</tr>
<tr>
<td>5 Jumper Insertion under Antenna Ratio and Timing Constraints</td>
<td>54</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>54</td>
</tr>
<tr>
<td>5.2 Preliminaries</td>
<td>58</td>
</tr>
<tr>
<td>5.3 Problem Formulation</td>
<td>58</td>
</tr>
<tr>
<td>5.3.1 Candidate location selection</td>
<td>59</td>
</tr>
<tr>
<td>5.3.2 Jumper library</td>
<td>59</td>
</tr>
<tr>
<td>5.3.3 An ILP Formulation</td>
<td>61</td>
</tr>
<tr>
<td>5.4 Algorithm</td>
<td>64</td>
</tr>
<tr>
<td>5.4.1 Overview</td>
<td>64</td>
</tr>
<tr>
<td>5.4.2 The classical Lagrangian relaxation method</td>
<td>64</td>
</tr>
<tr>
<td>5.4.3 The algorithm for the problem in Fig. 5.10</td>
<td>67</td>
</tr>
<tr>
<td>5.4.4 The algorithm for the problem in Fig. 5.11</td>
<td>73</td>
</tr>
<tr>
<td>5.4.5 The augmented Lagrangian relaxation method</td>
<td>76</td>
</tr>
<tr>
<td>5.4.6 Running time analysis</td>
<td>78</td>
</tr>
<tr>
<td>5.5 Experimental Results</td>
<td>78</td>
</tr>
<tr>
<td>5.5.1 Non-cumulative Antenna Ratio</td>
<td>79</td>
</tr>
<tr>
<td>5.5.2 Cumulative Antenna Ratio</td>
<td>81</td>
</tr>
<tr>
<td>5.6 Conclusions</td>
<td>82</td>
</tr>
<tr>
<td>6 Simultaneous Layer Reassignment and Wire Sizing for Yield Enhancement Based on Tree Decomposition and Multi-threaded Dynamic Programming</td>
<td>83</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>83</td>
</tr>
<tr>
<td>6.2 Overview</td>
<td>85</td>
</tr>
<tr>
<td>6.3 ECC Graph Model</td>
<td>86</td>
</tr>
<tr>
<td>6.3.1 Topology Description</td>
<td>87</td>
</tr>
<tr>
<td>6.3.2 Cost Definition</td>
<td>88</td>
</tr>
<tr>
<td>6.3.3 Graph Simplification</td>
<td>90</td>
</tr>
<tr>
<td>6.4 Algorithm</td>
<td>91</td>
</tr>
<tr>
<td>6.4.1 Preliminaries</td>
<td>91</td>
</tr>
<tr>
<td>6.4.2 General Description</td>
<td>93</td>
</tr>
<tr>
<td>6.4.3 Heuristic for Identifying the Largest Subgraph with Bounded Treewidth</td>
<td>94</td>
</tr>
<tr>
<td>6.4.4 Dynamic Programming</td>
<td>106</td>
</tr>
<tr>
<td>6.4.5 Speed up for Large Scale Layouts</td>
<td>109</td>
</tr>
<tr>
<td>6.5 Experimental Results</td>
<td>110</td>
</tr>
<tr>
<td>6.6 Applying the Proposed Algorithm to Multiple-Patterning Technologies</td>
<td>114</td>
</tr>
<tr>
<td>6.6.1 Problem 6.1</td>
<td>114</td>
</tr>
<tr>
<td>6.6.2 Problem 6.2</td>
<td>117</td>
</tr>
<tr>
<td>6.7 Conclusions and Future Work</td>
<td>119</td>
</tr>
<tr>
<td>7 Conclusions and Future Work</td>
<td>120</td>
</tr>
<tr>
<td>7.1 Conclusions</td>
<td>120</td>
</tr>
<tr>
<td>7.2 Future Work</td>
<td>121</td>
</tr>
<tr>
<td>7.2.1 Track Routing</td>
<td>121</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>7.2.2 Double-Patterning-Friendly Detailed Routing</td>
<td>122</td>
</tr>
<tr>
<td>7.2.3 Jumper Insertion</td>
<td>122</td>
</tr>
<tr>
<td>7.2.4 Simultaneously Layer Reassignment and Wire Sizing</td>
<td>122</td>
</tr>
<tr>
<td>A A Summary of the Existing Work</td>
<td>123</td>
</tr>
<tr>
<td>Bibliography</td>
<td>128</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 A summary of notation.</td>
<td>18</td>
</tr>
<tr>
<td>3.2 The number of nets in the benchmark circuits.</td>
<td>37</td>
</tr>
<tr>
<td>3.3 Comparison between the optimal solution and the solution from our heuristic.</td>
<td>38</td>
</tr>
<tr>
<td>3.4 Comparison between the solutions for different scenarios.</td>
<td>38</td>
</tr>
<tr>
<td>4.1 Grid state for DPT coloring.</td>
<td>43</td>
</tr>
<tr>
<td>4.2 Experimental results with “K” in the LCD algorithm set to 5.</td>
<td>52</td>
</tr>
<tr>
<td>5.1 A summary of the notations used in the ILP formulation in Fig. 5.7.</td>
<td>62</td>
</tr>
<tr>
<td>5.2 Benchmark information when using non-cumulative antenna ratio.</td>
<td>79</td>
</tr>
<tr>
<td>5.3 Experimental results when only optimizing the number of vias under non-cumulative antenna ratio.</td>
<td>79</td>
</tr>
<tr>
<td>5.4 Technology parameters.</td>
<td>80</td>
</tr>
<tr>
<td>5.5 Experimental results when optimizing both the number of vias and timing under non-cumulative antenna ratio.</td>
<td>80</td>
</tr>
<tr>
<td>5.6 Benchmark information when using non-cumulative antenna ratio.</td>
<td>81</td>
</tr>
<tr>
<td>5.7 Experimental results when only optimizing the number of vias under cumulative antenna ratio.</td>
<td>81</td>
</tr>
<tr>
<td>5.8 Experimental results when optimizing both the number of vias and timing under cumulative antenna ratio.</td>
<td>82</td>
</tr>
<tr>
<td>6.1 Information about the benchmark circuits.</td>
<td>111</td>
</tr>
<tr>
<td>6.2 Results of the first experiment on the effect of performing LA and WS simultaneously.</td>
<td>111</td>
</tr>
<tr>
<td>6.3 Results of the second experiment on the effect of treewidth upper bound.</td>
<td>112</td>
</tr>
<tr>
<td>6.4 A comparison of single-thread and multi-thread running times (in secs).</td>
<td>113</td>
</tr>
<tr>
<td>A.1 Global routing</td>
<td>123</td>
</tr>
<tr>
<td>A.2 Layer assignment (a)</td>
<td>123</td>
</tr>
<tr>
<td>A.3 Layer assignment (b)</td>
<td>124</td>
</tr>
<tr>
<td>A.4 Track routing</td>
<td>124</td>
</tr>
<tr>
<td>A.5 Detailed routing (a)</td>
<td>125</td>
</tr>
<tr>
<td>A.6 Detailed routing (b)</td>
<td>126</td>
</tr>
<tr>
<td>A.7 Post-routing (redundant via insertion)</td>
<td>126</td>
</tr>
<tr>
<td>A.8 Post-routing (Wire sizing, spreading and layout compaction)</td>
<td>127</td>
</tr>
</tbody>
</table>
A.9 Post-routing (jumper insertion) ............................................. 127
A.10 Post-routing (diode insertion) ............................................. 127
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A routing flow example.</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Two examples of hard errors caused by random defects.</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Two examples of soft errors caused by random defects.</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>CMP induced variations.</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>An example of poor image fidelity in nano-lithography.</td>
<td>6</td>
</tr>
<tr>
<td>1.6</td>
<td>An example of OPC and SRAF.</td>
<td>8</td>
</tr>
<tr>
<td>1.7</td>
<td>An example of OAI.</td>
<td>8</td>
</tr>
<tr>
<td>1.8</td>
<td>An illustration of the phase-shift mask technology.</td>
<td>9</td>
</tr>
<tr>
<td>1.9</td>
<td>An example of the conflicts in layout decomposition for PSM.</td>
<td>9</td>
</tr>
<tr>
<td>1.10</td>
<td>An example flow of the double-exposure-double-etch technology.</td>
<td>9</td>
</tr>
<tr>
<td>1.11</td>
<td>An example of layout decomposition in double patterning technology.</td>
<td>10</td>
</tr>
<tr>
<td>1.12</td>
<td>Example via structures.</td>
<td>10</td>
</tr>
<tr>
<td>1.13</td>
<td>An illustration of antenna effect.</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>An overall description of the research objective.</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>An example of track routing.</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>Some examples for illustrating the notation used in the problem formulation.</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>An example of Elmore delay.</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Formulation of the track routing problem for the optimization of timing and yield.</td>
<td>23</td>
</tr>
<tr>
<td>3.5</td>
<td>An overview of the algorithm.</td>
<td>24</td>
</tr>
<tr>
<td>3.6</td>
<td>An illustration of the edge weight function.</td>
<td>26</td>
</tr>
<tr>
<td>3.7</td>
<td>An example of wire segment ordering.</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>DPT decomposes the layout to relax pitch size.</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>Some possibilities of pitch reduction in double patterning.</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>An example of modeling the mask assignment problem as a coloring problem.</td>
<td>41</td>
</tr>
<tr>
<td>4.4</td>
<td>The basic flow of the algorithm in [1].</td>
<td>43</td>
</tr>
<tr>
<td>4.5</td>
<td>An example of the double-patterning-friendly detailed routing algorithm in [1].</td>
<td>44</td>
</tr>
<tr>
<td>4.6</td>
<td>An example for illustrating the impact of $BG$ grids coloring on routing.</td>
<td>45</td>
</tr>
<tr>
<td>4.7</td>
<td>Application of LCD to the example in Fig. 4.6.</td>
<td>47</td>
</tr>
<tr>
<td>4.8</td>
<td>Avoiding conflict during path searching.</td>
<td>51</td>
</tr>
<tr>
<td>4.9</td>
<td>Conflict from detours.</td>
<td>51</td>
</tr>
</tbody>
</table>
5.1 An illustration of the cumulative and non-cumulative antenna rules. ................................................................. 55
5.2 An illustration of the difference between antenna ratio and antenna size. .............................................................. 56
5.3 An example for illustrating the coupling between subtrees. .................................................................................. 56
5.4 An example of candidate location selection. ........................................................................................................ 59
5.5 Problem formulation. ........................................................................................................................................... 60
5.6 An example of using detailed routing tools to generate jumpers. ........................................................................ 60
5.7 An ILP formulation of the jumper insertion problem. ......................................................................................... 61
5.8 An example for illustrating the compatibility between the jumper insertion solutions of different subtrees. ......................................................................................................................... 63
5.9 The problem obtained by applying CLR on the problem in Fig. 5.7. ................................................................. 64
5.10 The subproblem of Fig. 5.9 with respect to $s_{p,i}$. ............................................................................................. 65
5.11 The subproblem of Fig. 5.9 with respect to $c_{q,j}$. ............................................................................................. 66
5.12 An example for illustrating the solution propagation procedure. ................................................................ 67
5.13 An explanation about the case where one subtree belongs to more than one $T_l$. ................................................... 70
5.14 The problem obtained by applying ALR on problem in Fig. 5.7. ...................................................................... 76
5.15 The subproblem of Fig. 5.14 with respect to $s_{p,i}$. ............................................................................................. 77
5.16 The subproblem of Fig. 5.14 with respect to $c_{q,j}$. ............................................................................................. 78
6.1 An example for the overall flow. ............................................................................................................................ 86
6.2 An example of the graph model. ........................................................................................................................... 87
6.3 Edge cost functions: (a) is for the connection edges; (b) is for the others. .............................................................. 89
6.4 An example of graph simplification. ..................................................................................................................... 91
6.5 An example of tree decomposition. ..................................................................................................................... 92
6.6 An example for the heuristic in Algorithm 6.1. ........................................................................................................ 97
6.7 An example of tree-decomposition simplification with the algorithm in [2]. .......................................................... 102
6.8 An example of dynamic programming. ................................................................................................................ 105
6.9 Impact of root selection on the depth of tree. ........................................................................................................ 107
6.10 Tree topology enhancement. ............................................................................................................................... 108
6.11 An example for illustrating the speed-up technique that slices the layout into overlapping windows. .................................................. ................................................................. 109
6.12 An illustration of the dependence of fault reduction on circuit density. ............................................................ 113
6.13 An example of layout decomposition in multiple-patterning technology. ........................................................... 115
6.14 An example of using layer reassignment in multiple-patterning technology. ......................................................... 118
Chapter 1

Introduction

As the IC industry enters the nano era, the problem of yield loss is exacerbated due to the complexity and difficulty of the manufacturing process and the prominent impact of design on manufacturing. To combat this problem, the concept of Design for Manufacturability and Yield (DFM&Y) has been proposed and employed in the circuit design practise to reduce the manufacturing cost and improve the overall yield and profit. Routing, as a stage of layout design, has a great potential to reduce the yield loss caused by the Back-End-Of-Line (BEOL) defects (i.e., the defects on metal wires). In the first part of this chapter, the concept of DFM&Y and its motivation are briefly introduced. Then, some basic knowledge about routing is presented. Finally, the impact of routing on manufacturing and yield is illustrated by some examples.

1.1 Design for Manufacturability and Yield

Design for manufacturability (DFM) in its broad definition stands for the methodology of ensuring that a product can be manufactured repeatedly, consistently, reliably, and cost effectively by taking all the measures needed for that goal starting at the concept stage of a design and implementing these measures throughout the design flow [3].

For the IC industry, the ultimate motivation of employing DFM&Y in the design flow is profit. In nano-scale technology, the overall circuit performance becomes much more sensitive to the variations in the manufacturing process. This makes the precise control of the manufacturing process a very challenging task. For example, for a MOS transistor with a gate length of 32 nm, a 3 nm variation budget means almost 10% change in the gate length. This variation in gate length will be reflected in the final circuit as variations in speed and power consumption. What is more unfortunate is that this small 3 nm budget has to be shared by all the possible variation sources such as lithography, etching, stress engineering, mask error and so on. As another example, in
a typical 65-nm technology the thickness of the gate dielectric film is on the order of four atom layers. However, for the current manufacturing technology, precisely placing four atom layers on all the transistor regions across the whole wafer is simply impossible.

As a result of the high precision requirement from the nano-scale circuits and the relatively poor control from the current manufacturing technology, the yield loss in the initial manufacturing is increasing. In 180-nm technology the first-round design success rate was around 80%, which then declined to 60% in 130-nm technology and continued downward since then [4]. What makes things worse is that, contrary to the declining first-time design success rate, the cost of product design rockets as the technology node moves into the nano scale. While being around $2-million at 250-nm node, the design cost is estimated at $75 million at 32-nm node [4]. Low initial yield is expensive not only because of the cost of circuit redesign, mask modification and re-spin but also due to the delay to market which may significantly reduce the life cycle of the product. Therefore, there is a strong desire in current IC industry to improve the manufacturability and yield of the circuits.

One important observation in nano-scale technology is that the yield loss caused by the design-related problems becomes prominent. As the feature size continues scaling down, the image quality on silicon suffers more from the proximity effect in lithography. As a result, the printed image of a polygon is interfered with other polygons in its neighborhood. Consequently, the printability of a circuit heavily depends on the layout design style. Other design-related manufacturing problems include, but are not limited to, the via and wire etch rate variation due to the micro and macro loading effect and the topography variation in the CMP (Chemical Mechanical Polishing) process, both of which are dependent on the pattern density distribution in the layout. Another source of design-manufacturing interaction, that is exacerbated these days, is the stress engineering technique, where a tensile or compressive stress is applied to the transistor channel to improve the mobility of the carriers (i.e., holes and electrons). The problem with the stress engineering technique is that the stress on the transistor channel is modulated by the volume and distance of the stress film to the channel. This makes the electrical characteristic of the transistors sensitive to the surrounding layout structures, e.g., poly, contact, shallow trench isolation, embedded silicon-germanium, and so on. This sensitivity poses new challenges to layout simulation and design.

The impact of design on manufacturing is reflected not only in the yield loss but also the cost of the mask used in lithography. In nano-scale technology, the mask cost has reached a level of multi-million dollars (estimated at $3 million for 32 nm node). This high cost is jeopardizing system-on-chip innovation from start-up companies and emerging market institutions. The biggest
contribution to the mask cost is from the mask write step, whose high cost is due to the expensive mask writer (several tens of millions) and the long mask writing time. Since the mask writing time is dependent on the complexity of the polygons in the layout, which is, in turn, dependent on the layout design style and the layout enhancement techniques, the circuit designers have the potential to reduce the mask cost by placing the polygons in the layout in a manufacturing-friendly manner.

In conclusion, DFM&Y has become a well-established trend in the IC industry. To apply DFM&Y to the design flow, the EDA (Electronic Design Automation) tool vendors have started developing and releasing the design tools and methodologies that can help analyze and enhance the manufacturability and yield of the circuits. Indeed, nowadays only optimizing the traditional circuit metrics such as timing, power, area and noise is no longer enough. The circuit designers are expected to be aware of the manufacturability and yield problems of the circuits. For a more detailed description of DFM&Y, please refer to [3, 4, 5, 6].

1.2 Routing

Routing is a key step in the physical design stage in the VLSI design flow. It is responsible for connecting the basic circuit blocks as specified in the netlist provided by the circuit designer while satisfying all the design rules provided by the circuit manufacturer. Nowadays, due to the popularity of system-on-chip (SOC), it is not unusual to have circuits with millions of nets to route. This poses stringent requirements of the running time of the routing algorithms. Furthermore, as the minimum feature size of the circuits continues scaling down, the impact of routing on circuit performance becomes more significant. In particular, the clock cycles in current circuits are mainly dominated by interconnection delay. Besides, as the spacing between the wires is shrinking, the coupling between neighboring wires becomes stronger, which aggravates the signal integrity problem of the circuits. Another problem that routing plays an important role in is manufacturability and yield, since routing determines how the metal patterns are distributed in the layout. This problem will be illustrated in more detail in Section 1.3.

Because of the above reasons, routing has become a very complex step in current VLSI design flow and is further divided into several sub-steps to improve its tractability. A typical routing flow is shown in Fig. 1.1.

Generally speaking, there are four major steps in routing, including global routing, layer assignment, detailed routing, and post-routing modification. The first step is global routing, which is used to determine the approximate connection paths of the nets. In this step, the multi-layer
layout is first sliced into tiles. Then, a three dimensional grid graph is built by abstracting the tiles into grid points. To improve the efficiency of the global routing algorithms, the 3D grid graph is usually projected onto a 2D plane, where the global routing algorithms are then applied to find the connection topologies of the nets. After that, layer assignment is performed to map the obtained 2D global routing paths back to the 3D routing grid graph. At this point, the regions where the wires should be placed have been approximately determined. Next, in detailed routing, the exact geometric positions and shapes of the wires are determined to satisfy the design rules. The last step is post-routing modification, where the connection structures are modified to further improve the quality of the routing solution. For example, in Fig. 1.1 a redundant via is added in the last routing step to improve the yield of the net.

It is worth mentioning that the routing flow presented in this section is based on the traditional flow that is widely used in design practice. There is another routing framework called multilevel routing (e.g., [7]), whose basic idea is from the multilevel optimization methodology widely used in solving integral and differential equations. However, the atomic operations in the multilevel routing framework are similar to the basic steps in the traditional routing flow. Therefore, in this dissertation only the flow in Fig. 1.1 is focused on.

1.3 Routing and DFM&Y

In this section, the impacts of routing on manufacturing and yield are briefly illustrated by some examples. For more detailed descriptions, please refer to [3] and [4].
1.3.1 Random Defects

In the fabrication processes such as lithography, etching, deposition, cleaning and CMP, contaminating particles may fall on the chip surface and cause interruptions (opens) of nets or unintended interconnections (shorts) between different nets. Fig. 1.2 shows two such examples. In Fig. 1.2a, the contaminating particle completely blocks the current path of the wire. If the particle is not conductive, an open defect will be induced. In Fig. 1.2b, a particle bridges two adjacent wires. If the particle is conductive, the two wires will be shorted. Besides causing opens and shorts, random defects may also induce parametric yield loss and reliability problems. This is illustrated in Fig. 1.3. In Fig. 1.3a, the contaminating particle is not big enough to cause an open defect. However, since the particle does block a part of the current path, the resistance of the wire becomes larger. As a result, the delay of the wire is increased and the circuit may not meet the timing requirements. Fig. 1.3b shows an example where contaminating particles cause circuit reliability problems. In the figure, the contaminating particle reduces the spacing between the adjacent wires. This reduction of wire spacing may cause the dielectric between the wires to breakdown earlier and hence tends to reduce the life time of the circuit.

Generally, the probabilities of the defects caused by contaminating particles are dependent on the particle position and size distribution, and wire widths and spacings. Intuitively, the wider a wire is, the less likely the wire will be broken by a particle. Similarly, the larger the spacing between two wires is, the less probable these two wires will be shorted. In the stage of routing, the router has the control over the spacings and widths of the metal wires, and hence the potential to reduce the probabilities of the defects induced by contaminating particles.

![Figure 1.2](image1.png)
(a) An example of open defect
(b) An example of short defect

Figure 1.2. Two examples of hard errors caused by random defects.

![Figure 1.3](image2.png)
(a)
(b)

Figure 1.3. Two examples of soft errors caused by random defects.
1.3.2 Electroplating (ECP) and Chemical Mechanical Polishing (CMP)

In current IC technology, the metal wires are typically manufactured by a dual-damascene process. Two key steps in this process are electroplating (ECP) and chemical mechanical polishing (CMP). In ECP the metal material is deposited into pre-defined trenches. Then CMP is used to planarize the surface of the chip. The problem with ECP and CMP is that topography variation may be generated due to the non-uniform distribution of the metal wires. One example is shown in Fig. 1.4. As shown in the figure, the post-ECP and post-CMP topography profiles are functions of the densities of the metal wires. This variation in the chip surface topography results in the variation of the metal resistance and reduces the depth-of-focus (DOF) margin in lithography. In practice, some dummy metal structures (i.e., metal structures that do not carry useful signals) are added to the layout to make the metal density distribution more uniform. However, the added dummy metals may degrade the circuit timing, power and signal integrity performances. Therefore, it is preferred that the metal wires are uniformly distributed in the stage of routing so that less dummy metals are needed.

![Figure 1.4. CMP induced variations.](image)

![Figure 1.5. An example of poor image fidelity in nano-lithography.](image)
1.3.3 Lithograph

In lithography, the polygons drawn in the layout are imaged and printed onto the silicon wafer. In current nano-scale technology, lithography is working in the sub-wavelength regime, since the wavelength used in exposure is much larger than the minimum feature size in the circuit. As a result, due to various optical effects, the features obtained on silicon may have poor fidelity to the polygons in the layout. One example is shown in Fig. 1.5. Note that in the example the line ends of the image printed on the silicon wafer are shortened. A lot of techniques have been proposed to improve the fidelity of the printed image, including but are not limited to Optical Proximity Correction (OPC), Sub-Resolution Assist Feature (SRAF), Off-Axis Illumination (OAI), Phase Shift Mask (PSM), immersion lithography, multiple patterning, Extreme UltraViolet lithography (EUV), maskless lithography, and imprint lithography. Among the mentioned techniques, OPC, SRAF, OAI, PSM and immersion lithography together have already been widely used in industry. Double patterning is believed to be introduced at 32-nm HP (Half Pitch) node. Whether and when EUV is going to be widely used in practice are still under debate. For reference, Intel is preparing to extend the 193-nm immersion lithography technique to 15-nm and possibly 11-nm by employing multiple-patterning technique [8].

Some of the techniques mentioned above have special requirements of the layout design style. These requirements and their impacts on routing are briefly explained in the rest of this section.

OPC and SRAF

The basic idea of OPC and SRAF is to modify the original polygons on the mask in such a way that the printed image on silicon is more faithful to the design intention. One example is shown in Fig. 1.6, where some extra structures such as serifs and hammerheads are added to the polygon on the mask. As a result, the line end shortening effect which has been observed in Fig. 1.5 is significantly suppressed. To use OPC or SRAF, there must be enough empty space around the polygons to be modified. This, in turn, requires the router to leave enough empty space around the metal patterns which are likely to suffer from severe lithography fidelity problems. Also note that the added structures by OPC and SRAF make the polygons on the mask more complex, which may increase the mask writing time and the total mask cost. Therefore, it is desirable that the router places the metal structures in the layout in such a manner that fewer OPC and SRAF structures are needed.
In OAI, the incident direction of the exposure light is tilted in such a way that more information (i.e., high frequency components) of the diffraction patterns can be collected by the optical system. One example is shown in Fig. 1.7. By using specific angle in the illumination, the imaging contrast of certain pitches can be enhanced, but with the sacrifice of the contrast of other spatial frequencies. As a result, the spacings and widths of the wires that can be well printed on the silicon wafer become discrete. This discreteness complicates the design of router.

In PSM, image resolution is improved by taking advantage of the interference generated by phase differences. This is illustrated in Fig. 1.8, where the destructive interference caused by the 180 degree phase difference is used to improve the resolution of the image. Generally, there exist two types of phase-shift masks, including alternating and attenuated phase-shift masks. Due to their simplicity, attenuated phase-shift masks have already been extensively used in industry. However, as the minimum feature size continues scaling down, the adoption of alternating phase-shift masks is becoming widespread. When applying the alternating phase-shift mask technology, the layout has to be decomposed to accommodate the two different phases. The problem is that some layouts can not be cleanly decomposed. One example is shown in Fig. 1.9. In the figure, the empty regions on the two sides of each wire are assigned to two different phases to improve the resolution. But, as shown in the figure, an assignment conflict exits in the layout. As a result, a dark line will be printed at the conflict position. This dark line can be removed by sending the silicon wafer through another exposure process. However, this additional exposure step lowers the throughput of manufacturing. Therefore, it is desirable that the router places the metal wires in a conflict-free manner.
Multiple-Patterning Technology

Multiple-Patterning Technology (MPT) is a class of techniques that achieve higher feature density by performing multiple exposures and/or multiple etchings. It includes dual tone photoresist, dual tone development, double/multiple exposure, self-aligned spacer, double-exposure-double-etch, and multiple-patterning [8]. An example of the double-exposure-double-etch process is shown in Fig. 1.10. In the figure, to achieve the feature density on “Hard mask2”, two rounds of exposures and etchings are performed. As a result, the feature density requirement in each round of exposure and etching is relaxed. Similarly to the alternating phase-shift mask technology, MPT also faces the problem of layout decomposition. Fig. 1.11 shows such an example for the double-patterning technology. In the figure, the spacing between the two branches of the U-shaped pattern is too small to be printed in a single round of exposure and etching. In double patterning technology, this problem is solved by decomposing the pattern into two parts and printing each part with a separate round of exposure and etching. However, as shown in the figure, a stitch is generated by the decomposition. This stitch is sensitive to the overlay errors from the manufacturing equipment and hence is a potential source of yield loss. Therefore, to improve the overall yield of the layout, the router should place the metal patterns in a decomposition-friendly manner to reduce the number of stitches.
1.3.4 Via

A *via* is a structure used to connect wires on different metal layers. Fig. 1.12a shows such an example. Vias may fail due to, for example, random defects, electromigration, cut misalignment, and thermal stress induced voiding effects. Therefore, it is desirable that the router reduces the number of vias in routing. For the vias that cannot be removed, the router should explore the possibility of replacing them with special via structures such as fat vias and redundant vias. These special vias make the layout more fault-tolerant. Fig. 1.12b shows an example redundant-via structure.

1.3.5 Antenna Effect

Figure 1.13. An illustration of antenna effect.

Antenna effect describes the phenomena of damage to MOS gates caused by electrical charge accumulation on conductor wires. In plasma etching, exposed conductor wires collect electrical charge. If the conductor wires are connected to gate oxides but no diffusion regions, the
charge will accumulate and a discharging current may tunnel through the gate oxides and damage the gates. One example is shown in Fig. 1.13a. In this example, before segment $e_2$ is manufactured, $e_1$ is connected to a gate but no diffusion region. As a result, the charge collected by $e_1$ will build up and may cause gate damage. There are several approaches to solve this problem, as shown in Fig. 1.13b, 1.13d and 1.13c. In Fig. 1.13b, a jumper is inserted on $e_1$. This jumper breaks $e_1$ by temporarily jumping to a higher layer. In this way, the size of the segment that collects electrical charge and connects to the gate is reduced and the robustness of the layout to antenna effect is improved. The approach in Fig. 1.13c is similar to the one in Fig. 1.13b. The only difference is that now a bigger portion of $e_1$ is moved to the higher layer. Compared with the approach in Fig. 1.13b, the one in Fig. 1.13d reduces the number of added vias but involves more changes to the rest of the net. Fig. 1.13c shows the approach of diode insertion, where a dummy diode is placed beside the gate. Before the gate is damaged by the tunnel current, the diode breaks down and a low resistance conductive path is formed to discharge the charge on $e_1$.

Among the three approaches mentioned above, those in Fig. 1.13b and 1.13d change the routing paths of the wires in the layout, and hence can be taken into account in the stage of routing in the design flow.

1.4 Conclusions

In this chapter, the basic concept of DFM&Y has been briefly introduced. Also, the possibilities of improving manufacturability and yield in the stage of routing have been described.
Chapter 2

Research Objective and Previous Work

2.1 Research Objective

In this dissertation, the problem of routing aware of manufacturability and yield is investigated, and several novel algorithms are proposed. Fig. 2.1 shows an overall description of the problems that are potentially involved in the topic. In the figure, the design metrics are divided into two groups. One group includes the traditional design metrics such as timing, power, noise, and area, and the other includes the manufacturability and yield metrics that have been introduced in the first chapter. To determine a specific problem, one can choose a routing stage from the middle yellow oval representing the routing stages, one or more metrics from the oval representing the manufacturability and yield metrics, and/or some metrics from the left oval representing the traditional design metrics. For example, if one chooses “track routing” from the middle oval, “random defects” from the right oval, and “timing” from the left oval, the problem becomes “how to design a track router to simultaneously optimize the timing performance and random-defect-induced yield loss of the circuits”.

2.2 Previous Work

A summary of the literature concerning integrating the manufacturability and yield considerations into routing is provided in Appendix A. Each table in the Appendix is divided into several sub-tables by the double-lines. The entries in each sub-table are closely related to each other and ordered chronologically. This ordering helps clarify how the formulations and algorithms for a particular topic have evolved over time.
Figure 2.1. An overall description of the research objective.

Note that only the research about routing is listed in the Appendix A tables. In particular, the non-routing DFM&Y techniques (e.g., dummy metal fill) are not included. In addition, the publications that only deal with traditional circuit design metrics, e.g., time, power, noise, etc., is not included either.

Most of the cited references in Appendix A are from prestigious conferences (e.g., DAC, ICCAD, DATE, ASPDAC, GLSVLSI and ISPD) in the area of IC design automation. Note that some of the papers cited in the Appendix may also have a journal version. However, in this dissertation, only the conference version of those papers are noted, since their publication time is typically earlier and hence more clearly shows the evolution process of the work in this area. The summary in the Appendix may not be complete, especially for the work that was published more than 20 years ago. However, the latest trend in the area of DFM&Y and routing should be clear.

The detailed analysis of the existing work is deferred to the later chapters, when the problems targeted by the dissertation are presented. This gives the analysis a context and makes it easier to understand.

2.3 Organization of the Rest of the Dissertation

In the next several chapters, some of my completed research is described in detail. In Chapter 3, the track routing algorithm that simultaneously optimizes timing and yield is presented. In Chapter 4, the techniques used to enhance the double-patterning-friendly detailed routers are
introduced. In Chapter 5, the jumper insertion algorithm that optimizes the number of vias while considering the timing constraints is explained. Finally, in Chapter 6, the concurrent wire sizing and layer reassignment algorithm for yield optimization is described.

Some of the content in this dissertation has been published in:


Some contents in Chapter 6 have been submitted to IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems (TCAD).

2.4 Conclusions

In this chapter, the research objective of this dissertation has been presented. In the next several chapters, some proposed algorithms for DFM&Y will be described in detail.
Chapter 3

Track Routing Optimizing Timing and Yield

3.1 Introduction

As the feature size of current MOS technology continues scaling down, the problem of interconnect timing becomes prominent. In contrast to device scaling, the reduction of wire size yields larger interconnect delay due to the shrinking cross-sectional area of the wires. As a result, interconnect delay dominates over transistor and logic delay and becomes the key factor affecting the overall chip performance. Interconnect not only plays a dominate role in circuit performance, but also has a significant impact on the chip yield. During the manufacturing of interconnects, contaminating particles may fall on the chip surface and cause the interruptions (opens) of nets or unintended interconnections (shorts) between different nets. These defects have been introduced in detail in Section 1.3.1.

One important design phase to optimize interconnect timing and yield is routing [9, 10]. Traditionally, routing is finished in two stages, including global routing and detailed routing. In global routing, a topological routing path for each net is determined. Following the global routing path, the pin-to-pin connection that satisfies the physical design rules is constructed for each net in detailed routing. Compared with detailed routing, global routing has more flexibility for optimization, since it has the view of the whole layout. However, to make the global routing problems tractable, the exact information of the wire widths and spacings is typically hidden from the global routers and the wires are abstracted as zero-width lines. To make sure that the layout regions are not overcrowded with wires, some congestion threshold values are assigned to the boundaries of each region to limit the number of wires that pass across the region boundaries. Since the wire width and
spacing information is not available in global routing, the ability of global routers to optimize the metrics that involve short range interaction between different nets is limited. Such metrics include, for example, the coupling capacitance between wires and the short defects caused by contaminating particles. In contrast to global routing, detail routing is responsible for determining the exact geometric shapes and positions of the wires. As a result, detailed routers must take care of the complex design rules regarding wire widths and spacings. To reduce its running time, the searching performed in detailed routing is typically limited to the regions that are close to the paths returned in global routing. Because of the availability of the wire width and spacing information, detailed routers have the potential to supplement global routers and optimize the metrics that involve the interaction between adjacent wires. However, due to the heavy burden of design rule awareness, detailed routing is usually done in a net-by-net manner. This makes it difficult for the detailed routers to consider more than one net simultaneously. To combat this problem, [11] proposed track routing as an intermediate stage between global routing and detailed routing. In track routing, rows and columns of global routing cells are merged together to form the so-called routing panels. The global routing paths are then decomposed into horizontal and vertical segments that reside in the routing panels. Finally, for each routing layer, the exact positions of the wire segments within each routing panel are determined. One example is shown in Fig. 3.1. Fig. 3.1a shows the global routing paths of three nets. During track routing, the L-shaped path is first decomposed into two wire segments: $BO$ and $B'O$. Then, the exact positions of the wire segments in the routing panels are determined by some track routing algorithm, as shown in Fig. 3.1b and 3.1c.

![Diagram](image)

Figure 3.1. An example of track routing.
Compared with global routing, track routing is aware of the relatively more accurate wire adjacency information and can be used to optimize the interaction between different nets. Compared with detailed routing, track routing is able to simultaneously consider all the segments in a routing panel and hence is more flexible for routing optimizations. As a result, track routing has become a popular design stage to optimize the timing delay [12] and noise [13] induced by coupling capacitance. Recently [14] proposed a track routing algorithm for yield optimization, where the wire widths and spacings are adjusted to reduce the Probability-Of-Failure (POF) due to opens and shorts. However, the adjustment of wire widths and spacings also has significant impacts on the timing and noise performance of the circuit, since wire resistance, area capacitance and coupling capacitance are sensitive to the widths and spacings of the wires. Therefore, it is necessary for the track routers to be aware of timing and noise impacts when doing yield optimization.

In this chapter, a track routing algorithm that takes into account both timing and yield is proposed. The major contributions of this work are summarized below.

1. Optimizations of timing and yield are unified in the same framework in track routing.

2. It is shown that the problem of wire sizing and spacing for timing and yield can be formulated as a Mixed Linear Geometric Programming Problem.

3. A heuristic is proposed to find a decent track routing solution within reasonable amount of running time.

The experimental results in Section 3.5 show that, compared with the algorithm that optimizes only yield, the algorithm proposed in this chapter is able to improve the minimum timing slack on average by 20%, with a yield penalty of only 3%. It has also been demonstrated that, compared with yield, timing is more sensitive to the ordering, sizing and spacing of the wires. Therefore, the track routing algorithms that focus only on yield optimization may significantly degrade the timing performance of the circuits.

The rest of this chapter is organized as follows. In Section 3.2, the notations and models used in the work are presented. The problem is formulated in Section 3.3. The algorithm is described in Section 3.4. The experimental results are reported in Section 3.5. Finally, the chapter is concluded in Section 3.6.
Table 3.1. A summary of notation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W$</td>
<td>The set of wire segments ${W_1, W_2, \ldots, W_i, \ldots}$</td>
</tr>
<tr>
<td>$W_i$</td>
<td>Wire segment $i$</td>
</tr>
<tr>
<td>$w_i$</td>
<td>Width of $W_i$</td>
</tr>
<tr>
<td>$p_i$</td>
<td>Position of the middle line of $W_i$</td>
</tr>
<tr>
<td>$M_i$</td>
<td>Preferred position of the middle line of $W_i$</td>
</tr>
<tr>
<td>$d_i$</td>
<td>Deviation of $p_i$ from $M_i$</td>
</tr>
<tr>
<td>$L_i$</td>
<td>Length of $W_i$</td>
</tr>
<tr>
<td>$L_{ij}$</td>
<td>Overlapped wire length between $W_i$ and $W_j$</td>
</tr>
<tr>
<td>$l_{ij}$</td>
<td>Overlapped and adjacent wire length between $W_i$ and $W_j$</td>
</tr>
<tr>
<td>$n_i$</td>
<td>${W_j</td>
</tr>
<tr>
<td>$s_{ij}$</td>
<td>Space between $W_i$ and $W_j$</td>
</tr>
<tr>
<td>$cc_{ij}$</td>
<td>Coupling capacitance between $W_i$ and $W_j$</td>
</tr>
<tr>
<td>$P_i$</td>
<td>Routing panel $i$</td>
</tr>
<tr>
<td>$T_i$</td>
<td>Top position of $P_i$</td>
</tr>
<tr>
<td>$B_i$</td>
<td>Bottom position of $P_i$</td>
</tr>
<tr>
<td>$W_{\text{min}}$</td>
<td>Minimum wire width rule of the layer</td>
</tr>
<tr>
<td>$W_{\text{max}}$</td>
<td>Maximum wire width rule of the layer</td>
</tr>
<tr>
<td>$S_{\text{min}}$</td>
<td>Minimum wire spacing rule of the layer</td>
</tr>
<tr>
<td>$N$</td>
<td>The set of nets ${N_1, N_2, \ldots, N_i, \ldots}$</td>
</tr>
<tr>
<td>$N_i$</td>
<td>Net $N_i$</td>
</tr>
<tr>
<td>$\text{SINK}_i$</td>
<td>The set of sinks of net $N_i$</td>
</tr>
<tr>
<td>$\text{SINK}_{ij}$</td>
<td>The $j^{th}$ sink in $\text{SINK}_i$</td>
</tr>
<tr>
<td>$\text{RAT}_{ij}$</td>
<td>Required arrival time of $\text{SINK}_{ij}$</td>
</tr>
<tr>
<td>$\text{delay}_{ij}$</td>
<td>Delay at sink $\text{SINK}_{ij}$</td>
</tr>
</tbody>
</table>

Figure 3.2. Some examples for illustrating the notation used in the problem formulation.
3.2 Preliminaries

3.2.1 Notation

The notation used in this chapter is listed in Table 3.1.

Table 3.1 is divided into two parts by a double-line. The meaning of the notations in the bottom part is straightforward. In the following, Fig. 3.2 is used to explain some notations in the top part of the table.

As defined in Table 3.1, \( p_i \) is the position of the horizontal or vertical middle line of wire \( W_i \). If \( W_i \) is in a horizontal (vertical) panel, \( p_i \) is the \( y(x) \) coordinate of the horizontal (vertical) middle line of \( W_i \). For example, in Fig. 3.2a \( p_1 \) is the position of the horizontal middle line of wire \( W_1 \).

In Table 3.1, \( M_i \) is used to denote the preferred position of \( W_i \). This position is defined to be the median of the \( x \) or \( y \) coordinates of the pins that fall into the same panel as \( W_i \). It has been shown in [14] that this position is desirable for shorter wire length. One example is shown in Fig. 3.2b. In the figure \( M_i \) is the median of the \( y \) coordinates of \( PIN_1, PIN_2 \) and \( PIN_3 \). If the wire is not assigned to \( M_i \) but assigned to \( p_i \), as shown in Fig. 3.2c, the total wire length is increased. Therefore, in general, \( p_i \) is preferred to be close to \( M_i \). The deviation of \( p_i \) from \( M_i \) is denoted by \( d_i \).

In Table 3.1, \( L_{ij} \) is defined to be the overlapped length between \( W_i \) and \( W_j \). Note that it is different from \( l_{ij} \) which is the overlapped and adjacent length between \( W_i \) and \( W_j \). This is demonstrated in Fig. 3.2a by the difference between \( L_{13} \) and \( l_{13} \). While \( l_{13} \) is dependent on wire ordering, \( L_{13} \) is not. The notation \( n_i \) represents the set of wires whose overlapped and adjacent lengths with \( W_i \) are larger than zero. For example, in Fig. 3.2a \( n_1 = \{ W_2, W_3 \} \).

3.2.2 Models

Yield Model

In this work, the yield models in [14] are used to estimate the probability of failure (POF) due to opens and shorts. These models are based on the assumption that the size of the contaminating particles follows the widely used distribution:

\[
F(x) = kx^{-r} \quad (x_{\text{min}} \leq x \leq \infty) \tag{3.2.1}
\]

19
where $x$ is the defect size and $x_{\text{min}}$ is minimum feature size that can be resolved in lithography, $r$ is a constant around 3, and $k$ is a constant for the normalization of the distribution, i.e. $\int_{x_{\text{min}}}^{\infty} F(x)dx = 1$. To obtain the probability of failure due to opens and shorts, the distribution function in (3.2.1) is multiplied with the probability that a particle falls into the critical area (i.e., the area where the particle may induce open or short defect), and the product is integrated over the range between $x_{\text{min}}$ and $\infty$. After the integration, the POF due to the open defects on wire $W_i$ is obtained as below.

$$\text{POF}^o_i = \frac{k L_i}{2A_{\text{chip}}} \left( \frac{w_i + S_{\text{min}}}{2w_i^2 + S_{\text{min}}w_i} \right)$$  \hspace{1cm} (3.2.2)$$

where $k$ has the same meaning as in (3.2.1), and $A_{\text{chip}}$ is the total area of the chip. Other notations are as defined in Table 3.1.

The POF due to the short defects between $W_i$ and $W_j$ is calculated as below in (3.2.3).

$$\text{POF}^s_{ij} = \frac{k l_{ij}}{2A_{\text{chip}}} \left( \frac{s_{ij} + W_{\text{min}}}{2s_{ij}^2 + W_{\text{min}}s_{ij}} \right)$$  \hspace{1cm} (3.2.3)$$

where $k$ and $A_{\text{chip}}$ have the same meaning as in (3.2.2). The other notations are as defined in Table 3.1.

It has been shown in [14] that the two equations in (3.2.2) and (3.2.3) can be approximated as below.

$$\text{POF}^o_i \approx \frac{k L_i}{2A_{\text{chip}}} \left( a \frac{S_{\text{min}}}{w_i} - b \right) \left( 1 \leq \frac{w_i}{S_{\text{min}}} \leq 20 \right)$$  \hspace{1cm} (3.2.4)$$

$$\text{POF}^s_{ij} \approx \frac{k l_{ij}}{2A_{\text{chip}}} \left( a \frac{W_{\text{min}}}{s_{ij}} - b \right) \left( 1 \leq \frac{s_{ij}}{W_{\text{min}}} \leq 20 \right)$$  \hspace{1cm} (3.2.5)$$

where $a$ and $b$ are constants, which respectively take the value of 0.7399 and 0.0453. The regression coefficients of the above approximation formulas are over 99.8% in the specified range.

As aforementioned, the deviation of the wire’s position from its preferred position tends to increase the wire length. Since longer wire length indicates higher POF due to the open defects, [14] uses the following formula to consider this deviation in the cost function.

$$\text{POF}^{\ast o}_i \approx \frac{k d_i}{2A_{\text{chip}}} \left( \frac{W_{\text{min}} + S_{\text{min}}}{2W_{\text{min}}^2 + S_{\text{min}}W_{\text{min}}} \right)$$  \hspace{1cm} (3.2.6)$$

The equation (3.2.6) is obtained by substituting $d_i$ and $W_{\text{min}}$ for $L_i$ and $w_i$ in (3.2.2).

In this work, the equations (3.2.4), (3.2.5) and (3.2.6) are used in the problem formulation.
Coupling Capacitance Model

The coupling capacitance between adjacent wires is taken into account to more accurately model the impact of track routing on timing. This capacitance is calculated by the following formula [12].

\[ cc_{ij} = \alpha \frac{l_{ij}}{s_{ij}} \]  

(3.2.7)

where \( \alpha \) and \( \beta \) are technology-dependent constants, and \( cc_{ij}, l_{ij} \) and \( s_{ij} \) are as defined in Table 3.1.

Timing Model Considering Coupling Capacitance

![Diagram of Elmore delay model](image)

Figure 3.3. An example of Elmore delay.

The well known Elmore delay model is used in this work to evaluate the timing performance of the nets. From the perspective of circuit timing analysis, Elmore delay is a simple approximation of the delay through an RC network and can be interpreted as the first time moment of the impulse response [15]. Elmore delay can well estimate the delay of the circuits whose response has a dominant time constant. However, for the circuits where multiple time constants affect the delay, the accuracy of Elmore delay is limited. Although the Elmore delay model has some drawbacks in terms of accuracy, its fidelity is still relatively high and, due to its simplicity, has been widely used in layout optimization algorithms.
In the Elmore delay model, the wires in a routing tree are modeled as \( \Pi \) equivalent circuits, as shown in Fig. 3.3a. The notation \( r_i \) and \( c_i \) in Fig. 3.3a represents the resistance and area capacitance of \( W_i \). They are modeled as below.

\[
    r_i = \rho \frac{L_i}{w_i} \quad (3.2.8)
\]

\[
    c_i = \sigma L_i w_i + 2fL_i \quad (3.2.9)
\]

where \( \rho \), \( \sigma \), and \( f \) are constants respectively representing sheet resistance, capacitance per unit area and fringe capacitance per unit perimeter, and \( L_i \) and \( w_i \) are the length and width of wire \( W_i \).

After replacing each wire with its \( \Pi \) model, the routing tree is converted to an RC tree. An example is shown in Fig. 3.3b, where \( r_d \) is the output resistance of the driver, \( c_D \), \( c_E \) and \( c_F \) are the node capacitance at \( D \), \( E \) and \( F \), and \( r_1 \) and \( r_2 \) are the resistance of \( W_1 \) and \( W_2 \). The Elmore delay at node \( i \) in an RC tree is calculated as in equation (3.2.10).

\[
    ED_i = \sum_{r_j \in \text{PATH}_i} r_j \sum_{k \in \text{downstream}(r_j)} c_k \quad (3.2.10)
\]

where \( \text{PATH}_i \) is the path connecting the source to node \( i \); \( \text{downstream}(r_j) \) is the set of the nodes whose paths to the source pass through \( r_j \). For example, in Fig. 3.3b, \( \text{downstream}(r_d) = \{D, E, F\} \).

To make problem formulation aware of the impact of coupling capacitance on timing, the method described as follows is used. In such method, the coupling capacitance between two wires is split into halves which are then attached to the two end nodes of the wire. An example is shown in Fig. 3.3(c). In the figure, \( W_2 \) is coupled to \( W_3 \). Half of the coupling capacitance is attached to node \( D \) and the other half is attached to node \( F \). After plugging this coupling capacitance into the RC tree, the Elmore delay formula in (3.2.10) is used to predicate the timing performance of the nets.

### 3.3 Problem Formulation

Given a set of nets \( N = \{N_1, N_2, \ldots, N_i, \ldots\} \) each of which contains a set of sinks \( SINK_i = \{SINK_{i1}, SINK_{i2}, \ldots, SINK_{ij}, \ldots\} \), the required arrival time of the sinks, the set
\[ W = \{ W_1, W_2, \ldots, W_i, \ldots \} \]

of the wire segments obtained by decomposing the global routing paths, and the wire spacing and width rules, perform track routing while simultaneously optimizing timing and yield. Mathematically, the problem is formulated as in Fig. 3.4.

\[
\text{min} \quad \mu \lambda_{\text{yield}} + (1 - \mu) \lambda_{\text{timing}}
\]

\[
\text{S.t.} \quad \sum_{W_i \in W} (POF_i^o + POF_i^{op}) + \sum_{W_i, W_j \in W, i > j} POF_{ij} \leq \lambda_{\text{yield}} \overline{POF}
\]

\[
|p_i - M_i| \leq d_i \quad (\forall W_i \in W)
\]

\[
delay_{ij} \leq \lambda_{\text{timing}} \text{RAT}_{ij} \quad (\forall N_i \in N, SINK_{ij} \in SINK_i)
\]

\[
B_k + \frac{w_i}{2} \leq p_i \leq T_k - \frac{w_i}{2} \quad (\forall W_i \in P_k)
\]

\[
W_{\text{min}} \leq w_i \leq W_{\max} \quad (\forall W_i \in W)
\]

\[
S_{\min} \leq s_{ij} \leq |p_i - p_j| - \frac{w_i + w_j}{2} \quad (\forall W_i, W_j \in W, j \in n_i)
\]

\[
\lambda_{\text{yield}}, \lambda_{\text{timing}} \geq 0
\]

Figure 3.4. Formulation of the track routing problem for the optimization of timing and yield.

In the objective function, \( \mu \) is a user defined parameter bounded to \([0,1]\). It is used to adjust the relative weight of yield and timing. The variables \( \lambda_{\text{yield}} \) and \( \lambda_{\text{timing}} \) are used to indicate the “criticality” of yield and timing. They are constrained in (3.3.1) and (3.3.3).

The variable \( \lambda_{\text{yield}} \) is constrained in (3.3.1). The notation \( \overline{POF} \) in (3.3.1) represents a constant that is obtained as follows. Given a track routing problem, an initial track assignment is performed with a simple algorithm based on interval packing without considering yield or timing. Minimum wire width and spacing are used in the initial track routing solution. The total \( POF \) is then calculated and used as the value of \( \overline{POF} \). The left side of (3.3.1) is the total \( POF \) based on the models in (3.2.4), (3.2.5) and (3.2.6) in Section 3.2.2. (3.3.2) is used to constrain the deviation of the wire’s position from its preferred position. This deviation is used in (3.3.1) to calculate the value of \( POF_i^{op} \) as defined in Section 3.2.2.

(3.3.3) is used to constrain \( \lambda_{\text{timing}} \). In (3.3.3), \( \text{delay}_{ij} \) and \( \text{RAT}_{ij} \) are the Elmore delay and required arrival time of the sink \( SINK_{ij} \).

The position of a wire is confined to the given routing panel by the constraint (3.3.4). Finally, (3.3.5) and (3.3.6) are the wire width and spacing constraints.
3.4 Algorithm

3.4.1 Basic Framework

Due to the complexity of the problem, the algorithm is divided into two steps. In the first step, the wire segments in each routing panel are ordered to minimize some cost function that is aware of both timing and yield. This is discussed in Section 3.4.2. In this step, the wire adjacency relationship is determined. In the second step, the positions and widths of the wire segments are optimized by the algorithm presented Section 3.4.3 to minimize the objective function in Fig. 3.4. This two-step procedure is illustrated in Fig. 3.5.

**Figure 3.5. An overview of the algorithm.**

```
Algorithm 3.1 The wire segment ordering algorithm.
Input: The set of routing panels and the set of wire segments that reside in the panels.
Output: The order of the wire segments in each panel.

1: for each routing panel $P_i$ do
2:   Build the Weighted Horizontal Constraint Graph (WHCG) for the wires in $P_i$;
3:   Find the Maximum Clique (MC) in the WHCG;
4:   Find the Minimum Hamiltonian Path (MHP) in the MC;
5:   Pick out the segments to the right of MC and put them into the set $S_{right}$;
6:   Sort the segments in $S_{right}$ according to their left end point coordinates (from left to right);
7:   for each segment $S$ in $S_{right}$ do
8:     Build a list $L$ of $MC$’s vertices that are mergeable to $S$;
9:     Calculate the $merge$-$cost$ for each vertex in $L$;
10:    Among the vertices in $L$, choose the one with the lowest $merge$-$cost$ and merge it with $S$;
11:  end for
12:  Pick out the segments to the left of $MC$ and put them into the set $S_{left}$;
13:  Sort the segments in $S_{left}$ according to their right end point coordinates (from right to left);
14:  Repeat the loop from line 7 to 11 for the segments in $S_{left}$;
15:  Find the $MHP$ for the merged clique;
16:  Determine the order of the wire segments according to the $MHP$ found in line 15;
17:  end for
```
3.4.2 Wire Segment Ordering

Algorithm 3.1 shows the algorithm for ordering the wire segments. Before explaining the algorithm, it is necessary to introduce some concepts in graph theory.

Definition 3.1. [16] Let \( \{I_1, I_2, ..., I_n\} \) be a set of intervals. The corresponding interval graph is \( G = (V, E) \), where

- \( V = \{I_1, I_2, ..., I_n\} \), and
- \( (I_\alpha, I_\beta) \in E \) if and only if \( I_\alpha \cap I_\beta \neq \emptyset \).

It is well known that interval graphs belong to the so-called perfect graphs [17]. As a result, some difficult combinatorial problems, such as graph coloring, maximum clique, and maximum independent set problem can all be solved in polynomial time [17, 18].

Definition 3.2. For an undirected graph, a Hamiltonian path is a path that visits each vertex exactly once.

Generally, the problem of determining the shortest Hamiltonian path in a complete graph is NP-hard [18]. This means that it is not likely that the problem can be solved in polynomial time. Therefore, heuristics are typically used to return a short Hamiltonian path within reasonable amount of running time.

Next, the wire ordering algorithm in Algorithm 3.1 is explained. The algorithm in Algorithm 3.1 processes the wire segments in a panel by panel manner. In line 2, a Weighted Horizontal Constraint Graph (WHCG) is built. The vertices in the graph represent the wire segments in the routing panel. There is an edge between two vertices if the corresponding wire segments overlap with each other. The weight of an edge is determined as follows. Assume that the wire segment \( W_i \) overlaps with \( W_j \) and the two wire segments respectively belong to net \( N_m \) and \( N_n \). The weight of the edge connecting \( W_i \) and \( W_j \) is defined by the following equations.

\[
weight(W_i, W_j) = (\mu \delta_{POF} + (1 - \mu)\delta_{delay})
\]

\[
\delta_{POF} = \frac{1}{POF} \left[ \frac{KL_{ij}}{2A_{chip}} \left( \frac{S_{min} + W_{min}}{2S_{min}^2 + W_{min}S_{min}} \right) \right]
\]

\[
\delta_{delay} = \max \{ \delta_{delay}^m, \delta_{delay}^n \}
\]
\[ \delta_{\text{delay}}^m = \frac{r_{\text{upstream,cc}}^{m}}{\min \{ \text{RAT}_{mk} | \text{SINK}_{mk} \in \text{downstream}(W_i) \}} \]  

(3.4.4)

\[ \delta_{\text{delay}}^n = \frac{r_{\text{upstream,cc}}^{n}}{\min \{ \text{RAT}_{nk} | \text{SINK}_{nk} \in \text{downstream}(W_j) \}} \]  

(3.4.5)

\[ cc_{ij} = \alpha \frac{L_{ij}}{S_{\text{min}}^\beta} \]  

(3.4.6)

Figure 3.6. An illustration of the edge weight function.

In (3.4.1) \( \mu \) is the same parameter as the one in the objective function in Fig. 3.4. Variables \( \delta_{\text{POF}} \) and \( \delta_{\text{delay}} \) indicate the sensitivities of POF and delay to the adjacency of \( W_i \) and \( W_j \). They are respectively defined in (3.4.2) and (3.4.3). In (3.4.2) the content in the square bracket is an estimation of the POF due to the short defects between \( W_i \) and \( W_j \) when the two wires are adjacent. It is obtained by substituting \( L_{ij} \) and \( S_{\text{min}} \) for \( l_{ij} \) and \( s_{ij} \) in (3.2.3). The value of \( \text{POF} \) is obtained in the same way as introduced in Section 3.3. In (3.4.4), \( \text{RAT}_{mk} \) is the required arrival time of the sink \( \text{SINK}_{mk} \). The variable \( r_{\text{upstream}}^{m} \) represents the upstream resistance of the wire \( W_i \). This is illustrated in Fig. 3.6. In the figure the upstream resistance of \( W_i \) is the resistance from the driver \( D_m \) to the node \( A \). The variable \( cc_{ij} \) represents the coupling capacitance between \( W_i \) and \( W_j \). The product \( r_{\text{upstream,cc}}^{m} \) is an estimation of the extra delay caused by the coupling capacitance. The variables in (3.4.5) are defined in a similar way as those in (3.4.4). Finally, (3.4.6) is an estimation of the coupling capacitance between \( W_i \) and \( W_j \). It is obtained by substituting \( L_{ij} \) and \( S_{\text{min}} \) for \( s_{ij} \) and \( l_{ij} \) in (3.2.7).

After the WHCG is built, its maximum clique (the clique [19] with the most vertices) is found in line 3 in Algorithm 3.1. Since WHCG is an interval graph, its maximum clique can be obtained efficiently in polynomial time by using the algorithm in [17]. If there are more than one maximum clique, we choose the one where the sum of the weight of the edges is the largest. In line 4 the minimum Hamiltonian path is obtained by the Lin-Kernighan heuristic in [18]. In line 5 the segments to the right of the maximum clique are picked out. This is explained in Fig. 3.7a. In the
Figure 3.7. An example of wire segment ordering.

figure, clique $BCD$ is chosen as the maximum clique. Since segment $E$ is to the right of the clique, it is inserted into the set $S_{right}$. In line 6, the segments in $S_{right}$ are sorted according to their left endpoint coordinates. In the loop from line 7 to 11, the segments in $S_{right}$ are merged to the maximum clique one-by-one. This merging process is described as follows.

**Definition 3.3.** Two vertices in a WHCG are mergeable if there is no edge connecting them.

In fact, if there is no edge between a pair of vertices in the WHCG, the two wire segments corresponding to the two vertices have no overlap. This means that the two wire segments can occupy the same routing track without causing any conflict. From the perspective of the conflict graph, merging the vertices that satisfy the criteria in Definition 1 does not increase the size of the maximum clique of the graph.

In line 8 in Algorithm 3.1, a list of the vertices that belong to the maximum clique and are mergeable to $S$ is built. Then, for each vertex in the list, a merge cost is calculated. The merge cost of a vertex $p$ is defined to be the length increase of the minimum Hamiltonian path (the one found in line 4) as a result of merging $p$ and $S$. For example, in Fig. 3.7c the maximum clique is the triangle. The minimum Hamiltonian path of the maximum clique is highlighted by the bold lines. If we merge segment $A$ to node $BE$, the length of the path will be increased by 4. So, in this case the merge cost of the node $\{BE\}$ is 4.

After merging the segments in the set $S_{right}$ to the maximum clique, the segments that are to the left of the maximum clique are picked out and put into the set $S_{left}$. These segments are
sorted according to their right end points and merged to the maximum clique in a similar way as described above.

After all the vertices in the conflict graph are merged to the maximum clique, the MHP in the final merged clique is found in line 15. Finally, the order of the wire segments is determined according to the minimum Hamiltonian path.

Fig. 3.7 shows an example of wire segment ordering. Fig. 3.7a shows the set of wire segments in the panel. The conflict graph of these segments is shown in Fig. 3.7b. The maximum clique in the conflict graph is the one containing nodes B, C, and D. The MHP of the maximum clique is the path from C through B to D. In the merging process, node E is merged to B, and node A is merged to D. The MHP of the merged clique is from node \{C\} to \{A, D\} through \{B, E\}. According to this path, the segments in the panel are ordered as shown in Fig. 3.7e.

**Running Time Analysis**

The algorithm in Algorithm 3.1 processes the segments in a panel-by-panel manner. For a single panel, the time complexity of constructing the conflict graph is $O(S \cdot \log(S) + E)$, where $S$ is the number of segments in the panel, and $E$ is the number of edges in the conflict graph. In practice, $E$ is bounded by $O(T^2 \cdot P)$, where $T$ is the maximum capacity of the global routing edges and $P$ is the number of global routing tiles in the panel. The value of $S$ is bounded by $O(T \cdot P)$. So, the time complexity of conflict graph construction is bounded by $O(T^2 \cdot P \cdot \log(P))$.

The time complexity of finding the maximum clique in the conflict graph is $O(S \cdot \log(S) + E \cdot D)$, where $S$ and $E$ are as previously defined, and $D$ is the number of maximal cliques that a node in the conflict graph belongs to. In practice, $D$ is bounded by $P$. So, the time complexity of finding the maximum clique is bounded by $O(T^2 \cdot P^2)$.

The minimum Hamiltonian path is found by the Lin-Kernighan heuristic, whose running time is empirically $S^2 \cdot 2^2$[18]. Since $S$ is bounded by $O(T \cdot P)$, the time complexity of searching for the minimum Hamiltonian path is bounded by $O(T^2 \cdot 2^2 \cdot P^2 \cdot 2)$.

The merging process takes in total $O(S \cdot M + E)$ time, where $M$ is the number of nodes in the maximum clique. In practice, $M$ is bounded by $T$. So, the time complexity of the merging process is bounded by $O(T^2 \cdot P)$.

In conclusion, the total running time of processing the segments in a panel is bounded by $O(T^{2.2} \cdot P^{2.2})$. 

28
3.4.3 Wire Positioning and Sizing for Yield and Timing

After the wire segments are ordered, the problem becomes wire positioning and sizing to minimize the objective function in Fig. 3.4. In the following, it is first shown that this problem can be formulated as a Mixed Linear Geometric Programming (MLGP) problem, which can be transformed into a convex optimization problem. Since general nonlinear convex optimization may take a long running time, a heuristic is then proposed to solve the problem much faster while still returning a good solution.

The MLGP Formulation

Some definitions related to MLGP are given below.

Definition 3.4. A monomial function is a real valued function of positive variables \((x_1, x_2, ..., x_n)\) with the form \(f(x) = cx_1^{a_1}x_2^{a_2}...x_n^{a_n}\) where \(c > 0\) and \(a_i \in R\).

Definition 3.5. A function is called a posynomial function if it can be represented as a sum of monomial functions.

Definition 3.6. A Mixed Linear Geometric Programming (MLGP) problem is an optimization problem of the form:

\[
\begin{align*}
\text{minimize} & \quad f_0(x) + h_0(z) \\
\text{s.t.} & \quad f_i(x) \leq h_i(z) \quad i = 1, ..., m \\
& \quad g_i(x) = 1 \quad i = 1, ..., p 
\end{align*}
\]

(3.4.7)

(3.4.8)

where \(f_i\) are posynomial functions, \(g_i\) are monomial functions and \(h_i\) are affine functions (i.e., linear functions plus a constant).

By using the logarithmic transformation (i.e., \(x = e^y\)) for the variables \(x_1, x_2, ..., x_n\), but keeping the variables \(z_1, z_2, ..., z_k\) as they are, an MLGP problem can converted to a convex optimization problem [20].

In the following, it is shown that, for a given ordering of the wire segments, the formulation in Section 3.3 can be transformed into an MLGP problem. In particular, it can be shown that the constraints (3.3.1) and (3.3.3)-(3.3.6) in Fig. 3.4 can all be expressed in the form as shown below.

\[
f(w_i, s_{ij}) \leq h(p_i, d_i, \lambda_{yield}, \lambda_{timing})
\]

(3.4.9)
where \( f(w_i, s_{ij}) \) represents a posynomial function of \( w_i \) and \( s_{ij} \), and \( h(p_i, d_i, \lambda_{yield}, \lambda_{timing}) \) represents an affine function of \( p_i, d_i, \lambda_{yield}, \) and \( \lambda_{timing} \).

Constraint (3.3.4) in Fig. 3.4 is equivalent to the following two inequalities in (3.4.10).

\[
\begin{align*}
\frac{w_i}{2} & \leq p_i - B_k \\
\frac{w_i}{2} & \leq -p_i + T_k
\end{align*}
\]  

(3.4.10)

It is obvious that both of the inequalities in (3.4.10) have the form shown in (3.4.9).

Constraint (3.3.5) is equivalent to the following two inequalities in (3.4.11).

\[
\begin{align*}
\frac{W_{\min}}{w_i} & \leq 1 \\
w_i & \leq W_{\max}
\end{align*}
\]  

(3.4.11)

Obviously, the inequalities in (3.4.11) take the form of (3.4.9).

To convert (3.3.6) into the form of (3.4.9), first note that when the wire ordering is determined, the absolute value operation in (3.3.6) can be solved. Assume that \( p_i \) is larger than \( p_j \). Then, (3.3.6) is equivalent to the following two inequalities in (3.4.12).

\[
\begin{align*}
\frac{S_{\min}}{s_{ij}} & \leq 1 \\
s_{ij} + \frac{w_i + w_j}{2} & \leq p_i - p_j
\end{align*}
\]  

(3.4.12)

It is easy to check that both of the inequalities in (3.4.12) are in the form of (3.4.9). Note that if \( p_i \) is smaller than \( p_j \), we can just switch the subscripts \( i \) and \( j \), and we will have the same inequalities as in (3.4.12).

In the following, it is shown that constraints (3.3.1), (3.3.2), and (3.3.3) can be expressed in the form of (3.4.9).

First note that, for a given wire segment ordering, \( l_{ij} \) (the overlapped and adjacent length between wire \( W_i \) and \( W_j \), refer to Section 3.2) becomes a constant for a given pair of wires. Therefore, the yield models (3.2.4), (3.2.5), and (3.2.6) can be written as below.

\[POF^o_i = \frac{\alpha_i}{w_i} - F_i, \quad POF^s_{ij} = \frac{\beta_{ij}}{s_{ij}} - G_{ij}, \quad POF^{os}_i = \gamma id_i,\]

where \( \alpha_i = a \frac{kl_i S_{\min}}{2A_{\text{chip}}} \), \( \beta_{ij} = a \frac{kl_{ij} W_{\min}}{2A_{\text{chip}}} \), \( \gamma_i = \frac{k(W_{\min} + S_{\min})}{2A_{\text{chip}}(2W_{\min} + S_{\min} W_{\min})} \), \( F_i = b \frac{kl_i}{2A_{\text{chip}}} \), and \( G_{ij} = b \frac{kl_{ij}}{2A_{\text{chip}}} \). Note that \( \alpha_i, \beta_{ij}, \gamma_i, F_i, \) and \( G_{ij} \) are all positive constants for a given pair of wires. By substitution, constraint (3.3.1) can be written as below.
\[
\sum_{W_i \in W} \left( \frac{\alpha_i}{w_i} \right) + \sum_{W_i, W_j \in W, i > j} \left( \frac{\beta_{ij}}{s_{ij}} \right) \leq \lambda_{\text{yield}}P_{\text{TOP}} - \sum_{W_i \in W} (\gamma_i d_i) + C \tag{3.4.13}
\]

The left-hand side of (3.4.13) is a posynomial function of \( w_i \) and \( s_{ij} \), and the right-hand side is an affine function of \( \lambda_{\text{yield}} \) and \( d_i \). The variable \( C \) is a constant and is equal to \( \sum_{W_i \in W} F_i + \sum_{W_i, W_j \in W, i > j} G_{ij} \). Therefore, constraint (3.3.1) can be represented in the form of (3.4.9).

To prove constraint (3.3.3) can also be expressed in the form of (3.4.9), we only need to show that the delay \( \text{"delay}_{ij} \) is a posynomial function. To prove that, first note that the resistance and capacitance models in (3.2.8) and (3.2.9) are posynomial functions of \( w_i \), and the coupling capacitance model in (3.2.7) is a posynomial function of \( s_{ij} \). In the Elmore delay model in (3.2.10), the sum of resistance is multiplied by the sum of capacitance. Since the product of two posynomial functions is still a posynomial function, \( \text{delay}_{ij} \), which is calculated by the Elmore delay model in (3.2.10), is a posynomial function of \( w_i \) and \( s_{ij} \).

Next, we deal with the constraint (3.3.2) in Fig. 3.4. This constraint can be rewritten as below.

\[
\begin{cases} 
0 \leq d_i + M_i - p_i \\
0 \leq d_i - M_i + p_i 
\end{cases} \tag{3.4.14}
\]

The inequalities in (3.4.14) are nothing but linear constraints involving \( d_i \) and \( p_i \), and do not affect the convexity of the convex programming problem that is obtained by the logarithmic transformation. The same analysis can also be used for the constraint (3.3.7).

In conclusion, the constraints (3.3.1), and (3.3.3)-(3.3.7) in Fig. 3.4 can all be expressed in the form of (3.4.9), and the linear constraints (3.3.2) and (3.3.7) do not affect the convexity of the problem. Therefore, by using the logarithmic transformation for the variables \( w_i \) and \( s_{ij} \), all the constraints become convex. Since the objective function in Fig. 3.4 is a linear function, the problem of wire positioning and sizing for a given wire segment order becomes a convex optimization problem.

Convex optimization problems can be solved optimally. But the running time for solving the general nonlinear convex optimization problems is typically very long. For example, for a benchmark circuit containing less than 10K nets, the running time of the convex optimization solver in MOSEK [21] is about 9 hours. Part of the overhead is due to the fact that, to use the optimization
engine in MOSEK, the convex optimization problem has to be converted to a so-called separable convex optimization problem. As a result, more linear constraints are added.

In the next section, a heuristic is proposed, which is able to return a decent solution within much shorter running time.

**Algorithm 3.2** A heuristic algorithm for wire positioning and sizing.

| Input: | A set of wire segments in a routing panel and the ordering of the wire segments. |
| Output: | The position and sizing solution of the wire segments. |

1: Use the SOCP formulation proposed in [14] to optimize yield;
2: Calculate the delay at each sink;
3: Insert sinks into a priority queue $Q$ based on their criticality;
4: while $Q$ is not empty do
5: Choose the sink $S$ with the highest priority;
6: Select the wires that are in the path from the driver to $S$ and put them into a set $W$;
7: while $W$ is not empty do
8: Choose the wire $W_i$ that is closest to the driver;
9: if $W_i$ has not been resized or repositioned before then
10: Resize $W_i$;
11: Reposition $W_i$;
12: end if
13: Remove $W_i$ from $W$;
14: end while
15: Remove $S$ from $Q$;
16: end while

**A Heuristic for Wire Positioning and Sizing**

A heuristic algorithm based on local adjustment is shown in Algorithm 3.2. In line 1, the efficient SOCP(Second Order Conic Programming) formulation proposed in [14] is used to do an initial track assignment for yield optimization. Then, the delay considering the coupling capacitance is calculated for each sink. In line 3, the sinks are inserted into a priority queue according to their criticality. In the algorithm, the criticality of a sink is defined as $\text{criticality} = \frac{\text{delay}}{\text{RAT}}$ where $\text{delay}$ and $\text{RAT}$ are the Elmore delay and required arrival time of the sink.

After the priority queue is formed, the algorithm processes the sinks one-by-one. First, the sink with the highest criticality is chosen. Then, the wire segments that are in the path from the driver to the chosen sink are selected. These wires are later resized and repositioned to improve the timing performance while being aware of the yield cost. In line 8 the wire that is closest to the driver
is picked out. If this wire has been resized or repositioned before, it will be ignored. Otherwise, its size and position will be adjusted as described subsequently.

For the wire sizing part, two basic steps are involved. In the first step, the coefficients in the derivation formulas (3.4.15) and (3.4.16) are calculated.

\[
\frac{\partial \text{delay}}{\partial w_i} = \alpha_i^\text{delay} - \beta_i^\text{delay} \frac{w_i^2}{w_i^2} + \sum_{W_j \in n_i} \gamma_i^\text{delay} \left( |p_j - p_i| - \frac{w_i}{2} - \frac{w_j}{2} \right)^3
\]

(3.4.15)

\[
\frac{\partial \text{POF}}{\partial w_i} = -\beta_i^\text{POF} \frac{w_i^2}{w_i^2} + \sum_{W_j \in n_i} \gamma_i^\text{POF} \left( |p_j - p_i| - \frac{w_i}{2} - \frac{w_j}{2} \right)^2
\]

(3.4.16)

where \( p_i, w_i, p_j \) and \( w_j \) are as defined in Table 3.1, \( \alpha_i^\text{delay}, \beta_i^\text{delay}, \gamma_i^\text{delay}, \beta_i^\text{POF} \) and \( \gamma_i^\text{POF} \) are positive coefficients that depend on \( W_i \)'s length, its upstream resistance and downstream capacitance, and its overlapped length with the neighborhood wires. Note that \( p_j, p_i, w_j, \alpha_i^\text{delay}, \beta_i^\text{delay}, \gamma_i^\text{delay}, \beta_i^\text{POF}, \) and \( \gamma_i^\text{POF} \) are regarded as constants when resizing the wire \( W_i \).

After obtaining the coefficients in (3.4.15) and (3.4.16), the following equation (3.4.17) is solved.

\[
\frac{\mu}{\text{RAT}} \frac{\partial \text{delay}}{\partial w_i} + \frac{1 - \mu}{\text{POF}} \frac{\partial \text{POF}}{\partial w_i} = 0
\]

(3.4.17)

where \( \mu \) is the parameter in the objective function in Fig. 3.4, \( \text{RAT} \) is the required arrival time of the sink selected in line 5, and \( \text{POF} \) is the value of the total \( \text{POF} \) in the initial track routing solution.

In this work, due to its simplicity, the bisection method in Algorithm 3.3 is used to solve the equation in (3.4.17). In the algorithm, \( g(w_i) \) represents the left-hand side function in (3.4.17). The variable \( \epsilon \) is a user specified tolerance value. In the first step, it is checked if the product between \( g(w_{\text{min}}) \) and \( g(w_{\text{min}}) \) is smaller than \( -\epsilon \). If so, the interval between \( w_{\text{min}} \) and \( w_{\text{max}} \) is recursively bisected to search for the value \( w_i \) such that \( |g(w_i)| \) is no larger than \( \epsilon \). To reduce the running time, the number of iterations in the loop is limited to 3. If no solution satisfying the stop criteria is found in the loop, the algorithm simply chooses the value such that the absolute value of the function \( g \) is minimum.

Repositioning is done in a similar way as resizing. The derivation formulas are shown in (3.4.18).
Algorithm 3.3 The bisection method used to solve the equation (3.4.17).

Input: The coefficients in (3.4.15) and (3.4.16).
Output: The solution to the equation in (3.4.17).

1: \( w_{\min} = W_{\min}; w_{\max} = W_{\max}; \)
2: \( \text{if } g(w_{\min}) \times g(w_{\max}) < -\epsilon \) then
3: \( i = 0; \)
4: \( \text{while } i < 3 \) do
5: \( w_{\text{middle}} = \frac{w_{\min} + w_{\max}}{2}; \)
6: \( \text{if } |g(w_{\text{middle}})| \leq \epsilon \) then
7: \( \text{return } w_{\text{middle}}; \)
8: \( \text{else} \)
9: \( \text{if } g(w_{\text{middle}}) \times g(w_{\max}) > 0 \) then
10: \( w_{\max} = w_{\text{middle}}; \)
11: \( \text{else} \)
12: \( w_{\min} = w_{\text{middle}}; \)
13: \( \text{end if} \)
14: \( i = i + 1; \)
15: \( \text{end if} \)
16: \( \text{end while} \)
17: \( \text{end if} \)
18: \( \text{if } |g(w_{\min})| < |g(w_{\max})| \) then
19: \( \text{return } w_{\min}; \)
20: \( \text{else} \)
21: \( \text{return } w_{\max}; \)
22: \( \text{end if} \)

\[
\frac{\partial \text{delay}}{\partial p_i} = \sum_{W_j \in n_i, p_j > p_i} \frac{\alpha_{ij}^{\text{delay}}}{(p_j - p_i - \frac{w_i}{2} - \frac{w_j}{2})^3} - \sum_{W_j \in n_i, p_j < p_i} \frac{\alpha_{ij}^{\text{delay}}}{(p_i - p_j - \frac{w_i}{2} - \frac{w_j}{2})^3} \tag{3.4.18}
\]

\[
\frac{\partial \text{POF}}{\partial p_i} = \sum_{W_j \in n_i, p_j > p_i} \frac{\alpha_{ij}^{\text{POF}}}{(p_j - p_i - \frac{w_i}{2} - \frac{w_j}{2})^2} - \sum_{W_j \in n_i, p_j < p_i} \frac{\alpha_{ij}^{\text{POF}}}{(p_i - p_j - \frac{w_i}{2} - \frac{w_j}{2})^2} + f(p_i, M_i) \tag{3.4.19}
\]

where \( p_i, w_i, p_j \) and \( w_j \) are as defined in Table 3.1, \( \alpha_{ij}^{\text{delay}} \) and \( \alpha_{ij}^{\text{POF}} \) are coefficients depending on \( W_i \)'s upstream resistance and its overlapped length with the neighborhood wires, and \( M_i \) is the preferred position of wire \( W_i \), as explained in Section 3.2.

The function \( f(p_i, M_i) \) in (3.4.19) is as defined in (3.4.20).
\[ f(p_i, M_i) = \begin{cases} 
C, & p_i > M_i \\
-C, & p_i < M_i 
\end{cases} \] (3.4.20)

In (3.4.20), \( C \) is a positive constant. This function is derived from the penalty cost of \( POF^\omega_i \) in Section 3.2.2.

After calculating the coefficients, (3.4.18) and (3.4.19) are substituted into (3.4.21) shown below.

\[ \frac{\mu}{RAT} \frac{\partial \text{delay}}{\partial p_i} + \frac{1 - \mu}{POF} \frac{\partial POF}{\partial p_i} = 0 \] (3.4.21)

In (3.4.21), \( \mu, RAT \), and \( POF \) have the same meaning as in (3.4.17). Finally, (3.4.21) is solved by the bisection method similar to the one in Algorithm 3.3.

**Running Time Analysis**

The running time of the algorithm in Algorithm 3.2 contains two parts, one for the SOCP solver and the other for the local wire sizing and positioning steps. The time complexity of the SCOP solver is bounded by \( O(N^{1.335}) \)[14], where \( N \) is the number of variables in the SOCP formulation. For the track routing problem, \( N \) is bounded by \( O(L) \), where \( L \) the total wire length measured in terms of the number of global routing tiles. So, the time complexity of the solver is bounded by \( O(L^{1.335}) \).

Local sizing and positioning are performed by solving the equations with the bisection method. Since there are only 3 iterations in the bisection algorithm in Algorithm 3.3, the running time of the bisection algorithm can be regarded as a constant. However, some tree traversals are needed to collect the coefficients of the equations and update the timing delay of the sinks in the routing trees. Therefore, the time complexity of local sizing and positioning in each iteration is bounded by \( O(L_t) \), where \( L_t \) is the wire length (measured in terms of the number of global routing tiles) of the routing tree whose sink is currently being processed. Since each sink will be processed only once, the total time contributed by the local sizing and positioning steps is bounded by \( O(K.L_{max}) \), where \( K \) is the total number of sinks in all the routing trees in the circuit, and \( L_{max} \) is wire length of the longest routing tree. Furthermore, since both \( K \) and \( L_{max} \) are bounded by \( O(L) \), the time complexity of local sizing and positioning is bounded by \( O(L^2) \).
Finally, the sinks are kept in a priority queue. Maintaining this queue takes in total $O(K \log(K))$ running time. As aforementioned, $K$ is bounded by $O(L)$. So, the time complexity of queue maintenance is bounded by $O(L \log(L))$.

In conclusion, the running complexity of the heuristic algorithm in Algorithm 3.2 is bounded by $O(L^2)$.

### 3.5 Experimental Results

The algorithm in this chapter has been implemented in C. The global routing results are generated by our history-driven global router based on rip-up and reroute. The defect size distribution is assumed to follow the formula $p(x) = kx^{-3}$, where $p(x)$ is the probability density function of the defective size distribution and $k$ is a value for normalization. The minimum and maximum wire width rules are respectively 0.2um and 0.4um. The minimum spacing rule is 0.2um. The IBM98 benchmark circuits are used in the experiments. Some basic information about these circuits is shown in Table 3.2. Since there is no timing information associated with the benchmark circuits, the required arrival time of the sinks is randomly generated between 3ns and 5ns. Monte Carlo simulation is performed by generating 10k random defects according to the defect size distribution function. The Second-Order-Conic-Programming (SOCP) problems in the algorithm in Algorithm 3.2 are solved by the MOSEK conic programming Application Programming Interface (API). The Mixed-Linear-Geometric-Programming (MLGP) problems are solved by the following procedure. First, the MLGP problems are converted into convex programming problems by using the logarithmic transformation. Then, the obtained convex programming problems are further converted into separable convex programming problems by substitution. The hash-table data structure from [22] is used to facilitate the substitution on the fly. Finally, the separable convex programming problems are solved by the “scopt” API in MOSEK. In Table 3.3 and 3.4, the unit of “Slack” is pico-seconds and the unit of “Time” is seconds.

Table 3.3 shows the experimental results comparing the optimal solutions of the MLGP problem and the solutions returned by the heuristic in Algorithm 3.2. Due to the long running time of general nonlinear convex optimization, subsets of wire segments rather than the whole benchmarks are used in the experiments for test. The subsets of wire segments are selected by randomly picking a region whose average wire density is close to that of the whole benchmark circuit. The value of $\mu$ in the objective function in Section 3.3 is set to 0.8, because it is found that this value in general gives a good trade-off between running time and solution quality.
Table 3.2. The number of nets in the benchmark circuits.

<table>
<thead>
<tr>
<th>Name</th>
<th>Number of Nets</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibm01</td>
<td>11k</td>
</tr>
<tr>
<td>ibm02</td>
<td>18k</td>
</tr>
<tr>
<td>ibm03</td>
<td>21k</td>
</tr>
<tr>
<td>ibm04</td>
<td>26k</td>
</tr>
<tr>
<td>ibm06</td>
<td>33k</td>
</tr>
<tr>
<td>ibm07</td>
<td>44k</td>
</tr>
<tr>
<td>ibm08</td>
<td>47k</td>
</tr>
<tr>
<td>ibm09</td>
<td>50k</td>
</tr>
<tr>
<td>ibm10</td>
<td>64k</td>
</tr>
</tbody>
</table>

From the results in Table 3.3, it can be observed that the heuristic is much faster (20x) and its solution is very close to the optimal solution. On average, the number of faults and the value of minimum timing slack are respectively within 2% and 7% of the optimal solution.

It can also be noticed that the gap between the heuristic solution and the optimal solution is larger in sparse circuits than in dense circuits. This is because in dense circuits there is less room to do segment positioning and sizing. Hence, in dense circuits, the solution quality relies more on segment ordering rather than segment positioning and sizing.

Table 3.4 shows the experimental results for different scenarios. First the weight of timing is set to zero to find out the solution that is optimized only for yield. Then, this weight is increased to 0.2 to make the algorithm aware of the timing cost. It can be observed that, with almost the same amount of running time, the minimum slack can be improved by 20% with the number of faults increased by 3%.

Overall, by adjusting the relative weight of timing and yield, the minimum timing slack varies, on average, by 26% while the number of faults by 8%. This shows that, compared with yield, timing seems to be more sensitive to the weight. This phenomena is believed to be caused by the fact that the minimum timing slack is mostly determined by a subset of nets (i.e., the nets that are most timing critical). In contrast, the number of faults is contributed by all the nets. Therefore, it is possible to significantly improve the timing performance of the circuits without hurting the overall yield very much.

3.6 Conclusions

In this chapter, a track routing algorithm that optimizes both timing and yield in an unified framework has been introduced. It has been shown that, for a given segment ordering, the problem
of wire sizing and positioning for yield and time can be formulated as a mixed linear geometric programming problem. To reduce the running time, a practical heuristic has been proposed.

Table 3.3. Comparison between the optimal solution and the solution from our heuristic.

<table>
<thead>
<tr>
<th>Test case</th>
<th>No. of wire segments</th>
<th>Density</th>
<th>Nonlinear convex optimization</th>
<th>Heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Faults</td>
<td>Slack</td>
</tr>
<tr>
<td>Case 1</td>
<td>2553</td>
<td>0.57</td>
<td>1060</td>
<td>279</td>
</tr>
<tr>
<td>Case 2</td>
<td>3382</td>
<td>0.51</td>
<td>1068</td>
<td>196</td>
</tr>
<tr>
<td>Case 3</td>
<td>2812</td>
<td>0.45</td>
<td>948</td>
<td>105</td>
</tr>
<tr>
<td>Case 4</td>
<td>2975</td>
<td>0.58</td>
<td>862</td>
<td>35</td>
</tr>
<tr>
<td>Case 5</td>
<td>4067</td>
<td>0.58</td>
<td>1022</td>
<td>72</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td>4960</td>
<td>687</td>
</tr>
<tr>
<td><strong>Norm</strong></td>
<td></td>
<td></td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 3.4. Comparison between the solutions for different scenarios.

<table>
<thead>
<tr>
<th>Name</th>
<th>For Yield</th>
<th>For Yield and Timing</th>
<th>For timing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Faults</td>
<td>Slack</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ibm01</td>
<td>951</td>
<td>109</td>
<td>15</td>
</tr>
<tr>
<td>Ibm02</td>
<td>984</td>
<td>43</td>
<td>92</td>
</tr>
<tr>
<td>Ibm03</td>
<td>899</td>
<td>81</td>
<td>55</td>
</tr>
<tr>
<td>Ibm04</td>
<td>1033</td>
<td>135</td>
<td>63</td>
</tr>
<tr>
<td>Ibm06</td>
<td>1198</td>
<td>179</td>
<td>97</td>
</tr>
<tr>
<td>Ibm07</td>
<td>851</td>
<td>293</td>
<td>144</td>
</tr>
<tr>
<td>Ibm08</td>
<td>1021</td>
<td>143</td>
<td>132</td>
</tr>
<tr>
<td>Ibm09</td>
<td>1014</td>
<td>265</td>
<td>121</td>
</tr>
<tr>
<td>Ibm10</td>
<td>824</td>
<td>18</td>
<td>232</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>8775</td>
<td>1266</td>
<td>951</td>
</tr>
<tr>
<td><strong>Norm</strong></td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Chapter 4

Enhancing Double-Patterning Detailed Routing with Lazy Coloring and Within-Path Conflict Avoidance

4.1 Introduction

As the current 193nm lithography [23] is approaching its fundamental printing limit [24], more advanced scale-down methods, such as immersion lithography using high refractive index fluid, double-patterning, and EUV lithography have been developed. Among them, double-patterning technology (DPT) is emerging as the most promising one to achieve the 32-nm node and beyond [25]. Similar to the double-exposure techniques, DPT relaxes the pitches by decomposing the design to dual masks. One example is shown in Fig. 4.1, where the lines in the original layout are first decomposed into two groups of line patterns which are then assigned to different masks. In this way, the minimum spacing requirement in each mask is relaxed. In contrast with double exposure, double patterning requires an intermediate development step, which eliminates the resolution degradation due to the cross-coupling that occurs in the latent images of multiple exposures [24]. An example double-patterning process has been shown in Fig. 1.10 of Chapter 1. Fig. 4.2 shows some pitch reduction possibilities of DPT [26]. Since DPT is normally based on current lithography technology, the manufacturing equipment and materials can be reused [27]. This makes DPT very attractive to the IC industry.

DPT requires that two polygons be assigned to different masks if the spacing between them is smaller than a threshold value $Min_{dp}[28]$. This mask assignment problem can be formu-
Figure 4.1. DPT decomposes the layout to relax pitch size.

Figure 4.2. Some possibilities of pitch reduction in double patterning.
Figure 4.3. An example of modeling the mask assignment problem as a coloring problem.

lated as a 2-coloring problem on a conflict graph. Fig. 4.3 shows such an example. Fig. 4.3a shows the original layout, which contains three polygons. The conflict graph of the layout is shown in Fig. 4.3b. In the conflict graph, the vertices correspond to the polygons in the layout and two vertices are connected by an edge if the spacing between the corresponding polygons is smaller than $\text{Min}_{dp}$. During mask assignment, the polygons whose spacing is smaller than $\text{Min}_{dp}$ have to be assigned to different masks. In the conflict graph model, the mask assignment problem for DPT corresponds to assigning two colors to the vertices such that no two vertices that are connected by an edge have the same color. In the formulation of graph coloring, the “colors” correspond to the masks. For example, in Fig. 4.3b, color gray corresponds to Mask A and black corresponds to Mask B. With the graph-coloring formulation, a layout is decomposable for DPT if and only if the conflict graph for the layout is bicolorable[29]. In graph theory, it is well known that a graph is bicolorable if and only the graph does not contain any odd cycle (i.e., the graph is bipartite). This is the reason why, in some papers about layout decomposition for DPT, odd cycles are sometimes called conflict cycles. Unfortunately, there is no guarantee that all layouts are free from conflict cycles. For example, the conflict graph in Fig. 4.3b is a triangle and hence contains a conflict cycle. One widely used technique to get rid of the conflict cycles is to slice some polygons in the layout. For example, the polygon $B$ in Fig. 4.3a is sliced into two polygons $B^1$ and $B^2$ in Fig. 4.3c. The conflict graph for
the sliced layout is shown Fig. 4.3d. It can be observed that no odd cycle exits in the new conflict graph. One of the coloring solutions for the sliced layout is shown in Fig. 4.3e. Note that there is a stitch between $B^1$ and $B^2$. This stitch may cause manufacturing problems such as overlay errors, tight overlay control, line edge errors and interference mismatch [28]. Therefore, it is desirable to assign “colors” to the polygons in a layout such that as few as possible stitches are generated.

In the literature, there are generally two categories of EDA solutions to tackle the DPT layout decomposition problem. The first one, similar to what we have done in the example in Fig. 4.3, consists of searching for the optimal decomposition for a given layout in the post-layout phase. In [28], the polygons of a layout are first fractured into non-overlapping rectangles. Then a conflict graph is constructed over the rectangles based on the spacing requirements of DPT. Conflict cycles of the graph are detected by an algorithm based on breath-first search[30] and are solved by splitting the nodes in the graph. Finally, integer linear programming (ILP)[31] is applied to find a coloring solution which minimizes the design rule violations and number of stitches, and maximizes the overlapping lengths between touching features of different colors. To improve the scalability of the ILP-based algorithm, a layout partitioning technique is used to divide the layout into small regions. Three different layout decomposition methods, including ILP, phase conflict detection and bipartition by node-deletion, are compared in [32]. A model-based method to perform layout decomposition is suggested in [33]. In this method, a physics-based model, instead of the normally used design rules, is used to predict the printability of the layout patterns obtained from decomposition. This model-based method is designed to handle complex geometries and is suitable for small regions where a global coloring fails.

The second category of EDA solutions for DPT layout decomposition tries to consider layout coloring during layout design. The authors in [1] describe the first DPT-friendly detailed routing algorithm. In the algorithm a stitch penalty is added to the cost function to reduce the number of potential stitches. After a net is routed, its path is colored and “shadowed”. In the shadowing operation, the allowed colors of the surrounding grids of the path are constrained based on the DPT spacing rules. This algorithm will be introduced in more detail in Section 4.2.

In this work, two enhancement techniques, including Lazy Color Decision (LCD) and Last Conflict Segment Recording (LCSR), for the DPT-friendly detailed routing algorithm in [1] are proposed. With the help of LCD, the detailed algorithm is able to consider several nets simultaneously, and thus has more flexibility to reduce the number of stitches and wire length. With the help of LCSR, the color conflicts within a single path are detected during path searching. This de-
tection gives the routing algorithm the capability to avoid within-path conflicts and thus improves the successful rate of net coloring.

The rest of the chapter is organized as follows. In Section 4.2 the DPT-friendly detailed routing algorithm proposed in [1] is briefly introduced and some of its limitations are pointed out. Then, in Section 4.3 and 4.4, the techniques of LCD and LCSR are described. Experimental results are shown in Section 4.5. Finally, the chapter is concluded in Section 4.6.

4.2 Background

In this section, the algorithm proposed in [1] is briefly introduced. In that work, every grid on the routing graph has an associated state as defined below in Table 4.1.

<table>
<thead>
<tr>
<th>Grid state</th>
<th>Description</th>
<th>Grid Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>BG</td>
<td>bi-colorable</td>
<td>black or gray</td>
</tr>
<tr>
<td>BG</td>
<td>black-colorable</td>
<td>black only</td>
</tr>
<tr>
<td>BG</td>
<td>gray-colorable</td>
<td>gray only</td>
</tr>
<tr>
<td>BG</td>
<td>uncolorable</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 4.1. Grid state for DPT coloring.

The basic flow of the algorithm in [1] is shown in Fig. 4.4.

1: decompose and color the routing blockages;
2: shadow the colored routing blockages;
3: for each net $n$ do
4: perform path-searching for $n$;
5: color all the grids along the path according to the state of the grids;
6: shadow the colored path to reflect the DPT spacing rules;
7: end for

Figure 4.4. The basic flow of the algorithm in [1].

A simple example for illustrating the algorithm in Fig. 4.5 is shown in Fig. 4.5. The rectangles $B_1$ and $B_2$ are routing blockages. The grids $S$ and $T$ are the two terminals of a net. As shown in Fig. 4.5a, the states of $S$ and $T$ are $BG$ at first, meaning that gray and black are both feasible colors for these grids. During the step of blockage coloring and shadowing, $B_1$ and $B_2$ are colored gray as shown in Fig. 4.5b. Since the spacing between $S$ and $B_1$ is smaller than the DPT minimum spacing rule $\text{Min}_{d_{dp}}$, the state of $S$ is changed to $BG$, meaning that $S$ can only be colored black. Similarly, the state of $T$ is changed to $BG$ by the shadowing of $B_2$. Then, in Fig. 4.5c a path-searching algorithm is applied to find a path connecting $S$ and $T$. After that, the obtained
path is colored according to the states of the grids. Since both $S$ and $T$ have the state $BG$, they are colored black. To reduce the number of stitches, the $BG$ grids along the path are colored black too, as shown in Fig. 4.5d. Finally, shadowing is done around the path to reflect the DPT spacing requirements imposed by the newly colored path grids. This is shown in Fig. 4.5e.

One of the drawbacks of the algorithm in [1] is the pre-emptive coloring of the $BG$ grids. In [1], a path is completely colored immediately after it has been found, even if there are $BG$ grids along the path. Although this immediate coloring is able to reduce the number of stitches along the new path, it may impose unnecessary constraints on the nets that have not been routed yet and lead to a worse solution. One such example is shown in Fig. 4.6. In Fig. 4.6, three nets are to be routed: \{S_1, T_1\}, \{S_2, T_2\} and \{S_3, T_3\}. The rectangle $B$ is a routing blockage and is pre-colored black. Now, assume that we route \{S_1, T_1\} first. After path searching, Path1 is obtained as the connection path between $S_1$ and $T_1$. There are more than one coloring option for this path. If it is colored as in Fig. 4.6b, a legal routing path from $S_3$ to $T_3$ will be at least 15 grids long, as shown in Fig. 4.6b. This
is because the coloring of Path1 in Fig. 4.6b forces Path2 to be colored gray, which then makes the grids $P_1$, $P_2$ and $P_3$ uncolorable. But if we color Path1 as in Fig. 4.6c, Path2 will be colored black. In this case, the shortest routing path from $S_3$ to $T_3$ spans only 7 grids. Longer paths not only consume more routing resources but also impose more color constraints. What this example has demonstrated is that the $BG$ grids should not be colored too soon, since more information may be needed to find a wise coloring solution. To address this problem, the LCD technique is proposed. This is described in the next section.

![Figure 4.6](image-url)

Figure 4.6. An example for illustrating the impact of $BG$ grids coloring on routing.

### 4.3 Lazy Color Decision

The basic idea of this technique is to delay the coloring of $BG$ grids until more information is available. Taking Fig. 4.6 as an example, if the $BG$ grids along Path1 are colored after...
Path3 is obtained, then the shorter solution in Fig. 4.6d can be found. To implement this idea, a conflict graph for the BG monotonic paths (defined below) is kept.

**Definition 4.1.** Given two points $X = (X_1, X_2, ..., X_n)$ and $Y = (Y_1, Y_2, ..., Y_n)$, the Manhattan distance between these two points is defined as: $\sum_{i=1}^{n} |X_i - Y_i|$.

**Definition 4.2.** A path connecting two grids on a routing layer is called a monotonic path if the length of the path is equal to the minimum Manhattan distance between the two grids.

**Definition 4.3.** Two grids on a routing layer are defined to conflict with each other if: a) The distance between them is smaller or equal to the DPT minimum spacing rule $Min_{dp}$; and b) There is no monotonic path connecting the two grids or the monotonic path contains both $\overline{BG}$ and $\overline{BG}$ grids.

**Definition 4.4.** Two BG monotonic paths are defined to conflict with each other if there are conflicting grids on the two paths.

The above definitions are explained in Fig. 4.7f. In the figure, $P_1P_2, P_3P_4, P_5P_6$ and $P_7P_8$ are BG monotonic paths. The path $P_1$ conflicts with $P_3$ because the spacing between them is smaller than $Min_{dp}$ and there is no monotonic path connecting the two grids. As a result, $P_1P_2$ conflicts with $P_3P_4$. Similarly, $P_5P_6$ and $P_7P_8$ also conflict with $P_3P_4$.

The conflict graph is constructed as follows. The vertices in the conflict graph represent the BG monotonic paths. There is an edge between two vertices if the corresponding paths conflict with each other. The conflict graph for Fig. 4.7f is shown in Fig. 4.7g.

Given these definitions, the overall flow of the DPT-friendly detailed routing algorithm proposed by us is as shown in Algorithm 4.1.

Note the difference between this algorithm and the one in [1]. In line 3, only $\overline{BG}$ and $\overline{BG}$ grids along the path are colored, while in [1] all the grids are colored. In line 4, this partly colored path is shadowed. Then the BG monotonic paths are picked out and inserted into the conflict graph. After that, a judgement is made about whether or not to solve the coloring of the components in the conflict graph. This judgment is based on three conditions:

1. The number of nodes in the component is larger than $K$, where $K$ is a user specified parameter.
2. There exits an odd cycle in the component.
Figure 4.7. Application of LCD to the example in Fig. 4.6.
**Algorithm 4.1** DPT-friendly detailed routing algorithm flow with the LCD technique integrated.

1. **for** each net $n$ **do**
2. \hspace{1em} perform path searching for $n$;
3. \hspace{1em} color $BG$ and $B\overline{G}$ grids along the path;
4. \hspace{1em} shadow the partly colored path;
5. \hspace{1em} pick out the $BG$ monotonic paths and insert them into the conflict graph;
6. \hspace{1em} **while** there exists any component $C$ in the conflict graph such that the number of nodes in $C$ is larger than $K$ or $C$ contains any odd cycle or the states of some grids in the $BG$ monotonic paths in $C$ have been changed **do**
7. \hspace{2em} color the paths in $C$;
8. \hspace{2em} shadow the newly colored paths;
9. \hspace{2em} remove $C$ from the conflict graph;
10. **end while**
11. **end for**

3. The states of some grids in the $BG$ monotonic paths in the component are now changed to $BG$ or $B\overline{G}$.

Condition 1 is used to reduce the running time. The reason for including condition 2 is that odd cycles indicate coloring conflicts (refer to Fig. 4.3). To solve these conflicts, some routing paths have to be sliced and stitches will be generated [28]. One consequence of adding a stitch is that the states of some grids around the stitch will become uncolorable. The router needs to be aware of the position of these uncolorable grids to find colorable routing paths for other nets. Odd cycles can be detected by using the linear time algorithm in [28]. Condition 3 is included to start the coloring as soon as there are color constraints on the $BG$ routing paths. If any component in the conflict graph satisfies one of the aforementioned conditions, the $BG$ routing paths belonging to the component are colored. Then, the newly colored routing paths are shadowed to reflect the DPT spacing rules.

Fig. 4.7 shows how the algorithm is applied to the example in Fig. 4.6. In this figure, shadowing is not shown for clearness. After $Path1$ is obtained, only $S_1$ and $T_1$ are colored (Fig. 4.7b). Then the $BG$ monotonic path $\{P_1, P_2\}$ is discovered and inserted into the conflict graph (Fig. 4.7c). Similarly, after the routing of $\{S_2, T_2\}$, the $BG$ path $\{P_3, P_4\}$ is also inserted into the conflict graph (Fig. 4.7e). Since $\{P_1, P_2\}$ conflicts with $\{P_3, P_4\}$, an edge is added between the corresponding nodes in the conflict graph. This procedure repeats until $\{P_5, P_6\}$ and $\{P_7, P_8\}$ are discovered and added to the conflict graph (Fig. 4.7g). After all the nets have been routed, the coloring of the component in the conflict graph in Fig. 4.7g is solved and the $BG$ paths are colored (Fig. 4.7h and Fig. 4.7i).
Next, an analysis of the running time of the algorithm in Algorithm 4.1 is presented. Since Dijkstra’s algorithm [30] is used for path searching, the running time complexity of the step in line 2 is \( O(g \cdot \log(g)) \), where \( g \) is the number of grids in the routing graph. Assuming there are in total \( N \) two-pin nets to be routed, the total running time of path searching is bounded by \( O(N \cdot g \cdot \log(g)) \). The complexity of the coloring and shadowing steps in line 3-4 and 7-8 is \( O(l) \), where \( l \) is the number of grids in the path. Therefore, the coloring and shadowing operations take in total \( O(L) \) time, where \( L \) is the sum of lengths (measured in terms of the number of grids) of all the nets. In line 5, finding out the \( BG \) monotonic paths also takes linear time and has a complexity of \( O(l) \). So, similarly to the coloring and shadowing operations, the operation of finding out the \( BG \) monotonic paths takes in total \( O(L) \) time. The components in the conflict graph are kept in a disjoint-set-forest data structure, with each component corresponding to a set. The heuristics of union-by-rank and path compression [30] have been implemented into the data structure. The data structure is associated with three basic operations, including make-set, union, and find-set. The make-set operation creates a new set. This operation is called when a new \( BG \) monotonic path is found. The union operation unites two disjoint sets. This operation is called when an edge is added between two vertices that belong to different components in the conflict graph. The find-set operation, when given a vertex in the conflict graph, returns the component that the vertex belongs to. This operation is called before adding an edge to the conflict graph to check if the two vertices that are going to be connected by the edge belong to the same component. For each net, the number of make-set, union, and find-set operations called in line 5 for maintaining the conflict graph data structure is bounded by \( O(l) \). Therefore, the total number of operations on the disjoint-set-forest data structure is bounded by \( O(L) \). Since for practical purposes the running time complexity of the operations on the disjoint-set-forest data structure is linear [30], the time taken by these operations is bounded by \( O(L) \). The condition that whether a component in the conflict graph contains more than \( K \) vertices needs to be checked only when a union operation is performed and can be answered in constant time. Similarly, the condition that whether a component contains odd cycles is checked only when an union operation is called. This check can be finished in \( O(K) \) time. Finally, checking whether the states of any \( BG \) grids have been changed to \( \overline{BG} \) or \( B\overline{G} \) can be done at the same as the shadowing operation, and hence does not cause any running time overhead. Therefore, the total running time overhead caused by the condition checking step in line 6 in Algorithm 4.1 is \( O(L \cdot K) \). After a component has been selected in line 6, the coloring of the component is done by using the Chaitin’s heuristic [34], which can be implemented in \( O(v + e + s) \) time, where \( v \) is the number of vertices in the component, \( e \) is the number of edges, and \( s \) is the total length (measured in terms
of the number of grids) of paths whose corresponding vertices are in the component. Since \( e \) is bounded by \( v^2 \), the running time of coloring a component is bounded by \( O(v^2 + s) \). Note that \( v \) is bounded by \( 2K \), because the number of vertices in a component is increased only when an union operation is performed and the components whose number of vertices is larger than \( K \) before the union operation have already been removed from the conflict graph in the previous rounds. Therefore, the running time overhead of coloring a single component is bounded by \( O(K^2 + s) \). Since the number of components in the conflict graph and the sum of \( s \) are both bounded by \( L \), the total running time overhead caused by coloring the components in the conflict graph is bounded by \( O(L.K^2) \). In conclusion, the overall running time of the algorithm in Algorithm 4.1 is bounded by \( O(L.K^2 + N.g.log(g)) \). Finally, since \( L \) is bounded by \( N.g \), the total running time complexity can be simplified to \( (N.g.(log(g) + K^2)) \).

### 4.4 Last Conflict Segment Recording

By shadowing, the path-searching algorithm is aware of the color constraints imposed by other nets. However, since coloring and shadowing are done after the path is obtained, the path-searching algorithm may not be able to detect within-path conflicts. One example is shown in Fig. 4.8. In the figure, Path1 and Path2 are two possible paths connecting \( S \) to \( T \). Both of them have the same number of stitches and wire length. However, Path1 is not colorable due to the potential color conflict as indicated in the figure. If the path searching algorithm is aware of this conflict, it will be able to choose Path2 over Path1. In this section, the Last Conflict Segment Recording (LCSR) method will be described. This method is able to help the path-searching algorithm detect within-path conflicts. The basic idea of this method is based on the following observations.

**Definition 4.5.** A monotonic path is called an “iso-state” monotonic path if the grids along the path have the same state.

**Observation 4.1.** There is no color conflict among the grids in a monotonic iso-state path. This is because there is no minimum spacing requirement between these grids.

**Observation 4.2.** There is no conflict between the grids of two monotonic iso-state paths touching at a stitch. This is because these two paths will be printed on different masks.

The above definition and observations are explained in Fig. 4.8. In the figure, Path1 contains three monotonic iso-state sub-paths: \( \{S, P_2\}, \{P_3, P_4\} \) and \( \{P_5, T\} \). As can be observed, there is no color conflict among the grids within each monotonic iso-state sub-path. Neither is there
any conflict between \{S, P_2\} and \{P_3, P_4\}. This is because \{S, P_2\} and \{P_3, P_4\} will be printed on different masks. Similarly, \{P_3, P_4\} does not conflict with \{P_5, T\}.

According to the above observations, conflicts should be searched for between grids belonging to different monotonic iso-state sub-paths which do not touch at a stitch. In Fig. 4.8, this indicates searching for conflicts between \{S, P_2\} and \{P_5, T\}. However, it would take too much time to search for the conflicts between all the grids belonging to the different monotonic iso-state sub-paths. To speed up the algorithm, we simply record the position of the last “conflict segment”. When the path searching algorithm tries to use a grid, it checks whether this grid conflicts with the “conflict segment”. If a conflict is detected, a penalty is added to the cost to discourage the routing algorithm from using the grid.

The core of the algorithm is how to update the position of the conflict segment when a detour or stitch is met. We use Fig. 4.8 and Fig. 4.9 to illustrate the basic update rules. For \textit{Path} 1 in Fig. 4.8, when the path searching algorithm propagates from \(P_4\) to \(P_5\), it meets a stitch. In this case, the conflict segment is updated to be the last segment of the last monotonic iso-state sub-path. In this example, the last monotonic iso-state sub-path before \{P_3, P_4\} is the one from \(S\) to \(P_2\). Its last segment is from \(P_1\) to \(P_2\). Therefore, the position of the conflict segment is updated to be from \(P_1\) to \(P_2\). The algorithm will then check if \(P_5\) is too close to the segment \(P_1P_2\). If so, a penalty is added to the cost of grid \(P_5\). Fig. 4.9 shows the case of a detour. In the figure, \{\(P_1, P_4\}\) is a monotonic iso-state sub-path. When the algorithm propagates from \(P_4\) to \(P_5\), it creates a detour which causes a color conflict. In this case, the conflict segment is updated to be the second to the last segment of

---

**Figure 4.8.** Avoiding conflict during path searching.

**Figure 4.9.** Conflict from detours.
the current monotonic iso-state sub-path. In this example, the current monotonic iso-state sub-path is \(\{P_1, P_4\}\). Its second to the last segment is the one from \(P_1\) to \(P_2\). So, when calculating the cost of grid \(P_5\), the algorithm will check if \(P_5\) conflicts with the segment \(P_1 P_2\).

In conclusion, if a stitch is met during path searching, the rule used in Fig. 4.8 is followed to update the position of conflict segment. Otherwise, if a detour is met, the rule used in Fig. 4.9 is followed. If neither a stitch nor a detour is found, the position of the conflict segment is not changed.

Finally, note that the update of the “conflict segment” does not increase the asymptotic time complexity of the path-searching algorithm, since the iso-state monotonic paths can be easily recognized during path propagation and only the positions of the last and the second to the last segments of the iso-state monotonic paths need to be recorded and updated.

### 4.5 Experimental Results

Table 4.2. Experimental results with “K” in the LCD algorithm set to 5.

<table>
<thead>
<tr>
<th>Design</th>
<th>Nets</th>
<th>Router</th>
<th>wire length (mm)</th>
<th>time (s)</th>
<th>No. of stitches</th>
<th>uncolorable wire length (um)</th>
<th>Ratio wire length</th>
<th>time (s)</th>
<th>No. of stitches</th>
<th>uncolorable wire length</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCC1</td>
<td>10855</td>
<td>LCD</td>
<td>52.5</td>
<td>500</td>
<td>33</td>
<td>1.57</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ LCSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DPFR</td>
<td>52.69</td>
<td>480</td>
<td>39</td>
<td>1.664</td>
<td>1.004</td>
<td>0.97</td>
<td>1.18</td>
</tr>
<tr>
<td>TCC2</td>
<td>9442</td>
<td>LCD</td>
<td>52.32</td>
<td>442</td>
<td>40</td>
<td>2.018</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ LCSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DPFR</td>
<td>52.65</td>
<td>412</td>
<td>48</td>
<td>2.112</td>
<td>1.006</td>
<td>0.93</td>
<td>1.2</td>
</tr>
<tr>
<td>TCC3</td>
<td>11074</td>
<td>LCD</td>
<td>41.61</td>
<td>396</td>
<td>34</td>
<td>1.62</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ LCSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DPFR</td>
<td>41.71</td>
<td>372</td>
<td>39</td>
<td>1.696</td>
<td>1.002</td>
<td>0.94</td>
<td>1.15</td>
</tr>
<tr>
<td>TCC4</td>
<td>10080</td>
<td>LCD</td>
<td>33.95</td>
<td>341</td>
<td>30</td>
<td>1.411</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ LCSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DPFR</td>
<td>34.08</td>
<td>324</td>
<td>35</td>
<td>1.504</td>
<td>1.004</td>
<td>0.95</td>
<td>1.17</td>
</tr>
<tr>
<td>TCC5</td>
<td>8999</td>
<td>LCD</td>
<td>33.98</td>
<td>427</td>
<td>30</td>
<td>1.364</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ LCSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DPFR</td>
<td>34.15</td>
<td>396</td>
<td>35</td>
<td>1.44</td>
<td>1.005</td>
<td>0.93</td>
<td>1.17</td>
</tr>
</tbody>
</table>
The techniques proposed in this paper have been implemented in C and tested on a 3.0GHz Linux machine with 1G RAM. Five circuits are scaled down to 32 nm for evaluation. These test circuits are generated from the ISPD07 benchmark circuits by randomly picking up the regions whose average wire density is close to that of the original benchmarks. First, global routing is performed on these nets by a sequential global router based on rip-up and reroute. Then the results are output to the detailed router for detailed routing. For comparison, the algorithm DPFR proposed in [1] has been implemented. The results are shown in Table 4.2. In the experiment, the “K” value used in the LCD technique is set to 5.

According to the experimental results, compared with DPFR, our algorithm (LCD+LCSR) is able to reduce the number of stitches by around 15~20%. This is mainly due to the wiser color assignment achieved by LCD. Another benefit from LCD is the shorter wire length, though the reduction is very small. Importantly, our algorithm produces shorter uncolorable wire length, as a result of LCSR. The reduction ranges from 5% to 7%. The running time of algorithm is on average 4% slower than DPFR, mainly due to the overhead of the maintenance and coloring of the conflict graph in the LCD technique.

### 4.6 Conclusions and Future Work

In this work, an enhanced DPT-friendly detailed routing algorithm has been described. In particular, two novel techniques, including lazy color decision and last conflict segment recording have been proposed. Experimental results have shown the efficiency of the algorithm.

One interesting future research direction is to integrate the Optical Proximity Correction (OPC) consideration into the algorithm. In the existing work about OPC-friendly detailed routing, the metal patterns on a layer are placed in such a way that the interference between the patterns is minimized. However, in double-patterning technology the polygons on a layer may not be on the same mask. This requires the OPC-friendly detailed routing algorithms to be aware of the mask assignment solutions produced by the DPT-friendly detailed routing algorithms.
Chapter 5

Jumper Insertion under Antenna Ratio and Timing Constraints

5.1 Introduction

Antenna effect is a phenomenon that may cause yield loss and reliability problems during the manufacturing of MOS integrated circuits. Some preliminary knowledge about antenna effect has been introduced in Section 1.3.5. Typically, antenna effect is evaluated by the ratio between the area of the conductor wires that are not connected to any diffusion region and the area of the gates connected to the conductor wires [35]. This ratio is called the antenna ratio. To reduce the probability of gate damage caused by the antenna effect, chip manufacturers usually provide some antenna rules which specify the maximum allowable value of the antenna ratio. A violation of such rules is called an antenna violation.

As introduced in Section 1.3.5, one of the popular approaches to fixing the antenna violations is jumper insertion. Some algorithms, including [36], [37], [38], [39] and [40], have been proposed recently to solve the jumper insertion problem. To understand the differences between the existing work, it is important to have a clear idea about the different definitions of antenna rules. Based on whether the charge collected by the lower metals is counted, antenna rules can be divided into two types: cumulative and non-cumulative. This is illustrated in Fig. 5.1. Fig. 5.1a shows an example routing tree. Fig. 5.1b shows the manufacturing steps of the metal wires. In the first step, segment $e_1$ on M1 is manufactured. After this step, for both cumulative and non-cumulative antenna rules, the antenna size consist of the area of $e_1$ that is exposed to the plasma environment. For simplicity, let us assume that only the top area of $e_1$ is exposed and hence counted in the antenna size. In the second step of the process, segment $e_2$ on M2 is manufactured. After this step,
Figure 5.1. An illustration of the cumulative and non-cumulative antenna rules.

Based on the type of the antenna rule, the antenna size has two possible values. If the antenna rule is cumulative, the antenna size is the sum of the top areas of $e_1$ and $e_2$. On the other hand, if the antenna rule is non-cumulative, the antenna size only consists of the top area of $e_2$. This is because, in the processes where the antenna effect is non-cumulative, the charge collected by the metal on lower layers has dissipated prior to the manufacturing of the next layer [41]. Finally, after the manufacturing of the metal on M3, the charge collected in the plasma process is discharged through the diffusion region of the driver. As can be observed in the example in Fig. 5.1, different antenna rule definitions calculate the antenna size in different ways.

Besides classifying the antenna rules as cumulative and non-cumulative, there is another kind of classification based on whether the charge sharing effect (i.e., the charge collected by a conductor wire is shared by all the gates connected to the wire) is considered. For the antenna rules that do not consider the charge sharing effect, the severeness of antenna effect is measured in terms of antenna size, while for those that take into account the charge sharing effect, antenna effect is measured by the ratio between the antenna size and the total gate area. This is explained in Fig. 5.2, which shows the different antenna effect expressions for the routing tree in Fig. 5.1. Note that after the manufacturing of $e_2$, there are four possible expressions for the antenna effect depending on the type of the antenna rules.
Based on the aforementioned classification of antenna rules, the existing work of jumper insertion can be divided into two categories. In the first category (including [36], [37] and [40]), the antenna rule is expressed as antenna size rather than antenna ratio. For this kind of antenna rule, the jumper insertion problem can be solved optimally by a greedy algorithm based on tree partitioning. The problem with this antenna rule definition is that the charge sharing effect is not considered [39] and, therefore, tends to be conservative [40]. In the second category, the antenna rule is expressed as an antenna ratio. An optimal algorithm is proposed in [39] to solve the jumper insertion problem under the constraint of bounded antenna ratios. However, the algorithm in [39] is developed for a single subtree, while in practice we need to simultaneously deal with multiple subtrees on different layers. This is explained with the example in Fig. 5.3. Fig. 5.3a shows an example routing tree. Fig. 5.3b and Fig. 5.3c show the produced subtrees after the manufacturing of M1 and M2. To free the routing tree in Fig. 5.3a from antenna violation, the antenna ratios of all the subtrees in Fig. 5.3b and Fig. 5.3c must be smaller than the value specified by the antenna rule. Otherwise, some gates may be damaged. For example, if the subtree t1 in Fig. 5.3b violates the antenna rule, the gate S1 may be damaged during the manufacturing of M1. Similarly, if t2 in Fig. 5.3c violates the antenna rule, S1 and S2 may be damaged during the manufacturing of M2. What makes things complicated is that the jumper insertion solutions of those subtrees are “coupled” with
each other. In Fig. 5.3, both t1 and t2 contain e1. This indicates that a jumper on e1 may change the antenna ratios of both t1 and t2. This coupling between the jumper insertion solutions of different subtrees makes it necessary to consider all the subtrees simultaneously. While the algorithm in [39] is able to optimally solve the jumper insertion problem for a single subtree, it does not mention how to consider all the subtrees simultaneously. Another existing work that uses antenna ratio as the antenna rule is [38]. The problem with [38] is that only the maximal subtrees are considered in the antenna rule definition. For the example in Fig. 5.3, this means that [38] only considers the subtree in Fig. 5.3c but not the subtrees in Fig. 5.3b. However, depending on the nature of the antenna effect (i.e., cumulative or non-cumulative) and the area of the gates S1 and S2, it is possible that the antenna ratio of the subtree in Fig. 5.3c satisfies the antenna rule while those of the subtrees in Fig. 5.3b break the antenna rule. This means that freeing only the maximal subtrees from antenna violation is not enough to guarantee that the whole routing tree is free from antenna violations.

Another drawback of the existing work is that the jumpers of a routing tree are always assumed to be placed on or above the highest layer of the tree. This assumption is reasonable when the antenna effect is expressed as cumulative antenna size. However, in the situations where the antenna rules are expressed as antenna ratio, this assumption tends to significantly increase the number of vias added by the jumpers. Finally, adding jumpers causes extra delay to the circuits because of the large via resistance. But none of the existing algorithms is aware of the impact of this delay.

In this work, a jumper insertion algorithm is proposed to solve the problems mentioned above. The major features of the algorithm are summarized below.

1. The algorithm considers all the subtrees simultaneously by performing Lagrangian relaxation and using an algorithm similar to the one in [39] as one of the internal engines.

2. Unlike the existing work, the algorithm allows the jumpers to be placed on any routing layer, which helps to significantly reduce the number of vias added by the jumpers.

3. The algorithm is aware of the delay caused by the resistance of the vias in the jumpers.

Experimental results show that, by allowing the jumpers to be placed on any layer, the algorithm is able to reduce the number of vias by 50%. The experiments also show that the timing-aware version of the algorithm is better at satisfying the timing constraints.

This chapter is organized as follows. The formal definitions of antenna ratio and antenna violation are given in Section 5.2. The problem formulation is presented in Section 5.3. The details
of the algorithm are described in Section 5.4. The experimental results are reported in Section 5.5. Finally, this chapter is concluded in Section 5.6.

5.2 Preliminaries

In this section, the formal definitions of antenna ratio and antenna violation are presented.

Definition 5.1. Given a routing tree $T$, the notation $T^l$ is used to denote the set of maximal subtrees of $T$ that are formed by the wires on or below layer $l$.

Definition 5.2. Given a wire segment $e$, the notation $A(e, l)$ is used to represent the contribution of $e$ to the antenna area of the routing tree $t$ that covers $e$ and belongs to $T^l$.

Note that the form of $A(e, l)$ is dependent on the manufacturing technology (e.g., whether the antenna effect is cumulative, whether the sidewalls of a wire contribute to the antenna effect, etc.). In this work, it is assumed that the function $A(e, l)$ has already been provided by the circuit manufacturer.

Definition 5.3. The antenna ratio of a subtree $t$ in $T^l$ is defined as: $AR = \sum_{e \in t} A(e, l)/GA(t)$ where $A(e, l)$ is as defined in Definition 5.2 and $GA(t)$ is the total area of the gates connected to $t$.

The above definitions can be illustrated with the example in Fig. 5.3. Fig. 5.3a shows the original tree. Fig. 5.3b shows the set of subtrees in $T^{M1}$ and Fig. 5.3c shows the subtrees in $T^{M2}$. The antenna ratio of the subtree $t2$ is: $(A(e1, M2) + A(e2, M2) + A(e3, M2) + A(e4, M2))/GT(t2)$. $A(e1,M2)$ is the contribution of segment $e1$ to the antenna area of the subtrees in $T^{M2}$. $A(e2,M2)$, $A(e3,M2)$ and $A(e4,M2)$ are similarly defined.

The definition of antenna violation is presented below.

Definition 5.4. Given a routing tree $T$, the set $M$ of routing layers and a maximum allowable antenna ratio $AR_{max}$, $T$ is suffering from antenna violation if there exists a subtree $t$ in $T^l (\forall l \in M)$ that connects to gate oxides but no diffusion region and the antenna ratio of $t$ is larger than $AR_{max}$.

5.3 Problem Formulation

In this work, the following problem is targeted.
Problem 5.1. Given a routing tree $T$ that suffers from antenna violations, a set $Q$ of candidate locations of jumper insertion, and a library $J_q$ of jumpers for each candidate location $q \in Q$, select jumpers from the jumper libraries to fix the antenna violations such that the timing slacks of the sinks are still positive and the total number of vias added by the selected jumpers is minimized.

The above problem formulation is illustrated below.

5.3.1 Candidate location selection

The candidate locations of jumper insertion can be chosen from the low congested regions round the given routing tree. One method is to, for each segment, select the two low congested regions that are closest to the end points of the segment. One example is shown in Fig. 5.4. In the figure, the routing tree contains three segments. The rectangles $q_1$, $q_2$, $q_3$ and $q_4$ represent the low congested regions. For segment $e_1$, $q_1$ is the region that is closest to the left end point. Similarly, $q_3$ is the region that is closest to the right end point. Therefore, $q_1$ and $q_3$ will be selected as the candidate locations of jumper insertion. The reason why the positions closest to the end points of a segments are selected is that these positions can potentially isolate most part of the segment from the rest of the routing tree.

![Figure 5.4](image.png)

Figure 5.4. An example of candidate location selection.

5.3.2 Jumper library

The jumper library at a candidate location is organized as a table. In such a table, each entry contains the information of a jumper. One single jumper is recorded for each possible highest layer that it passes through (so there are exactly $H - l + 1$ entries, where $l$ is the layer that the original routing path in this candidate location passes through, and $H$ represents the highest layer in the layout). Some entries in the table may be empty, indicating that no jumper is allowed to pass through the corresponding layers. Note that, since the original routing path in the candidate location is also stored in the table (in the entry for layer $l$), at least one entry is not empty.
Fig. 5.5 shows an example of jumper insertion problem. Fig. 5.5a shows a given routing tree. There are five routing layers: M1, M2, M3, M4, and M5. \{e1, e2, e3, e4, e5\} is the set of segments. S1 and S2 are the sinks, and S0 is the source. The squares q1, q2, and q3 are the three candidate locations of jumper insertion. Fig. 5.5b shows an example jumper library for the candidate location q2. Since the original routing path in q2 is on layer M3 and the highest routing layer of the layout is M5, the table in Fig. 5.5b contains three entries from M3 to M5. In the entry of M3, the original routing path in q2 is stored. Because of the routing blockage \( B \) on layer M4, the entry M4 is empty. The entry for M5 stores the routing path of the jumper whose highest layer is M5.

The jumper library can be directly provided by the circuit designer or by some detailed routing tools. One method to use the detailed routing tools to generate the jumper library is illustrated in Fig. 5.6. Fig. 5.6a shows a simple routing tree. The rectangle q1 is the candidate location of jumper insertion. To build the jumper library in q1, one can first build a 3D routing grid graph for q1. This is shown in Fig. 5.6b. The dashed segment in Fig. 5.6b represents the original routing path. The bold segments in Fig. 5.6c show a jumper whose highest layer is M2. Note that this jumper is nothing but a pair of vertex-disjoint paths from v to n. Therefore, to find the jumper whose highest layer is M2 and number of vias is minimum, one can simply disable the nodes above M2 in the 3D routing graph, and then search for the shortest pairs of vertex-disjoint paths from v to all the nodes.
on M2, and finally, among these pairs, choose the one with the least number of vias as the jumper. The problem of shortest pair of vertex-disjoint paths can be solved by the Suurballe’s algorithm and the vertex-splitting technique [42].

The objective of the problem formulation is to select some jumpers from the jumper libraries to fix the antenna violations while satisfying the timing constraints and minimizing the total number of vias in the jumpers.

### 5.3.3 An ILP Formulation

In this section, an ILP formulation for the problem is introduced. The notations used in the formulation are summarized in Table 5.1.

In the table, the variables above $C_{i,j,q}^p$ (including $C_{i,j,q}^p$) are constants for a given jumper insertion problem. The variables below $C_{i,j,q}^p$ are adjusted in the ILP formulation to optimize the number of vias while satisfying the antenna ratio and timing constraints. In the table, $D_j^q$ is the delay caused by the jumper $j$. In this work, this delay is estimated as $R.C_{\text{down}}$, where $R$ is the total resistance of the vias in the jumper and $C_{\text{down}}$ is the total downstream capacitance at the jumper position in the original routing tree.

The ILP formulation is shown in Fig. 5.7. Before explaining the formulation, it is needed to first define the *compatibility* between the jumper selection solutions of two subtrees.

$$\min \sum_{q \in Q} \sum_{j \in J^q} V_j^q \times c_j^q$$

$$\text{S.t.} \quad \sum_{q \in Q_k} \sum_{j \in J^q_k} D_j^q c_j^q \leq W_k \quad (\forall k \in K) \quad (5.3.1)$$

$$\sum_{p \in P} s_{p,i} = 1 \quad (\forall i \in S^p) \quad (5.3.2)$$

$$c_j^q = \sum_{i \in S^p} C_{p,i,j}^{q,p} s_{p,i} \quad (\forall p \in P, q \in Q^p, j \in J^p) \quad (5.3.3)$$

$$\sum_{j \in J^q} c_j^q = 1 \quad (\forall q \in Q) \quad (5.3.4)$$

$$c_{p,j}^q - c_j^q = 0 \quad (\forall q \in Q, j \in J^q, p \in P^q) \quad (5.3.5)$$

$$s_{p,i} = 0 \text{ or } 1 \quad (\forall p \in P, i \in S^p) \quad (5.3.6)$$

$$c_j^q = 0 \text{ or } 1 \quad (\forall q \in Q, j \in J^q) \quad (5.3.7)$$

Figure 5.7. An ILP formulation of the jumper insertion problem.
Table 5.1. A summary of the notations used in the ILP formulation in Fig. 5.7.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>The given routing tree.</td>
</tr>
<tr>
<td>$M$</td>
<td>The set of routing layers in the layout.</td>
</tr>
<tr>
<td>$T^l$</td>
<td>The set of maximal subtrees of $T$ formed by the wires on or below layer $l$.</td>
</tr>
<tr>
<td>$P$</td>
<td>$\bigcup_{l \in M} T^l$</td>
</tr>
<tr>
<td>$K$</td>
<td>The set of sinks in $T$.</td>
</tr>
<tr>
<td>$W^k$</td>
<td>The timing slack of sink $k \in K$.</td>
</tr>
<tr>
<td>$Q$</td>
<td>The set of candidate locations in $T$ for jumper insertion.</td>
</tr>
<tr>
<td>$J^q$</td>
<td>The set of jumpers in the jumper library at $q \in Q$.</td>
</tr>
<tr>
<td>$V^q_j$</td>
<td>The number of vias in the jumper $j \in J^q$.</td>
</tr>
<tr>
<td>$D^q_j$</td>
<td>The delay of jumper $j \in J^q$.</td>
</tr>
<tr>
<td>$Q^p$</td>
<td>The set of candidate locations in $Q$ that are covered by the subtree $p \in P$.</td>
</tr>
<tr>
<td>$Q_k$</td>
<td>The set of candidate locations in $Q$ that reside along the path between the source of $T$ and the sink $k \in K$.</td>
</tr>
<tr>
<td>$P^q$</td>
<td>The set of subtrees in $P$ that cover the candidate location $q \in Q$.</td>
</tr>
<tr>
<td>$S^p$</td>
<td>The set of jumper selection solutions that are able to fix the antenna violations of the subtree $p \in P$.</td>
</tr>
<tr>
<td>$C^q_{p,i,j}$</td>
<td>Integer variable. 1 if jumper $j \in J^q$ is selected in solution $i \in S^p$; 0 otherwise.</td>
</tr>
<tr>
<td>$s^q_{p,i,j}$</td>
<td>Integer variable. 1 if solution $i \in S^p$ is selected; 0 otherwise.</td>
</tr>
<tr>
<td>$c^q_{p,j}$</td>
<td>Integer variable. 1 if jumper $j \in J^q$ is selected by the chosen solution in $S^p$; 0 otherwise.</td>
</tr>
<tr>
<td>$c^q_j$</td>
<td>Integer variable. 1 if jumper $j \in J^q$ is selected in the final solution for the routing tree $T$; 0 otherwise.</td>
</tr>
</tbody>
</table>
**Definition 5.5.** Given two subtrees and the jumper selection solution for each of them, the two solutions are called to be **compatible** with each other if they select the same jumpers in the candidate locations shared by the two subtrees.

![Diagram](image)

**Figure 5.8.** An example for illustrating the compatibility between the jumper insertion solutions of different subtrees.

The definition of compatibility is explained with the example in Fig. 5.8. There are four layers in Fig. 5.8. Fig. 5.8a shows a routing tree $T$. The rectangles $q_1$, $q_2$, and $q_3$ are the candidate locations of jumper insertion. Fig. 5.8b and 5.8c show the subtrees of $T$ formed after the manufacturing of $M_2$ and $M_3$. Note that $q_1$ and $q_2$ are shared by the two subtrees. To free $T$ from antenna violations, both of the subtrees in Fig. 5.8b and 5.8c must satisfy the antenna rule. The jumper selection solutions of the two subtrees are called to be **compatible** with each other if the jumpers selected in $q_1$ and $q_2$ in the solution for the subtree in Fig. 5.8b are also selected in the solution for the subtree in Fig. 5.8c.

Now, the ILP formulation in Fig. 5.7 can be expressed as follows: for each subtree $p \in P$ ($P$ is as defined in Table 5.1), choose a solution from $S^p$ ($S^p$ is as defined in Table 5.1) such that the total number of vias added by the jumpers is minimized, the chosen jumper selection solutions are compatible with each other, and the total extra delay caused by the jumpers along the path between a sink and the source is smaller than the sink’s timing slack.

In Fig. 5.7, inequation (5.3.1) represents the timing constraint. Note that, when calculating the delay caused by a jumper, only the impact of the via resistance is considered. With this simplification and the Elmore delay model, the extra delay along a sink-source path can be obtained by adding together the delays of the jumpers on the path. Equation (5.3.2) states that only one solution in $S^p$ is selected. Equation (5.3.3) is for the calculation of $C^{p}_{i,j,q}$. Equation (5.3.4) requires that only one jumper is selected from the jumper library at each candidate location. Note that it can be guaranteed that at least one “jumper” exists in each jumper library. This is because the jumper library of a location also includes the original routing path. Equation (5.3.5) is used to enforce
the compatibility among the chosen jumper selection solutions. (5.3.6) and (5.3.7) are the integer constraints.

5.4 Algorithm

5.4.1 Overview

In this section the algorithm to solve the ILP problem in Fig. 5.7 is introduced. The algorithm is based on Lagrangian Relaxation (LR). Within the LR method, there are two main approaches: the Classical Lagrangian Relaxation (CLR) method and the Augmented Lagrangian Relaxation method (ALR)[43]. The basic idea of CLR is to remove difficult constraints by adding linear penalties for violations to the objective function. One important property of the CLR method is that it yields a lower bound on the optimal solution. However, the CLR method may not return a feasible solution due to the duality gap. In contrast to CLR, ALR additionally adds a quadratic penalty for violations to the objective function. As a result, the ALR method is more likely to yield feasible solutions. The problem with the ALR method is that no lower bound on the optimal solution is obtained. Please refer to [44] for a more comprehensive introduction of the Lagrangian relaxation technique.

In this chapter, CLR and ALR are combined together to solve the ILP problem. In the first step, the CLR method is used to find an initial solution and a lower bound on the optimal solution. If the obtained solution is feasible, the algorithm stops. Otherwise, for those difficult problems the ALR method is used to solve the ILP problem. The initial solution used in ALR is the solution returned by CLR. The details are presented in the rest of this section.

5.4.2 The classical Lagrangian relaxation method

\[
\max_{\lambda_{p,j}} \left[ \min_{s_{p,i}, c_{j}} \left( \sum_{p \in P} \sum_{q \in Q} \sum_{j \in J} \left( c_{p,j}^q \left( \frac{V_{j}^q}{2|P_{j}|} \right) + \lambda_{p,j}^q \right) \right) \right]
\]

S.t. (5.3.1),(5.3.2),(5.3.3),(5.3.4),(5.3.6),(5.3.7) in Fig.5.7

Figure 5.9. The problem obtained by applying CLR on the problem in Fig. 5.7.
The problem obtained after performing classical Lagrangian relaxation is shown in Fig. 5.9.

**Algorithm 5.1** The subgradient algorithm to solve the problem in Fig. 5.9.

1: initialize $\lambda^q_{p,j}$ ($\forall q \in Q, p \in P^q, j \in J^q$);
2: initialize step;
3: n=0;
4: max_dual_cost = 0;
5: best_dual_sol = NULL;
6: while $n < \text{MAX\_ITERATION}$ do
7: For the given value of $\lambda^q_{p,j}$ solve the problem in Fig. 5.9 by solving the problems in Fig. 5.10 and Fig. 5.11 individually. Suppose that the obtained value of the objective function in Fig. 5.10 is $\text{dual\_cost}$ and the solution to the problem in Fig. 5.11 is $\text{dual\_sol}$;
8: if ($\text{dual\_cost} > \text{max\_dual\_cost}$) then
9: $\text{max\_dual\_cost} = \text{dual\_cost}$;
10: $\text{best\_dual\_sol} = \text{dual\_sol}$;
11: end if
12: $\text{tmp}\_pc = \sum_{q \in Q} \sum_{j \in J} \sum_{p \in P^q} (c^q_{p,j} - c^q_j)^2$;
13: if ($\text{tmp}\_pc == 0$) then
14: return($\text{max\_dual\_cost}, \text{best\_dual\_sol}$);
15: else
16: $\lambda^q_{p,j} = \lambda^q_{p,j} + \text{step} \times \left( c^q_{p,j} - c^q_j \right) / \sqrt{\text{tmp}\_pc}$;
17: $\text{step} = \text{step} \times 0.8$;
18: end if
19: $n = n + 1$;
20: end while
21: return ($\text{max\_dual\_cost}, \text{best\_dual\_sol}$);

$$
\min_{s_{p,i}} \sum_{p \in P} \sum_{q \in Q} \sum_{j \in J^q} c^q_{p,j} \left( \frac{V^q_j}{2|P^q|} + \lambda^q_{p,j} \right)
$$

S.t. (5.3.2),(5.3.3) and (5.3.6) in Fig.5.7

Figure 5.10. The subproblem of Fig. 5.9 with respect to $s_{p,i}$.

In Fig. 5.9, $\lambda^q_{p,j}$ ($\forall q \in Q, p \in P^q, j \in J^q$) are the Lagrangian multipliers. The problem in Fig. 5.9 is obtained from the problem in Fig. 5.7 by relaxing constraint (5.3.5) and doing some substitutions. It is the well known that classical Lagrangian relaxation gives a concave optimization problem, which can then be solved with the subgradient method [44]. This is described in Algorithm 5.1.
\[
\min_{c_j^q} \sum_{q \in Q} \sum_{j \in J^q} \left[ c_j^q \left( \frac{V_j^q}{2} - \sum_{p \in P^q} \lambda_{p,j}^q \right) \right]
\]
S.t. (5.3.1), (5.3.4) and (5.3.7) in Fig. 5.7

Figure 5.11. The subproblem of Fig. 5.9 with respect to \( c_j^q \).

After proper initialization (lines 1-5), the loop between 6 and 19 solves the problem in Fig. 5.9 in an iterative manner. In line 7 of Algorithm 5.1, we solve the problem in Fig. 5.9 for a given value of \( \lambda_{p,j}^q \). In this case, the problem in Fig. 5.9 can be decomposed into the two independent subproblems in Fig. 5.10 and Fig. 5.11.

The problem in Fig. 5.10 is obtained from Fig. 5.9 by keeping only the part that is related to \( s_{p,i} \). Similarly, the problem in Fig. 5.11 is obtained by keeping only the part that related to \( c_{j}^q \).

The problem in Fig. 5.10 can be expressed as follows: perform jumper selection for each subtree in \( P \) (\( P \) is as defined in Table 5.1) to fix the antenna violations of the subtrees while minimizing the total cost of the selected jumpers. In this problem, the cost of a jumper is as defined in the parenthesis in the objective function in Fig. 5.10. Note that in this problem the jumper selection solutions of different subtrees do not need to satisfy the compatibility constraint. This problem can be solved by independently applying a modified version of the algorithm in [39] to each subtree. This will be introduced in detail in Section 5.4.3.

The problem in Fig. 5.11 can be expressed as follows: select a jumper from each jumper library such that the total cost of the jumpers is minimized and the timing constraints are satisfied. In this problem, the cost of a jumper is as defined in the parenthesis in the objective function in Fig. 5.11. Note that in this problem the selected jumpers are not required to fix the antenna rule violations. In this work, an algorithm similar to the one in [45] is used. The basic steps of the algorithm will be described in Section 5.4.4.

In Algorithm 5.1, after solving the problems in Fig. 5.10 and Fig. 5.11, the value of \( \text{max_dual} \) and \( \text{best_dual_sol} \) is updated. Then it is tested whether the obtained solution satisfies the relaxed constraint ((5.3.5) in Fig. 5.7). This test is performed in line 12 and 13. If the test is passed, the program returns because the obtained solution is a feasible and optimal solution to the ILP problem. Otherwise, the value of \( \lambda_{p,j}^q \) is updated along the subgradient, and the next round of iteration starts.
5.4.3 The algorithm for the problem in Fig. 5.10

In this section, the algorithm for the problem in Fig. 5.10 is introduced. Note that in Fig. 5.10 the jumper insertion problems of different subtrees are decoupled. Actually, this decoupling is the reason why the technique of Lagrangian relaxation is used. Therefore, in this section only the jumper insertion algorithm for a single subtree is introduced. This algorithm can then be used to solve the jumper insertion problem for each subtree. Also note that the algorithm in this section is different from the one in [39], although they use similar solution forms. In particular, the algorithm in [39] is for the minimization of the total number of jumpers while the algorithm in this section is to minimize the total cost of the jumpers. This difference is important, because the number of jumpers is a discrete variable and is bounded by a polynomial function of the number of edges in the tree. As a result, the algorithm in [39] is able to guarantee a polynomial time complexity. However, when the total cost of jumpers is used as the optimization objective, the polynomial bound no longer holds. Besides, the algorithm in this section is more general. In [39], an enumeration procedure is used to propagate the candidate solutions along the edges of the routing tree. In the algorithm of this section, this enumeration procedure is replaced by a more general solution-propagation step. The details of the algorithm are introduced in the rest of this section.

![Diagram](image_url)

Figure 5.12. An example for illustrating the solution propagation procedure.

The algorithm follows the basic flow of dynamic programming. Candidate solutions are propagated along the subtree from the leaves towards the root. Then, at the root, the candidate solution with the minimum cost is chosen. Finally, a back-trace procedure is performed to choose the optimal jumper insertion solution at each candidate location of jumper insertion. The example in Fig. 5.12 illustrates this basic flow. Fig. 5.12a shows a routing tree, where \( n_0 \) is the source. The squares \( p_1 \) and \( p_2 \) represent the candidate locations of jumper insertion. The green nodes and edges in the figure highlight a subtree. Fig. 5.12b and 5.12c show how the dynamic programming
algorithm is performed on this subtree. In the first step, the algorithm chooses a node in the subtree as the root. This node is the one that is closest to the source of the original routing tree. For example, in the subtree in Fig. 5.12b, node $n_3$ is chosen as the root since it is the closest node to the source $n_0$ of the original routing tree. After the root is chosen, new nodes are added at the intersection points between the subtree edges and the candidate locations of jumper insertion. For example, in Fig. 5.12b, nodes $A$, $B$, $C$, and $D$ are added to the subtree. Then, a bottom-up solution propagation procedure is performed. The arrows in Fig. 5.12b show the directions of the propagation. Finally, the algorithm does a back-trace from the root of the subtree towards the leaves to choose the optimal jumper insertion solution at each candidate location. This is indicated by the arrows in Fig. 5.12c.

Generally, when propagating the candidate solutions from the leaves towards the root, three basic steps are involved. They are solution initialization, solution growth along an tree edge, and solution merging. These basic steps are explained as follows.

**Solution Initialization**

In this step, a set of candidate solutions are initialized at the leaves of the subtree. The candidate solution is a heptuple $(cost, as, ga, g, children, jumper)$, where “cost” represents the total cost of the jumpers that have been chosen in the candidate solution, “as” and “ga” are the total antenna size and gate area that are connected to the current node and have not been blocked by the chosen jumpers, “g” is a mark indicating if “ga” is 0, “children” is a pointer that points to the list of child solutions of the current solution, and “jumper” is a pointer pointing to the jumper selected at the current location. This pointer is used in back-trace to find out the optimal jumper solution at each candidate location of jumper insertion. In the initialization step, “cost” and “as” are set to 0. “children” and “jumper” are set to NULL. If a leaf in the subtree is a sink in the original routing tree, then, in the initial candidate solution for this leaf, “ga” is set to the area of the gate at the sink and “g” is set to 1. Otherwise, both “ga” and “g” are set to 0.

**Solution Growth Along an Tree Edge**

There are two cases in this step. One is to grow the solution set along a normal tree edge while the other one passes through a candidate location of jumper insertion. For example, in Fig. 5.12b, these two cases respectively correspond to propagating the solution set along $n_1A$ and $AB$. The algorithms for these two cases are described in Algorithm 5.2 and 5.3.
Algorithm 5.2 The algorithm for propagating the solutions along a normal tree edge.

Input: The set $C_m$ of the candidate solutions at node $m$ and a normal tree edge $mn$.
Output: The set $C_n$ of the candidate solutions at node $n$.

1: $C_n = \emptyset$;
2: for each candidate solution $sol_m$ in $C_m$ do
3:   build a new solution $sol_n$;
4:   $sol_n\.cost = sol_m\.cost$;
5:   $sol_n\.as = sol_m\.as + AS(mn)$;
6:   $sol_n\.ga = sol_m\.ga$;
7:   $sol_n\.g = sol_m\.g$;
8:   add $sol_m$ to $sol_n\.children$;
9:   $sol_n\.jumper = NULL$;
10: end for
11: return $C_n$;

Algorithm 5.3 The algorithm for propagating the solutions along a tree edge containing a candidate jumper insertion location.

Input: The set $C_m$ of candidate solutions at node $m$ and a tree edge $mn$ that passes through a candidate location $p$ of jumper insertion.
Output: The set $C_n$ of candidate solutions at node $n$.

1: for each candidate solution $sol_m$ in $C_m$ do
2:   for each jumper $j$ in the jumper library for $p$ do
3:     build a new solution $sol_n$;
4:     $sol_n\.cost = sol_m\.cost + j\.cost$;
5:     $sol_n\.as = sol_p\.as + AS(mn)$;
6:     if $j$ is a real jumper for the current subtree then
7:       if $(sol_m\.as - AR_{max}.sol_m\.ga) \leq 0$ then
8:         $sol_n\.ga = 0$;
9:         $sol_n\.g = 0$;
10:     else
11:       break;
12:     end if
13: else
14:     $sol_n\.ga = sol_m\.ga$;
15:     $sol_n\.g = sol_m\.g$;
16: end if
17: add $sol_m$ to $sol_n\.children$;
18: $sol_n\.jumper = j$;
19: add $sol_n$ to $C_n$;
20: end for
21: end for
In Algorithm 5.2, \( AS(\overline{mn}) \) represents the contribution of the edge \( \overline{mn} \) to the antenna size. Note that in the algorithm in Algorithm 5.2, when propagating the solution along a normal tree edge, only the antenna size is incremented.

![Diagram](image)

Figure 5.13. An explanation about the case where one subtree belongs to more than one \( T^l \).

Algorithm 5.3 shows the algorithm for propagating the solutions along an edge containing a candidate location of jumper insertion. The if-else structure between line 6 and 16 is the key in the algorithm. In line 6, it is checked if the jumper \( j \) is a real jumper for the current subtree. A jumper is called a real jumper for a subtree \( t \), if \( t \) belongs to \( T^l \) and the highest layer of the jumper is higher than \( l \), where \( T^l \) is as defined in table 5.1. If \( j \) is a real jumper, it serves as a cut in the subtree. In this case, it is further checked if adding this jumper breaks the antenna rule. The notation \( AR_{max} \) in line 6 represents the maximum allowable antenna ratio. If the antenna rule is not broken, \( sol_n.ga \) and \( sol_n.g \) are set to 0. Otherwise, the program jumps out of the inner loop and continues to process the next jumper in the jumper library. One thing to note here is that a subtree may belong to more than one \( T^l \). This makes the judgement of real jumpers a little complicated. This is explained in Fig. 5.13. Fig. 5.13a shows a routing tree. Fig. 5.13b shows the subtree on layer \( M1 \). Note that this subtree belongs to both \( T^{M1} \) and \( T^{M2} \). Therefore, this subtree will be considered twice in the algorithm. When the subtree is considered as a member of \( T^{M1} \), the jumper whose highest layer is \( M2 \) is a real jumper. However, when the subtree is considered as a member of \( T^{M2} \), the jumper whose highest layer is \( M2 \) is not a real jumper any more.

**Solution Merging**

When a node in the subtree has more than one child, the solutions from the child nodes are merged together. For example, in Fig5.12, to form the solutions at node \( n_3 \), the solutions propagated from \( B \) and \( D \) have to be merged with each other. This merging process is described in Algorithm 5.4. Note that in the algorithm in Algorithm 5.4 it is assumed that node \( q \) has only two children: node \( n \) and node \( m \). This assumption can always be satisfied by add some virtual nodes to make the tree binary.
Algorithm 5.4 The algorithm for merging the solutions propagated from child nodes.

Input: The set $C^m_q$ of candidate solutions at node $q$ propagated from node $m$, and the set $C^n_q$ of candidate solutions at node $q$ propagated from node $n$.

Output: The set $C^*_q$ of candidate solutions at node $q$ merged from $C^m_q$ and $C^n_q$.

1: $C_q = \emptyset$;
2: for each candidate solution $sol^m_q$ in $C^m_q$ do
3: for each candidate solution $sol^n_q$ in $C^n_q$ do
4: build a new solution $sol_q$;
5: $sol_q$.cost = $sol^n_q$.cost + $sol^m_q$.cost;
6: $sol_q$.as = $sol^n_q$.as + $sol^m_q$.as;
7: $sol_q$.ga = $sol^n_q$.ga + $sol^m_q$.ga;
8: $sol_q$.g = max($sol^n_q$.g, $sol^m_q$.g);
9: add $sol^m_q$ and $sol^n_q$ to $sol_q$.children;
10: $sol_q$.jumper = NULL;
11: add $sol_q$ to $C_q$;
12: end for
13: end for
14: return $C_q$;

Solution Pruning

To reduce the running time of the algorithm, some pruning operations are performed after each solution growing and merging step. One criteria for solution pruning is shown in Lemma 5.1 [39].

**Lemma 5.1.** Given two candidate solutions $sol_1$ and $sol_2$ for a node $q$, if $sol_1$.cost > $sol_2$.cost and $sol_1$.as − $AR_{\text{max}}$.sol_1.ga > $sol_2$.as − $AR_{\text{max}}$.sol_2.ga, and $sol_1$.g is equal to $sol_2$.g, then $sol_1$ can be safely removed from the set of candidate solutions without hurting the optimality of the algorithm.

In Lemma 5.1, $AR_{\text{max}}$ is the maximum allowable antenna ratio. To use the lemma to prune the set of candidate solutions, one can first divide the set into two subsets, one with $sol.g$ equal to 0 and the other with $sol.g$ equal to 1, and then order the candidate solutions in each subset according to the value of $sol.cost$, and finally traverse through each ordered subset to remove those redundant candidate solutions. Note that the criteria in Lemma 5.1 alone can not guarantee that the number of solutions kept at a node is a polynomial function of the size of the jumper insertion problem. Therefore, to reduce the running time further, a sampling operation is performed after the pruning step. Basically, the solutions are first divided into $L$ slots according to their cost value, where $L$ is a user-specified parameter. Then, among the solutions in each slot, only the one with
the minimum antenna size overhead is chosen. Given a solution $sol$, the antenna size overhead is defined as: $sol.as - AR_{\text{max}}.sol.ga$. The overall pruning algorithm is described in Algorithm 5.5.

Note that the optimality of the dynamic programming algorithm can not be guaranteed if the sampling operation is called in the algorithm (i.e., if the program from line 21 to 24 in Algorithm 5.5 is executed).

**Algorithm 5.5** The pruning procedure.

Input: A set $C$ of candidate solutions and a user-specified parameter $L$.
Output: The set $C_{\text{pruned}}$ obtained by pruning the redundant solutions in $C$.

1: $C_{\text{pruned}} = \emptyset$
2: build two new sets $C_0$ and $C_1$
3: for each solution $sol$ in $C$ do
4: if $sol.g == 0$ then
5: add $sol$ to $C_0$
6: else
7: add $sol$ to $C_1$
8: end if
9: end for
10: order the solutions in $C_0$ according to their cost value (from low to high);
11: pick out the first solution in $C_0$; assume this solution is $sol_0$;
12: $as = sol_0.as - AR_{\text{max}}.sol_0.ga;$ // calculate the antenna size overhead of $sol$
13: for each solution $sol$ in $C_0$ do
14: $as_{\text{tmp}} = sol.as - AR_{\text{max}}.sol.ga;$
15: if $as_{\text{tmp}} > as$ then
16: remove $sol$ from $C_0$; // a redundant solution is found and removed
17: else
18: $as = as_{\text{tmp}}$
19: end if
20: end for
21: if the number of solutions in $C_0$ is larger than $L$ then
22: divide the solutions in $C_0$ into $L$ slots;
23: Among the solutions in each slot, choose the one with the minimum antenna size overhead;
24: end if
25: repeat the operations from line 10 to 24 for the set $C_1$
26: add $C_0$ and $C_1$ to $C_{\text{pruned}}$
27: return $C_{\text{pruned}}$

**Choosing the Best Solution at the Root**

When the propagation procedure arrives at the root of the subtree, the best solution is chosen. Depending on whether the root is the source of the original routing tree, there are two types
Algorithm 5.6 The algorithm of choosing the best solution at the root.

Input: The set $C$ of candidate solutions at the root.
Output: The best solution $sol_{best}$ in $C$.

1: $sol_{best} = NULL$;
2: for each solution $sol$ in $C$ do
3: if the root is not the source of the original routing tree then
4: if $sol.as - AR_{max}.sol.ga \leq 0$ then
5: if $sol_{best} == NULL$ or $sol.cost < sol_{best}.cost$ then
6: $sol_{best} = sol$;
7: end if
8: end if
9: else
10: if $sol.cost < sol_{best}.cost$ then
11: $sol_{best} = sol$;
12: end if
13: end if
14: end for
15: return $sol_{best}$;

of standards for choosing the best solution. If the root is the source of the original routing tree, the solution with the minimum cost is simply chosen. On the other hand, if the root is not the source, the best solution is the one whose cost is minimum and antenna size overhead is no larger than 0.

The algorithm for choosing the best solution is shown in Fig. 5.6.

Back-trace

The back-trace procedure is mostly straightforward, just following the list pointed by the “children” pointer of the solution. If the “jumper” pointer of the solution is not NULL, then the jumper pointed by this pointer is selected in the final solution. The algorithm for the back-trace step is shown in Algorithm 5.7.

5.4.4 The algorithm for the problem in Fig. 5.11

The algorithm used to solve the problem in Fig. 5.11 is also based on dynamic programming and has the same basic flow as the algorithm in the last section. The basic steps, such as solution initialization, propagation, merging, pruning and back-trace, are also similar to those of the algorithm in the last section. The only difference is the solution form. Therefore, to the make
Algorithm 5.7 The back-trace procedure.

Input: The best solution $sol_{best}$ at the root of the subtree.
Output: The set $C$ of selected jumpers in the final solution.

1: $C = \emptyset$;
2: $S = \{sol_{best}\}$;
3: while $S$ is not empty do
4:   pick a solution $sol$ from $S$;
5:   remove $sol$ from $S$;
6:   if $sol.jumper$ is not NULL then
7:     add $sol.jumper$ to $C$;
8:   end if
9:   add all the solutions in $sol.children$ to $S$;
end while
11: return $C$;

the presentation compact, only the changes caused by this difference are shown in the rest of this section.

For the problem in Fig. 5.11, a candidate solution is a pair $(cost, slack)$, where $cost$ and $slack$ respectively represent the cost and the minimum timing slack of the solution. In the beginning, a candidate solution $(0, W^k)$ is built for each sink $k$ in the routing tree, where $W^k$ is the available timing slack of sink $k$.

Assume that the candidate solution $sol$ at node $U$ is to be propagated to node $V$. If there is no candidate location of jumper insertion along the edge from $U$ to $V$, we simply copy $sol$ to the list of candidate solutions at node $V$. Otherwise, for each jumper $j$ in the jumper library at the candidate location of jumper insertion, a new solution $sol_{new}$ is created as follows: $sol_{new}.cost = sol.cost + j.cost$, and $sol_{new}.slack = sol.slack - j.delay$, where $j.cost$ and $j.delay$ represent the cost and delay of the jumper $j$. This new solution is then added to the list of candidate solutions at node $V$.

Next, the merging operation is illustrated. Suppose that two candidate solutions $sol_u$ and $sol_v$ are to be merged together. A new solution $sol$ is created as follows: $sol.cost = sol_u.cost + sol_v.cost$, and $sol.slack = \min(sol_u.slack, sol_v.slack)$.

Note that this merging process involves a “min” operation. As a result, the merging operation can be done in linear time, assuming that the two sets of candidate solutions to be merged have already been sorted according to the timing slack value. This is shown in Algorithm 5.8.

One of the reasons why the algorithm in Algorithm 5.8 has linear running time complexity is that there is a dominance relationship between the solutions. This relationship is shown below.
Algorithm 5.8 The merging procedure when solving the problem in Fig. 5.11.

Input: Two sets $C_m$ and $C_n$ of candidate solutions sorted according to the value of slack (from low to high)

Output: The set $C_q$ of candidate solutions merged from $C_m$ and $C_n$. 

1: $i = j = 0$
2: $C_q = \emptyset$
3: while $i < |C_m|$ and $j < |C_n|$ do
4: set $sol_u$ to the $i^{th}$ solution in $C_m$;
5: set $sol_v$ to the $j^{th}$ solution in $C_n$;
6: build a new solution $sol$
7: $sol.cost = sol_u.cost + sol_v.cost$;
8: if $sol_u.slack < sol_v.slack$ then
9: $sol.slack = sol_u.slack$
10: $i = i + 1$;
11: else
12: $sol.slack = sol_u.slack$
13: $j = j + 1$;
14: end if
15: add $sol$ to $C_q$
16: end while
17: return $C_q$

Definition 5.6. Given two candidate solutions $sol_1$ and $sol_2$ at a node $U$, it is defined that $sol_1$ dominates $sol_2$ if $(sol_1.cost < sol_2.cost and sol_1.slack > sol_2.slack)$.

It is obvious that the dominated candidate solutions can be discarded without hurting the optimality of the algorithm. Therefore, we can perform pruning after the propagation and merging operations to reduce the running time.

According to the dominance relationship, if a set of candidate solutions are sorted from low to high according to the value of “slack” but the value of “cost” of the solutions is not in the order from high to low, then there exist some dominated solutions. These dominated solutions can be safely removed. This is the reason why, in the algorithm in Algorithm 5.8, the indices $i$ and $j$ are always incremented in the while loop.

However, the dominance relationship alone can not guarantee that the number of candidate solutions at a node is polynomial. Therefore, to limit the running time of the algorithm, at most $L$ candidate solutions are kept at each node. If, after pruning, the number of candidate locations at the node is larger than $L$, we do a sampling operation. Basically, the candidate solutions are divided
max \lambda^q_{p,j} \left[ \min_{s_{p,i},c^q_j} \left( \sum_{p \in P} \sum_{q \in Q^p} \sum_{j \in J^q} c^q_{p,j} \left( \frac{V^q_{j}}{2(Pq)} + \lambda^q_{p,j} \right) \right) \\
+ \sum_{q \in Q} \sum_{j \in J^q} \left( c^q_j \left( \frac{V^q_{j}}{2} - \sum_{p \in P^q} \lambda^q_{p,j} \right) \right) \\
+r \sum_{q \in Q} \sum_{j \in J^q} \sum_{p \in P^q} \left( c^q_{p,j} - c^q_j \right)^2 \right] \right]

S.t. (5.3.1),(5.3.2),(5.3.3),(5.3.4),(5.3.6) and (5.3.7) in Fig. 5.7

Figure 5.14. The problem obtained by applying ALR on problem in Fig. 5.7.

into \( L \) slots based on their timing slacks and in each slot we only keep the candidate solution that has the smallest cost.

5.4.5 The augmented Lagrangian relaxation method

As aforementioned in Section 5.4.1, our algorithm uses both CLR and ALR. In this section, it is shown how the ALR method is used.

The ALR of the problem in Fig. 5.7 is shown in Fig. 5.14. It can be observed that the problem in Fig. 5.14 is very similar to the CLR problem in Fig. 5.9. The only difference is that in the objective function in Fig. 5.14 there is a quadratic penalty term. Due to the product between \( c^q_{p,j} \) and \( c^q_j \) in the quadratic term, the problem in Fig. 5.14 can not be decomposed into independent subproblems as we have done for the problem in Fig. 5.9. Instead, the block coordinate descent method is used to solve the problem in Fig. 5.14. The algorithm is shown in Fig. 5.9.

In Fig. 5.9, the algorithm first performs an initialization (lines 1-4). The loop from line 5 to 16 solves the problem in Fig. 5.14 in an iterative manner. The number of iterations is bounded by \( MAX\_\_ITERATION \), which is a user specified parameter. In our implementation, \( MAX\_\_ITERATION \) is set to 20.

In line 6, we solve the problem shown in Fig. 5.15. This problem is obtained from the problem in Fig. 5.14 by picking the part that is related to the variable \( c^q_{p,j} \). Note that in this problem the variables \( c^q_j, \lambda^q_{p,j} \) and \( r \) are regarded as constants. It can be observed that the problem in Fig. 5.15 is similar to the problem in Fig. 5.10. The only differences are the costs of jumpers. Therefore, these two problems can be solved with the same algorithm.
Algorithm 5.9 The algorithm used to solve the ALR problem in Fig. 5.14.

1: initialize $\lambda_{p,j}^q (\forall q \in Q, p \in P^q, j \in J^q)$ and $r$;
2: Initialize $c_j^q$ with the solution returned by CLR;
3: $n = 0$;
4: $pc = 0$;
5: while $n < MAX\_ITERATION$ do
6: For the given $c_j^q$, solve the problem in Fig. 5.15 for $c_{p,j}^q$;
7: For the given $c_{p,j}^q$, solve the problem in Fig. 5.16 for $c_j^q$;
8: $tmp_{pc} = \sum_{q \in Q} \sum_{j \in J^q} \sum_{p \in P^q} \left( c_{p,j}^q - c_j^q \right)^2$;
9: if ($tmp_{pc} == 0$) then
10: return the solution;
11: else
12: Set $\lambda_{p,j}^q = \lambda_{p,j}^q + 2r(c_{p,j}^q - c_j^q)$;
13: Set $r = 2r$ if $n > 0$ and $tmp_{pc} > 2.5pc$;
14: $pc = tmp_{pc}$;
15: end if
16: $n = n + 1$;
17: end while
18: Report that no feasible solution has been found.

$$\min_{s_{p,i}} \sum_{p \in P} \sum_{q \in Q} \sum_{j \in J^q} c_{p,j}^q \left( \frac{V_j^q}{2|P^q|} + \lambda_{p,j}^q + r - 2rc_j^q \right)$$
S.t. (5.3.2), (5.3.3) and (5.3.6) in Fig. 5.7

Figure 5.15. The subproblem of Fig. 5.14 with respect to $s_{p,i}$.

In line 7, we solve the problem shown in Fig. 5.16, which is obtained from the problem in Fig. 5.14 by picking the part that is related to the variable $c_j^q$. In this problem the variables $c_{p,j}^q$, $\lambda_{p,j}^q$ and $r$ are regarded as constants. The problem in Fig. 5.16 is similar to the one in Fig. 5.11. So we can use the algorithm introduced in Section 5.4.4 to solve it.

In line 8 and 9, we test whether a feasible solution to the problem in Fig. 5.7 is found. If so, the obtained solution is returned. Otherwise, the values of $\lambda_{p,j}^q$ and $r$ are updated, and the next iteration starts. Finally, if no feasible solution is found after the iteration number limit is reached, the program stops and reports to the user that no solution is obtained.
$$\min \sum_{q \in Q} \sum_{j \in J_q} c_{j}^{q} \left( \frac{V_{q}}{2} - \sum_{p \in P_{q}} \lambda_{p,j}^{q} + r - 2r \sum_{p \in P_{q}} c_{p,j}^{q} \right)$$

s.t. (5.3.4) and (5.3.7) in Fig.5.7

Figure 5.16. The subproblem of Fig. 5.14 with respect to $c_{j}^{q}$.

5.4.6 Running time analysis

The running time of the overall algorithm is determined by the running time of CLR and ALR. Since the problems in Fig. 5.10 and 5.11 are similar to those in Fig. 5.15 and 5.16, both CLR and ALR have the same running time complexity. Therefore, in this section only analyze the running time of CLR is analyzed. The running time of CLR is determined by the algorithms for the problems in Fig. 5.10 and 5.11. The algorithms for the problems in Fig. 5.10 and 5.11 have been presented in Section 5.4.3 and 5.4.4. For the algorithm in Section 5.4.4, since at most $L$ candidate solutions are kept at every node, the merging operation at a node takes $O(L)$ time, assuming the candidate solutions at a node are always sorted based on timing slacks. The propagation operation along an edge takes $O(L.B.log(L.B))$ time, where $B$ is the maximum number of jumpers in a jumper library. The logarithm factor is due to the sorting operation. Therefore the total running time complexity of the algorithm in Section 5.11 is $O(V.L.B.log(L.B))$, where $V$ is the number of nodes in the tree. The algorithm in Section 5.4.3 is also based on dynamic programming. However, its merging operation is more complex. As a result, the running time complexity is $O(V.L^2.log(L))$. Since one node in the original routing tree may occur in at most $M$ subtrees ($M$ is the number of routing layers in the layout), the total running time complexity of the algorithm in Section 5.4.3 is $O(M.V.L^2.log(L))$.

In conclusion, the time complexity of each iteration in the CLR and ALR algorithm is bounded by $O(M.V.L^2.log(L))$. Therefore, the total running time of the jumper insertion algorithm is bounded by $O(K.M.V.L^2.log(L))$, where $K$ is the number of iterations.

5.5 Experimental Results

The algorithm proposed in this paper has been implemented in C on a machine running 32-bit Debian Linux with kernel 2.6.32-5-686 on an Intel i7-2620M 2.7GHz CPUry. The compiler used is gcc-4.4.5. The five benchmark circuits are obtained from the ISPD benchmark suite [46]
by picking the nets that violate the antenna rule. The ISPD circuits are first routed by an in-house sequential router based on rip up and reroute. Then the nets that suffer from antenna violations are picked out to form the benchmark circuits. The jumpers are allowed to be placed on any routing layer. Since there is no information in the benchmark circuits about the area of the gates, it is assumed that the gate area of all the sinks is equivalent to 2 units of minimum wire width (1 unit for the PMOS transistor and 1 unit for the NMOS transistor). The antenna ratio upper bound is set to 300:1, because this number is close to what is widely accepted as the antenna rule design maximum [47].

5.5.1 Non-cumulative Antenna Ratio

Table 5.2. Benchmark information when using non-cumulative antenna ratio.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>From</th>
<th>#layers</th>
<th>#nets(av)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>adapte1</td>
<td>6</td>
<td>6021</td>
</tr>
<tr>
<td>C2</td>
<td>adapte2</td>
<td>6</td>
<td>4843</td>
</tr>
<tr>
<td>C3</td>
<td>adapte3</td>
<td>6</td>
<td>13709</td>
</tr>
<tr>
<td>C4</td>
<td>adapte4</td>
<td>6</td>
<td>12814</td>
</tr>
<tr>
<td>C5</td>
<td>newblue1</td>
<td>6</td>
<td>1749</td>
</tr>
</tbody>
</table>

In the first set of experiments, it is assumed that the antenna rule is non-cumulative. The information of the five benchmark circuits is shown in Table 5.2. In Table 5.2, the column of “#nets(av)” shows the number of nets that suffer from antenna violations. The “From” column shows the corresponding ISPD benchmark circuits.

Table 5.3. Experimental results when only optimizing the number of vias under non-cumulative antenna ratio.

<table>
<thead>
<tr>
<th>Name</th>
<th>Any layer</th>
<th>Above the highest layer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lb sol</td>
<td>ST MT</td>
</tr>
<tr>
<td>C1</td>
<td>16330 16732</td>
<td>68 39</td>
</tr>
<tr>
<td>C2</td>
<td>11742 11872</td>
<td>44 24</td>
</tr>
<tr>
<td>C3</td>
<td>33842 34216</td>
<td>160 83</td>
</tr>
<tr>
<td>C4</td>
<td>30902 31204</td>
<td>170 89</td>
</tr>
<tr>
<td>C5</td>
<td>4050 4080</td>
<td>12 7</td>
</tr>
<tr>
<td>total</td>
<td>9686 9810</td>
<td>454 242</td>
</tr>
<tr>
<td>ratio</td>
<td>0.99 1.00</td>
<td>1.00 0.53</td>
</tr>
</tbody>
</table>

79
Table 5.4. Technology parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gate capacitance</td>
<td>2fF-8fF</td>
</tr>
<tr>
<td>Wire capacitance</td>
<td>6e-3fF per unit</td>
</tr>
<tr>
<td>Via resistance</td>
<td>100hm per via</td>
</tr>
<tr>
<td>Timing slack</td>
<td>3ps-30ps</td>
</tr>
</tbody>
</table>

Table 5.5. Experimental results when optimizing both the number of vias and timing under non-cumulative antenna ratio.

<table>
<thead>
<tr>
<th>Name</th>
<th>timing-aware</th>
<th>Not timing-aware</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#nets(ng)</td>
<td>sol</td>
<td>time(s)</td>
<td>#nets(ng)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ST</td>
<td>MT</td>
</tr>
<tr>
<td>C1</td>
<td>2</td>
<td>17038</td>
<td>72</td>
<td>38</td>
</tr>
<tr>
<td>C2</td>
<td>1</td>
<td>12558</td>
<td>46</td>
<td>25</td>
</tr>
<tr>
<td>C3</td>
<td>3</td>
<td>36338</td>
<td>164</td>
<td>94</td>
</tr>
<tr>
<td>C4</td>
<td>5</td>
<td>33612</td>
<td>182</td>
<td>94</td>
</tr>
<tr>
<td>C5</td>
<td>0</td>
<td>4166</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>total</td>
<td>11</td>
<td>103712</td>
<td>477</td>
<td>258</td>
</tr>
<tr>
<td>ratio</td>
<td>0.01</td>
<td>1.06</td>
<td>1.11</td>
<td>1.14</td>
</tr>
</tbody>
</table>

As aforementioned, the existing work assumed that the jumpers are placed on or above the highest layer of routing tree. In the first experiment, the impact of this assumption is investigated. The results are shown in Table 5.3.

The meaning of the columns in Table 5.3 is explained below. The columns under “Any layer” list the information of the solutions when jumpers are allowed to be placed on any routing layer, while the columns under “Above the highest layer” list the information of the solutions when jumpers are only placed on or above the highest layer of a routing tree. The “lb” columns lists the lower bounds on the number of vias in the optimal solutions. These bounds are returned by the CLR method. The “sol” columns show the number of vias in the final solutions returned by our algorithm. The column “ST” shows the running time of the algorithm when only one thread is used, while the column “MT” shows the running time when multiple threads are used. Note that our algorithm has successfully fixed all the nets suffering from antenna violations. From the table, it can be observed that the assumption that the jumpers are only placed on or above the highest layer of a routing tree increases the via number by almost 50%. The running time in this case is reduced by about 15% due to the smaller solution space. Also, note that the solutions returned by our algorithm are on average within 7% of the optimal solution. One advantage of Lagrangian relaxation is that the obtained sub-problems are decoupled and hence can be easily solved in parallel. As can be observed in Table 5.3, the multi-threaded version reduces the running time by almost 50%.
In the second experiment, investigate the capability of our algorithm to satisfy the timing constraints is investigated. The technology parameters used in the experiment are show in Table 5.4. The experimental results are shown in Table 5.5.

The meaning of the columns in Table 5.5 is explained below. The columns under “timing-aware” lists the information of the solutions when the algorithm is aware of the delay caused by the jumpers, while the columns under “Not timing-aware” contain the information of the solutions when the algorithm is not aware of this delay. “nets(ng)” columns list the number of nets with negative timing slacks after jumper insertion. All the other columns have the same meanings as in Table 5.3. In the experiment, the jumpers are allowed to be placed on any routing layer. From the table, it can be observed that our timing-aware algorithm is able to fix most of the antenna violations without breaking the timing constraints. The penalties are 6% increase in via number and 14% increase in running time.

### 5.5.2 Cumulative Antenna Ratio

Table 5.6. Benchmark information when using non-cumulative antenna ratio.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>From</th>
<th>#layers</th>
<th>#nets(av)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>adaptec1</td>
<td>6</td>
<td>7323</td>
</tr>
<tr>
<td>C2</td>
<td>adaptec2</td>
<td>6</td>
<td>5526</td>
</tr>
<tr>
<td>C3</td>
<td>adaptec3</td>
<td>6</td>
<td>15825</td>
</tr>
<tr>
<td>C4</td>
<td>adaptec4</td>
<td>6</td>
<td>14110</td>
</tr>
<tr>
<td>C5</td>
<td>newblue1</td>
<td>6</td>
<td>1974</td>
</tr>
</tbody>
</table>

Table 5.7. Experimental results when only optimizing the number of vias under cumulative antenna ratio.

<table>
<thead>
<tr>
<th>Name</th>
<th>Any layer</th>
<th>Above the highest layer</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lb</td>
<td>sol</td>
<td>time(s)</td>
</tr>
<tr>
<td></td>
<td>ST</td>
<td>MT</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>26932</td>
<td>27630</td>
<td>82</td>
</tr>
<tr>
<td>C2</td>
<td>16596</td>
<td>16918</td>
<td>53</td>
</tr>
<tr>
<td>C3</td>
<td>49756</td>
<td>50848</td>
<td>175</td>
</tr>
<tr>
<td>C4</td>
<td>43678</td>
<td>44532</td>
<td>195</td>
</tr>
<tr>
<td>C5</td>
<td>5164</td>
<td>5232</td>
<td>12</td>
</tr>
<tr>
<td>total</td>
<td>142126</td>
<td>145160</td>
<td>517</td>
</tr>
<tr>
<td>ratio</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

In the second set of experiments, the antenna rule is assumed to be cumulative. The information of the five benchmark circuits is shown in Table 5.6. The experimental results are
Table 5.8. Experimental results when optimizing both the number of vias and timing under cumulative antenna ratio.

<table>
<thead>
<tr>
<th>Name</th>
<th>timing-aware</th>
<th></th>
<th></th>
<th>Not timing-aware</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#nets(ng)</td>
<td>sol</td>
<td>time(s)</td>
<td>#nets(ng)</td>
<td>sol</td>
<td>time(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ST</td>
<td>MT</td>
<td></td>
<td>ST</td>
</tr>
<tr>
<td>C1</td>
<td>6</td>
<td>28084</td>
<td>94</td>
<td>51</td>
<td>167</td>
<td>27630</td>
</tr>
<tr>
<td>C2</td>
<td>4</td>
<td>18320</td>
<td>65</td>
<td>38</td>
<td>199</td>
<td>16918</td>
</tr>
<tr>
<td>C3</td>
<td>8</td>
<td>53384</td>
<td>245</td>
<td>125</td>
<td>596</td>
<td>50848</td>
</tr>
<tr>
<td>C4</td>
<td>4</td>
<td>47282</td>
<td>229</td>
<td>130</td>
<td>647</td>
<td>44532</td>
</tr>
<tr>
<td>C5</td>
<td>1</td>
<td>5364</td>
<td>17</td>
<td>8</td>
<td>20</td>
<td>5232</td>
</tr>
<tr>
<td>total</td>
<td>23</td>
<td>152434</td>
<td>650</td>
<td>352</td>
<td>1629</td>
<td>145160</td>
</tr>
<tr>
<td>ratio</td>
<td>0.01</td>
<td>1.05</td>
<td>1.17</td>
<td>0.63</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

shown in Table 5.7 and 5.8. The meaning of the columns in the tables are the same as that in Table 5.3 and 5.5.

From the results in Table 5.7, it can be observed that in the case of cumulative antenna ratio the assumption that the jumpers have to be placed on or above the highest layer increases the number of vias by 26%. Also, from the results in Table 5.8, it can be found that the timing-aware version of the algorithm is much better at satisfying the timing constraints. The penalty is 17% increase in running time and 5% increase in via number.

5.6 Conclusions

In this chapter, a timing-aware jumper insertion algorithm has been proposed. The algorithm targets the antenna rules expressed in the form of antenna ratios and allows the jumpers to be placed on any routing layer. The techniques of Lagrangian relaxation and dynamic programming have been used. Experimental results have demonstrated the algorithm’s efficiency.
Chapter 6

Simultaneous Layer Reassignment and Wire Sizing for Yield Enhancement Based on Tree Decomposition and Multi-threaded Dynamic Programming

6.1 Introduction

As the integrated circuit technology enters the nano-scale era, VLSI designs become more vulnerable to yield loss. Generally, it is believed that the yield loss caused by systematic sources is higher than that caused by the random defects. However, since systematic yield loss can be largely compensated, the yield loss induced by the random defects is relatively more prominent in mature processes [48]. Among the random defects, those related to the interconnect wires play an important role [49]. During the manufacturing of interconnects, contaminating particles may fall on the chip surface and cause the interruption (open) of nets or the unintended interconnection (short) between different nets. For a more detailed description of random defects, please refer to Section 1.3.1.

The problem of improving the interconnection-related yield has been investigated in the stages of track routing [50], detailed routing [49] and post-routing layout modification. Each stage has its advantages and disadvantages in terms of optimization flexibility and yield prediction accuracy. In track routing, the widths and spacings of the wires are adjusted to reduce the probability of opens and shorts. However, the current track routing algorithms only focus on the wires on a single layer. As a result, the track routers lack the capability of moving wires to other layers for
yield improvement. Moreover, the exact positions and shapes of the vias are not known to the track routers. This limits the accuracy of the yield models used in track routing. In detailed routing, the exact geometric shapes and positions of the wires and vias are determined. Therefore, detailed routing has very accurate layout information for yield estimation. However, since detailed routing is typically done in a net-by-net manner, its view of the layout is substantially local. This limits the optimization flexibility of detailed routers. Similar to detail routing, the techniques used in post-routing modification also have access to very accurate layout information. The disadvantage of these techniques is that their flexibility of layout modification is relatively limited when it comes to optimization of a single net. This is due to the fact that the major geometric relationship between the wires has already been fixed by routing. One the other hand, thanks to this limit of optimization flexibility, these post-routing techniques are able to afford considering several nets simultaneously.

In this chapter, we focus on the post-routing layout synthesis techniques. Some examples of these techniques include layer reassignment, wire sizing and wire spreading. The authors in [51] formulate the problem of layer assignment for critical area reduction as a network partitioning problem, which is then solved by a heuristic. The author in [52] investigates the usage of layer reassignment in the context of channel routing to reduce the critical area. Several layout synthesis techniques for yield enhancement, including wire spreading, wire sizing, and layout compaction, are summarized in [53]. The authors in [54] leverage data structures used in layout compaction and propose an enhanced network flow algorithm to more reasonably distribute the objects in the layout for yield improvement. A new layout modification tool is reported in [55] to improve yield and reliability, which takes advantage of redundant vias addition and metal displacement. An algorithm is proposed in [48] to do concurrent wire spreading and sizing within one layer. In addition, a metal fill procedure is also proposed in [48] to improve the uniformity of metal densities.

One of the drawbacks in previous work is that layer reassignment and wire sizing are not considered simultaneously. This may limit the quality of the obtained solutions. For example, when we decide to move a wire to a different layer, we may prefer the layer which allows wider width because then we can widen the wire to make it more robust to the open defects. This requires the layer reassignment algorithm to be aware of the wire sizing options. Similarly, the wire sizing algorithm should be informed of the layer reassignment options to explore the possibility of making the wire wider by moving it to another layer.

In this chapter, an algorithm which simultaneously performs layer reassignment and wire sizing is proposed. The major contributions of this work include:
1. The techniques of layer reassignment and wire sizing are used simultaneously to maximize the yield enhancement.

2. The graph theory concept of tree decomposition is employed in the algorithm design, which, when combined with dynamic programming, allows the layer assignment and wire sizing problem to be solved optimally on non-tree graph models. This is in contrast to existing dynamic-programming-based layer assignment algorithms (e.g., [56] and [57]) whose optimality is limited to tree graphs. To improve the running time performance of the algorithm, a heuristic is also proposed to find the largest subgraph with bounded treewidth.

3. To take advantage of the current multi-core CPUs, the dynamic programming algorithm is implemented with multiple threads. A technique is also proposed to make the obtained tree decompositions more friendly to the multi-threaded implementation.

4. The potential usage of the proposed algorithm in multiple patterning technology is pointed out.

Experimental results show that by performing layer reassignment and wire sizing simultaneously the number of faults can be reduced by 14%. Furthermore, the experimental results also show that a reduction of 32%~56% in the running time of the dynamic programming algorithm can be achieved by the multi-threaded implementation.

The rest of this chapter is organized as follows. In Section 6.2, the overall flow of the algorithm is explained. In Section 6.3, the graph used in this chapter to model the geometric relationship between the routing options of the wires and vias in the layout is described. The optimization algorithm based on tree decomposition and multi-threaded dynamic programming is presented in Section 6.4. The experimental results are shown in Section 6.5. The discussion about using the proposed algorithm in multiple patterning technology is presented in Section 6.6. Finally, the chapter is concluded in Section 6.7.

6.2 Overview

The overall flow is explained with the example in Fig. 6.1.

Fig. 6.1(a) shows a reference layout. There are three routing layers $M_1$, $M_2$ and $M_3$. This given layout contains five wire segments $S_i (i = \{1, ..., 5\})$ and a via $V_1$. The segments $S_2$ and $S_5$ are connected through $V_1$. In the first step of the flow, a set of candidate routing options are
built for the wire segments and vias in the layout. These routing options are provided by the user according to the manufacturing technology. Some examples are shown in Fig. 6.1(b). Among the routing options in Fig. 6.1(b), $S_3^1$ and $S_3^2$ are for the segment $S_3$. The options $S_2^1$ and $S_2^2$ are for $S_2$, and $V_1^1$ and $V_1^2$ are for $V_1$. The option $V_1^1$ is a via from $M_1$ to $M_2$, while $V_1^2$ is from $M_2$ to $M_3$. In Fig. 6.1(b) $S_3^1$, $S_3^2$ and $V_1^1$ represent the original routing options used in Fig. 6.1(a). The options $S_3^2$ and $S_2^2$ are the widened versions of $S_3^1$ and $S_2^1$. After building the candidate routing options for the wires and vias in the layout, an extended continuous-conflict (ECC) graph is constructed. This is shown in Fig. 6.1(c). The design rule constraints, the connection requirements and the failure probabilities due to opens and shorts are reflected in the costs of the edges and vertices in the graph model. After the ECC graph is built, an algorithm is used to select a routing option for each segment and via such that the total cost is minimized. This algorithm is based on tree decomposition and multi-threaded dynamic programming. Fig. 6.1(d) shows a solution where the segment $S_2$ takes the routing option $S_2^2$. This routing option is on layer $M_3$ and has a larger width.

In the following, the ECC graph model is described in Section 6.3. The algorithm for choosing the candidate routing options to minimize the total cost is presented in Section 6.4.

### 6.3 ECC Graph Model

The graph model used in our work is based on the one in [56], with modifications to include the cost due to opens and shorts. In the following, the graph topology in Section 6.3.1 is
first described. The cost functions of the edges and vertices are introduced in Section 6.3.2. Finally, in Section 6.3.3, how to simplify the graph is explained.

### 6.3.1 Topology Description

In the graph model, each vertex represents a wire segment or a via. Associated with each vertex are the set of possible routing options of the corresponding segment or via. Two vertices are connected by an edge if some routing options of the two vertices suffer from connection or conflict constraints or are vulnerable to the short defects.

![Graph Model Examples](image)

Figure 6.2. An example of the graph model.

An example is shown in Fig. 6.2. Fig. 6.2(a) shows a reference layout, which contains three layers M1, M2 and M3. There are 5 segments in the layout. The segments \( S_2 \) and \( S_3 \) are connected by the via \( V_1 \). Fig. 6.2(c) shows some routing options for the segments and vias in the
layout. Among these options, \( \{S_3^1, S_3^2, S_3^3, S_3^4\} \) are for \( S_3 \); \( \{S_2^1, S_2^2\} \) are for \( S_2 \); \( \{S_5^1, S_5^2\} \) are for \( S_5 \); and \( \{V_1^1, V_1^2, V_1^3\} \) are for the via \( V_1 \). The layer notations beside the routing options of \( V_1 \) show the bottom and top layers of the via structures. For example, \( V_1^1 \) has M1 as its bottom layer and M3 as its top layer. The layer and width of \( S_1 \) and \( S_2 \) are assumed to be fixed.

Fig. 6.2(d)-(g) show some examples of the edges in the graph model. Fig. 6.2(d) shows the edge modeling the possible conflict between \( S_3 \) and \( S_4 \). In the figure, \( sp_{min} \) represents the minimum spacing rule. In the original layout \( S_3 \) and \( S_4 \) are on different layers and hence do not conflict with each other. However, if \( S_3 \) takes the routing option \( S_3^3 \), \( S_3 \) and \( S_4 \) will be both on M2 and a conflict will arise. To model this potential conflict, an edge is added between \( S_3 \) and \( S_4 \) in the graph model as shown at the right of Fig. 6.2(d). Fig. 6.2(e) shows how to model the connection requirement among \( S_2, S_3, \) and \( V_1 \). In the original layout, \( S_2 \) and \( S_3 \) are connected through \( V_1 \). This connection requirement is reflected in the graph model by the edges between \( S_2(S_3) \) and \( V_1 \). Fig. 6.2(f) shows an example of the edges involving a stacked via. The option \( V_1^1 \) is a stacked via from M1 to M3. The option \( S_5^2 \) is a widened version of \( S_5 \) on layer M2. Their spacing on M2 is smaller than the minimum spacing rule. As a result, an edge is added between \( V_1 \) and \( S_5 \) in the graph model to reflect this potential conflict. Fig. 6.2(g) shows an example of the edges modeling the probability of failure (POF) caused by the short defects. Since the POF due to the short defects decreases rapidly as the spacing between the metal structures increases, a user defined parameter \( sp_{short} \) is introduced such that if the spacing between the metal wires is larger than this value, the POF caused by the short defects will be relatively insignificant and can be ignored. In our implementation, this value is set to be two times the minimum spacing rule. In Fig. 6.2(g), \( S_2^2 \) is a widened version of \( S_2 \). Since the spacing between \( S_2^2 \) and \( S_1 \) is smaller than \( sp_{short} \), an edge is added between \( S_2 \) and \( S_1 \) in the graph to model this potential yield loss caused by the short defects.

### 6.3.2 Cost Definition

To guide the algorithm to optimize the yield of the layout, we need to assign cost to the edges and vertices in the graph model. These cost functions are described in this section.
Vertex Cost Definition

Assume that in the ECC graph there is a vertex $S_p$ corresponding to a wire segment and the set of routing options of $S_p$ are $\{S_p^i | i \in \{1, ..., n\}\}$. The cost function of $S_p$ is defined as below.

\[
C(S_p^i) = \frac{k L_i}{2 A_{chip}} \left( \frac{w_i + S_{min}}{2 w_i^2 + S_{min} w_i} \right)
\]  

(6.3.1)

The above formula is from [50] to model the probability of failure caused by the open defects. In the formula, $k$ is a technology parameter related to the distribution of the defects, $L_i$ is the length of $S_p^i$, $A_{chip}$ the total chip area, $w_i$ is the width of $S_p^i$, and $S_{min}$ is the minimum spacing rule.

If vertex $S_p$ corresponds to a via, the formula (6.3.1) cannot be directly used. This is because a via passes through several layers and therefore has more than one cross section. In this case, the formula (6.3.1) is used for each cross section and the obtained values are added together as the final cost.

\[
C(S_p^i, S_q^j) = \begin{cases} 
0, & S_p^i \text{ and } S_q^j \text{ are on the same layer and satisfy the overlap constraints.} \\
\infty, & \text{otherwise.} 
\end{cases} 
\]  

(a)

\[
C(S_p^i, S_q^j) = \begin{cases} 
\infty, & S_p^i \text{ and } S_q^j \text{ are on the same layer and } sp_{S_p^i, S_q^j} < sp_{min}. \\
P_{s}(S_p^i, S_q^j), & \text{same layer and } sp_{min} \leq sp_{S_p^i, S_q^j} \leq sp_{short}. \\
0, & \text{otherwise.} 
\end{cases} 
\]  

(b)

Figure 6.3. Edge cost functions: (a) is for the connection edges; (b) is for the others.

Edge Cost Definition

Assume that in the ECC graph there is an edge $e$ connecting a pair of vertices $S_p$ and $S_q$ and the sets of routing options of the two vertices are $\{S_p^i | i \in \{1, ..., m\}\}$ and $\{S_q^j | j \in \{1, ..., n\}\}$, the cost function of the edge is defined as in Fig. 6.3.

Based on the type of the edge, the cost function has two forms. If the edge reflects a connection constraint, the cost function has the form shown in Fig. 6.3(a). Otherwise, the cost function has the form shown in Fig. 6.3(b). In Fig. 6.3(b), $sp_{S_p^i, S_q^j}$ is the spacing between $S_p^i$...
and $S_{q}^{i}$, and $sp_{min}$ is the minimum spacing rule. The notation $sp_{short}$ represents a user specified parameter used to model the short defects. It has the same meaning as in Fig. 6.2(g). The notation $P^{s}(S_{p}^{i}, S_{q}^{j})$ represents the probability of failure due to the shorts between $S_{p}^{i}$ and $S_{q}^{j}$. Its value is calculated as below [50].

$$P^{s}(S_{p}^{i}, S_{q}^{j}) = \frac{k_{l_{ij}}}{2A_{chip}} \left( \frac{s_{ij} + W_{min}}{2s_{ij}^{2} + W_{min}s_{ij}} \right)$$ (6.3.2)\

In the above equation, $k$ is a technology parameter related to the distribution of the defects, $A_{chip}$ is the total chip area, $l_{ij}$ and $s_{ij}$ are the overlapped length and spacing between $S_{p}^{i}$ and $S_{q}^{j}$, and $W_{min}$ is the minimum width rule.

If the edge models the potential conflict between two via structures, the cost function in Fig. 6.3 can not be directly used because the two vias may share more than one layer. In this case the function in Fig. 6.3 is calculated for all the shared layers and the obtained values are added together as the final cost.

### 6.3.3 Graph Simplification

In realistic layouts, there are some nets whose widths and layers are not allowed to be changed. These nets may include, for example, power nets, clock nets, within-cell routing wires and timing-critical nets. We can simplify the graph model by removing from the graph the vertices corresponding to these wires and adjusting the cost of their neighbor vertices to reflect the cost of the edges incident to the removed vertices. Assuming that vertex $S_{p}$ in the graph model has only one routing option $S_{p}^{1}$ (i.e., $S_{p}$ is fixed) and vertex $S_{q}$ is a neighbor of $S_{p}$, for each routing option $S_{q}^{i}$ of $S_{q}$ the cost of $S_{q}^{i}$ is added by $C(S_{q}^{i}, S_{p}^{1})$, where $C(S_{q}^{i}, S_{p}^{1})$ is calculated as described in Section 6.3.2. One example is shown in Fig. 6.4. In the figure, $S_{p}$ and $S_{q}$ represent two vertices, and $S_{p}^{1}$, $S_{q}^{1}$ and $S_{q}^{2}$ are the routing options of the vertices. Assume that the wire or via corresponding to $S_{p}$ is fixed. To remove $S_{p}$ from the graph, the cost of the routing options of $S_{q}$ is updated as shown in Fig. 6.4b. Since the edge from $S_{p}^{1}$ to $S_{q}^{1}$ has a cost of 3, the cost of $S_{q}^{1}$ is added by 3. Similarly, the cost of $S_{q}^{2}$ is added by 2.

Note that it is possible that some routing options of the vertices in the graph model have infinitely large cost (e.g., the routing options that conflict with the fixed wires). These options can be safely removed.
6.4 Algorithm

In this section, the optimization algorithm is introduced. It selects a routing option for each vertex in the graph model to minimize the total cost. Some preliminaries about the concept of tree decomposition are introduced in Section 6.4.1. The algorithm details are presented in the rest of the section.

6.4.1 Preliminaries

Tree decomposition is a concept in graph theory. It maps a graph into a tree that can be used to speed up solving certain problems on the original graph. Please note that tree decomposition does not mean decomposing a graph into trees. Instead, it is a concept that measures the “likeness” of a graph to trees. In a tree decomposition, the vertices and edges of a graph are covered by the union of several “bags” such that for each vertex the bags that cover it form a tree. Its formal definition is presented below [58].

Definition 6.1. A tree decomposition of a graph $G = (V, E)$ is a pair $(\{K_i, i \in I\}, T = (I, F))$ with $K_i \subseteq V, i \in I$ and $T = (I, F)$ a tree, such that:

- (TD1) $\bigcup_{i \in I} K_i = V$;
- (TD2) for all $vw \in E$, there is an $i \in I$ with $v, w \in K_i$;
- (TD3) for all $v \in V$, $\{i \in I : v \in K_i\}$ forms a connected subtree of $T$.

In the above definition, $\{K_i, i \in I\}$ represents the set of “bags” and $T = (I, F)$ represents the connection between the bags.

The width of a tree decomposition $(\{K_i, i \in I\}, T = (I, F))$ is $\max_{i \in I}|K_i| - 1$. The treewidth of a graph is the minimum width over all the tree decompositions of the graph. With this definition, the treewidth of a connected graph is 1 if and only if the graph is a tree.

Figure 6.4. An example of graph simplification.
One example of tree decomposition is shown in Fig. 6.5. Fig. 6.5(a) shows a graph and Fig. 6.5(b) shows one of its tree decompositions. The maximum cardinality of the “bags” in Fig. 6.5(b) is 3, so the width of the tree decomposition is 2. One can check that, among all the tree decompositions of the graph in Fig. 6.5(a), the one in Fig. 6.5(b) has the minimum width. In other words, the treewidth of the graph is 2.

One interesting aspect of tree decomposition is that some NP-hard problems can be solved in polynomial time if the treewidth of the graph is bounded (i.e., the treewidth is regarded as a constant) [59]. A general guideline is given in [58] for judging whether a problem can be solved in polynomial time on graphs with bounded treewidth. Basically, given a vertex cut set, if the solutions on one side of the cut set only depends on the solutions in the cut set, the problem likely can be solved optimally in polynomial time by applying dynamic programming to the tree decomposition of the graph.

Next the example in Fig. 6.5 is used to illustrate the basic idea. First note that the vertices in a “bag” in a tree decomposition form a cut in the original graph if the “bag” is not a leaf. This is due to the following lemma.

**Lemma 6.1.** Given a tree decomposition \( \{K_i, i \in I\}, T = (I, F) \) for graph \( G \), let \( i_1i_2, (i_1, i_2 \in I) \) be any edge of \( T \), and let \( T_1, T_2 \) be the components of \( T - i_1i_2 \), with \( i_1 \in T_1 \) and \( i_2 \in T_2 \), then \( K_{i_1} \cap K_{i_2} \) separates \( U_1 = \bigcup_{i \in T_1} K_i \) from \( U_2 = \bigcup_{i \in T_2} K_i \) in \( G \). [19]

From Lemma 6.1, it can be obtained that \( K_{i_1} \cap K_{i_2} \) forms a cut in \( G \). Since \( K_{i_1} \cap K_{i_2} \) is a subset of \( K_{i_1} \) and \( K_{i_2} \), \( K_{i_1} \) and \( K_{i_2} \) are also vertex cuts if \( U_1 - K_{i_1} \) and \( U_2 - K_{i_2} \) are not empty. For example, in Fig. 6.5(b) the vertices \( \{d, e, g\} \) in the bag \( K_3 \) compose a vertex cut in the graph in Fig. 6.5(a). Now assume that in the combinatorial problem the solution on one side of a cut set only depends on the solution of the vertices in the vertex cut. In Fig. 6.5 this means that to find out the solutions of the vertices \( \{f, h, i\} \) we only need to know the solutions of \( \{d, e, g\} \). In other words, the solutions of \( \{a, b, c\} \) do not need to be considered. This fact helps greatly reduce the
running time of the algorithm and is one of the reasons why some NP-hard problems can be solved in polynomial time if the treewidth of the graph is regarded as a constant.

The concept of tree decomposition has also been used in several work in VLSI. The usage of tree decomposition in the Gate Matrix Layout problem was mentioned in [60]. The authors in [61] used Bayesian networks and junction trees for the switch activity estimation in VLSI networks.

6.4.2 General Description

In this chapter, the focus is on the problem of post-routing layer reassignment and wire sizing for yield enhancement. This problem can be regarded as a generalized post-routing layer assignment problem where the wire sizing options are considered. It is well known that the general problem of post-routing layer assignment is NP-complete [62]. In [56], this problem is solved by the heuristic which iteratively applies the Dynamic Programming (DP) technique to the maximally induced subtrees in the ECC graph. However, by using the tree decomposition technique we can actually use DP to optimally solve the problem on non-tree subgraphs. This allows the algorithm to optimally solve larger subgraphs, which, intuitively, may give better results.

One of the major drawbacks of the DP algorithm based on Tree Decomposition (TD) is that its running time is exponential to the width of the TD. To combat this problem, instead of applying the DP algorithm to the TD of the whole graph, the DP algorithm is iteratively applied to the maximal subgraph whose treewidth is bounded by a user specified parameter. Note that if the treewidth upper bound specified by the user is 1, then the maximal induced subgraph with bounded treewidth is the same as the maximal induced subtree. In this case, our algorithm is similar to that in [56]. However, in the case that the treewidth upper bound is larger than 1, the maximal treewidth-bounded subgraph may contain cycles. As a result, in this case our algorithm is able to optimally solve the problem on subgraphs larger than the maximal induced subtrees. Obviously, one of the key steps of our algorithm is to identify the largest subgraph with bounded treewidth. This is described in Section 6.4.3.

In summary, the algorithm works as follows. It iteratively finds the largest subgraph with bounded treewidth and then optimally solves the problem of layer reassignment and wire sizing on this subgraph by applying the DP algorithm to the subgraph’s tree decomposition. The wire segments and vias that correspond to the vertices in the solved largest subgraph are then fixed and removed from the ECC graph and the next iteration then starts.

To take advantage of the current multi-core CPUs, the dynamic programming algorithm is implemented with multiple threads. A technique is also proposed to make the obtained tree
decomposition more friendly to the multi-threaded implementation. This is presented in Section 6.4.4.

6.4.3 Heuristic for Identifying the Largest Subgraph with Bounded Treewidth

In this section, we focus on the problem of identifying the largest Induced Subgraph With Bounded Treewidth (ISWBT). First note that this problem is generally NP-hard. The NP-hardness can be obtained by noticing that, when the treewidth upper bound is set to 1, the problem is equivalent to finding the maximum induced forest of a graph, which is, in turn, equivalent to finding the minimum Feedback Vertex Set (FVS) of the graph. Since finding the minimum FVS of a graph is well known to be NP-hard [63], the problem of identifying the largest induced subgraph with bounded treewidth is also NP-hard. An exact algorithm for finding the largest ISWBT is described in [64]. However, the running time complexity of that algorithm is $1.734601^n \cdot n^{O(t)}$, where $t$ is the treewidth upper bound and $n$ is the number of vertices in the graph. To make the running time practical, a heuristic is proposed.

The heuristic is shown in Algorithm 6.1, which takes a graph $G$ and a user specified parameter $k$ as inputs and returns a subgraph $G_X$ and a tree decomposition $T_X$ of the subgraph such that the width of $T_X$ is no larger than $k$.

To explain the algorithm in Algorithm 6.1, some definitions and theorems are necessary.

**Definition 6.2.** A graph is chordal if every cycle of length $> 3$ has a chord.

A chordal graph is also called a triangulated graph.

**Theorem 6.2.** Let $G=(V,E)$ be an undirected graph, and let $\mathcal{K}$ be the set of maximal cliques of $G$, with $\mathcal{K}_v$ the set of all the maximal cliques that contain vertex $v$ of $G$. The following statements are equivalent: (i) $G$ is chordal; (ii) there exists a tree $T = (\mathcal{K}, \mathcal{E})$ whose vertex set is the set of maximal cliques of $G$ such that each of the induced subgraphs $T[\mathcal{K}_v]$ is connected. [65]

A tree with the property described in THEOREM 6.2 (ii) is in fact a clique tree of $G$. For chordal graphs, the clique tree is also a tree decomposition and the width of the tree decomposition is equivalent to the size of the maximum clique minus one.

However, in real world applications the graph is not necessarily chordal. We can get around this problem by using the fact that a tree decomposition of a graph $G$ is also a tree decomposition of the subgraphs of $G$.

Based on the above discussion, it is natural to come up with the following procedure to find out the tree decomposition of a graph. First, make the given graph chordal by adding some
Algorithm 6.1 The heuristic for identifying the largest subgraph with bounded treewidth.

Input: A graph G=(V,E) and a user specified parameter k.
Output: A subgraph G_X induced by the nodes in X ∈ V and a tree decomposition T_X={K_X,E_X} of G_X such that the width of T_X is no larger than k.

1: prev_card = 0; X = ∅; s = 0; K_X = ∅; E_T = ∅; F = ∅;
2: for all v ∈ V do
3:   w(v) = 0; l(v) = 0; a(v) = 1; clique(v) = ∅;
4: end for
5: L_{|V|+1} = ∅;
6: for i = |V| → 1 do
7:   U = {v|v ∈ V ∧ l(v) == 0 ∧ a(v) == 1};
8:   choose a vertex x such that w(x) = max{w(v)|v ∈ U};
9: if w(x) > k then
10:   a(x) = 0; //deactivate this vertex
11: continue; //don’t add this vertex to the subgraph
12: end if
13: l(x) = i; //label this vertex
14: X = X ∪ {x}; //add x to X
15: S = ∅;
16: for all vertices y ∈ U do
17:   if there is an edge xy or a path x,v_1,v_2,...,v_k,y in G such that w(v_i) < w(y) and v_i ∈ U(∀i ∈ {1,...,k}) then
18:      S = S ∪ y;
19:   end if
20: end for
21: for all vertices y ∈ S do
22:   w(y) = w(y) + 1;
23:   if xy ∉ E then F = F ∪ xy; //triangulation
24: end for
25: new_card = w(x);
26: adj(x) = {v|vx ∈ E ∪ F};
27: if new_card ≤ prev_card then
28:   K_s = adj(x) ∩ L_{i+1};
29:   K_X = K_X ∪ K_s; //add K_s to K_X
30: if new_card ≠ 0 then
31:   choose y ∈ K_s s.t. l(y) = min{l(v)|v ∈ K_s};
32:   K_p = clique(y);
33:   E_X = E_X ∪ {K_s,K_p};
34: end if
35: s = s + 1;
36: end if
37: clique(x) = K_s;
38: K_s = K_s ∪ {x}; L_i = L_{i+1} ∪ {x}; prev_card = new_card;
39: end for
40: reactivate the vertices that have been deactivated in line 10.
edges. Then, find out the clique tree of the chordal graph. Finally, return the obtained clique tree as a tree decomposition of the original graph. What the algorithm in Algorithm 6.1 does is to merge the above procedure together into one single step.

As aforementioned, we need to limit the width of the returned tree decomposition because the running time of the dynamic programming algorithm is exponential to this number. This limited width is achieved in the algorithm in Algorithm 6.1 by limiting the size of the maximum clique in the graph triangulation step.

Next, the statements in the algorithm in Algorithm 6.1 are briefly explained. The lines 1-5 are for initialization. The notation \( w(v) \) represents the weight of vertex \( v \). It is used to identify which vertex should be labelled next. The notation \( l(v) \) represents the label of \( v \). The notation \( a(v) \) is a mark indicating whether \( v \) is active. The notation \( \text{clique}(v) \) is a pointer. It points to the clique that \( v \) is first added to. The loop from line 6 to 39 labels the vertices one-by-one. In line 8, the active vertex that has not been labelled and has the highest weight is chosen. Then in line 9, it is checked if this vertex can be added into the subgraph without breaking the limit of maximum clique size. If the limit is broken, this vertex is deactivated in line 10, and the next round starts. Otherwise, the vertex is labelled in line 13 and then added to the subgraph. In the statements from line 16 to 24 some extra edges are introduced to locally make the graph chordal. In line 27, it is checked if another maximal clique is detected. If so, a new clique is built and initialized in line 28. Then in the statements from line 30 to 34, this new clique is connected to the clique tree. Finally, in the statements from line 37 to line 38 the vertex \( x \) is added to the clique and \( \text{prev\_card} \) is updated. In line 40, the deactivated vertices are reactivated to prepare for finding the next largest subgraph with bounded treewidth.

A simple example for the heuristic is shown in Fig. 6.6. Fig. 6.6a shows a reference graph, which contains five vertices \( \{a, b, c, d, e\} \). Beside every vertex \( v \) is shown the triple \( \{w(v), l(v), a(v)\} \), where \( w(v) \), \( l(v) \) and \( a(v) \) have the same meanings as in the algorithm in Algorithm 6.1. The value of \( w(v) \) and \( l(v) \) is initialized to be zero while \( a(v) \) is initialized to be one. Assume that we want to find the largest subgraph whose treewidth is no larger than 2. Fig. 6.6b shows the result after the first round. Vertex \( a \) is labelled \( 5 (= |V|) \) and added to the clique \( K_0 \). The weights of \( b \) and \( c \) are increased because they are the neighbors of \( a \). Fig. 6.6c shows the result after the second round. In this round \( b \) is labelled. A virtual edge from \( b \) to \( c \) is added to make the graph chordal. The weights of the neighbors of \( b \) are incremented and \( b \) is added to \( K_0 \). Fig. 6.6d shows the result after the third round, where \( c \) is labelled and added to \( K_0 \). Fig. 6.6e shows the result after the fourth round. In this round a new maximal clique \( K_1 \) is detected and added to the clique tree. Fig. 6.6f shows the result of the final round. Note that in this round vertex \( e \) is deactivated (i.e.,
Figure 6.6. An example for the heuristic in Algorithm 6.1.

\( a(e) \) is set to 0). This is because \( w(e) \) is larger than 2, which means that adding \( e \) to the subgraph will make the width of the tree decomposition larger than 2. The obtained subgraph and its tree decomposition are shown in Fig. 6.6g.

Next, the correctness of the algorithm in Algorithm 6.1 is proved. Before the proof, we need to define some notation. Given a graph \( G = (V, E) \) and a user specified parameter \( k \) as inputs to the algorithm in Algorithm 6.1, let \( G_X = (X, E_X) \) be the subgraph returned by the algorithm, \( F \) be the set of virtual edges added in line 23, and \( E_X \) be the set \( \{uv|uv \in F, u, v \in X\} \). In the following, when we say that a vertex is “processed” in the \( j^{th} \) round of iteration, we mean that the vertex is chosen in line 8 in the \( j^{th} \) round of iteration in the algorithm in Algorithm 6.1. Note that a processed vertex is not necessarily labelled, since it is possible that the processed vertex is deactivated in line 10 rather than labelled in line 13. To simplify the presentation, we use \( L^j(1 \leq j \leq |V|) \) to denote the set of vertices that have been labelled before the \( j^{th} \) round of iteration, and \( M^j \) to denote the set of edges \( \{uv|uv \in E_X \cup F_X, u, v \in L^j\} \), and \( T^j \) to denote the set of vertices that have not been processed before the \( j^{th} \) round of iteration. For a vertex \( v \in T^j \), we use \( N^j(v) \) to denote the set of nodes \( \{u|u \in L^j, uv \in E \cup F\} \). For a vertex \( v \in V \), we use \( w^j(v) \) to denote the weight of \( v \) at the beginning of the \( j^{th} \) round of iteration. A path \( u, x_1, \ldots, x_r, v \) in \( G \) is called
a lower-weight path between \( u \) and \( v \) in the \( j^{th} \) iteration if \( w^j(x_i) < \min\{w^j(u), w^j(v)\} \) (\( \forall i = 1, ..., r \)) and \( x_i (\forall i = 1, ..., r) \) is in \( T' \).

**Lemma 6.3.** Let \( u \) and \( v \) be two vertices that are connected by an edge \( uv \in E \). Assume that \( u \) and \( v \) are respectively processed in the \( m^{th} \) and \( n^{th} \) iteration. If \( m < n \), then \( w^i(v) \leq w^i(u) \) (\( \forall i = 1, ..., m \)) and \( N^m(v) \subseteq N^m(u) \).

**Proof.** Note that in line 8 in the algorithm in Algorithm 6.1, we always choose the vertex with the highest weight to process. Since \( m < n \), obviously we have \( w^m(v) \leq w^m(u) \).

Now suppose that \( w^i(v) > w^i(u) \) for some \( i \in 1, ..., m - 1 \). Then, for any \( j \in i, ..., m \), we would have \( w^j(v) > w^j(u) \). This is explained as follows. Assume that \( j \) is the smallest number in \( i, ..., m \) such that in the \( j^{th} \) round of iteration the weight of \( u \) is incremented in line 22 in Algorithm 6.1. Also assume that in the \( j^{th} \) round of iteration vertex \( x \) is chosen in line 8 as the one to be processed. If there is an edge \( xv \in E \), the weight of \( v \) will also be incremented and we will have \( w^{j+1}(v) > w^{j+1}(u) \). In the following, we show that if there is no such edge between \( x \) and \( v \) in \( E \), we will still have \( w^{j+1}(v) > w^{j+1}(u) \). Since the weight of \( u \) is incremented, either there is an edge \( xu \in E \) or there exists a lower-weight path \( x, y_1, ..., y_r, u \) between \( x \) and \( u \) in the \( j^{th} \) round of iteration. If \( xu \in E \), then the path \( x, u, v \) is a lower-weight path between \( x \) and \( v \) in the \( j^{th} \) round of iteration (note that according to the supposition we have \( w^j(x) \geq w^j(v) > w^j(u) \)). As a result, the weight of \( v \) will be incremented in the \( j^{th} \) round of iteration and we will have \( w^{j+1}(v) > w^{j+1}(u) \). On other hand, if there exists a lower-weight path \( x, y_1, ..., y_r, u \) between \( x \) and \( u \) in the \( j^{th} \) round of iteration, then the path \( x, y_1, ..., y_r, u, v \) is a lower-weight path between \( x \) and \( v \). So, in this case, we will still have \( w^{j+1}(v) > w^{j+1}(u) \). From the above arguments, we can get that, under the supposition that \( w^i(v) > w^i(u) \) for some \( i \in 1, ..., m - 1 \), whenever the weight of \( u \) is incremented, the weight of \( v \) will also be incremented. Therefore, the supposition would lead to \( w^m(v) > w^m(u) \), which causes a contradiction. So, we have the conclusion that \( w^i(v) \leq w^i(u) \) (\( \forall i = 1, ..., m \)).

To prove that \( N^m(v) \subseteq N^m(u) \), suppose that there exists a vertex \( x \) such that \( x \in N^m(v) \) and \( x \notin N^m(u) \). Assume that \( x \) is labelled in the \( j^{th} \) round of iteration. Obviously, we have \( j < m < n \). Therefore, at the beginning of the \( j^{th} \) round of iteration \( w^j(x) \geq w^j(u) \geq w^j(v) \).

If \( w^j(u) > w^j(v) \), \( x, v, u \) form a lower-weight path between \( x \) and \( u \) and a virtual edge will be added between \( x \) and \( v \), which contradicts to the supposition that there is no edge between \( x \) and \( u \). If \( w^j(u) = w^j(v) \) and there is no edge between \( x \) and \( u \), the weight of \( u \) will not be incremented in the \( j^{th} \) round of iteration (i.e., \( w^{j+1}(u) = w^j(u) \)). However, since \( x \in N^m(v) \), the weight of
which contradicts to the fact that $w^i(v) \leq w^i(u) \ (\forall i = 1, \ldots, m)$. Therefore, we have $N^m(v) \subseteq N^m(u)$.

The following Lemma is an extension of Lemma 6.3.

**Lemma 6.4.** Let $u$ and $v$ be two vertices in $G$. Assume that $u$ and $v$ are respectively processed in the $m^{th}$ and $n^{th}$ round of iteration. If $m < n$ and there exist a lower-weight path $P = u, x_1, \ldots, x_r, v$ between $u$ and $v$ in the $m^{th}$ iteration, then $w^i(v) \leq w^i(u)$, $w^i(x_j) \leq w^i(u)$, $w^i(x_j) \leq w^i(v) \ (\forall j = 1, \ldots, r, \forall i = 1, \ldots, m)$ and $N^m(v) \subseteq N^m(u)$.

**Proof.** First, we prove that $w^i(x_j) \leq w^i(u) \ (\forall j = 1, \ldots, r, \forall i = 1, \ldots, m)$. Suppose that there exists a number $a \in \{1, \ldots, m\}$ and a set $B \subseteq \{x_1, \ldots, x_r\}$ such that $w^a(b) > w^a(u)$ for each vertex $b$ in $B$. Among the vertices in $B$, pick out the subset $B_{\text{sub}}$ such that $w^a(c) \geq w^a(b)$ ($\forall b \in B, \forall c \in B_{\text{sub}}$). Then, among the vertices in $B_{\text{sub}}$, choose as $y$ the vertex that is closest to $u$ in the lower-weight path $P$. Assume that $y = x_l \ (l \in \{1, \ldots, r\})$. Then, for each vertex $z \in \{x_1, \ldots, x_r, u\}$, whenever the weight of $z$ is incremented in some iteration $j \in \{a, \ldots, m\}$, there exists a lower-weight path reaching $y$ in the same iteration and the weight of $y$ is also incremented. So, we have $w^i(y) > w^i(u)$ and $w^i(y) > w^i(x_j) \ (\forall i \in \{a, \ldots, m\}, \forall j \in \{l + 1, \ldots, r\})$, which contradicts to the fact that $w^m(y) < w^m(u)$ (note that $y$ is in $\{x_1, \ldots, x_r\}$ and the path $u, x_1, \ldots, x_r, v$ is a lower-weight path in the $m^{th}$ iteration). Therefore, the supposition is false and $w^i(x_j) \leq w^i(u) \ (\forall j = 1, \ldots, r, \forall i = 1, \ldots, m)$.

Similarly, we can prove that $w^i(x_j) \leq w^i(v)$ and $w^i(v) \leq w^i(u) \ (\forall j = 1, \ldots, r, \forall i = 1, \ldots, m)$.

To prove that $N^m(v) \subseteq N^m(u)$, suppose that there exists a vertex $y$ such that $y \in N^m(v)$ and $y \notin N^m(u)$. Assume that $y$ is labelled in the $j^{th}$ iteration (obviously $j < m$). Then, according to the supposition, there is an edge between $y$ and $v$ but no edge between $y$ and $u$ in the $j^{th}$ iteration. If $w^j(v) = w^j(u)$, we would have $w^{j+1}(v) > w^{j+1}(u)$, which contradicts to the fact that $w^j(v) \leq w^j(u) \ (\forall i = 1, \ldots, m)$. Therefore, $w^j(v) < w^j(u)$. Since there is no edge between $y$ and $u$ in the $j^{th}$ iteration, there exits a vertex set $B \in \{x_1, \ldots, x_r\}$ such that $w^j(b) = w^j(u) \ (\forall b \in B)$. Assume that $z \in B$ is the one that is closest to $v$. Then, the path $y, v, x_1, \ldots, z$ is a lower-weight path between $y$ and $z$ in the $j^{th}$ iteration. As a result, $w^{j+1}(z) = w^j(z) + 1 > w^j(z) = w^{j+1}(u)$, which contradicts to the fact that $w^{j+1}(z) \leq w^{j+1}(u)$ (note that $z$ is a vertex in $\{x_1, \ldots, x_r\}$ and we have proofed $w^i(x_j) \leq w^i(u) \ (\forall j = 1, \ldots, r, \forall i = 1, \ldots, m)$). Therefore, the supposition is false and $N^m(v) \subseteq N^m(u)$.

99
Lemma 6.5. For any \( j \) in \( 1, \ldots, |V| \), the graph \( G^j = (L^j, M^j) \) is a chordal graph and for each \( v \in \overline{T}^j \), the vertices in \( N^j(v) \) form a clique in \( G^j \).

Proof. In the following, the lemma is proved by induction. To start, when \( j = 1 \), both \( L^j \) and \( M^j \) are empty. So, in this case the lemma is true.

Now assume that the lemma is true when \( j = m \) \((m \geq 0)\). And also assume that in the \( m^{th} \) iteration, vertex \( v \in \overline{T}^m \) has been chosen in line 8 in Algorithm 6.1 as the vertex to be processed next.

If, in the \( m^{th} \) iteration, vertex \( v \) is deactivated in line 10 in Algorithm 6.1, then \( G^{m+1} = G^m \). Since, according to the induction assumption, \( G^m \) is a chordal graph, \( G^{m+1} \) is also a chordal graph. Furthermore, since the deactivation of \( v \) does not affect \( N^m(u) (\forall u \in \overline{T}^{m+1}) \), \( N^{m+1}(u) = N^m(u) (\forall u \in \overline{T}^{m+1}) \). From the induction assumption, the vertices in \( N^m(u) \) form a clique in \( G^m \).

Since \( G^{m+1} = G^m \) and \( N^{m+1}(u) = N^m(u) (\forall u \in \overline{T}^{m+1}) \), the vertices in \( N^{m+1}(u) \) form a clique in \( G^{m+1} \). Therefore, in this case, the lemma is true for \( j = m + 1 \).

If vertex \( v \) is not deactivated in the \( m^{th} \) iteration, it will then be labelled in line 13 in Algorithm 6.1. Since, according to the induction assumption, the vertices in \( N^m(v) \) form a clique in \( G^m \), \( v \) is a “simplicial [65]” vertex as to the chordal graph \( G^m \). Since adding a simplicial vertex to a chordal graph still gives a chordal graph, graph \( G^{m+1} \) is chordal. In the following, we show that the vertices in \( N^{m+1}(u) (\forall u \in \overline{T}^{m+1}) \) form a clique in \( G^{m+1} \).

Assume that \( u \) is an arbitrary vertex in \( \overline{T}^{m+1} \). If \( u \) is connected to \( v \) with an edge in \( E \), then, according to Lemma 6.3, \( N^m(u) \subseteq N^m(v) \). Since, according to the induction assumption, the vertices in \( N^m(u) \) form a clique, the vertices in \( N^m(u) \cup \{v\} = N^{m+1}(u) \) also form a clique.

If \( u \) is connected to \( v \) through a lower-weight path in the \( m^{th} \) iteration, then, according to Lemma 6.4, \( N^m(u) \subseteq N^m(v) \). Also, since \( u \) is connected to \( v \) through a lower-weight path, a virtual edge will be added between \( u \) and \( v \) in the \( m^{th} \) iteration and \( v \) is in \( N^{m+1}(u) \). Since, \( N^m(u) \subseteq N^m(v) \) and, according to the induction assumption, the vertices in \( N^m(u) \) form a clique, the vertices in \( N^m(u) \cup \{v\} = N^{m+1}(u) \) also form a clique.

If neither \( u \) connects to \( v \) by an edge nor there exists a lower-weight path between \( u \) and \( v \) in the \( m^{th} \) iteration, we have \( N^{m+1}(u) = N^m(u) \). Since, according to the induction assumption, the vertices in \( N^m(u) \) form a clique, the vertices in \( N^{m+1}(u) \) also form a clique.

In summary, the vertices in \( N^{m+1}(u) \) always form a clique. Because the vertex \( u \) is arbitrarily chosen, the vertices in \( N^{m+1}(u) (\forall u \in \overline{T}^{m+1}) \) form a clique. \( \square \)
Lemma 6.6. Given a graph $G = (V, E)$ and a user specified parameter $k$ as inputs to the algorithm in Algorithm 6.1, let $G_X = (X, E_X)$ be the subgraph returned by the algorithm, $F$ be the set of virtual edges added in line 23, and $F_X$ be the set $\{uv| uv \in F, u, v \in X\}$. The graph $G_X^+ = (X, E_X \cup F_X)$ is a chordal graph.

Proof. The lemma can be derived from Lemma 6.5 by noticing that $G_X^+ = G |^{V \setminus L|^{V \setminus M|^{V}. \square$

Theorem 6.7. The algorithm in Algorithm 6.1 is correct. Specifically, given a graph $G = (V, E)$ and a user specified parameter $k$ as inputs, the algorithm returns a subgraph $G_X = (X, E_X)$ and a tree decomposition $T_X = \{K_X, E_X\}$ of $G_X$ such that the width of $T_X$ is no larger than $k$.

Proof. In the lines 25-38 in the algorithm in Algorithm 6.1, we use the algorithm in [65] for clique tree construction. The original clique-tree-construction algorithm in [65] is developed for chordal graphs and is based on maximal cardinality search. Note that the triangulation steps in the lines 16-24 in Algorithm 6.1 are also based maximal cardinality search. As a result, the order in which the vertices in $G_X^+$ are processed in the algorithm in Algorithm 6.1 is one of the valid orders that may be obtained if the algorithm in [65] is applied directly to the graph $G_X^+$ (note that, from Lemma 6.6, we know that $G_X^+ = (X, E_X \cup F_X)$ is a chordal graph). Furthermore, because of the triangulation steps (i.e., lines 16-24 in Algorithm 6.1), the set “$L_{i+1}$” used in line 28 in Algorithm 6.1 always contains the vertices whose labels are larger than $i$. This guarantees that the maximal cliques are detected correctly. Finally, since the value of $l(v)(v \in X)$ used in line 31 in Algorithm 6.1 contains the correct order label, the connection between the maximal cliques in the algorithm in Algorithm 6.1 is finished in the same way as the algorithm in [65]. Therefore, the clique tree $T_X = \{K_X, E_X\}$ returned by the algorithm in Algorithm 6.1 is one of the clique trees that may be produced if the algorithm in [65] is applied directly to the graph $G_X^+$. Since the clique-tree-construction algorithm in [65] is correct, the clique tree returned by the algorithm in Algorithm 6.1 is also correct.

Furthermore, because of the if-structure in the lines 9-13 in Algorithm 6.1, none of the cliques in the clique tree has a size larger than $k + 1$. This means that, if treating the clique tree $T_X$ as a tree decomposition of $G_X^+$, the width of $T_X$ is no larger than $k$.

Finally, since $G_X$ is a subgraph of $G_X^+$, $T_X$ is also a tree decomposition of $G_X$. \square

Next, we will prove a property of the tree decompositions returned by the algorithm in Algorithm 6.1. This property makes the returned tree decompositions “immune” to the simplification algorithm in [2]. Before presenting the property, we first need to introduce the simplification algorithm in [2].
Given a tree decomposition, the algorithm in [2] tries to reduce the sizes of the “bags” in the tree decomposition. It has been shown that, if there exists a “bag” in the tree decomposition such that the “expanded graph” of the bag is not a clique, then the tree decomposition can be simplified by replacing the bag with smaller bags. This is formally described below.

**Definition 6.3.** Given a graph \( G = (V, E) \) and a tree decomposition \( T = (K, E) \) of \( G \), for each node \( K_i \) in \( K \), the expanded graph of \( K_i \) is a graph that consists of the induced graph of the vertices in \( K_i \) and the additional edges \( \bigcup_{K_j \in N(K_i)} C(K_i \cap K_j) \), where \( N(K_i) = \{K_j | K_j \in K, K_i K_j \in E\} \) and \( C(K_i \cap K_j) \) is the set of edges in the complete graph on the vertices in \( K_i \cap K_j \).

![Figure 6.7. An example of tree-decomposition simplification with the algorithm in [2].](image)

Definition 6.3 is explained with the example in Fig. 6.7. Fig. 6.7b shows a graph \( G \) and Fig. 6.7a shows one of the graph’s tree decompositions. The expanded graph of \( K_2 \) is built as follows. First, the induced graph of the vertices in \( K_2 \) is the graph \( G_1 \) in Fig. 6.7c. The intersection set between \( K_2 \) and \( K_3 \) is \( \{e, g\} \). The complete graph of the vertices in this intersection set is the graph \( G_3 \) in Fig. 6.7c. Similarly, \( G_2 \) and \( G_4 \) in Fig. 6.7c show the complete graphs of the vertices
in \( K_2 \cap K_1 \) and \( K_2 \cap K_5 \). By merging the graphs in Fig. 6.7c together, we have the expanded graph of \( K_2 \) as shown in Fig. 6.7d.

An algorithm is proposed in [2] to simplify the tree decompositions that have some specific property. Generally, given a graph \( G \) and its tree decomposition \( T = ( \mathcal{K}, \mathcal{E} ) \), if there exists a node \( K_i \in \mathcal{K} \) such that the expanded graph of \( K_i \) is not a clique, then \( T \) can be simplified by replacing \( K_i \) with smaller nodes. For example, Fig. 6.7a shows a tree decomposition of the graph in Fig. 6.7b. The expanded graph of \( K_2 \) in the tree decomposition is shown in Fig. 6.7d. Note that the expanded graph is not a clique. As a result, by using the algorithm in [2], the tree decomposition in Fig. 6.7a can be simplified by replacing \( K_2 \) with “smaller” nodes. This replacement is shown in Fig. 6.7e, and the new tree decomposition is shown in Fig. 6.7f. For the details of the simplification algorithm, please refer to [2].

The following lemma can be obtained from [2].

**Lemma 6.8.** Given a graph \( G \) and its tree decomposition \( T = ( \mathcal{K}, \mathcal{E} ) \), \( T \) can be simplified by the algorithm in [2] if and only if there exists a node \( K_i \in \mathcal{K} \) such that the expanded graph of \( K_i \) is not a clique.

Next, we will prove that it is impossible to simplify the tree decompositions returned by the algorithm in Algorithm. 6.1 with the algorithm in [2].

From Lemma 6.6, we know that the graph \( G_X^+ = (X, E_X \cup F_X) \) is a chordal graph. In the next lemma, we show that for each edge \( uv \) in \( F_X \) there exist two nodes \( K_i \) and \( K_j \) in the returned tree decomposition in Algorithm. 6.1 such that \( \{u, v\} \subseteq K_i \cap K_j \).

**Lemma 6.9.** Given a graph \( G = (V, E) \) and a user specified parameter \( k \) as inputs to the algorithm in Algorithm. 6.1, let \( G_X = (X, E_X) \) and \( T_X = (\mathcal{K}_X, \mathcal{E}_X) \) be respectively the subgraph and tree decomposition returned by the algorithm, \( F \) be the set of virtual edges added in line 23, and \( F_X \) be the set \( \{uv|uv \in F, u, v \in X\} \). For each edge \( uv \in F_X \), there exist two nodes \( K_i \) and \( K_j \) in \( \mathcal{K}_X \) such that \( \{u, v\} \subseteq K_i \cap K_j \). Furthermore, for each node \( K_s \in \mathcal{K}_X \) such that \( \{u, v\} \subseteq K_s \), \( K_s \) has a neighbor \( K_t \) in \( T_X \) such that \( \{u, v\} \subseteq K_s \cap K_t \).

**Proof.** Assume that \( u \) is labelled in the \( m^{th} \) iteration in the algorithm in Algorithm. 6.1 and \( v \) is labelled in the \( n^{th} \) iteration. And, without loss of generality, assume that \( m < n \). So, \( u \) is labelled earlier than \( v \). According to Lemma 6.5, \( v \) and the vertices that are connected to \( v \) and labelled before \( v \) form a clique. We use \( K_i \in \mathcal{K}_X \) to denote the node that contains the vertices in the clique. Note that both \( u \) and \( v \) are in \( K_i \).
Since the edge $uv$ is in $F_X$ (i.e., $uv$ is not in $E$), there exists a lower-weight path $u, x_1, ..., x_r, v$ between $u$ and $v$ in the $m^{th}$ iteration. Among the vertices in $\{x_1, ..., x_r\}$, pick those that have the highest weight at the beginning of the $m^{th}$ iteration. Then, among those picked vertices, choose the one that is closest to $u$. Assume that this vertex is $y$. Because of the way $y$ is chosen, there exists a lower-weight path between $u$ and $y$ in the $m^{th}$ iteration. Therefore, if there is no edge between $u$ and $y$, a virtual edge between them will be added. Also, the weight of $y$ will be incremented in the $m^{th}$ iteration. As a result, at the end of the $m^{th}$ iteration, $y$ has the unique highest weight among the vertices in $\{x_1, ..., x_r\}$. In addition, because the weight of $v$ is also incremented in the $m^{th}$ iteration, the weight of $y$ is still strictly less than that of $v$ at the end of the $m^{th}$ iteration. Assume that $y$ is processed in the $p^{th}$ iteration in the algorithm in Algorithm 6.1. According to Lemma 4.1 in [66], $n < p$ and $y$ still has the unique highest weight among the vertices in $\{x_1, ..., x_r\}$ at the beginning of the $n^{th}$ iteration. Therefore, there exists a lower weight path between $v$ and $y$ in the $n^{th}$ iteration. So, if there is no edge between $y$ and $v$, a virtual edge will be added between them in the $n^{th}$ iteration. From Lemma 4.1 in [66], we also have that the weight of $y$ is at most equal to that of $v$ at the end of the $n^{th}$ iteration. This means that $y$ is not in $K_i$. In the $p^{th}$ iteration, $y$ is processed. According to Lemma 6.5, the vertices that are connected to $y$ and labelled before $y$ form a clique. We use $K_j \in \mathcal{K}_X$ to denote the node that contains the vertices in the clique. Note that both $u$ and $v$ are in $K_j$. So, $K_j$ is not empty. Also note that $K_j$ is different from $K_i$, because $y$ forms a clique with the vertices in $K_j$ but does not form a clique with the vertices in $K_i$. So, there exist two nodes $K_i$ and $K_j$ in $\mathcal{K}_X$ such that $\{u, v\} \subseteq K_i \cap K_j$.

Now assume that $\{u, v\} \subseteq K_s$. From the above argument, there must exist another node $K_q \in \mathcal{K}_X$ such that $\{u, v\} \subseteq K_q$. Assume that $K_s$ and $K_q$ are connected by the path $P : K_s, K_t, ..., K_q$ in $T_X$. According to the definition of tree decomposition in Definition 6.1, all the nodes in $P$ contain both $u$ and $v$. In particular, $\{u, v\} \subseteq K_t$. In other words, $K_s$ has a neighbor $K_t$ such that $\{u, v\} \subseteq K_s \cap K_t$.

**Theorem 6.10.** In the algorithm in Algorithm 6.1, the returned tree decomposition $T_X = (\mathcal{K}_X, \mathcal{E}_X)$ for the subgraph $G_X = (X, E_X)$ can not be simplified by the algorithm in [2].

**Proof.** First, according to Lemma 6.6, the graph $G_X^+ = (X, E_X \cup F_X)$ is a chordal graph. And, from the proof of Theorem 6.7, $T_X$ is a clique tree of $G_X^+$. This means that, for any node $K_i \in \mathcal{K}_X$, if the edges in the set $\{uv | u, v \in K_i, uv \in F_X\}$ are added to the $G_X$’s subgraph that is induced by the vertices in $K_i$, the subgraph will become a clique.
Second, according to Lemma 6.9, for any edge \( uv \in F_X \), if both \( u \) and \( v \) are in \( K_i \), then \( K_i \) must have a neighbor \( K_j \) that also contains both \( u \) and \( v \). Therefore, we have that \( \{uv|u,v \in K_i, uv \in F_X \} \) is a subset of \( \bigcup_{K_j \in N(K_i)} C(K_i \cap K_j) \), where \( N(K_i) \) is the set of neighbors of \( K_i \) and \( C(K_i \cap K_j) \) is the set of edges in the complete graph on the vertices in \( K_i \cap K_j \).

Combining the above two arguments, we know that the expanded graphs of the nodes in \( K_X \) are always cliques. So, according to Lemma 6.8, it is impossible to simplify \( T_X \) with the algorithm in [2].

According to Theorem 6.10, it is unnecessary to process the tree decompositions returned by the algorithm in Algorithm 6.1 with the algorithm in [2].

The running time of the algorithm in Algorithm 6.1 is limited by the step in line 17, whose complexity is \( O(|E|) \) [67], where \( |E| \) is the number of edges in the graph. Therefore, the running time complexity of the whole algorithm in Algorithm 6.1 is \( O(|V| \cdot |E|) \), where \( |V| \) is the number of vertices in the graph.

As can be observed in Fig. 6.6g, the obtained subgraph is not necessarily a tree. So the traditional Dynamic Programming (DP) algorithms in [57] and [56] can not optimally solve the problem on this subgraph. However, when combined with the tree decomposition technique, the DP algorithm can be extended to optimally solve the problem on this kind of general graphs. This is explained in the next section.

\[
D_1 = \{(S_1^1, S_2^1, S_3^1, S_4^1), (S_1^1, S_2^1, S_3^1, S_4^1), (S_1^1, S_2^1, S_3^1, S_4^1)\}
\]

\[
K_1 = \{S_2, S_4\}
\]

\[
K_2 = \{S_1, S_2\} \quad K_3 = \{S_2, S_3\}
\]

\[
D_2 = \{(S_1^1, S_1^1, S_2^2, S_2^2), (S_3^2, S_3^2), (S_1^1, S_1^1, S_2^2, S_2^2), (S_3^2, S_3^2)\}
\]

\[
D_3 = \{(S_1^1, S_2^1, S_3^1, S_4^1), (S_1^1, S_2^1, S_3^1, S_4^1), (S_1^1, S_2^1, S_3^1, S_4^1)\}
\]

Figure 6.8. An example of dynamic programming.
6.4.4 Dynamic Programming

In this section, we introduce the dynamic programming technique used to solve the following problem. Given a subgraph $G_X$ and the tree decomposition $T_X$ of $G_X$, select a routing option for each vertex in $G_X$ such that the total cost of the edges and nodes as defined in Section 6.3 is minimized.

Basic Description

Since using dynamic programming on tree decomposition is a well studied topic, in this section we only explain the basic idea. The reader is referred to [2] for an example of the formal description.

Given a tree decomposition $T_X$, the dynamic programming algorithm proceeds from the leaves of $T_X$ towards the root. Given a node $K_i$ in $T_X$, we use $D_i$ to denote the set of all the possible combinations of the routing options of the vertices in $K_i$. If $K_i$ is a leaf, the set $D_i$ is built at the start of the algorithm. On other hand, if $K_i$ is not a leaf and has two children $K_m$ and $K_n$, the costs of the combinations in $D_i$ are computed by merging the combinations in $D_m$ and $D_n$. Basically, every combination in $D_m$, every combination in $D_n$ that shares the same routing options for the vertices in $K_m \cap K_n$, and every combination of the routing options of the vertices in $K_i \setminus (K_m \cup K_n)$ are combined together. This process is explained with the example in Fig. 6.8.

The tree decomposition in Fig. 6.8 contains three “bags” $K_1$, $K_2$ and $K_3$. The bag $K_1$ is the predecessor of $K_2$ and $K_3$. The vertices $S_1$, $S_2$, $S_3$ and $S_4$ are covered by the tree decomposition. The bag $K_2$ contains $S_1$ and $S_2$, while $K_3$ contains $S_2$ and $S_3$, and $K_1$ contains $S_2$ and $S_4$. To simplify the presentation, we assume that each vertex $S_i$ ($i \in \{1, 2, 3, 4\}$) has only two candidate routing options $S_i^1$ and $S_i^2$. All the four possible combinations of the routing options of the vertices in $K_2$ are shown in $D_2$. Similarly, $D_3$ shows all the possible combinations of routing options of the vertices in $K_3$. To merge $D_2$ with $D_3$, we first pick up a combination in $D_2$. Assume that $(S_1^1, S_2^1)$ is picked. Because $S_2$ is shared between $K_2$ and $K_3$, we must ensure that the combination we pick from $D_3$ selects the same routing option for the vertex $S_2$. So we can choose $(S_2^1, S_3^1)$ and $(S_2^1, S_3^2)$ from $D_3$. Merging $(S_1^1, S_2^1)$ with these two combinations produces $(S_1^1, S_2^1, S_3^1)$ and $(S_1^1, S_2^1, S_3^2)$. Repeating the above procedure for each combination in $D_2$, we obtain in total 8 combinations, as shown at the bottom of the figure. These combinations can be divided into two categories: the one that selects the routing option $S_2^1$ for $S_2$ and the other that selects $S_2^2$ for $S_2$. Among each category, we only need to keep the one with the minimum cost. Assume that the combinations $(S_1^1, S_2^1, S_3^1)$...
and \((S^1_1, S^2_2, S^3_1)\) are the two with the minimum cost. In the final step, we combine these two combinations with the two possible routing options of \(S_4\) to obtain the set \(D_1\), which is shown at the top of the figure. Also note that not all the data associated with the combinations in \(D_1\) is stored at \(K_1\). For example, in the combination \((S^1_1, S^3_2, S^1_1, S^2_4)\) in \(D_1\), the sub-combination \((S^1_1, S^1_2)\) has been stored at \(K_2\) and the sub-combination \((S^1_1, S^1_3)\) has been stored at node \(K_3\). Therefore, for the combination \((S^1_1, S^1_2, S^3_4, S^2_4)\), at \(K_1\) we only need to store \(S^1_4\) and the pointers to \((S^1_1, S^1_2)\) and \((S^1_1, S^1_3)\). In this way, the memory usage of the algorithm is reduced.

In the above we assume that \(K_1\) has two children. The case that \(K_1\) has only one child or more than two children can be derived similarly and is therefore skipped. After the bottom-up propagation arrives at the root, the solution with the minimum cost is chosen and a trace back is performed.

The running time complexity of the algorithm is \(O(|K| \cdot |R|^k \cdot k^2)\), where \(|K|\) is the number of “bags” in the tree decomposition, \(|R|\) is the maximum number of routing options a vertex has, and \(k\) is the width of the tree decomposition. The \(k^2\) factor is the time bound on calculating the total cost caused by a single combination. One can observe that the running time is exponential to \(k\). This is why we propose the heuristic in Section 6.4.3 to limit the width of the tree decomposition.

**Multithreaded Implementation**

To take advantage of the current multi-core CPUs, we have implemented a multi-threaded version of the dynamic programming algorithm. The basic idea is that the different child branches of a node can be processed simultaneously.

![Figure 6.9](image.png)

**Figure 6.9. Impact of root selection on the depth of tree.**

One thing to note is that the selection of the root of the tree decomposition may have a significant impact on the performance of the multi-threaded implementation. This is illustrated in Fig. 6.9. Fig. 6.9a shows a tree decomposition. Fig. 6.9b and Fig. 6.9c respectively show the directed trees obtained from the tree decomposition by choosing \(K_1\) and \(K_2\) as the root. In Fig.
6.9b, the multi-threaded implementation can not help much because each node has only one child branch. However, in Fig. 6.9c $K_2$ has two child branches, which can be processed simultaneously. Note that we are allowed to change the root because the ECC graph model used in our work is an undirected graph.

Figure 6.10. Tree topology enhancement.

One characteristic of the directed tree structures that are friendly to multi-threaded implementation is that they are typically shallower. For example, in Fig. 6.9, the depth of the tree in Fig. 6.9b is smaller than that of the tree in Fig. 6.9c. Therefore we face the problem of selecting as root a node in the tree decomposition such that the depth of the produced directed tree is as small as possible. One simple method to solve this problem is to try setting as root each node in the tree and use the depth-search-algorithm to find out the depths of all the obtained directed trees and finally choose the node which induces the minimum-depth tree. The advantage of this method is that it guarantees to give the optimal solution. However, the running time of this method is $\Theta(n^2)$ ($n$ is the number of nodes in the tree), which can be significantly long for large tree decompositions. To reduce the running time, we have developed a heuristic algorithm. In the algorithm, we first randomly select a node as the root. Then we find the deepest path of the produced directed tree topology. Finally, we try flipping at the nodes in the deepest path to reduce the depth as much as possible. This procedure is explained in Fig. 6.10. Fig. 6.10a shows a tree. Assuming that $K_1$ is initially chosen as the root, the produced directed tree is the one in Fig. 6.10b. The deepest path in this directed tree is the one from $K_1$ to $K_6$. In our tree-structure-enhancement algorithm, a traverse is performed along the deepest path starting from the root towards the leaf, and the node where flipping induces the minimum depth is chosen as the new root. In Fig. 6.10b, the new root is node $K_2$. Finally, the direction of the edges in the tree are changed in favor of the new root. In Fig. 6.10, the finally obtained directed tree is the one in Fig. 6.10c. The running time complexity of this tree-structure-enhancement algorithm is $O(n)$, since all the steps (i.e., finding the deepest path,
finding the best position in the deepest path to flip, and changing the direction of the edges) can be finished in linear time.

### 6.4.5 Speed up for Large Scale Layouts

While the heuristic in Section 6.4.3 can be used to limit the running time of the dynamic programming algorithm, the running time of the heuristic itself can be significantly long for large-scale layouts. This is due to its $O(|V| \cdot |E|)$ running time complexity. To solve this problem, the layout is divided into overlapping windows. Then for each window the wire segments and vias that overlap with it are picked out and their layer reassignment and wire sizing solutions are solved by the algorithms in Section 6.4.3 and 6.4.4.

![Diagram](image)

Figure 6.11. An example for illustrating the speed-up technique that slices the layout into overlapping windows.
This speed up technique is illustrated in Fig. 6.11. Fig. 6.11a shows a layout. In the first step, the conflict graph for the whole layout is constructed. This is shown in Fig. 6.11b. In the second step, the layout is sliced into overlapping windows. This is shown in Fig. 6.11c, where three overlapping windows $W_1$, $W_2$, and $W_3$ are formed to cover the layout. Then, for each window, the subgraph induced by the vertices that correspond to the wire segments overlapping with the window is picked out, and the layer reassignment and wire sizing solutions for the vertices in this subgraph are solved by the algorithms in Section 6.4.3 and 6.4.4. For example, the subgraph induced by the wire segments overlapping with the window $W_1$ in Fig. 6.11c is $G_{sub}$ in Fig. 6.11d. To solve the layout reassignment and wire sizing problem for $G_{sub}$, we first “fix” all the other vertices that are not in $G_{sub}$ (i.e., the layers and sizes of the wire segments not belonging to $G_{sub}$ are fixed). In Fig. 6.11d, the fixed vertices are colored grey. Then, we use the simplification technique introduced in Section 6.3.3 to remove the edges between $G_{sub}$ and those fixed vertices and move the cost of the removed edges into the cost of the vertices in $G_{sub}$. In this way, $G_{sub}$ in “detached”. Finally, the algorithms introduced in Section 6.4.2, 6.4.3 and 6.4.4 are used to find the layer reassignment and wire sizing solutions for the vertices in $G_{sub}$.

### 6.5 Experimental Results

The proposed algorithm has been implemented in C on a machine running 64-bit Debian Linux with kernel-3.0 on an Intel i7-2630QM Quad Core CPU at 2.00GHz. Three benchmark circuits have been designed by Alliance [68] using a set of 0.18um rules. Information about those layouts is summarized in Table 6.1. The last column in the table lists the number of objects (e.g., wire segments and vias) in each layout. Monte Carlo simulation is performed to test the robustness of those layouts to random defects. In the simulation 10K particles are randomly generated. It is assumed that the random particles are uniformly distributed across the layout area and layers. The size distribution of the random particles is assumed to follow the function in [50]. The wires in the layouts are allowed to be placed on any layer as long as the preferred routing direction rules are satisfied. The widths of the wires are allowed to be one or two times the minimum width rules. Power nets, clock networks, and within-cell routings are not touched by the algorithm.

In the existing work, either the layer reassignment technique is used to reduce the shorts (e.g., [51] and [52]) or the wire sizing technique is used to reduce the opens (e.g., [48] and [53]). But none of them performs concurrent layer reassignment and wire sizing to simultaneously reduce opens and shorts. So in the first experiment we test how much yield improvement can be achieved.
Table 6.1. Information about the benchmark circuits.

<table>
<thead>
<tr>
<th>Name</th>
<th>Layer</th>
<th>No. of objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>8</td>
<td>247k</td>
</tr>
<tr>
<td>C2</td>
<td>8</td>
<td>204k</td>
</tr>
<tr>
<td>C3</td>
<td>8</td>
<td>142k</td>
</tr>
</tbody>
</table>

Table 6.2. Results of the first experiment on the effect of performing LA and WS simultaneously.

<table>
<thead>
<tr>
<th>Name</th>
<th>original layout</th>
<th>LA</th>
<th>WS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>faults</td>
<td>time</td>
<td>faults</td>
</tr>
<tr>
<td></td>
<td>open</td>
<td>short</td>
<td>sum</td>
</tr>
<tr>
<td>C1</td>
<td>383</td>
<td>315</td>
<td>698</td>
</tr>
<tr>
<td>C2</td>
<td>565</td>
<td>470</td>
<td>1035</td>
</tr>
<tr>
<td>C3</td>
<td>315</td>
<td>155</td>
<td>470</td>
</tr>
<tr>
<td>sum</td>
<td>1263</td>
<td>940</td>
<td>2203</td>
</tr>
<tr>
<td>scale</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Name</th>
<th>LA&amp;WS</th>
<th>time</th>
<th>faults</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>faults</td>
<td>(s)</td>
<td>open</td>
</tr>
<tr>
<td>C1</td>
<td>152</td>
<td>301</td>
<td>286</td>
</tr>
<tr>
<td>C2</td>
<td>112</td>
<td>492</td>
<td>427</td>
</tr>
<tr>
<td>C3</td>
<td>101</td>
<td>284</td>
<td>111</td>
</tr>
<tr>
<td>sum</td>
<td>365</td>
<td>1077</td>
<td>824</td>
</tr>
<tr>
<td>scale</td>
<td>1.00</td>
<td>0.85</td>
<td>0.88</td>
</tr>
</tbody>
</table>

(b)

by doing Layer reAssignment (LA) and Wire Sizing (WS) simultaneously. The results are shown in Table 6.2. In the experiment, the treewidth upper bound used in the heuristic in Section 6.4.3 is set to be 2 because we find this number generally gives the best tradeoff between running time and solution quality. In this table, “LA&WS” represents doing LA and WS simultaneously. From the results it can be observed that LA is effective in reducing shorts but has little impact on the number of opens. On the contrary, WS is effective in reducing opens but can do little about shorts. Performing LA and WS concurrently combines the advantages of the two techniques and are able to reduce opens and shorts simultaneously. According to the results, “LA&WS” is able to reduce the number of faults by 14%, while “LA” and “WS” reduce the number of faults by 6% and 8% respectively. However, the running time of “LA&WS” is longer due to its larger solution space.

In the second experiment, we look at the impact of the treewidth upper bound used in the heuristic in Section 6.4.3 on the solution quality and running time of the algorithm. The results are
shown in Table 6.3. Note that the case where the treewidth upper bound is 1 corresponds to applying the dynamic programming algorithm to the maximal induced subtrees. In this case, the algorithm becomes similar to the one in [56]. However, when the treewidth upper bound is larger than 1, the algorithm can optimally solve the subgraphs larger than the maximal subtrees and hence has better results. According to the experimental results, when the treewidth upper bound is increased from 1 to 2, the average number of faults is reduced by 8%. For the running time part, the “TD” columns show the running time contributed by tree decomposition (i.e., the algorithm in Section 6.4.3) and the “DP” columns show the running time from dynamic programming. One thread is used in this experiment to isolate the impact of multi-threads. From the results, it can be observed that the running time of the dynamic programming algorithm generally increases exponentially with the treewidth upper bound.

Table 6.3. Results of the second experiment on the effect of treewidth upper bound.

<table>
<thead>
<tr>
<th>Name</th>
<th>Treewidth=1</th>
<th></th>
<th></th>
<th></th>
<th>Treewidth=2</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>faults</td>
<td>time(s)</td>
<td></td>
<td>faults</td>
<td>time(s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TD</td>
<td>DP</td>
<td>Total</td>
<td>TD</td>
<td>DP</td>
<td>Total</td>
<td>TD</td>
<td>DP</td>
</tr>
<tr>
<td>C1</td>
<td>657</td>
<td>146</td>
<td>20</td>
<td>166</td>
<td>587</td>
<td>132</td>
<td>41</td>
<td>173</td>
</tr>
<tr>
<td>C2</td>
<td>992</td>
<td>105</td>
<td>14</td>
<td>119</td>
<td>919</td>
<td>96</td>
<td>25</td>
<td>121</td>
</tr>
<tr>
<td>C3</td>
<td>409</td>
<td>86</td>
<td>10</td>
<td>96</td>
<td>395</td>
<td>82</td>
<td>22</td>
<td>104</td>
</tr>
<tr>
<td>Sum</td>
<td>2058</td>
<td>337</td>
<td>44</td>
<td>381</td>
<td>1901</td>
<td>310</td>
<td>88</td>
<td>398</td>
</tr>
<tr>
<td>Scale</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.92</td>
<td>0.92</td>
<td>2.00</td>
<td>1.04</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Name</th>
<th>Treewidth=3</th>
<th></th>
<th></th>
<th></th>
<th>Treewidth=4</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>faults</td>
<td>time(s)</td>
<td></td>
<td>faults</td>
<td>time(s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TD</td>
<td>DP</td>
<td>Total</td>
<td>TD</td>
<td>DP</td>
<td>Total</td>
<td>TD</td>
<td>DP</td>
</tr>
<tr>
<td>C1</td>
<td>582</td>
<td>128</td>
<td>144</td>
<td>272</td>
<td>584</td>
<td>129</td>
<td>568</td>
<td>697</td>
</tr>
<tr>
<td>C2</td>
<td>884</td>
<td>90</td>
<td>86</td>
<td>176</td>
<td>893</td>
<td>95</td>
<td>360</td>
<td>455</td>
</tr>
<tr>
<td>C3</td>
<td>394</td>
<td>75</td>
<td>76</td>
<td>151</td>
<td>392</td>
<td>72</td>
<td>310</td>
<td>382</td>
</tr>
<tr>
<td>Sum</td>
<td>1860</td>
<td>293</td>
<td>306</td>
<td>599</td>
<td>1869</td>
<td>296</td>
<td>1238</td>
<td>1534</td>
</tr>
<tr>
<td>Scale</td>
<td>0.90</td>
<td>0.87</td>
<td>6.95</td>
<td>1.57</td>
<td>0.91</td>
<td>0.88</td>
<td>28.14</td>
<td>4.03</td>
</tr>
</tbody>
</table>

(b)

Another interesting phenomena in the second experiment is the dependence of fault reduction on circuits. For example, for benchmark C1, when the treewidth upper bound is increased from 1 to 2, the number of faults is reduced from 657 to 587, a reduction of 11.7%. However, for benchmark C3, only 3% reduction is achieved when the treewidth upper bound is increased from 1 to 2. This indicates that the reduction of faults is dependent on some circuit properties. By investi-
gating deeper into the benchmark circuits, we find that the fault reduction is actually affected by the density distribution of the circuits. This is illustrated in Fig. 6.12. Fig. 6.12a and Fig. 6.12b show a layout pattern and its conflict graph. Note that in Fig. 6.12a rectangle $D$ is far away from $A$, $B$, and $C$. As a result, the conflict graph in Fig. 6.12b contains no cycles. Therefore, to optimally solve the problem on the conflict graph in Fig. 6.12b, a tree decomposition of width 1 is enough. However, if polygon $D$ is close to $A$ and $C$, as shown in Fig. 6.12c, a cycle is formed in the conflict graph, as shown in Fig. 6.12d. In this case, the width of the tree decomposition needs to be increased to optimally solve the problem. In general, the more “complex” the conflict graph is (i.e., the conflict graph contains more subgraphs whose connectivity is high), the more likely that increasing the treewidth upper bound will significantly reduce the number of faults.

![Figure 6.12](image)

**Figure 6.12.** An illustration of the dependence of fault reduction on circuit density.

In the final experiment, we test the impact of the multi-threaded implementation on the running time of the dynamic programing algorithm. The results are shown in Table 6.4, “TW” represents the maximum allowed width of the tree decompositions to which the dynamic programming algorithm is applied. The “ST” columns show the running time of the dynamic programming algorithm when only one thread is used, while the “MT” column shows the running time of the multi-threaded version. It can be observed that 32%~56% reduction in the running time has been achieved by the proposed multi-threaded implementation.

**Table 6.4.** A comparison of single-thread and multi-thread running times (in secs).

<table>
<thead>
<tr>
<th>Name</th>
<th>TW=1</th>
<th>TW=2</th>
<th>TW=3</th>
<th>TW=4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ST</td>
<td>MT</td>
<td>ST</td>
<td>MT</td>
</tr>
<tr>
<td>C1</td>
<td>20</td>
<td>7</td>
<td>41</td>
<td>20</td>
</tr>
<tr>
<td>C2</td>
<td>14</td>
<td>12</td>
<td>25</td>
<td>16</td>
</tr>
<tr>
<td>C3</td>
<td>10</td>
<td>11</td>
<td>22</td>
<td>19</td>
</tr>
<tr>
<td>Sum</td>
<td>44</td>
<td>30</td>
<td>88</td>
<td>55</td>
</tr>
<tr>
<td>Scale</td>
<td>1.00</td>
<td>0.68</td>
<td>1.00</td>
<td>0.63</td>
</tr>
</tbody>
</table>
6.6 Applying the Proposed Algorithm to Multiple-Patterning Technologies

In this section, we discuss applying the proposed algorithm to multiple-patterning technologies.

In the multiple-patterning technologies such as double patterning, triple patterning and quintuple patterning, the polygons on a critical layer are assigned to different masks. Basically, if the spacing between two polygons is smaller than a threshold value, the two polygons must be assigned to two different masks. If there exist two polygons that are close to each other but, due to space limit, cannot be assigned to different masks, then there is a conflict. One example is shown in Fig. 6.13. Fig. 6.13a shows a layout layer that contains five polygons. Assuming that $s_p$ is smaller than the threshold value, polygon $A$ and $B$ are not allowed to be assigned to the same mask. Otherwise, a conflict would be generated. For a more detailed description of the multiple-patterning technologies, please refer to Section 1.3.3 and Chapter 4.

Some problem formulations in multiple-patterning technologies are:

**Problem 6.1.** Assign the polygons on a layout layer to different masks to minimize the number of conflicts. (See references [28], [32], and [69])

**Problem 6.2.** Perform layer reassignment for the polygons in the layout such that the polygons on the critical layers can be assigned to different masks to minimize the number of conflicts. (e.g., [70])

In the following two sections, it is shown how the above two problem formulations can be solved by the algorithm proposed in this chapter.

6.6.1 Problem 6.1

For the formulation in Problem 6.1, the polygons on a critical layer are assigned to different masks to minimize the number of conflicts. Taking the layout layer in Fig. 6.13a as an example and assuming that this layer is manufactured by double-patterning technology (i.e., at most two masks can be used to print the patterns on this layer), the formulation in Problem 6.1 can be stated as follows:

*Assign the polygons A, B, C, D, and E to two masks such that the number of conflicts is minimized.*

The existing work typically formulates the above problem as a coloring problem on a conflict graph, where the vertices in the conflict graph correspond to the polygons and two vertices
Figure 6.13. An example of layout decomposition in multiple-patterning technology.
are connected by an edge if the spacing between the polygons corresponding to the two vertices is smaller than the threshold value. The “colors” in the coloring formulation correspond to the different masks that the polygons can be assigned to, and a color conflict is produced if the two vertices connected by an edge have the same color. In the Fig. 6.13, the conflict graph for the polygons in Fig. 6.13a is shown in Fig. 6.13b. One coloring solution for the conflict graph is shown in Fig. 6.13c, where \( A, B, \) and \( E \) are assigned to \( MASK_1 \), and \( C \) and \( D \) are assigned to \( MASK_2 \). Note that there is a color conflict between \( A \) and \( B \) due to the odd cycle \( ABC \).

To remove those color conflicts, the existing algorithms usually includes a fracture step before coloring. For example, Fig. 6.13d shows a fracture possibility for the layout in Fig. 6.13a. Note that polygon \( A \) is fractured into two rectangles \( A_1 \) and \( A_2 \). The conflict graph for this fractured layout is shown in Fig. 6.13e. It can be observed that the odd cycle \( ABC \) in Fig. 6.13b has been removed in Fig. 6.13e.

The problem with the fracture operation is that it makes the layout manufacturing vulnerable to overlay error. For example, in Fig. 6.13d, it is required that \( A_1 \) and \( A_2 \) touch each other. However, if \( A_1 \) and \( A_2 \) are printed by two masks and the two masks are not perfectly aligned, it is possible that \( A_1 \) and \( A_2 \) are not connected. This is shown in Fig. 6.13f, where \( OE \) represents the overlay error. Some sub-wavelength lithography phenomena, such as line-end shortening, make the overlay problem even more serious. To compensate for overlay errors, some rules are developed by the circuit manufacturers. Basically, if two polygons are printed by two different masks and are supposed to connect with each other, the overlapping area between the two polygons must be no less than a minimum value. Taking into account this overlapping constraint, the compatibilities between the mask assignment options of \( A_1 \) and \( A_2 \) are shown in Fig. 6.13g-6.13n, where \( \{ A_{11}, A_{12}, A_{13}, A_{14} \} \) is the set of mask assignment options of \( A_1 \) and \( \{ A_{21}, A_{22} \} \) is the set of options of \( A_2 \). Note that \( A_{11} \) is compatible with \( A_{21} \) because they are printed by the same mask and hence do not suffer from the overlay error problem. The polygon \( A_{31} \) is compatible with \( A_{22} \) because their overlapping area is larger than the minimum requirement. The polygon \( A_{22} \) is not compatible with \( A_{11} \), since they do not have any overlapping area. Two special cases are shown in Fig. 6.13m and 6.13n. In the figure, these two cases are labelled as \( \text{incompatible} \), because they are actually the same as those combinations in Fig. 6.13g and 6.13h and hence redundant. As a result, we can safely ignore them and treat them as incompatible.

Note that, even with the consideration of overlapping constraint, the mask assignment problem can still be formulated as a generalized coloring problem. What we need to do is to expand the color sets to include the mask assignment options developed for the overlapping constraint.
and modify the definition of “color conflict” accordingly. For example, to model the relationship between $A_1$ and $A_2$ in Fig. 6.13d while taking into account the overlapping constraint, we can add an edge between $A_1$ and $A_2$ in the conflict graph in Fig. 6.13e and use $\{A_1^1, A_2^2, A_3^3, A_4^4\}$ as the color set for $A_1$ and $\{A_1^1, A_2^2\}$ as the color set for $A_2$. The compatibility between the different “colors” of $A_1$ and $A_2$ is as defined in Fig. 6.13g-6.13n, and a “color conflict” is produced if the color of $A_1$ is not compatible with that of $A_2$.

From the previous discussion, we have shown that Problem 6.1 can be formulated as a coloring problem. Generally, the coloring problems are NP-complete. However, it is well known that the tree decomposition technique can be used to solve the coloring problems optimally [71]. Nevertheless, as expected, the running time of the optimal algorithm is exponential to the width of the tree decomposition. To make the running time practical, one can use the algorithm proposed in this chapter in Section 6.4, which iteratively uses the heuristic in Section 6.4.3 to identify the largest subgraph with bounded treewidth and then uses dynamic programming to optimally solve the problem on the obtained subgraph. Furthermore, as having been shown in Section 6.4.4, the dynamic programming algorithm can be sped up by being implemented with multiple threads. This speed-up means that we can tolerate tree decompositions with larger width. As a result, larger subgraphs can be solved optimally within a reasonable amount of time.

6.6.2 Problem 6.2

In the formulation in Problem 6.2, the layers of the wire segments are allowed to be changed to reduce the number of conflicts. One such example is shown in Fig. 6.14. Fig. 6.14a shows the original layout, which contains two layers $M1$ and $M2$. Polygon $A$, $B$ and $C$ are on $M1$, while $C$ and $D$ are on $M2$. In this example, it is assumed that $sp$ is smaller than the minimum spacing that can be printed by a single mask. As result, $A$ conflicts with $B$ and they must be assigned to different masks. Similarly, $B$ conflicts with $C$, and $C$ conflicts $A$. Assuming that at most two masks can be used to print the polygons on $M1$, it is impossible to avoid all the conflicts in mask assignment. This is because $A$, $B$ and $C$ form an odd conflict cycle. One way to solve this problem is to fracture the polygons on $M1$. This method has been introduced in Section 6.6.1. Another way is to do layer reassignment for the polygons in the layout. For example, if we switch the layers of $A$ and $D$ in Fig. 6.14a, we will obtain the layout in Fig. 6.14b. In this new layout, $D$ conflicts with $B$ but does not conflicts with $C$. Thus, no odd conflict cycle is formed and all the conflicts can be avoided in mask assignment.
To model the layer reassignment problem for multiple-patterning technologies, one can extend the ECC graph model introduced in Section 6.3 to include the mask assignment options. For example, the extended ECC graph for the layout in Fig. 6.14a is shown in Fig. 6.14c. The sets of layer and mask assignment options for the polygons \( B \) and \( C \) are shown in Fig. 6.14d. Note that in the example it is assumed that double patterning is used to manufacture the polygons on layer \( M_1 \) while single patterning is used on layer \( M_2 \). This means that the polygons on \( M_1 \) can be assigned to two different masks while those on \( M_2 \) are always assigned to the same mask.

In Fig. 6.14d, \( B^1 \) and \( B^2 \) respectively represent assigning \( B \) to \( Mask1 \) and \( Mask2 \) of layer \( M_1 \), while \( B^3 \) represents assigning \( B \) to layer \( M_2 \). In the extended ECC graph in Fig. 6.14c, the edge set \( \{AB, AC, BC, BD, BE\} \) models the conflict relationships in mask assignment while the edge set \( \{AD, CE\} \) models the conflict relationships in layer assignment. To simplify the following presentation, the edges that model the conflicts in mask assignment are called “MACEs” (Mask Assignment Conflict Edges), while the edges the model the conflicts in layer assignment are called “LACEs” (Layer Assignment Conflict Edges). If two vertices are connected by a LACE, the polygons corresponding to these vertices can not be assigned to the same layer. On the other hand, if two vertices are connected by a MACE, the polygons corresponding to the vertices are allowed to be on the same layer but not allowed to be on the same mask. For example, in Fig. 6.14c, \( B \) and
$C$ are connected by a MACE. This means that polygon $B$ can not be on the same mask as polygon $C$. Since the polygons on layer $M2$ are manufactured by single patterning (i.e., single mask), the edge between $B$ and $C$ also means that the two polygons can not simultaneously appear on layer $M2$. The conflict relationships between the layer and mask assignment options of $B$ and $C$ are summarized in Fig. 6.14e. The edge between $C$ and $E$ in Fig. 6.14c is an example of LACE. The conflict relationships between the layer and mask assignment options of $C$ and $E$ are summarized in Fig. 6.14f and 6.14g.

Note that the conflict relationships between the layer and mask assignment options of the polygons can be expressed in the form of edge cost functions in the extended ECC graph. For example, we can define the edge cost function as follows: if the two chosen options conflict with each other, the cost is $\infty$. Otherwise, the cost is zero. Then, the layer and mask assignment problem becomes that of choosing a layer and mask assignment option for each vertex in the extended ECC graph such that the total cost is minimized. This problem is very similar to the layer assignment problem that was formulated in Section 6.3, and hence can also be solved by the algorithms in Section 6.4.

In the above example, it is assumed that double-patterning technology is used in manufacturing. The cases where multiple-patterning technologies, such as triple patterning and quintuple patterning are used, can be solved in a similar way.

In conclusion, the algorithms proposed in Section 6.4 can also be used to solve the problems formulated as Problem 6.2.

### 6.7 Conclusions and Future Work

In this paper, a simultaneous layer reassignment and wire sizing algorithm for yield enhancement has been proposed. The algorithm takes advantage of the tree decomposition technique to solve non-tree graphs optimally. The dynamic programming algorithm has been implemented with multiple threads. Experimental results have demonstrated the efficiency of the algorithm. In addition, the potential benefit of using this algorithms in the problems in multiple patterning technologies has been discussed. One interesting future research direction is to integrate other layout enhancement techniques such as redundant via insertion and wire spreading into the optimization framework.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this dissertation, some problem formulations and algorithms for optimizing manufacturability and yield in routing stage have been proposed.

A track routing algorithm that simultaneously optimizes timing and yield has been described in Chapter 3. This algorithm tackles the problem by performing the optimization in two steps: wire ordering, and wire positioning and sizing. It has been shown that, for a given wire ordering, the problem of wire positioning and sizing for yield and timing can be formulated as a mixed linear geometric programming problem, assuming that the Elmore delay model and the analytical yield models are used. Although the mixed linear geometric programming problem can be solved optimally after being converted into a convex optimization problem, the running time of the optimal solver is too long. A heuristic has also been proposed to return a decent solution within much shorter time. It has been shown that, compared with yield, timing is more sensitive to the spacings and widths of the wires. Therefore, it is possible to significantly improve the timing performance of the circuits without hurting the overall yield very much.

Two techniques for enhancing the double-patterning-friendly detailed algorithm have been presented in Chapter 4. The Lazy Color Decision technique delays the coloring and shadowing of the grids that can take two colors. In this way, the coloring flexibility of these grids is fully used and less coloring conflicts and stitches are generated. The Last Conflict Segment Recording technique is a heuristic which record some extra information during path searching. This piece of information helps the path search algorithm to be aware of the generation of within-path conflicts. The experimental results have demonstrated that these two techniques are able to reduce the number of stitches by 15～20%.
A jumper insertion algorithm that targets the practical antenna rules used in industry has been introduced in Chapter 5. The problem has been formulated as an Integer Programming problem. Timing constraints have been incorporated to make the formulation more reasonable. The Lagrangian relaxation technique has been used to reduce the integer programming problem to several simpler combinatorial problems that can be solved with dynamic programming. It has been shown in the experiments that, by allowing the jumpers to be placed on any layer, the number of vias can be reduced by around 50% if the antenna rule is non-cumulative and 20% if the rule is cumulative. Experimental results have also shown that the algorithm is much better at satisfying the timing constraints.

A concurrent layer reassignment and wire sizing algorithm has been proposed in Chapter 6. In contrast to the existing layer assignment algorithms that perform dynamic programming only on trees, the proposed algorithm extends the dynamic programming algorithm to non-tree graphs by using the tree decomposition concept in graph theory. To make the running time practical, a heuristic based on the maximal cardinality search algorithm has been developed to find the largest subgraph with bounded treewidth. The dynamic programming algorithm has been implemented with multiple threads to reduce the running time further. Compared with the existing algorithms, the proposed algorithm is able to optimally solve larger subgraphs and hence returns better solutions. It has also been pointed out that the proposed algorithms can be readily used to solve the coloring problems in multiple patterning technologies.

7.2 Future Work

7.2.1 Track Routing

While the track routing algorithm proposed in Chapter 3 is able to simultaneously optimize timing and yield, it fails to consider the impact of track routing on noise. Even though it is likely that the noise metrics can be integrated into the mixed linear geometric formulation, it is unknown how the heuristic performs when the noise consideration is incorporated. Therefore, one interesting extension of the heuristic in Chapter 3 would be to modify the cost function to make the algorithm aware of the noise impact.
7.2.2 **Double-Patterning-Friendly Detailed Routing**

The techniques proposed in Chapter 4 have been shown to be effective in reducing the number of coloring conflicts and stitches in double-patterning technology. However, the mathematical formulations behind these techniques are still not very clean. In particular, it is still unknown if there exists any efficient mathematical formulation that can seamlessly combine path searching and coloring. Furthermore, it is required that this formulation can be extended to multiple (instead of being limited to only double) coloring problems.

7.2.3 **Jumper Insertion**

It has been shown that the algorithm in Chapter 5 is able to solve the timing-aware jumper insertion problem effectively. However, since this algorithm is based on Lagrangian relaxation, tens of iterations are needed for the convergence of the subgradient method. The next step is to investigate the possibility of developing a faster heuristic.

7.2.4 **Simultaneously Layer Reassignment and Wire Sizing**

In Chapter 6, it has been shown that the problem of concurrent layer reassignment and wire sizing can be solved elegantly by the algorithm based on tree decomposition. One drawback of the proposed algorithm is that the impact of wire sizing on timing is not considered. This drawback can be partly compensated for by prohibiting the change of timing-critical nets. But this limits the flexibility of the optimization algorithm. One natural question is whether it is possible to directly incorporate the consideration of timing into the optimization framework.

Another interesting research direction in the future is to use the tree-decomposition technique to solve other layout enhancement problems, such as wire spreading, redundant via insertion, and lithograph-friendly layout pattern improvement.
Appendix A

A Summary of the Existing Work

Table A.1. Global routing

<table>
<thead>
<tr>
<th>Year</th>
<th>Metrics</th>
<th>Algorithms &amp; Techniques &amp; Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010[73]</td>
<td>CMP</td>
<td>Using maximum effective wire density in the cost function.</td>
</tr>
<tr>
<td>2009[74]</td>
<td>Via</td>
<td>Via-aware Steiner tree construction and 3-bend pattern routing.</td>
</tr>
</tbody>
</table>

Table A.2. Layer assignment (a)

<table>
<thead>
<tr>
<th>Year</th>
<th>Metrics</th>
<th>Algorithms &amp; Techniques &amp; Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1986[76]</td>
<td>Via</td>
<td>Two-layer constrained via minimization and graph bipartization formulation.</td>
</tr>
<tr>
<td>1987[77]</td>
<td>Via</td>
<td>Showing that the two-layer constrained via minimization problem is in general NP-complete.</td>
</tr>
<tr>
<td>1987[78]</td>
<td>Via</td>
<td>Showing that the two-layer via minimization problem is in general NP-complete.</td>
</tr>
</tbody>
</table>
Table A.3. Layer assignment (b)

<table>
<thead>
<tr>
<th>Year</th>
<th>Metrics</th>
<th>Algorithms &amp; Techniques &amp; Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988[79]</td>
<td>Via</td>
<td>Two layer constrained via minimization, integer linear programming formulation, transforming into a maximum cut problem.</td>
</tr>
<tr>
<td>1997[56]</td>
<td>Via</td>
<td>Multilayer constrained via minimization, the extended conflict-continuation graph model, a heuristic based on dynamic programming on the maximal induced subtrees.</td>
</tr>
<tr>
<td>1998[81]</td>
<td>Via &amp; cross talk</td>
<td>Three-layer constrained via minimization, a modified version of the segment cross graph.</td>
</tr>
<tr>
<td>1999[83]</td>
<td>Via</td>
<td>Two layer constrained via minimization, signed hyper-graph model.</td>
</tr>
<tr>
<td>1999[84]</td>
<td>Via</td>
<td>Two layer constrained via minimization, LAP graph model.</td>
</tr>
<tr>
<td>1999[85]</td>
<td>Via</td>
<td>The same graph model as [84], genetic algorithm.</td>
</tr>
<tr>
<td>2001[86]</td>
<td>Via</td>
<td>Two layer constrained via minimization, combing hill-climbing and simulated annealing.</td>
</tr>
<tr>
<td>2004[87]</td>
<td>Via</td>
<td>Multilayer constrained via minimization, conjugate conflict graph model.</td>
</tr>
<tr>
<td>2008[57]</td>
<td>Via &amp; wire congestion</td>
<td>Via minimization under wire congestion constraints by dynamic programming.</td>
</tr>
<tr>
<td>2009[74]</td>
<td>Via</td>
<td>Carefully ordering nets and edges for via number minimization.</td>
</tr>
<tr>
<td>2009[88]</td>
<td>Via &amp; wire congestion &amp; via capacity</td>
<td>An extension of [57].</td>
</tr>
<tr>
<td>2005[90]</td>
<td>Antenna &amp; timing</td>
<td>An improved probabilistic model for the coupling capacitance, tree partitioning to solve the antenna problem.</td>
</tr>
</tbody>
</table>

Table A.4. Track routing

<table>
<thead>
<tr>
<th>Year</th>
<th>Metrics</th>
<th>Algorithms &amp; Techniques &amp; Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007[72]</td>
<td>CMP</td>
<td>Proposing an algorithm based on incremental Delaunay triangulation to uniformly distribute the wires.</td>
</tr>
<tr>
<td>Year</td>
<td>Metrics</td>
<td>Algorithms &amp; Techniques &amp; Features</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2008</td>
<td>CMP&amp;ECP</td>
<td>The same algorithm as in [92].</td>
</tr>
<tr>
<td>1985</td>
<td>Random defects</td>
<td>Channel routing, horizontal conflict graph.</td>
</tr>
<tr>
<td>1993</td>
<td>Random defects</td>
<td>Channel routing, net burying, net bumping and via shifting.</td>
</tr>
<tr>
<td>1995</td>
<td>Random defects</td>
<td>The same as [95].</td>
</tr>
<tr>
<td>1995</td>
<td>Random defects</td>
<td>ILP formulation to consider the wire length in the vertical layer.</td>
</tr>
<tr>
<td>2002</td>
<td>Random defects</td>
<td>Non-tree routing, ILP formulation and greedy heuristic.</td>
</tr>
<tr>
<td>2006</td>
<td>Random defects and timing</td>
<td>ILP formulation, traveling salesman problem heuristic for a good solution, and MinMaxMin heuristic for delay improvement.</td>
</tr>
<tr>
<td>2008</td>
<td>Random defects and timing</td>
<td>Worst case Elmore delay calculation by selecting a subset of the possible open positions.</td>
</tr>
<tr>
<td>2009</td>
<td>Random defects</td>
<td>Incorporating shorts into the non-tree methods.</td>
</tr>
<tr>
<td>2009</td>
<td>Random defects</td>
<td>L-shape and U-shape redundant wire insertion.</td>
</tr>
<tr>
<td>2005</td>
<td>Via</td>
<td>Maximizing the redundant via insertion rate and using Lagrangian relaxation to solve the problem.</td>
</tr>
<tr>
<td>2005</td>
<td>Via</td>
<td>Grid density guided maze routing for redundant via insertion.</td>
</tr>
<tr>
<td>2006</td>
<td>Via</td>
<td>A cost function aware of redundant vias.</td>
</tr>
<tr>
<td>2009</td>
<td>Via and DPT</td>
<td>Considers redundant via insertion and double patterning simultaneously.</td>
</tr>
<tr>
<td>1988</td>
<td>Via</td>
<td>Two layer channel routing aware of via minimization.</td>
</tr>
<tr>
<td>1989</td>
<td>Via</td>
<td>Ripping up vias and rerouting, via shifting.</td>
</tr>
<tr>
<td>1990</td>
<td>Via</td>
<td>Multilayer unconstrained via minimization, stack vias not allowed.</td>
</tr>
<tr>
<td>1991</td>
<td>Via</td>
<td>Two layer unconstrained via minimization.</td>
</tr>
</tbody>
</table>
### Table A.6. Detailed routing (b)

<table>
<thead>
<tr>
<th>Year</th>
<th>Metrics</th>
<th>Algorithms &amp; Techniques &amp; Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1998[110]</td>
<td>Antenna</td>
<td>Partially modifying the nets to reduce cumulative antenna ratio.</td>
</tr>
<tr>
<td>2005[113]</td>
<td>OPC</td>
<td>Maze routing for minimum length minimum light interference.</td>
</tr>
<tr>
<td>2005[114]</td>
<td>OPC</td>
<td>Similar to [113].</td>
</tr>
<tr>
<td>2008[116]</td>
<td>OPC</td>
<td>Proposing a analytical formula obtained by empirical fitting, quasi-inverse technique.</td>
</tr>
<tr>
<td>2010[118]</td>
<td>OPC</td>
<td>Pattern based hotspot detection, boolean satisfiability solver.</td>
</tr>
<tr>
<td>2009[105]</td>
<td>DPT and via</td>
<td>Considering redundant via insertion and DPT simultaneously.</td>
</tr>
</tbody>
</table>

### Table A.7. Post-routing (redundant via insertion)

<table>
<thead>
<tr>
<th>Year</th>
<th>Metrics</th>
<th>Algorithms &amp; Techniques &amp; Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004[55]</td>
<td>Random defects &amp; Via</td>
<td>Reporting an new tool that does wire displacement and redundant via insertion.</td>
</tr>
<tr>
<td>2006[121]</td>
<td>Via</td>
<td>Redundant via insertion, conflict graph construction based on orthogonal range searching on R-trees, and maximum independent set formulation.</td>
</tr>
<tr>
<td>2006[122]</td>
<td>Via</td>
<td>Redundant via insertion, maximum weighted maximum independent set formulation.</td>
</tr>
<tr>
<td>2006[104]</td>
<td>Via</td>
<td>Redundant via insertion, proposing a minimum weighted maximum matching formulation.</td>
</tr>
<tr>
<td>2007[105]</td>
<td>Via</td>
<td>Redundant via insertion, linear programming and relaxation.</td>
</tr>
<tr>
<td>2008[124]</td>
<td>Via</td>
<td>Redundant via insertion, ILP formulation and speed up techniques.</td>
</tr>
<tr>
<td>2008[125]</td>
<td>Via</td>
<td>Redundant via insertion, network flow formulation, and min-cost maximum flow algorithm.</td>
</tr>
<tr>
<td>2009[128]</td>
<td>Via</td>
<td>Redundant via insertion, considering the possibility of wire bending.</td>
</tr>
<tr>
<td>2009[105]</td>
<td>Via and DPT</td>
<td>Similar to [124]</td>
</tr>
</tbody>
</table>
### Table A.8. Post-routing (Wire sizing, spreading and layout compaction)

<table>
<thead>
<tr>
<th>Year</th>
<th>Metrics</th>
<th>Algorithms &amp; Techniques &amp; Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995</td>
<td>Random defects</td>
<td>Layout compaction</td>
</tr>
<tr>
<td>2004</td>
<td>Random defects and via</td>
<td>Reporting an new tool that does wire displacement and redundant via insertion.</td>
</tr>
<tr>
<td>2007</td>
<td>Random defects and CMP</td>
<td>Concurrent wire sizing and spreading, dummy fill for wire density uniformity.</td>
</tr>
<tr>
<td>2005</td>
<td>OPC</td>
<td>Wire spreading and ripping-up-and-reroute guided by edge placement error.</td>
</tr>
<tr>
<td>1998</td>
<td>PSM</td>
<td>Formulated as a minimum distortion problem, an algorithm based on the Voronoi method.</td>
</tr>
<tr>
<td>1999</td>
<td>PSM</td>
<td>T-joint problem formulation.</td>
</tr>
<tr>
<td>2008</td>
<td>PSM</td>
<td>Wire spreading, multi-layer odd face pairing formulation, mixed integer linear programming formulation.</td>
</tr>
</tbody>
</table>

### Table A.9. Post-routing (jumper insertion)

<table>
<thead>
<tr>
<th>Year</th>
<th>Metrics</th>
<th>Algorithms &amp; Techniques &amp; Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004</td>
<td>Antenna</td>
<td>Greedy algorithm on spanning trees.</td>
</tr>
<tr>
<td>2005</td>
<td>Antenna</td>
<td>Optimal greedy algorithm on Steiner trees.</td>
</tr>
<tr>
<td>2006</td>
<td>Antenna</td>
<td>Similar to [132] with extension to deal with blockages.</td>
</tr>
<tr>
<td>2006</td>
<td>Antenna</td>
<td>The same as [132].</td>
</tr>
<tr>
<td>2006</td>
<td>Antenna</td>
<td>Considering charge sharing effect.</td>
</tr>
</tbody>
</table>

### Table A.10. Post-routing (diode insertion)

<table>
<thead>
<tr>
<th>Year</th>
<th>Metrics</th>
<th>Algorithms &amp; Techniques &amp; Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>Antenna</td>
<td>Min-cost flow algorithm.</td>
</tr>
<tr>
<td>2006</td>
<td>Antenna</td>
<td>Similar to [136].</td>
</tr>
</tbody>
</table>
Bibliography


136


