To the Graduate Council:

I am submitting herewith a thesis written by Cheng Chang Yu entitled "Path-Constrained and Incrmental Floorplanning Using Sequence Pairs". I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

D P Mehta

Dr. Dinesh Mehta, Major Professor

We have read this thesis
and recommend its acceptance:

Ben Walsh

Bruce Bowar

Accepted for the Council:

Associate Vice Chancellor
and Dean of the Graduate School

Path-Constrained and Incremental Floorplanning Using Sequence Pairs

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Cheng Yu

May 2000

Dedication

This work is dedicated to my parents, who have given me strength and hope for everyday of my life.

Acknowledgments

I would like to thank my advisor, Dr. Dinesh Mehta, for his guidance and support in the development of this thesis. I would also like to thank Dr. Bruce Whitehead and Dr. Bruce Bomar for serving on my thesis committee.

Abstract

This research focuses on the floorplanning stage of the VLSI physical design cycle. The first part of this research is on path-constrained floorplan. In order to maximize CPU performance and improve clock cycle time, modules on critical paths must be placed in a straight line from an input pin to an output pin. This technique uses the sequence-pairs method. The second part of this research is on incremental floorplan. Given a floorplan, we want to generate a different floorplan that is very similar to the original floorplan after incremented changes in module sizes have been made. Once again, we use sequence-pairs with various cost functions to solve the problem. We have been successful in obtaining a provably correct solution for a limited version of the path-constrained problem. Experimental results demonstrating the efficiency of our methods are also presented.

Contents

1 Introduction 1

2 Literature Survey and Analysis 4

2.1 Slicing Tree and Reverse Polish Expression 5

2.2 Sequence Pair Method 9

2.3 Bounded-Sliceline Grid Method 13

2.4 Placement With Obstacles Based On Sequence Pair 16

2.5 Sequence-Pair Placement Method For Hard/Soft/Pre-placed Modules . 17

2.6 Slicing Floorplans with Boundary Constraints 19

2.7 Slicing Floorplans with Range Constraints 20

2.8 How good are Slicing Floorplans? 21

3 Incremental Floorplanning 23

3.1 Motivation for Incremental Floorplanning 23

3.2 Range constraints without increment 26

3.3 Range constraints with increment 29

3.4 Incremental Floorplanning with all modules 31

4 Path-Constrained Floorplanning **45**

4.1 Motivation 45

4.2 Problem Formulation 46

4.3 Path-Constrained Floorplanning Algorithm 48

4.4 An Illustrative Example 52

4.5 Proof of Correctness 65

4.6 Experimental Results 67

5 Conclusion **71**

Bibliography **73**

Vita **75**

List of Tables

3.1 Range Constraint using Cost Functions. 27

3.2 More Rigid Range Constraint using Cost Functions. 28

3.3 More Relaxed Range Constraint using Cost Functions. 28

3.4 More Rigid Range Constraint using Distance Cost Functions. 29

3.5 Range Constraint using Incremental Cost Functions. 30

3.6 More Rigid Range Constraint using Incremental Cost Functions. 30

3.7 Range Constraint using Cost Function With Distance Factor only. . . . 31

3.8 Incremental Floorplanning Using Cost Functions. 34

3.9 Incremental Floorplanning Using Distance Functions. 35

3.10 Lower Bound on Area Using Cost Functions. 36

3.11 Incremental Floorplanning Using Same Sequence Pair. 36

3.12 Incremental Floorplanning With Scenario 1. 37

3.13 Incremental Floorplanning With Scenario 2. 37

3.14 Incremental Floorplanning With Scenario 3. 38

3.15 Incremental Floorplanning With Scenario 4. 38

3.16 Incremental Floorplanning Using Cost Function With Area and Distance Factor 42

3.17 Incremental Floorplanning Using Input ami49 with scenario 1. 42

3.18 Incremental Floorplanning Using Input ami49 with scenario 2. 43

3.19 Incremental Floorplanning Using Input ami49 with scenario 3. 43

3.20 Incremental Floorplanning Using Input ami49 with scenario 4. 44

3.21 Incremental Floorplanning Using Different Orientation 44

4.1 Module Specification for Path-Constrained Floorplanning Example . . . 53

4.2 Path Constraint Satisfied With Different Path Input 1 68

4.3 Path Constraints Satisfied With Different Path Input 2 69

List of Figures

2 1 Slicing Floorplan Example 5

2.2 Non Slicing Floorplan Example 6

2.3 Slicing Tree Example 7

2 4 Skewed Slicing Tree Example 7

2.5 Example of a packing on an oblique grid 10

2.6 Sequence Pair Packing Example 12

2.7 Horizontal BSG Graphs Example 15

2.8 Vertical BSG Graphs Example 15

3.1 Distance Factor Example 27

3.2 Manhattan distance and Direct Distance Example 35

3.3 Resulting Floorplan In First Design Iteration 39

3.4 Resulting Floorplan In Second Design Iteration With Scenario 0 39

3.5 Resulting Floorplan In Second Design Iteration With Scenario 1 40

3.6 Resulting Floorplan In Second Design Iteration With Scenario 2 40

3.7 Resulting Floorplan In Second Design Iteration With Scenario 3 40

3.8 Resulting Floorplan In Second Design Iteration With Scenario 4 41

4.1 An Example of A Critical Path 47

4.2 Horizontal and Vertical Bands 49

4 3 Propped Edge Example 50

4.4 An Path-Constrained Floorplanning Example: Step 1 54

4.5 An Path-Constrained Floorplanning Example: Step 2-1 55

4.6 An Path-Constrained Floorplanning Example: Step 2-2 56

4.7 An Path-Constrained Floorplanning Example: Step 3 57

4.8 An Path-Constrained Floorplanning Example: Step 4 58

4.9 An Path-Constrained Floorplanning Example: Step 5 59

4.10 An Path-Constrained Floorplanning Example: Step 6-1 60

4.11 An Path-Constrained Floorplanning Example: Step 6-2 61

4.12 An Path-Constrained Floorplanning Example: Step 7 62

4.13 An Path-Constrained Floorplanning Example: Step 8 63

4.14 An Path-Constrained Floorplanning Example: Step 9 64

4.15 An Path-Constrained Floorplanning Example: Step 10 65

Chapter 1

Introduction

Chip performance, as measured by clock frequency has significant commercial value in the VLSI industry. The clock frequency is inversely proportional to the clock cycle time, which is determined by the time taken by a signal to travel from source to destination on a certain number of critical paths. The total path delay is determined by the gate delay and the interconnection delay. Advances in semiconductor technology have reduced gate delay to the point that gate delay and interconnection delay are comparable. Since interconnection delay is proportional to interconnection length, it is necessary to place modules in a straight line from the input pin to the output pin of these critical paths.

The first objective of this thesis is to develop software that will successfully place modules onto critical paths and at the same time minimize the area and wire length of the whole chip.

The second objective of this thesis is to use incremental floorplanning to improve

the design productivity in the physical design environment.

The typical physical design cycle consists of partitioning, floorplanning and place-and-route steps. In floorplanning, a floorplan architect uses area *estimates* to allocate area for each module because the actual area is not known until after place-and-route. Because of the difference between the area *estimate* and the exact area requirement for the floorplan, the floorplan has to go through another design iteration.

There is no guarantee that the resulting modules from the next design iteration will have similarity in shape and location to those obtained in the previous iteration. The area and timing requirements of a module are sensitive to the relative locations of the modules in the floorplan. Thus, if shapes and locations of modules fluctuate from one iteration to the next, so does the area of the laid out module. This results in a process which is uncertain to converge. Also, a drastic change in the layout resulting from changes to module shapes and locations would result in a loss of time as the designer would have to refamiliarize with the new layout.

Our research is to design an incremental floorplanner that can be used by a floorplan architect to perform tasks that would otherwise have been performed manually. Our approach is to use new cost functions in the simulated annealing process to ensure that the shape, orientation, and location of the new module are similar to those of the old module to facilitate convergence

Thus, the contribution of this thesis is to present:

1. A Path-Constrained Algorithm that has successfully placed modules onto hori-

zontal/vertical path. A theoretical proof is provided to prove the correctness of the algorithm.

2. Simulated Annealing based solution to generate a new floorplan with modules at similar locations to those in the previous design iteration.

3. We have implemented the software in C under UNIX that handles the path-constrained and incremental floorplanning problem.

Chapter 2 provides an extended review of previous scholarly studies that have focused on floorplanning in VLSI design. Chapter 3 describes incremental floorplanning. Chapter 4 discusses the path-constrained floorplanning problem. Finally, Chapter 5 provides concluding remarks about this work and outlines possible future work.

Chapter 2

Literature Survey and Analysis

Floorplanning is one of the stages of VLSI physical design. It tries to place a number of modules onto a two-dimensional plane such that no two modules overlap with each other. The goal is to optimize certain objectives such as area or wirelength or a combination of both to produce an efficient circuit During the floorplanning process, modules can be rotated around to change their orientation. The final area is the area of the smallest enclosing rectangle that contains all of the modules. The most popular method involves simulated annealing in which an iterative computation is carried out to search the solution space to find an optimal solution.

There has been much research on floorplanning in the last two decades. In the following sections, we will describe three simulated-annealing-based techniques used for floorplanning. Also included are techniques for solving constrained versions of the floorplanning problem.

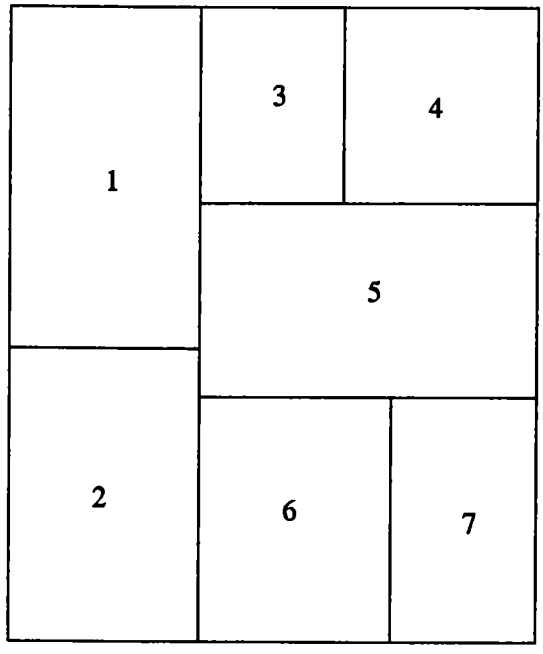

Figure 2.1: Slicing Floorplan Example

2.1 Slicing Tree and Reverse Polish Expression

There are two kinds of floorplans: slicing and non-slicing. A slicing floorplan is one that can be recursively subdivided by horizontal and vertical cutlines. Figure 2.1 is a slicing floorplan that contains 7 modules.

A non-slicing floorplan is one that is not slicing. There are known floorplans that are not slicing, for example, the wheels are the smallest non-slicing floorplan. Figure 2.2 is a non-slicing floorplan that contains 7 modules.

A slicing tree can be used to represent a slicing floorplan. A slicing tree is a binary tree with n leaves and $n-1$ internal nodes, where each internal node represents a horizontal cut or a vertical cut line, and each leaf represents a module [7].

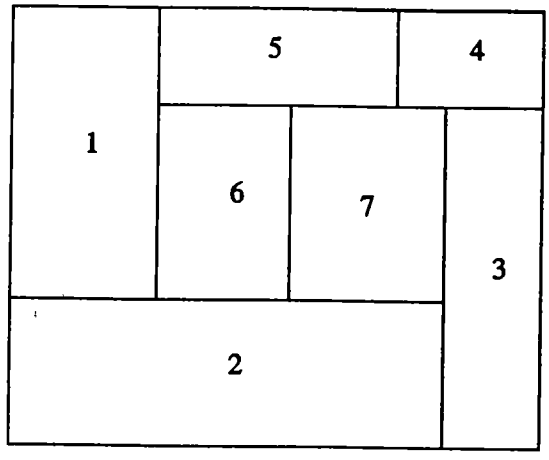

Figure 2.2: Non Slicing Floorplan Example

Figure 2.3 is a slicing tree corresponding to the slicing floorplan represented in Figure 2.1.

A skewed slicing tree is one in which no node and its right child are the same. Figure 2.4 is a skewed slicing tree corresponding to the slicing floorplan in Figure 2.1.

A Polish expression can be obtained by performing a postorder traversal of a slicing tree. A postorder traversal of a tree is computed by recursively traversing the left subtree, the right subtree, and then the root. A Polish expression is defined as follows: an expression $E = e_1 e_2 \dots e_{2n-1}$, where each e_i belongs to $\{1, 2, \dots n, H, V\}$, $1 \leq i \leq 2n-1$, is a Polish expression of length $2n-1$ if and only if:

1. Every operand j, $1 \leq j \leq n$, appears exactly once in the expression; and

2. The expression E has the balloting property, i.e., for every subexpression $E_i = e_1 \dots e_i$, $1 \leq i \leq 2n-1$, the number of operands is greater than the number of operators.

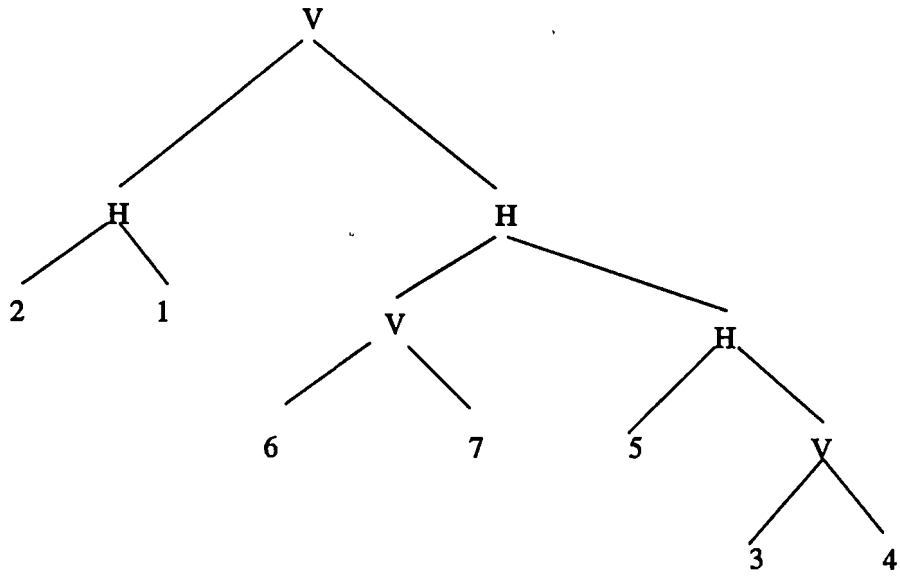

Figure 2.3: Slicing Tree Example

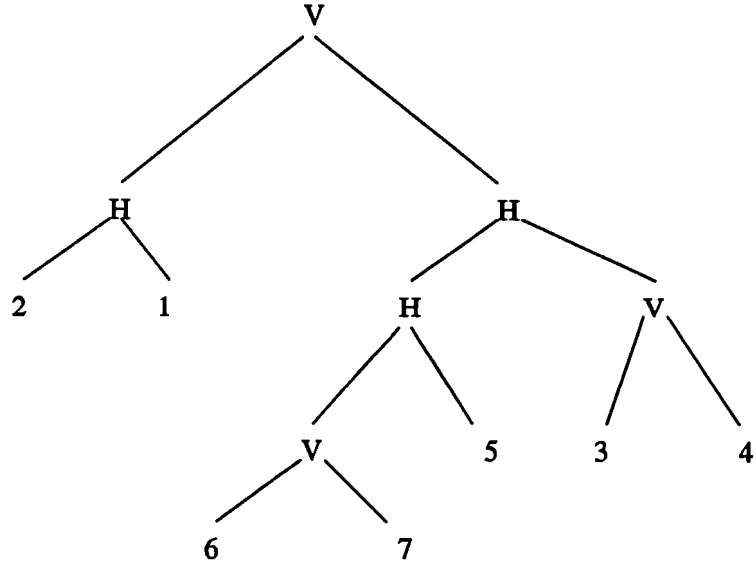

Figure 2.4: Skewed Slicing Tree Example

The operators H and V have the following meanings: ijH means rectangle j is on the top of rectangle i. ijV means rectangle i is to the left of rectangle j.

For example, the Polish expression corresponding to Figure 2.3 is $21H67V534VHHV$. observe that module 1 is on top of module 2 and module 6 is to the left of module 7. Also, notice that there is a pair of consecutive H's in this expression.

A Polish expression E $= e_1 e_2 \ldots e_{2n-1}$ is called normalized if and only if E has no consecutive H's or V's. For example, the Polish expression corresponding to Figure 2.4 is $21H67V5H34VHV$. Notice that there are no consecutive H's or V's in this Polish expression.

Wong and Liu showed that:

1. There is a one-to-one correspondence between the set of skewed slicing trees with n leaves and the set of normalized Polish expressions of length $2n-1$.

2. There is a one-to-one correspondence between the set of normalized Polish expressions of length $2n-1$ and the set of slicing structures with n basic rectangles.

Floorplan solutions are represented by normalized Polish expressions. There are three types of moves associated with a normalized Polish expression:

1. swap two adjacent operands

2. complement some chain of operators of non-zero length

3. swap an adjacent operand-operator pair

The first two moves always produce a normalized Polish expression from a normalized Polish expression However, the third move may result in a non-normalized Polish expression. Therefore, whenever to make the third move, the resulting expression after move must be checked to assure it is a normalized Polish expression.

A cost function is evaluated for each intermediate solution. The cost function normally is a linear combination of area and interconnection length. For example, $Cost = \alpha * Area + (1-\alpha) * Wirelength$ where *Area* is the area of the smallest rectangle enveloping the n basic rectangles, and *Wirelength* is a measure of the overall wiring length using half-perimeter method.

2.2 Sequence Pair Method

The sequence pair is another abstract representation of a floorplan. The sequence pair has two components: one positive module name sequence and one negative module name sequence. A sequence pair is a set of two strings such that each module appears once in each string. The first sequence is denoted by Γ^+ and the second by Γ^-. For example, (Γ^+, Γ^-) = (2146357, 5613742) is a sequence pair denoting a floorplan with seven modules.

A floorplan is constructed from a sequence pair as follows: Draw a grid with horizontal grid lines labeled with module names in Γ^+ from the top and vertical lines labeled with module names in Γ^- from the left. Each module is put on the grid with its center at the intersection of grid lines with that module name. Then , rotate the resultant grid

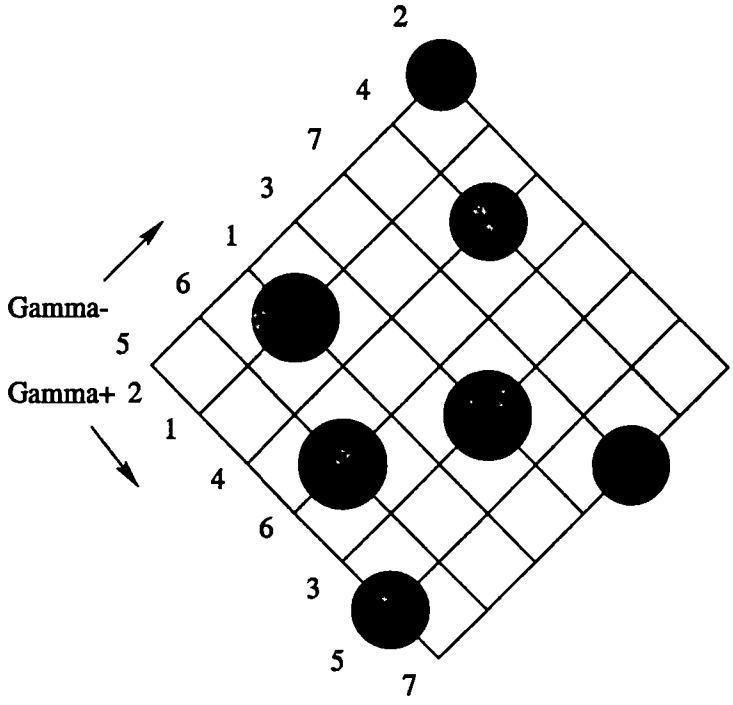

Figure 2.5: Example of a packing on an oblique grid

by 45 degrees counter clockwise to obtain an oblique grid. Figure 2.5 shows an oblique grid for $(\Gamma^+, \Gamma^-) = (2146357, 5613742)$.

By observing the oblique grid, the spatial relationship of two modules from those two sets has been revealed.

If a module A appears before module B in Γ^+, then it implies module A is to the left of module B or module A is above module B. If a module A appears before module B in Γ^-, then it implies module A is below module B or module A is to the left of module B. If a module A appears after module B in Γ^+, then it implies module A is to the right of module B or module A is below module B. If a module A appears after module B in Γ^-, then it implies module A is to the right of module B or module A is

above module B.

Two modules a and b are related in four ways. a is either before or after b in each of Γ^+ and Γ^-. Therefore four sets have been defined as follows:

1. Left_Of(a) = $\{b \mid b$ is before a in both Γ^+ and $\Gamma^-\}$,

2. Right_Of(a) = $\{b \mid b$ is after a in both Γ^+ and $\Gamma^-\}$,

3. Above_Of(a) = $\{b \mid b$ is before a in Γ^+ and after a in $\Gamma^-\}$,

4. Below_Of(a) = $\{b \mid b$ is after a in Γ^+ and before a in $\Gamma^-\}$.

By taking the intersection of the oblique grid relationships, if a module b$\in$ $Left_Of$(a), then b is left of a in two-dimensional space. If a module b$\in$ $Right_Of$(a), then b is right of a in two-dimensional space. If a module b$\in$ $Above_Of$(a), then b is above a in two-dimensional space. If a module b$\in$ $Below_Of$(a), then b is below a in two-dimensional space.

For example, module 3$\in$ $Right_Of(1)$ in the sequence pair and module 3 indeed appears exactly right of module 1 in Figure 2.6.

From a given sequence pair a rectangle packing can be obtained by applying longest path algorithm. This consists of constructing two graphs: a horizontal-constraint graph and a vertical-constraint graph. The horizontal graph consists of vertex set and edge set. The vertex set contains a source s, a sink t, and n vertices labeled with module names. The edge set contains edges from source to every module and edges from every module to sink. Additionally if module a is left of module b then there is an edge from a

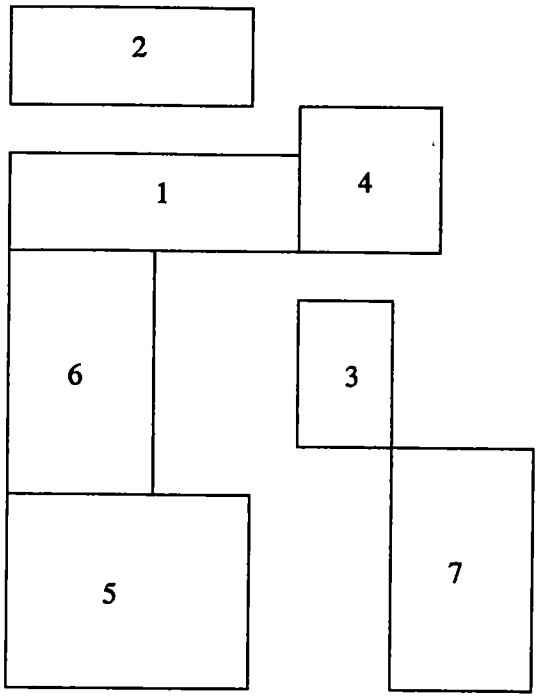

Figure 2.6: Sequence Pair Packing Example

to b in the graph. Every vertex has a weight that is its width. The source and sink have weight 0. The X-coordinate of a module is the longest path length from source to that module in the horizontal-constraint graph. Similarly, the vertical-constraint graph is constructed except that the height of a module is used as the weight and an edge exists between two modules if one is above the other. The Y-coordinate of a module is the longest path length from source to that module in the vertical-constraint graph. The width and length of the chip can be obtained from the longest path length between the source and the sink in both graphs. There are moves that can be used to alter sequence pair in the simulated annealing:

1. swap two modules in $\Gamma+$
2. swap two modules in both $\Gamma+$ and $\Gamma-$
3. change orientation of a module

By using sequence pair method, 500 rectangles were packed successfully and efficiently [6].

2.3 Bounded-Sliceline Grid Method

The bounded-sliceline grid method consists of special orthogonal segments that divide the plane into "bsg-rooms". The orthogonal line segments are called BSG-segs, and they are defined in the following ways:

1. $H_{i,j} = \{(x,y)|i-1 < x < i+1, y = j, i+j : odd\}$

2. $V_{i,j} = \{(x,y)|x = i, j-1 < y < j+1, i+j : even\}$

The BSG is a system of BSG-segs. The rectangular space surrounded by adjacent pairs of BSG-segs is called a BSG-room. The BSG is theoretically infinite grid and realistically it needs to be bound to a finite space. A BSG table of size $p \times q$ has origin at (0, 0) and the above-right corner at (p, q) [8].

There are horizontal and vertical BSG graphs that are associated with BSG rooms. The horizontal BSG graph contains vertex set and edge set. The vertex set $V_h = \{source, sink, H_{i,j}\}$. The edge set contains three parts. The source is connected to the center of the bottom segs. The sink is connected to the center of the top segs. The center of one horizontal segment is connected to another horizontal segment if two horizontal segments are adjacent. The vertical BSG graph is defined in a similar way. The Figure 2.7 and Figure 2.8 show horizontal and vertical BSG graphs. Each room is crossed exactly once in each of the two graphs.

A BSG-assignment is a mapping of modules into the rooms of BSG table such that no two modules occupy the same room.

Each edge in the BSG-graph has a weight. If an edge in a horizontal BSG graph crosses a room that contains a module, it has a weight equal to the height of that module. Similarly, if an edge in a vertical BSG graph crosses a room that contains a module, it has a weight equal to the width of that module. The longest-path computation determines the height and width of the chip and the position of each module.

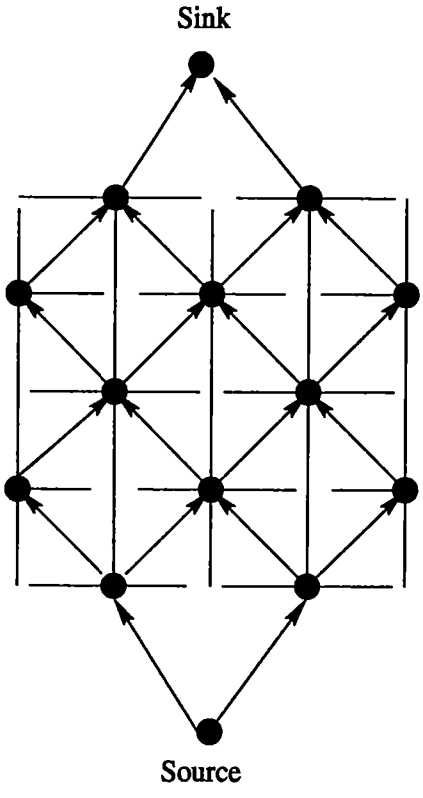

Figure 2.7: Horizontal BSG Graphs Example

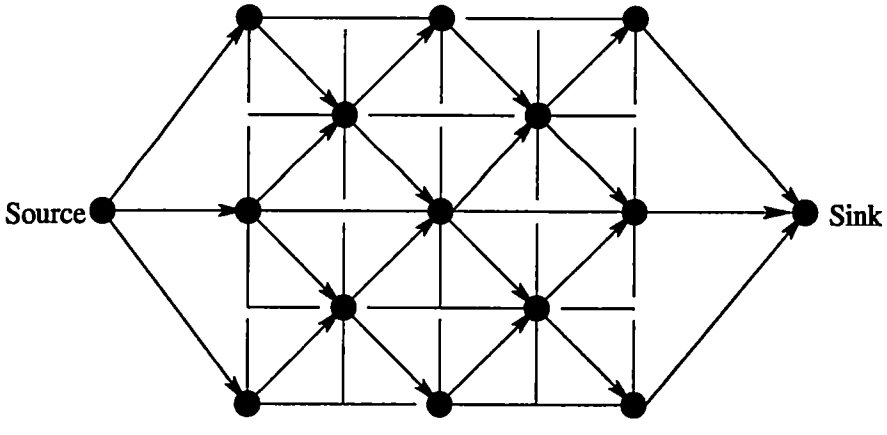

Figure 2.8: Vertical BSG Graphs Example

The BSG is also a simulated-annealing-based approach and a move is defined as swapping the content of two rooms, at least one of which should not be empty. Details may be found in [8].

2.4 Placement With Obstacles Based On Sequence Pair

In some floorplans, a number of modules have been preplaced and all other modules will need to be placed in the available area. An arbitrary sequence pair will not result in the preplaced modules being positioned correctly. So a procedure called "adaptation" has been used to transform an inconsistent sequence pair to a feasible sequence pair. The first step of the algorithm is to guarantee that the coordinates of the preplaced module from the sequence pair will not be smaller than its predefined coordinates. This has been achieved by creating an additional edge directly from the source to the preplaced module in the constraint graphs. The adaptation procedure also introduces the notion of dominance. The use of dominance relation gives us a topologically sorted sequence pair. For any two preplaced modules a and b, if a dominates b, then a is before b in Γ^-. This is used to transform an invalid sequence pair to a valid sequence pair by moving modules along Γ^+ while maintain Γ^- as constant [5].

2.5 Sequence-Pair Placement Method For Hard/Soft/Preplaced Modules

The convex optimization problem is solved to optimize the aspect ratio of the soft modules. The method can be used in two ways: a) it is used as the exact placement method. b) it is used as the post process in an approximate placement method.

There are three types of modules that form the input pool to the floorplan problem:

1. *Hard Module*: The width and height of module are specified.

2. *Soft Module*: The width and height of the module are unspecified. The area of the module is specified. The ratio of the height to the width has to be inside an range.

3. *Preplaced Module*: The width, height and position of the module are specified.

The problem is to get a feasible packing of all those kinds of modules such that all the preplaced modules are placed at specified positions and the aspect ratio and the area of soft module meet the constraints.

The soft-module-problem is transformed into a convex optimization problem with convex constraints by an exponential variable transformation. The implementation of Vaidya's algorithm is employed to solve the Convex-Problem in a series of steps. First a convex polytope has to be constructed using the aspect ratio constraints. Second the gradient of the objective function is calculated along the longest paths. Then the separating hyper plane is constructed and the increasing part is cut off and the remaining

part replaces the current polytope. The algorithm repeats until the polytope becomes very small. The feasibility of a point in the resultant small polytope is tested. If it is feasible the point is the output of the solution. Otherwise, there is no solution to the problem [4].

The problem is proved to be solved in polynomial time depending on the number of modules, the wideness of the range of the aspect ratio constraint, and the accuracy used in the stopping criterion.

There are two methods that are implemented to solve the problem: the exact method and the approximate method. The exact method uses the Vaidya's algorithm in a simulated annealing. In some iterations the sequence pair may become infeasible when there are some preplaced modules. The adaptation procedure is extended to consider the soft modules also. Because of the large CPU time for inputs the approximate method is also used. There are four methods for approximate placement. They are called "square", "discrete", "sq+post", "dis+post". The "square" method shapes each module into a square hard module. The "discrete" method starts with square modules and increases or decreases aspect ratio of the module by 10 percent. The "sq+post" method is "square" followed by the numerical optimization. "dis+post" method is "discrete" followed by the numerical optimization.

2.6 Slicing Floorplans with Boundary Constraints

It is sometimes useful in VLSI design to place some modules on one of four sides: on the left, on the right, at the bottom, or at the top of the final floorplan. These are called boundary constraints. A slicing floorplan algorithm can be used to handle such constraints. The normalized Polish expression is used to define a state in the simulated annealing method. In the simulated annealing process, the normalized Polish expression is checked to see whether the boundary constraints are satisfied. This is done by scanning the Polish expression using check-boundary-constraint algorithm. If boundary constraints have been violated, the algorithm attempts to remove the violation.

The check-boundary-constraint algorithm uses a lemma that states that a module is on the right boundary of the floorplan if and only if it is not in the left subtree of any internal node in the slicing tree labeled V. Similar assertion are made for the other three boundaries [3].

The Polish expression is scanned from right to left and the algorithm is implemented using a stack. The stack element p has four bits: p.left, p.right, p.top, and p.bottom. Each time if it an operator it is pushed onto the stack and set p.left or p.bottom to 1 depending on whether it is a V operator or H operator. When the right subtree is done the left subtree is traversed to set up the stack element bits. After both left and right subtree of a stack element p have been traversed that stack element is poped off the stack. On completion of the algorithm, each module will have four bits to indicate if there exists any other module to its left, right, top, or bottom.

The fixing of a Polish expression is done by shuffling modules. In general, if a module X does not meet the boundary constraint it will be shuffled with another module Y which is closest to X in the Polish expression such that Y's position satisfies X's boundary constraint. However, it is possible that some constraints are still violated after all the possible shufflings. So a boundary constraint term is included in the cost function to penalize the remaining violating constraint.

2.7 Slicing Floorplans with Range Constraints

The range constraint requires a module to be placed within a given rectangular region in the final floorplan. A pre-placed constraint is just a special range constraint where the rectangular region is of the same dimensions as the module itself.

A range constraint is defined as follows: given a hard module A and a rectangular region $R = \{(x,y)|x1 \leq x \leq x2, y1 \leq y \leq y2\}$, A must be placed inside R in the final floorplan.

The Wong-Liu algorithm handles soft module by representing the shape of each module with a shape curve. A shape curve is a piecewise linear decreasing curve which represents the tradeoff between the height and the width of a module. Starting from the leaves of the slicing tree work upward to the root, computing the shape curve at each internal node. Eventually the shape curve at the root represents all the possible shapes of the final floorplan.

An important observation is that when two modules are put together and at least

one of them has range constraint the combined supermodule will also have range constraint. So the idea here is the shape curve is computed at each internal node from the shape curve of its two children, taking into account the placement constraint information. For example, assume only module A has range constraint and module B has no placement constraint. Let X be the combined supermodule, i.e. $X = ABH$, then $h(X) = h(A) + h(B)$, $w(X) = max\{w(A), w(B)\}$, $right(X) = max\{w(X), right(A)\}$, $left(X) = left(A)$, $top(X) = top(A) + h(B)$ and $bottom(X) = bottom(A)$ [2].

Equations are derived for all cases and incorporated into the shape curve computation for the slicing tree A penalty term is introduced in the cost function to penalize the violation of range constraints.

2.8 How good are Slicing Floorplans?

Since the set of slicing floorplans is a proper subset of the set of all valid floorplans, it is possible to argue that slicing floorplan formulations ignore high quality non-slicing floorplan. So it is important to have mathematical analysis of slicing floorplan to assure their performance.

Height(R), width(R), and area(R) denote the height, the width, and the area of module R respectively. The aspect ratio is defined as $height(R)/width(R)$. A soft module has a shape flexibility which specifies the range of its aspect ratio. In this floorplan design problem, given n soft modules of area A_i, for $i = 1, 2, \ldots, n$ and a shape flexibility r, an upper bound on the area of the optimal floorplan needs to be

established. The slicing floorplan F is constructed such that every module satisfies the aspect ratio constraint and the area is as small as possible. A_{total} denotes $\sum_{i=1}^{n} A_i$ and A_{max} denotes $max_{1 \leq i \leq n}\{A_i\}$.

Given a set of soft modules of total area A_{total}, maximum area A_{max} and shape flexibility $r \geq 2$, there exists a slicing floorplan F of these modules such that

$area(F) \leq min\{(1 + 1/\lfloor\sqrt{r}\rfloor, 5/4, (1 + \alpha)\}A_{total}$

where $\alpha = \sqrt{2A_{max}/rA_{total}}$

In the previous analysis, the shape flexibility r is required to be at least two. This is justified by the assumption that each rectangle has a considerable amount of flexibility in its shape. The packing technique is slightly modified to accommodate the case when $r = 2 - \epsilon$ where ϵ is a small positive number. So given a set of soft modules of total area A_{total}, maximum area A_{max} and shape flexibility $r = 2 - \epsilon$ where ϵ is a small positive number, there exists a slicing floorplan F such that

$area(F) \leq min\{5/4(1 + \epsilon/2), (1 + \alpha + \epsilon/2)\}A_{total}$

where $\alpha = \sqrt{2A_{max}/rA_{total}}$

Experimental results show that slicing floorplan can perform better than what is proven mathematically. Experiments have been performed to obtain 2.2% of dead space on an input problem with 100 soft retangles of shape flexibility two [1].

Chapter 3

Incremental Floorplanning

In this chapter, we present the incremental floorplanning research problem. Incremental floorplanning is a problem in which a floorplan in a particular design iteration has to be made structurally similar to the floorplan in previous design iteration to improve design productivity.

3.1 Motivation for Incremental Floorplanning

The incremental floorplanning strategy differs from traditional floorplanners in that it seeks to improve design productivity in an aggressive physical design environment, where a significant investment of design time is made in order to meet tight goals for area, performance, power, etc.

Design productivity, which may be defined as the inverse of design time, is adversely affected by (1) number of design iterations and (2) time per iteration. Our incremental

floorplanning strategy was designed with a view to reducing both of these quantities. In order to motivate the discussion, we consider a simplified scenario consisting of two steps of the traditional physical design cycle: floorplanning and place-and-route.

Design Scenario. A floorplanner dissects the chip area into modules. These modules correspond to sub-circuits which were obtained by a previous partitioning step. The floorplanning algorithms use area *estimates* for these sub-circuits to allocate area for each module. Estimates are needed because the exact area requirements of a subcircuit are unknown until after the subsequent place-and-route step. However, the actual area of the laid-out circuit typically differs from the estimate used by the floorplanner by a small amount. This invalidates the floorplan and necessitates having to go to another iteration of the design cycle.

Popular floorplanning algorithms use techniques such as simulated annealing and genetic algorithms. When these techniques are used in a given iteration of the design cycle, there is no assurance that the resulting modules have any resemblance in shape and location to those obtained in the previous iteration. The area and timing requirements of a module are sensitive to its shape and its pin locations and therefore to the relative locations of the modules in the floorplan. The area of a module consists of the cell area and the routing area The cell area remains constant, but the routing area depends on the module shape and pin positions. Pin positions clearly impact the routing. The module shape has an impact because it is used to determine the number of standard cell rows needed. This, in turn, affects the routing area. Thus, if shapes

and locations of modules fluctuate from one iteration to the next, so does the area of the laid out module. This results in a process which is uncertain to converge. Also, aggressive design environments typically require significant human input to the design process. A layout designer may spend hours becoming familiar with a proposed layout. A drastic change in the layout resulting from changes to module shapes and locations would result in much time being lost as the designer would have to refamiliarize him/her self with the new layout.

In practice, this problem is addressed by manually making small changes to the floorplan to accommodate the actual area requirements with minimal change to the shapes and locations of the modules.

Our strategy is to design an incremental floorplanner that can be used by a floorplan architect to perform tasks that would otherwise have been performed manually. Specifically, the input to the incremental floorplanner consists of

1. The floorplan computed in the previous design iteration based on area estimates.
2. The actual area required for each sub-circuit (obtained by running the place-and-route software on the modules computed by the floorplan).

The output of the incremental floorplanner is a new floorplan which uses the actual areas computed by the place and route software. The floorplanner attempts to alter the shapes of individual modules to meet the actual area requirement while simultaneously ensuring that the shape, dimensions, and location of the new module are similar to those of the old module to facilitate convergence.

The basic approach in my research is to use various cost functions in the context of simulated annealing to control the degree of similarity without too much sacrifice for chip area and wirelength. The following sections describe various experiments.

3.2 Range constraints without increment

First we test the feasibility of using cost functions to solve this problem. Our input consists of five modules and five bounding rectangles. The five modules are to be placed inside the corresponding bounding rectangles. We define the distance factor d to be the sum of the Manhattan distances from the center of each module to the nearest edge of its rectangle. If the distance factor is zero, the module has satisfied its range constraint. Figure 3.1 illustrates distance factor. In Figure 3.1, we have a bounding rectangle B and a module A. The dotted line is the Manhattan distance from the center of module A to its nearest edge of B.

The distance factor is included as a term of the cost function. It can also be regarded as a penalty term appended to our cost function.

Table 3.1 shows the results of using various cost functions with different weights associated with the distance factor in the simulated annealing process. F denotes cost function. *Cost* denotes final cost of the floorplan. *Area* denotes final area of the floorplan. P denotes the fraction of modules that have satisfied the range constraint. Q denotes the sum of Manhattan distance from the center of each module to its nearest edge of corresponding bounding rectangle.

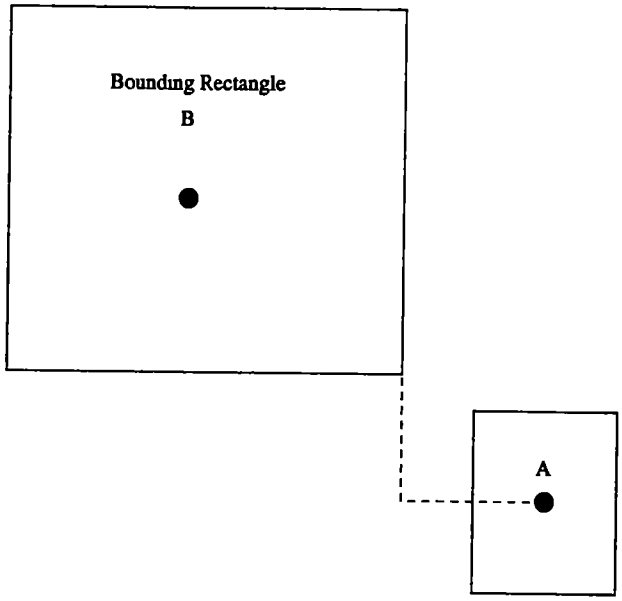

Figure 3.1: Distance Factor Example

Table 3.1: Range Constraint using Cost Functions.

Experiment	F	Cost	Area	P	Q
0	a + 300 * d	1347879	1316679	3/5	104
1	a + 500 * d	1294316	1243816	3/5	101
2	a + 600 * d	1330735	1313935	4/5	28
3	a + 700 * d	1300901	1300901	5/5	0
4	a + 800 * d	1306732	1306732	5/5	0
5	a + 1000* d	1286152	1286152	5/5	0

Table 3.2: More Rigid Range Constraint using Cost Functions.

Experiment	F	Cost	Area	P	Q
0	a + 300 * d	1277724	1277724	5/5	0
1	a + 500 * d	1335414	1292914	3/5	85
2	a + 600 * d	1370024	1331624	3/5	64
3	a + 700 * d	1327536	1277136	2/5	72
4	a + 800 * d	1340552	1325352	4/5	19
5	a + 1000* d	1318952	1295952	3/5	23

Table 3.3: More Relaxed Range Constraint using Cost Functions

Experiment	F	Cost	Area	P	Q
0	a + 300 * d	1260531	1245531	3/5	50
1	a + 500 * d	1324323	1324323	5/5	0
2	a + 600 * d	1339410	1327410	3/5	20
3	a + 700 * d	1263073	1263073	5/5	0
4	a + 800 * d	1278165	1278165	5/5	0
5	a + 1000* d	1381220	1312220	3/5	69

Next, we reduced the dimension of each enclosing rectangle by a factor of 2. Table 3.2 shows the result. We used the same cost functions and observed that applying more rigid constraints increases Q in most cases.

Next, we increased the dimensions of each enclosing rectangle to see if there is any impact on the modules which need to be placed within range constraint. Table 3.3 shows the result. We still used the same cost functions and observe that Q decreases in most cases.

In the last experiment we used the distance factor to be the sole term in the cost function and we reduce the dimension of each enclosing rectangle by one third to one fourth. It turns out that all the modules are within their corresponding rectangles.

Table 3.4: More Rigid Range Constraint using Distance Cost Functions.

Experiment	F	Cost	Area	P	Q
0	distance_sum_edge	0	2697303	5/5	0

Table 3.4 shows the result.

From above results we can conclude that by putting the distance factor into the cost function, the range constraints can be more easily satisfied.

3.3 Range constraints with increment

We use the same 5 modules and bounding rectangles from the previous section. Each module, additionally, has two variables associated with it that represent the desired increase in its width and height.

We simulate an iterative design environment as follows: first, a regular floorplanner, which uses the traditional cost function consisting of area and wirelength, is run on the input. The output of this run includes the location, shape, and orientation of each module. This step represents the first iteration of the design cycle.

The second iteration uses our incremental floorplanner. The input to the incremental floorplanner consists of the output of the first iteration and the desired increments in the width and height of each module. The incremental floorplanner differs from the traditional floorplanner in that (1). The cost function includes a distance term (2). The orientation of each module is not permitted to change

A number of cost functions with the distance factor are presented to show how

Table 3.5: Range Constraint using Incremental Cost Functions.

Experiment	F	Cost	Area	P	Q
0	a + 300 * d	1457332	1371832	2/5	285
1	a + 500 * d	1650651	1438151	3/5	425
2	a + 600 * d	1626708	1447308	4/5	299
3	a + 700 * d	1524635	1395135	4/5	185
4	a + 800 * d	1322864	1321264	4/5	2
5	a + 1000* d	1548560	1546560	4/5	2

Table 3.6: More Rigid Range Constraint using Incremental Cost Functions.

Experiment	F	Cost	Area	P	Q
0	a + 300 * d	1504400	1324400	3/5	600
1	a + 500 * d	1423588	1397088	4/5	53
2	a + 600 * d	1554984	1458984	3/5	160
3	a + 700 * d	1545056	1409256	4/5	194
4	a + 800 * d	1479290	1476090	4/5	4
5	a + 1000* d	1413720	1413720	5/5	0

various cost functions affect the outcome. Table 3 5 shows the result.

Next, we reduced the dimension of each bounding rectangle by a factor of 2. Table 3.6 shows the result.

Next, we further reduced the dimension of each bounding rectangle and we used the distance factor as the only term in the cost function. Table 3.7 shows the result. *Size* denotes the dimensions of the bounding rectangles. (In this case, the bounding rectangles are made square.)

The best result is obtained by making dimensions of bounding rectangle to zero and by using a cost function with only the distance factor term in simulated annealing process. The reason is that the smaller the bounding rectangles, the harder it is to

Table 3.7: Range Constraint using Cost Function With Distance Factor only.

Experiment	Size	F	Area	d
0	0	d	3444336	57
1	5	d	3666240	34
2	15	d	3359048	12
3	25	d	2977864	0
4	35	d	3084896	0
5	50	d	4310950	0

place the modules within them. Thus the penalty is larger for the smaller rectangles and the simulated annealing process is more sensitive to the distance factor. When the cost function consists of only the distance factor, optimal results will be obtained for location similarity. As a result, experiments in the following sections will only be concerned with the Manhattan distance or direct distance or variations of direct distance between the center of a module and its center position in the previous design iteration.

We also believe that if we obtain an optimal floorplan with respect to the area in the previous design iteration, then a good incremental floorplanner should give a good measure of area for the incremental floorplan because of geometric similarity.

3.4 Incremental Floorplanning with all modules

The preceding experiments dealt with five modules that were manually selected. To simulate an industrial scenario, we randomly select modules for various levels of percentage increments. For example, in one scenario, 60% of all modules have a 0% area increment, 30% have a 10% area increment and the remaining 10% have a 20% area

increment. If a module is chosen for an area increase by a random number generator, we again employ a random number generator to either increase the height or the width of that module with equal probability.

Next, we describe the input files used by our incremental floorplanner assuming that our design consists of n modules. There are two input files to our incremental floorplanning program: The first consists of n lines with each line corresponding to a module. Each line has a fixed format: *module_number*, *width*, *height*, *incrementPercentage*, *incrementDimension*. *module_number* denotes the sequential order of modules. *Width* and *height* are the dimensions of bounding rectangles for n modules and they are all set to zero. If *incrementPercentage* is 0, then there is no increment on both width and height of the module. If *incrementPercentage* is greater than 0, then $newWidth = (1 + incrementPercentage) * oldWidth$, or $newHeight = (1 + incrementPercentage) * oldHeight$. The *newWidth* and *newHeight* denote the dimensions of modules in the incremental floorplan. The *oldWidth* and *oldHeight* denote the dimensions of modules in the regular floorplan. The *incrementDimension* controls which dimension to increase the size. If *incrementDimension* is 0, we increment the width by *incrementPercentage*. Else if *incrementDimension* is 1, we increment the height by *incrementPercentage*. The second input file is generated by running Sequence-Pair method on the module set without any consideration of incremental floorplan. It has the following format for each of its n lines: *centerXcoordinate*, *centerYcoordinate*, *shape*, *orientation*. *centerXcoordinate* and *centerYcoordinate* are center coordinates

of a module in the resulting floorplan in the first design iteration. *shape* and *orientation* denote the shape and orientation of module in the resulting floorplan in the first design iteration respectively. Next, we run our incremental floorplanning program to make each module with same shape and orientation close to where it was in the first floorplan. In this set of experiments we tried a number of cost functions and mixed the area component with the distance component. We also tried using only the distance component itself and change the temperature parameter for the simulated annealing to see if we can get an incremental floorplan that is optimal in area and distance. We find out that if only distance component is included in the cost function then the resulting floorplan is very similar to the original floorplan in terms of geometry of each module. Also not surprisingly if we reduce the temperature more slowly then the program runs longer and produces a better incremental floorplan. We used ami33.cel, a well-known benchmark that consists of 33 modules for our experiments here.

Table 3.8 shows the result. F denotes various cost functions. *Cost* denotes the final cost of the floorplan. *Area* denotes the final area of the floorplan. P denotes the sum of the Manhattan distance from the center position of each module to the center position of each module in previous design iteration. W denotes the white space ratio of the new floorplan. D^2 denotes the sum of the square of direct distance from the center position of each module to the center position of each module in previous design iteration. D denotes the sum of the direct distance from the center position of each module to the center position of each module in previous design iteration. D^3 denotes the sum of the

Table 3.8: Incremental Floorplanning Using Cost Functions.

Experiment	F	Cost	Area	P	W
0	a + 300 * P	4007970	1580670	8091	23.3%
1	a + 500 * P	5244374	1644874	7199	26.3%
2	a + 600 * P	4128306	1675506	4088	27.6%
3	a + 700 * P	5514718	1531018	5691	20.8%
4	a + 800 * P	4230083	1708483	3152	29.1%
5	a + 900 * P	5063594	1729994	3704	29.9%
6	P	4935	2059050	4935	41.1%
7	D^2	239416	1568150	2044	22.7%

cube of direct distance from the center position of each module to the center position of each module in previous design iteration. We can see that the cost function D^2 performs the best in terms of a combination of area and geometric similarity. In Figure 3.2, the Manhattan distance is $a + b$ where a denotes the vertical distances between the centers of module A in the different iterations and b denotes the horizontal distance. d denotes the direct distance between the centers of module A in the different iterations.

Then we tried various cost functions that consist of only the distance factor. Table 3.9 shows the result. In this set of experiments, we use $npt = 800$ and $nrc = 45$ for our simulated annealing process. npt is the number of moves per temperature. nrc is the number of cooling steps. As we can see, we obtain a very good incremental floorplan because we reduce the temperature more slowly.

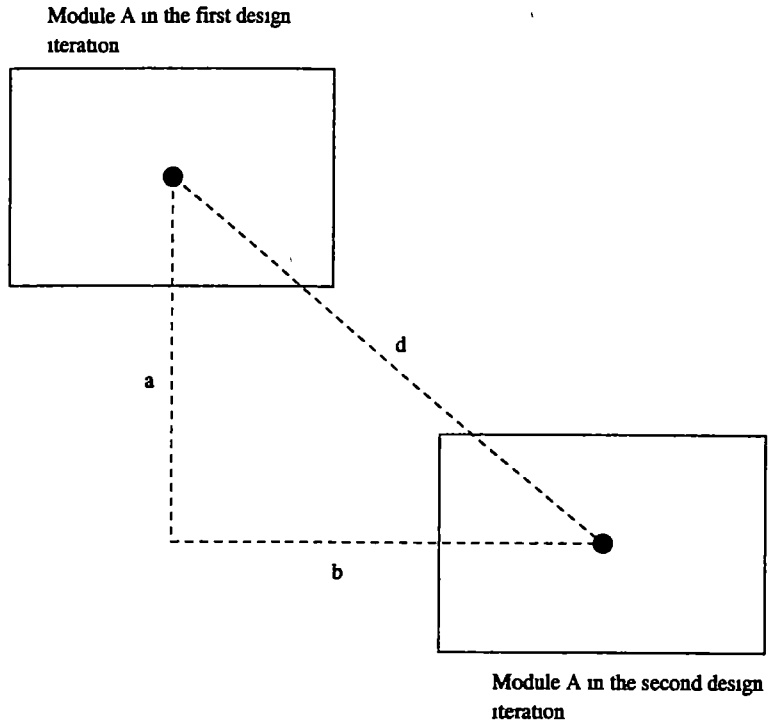

Figure 3.2: Manhattan distance and Direct Distance Example

Table 3.9: Incremental Floorplanning Using Distance Functions

Experiment	F	Area	P
0	D	1488410	2035
1	D^2	1542912	1777
2	D^3	1509456	1636

Table 3.10: Lower Bound on Area Using Cost Functions.

Experiment	Area
0	1398600
1	1382255
2	1339128

Table 3.11: Incremental Floorplanning Using Same Sequence Pair.

Experiment	Original Area	New Area	P	Q
0	1340640	1485392	2038	18.3%

We also include the cost function that consists of only area term to obtain a lower bound on the performance of the area of incremental floorplan. Table 3.10 shows the result.

The other incremental floorplanning technique includes keeping the sequence pair the same as the final sequence pair in the regular floorplanning to see if it will give us a good result in terms of area and similarity of module location. The reason is that by keeping the sequence pair the same, the geometrical relationship between every pair of modules such as right-of or above-of is kept constant and consequently their locations should be similar. Table 3.11 shows the result. As we can see, this method does yield a good result for small increments.

So far, our experiments used the increment scenario described on Page 31. In the following experiments, we consider four other increment scenarios.

For each scenario, we carry out 5 experiments. three use cost functions with various distance measures, one uses the sequence-pair generated by the regular floorplanner,

Table 3.12: Incremental Floorplanning With Scenario 1.

Experiment	F	Area	P	Q
0	D	2219917	5826	30.0%
1	D^2	2215785	5702	29.9%
2	D^3	2156904	6355	28.0%
3	S	2565972	12338	39.5%
4	*Area*	1715792	27030	9.5%

Table 3.13: Incremental Floorplanning With Scenario 2.

Experiment	F	Area	P	Q
0	D	2741555	7119	34.4%
1	D^2	2666558	10801	32.5%
2	D^3	2610448	10498	31.1%
3	S	2859516	16361	37.1%
4	*Area*	1906065	28996	5.6%

and one uses a cost function consisting of only an area term to obtain a lower bound.

There are four scenarios.

In scenario 1, 30% of all modules have a 0% area increment, 50% have a 50% area increment and the remaining 20% have a 80% area increment. Table 3.12 shows the result. S denotes the technique where we kept the sequence-pair the same as the final sequence-pair in the regular floorplan.

In scenario 2, 10% of all modules have a 0% area increment, 60% have a 50% area increment and the remaining 30% of modules have a 80% area increment. Table 3.13 shows the result.

In scenario 3, 10% of all modules have a 0% area increment, 60% have a 80% area increment and the remaining 30% have a 100% area increment. Table 3.14 shows the

Table 3.14: Incremental Floorplanning With Scenario 3.

Experiment	F	Area	P	Q
0	D	2979808	8830	32.8%
1	D^2	2830377	10200	29.2%
2	D^3	2669748	11291	25.0%
3	S	3280881	19462	38.9%
4	*Area*	2160900	29686	7.4%

Table 3.15: Incremental Floorplanning With Scenario 4.

Experiment	F	Area	P	Q
0	D	4410000	14364	42.2%
1	D^2	3254866	15187	21.7%
2	D^3	3425512	17311	25.6%
3	S	4751378	28556	46.3%
4	*Area*	2835170	33376	10.1%

result.

In scenario 4, 0% of all modules have a 0% area increment, 50% have a 100% area increment and the remaining 50% have a 150% area increment. Table 3.15 shows the result.

Notice that in going from scenario 1 to scenario 4 more modules change size and the amount of increase is greater. After obtaining all the results, we discovered that there is no universal cost function that wins on both area and distance factor all the time. The cost function with only direct distance factor gives a fairly good estimate for the final distance value. The various cost functions are unpredictable. However the cost function with only area component obtains best area at all the time. We also found out that by keeping sequence pair the same will not give an optimal result. All the above

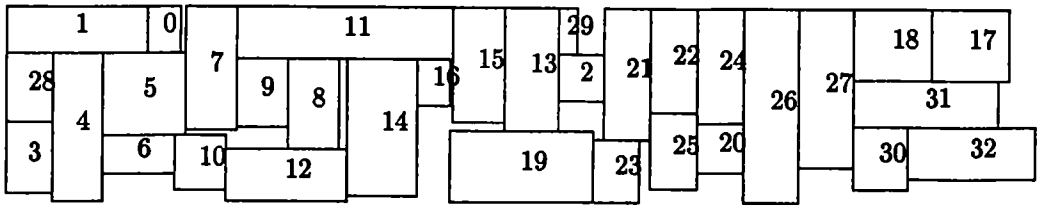

Figure 3.3: Resulting Floorplan In First Design Iteration

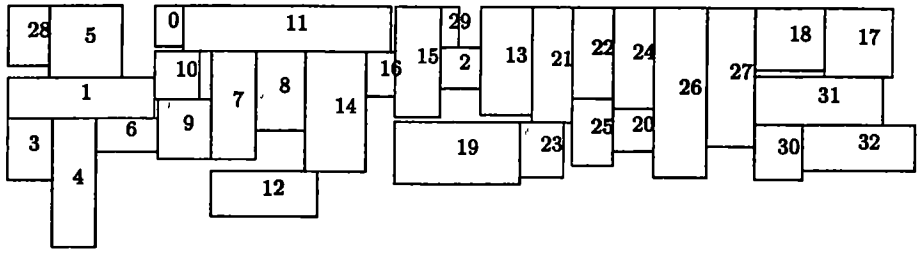

Figure 3.4: Resulting Floorplan In Second Design Iteration With Scenario 0

experiments use npt=800 and nrc=45 for the simulated annealing process.

We chose the best cost function and obtained the graphs corresponding to the five scenarios.

Figure 3.3 is a resulting floorplan that is using cost function of area and wirelength in the first design iteration.

Figure 3.4 is a resulting floorplan that corresponds to scenario 0.

Figure 3.5 is a resulting floorplan that corresponds to scenario 1.

Figure 3.6 is a resulting floorplan that corresponds to scenario 2.

Figure 3.7 is a resulting floorplan that corresponds to scenario 3.

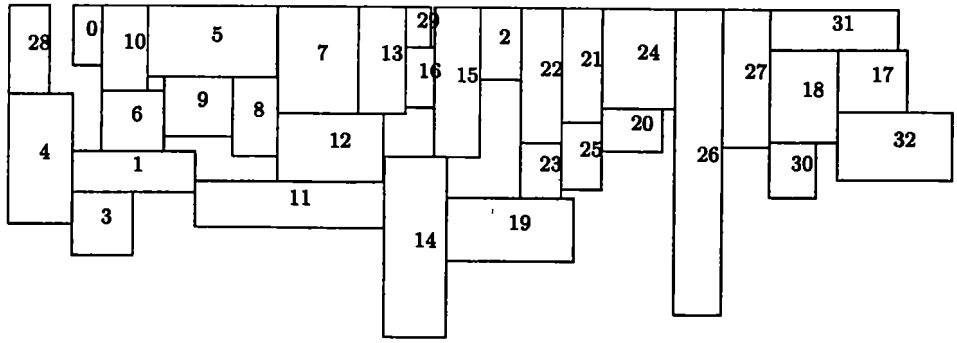

Figure 3.5: Resulting Floorplan In Second Design Iteration With Scenario 1

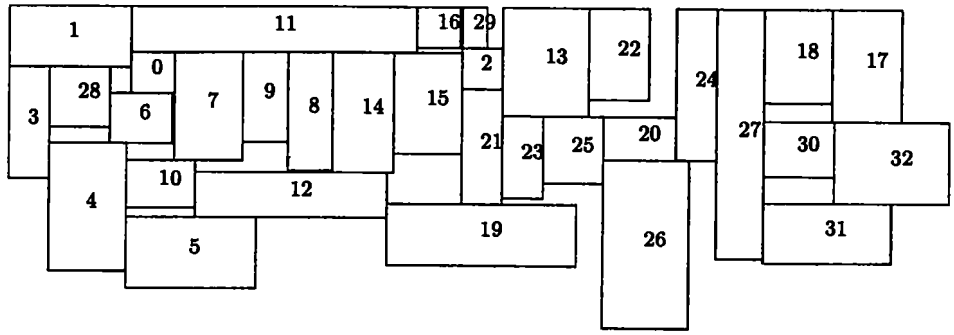

Figure 3.6: Resulting Floorplan In Second Design Iteration With Scenario 2

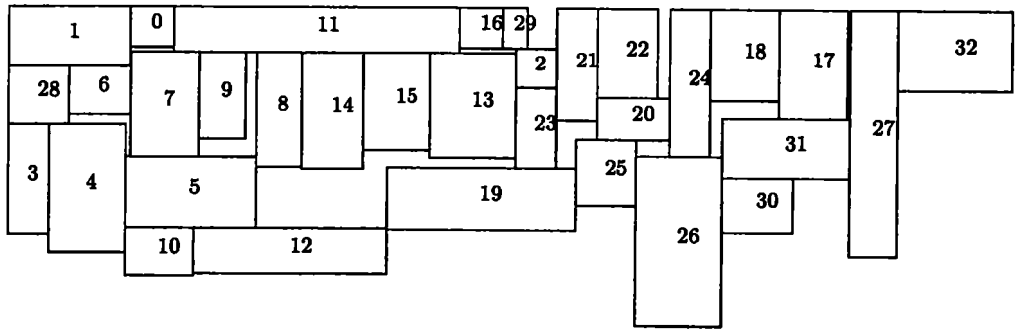

Figure 3.7: Resulting Floorplan In Second Design Iteration With Scenario 3

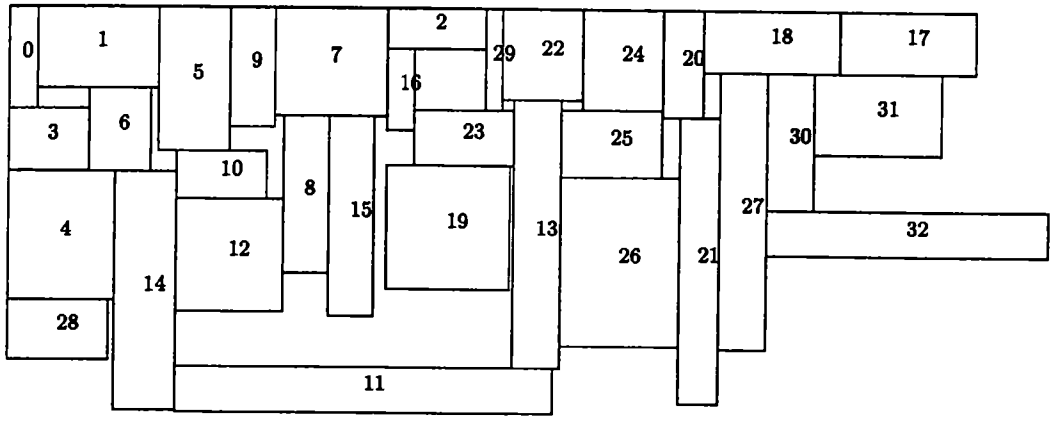

Figure 3.8: Resulting Floorplan In Second Design Iteration With Scenario 4

Figure 3.8 is a resulting floorplan that corresponds to scenario 4.

Because the white space percentage was large enough to cause concern over the area component, we included the area in the cost function to try to balance both area and distance factor. Thus we tried three cost functions:

1. Cost function A = $area * sqrt(numberOfcells) + sumOfdirectdistance^2$

2. Cost function B = $area + sumOfdirectdistance^2/(numberOfcells)$

3. Cost function C = $area * (numberOfcells) + sumOfdirectdistance^2$

Table 3.16 shows the result of using above cost functions.

We find out that Cost function C gives a good measure of area as well as distance factor. Now we perform experiments on another well-known benchmark ami49.cel which consists of 49 modules. Because the change of size reflected in the above experiments are a little bit large than usual, so we design four scenarios that is close to real life situation:

Table 3.16: Incremental Floorplanning Using Cost Function With Area and Distance Factor

Experiment	F	Area	P
0	A	1602048	8131
1	B	1445500	21774
2	C	1404711	13019

Table 3.17: Incremental Floorplanning Using Input ami49 with scenario 1.

Experiment	F	Area	P
0	$F1$	41963600	125378
1	$F2$	41796860	134742
2	$F3$	43751008	82828
3	$F4$	44959320	108170
4	$F5$	44779200	32330

Scenario 1 is 60% of all modules have a 0% area increment, 30% have a 10% area increment and the remaining 10% have a 20% area increment.

Table 3.17 shows the result of using various cost functions with benchmark ami49.cel. $F1$ denotes the cost function $49 * Area + D^2$. $F2$ denotes the cost function $Area$. $F3$ denotes the cost function $30 * Area + D^2$. $F4$ denotes the cost function $40 * Area + D^2$. $F5$ denotes the cost function D^2. D^2 denotes the sum of the square of direct distance from the center position of each module to the center position of each module in previous design iteration. The modules' orientation and shape are kept constant.

Scenario 2 is 30% of all modules have a 0% area increment, 30% have a 10% area increment, 20% have a 20% area increment and the remaining 20% have a 40% area increment. Table 3.18 shows the result.

Table 3.18: Incremental Floorplanning Using Input ami49 with scenario 2

Experiment	F	Area	P
0	$F1$	51427376	116153
1	$F2$	49003812	154711
2	$F3$	49540040	143445
3	$F4$	49395936	94123
4	$F5$	62759704	45179

Table 3.19: Incremental Floorplanning Using Input ami49 with scenario 3.

Experiment	F	Area	P
0	$F1$	53816832	134641
1	$F2$	47251440	156761
2	$F3$	49359168	116189
3	$F4$	49824640	133775
4	$F5$	71115556	51775

Scenario 3 is 20% of all modules have a 0% area increment, 30% have a 10% area increment, 20% have a 20% area increment, 20% have a 40% area increment and the remaining 10% have a 100% area increment. Table 3.19 shows the result.

Scenario 4 is 50% of all modules have a 0% area increment, 30% have a 10% area increment, 10% have a 20% area increment and the remaining 10% have a 100% area increment. Table 3.20 shows the result.

From the above experiments we can see the cost function with only area term gives the best area for all scenarios. And the cost function with only distance_factor term gives the best distance value for all the scenarios.

In the above experiments, the orientation of all modules are as same as those obtained from the first design iteration. Next we permitted the orientation of all the

Table 3.20: Incremental Floorplanning Using Input ami49 with scenario 4.

Experiment	F	Area	P
0	$F1$	45135160	107517
1	$F2$	43179648	157665
2	$F3$	45658288	120429
3	$F4$	45181136	101191
4	$F5$	53707500	48889

Table 3.21: Incremental Floorplanning Using Different Orientation

Experiment	F	A1	D1	A2	D2
1	f	1404711	13019	1401964	19431
2	f	1480440	20554	1549530	19274
3	f	1631932	21627	1674400	22648
4	f	1528028	23583	1564254	20810

modules to change in the hope that we would get a better area measurement because of this additional flexibility. However, the results do not confirm this intuition. Table 3.21 shows the result. This set of experiments deal with ami33.cel. Here f denotes the cost function $33 * Area + D^2$. $A1$ denotes the area when the orientation is fixed to make all modules' orientation as same as what they are in the first iteration. $D1$ denotes the distance factor value for the fixed orientation floorplanning. $A2$ denotes the area when the orientation changes in the second iteration. $D2$ denotes the the distance factor value for the free orientation floorplanning. The distance factor here is the sum of the Manhattan distance from the center of all modules in the second iteration to the center of all modules in the first iteration.

Chapter 4

Path-Constrained Floorplanning

4.1 Motivation

In VLSI design, it is very important to improve chip performance. This is because:

1. Improved chip performance enables more complex software systems to become commercially viable.

2. The general user has an expectation that chip performance increases every few months. This results in the commercial success of the chip being closely tied to its performance.

Chip performance is usually measured by its clock cycle frequency. The clock cycle frequency is the inverse of clock cycle time. The clock cycle time is determined by the delay on certain paths on the chip. The delay of a path is the sum of the delays of the cells in the path and the delays due to the wires used for interconnecting these

cells. The delay between two successive logic stages is composed of three elements (1) The intrinsic delay due to switching a cell ON or OFF (2) Delay due to charging or discharging the fan out and load capacitance and (3) Delay due to distributed RC of interconnection. The delay components (1) and (2) are typically grouped as gate delay.

With advances in semiconductor technology and the consequent shrinkage of device geometry, interconnect delay has become comparable to gate delay. Interconnect delay is related to the length of the interconnect in the design, which is itself related to the relative locations of modules in a floorplan. In this chapter, we will be concerned with the delay of a small subset of paths (called critical paths). A critical path is one whose modules and interconnect must be arranged in a near-optimal fashion for the chip to meet its performance bounds.

4.2 Problem Formulation

We denote the set of all modules $M = \{M_1, M_2, ..., M_n\}$. A path in a circuit, is an ordered sequence of distinct modules, $(M_{i_1}, M_{i_2}, ..., M_{i_k})$, where $M_{i_{j+1}}$ is a fanout of M_{i_j} for $1 \leq j \leq k-1$. There is a set of (input and output) pins for the chip $p = \{p_1, p_2, ..., p_q\}$. A critical path is a path that originates at an input pin and ends at an output pin, such that the modules and the interconnect in the path must be placed in near-optimal fashion. We use the term "path-module" to denote a module that belongs to at least one critical path. Figure 4.1 shows an example of a critical path with modules A, B, C, D, E, F. The black dots on the boundary of the chip are the input pin and the

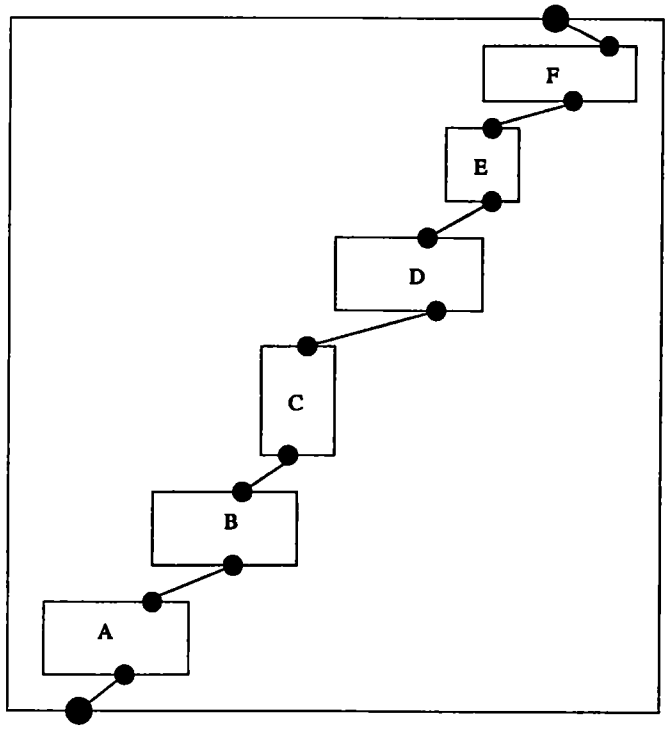

Figure 4.1: An Example of A Critical Path

output pin of the chip. The gray dots on each module mark the input pin and output pin positions on individual modules.

The objective of this research is to solve a constrained version of the floorplanning problem, where in addition to the usual goals of minimizing area and wirelength, the sequence of modules that belong to a critical path are placed on an approximate straight line between the input pin and the output pin.

We used simulated annealing based on sequence-pairs to solve the problem. A randomly generated sequence-pair will, in general, violate path constraints. There are three strategies for dealing with this problem.

1. Convert an infeasible sequence-pair to a feasible one

2. Ignore infeasible sequence-pairs

3. Introduce a penalty term that discourages path violations

We use the first strategy to handle path constraints.

4.3 Path-Constrained Floorplanning Algorithm

The problem, as stated above, is rather complex. We make two simplifying assumptions:

1. The straight line along which critical path modules must be placed is either horizontal or vertical. Ideally, the centers of the modules must lie on this line. However, this is restrictive, so we permit module centers to lie within a small specified distance from the line. (We will refer to this thick line as a band.)

2. A module may exist on at most one critical path.

The input to our procedure for converting an infeasible sequence-pair to a feasible one consists of:

1. A sequence pair

2. A set of n modules and their dimensions and

3. A set of critical paths. Each critical path includes a module sequence and a specification of the horizontal/vertical band on which the modules must be placed.

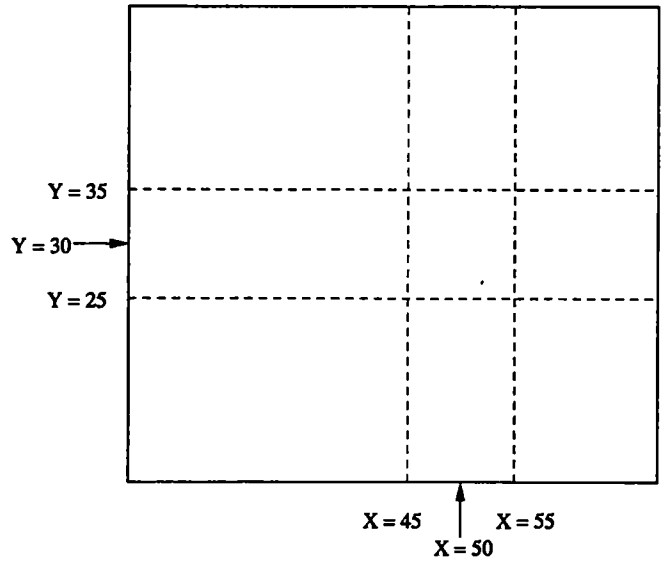

Figure 4.2: Horizontal and Vertical Bands

Figure 4.2 shows horizontal and vertical bands centered at $y = 30$ and $x = 50$, respectively. In both cases, the band width $\alpha = 10$. The dotted lines denote the extents of the two bands.

Output: A feasible sequence-pair such that path-modules are placed within the corresponding bands.

Our algorithm to convert an infeasible sequence-pair to a feasible one traverses Γ^- from left to right. The algorithm maintains partial horizontal and vertical constraint graphs during this traversal. At any given point in the traversal, the graphs contain precisely those modules which have been traversed in Γ^-. The location of these modules can be determined from the graphs. Path-modules that are outside their band are relocated so that they are in their band. This relocation is achieved by rearranging modules in Γ^+. Thus, our algorithm leaves Γ^- unchanged.

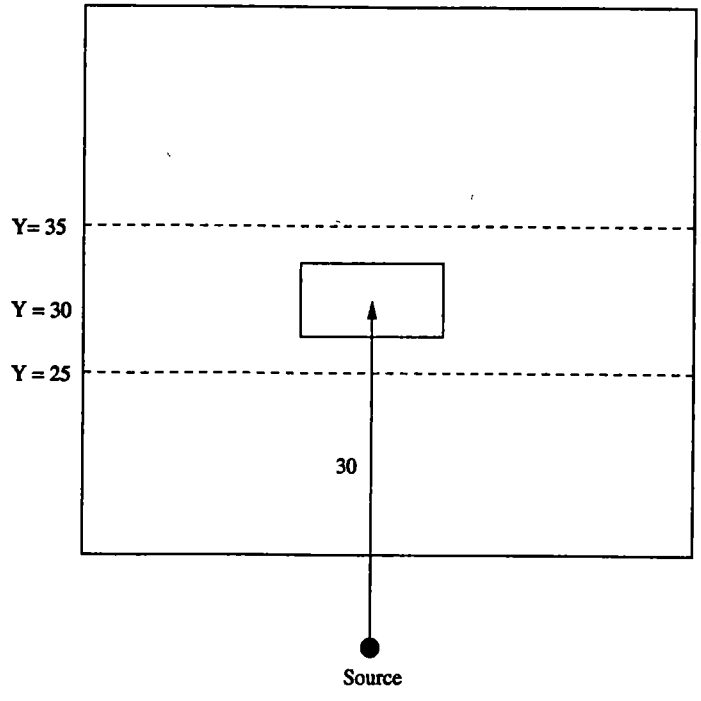

Figure 4.3: Propped Edge Example

For all modules on a critical path, we draw a propped edge from the source to the path module with a weight equal to the center of band so that the path module will never be placed below the band. Figure 4.3 shows an example of a propped edge. The module is placed onto a horizontal band with center position $Y = 30$. The propped edge from the source to the module has a weight of 30, ensuring that the path-module can not be below the band.

Step 1: Sort path modules in Γ^+ and Γ^-

1. if a given path is horizontal, sort both Γ^+ and Γ^- to make sure the path-modules are occurring in the same sequence. For example, if modules *a*, *b*, *c*, *d* are path modules that are to be placed onto a horizontal path from left to right, then our

sequence-pair would look like the following: $(\Gamma^+, \Gamma^-) = (...a...b...c...d..., ...a...b...c...d...)$

2. else if a given path is vertical, sort both Γ^+ and Γ^- to make sure the path-modules are occurring in the opposite sequence. For example, if modules p, q, s, t are path modules that are to be placed onto a vertical path from bottom to top, then our sequence-pair would look like the following: $(\Gamma^+, \Gamma^-) = (...t...s...q...p..., ...p...q...s...t...)$

While(Path violation has occurred or Γ^- has not been totally traversed), perform step 2.

Step 2: for (int $k = 1$; $k \leq n$; $k++$)

1. Construct partial horizontal and vertical constraint graphs which include modules up to kth position in Γ^-. After the graph construction, we would obtain X/Y coordinates of all modules in Γ^- up to kth position. We denote the coordinates of module A by (X_A, Y_A). Set kth module A in Γ^- to its corresponding position at (X_A, Y_A).

2. if kth module A is a path module, then:

   (a) if A is on a horizontal band with center Y_p and $Y_A - Y_p > \alpha/2$, move A and all the necessary modules on A's path after the "violation module" in Γ^+. The "violation module" is a module that is required to be below A by the sequence-pair and has pushed path-module A out of its horizontal band. The "necessary" modules here are path-modules on same critical path as module

A that are located between the "violation module" and A in Γ^+.

(b) else if A is on vertical band with center X_p and $X_A - X_p > \alpha/2$, move A and all the necessary modules on A's path before the "violation module" in Γ^+. The "violation module" is a module that is required to be left of A by the sequence-pair and has pushed path-module A out of its vertical band. The "necessary" modules here are path-modules on same critical path as module A that are located between the "violation module" and A in Γ^+.

4.4 An Illustrative Example

To better understand the problem, we present an example to show how our algorithm works. In this example, we have one horizontal and one vertical band. We have modules H = (a, b, c, d) placed on the horizontal band from left to right and modules V = (p, q, s, t) on the vertical band from bottom to top. In addition, we have two free modules m, n. The center of horizontal band is $y = 30$ and the center of vertical band is $x = 30$. To guarantee the ordering of horizontal path-modules, we sort the sequence-pair such that $(\Gamma^+, \Gamma^-) = (...a...b...c...d..., ...a...b...c...d...)$. To guarantee the ordering of vertical path-modules, we sort the sequence pair such that $(\Gamma^+, \Gamma^-) = (...t...s...q...p..., ...p...q...s...t...)$. Both horizontal and vertical bands have width 10. We use (X_a, Y_a) to denote the coordinates of module a after it has been traversed in Γ^-. To be considered within horizontal band, $|Y_u - 30| \leq 5, u \in H$. Similarly, to be considered within vertical band, $|X_w - 30| \leq 5, w \in V$. Otherwise, we use term "out of path" to indicate that a path

Table 4.1: Module Specification for Path-Constrained Floorplanning Example

Modules	Width	Height
m	15	30
n	8	6
a	12	15
b	6	4
c	8	4
d	10	8
p	6	9
q	4	8
s	10	4
t	5	10

violation has occurred. Table 4.1 shows the modules that are used in our example and their dimensions.

Initially, we have a random sequence-pair $(\Gamma^+, \Gamma^-) = (cmdqbptsna, mbntcpsaqd)$. After our sorting procedure, we have $(\Gamma^+, \Gamma^-) = (ambtcsqpnd, manpbqsctd)$ as our initial sequence-pair. Note that the position of free modules in Γ^+ and Γ^- are unaffected by this step. We now traverse Γ^- for each individual module.

k = 1: $(\Gamma^+, \Gamma^-) = (ambtcsqpnd, manpbqsctd)$. First module in Γ^- is m. Since m is a free module, set it to its corresponding position. Figure 4.4 shows the resulting configuration.

k=2: $(\Gamma^+, \Gamma^-) = (ambtcsqpnd, manpbqsctd)$. Now we are dealing with module a, a is a horizontal path-module and m has pushed a out of path. We use the term "violation-module" to denote the module that has caused a path-module to be pushed out of path. In this case, m is the "violation-module". Figure 4.5 illustrates the situation.

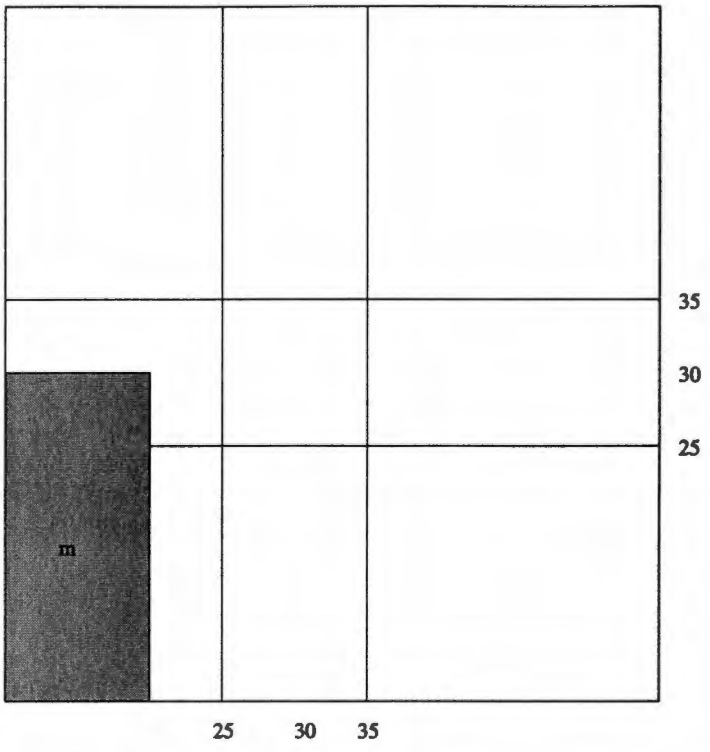

Figure 4.4: An Path-Constrained Floorplanning Example: Step 1

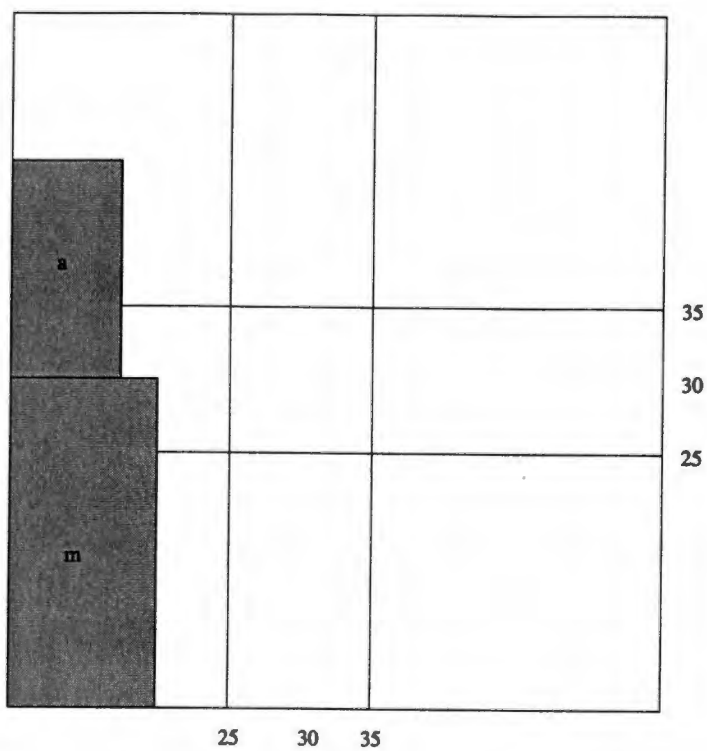

Figure 4.5: An Path-Constrained Floorplanning Example: Step 2-1

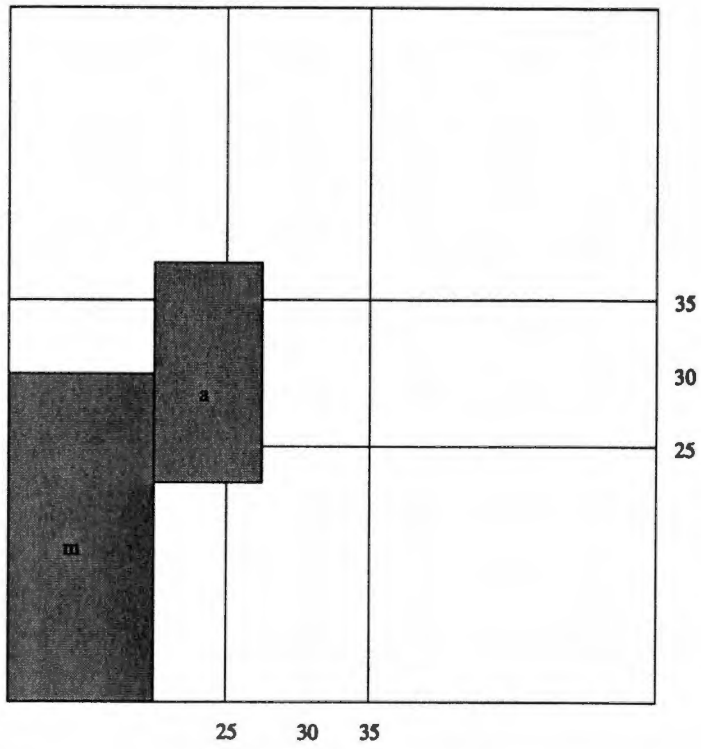

Figure 4.6: An Path-Constrained Floorplanning Example: Step 2-2

Because a is on horizontal path, we move path-module a after the "violation-module" m in Γ^+. There are no other modules that are between a and m in Γ^+ that are also on the horizontal path. $(\Gamma^+, \Gamma^-) = (ambtcsqpnd, manpbqsctd) \Rightarrow (\Gamma^+, \Gamma^-) = (mabtcsqpnd, manpbqsctd)$ Figure 4.6 shows the resulting configuration.

k=3: $(\Gamma^+, \Gamma^-) = (mabtcsqpnd, manpbqsctd)$. Module n is a free module, we set it to its corresponding position. Figure 4.7 shows the resulting configuration.

k=4: $(\Gamma^+, \Gamma^-) = (mabtcsqpnd, manpbqsctd)$. Module p is a vertical path-module. We set it to its corresponding position. No path violation has occurred. Figure 4.8 shows the resulting configuration.

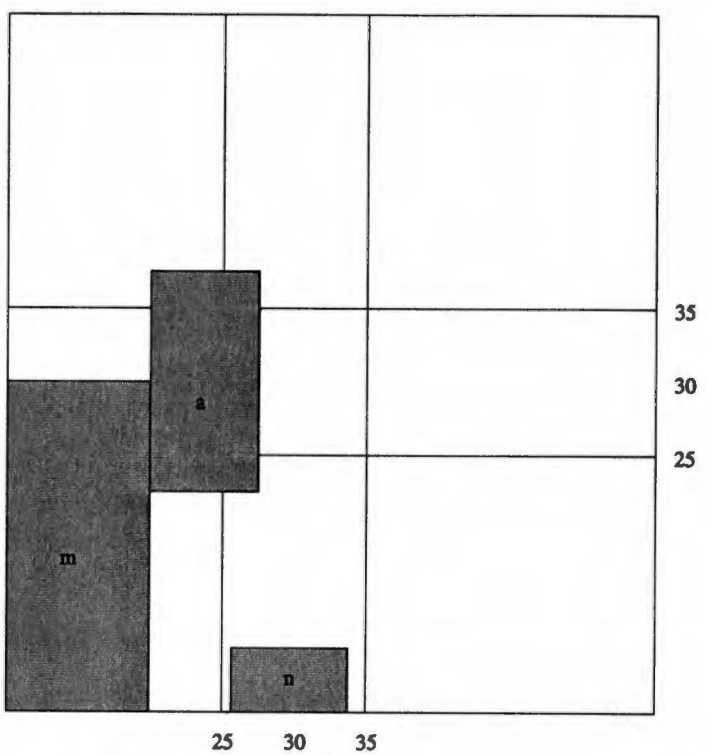

Figure 4.7: An Path-Constrained Floorplanning Example: Step 3

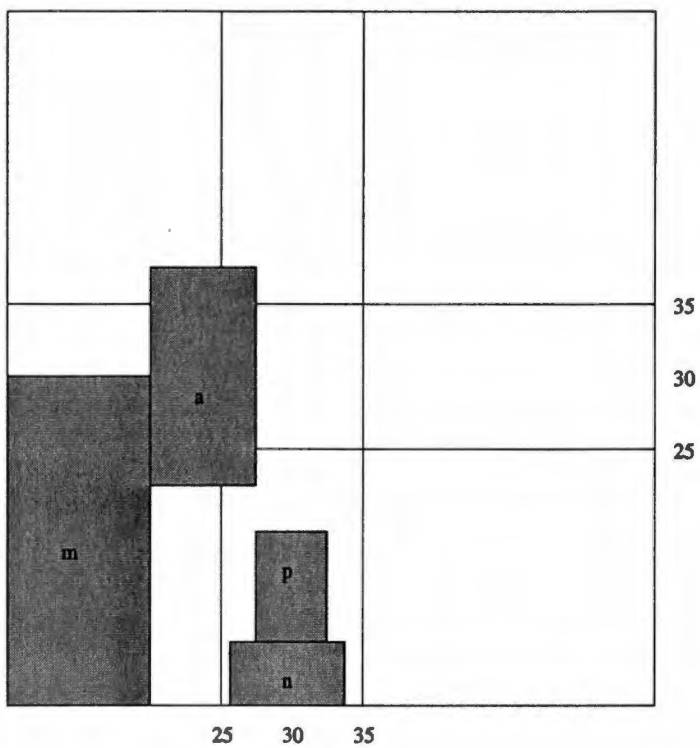

Figure 4.8: An Path-Constrained Floorplanning Example: Step 4

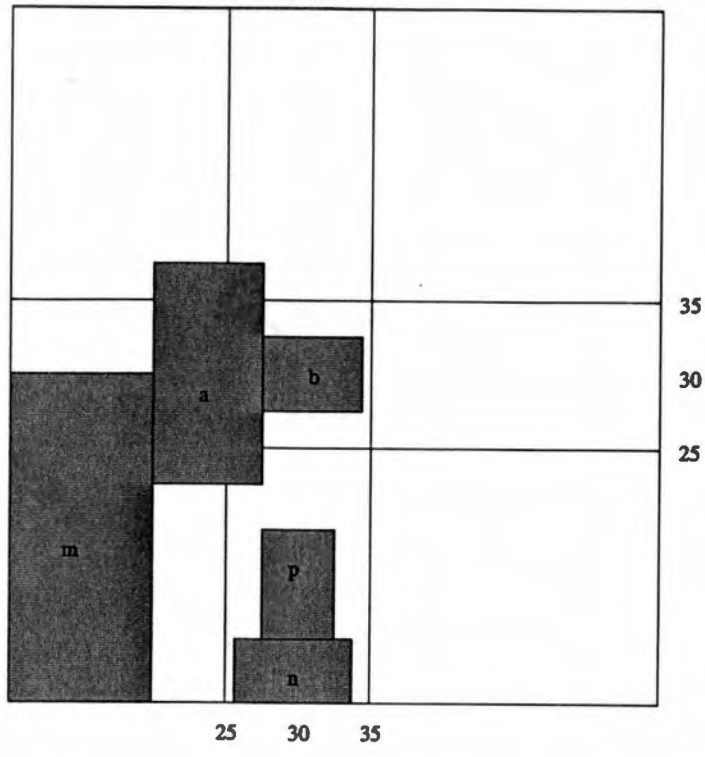

Figure 4.9: An Path-Constrained Floorplanning Example: Step 5

k=5: $(\Gamma^+, \Gamma^-) = (mabtcsqpnd, manpbqsctd)$. Module b is a horizontal path-module. We set it to its corresponding position. Figure 4.9 shows the resulting configuration.

k=6: $(\Gamma^+, \Gamma^-) = (mabtcsqpnd, manpbqsctd)$. Module q is a vertical path-module. Module b has pushed module q out of path. Figure 4.10 shows the resulting configuration. Because module q is on vertical band and path violation has occurred, we move path-module q in the Γ^+ before "violation-module" b. Modules t and s are modules that are between b and q in Γ^+ and they are also on same vertical path with module q. We thus move module t, s, and q before b in Γ^+. $(\Gamma^+, \Gamma^-) = (mabtcsqpnd, manpbqsctd) \Rightarrow$ $(\Gamma^+, \Gamma^-) = (matsqbcpnd, manpbqsctd)$. Figure 4.11 shows the resulting configuration.

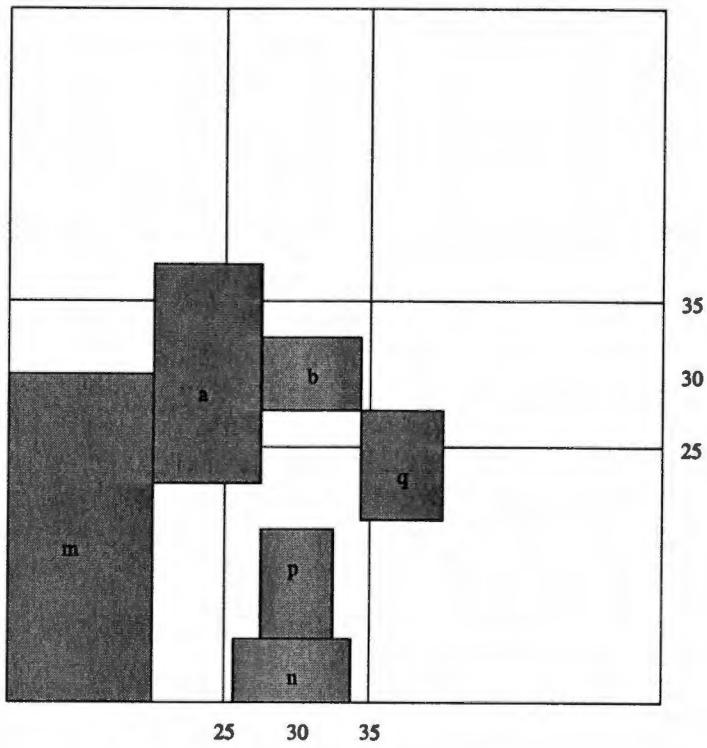

Figure 4.10: An Path-Constrained Floorplanning Example: Step 6-1

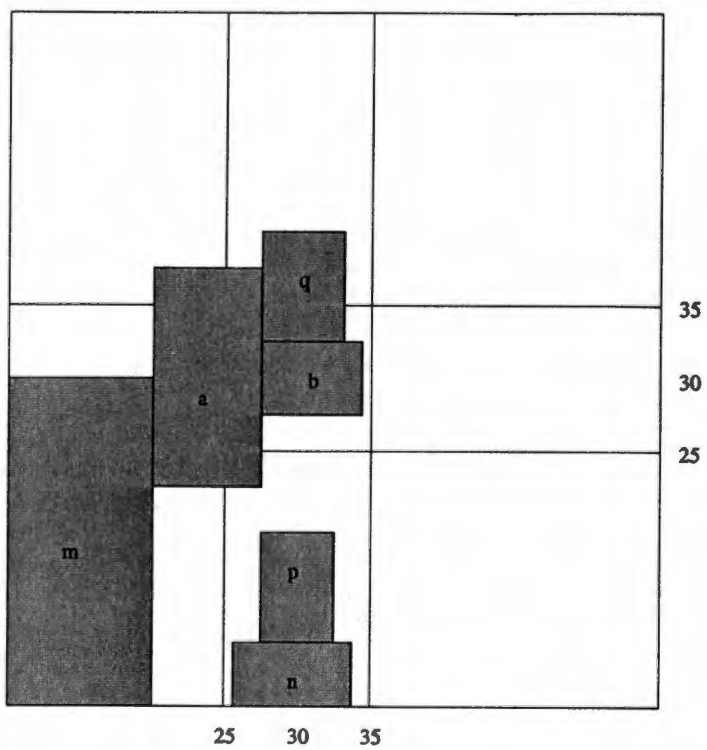

Figure 4.11: An Path-Constrained Floorplanning Example: Step 6-2

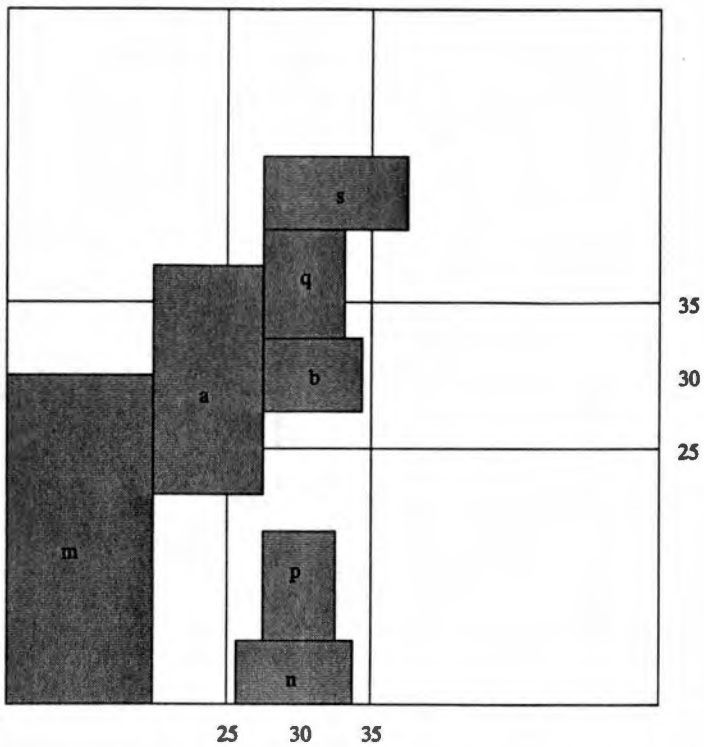

Figure 4.12: An Path-Constrained Floorplanning Example: Step 7

k=7: $(\Gamma^+, \Gamma^-) = (matsqbcpnd, manpbqsctd)$. Module s is a vertical path-module. We set it to its corresponding position. No path violation has occurred. Figure 4.12 shows the resulting configuration.

k=8: $(\Gamma^+, \Gamma^-) = (matsqbcpnd, manpbqsctd)$. Module c is a horizontal path-module. We set it to its corresponding position. No path violation has occurred. Figure 4.13 shows the resulting configuration.

k=9: $(\Gamma^+, \Gamma^-) = (matsqbcpnd, manpbqsctd)$. Module t is a vertical path-module. We set it to its corresponding position. No path violation has occurred. Figure 4.14 shows the resulting configuration.

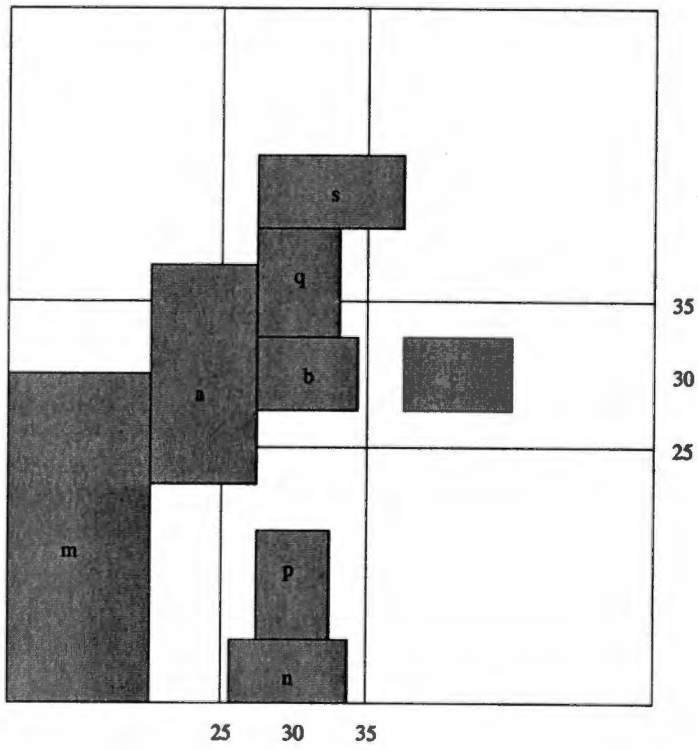

Figure 4.13: An Path-Constrained Floorplanning Example: Step 8

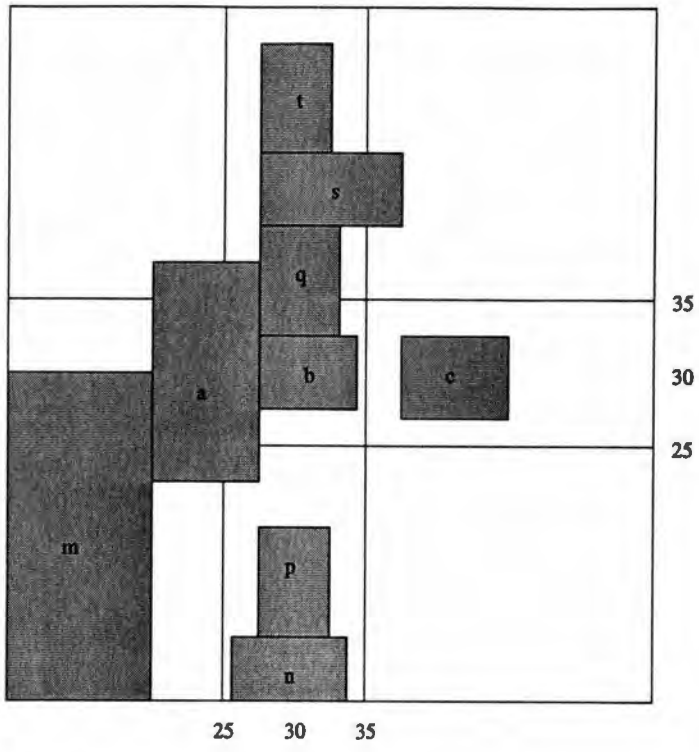

Figure 4.14: An Path-Constrained Floorplanning Example: Step 9

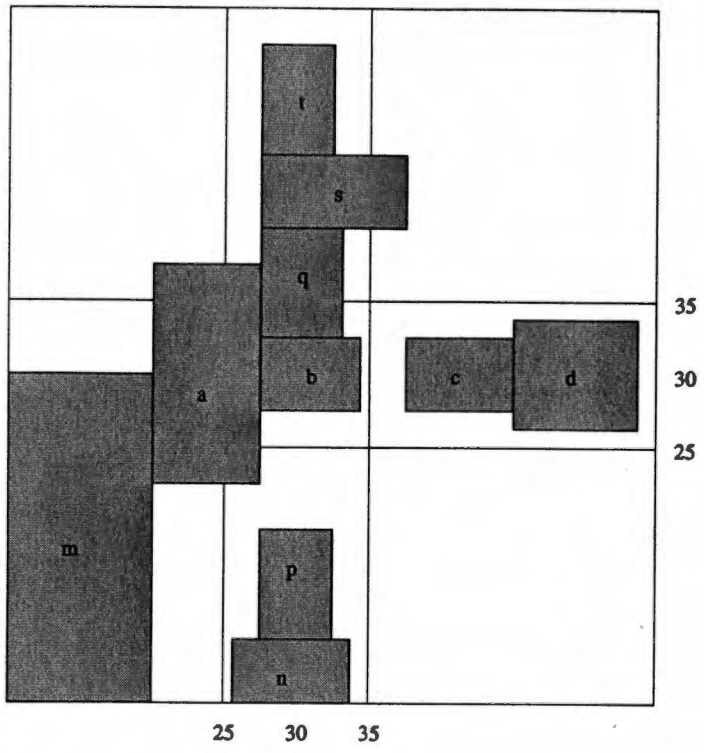

Figure 4.15: An Path-Constrained Floorplanning Example: Step 10

k=10: $(\Gamma^+, \Gamma^-) = (matsqbcpnd, manpbqsctd)$. Module d is a horizontal path-module. We set it to its corresponding position. No path violation has occurred. Figure 4.15 shows the resulting configuration. After we are done with traversing Γ^-, we have obtained a sequence-pair that satisfies the path constraint.

4.5 Proof of Correctness

Theorem: Assume modules may be on at most one critical path, given a sequence-pair with relative position of path modules correctly specified, Path-Constrained Floorplanning Algorithm assures the path constraints always satisfiable.

Fact 1: Let module Y be a horizontal path module, Y would not be pushed out of path by modules before it in Γ^- after Path-Constrained Floorplanning Algorithm finishes.

Proof: All the modules after module Y in Γ^- will be either right of or above module Y, therefore modules after Y in Γ^- can not cause a path violation. Suppose after running Path-Constrained Floorplanning Algorithm , path module Y is still out of horizontal band (ie. above band). The situation implies that there exists a sequence of modules $X_1...X_n$ below Y which push Y out of band. For that to happen, final sequence pair would contain a relative order of Y and $X_1...X_n$ in the following way: $(...Y...X_n...X_1...$, $...X_1...X_n...Y...)$, but according to our algorithm we find least value of index i such that X_i has pushed Y out of path, then our final sequence-pair would look like $(...X_n...X_iY...X_{i-1}...X_1...$, .. $X_1...X_n...Y...)$ So a contradiction has occurred. A similar proof leads the same conclusion when a path-module Y is out of vertical band.

Fact 2: Y is traversed in Γ^-, its position will never be changed.

Proof: Suppose there is a sequence-pair $(...Y...$, $X_0...X_nY...Z...)$. Y's position is solely determined by its predecessors in Γ^-. Let module Z be a horizontal path module. Also let some modules $(X_{i_1}, X_{i_2}, ...X_{i_k})$ in $(X_0...X_n)$ be on the same horizontal path with Z. Because we ensure that the relative position of the path-modules are maintained, modules $(X_{i_1}, X_{i_2}, ..X_{i_k})$ will appear before Z in Γ^+. The violation module that has pushed module Z out of path will be after Z. As a result, there is no module between Z and the violation module in Γ^+ that belong to $(X_{i_1}, X_{i_2}, ...X_{i_k})$, therefore when we

move Z and those modules between Z and violation module in Γ^+ that are on same horizontal path with Z together after the violation module, we are not changing any module's position among $(X_0..X_n)$ in Γ^+. Hence, the position of module Y will not be changed. A similar proof leads the same conclusion when a path module Z is on vertical path.

Because of *Fact 1* and *Fact 2*, we have proved after running Path-Constrained Floorplanning Algorithm, all path modules will be placed onto critical paths correctly.

4.6 Experimental Results

Two sets of experiments have been performed on the well-known floorplanning benchmark ami33.cel which consists of 33 cells. All modules reside on at most one path, and the number of paths range from 1 to 4.

Table 4.2 shows the results of 16 experiments.

1. Experiments 1-4 are performed on an input path specification that has 1 horizontal path with 5 modules on it.

2. Experiments 5-8 are performed on an input path specification that has 1 horizontal path with 5 modules on it and 1 vertical path with 5 modules on it.

3. Experiments 9-12 are performed on an input path specification that has 2 horizontal paths and 1 vertical path, with 5 modules on each path.

4. Experiments 13-16 are performed on an input path specification that has 2 hori-

Table 4.2: Path Constraint Satisfied With Different Path Input 1

Experiment	PN	Satisfied	npt	nrc	Cost	Area	Wirelength
0	0	Don't care	100	15	1.140841	1319325.00	120109.00
1	1	Yes	100	15	2.001780	2314956.00	152545.00
2	1	Yes	100	15	1.853057	2142966.00	110485.00
3	1	Yes	500	15	2.098089	2426333.00	140060.00
4	1	Yes	800	15	1.733232	2004394.00	126986.00
5	2	Yes	100	15	2.325265	2689050.00	124404.00
6	2	Yes	100	15	2.293270	2652050.00	146862.00
7	2	Yes	500	15	2.082115	2407860.00	139587.00
8	2	Yes	800	15	2.168128	2507330.00	122285.00
9	3	Yes	100	15	1.988487	2299584.00	147811.00
10	3	Yes	100	15	2.205627	2550695.00	140314.00
11	3	Yes	500	15	2.572772	2975280.00	141101.00
12	3	Yes	800	15	1.809457	2092545.00	137359.00
13	4	Yes	100	15	3.219161	3722796.00	173178.00
14	4	Yes	100	15	2.724146	3150336.00	154192.00
15	4	Yes	500	15	2.463727	2849175.00	155821.00
16	4	Yes	800	15	2.957018	3419640.00	166504.00

zontal paths and 2 vertical paths, with 5 modules on each path.

PN denotes the number of paths, *Satisfied* indicates whether or not the path constraint is satisfied in the final floorplan, *npt* denotes the number of moves per temperature in the simulated annealing process, *nrc* denotes the number of cooling steps in the simulated annealing process. Experiment 0 is the result of a pure floorplanner without considering the path constraint.

As you can see, all path constraints are satisfied. With number of path increasing, the area of the floorplan is likely to increase. Within one path specification, increase of npt(number of moves per temperature) would likely give a better result(smaller area).

Table 4.3 shows the results of another 16 experiments performed on ami33.cel

Table 4.3: Path Constraints Satisfied With Different Path Input 2

Experiment	PN	Satisfied	npt	nrc	Cost	Area	Wirelength
0	0	Don't care	100	15	1.140841	1319325.00	120109.00
1	1	Yes	100	15	2.040210	2359399.00	132929.00
2	1	Yes	800	15	1.816660	2100875.00	120807.00
3	1	Yes	100	45	1.709758	1977248.00	138453.00
4	1	Yes	800	45	1.963900	2271150.00	128408.00
5	1	Yes	100	15	2.610906	3019380.00	138014.00
6	1	Yes	800	15	2.206686	2551920.00	158915.00
7	1	Yes	100	45	2.052982	2374169.00	129074.00
8	1	Yes	800	45	2.669167	3086755.00	145449.00
9	4	Yes	100	15	2.629947	3041400.00	176844.00
10	4	Yes	800	15	2.204901	2549855.00	152002.00
11	4	Yes	100	45	2.739345	3167913.00	159374.00
12	4	Yes	800	45	2.504882	2896768.00	148629.00
13	4	Yes	100	15	2.335107	2700432.00	140146.00
14	4	Yes	800	15	1.951854	2257220.00	132160.00
15	4	Yes	100	45	1.839946	2127804.00	132215.00
16	4	Yes	800	45	1.921952	2222640.00	121927.00

1. Experiments 1-4 are performed on an input path specification that has 1 horizontal path with 5 modules on it.

2. Experiments 5-8 are performed on an input path specification that has 1 horizontal path with 10 modules on it.

3. Experiments 9-12 are performed on an input path specification that ha 2 horizontal paths and 2 vertical paths, with 5 modules on each path.

4. Experiments 13-16 are performed on an input path specification that has 4 horizontal paths, with 5 modules on each path.

In Table 4.3, we also tried using various combinations of *npt* and *nrc* for individual

experiments. The results show that all path constraints are satisfied. Generally, an increase in the number of paths increases the area of the floorplan. Also if we have a mix of horizontal paths and vertical paths, the area is likely to increase due to the conflict among different directional paths.

Chapter 5

Conclusion

In conclusion, the Path-Constrained Floorplanning algorithm developed in this thesis provides provably feasible solutions for single path-module placement. The numerous incremental floorplanning cost functions have generated floorplans with modules at similar locations to their locations in the previous design iteration.

Some additional comments and ideas for further research and implementation are listed below:

1. A band would be permitted to have different directions other than horizontal or vertical in the path-constrained floorplanning problem.

2. Modules should be permitted to be placed onto multiple paths at the same time. However, this introduces some complexity to the problem. There might exist invalid input path requirement where there is no feasible sequence-pair as solution. It is also difficult to provide a fixing routine to convert an infeasible sequence pair

to a feasible sequence pair. We are considering using cost functions to handle this problem structure by appending penalty term into the cost function to force modules onto various paths.

3. Finally, future extensions would consider employing a more complex cost function to generate a better incremental floorplan.

Bibliography

Bibliography

[1] F. Y. Young and D. F. Wong. How Good Are Slicing Floorplans? In *Proc. of 1997 International Symposium on Physical Design*, pages 144–149, 1997.

[2] F. Y. Young and D. F. Wong. Slicing Floorplans with Range Constraint. In *Proc. of 1999 International Symposium on Physical Design*, pages 97–102, 1999.

[3] F. Y. Young, D. F. Wong, and H. H. Yang. Slicing Floorplans with Boundary Constraints. *IEEE Transactions on Computer-Aided Design*, 18(9):1385–1389, September 1999.

[4] H. Murata and E. Kuh. Sequence-Pair Based Placement Method For Hard/Soft/Pre-Placed Modules. In *Proc. of 1998 International Symposium on Physical Design*, pages 167–172, 1998.

[5] H. Murata, K. Fujiyoshi, M. Kaneko. VLSI/PCB Placement with Obstacles Based on Sequence Pair. *IEEE Transactions on Computer-Aided Design*, 17(1):60–68, January 1998.

[6] H. Murata, K Fujiyoshi, S. Nakatake, Y. Kajitani. VLSI Module Placement based on Rectangle Packing by the Sequence-Pair. *IEEE Transactions on Computer-Aided Design*, 15(12):1518–1524, December 1996.

[7] S. M. Sait and H. Youssef. *VLSI Physical Design Automation*. IEEE Press, New York, 1995.

[8] S. Nakatake, K. Fujiyoshi, H. Murata, Y. Kajitani. Module Packing Based on the BSG-Structure and IC Layout Applications. *IEEE Transactions on Computer-Aided Design*, 17(6):519–530, June 1998.

Vita

Cheng Yu was born on January 28, 1975 in Shanghai, China. He attended schools in the Springfield Public School System where he graduated from Kickapoo High School in Springfield, Missouri in 1994. He received his Bachelor of Science degree while majoring in Computer Science and minoring in Mathematics from Southwest Missouri State University in 1998. In May of 2000 he received his Master of Science in Computer Science from the University of Tennessee Space Institute.