TWO-DIMENSIONAL COMPACTION: COMPUTING THE SHAPE FUNCTION OF A SLICING LAYOUT

Fook-Luen Heng
(Thesis)

CS-TR-175-88

October 1988

Two-dimensional Compaction: Computing the Shape Function of a Slicing Layout

Fook-Luen Heng

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

OCTOBER 1988

© Copyright by Fook-Luen Heng 1988
All Rights Reserved

to my parents

Abstract

We investigate a two-dimensional compaction scheme for a slicing layout. The compaction scheme treats wires as topological entities, i.e. only the paths of wires that connect terminals of the same nets are known but not their physical locations. Wires are not present physically in the layout during compaction, but they are expressed as constraints which describe the actual space required to accommodate them. This amounts to computing the shape function that captures the routing requirement of the layout.

We present an $O(k \cdot n^3)$ algorithm to compute the shape function of a stack of k two-component river routable channels with n nets, and a heuristic to approximate the shape function of a stack of multiple component river routable channels.

We develop heuristics that use the algorithm and the heuristic above as their subroutines to approximate the shape function of a slicing layout under the restriction of river routing. The experimental results show the potential of the proposed compaction scheme.

We study the asymptotic behavior of river routing and pitch aligning in uniform stacks and uniform arrays. We derive the condition under which river routing is better than pitch aligning for such uniform layouts.

We discuss the problem of computing the shape function of a slicing layout with general channel routing. We use channel density as an estimate of channel width. We show an NP-completeness result for computing such a shape function. We present a pseudo-polynomial time algorithm to compute the shape function (using channel density as channel width) for a slicing layout of depth one, and we provide an efficient heuristic to compute such a shape function. Experimental results show that the heuristic produces shape functions which are very close to the exact shape function.

Acknowledgements

My most sincere thanks goes to my research advisor, Professor Andrea LaPaugh whose patience, support, inspiring discussions and careful reading have made this thesis possible.

I thank the readers of this thesis, Professor Kenneth Steiglitz and Dr. Ravi Nair, for their valuable suggestions and helpful comments.

I thank all my friends and colleagues who have made my years in Princeton memorable, the SFC, the Rogue Warriors, the CGSA...

I thank Joseph Wang who provided me with accommodation during my numerous trips to Princeton. I also thank Susan Yeh, for many favors, and Sharon Rodger, who has always been very helpful.

Finally, I wish to thank my parents for their encouragement and constant support through many years, and I also wish to thank a special someone, Ling-Chwun, who cares.

This work was supported in part by DARPA Contract N00014-82-K-0549 and NSF Grants MIP8619335.

Table of Contents

Introduction

The *compaction* of a VLSI layout is to find a minimum area layout subject to *design rules* and *topology* imposed on the layout. The design rules specify displacement constraints between layout components, and the topology specifies the relative positions of the layout components. To be able to obtain a minimum area layout is very important in a VLSI system because area is one of the major cost in fabricating a VLSI chip [MeCo].

A custom VLSI layout consists of layout components (transistors, contacts, independently designed black boxes), and wires interconnecting them. It is typically produced in two phases, namely top down floor-planning and bottom up design. In the floor-planning phase, a circuit is partitioned into smaller subcircuits, each subcircuit is further partitioned into smaller subcircuits. The partitioning process stops when a subcircuit, called the *leaf cell*, is of manageable size. It produces a topology of the layout. The floor-planning phase is completed with a global routing in which the paths of the interconnecting wires between leaf cells are specified. A floor-plan of a layout can either be obtained automatically [LaPo, Lau, Ott, SzOt, WoLi] or manually.

In the bottom up design, each leaf cell is designed independently. Then the detailed routing, in which the actual physical wires that interconnect the leaf cells are laid out, is performed. The design of leaf cells and the detailed routing can be carried out automatically or manually with the aid of layout tools [Eic, LaPo, Mat, LeMe, Ouha]. The layout process described above is not a generic layout process but rather a summary of a general layout process which we will use to show when and where the conventional compaction of a layout occurs, and to point out the difference between our compaction and the conventional one.

The compaction process of a layout can occur at different stages during the layout process. It can be carried out after all leaf cells are designed and the detailed routing is performed. Or it can be carried out in two stages: each leaf cell is compacted independently, and after the detailed routing, the whole layout is compacted and the leaf cells are left intact. In both cases, the compaction occurs after the detailed routing is performed. We propose and investigate a compaction scheme which is different from the conventional one in that it is performed before the detailed routing, but it captures the necessary routing space. The actual detailed routing is performed after the compaction.

We investigate such a compaction scheme for the *slicing layout topology*. A slicing layout topology is a hierarchical partitioning of the layout. In Chapter 1 we give the formal definition of the slicing layout topology. The compaction scheme we consider exploits the inherent hierarchy of the slicing layout topology, and it is a two-dimensional compaction at each level of the hierarchy. The two-dimensional compaction optimizes layout area, allowing both dimensions to vary simultaneously. This is different from a one-dimensional compaction in which the layout is compacted in one dimension at a time.

The advantage of our compaction scheme is twofold. Firstly it treats wires as topological rather than geometrical entities -- this reflects more closely the important characteristics of the actual wires in the compaction stage of the design. And it delays the detailed routing, which very often affects the performance of a compaction scheme both in terms of final area and efficiency, until the later stage of the design. Secondly, it is a two- dimensional compaction; it can find a better solution than the one-dimensional compaction, since the solution space of a two-dimensional compaction is a superset of the solution space of a one-dimensional compaction.

Chapter 1

Compaction and Slicing Layout

A brief overview of constraint-based compaction is given. Our layout model and a slicing layout are defined formally. Some layout tools that employ the slicing paradigm are briefly introduced and their inadequacies are explained. The notion of shape function of a layout is introduced and its application to layout optimization is discussed.

1.1 Constraint-based Compaction and Jog Introduction

A layout consists of layout components interconnected by wires. A layout component can be a transistor, a contact, or a black box which is a collection of components. A component is usually represented by a rectangle with given length and width. A wire consists of wire segments; for compaction, the width of a segment is fixed but the length of the segment is determined by the compaction. The displacements between layout components have to satisfy design rules dictated by the technology. For example in Mead and Conway rules [MeCo] an nMos transistor must be at least 2λ from a buried contact; metal wires must be at least 3λ apart, where λ is a basic unit depends on the technology. See Figure 1.1a and 1.1b.

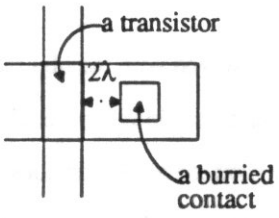


Figure 1.1a

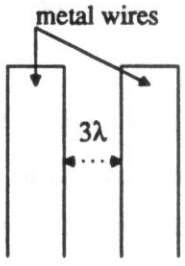
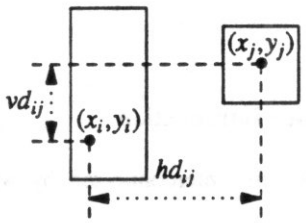


Figure 1.1b

The compaction of a layout is to find a minimum-area layout of components subject to the design rules and the topology of the layout. There are various approaches to layout compaction. In virtual grid compaction [Wes, Ros], the layout components are placed on a grid, and the displacement between each pair of grid lines is determined so that no design rule is violated and the area of the layout is minimized. In

constraint-based compaction [Eic, Hsu, KeWa, Mat], the displacement constraints between layout components are described by a system of linear constraints, and the system of constraints is solved to obtain a compacted layout. We will be focusing on constraint-based compaction to illustrate the merit of our work.

In constraint-based compaction, a layout component is represented by a point inside the component and displacement constraints between components are represented by linear constraints between coordinates of the corresponding points in the components. For example, in Figure 1.2a the minimum horizontal displacement constraint between component i and j is represented by the horizontal linear constraint $x_j - x_i \geq hd_{ij}$. Similarly, the minimum vertical displacement constraints between components are represented by linear constraints of the form $y_j - y_i \geq vd_{ij}$.



minimum horizontal displacement constraint: $x_j - x_i \geq hd_{ij}$
minimum vertical displacement constraint: $y_j - y_i \geq vd_{ij}$

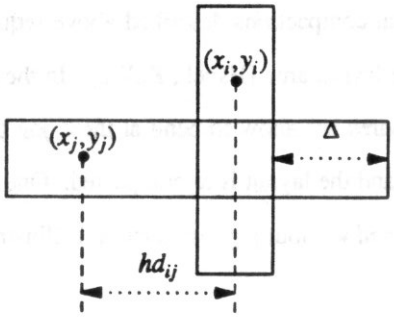
Figure 1.2a

There are other types of constraints which can also be described by linear constraints of similar form. The connectivity constraint is illustrated in Figure 1.2b, and the rigid constraint is illustrated in Figure 1.2c.

A vertical wire is represented by a horizontal coordinate, its width is fixed and it is allowed to stretch along the vertical dimension. This is illustrated in Figure 1.3. Similarly, a horizontal wire is represented by a vertical coordinate.

Typically, the system of horizontal (vertical) linear constraints is solved with the objective of minimizing the horizontal (vertical) dimension of the layout. The solution determines the absolute horizontal (vertical) positions of the components. Then the system of vertical (horizontal) linear constraints is generated and solved. This is regarded as one iteration of the compaction process. The process can be repeated for many iterations until no improvement is made. The solution to the system of linear constraints that minimizes the corresponding dimension of the layout can be computed by different longest path algorithms depending on the assumptions on the linear constraints. [Hsu, Mat, LiWo].

The constraint-based compaction described above is regarded as one dimensional compaction. In one dimensional compaction the systems of horizontal and vertical constraints are solved independently.

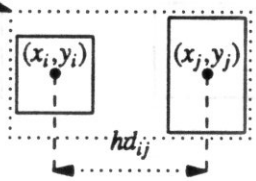


at least Δ units of component j must be on the right of component i to ensure connectivity

connectivity constraint: $x_i - x_j \leq hd_{ij}$
i.e. $x_j - x_i \geq -hd_{ij}$

Figure 1.2b

a bounding box -- a component created from existing components

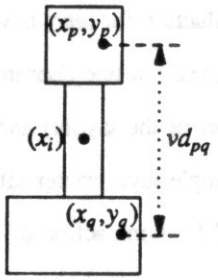


component i and component j must be exactly hd_{ij} units apart to enforce the horizontal dimension of the bounding box

rigid constraint: $x_j - x_i = hd_{ij}$

Figure 1.2c

Value of x_i determines the horizontal position of the wire. Values of y_p and y_q determine the length of the wire



a vertical wire with horizontal coordinate x_i

Figure 1.3

Kedem and Watanabe describe constraints between components by mixed linear-integer inequalities [KeWa]. In their formulation, horizontal and vertical constraints between two adjacent components are specified simultaneously, and a decision variable (with value 1 or 0) is used to determine which of the horizontal and vertical constraints is active. This allows the compaction of the layout to be carried out in both dimensions simultaneously. Their compaction is regarded as two dimensional compaction.

Both the one and two dimensional compactions described above require a jog introduction and a re-compaction step to further improve the layout area [Hsueh, KeWa]. In the jog introduction, jog points are inserted at appropriate wires and the wires are allow to bend at these jog points. After the jog points are inserted new constraints are generated and the layout is re-compacted. One example in which the improvement of the layout area cannot be achieved without jog introduction is illustrated in Figure 1.4.

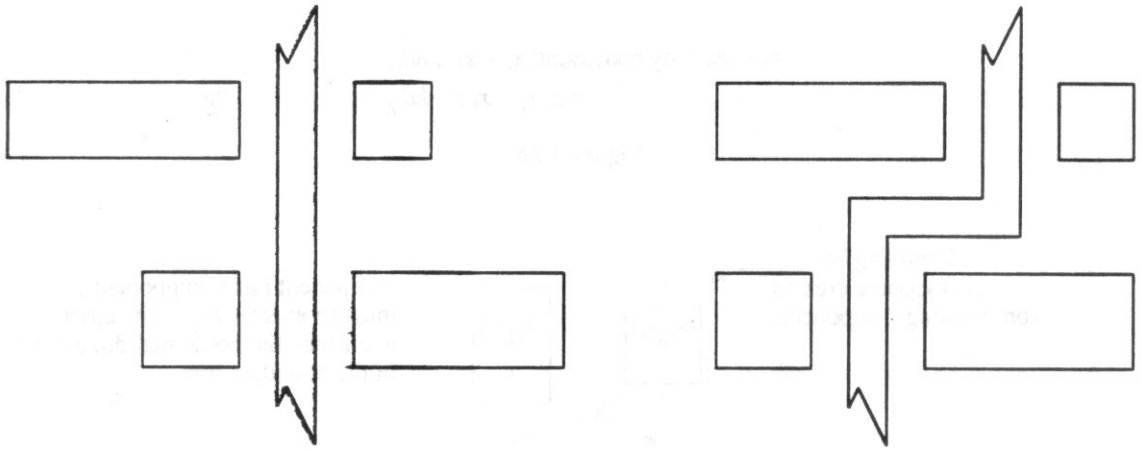


Figure 1.4

Maley studies automatic jog introduction for layouts with single layer routing [Mal]. He treats wires as topological objects and compacts a layout in one chosen dimension. During the compaction, wires are translated into constraints (no longer part of the layout) and the necessary and sufficient routing space is captured. The wires are restored by a single-layer router after the compaction. Mehlhorn and Näher provided an improved algorithm in [MeNä] for compaction of this type. In our compaction scheme, we also treat wires as topological objects, but out compaction is two dimensional.

1.2 Layout Model and Definitions

The VLSI layout we consider consists of two types of objects, namely rectangles and wires. A rectangle can represent a transistor, a contact or a black box which is a layout of a subcircuit. A rectangle has terminals, which are signals coming into it or signals going out of it, on its boundaries. A *net* is a set of terminals that carry the same signal. A wire connects terminals in the same net. Interconnecting terminals of the layout is called *routing* of the layout. The placement and routing of rectangles of a layout obey certain design rules dictated by the technology.

To be able to discuss the compaction problem of a VLSI layout we will require a layout model. A layout model provides a framework for the placement of the basic objects. While there are a variety of VLSI layout models, we shall choose the *grid model*. In the grid model, rectangles and their corners are placed on integral grid points on a rectangular grid. The *routing region* is the region that is not occupied by the rectangles. In addition, wires connecting the rectangles are assumed to have zero width and to run along grid lines of the rectangular grid. The "design rules" in the grid model require that rectangles do not overlap, wires run only in the routing region, and only one wire is allowed in one grid line, i.e. wires are not allowed to overlap. Crossing of wires is allowed at grid points. With the more specific assumptions that horizontal wire segments are in one routing layer and vertical wire segments are in another, the routing model is known as *rectilinear, two-reserved-layer* routing model *. See Figure 1.5 for a layout in the grid model.

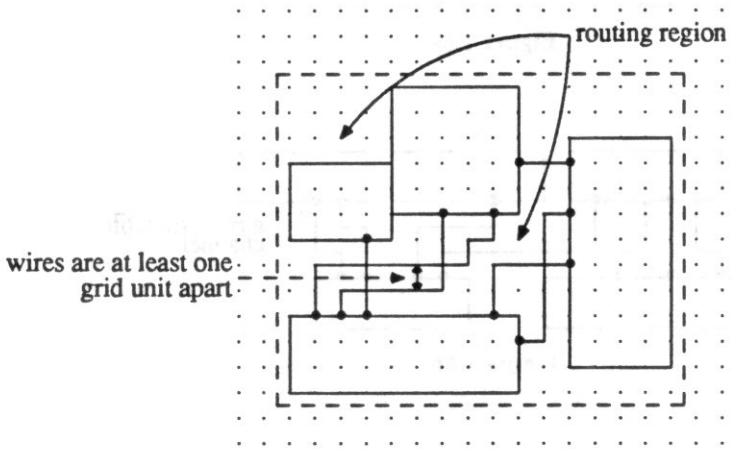


Figure 1.5

The area of a layout is the area of the minimum bounding rectangle that encompasses the layout. Notice that in the grid model, the minimum bounding rectangle can only have integral dimensions. This simplified model will allow us to investigate the algorithmic aspect of compaction, and to evaluate the performance of proposed solutions.

A *routing channel* is a rectangular routing region. Terminals are distributed on the opposite boundaries of the channel (top and bottom, or left and right). The terminals come from rectangles on the opposite boundaries. In general, wires may enter or exit a channel at boundaries which are orthogonal to the

* The rectilinear, two-reserved-layer routing model is not critical to our work. The results in Chapter 5 can be easily extended to other routing models, for example, the *knock-knee* model in which corners of two wires are allowed to overlap at a grid point.

boundaries on which terminals are distributed. The *channel width* of a routing channel is the minimum number of grid lines parallel to the channel, called *tracks*, required to interconnect the terminals. See Figure 1.6a. The *channel routing problem* is to compute the channel width and find a routing that achieves the channel width. If a channel consists only of two-terminal nets, the two terminals of a net are on the opposite sides of the channel, and the terminals on the opposite sides of the channel have the same net ordering, the routing can be done without wire crossing. This class of channel routing problem is called the *river routing problem* and this type of channel is called a *river routable channel*. See Figure 1.6b.

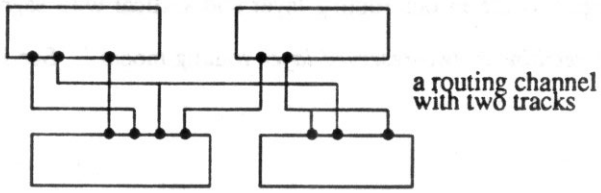


Figure 1.6a

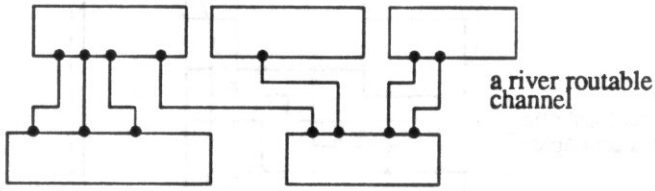
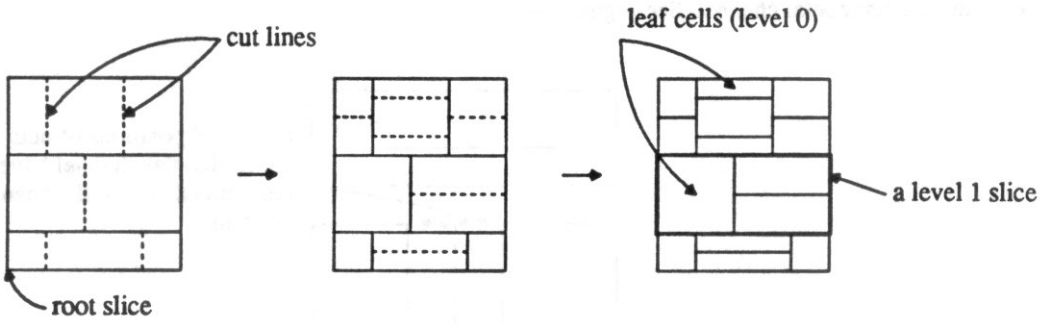


Figure 1.6b

1.3 Slicing Structure

In this thesis we investigate the compaction problem for a type of layout topology known as *slicing structure*. A *slicing structure* is a partition of a rectangle into smaller rectangles, called *slices*, by using a set of parallel lines, called *cut lines*. Each of the slices is further partitioned by another set of cut lines which is orthogonal to the previous set, and so on. The total number of alternate vertical and horizontal slicing operations is the *depth* of the slicing structure. The slicing process can be carried out to any depth. Slices resulting from a partition of a slice s are called *children* of s, and s is called the *parent* of its child slices. The bounding rectangle on which the partition begins is called the *root slice*, and slices on which no further partitioning is performed are called *leaf cells*. The *level* of a slice is the distance of the slice from its furthest descendent. The level of a leaf cell is zero, the level of a slice is the largest level of its child slices plus one. See Figure 1.7a.



A slicing structure of depth 3

Figure 1.7a

One natural representation of a slicing structure is the *slicing tree*. It is a rooted ordered tree. Each node of the tree corresponds to a slice in the slicing structure, and the children of the node correspond to children of the slice. The root node of the slicing tree corresponds to the root slice, and leaf nodes correspond to leaf cells. The depth of the tree corresponds to the depth of the slicing structure. The level of a node is the level of the corresponding slice. See Figure 1.7b.

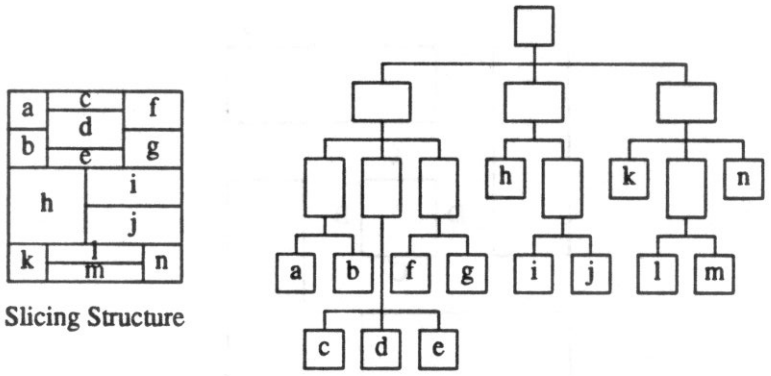


Figure 1.7b

A *slicing layout* is a layout in which rectangles (circuit components) are placed according to a slicing structures. A slicing layout is described by a slicing structure, i.e. each leaf cell of the slicing structure is an imaginary bounding box of a rectangle of the layout. From here on we use slicing structure and slicing layout interchangeably, although slicing structure sometimes refers to the topology of the slicing layout. This should be clear from the context. Each cut line represents a routing channel. A slicing layout in which all routing channels (cut lines) are river routable channels is called a *river slicing layout*.

During the detailed routing phase of a layout process, the channels have to be routed in a certain order. For example, in a T intersection, the stem must be routed first in order to determine the order of nets

entering the horizontal channel. See Figure 1.8.

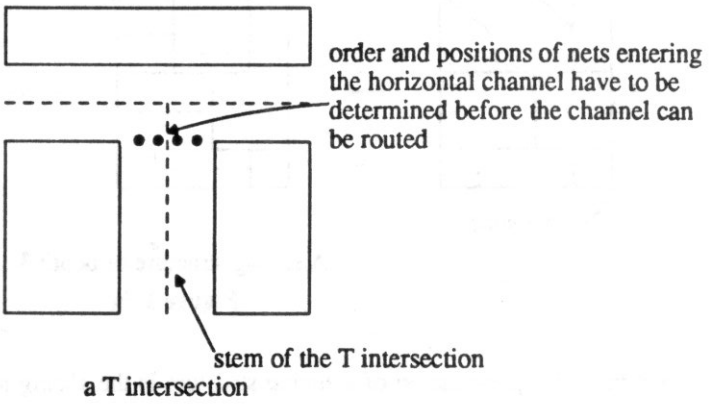


Figure 1.8

Sometime, cycles in the channel routing order may be resulted. See Figure 1.9 for an example of a cycle in the order of channels. In the example, channel A must be routed before B, B before C, C before D, and D before A; hence a cycle.

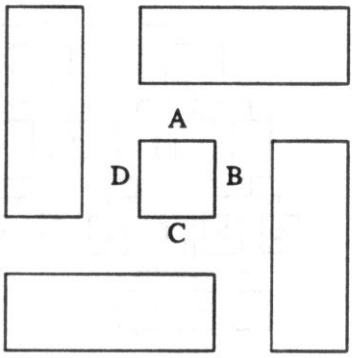


Figure 1.9

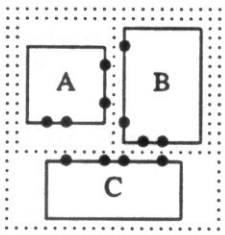
A slicing layout has two major advantages over a non-slicing one. First, it is free of cycles in the channel routing order [SuSl]. This implies a cycle breaking scheme can be avoided during the detailed routing and no time-consuming iteration of re-routing is required. Secondly, a slicing layout has an inherent hierarchy which is needed for a VLSI layout with a large number of circuit elements.

1.4 Layout Tools which use the Slicing Paradigm

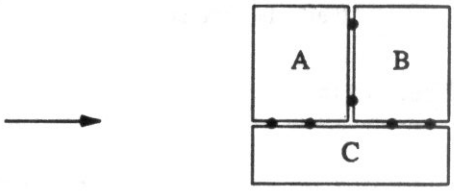
In this section we describe layout tools which use the slicing paradigm and their inadequacies. These inadequacies motivate our work on a new two dimensional compaction scheme.

1.4.1 Allende: a layout language

Allende [Mat] is a layout language which uses the slicing paradigm. It describes a layout structure by specifying relative position *(left, above, right, below)* of layout components called *cells*, for example (A left B) above C, see Figure 1.10a.



(A left B) above C
Figure 1.10a



Pitch aligning cell composition scheme
Figure 1.10b

A larger cell is created by composing smaller cells. Terminals on the opposite sides of a common cut line are required to have the same net order. The composition of cells is done by *pitch aligning* in which terminals of adjacent cells are aligned and cells are abutted. Cell stretching is very often necessary for the alignment to occur. See Figure 1.10b. In order to apply the pitch aligning cell composition scheme, each cell is assumed to be *stretchable*. In a *stretchable component*, the minimum displacement constraints are the only constraints between terminals. There is a minimum constraint between two adjacent terminals on the same boundary; there may or may not be minimum displacement constraints between terminals on opposite boundaries. When there is a constraint between two terminals on the opposite boundaries, the movement of one terminal on one boundary due to cell stretching will affect the terminal on the opposite boundary, as illustrated in Figure 1.11a. When there is no constraint between two terminals on the opposite boundaries, the movement of one terminal on one boundary will not affect the terminal on the opposite boundary as illustrated in Figure 1.11b.

Allende allows a compact and relatively simple description of a layout and the layout produced is guaranteed to free from design rule violation. One drawback of its simple cell composition scheme is that undesirable stretching of cells occurs and this stretching may propagate, e.g. Figure 1.12.

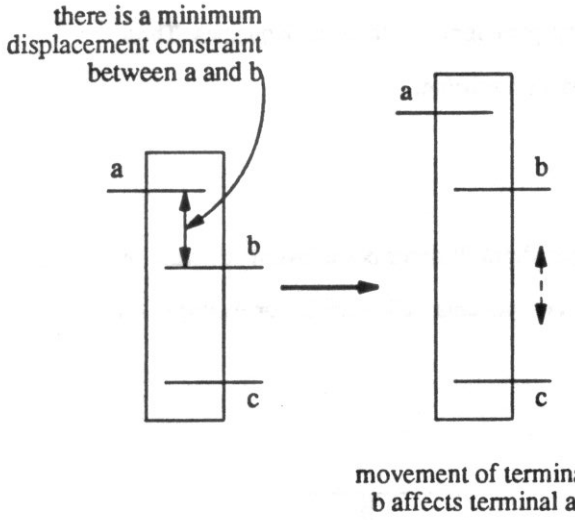


Figure 1.11a

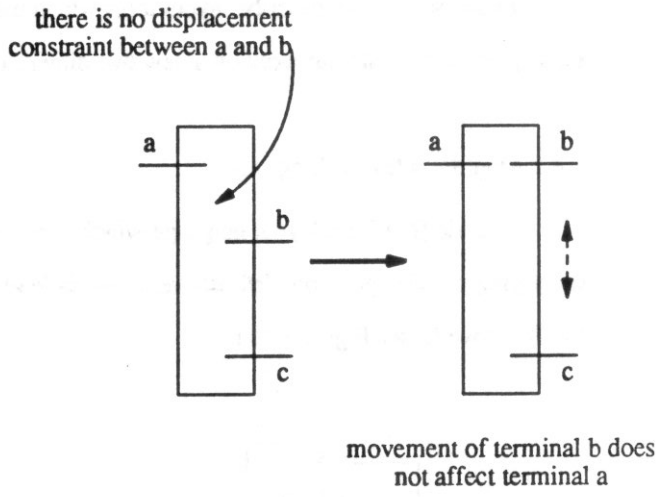


Figure 1.11b

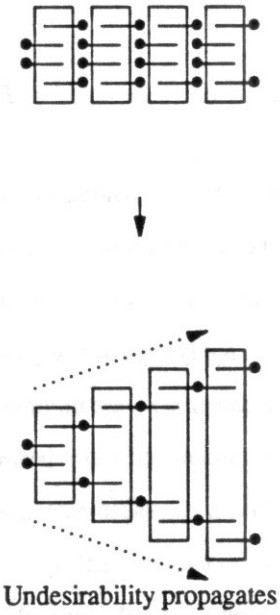
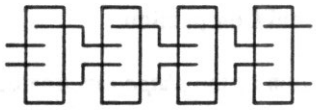


Figure 1.12

One remedy is to introduce single layer routing in an optimal fashion between adjacent cells, e.g. Figure 1.13.

LAVA and HILL [Eic, LeMe] are two other layout languages that use pitch aligning as one of their cell composition schemes.



Single layer routing
avoids cell stretching

Figure 1.13

1.4.2 Floor-planning Tools

Many floor-planning algorithms use the slicing paradigm [Lau, LaPo, Ott, SzOt, WoLi]. They partition a circuit into smaller subcircuits using a certain cost criterion, typically the min-cut criterion in which the number of nets crossing the partitioned boundary is minimized. Each subcircuit is further partitioned by the same cost criterion until the subcircuits are leaf cells. After the partitioning process, a slicing description of the circuit is obtained and the global routing is performed to determine the path of each routing wire. None of the authors in [Lau, LaPo, Ott, SzOt, WoLi] addresses the issue of the detailed routing.

After all leaf cells are designed, the dimensions of the leaf cells and absolute positions of terminals on the boundary of the leaf cells are known. The width of a routing channel is determined by the wires that pass through the channel and the wires that connect leaf cells adjacent to the channel. Relative positions of leaf cells adjacent to a routing channel can have a major effect on the width of the channel, as illustrated by an example in Figure 1.14.

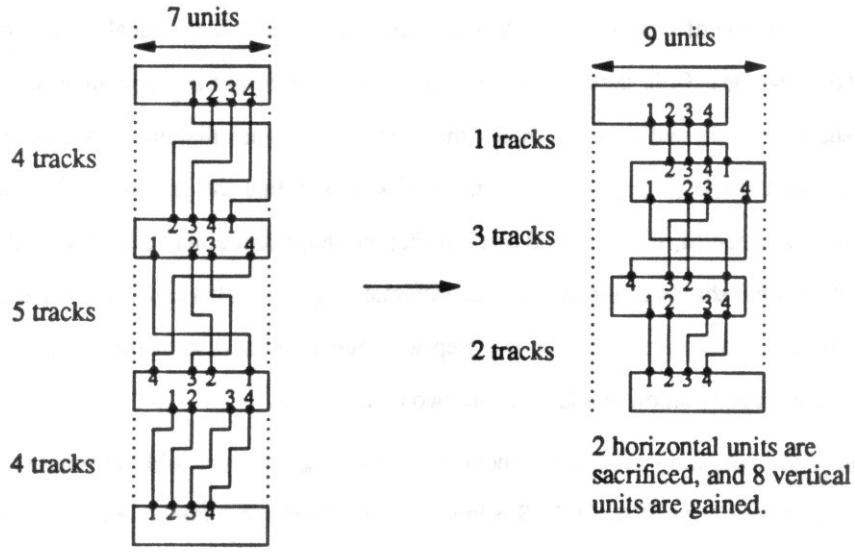


Figure 1.14

1.5 Shape Function and Two-Dimensional Compaction

In this section we describe the notion of the *shape function* of a layout and some previous work in computing the *shape function* of a slicing layout.

Given a topology of a layout, i.e. a relative placement of rectangles and the global routing of the layout, we would like to find the smallest area bounding box that *accommodates* the layout, in other words, the smallest shape at which the layout can be realized. Another scenario is, given a bounding box with a fixed horizontal dimension, find the minimum vertical dimension of the bounding box that will accommodate the layout, or vice versa.

The *horizontal shape function* of a layout is a function that gives the minimum vertical dimension of the layout for any given horizontal dimension, and the *vertical shape function* is a function that gives the minimum horizontal dimension of the layout for any given vertical dimension. The horizontal shape function of a layout is a monotonically non-increasing function in the horizontal dimension, this is because a bounding box with horizontal dimension h can accommodate a layout in vertical dimension v, then a bounding box with horizontal dimension $h' > h$ can also accommodate the layout with vertical dimension v. Here we assume the bounding box does not have to be tight, i.e. there may be white space between the actual layout and the bounding box. Similarly, the vertical shape function of a layout is monotonically non-increasing in the vertical dimension.

By the *shape function* of a layout we mean either the horizontal or the vertical shape function of the layout, and it should be clear from the context. A shape (h, v) of a layout is a bounding box with dimensions $h \times v$. The shape (h, v) is said to be minimal if there is no $h' < h$ such that the layout can be realized in the bounding box with shape (h', v), and there is no $v' < v$ such that the layout can be realized in the bounding box with the shape (h, v'). It should be clear that the shape function of a layout is fully characterized by the set of minimal shapes of the layout. The minimal shapes are also called the *break points* of the shape function. Notice that a shape function is a step-wise function because of the integral break points. See Figure 1.15 for a shape function of a layout with two components.

The problem of computing the shape function for a slicing structure without interconnections has been considered in the literature [Ott, Sto]. Stockmeyer considered the *optimal orientation problem* of a slicing structure [Sto]. In the optimal orientation problem, each leaf cell has a fixed size and free orientation. And each routing channel has zero width (no interconnections). The problem is to choose one of the two orientations (horizontal or vertical) for each leaf cell so that the area of the root slice is minimized. A more general problem is considered by Otten in [Ott]. In [Ott], each leaf cell has a set of shapes and fixed orientation. The problem is to compute the set of minimal shapes (shape function) of the root slice. The

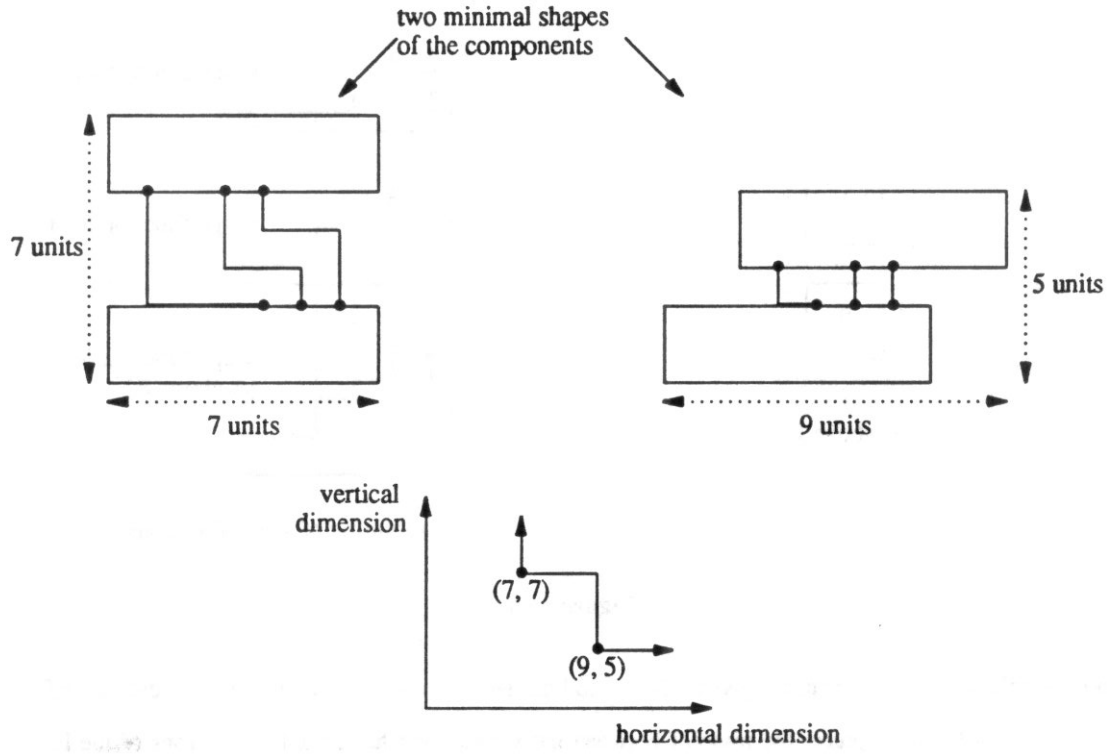


Figure 1.15

optimal orientation problem is a special case of this problem: each leaf cell has two shapes (one for its horizontal orientation and one for its vertical orientation) and the minimal shape of the root slice with the smallest area is the solution.

There is an efficient method to compute the shape function of a parent slice given shape functions of its child slices. It is the summing of individual shape functions along their common dimension. The summing operation is illustrated in Figure 1.16. To compute the shape function of the root slice of a slicing structure, a postorder traversal is performed on the slicing tree, and the shape function of each tree node is computed by summing the shape functions of its children. This method is presented by Otten in [Ott]. The complexity of the algorithm is $O(n \cdot b \cdot d)$, where n is the number of leaf cells, b is the maximum number of break points in the shape functions of leaf cells and d is the depth of the slicing structure.

Luk et al [LSW] present an algorithm to compute the shape function of a slicing layout with a single multiple-terminal net, i.e. all terminals carry a single signal.

After the shape function of a slicing layout is computed, the compaction problem is to find the minimum-area shape among the set of minimal shapes of the layout. Another scenario is to find a shape

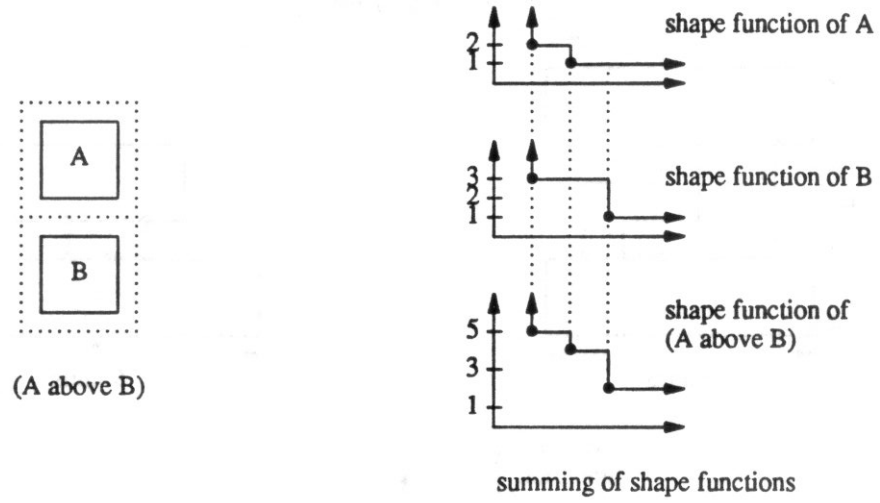


Figure 1.16

with the smallest vertical dimension given a horizontal dimension or vice versa. In essence, the shape function of a slicing layout captures the tradeoff between the vertical and horizontal dimensions of the layout. It represents truly two-dimensional compaction. This is different from one-dimensional compaction in which a layout is compacted in each dimension independently.

1.6 Summary and Organization of the Thesis

In Section 1.1 we gave an overview of constraint-based compaction and pointed out the need of jog point insertion to get a more compact layout. We assert that treating wires as topological entities in a layout is essential for better compaction, because it decouples the compaction and the detailed routing phase, and gives more interaction between the two phases.

In Section 1.2 and Section 1.3 we defined the layout model and the layout topology that we will be considering -- the grid model and the slicing layout topology. We described layout tools which use the slicing paradigm and their inadequacies in Section 1.4.

In Section 1.5 we introduced the notion of shape function that includes routing requirements. We mentioned previous work in computing the shape function of a slicing layout without interconnections and with limited interconnections.

In this thesis we consider a two-dimensional compaction scheme of a slicing layout. The compaction scheme treats wires as *topological entities*, i.e. only the paths of the wires that connect terminals of the same nets are known but not their physical locations. The path of a wire is assumed to be given (obtained from the global routing phase). Wires are not present physically in the layout during compaction, but they

are expressed as constraints which describe the actual space required to accommodate them. This amounts to computing the shape function that captures the routing requirements of the layout. For the rest of the thesis, by *shape function* we mean a shape function that includes the routing requirement unless specified otherwise.

We first study the two-dimensional compaction problem of the simplest slicing layout which is a stack of components. In Chapter 2, we consider the problem of computing the shape function of a stack of components under the restriction that each channel in the stack is river routable. A stack of single components under this restriction is a river slicing layout of depth one. This is a natural extension of single channel river routing problem. We present an algorithm to compute the shape function of a river slicing layout of depth one and a heuristic to approximate the shape function of a river slicing layout of depth two.

In Chapter 3, we present heuristics that use the algorithm and the heuristic of Chapter 2 as their basic subroutines to approximate the shape function of a more general river slicing layout. In a river slicing layout, terminals on opposite sides of a routing channel must have the same net ordering; this is the same requirement in pitch aligning. We compare the performance of the heuristics with pitch aligning by comparing the smallest area of layouts produced by each of the heuristics to the area of the layout produced by pitch aligning. This comparison shows the potential of our compaction scheme which treats wires as topological entities.

A *uniform layout* is a layout whose leaf cells are identical. In Chapter 4, we study the asymptotic behavior of river routing and pitch aligning in uniform stacks and uniform arrays. We derive the condition under which river routing composition is better than pitch aligning.

The channel routing problem under the rectilinear, two-reserved-layer routing model is known to be an NP-complete problem [Syz]. It is unlikely that the channel width can be computed efficiently. Instead of computing the channel width, the metric known as the *channel density* is often used to estimate the actual channel width. Intuitively, the channel density measures the minimum number of necessary net crossings at any grid line perpendicular to the channel. In general the channel density is a fairly good estimate for the channel width [Lei]. In fact Rivest and Fiduccia claimed that their channel routing algorithm usually uses no more than one track more than the channel density [RiFi].

In chapter 5, we discuss the problem of computing the shape function of a slicing layout using channel density as channel width. We show an NP-completeness result for computing such a shape function.

And we present a pseudo-polynomial time algorithm * to compute the shape function (using channel density as channel width) for a slicing layout of depth one. An efficient heuristic is proposed and the performance of the heuristic is compared against the result of the pseudo-polynomial time algorithm.

We summarize the approach of computing the shape function that captures the routing requirement of a slicing layout and address future research directions in chapter 6.

* Let I be an instance of a computational problem A -- typically I will be a sequence of combinatorial objects such as graphs, sets, or integers. In our case, I is positions of terminals, lengths of components and the number of components. Let MAX(I) be the largest integer appearing in I. An algorithm for A is *pseudo-polynomial* if the algorithm solves any instance of I of A in time bounded by a polynomial in $|I|$, the length of the encoding of I, and MAX(I) [PaSt].

Chapter 2

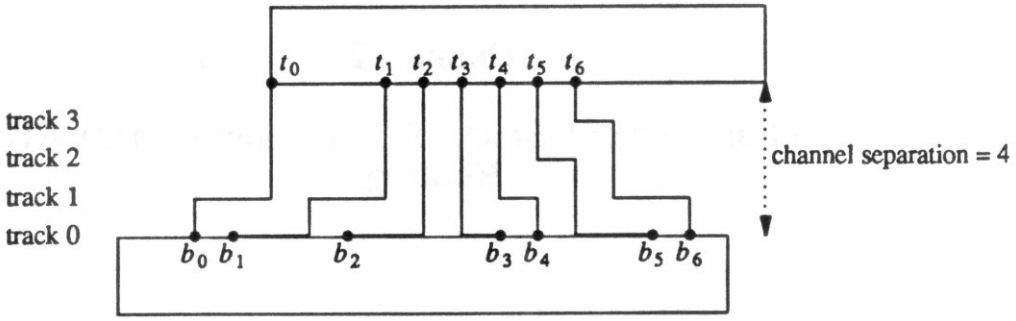
Shape Function of a Stack of Components under River Routing

In this chapter the single channel river routing problem is introduced. A necessary and sufficient condition for river routability is described. We extend the result of single channel river routing to a stack of river routable channels. An algorithm to compute the shape function of a stack of two-component channels is presented. A heuristic to compute the shape function of a stack of multiple component channels is discussed.

2.1 Single Channel River Routing

Single channel river routing is studied extensively in the literature, see in particular [DKSSU, Mir, Pin, SiDo]. We will give a brief review of the problem. The problem is stated as follows: given a channel that is defined by two rows of components, terminals $\{ t_0, t_1, \ldots, t_{m-1} \}$ are distributed along the bottom edges of the components on the top row; terminals $\{ b_0, b_1, \ldots, b_{m-1} \}$ are distributed on the top edges of the components on the bottom row. Terminals are allowed at the left ends of components but are not allowed at the right ends of components. t_i is to be connected to b_i by wire w_i. We are using the grid model in which wires run along grid lines. In a river routing channel, wires are not allowed to cross or overlap one another. *Tracks*, the grid lines parallel to the channel, are numbered from 0 to $t-1$, bottom to top, $t \geq 0$, and t is called the *channel separation* of the channel. For $t > 0$, top components are t grid units from the bottom components, and wires are extended vertically from track $t-1$ to connect corresponding terminals on the bottom edges of the top components, i.e. there are no horizontal pieces of wires on the bottom edges of the top components. However horizontal wire segments are allowed on the top edges to the bottom components (track 0). See Figure 2.1 for an instance of river routing.

The *optimal placement problem* at separation t in a river routable channel is to find an optimal placement of the components so that the *horizontal span* of the channel, i.e. the distance from the left edge of the leftmost component to the right edge of the rightmost component is minimized. The *minimum separation* of the channel is the minimum number of tracks required to route the channel. A channel separation is said to be *legal* if it is no less than the minimum separation. Notice that if we compute the minimum horizontal

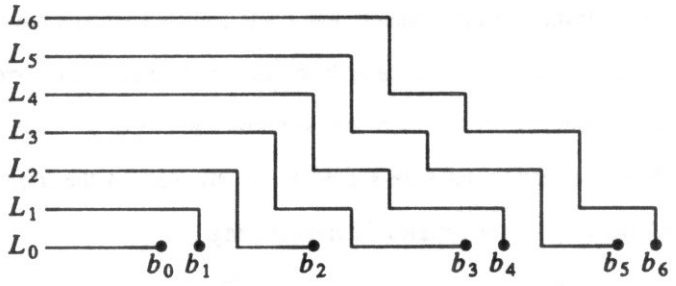


An instance of river routing

Figure 2.1

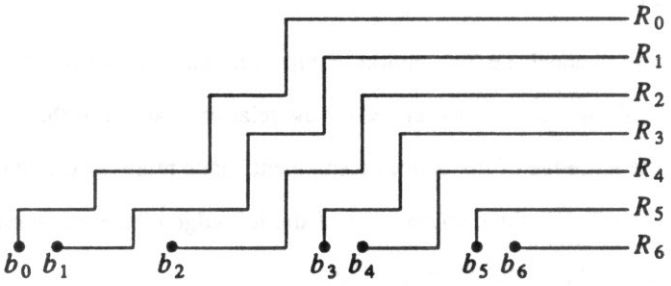
span for each legal separation, we get the shape function of the two rows of components. Another variation of the problem is to find the minimum legal separation of the channel. This can be done by a binary search on the channel separations.

All solutions to river routing problems are based on a necessary and sufficient routability condition of a river routable channel. We will proceed to give an informal derivation of the necessary and sufficient routability condition. See [Mir, Pin] for more formal derivations. We use here the notations and terminologies in [Mir]. Consider the *left base wires* $\{ L_0, L_1, \ldots, L_{m-1} \}$, emanating from the bottom terminals, as shown in Figure 2.2a. The left base wires are obtained by routing all bottom terminals to the left of the channel by using as few tracks as possible without violating the wiring model. This can be done by extending each of the m bottom terminals in turn, beginning from terminal b_0, to the left or upward if it is obstructed by a terminal or another wire. Similarly, the *right base wires* $\{ R_0, R_1, \ldots, R_{m-1} \}$ are obtained by routing the bottom terminals to the right of the channel optimally. See Figure 2.2b.



left base wires

Figure 2.2a



right base wires

Figure 2.2b

Let $ll_i(t)$ be the position on track t where the left base wire L_i intersects track t from below, and $rr_i(t)$ be the position on track t where the right base wire R_i intersects t from below. $ll_i(t)$ is b_{i-t} shifted right by t units and $rr_i(t)$ is b_{i+t} shifted left by t units, therefore

$$ll_i(t) = b_{i-t} + t \text{ and } rr_i(t) = b_{i+t} - t$$

At separation t, wire w_i that connects t_i to b_i is obtained by dropping a vertical wire segment from t_i at track t to the base wire L_i or R_i (depending on whether t_i is to the left or right of b_i). A necessary condition for w_i to be a valid wire (run along grid lines, no crossing or overlapping with other wires) is that t_i is to the right of $ll_i(t)$ and to the left of the rr_i, as shown in Figure 2.3.

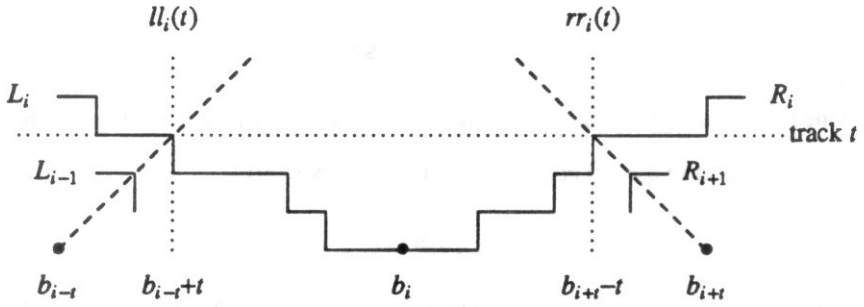


Figure 2.3

In fact,

$$b_{i-t} + t \le t_i \le b_{i+t} - t \qquad (2.1.1)$$

is a necessary and sufficient condition for w_i to be a valid wire at separation t [Mir, Pin].

2.1.1 Two-Component Channel

In a two-component channel, i.e. one bottom component and one top component, relative positions of top and bottom terminals are fully characterized by the relative position of the left edge of the top component with respect to the left edge of the bottom component, since relative positions of terminals within a component are fixed. An *offset* is the displacement of the left edge of the top component with respect to the left edge of the bottom component. See Figure 2.4.

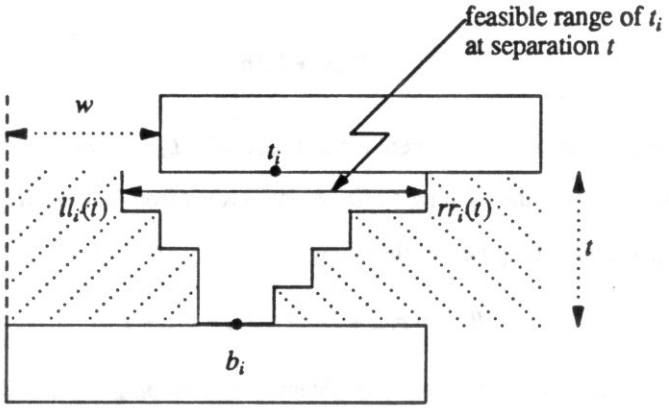


Figure 2.4

At offset w terminal t_i is at position $t_i + w$. At separation t, condition (2.1.1) describes a feasible range for top terminal t_i, i.e.

$$b_{i-t} + t \le t_i + w \le b_{i+t} - t$$

The feasible range of t_i also defines an offset range for the top component (i.e. range of the left edge of the top component with respect to the left edge of the bottom component)

$$b_{i-t} + t - t_i \le w \le b_{i+t} - t - t_i$$

The intersection of all the offset ranges of top components at separation t defined by each of the m terminals is the *feasible offset range* of the top component at separation t. The left endpoint of the feasible offset range at separation t is called the *left constraint* at separation t, $L(t)$, and the right end point of the feasible offset range at separation t is called the *right constraint* at separation t, $R(t)$:

$$L(t) = \max\{\, b_{i-t} + t - t_i \mid t \le i < m \,\}, 0 \le t < m; L(m) = -\infty$$

$$R(t) = \min\{\, b_{i+t} - t - t_i \mid 0 \le i < m-t \,\}, 0 \le t < m; R(m) = +\infty$$

Let dBT be the $m \times m$ matrix whose ij^{th} entry is $b_i - t_j$. Then $L(t) - t$ is the maximum of all elements on the sub-diagonal in dBT at distance t above the main diagonal, likewise $R(t) + t$ is the minimum of all

elements on the sub diagonal in dBT at distance t below the main diagonal. Therefore $L(t)$ and $R(t)$ can be computed by a diagonal sweep of the matrix $B-T$. This requires $O(m^2)$ time.

The geometrical interpretation of the left and right constraint at separation t is that, $L(t)$ is the leftmost position and $R(t)$ is the rightmost position of the top component with respect to the bottom component such that the routing can be realized in t tracks.

For a separation $t \geq m$, the feasible range of the top component is $(-\infty, +\infty)$, i.e. the routing can always be realized in m tracks. And for a channel with separation $t > m$, we can reduce the separation to m without affecting the feasibility of the channel. Therefore without loss of generality, we only consider channels whose separations are no greater than the number of nets in the channels.

$L(t)$ is non-increasing in t, since larger separation will allow the top component to move further to the left. Formally, observe that in the grid model

$$b_{i+1} \geq b_i + 1,$$

$$\therefore b_{i-t} + t - t_i \geq b_{i-(t+1)} + (t+1) - t_i, \text{ for } t < m$$

$$\therefore L(t) \geq L(t+1)$$

Similarly, $R(t)$ is non-decreasing in t, i.e. $R(t) \leq R(t+1)$.

The minimum separation is the smallest separation t which defines a feasible offset range for the top component. In terms of $L(t)$ and $R(t)$, it is the separation t such that,

$$L(t) \leq R(t)$$

and

$$L(t-1) > R(t-1)$$

Since $L(t)$ is non-increasing and $R(t)$ is non-decreasing, the minimum separation can be computed by a linear scan on $L(t)$ and $R(t)$.

Knowing how to compute the feasible offset range of the top component at separation t, we can compute the minimum horizontal span of the channel at separation t. There are four cases:

(1) $L(t) \leq 0 \leq R(t)$, i.e. 0 is a feasible offset. Aligning components on the left gives the smallest horizontal span.

(2) $L(t)+length_{top} \leq length_{bottom} \leq R(t)+length_{top}$, i.e. $length_{bottom}-length_{top}$ is a feasible offset. Aligning the components on the right gives the smallest horizontal span.

(3) $0 \le L(t) \le R(t)$ and not case (2), then the channel has smallest horizontal span at offset $L(t)$.

(4) $L(t) \le R(t) \le 0$ and not case (2), then the channel has smallest horizontal span at offset $R(t)$.

Therefore the shape function of a two-component channel can be obtained by computing the minimum horizontal span for each legal channel separation.

2.1.2 Multiple Component Channel

The *optimal placement problem* for a multiple component channel at separation t is to find a placement for each component at separation t such that the horizontal span of the channel is minimized. Leiserson and Pinter reduce this problem to a longest path problem by deriving constraints between components from condition (2.1.1) [LePi].

Consider a channel with k components and m terminals. The components are numbered from 1 to l on the bottom row, and from $l+1$ to k on the top row. The order of the components on the same row is fixed. The *placement graph* of such a channel is a weighted single-source single-sink directed graph that describes constraints on relative placement between components in the channel. The source v_0 corresponds to the left boundary of the channel, and the sink v_{k+1} corresponds to the right boundary of the channel. An internal vertex v_i corresponds to the left edge of component i. A vertex v_i is regarded as a variable whose value represents the horizontal position of the object to which it corresponds.

A constraint from v_j to v_i, $v_i - v_j \ge w_{ij}$ describes a relative placement between component i and component j. It corresponds to a directed edge from v_j to v_i with weight w_{ij}. There are two types of constraints in a placement graph: constraints between vertices on the same row, and constraints between vertices on different rows. A constraint for two adjacent vertices v_i and v_{i+1} on the same row is $v_{i+1} - v_i \ge l_i$ where l_i is the length of component i; there are k such constraints. A constraint between v_i and v_j on different rows is established by constraints (condition (2.1.1)) between terminals on component i and component j. Since relative positions of terminals within a component are fixed, the constraint between a top terminal t_i and a bottom terminal b_i can be translated into a constraint between left edges of components that contain the terminals. So a constraint from v_i to v_j is a maximal constraint between terminals on component i and component j. This set of constraints can be established in $O(m)$ time. In addition it can be easily seen that if v_i and v_j are two vertices on opposite rows, $v_{i'}$ on the same row as v_i and $i < i'$, $v_{j'}$ on the same row as v_j and $j < j'$, and $v_{j'} - v_i$ is a constraint in the set of constraints, then $v_j - v_{i'}$ is not in the set of constraints. Therefore the number of edges from the bottom row to the top row is at most $2k$, and the number of edges from the top row to the bottom is at most $2k$. Thus the total number of edges in the placement graph is $O(k)$. Figure 2.5 shows a 5 component channel and the placement graph at separation 2. In the example,

the constraint from v_3 to v_1 is the maximal constraint of $b_2 - t_0 \geq 2$ (equivalently $v_1 - v_3 \geq 0$), and $b_3 - t_1 \geq 2$ (equivalently $v_1 - v_3 \geq 1$), i.e. $v_1 - v_3 \geq 1$.

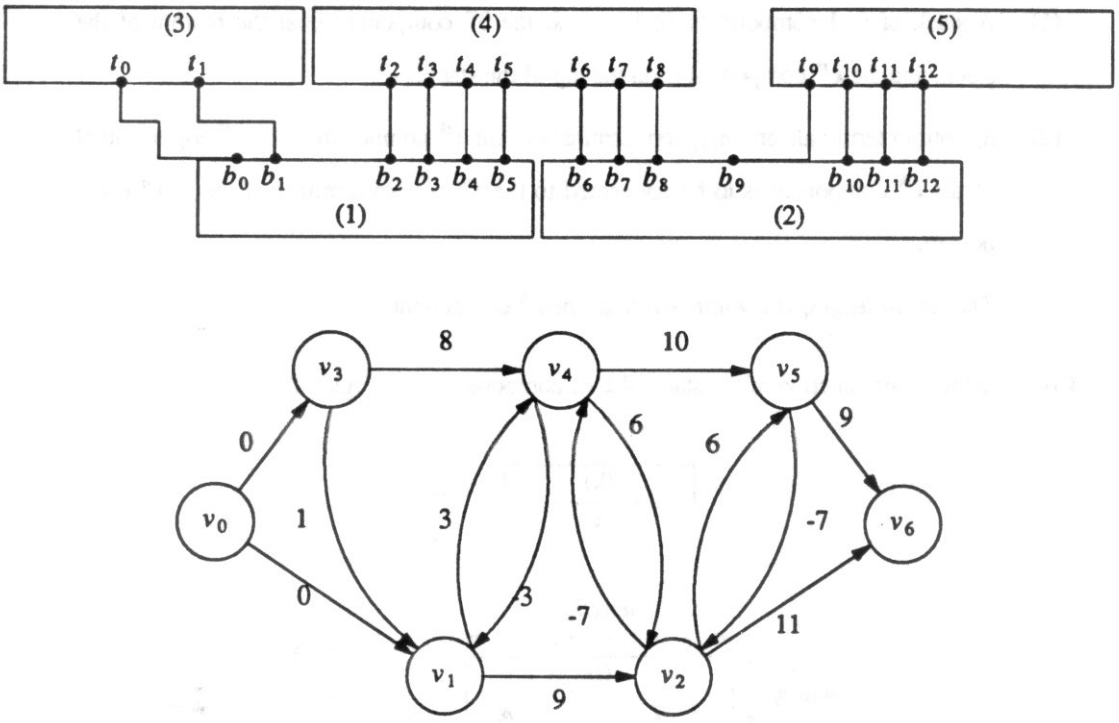


Figure 2.5

Leiserson and Pinter find a partial ordering of the edges in the placement graph [LePi]. The partial order leads to an algorithm which computes the longest path of the placement graph in linear time in the number of edges in the graph. The shape function of a multiple component channel can then be computed by solving the optimal placement problem for each legal channel separation.

In the next two sections we consider the river routing problem of a stack of components; we compute the shape function of the stack.

2.2 Shape Function of a Stack of Single Component

In this section we compute the shape function of a stack of single components in which each channel, defined by two adjacent components in the stack, is river routable. The *shape-of-stack problem* is stated as follows:

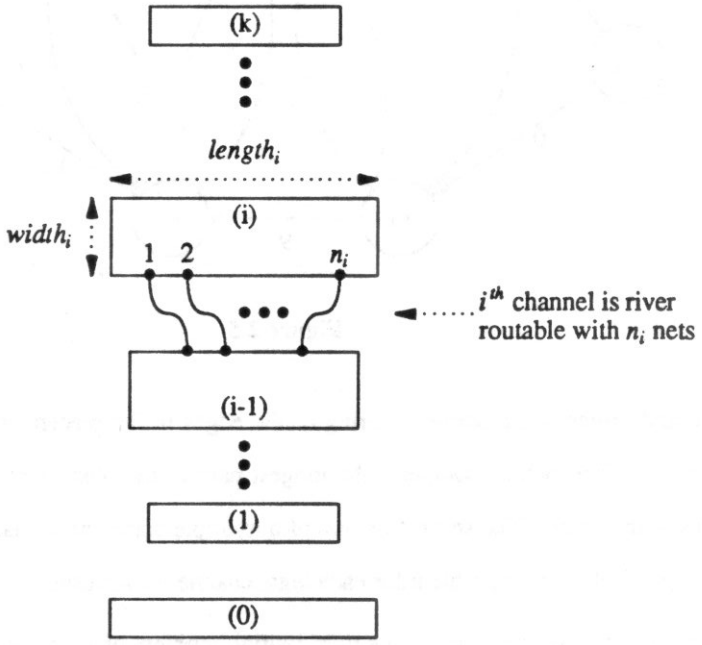
Given,

(1) A stack of $k+1$ components, 0, 1, . . . , k, the 0^{th} component is at the bottom of the stack, and the k^{th} component is on the top of the stack.

(2) n_i bottom terminals and n_{i+1} top terminals of the i^{th} component. The j^{th} top terminal of the i^{th} component is to be connected to the j^{th} bottom terminal of the $i+1^{st}$ component.

(3) The length $length_i$ and width $width_i$ of the i^{th} component.

Compute the shape function of the stack of $k+1$ components. See Figure 2.6.



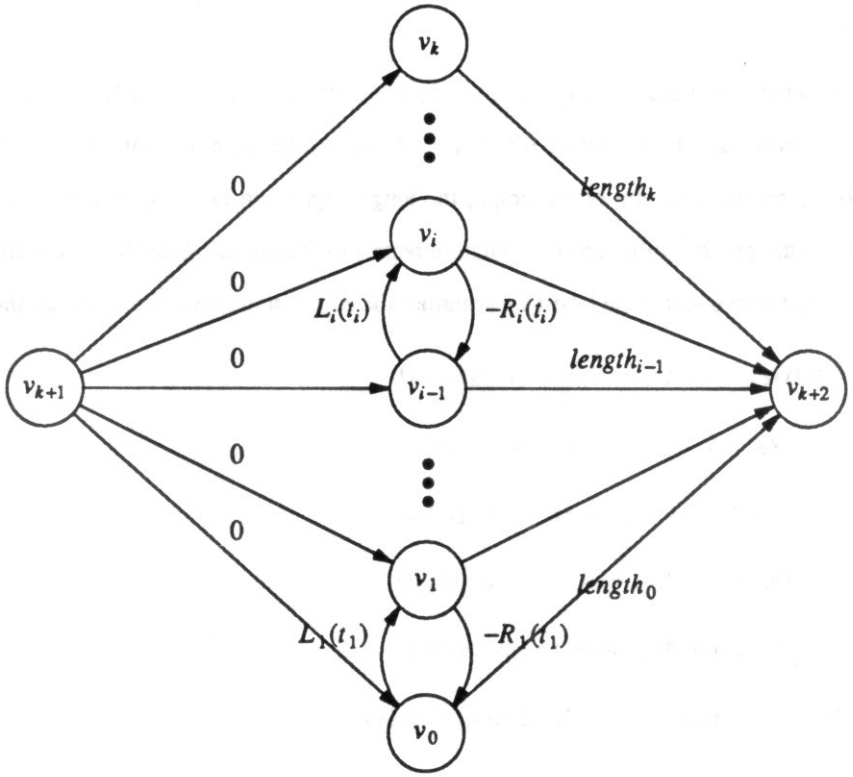
an instance of the shape-of-stack problem with $k+1$ components

Figure 2.6

We will first define some of the terms that we will be using. Recall the left and right constraints of a two-component channel. They describe the necessary and sufficient routability condition of a channel, i.e. a channel is routable at separation t if and only if the offset of the top component with respect to the bottom component is greater than or equal to the left constraint $L(t)$ and less than or equal to the right constraint $R(t)$. In a stack of components, let $L_i(t)$ and $R_i(t)$ denote the left and right constraints of the i^{th} channel respectively. A *separation vector* $(t_1, t_2, \ldots, t_k)$ for a stack of $k+1$ components describes channel

separations of the k channels of the stack. A separation vector $(t_1, t_2, \ldots, t_k)$ is said to be *legal* if all channel separations are legal, i.e. $L_i(t_i) \leq R_i(t_i)$ for i from 1 to k.

For a given separation vector $(t_1, t_2, \ldots, t_k)$, the *placement graph* of a stack of $k+1$ components is a weighted directed graph which represents constraints between adjacent components. There are $k+3$ vertices; $v_0, v_1, \cdots, v_k$ correspond to the left edge of each component in the stack, v_{k+1} and v_{k+2} correspond to the left and the right boundaries of the stack respectively. A vertex v_i is also regarded as a variable whose value represents the horizontal position of the object to which it corresponds. The weight of the directed edge (v_{i-1}, v_i) is the left constraint of the i^{th} channel, i.e. $weight(v_{i-1}, v_i) = L_i(t_i)$, the weight of the directed edge (v_i, v_{i-1}) is the negative right constraint of the i^{th} channel, i.e. $weight(v_i, v_{i-1}) = -R_i(t_i)$, $weight(v_{k+1}, v_i) = 0$, and $weight(v_i, v_{k+2}) = length_i$. See Figure 2.7.



placement graph for a stack of $k+1$
components with the separation vector
$(t_1, t_2, \ldots, t_k)$

Figure 2.7

A *configuration* of a stack for the separation vector $(t_1, t_2, \ldots, t_k)$ is a placement of the $k+1$ components; the i^{th} component at position v_i, and the i^{th} channel with separation t_i. A *legal configuration* is a configuration in which relative offsets of all adjacent components do not violate the left and right

constraints between them, i.e. for a $i-1^{st}$ and i^{th} components pair,

$$L_i(t_i) \le v_i - v_{i-1} \le R_i(t_i)$$

Without loss of generality, we assume each channel separation in a legal configuration does not exceed the number of nets in the channel. For the rest of the chapter, when we refer to a legal configuration, we mean a legal configuration with the above assumption.

In terms of the placement graph, a legal configuration is an assignment of values to the nodes in the graph subjects to the set of constraints. An *optimal configuration* for a separation vector is a legal configuration of the vector with minimum horizontal dimension. The optimal configuration can be constructed by finding an assignment of values to the nodes on the placement graph subjects to the set of constraints, such that $v_{k+2} - v_{k+1}$ is minimized. It is equivalent to solving the longest path problem on the placement graph.

For a legal separation vector $(t_1, t_2, \ldots, t_k)$, $L_i(t_i) \le R_i(t_i)$ for i from 1 to k; therefore, there is no non-negative cycle in the graph. Therefore we only have to consider edge disjoint longest paths from v_{k+1} to v_{k+2}. Due to the special structure of the graph, the longest path can be computed in time linear in the number of edges in the graph [LePi]. The computation begins by initializing *label* (v_{k+1}) to zero and all others *label* (v_i) to $-\infty$, it then updates the labels by scanning the edges in the order specified by the edge list ξ,

$$\begin{aligned}\xi = \{&(v_{k+1}, v_0), (v_{k+1}, v_1), \ldots, (v_{k+1}, v_k),\\ &(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k),\\ &(v_k, v_{k-1}), (v_{k-1}, v_{k-2}), \ldots, (v_1, v_0),\\ &(v_0, v_{k+2}), (v_1, v_{k+2}), \ldots, (v_k, v_{k+2})\}\end{aligned}$$

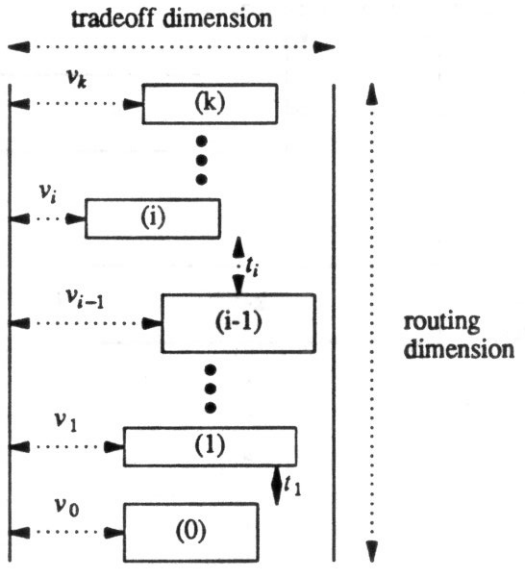
For each edge (v_i, v_j) in ξ it updates *label* (v_j) as follows,

$$label(v_j) = \max\{label(v_j), label(v_i) + weight(v_i, v_j)\}$$

The correctness of the computation is due to the fact that *each edge disjoint path from* v_{k+1} *to* v_{k+2} *is a subsequence of* ξ [Pinter, Yen].

The *routing dimension* of a configuration is the dimension that is orthogonal to the routing channels and the *tradeoff dimension* of a configuration is the dimension that is parallel to the routing channels. The *total separation* of a configuration is the total of all channel separations. Because the total of all component widths is constant in a given stack, the routing dimension varies with the total separation. From here on, we shall ignore the constant total widths of the stack and use the total separation as the routing dimension of

the stack. See Figure 2.8.



A configuration for the separation vector $(t_1, t_2, \ldots, t_k)$

Figure 2.8

We have defined the legal configuration of a stack of components for a given separation vector and have described a method to compute the minimum tradeoff dimension of the stack. We now consider sets of legal configurations that have the same total separations. First we define the *minimal shape* of a stack of components and describe how it can be computed. A shape $(s, t)_k$ describes a set of legal configurations, whose tradeoff dimension does not exceed s and whose routing dimension (total separation) does not exceed t, of a stack of $k+1$ components. A position w_k of the k^{th} component (top) is the position of the top component relative to the left boundary of the stack from which s is measured. A 3-tuple $(s, t, w_k)_k$ describes a set of legal configurations in $(s, t)_k$ whose top components are at position w_k. We say $(s, t, w_k)_k$ is *legal* if it contains at least one legal configuration. The legality of $(s, t, w_k)_k$ can be defined in terms of the legality of $(s, t-q, w_{k-1})_{k-1}$ as follows,

$(s, t, w_k)_k$ is legal if and only if for some w_{k-1} and q :

(1) $s \le w_k + length_k$, i.e. the k^{th} component is within the tradeoff dimension s.

(2) $(s, t-q, w_{k-1})_{k-1}$ is legal.

(3) $L_k(q) \le w_k - w_{k-1} \le R_k(q)$, i.e. placement of the k^{th} component with respect to the $k-1^{st}$ component does not violate the left and right constraints. See Figure 2.9.

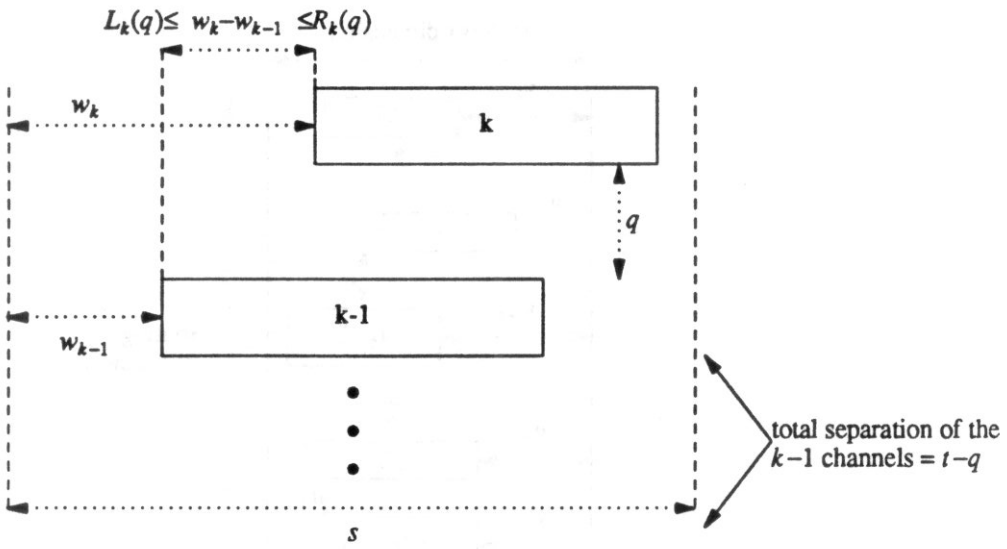


Figure 2.9

We say $(s, t)_k$ is *minimal* for a given tradeoff dimension s if

(1) There is some w_k such that $(s, t, w_k)_k$ is legal.

(2) There is no w_k such that $(s, t-1, w_k)_k$ is legal.

Likewise, we say $(s, t)_k$ is minimal for a given routing dimension t if

(1) There is some w_k such that $(s, t, w_k)_k$ is legal.

(2) There is no w_k such that $(s-1, t, w_k)_k$ is legal.

If $(s, t)_k$ is minimal in both s and t, then it is said to be a *minimal shape*. A minimal shape $(s, t)_k$ of a stack corresponds to a break point of the shape function of the stack.

2.2.1 The Boundary Functions

Recall that in Chapter 1, the shape function of a layout is characterized by the set of minimal shapes of the layout. Similarly, the shape function of a stack of $k+1$ components is fully described by the set of minimal shapes of the stack. Computing the shape function of a stack of $k+1$ components amounts to computing the set of minimal shapes of the stack.

First we will show how to compute the minimal routing dimension of a stack of $k+1$ components for a given tradeoff dimension s. To do so, we compute the *left boundary function* $l_k(s, t)$ and the *right boundary function* $r_k(s, t)$. For a non-empty set $(s, t)_k$ of legal configurations, $l_k(s, t)$ describes the leftmost position of the k^{th} component and $r_k(s, t)$ describes the rightmost position of the k^{th} component of all the configurations in $(s, t)_k$. If s is less than $\max_{0 \le i \le k}\{ length_i \}$, then $(s, t)_k$ is empty. In the following definitions we will assume s is no less than $\max_{0 \le i \le k}\{ length_i \}$. Let,

n_i denote the number of nets in the i^{th} channel,

$N_i = \sum_{j=1}^{j=i} n_j, N_0=0.$

$L_i(q)$ for $q = 0$ *to* n_i denote the left constraint of the i^{th} channel.

$R_i(q)$ for $q = 0$ *to* n_i denote the right constraint of the i^{th} channel.

$Tmin_i$ denote the minimum channel separation of the i^{th} channel.

$tmin_0(s) = 0$

$l_0(s, 0) = 0; r_0(s, 0) = s - length_0$

$range_i(s)$ denote the range $[tmin_{i-1}(s)+Tmin_i, N_i]$. $range_0(s)=[0,0]$.

For $t \in range_i(s)$,

$lowq_i(s, t) = \max \{ t-N_{i-1}, Tmin_i \}$

$highq_i(s, t) = \min \{ n_i, t-tmin_{i-1}(s) \}$

Since $L_i(n_i) = -\infty$ and $R_i(n_i) = +\infty$, we will need to add $-\infty$ and $+\infty$ to integers; we never need $(+\infty) + (-\infty)$. For a finite number x, we define $x+(-\infty) = -\infty$, and $x+(+\infty) = +\infty$.

We will now give the inductive definition of the boundary functions. For a stack of $i+1$ components the boundary functions $l_i(s, t)$ and $r_i(s, t)$ of the tradeoff dimension s are defined as follows:

For $t \in range_i(s)$,

$l'_i(s, t) = \min \{ l_{i-1}(s, t-q) + L_i(q) \mid lowq_i(s, t) \le q \le highq_i(s, t) \}$

$r'_i(s, t) = \max \{ r_{i-1}(s, t-q) + R_i(q) \mid lowq_i(s, t) \le q \le highq_i(s, t) \}$

$l_i(s, t) = \max \{0, l'_i(s, t)\}$

$r_i(s, t) = \min \{ s-length_i, r'_i(s, t) \}$

$tmin_i(s)$ = smallest t in $range_i(s)$ such that $l_i(s, t) \le r_i(s, t)$.

The following lemma shows that the boundary functions are well-defined.

Lemma 2.1 For a stack of $i+1$ components and for t in $range_i(s)$

(1) $l_i(s, t)$ and $r_i(s, t)$ are well-defined.

(2) $l_i(s, t)$ is non-increasing, and $r_i(s, t)$ is non-decreasing in t and $l_i(s,N_i) = 0$, $r_i(s,N_i) = s\text{-}length_i$

(3) $tmin_i(s)$ exists

Proof. We shall prove the lemma by induction on i. The basis, $i = 0$, is trivially satisfied: $range_0(s) = [0,0]$, $l_0(s, 0) = 0, r_0(s, 0) = s\text{-}length_0$, $tmin_0(s) = 0$. From the induction hypothesis, $tmin_{i-1}(s)$ exists and $tmin_{i-1}(s)+Tmin_i \le N_{i-1}+n_i \le N_i$, therefore $range_i(s)$ is a legitimate range. To show that $l_i(s, t)$ is well-defined, consider $l'_i(s, t)$:

$$l'_i(s, t) = \min \{l_{i-1}(s, t-q)+L_i(q) \mid lowq_i(s, t) \le q \le highq_i(s, t)\}$$

For $t \in range_i(s)$,

(1) $Tmin_i \le n_i$

(2) $t-N_{i-1} \le N_i-N_{i-1} = n_i$

(3) $Tmin_i \le t-tmin_{i-1}(s)$

(4) $t-N_{i-1} \le t-tmin_{i-1}(s)$

(1) is true because minimum channel separation is no larger than the number of nets in the channel, (2) is true because N_i is the largest t in $range_i(s)$, (3) is true because $Tmin_i + tmin_{i-1}(s)$ is the smallest t in $range_i(s)$, and (4) is true because $tmin_{i-1}(s)$ is no larger than N_{i-1}. Thus $lowq_i(s, t) \le highq_i(s, t)$. In addition for $q \in [lowq_i(s, t), highq_i(s, t)]$,

$$Tmin_i \le q \le n_i$$

Therefore $L_i(q)$ is defined. In addition,

$$t-N_{i-1} \le q \le t-tmin_{i-1}(s)$$

therefore,

$$tmin_{i-1}(s) \le t-q \le N_{i-1}$$

Therefore $l_{i-1}(s, t-q)$ is well-defined by the induction hypothesis. Hence, $l'_i(s, t)$ exists and has a unique value. Thus $l_i(s, t)$ is well-defined for t in $range_i(s)$. Similarly $r_i(s, t)$ is well-defined.

Now we show that $l_{i-1}(s, t)$ is non-increasing. Let q be the channel separation that defines $l'_i(s, t)$. Then,

$$l'_i(s, t) = l_{i-1}(s, t-q) + L_i(q)$$

If $t+1$ is in $range_i(s)$, then $t+1 \le N_i$. There are two cases:

Case 1: $q < n_i$.

Then $q+1 \le n_i$. And $L_i(q) \ge L_i(q+1)$, since $L_i(q)$ is a non-increasing function. Therefore,

$$l'_i(s, t) = l_i(s, t-q) + L_i(q) \ge l_i(s, t+1-(q+1)) + L_i(q+1) \ge l'_i(s, t+1)$$

Case 2: $q = n_i$.

Then $t+1-q \le N_i-q = N_i-n_i = N_{i-1}$. And, from $q \le highq_i(s, t))$, $tmin_{i-1}(s) \le t-q \le t+1-q$. So, $l_{i-1}(s, t-q) \ge l_{i-1}(s, t+1-q)$, since $l_{i-1}(s, t)$ is a non-increasing function by the induction hypothesis. In addition $n_i \in [lowq_i(s, t), highq_i(s, t)]$. Therefore,

$$l'_i(s, t) = l_i(s, t-q) + L_i(q) \ge l_i(s, t+1-q) + L_i(q) \ge l'_i(s, t+1)$$

If $l'_i(s, t) < 0$, then $l_i(s, t) = 0$, and if $l'_i(s, t) \ge 0$, then $l_i(s, t) = l'_i(s, t)$. Therefore $l_i(s, t) \ge l_i(s, t+1)$ for t and $t+1$ in $range_i(s)$. Similarly, $r_i(s, t) \ge r_i(s, t+1)$ for t and $t+1$ in $range_i(s)$. Since,

$$l'_i(s,N_i) \le l_{i-1}(s,N_{i-1}) + L_i(n_i) = 0 + (-\infty)$$

$l_i(s,N_i) = 0$. Since,

$$r'_i(s,N_i) \ge r_{i-1}(s,N_{i-1}) + R_i(n_i) = s\text{-}length_{i-1} + \infty = +\infty$$

$r_i(s,N_i) = s\text{-}length_i$. Now we show the existence of $tmin_i(s)$. We have $l_i(s,N_i) \le r_i(s,N_i)$. In addition, $l_i(s, t)$ is non-increasing and $r_i(s, t)$ is non-decreasing in $range_i(s)$, there is a smallest t in $range_i(s)$ such that $l_i(s, t) \le r_i(s, t)$. □

Corollary 2.1 For t in $range_i(s)$ and $q \in [lowq_i(s, t), highq_i(s, t)]$,

$$t-q \in range_{i-1}(s),$$

and

$$l_{i-1}(s, t-q) \le r_{i-1}(s, t-q)$$

Proof. $q \in [lowq_i(s, t), highq_i(s, t)]$ implies $tmin_{i-1}(s) \le t-q \le N_{i-1}$, and the result follows from the definition of $tmin_{i-1}(s)$. □

We shall show that the left boundary function $l_i(s, t)$ and the right boundary function $r_i(s, t)$ of a stack of $i+1$ components *capture* the shape $(s, t)_i$, i.e. for any w_i such that $l_i(s, t) \le w_i \le r_i(s, t)$, we can construct a legal configuration in $(s, t, w_i)_i$, and for $l_i(s, t) > r_i(s, t)$ there is no w_i such that $(s, t, w_i)_i$ is legal. See Figure 2.10.

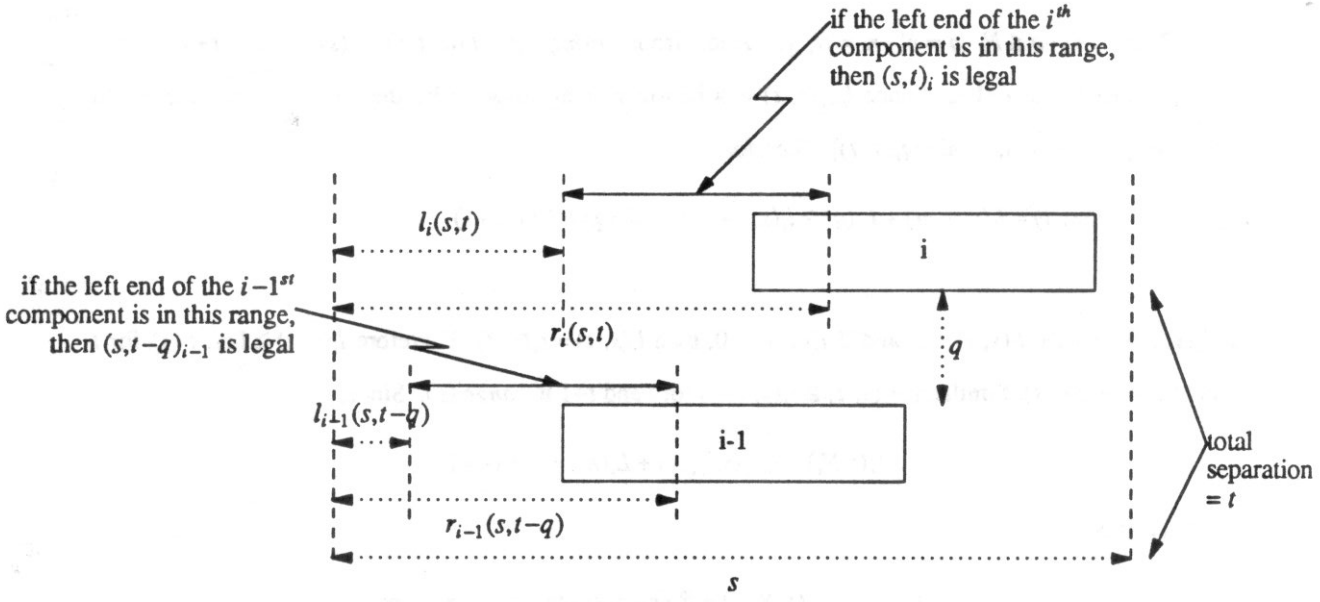


Figure 2.10

Lemma 2.2 For t in $range_i(s)$ $(s, t, w_i)_i$ is legal if and only if $l_i(s, t) \le w_i \le r_i(s, t)$.

Proof. We shall prove the lemma by induction on i. The basis, $i = 0$, is trivially satisfied, since $l_0(s, 0) = 0$ and $r_0(s, 0) = s\text{-}length_0$, and $(s, 0, w_0)_0$ is legal if and only if $0 \le w_0 \le s\text{-}length_0$.

Assume $(s, t, w_i)_i$ is legal. Let, w_{i-1} be the the position of the $i-1^{st}$ component of a legal configuration in $(s, t, w_i)_i$, and q be the channel separation of the i^{th} channel. Without loss of generality we assume $q \le n_i$, and $t-q \le N_{i-1}$. See the definition of a legal configuration. Then,

$$L_i(q) \le w_i - w_{i-1} \tag{2.1}$$

And, $l_{i-1}(s, t-q) \le w_{i-1} \le r_{i-1}(s, t-q)$ by the induction hypothesis. Therefore $tmin_{i-1}(s) \le t-q$ and so $q \in [lowq_i(s, t), highq_i(s, t)]$. Thus,

$$l'_i(s, t) \le l_{i-1}(s, t-q) + L_i(q) \le w_{i-1} + L_i(q) \le w_i \quad \text{[by (2.1)]}$$

If $l'_i(s, t) > 0$ then $l_i(s, t) = l'_i(s, t) \le w_i$, if $l'_i(s, t) < 0$ then $l_i(s, t) = 0 \le w_i$, since $w_i \ge 0$. Therefore, $l_i(s, t) \le w_i$. Similarly, $r_i(s, t) \ge w_i$.

To prove the sufficient condition for legality we first show the following claim.

Claim If $l_{i-1}(s, t-q) + L_i(q) \le w_i \le r_{i-1}(s, t-q) + R_i(q)$, for $t \in range_i(s)$ and $q \in [lowq_i(s, t), highq_i(s, t)]$, then there are w and w_{i-1} such that, $w_i = w_{i-1} + w$, $L_i(q) \le w \le R_i(q)$, and $(s, t-q, w_{i-1})_{i-1}$ is legal. This implies that if $0 \le w_i \le s - length_i$ then $(s, t, w_i)_i$ is legal.

Proof (claim). The interval $[l_{i-1}(s, t-q) + L_i(q), r_{i-1}(s, t-q) + R_i(q)]$ can be divided into two parts, since,

$$l_{i-1}(s, t-q) + L_i(q) \le r_{i-1}(s, t-q) + L_i(q) \quad \text{[by corollary 2.1]}$$

$$\le r_{i-1}(s, t-q) + R_i(q)$$

If w_i falls in the first part of the interval, i.e.

$$l_{i-1}(s, t-q) + L_i(q) \le w_i \le r_{i-1}(s, t-q) + L_i(q)$$

choose $w = L_i(q)$ and $w_{i-1} = w_i - w$, then

$$L_i(q) = w \le R_i(q)$$

and

$$l_{i-1}(s, t-q) \le w_{i-1} = w_i - w \le r_{i-1}(s, t-q)$$

If w_i falls in the second part of the interval, i.e.

$$r_{i-1}(s,\ t-q) + L_i(q) \le w_i \le r_{i-1}(s,\ t-q) + R_i(q)$$

choose $w_{i-1} = r_{i-1}(s,\ t-q)$ and $w = w_i - w_{i-1}$, then

$$l_{i-1}(s,\ t-q) \le w_{i-1} = r_{i-1}(s,\ t-q)$$

and

$$L_i(q) \le w_i - w_{i-1} = w \le R_i(q)$$

End of claim.

Assume $l_i(s,\ t) \le w_i \le r_i(s,\ t)$. The sufficient condition implies that $0 \le w_i \le s - length_i$. In addition,

$$l_i'(s,\ t) \le l_i(s,\ t) \le w_i \le r_i(s,\ t) \le r_i'(s,\ t-q)$$

Let, q_1 and q_2 be channel separations of the i^{th} channel such that

$$l_i'(s,\ t) = l_{i-1}(s,\ t-q_1) + L_i(q_1)$$

and

$$r_i'(s,\ t) = r_{i-1}(s,\ t-q_2) + R_i(q_2)$$

Notice that $q_1,\ q_2 \in [lowq_i(s,\ t),\ highq_i(s,\ t)]$. Then we have the following,

$$l_i'(s,\ t) = l_{i-1}(s,\ t-q_1) + L_i(q_1) \le w_i$$

$$\le r_{i-1}(s,\ t-q_2) + R_i(q_2) = r_i'(s,\ t)$$

In addition,

$$l_{i-1}(s,\ t-q_1) + L_i(q_1) \le r_{i-1}(s,\ t-q_1) + R_i(q_1)$$

and

$$l_{i-1}(s,\ t-q_2) + L_i(q_2) \le r_{i-1}(s,\ t-q_2) + R_i(q_2)$$

We will show the intersection of the following two intervals,

(1) $[l_{i-1}(s,\ t-q_1) + L_i(q_1),\ r_{i-1}(s,\ t-q_1) + R_i(q_1)]$

(2) $[l_{i-1}(s,\ t-q_2) + L_i(q_2),\ r_{i-1}(s,\ t-q_2) + R_i(q_2)]$

is not empty by showing,

$$l_{i-1}(s,\ t-q_2) + L_i(q_2) \le r_{i-1}(s,\ t-q_1) + R_i(q_1)$$

There are two cases,

Case 1: if $q_1 \le q_2$, i.e. $t - q_2 \le t - q_1$

Since $L_i(q)$ is non-increasing and $r_{i-1}(s, t)$ is non-decreasing,

$$l_{i-1}(s, t - q_2) + L_i(q_2) \le l_{i-1}(s, t - q_2) + L_i(q_1)$$

$$\le r_{i-1}(s, t - q_2) + R_i(q_1)$$

$$\le r_{i-1}(s, t - q_1) + R_i(q_1)$$

Case 2: if $q_1 \ge q_2$, i.e. $t - q_2 \ge t - q_1$

Since $l_{i-1}(s, t)$ is non-increasing and $R_i(q)$ is non-decreasing,

$$l_{i-1}(s, t - q_2) + L_i(q_2) \le l_{i-1}(s, t - q_1) + L_i(q_2)$$

$$\le r_{i-1}(s, t - q_1) + R_i(q_2)$$

$$\le r_{i-1}(s, t - q_1) + R_i(q_1)$$

Therefore w_i is either in interval (1) or in interval (2); in either case, by the above claim, $(s, t, w_i)_i$ is legal. □

Lemma 2.2 implies that $(s, tmin_i(s))_i$ is a minimal shape for the tradeoff dimension s. Lemma 2.2 also provides a mechanism to compute the position of the $i-1^{st}$ component, w_{i-1}, from the position of the i^{th} component, w_i, of a legal configuration in $(s, t, w_i)_i$. This mechanism can be applied repeatedly until positions of all components are known, and thus a legal configuration is constructed. Another approach to constructing a legal configuration in $(s, t, w_i)_i$ is to keep the separation vector $(t_1, t_2, \ldots, t_i)$ for each total separation t, where $t = \sum_{j=1}^{j=i} t_j$, while computing $l_i(s, t)$ and $r_i(s, t)$. Then an optimal configuration can be constructed by solving for the longest path of the corresponding placement graph.

For a stack of $k+1$ components, $l_k(s, t)$ and $r_k(s, t)$ can be computed straightforwardly from their definitions as follows:

ComputeIr:

(1) **for** $i = 1$ to k

(2) Compute:

the left constraint $L_i(t)$, for $t = 0$ to n_i

the right constraint $R_i(t)$, for $t = 0$ to n_i

and the minimum separation $Tmin_i$ of the i^{th} channel.

(3) $tmin_0(s) = 0$; $l_0(s, 0) = 0$, $r_0(s, 0) = s\text{-}length_0$;

$Tmin_0 = 0$; $N_0 = 0$.

(4) **for** $i = 1$ to k

(5) **for** $t = tmin_{i-1}(s) + Tmin_i$ to N_i

(6) $lowq_i(s, t) = \max\{t - N_{i-1}, Tmin_i\}$

(7) $highq_i(s, t) = \min\{n_i, t - tmin_{i-1}(s)\}$

(8) $l'_i(s, t) = \min\{l_{i-1}(s, t-q) + L_i(q) \mid lowq_i(s, t) \le q \le highq_i(s, t)\}$

(9) $r'_i(s, t) = \max\{r_{i-1}(s, t-q) + R_i(q) \mid lowq_i(s, t) \le q \le highq_i(s, t)\}$

(10) $l_i(s, t) = \max\{0, l'_i(s, t)\}$

(11) $r_i(s, t) = \min\{s\text{-}length_i, r'_i(s, t)\}$

(12) $tmin_i(s)$ = smallest t such that $l_i(s, t) \le r_i(s, t)$.

We will now analyze the complexity of computing $l_k(s, t)$ and $r_k(s, t)$ for t in $range_i(s)$. At line (2) in **ComputeIr**, $L_i(t)$ and $R_i(t)$ for t ranging from 0 to $n_i - 1$ can be computed by a sweep of the matrix dBT; this takes $O(n_i^2)$ time. See Section 2.1. $Tmin_i$ can be computed in $O(n_i)$ time, due to the fact that $L_i(t)$ is non-increasing and $R_i(t)$ is non-decreasing. Let n be the total number of nets $n = N_k$. Thus the total time consumed at line (2) is

$$C \cdot \sum_{i=1}^{i=k} n_i^2 \le C \cdot (\sum_{i=1}^{i=k} n_i)^2 \le C \cdot n^2 = O(n^2)$$

for some constant C.

From line (5) to line (11), $l'_i(s, t)$ is computed by an off diagonal sweep. This takes $O(N_{i-1} \cdot n_i)$ time. Similarly, $r'_i(s, t)$ is computed in $O(N_{i-1} \cdot n_i)$ time. $tmin_i(s)$ can be computed in $O(N_i)$ time due to the fact that $l_i(s, t)$ is non-increasing and $r_i(s, t)$ is non-decreasing. Therefore the total time required from line (4) to line (12) is,

$$\sum_{i=1}^{i=k} C \cdot (N_{i-1} \cdot n_i + N_i) \le C \cdot \sum_{i=1}^{i=k} (n \cdot n_i + n)$$

$$\le C \cdot n \cdot \sum_{i=1}^{i=k} (n_i + 1)$$

$$\leq C \cdot (n^2 + k \cdot n) = O(n^2 + k \cdot n)$$

for some constant C.

Without loss of generality, we may assume there is at least one net in each channel, therefore $k \leq n$. And the total time required to compute $l_k(s, t)$, $r_k(s, t)$ for t in $range_i(s)$ and $tmin_k(s)$ is $O(n^2)$.

2.2.2 Maximal Refinement of Well-behaved Intervals

ComputeIr computes the left boundary function, the right boundary function and the minimum total separation of a stack of $k+1$ components for a given tradeoff dimension s. All minimal shapes of the stack can be computed by applying **ComputeIr** to compute the minimal shape of each tradeoff dimension s. However, a minimal shape $(s, t)_k$ for s may not be a minimal shape for t, i.e. it may not be an actual break point in the shape function. Therefore this approach may have potentially many redundant computations. In addition, the range of tradeoff dimension one needs to consider by the above approach is from $\max_{0 \leq i \leq k} \{length_i\}$ to $\sum_{i=0}^{i=k} length_i$. This could be exponential in the length of the input.

Our algorithm computes the left and the right boundary functions $l_k(s, t)$ and $r_k(s, t)$ at *potential break points*, which are tradeoff dimensions where break points (minimal shapes) of the stack are likely to occur. As we will show in the following lemmas, the number of potential break points of the stack is approximately $O(k \cdot n)$; this results in an $O(k \cdot n^3)$ algorithm.

Two consecutive potential break points define a *well-behaved interval*, in which no break point occurs except possibly at the left boundary. The basic skeleton of the algorithm is a refinement of well-behaved intervals of a stack of i components into well-behaved intervals for a stack of $i+1$ components. Formally, a half-open interval $[s, s')$ is said to be a well-behaved interval of a stack of $i+1$ components, if

For $\Delta \geq 0$ and $s + \Delta < s'$

(1) $tmin_i(s + \Delta) = tmin_i(s)$

(2) $l_i(s + \Delta, t) = l_i(s, t)$, for $t \geq tmin_i(s)$

(3) $r_i(s + \Delta, t) = r_i(s, t) + \Delta$, for $t \geq tmin_i(s)$

A quick observation is that if $[s, s')$ is a well-behaved interval, then $[s + \delta_1, s + \delta_2)$ is also a well-behaved interval when $s + \delta_1 < s + \delta_2 \leq s'$.

Consider a legal configuration in the shape $(s, tmin_i(s))_i$, the geometrical interpretation of a well-behaved interval $[s, s')$ is that, as we increase the tradeoff dimension by Δ, it is not sufficient to accommodate a legal configuration with the position of the top component closer to the left boundary, so the left

boundary function remains unchanged, on the other hand we can shift the whole configuration to the right by Δ, so the right boundary function increases by Δ. See Figure 2.11.

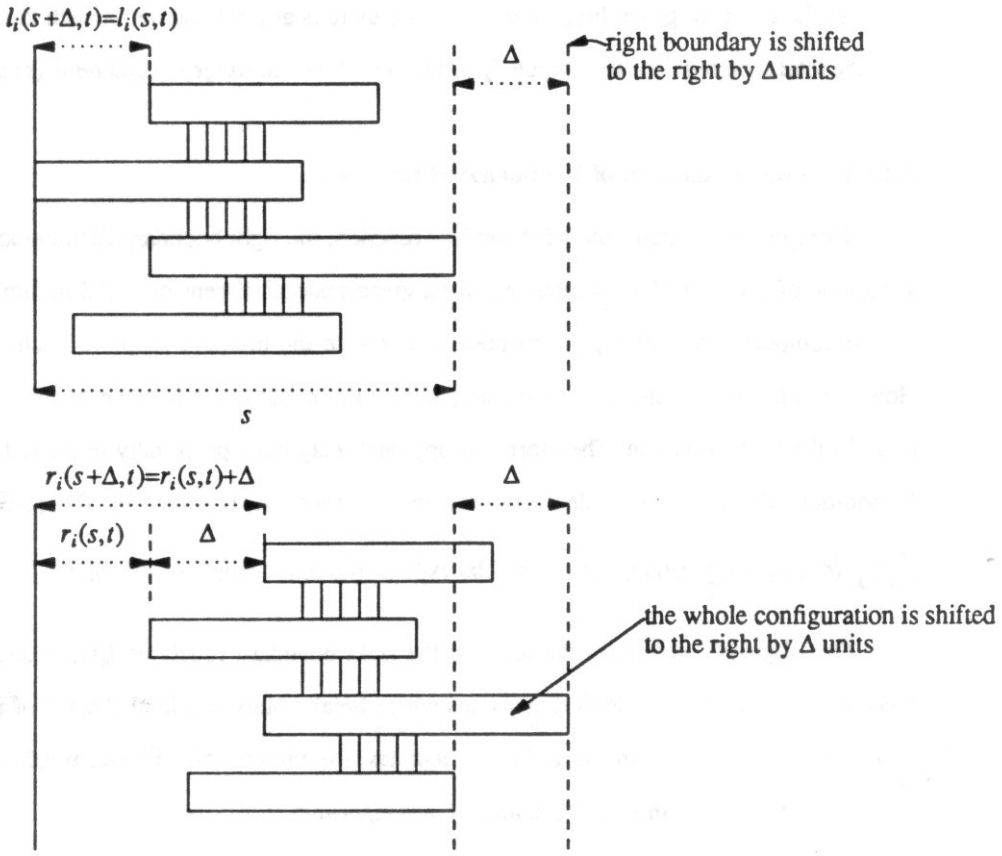


Figure 2.11

Lemma 2.3 For any tradeoff dimension s, $\Delta \geq 0$, and t in $range_i(s)$,

$$t \in range_i(s+\Delta),$$

$$l_i(s+\Delta, t) \leq l_i(s, t)$$

$$r_i(s+\Delta, t) \geq r_i(s, t) + \Delta$$

$$tmin_i(s+\Delta) \leq tmin_i(s)$$

Proof. The lemma can be proved by simple induction on i. We will only show the induction step. Assume the lemma is true for $i-1$, then

$$lowq_i(s+\Delta, t) = \max\{ t-N_{i-1}, Tmin_i \} = lowq_i(s, t)$$

$$highq_i(s+\Delta,\ t) = \min\{\ n_i, t - tmin_{i-1}(s+\Delta)\ \}$$

$$\geq \min\{n_i, t - tmin_{i-1}(s)\ \} = highq_i(s,\ t)$$

$$tmin_{i-1}(s+\Delta) + Tmin_i \leq tmin_{i-1}(s) + Tmin_i \text{ implies } range_i(s) \subset range_i(s+\Delta)$$

$$l'_i(s,\ t) = \min\{\ l_{i-1}(s,\ t-q) + L_i(q) \mid lowq_i(s,\ t) \leq q \leq highq_i(s,\ t)\ \}$$

$$l'_i(s+\Delta,\ t) = \min\{\ l_{i-1}(s+\Delta,\ t-q) + L_i(q) \mid lowq_i(s+\Delta,t) \leq q \leq highq_i(s+\Delta,t)\ \}$$

$$\leq \min\{\ l_{i-1}(s,\ t-q) + L_i(q) \mid lowq_i(s,\ t) \leq q \leq highq_i(s,\ t)\ \}$$

$$= l'_i(s,\ t)$$

Therefore, $l_i(s+\Delta,\ t) = \max\{\ 0, l'_i(s+\Delta,\ t)\ \} \leq \max\{\ 0,\ l'_i(s,\ t)\ \} = l_i(s,\ t)$. And,

$$r'_i(s,\ t) = \max\{\ r_{i-1}(s,\ t-q) + R_i(q) \mid lowq_i(s,\ t) \leq q \leq highq_i(s,\ t)\ \}$$

$$r'_i(s+\Delta,\ t) = \max\{\ r_{i-1}(s+\Delta,\ t-q) + R_i(q) \mid lowq_i(s+\Delta,t) \leq q \leq highq_i(s+\Delta,t)\ \}$$

$$\geq \max\{\ r_{i-1}(s,\ t-q) + \Delta + R_i(q) \mid lowq_i(s,\ t) \leq q \leq highq_i(s,\ t)\ \}$$

$$= \max\{\ r_{i-1}(s,\ t-q) + R_i(q) \mid lowq_i(s,\ t) \leq q \leq highq_i(s,\ t)\ \} + \Delta$$

$$= r'_i(s,\ t) + \Delta$$

Therefore,

$$r_i(s+\Delta,\ t) = \min\{\ s+\Delta - length_i,\ r'_i(s+\Delta,\ t)\ \}$$

$$\geq \min\{\ s+\Delta - length_i,\ r'_i(s,\ t) + \Delta\ \}$$

$$= \min\{\ s - length_i,\ r'_i(s,\ t)\ \} + \Delta$$

$$= r_i(s,\ t) + \Delta$$

In addition,

$$l_i(s+\Delta,\ tmin_i(s)) \leq l_i(s,\ tmin_i(s)) \leq r_i(s,\ tmin_i(s)) \leq r_i(s+\Delta,\ tmin_i(s))$$

therefore, $tmin_i(s+\Delta) \leq tmin_i(s)$. □

Corollary 2.2 In a well-behaved interval $[s, s')$ for a stack of i components, if $s + \Delta \le s'$ then for a stack of $i+1$ components and for t in $range_i(s)$

$$l_i(s + \Delta, t) = l_i(s, t),$$

$$r_i(s + \Delta, t) = r_i(s, t) + \Delta, \text{ and}$$

$$range_i(s + \Delta) = range_i(s)$$

Proof. Replace the inequalities by equalities of the inductive step above. □

For a well-behaved interval $[s, s')$ of a stack of i components $(0, 1, 2, \ldots, i-1)$ we say the sequence of tradeoff dimensions

$$s = s_0 < s_1 < \cdots < s_m = s'$$

is a *maximal refinement* of $[s, s')$ after adding the i^{th} component (now a stack of $i+1$ components) if

(1) $[s_j, s_{j+1})$ is a well-behaved interval of the stack of $i+1$ components, for $j = 0$ to $m-1$

(2) $tmin_i(s_j) > tmin_i(s_{j+1})$, for $j = 0$ to $m-2$.

(3) $tmin_i(s_{m-1}) \ge tmin_i(s_m)$

Figure 2.12 shows an example of a maximal refinement of a well-behaved interval.

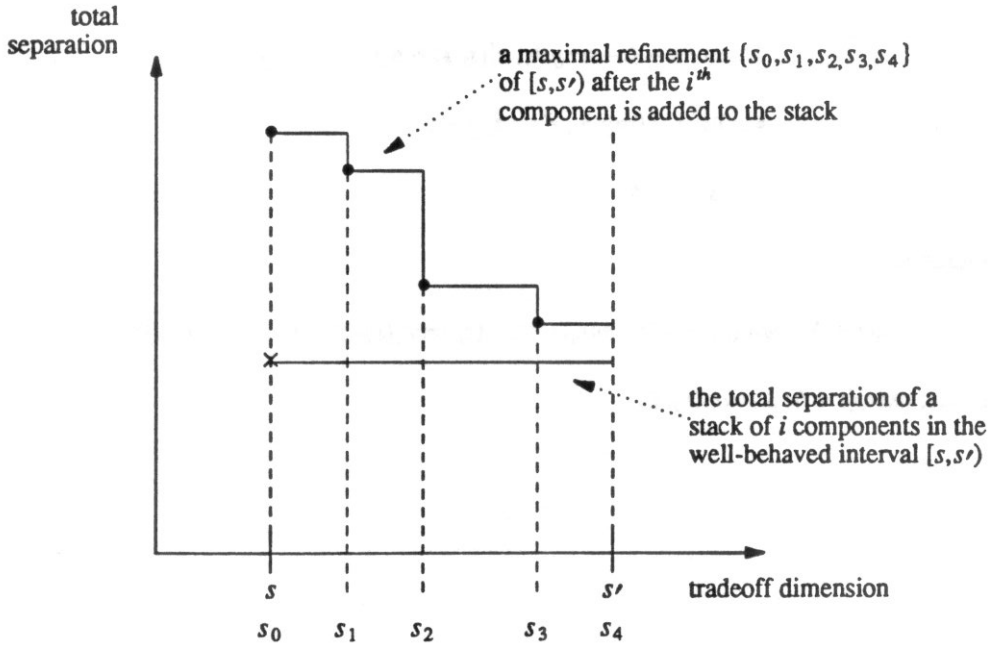


Figure 2.12

Lemma 2.4 Given a well-behaved interval $[s, s')$ of a stack of i components, adding one more component to the stack we can compute the maximal refinement

$$s = s_0 < s_1 < \cdots < s_m = s'$$

of $[s, s')$ of the new stack in time $O(N_i^2)$.

Proof. We shall construct such a refinement. Intuitively, beginning at the left boundary $s = s_0$, we look for maximal tradeoff dimension s_1 such that $[s_0, s_1)$ is a well-behaved interval of the stack of $i+1$ components and $tmin_i(s_0) > tmin_i(s_1)$ and continue the process until the last well-behaved interval is found.

Formally, compute $l_i(s, t)$ and $r_i(s, t)$ for t in $range_i(s)$, and locate $tmin_i(s)$, i.e. find the smallest t in $range_i(s)$ such that $l_i(s, t) \le r_i(s, t)$. By Corollary 2.2, $l_i(s + \Delta, t) = l_i(s, t)$ and $r_i(s + \Delta, t) = r_i(s, t) + \Delta$ for $s + \Delta < s'$. If $tmin_i(s)$ equals its minimum achievable value over $[s, s')$, $tmin_{i-1}(s) + Tmin_i$, then the maximal refinement is $[s, s')$. Otherwise increase the tradeoff dimension by Δ until $tmin_i(s) > tmin_i(s + \Delta)$. This occurs when

$$\Delta = l_i(s, tmin_i(s)-1) - r_i(s, tmin_i(s)-1).$$

Since,

$$r_i(s + \Delta, tmin_i(s)-1) = r_i(s, tmin_i(s)-1) + \Delta$$

$$= l_i(s, tmin_i(s)-1) = l_i(s + \Delta, tmin_i(s)-1)$$

Let $s_0 = s$ and $s_1 = s_0 + l_i(s, tmin_i(s)-1) - r_i(s, tmin_i(s)-1)$. Then $[s_0, s_1)$ is a well-behaved interval of the stack of i channels. This is because, for $s_0 + \delta < s_1$,

$$r_i(s + \delta, tmin_i(s)-1) = r_i(s, tmin_i(s)-1) + \delta$$

$$< r_i(s, tmin_i(s)-1) + l_i(s, tmin_i(s)-1) - r_i(s, tmin_i(s)-1)$$

$$= l_i(s, tmin_i(s)-1) = l_i(s + \delta, tmin_i(s)-1)$$

So $tmin_i(s_0 + \delta) = tmin_i(s_0)$.

In general,

$$s_{j+1} = s_j + l_i(s_j, tmin_i(s_j)-1) - r_i(s_j, tmin_i(s_j)-1)$$

and $[s_j, s_{j+1})$ is a well-behaved interval of the stack when $s_{j+1} \le s'$. This is because, for $s_j + \delta < s_{j+1}$

$$r_i(s_j + \delta, tmin_i(s_j)-1) = r_i(s_j, tmin_i(s_j)-1) + \delta \qquad (I)$$

$$< r_i(s_j, tmin_i(s_j)-1) + l_i(s_j, tmin_i(s_j)-1) - r_i(s_j, tmin_i(s_j)-1)$$

$$= l_i(s_j, tmin_i(s_j)-1) = l_i(s_j + \delta, tmin_i(s_j)-1)$$

The equality (I) is due to the fact that $[s_j, s_{j+1})$ is a well-behaved interval for the stack of i channels.

We halt when $tmin_i(s_j)$ reaches its minimum achievable value $tmin_{i-1}(s) + Tmin_i$ or when s_{j+1} is greater than or equal to s', in either case set $s_{j+1} = s'$. Lemma 2.3 implies $tmin_i(s_{m-1}) \ge tmin_i(s_m)$.

The actual refinement process is described as follows,

Refine (s, s')

(1) Compute $l_i(s, t)$ and $r_i(s, t)$ for t in $range_i(s)$.
(2) $j = 0$; $s_j = s$
(3) **while** $(s_j < s')$
(4) locate $tmin_i(s_j)$
(5) **if** $(tmin_i(s_j) = Tmin_i + tmin_{i-1}(s))$ **then**
(6) $s_{j+1} = s'$
(7) **else**
(8) $s_{j+1} = s_j + l_i(s_j, tmin_i(s_j)j)-1) - r_i(s_j, tmin_i(s_j)-1)$
(9) $j = j + 1$
(10) $s_j = s'$

The complexity of **Refine** is $O(N_i^2)$ for a stack of $i+1$ components. It requires $O(N_i^2)$ to compute $l_i(s, t)$ and $r_i(s, t)$. The time consumed in the while loop is equal to the number of resulting well-behaved intervals. Since $tmin_i(s_j)$ is strictly decreasing, this number does not exceed N_i. In addition locating $tmin_i(s_j)$ takes no more than the size of $range_i(s)$. Thus the complexity of **Refine** is $O(N_i^2 + N_i)$. If we assume $l_{i-1}(s, t)$ and $r_{i-1}(s, t)$ are given, then it requires $O(N_{i-1} \cdot n_i)$ to compute $l_i(s, t)$ and $r_i(s, t)$. See Lemma 2.2. Thus the complexity in this case is $O(N_{i-1} \cdot n_i + N_i)$. □

2.2.3 The Main Theorem

Theorem The shape-of-stack problem can be solved in $O(k \cdot n^3)$ time.

Proof. We show this by providing an algorithm to compute a sequence of tradeoff dimensions,

$$\max\{ length_i \} = s_0 < s_1 < s_2 < \cdots < s_m = \sum_{i=0}^{i=k} length_i$$

such that

$$tmin_k(s_0) > tmin_k(s_1) > tmin_k(s_2) > \cdots > tmin_k(s_m)$$

and when $s_i+\Delta<s_{i+1}$

$$tmin_k(s_i+\Delta)=tmin_k(s_i), for\ i = 1\ to\ m-1$$

Notice that the above sequence of tradeoff dimensions covers the range of tradeoff dimension that is needed to be considered, since for any tradeoff dimension $s > s_m$, the minimum total separation $tmin_k(s)$ achieves its minimum achievable value $\sum_{i=1}^{i=k} Tmin_i$.

We construct the sequence incrementally as follows, Begin with the 0^{th} component, $s_0^0 = \max\{ length_i \}$, $s_1^0 = (\sum length_i)+1$. $[s_0^0, s_0^1)$ is a well-behaved interval of the stack of one component. Adding the 1^{st} component, by Lemma 2.4, we can construct a maximal refinement of $[s_0^0, s_1^0)$:

$$s_0^0 = s_0^1 < s_1^1 < s_2^1 < \cdots < s_{m_1}^1 = s_1^0$$

$$tmin_1(s_0^1) > tmin_1(s_1^1) > \cdots > tmin_1(s_{m_1}-1^1) \geq tmin_1(s_{m_1}^1),$$

Each of $[s_j^1, s_{j+1}^1)$ is a well-behaved interval of the stack of two components. In general, after adding the next top component, **Refine** can be applied to construct the maximal refinement of each of the well-behaved interval. The process is continued until the refinement is obtained for the whole stack. After the last refinement, we obtain the sequence $\{ s_0^k, s_1^k, \ldots, s_m^k \}$ and $tmin_k(s_j^k)$ is computed in the course of computing each s_j^k. The sequence has the property that $tmin_k(s_i^k+\Delta)=tmin_k(s_i^k)$, when $s_i^k+\Delta<s_{i+1}^k$. Thus we can scan the sequence once and merge the intervals whose minimum total separations at the left boundaries are equal. This gives us the required sequence and thus the shape function. A typical refinement of the shape function for a stack of i components after the i^{th} component is added is shown in Figure 2.13. The algorithm is described as follows,

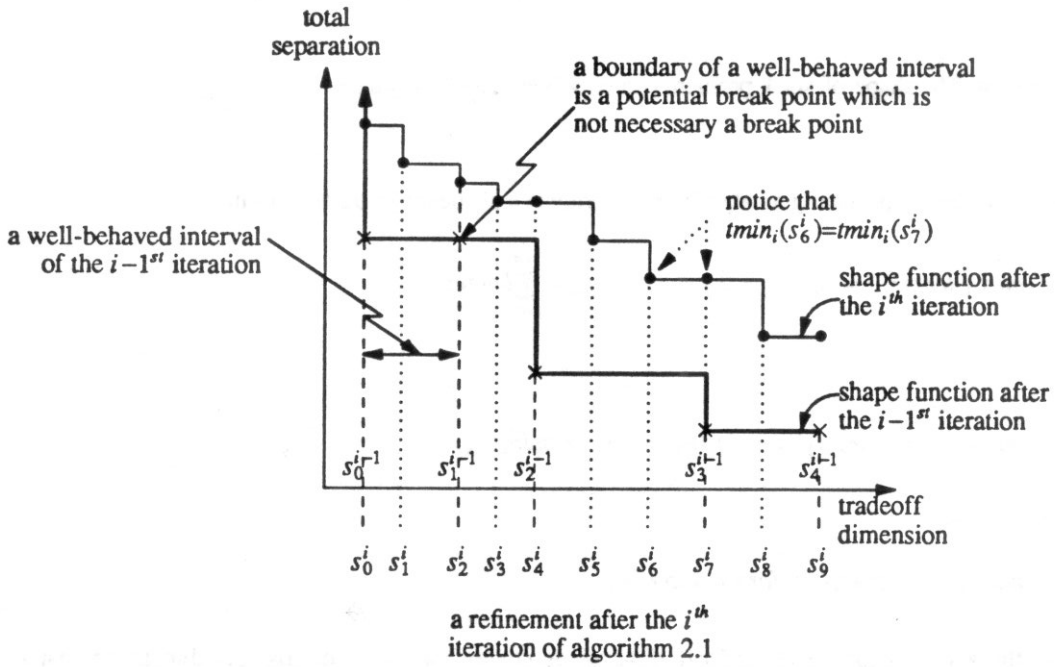


Figure 2.13

Algorithm 2.1:

(0) $s_0^0 = \max\{ length_i \}$; $s_1^0 = \sum_{i=0}^{i=k} length_i + 1$

(1) Initialize $List^0$: $List^0 = \{ s_0^0, s_1^0 \}$

(2) $i = 1$

(3) **while** $(i \le k)$

(4) add the i^{th} component to stack

(5) **for** (each well-behaved interval $[s_j^{i-1}, s_{j+1}^{i-1})$ in $List^{i-1}$)

(6) **Refine** $(s_j^{i-1}, s_{j+1}^{i-1})$

(7) add the maximal refinement into $List^i$

(8) $i = i + 1$

(9) Scan $List^k$ and merge intervals whose minimum total separations at the left boundaries are equal.

The complexity of Algorithm 2.1 depends on the number of times **Refine** is invoked. This number is equal to the total number of intervals in all $List^i$ for i=0 *to* $k-1$. Let $size\ (List^i)$ denotes the number of intervals in $List^i$. For $i>1$, we show in the following

$$size\ (List^i) \le N_i + size\ (List^{i-1}) = O\ (i \cdot N_i) \qquad (2.2.1)$$

After the sequence of s_j^i in $List^i$ is constructed, $tmin_i(s_j^i)$ is strictly decreasing except at some $s_\alpha^i = s_\beta^{i-1}$, where $tmin_i(s_{\alpha-1}^i) \geq tmin_i(s_\alpha^i)$. Therefore $size(List^i) \leq N_i + 1 + size(List^{i-1}) - 1 = N_i + size(List^{i-1})$, because there may be N_i+1 intervals due to the monotonicity of $tmin_i(t)$ and the 1^{st} entry s_0^i is not duplicated. We shall show (2.2.1) for $i>1$ by induction. $size(List^1) \leq N_1+1$, $size(List^2) \leq N_2+N_1+1 \leq 2 \cdot N_2$ (we assume there is no empty channel). Assume true for $i-1$,

$$size(List^i) \leq N_i + (i-1) \cdot N_{i-1}, \text{ for } i > 2$$

$$\leq N_i + (i-1) \cdot N_i \leq i \cdot N_i$$

thus $size(List^i) = O(i \cdot N_i)$ for $i>1$.

We keep $l_{i-1}(s, t)$ and $r_{i-1}(s, t)$ after the $i-1^{st}$ iteration. So at the i^{th} iteration the time required for **Refine**$(s_j^{i-1}, s_{j+1}^{i-1})$ is $O(N_{i-1} \cdot n_i + N_i)$. The time required to refine $List^0$ is $O(n_1^2)$, and the time required to refine $List^1$ is $O((N_1+1) \cdot (N_1 \cdot n_2 + N_2))$. The total time required to construct the maximal refinement of all intervals in $List^{i-1}$ for $i-1>1$ is $O(size(List^{i-1}) \cdot (N_{i-1} \cdot n_i + N_i))$, and the complexity to establish the shape function is therefore,

$$C_1 \cdot n_1^2 + C_2 \cdot (N_1+1) \cdot (N_1 \cdot n_2 + N_2) + \sum_{i=2}^{i=k-1} C_3 \cdot i \cdot (N_i^2 \cdot n_{i+1} + N_i \cdot N_{i+1})$$

$$\leq C_1 \cdot n_1^2 + C_2 \cdot (N_1 \cdot n_2 + N_2) + \sum_{i=1}^{i=k-1} C_4 \cdot i \cdot (N_i^2 \cdot n_{i+1} + N_i \cdot N_{i+1})$$

$$\leq C_1 \cdot n_1^2 + C_2 \cdot (N_1 \cdot n_2 + N_2) + C_4 \cdot k \cdot n^2 \cdot \sum_{i=1}^{i=k-1} (n_{i+1} + 1)$$

$$\leq C \cdot (k \cdot n^3 + k^2 \cdot n^2)$$

for some constants C_1, C_2, C_3, C_4 and C.

Without loss of generality we assume there is at least one net in each channel, otherwise the stack can be divided into two independent stacks and the shape function of the whole stack is the sum of the two individual shape functions. So we assume $n \geq k$ and the complexity of Algorithm 2.1 is $O(k \cdot n^3)$. □

2.2.4 Simplification on One-Sided Constraints

Algorithm 2.1 can be sped up by a factor of two if the left and the right constraints of all channels in a stack are one-sided. We say the left and right constraints are one-sided if $R_i(Tmin_i)$ is non-positive or $L_i(Tmin_i) \geq length_{i-1} - length_i$, for $i = 1,...,k$.

When $R_i(Tmin_i)\leq 0$, for $i = 1,...,k$, all the left boundary functions are at their minimum, i.e. $l_i(s,t)=0$, and we need only to compute the right boundary functions when computing the shape function. And when $L_i(Tmin_i)\geq length_{i-1}-length_i$, for $i = 1,...,k$, all the right boundary functions are always at their maximum, i.e. $r_i(s,t)=s-length_i$. Formally,

Lemma 2.5 In a stack of $k+1$ components, if $R_i(Tmin_i)\leq 0$ for $i = 1,...,k$. Then $l_i(s,t)=0$ for $\max\{length_i\}\leq s\leq\sum_{i=0}^{i=k} length_i$ and $t \in range_i(s)$.

Proof. By simple induction on i. $R_i(Tmin_i)\leq 0$ implies $L_i(Tmin_i)\leq 0$. Since $L_i(t)$ is non-increasing, this implies $L_i(t)\leq 0$ for all channel separations $t\geq Tmin_i$. Assume $l_{i-1}(s,t)=0$ then,

$$l'_i(s,t)=\min\{l_{i-1}(s,t-q)+L_i(q) \mid lowq_i(s,t)\leq q\leq highq_i(s,t)\}\leq 0$$

Thus, $l_i(s,t)=0$. □

When $L_i(Tmin_i)\geq length_{i-1}-length_i$ the result on the right boundary function can be shown similarly. Therefore when we have one-sided constraints, only one boundary function needs to be computed and the running time is sped up by a factor of two.

2.3 Stack of Multiple Components

In the previous section we considered stacks with single component entries. In this section we consider stacks with multiple component entries. The *shape-of-rows problem* is described as follows,

> Given a stack of $k+1$ rows, each row has one or more components with predetermined order, and each channel defined by two adjacent rows is river routable. Compute the shape function of the stack of rows. See Figure 2.14.

For simplicity we assume components in a row have the same widths. First we generalize the terms we used in the stack of single component to the stack of rows. A *separation vector* $(t_1, t_2, \ldots, t_k)$ describes the channel separation of a stack of $k+1$ rows. A *configuration* of the stack of $k+1$ rows for a separation vector $(t_1, t_2, \ldots, t_k)$ is a placement of each components such that the structure of the stack is preserved -- the i^{th} channel has channel separation t_i for i from 1 to k, components on the same row do not overlap and the order of the components is preserved.

Notice that each channel in the stack of rows is an instance of a multiple component channel. (See Section 2.1.2.) If the channel separation of a channel is given, the placement graph that describes

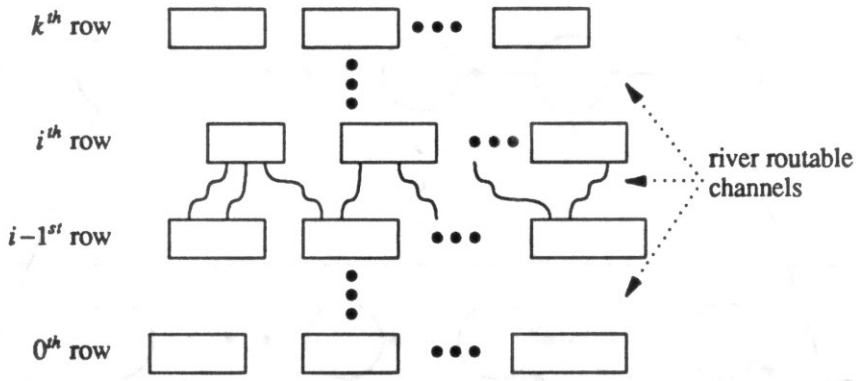


Figure 2.14

constraints between components that define that channel can be derived as in Section 2.1.2. For a stack of $k+1$ rows if a separation vector $(t_1, t_2, \ldots, t_k)$ is given, the placement graph of each channel can be derived, and the *placement graph* of the stack of rows which expresses constraints between components in the stack can then be established by merging the placement graphs of each individual channel of the stack. This is done by identifying vertices which represent the positions of the same components, vertices which represent the left boundary, and vertices which represent the right boundary. This results in a single-source-single-sink weighted directed graph. See Figure 2.15.

For a given separation vector $(t_1, t_2, \ldots, t_k)$, a *legal configuration* is a configuration which does not violate the constraints imposed by the placement graph of the stack. An optimal configuration for a stack of $k+1$ rows for the separation vector is a legal configuration whose tradeoff dimension is minimum. Let v_l be the vertex that represents the left boundary and v_r be the vertex that represents the right boundary, in the placement graph associated with the separation vector $(t_1, t_2, \ldots, t_k)$, The minimum tradeoff dimension for the separation vector can be computed by assignment of values to vertices in the placement graph, subject to the constraints, such that $v_r - v_l$ is minimized. Again this amounts to finding the longest path in the placement graph and can be solved by the standard Bellman-Ford algorithm [Law]. In addition, the values of the vertices give us the optimal configuration.

A *total separation* for a separation vector is the sum of each channel separation. A shape $(s, t)_k$ describes a set of legal configurations, of a stack of $k+1$ rows whose tradeoff dimension does not exceed s and whose total separation does not exceed t. We say $(s, t)_k$ is *legal* if there is at least one legal configuration in $(s, t)_k$; we say $(s, t)_k$ is *minimal* if no legal configurations in $(s, t)_k$ has tradeoff dimension less than s or total separation less than t.

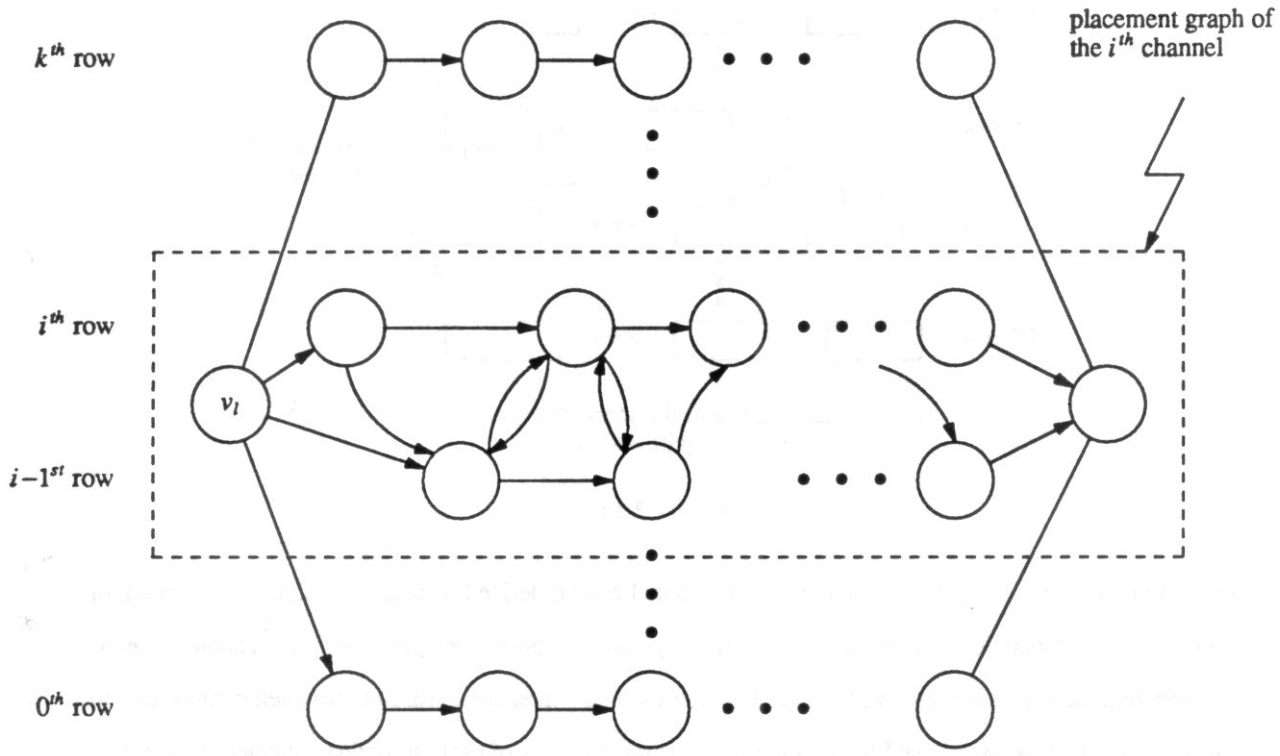


Figure 2.15

One of the difficulties in computing the minimum tradeoff dimension of a stack of rows for a total separation t is that we do not know of an efficient way to compute the channel separation of each individual channel. In fact Pinter and Sipser showed that given a stack of rows, a tradeoff dimension s and a total separation t, deciding if $(s, t)_k$ is legal is NP-complete [Pin].

Since computing the shape function of a stack of rows is difficult, we will approximate the shape function by a heuristic. The goal of the heuristic is to find a separation vector for each total separation t. The following are terms and notation required to describe the heuristic.

Consider a row in the stack, two adjacent components on the same row can define a routing channel. To avoid confusion, we call the routing channels that are directly involved in the computation of separation vectors the *direct routing channels* -- these are the channels defined by adjacent rows in the stack. We call the nets in the direct routing channels the *direct routing nets*. For a stack of $k+1$ rows, let

$Tmin_i$ the minimum channel separation of the i^{th} channel.

$tmin_i$ the smallest possible total separation of the first i channels of the stack. $tmin_i = \sum_{j=1}^{j=i} Tmin_j$.

n_i the number of nets in the i^{th} channel.

n_{max} the maximum of all n_i.

N_i the number of direct routing nets for the first i channels. $N_i = \sum_{j=1}^{j=i} n_i$.

m_i the number of components in the i^{th} row.

M_i the total number of components for the first $i+1$ rows. $M_i = \sum_{j=0}^{j=i} m_i$.

n the total number of routing nets. $n = N_k$.

m the total number of components in the stack. $m = M_k$.

$v_i(t)$ a separation vector $(t_1, t_2, \ldots, t_i)$ for a stack of $i+1$ rows whose total separation is t. $v_i(t) = (t_1, t_2, \ldots, t_i)$ and $t = \sum_{j=1}^{j=i} t_j$.

$v_i(t) \mid q$ the concatenation of a vector in $\mathbb{N}^i$ with a scaler to form a vector in $\mathbb{N}^{i+1}$. $v_i(t) \mid q = (t_1, t_2, \ldots, t_i) \mid q = (t_1, \ldots, t_i, q)$.

The heuristic computes a separation vector whose total separation is t for a stack of $i+1$ rows by considering all combinations of $v_{i-1}(t-q)$ and q, where q is a legal separation of the i^{th} channel. It chooses the best $v_{i-1}(t-q) \mid q$ pair, i.e. the pair that result in minimum tradeoff dimension of a stack of $i+1$ rows. This is done for i from 1 to k. The heuristic is described in Algorithm 2.2.

In line 7 of algorithm 2.2, $low_i(t)$ and $high_i(t)$ are the smallest and largest channel separations needed to be consider respectively at the i^{th} channel. Obviously $q \geq Tmin_i$ and $q \leq t - tmin_{i-1}$. Intuitively when the total separation of a stack of i rows is N_{i-1}, each channel has full flexibility, i.e. components on adjacent rows do not impose any constraints on each other, so the minimum tradeoff dimension achieves its smallest achievable value. For instance, for the separation vector $(n_1, n_2, \ldots, n_{i-1})$, each channel in the stack has full flexibility and the tradeoff dimension is equal to the length of the longest row. When $t-q > N_{i-1}$, i.e. $q < t-N_{i-1}$, the tradeoff dimension of the stack for the separation vector $v_{i-1}(t-q) \mid q$ is no better than the tradeoff dimension for $v_{i-1}(N_{i-1}) \mid t-N_{i-1}$; therefore we only consider $q \geq \max\{ tmin_i, t-N_{i-1} \}$. When $q > n_i$ the minimum tradeoff dimension of the stack for the separation vector $v_{i-1}(t-q) \mid q$ is no better than the tradeoff dimension for $v_{i-1}(t-n_i) \mid n_i$, therefore we only consider $q \leq \min\{ n_i, t-Tmin_{i-1} \}$.

Algorithm 2.2 (stack of $k+1$ rows)

(1) **for** $i = 1$ to k

(2) Compute $Tmin_i$.

(3) **for** $t = Tmin_1$ to n_1

(4) $v_1(t) = t$

(5) **for** $i = 2$ to k

(6) **for** $t = tmin_i$ to N_i

(7) $low_i(t) = \max\{Tmin_i, t - N_{i-1}\}$; $high_i(t) = \min\{n_i, t - tmin_{i-1}\}$

(8) $optq = high_i(t)$

(9) **for** $q = low_i(t)$ to $high_i(t)$

(10) Establish placement graph for the separation vector $v_{i-1}(t-q) \mid q$.

(11) Apply Bellman-Ford to solve longest path.

(12) **if** (shorter longest path is obtained) **then**

(13) $optq = q$

(14) $v_i(t) = v_{i-1}(t - optq) \mid optq$

The time required to compute $Tmin_i$ is $O(n_i \cdot log(n_i))$, so the total time spent at the loop at line (1) is $O(n \cdot \log(n_{max}))$. The time required to establish the placement graph at line (10) is $O(N_i)$, and the time for Bellman-Ford algorithm at line (11) is $O(M_i^2)$ since the number of edges in the placement graph at the i^{th} iteration is $O(M_i)$. See section 2.1.2. The range of q at line (9) is $O(n_i)$, and the range of t at line (6) is $O(N_i)$, therefore the complexity of algorithm 2.2 is,

$$C \cdot \sum_{i=1}^{i=k} N_i \cdot n_i \cdot (N_i + M_i^2)$$

$$\leq C \cdot \sum_{i=1}^{i=k} n_i \cdot (n^2 + n \cdot m^2)$$

$$\leq C \cdot (n^3 + n^2 \cdot m^2) = O(n^3 + n^2 \cdot m^2)$$

for some constant C.

In the above shape-of-rows problem we have assumed that components on the same row have same widths, and there are no displacement constraints between adjacent components on the same row. In practice, components in the same row may have different widths, and displacements between adjacent components on the same row may not be less than certain units due to routing requirements of the non-direct routing channels. The first assumption is required to ensure that the total width contributed by the row

widths to the routing dimension is constant, so for a row with non-uniform component widths, we use the width of the smallest rectangle that encompasses the row as its width. The second assumption can be reflected in the placement graph if the displacement constraints are known for all the non-direct routing channels.

We have not evaluated algorithm 2.2 directly. However, algorithm 2.2 has been used as a subroutine in heuristics that compute approximate shape functions of slicing structures. From the performance of those heuristics we conclude that algorithm 2.2 is a useful heuristic. This work is presented in the next chapter.

Chapter 3

Shape Function of River Slicing Layouts

In this chapter the algorithms developed in Chapter 2 are applied to approximate the shape function a river slicing layout. Three heuristics using combinations of the algorithms in Chapter 2 are described and the performance of each approach is discussed.

3.1 Introduction

While the shape function of a slicing layout without interconnections can be computed efficiently [Ott, Sto], to compute the exact shape function that includes all the necessary river routing space of a slicing layout is NP-complete [Pin]. Our goal here is to approximate the actual shape function. Recall that the exact shape function of a slicing layout captures a set of minimal dimensions (break points) of the layout, i.e. given a horizontal dimension of the layout we can find the minimum vertical dimension and vice versa. Since finding the exact set of minimal dimensions of a slicing layout with interconnections is difficult, we develop heuristics to find a set of shapes which are area efficient. Three heuristics are developed to compute an approximate shape function of a river slicing layout. In a river slicing layout, terminals on opposite sides of a routing channel must have the same net ordering, this is the same requirement in pitch aligning. We compare the performance of the heuristics with pitch aligning by comparing the smallest area of layouts produced by each of the heuristics to the area of the layout produced by pitch aligning. This comparison is intended to show the potential of our compaction scheme which treats wires as topological entities.

3.2 The Rigid Heuristic

To approximate the shape function of a river slicing layout we employ an approach similar to that used by Otten [Ott] to compute the shape function of a slicing layout without interconnection. In [Ott] the shape function of a slicing layout is computed by performing a postorder traversal on the slicing tree that represents the slicing layout. When a slice is visited the shape function of the slice is computed by summing shape functions of its child slices. After the root slice has been visited, the shape function of the slicing layout is obtained. In general, when computing the shape function of a parent slice that includes the

routing space between its child slices, a simple summing operation will not work. This is because the routing requirements between two adjacent child slices may alter when different shapes of the child slices are chosen. However, if a fixed shape is chosen for each child slice, computing the shape function of the parent reduces to the shape-of-stack problem. Our approach is based on a shape-choosing strategy in which a "good" shape is chosen for each of the child slices, and the shapes are used to compute the shape function of their parent slice.

Recall that in Chapter 2, algorithm 2.1 provided a solution to the shape-of-stack problem. We will first illustrate how algorithm 2.1 is applied to approximate the shape function of a slicing layout of depth two. We assume leaf cells of a slicing layout have rigid shapes and their terminal positions are given. Notice that a slicing layout of depth two is a stack of rows of leaf cells.

To approximate the shape function of a slicing layout of depth two, algorithm 2.1 is applied to compute the shape function of each row of components on the stack. Then a shape is chosen for each row and the row is *frozen* at that shape, i.e. the optimal configuration is constructed for the row of components and the row is treated as a rigid component. We choose the shape with the smallest area. The shape choosing strategy will be discussed in Section 3.5. Now algorithm 2.1 is applied to compute the shape function of this stack of rigid components. See Figure 3.1 for an illustration. The resulting shape function is an approximation to the actual shape function of the slicing layout.

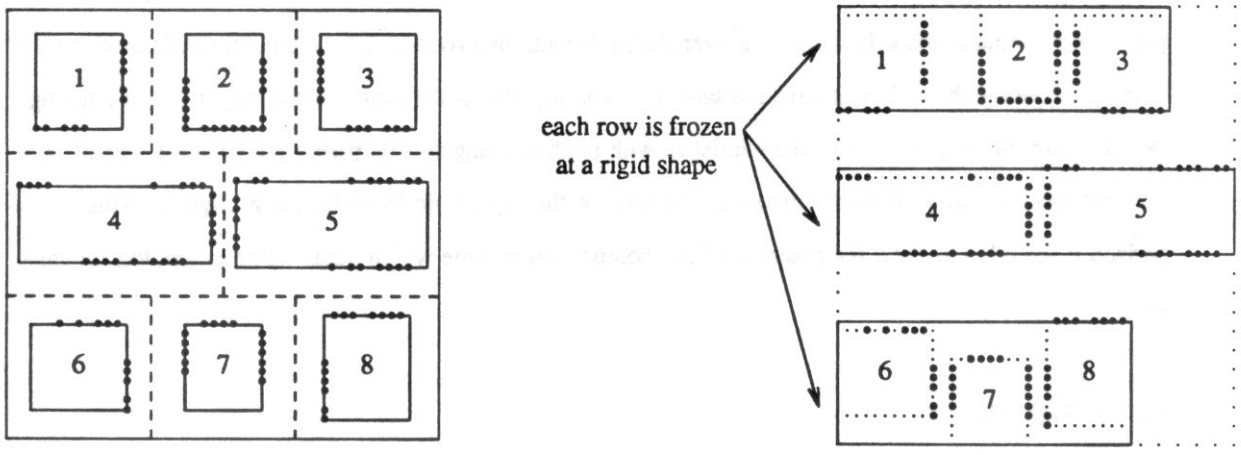


Figure 3.1

In general the above approach to approximate the shape function of a slicing layout of depth two can be applied to approximate a shape function of a slicing layout of any depth. The basic skeleton of the

heuristic is a postorder traversal on the slicing tree that represents the slicing layout. At a visit of a slice, after a shape is chosen for each of its child slices, the shape function of the slice is computed by the rigid algorithm (algorithm 2.1). Formally the heuristic is described as follows:

```
Rigid_Shape_Function( slice )
(1)   if ( slice is a leaf cell )
(2)         return( dimensions of slice )
(3)   else
(4)         for ( each child of slice )
(5)               Rigid_Shape_Function( child )
(6)               Choose a shape on the shape function.
(7)   Apply Algorithm 2.1 to compute shape function.
(8)   return( shape function computed )
```

3.3 The Thawing Heuristic

A better approximation of the shape function for a stack of rows of components can be obtained if we do not freeze the rows in their chosen shape. That is, after a shape is chosen for a row and a configuration is constructed for that row, we allow flexible displacements between two adjacent components on the row. To be more precise, the minimum displacements between two adjacent components on the row are enforced at the channel separations that are determined by the chosen configuration, but there is no constraint on the maximum displacement of the adjacent components on the row. We call a row with flexible displacements between adjacent components a *thawable* row. Now Algorithm 2.2 can be applied to approximate the shape function of this stack of thawable rows. To apply algorithm 2.2 the width of each row is required. We use the tradeoff dimension of the chosen shape of each row as its width. See Figure 3.2 for a configuration of the slicing layout in Figure 3.1 when algorithm 2.2 is used to compute the shape function.

The routing dimension of the shape function obtained by the above approach can be reduced. Recall that there is a separation vector associated with a given break point. From the separation vector a configuration of the stack of thawable rows can be constructed at that break point. After the configuration is constructed, the actual displacements between two adjacent components on a row may be larger than the predetermined minimum displacements. The accurate width of a row can be obtained by the *squeezing operation* which is the construction of the optimal configuration of the row using actual channel displacements as channel separations. This does not affect routability of the routing channels of the row. To obtain

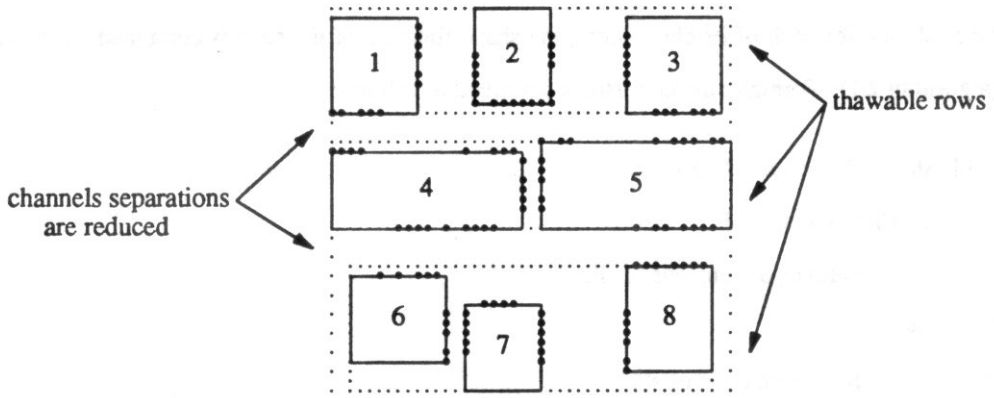


Figure 3.2

a more accurate shape function, the squeezing operation is applied to each row of components to obtain the minimum routing dimension at each break point. See Figure 3.3. We refer to the application of algorithm 2.2 and the squeezing operation as the *thawing operation*.

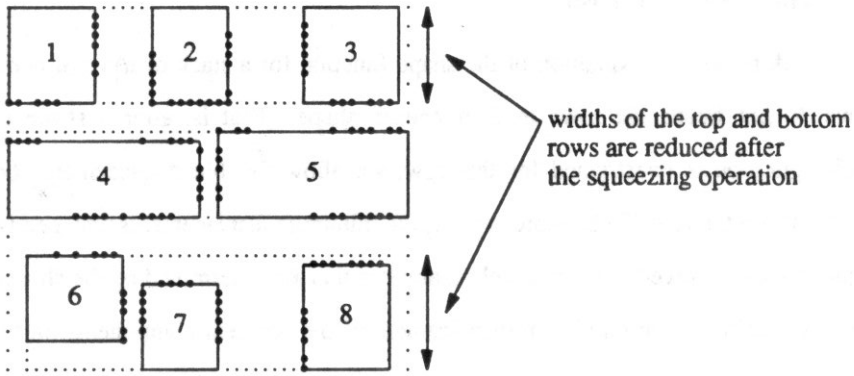


Figure 3.3

The basic skeleton of the thawing approach is the same as that of the rigid approach. Although algorithm 2.2 can potentially produce better shape functions, it is computationally more expensive than algorithm 2.1. A predetermined parameter, *Thaw_level* is used to control the level at which the thawing operation begins. Thaw_level equal to one means the thawing operation begins on slices with level higher than one, i.e. slices which have at least one non-leaf child slice. Thaw_level equal to the level of the root slice means no thawing operation is applied; this is equivalent to the rigid heuristic. Formally the thawing heuristic is described as follows:

```
Thawing_Shape_Function( slice )
(1)   if ( slice is a leaf cell )
(2)         return( dimensions of slice )
(3)   else
(4)         for ( each child of slice )
(5)               Thawing_Shape_Function( child )
(6)               Choose a shape on the shape function.
(7)   if ( depth of slice is ≤ Thaw_level )
(8)         Apply Algorithm 2.1 to compute shape function.
(9)   else
(10)        Apply Algorithm 2.2 to compute shape function.
(11)        Apply squeezing operation for each child slice
            to obtain more accurate shape function.
(12)  return( shape function computed )
```

3.4 The Dynamic Thawing Heuristic

The above heuristic requires a predetermine parameter, *Thaw_level*, which is the level at which the thawing operation begins. From our empirical study of 60 randomly generated slicing layouts we see no direct correlation between Thaw_level and the compactness of the shape function, i.e. a lower Thaw_level does not necessary produce a better shape function. We compare shape functions of a layout produced by the above heuristics with different values of Thaw_level by comparing the area of the smallest shape on each shape function.

The *optimal thawing* strategy is to choose the best Thaw_level for a given layout. This can be done by running the heuristic for each value of Thaw_level. However, this is very inefficient computationally. This motivates the *dynamic thawing* scheme in which a criterion is used to determine if the thawing operation or the rigid computation is applied at the computation of the shape function of a slice.

We will define some terms that we will use in the criterion for dynamic thawing. In a stack of rows of components with known minimum displacements between adjacent components on each row, define the following:

$rminsep_i$ the minimum channel separation of the i^{th} channel if the i^{th} row and $i+1^{st}$ row are rigid.

$tminsep_i$ the minimum channel separation of the i^{th} channel if the i^{th} row and $i+1^{st}$ row are thawable.

mtts $mtts = \sum tminsep_i$.

mtrs $mtrs = \sum rminsep_i$.

The dynamic thawing heuristic decides if the rigid computation (algorithm 2.1) or the thawing computation (algorithm 2.2 + squeezing operation) is applied. From empirical observations we conclude that the thawing computation is beneficial when the *mtts* of a slice is smaller than the *mtrs* of the slice. The explanation of this conclusion is that in a rigid computation channel separations of some routing channels in the slice are at their minima, i.e. $rminsep_i$, and if $tminsep_i$ is much less than $rminsep_i$ at these channels, a better routing dimension can be expected if the thawing computation is applied. Since both $rminsep_i$ and $tminsep_i$ can be computed very efficiently [Mir, Pin], *mtts* and *mtrs* serve as a convenient indicator for when to apply the thawing computation. We do not know an efficient way to decide if thawing will improve the tradeoff dimension of a slice. Therefore, we focus on the channel dimension as our criterion for the dynamic thawing,

$$melting\_point = \frac{mtrs - mtts}{mtrs}.$$

If *melting_point* is greater than a tuned parameter, Thaw_threshold, then the thawing computation will take place. The Thaw_threshold is empirically determined to be 0.10. Formally the dynamic thawing heuristic is described as follows:

Dynamic_Shape_Function(slice)

(1) **if** (slice is a leaf cell)

(2) **return**(dimensions of slice)

(3) **else**

(4) **for** (each child of slice)

(5) **Dynamic_Shape_Function**(child)

(6) Choose a shape on the shape function.

(7) Compute *mtts* and *mtrs*.

(8) $melting\_point = \frac{mtrs - mtts}{mtrs}$

(9) **if** ($melting\_point \leq Thaw\_threshold$)

(10) Apply Algorithm 2.1 to compute shape function.

(11) **else**

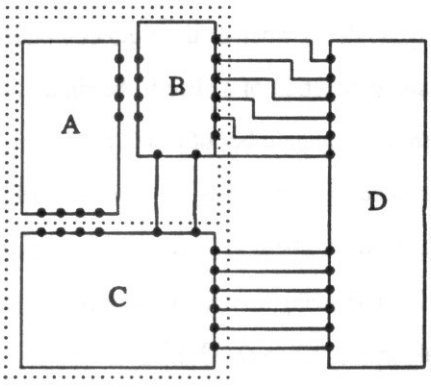
(12) Apply Algorithm 2.2 to compute shape function.

(13) Apply squeezing operation for each child slice
to obtain more accurate shape function.

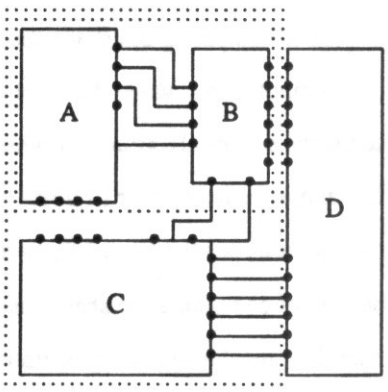
(14) **return**(shape function computed)

3.5 Shape-Choosing Strategy

Choosing good shapes of child slices for the computation of the shape function of their parent slice is a non-trivial task. The shapes of child slices affect the shape of their parent and in turn the shape function of the parent slice affects its parent slice. We will illustrate this by the following example:



Example 3.1a



Example 3.1b

In the above examples, the slice that contains A and B has two shapes, if the shape in Example 3.1a is chosen the composite slice that contains A, B and C requires no routing area. However the composite slice that contains A, B, C and D will require 6 routing tracks. On the other hand if the shape in Example 3.1b is chosen for the slice that contains A and B, extra routing area is required in the composite slice that contains A, B and C, however no routing area is required between the composite slice and D.

The moral of the above example is that good shapes of child slices chosen for their parent slice may not be good shapes for higher level slices that contain them. To consider all combinations of shapes is infeasible. For example, in a stack of n rows, if the average number of shapes to be considered for each row is k, to compute shape functions of the stack for each combination of shapes of the n rows will require k^n computations of shape functions.

The shape-choosing strategy we use is to choose the shape with minimum area. We believe this is a good strategy in terms of producing a shape with minimum area in the final layout. In our empirical study of 60 randomly generated layouts, the rigid heuristic and the dynamic heuristic with the best area shape-choosing strategy produce layouts with areas much smaller than pitch aligning.

3.6 Complexities of the Heuristics

The complexity of each of the three heuristics is the sum of the cost of computing the shape function of each slice over all the slices in a slicing layout. At a visit of a slice, a shape is chosen for each of its child slices. The shape choosing strategy on a slicing layout is linear in the number of break points of the shape function, and the number of break points of a slice's shape function is at most the number of its direct routing nets. The sum of the number of direct routing nets over all the slices is equal to the total routing nets of the slicing layout, n.

The cost of computing the shape function of a slice s, depends on whether the rigid computation or the thawing computation is used. The cost of computing the shape function of a slice if the rigid computation is used is the cost of algorithm 2.1 which is $O(k_i n_i^3)$ where k_i is the number of direct routing channels of the slice, and n_i is the number of direct routing nets of that slice. Notice that k_i is the number of slicing operations that partitions the slice, $\sum k_i$ over all slices is equal to $m-1$ where m is the total number of leaf cells. The cost of computing the shape function of the slice if the thawing computation is used is the cost of algorithm 2.2 plus the squeezing operation. The cost of algorithm 2.2 is $O(n_i^3+n_i^2 \cdot g_i^2)$, where g_i is the number of components involved in the thawing operation of s, i.e. the total number of child slices of child slices of s, they are the child slices which are two levels below s. The cost of the squeezing operation on a slice is equal to the number of direct routing channels of the slice, assuming the child slices two levels

below s can be looked up in $O(1)$ time. The sum of the cost of the squeezing operations over all the slices is $m-1$.

In the dynamic thawing heuristic extra computation is required to determine which of the rigid computation and the thawing computation is used to compute the shape function of a slice. This involves the computation of minimum rigid channel separations and the minimum thawing channel separations of each direct routing channel of the slice. The minimum rigid channel separation of a channel can be computed in linear time in the number of nets of the channel [Mirzaian], and the minimum thawing channel separation can be computed in $O(t \cdot \log t)$ where t is the number of nets of the channel [LePi]. So the total cost of the decision making over all slices is $O(n+n \cdot \log t_{max})$ where t_{max} is the maximum number of nets in a channel over all routing channels.

Given a slicing layout, let

n	the total number of routing nets in the slicing layout.
m	the total number of leaf cells in the slicing layout.
S_r	the set of slices which shape function is computed by the rigid heuristic.
S_t	the set of slices which shape function is computed by the thawing heuristic.
k_i	the number of direct routing channels of slice i
m_r	the number of channels involve in the rigid computation, i.e. $\sum_{i \in S_r} k_i$. $m_r = m-1$ for the rigid heuristic.
n_i	the number of direct routing nets of slice i.
g_i	the total number of child slices two levels below the slice i.
n_{max}	the maximum of n_i over all slices.
n_{rmax}	the maximum of n_i over all slices in S_r.
n_{tmax}	the maximum of n_i over all slices in S_t.
g_{tmax}	the maximum of g_i over all slices in S_t.
t_{max}	the maximum number of nets in each channel over all channels.

The complexity of the rigid heuristic is

$$\textit{cost of algorithm 2.1} + \textit{cost of shape choosing}$$

$$\leq C_1 \cdot \sum_{\textit{all slices}} k_i n_i^3 + C_2 \cdot \sum_{\textit{all slices}} \textit{direct routing nets}$$

$$\leq C_1 \cdot (\sum_{\textit{all slices}} k_i) \cdot n_{\max}^3 + C_2 \cdot n = O(m \cdot n_{\max}^3 + n)$$

for some constants C_1 and C_2.

The complexity of the thawing heuristic is

$$\textit{cost of rigid} + \textit{cost of thawing} + \textit{cost of shape choosing}$$

$$= C_1 \cdot \sum_{i \in S_r} k_i n_i^3 + C_2 \cdot \sum_{i \in S_t} (n_i^3 + g_i^2 \cdot n_i^2) + \textit{cost of squeezing} + C_4 \cdot n$$

$$\leq C_1 \cdot m_r \cdot n_{rmax}^3 + C_2 \cdot (n^3 + g_{tmax}^2 \cdot n^2) + C_3 \cdot n + C_4 \cdot n$$

$$= O(m_r \cdot n_{rmax}^3 + n^2 \cdot (n + g_{tmax}^2) + n)$$

for some constants C_1, C_2, C_3 and C_4.

The complexity of dynamic thawing is similar to that of the thawing with an extra overhead on decision making. The total cost of decision making is $O(n+n \cdot \log t_{\max})$. Therefore the complexity of the dynamic thawing is $O(m_r \cdot n_{rmax}^3 + n^2 \cdot (n + g_{tmax}^2) + n + n \cdot \log(t_{\max}))$.

Although the complexity of the rigid heuristic is $O(m \cdot n_{\max}^3 + n)$, it can be much faster in practice. This depends on two factors. The first factor is the value of $n_{\max}$ with respect to the actual number of nets, n, in the slicing layout. The second is the actual number of potential break points in each individual calculation of shape function of a slice. Recall in chapter 2, $(k_i n_i)$ is the total number of potential break points in the calculation of the shape function, but it can be small in practice. Our empirical results show this is the case. The results are tabulated in section 3.7.

3.7 Performance of Heuristics

In this section we summarize the performance of each heuristic developed above. Sixty depth-one to depth-six slicing layouts were generated randomly, ten for each depth. Typically, a slice contains four to six child slices. See appendix A for the generation of the examples. We compute an approximate shape function for each slicing layout, and the area of the minimum area layout on the shape function is

compared against the area of the layout produced by pitch aligning. Pitch aligning is chosen because it also requires terminals on opposite sides of a channel to have the same net ordering. The comparison is intended to show the potential of the compaction scheme we propose. Table 3.1 presents the total layout area of the sixty slicing layouts produced by each scheme. The total routing area is the total area minus the total area of all leaf cells, i.e. the total non-cell area.

	total layout area	total routing area	normalized layout area	normalized routing area	normalized time
Pitch (P) Aligning	2712870	1925517	1.000	1.000	1.000
Rigid (R) Heuristic	1662586	875233	0.613	0.455	0.920
Dynamic (D) Thawing	1586263	798910	0.585	0.415	25.81
Optimal (O) Thawing	1585368	198015	0.584	0.414	76.00

Table 3.1

Since most of the heuristics were coded for their correctness and are by no means in their most efficient forms, the normalized total running times are provided to illustrate the tradeoff between the time and area. The result of the optimal thawing is obtained by running thawing heuristic for all values of the parameter Thaw_level; for each slicing layout the values of Thaw_level range from one to the level of the root slice. For each slicing layout the Thaw_level that produces the shape function with the smallest minimum area shape is chosen.

Observe that the running time of the rigid heuristic and the running time of pitch aligning are of the same order of magnitude. The running time of pitch aligning is linear in the number of constraints, which is linear in the number of terminals. It is essentially solving the longest path problem in a graph with no positive cycle. See Appendix A. Therefore the actual running time of the rigid heuristic is approximately linear in the number of terminals on the average. This is because the rigid heuristic makes use of the explicit hierarchical structure of the slicing layouts.

Table 3.2 gives the statistics based on individual comparisons. Here the samples are $\frac{X_i}{P_i}$ where X_i is the best area produced by a heuristic for the i^{th} example, and P_i is the area produced by pitch aligning. There are a few examples in which the heuristics actually produced layouts with larger area than the layouts produced by pitch aligning. These only occur for slicing layouts with small depth (1 or 2). From Table 3.2 we see that the dynamic thawing heuristic is as good as the optimal thawing. And it is better than the

thawing heuristic with a fixed value of Thaw_level.

	$E(\frac{X}{P})$ mean	median	standard deviation	min	max
Rigid Heuristic	0.703	0.697	0.142	0.435	1.242
Dynamic Thawing	0.677	0.673	0.135	0.418	1.216
Optimal Thawing	0.677	0.673	0.135	0.418	1.216

Table 3.2

Table 3.3 gives the statistics when the dynamic thawing heuristic and the optimal thawing heuristics are compared with the rigid heuristic for each individual example. The comparison is done for the second half of the sixty slicing layouts, i.e. slicing layouts with depth four or more. This is because the dynamic thawing heuristic is almost equivalent to the rigid heuristic for slicing layouts with small depths, since only very few thawing operations are performed on slicing layouts with small depths. For example, there is no thawing operation for a depth-one slicing layout. Notice in Table 3.3, the number of times Algorithm 2.2 called by the dynamic thawing heuristic is less than that of the optimal thawing heuristic. The much more expensive Algorithm 2.2 is only called by the dynamic heuristic at about 22% of the time. The dynamic thawing heuristic produces layouts 5% smaller than the rigid heuristic but take more time by a factor of 25.

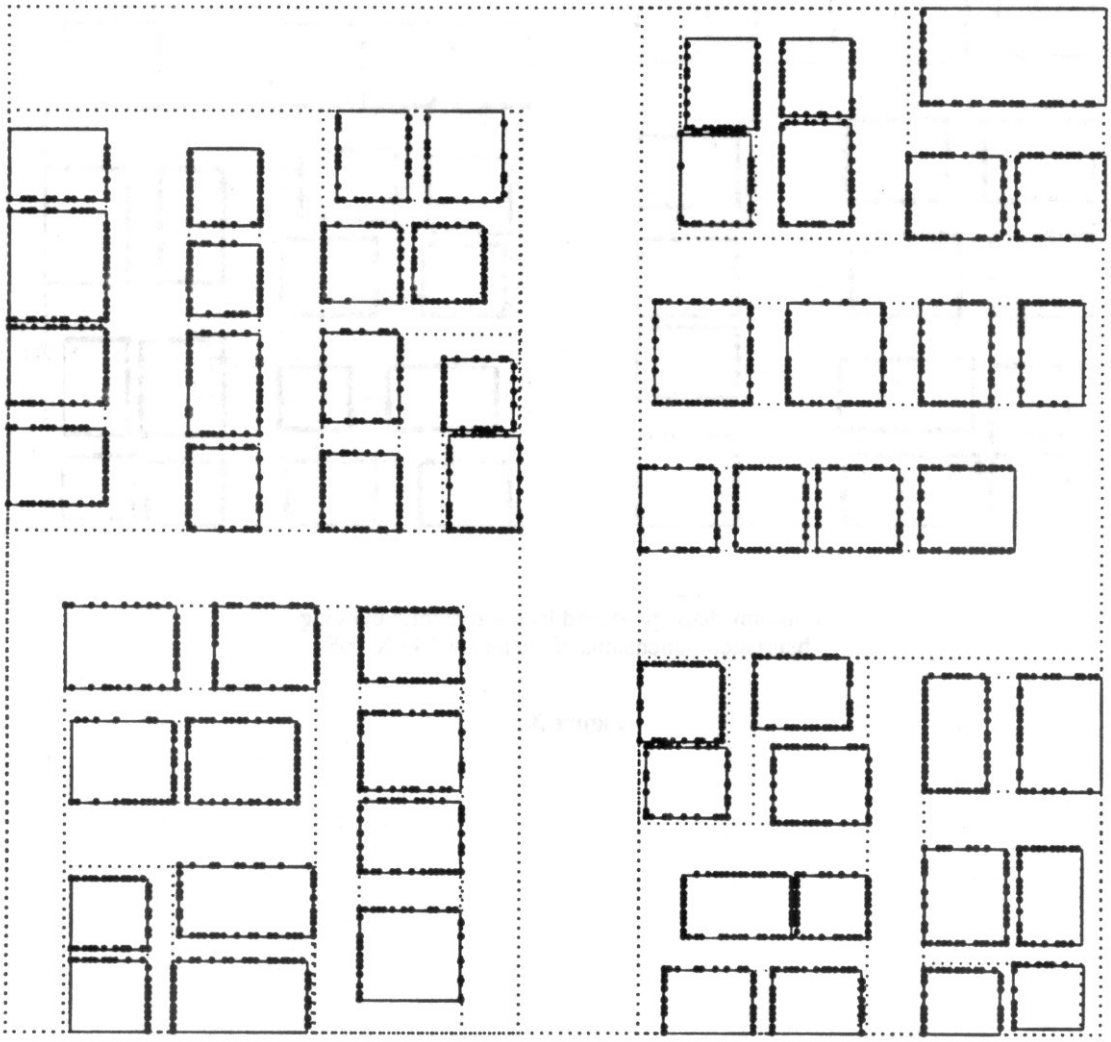
	$E(\frac{X}{R})$ mean	median	standard deviation	min	max	# of Algo. 2.1	# of Algo. 2.2
Dynamic Thawing	0.951	0.960	0.037	0.861	1.000	814	235
Optimal Thawing	0.950	0.960	0.037	0.861	0.994	772	277

Table 3.3

In summary, the hierarchical approach of the rigid heuristic is as efficient as the linear time pitch aligning scheme on the average and it also produces layouts with smaller area than pitch aligning. This shows that treating wires as topological entities during compaction can be beneficial without sacrificing the running time. In addition, if machine time is not a scarce resource, the dynamic thawing can be applied to reduce the routing area in the rigid heuristic by about 5% on the average.

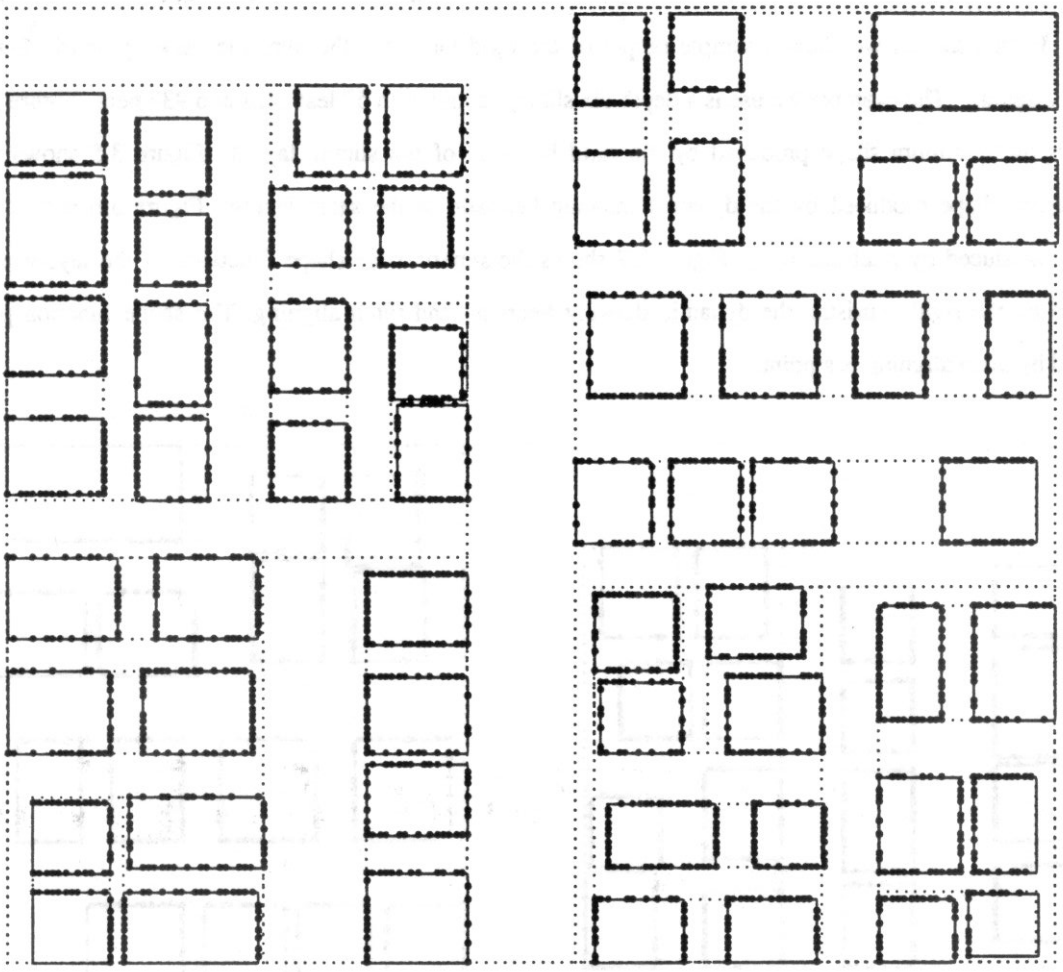
3.8 A Sample Output

In this section we show a sample output of the rigid heuristic, the dynamic thawing heuristic and pitch aligning. The example we use is a depth-six slicing layout with 57 leaf cells and 937 nets. Figure 3.4 shows the minimum shape produced by the rigid heuristic of the slicing layout. Figure 3.5 shows the minimum shape produced by the dynamic thawing heuristic of the same layout. Figure 3.6 shows the shape produced by pitch aligning. Figure 3.7 shows the superimpose shape functions of the layout produced by the rigid heuristic, the dynamic thawing heuristic and pitch aligning. The shape function produced by pitch aligning is a point.



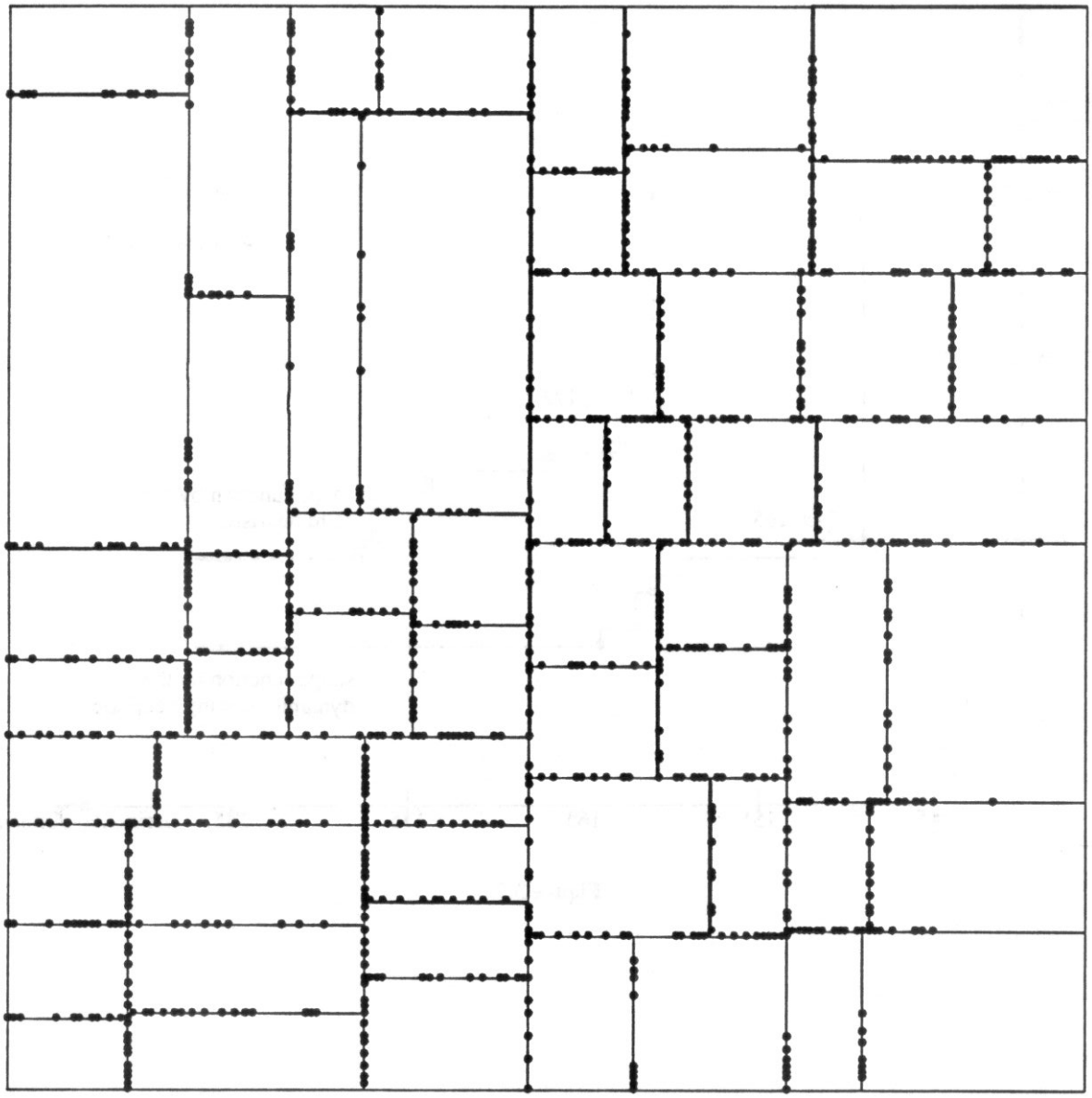
Minimum shape produced by the rigid heuristic
Dimensions of shape are 161 X 172

Figure 3.4



Minimum shape produced by the dynamic thawing heuristic. Dimensions of shape are 149 X 165

Figure 3.5



Shape produced by the pitch aligning scheme
Dimensions of shape are 183 X 184

Figure 3.6

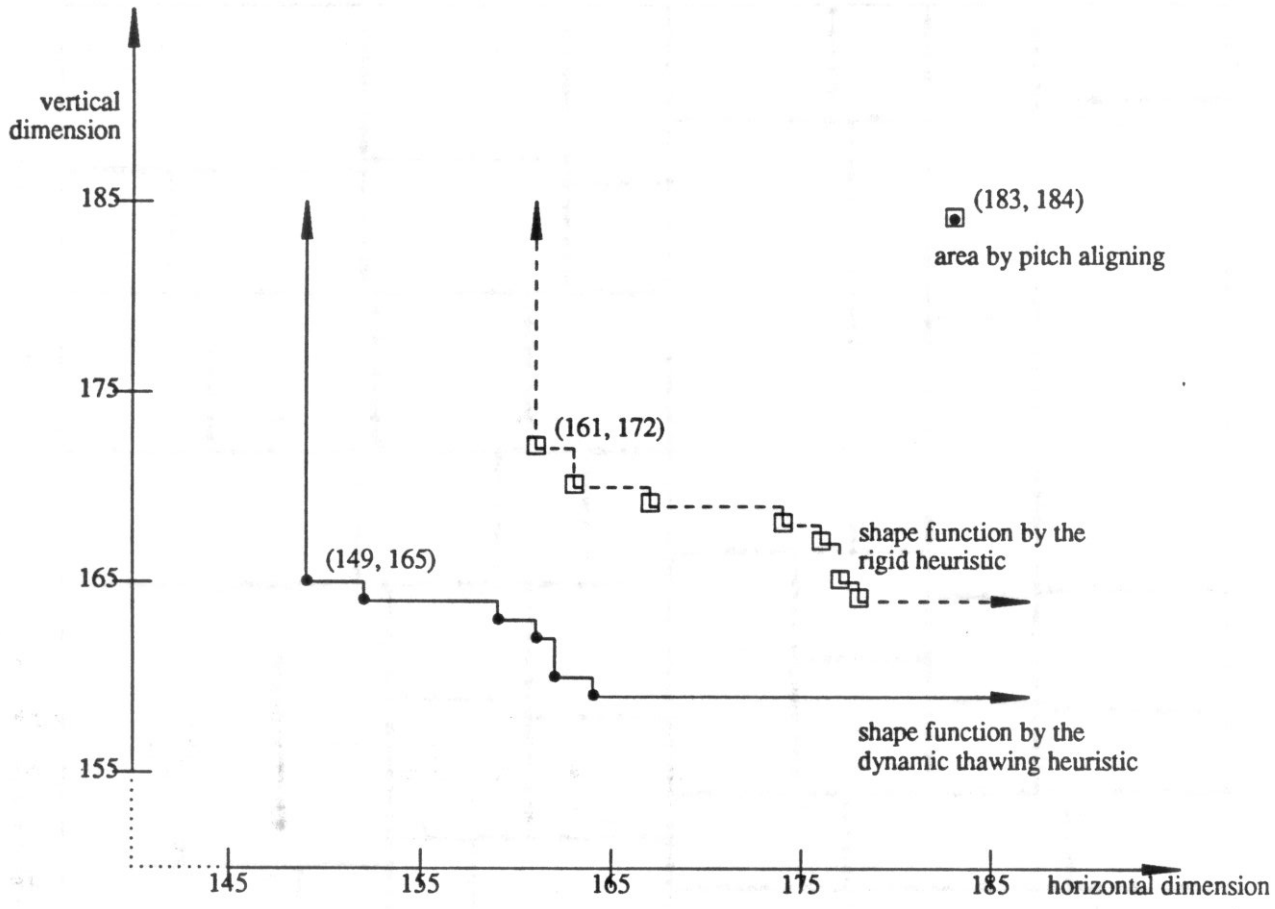

vertical
dimension
185
175
165
155
145
155
165
175
185
horizontal dimension
(183, 184)
area by pitch aligning
(161, 172)
shape function by the
rigid heuristic
(149, 165)
shape function by the
dynamic thawing heuristic

Figure 3.7

Chapter 4

Asymptotic Behavior of River Routing and Pitch Aligning in Uniform Layouts

We address the issues of using the mixture of pitch aligning and river routing cell composition. We study the asymptotic behavior of river routing and pitch aligning in uniform layouts, and we derive the condition under which river routing is better than pitch aligning.

4.1 Introduction

In Chapter 3 we presented three heuristics that compute the approximate shape function of a river slicing layout. The basic skeleton of the heuristics is a postorder traversal on the slicing tree that represents the slicing layout. At a visit of a slice s, the shape function of s is computed by different algorithms assuming that the river routing cell composition scheme is used. During the computation of the shape function of the parent slice of s, the smallest area shape of s is chosen.

If all child slices of s are stretchable, the pitch aligning cell composition scheme can be used to produce a shape for s. The shape produced by the pitch aligning scheme is then compared to the smallest area shape on the shape function of s. The shape with smaller area is then chosen for the shape function computation of the parent slice of s. This gives us a mixture of the pitch aligning and river routing cell composition strategies. Because of the hierarchical nature of the heuristics in Chapter 3, they can be modified easily to include the extra procedure that computes the shape of a slice whose child slices are stretchable.

The modified heuristics would be more efficient if we had some criteria to decide which scheme would give us a better shape without actually computing the shape of a slice using both cell composition schemes. We know of no good criterion that will allow us to tell which scheme is better in general without carrying out the actual computation. But for uniform layouts this may be possible.

In this chapter we will derive bounds for the area of a uniform layout without the actual computation of the shape function or the longest path of the constraint graph that describes the layout. In particular we consider stacks with uniform basic components and two dimensional arrays with uniform basic components. We derive the condition under which river routing is better than pitch aligning. We do this by

comparing an upper bound for the area of a uniform layout using the river routing cell composition scheme to a lower bound for the area of the layout using the pitch aligning scheme.

4.2 Bounds for the Area of a Stack of Uniform Components

We will derive bounds on the area of a stack of uniform river routable components. A stack of uniform river routable components consists of a *basic component* which is replicated many times to produce a stack, and two adjacent components in the stack are river routable. We shall refer to this type of stack as a *uniform stack*. We can use the river routing cell composition scheme to compose the stack. If the basic component is stretchable, the pitch aligning cell composition scheme can also be used.

In a uniform stack with n basic components, when the river routing cell composition is used, we derive an upper bound for the smallest area of the stack. The upper bound is an expression in terms of n and the left and right constraints of two adjacent components. If pitch aligning cell composition is used we derive a lower bound for the area in terms of n and the minimum displacement constraints between terminals of the basic component.

4.2.1. Upper Bound on River Routing

Recall that in Chapter 2 the routing requirement of two river routable components is captured by the left and right constraints of the top component with respect to the bottom. The left and right constraints define an offset range, $[L(t), R(t)]$, for the top component at channel separation t.

Consider two uniform basic components. Let,

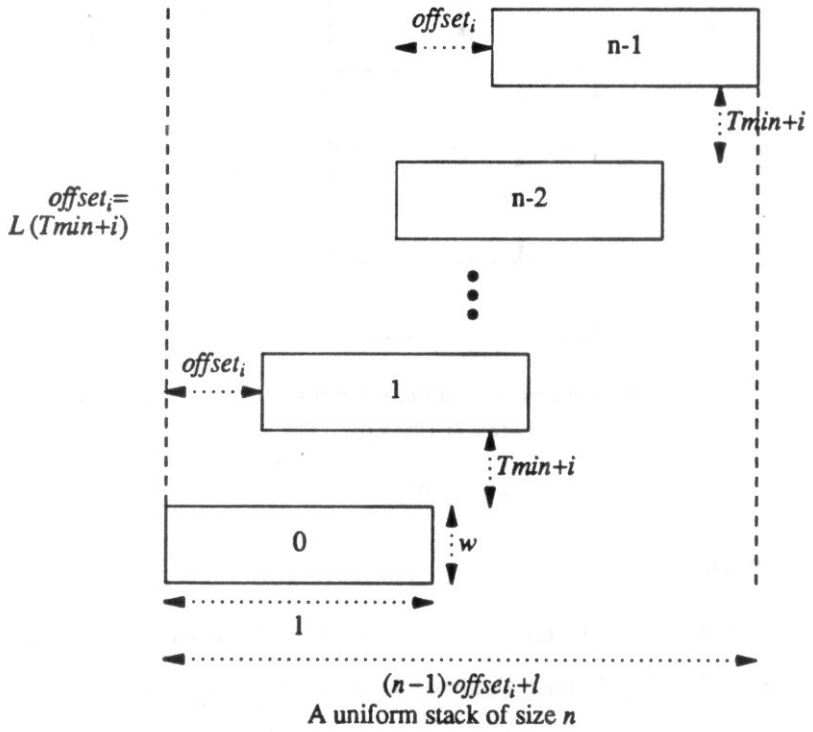
- k be the number of nets between the two components.
- $Tmin$ be the minimum channel separation.
- $Tmax$ be the smallest channel separation such that $L(t) \le 0 \le R(t)$, i.e. the smallest channel separation such that the top and bottom components can be aligned.
- w be the width of the basic component.
- l be the length of the basic component.

In a uniform stack of n components, we want to find bounds for the areas of the stack at different total separations. For the purpose of our analysis, we only find bounds for the areas of the stack at total separations where simple bounds exist. We consider the total separations, $(n-1)\cdot Tmin$, $(n-1)\cdot(Tmin+1)$,..., $(n-1)\cdot Tmax$. Without loss of generality we can assume $0 \le L(Tmin) \le R(Tmin)$. When

$L(Tmin) \le R(Tmin) \le 0$, we can flip the stack and get the same bound. When $L(Tmin) \le 0 \le R(Tmin)$, $Tmax$ is equal to $Tmin$ and we only consider the total separation $(n-1)\cdot Tmax$ which is covered in the following analysis. Let $offset_i$ be the least possible absolute offset of a basic component with respect to the previous component at the channel separation $Tmin+i$. $offset_i = L(Tmin+i)$ for $i = 0,1,...,Tmax-Tmin-1$, and $offset_{Tmax-Tmin}=0$. An upper bound for the area of the stack at the total separation $(n-1)\cdot(Tmin+i)$ is

$$(n\cdot w+(n-1)\cdot(Tmin+i))\cdot((n-1)\cdot offset_i+l)$$

See Figure 4.1.



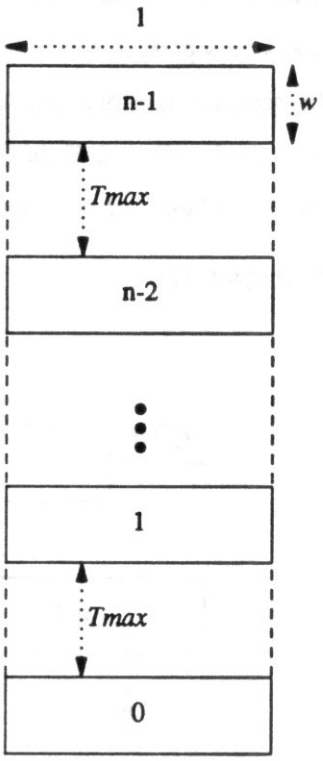
A uniform stack of size n

Figure 4.1

The bound at the total separation $(n-1)\cdot(Tmin+i)$ is established by distributing $Tmin+i$ tracks to each channel. This is a simplistic but sufficient bound for our purpose. In fact, it is not known whether the optimal distribution of the tracks at a given total separation can be computed in time polynomial in $k\cdot\log(l)+\log(n\cdot w\cdot l)$, the length of the input. The smallest upper bound can be used as upper bound for the smallest area of the stack. We choose to use the bound at total separation $(n-1)\cdot Tmax$,

$$(n\cdot w+(n-1)\cdot Tmax)\cdot l$$

See Figure 4.2.



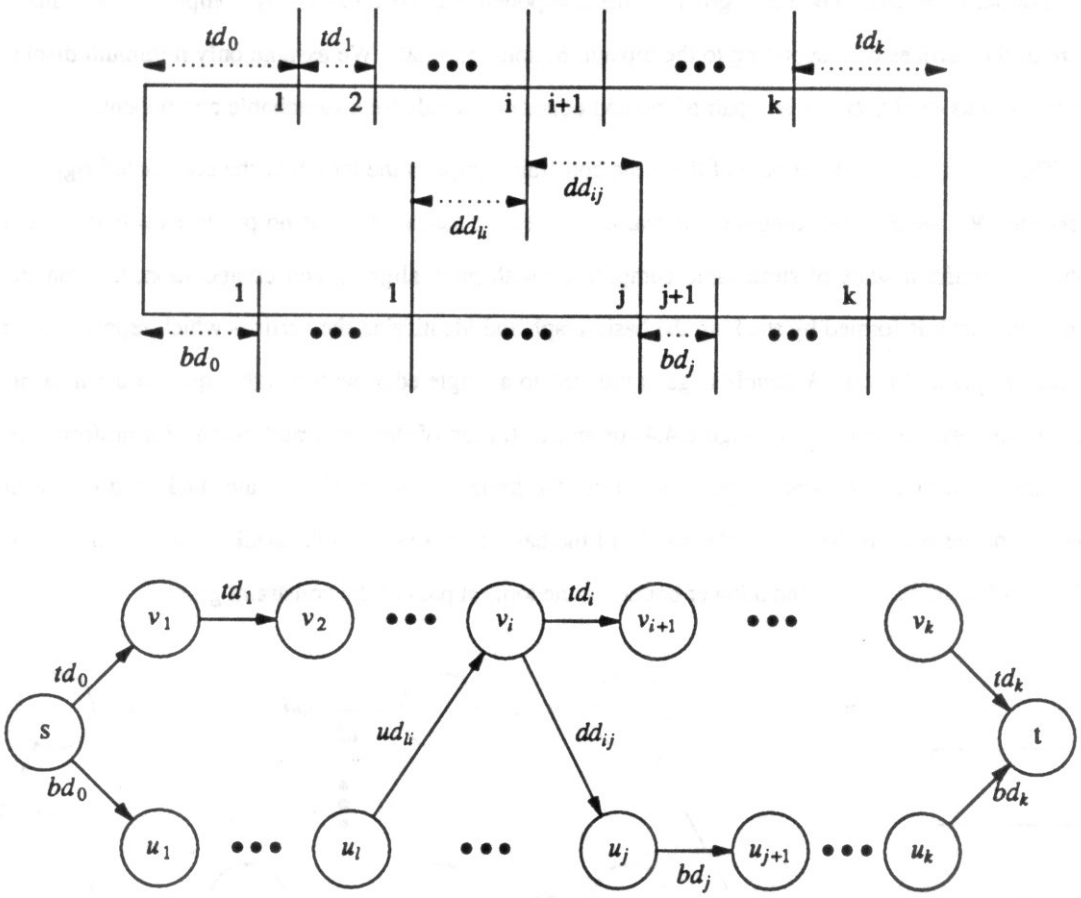
area at total separation $n \cdot Tmax =$
$[n \cdot w + (n-1) \cdot Tmax] \cdot l$

Figure 4.2

4.2.2. Lower Bound on Pitch Aligning

In this section we will derive a lower bound for the area for a uniform stack if the pitch aligning cell composition scheme is used. To apply the pitch aligning scheme, the basic component in the stack has to be stretchable. A stretchable basic component with k terminals on the top boundary and k terminals on the bottom boundary with minimum displacement constraints between neighboring terminals can be described by the commonly used constraint graph shown in Figure 4.3. We call this the *basic graph* of the basic component.

The source s in the basic constraint graph represents the left boundary of the component, the sink t represents the right boundary of the component, vertex v_i represents the i^{th} terminal of the top boundary, vertex u_i represents the i^{th} terminal on the bottom boundary. A directed edge from vertex u to v with a non-negative weight represents a minimum displacement constraint from u to v. Terminals on the top boundary are separated by the minimum displacement constraints $td_0, td_1, \ldots, td_k$, The minimum displacement constraint td_i between the i^{th} terminal and the $i+1^{st}$ terminal is represented in the basic constraint graph by an edge from v_i to v_{i+1} with weight td_i. Similarly terminals on the bottom boundary are separated



A stretchable component and its constraint graph

Figure 4.3

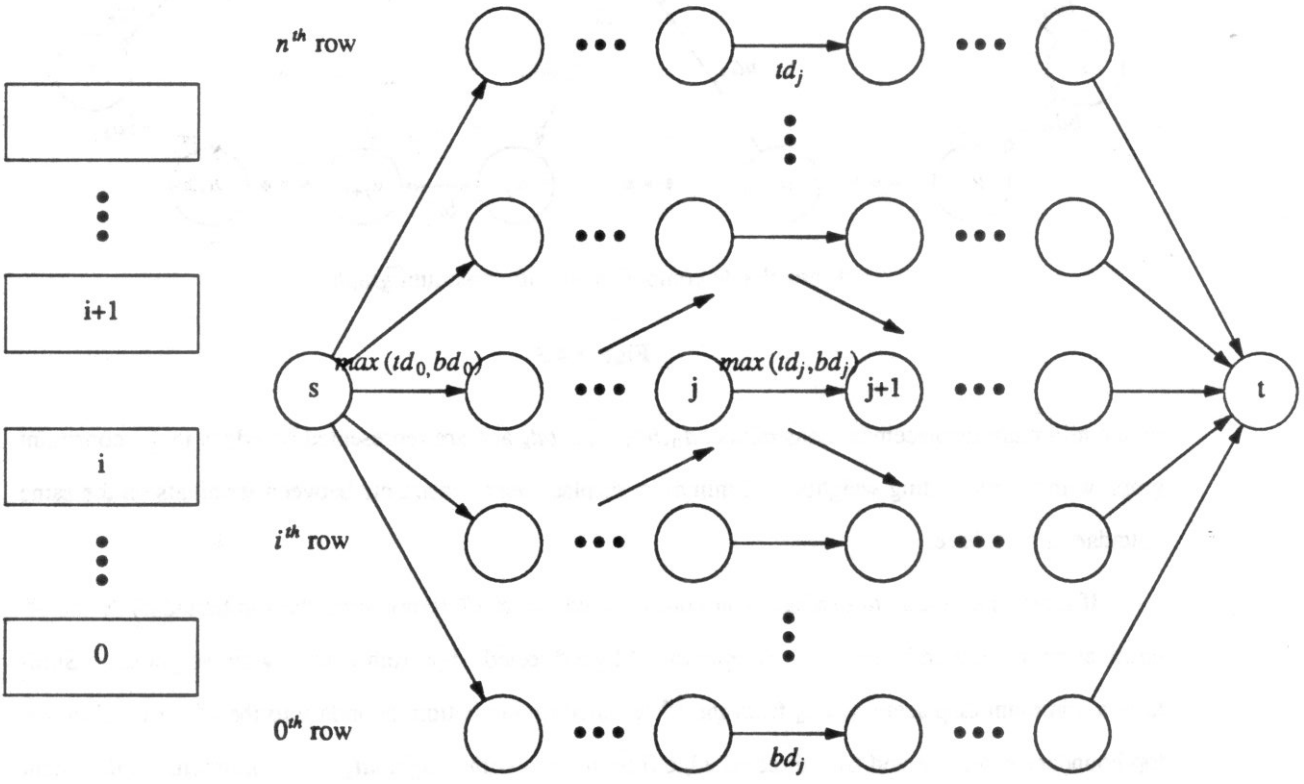
by the minimum displacement constraints $bd_0, bd_1, \ldots, bd_k$ and are represented by edges in the constraint graph with corresponding weights. All minimum displacement constraints between terminals on the same boundary are positive.

If there is a minimum displacement constraint dd_{ij} from i^{th} terminal on the top boundary to the j^{th} terminal on the bottom boundary, it is represented by a directed edge from v_i to u_j with weight dd_{ij}. Similarly a minimum displacement ud_{li} from the l^{th} terminal on the bottom boundary to the i^{th} terminal on the top boundary is represented by a directed edge from u_l to v_i with weight ud_{li}. The minimum displacement constraints between terminals on opposite boundaries are non-negative.

In practice, the horizontal constraint graph of a component describes the horizontal displacement constraints between sub-components of the component (terminals on the boundaries are sub-components of the component). And the constraint graph includes vertices corresponding to sub-components which are not

top or bottom terminals. The basic graph of the component can be obtained by computing the transitive closure of the vertices corresponding to the top and bottom terminals. We assume only minimum displacement constraints exist between any pair of top and bottom terminals for a stretchable component.

The length of the longest path of the basic constraint graph is the length of the compacted rigid basic component. We assume the component is realizable, i.e. we assume there is no positive cycle in the basic graph. In a uniform stack of stretchable components with pitch aligning cell composition, the constraint graph of the stack is formed by stacking the basic graph and identifying the vertices which represent terminals that are pitch aligned. A double edge is reduced to a single edge with weight equal to the maximum weight of the original edges. See Figure 4.4 for an illustration of the constraint graph of a uniform stack. The longest path in the constraint graph determines the horizontal dimension of the stack, and the vertical dimension of the stack is the sum of the widths of the basic components in the stack. To find a lower bound for the area for the stack, we find a lower bound on the longest path of the constraint graph.



A uniform stack with *n* basic components

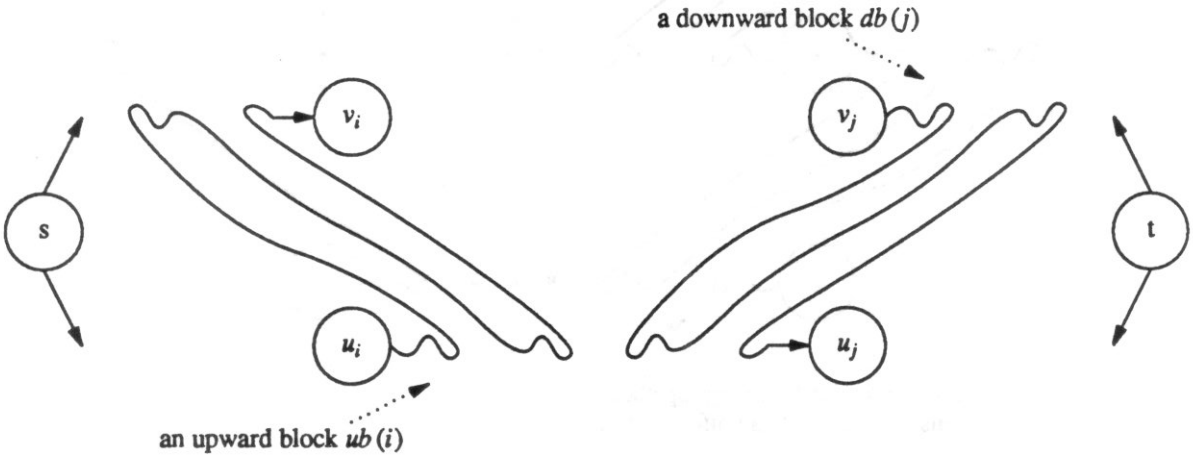
constraint graph of the uniform stack with *n* components

Figure 4.4

The constraint graph of a uniform stack is a graph with a stack of rows of vertices, and vertices in the same row have a natural topological ordering, namely the order of the terminals which the vertices represent. Edges in the graph only exist between vertices in the same row and vertices in the adjacent rows. It is a very regular graph with the basic graph replicated many times. We can exploit the regularity of the constraint graph to find a long path in the graph and use the length of this long path as a lower bound for the actual longest path. The basic idea to find a long path in the constraint graph is to find subgraphs in the basic graph that can be strung together in the constraint graph.

For a uniform stack of n stretchable components, we do not compute the actual longest path of the constraint graph because such computation needs to be carried out for each value of n. This is because the longest path for a given n may not reflect the longest path for another value of n. In fact it is not known whether an expression exists for the longest path of the constraint graph of a uniform stack of n components in terms of the n and the minimum displacement constraints between neighboring terminals of the basic component and can be computed in time polynomial in the length of the input, i.e. $k \cdot log\,(l)+\log(n \cdot l)$, where l is the length of the basic component.

We introduce the notion of a *block* in the basic graph. Intuitively, a block is a subgraph in the basic graph that can be strung together to form a long path in the constraint graph of the uniform stack. An *upward block* in a basic graph is a longest path from u_i to v_i and i is said to be the *index* of the block. A *downward block* is a longest path from v_j to u_j. An upward block with index i is denoted by $ub\,(i)$, a downward block with index j is denoted by $db\,(j)$. See Figure 4.5a.



blocks in a basic graph

Figure 4.5a

In the constraint graph of a uniform stack, an upward block $ub\,(i)$ of the basic graph can be strung together

to form a long path from the bottom row to the top row, this is because $ub(i)$ ends at v_i which is the beginning of an equivalence upward block of the next basic graph. Similarly a downward block $db(i)$ can be strung together to form a long path from the top row to the bottom row. See Figure 4.5b. Note that these blocks may not exist.

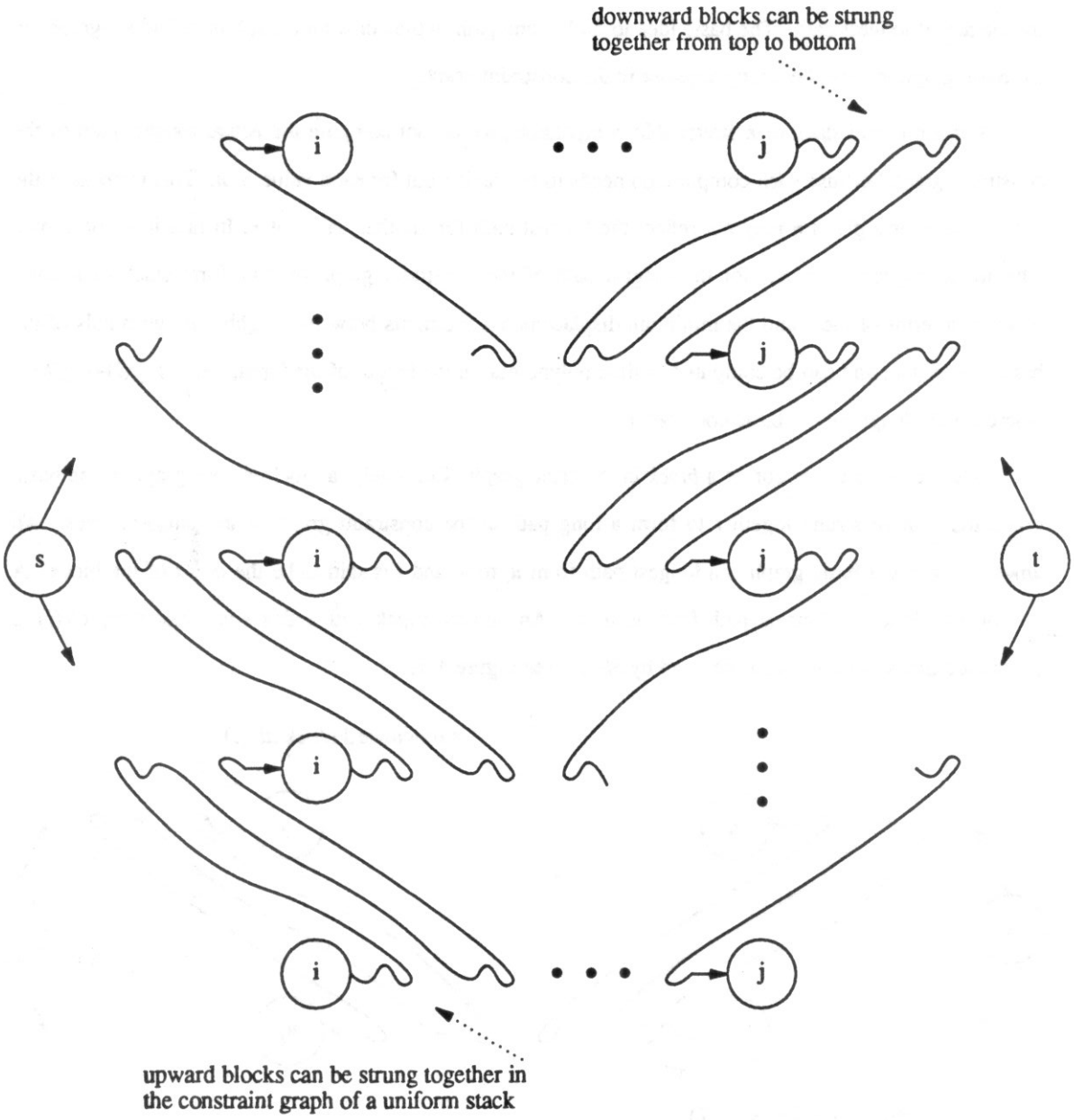


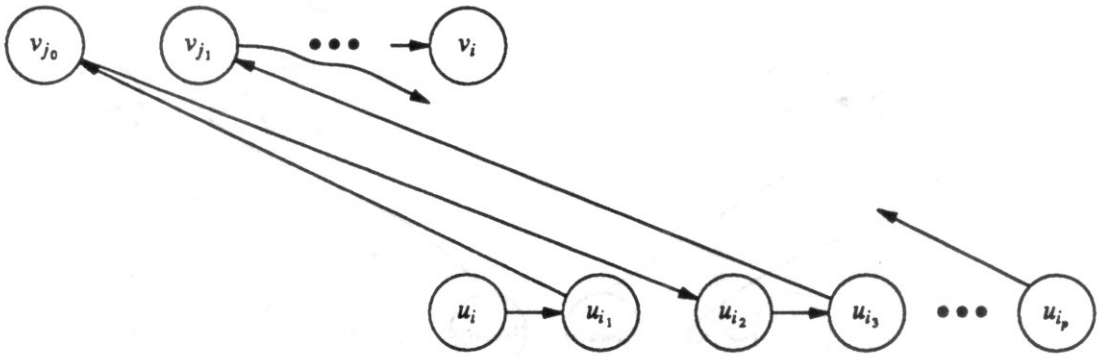
Figure 4.5b

Let $u_i=u_{i_0},u_{i_1},\ldots,u_{i_p}$ be the vertices on the bottom row of the basic graph that appear in $ub(i)$, and they appear in the order that they are specified. Let $v_{j_0},v_{j_1},\ldots,v_{j_q}=v_i$ be the vertices on the top row of the

basic graph that appear in $ub(i)$, then

Lemma 4.1 $i=i_0<i_1<\cdots<i_p$ and $j_0<j_1<\cdots<j_q=i$

Proof. Assume the contrary, then there is $i_l>i_{l+1}$, i.e. there is a non-negative path from u_{i_l} to $u_{i_{l+1}}$ in $ub(i)$, but there is a positive path from $u_{i_{l+1}}$ to u_{i_l}. This give us a positive cycle which contradicts the assumption on the basic graph. Similarly, $j_0<j_1<\cdots<j_q=i$. See Figure 4.6.



An upward block $ub(i)$:

$ub(i)=u_{i_0}\ u_{i_1}\ v_{j_0} u_{i_2}\ u_{i_3}\ v_{j_1}\ \cdots\ u_{i_p}\ \cdots\ v_{j_q}$, where $i_0=j_q=i$

Figure 4.6

Similarly in a downward block $db(i)$, vertices on the same row that belong to the block appear in their topological order. □

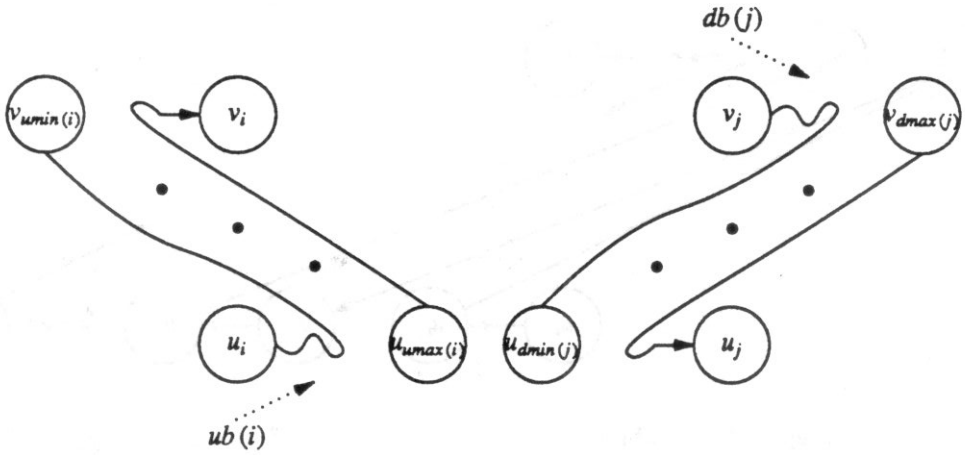
In an upward block $ub(i)$ let $umax(i)$ denotes the largest index of all vertices in $ub(i)$ and $umin(i)$ denotes the smallest index of all vertices in $ub(i)$. From Lemma 4.1, $u_{umax(i)}$ and $v_{umin(i)}$ are both in $ub(i)$. Similarly in a downward block $db(j)$, let $dmax(j)$ denotes the largest index of all vertices in $db(j)$ and $dmin(j)$ denotes the smallest index of all vertices in $db(j)$ and $v_{dmax(j)}$ and $u_{dmin(j)}$ are both in $db(j)$. See Figure 4.7a.

Lemma 4.2 Consider an upward block $ub(i)$ and a downward block $db(j)$. The following are true

(1) if $i<j$ then $umax(i)<dmin(j)$. See Figure 4.7a.

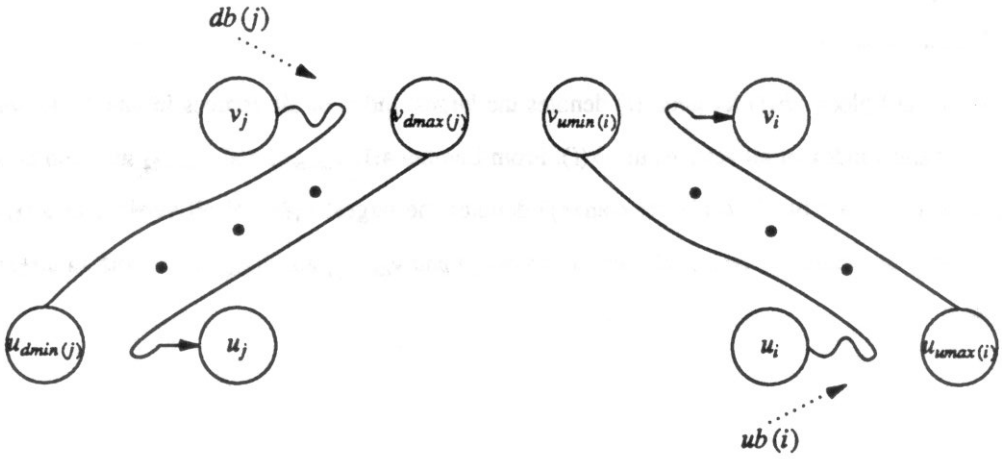
(2) if $i>j$ then $dmax(j)<umin(i)$. See Figure 4.7b.

(3) if $i=j$ then $umax(i)=umin(i)=dmax(j)=dmin(j)=i$. See Figure 4.7c.



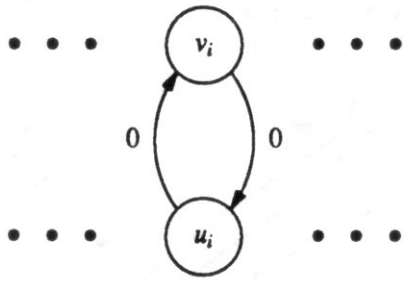
$i < j$

Figure 4.7a



$i > j$

Figure 4.7b



$i = j$

A barrier

Figure 4.7c

Proof.

(1) Assume $umax(i) \geq dmin(j)$. Then there is a non-negative path from $u_{dmin(j)}$ to $u_{umax(i)}$, there is a non-negative path from $u_{umax(i)}$ to v_i, there is a positive path from v_i to v_j, and there is a non-negative path from v_j to $u_{dmin(j)}$. This is a positive cycle in the basic graph which contradicts the assumption on the basic graph.

(2) Similar to (1).

(3) Since there is a non-negative path from u_i to v_i and there is a non-negative path from v_i to u_i, there is a non-negative cycle. For the cycle to be also non-positive, $ub(i)$ is a path of zero weight. Assume there is a third vertex in $ub(i)$, WLOG let the vertex be v_q and $q < i$. Since there is a positive path from v_q to v_i, therefore $ub(i)$ has a positive weight (since $ub(i)$ is a longest path). A contradiction. Hence $ub(i)$ consists of the edge (u_i, v_i) with zero weight. Similarly, $db(i)$ consists of the edge (v_i, u_i) with zero weight. □

Call the type of blocks in (3) of Lemma 4.2 a *barrier*. A barrier is a zero cycle $u_i v_i u_i$. Lemma 4.2 implies that if we order the blocks by their indices, we have a natural partitioning of the sequences of upward and downward blocks, i.e. we have a sequence of blocks with one orientation followed by a sequence of blocks with another orientation and so on. In addition two sequences of blocks with different orientations are either disjoint or they are intersecting at a barrier. We call a maximal sequence (consecutive blocks) of upward blocks an *upward cluster*, and a maximal sequence of downward blocks an *downward cluster*. An upward cluster is followed by a downward cluster and vise versa.

By Lemma 4.2, two adjacent clusters are either disjoint or they are intersecting at a barrier. Assume there are x clusters in the basic graph and the clusters are numbered from 1 to x. Let $c(i)$ denotes the i^{th} cluster. Let $b(i)$ denotes a block with maximum weight in $c(i)$. See Figure 4.8.

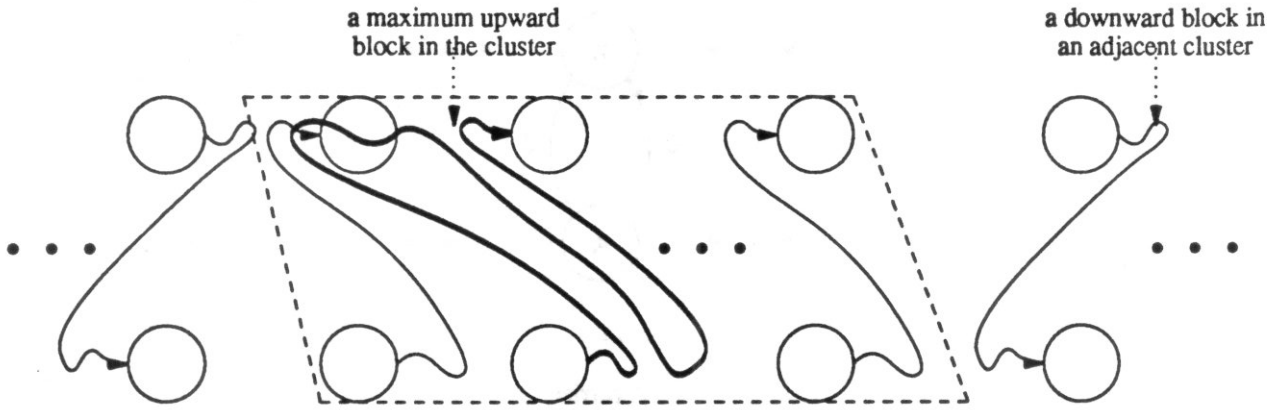


Figure 4.8

The upward and downward blocks in the basic graph can be computed by an all-pairs longest path algorithm. To construct a long path in the constraint graph of an uniform stack, we first compute all the blocks in the basic graph and then compute the clusters $c(1), c(2), \ldots, c(x)$ in the basic graph and the maximum blocks $b(1), b(2), \ldots, b(x)$ in the clusters. If $b(i)$ is an upward block, let $b(i)$ be $ub(b_i)$. Compute the longest path, $lpath(i)$, from v_{b_i} to $v_{b_{i+1}}$. If $b(i)$ is a downward block, let $b(i)$ be $db(b_i)$. Compute the longest path, $lpath(i)$, from u_{b_i} to $u_{b_{i+1}}$. Let $lpath(0)$ be the longest path from the source s to $b(1)$. If $b(1)$ is an upward block, $lpath(0)$ is the longest path from s to u_{b_1}, if $b(1)$ is a downward block, $lpath(0)$ is the longest path from s to v_{b_1}. Similarly, $lpath(x)$ is the longest path from $b(x)$ to the sink t. See Figure 4.9.

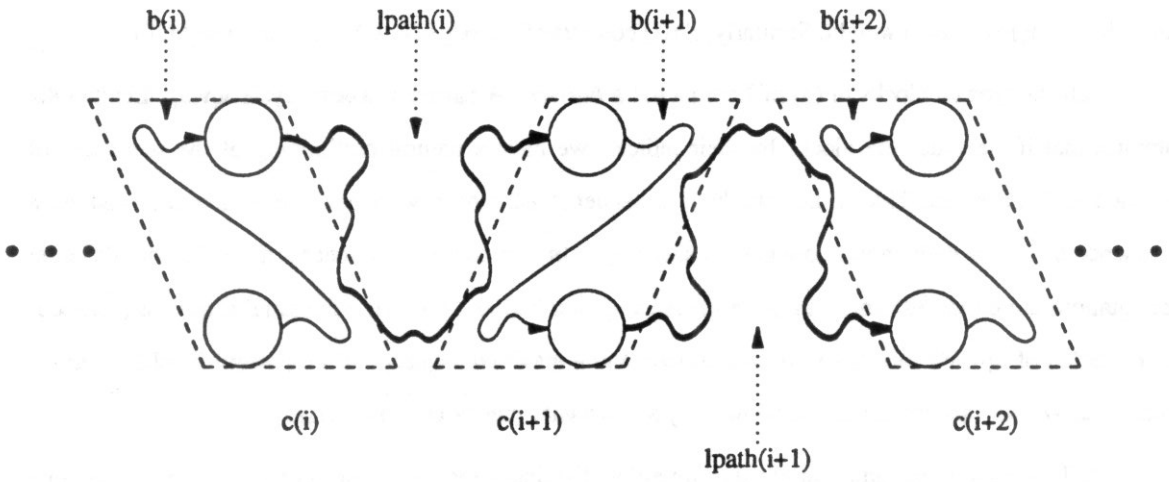
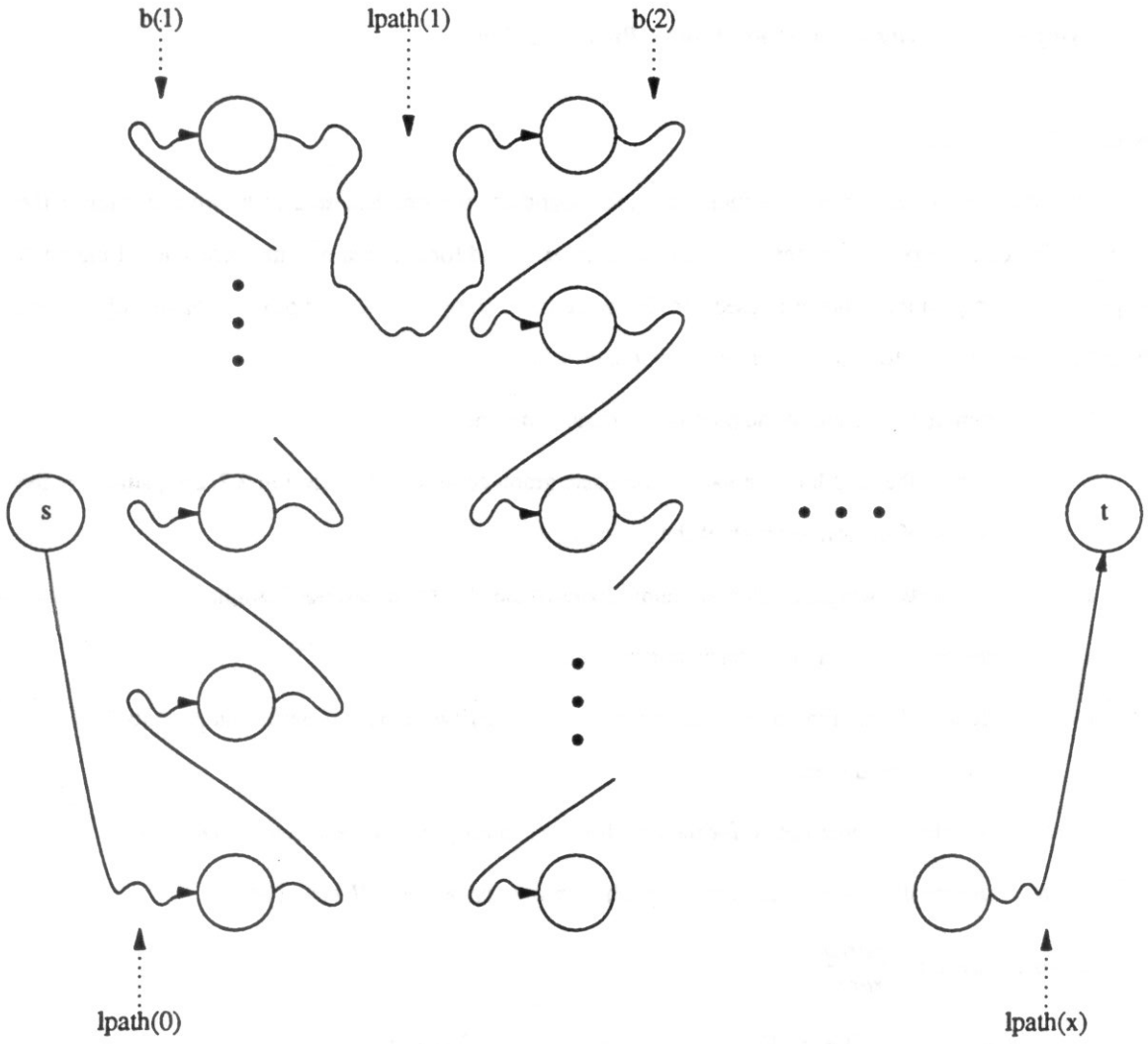


Figure 4.9

Now we begin to construct a long path in the constraint graph of a uniform stack with n components using $lpath(0), lpath(1), .., lpath(x)$, and $b(1), b(2), ..., b(x)$. Assume the first cluster is an upward cluster. The long path consists of the following pieces, $lpath(0)$, n copies of $b(1)$ strung together from bottom to top row, $lpath(1)$, n copies of $b(2)$ strung together from top to bottom row, and so on. See Figure 4.10.



a long path in the constraint graph

Figure 4.10

Let $W(path)$ denotes the length of *path*. The length of the long path constructed above is

$$\sum_{i=0}^{i=x} W(lpath(i)) + n \cdot \sum_{i=1}^{i=x} W(b(i))$$

Observe that the concatenation of $lpath(0)$, $b(1)$, $lpath(1)$, $b(2)$,..., $b(x)$, $lpath(x)$ is a path in the basic graph. Let L be the length of this path. Let $B = \sum_{i=1}^{i=x} W(b(i))$ Then the length of the long path becomes

$L+(n-1)\cdot B$. In a uniform stack of n components, with the pitch aligning cell composition scheme, the vertical dimension of the stack is $w\cdot n$ where w is the width of the basic component. Thus, a lower bound for the area of the stack is $w\cdot n\cdot(L+(n-1)\cdot B)$.

4.3 Asymptotic Behavior of River Routing vs. Pitch Aligning

4.3.1 Uniform Stack

We derived, in the previous section, an upper bound for the smallest area of a uniform stack if the river routing cell composition scheme is used and a lower bound for the area of a uniform stack if the pitch aligning cell composition scheme is used. In this section we compare the asymptotic behavior of the two bounds. Consider a uniform stack with n components. Let

- l denote the length of the compacted basic component.
- L denote the length of the path in the basic graph constructed in section 2.2. $L \le l$, since l is the length of the longest path of the basic graph.
- B denote the weight of all maximum blocks in the clusters of the basic graph.
- w denote the width of the basic component.
- $Tmax$ denote the minimum channel separation between two adjacent basic components such that they can be aligned.
- $rarea$ denote the upper bound for the area for river routing. $rarea=(n\cdot w+(n-1)\cdot Tmax)\cdot l$.
- $parea$ denote the lower of the area for pitch aligning. $parea=n\cdot w\cdot(L+(n-1)\cdot B)$.

Consider the ratio $\frac{parea}{rarea}$,

$$\frac{parea}{rarea}=\frac{n\cdot w\cdot(L+(n-1)\cdot B)}{(n\cdot w+(n-1)\cdot Tmax)\cdot l}=\frac{n\cdot w}{n\cdot w+(n-1)\cdot Tmax}\cdot\frac{(L+(n-1)\cdot B)}{l}$$

$$\approx(n-1)\cdot\frac{w}{w+Tmax}\cdot\frac{B}{l}$$

for large n and $B>0$. The ratio holds when $\frac{(n-1)}{l}\cdot B \gg \frac{L}{l}$. Since $\frac{L}{l}<1$, the ratio holds when $\frac{(n-1)}{l}\cdot B>1$, i.e. when $(n-1)>\frac{l}{B}$. The ratio is growing with n, this means river routing scheme is better than the pitch aligning scheme for a uniform stack with large number of components. $parea$ begins to exceed $rarea$ when

$$(n-1)\ge\frac{w+Tmax}{w}\cdot\frac{l}{B}$$

Intuitively, the blocks in the basic graph represent the portions of the basic component which will be stretched. And the stretching propagates to produce a layout with large area.

When $B=0$, there are no stretchable portions of the basic component that will propagate. Let *Hdim* denotes the horizontal dimension, the length of the longest path in the constraint graph, of a uniform stack with n components. The ratio $\frac{parea}{rarea}$ becomes,

$$\frac{parea}{rarea}=\frac{n\cdot w}{n\cdot w+(n-1)\cdot Tmax}\cdot\frac{Hdim}{l}$$

Tmax is the channel separation of a stack of two basic components at minimum horizontal dimension (when they aligned). Since $B=0$, there is no upward or down block in the constraint graph, we can compute the longest path of the constraint graph of a stack of two basic components and use this as a lower bound for *Hdim*. We can then use the ratio $\frac{parea}{rarea}$ as a guideline on when to use pitch aligning and when to use river routing cell composition.

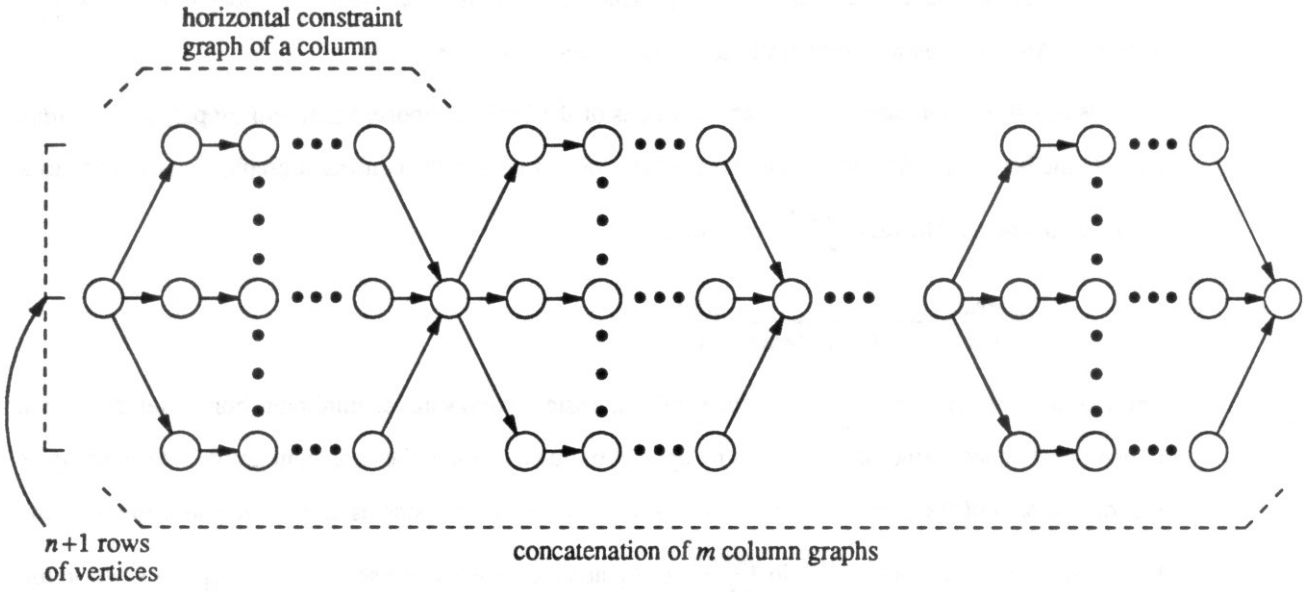
In summary, when $B>0$ river routing cell composition is superior to pitch aligning cell composition for large uniform stacks. When $B=0$, the performance the two schemes is determined by the value of n, w, l, *Tmax* and *Hdim*.

4.3.2 Uniform Two Dimensional Array

Consider a $n\times m$ array and assume the array is sliced vertically first and then horizontally. The column graph, the horizontal constraint graph of a column of the array, is similar to the constraint graph of a uniform stack. The horizontal constraint graph of the array is formed by placing the column graphs in a sequence and identifying the sources and sinks of adjacent graphs. See Figure 4.11. A long path in a column graph can be computed the same way we compute the long path in the constraint graph of a uniform stack. Recall that when we compute the long path of a uniform stack, we compute $lpath(0),...,lpath(x)$, $b(1),...,b(x)$, where $b(i)$ denote the weight of the maximal block in the i^{th} cluster of the basic graph, and $lpath(i)$ is the length of a path that link the i^{th} and the $i+1^{st}$ maximal block, $lpath(0)$ is the length of a path from the source to $b(0)$ and $lpath(x)$ is the length of a path from $b(x)$ to the sink. Let B_h be the sum weight of the maximal blocks and L_h be the length of the long path that links all the blocks.

A long path of the horizontal constraint graph of the array can be obtained by concatenating the long paths in the column graphs. Thus a lower bound for the horizontal dimension of the array is,

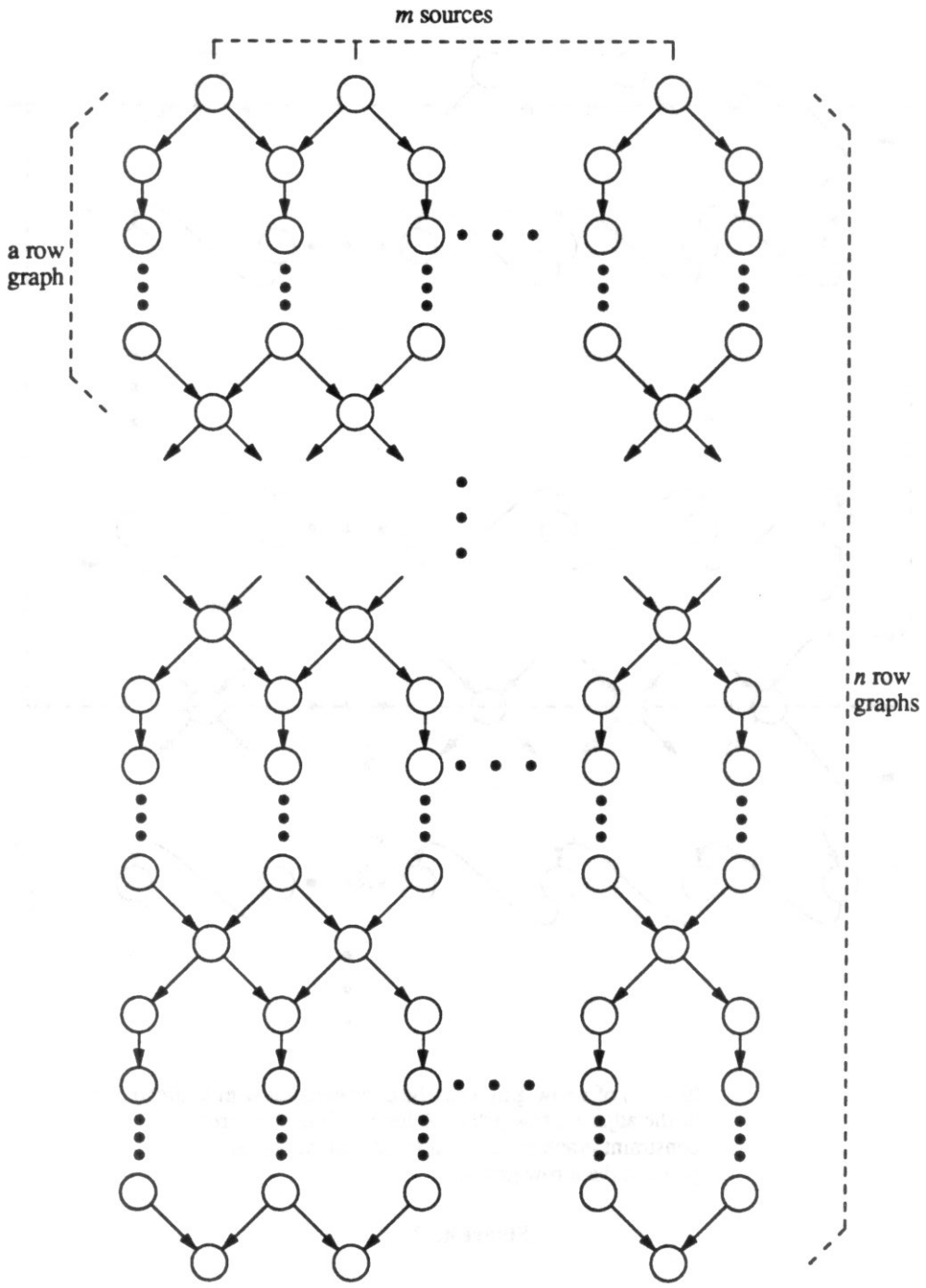
$$m\cdot(L_h+(n-1)\cdot B_h)$$



Horizontal constraint graph of an
$n \times m$ array of uniform components

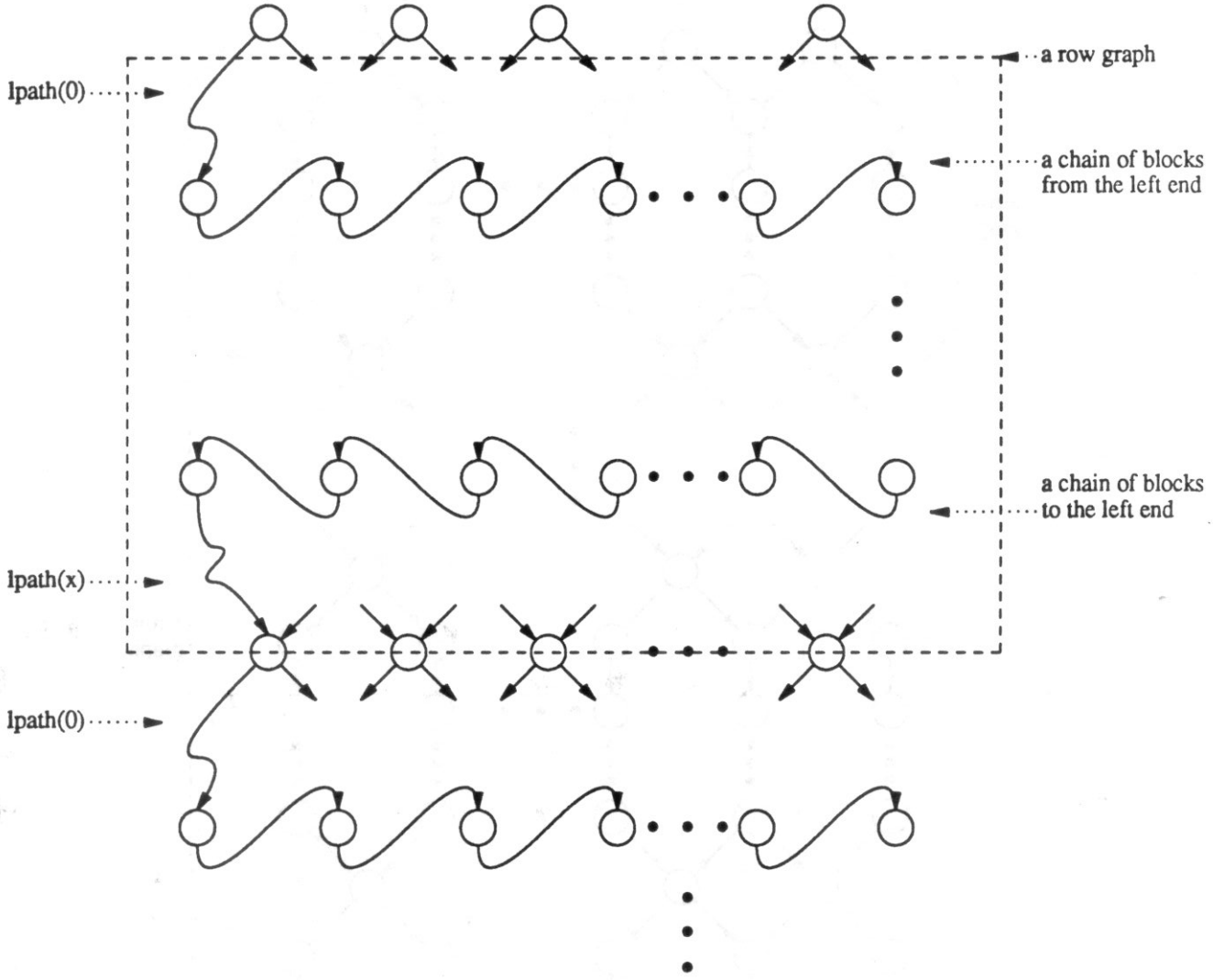
Figure 4.11

Now consider the vertical constraint graph of the array. It is constructed by concatenating the vertical constraint graph of each column. The following is another equivalent construction which is more suitable for our analysis. The row graph, the vertical constraint graph of a row of the array, is similar to the constraint graph of a uniform stack rotated ninety degrees, but instead of one source and one sink it has m sources and m sinks. See Figure 4.12. The vertical constraint graph of the array can be formed by stacking up the row graphs and identifying the sources and sinks of adjacent row graphs. See Figure 4.12. A long path in a row graph can be computed the same way we compute the long path in a column graph; we compute the sequence of maximal blocks and the paths that link the adjacent blocks. If the first maximal block and the last maximal block have opposite orientations (one is to the left and one is to the right or vise versa), so they end up at the same column and a long path of the vertical constraint graph can be constructed by concatenating the long paths of the row graphs as illustrated in Figure 4.13. If the first block and the last block have the same orientation, we can eliminate the first (or the last block) by treating it as part of $lpath(0)$ (or $lpath(x)$), and we get a sequence of maximal blocks such that the first and the last block have opposite orientation. A long path of the vertical constraint graph can then be constructed similarly. Let B_v be the sum weight of the sequence of maximal blocks of a row graph (first and last block have opposite orientation), and L_v be the length of the path that links all the blocks. Then a lower bound for the



vertical constraint graph of an $n \times m$ array of uniform components

Figure 4.12



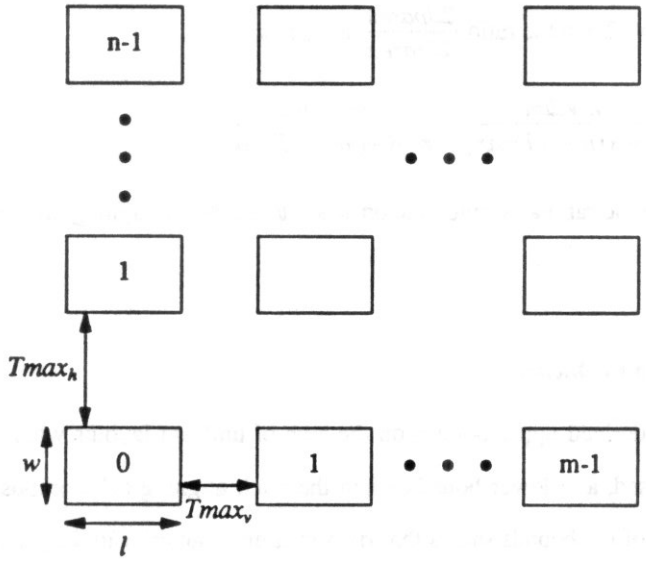
lpath(x) of a row graph can be concatenated with lpath(0) of the adjacent row graph. A long path of the vertical constraint graph is obtained by concatenating all long paths in the n row graphs

Figure 4.13

vertical dimension of the array is,

$$n\cdot(L_v+(m-1)\cdot B_v)$$

Let, $Tmax_h$ denotes the minimum horizontal channel separation such that the adjacent components can be aligned horizontally, and let $Tmax_v$ denotes the minimum vertical channel separation such that the adjacent components can be aligned vertically. See Figure 4.14.



A uniform $n \times m$ array

Figure 4.14

With the same approach as in section 2.1, we can derive an upper bound for the smallest area of the array if the river routing cell composition scheme is used. Let *2drarea* denotes the upper bound, then

$$2drarea=(n\cdot w+(n-1)\cdot Tmax_h)\cdot(m\cdot l+(m-1)\cdot Tmax_v)$$

Let *2dparea* denotes the lower bound for the area of the array when the pitch aligning cell composition is used. Then,

$$2dparea=n\cdot(L_v+(m-1)\cdot B_v)\cdot m\cdot(L_h+(n-1)B_h)$$

The ratio $\frac{2dparea}{2drarea}$ when n and m are large and B_v, $B_h>0$ is then,

$$\frac{2dparea}{2drarea}=\frac{n\cdot(L_v+(m-1)\cdot B_v)\cdot m\cdot(L_h+(n-1)B_h)}{(n\cdot w+(n-1)\cdot Tmax_h)\cdot(m\cdot l+(m-1)\cdot Tmax_v)}$$

$$\approx\frac{(L_v+(m-1)\cdot B_v)}{w+Tmax_h}\cdot\frac{(L_h+(n-1)\cdot B_h)}{l+Tmax_v}$$

$$\approx(n-1)\cdot(m-1)\cdot\frac{B_v}{w+Tmax_h}\cdot\frac{B_h}{l+Tmax_v}$$

Again the ratio is growing with n and m. This shows again that river routing scheme is better than the pitch aligning scheme for uniform two dimensional array. If $B_v=0$ and $B_h>0$ the ratio grows with n, if $B_h=0$ and $B_v>0$ the ratio grows with m. If $B_h=B_v=0$, let *Hdim* denotes the length of the longest path of a column graph, and *Vdim* denotes the length of the longest path of a row graph from i^{th} source to i^{th} sink for some

$i \le n$, $Hdim \ge l$ and $Vdim \ge w$. Then the ratio $\frac{2dparea}{2drarea}$ becomes,

$$\frac{2dparea}{2drarea} = \frac{n \cdot Vdim}{n \cdot w + (n-1) \cdot Tmax_h} \cdot \frac{m \cdot Hdim}{m \cdot W + (m-1) \cdot Tmax_v}$$

In such a case we can use the ratio as a guideline on when to use pitch aligning and when to use river routing.

4.4 Conclusion and Open Problems

In this chapter we derived upper bounds on the area of uniform layouts when the river routing cell composition scheme is used, and lower bounds when the pitch aligning cell composition scheme is used. The asymptotic behavior of the bounds shows that river routing is superior to the pitch aligning scheme for uniform layouts with a large number of basic components if stretching causes terminals to spread from bottom to top. For uniform layouts with a small number of basic components, the bounds can be used as guidelines to determine which cell composition scheme should be used.

In the derivation of the bounds we used very crude estimates to get the bounds that are sufficient for our purpose. An open problem is to exploit the regularity of the uniform layouts to compute the exact shape functions for uniform layouts more efficiently. Initial work on this has been done by Iwano [Iwa] who considers the problem of solving the longest path problem of regular graphs in time independent of the size of the regular graphs. He proposes a method to generate the longest path of a regular graph based on the longest path of a subgraph of the regular graph. He shows the condition when such a subgraph exists and shows how large must the subgraph be. For the case of the river routing cell composition scheme, it is not known whether an expression exists for the shape function of a uniform layout in terms of the number of basic components in a layout and the left and right constraints between adjacent basic components. If the expression exists, it is interesting to know if it can be computed in time polynomial in the length of the input. For the case of the pitch aligning cell composition scheme, it is not known whether an expression exists for the area of the uniform layout in terms of the number of basic components in the layout and the minimum displacement constraints between neighboring terminals of the basic component, again we would like to compute such an expression in time proportional to the length of the input.

Chapter 5

Shape of Density Stack

In this chapter we consider the computation of the shape function for a stack of single components using channel density as an estimate of channel width. We show that to compute such a shape function is NP-complete and provide a good heuristic to compute an approximate shape function.

5.1. Introduction

In the previous two chapters we considered the shape functions of river slicing layouts. In a river slicing layout all routing channels are river routable, i.e. terminals on opposite sides of a channel have the same net ordering. In this chapter we address the problem of computing the shape functions of more general slicing layouts (routing channels may not be river routable).

In a general routing channel the channel width, the minimum number of tracks required to route the channel, is difficult to compute. In fact, Szymanski showed that computing the channel width under the rectilinear, two reserved layer routing model is NP-complete [Szy]. It is unlikely that the channel width can be computed efficiently. Instead of computing the channel width, a metric known as the *channel density* is often used to estimate the actual width. Intuitively, the channel density measures the minimum number of necessary net crossings at any grid line perpendicular to the channel. Formally, in a channel with n nets, given the positions of terminals (top and bottom) in the channel, the channel density d is defined as,

$$d = \max_x \left\{ \sum_{i=0}^{i=n-1} f_i(x) \right\}$$

where,

$$f_i(x) = \begin{cases} 1 \text{ if } l_i \le x \le r_i \text{ and } l_i \ne r_i \\ 0 \text{ otherwise} \end{cases}$$

and l_i is the horizontal position of the leftmost terminal of net i, r_i is the horizontal position of the rightmost

terminal of net i.

In general, the channel density is a fairly good estimate of the channel width. In fact Rivest and Fiduccia claimed that their channel routing algorithm usually uses no more than one more track than the channel density [RiFi]. Theoretical results relating the channel density and the channel width are described in [Lei].

In this chapter, we shall use the channel density as an estimate of the actual channel width, the minimum number of tracks required to route the channel. We consider the problem of computing the shape function of a stack of components using channel density as channel width. We show that such a problem is NP-complete. We present a pseudo-polynomial time algorithm and an efficient heuristic for the problem. We compare the performance of the heuristic against the pseudo-polynomial time algorithm. Finally, we discuss the problem of computing the shape functions of slicing layouts using the heuristic as a basic sub-routine.

5.2 The Density Function

In a two-component channel with n nets, the i^{th} terminal on the bottom edge of the top component is given as an ordered pair (t_i, n_i) where t_i denotes the position of the terminal from the left edge of the top component and n_i denotes the net in which the terminal belongs. Similarly, the i^{th} terminal on the bottom component is given as (b_i, n_i). For a given offset w, the displacement of the left edge of the top component with respect to the left edge of the bottom component, the absolute position of the i^{th} top terminal, (t_i, n_i), is t_i+w. We assume the left edge of the bottom component is at absolute position 0; therefore the absolute position of the i^{th} bottom terminal is b_i. At offset w, the leftmost position of net i, $l_i(w)$ and the rightmost position of net i, $r_i(w)$ are well-defined and the *density function* is defined as,

$$D(w) = \max_x \left\{ \sum_{i=0}^{i=n-1} f_i(x,w) \right\}$$

where

$$f_i(x, w) = \begin{cases} 1 & \text{if } l_i(w) \le x \le r_i(w) \text{ and } l_i(w) \ne r_i(w) \\ 0 & \text{otherwise} \end{cases}$$

See Figure 5.1. Notice that only the leftmost and rightmost terminals of a net are relevant in the density calculation. We preprocess the terminals and only keep track of four (or fewer) terminals of each net, namely the leftmost and rightmost terminals of the net in the top component, the leftmost and rightmost terminals of the same net in the bottom component. These four (or fewer) terminals of a net i are sufficient to

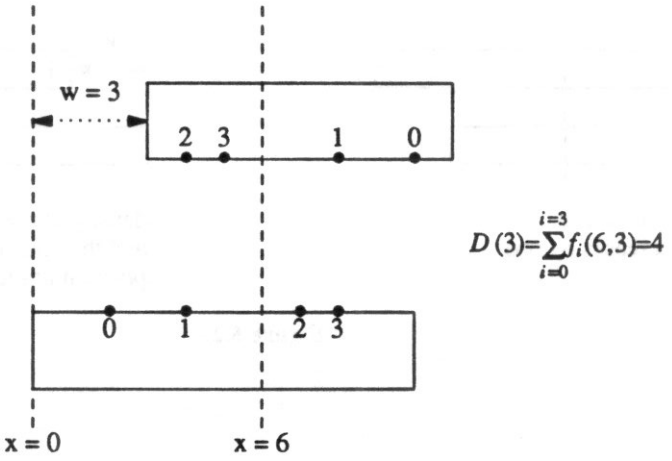


Figure 5.1

define $l_i(w)$ and $r_i(w)$ at a given offset w. Therefore we only have to keep track of at most $4 \cdot n$ terminals, $2 \cdot n$ on the top component and $2 \cdot n$ on the bottom component. The preprocessing time is $O(m_t+m_b)$ where m_t is the number of terminals on the top component, and m_b is the number of terminals on the bottom component. We shall consider the density computation after the preprocessing and shall express the computation time in terms of n.

At a given offset w, $f_i(x, w)$ changes from 0 to 1 as x increases from $x<l_i(w)$ to $l_i(w)$ and $f_i(x, w)$ changes from 1 to 0 as x increases from $l_i(w) \le x \le r_i(w)$ to $x>r_i(w)$. To compute $D(w)$, we sort the list $\{l_0(w),..., l_{n-1}(w), r_0(w),..., r_{n-1}(w)\}$ and scan the channel from left to right, update the density (the sum of $f_i(x, w)$) at the positions in the sorted list in order, and remember the maximum. Let p be the current position being scanned, the density at p and the position immediately to the right of p is updated as follows,

(i) Only one terminal is at p. If $p=l_i(w)$ for some i, the density increases by one; if $p=r_i(w)$ it remains unchanged at p and then decreases by one, otherwise the density stays the same. See Figure 5.2a.

(ii) Two terminals are at p. If the terminals belong to the same net, say i and they are the only two terminals of the net, then the density remains unchanged. If net i has more than two terminals, and if $p=l_i(w)$ then the density increases by one, if $p=r_i(w)$ then it and remains unchanged at p and then decreases by one. If the terminals belong to different nets, say i and j, if $p=l_i(w)=l_j(w)$ then the density increases by two, if $p=r_i(w)=r_j(w)$ then the density remains unchanged at p and then decreases by two. If $p=l_i(w)=r_j(w)$ the density increases by one at p and then decreases by one. See Figure 5.2b.

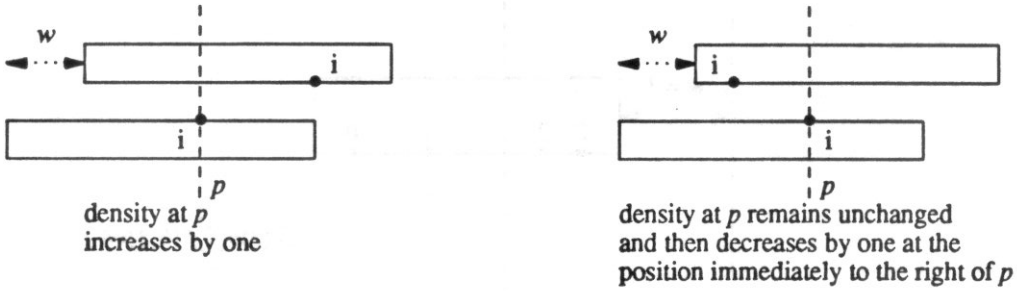
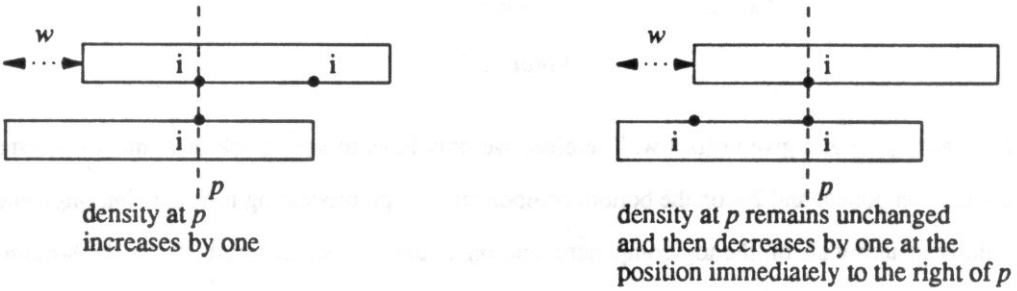
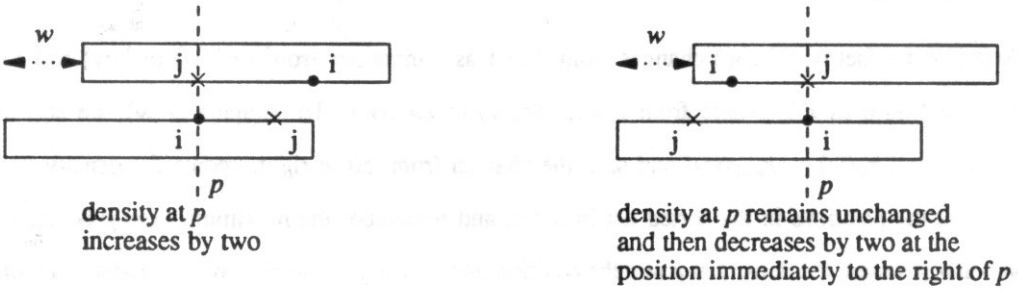


Figure 5.2a





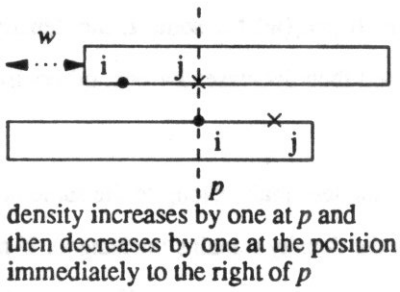
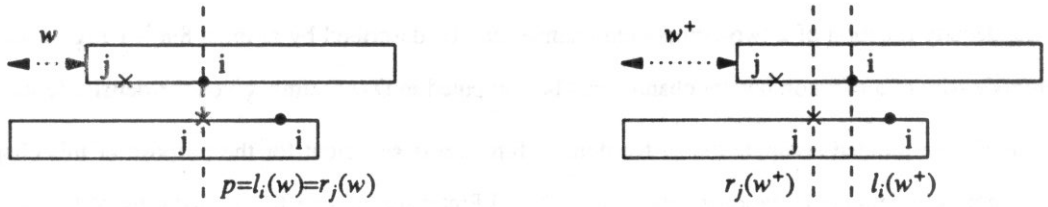


Figure 5.2b

This can be done in $O(n \cdot log(n))$ time.

Let w denote an offset in which alignments of terminals occur and w^+ denote an offset between w and the next offset in which alignments of terminals occur. While computing $D(w)$ we also compute

$D(w^+)$. In our grid model, there may be no legal w^+, but the computation of $D(w^+)$ is necessary for the updating. The updating of $D(w^+)$ differs from $D(w)$ only when the following cases occur at the current scanning position p. Case (i): there are two terminals of different nets at p, say the top terminal is in net i and the bottom terminal is in net j, and $p=l_i(w)=r_j(w)$. The densities of offset w^+ at $l_i(w^+)$ and $r_k(w^+)$ are one less than the density of offset w at $p=l_i(w)=r_j(w)$. See Figure 5.2c.

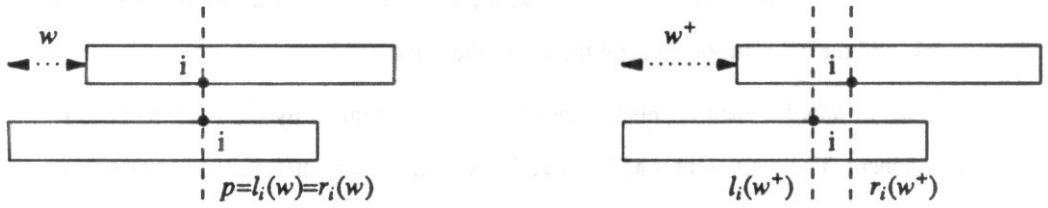


$$\sum_{l=0}^{l=n-1} f_l(p,w) - 1 = \text{density of offset } w^+ \text{ at } r_j(w^+) = \sum_{l=0}^{l=n-1} f_l(r_j(w^+), w^+)$$

$$= \text{density of offset } w^+ \text{ at } l_i(w^+) = \sum_{l=0}^{l=n-1} f_l(l_i(w^+), w^+)$$

Figure 5.2c

Case (ii): there are two terminals of the same net, say net i, at p and they are the only two terminals of the net. In this case the density of offset w^+ between $r_i(w^+)$ and $l_i(w^+)$ is one more than the density of offset w at $p=l_i(w)=r_i(w)$. See Figure 5.2d.



$$\sum_{l=0}^{l=n-1} f_l(p,w) + 1 = \sum_{l=0}^{l=n-1} f_l(r_j(w^+), w^+)$$

$$= \sum_{l=0}^{l=n-1} f_l(l_i(w^+), w^+)$$

Figure 5.2d

Therefore $D(w^+)$ can be updated with little extra accounting.

There are at most $4 \cdot n^2$ offsets at which alignments of terminals occur. Therefore only $4 \cdot n^2$ offsets are of interest, and the density function can be stored in at most $8 \cdot n^2$ space, $4 \cdot n^2$ at the offsets w of interest and $4 \cdot n^2$ at w^+. A straightforward method to compute the density function is to compute the $4 \cdot n^2$ offsets and compute $D(w)$ and $D(w^+)$ at each of the $4 \cdot n^2$ offsets in order. We can sort the $4 \cdot n^2$ offsets in

$O(n^2 \cdot \log(n))$ time. The sorting of the list $\{l_0(w),..., l_{n-1}(w), r_0(w),..., r_{n-1}(w)\}$ need only be done once, since we can update the list when we goes from one offset to the next. The total time required by the simple-minded computation is $O(n^3)$, since the update at each offset can be done in $O(n)$ time except the first one. The density function is defined over the range $(-\infty, +\infty)$. And $D(w) = n$ for w greater than the length of the bottom component or w less than the negative length of the top component, since all n nets have to cross the vertical column at the left ($w<0$) or right ($w>0$) boundary of the bottom component. Since the density function of a two-component channel can be described by at most $8 \cdot n^2+1$ segments, the minimum density of a two-component channel can be computed in $O(n^2)$ time, given the density function.

The simple-minded computation of the density function is sufficient for the purpose of this chapter although more efficient computation exists. LaPaugh and Pinter use more efficient updating of $D(w)$ from one offset to the next. They present an $O(n^2 \cdot \log(n))$ algorithm to compute the density function in [LaPi].

5.3 The Shape of Density Stack

In this section we describe the problem of computing the shape function of a stack of single components using channel density as channel width and show that the problem is NP-complete. The *shape of density stack problem* is stated as follows:

Given,

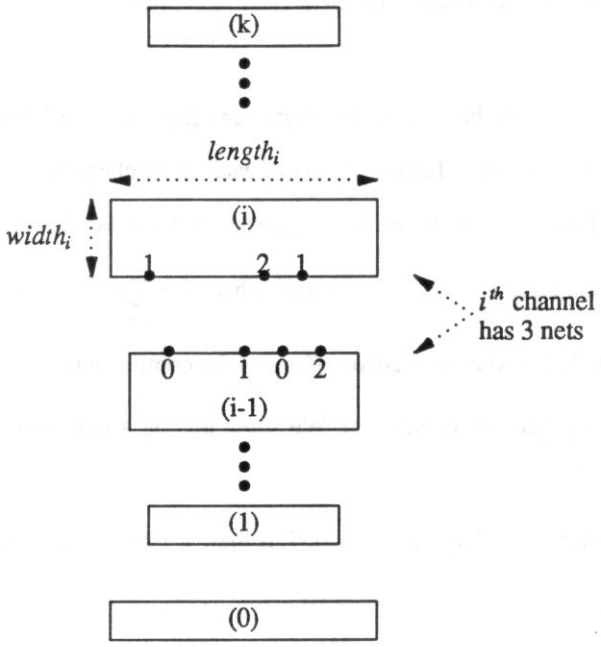
(1) A stack of $k+1$ components, $0, 1, \ldots, k$, the 0^{th} component is at the bottom of the stack, and the k^{th} component is on the top of the stack.

(2) The set of nets for each channel. The i^{th} channel is defined by the $i-1^{st}$ and the i^{th} component. There are n_i nets and m_i terminals in the i^{th} channel. Each net has at least two terminals. The terminals are distributed on the bottom edge of the i^{th} component and top edge of the $i-1^{st}$ component. The position of a terminal with respect to the left edge of the component it is on and the net in which the terminal belongs are given.

(3) The length $length_i$ and width $width_i$ of the i^{th} component.

Compute the shape function of the stack of $k+1$ components using channel density as channel width. See Figure 5.3.

Since we are using the channel density to approximate the actual width of a channel, from here on, by density of a channel we also mean the actual vertical dimension of the channel. A configuration of a stack of $k+1$ components is specified by a density vector $(d_1, d_2, \ldots, d_k)$ and an offset vector



an instance of the shape-of-density-stack
problem with $k+1$ components

Figure 5.3

$(w_1, w_2, \ldots, w_k)$ such that d_j is the channel density of the i^{th} channel and w_i is the offset of the $i+1^{st}$ component with respect to the i^{th} component for i from 1 to k. Let $D_i(w)$ denotes the density function of the i^{th} channel. A configuration of a stack of $k+1$ components is said to be *legal* if $D_i(w_i) \leq d_i$ for i from 1 to k. The *total density* of a configuration is the total of all channel densities. A shape $(s, d)_k$ describes a set of legal configurations of a stack of $k+1$ components, whose tradeoff dimension does not exceed s and whose total densities do not exceed d. A shape $(s, d)_k$ is said to be *legal* if it contains at least one legal configuration.

We now show that the shape of density stack problem is NP-complete by showing the following decision problem, Stack of Density, is NP-complete:

Given a stack of $k+1$ components as in the shape of density stack problem, a total separation d and a trade-off dimension s, is $(s, d)_k$ legal ?

Theorem 5.1 The Stack of Density problem is NP-complete.

Proof. (i) The stack of density problem is in NP. Guess an offset for the i^{th} component with respect to the $i-1^{st}$ component for $i = 1,..., k$. The channel density of each channel can be computed in $O(n_i \cdot log(n_i)+m_i)$ time where n_i is the number of nets and m_i is the number of terminals in the i^{th} channel. Therefore the total density can be computed in $O(n \cdot log(n)+m)$ time where $n = \sum_{i=1}^{i=k} n_i$ and $m = \sum_{i=1}^{i=k} m_i$. The tradeoff dimension can be computed in $O(k)$ given the relative offsets of the components.

(ii) The stack of density problem is NP-hard. We show this by a reduction from PARTITION [GaJo]. An instance of PARTITION:

Given a set of positive integer $S = \{ a_1, a_2, \ldots, a_k \}$, is there a subset $A \subseteq S$ such that

$$\sum_{a_i \in A} a_i = \sum_{a_i \in S-A} a_i \, ?$$

Construct the following instance of stack of density problem:

Let, $L = \sum_{i=1}^{i=k} a_i$, $M = \max_{a_i \in S} \{ a_i \}$.

0^{th} component:

$length_0 = 2 \cdot L + M$,

top terminals : $(L, 1)$, $(L+a_1, 2)$,

bottom terminals : $\varnothing$

i^{th} component: $(i = 1,..., m-1)$

$length_i = 2 \cdot (\sum_{j=1}^{j=i} a_j) + M$,

top terminals : $(0, 1)$, $(a_{i+1}, 2)$,

bottom terminals : $(a_i, 2)$, $(2 \cdot a_i, 1)$

k^{th} component:

$length_k = 2 \cdot (\sum_{j=1}^{j=k} a_j) + M = 2 \cdot L + M$,

top terminals : $\varnothing$,

bottom terminals : $(a_k, 2)$, $(2 \cdot a_k, 1)$

Is $(2 \cdot L+M, k)_k$ legal ? See Figure 5.4.

× : terminal of net 1 in a channel

• : terminal of net 2 in a channel

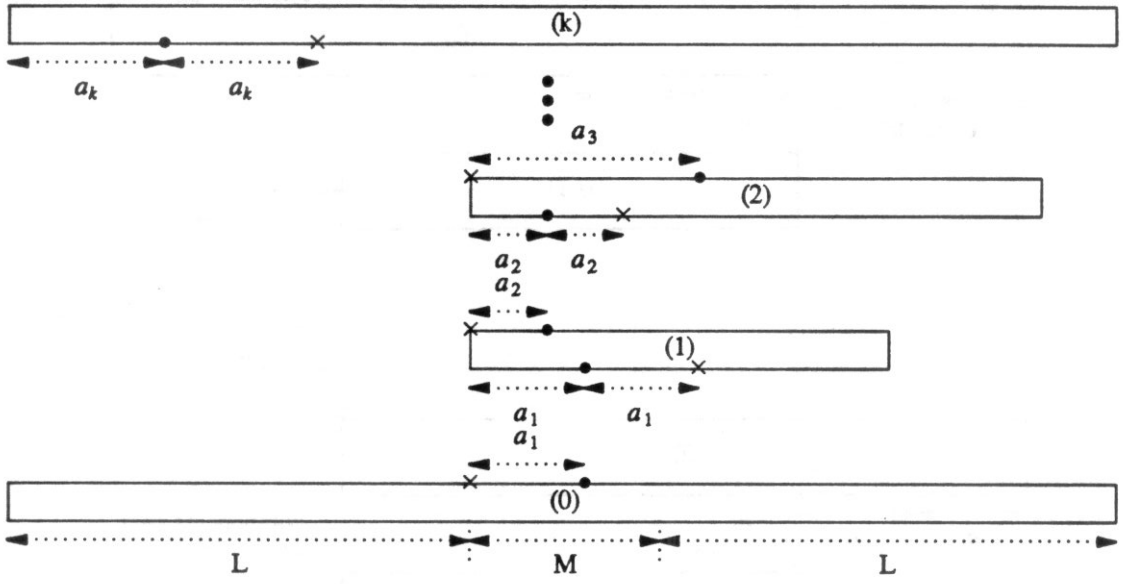


Figure 5.4

Observe that the minimum channel density of each channel is 1. For $i = 2,...,k$, the i^{th} channel achieves density 1 when the i^{th} component is $2 \cdot a_i$ units to the left of the $i-1^{st}$ component in which case they are aligned on the right, or when the two components are aligned on the left in which case the right end of the i^{th} component is $2 \cdot a_i$ units to the right of the right end of the $i-1^{st}$ component. See Figure 5.5. Suppose $(2 \cdot L+M,\ k)_k$ is legal; choose a legal configuration in $(2 \cdot L+M,\ k)_k$. Notice that at total density k, the channel density at each channel is 1. Let, l_i denote the position of the left end of the i^{th} component to the left of the left end of the $i-1^{st}$ component for $i = 2,...,\ k$. And l_1 denotes the position of the left end of the 1^{st} component to the left of position L of the 0^{th} component. For $i = 2,...,\ m$,

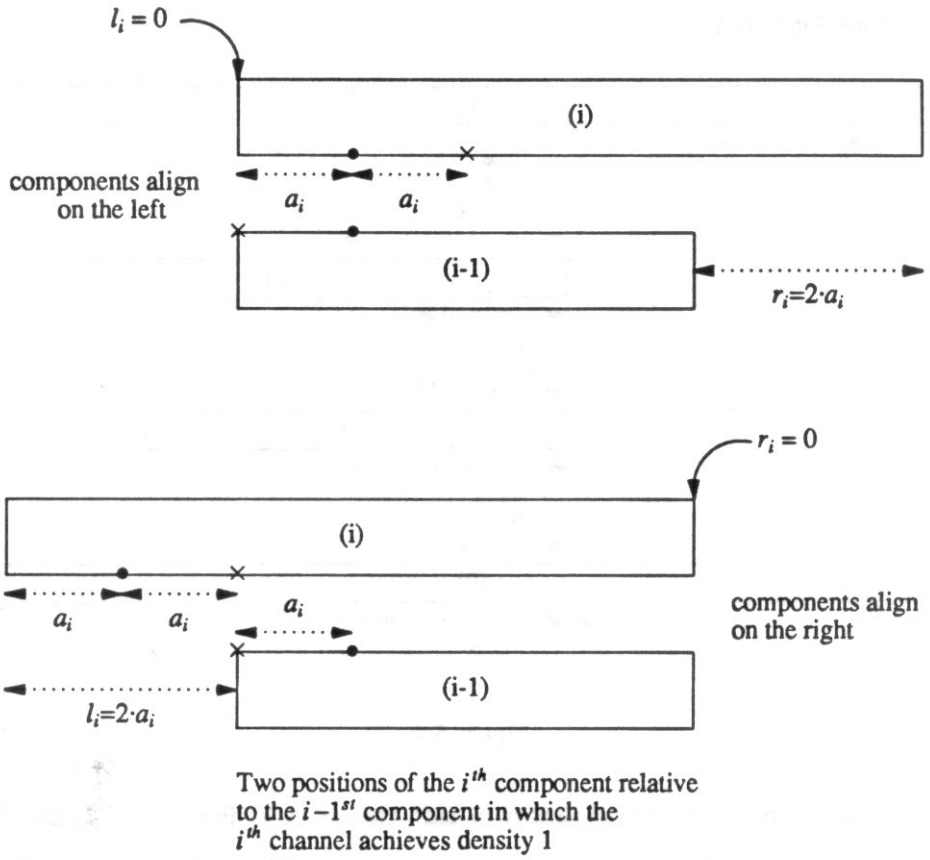
$$l_i = \begin{cases} 0 & i^{th} \text{ and } i-1^{st} \text{ component align on the left} \\ 2 \cdot a_i & i^{th} \text{ and } i-1^{st} \text{ component align on the right} \end{cases}$$

and,

$$l_1 = \begin{cases} 0 & 1^{st} \text{ component at position } L \\ 2 \cdot a_1 & 1^{st} \text{ component at position } L-2 \cdot a_1 \end{cases}$$

See Figure 5.5.

Let, r_i denote the position of the right end of the i^{th} component to the right of the the right end of the $i-1^{st}$ component for $i = 2,...,\ m$. And r_1 denotes the right end of the 1^{st} component to the right of position $L+M$



Two positions of the i^{th} component relative to the $i-1^{st}$ component in which the i^{th} channel achieves density 1

Figure 5.5

of the 0^{th} component. For $i = 2,..., k$,

$$r_i = \begin{cases} 2 \cdot a_i & i^{th} \text{ and } i-1^{st} \text{ component align on the left} \\ 0 & i^{th} \text{ and } i-1^{st} \text{ component align on the right} \end{cases}$$

and,

$$r_1 = \begin{cases} 2 \cdot a_1 & 1^{st} \text{ component at position } L \\ 0 & 1^{st} \text{ component at position } L - 2 \cdot a_1 \end{cases}$$

See Figure 5.5.

Let, ll_i denote the position of the left end of the i^{th} component to the left of position L of the 0^{th} component. Notice that $ll_i = ll_{i-1} + l_i$. Therefore,

$$ll_k = ll_{k-1} + l_k = ll_{k-2} + l_{k-1} + l_k \cdots = \sum_{i=1}^{i=k} l_i$$

Let, rr_i denote the position of the right end of the i^{th} component to the right of position $L+M$ of the 0^{th} component. Notice that $rr_i = rr_{i-1} + r_i$. Therefore,

$$rr_k = rr_{k-1}+r_k = rr_{k-2}+r_{k-1}+r_k \cdots = \sum_{i=1}^{i=k} r_i$$

Since $length_k = 2 \cdot L+M$, $ll_k = rr_k = L$, i.e. the m^{th} component is aligned with the 0^{th} component. Thus, $\sum_{i=1}^{i=k} l_i = \sum_{i=1}^{i=k} r_i$. But from the definitions of l_i and r_i, $l_i = 0$ if and only if $r_i = 2 \cdot a_i$, and $l_i = 2 \cdot a_i$ if and only if $r_i = 0$. Therefore l_i and r_i give a solution to the instance of PARTITION.

Now suppose there is a subset $A \subseteq S$ which is a solution of the instance of PARTITION. We can construct a legal configuration in $(2 \cdot L+M, k)_k$ by choosing $l_i = 2 \cdot a_i$ if $a_i \in A$ and $l_i = 0$ otherwise. Similarly, $r_i = 0$ if $a_i \in A$ and $r_i = 2 \cdot a_i$ otherwise. Hence, the stack of density problem is NP-complete. □

5.4 A Pseudo-polynomial Time Algorithm

In this section we present a pseudo-polynomial time algorithm for the shape of density stack problem. Recall that the input is the size of the stack, the lengths and widths of components in the stack, the terminals of components in the stack. Let l be the sum of all the component lengths, w be the sum of all the component widths, m be the total number of terminals and l_{max} be the largest integer to describe a terminal position, then the length of the input is $log(w \cdot l \cdot k)+m \cdot \log(l_{max})$. The running time of the algorithm depends on the sum of the lengths of all the components, which can be exponential in the length of the input. The structure of the algorithm is similar to Algorithm 2.1: for a given tradeoff dimension s it proceeds component by component and computes the minimum total density of a density stack at s. However, unlike the river routing problem, for a given channel density the offset range is not a contiguous range, we cannot apply the same method to compute the potential break points of the density stack. Therefore we compute the minimum total density of the stack at each tradeoff dimension. First we define some notation in the algorithm. Let,

$D_i(x)$ denotes the density function of the i^{th} channel.

$Dmin_i$ denotes the minimum channel density of the i^{th} channel.

$d_i(s, x)$ denotes the minimum total density of a stack of $i+1$ components at tradeoff dimension s with the i^{th} component at position x relative to the left boundary of the stack.

$dmin_i(s)$ denotes the minimum total density of a stack of $i+1$ components at tradeoff dimension s. $dmin_i(s) = \min_{0 \le x \le s-length_i} \{d_i(s, x)\}$.

It should be clear that $d_i(s, x)$ is only defined when $s \geq \max_{0 \leq j \leq i}\{length_j\}$. In addition, when $s \geq \sum_{j=0}^{j=i} length_j$, $dmin_i(s) = \sum_{j=0}^{j=i} Dmin_i$. This is because, at the j^{th} channel, the minimum horizontal span such that the channel density $Dmin_j$ is achieved is at most $length_{j-1}+length_j$. Therefore it is sufficient to consider the tradeoff dimension s such that,

$$\max_{0 \leq j \leq k}\{length_j\} \leq s \leq \sum_{j=0}^{j=k} length_j$$

when computing $dmin_k$.

For $\max_{0 \leq i \leq k}\{length_i\} \leq s \leq \sum_{i=0}^{i=k} length_i$ and $0 \leq x \leq s - length_0$, we define,

$$d_0(s, x) = 0$$

Then, for $0 \leq x \leq s - length_i$,

$$d_i(s, x) = \min_{0 \leq y \leq s - length_{i-1}}\{d_{i-1}(s, y) + D_i(x-y)\}$$

See Figure 5.6.

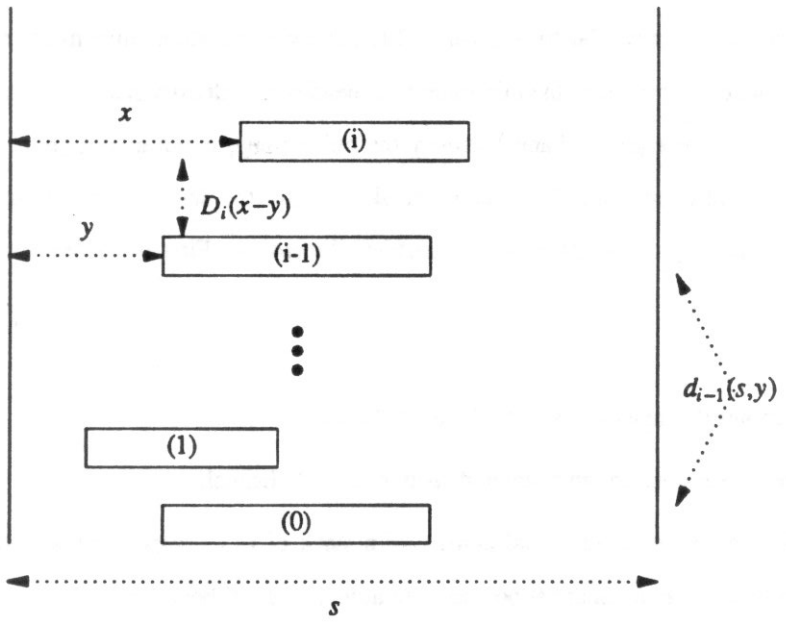


Figure 5.6

The validity of the algebraic definition of $d_i(s, x)$ can be verified by simple induction on the number of components in the stack.

To compute the shape function for a density stack, we compute the minimum total density at each tradeoff dimension s in the relevant range, i.e. $\max_{0 \le i \le k}\{length_i\} \le s \le \sum_{i=0}^{i=k} length_i$. Formally the algorithm is described as follows:

Algorithm 5.1:

(1) $Mins = \max_{0 \le i \le k}\{length_i\}$

(2) $Maxs = \sum_{i=0}^{i=k} length_i$

(3) **for** $i = 1$ to k

(4) Compute $D_i(x)$

(5) **for** $s = Mins$ to $Maxs$

(6) **for** $x = 0$ to $s - length_0$

(7) $d_0(s, x) = 0$

(8) **for** $i = 1$ to k

(9) **for** $x = 0$ to $s - length_i$

(10) $d_i(s, x) = \min_{0 \le y \le s - length_i}\{d_{i-1}(s, y) + D_i(x - y)\}$

(11) **for** $s = Mins$ to $Maxs$

(12) $dmin_k(s) = \min_{0 \le x \le s - length_k}\{d_k(s, x)\}$

$dmin_k(s)$ gives the minimum total density at tradeoff dimension s, and this describes the shape function of the density stack.

For a fixed tradeoff dimension s, the time required to compute $d_i(s, x)$ for x in the range $[0, s - length_i]$ is $O(s^2)$. Therefore the time required to compute $d_k(s, x)$ is $O(k \cdot s^2)$. Let $l = \sum_{i=0}^{i=k} length_i$, then the complexity of Algorithm 5.1 is $O(k \cdot l^3)$.

5.5 A Heuristic to Compute the Shape Function

In this section we present a heuristic to compute an approximate shape function of a density stack. We first outline the basic steps of the heuristic and explain the steps in detail. Let $dmin = \sum_{i=1}^{i=k} Dmin_i$, and let $s(d)$ denotes the tradeoff dimension of the approximate shape function at the total density d. The basic steps of the heuristic are as follows:

(1) Construct the initial configuration with the total density *dmin* greedily. In this step, we have a density vector $(Dmin_1, Dmin_2, \ldots, Dmin_k)$, and heuristically obtain a good tradeoff dimension $s(dmin)$. $(s(dmin), dmin)_k$ will be a break point of the approximate shape function we are computing.

(2) Find a channel whose density increased by one unit will result in the maximum reduction in the tradeoff dimension. In this step we obtain a new density vector for a total density which is one unit more than the previous, and a new tradeoff dimension which is no greater than the previous one.

(3) Repeat step (2) with the new density vector until the total density *dmax*, where *dmax* is the smallest total density such that $s(d)=s(dmax)$ for all total density d greater than *dmax*, $s(dmax)=\max_{0\le i\le k}\{length_i\}$. $dmax\le\sum_{i=1}^{i=k} n_i$, where n_i is the number of nets in the i^{th} channel.

Before we go into the details of how to carry out each step of the heuristic, we first show how the tradeoff dimension of a density stack can be computed given a density vector $(d_1, d_2, \ldots, d_k)$, and how a configuration of a density stack can be constructed with this density vector $(d_1, d_2, \ldots, d_k)$. Recall that in a river routable channel, the left and right constraint of the channel at a channel separation t, $L(t)$ and $R(t)$, represent the leftmost and rightmost position of the top component with respect to the bottom. It is a contiguous feasible offset range of the top component with respect to the bottom. In a general channel however, there may be more than one feasible offset range of the top component at a channel density d, i.e. at channel density d, the offset x where the density function $D(x)$ equal to d falls in a collection of feasible offset ranges instead of one contiguous feasible offset range. Let $l_p(d)$ and $r_p(d)$ represent the p^{th} feasible offset range of the channel at density d, then

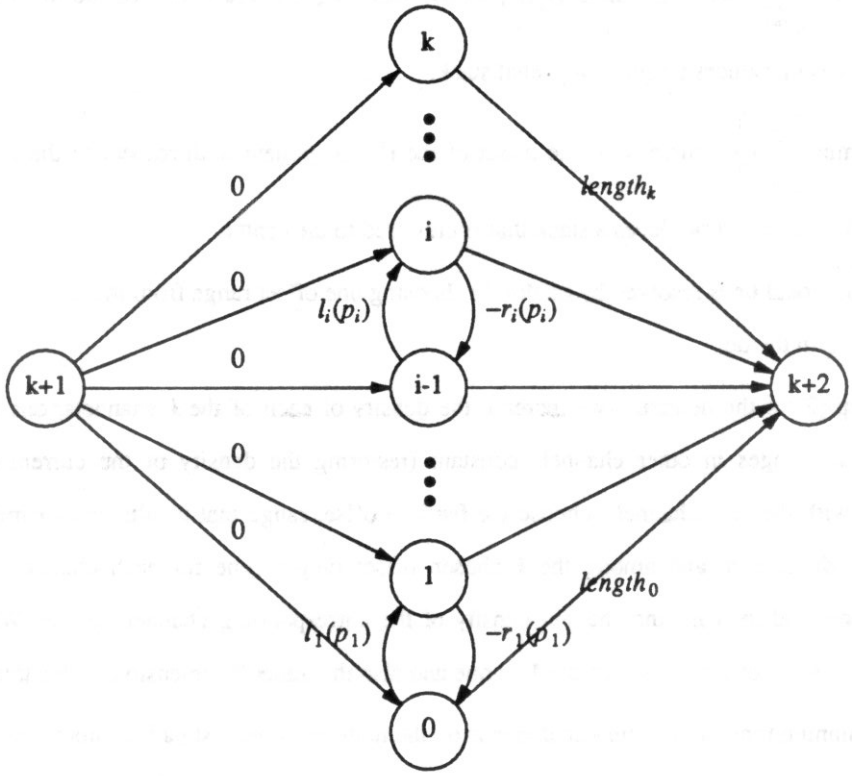
$$l_p(d) \le x \le r_p(d)$$

implies

$$D(x) \le d$$

Also recall the placement graph of a river routable stack of $k+1$ components for a separation vector $(t_1, t_2, \ldots, t_k)$. The placement graph consists of $k+3$ vertices, $v_0, \ldots, v_{k+2}$, v_i represents the position of the i^{th} component from the left boundary of the stack, for $i = 0,\ldots,k$, and v_{k+1} and v_{k+2} represent the left and right boundary of the stack respectively, $weight(v_{i-1}, v_i)=L_i(t_i)$, and $weight(v_i, v_{i-1})=-R_i(t_i)$. In a density stack, for a density vector $(d_1, d_2, \ldots, d_k)$, we have a collection of feasible offset ranges at a given channel density. However, if one is chosen for each channel we can construct a similar placement graph to

that of a river routable stack. The length of the longest path of such a placement graph is the tradeoff dimension of the stack for the chosen set of k feasible offset ranges (one for each channel). The labels of the non-boundary vertices give the configuration of the stack. See Figure 5.7. In general, it is infeasible to compute the shape function by solving for the longest path of the placement graph for each combination of the offset ranges of the k channels. Since the number of such combinations can be $n_1^2 \cdot n_1^2 \cdots n_k^2 = O(n_{\max}^{2 \cdot k})$, where n_i is the number of terminals in the i^{th} channel and $n_{\max} = \max_{1 \le i \le k} \{n_i\}$.



$D_i(d_i)$ is a collection of feasible offset ranges:

$$[l_i(1),r_i(1)],\ [l_i(2),r_i(2)],\ ...,\ [l_i(q_i),r_i(q_i)]$$

Let $[l_i(p_i),r_i(p_i)]$ be chosen for the i^{th} channel, then the above is the placement graph of the density stack with the density vector $(d_1,d_2,\ldots,d_k)$

Figure 5.7

We now describe each step of the heuristic in detail. In step (1) of the heuristic, we construct an initial configuration of a density stack for the density vector $(d_1, d_2, \ldots, d_k)$ by a greedy approach. We begin at the 1^{st} channel choosing the feasible offset range, from the collection of feasible offset ranges at density $Dmin_1$, which will result in a good tradeoff dimension. We add one component at a time, choose the feasible offset range of the top channel at the given channel density of that channel such that the

tradeoff dimension resulted by adding the new component is minimum. In case of a tie while adding the i^{th} component, we break the tie by one of the following cost criterion,

(1) $\min\left\{\sum_{j=1}^{j=i} label(v_j)\right\}$, where $label(v_j)$ is the label of v_j after the computation of the longest path on the placement graph of the current stack of components. This in effect produces a left compacted stack.

(2) $\min\left\{\sum_{j=1}^{j=i} (label(v_{i+2}) - label(v_j))\right\}$, where $label(v_{i+2})$ is the length of the longest path. This in effect produces a right compacted stack.

(3) $\min\left\{\sum_{i=1}^{i=k} w_i\right\}$, where w_i is the offset of the i^{th} component with respect to the $i-1^{st}$ component. This in effect produces a stack that is clustered to the center.

Any additional tie is resolved by randomly choosing one offset range from the set of offset ranges participating in the tie.

In step (2) of the heuristic, we increase the density of each of the k channels, one at a time while keeping offset ranges in other channels constant (restoring the density of the current channel before proceeding with the next channel), choose the feasible offset range that results in maximum reduction in the tradeoff dimension, and among the k chosen offset ranges (one for each channel) keep the most beneficial one, and increase the channel density of the corresponding channel by one. We obtain a new density vector with total density increased by one and also the tradeoff dimension of this total density.

The running time of the heuristic depends on the number of longest path computations. At one iteration, a longest path computation is done for each feasible offset range under consideration. There are at most $O(n_i^2)$ feasible offset ranges for the i^{th} channel and therefore the total feasible offset ranges is $O(n^2)$, where n_i is the number of terminals in the i^{th} channel, and n is the total number of terminals in the density stack. At each iteration the total density is increased by one, therefore there are at most n iterations. The time required for a longest path computation is $O(k)$ (See chapter 2). Therefore the total running time of the heuristic is $O(k \cdot n^3)$. In practice, the number of feasible ranges under consideration is much smaller than n^2, and the total number of iteration is $dmax - dmin$, therefore the actual running time is much more efficient than the worst case time.

5.6 Performance

We compare the performance of the heuristic by comparing the approximate shape functions produced by the heuristic to the exact shape functions produced by the pseudo-polynomial algorithm on sixty randomly generated density stacks. The size of the density stacks ranges from four components to nine components, ten were generated for each size. The length of the components ranges from 25 to 35 units. The average number of nets in a channel is 12. The minimum channel density ranges from 1 to 8. And the minimum density such that a channel has minimum horizontal span ranges from 1 to 11. See appendix B for the generation of the random density stacks.

For each density stack we compute the percentage that the total densities on the approximate shape function deviate from the total densities on the exact shape function. We compute the percentage that the total densities deviate for each tradeoff dimension s for s ranges from $s(dmax)$ to $s(dmin)$ and take the average over the range of s, where $s(d)$ denotes the tradeoff dimension on the approximate shape function at total density d, $dmin$ denotes the minimum total density on the approximate shape function and $dmax$ denotes the smallest total density on the approximate shape function such that $s(d)=s(dmax)$ for $d>dmax$. Let, $dheu(x)$ and $dopt(x)$ denote the total density on the approximate shape function and the exact shape function at tradeoff dimension x respectively. Let the fraction that the total densities deviate be d_{dev}, then,

$$d_{dev}=\frac{\left[\sum_{x=s(dmax)}^{x=s(dmin)} \frac{dheu(x)-dopt(x)}{dopt(x)}\right]}{s(dmax)-s(dmin)+1}$$

Similarly we compute the percentage that the tradeoff dimensions deviates. Let $sheu(x)$ and $sopt(x)$ denote the tradeoff dimensions of the approximate shape function and the exact shape function at total density x respectively. Let the fraction that the tradeoff dimensions deviates be s_{dev}, then,

$$s_{dev}=\frac{\left[\sum_{x=dmin}^{x=dmax} \frac{sheu(x)-sopt(x)}{sopt(x)}\right]}{dmax-dmin+1}$$

The results are tabulated in Table 5.1.

	mean	median	min	max	standard deviation
d_{dev}	0.017	0.007	0.000	0.123	0.0251
s_{dev}	0.024	0.008	0.000	0.158	0.0346

Table 5.1

On the average the heuristic produces shape functions that are very close to the exact shape functions. The average percentage that the approximate total densities deviates from the exact total densities is 1.7%, and the average percentage that the approximate tradeoff dimensions deviates from the exact tradeoff dimensions is 2.4%. In fact, the heuristic produced the exact shape functions almost 50% of the time. There are a few cases where the heuristic produced approximate shape functions that deviate from the exact shape functions by more than 10%, both in terms of total density and tradeoff dimension. The shape functions of one such example is shown in Figure 5.8. In Figure 5.9 we show shape functions of an example at the median of the sixty randomly generated examples. The actual running time of the heuristic is about an order of magnitude faster than the pseudo-polynomial time algorithm. This is because the number of feasible offset ranges under consideration at each iteration is small.

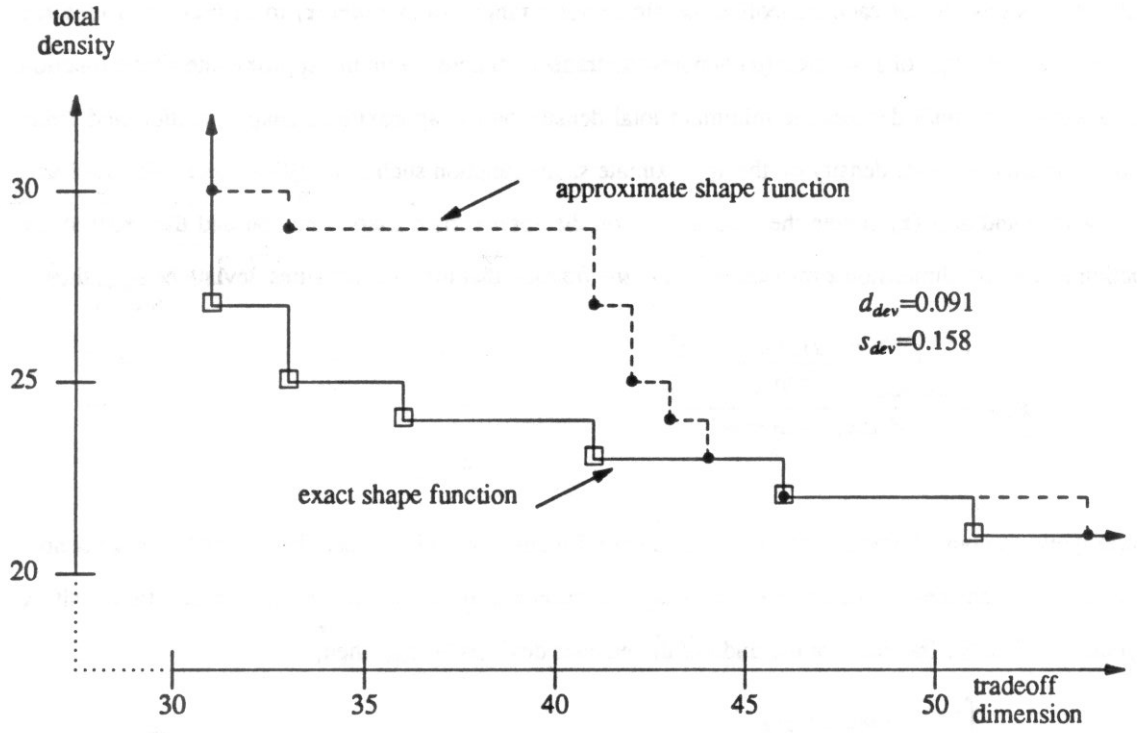


Figure 5.8

We observed that in general the heuristic is likely to produce the exact shape function when there are very few break points on the shape function. From the empirical study of the sixty randomly generated density stacks, we conclude that the heuristic produced good approximate shape functions.

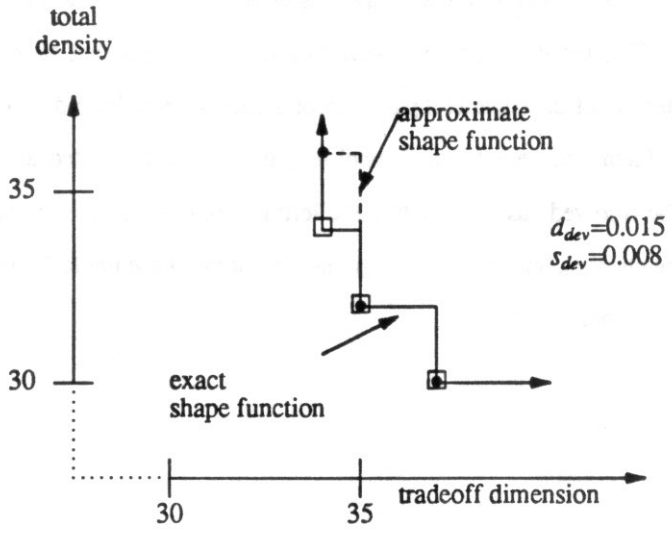


Figure 5.9

5.7 Conclusion and Open Problems

In this chapter we addressed the problem of computing the shape function of a stack of components using channel density as channel width. We showed that an efficient solution for such a problem is unlikely to exist by showing the decision problem Stack Of Density is NP-complete. We presented a pseudo-polynomial time algorithm and an efficient heuristic to compute the shape function of a density stack. We compared the approximate shape functions produced by the heuristic to the exact shape functions produced by the pseudo-polynomial algorithm. The empirical results show that the heuristic produces shape functions very close to the exact shape function.

Another variation of the problem of computing the shape function using channel density as channel width is to consider a multiple-component channel and compute the shape function of such channel. Johnson, LaPaugh and Pinter provide an $O(n^3)$ time algorithm to compute the minimum channel density of a multiple-component channel, where n is the number of nets in the channel [JLP]. Their algorithm does not compute the tradeoff dimension of the channel and it is not known whether the shape function of a multiple-component channel can be computed in polynomial time.

Finally, we will close this chapter by addressing a few technical details that arize when computing the shape function of slicing layouts using channel density as channel width. To compute the shape function of a slicing layout using channel density as channel width, we can apply the bottom up approach similar to the rigid algorithm in chapter 3. The heuristic presented above can be applied as a basic subroutine to compute the shape function of a stack. However, in the formulation of the shape of density stack problem we have assumed each channel only has top and bottom terminals. In general, after a global routing step of

a slicing layout, a wire connecting two terminals may pass through a channel and the net may not have any terminal in the channel. This problem can be resolved easily by the fact that such a net will contribute exactly one unit to the density of the channel it passes through; it does not depend on the offset of the channel. There is another problem when a net in a channel may have to be connected to a terminal outside the channel. This can also be resolved easily by letting the leftmost position of the net be $-\infty$ if the net has a terminal to the left of the channel, and letting the rightmost position of the net to be $+\infty$ if the net has a terminal to the right of the channel.

Chapter 6

Conclusions and Future Research

6.1 Conclusion

Due to the complexity of custom VLSI layouts, hierarchy is necessary for custom VLSI layout description. The slicing paradigm is a widely used hierarchical layout description. In this thesis we investigated the compaction problem for slicing layouts.

Our compaction approach is different from the conventional compaction approach. Conventional compaction is carried out after the detailed routing of the layout has been done. For river slicing layouts, our compaction approach captures the necessary routing space of the layout without the routing being present. The routing is deferred until after the layout is compacted. For more general slicing layouts we use the channel density to estimate the actual routing space and again the routing is deferred until after the compaction. We introduced the notion of shape function of a layout that includes the routing requirement between components in the layout. Computing such a shape function of a layout decouples the compaction and the detailed routing phase of the layout process, while giving interaction between the two phases.

For river slicing layouts with the river routing cell composition scheme, we presented methods of computing (sometimes approximate) shape functions. We presented an $O(k \cdot n^3)$ time algorithm to solve the two dimensional compaction problem for a stack of river routable channels. The solution provided an efficient computation of the shape functions for river slicing layouts of depth one. We provided heuristics to compute the shape functions of river slicing layouts of general depth. This is a generalization of the compaction problem of slicing layouts considered by Otten and Stockmeyer where they assumed there are no interconnections between components [Ott, Sto], and the compaction problem considered by Luk, Sipala and Wong where they considered slicing layouts with a single multiple-terminal net [LSW]. We compared the heuristic computation of the shape function for a river slicing layout with with the conventional pitch aligning cell composition scheme through 60 randomly generated examples. On the average, the minimum area layouts produced by the best heuristic are 33% smaller than the layouts produced by the pitch aligning scheme.

In Chapter 4 we addressed the issues of using a mixture of pitch aligning and river routing cell composition. We studied the asymptotic behavior of river routing and pitch aligning in uniform layouts, and we derived a condition under which river routing is better than pitch aligning.

We showed the intractability of computing the shape function of a stack of components, using channel density as channel width, when the routing channels are not river routable. We provided efficient heuristics for computing such a shape function.

6.2 Future Research

In our layout model we assumed wires have zero widths. This assumption allowed us to apply the relatively simple routability condition for a river routable channel that has wires with uniform widths. To implement the algorithms to compute the shape function in a real layout, necessary and sufficient routability conditions that capture non-uniform wire widths needs to be developed, and versions of algorithms 2.1 and 2.2 based on such conditions need to be investigated.

Our results on the shape function that captures the routing requirement are based on the routability condition of a channel -- the channel separation for a river routable channel and the channel density for a more general routing channel. We assumed a channel is a rectangular routing region, i.e. components that define the channel touch the boundaries of the channel and there is no empty space between the components and the boundaries of the channel. To compute more accurate shape function of a layout, routability conditions for non-rectangular channels need to be developed. We also assumed components in a layout to be rectangular, computing the shape function of a layout that contains non-rectangular components is an area that needs to be explored. Related work in this area is considered by Maley [Mal]. Maley presents necessary and sufficient conditions for single layer routing when the layout which contains non-rectangular components is compacted in a chosen dimension.

The core of our approach to compute the shape function of a river slicing layout is Algorithm 2.1 and 2.2, which compute the shape function of a stack of components. In the stack we considered, connections only exist between adjacent rows of components. In a more general stack, connections exist between non-adjacent rows. The ability to compute the shape function of such a stack will add more versatility to the compaction scheme we proposed in the thesis. Initial work in this area is considered by Joseph and Pinter who present an efficient algorithm to compute the minimal area of such a stack when the channel separations of routing channels are given [JoPi].

In the heuristics that compute the approximate shape function of a river slicing layout, we choose the best area shapes of the child slices for the computation of the shape function of their parent slice. One

direction of future research is to investigate more sophisticated shape choosing strategies. For example, when we know a priori the desired aspect ratio of the final layout, we can choose shapes of child slices so that the final aspect ratio can be achieved with smaller area. Another possibility is to ignore the routing requirements between child slices and apply the cheaper shape-summing operation presented by Otten [Ott] to sum the shape functions of the child slices, and obtain a "pseudo-shape" function of the parent slice. The shape summing operation can also be applied to obtained pseudo-shape function of slices at higher level of the hierarchy. The pseudo-shape function may provide more information for the slices at higher level of the hierarchy and enable us to choose more appropriate shapes of the child slices.

In a slicing layout, at each level of the hierarchy we compute the shape function of a stack. In a non-slicing hierarchical layout representation, at each level of the hierarchy we will be dealing with a more general layout topology, for example the topology that gives the cyclic routing order (Chapter 1). To apply the compaction scheme in this thesis to a non-slicing layout, algorithms that compute the shape function of a layout topology other than a stack need to be developed.

One final research issue we will address is hierarchical versus flat compaction. Due to the size of a VLSI layout, hierarchical compaction is needed to reduce the amount of information to be dealt with at any instant during the course of compaction. One drawback is the "boundary enforcement" of the hierarchy. In the case of a slicing layout; the hierarchy defines the routing channels of the layout, the routing channels are the boundaries of the hierarchy we enforce. It can be beneficial if we do not enforce the boundaries of the hierarchy, i.e. we can consider a boundary of the hierarchy as a flexible routing region rather than a routing channel as illustrated by Figure 6.1. This will allow the possibility of slices with jagged edges to be composed in a more compact way. The same remarks apply to non-slicing hierarchical compaction.

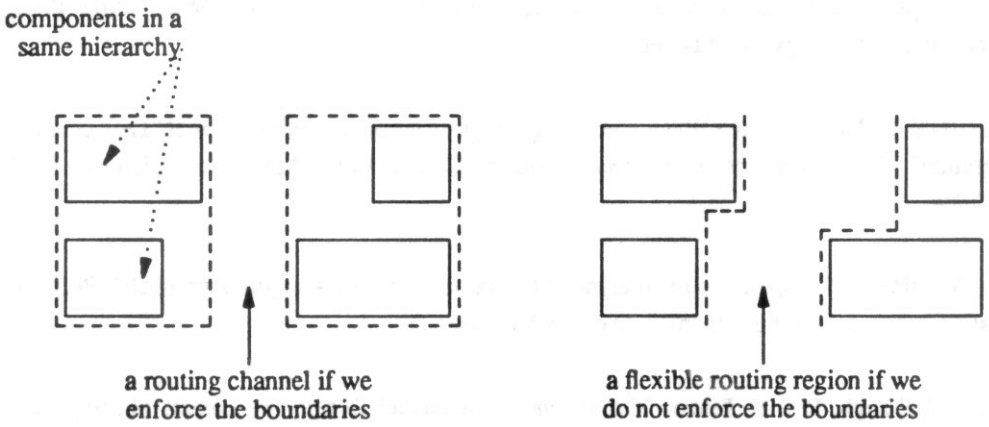


Figure 6.1

References

[DKSSU]Danny Dolev, Kevin Karplus, Alan Siegel, Alex Strong, Jeffrey D. Ullman: "Optimal Wiring Between Rectangles", *Proceedings of* 13^{th} *ACM Symposium on Theory of Computing*, pp. 312-317, 1981.

[Eic] P. A. Eichenberger: "Fast Symbolic Layout Translation for Custom VLSI Integrated Circuits", *Technical Report No. 86-295*, April 1986, Stanford University.

[GaJo] Michael R. Garey, David S. Johnson: "Computer and Intractability - A Guide to the Theory of NP-Completeness", W. H. Freeman and Company, San Francisco, 1979.

[HeLa] Fook-Luen Heng, A. S. LaPaugh: "Optimal Compaction of Multiple Two Component Channels under River Routing", *Technical Report No. CS-TR-068-86*, December 1986, Princeton University.

[HsPe] Min-Yu Hsueh, Donald O. Pederson: "Computer-Aided Layout of LSI Circuit Building-Blocks", *Proceeding of 1979 ISCAS*, pp. 474-477.

[Hsu] Min-Yu Hsueh: "Symbolic Layout and Compaction of Integrated Circuits", Ph.D. dissertation, 1979, EECS Division, University of California, Berkeley.

[Iwa] Kazuo Iwano: "Two Dimensional Dynamic Graphs and their VLSI Applications", Ph.D. dissertation, October 1987, Princeton University.

[JLP] David Jonhson, A. S. LaPaugh, Ron Y. Pinter: "Optimal lateral placement", unpublished manuscript.

[JoPi] A. Joseph, R. Y. Pinter: "Feed-through River Routing", *Technical Report 88.243*, June 1988, Science and Technology, IBM Israel.

[KeWa] Gershon Kedem, Hiroyuki Watanabe: "Graph-Optimization Techniques for IC Layout and Compaction", *IEEE Transaction on Computer-Aided Design*, Vol. CAD-3, No. 1, January 1984, pp. 12-19.

[LaP] A. S. LaPaugh: "Algorithms for Integrated Circuit Layout: An Analytic Approach", Ph.D. dissertation, 1980, Massachusetts Institute of Technology.

[LaPi] A. S. LaPaugh, Ron Y. Pinter: "On Minimizing Channel Density by Lateral Shifting", *ICCAD Digest of Technical Papers*, 1983, pp. 123-124.

[LaPo] D. P. LaPotin: "A Global Floor-Planning Approach for VLSI Design", Ph.D. dissertation, December 1985, Carnegie-Mellon University.

[Lau] Ulrich Lauther: "A min-cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation", 16th *Design Automation Conference*, 1979, pp. 1-9.

[Law] E. L. Lawler: "Combinatorial Optimization: Networks and Matroids", Holt, Rinehart and Winston, New York, 1976.

[Lei] Tom Leighton: "A Survey of Problems and Results for Channel Routing", paper presented at 1986 Aegean Workshop on Computing.

[LeMe] T. Lengauer, K. Mehlhorn: "The Hill System: A Design Environment for the Hierarchical Specification, Compaction, and Simulation of Integrated Circuit Layouts", *1984 Conference on Advanced Research in VLSI, M.I.T.*

[LePi] Charles E. Leiserson, Ron Y. Pinter: "Optimal Placement for River Routing", *SIAM J. Comput.*, Vol. 12, No. 3, August 1983, pp.447-462.

[LSW] W. K. Luk, P. Sipala, C. K. Wong: "Minimum-Area Wiring for Slicing Structures", *Technical Report RC 11477*, IBM Thomas J. Watson Research Center, 1985.

[LTW] W. K. Luk, D. T. Tang, C. K. Wong : "Hierarchical Global Wiring for Custom Chip Design", 23rd *Design Automation Conference*, 1986, pp 481-489.

[Mal] F. Miller Maley: "Single Layer Wire Routing", Ph.D. dissertation, August, 1987, Massachusetts Institute of Technology.

[Mat] J. M. da Mata: "A Methodology for VLSI Design and a Constraint-based Layout Language", Ph.D. dissertation, October 1984, Princeton University.

[MeCo] Carver Mead, Lynn Conway: "Introduction to VLSI Systems", Addison-Wesley Publishing Company, Menlo Park, California, 1980.

[MeNä] Kurt Mehlhorn, Stefan Näher: "A Faster Compaction Algorithm with Automatic Jog Insertion", *M.I.T. Conference on Advanced Research in VLSI*, 1988, pp. 297-314.

[Mir] A. Mirzaian: "Channel Routing in VLSI", *Proceedings of* 16th *ACM Symposium on Theory of Computing*, 1984, pp 101-107.

[OHMST] J.K. Ousterhout, G. Hamachi, R.N. Mayo, W.S. Scott, G.S. Taylor: "1986 VLSI Tools: Still More Works by the Original Artists", *Technical Report No. 86/272*, 1986, UCB/CSE.

[Ott] Ralph H.J.M. Otten: "Efficient Floorplan Optimization", *International Conference on Computers and Design*, pp. 499-502, 1983.

[PaSt] Christos H. Papadimitriou, Kenneth Steiglitz: "Combinatorial Optimization - Algorithm and Complexity", Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1982.

[Pin] Ron Y. Pinter: "The impact of Layer Assignment Methods on Layout Algorithms for Integrated Circuits", Ph.D. dissertation, 1982, Massachusetts Institute of Technology.

[RiFi] R. L. Rivest, C. M. Fiduccia: "A Greedy Channel Router", 19^{th} *Design Automation Conference*, 1982, pp. 418-424.

[Ros] Jonathan B. Rosenberg: "Chip Assembly Techniques for Custom IC design in a Symbolic Virtual-Grid Environment", *M.I.T. Conference on Advanced Research in VLSI*, 1984, pp. 213-225.

[SiDo] Alan Siegel, D. Dolev: "The separation for general single-layer wiring barriers", *Carnegie-Mellon Conference on VLSI Systems and Computations*, October 1981, pp. 31-41.

[Sto] Larry Stockmeyer: "Optimal Orientation of Cells in Slicing Floorplan Designs", *Information and Control 57*, 1983, pp. 91-101.

[Susl] Kenneth J. Suppowit, Eric A. Slutz: "Placement Algorithm for Custom VLSI", 20^{th} *Design Automation Conference*, 1983, pp. 164-170.

[Syz] T. G. Szymanski: "Dogleg Channel Routing is NP-Complete", *IEEE Transaction on Compute-Aided Design*, Vol. CAD-4, No. 1, January 1985, pp. 31-41.

[SzOt] Antoni A. Szepieniec, Ralph H.J.M. Otten: "The Genealogical Approach To the Layout Problem", 17^{th} *Design Automation Conference*, 1980, pp. 535-542.

[Wes] Neil Weste: "Virtual Grid Symbolic Layout", 18^{th} *Design Automation Conference*, 1981, pp. 225-233.

[WoLi] D.F. Wong, C.L. Liu: "A New Algorithm for Floor-plan Design", 23^{rd} *Design Automation Conference*, 1986, pp. 101-107.

[Yen] J. Y. Yen: "An algorithm for finding shortest routes from all source nodes to a given destination in general networks", *Quarterly of Applied Mathematics*, Vol. 27, No. 4, July 1970, pp. 526-530.

Appendix A

In this appendix we describe the generation of river slicing layouts that we used in the empirical study of the performance of the heuristics in Chapter 3. We generate slicing layouts that mimic the partitioning of a true design.

We first describe a general procedure that generates the slicing layouts and then describe the parameters that control the generation in detail. To generate a slicing layout we begin with a root slice whose horizontal and vertical dimensions are parameters. The value of the parameters are supplied at each generation. The root slice is partitioned into smaller slices according to some criterion. The criterion is controlled by some parameters which will be described later. The smaller slices are then further partitioned with the same criterion. In general a slice s with the dimensions and the number of terminals on each boundary known is partitioned as follows,

(1) Determine the orientation and the number of the cut lines that will partition the slice. If the cut lines have the same orientation as the slice, the resulting child slices remain in the same level, otherwise the child slices become one level lower in the hierarchy. See Figure A.1. The orientation of the cut lines is determined by the dimensions of s; when the horizontal (vertical) dimension is greater than the vertical (horizontal) dimension, the vertical (horizontal) cut lines are generated with higher probability. This has an effect of reducing the probability of generating slicing layouts with a lot of long and thin slices which in general are not present in a true design.

(2) After the child slices are obtained, distribute the terminals on the boundaries of s that are perpendicular to the cut lines to the child slices uniformly. We ensure that the terminals distributed to the boundaries (perpendicular to the cut lines) of the child slices can be accommodate by the boundaries.

(3) Determine the number of nets between the common boundary of two adjacent slices.

(4) If slice is a leaf cell, deform the slice but preserve the area of the slice, and then determine the displacements between adjacent terminals.

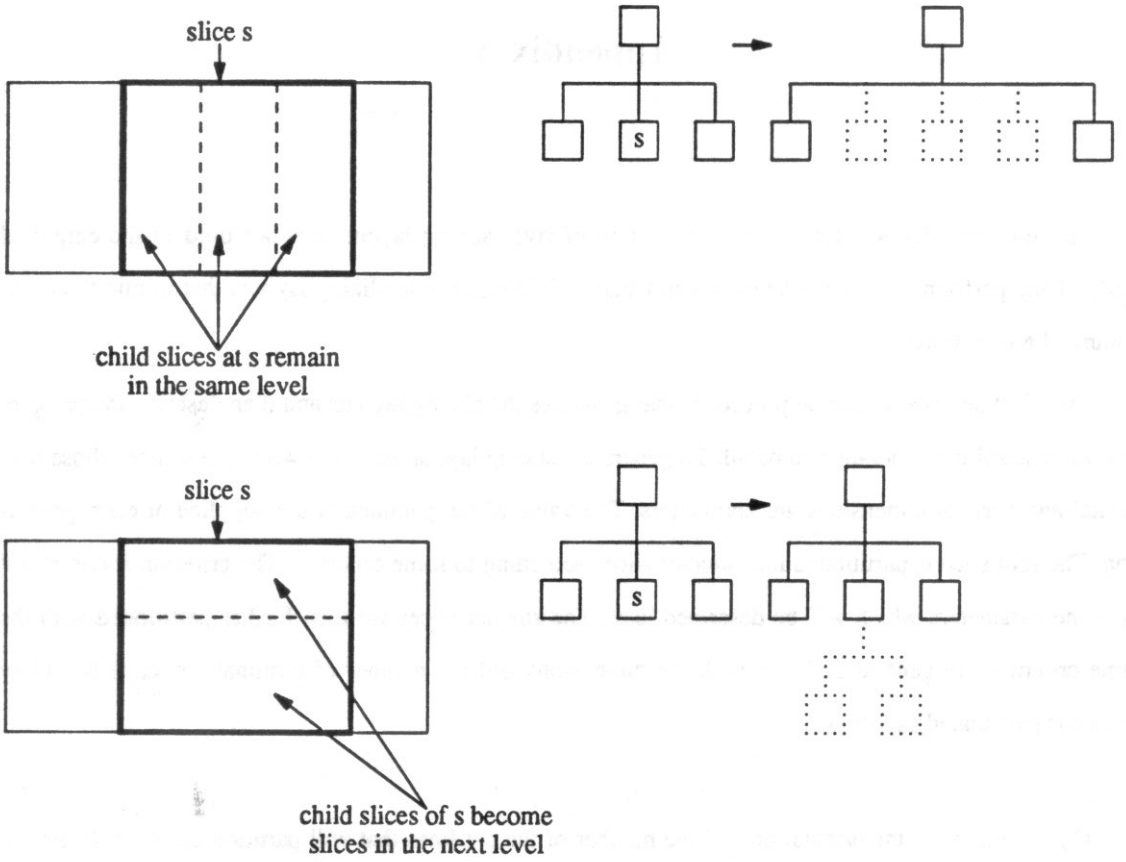


Figure A.1

The following are the parameters that control step (1) to (4) above,

seed, step

These provide seeds for the random number generators used.

nslice This is the maximum number of slices in one partitioning process.

level This determines the level of the resulting slicing layout.

WellFit This determines the degree of deformation of a leaf cell. If *WellFit* = 100% then no deformation is done. The deformed leaf cell is ensured to accommodate all terminals on its boundaries.

CutRatioLow, CutRatioHigh

This determines the number of nets in a common boundary. Let *dim* denotes the size of the common boundary, then the number of nets is uniformly chosen between *dim·CutRatioLow* and *dim·CutRatioHigh*.

CN, DN

These describe the functions $f(x)$ and $F(x)$, where

$$f(x)=CN+\frac{DN-CN}{n}\cdot x$$

and

$$F(x)=\frac{f(x)}{\sum_{i=1}^{i=n} f(i)}$$

For a boundary with n terminals, we want to generate $(n+1)$ pieces of displacements which add up to the length of the boundary. We use the probability function $F(x)$ to generate displacements between adjacent terminals, i.e. the i^{th} displacement has an expected value $F(i)\cdot$*length of boundary*. The $(n+1)$ pieces of displacement are then permuted randomly. This is intended to generate non-uniform displacements. When *DN* is equal to *CN*, the displacements are uniformly distributed, and when *DN* is not equal to *CN* the displacements have linear variations.

CS, DS

These describe similar functions $g(x)$ and $G(x)$. $G(x)$ governs the distribution of the displacements between cut lines on a slice.

With the parameters described above we will be able to generate a large variety of slicing layouts. When *WellFit* is 100%, the slicing layout generated is tight, i.e. if we ignore the routing between cells, all leaf cells are perfectly fit and there is no empty space in the layout. When *WellFit* is low, it may have a lot of empty spaces between cells. *CutRatioLow* and *CutRatioHigh* control the number of nets in a common boundary, when *CutRatioLow* = *CutRatioHigh* = 100%, the number of nets is equal to the size of the boundary. The probability function $G(x)$ governs the sizes of the leaf cells. When $CS \ll NS$ the leaf cells will have very different sizes.

We have described the generation of river slicing layouts for the river routing cell composition scheme. In order to use pitch aligning cell composition we need to generate one horizontal and one vertical constraint graph, which model a stretchable component, for each leaf cell. To generate a constraint graph for a leaf cell we need to determine the minimum displacement constraints between terminals. For terminals on the same boundary we use the actual displacements as the displacement constraints. For two terminals on the opposite boundaries, with probability p we generate a displacement constraint using the actual displacement between the two terminals. See Figure A.2 for an illustration. The number of constraints

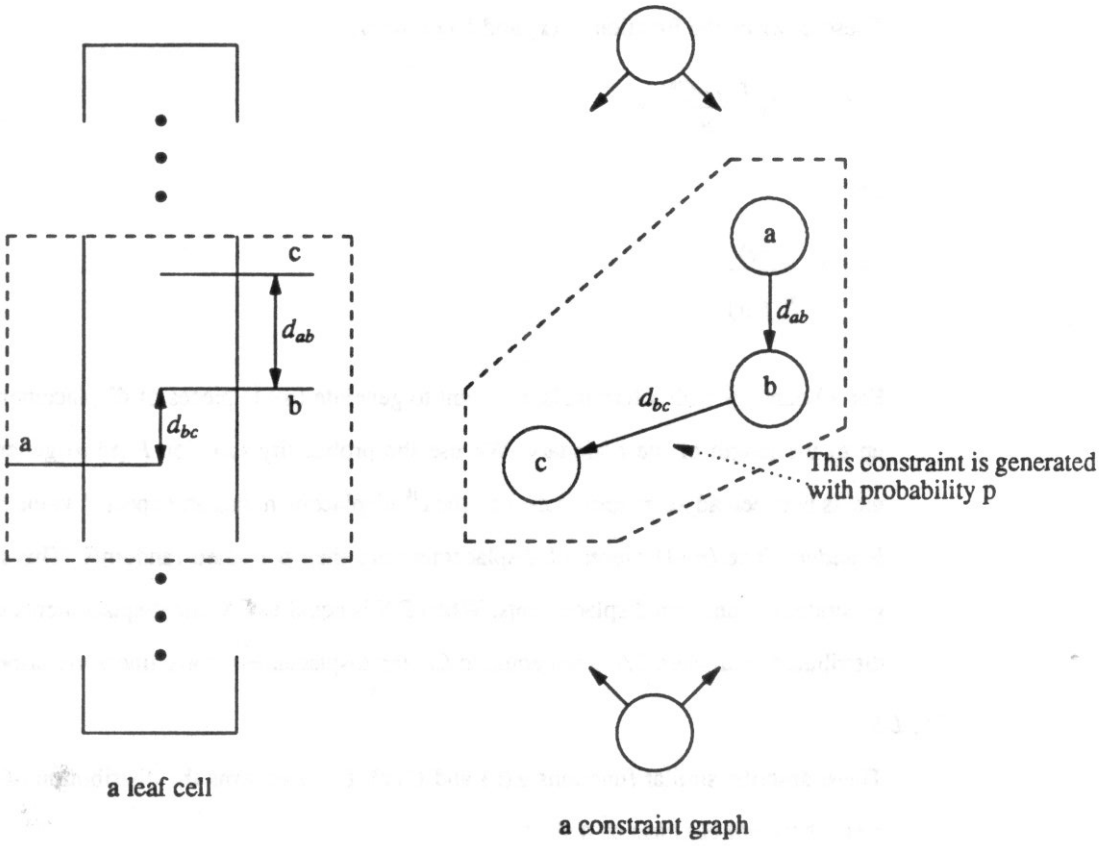


Figure A.2

generated with this assumption is linear in the number of terminals of the leaf cell. In addition, there is no cycle in such a constraint graph, therefore there is no cycle in the constraint graph of the whole layout and the longest path problem in such a graph can be solved in time linear in the number of constraints.

When p=0 there is no minimum displacement constraints between terminals on the opposite boundaries. In a layout generated with p=0, movement of terminals on one boundary of a component due to the stretching of the component will not affect movement of terminals on the opposite boundary. In this type of layouts, the undesirability of stretching is kept to the minimum. On the other hand, when p=1, there is a minimum displacement constraint between two neighboring terminals on the opposite boundaries. In a layout generated with p=1, movement of a terminal on one boundary of a component due to stretching will force a terminal on the opposite boundary to move. In this type of layouts the undesirability of stretching is at its maximum. We use p=0.5 in our examples.

Appendix B

In this appendix we describe the generation of density stacks that we used in the empirical study of performance of the heuristic in Chapter 4. We generate density stacks with two-point nets. A density stack is generated according to the following parameters,

seed The seed of the random number generator used.

k The number of channels in the stack.

minlength, *maxlength*

The lengths of the components in the stack are uniformly distributed between *minlength* and *maxlength*.

nlow, *nhigh*

Let *smlength* denotes the smaller length of the two adjacent components. Then the number of nets in the channel is uniformly distributed between *nlow*·*smlength* and *nhigh*·*smlength*.

Terminals on a boundary are randomly distributed on the boundary. 20% of the time the terminals are cluster to the left, 20% of the time they are cluster to the right and 60% of the time they are uniformly distributed on the boundary. In addition, the order of the terminals on one side of the channel is randomly permuted.