# UNIMODULARITY IN SOLVING ILP MODEL

# UNIMODULARITY IN SOLVING ILP MODELS OF THE GLOBAL ROUTING PROBLEM

By
MIN JING JESSIE LIU.

Computational Eng. & Sci

A Thesis

Submitted to the School of Graduate Studies

In Partial Fulfillment of the Requirements

For the Degree

Master of Applied Science

McMaster University

MASTER OF APPLIED SCIENCE, (Aug, 2009) McMaster University

(Computational Eng & Sci.) Hamilton, Ontario

TITLE: UNIMODULARITY IN SOLVING ILP MODELS OF THE GLOBAL ROUTING PROBLEM

AUTHOR: MIN JING JESSIE LIU

SUPERVISOR: Dr. Tamás Terlaky

Dr. Antoine Deza

NUMBER OF PAGES: ix & 48

# Abstract

The global routing problem is becoming more and more important in the design of today's integrated circuits. A small chip may contain up to millions of components and wires. Although global routing can be formulated as an integer linear programming problem, it is hard to solve directly using currently available solvers. We discuss a relaxation of the problem to a linear programming (LP) formulation with a fractional solution. However, the relaxation yields an NP-hard problem. In this thesis, we introduce three relaxations: the primal *(Pc)*, the Lagrange dual (*Dc*), and the unimodular (*Pl*) formulation. At optimality, all three problems have the same objective value. A new way to tackle the LP problem is introduced: first solve the *Dc* and try to find Lagrange multipliers in order to build the *Pl* model, from which an integer solution can be obtained directly. An implementation based on the discussed approaches was tested using IBM benchmarks.

# Acknowledgments

I would like to express my gratitude to all of those who made it possible to complete this thesis: Dr Tamás Terlaky, Dr Laleh Behjat, Dr. Antoine Deza, and Logan Rakai.

My special thanks go to JiaPing Zhu whose valuable advices and extended knowledge helped me a lot.

I appreciate the great aid and support from all the members of the Advanced Optimization Laboratory.

I would also like to thank my parents for their understanding and continuous support.

Finally, I would like to thank my friend Huang Huang and William Hua for the help and support.

**Contents**

# List of Figures

# List of Table

# Chapter 1 Introduction

Integrated Circuits (also known as ICs, microcircuits, or chips) play an important role in today's manufacturing integrated circuit design. An integrated circuit may contain thousands or millions of components and interconnecting wires. However, how to properly layout these components for optimal circuit performance is still a problem of great interest in industry. In general, such problems are called VLSI problems (very large scale integration circuit design problems), and although they can be presented as linear programming problems, they tend to be extremely complex and time-consuming. [1]

## 1.1 History of IC

In the beginning of the 1960s, Small-Scale Integrated (SSI) circuits were produced, usually containing no more than ten transistors. In the late 1960s, Medium-Scale Integrated (MSI) circuits were produced, where the number of the transistors increased into the hundreds. Although their production costs were higher than those of SSI devices, MSI allowed more complex systems to be produced using smaller circuit boards, and less assembly work. In the mid 1970s, the number of the transistors increased to the tens of thousands, at which point it was coined as Large-Scale Integration (LSI). The final step in the development process, starting in the 1980s and continuing until today, is referred to as "very large-scale integration" (VLSI). These circuits may contain transistors on the scale of hundreds to several billions.

## 1.2 Physical Design

In general, digital IC design can be divided into three parts. The first step is Electronic System Level (ESL) design. In this step, the user's functional specifications are created using a variety of languages and tools. Then in the next step, Register Transfer Level (RTL) design, the user specifications will be converted into a register transfer level description. The last step is Physical Design, which is used to determine the physical shape of a circuit, location of the cells, and path of the nets in the circuits. Physical Design also consists of three stages: floor planning, placement, and routing.

The first stage is floor planning. In this stage, we determine the size, shape and the location for each small cell. In general, the circuit will be segmented into small rectangular blocks. The second stage is placement. Since the entire block has been fixed in the floor planning stage, a net is used to determine where each block needs to be connected. This net is called a benchmark.

The last stage is routing. Since a larger percentage of the delay of circuits occurs in the wires, routing is the most important stage in VLSI Design. This stage is performed in two steps: global routing and detail routing. During global routing, a solution will be generated from the benchmark. Then the detail routing will take the solution from the global routing and route it physically. The aim of global routing is to find an optimal path without violating any of the circuit's physical constraints, i.e. the minimum length of the wires, routing of all nets, and minimizing the overflow. The global routing problem is NP-hard [10]. Given its size tens of thousands, it usually solved using heuristic algorithms.

# Chapter 2 Global routing

The goal of global routing is to generate an optimal solution for the circuit that satisfies different objectives. The input of the global routing problem consists of a net-list that indicates the interconnections between blocks and the position of each block. Usually the global routing problem is presented as a graph problem. And there are three sub-problems in the global routing problem: graph generation, tree generation and route generation [19].

## 2.1 Graph Generation

Let us begin with an introduction to graph theory. Usually a graph is represented by drawing a dot for every vertex, and drawing an arc between two vertices if they are connected by an edge. A directed graph G= (V, E) consists of a finite set of vertices V and a finite set of edges (arcs) E. If an edge leads from u to v, then we record the edge e as u → v. We also say that v is adjacent to u.

We also define an undirected graph G= (V, E) with V recording the set of the vertices and E recording the set of the edges that need to be connected. Thus each edge in the undirected graph is an unordered pair of vertices. So if (u, v) is an edge in G, then (v, u) is also an edge in G.

The global routing problem can be presented as an undirected graph G = (V, E). Figure 1 below explains the problem. In Figure 1, the left graph shows a typical routing area of the circuit, and its overlaying graph is shown on the right side. After we obtain the overlaying graph, we transfer the overlaying graph to a grid graph (left graph of Figure 2). Each vertex represents a block while each edge represents the routing channel that the wires can be placed.

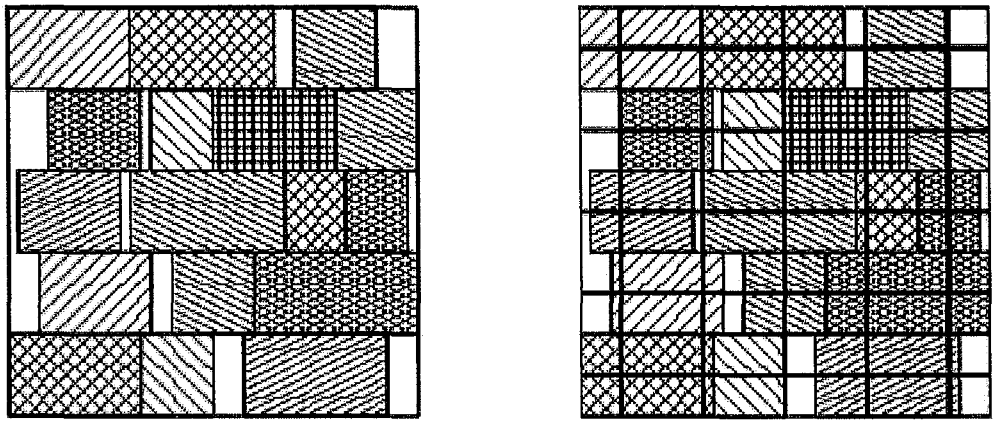


*Figure 1 A typical routing area and its overlaying graph*

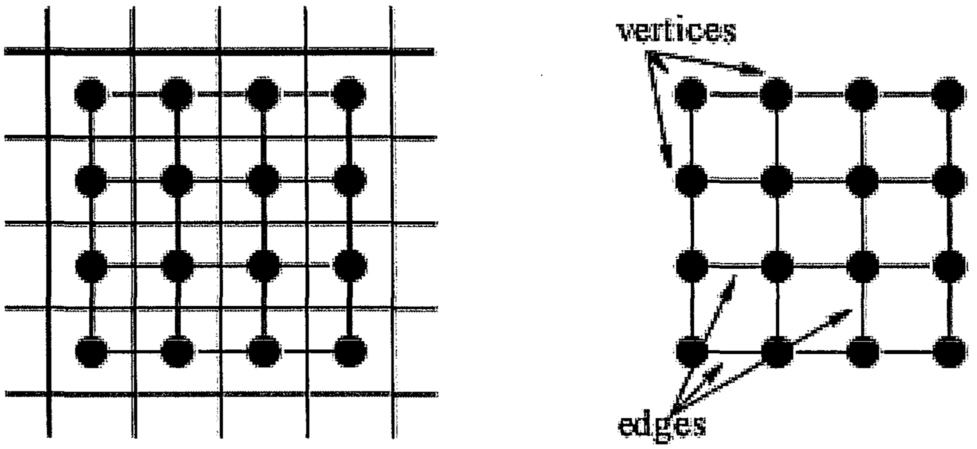


*Figure 2 Grid graph and the final graph*

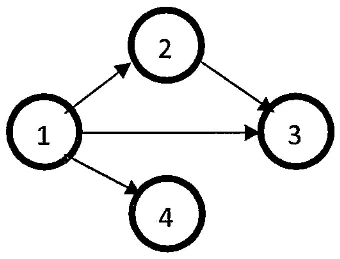
In a real circuit, the channels are placed in several metal or poly silicon layers on top of the transistors, which are made of silicon material. Thus each edge will have a limited number of wires that can be placed on it. This limit is the capacity of the edge. Once we finish building the final graph, we can record the circuit as G = (V, E).

There are two ways to represent a graph: an adjacency list or an adjacency matrix.

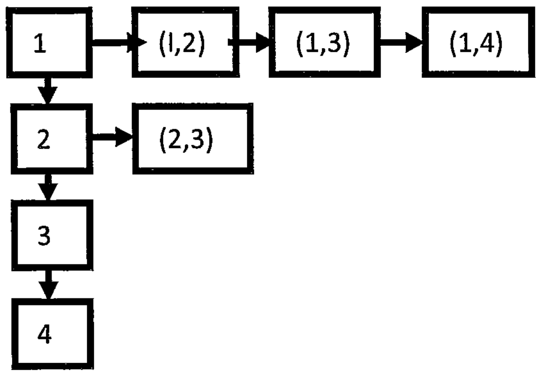
An adjacency list is the representation of all edges or arcs in a graph as a list. Usually, adjacency lists are unordered. An adjacency list that represents a graph G = (V, E) consists of an array L, for each $i \in V$, $L_i$ is a list of all vertices j such that $(i,j) \in E$.

An adjacency matrix is the representation of all edges or arcs in a graph as a matrix. For a finite graph with n vertices, the adjacency matrix is a $n \times n$ matrix. Suppose we have V = {1, 2, ..., n}, the non-diagonal entry $a_{ij}$ in the adjacency matrix is the number of edges from vertex i to vertex j, and the diagonal entry $a_{ii}$ is either once or twice the number of edges (loops) from vertex i to itself. Typically, an undirected graph may use the former convention of counting loops twice while a directed graph may use the latter convention.

An adjacency matrix requires $\Theta(|V|^2)$ memory usage while an adjacency list requires $\Theta(|V|+|E|)$ memory usage. Thus an adjacency matrix is suited for representing dense graphs and an adjacency list is suited for representing sparse graphs. An example of different representations is shown below.



(a) A graph G



$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

(b) Adjacency list of (a)

(c) ) Adjacency matrix of (a)

*Figure 3 Different representations of a directed graph G*

In the global routing problem, the graph generated from the circuit is an undirected graph. From the benchmark, we will know the number and the location of the blocks on each layer. We will also know for each layer how many blocks need to be connected. Thus, in the next step, tree generation, we can generate the tree for each layer (net), and obtain the full graph of the circuit.

## 2.2 Tree Generation

A tree is an undirected simple graph in which any two vertices are connected by exactly one simple path. The edges of a tree may or may not be weighted. Thus a tree can also be represented as an adjacency list or an adjacency matrix.

During tree generation, we need to generate a single tree that connects all the vertices in each layer according to the requirements for routing. Some graph techniques can be used to produce these trees.

### 2.2.1 Spanning tree

A spanning tree is a tree composed of all the vertices and some of the edges of G. A minimum spanning tree is a spanning tree with minimum weight. There are two classic minimum spanning tree algorithms, namely Prim's algorithm [17] and Kruskal's algorithm [13]. These algorithms run in polynomial time and are greedy algorithms. In general, a spanning tree can be built very fast [14], but sometimes it is not the optimum choice.

### 2.2.2 Minimum Steiner tree

We introduce here another tree called the minimum Steiner tree. The minimum Steiner tree problem is very similar to the minimum spanning tree problem. In the Steiner tree problem, we introduce extra intermediate vertices and edges to the graph in order to reduce the total length of the tree. The extra intermediate vertices are called Steiner points or Steiner vertices. A Steiner tree may not be unique. Figure 4 is used to show the difference between a spanning tree and a Steiner tree. The spanning tree is drawn with solid lines, while the Steiner tree is drawn with dashed lines. The white points are the Steiner vertices.

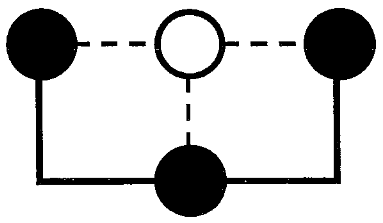


*Figure 4 A net shows a spanning tree and a Steiner tree.*

A VLSI circuit may contains hundreds of thousands cells/blocks, and the general minimum Steiner tree problem is NP-hard. The GeoSteiner Algorithm [20] is used to find the Steiner tree with the minimum length. There are also some other algorithms for finding the minimum Steiner tree such as Mix-flute [5], and GOBLIN [8].

## 2.3 Route Generation

A VLSI circuit consists of several layers. If we only produce a single tree in each layer during the tree generation, the whole net may not be a valid solution. For example, imagine we have two layers in the circuit with six vertices in each layer. A figure below shows two layers (in the real physical circuit, the left layer overlaps the right layer), while the black points are the vertices that need to be connected. The capacity for each edge of the circuit is one. Here, we only consider the 2D model.

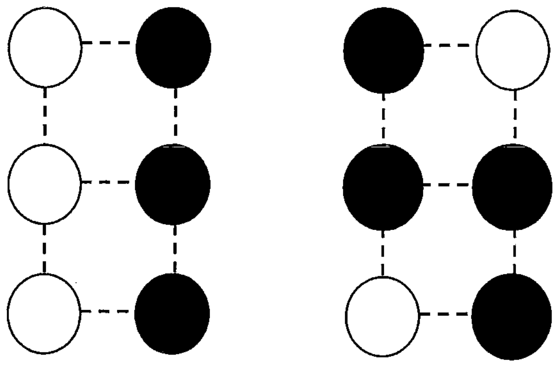


*Figure 5 Two layers on a simple circuit example*

If we generate the tree on the layers separately, the solution appears as in Figure 6. The solid line is the tree for each layer. However, the bottom-most right edge of the circuit appears twice, while the capacity for that edge is only one. Therefore, the solution is not valid.

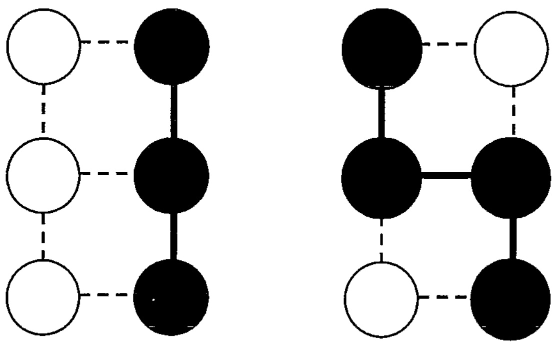


*Figure 6 One possible solution for the circuit if generate the tree separately.*

In the following example, two techniques are introduced: sequential [12][2][3] and concurrent techniques [15][4].

## 2.3.1 Sequential techniques:

At first, we order all the nets based on their importance, then each net is routed separately based on the constraints from previous nets. Several global routing techniques have been developed. Most techniques are based on maze search [9] or line search approaches. However, these routers can only be used to produce two-terminal nets and usually take a very long time to run. A shortest path algorithm can be used to route multi-terminal nets [6][11], while reducing the running time. These techniques are called sequential algorithms. Sequential routers can produce the solution directly without building the LP problem, however, due to their sequential nature; the solution may not be optimal. If the order of the nets is different, the solution may change drastically.

Using Figure 5 as an example:

If we generate the tree on the left layer of Figure 5 first, the solution of circuit will be

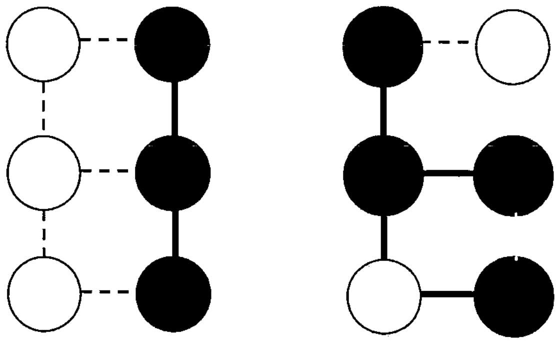


*Figure 7 Solution of the circuit if generate the tree on the left one first*

However, if we generate the tree on the right layer of Figure 5 first, the optimal solution does not exist.

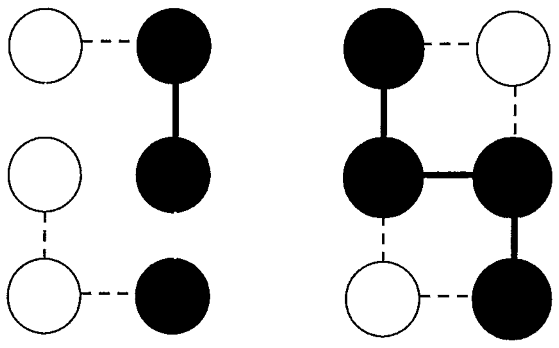


*Figure 8 After generating the tree on the right layer, the vertices on the left layer are disconnected*

### 2.3.2 Concurrent Algorithms:

Concurrent algorithms are another way to find an optimal solution of a VLSI problem. At first, it will produce a lot of possible trees in each net, allowing us to formulate the problem as an integer linear problem (ILP) with minimization or maximization under some constraints. Although there are a lot of linear programming solvers that can be used to solve the ILP problem, due to the size of the ILP problem, it is hard to obtain the integer solution directly. In this thesis, we will introduce a Lagrangian relaxation technique to solve the problem, and compare it with the classical techniques. Three different concurrent routers are used to generate the tree: the congestion router, the subnet router and the mix-flute router.

# Chapter 3 Unimodularity in the LP problem

In this chapter, we present some special cases of the LP problem which may help us to solve the VLSI ILP problem.

## 3.1 Unimodular matrix

**Definition 3.1.1**

A unimodular matrix M is a square integer matrix with determinant +1 or -1.

The simplest example is the identity matrix I, since Det (I) =1.

**Definition 3.1.2**

A totally unimodular matrix is a matrix for which every non-singular submatrix is unimodular.

From the definition, we also know that any totally unimodular matrix has only 0,+1 or -1 entries.

A small example shown below is totally unimodular:

$$A = \begin{bmatrix} -1 & -1 & 0 & 0 & 0 & +1 \\ +1 & 0 & -1 & -1 & 0 & 0 \\ 0 & +1 & +1 & 0 & -1 & 0 \\ 0 & 0 & 0 & +1 & +1 & -1 \end{bmatrix}.$$

## 3.2 Total Dual Integrality

Consider the linear program defined as

Primal problem:

$$\begin{aligned} \max \quad & \mathrm{c}^{\mathrm{T}} x \\ s.t. \quad & Ax \leq b \end{aligned}$$

Where A and b are rational and the dual problem may be written as:

Dual problem:

$$\begin{aligned} \min \quad & b^{T} y \\ s.t. \quad & Ay = c \\ & y \geq 0 \end{aligned}$$

We will use the following definition:

**Definition 3.2.1** [ 7]

The system of inequalities by $Ax \leq b$ is Total Dual Integral (TDI) if for all integral vectors c, the dual program has an integral optimal solution whenever the optimal value is finite.

It can then be shown that the primal problem also has an integral optimal solution for all c, if A is rational and b is integral.

**Definition 3.2.2**

A matrix A is totally unimodular if and only if the system $\{Ax \leq b, x \geq 0\}$ is TDI for all integral vectors b.

From definition 3.2,1 and 3.2.2, we can get the following corollary:

**Corollary 3.2.3**

If we have a primal LP problem in which c is an integer vector, and A is a totally unimodular matrix, the primal LP problem will have an integer optimal solution.

*Proof:*

Consider a totally unimodular matrix A, then the system $\{Ax \leq b, x \geq 0\}$ is TDI from the definition 3.2.2. Thus, the dual problem and the primal problem will have an integer optimal solution.

# Chapter 4 Modeling

The routing technique we used in this paper is based on the ILP model. Some of the most common notation used in the formulation is introduced.

n: Total number of the nets.

p: Total number of edges.

t: Total number of trees.

*Pd*: Primal Discrete global routing formulation.

*Pc*: Primal Continuous relaxed global routing formulation.

*Dc:* Dual Continuous relaxed global routing formulation.

*Pl*: Primal Lagrange of the partial Lagrange relaxation of the global Routing formulation.

*Dl*: Dual Lagrange of the partial Lagrange dual relaxation of the global Routing formulation.

*Pc-u*: Primal Continuous relaxed global routing formulation with added upper bounds on penalty.

*Dc-u*: Dual Continuous relaxed global routing formulation with added upper bounds on penalty.

*Pl-u*: Primal Lagrange of the partial Lagrange relaxation of the global Routing formulation with added upper bounds on penalty.

## 4.1 Modeling generation

Initially, we generate the set of trees for each net k, and the total number of trees is t. $y_j$ is used to record the status of tree j. It equals one when the tree is used and zero otherwise. $w_j$ is the weight variable for tree j. Usually the weight is a combination of length, the number of bends and congestion. $a_{ij}$ is a binary constant that is one if tree j passes through edge i and zero otherwise.

The global routing problem is then formulated as:

$$
\begin{array}{lll}
P_d & \min & \sum_{j=1}^{t} w_j y_j \qquad (1) \\
 & s.t.: & \sum_{y_j \in N_k} y_j = 1 \quad k = 1,2,\ldots,n \qquad (2) \\
 & & \sum_{j=1}^{t} a_{ij} y_j \le cap_i \quad i = 1,2,\ldots,p \ (3) \\
 & & y_j \in \{0,1\}, \qquad j = 1,2,\ldots,t \quad (4)
\end{array}
$$

Our aim is to minimize the total wire length and the congestion of the circuit. The constraints (2) ensure that for each net only one tree can be chosen. The constraints (3) enforce that the number of trees passing through an edge does not exceed its capacity. In the end, the solution of this model should be binary. We could rewrite *Pd* in matrix format:

$$
\begin{aligned}
P_d \quad & \min \quad w^T y \\
& s.t.: \quad \mathbf{H}\mathbf{y} = 1 \\
& \qquad\quad \mathbf{A}\mathbf{y} \le \boldsymbol{Cap} \\
& \qquad\quad y \in \{0,1\}
\end{aligned}
$$

***w*** is the weight vector composed of $w_j$, **H** is a totally unimodular matrix from the definition 3.1.2. **A** is a matrix such that each column represents a given tree and each row represents the edges that have been used by that tree. ***Cap*** is the maximum capacity for each edge.

During testing, we found most of the LP problems are infeasible. Usually, there are several ways to resolve this. We could generate more trees in the routing generation step, add an artificial variable vector to satisfy the condition on the capacity (in this case, overflow may occur), or apply both. Here, we introduce a penalty vector **z**:

$$
\begin{aligned}
P_f \quad & \min \quad w^T y + m^T z \\
& s.t.: \quad \mathrm{Hy} = 1 \\
& \qquad\quad \mathrm{Ay} \le Cap + \mathrm{z} && (5) \\
& \qquad\quad y \in \{0,1\}, && (6) \\
& \qquad\quad \mathrm{z} \ge 0
\end{aligned}
$$

m is the weight for the penalty **z**. After we add the penalty **z**, the model will always have an optimal solution.

## 4.2 Solve the Model

Most ILP problems can be solved directly using some solver. Typically, the size of the VLSI ILP problem is extremely large; it cannot be solved directly like other small problems. We introduce some models and will see how they can be used on based the solver technology.

### 4.2.1 Relaxed Primal Problem (Pc):

We relax the binary constraints of *Pf* so that the feasible global ILP problem *Pf* is relaxed as a LP problem *Pc*. Typically, the time for solving the LP problem will be less than the time for solving ILP problem.

$$\begin{array}{llll} P_c & \min & w^T y + m^T \mathbf{z} & (7) \\ & s.t.: & \mathbf{H}\mathrm{y} = 1 & (8) \\ & & \mathrm{A}\mathrm{y} \le \boldsymbol{Cap} + \mathbf{z} & (9) \\ & & 0 \le y \le 1 & (10) \\ & & 0 \le \mathbf{z} & \end{array}$$

In here, we assume all the values of m are the same. The total number of fractional coordinates in the solution of *Pc* depends on the LP method that been used. Usually, interior point method (IPM) will produce more fractional solutions than other methods. Then we can proceed with rounding to find an integer solution. For our case, we can directly obtain an integer solution using the structure of H.

### 4.2.2 Lagrange Dual Problem (*Dc*):

The Lagrange dual problem *Dc* of *Pc* will have the following form:

$$
\begin{array}{lll}
Dc \quad \max & -\mu^{\mathrm{T}}1 - v^T \boldsymbol{Cap} - \lambda^T 1 & (11) \\
\quad s.t.: & -\mathbf{H}^{\mathrm{T}}\mu - \mathbf{A}^T v - \lambda^T \le w, & (12) \\
 & m^{\mathrm{T}} \ge v^T \ge 0, & (13) \\
 & \lambda \ge 0, & (14)
\end{array}
$$

Note that the variable $\mu$ is free. We get this form by writing out the Lagrange formula of Pc. Then we can get the Dc by using the KKT (Karush-Kuhn-Tucker) conditions. See for example: C. Roos [18], S. J. Wright[21] and Y. Ye [22] and references therein.

### 4.2.3 Unimodular Problem (Pl)

We want to utilize the unimodularity of the part of the constraints of the original relaxed problem, *Pc*, so that we could obtain the binary solution directly. The partial Lagrange relaxation of *Pc* can be written as:

$$
\begin{array}{lll}
P_l & \min & w^T y + m^T \mathbf{z} + \alpha^{\mathrm{T}}(\mathrm{A}y - \boldsymbol{Cap} - \mathbf{z}) \\
 & s.t.: & \mathbf{H}y = 1 \\
 & & 0 \le y \le 1
\end{array}
$$

Since H is totally unimodular, if we know the value of **z**, *m*, and $\alpha$, we could write the solution directly without using a solver.

### 4.2.4 Partial Lagrange dual problem (*Dl*)

According to Theorem 5.1, which is given in the next chapter, the optimal value of Pc and the optimal value of the partial Lagrange dual of *Pc* will be the same. The partial Lagrange dual *Dl* can be written as:

$$D_l \qquad \begin{array}{ll} \max & g(\alpha) \\ s.t. & \alpha \geq 0 \end{array} \qquad (16)$$

$g(\alpha)$ is the optimal value of Pl. Since the matrix is totally unimodular, this problem is a unimodular optimization problem and a zero-one solution can be obtained by using the simplex method.

# Chapter 5 Theorem and solving technique

## 5.1 Theorem

We already built several models based on the *Pc* problem. Theorem 5.1.1 - 5.1.3 help us find out the relationship between these models. We can use these theorems to solve the *Pc* problem, and obtain an integer solution of *Pc*.

**Theorem 5.1.1:** The optimal objective value of *Pc*, *Dc* and *Dl* are equal, i.e, we have

$OPT_{Pc} = OPT_{Dc} = OPT_{Dl}$.

*Proof:*

Any feasible solution of *Pc* is a feasible solution of *Dl*, we will have $OPT_{Pc} \geq g(\alpha)$, for any $\alpha \geq 0$. So $OPT_{Pc} \geq OPT_{Dl}$

For any $\alpha = \upsilon$, and any $\lambda \geq 0$, since the feasible solution set of the problem *Pl* is contained in that of *Dc*. We have $g(\alpha) \geq OPT_{Dc}$. As a result, $OPT_{Dl} \geq OPT_{Dc}$.

Since *Pc* is a linear programming problem, the strong duality theorem holds. We have $OPT_{Pc} = OPT_{Dc} = OPT_{Dl}$.□

When we solve the model *Dl*, the value of opt $\alpha$ will be determined. We could use $\alpha$ to solve Pl, which is a unimodular problem. And the solution from *Pl* is binary for sure according to Corollary 3.2.3.

**Theorem 5.1.2:**

If $(\mu^*, v^*, \lambda^*)$ is an optimal solution of the dual problem *Dc*, $\alpha = v^*$ is an optimal solution of partial Lagrange dual problem *Dl*.

*Proof:*

Assume $(\mu^*, v^*, \lambda^*)$ is an optimal solution of the Dc. And $(y^*, z^*)$ is the optimal solution of *Pc*. We will have $v^{*T}(Ay^* - cap - z) = 0$. If $\alpha = v^*$, with $(y^*, z^*)$, and we solve *Pl*, the objective value of $g(v^*)$ will reach its upper bound of $OPT_{Pc}$. Thus, $v^*$ is an optimal solution of *Dl*. □

**Theorem 5.1.3:**

If $\alpha = v^*$ is an optimal solution of *Dl*, there exits $(\mu^*, v^*, \lambda^*)$ that is an optimal solution of *Dc*.

*Proof:*

Assume $\alpha^*$ is the optimal solution of *Dl*, and $(y^*, z^*)$ is the corresponding optimal solution of the problem *Pl*. If $(\mu^*, v^*, \lambda^*)$ is an optimal dual solution of the linear problem *Pl* with $\alpha = \alpha^*$, by the complementarily condition, we will have:

$v^{*T}(Hy^* - 1) = \lambda^{*T}(y^* - 1) = 0,$

Thus, the objective value of *Dc* will reach its upper bound $OPT_{Dl}$. Therefore, $(\mu^*, \alpha^*, \lambda^*)$ is an optimal solution of *Dc*.□

## 5.2 Solution Technique

There are several ways to find a solution for the ILP model of the global routing problem. Because of the size of the ILP problem, it is hard to get the binary solution directly from the solver. Based on Theorem 5.1.1-5.1.3 that introduced before and the models that we have, we propose a new way to solve the problem.

First, we will solve the primal problem Pc. There are two situations that can occur when *Pc* is solved:

1) An optimal binary solution is found.

2) An optimal fractional solution is found.

Here we discuss different approaches according to those two situations.

### 5.2.1 Optimal binary solution:

The primal relaxed problem *Pc* can have multiple optimal solutions by using different methods. If we do get the binary solution, we do not need any further calculations. Usually, IPMs may get more fractional solutions than dual simplex or primal simplex.

### 5.2.2 Optimal fractional solution:

Most of the times, we get a fractional solution from *Pc*. However, what we need is a binary solution. Figure 9 shows the classical technology to solve the VLSI problem. After we obtain the fractional solution, we could round the solution to zero or one by using the condition of H.

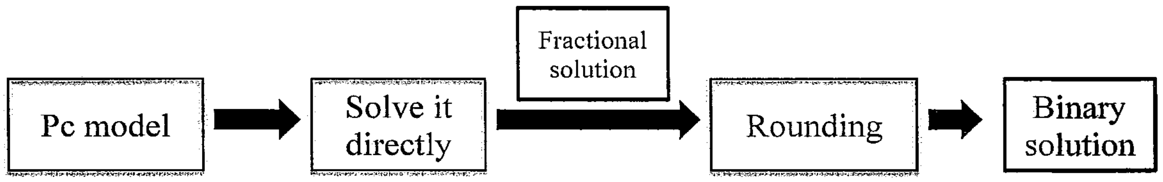


*Figure 9 Directly solve Pc and round to obtain a binary solution.*

According to Theorems 5.1.1, 5.1.2 and 5.1.3, the method can be applied. We could find the optimal Lagrange multiplier $v^*$ by solving *Dc*. Then let $\alpha = v^*$, and solve *Pl*. From chapter 3, we know that the solution of *Pl* will be an integer array. In the end, there still $OPT_{Pl} = OPT_{Pc}$.

Figure 10 shows the flow chart of the unimodular technology

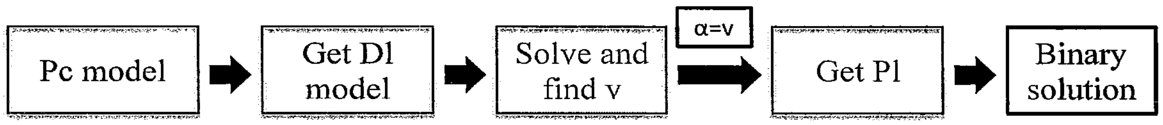


*Figure 10 Unimodularity solver*

# Chapter 6 Testing and conclusions

## 6.1 Testing Detail

We tested our techniques by solving four sample circuits. The four circuits were chosen from the IBM-Place 2.0 benchmarks suite and the placement results were generated by server Dragon that used at the University of Calgary.

Three routers were used during this testing: Congestion, Mix-flute, and Subnet. Each router has its own objective function in order to satisfy some characteristics of the circuits. Thus, the coefficients of the objective function are different.

Cplex is the main solver during the test. Three LP methods were used to solve the problem: Primal simplex, Dual simplex, and interior point method (IPM).

The tests were performed on the server Dantzig, 8 * AMD Opteron 885, with 64GB RAM.

## 6.2 Data results and conclusions

During the experiments, three different methods were used to solve the LP problem, *Pc* and *Dc*. Due to the unimodularity of the *Pl*, we could directly write out the solution if we knew the value of the Lagrange multiplier from *Dc*.

| Method | Problem name | |
|---|---|---|
| | Pc | Dc |
| 1-DD | Dual simplex | Dual simplex |
| 2-PP | Primal simplex | Primal simplex |
| 3-II | IPM | IPM |

*Table 1 Different LP Methods used to solve Pc and Dc*

### 6.2.1 Compare for the overflow

The number of overflows is defined as *Ay - Cap*. Since the original problem *Pd* is infeasible, overflow may occur on some edges. If the value in the *Ax-Cap* is larger than zero, it indicates that overflow occurred.

Table 2 shown below shows the amount of overflow obtained by using the Mix-flute router with different solving methods. The value of weight is the value m in the model *Pc*. For results generated using the subnet router and the congestion router, refer to Appendix A.

$$\%\text{ of increase} = \frac{\text{number of overflow in } Pl - \text{number of overflow in } Pc}{\text{number of overflow in } Pc} \times 100\%$$

| Mix-flute | | weight 1 mix_flute | | | weight 100 mix_flute | | |
|---|---|---|---|---|---|---|---|
| filename | method | number of overflow (Pc)(integer solution) | number of overflow (Pl) (integer solution) | % of increase | number of overflow (Pc)(integer solution) | number of overflow (Pl) (integer solution) | % of increase |
| ibm01 | D | 578 | 750 | 29.76% | 455 | 768 | 68.79% |
| | P | 565 | 771 | 36.46% | 457 | 768 | 68.05% |
| | I | 616 | 623 | 1.14% | 537 | 545 | 1.49% |
| ibm02 | D | 813 | 1063 | 30.75% | 789 | 1091 | 38.28% |
| | P | 802 | 1069 | 33.29% | 775 | 1080 | 39.35% |
| | I | 882 | 889 | 0.79% | 868 | 875 | 0.81% |
| ibm03 | D | 311 | 686 | 120.58% | 321 | 716 | 123.05% |
| | P | 291 | 662 | 127.49% | 307 | 719 | 134.20% |
| | I | 348 | 357 | 2.59% | 368 | 383 | 4.08% |
| ibm04 | D | 638 | 1271 | 99.22% | 625 | 1269 | 103.04% |
| | P | 662 | 1277 | 92.90% | 606 | 1284 | 111.88% |
| | I | 775 | 793 | 2.32% | 764 | 765 | 0.13% |

*Table 2 Amount of overflow by using Mix-flute with different solving techniques.*

From Table 2, it seems that the classical technology is better. The classical technology gets fewer overflows than the unimodularity method. And it shows the same result when we use the subnet router or the congestion router. Another thing we noticed was that when the weight of z increased, the number of the overflow decreased.

## 6.2.2 Compare for the objective value

Table 3 shows the objective value of *Pc* and *Pl* when using the mix-flute router. This table shows that the objective value is almost the same. This is consistent with the theorem 5.1.1. Appendix A contains the objective value using the congestion router.

Percentage increase between *Pl* and *Pc* = $\frac{\text{obj value in } Pl - \text{obj value in } Pc}{\text{obj value in } Pc} \times 100\%$

| | | weight 1 mix_flute | | | weight 100 mix_flute | | |
|---|---|---|---|---|---|---|---|
| filename | method | obj value in Pc(integer number) | obj value in Pl(integer number) | percentage increase in two model | obj value in Pc(integer number) | obj value in Pl(integer number) | percentage increase in two model |
| ibm01 | D | 26670 | 26863 | 0.72% | 27461 | 27870 | 1.49% |
| | P | 26667 | 26865 | 0.74% | 27468 | 27850 | 1.39% |
| | I | 26643 | 26660 | 0.06% | 27395 | 27397 | 0.01% |
| ibm02 | D | 82428 | 82486 | 0.07% | 83009 | 83170 | 0.19% |
| | P | 82421 | 82496 | 0.09% | 83008 | 83167 | 0.19% |
| | I | 82424 | 82426 | 0.00% | 82980 | 82978 | 0.00% |
| ibm03 | D | 73927 | 74008 | 0.11% | 74553 | 74885 | 0.45% |
| | P | 73924 | 74012 | 0.12% | 74566 | 74908 | 0.46% |
| | I | 73911 | 73910 | 0.00% | 74516 | 74533 | 0.02% |
| ibm04 | D | 99659 | 99775 | 0.12% | 100325 | 100687 | 0.36% |
| | P | 99664 | 99760 | 0.10% | 100328 | 100655 | 0.33% |
| | I | 99654 | 99640 | -0.01% | 100277 | 100298 | 0.02% |

*Table 3 Object value of Pl and Pc by using the Mix-flute router*

### 6.2.3 Comparing the number of fractional elements

When we add the penalty inside the model Pd, the weight of the penalty may also affect the solution. Table 4 shows what happens when we increase the value of weight. The number of fractional elements in the solution of *Pc* also changed. Table 5 shows the size for each problem.

| | | weight 1 mix_flute | weight 100 mix_flute | weight 1 cong | weight 100 cong | weight 1 subnet | weight 100 subnet |
|---|---|---|---|---|---|---|---|
| filename | method | # of fraction | # of fraction | # of fraction | # of fraction | # of fraction | # of fraction |
| ibm01 | D | 253 | 629 | 4 | 513 | 4 | 3198 |
| | P | 87 | 537 | 4 | 511 | 1079 | 1845 |
| | I | 3655 | 3865 | 16 | 796 | 32 | 718 |
| ibm02 | D | 469 | 603 | 1 | 632 | 0 | 132 |
| | P | 270 | 502 | 0 | 339 | 438 | 392 |
| | I | 9062 | 9436 | 17 | 537 | 2 | 279 |
| ibm03 | D | 959 | 1525 | 0 | 1108 | 893 | 1546 |
| | P | 836 | 1461 | 0 | 1102 | 0 | 4743 |
| | I | 9221 | 10430 | 16 | 1403 | 12 | 1024 |
| ibm04 | D | 1105 | 1340 | 3990 | 1129 | 4024 | 1156 |
| | P | 690 | 1236 | 0 | 1132 | 29 | 768 |
| | I | 14855 | 16398 | 28 | 1516 | 12 | 976 |

*Table 4 Number of fractional elements in the Pc*

| | number of rows in H | number of rows in A | number of columns in A |
|---|---|---|---|
| ibm01 | 3006 | 4822 | 25149 |
| ibm02 | 5358 | 6283 | 36331 |
| ibm03 | 5121 | 6735 | 42398 |
| ibm04 | 7431 | 9356 | 59977 |

*Table 5 Size of the model*

Most of the testing data results show that when the value of the weight increases, the number of fractions in the solution also increases, though a few data sets show a decrease, e.g. ibm02 using the subnet router and the primal simplex method. Some other improvements can also be applied to the model. e.g. in model *Pc*, the weight of m is an array, by default, all the weight of z is the same. If we use different values of m according to the weight of the edges in one model, the result may change significantly. However, there currently is no theory that indicates how to find a better value.

## 6.2.4 Comparing the running times

The running time for the classical solving technology is the running time of solving *Pc* plus the running time of the rounding solution. The running time for the new technology is the sum of the running times for *Dc* and *Pl*. Due to the unimodularity of *Pl*, the running time for solving *Pl* is negligible. Table 6 shows the running time for each method with Mix-flute router.

| Mix-flute | | weight 1 mix_flute | | weight 100 mix_flute | |
|---|---|---|---|---|---|
| filename | method | running time in pc | running time in Dc | running time in pc | running time in Dc |
| ibm01 | D | 0.7 | 1.15 | 1.93 | 4.47 |
| | P | 1.94 | 8.56 | 6.48 | 18.29 |
| | I | 19.14 | 22.86 | 22.5 | 30 |
| ibm02 | D | 2.06 | 3.5 | 4.21 | 10.44 |
| | P | 7.17 | 88.97 | 13.6 | 137.99 |
| | I | 59.22 | 77.99 | 100.15 | 112.25 |
| ibm03 | D | 44.06 | 47.57 | 103.7 | 227.04 |
| | P | 102.81 | 392.03 | 531.35 | 724.41 |
| | I | 358.59 | 376.06 | 502.59 | 511.26 |
| ibm04 | D | 28.51 | 49.32 | 56.8 | 152.45 |
| | P | 116.79 | 641.09 | 303.38 | 992.65 |
| | I | 453.92 | 583.15 | 586.82 | 662.91 |

*Table 6 Running times for different methods using the mix-flute router*

From the times shown, the running time by using unimodular technology is larger than the time by using the classical technology. There are two other tables which record the data result from the subnet and congestion routers in Appendix A.

Overall, it seems that the unimodularity technology does not play well in solving the VLSI problem. However, there are some possible improvements, which will be discussed next.

# Chapter 7 Further Testing

## 7.1 Upper bound discussion

If z is unbounded from above, *Pc* will always have an optimal solution. If we bound z from above by UB, the set of solutions will reduce, and the Lagrange dual problem will also be changed. However, theorems 5.1.1-5.1.3 will still hold this. We need further testing to determine what will happen if UB decrease.

After we add the upper bound of penalty z in the Pc model, the new model Pc-u can be written as:

$$
\begin{aligned}
P_c\text{-}u \quad & \min \quad w^T y + m^T \mathbf{z} \\
& s.t.: \quad \mathbf{H}\mathrm{y} = 1 \\
& \qquad \mathrm{A}\mathrm{y} \le \boldsymbol{Cap} + \mathbf{z} \\
& \qquad 0 \le y \le 1 \\
& \qquad 0 \le \mathbf{z} \le UB
\end{aligned}
$$

Suppose UB is an array. The value of UB should be large enough to maintain the feasibility of the model.

The new model *Dc-u* and the model *Pl-u* will be:

$$
\begin{aligned}
Dc\text{-}u\text{:} \quad & \max \quad -\mu^{\mathrm{T}} 1 - v^T cap - \lambda^T 1 - \gamma^{\mathrm{T}}\mathrm{UB} \\
& s.t.: \quad -\mathrm{H}^{\mathrm{T}}\mu - A^T v - \lambda \le w, \\
& \qquad v^{\mathrm{T}} - \gamma^{\mathrm{T}} \le m^T, \\
& \qquad v, \lambda, \gamma \ge 0,
\end{aligned}
$$

And

$$\begin{aligned} P_l\text{-}u: \quad & \min \quad w^T y + m^T z + \alpha^T(\text{Ay-cap-z}) \\ & s.t.: \quad Hy = 1 \\ & \qquad 0 \le y \le 1 \\ & \qquad 0 \le z \le UB \end{aligned}$$

## 7.2 Testing process

At the beginning of the experiment, we solve the LP problem (*Pc*) when the upper bound of z is infinite using the dual simplex method, and record the value of array **z** and the solution y. There are five methods that are used to change the upper bound of penalty **z**.

**Method one**: Find the maximum value in the array z and record it as MaxOfZ, then calculate Ay - *Cap*. Ay- *Cap* is an array that represents the overflow status of each edge.

For $i \in [1,p]$, if (Ay- *Cap*)$_i$ >0, the new UB[i] is set to MaxOfZ. Otherwise, we set the new UB[i] to 0.

Thus if we were to obtain, for example, z = [3.1, 2.3, 1, 0, 6.5, 7] and overflow = [1, 0, 2, 0, 0, 1]; the new UB will be [7, 0, 7, 0, 0, 7].

**Method two**: Find the maximum value in the array z and record it as MaxOfZ, then calculate Ay- *Cap*.

For $i \in [1,p]$, if $(Ay- \mathit{Cap})_i > 0$, the new UB[i] is set to MaxOfZ-1. Otherwise, we set the new UB[i] to 0.

Thus if we were to obtain z = [3.1, 2.3, 1, 0, 6.5, 7] and overflow = [1, 0, 2, 0, 0, 1]; the new UB will be [6, 0, 6, 0, 0, 6].

**Method three**: Let new UB = floor (z),

Thus if we obtain z = [3.1, 2.3, 1, 0, 6.5, 7], the new UB will be [3, 2, 1, 0, 6, 7].

**Method four**: Let new UB = floor (z) -1,

Thus if we obtain z = [3.1, 2.3, 1, 0, 6.5, 7] , the new UB will be [2, 1, 0, 0, 5, 6].

**Method five**: Find the maximum value in the array z and record it as MaxOfZ,

For $i \in [1,p]$, if floor $(z_i)$ = MaxOfZ, the new UB[i] is set to floor( MaxOfZ ) - 1. Otherwise, we set the new UB[i] to floor $(z_i)$.

Thus if we obtain z = [3.1, 2.3, 1, 0, 6.5, 7], the new UB will be [3, 2, 1, 0, 6, 6].

After the new UB has been obtained, the new Dc-u model can be solved using IPM. The *Pl-u* model is also a unimodular optimization model. We can then obtain the integer solution directly.

## 7.3 Testing Result

The table in this chapter are used to show the number of overflow with different upper bounds for penalty z. The entire primal and dual problems were solved using IPM. The testing router is mix-flute. The LP problem used for the testing is a little different from chapter 6, since we did some improvements on the router.

| | Infbounds | Method1 | Method2 | Method3 | Method4 | Method5 |
|---|---|---|---|---|---|---|
| ibm01 | 465 | 519 | 519 | INFEASIBLE | INFEASIBLE | INFEASIBLE |
| ibm02 | 772 | 829 | 829 | INFEASIBLE | INFEASIBLE | INFEASIBLE |
| ibm03 | 300 | 351 | 351 | INFEASIBLE | INFEASIBLE | INFEASIBLE |
| ibm04 | 634 | 702 | 702 | INFEASIBLE | INFEASIBLE | INFEASIBLE |

*Table 7 Amount of overflow with different upper bounds of z*

| | Infbounds | Method1 | Method2 | Method3 | Method4 | Method5 |
|---|---|---|---|---|---|---|
| ibm01 | 1433 | 1532 | 1532 | INFEASIBLE | INFEASIBLE | INFEASIBLE |
| ibm02 | 3655 | 3790 | 3790 | INFEASIBLE | INFEASIBLE | INFEASIBLE |
| ibm03 | 529 | 659 | 659 | INFEASIBLE | INFEASIBLE | INFEASIBLE |
| ibm04 | 2178 | 2317 | 2317 | INFEASIBLE | INFEASIBLE | INFEASIBLE |

*Table 8 Total overflow for the entire circuit with different upper bounds of z*

Table *8* shows the sum of the overflow for the entire circuit with different upper bounds of z.

From Table 7 and Table 8, we can notice that the infinity upper bounds on z play better than other bounds.

| | Infbounds | Method1 | Method2 | Method3 | Method4 | Method5 |
|---|---|---|---|---|---|---|
| ibm01 | 17 | 16 | 16 | INFEASIBLE | INFEASIBLE | INFEASIBLE |
| ibm02 | 22 | 21 | 21 | INFEASIBLE | INFEASIBLE | INFEASIBLE |
| ibm03 | 10 | 10 | 10 | INFEASIBLE | INFEASIBLE | INFEASIBLE |
| ibm04 | 20 | 18 | 18 | INFEASIBLE | INFEASIBLE | INFEASIBLE |

*Table 9 The maximum amount of overflow on the edges with different upper bounds of z*

Table **9** shows the maximum amount of overflow with different upper bounds on z. Here, method one and method two present a better solution than others.

We continue to try find a better upper bound on z, but much progress still needs to be made.

# Chapter 8 Future work

According to the testing results obtained, the unimodular methodology does not show any particular advantage at this moment compared to the classical methodology. The running time and the overflow did increase when we used the unimodular methodology. We continue searching for a way to improve the results. There are several potential avenues that may be explored, but more time is needed for testing and gathering the results.

## 8.1 The weight of the penalty

When solving the model *Pc*, we used an array *m* (weight). In the previous experiments, we set all values in *m* to be the same. However, the weight of **z** could be different, which also changes the obtained solution. If we can find a better way for determining a suitable value of *m*, it may improve the unimodular methodology. One promising idea is to give different weights according to the weights of the edges in the circuit. However, more testing and analysis are needed.

## 8.2 Lagrange multipliers

According to the definition of the primal and dual LP problems, the Lagrange multipliers are part of the solution to the dual problem. When Cplex solves the LP problem, it will also compute the dual solution simultaneously. Thus we do not need to generate the *Dc* model and solve it again. We only need to find the dual solution by solving the *Pc* problem and indicate which part of the dual solution could be used to generate the *Pl* problem. Thus, the running time for both technologies will be the same.

## 8.3 Preprocessing methodology.

Since the size of Pc and Dc is extremely large, if we could apply some preprocessing steps prior to solving it, we may be able to reduce the running time.

Above all, we still need a lot of work on this unimodular methodology. Hopefully, we can improve the method and make it more effective in the future.

# Appendix A

| | | weight 1 cong | | | weight 100 cong | | |
|---|---|---|---|---|---|---|---|
| filename | metho d | number of overflow (Pc)(intege r solution) | number of overflo w (pl) (integer solution) | % of increas e | number of overflow (Pc)(intege r solution) | number of overflo w (pl) (integer solution) | % of increas e |
| ibm01 | D | 747 | 752 | 0.67% | 522 | 691 | 32.38% |
| | P | 747 | 782 | 4.69% | 522 | 667 | 27.78% |
| | I | 744 | 746 | 0.27% | 549 | 549 | 0.00% |
| ibm02 | D | 1110 | 1114 | 0.36% | 838 | 967 | 15.39% |
| | P | 1110 | 1117 | 0.63% | 838 | 968 | 15.51% |
| | I | 1111 | 1113 | 0.18% | 844 | 872 | 3.32% |
| ibm03 | D | 739 | 746 | 0.95% | 290 | 527 | 81.72% |
| | P | 739 | 751 | 1.62% | 288 | 502 | 74.31% |
| | I | 739 | 739 | 0.00% | 314 | 353 | 12.42% |
| ibm04 | D | 1057 | 1061 | 0.38% | 591 | 838 | 41.79% |
| | P | 1057 | 1076 | 1.80% | 594 | 888 | 49.49% |
| | I | 1057 | 1059 | 0.19% | 620 | 648 | 4.52% |

*Table 10 Overflow by using congestion router*

| | | weight 1 subnet | | | weight 100 subnet | | |
|---|---|---|---|---|---|---|---|
| filename | method | number of overflow (Pc)(integer solution) | number of overflow (pl) (integer solution) | % of increase | number of overflow (Pc)(integer solution) | number of overflow (pl) (integer solution) | % of increase |
| ibm01 | D | 743 | 757 | 1.88% | 541 | 673 | 24.40% |
| | P | 743 | 757 | 1.88% | 535 | 677 | 26.54% |
| | I | 745 | 745 | 0.00% | 536 | 544 | 1.49% |
| ibm02 | D | 1073 | 1078 | 0.47% | 855 | 949 | 10.99% |
| | P | 1073 | 1080 | 0.65% | 855 | 940 | 9.94% |
| | I | 1073 | 1073 | 0.00% | 844 | 855 | 1.30% |
| ibm03 | D | 685 | 687 | 0.29% | 294 | 458 | 55.78% |
| | P | 685 | 711 | 3.80% | 294 | 496 | 68.71% |
| | I | 685 | 685 | 0.00% | 302 | 391 | 29.47% |
| ibm04 | D | 1053 | 1058 | 0.47% | 618 | 842 | 36.25% |
| | P | 1053 | 1066 | 1.23% | 616 | 843 | 36.85% |
| | I | 1053 | 1054 | 0.09% | 635 | 681 | 7.24% |

*Table 11 Overflow by using the subnet router*

| Congestion | | weight 1 cong | | | weight 100 cong | | |
|---|---|---|---|---|---|---|---|
| filename | method | obj value in Pc(integer number) | obj value in Pl(integer number) | percentage increase in two model | obj value in Pc(integer number) | obj value in Pl(integer number) | percentage increase in two model |
| ibm01 | D | 39036.81 | 39018.92 | -0.05% | 31408.01 | 30193.64 | -3.87% |
| | P | 39036.81 | 39026.64 | -0.03% | 31355.08 | 30424.73 | -2.97% |
| | I | 39035.56 | 39033.77 | 0.00% | 31595.83 | 31613.27 | 0.06% |
| ibm02 | D | -422530 | -422525 | 0.00% | -459375 | -460644 | 0.28% |
| | P | -422530 | -422526 | 0.00% | -459375 | -462335 | 0.64% |
| | I | -422529 | -422529 | 0.00% | -459389 | -458503 | -0.19% |
| ibm03 | D | 320977.2 | 320982.5 | 0.00% | 277126.8 | 275046.3 | -0.75% |
| | P | 320977.2 | 320993.2 | 0.00% | 277055.9 | 274233.9 | -1.02% |
| | I | 320977.2 | 320977.2 | 0.00% | 277244.7 | 278863.6 | 0.58% |
| ibm04 | D | 35740.08 | 35729.74 | -0.03% | -18208.3 | -21446.7 | 17.79% |
| | P | 35740.08 | 35735.63 | -0.01% | -18733.5 | -22541.6 | 20.33% |
| | I | 35740.98 | 35739.41 | 0.00% | -18443.3 | -17945.2 | -2.70% |

*Table 12 Object value of Pl and Pc by using the congestion router*

| congestion | | weight 1 cong | | weight 100 cong | |
|---|---|---|---|---|---|
| filename | method | running time in pc | running time in Dc | running time in pc | running time in Dc |
| ibm01 | D | 0.58 | 0.96 | 1.58 | 3.77 |
| | P | 1.4 | 2.55 | 5.84 | 26.29 |
| | I | 13.79 | 22.46 | 22.99 | 32.61 |
| ibm02 | D | 0.87 | 2.04 | 2.18 | 4.91 |
| | P | 2.68 | 3.94 | 8.34 | 30.05 |
| | I | 54.28 | 68.98 | 69.13 | 87.94 |
| ibm03 | D | 0.99 | 1.69 | 34.36 | 118.28 |
| | P | 7.32 | 13.14 | 190.35 | 1040.88 |
| | I | 272.88 | 314.21 | 458.92 | 399.45 |
| ibm04 | D | 1.51 | 3.46 | 43.95 | 104.23 |
| | P | 7.9 | 14.8 | 220.65 | 1404.91 |
| | I | 322.92 | 408.67 | 566.19 | 504.95 |

*Table 13 Running times for using the congestion router (in seconds)*

| Subnet | | weight 1 subnet | | weight 100 subnet | |
|---|---|---|---|---|---|
| filename | method | running time in pc | running time in Dc | running time in pc | running time in Dc |
| ibm01 | D | 0.17 | 0.3 | 0.39 | 0.58 |
| | P | 0.39 | 0.46 | 0.9 | 2.72 |
| | I | 4.57 | 18.56 | 7.84 | 20.58 |
| ibm02 | D | 0.56 | 0.85 | 1.01 | 1.47 |
| | P | 1.13 | 1.25 | 2.93 | 8.31 |
| | I | 29.06 | 45.64 | 41.26 | 55.43 |
| ibm03 | D | 0.57 | 0.74 | 11.08 | 19.48 |
| | P | 2.04 | 1.37 | 51.91 | 190.02 |
| | I | 128.15 | 686.98 | 243 | 1089.93 |
| ibm04 | D | 0.8 | 1.41 | 11.56 | 16.5 |
| | P | 2.48 | 2.24 | 41.9 | 238.68 |
| | I | 198.05 | 252.75 | 301.93 | 367.48 |

*Table 14 Running times for using the subnet router (in seconds)*

# Bibliography

[1] L. Behjat, A. Vannelli and W. Rosehart, “Integer linear programming models for global routing,” INFORMS Journal on Computing, vol. 18, No. 2, Spring 2006, pp.137-150.

[2] E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh, “Creating and exploiting flexibility in rectilinear Steiner trees,” IEEE Transactions. Computer-Aided Design, vol. 22, 2003, pp. 605– 615.

[3] Z. Cao, T. Yang, J. Yang, Y. He, L. He, and X. Hong, “DpRouter: A fast and accurate dynamic pattern based global routing algorithm,” in ASP-DAC, 2007, pp. 256–261.

[4] R. C. Carden IV, J. Li, and C.-K. Cheng. "A global router with a theoretical bound on the optimal solution," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 15, no 2, Feb. 1996, pp. 208-216,

[5] C. Chu, “FLUTE: Fast lookup table based wirelength estimation technique,” ICCAD, 2004, pp. 696–701.

[6] E. W. Dijkstra, “A note on two problems in connection with graphs,” Numerische Mathematik, vol. 1, 1959, pp. 269–271.

[ 7] J. Edmonds and R. Giles, “A min-max relation for submodular functions on graphs," Annals Discrete Mathematics, vol 1, 1977, pp.185-204.

[8] C. Fremuth-Paeger, “Goblin: a graph object library for network programming problems,” http://www.math.uni-augsburg.de/~fremuth/goblin.html, [retrieved November 2007] .

[9] S. Golzari and M. R. Meybodi, "A Maze Routing Algorithm Based on Two Dimensional Cellular Automata," in ACRI, 2006, pp. 564-570.

[10] W. L. Hare, M. J. J. Liu, and T. Terlaky, "Efficient preprocessing for VLSI optimization problems," Computational Optimization and Applications, Published online: 15 March 2008.

[11] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," IEEE Transactions on Systems Science and Cybernetics, 1968, pp. 100–107.

[12] R. Kastner, E. Bozorgzadeh, and M. Sarrafzadeh, "An exact algorithm for coupling-free routing," ISPD, 2001, pp. 10–15.

[13] J. B. Kruskal, "On the shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," Proceedings of the American Mathematical Society, vol 7, No. 1 , Feb 1956, pp. 48–50.

[14] E. S. Kuh and M. Marek-Sadowska, "Global Routing," Elsevier Science Publishers B.V., Amsterdam, vol. 1, 1985, pp. 133–168.

[15] T. Lengauer and M. Lungering, "Provably good global routing of integrated circuits," SIAM Journal of Optimization, vol. 11, no. 1, 2000, pp.1–30.

[16] C. H. Papadimitriou and K. Steiglitz. "Combinatorial optimization: algorithms and complexity," Dover Publications Publishers, 1998, page 316.

[17] R. C. Prim, "Shortest connection networks and some generalizations," Bell System Technical Journal, vol. 36, 1957, pp. 1389–1401.

[18] C. Roos, T. Terlaky and J-Ph. Vial, "Theory and Algorithms for Linear Optimization: An Interior Point Approach." Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley, 1997.

[19] N. A. Sherwani, "Algorithms for VLSI physical design automation," Kluwer Academic Publishers, 3rd edition, 1999.

[20] D. M. Warme, P. Winter, and M. Zachariasen, "Advances in Steiner Trees," Kluwer Academic Publishers, 1998.

[21] S. J. Wright, "Primal-Dual Interior-Point Methods," SIAM Publications, 1997.

[22] Y. Ye, "Interior-Point Algorithms: Theory and Analysis," Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley, 1997.