Dominance Constrained Drawing of Complete Binary Trees and Directed Acyclic Graphs on a 2-D Planar Grid

Lisa Priyanka,
CS0613
2006 – 2008

July 9, 2008

Contents

Diss/08/01/212

Key Words: Graph drawings, Dominance drawings, Timing Skew, VLSI Placement

Chapter 1

Introduction

1.1 VLSI Placement and Graph Drawing

In synchronous VLSI chips, signal from a single clock is fed to a number of modules. The clock routing scheme used determines the clock arrival time for different modules. In general, all functional units do not receive the clock signals simultaneously. The maximum difference in the clock arrival times at two components is called *clock skew*. The problem of timing skew between the clock and signal arrival is to be managed, to prevent increase in delay. *H-tree* [7, 8] layout, example for 4 modules in Figure 1.1, is commonly used but we advocate an alternate method: to route the signal wires and the clock wires together in parallel so that the skew between the clock and the signal vanishes, effectively. Figure 1.2 gives a schematic representation of the timing skew.

Having the clock wires routed parallel to the signal wires, the clock arrival time is matched with that of the signal. Due to inherent dependence among the modules, the

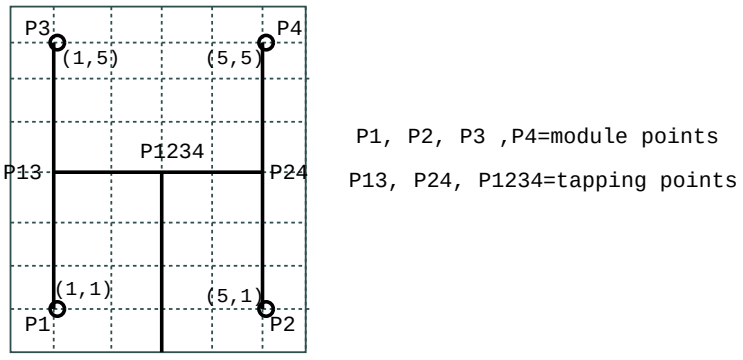

Figure 1.1: H-tree over 4 module points

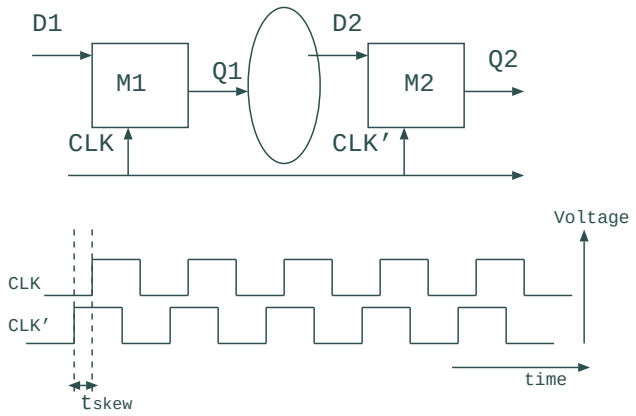

Figure 1.2: The time skew effect in VLSI chips

signal wires must obey the data dependence and so, must the clock wires. As the signal net is specific to the component for which it is being routed, while clock wire feeds all the modules, the clock routing is done separately. To still have parallel clock and signal wires, the data dependence forces some constraints into the VLSI placement phase. It is required that gates or modules must be placed in a way that their signal arrival times are greater than of those, from which they receive their input signals. We wish to analyze the area requirement for the placement of gates under these precedence constraints.

These placement constraints are covered by *dominance constraints*. Dominance drawing is a type of *graph drawing* where a given directed graph is drawn under dominance constraints. We shall formally define dominance constraints later in chapter 2, for the moment we give a brief introduction of *graph drawing* and its various objectives.

In graph drawing the vertices and edges of the input graph are mapped onto another graph to satisfy some constraints. Wide varieties of graph drawings have been described in the literature.

1.1.1 Types of graph drawings

Polyline drawing : Each edge is a polygonal chain (Figure 1.3(a)).

Straight-line drawing : Each edge is a straight-line segment (Figure 1.3(b)).

Orthogonal drawing : Each edge is a chain of horizontal and vertical segments (Figure 1.3(c)).

Grid drawing : Polyline drawing such that vertices, crossings and bends have integer coordinates.

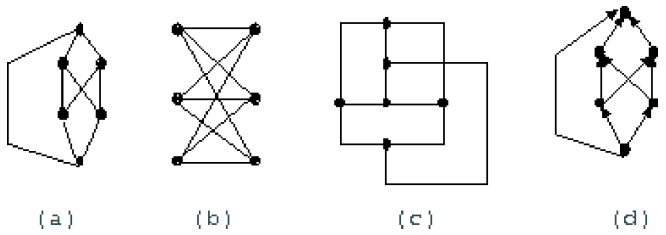

Figure 1.3: Types of drawings; (a) Polyline drawing of $K_{3,3}$; (b) Straight-line drawing of $K_{3,3}$; (c) Orthogonal drawing of $K_{3,3}$; (d) Planar Upward drawing of an acyclic digraph

Upward drawing : Drawing of a digraph where each edge is monotonically nondecreasing in the vertical direction (see Figure 1.3(d)).

Dominance drawing : Upward drawing of an acyclic digraph such that there exists a directed path from vertex u to vertex v; if and only if $x(u) \leq x(v)$ and $y(u) \leq y(v)$, where $x(.)$ and $y(.)$ denote the coordinates of a vertex.

hv-drawing : Upward orthogonal straight-line drawing of a binary tree such that the drawings of the subtrees of each node are separated by a horizontal or vertical line.

Bend : In a polyline drawing, point where two segments part of the same edge meet (Figure 1.3(a)).

Crossing : Point where two edges intersect (Figure 1.3(b)).

Most of the works done earlier in domain of graph drawing achieve almost planar drawings, i.e. with a limited number of edge crossings. This is to ensure effective visualization of the drawing and address the problem of constructing geometric representations of graphs, a major aspect of the emerging field of information visualization. Graph drawings have a wide variety of applications in various fields such as Software Engineering, Databases, VLSI Routing, and Biology.

1.2 Scope of the work

Our intention is to use the graph drawing, more specifically dominance drawing, to analyze the area requirement for the placement of gates under the dominance constraints. As the problem comes from the domain of VLSI, we allow non-planarity into our drawings. In fact, the crossings would be taken care by a change of layer in VLSI chips. The aim of the work is to design an algorithm that finds the least area required for the dominance drawings of complete binary trees and directed acyclic graphs.

1.3 Organization of the thesis

In chapter 2, we define the terms useful for our work. Chapter 3 contains a brief overview of some related previous works. In chapter 4, we explain our proposed algorithm for minimal area dominance constrained drawing of a complete binary tree, in chapter 6 we suggest an improvement over this algorithm and verify the area-efficiency obtained by the former. Chapters 5 and 7 discuss algorithms for minimizing edge congestion during edge mapping and for the dominance constrained placement of DAGs, respectively. The next chapters 8 and 9, discuss the implementations issues for the proposed algorithms and the obtained experimental results, respectively. Then we provide a comparison of the work with the results of the previous ones, briefly. Last chapter 11 concludes the thesis after briefing about the achievements made and the pending issues to be taken up in future.

Chapter 2

Problem Definitions

We study a variant of one of the graph drawing problems and discuss some solutions in this dissertation thesis. We consider a 2-dimensional grid as our host graph, on which the vertices and edges of a guest graph are mapped to satisfy the following constraints :

Let G' be the grid graph, with the grid points in the plane as the vertices of G' and line segments joining these grid points as the edges of G'. For a guest graph G, which is either a directed acyclic graph or a tree with directed edges from parent to the child vertices, let $Z: G \rightarrow G'$ denote an assignment of vertices of G onto the grid points. An assignment $Z(v) = (Z_x(v), Z_y(v))$ for $v \in V(G)$ comprises the x- and the y- coordinate of the grid point corresponding to v.

Definition 1 *Positive assignment: An assignment Z is defined to be a positive assignment if all the assigned grid points are in the positive quadrant of the rectangular coordinate system.*

Definition 2 *Dominance constraints: An assignment Z of the binary tree to the grid points is said to satisfy the dominance constraints if for each edge $e = (u, v) \in E(G)$ directed from vertex u to vertex v*

1. *$Z_x(u) \leq Z_x(v)$ and $Z_y(u) < Z_y(v)$;*
2. *e is mapped onto a nondecreasing vertical monotone path along the grid lines, directed from $Z(u)$ to $Z(v)$.*

In this definition of dominance constraints we have modified the condition on y-coordinate of the assignment from one present in the popular definition of dominance drawing as given in Chapter 1. This modification is required for our purpose since it must

be ensured that clock signal arrival time for a gate must be strictly greater than that for its predecessor gates in the digraph capturing the circuit. Having such a constraint on the placement of VLSI modules, imply placement based on precedence relations (created by data dependence) among the modules. As said earlier, we wish to have the clock arrival time equal to the signal arrival time, so data dependence among the modules is used. This would help in doing away with the *clock skew*.

In general graph drawings the objective is to do either or some of the following: minimize the area of the drawing, minimize the bends, minimize the crossings, etc. For studying the area of the drawing, length and width of the drawing is measured, and area is the product of the length and the width. We shall be discussing the drawings for complete binary trees and directed acyclic graphs. Under the dominance constraints, the minimum length of the drawing for an N- node complete binary tree is equal to the height of the tree $O(logN)$, which gives a width of $O(N)$, hence an area of $O(NlogN)$. To minimize the area, we consider the case where the length is equal to the width, i.e. we try to minimize the dimension of the square that contains the grid points assigned under Z. We consider the square bound for a Z, defined below, as a yardstick for area of the drawing.

Definition 3 *Square bound: For a positive assignment Z satisfying the dominance constraint, the minimum integer s satisfying the relation $(Zx(v), Zy(v)) \leq (s, s)\ \forall\ v \in V$, is the* ***square bound*** *of Z and the area of Z is given by s^2.*

Among other objectives, especially with applications in the domain of VLSI routing, is to minimize the *congestion* and sometimes to minimize the *dilation* of the drawings too. We define these terms for a grid drawing below.

Definition 4 *Congestion: For mapping of edges of the guest graph G on the edges of the host graph G', the number of times an edge $e' \in G'$, is used for some edge $e = (u, v) \in G$ is termed as congestion of the edge e'.*

Definition 5 *Dilation: For a given assignment Z, the manhattan distance between the vertices $u' = Z(u) \in G'$ and $v' = Z(v) \in G'$ for an edge $e = (u, v) \in G$, is the dilation associated with edge e.*

Here we mention that we shall also study the congestion and dilation of our drawing, and try to map the edges so as to minimize the congestion as permitted under our main objective to minimize the area.

Chapter 3

Previous Works

Of different works published for many kinds of graph drawings, upward drawings are closest to the dominance drawings, which is the prime focus of the work presented in this dissertation thesis. In [2], the authors develop a linear time algorithm for an area optimal, i.e $O(N)$ area, upward tree drawings. They obtain a planar polyline grid drawing for a bounded degree rooted tree with N- nodes. They also cite [5] which proves a lower bound of $\Omega(NlogN)$ on area for a strictly upward planar tree drawings, if the left to right oredr of the siblings in the tree is preserved.

Another paper [4] studies the upward planar drawings for single source acyclic digraphs. It presents an efficient algorithm based on *Thomassen*'s [6] graph theoretic charaterization for single source digraphs that admits an upward planar drawing.

A different work [3] on layouts of complete binary trees on an a by b grid, obtains an area efficient layout, i.e a layout with small *expansion ratio* (ratio of number of grid points to the number of tree points). It studies embedding of a guest graph (binary tree) G, into a host graph (grid with k layers, k=1 or 2) H to obtain an area efficient layout with minimum node-congestion. *Node-congestion* for a host vertex $v \in H$ is the number of edges of G that have been mapped on the host graph H through the vertex v.

Some earlier works on VLSI graph layouts [9], find lower bound on area using crossing number and wire area arguments. A lower bound of $\Omega(NlogN)$ has been found for non-planar graph layouts of planar graphs under this consideration.

Chapter 4

Embedding Complete Binary Trees on a 2-D Grid

4.1 Placement of Complete Binary Tree under dominance constraints

We first solve the problem of minimizing the square bound for dominance grid drawing of a complete binary tree and later extend the algorithm for dominance drawing of directed acyclic graphs.

The proposed method uses a greedy incremental approach starting with the root vertex of the tree G, and traversing it in a breadth first manner. All vertices of G occurring at level $l-1$ are assigned to the respective grid points, prior to moving to the next level l. During each set of assignments of the vertices at a level of G, the proposed method uses a set of rules to ensure minimizing the square bound of the grid drawing. A formal description of the proposed algorithm and the functions used by it are given below.

The proposed algorithm consists of the following phases:

1. Determine the region, on 2-D plane, to be assigned to a level in the drawing, so as to minimize the square bound.
2. Place the nodes of a level in the region determined above.
3. Map the tree edges along the grid edges so as to minimize the maximum congestion.

Phases 1 and 2 are implemented for each level of the tree, and phase 3 is implemented after complete assignment of the nodes of the complete binary tree.

The root vertex of G is assigned to the grid point having coordinate $(0, 0)$. Then, $Z(root) = (0, 0)$. The assignments of the successor vertices therefore must have $Z_x \geq 0, Z_y > 0$. In general, at any level of the tree G, the dominance constraints indicate that if the lowest row assigned for a vertex of a level of G is y', then all the vertices at next level can only be placed in rows $y \geq y' + 1$. The lowest row at which a vertex at level l of G can be assigned is $y = l$. Formal description of the algorithm for constructing a dominance constrained drawing is as follows:

ALGORITHM 1: CONSTRUCTING DOMINANCE CONSTRAINED DRAWING Z	
Input:	A complete binary tree G, its maximum level l_{max}, and a grid
Output:	Z and s, the minimum square bound of the drawing
M1.	(* Assign root vertex of G to origin of grid *) $Z(root) := (0, 0)$, $l := 1$, $forward(l) := 0$, $s := 0$
M2.	Repeat while $l \leq l_{max}$ *find_region*$(forward(l), N_l, s)$ (* Determine region on the grid for assignment of vertex set N_l of G at level l, where s is current value of the square bound *) *assign_current*$(prev\_assgn, s)$; $l := l + 1$ (* Place level l vertices of G in the region determined for level l *)
M3.	Print s
M4.	*Edge_mapper*(G, Z, l_{max}, s)

Phase 1 is implemented by *find_region* module in **Algorithm 1**. It determines the region required for nodes of a level l, by doing the following:

1. Calculates the number of unassigned nodes in the level l , after the forwarded unused points (with ordinate $\geq l$) within the square bound achieved for the previous level $l-1$ are used. Starts with the square bound of the previous level as the initial square bound of the current level l.

2. Increments the square bound of the current level l and includes the grid points, with ordinate equal or greater than $y = l$ and on the new boundary line in the region for level l. Step 2 is repeated until as many grid points as there are nodes in the level l have been included in the region for level l.

This function first uses the unused grid points, if any, given by $forward(l)$. Such grid points, if it exists, must lie on the periphery of the square bound s, i.e. along the lines $x = s$ or $y = s$. If all the vertices are not placed, the square bound is increased by one, and the grid points on the new boundary line are included in the region. The final region on the grid that ***find_region*** (Phase 1) reserves for each level is in general L- shaped, as

shown in Fig. 4.1. In the figure, the shaded L- shaped portion is the reserved region for the level $l-1$, and we call it the parent region. The unshaded L- shaped portion denotes the reserved region for the current level l. Also, the placemnt of nodes from level $l-1$ into the parent region is completed before reserving a potential region on the grid for the current level.

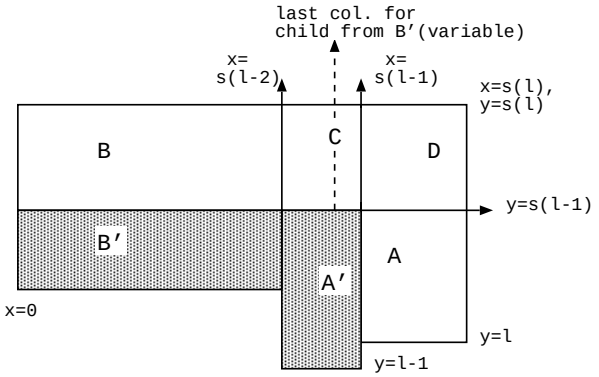

Figure 4.1: Abstract description of subregions in the parent and current level regions

FUNCTION 1: FINDING POTENTIAL REGION FOR VERTICES AT LEVEL L
find_region(forward(l),s,N_l)
(* Use the unused *forward(l)* grid points within region bounded by s *)
F1. $N_l := N_l - forward(l)$
F2. if $N_l := 0$ then $count := N_l$; Return
F3. $s := s + 1$ (* Increase the square bound *)
F4. Count grid points, k, on the new boundary line with ordinate values $\geq l$
F5. if k (in Step **F4**) $< N_l$ then
$N_l = N_l - k$;
(* Use the counted points *)
go to Step **F3**
else $forward(l+1) := k - N_l$; Return

Phase 2 is implemented by the function *assign_current*. This function assigns integer coordinates to the nodes of the current level l, after confirming that the dominance constraints are satisfied. As the constraints are checked here, this function forms the crux of the algorithm. It aims at assigning grid points included in the region for level l so that the square bound remains minimum and the constraints are never overlooked. To achieve this, it divides the region allotted to the previous level $l-1$ into *subregions* A' and B', where A' consists of grid points with abscissa greater than $s(l-2)$, the square bound achieved up to level $l-2$, and B' consists of the remaining grid points. Also the current level l, is divided into *subregions* A, B, C and D such that

subregion A has grid points with $s(l-1) \leq x \leq s(l), l \leq y \leq s(l-1)$;

subregion B has grid points $0 \leq x \leq s(l-2), s(l-1) \leq y \leq s(l)$;

subregion C has grid points $s(l-2) < x \leq s(l-1), s(l-1) \leq y \leq s(l)$; and,

subregion D has grid points $s(l-1) < x \leq s(l), s(l-1) \leq y \leq s(l)$.

It places the nodes under the constraints by following the rules:

1. The grid points with lowest ordinate value are assigned first.
2. For columns containing previous level nodes, the children of nodes in a column is preferred for placement in the grid points on the same column.
3. For columns not containing previous level nodes but having them in the immediate previous row the grid points are assigned to the children of nodes in the previous row, preferably.

Function 2: Assignment of Vertices at a Level to Grid Points	
assign_current(prev_assgn,s)	
Input:	Previous level assignment, square bound
Output:	Drawing of all vertices at level l
A1.	For grid points on rows with previous level vertices on them (*subregion* A), place children of vertices in the previous row(preferably from the farthest column that has a vertex with unassigned child).
A2.	For grid points on columns with previous level vertices on them and with $\leq s(l-2)$ (*subregion* B), place children of vertices in the same column (preferably from the nearest row that a vertex with unassigned child).
A3.	For all grid points that are not yet assigned (*subregion* C and D), place children of vertices in the previous columns.
A4.	Return

The search for an appropriate child vertex that should be assigned to a grid point during steps $A1$ and $A2$ of the function ***assign_current*** involves a simple table look-up into an array of children of vertices, of level $l-1$, for each row and column. For convenience, we term this search method as *Direct_Search*. During step $A3$, the search for an appropriate unassigned child may in addition to *Direct_Search* may require, scanning through arrays associated with many previous columns. So, we term this search method as *Lin_Scan* (for linear scan).

In Step **A1** the assignment from previous row is preferred from farthest column to permit completion of the assignment in the region determined in phase 1 as proved in **Theorem 1** (discussed later). This is not a restriction on the assignment and small variations in assignment is possible by appropriately changes. We shall show the variations in assignment later with the experimental results.

After the placement of vertices, the *Edge_mapper* function discussed later in chapter 5, completes the dominance drawing by mapping the tree edges on the grid edges. A scheme used to map the edges along the grid lines that ensures a vertical monotone edge mapping is that the mapped edges traverse rightwards and upwards, only.

Rectangular windows drawn for each edge $e = (u, v)$, with $Z(u), Z(v)$ as bottom-left, top-right corners respectively, are overlapping. Therefore the drawing obtained as a result of this algorithm is non-planar.

To explain the working of the **Algorithm 1**, we take as an example a full, complete binary tree of height $l_{max} = 5$. Refer figure 4.2. As a result of ***find_region*** the grid points are reserved for the vertices of the current level, and then the assignment is completed as per rules of ***assign_current***.

4.2 Analysis of Square bound

Let $s(l)$ be the square bound of the complete binary tree of height l, $\Delta s(l)$ the increment over $s(l-1)$ to get $s(l)$, then

$$\Delta s(l) = s(l) - s(l-1) \tag{4.1}$$

Let $count_k(l)$ be the count of new grid points found after k^{th} iteration of increasing $s(l-1)$ by 1, then

$$\begin{aligned} count_0(l) &= count_{\Delta s(l-1)}(l-1) + 1 \\ count_{k+1}(l) &= count_k(l) + 2 \qquad \text{for } k \geq 0 \end{aligned} \tag{4.2}$$

As an increment by 1 in the square bound increases both the number of columns and the number of rows by 1, there is an increment of 2 in $count_k$. This is because corresponding to each point on the previous boundary there is a grid point on the new boundary line $x = s$ and $y = s$, which has either the same x- or the same y- coordinate, except for $(s-1, s-1)$ which has two points $(s, s-1)$ and $(s-1, s)$ and one more grid point (s, s) which has no corresponding point on previous boundary line.

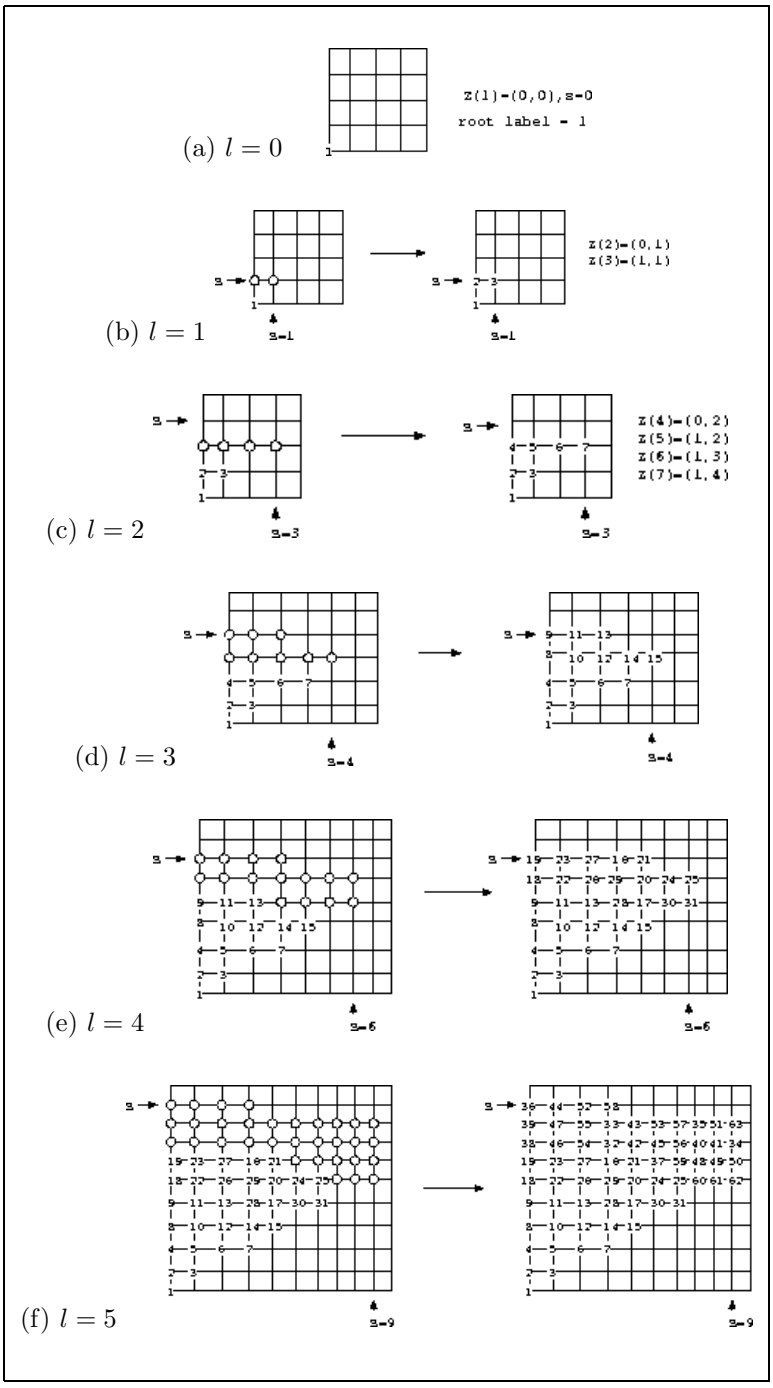

Figure 4.2: Illustration of *Algorithm 1* on complete binary tree with $l_{max} = 5$.

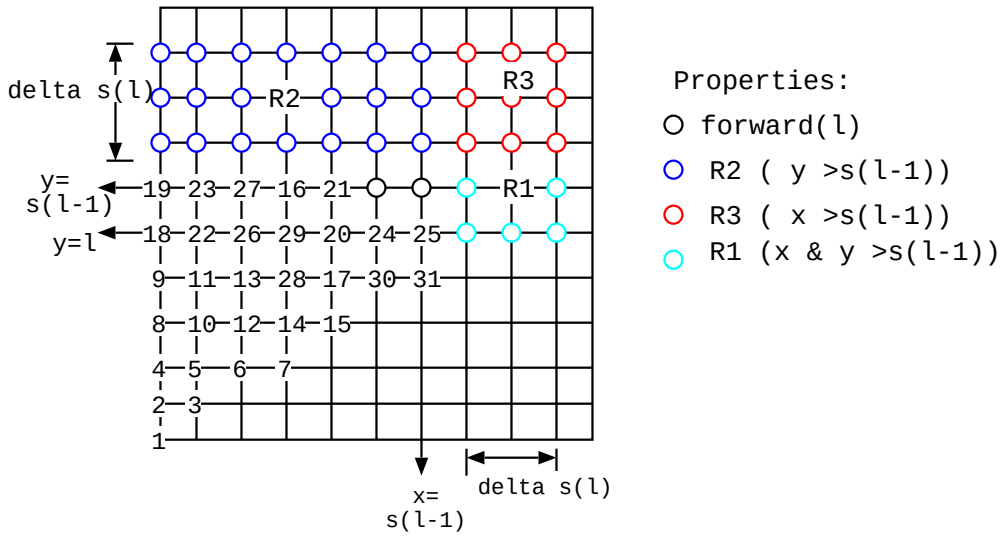

Figure 4.3: Subregions *R1,R2* and *R3* within a region allotted to a level

The first increment in $s(l-1)$ requires an increment by 1 over $count_{\Delta s(l-1)}(l-1)$ due to a shift of 1 in the lowest row. Over $\Delta s(l)$ increments, $count_{\Delta s(l)}(l)$ is given by (using equation 4.2)

$$count_{\Delta s(l)}(l) = count_0(l) + 2 * (\Delta s(l) - 1) \tag{4.3}$$

and the total number of new grid points counted is,

$$\Delta s(l) * count_0(l) + \Delta s(l) - 1 \tag{4.4}$$

Let $forward(l)$ denote the number of grid points that are unused by level $l-1$ and forwarded to the next level. Let $rem(l)$ be the number of vertices at level l which are not placed after having used $forward(l)$, $rem(l)$ is given by

$$rem(l) = N_l - forward(l) \tag{4.5}$$

Also, in each iteration of ***find_region*** function, at least $count_0(l)$ many grid points are counted, this implies that,

$$\begin{aligned} \Delta s(l) \quad &\leq \quad \lceil rem(l)/count_0(l) \rceil \qquad \text{for } l > 0, \\ & \qquad rem(0) = 1, \quad count_0(0) = 1 \end{aligned} \tag{4.6}$$

Graphically, the region allotted to a level by ***find_region*** function consists of the following three regions (Fig. 4.3):

$$\begin{aligned} \mathbf{R1} &= \{(x,y) | s(l-1) < x \leq s(l), l \leq y \leq s(l-1)\}, \\ \mathbf{R2} &= \{(x,y) | 0 \leq x \leq s(l-1), s(l-1) < y \leq s(l)\}, \\ \mathbf{R3} &= \{(x,y) | s(l-1) < x \leq s(l), s(l-1) < y \leq s(l)\} \end{aligned}$$

and the grid points forwarded from previous level $l-1$. **R1** is same as *subregion A* of Fig. 4.1, **R2** is the union of *subregions B* and *C*, and **R3** is same as *subregion D*. The number of grid points from the three regions **R1, R2, R3** are:

$$\begin{aligned} \textbf{from R1:} \quad & \Delta s(l) * (s(l-1) - l + 1) \\ \textbf{from R2:} \quad & \Delta s(l) * (s(l-1) + 1) \\ \textbf{from R3:} \quad & \Delta s(l) * \Delta s(l) \end{aligned}$$

$$\textbf{total} \quad = \quad \Delta s(l) * 2 * s(l-1) + 2 - l + \Delta s(l) \tag{4.7}$$

From equation(4.7), we get

$$forward(l+1) = forward(l) + \Delta s(l) * 2 * s(l-1) + 2 - l + \Delta s(l) - N_l \tag{4.8}$$

To minimize the square bound at each level, we need $forward(l+1) \geq 0$. Using the values of $forward(l)$, $s(l-1)$ that is known after ***find_region*** is executed for level $l-1$, and applying $forward(l+1) \geq 0$ in equation (4.8), we get

$$\begin{aligned} \lceil 2^{(l+1)/2} \rceil \quad &\leq \quad s(l) \ll \lceil 2^{(l+2)/2} \rceil \quad \text{and,} \\ \Delta s(l) \quad &= \quad 2^{(l+1)/2} - 2^{l/2} \quad \text{(approx.)} \end{aligned} \tag{4.9}$$

This gives the order of the square bound achieved by the proposed algorithm. If l_{max} be the last level in the given Complete Binary tree, the square bound for level l_{max} gives the square bound for the Complete Binary tree with $N = 2 * 2^{l_{max}} - 1$ vertices.

$$\begin{aligned} \Delta s(l_{max}) \quad &= \quad O(2^{l_{max}/2}) \\ s(l_{max}) \quad &= \quad \Sigma_{l=0}^{lmax} \Delta s(l) \geq 2^{(l_{max}+1)/2} - 1 \\ &= \quad (N+1)^{1/2} - 1 \\ s(l_{max}) \quad &= \quad O(2^{l_{max}/2}) \\ \textbf{(sq. bound)}\ s \quad &= \quad O(N^{1/2}) \end{aligned} \tag{4.10}$$

Thus, the minimum square bound obtained is $O(N^{1/2})$, where N is the total number of vertices in the binary tree G. Hence, the upper bound on the square bound s, achieved by the algorithm, is $O(N^{1/2})$.

4.3 Time Complexity

Steps **M1** and **M3** of **Algorithm 1** takes $O(1)$ time. Step **M2** is repeated for levels $l = 1$ to $l_{max}(= logN)$. Function ***find_region*** requires $\Delta s(l)$ constant time iterations. Steps **A1** and **A2** of ***assign_current*** takes constant time for each grid point (provided

proper update of index on *prev_assign* array is done), Step **A3** requires $O(s(l-1))$ to traverse through all the previously assigned columns, for each grid point. Each step of the ***assign_current*** places $O(\Delta s(l) * s(l-1))$ vertices in the worst case. Therefore, the worst case time complexity for the execution of the proposed **Algorithm 1** on l_{max} levels of the complete binary tree is, $O(N^{3/2})$.

4.4 Correctness and Optimality of the Algorithm

4.4.1 Correctness of the Algorithm

Theorem 1 *Direct_Search and Lin_Scan are sufficient to determine an assignment for all the vertices in a level of a complete binary tree within the region allotted by find_region.*

Proof (by induction) The region allotted to a level contains

1. some grid points that are forwarded from region bounded by previous square bound (on its periphery).

2. And the grid points from region allotted to the level by ***find_region*** which is an L- shaped region in general, and bounded by the lines $x = s(l); y = s(l); y = l; x = 0; x = s(l-1); y = s(l-1)$.

Divide the region into following subregions (same as in Section 4.1).

$$\begin{aligned}
A\textbf{:}&\quad s(l-1) \leq x \leq s(l), l < y \leq s(l-1)\\
B\textbf{:}&\quad 0 \leq x \leq s(l-2), s(l-1) \leq y \leq s(l)\\
C\textbf{:}&\quad s(l-2) < x \leq s(l), s(l-1) \leq y \leq s(l)\\
D\textbf{:}&\quad s(l-1) < x \leq s(l), s(l-1) < y \leq s(l)
\end{aligned}$$

Step **A1** of ***assign_current*** assigns grid points from *subregion A*, Step **A2** of ***assign_current*** assigns the grid points from *subregion B*, Step **A3** of ***assign_current*** assigns grid points from *subregions C* and *D*.

Base case: A Complete Binary tree with $l_{max} = 4$. From the example (in Figure 4.2), it is clear that *Direct_Search* and *Lin_Scan* are sufficient for embedding vertices up to level $l = 4$.

Induction Hypothesis:
We assume that all vertices up to level $l-1$, are successfully placed using *Direct_Search* and *Lin_Scan* in region allotted to levels up to level $l-1$. The lowest row occupied by a vertex of level $l-1$ is $y = l-1$, and width of the region for this level is $\Delta s(l-1) \approx$

$(2^{1/2}-1)*2^{(l-1)/2}$, using the approximate values given by equation 4.10 in section 4.2.

Induction Step:
For l^{th} level, the lowest row occupied by its vertices is $y = l$, and width of the region allotted to it is $\Delta s(l) \approx (2^{1/2}-1)*2^{l/2}$.

We divide the region occupied by the previous level also, into subregions:

$$\begin{aligned} A': &\quad s(l-2) \le x \le s(l-1), l-1 \le y \le s(l-1) \\ B': &\quad 0 \le x \le s(l-2), l-1 < y \le s(l-1) \end{aligned}$$

So the row-width (i.e the number of columns in A') of *subregion* A' is $\Delta s(l-1)$, implying that each row in A' has almost $2*\Delta s(l-1)$ number of children. The row-width of *subregion* A is $\Delta s(l)$ $(< 2*\Delta s(l-1))$, therefore each row in *subregion* A can be completely assigned with the children of vertices placed in the previous row of *subregion* A'.

Similarly, the column-width (i.e the number of rows in B') of *subregion* B' is $\Delta s(l-1)$, while that of *subregion* B is $\Delta s(l)$, or one more than $\Delta s(l)$, therefore each column in *subregion* B can be completely assigned with the children of vertices in the same column of *subregion* B'.

Note that the column-width of *subregion* B can be one more than $\Delta s(l)$ due to the forwarded grid points from the previous level. Since $2*\Delta s(l-1) - \Delta s(l) > 1$ for $l > 4$, the columns in *subregion* B are completely assigned using child vertices from the same columns for all levels higher than $l = 4$.

The number of child vertices (from *subregion* A') that are not assigned after assigning all required grid points of *subregion* A is given by:

$$\begin{aligned} 2* &\quad \Delta s(l-1)*(s(l-1)-l+1) - \Delta s(l)*(s(l-1)-l) \\ = &\quad \Delta s(l)*((s(l-1)-l)*(2^{1/2}-1)+2^{1/2}) \end{aligned} \tag{4.11}$$

and, number of child vertices (from *subregion* B') that are not assigned after assigning all required grid points of *subregion* B is given by:

$$\begin{aligned} 2 &\quad *\Delta s(l-1)*(s(l-2)+1) - \Delta s(l)*(s(l-2)+1) \\ = &\quad \Delta s(l)*(s(l-2)+1)*(2^{1/2}-1) \end{aligned} \tag{4.12}$$

This is true even after taking variation of one in column-width of *subregion* B into account, for all $l > 4$.

To assign grid points in *subregion* C, all the unassigned child vertices from *subregion* B' are used at first. Next, if some unassigned grid points are left in *subregion* C, unassigned child vertices from same column (that contains the grid point) of *subregion* A' is used. This can be safely done due to the following.

1. The number of columns in *subregion* C that can be completely assigned using unassigned children from *subregion* B' is

$$\begin{aligned} & \frac{\text{number of unassigned child vertices from } \textit{subregion } B' \text{ (equation 4.12)}}{\text{column width of } \textit{subregion } C \ (\approx \Delta s(l))} \\ \geq \quad & (2^{(l-1)/2}+1) * (2^{1/2}-1) > \frac{\Delta s(l-1)}{2^{1/2}} \end{aligned} \tag{4.13}$$

2. The number of child vertices from *subregion* A' that share the completely assigned columns of *subregion* C (equation 4.13) is given by

$$\begin{aligned} 2* \quad & \Delta s(l-1)/2^{1/2} * (s(l-1)-l+1) \\ = \quad & \Delta s(l) * (s(l-1)-l+1) \end{aligned} \tag{4.14}$$

$$\begin{aligned} > \quad & \Delta s(l) * (s(l-1)-l) \\ & \text{(i.e, } > \text{ number of grid points in } \textit{subregion } A) \end{aligned} \tag{4.15}$$

3. From equation 4.15 it can be said that if the assignment of grid points in *subregion* A preferably takes unassigned children of vertices in farthest column of *subregion* A' (starting from column $x = s(l-2)+1$), the children of vertices in the nearer columns can be used for assignment of grid points on the remaining unassigned columns (after having used all child vertices from *subregion* B') of *subregion* C.

Therefore to ensure that all the grid points in the *subregion* C are assigned, the assignment of grid points in *subregion* A preferably takes unassigned children of vertices in farthest column of *subregion* A'. For the same reason, the assignment of grid points in *subregion* B preferably takes unassigned children of vertices in nearest row of *subregion* B'.

Now, the assignment of grid points in *subregion* D can be completed using all the unassigned child vertices without any preference (this is because every grid point (x, y) of this subregion has both $x > Z_x(v), y > Z_y(v)$ for all vertices v of previous level).

Therefore the Steps **A1, A2, A3** are sufficient to completely assign all the grid points in the region determined by the ***find_region*** function for level l.

4.4.2 Optimality of the obtained placement

Theorem 2 *The grid drawing of the complete binary tree obtained by the proposed algorithm satisfies dominance constraints and has minimum square bound.*

Proof (By Contradiction)

Assumption : Let us assume that the square bound obtained by the proposed algorithm is sub-optimal. This implies that there exist some unused grid points within the square bound obtained, which when used could further minimize the square bound.

The proposed algorithm starts assigning vertices of level l from row $y = l$. After **Algorithm 1** completes the assignment of all vertices, the unused grid points within the *square bound* $s(l)$ for the level l, are of two types:

Type 1 : present *above* row $y = l$, and

Type 2 : present *below* row $y = l$.

(All grid points on $y = l$ lying within $s(l)$, are completely used i.e all points such that $0 \leq x \leq s(l); y = l$ are used up.)

Type 1 unused grid points occur either on $x = s(l)$ or on $y = s(l)$. These grid points could be used for some level l vertices without violating the dominance constraints, but it does not minimize the square bound. Usage of *Type 1* points rather vacates an equal number of points within $s(l)$. *Type 2* unused points lie on rows $y = k;$ for $k = 0$ to $l - 1$.

Observation 1 *The unused points on row $y = k$ have abscissa value in the range $s(k) + 1 \leq x \leq s(l)$.*

For levels $l = 0$ to 2, the vertices are placed on the row $y = 0$ to 2, respectively, therefore usage of the unused points on rows $y = k$ ($k = 0$ to 2) by vertices of level k increases the square bound of these levels and vacates equal number of grid points on the row $y = k$. Each vertex from levels 3 to $l - 1$ is either a child or a grandchild of some vertex of level 2. Thus, it cannot be placed on the unused grid points on the rows $y = 0$ to 2. Same holds for placing a level l vertex below row $y = l$. Hence, no level l vertex can be placed on row $y = k;$ for $k \leq l - 1$.

This implies that unused points on row $y = k$ can be assigned only to vertices of levels less than or equal to k. When a level k vertex is assigned to an unused grid point on $y = k$, the square bound of the level k increases beyond the achieved square bound $s(k)$

(due to the above **Observation**).

If the unused grid points on row $y = k$ are used to assign a vertex from level j; for $j < k$, then the square bound of level j would become higher than $s(j)$. Moving any vertex of level j to row $y = k$ vacates used grid points, therefore it does not minimize the percentage of unused points, too. Also, the new place for the j^{th} level vertex prevents its child vertex to be placed on rows $y \leq k$, which may increase the square bound of the next levels too. Thus, neither the usage of *Type 1* nor the usage of *Type 2* unused grid points obtains a lower square bound than $s(l)$, while keeping minimum square bounds for all intermediate levels.

Therefore the square bound achieved by the proposed algorithm is optimal if all levels must also be placed within their minimum square bounds.

4.5 Analysis of the dilation for the grid drawing

Once the placement of the nodes of the tree is determined, the dilation of the tree edges over the grid edges for the embedding gets fixed. To measure the dilation for tree edges that connects a node form level $l - 1$ to its child in level l, we use the subdivisions made in the function ***assign_current***.

For assignment of grid points in A and B the children of nodes from the previous row in A' and same column in B', respectively, are used. The fraction of the children from nodes in a row of A' (column of B') that are assigned to some grid point in the next row of A (same column of B) is given by

$$f' = \Delta s(l)/(2 * \Delta s(l-1)) = 2^{-1/2} \text{ (approx.)} \tag{4.16}$$

hence the fraction of all children from a single row of A' (column of B') that is not assigned to a grid point in A (B) is

$$f = 1 - f' = 1 - 2^{-1/2} \tag{4.17}$$

Preferring placement of a child of node in A' from the farthest column, the dilation for tree edges connecting a node in A' with its child in A is bounded above by

$$\Delta s(l) + f * \Delta s(l-1) + 1 \tag{4.18}$$

For tree edges connecting a node in B to its child assigned to some grid point in B', the dilation occurs only due to the vertical distance between the edge-vertices. The dilation for such edges, preferring placement of a child of node in nearest rows, is bounded above

by

$$\Delta s(l) + f' * \Delta s(l-1) \tag{4.19}$$

The scheme of placement used by ***assign_current***, causes maximum distance separation for tree edges that connects a vertex placed in A' or B' to a vertex placed in C or D. These edges may in worst case have to pass through the regions allotted to both the levels $l-1$ and l. The maximum dilation for a tree edge originating in subdivision B' occurs when it has to terminate in subdivision C or early in columns of D. For a tree edge originating in A', the worst case occurs for one connecting a node in A' to a node in D.

The maximum dilation for a tree edge originating in B' is bounded above by

$$s(l-2) + \Delta s(l) + \Delta s(l-1) \tag{4.20}$$

and for a tree edge originating in A' the maximum dilation is bounded above by

$$s(l) - l + 1 + \Delta s(l) + f * \Delta s(l-1). \tag{4.21}$$

Hence the maximum dilation for the placement attained is bounded above by

$$\begin{aligned} max \quad \{ \quad & s(l-2) + \Delta s(l) + \Delta s(l-1), s(l) - l + 1 + \Delta s(l) + f * \Delta s(l-1)\} \\ = \quad & O(N^{1/2}) \end{aligned} \tag{4.22}$$

Chapter 5

Edge Mapping

5.1 Congestion Analysis

Here we are going to find the maximum value for congestion that is present for any possible edge mapping along grid edges. To analyze the congestion, we divide the mapping into essential mapping, and non-essential mapping. Essential mapping of edges are those for which no alternate paths along the grid edges would be shortest and non-decreasing. The mapping of the edges must be non-decreasing so that the routing of edges do not undo the property achieved by dominance constraint. Keeping the path shortest imply a detour free path.

Non-essential mapping of edges is the one for which some alternate non-decreasing paths of same length exists.

The tree edges which are mapped under essential mapping, contribute to essential congestion. This happens for a tree edge that connects vertices assigned to two grid points in the same column. Such an edge needs to be mapped vertically from the source to the target vertex. Now, the assignment of all grid points in *subregion B* has been done so that the child vertex lie in the same column as its parent. This gives that at least $\Delta s(l)$ edges corresponding to vertices assigned in a column of *subregion B* have to be mapped along the grid edges of that column, vertically. The lowest grid edge in a column, i.e one that connects a grid point from region allotted to level $l-1$ to one from the region allotted to level l, is used by all the tree edges that require to be mapped vertically. Hence the maximum congestion that occurs due to essential mapping of edges is $\Delta s(l)$.

Therefore, the contribution of essential mapping towards the maximum value of con-

gestion is

$$= \Delta s(l) = 2^{l/2}(2^{1/2} - 1) = O(N^{1/2}) \tag{5.1}$$

For a tree edge $e = (u, v)$ directed from u to v, such that their mapping on the grid, given by $Z(u)$ and $Z(v)$ respectively, are in different columns, we define a *bounding rectangle* for the pair (u, v) as a rectangle formed with $Z(u)$ as the bottom-left corner and $Z(v)$ as the top-right corner. A non-decreasing detour-free path for the edge $e = (u, v)$ will lie within the *bounding rectangle* for (u, v).

The ***assign_current*** module uses a grid point from a different column than one containing the parent vertex, during assignments into the *subregions* A, C and D. Of these the grid points in rows of A have been assigned a child vertex of a vertex placed in the immediate previous row in A', therefore the edges for these parent-child vertex pair have a choice in selecting the location of its vertical grid edge, to move from the lower row to the next one. The sub-path of the edge map lying before and after the vertical edge is essentially horizontal, again contributing to essential congestion in the horizontal direction.

Despite a choice, the congestion contribution of mapping edges for vertices assigned to *subregion* A is essential in nature. Again, the grid edge that connects the rightmost grid point from A' to the leftmost grid point from A in the same row is used by $\Delta s(l)$ edges, half of which is for edges corresponding to a vertex in the same row, on an average, and other half for those in the next row.

The *bounding rectangles* for parent-child vertex pairs of which the parent is in B' or A' and child is in C or D, have possibility of many different monotone paths. Some of these rectangles are overlapping which imply that of many different paths possible there are some which require passing through a region required by paths for other parent-child pairs. Inorder to minimize the congestion we want to select such paths for different tree edges so that the number of overlaps on a single grid edge is minimized. We are in search of an edge mapping algorithm for our scheme of placement that does an optimal selection of the paths for the tree edges and thereby minimizes the congestion.

For average analysis of the congestion that could occur by a good selection of the paths for the tree edges we do the following theoretical study.

From earlier sections, Sec. 4.5, we know that about $f * 2 * \Delta s(l-1)$ child vertices from level $l-1$, of those in each row of A' or a single column of B', are not placed in the immediate next row orin the same column, respectively. The *bounding rectangles* corresponding to these parent-child pairs are overlapping. The maximum number of overlaps among the

bounding rectangles with their bottom-left corner in *subregion* B', is given by

$$\begin{aligned} & (s(l-2)+1) * f * 2 * \Delta s(l-1) \\ = \quad & (s(l-2)+1) * (2^{1/2}-1) * \Delta s(l) \end{aligned} \tag{5.2}$$

similarly, the maximum number of overlapping *bounding rectangles* with their bottom-left corner in the *subregion* A' is given by,

$$\begin{aligned} & (s(l-1)-l+2) * f * 2 * \Delta s(l-1) \\ = \quad & (s(l-1)-l+2) * (2^{1/2}-1) * \Delta s(l) \end{aligned} \tag{5.3}$$

There can also be some overlap among *bounding rectangles* of some edges originating in A' and B'. This happens when children of a vertex in B' are assigned in D or assigned in C on a column that has nodes with unassigned children in A' after complete assignment of *subregion* A.To understand it, refer to Figure 5.1.

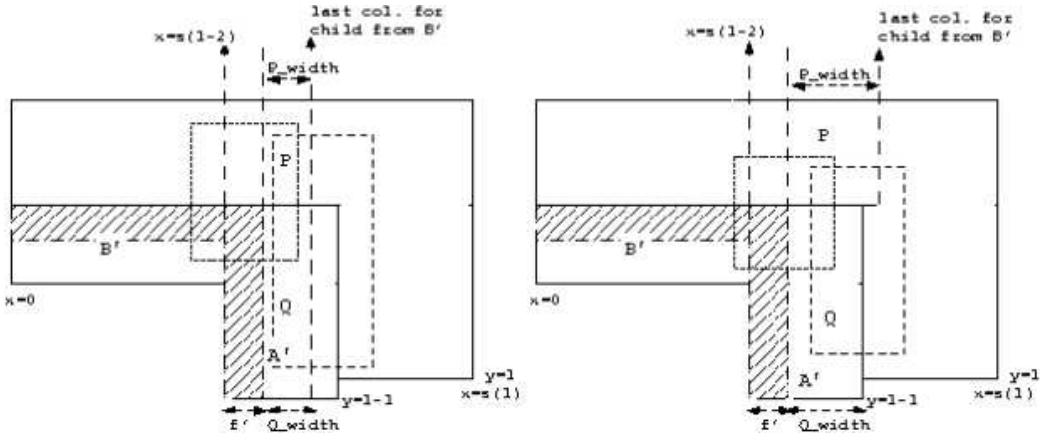

Figure 5.1: Overlap between *bounding rectangles* originating in B' and A'

In the figure the shaded region is the f' fractional part of A' or B' that has nodes with its children assigned to A and B respectively. P denotes subpart of C on or beyond column $x = s(l-2) + f' * \Delta s(l-1)$ and of D that has been assigned a child of node in B'. Q is that subpart of A' such that the *bounding rectangles* originating in A' overlaps with one terminating in P. The number of columns in P is obtained using number of columns beyond $x = s(l-2)$ required to place the children from B' and number of columns in shaded part of A'. Using equation 4.13 and 4.16, we obtain the number of columns in P is $\geq (1-f) * \Delta s(l-1) + (2^{1/2}-1)$. The columns shared by P with columns in *subregion* A', are the ones present in Q, therefore the *bounding rectangles* terminating in P or originating in Q are overlapping.The maximum number of such overlaps among *bounding rectangles* is given by,

$$\Delta s(l) * (\Delta s(l-1) * (1-2^{-1/2}) + 2^{1/2} - 1)$$

$$+2 * (s(l-1) - l + 2) * (\Delta s(l-1) * (1 - 2^{-1/2}) + 2^{1/2} - 1)$$
$$= (\Delta s(l) + 2 * (s(l-1) - l + 2)) * (\Delta s(l-1) * (1 - 2^{-1/2}) + 2^{1/2} - 1) \quad (5.4)$$

For the rectangle with largest number of overlaps of *bounding rectangles*, either the height or the width is $\leq \Delta s(l) + \Delta s(l-1)$. If the height or width is less than $\Delta s(l) + \Delta s(l-1)$, all the tree edges corresponding to the overlapping *bounding rectangles* may not mandatorily pass through this rectangular region. But otherwise, each such tree edge needs to cross the heavily used rectangular region. So, it would take its path through one of the $\Delta s(l) + \Delta s(l-1)$ rows or columns in the rectangle. On an average, a single grid edge on a row (column) of such a rectangle would be used for

$$\begin{aligned} & \frac{(s(l-2)+1) * (2^{1/2} - 1) * \Delta s(l)}{\Delta s(l) + \Delta s(l-1)} \quad or, \\ & \frac{(s(l-1) - l + 2) * (2^{1/2} - 1) * \Delta s(l)}{\Delta s(l) + \Delta s(l-1)} \quad or, \\ & \frac{(\Delta s(l) + 2 * (s(l-1) - l + 2)) * (\Delta s(l-1) * (1 - 2^{-1/2}) + 2^{1/2} - 1)}{\Delta s(l-1)} \\ = \quad & O(N^{1/2}) \end{aligned} \quad (5.5)$$

times.

The essential congestion given by equation 5.1 gives a lower bound on the value of maximum congestion. The equation 5.5, gives a theoretical lower bound on the maximum congestion caused by the non-essential edge mapping. However, such a uniform distribution of tree edges over the grid edges might not be possible without a detour. So, we have obtained a lower bound of $O(N^{1/2})$ on the maximum value of the congestion for the dominance constrained placement of complete binary trees given by **Algorithm 1**.

Our aim in the next section will be to achieve a scheme for edge mapping that minimizes the maximum congestion.

5.2 The Edge-Mapping Algorithm

Inorder to complete the drawing, we need to map the tree edges along the grid edges. This we call as the edge-mapping. During edge-mapping we need to ensure that no detours exist, else the delay through the path would increase. For this we follow the grid edges in a non-decreasing manner both vertically and horizontally, so we map the tree edges over the grid edges in rightwards and upwards direction.

Presence of dilation in the drawing indicate that a single tree edge would be mapped over a set of continous grid edges. As described earlier, the placement of the vertices over the planar grid is a non-planar drawing, therefore a single grid edge may be used for a number of tree edges. The overlap of tree edges implies a number of tracks in the channel of the VLSI chip, this forces an increase in the channel width. Increment in the channel width itself leads to an increase in the area of the VLSI chip. Therefore, during the edge mapping phase we also try to minimize the congestion.

Though it is difficult to develop an edge mapping algorithm that optimally selects the path for each tree edge so as to evenly distribute the congestion, we propose an edge-mapping algorithm that does a local search for each tree edge to follow a direction that keeps the congestion low. It achieves the edge-mapping without altering the area achieved in the placement. Later on we would want to study the effect of imposing a bound on the congestion, on the area requirement for the placement. We may even choose a minimum detour path to satisfy a bounded congestion edge mapping.

In this section we discuss the edge mapping algorithm followed by us in ***Edge_mapper***.

A grid edge g can be recognized by its starting grid point $st\_pt(g)$ and its end grid point $e\_pt(g)$. Of the four grid edges intersecting at any grid point, detour free path permits selection of only two: one moving upwards and other rightwards only. As we intend to keep the congestion low we store the number of times a grid edge g has been used in an associated variable $congestion(g)$. Also the maximum value of $congestion(g)$, over all used grid edges g, is stored in variable max_cong. We start the mapping of a tree edge $e = (u, v)$ directed from u to v, from the grid point $Z(u)$ as the source src_pt. The mapping of the tree edge is complete when the tree edge e reaches the destination grid point $Z(v)$, we call it $dest\_pt$. The current status for the tree edge e is known by the end point $e\_pt(g)$ of the last grid edge followed by e. If $e\_pt(g)$ of the last grid edge followed by the tree edge is not equal to the $dest\_pt$, $e\_pt(g)$ becomes the next source point src_pt for the path of the tree edge.

At any src_pt, the direction of the path that the tree edge should take is to be decided. Before choosing the direction of the path for the tree edge, we need to determine if there exists some option for the edge direction or not. This is done by testing if the current status src_pt of the tree edge lies on the final row or the final column of the *bounding rectangle*. When this is true, the rest of the path to be taken by the tree edge becomes fixed. If the new src_pt and the $dest\_pt$ lies on the same column the rest of the path must be strictly upwards.

ALGORITHM 2: MAPS TREE EDGES ONTO GRID EDGES

Edge_mapper(G, Z, l_{max}, s)

Input: Z, the dominance constrained drawing after the placement phase

Output: Completed dominance drawing with mapped tree edges

E1 Initialize congestion value for all grid edges to zero, and $max\_cong := 0$

E2 for each level l of complete binary tree do

for each tree edge e terminating at vertex $v$of the level l do

a. Let u be the parent vertex for v, then

$src\_pt := Z_x(u), Z_y(u)$; $dest\_pt := Z_x(v), Z_y(v)$

b. Let $st\_pt(g), e\_pt(g)$ denote the start grid point and the end grid point of the grid edge g

if $src\_pt(x) = dest\_pt(x)$ then

choose vertical edge g' with $st\_pt(g')$ equal to src_pt

$congestion(g') := congestion(g') + 1$

else if $src\_pt(y) = dest\_pt(y)$ then

choose horizontal edge g' with $st\_pt(g')$ equal to src_pt

$congestion(g') := congestion(g') + 1$

else

choose the edge g' with $st\_pt(g')$ equal to src_pt and smaller $congestion(g')$ value, prefer horizontal edge if edges in both directions have same congestion value

c. $src\_pt := e\_pt(g')$

Update the $pathcode(e)$

if $congestion(g') > max\_cong$ then

$max\_cong := congestion(g')$

end if

Repeat from Step **E2 b.** until $src\_pt = e\_pt$

end for

end for

E3 for graph edge e using a grid edge g with $congestion(g) = max\_cong$ do

Rip_reroute$(e, pathcode(e))$

end for

Similarly, if the new *src_pt* lies on the same row as the *dest_pt* the rest of the path must be strictly rightwards. The choice for the direction of the path stays for other status values (*src_pt*) of the path.

Selection of a direction for the path, is done to ensure that it minimizes the usage of a

grid edge that is already used for a higher number of tree edges. The tree edges that are mapped earlier than others would mostly find a low congestion value for both horizontal and vertical grid edge starting at the *src_pt*. In such cases when both of the possible grid edge that could be taken are equally congested, preference is given to the horizontal edge. This is done since many of the tree edges would be having essential mapping in the vertical direction. The path taken by the tree edge e is maintained with the tree edge as a $pathcode(e)$. For every selection of the direction the $pathcode(e)$ is updated. The $pathcode(e)$ is a binay string with: 0 specfying horizontal direction and 1 the vertical direction. The length of the $pathcode(e)$ is equal to the value of the dilation present in the tree edge e.

FUNCTION 1 : FINDS AN ALTERNATE LOW CONGESTED PATH	
Rip_reroute$(e, pathcode(e))$	
Input:	tree edge e with its $pathcode(e)$
Output:	A new $pathcode(e)$ for e with lower max. congestion
R1	Follow the grid edges g in $pathcode(e)$ and decrement $congestion(g)$
R2	Use Step **E2** in **Algorithm 2** to update $new\_pathcode(e)$
	(* max_cong is unaltered *)
R3	If $congestion(g)$ for all grid edges in $new\_pathcode(e) < max\_cong$ then
	decrement max_cong value
	$pathcode(e) := new\_pathcode(e)$
	else Follow grid edges g in $pathcode(e)$ to increment $congestion(g)$
	end if
R4	Return $pathcode(e)$

As another atempt in minimizing the congestion, we remove the mapped tree edge that has been moved through a highly congested grid edge and re-route it through some other path along the grid edges. This path may be a new one with grid edges that have a lower congestion value than the maximum congestion, or remain the same if no alternate path can bypass a highly congested grid edge. Re-routing, done by ***Rip_reroute***, uses the same strategy as the initial routing. The initial routing has been done in absence of many tree edges, therefore the path taken by the early mapped tree edges uses the first least congested direction. Due to this many options for the path, that the tree edge could take, have not been tested. We would like to try some of these options for the already mapped edges to check if the maximum congestion value could be decreased. Re-routing is an attempt to minimize the maximum value of the congestion by re-mapping some of the tree edges. The tree edges that have been mapped early, may later on end having used most congested grid edges. So we search for an alternate path for such tree edges, after all the tree edges have been mapped.

If an alternate path that decreases the use of highly congested grid edge, could be found, the $pathcode(e)$ and the congestion values are duly updated, otherwise the old $pathcode(e)$ is left unchanged. As a result we could decrease the maximum congestion achieved or atleast the frequency of occurrence of the maximum congestion. This step could be repeated a number of times, involving repeated scan for a low congestion path for each tree edge. This would be not be cost-effective, (as the number of tree edges per level that are actually re-mapped are only a few), hence we use ***Rip_reroute*** only a few times.

Chapter 6

Offline Improvement of Placement Area

The proposed algorithm **Algorithm 1** in section 4.1 minimizes the square bound for each level in the complete binary tree. Due to this, there are many unused grid points with ordinate value $y \geq k$ and lying between columns $x = s(k)$ and $x = s(l)$, for $k < l$. These are unused during assignment of nodes in level k, as it would increase the square bound $s(k)$, for the level k. To minimize the square bound for placement of an N- node complete binary tree, there is still the scope of using these unused grid points, without trying to minimize the square bound at each level.

To achieve a global minimizaion of the square bound, we insert some additional steps, into **Algorithm 1**.

6.1 Algorithm for improvement

Our approach is to first find the square bound s, required for the dominance drawing of an N- node complete binary tree, by the repeated use of ***find_region*** without any actual assignments made. The final value of s gives an initial value for the square bound and is the one achieved by **Algorithm 1**.

We do not store the square bounds found for the intermediate levels. Before we begin with the assignments, the square bound s is reduced, so as to minimize the number of unused grid points enclosed within the square bound s. For this, we assume that maximum number of nodes of a single level can be placed on a single row, in the found region. The number of grid points on a single row of the found region is $s + 1$, so levels having less

than $s+1$ nodes would be completely assigned to grid points on a single row. If there are some unused grid points left on the row corresponding to a level with $< s+1$ nodes, they cannot be used by the next level nodes.

Let k' be the last level having $< s+1$ nodes. So each level l up to level k' occupies a single row given by the row $y = l$. All usable grid points on the rows $y \leq k'$ have been reserved for assignment. The number of reserved grid points on these rows is more than the number of grid points (on rows $y \leq k'$) used in **Algorithm 1**. So if we continue to use all usable grid points on the next rows too, without changing s, a number of unused grid points will appear on the top rows, i.e rows nearer to $y = s$. If the number of unused grid points on the non-boundary top rows (boundary row is $y = s$), is more than the number of grid points reserved for use, on the column $x = s$, the square bound s can be decremented. This is repeatedly done until there are not as many unused grid points on the non-boundary top rows as there are used grid points on the boundary column $x = s$.

ALGORITHM: IMPROVES ON *Algorithm 1* TO GET GLOBAL MINIMUM AREA

Input: G, a complete binary tree with height l_{max}

Output: s, minimum square bound, Z

K1. (*Get the square bound s obtained in *Algorithm 1**)
$Z := (0,0); l := 1; s := 0; forward(l) := 0;$
Repeat while $l \leq l_{max}$
  find_region$(s, forward(l))$
  $l := l + 1$

K2. Calculate largest k' such that $2^{k'}$(# nodes in level k') $\leq s+1$
$N' := N - (2^{k'+1} - 1)$; (N, the number of nodes in G)
$n\_rows := \lceil N'/(s+1) \rceil$;
if $(k' + n\_rows < s)$ then
  Count number of unused grid points, *count*, on rows $(k' + n\_rows) \leq y < s$
  if *count* > number of points on column $x = s$ with $y \leq k' + n\_rows$
    $s := s - 1$;
    Repeat from step **K2**.
  end if
end if

K3. *assign*(s, l_{max}, G)

This gives us the globally minimum square bound for the placement of the N- node complete binary tree, with least number of unused grid points. Also, that these unused grid points cannot be used at all under the dominance constraints. Here too, the assignments are done level by level, with the lower rows being used before utilizing the higher rows.

Due to this the nodes of a level l, are placed above the region containing the nodes from level $l-1$, for all l. The method followed to complete the assignment of *subregion* B and D, by ***assign_current*** of **Algorithm 1**, is sufficient to completely assign all nodes of a level in this scheme too. Therefore the function ***assign_current*** needs no modification except that only the steps pertaining to subregion **B** and **D** is applicable.

FUNCTION 1: DOES THE ASSIGNMENT FOR THE VERTICES OF THE TREE

assign(s, l_{max}, G)

Input: improved square bound s, l_{max} and G

Output: Z, the completed assignment

L1. For levels l, with $2^l \leq s+1$,(i.e $l \leq k'$) place all vertices on row $y = l$ (with one vertex on the same column as its parent and other to the right of all columns containing a parent vertex using grid points from left to right). Keep in *unused*, the count of unused points on row $y = k'$.

L2. For levels l, with $2^l > s+1$

if $l = k'+1$ then

for columns $x \leq s - 2 * unused$, place both children on the same column as their parent vertex.

for columns $s - 2 * unused > x \leq s - unused$, place one child vertex on the same column as its parent, and remaining vertices of level l into the columns $x > s - unused$.

else if $l < l_{max}$ then

place all child vertices on the same column as their parent (preferably children of nodes in the nearer rows).

else

place as many child vertices in the same column as their parent, till the grid points have ordinates $\leq s$ and the remaining child vertices on the columns $x > (s + 1 - 2 * unused)$ using *Lin_Scan* (refer section 4.1)

end if

L3. Return

During assignment phase, the vertices in levels $l \leq k'$ are placed in a single row $y = l$, from left to right, placing a child node into the same column as its parent, if possible. The number of unused grid points that are left on the row $y = k'$ after assignment of all vertices from level k', is stored as *unused*. For the next level $l = k' + 1$, the child vertices of nodes in the columns $x \leq s - 2 * unused$ are placed in the same columns as their parent. The child vertices of nodes in columns $x > s - 2 * unused$ are equally distributed on the remaining (free) grid points on row $y = k' + 1$, i.e one child vertex is placed in the same column and the other in the columns not having any parent level node.

For the higher levels $l > k' + 1$, we place all the child vertices in the same column as their children. These assignments would be on grid points within the square, formed by the square bound s, for all levels except for the last level $l = l_{max}$. Hence we only place as many vertices of level l_{max} as could be placed on available free grid points with ordinate values $\leq s$, in each column.

The number of grid points in each column, that is used for assignment, is not equal. As we have been using more number of grid points from columns $x \leq s - 2 * unused$ than from the other columns to the right, there are unused grid points available on the columns $x > s - 2 * unused$). Throughout the assignments for levels $l \geq k' + 1$, the total number of grid points that have been used in each column $x \leq s - 2 * unused$, has been greater by $2^{l-k'} - 1$, from the number of grid points used in each of the remaining columns. Therefore, the remaining unassigned vertices of level l_{max} can be placed into these unused grid points using *Lin_Scan*, described in section 4.1. Also, to be cautious that all remaining vertices could be placed at some of these grid points under dominance constraints, we include the preference of placing the child vertices of parent level nodes in the nearer rows (nearer to the rows that are beng assigned) during their assignment to the same column as their parent vertex.

Chapter 7

Extension to DAGs

7.1 Need for modification

In the previous sections, the algorithm for placing nodes in a complete binary tree, and edge-mapping on the obtained drawing, both under the constraints of dominance drawing have been discussed. We have obtained an area-efficient algorithm for the complete binary tree, we would now like to analyze the square bound and area requirement for directed acyclic graphs. Directed acyclic graphs differ from a complete binary tree but are a closer depiction of the VLSI circuits.

The difference between the two graphs (complete binary tree and directed acyclic graph), that effects the proposed method applicable to complete binary tree and requires modifications is:

1. No bound on the indegree and outdegree.
2. Data dependence is not restricted to the immediate previous topological level.

Complete binary tree has a fixed indegree and outdegree of one and two respectively, this gives the analysis for the square bound of the kind described in section 4.2. In absence of a bound on degree, any vertex with in-degree > 1 and out-degree ≥ 1, can become a source for pushing the square bound higher. This happens as the vertex having an indegree > 1 is a convergence point for its incoming edges. Due to the convergent edges that may be present, the vertex becoming the point of convergence can only be placed on or after the last column containing one of its predecessor vertices. Also, such a vertex must be placed above any row containing its predecessor. As a result, many grid points on lower rows and columns cannot be used, forcing the assignments to be denser along the right diagonal of the square grid.

To recognize the data dependence in the directed acyclic graphs, topological order of the vertices is considered. Topological order defines the levels in a DAG. Like for assignment of a complete binary tree, the assignment of nodes in a DAG under the dominance constraints, require that a level l node be placed on or above a row $y = l$. Again, in absence of a restriction on out-degree of vertices, the number of vertices in different topological levels varies, sometimes to very low and sometimes very high. A very less number of nodes in a level may also lead to sparse usage of the grid points within the obtained square bound.

The data dependence is not limited to be from the immediate previous topological level, rather the predecessors can be from any of the previous topological levels. Therefore the region on the grid that contains the predecessors of nodes in a particular level, may span whole of the square region defined by the assignments made before the assignments for nodes of that level.

These differences with respect to the complete binary tree, causes us to modify our algorithm for the dominance drawing of a directed acyclic graph.

7.2 Placement of DAG vertices

Let G denote a single source connected directed acyclic graph. The assignment $Z(v)$ of a vertex $v \in G$, must satisfy the dominance constraints defined earlier. According to which, the y- coordinate $Z_y(v)$ of a node v must be strictly greater than that of its predecessor nodes, and the x- coordinate $Z_x(v)$ must be equal or greater than that of its predecessors. The proposed algorithm for the directed acyclic graph is also incremental, i.e assignments to the vertices are made level after level.

We place the source vertex, src, (with indegree 0) at the origin. As G is single sourced, the square bound for the level zero is $s = 0$. For assigning the vertices in the next level $l = 1$, we need to have the grid points on the rows $y \geq 1$. There are no unused grid points in the achieved square bound that is usable in the new level, so the number of forwarded grid points is zero, $n\_forwardpt = 0$.

The region on the grid that contains all the predecessors of a node in level l, is not limited to the region containing nodes from level $l - 1$ only. The predecessors may be from any of the levels preceding level l. As the edge connections may be random, it is not possible to strictly define the predecessor region. Rather, we assume that any assigned node from the previous levels that has some unassigned successor node can be one of the predecessors for nodes in the current level l. Therefore, we keep all assigned

nodes with some unassigned successor into the array *prev_assgn* to define the region for predeccessor nodes. The array *prev_assgn* is updated after assignments to all the nodes of a level is complete: including new nodes just assigned and deleting nodes with all assigned successors.

ALGORITHM 3: FINDS DOMINANCE CONSTRAINED DRAWING FOR DAG

Input: topologically sorted DAG G, the number of topological levels l_{max}, a grid

Output: s, the minimum square bound for the drawing, Z

M1 (*Assign the src vertex to origin*)
$Z(src) := (0,0)$; $l := 1$; $s := 0$; $n\_forwardpt := 0$; $forward\_list :=$ NULL;
prev_assign $:= src$; $xmin := 0$; $ymin := 1$;

M2 Repeat while $l \leq l_{max}$
$N_l :=$ Number of nodes in level l
build_arr(***prev_assign, Xarr, Yarr***)
if $n\_forwardpt > 0$ then
$k :=$ ***f_assign_current(forward_list, Xarr)***
$n\_forwardpt := n\_forwardptk$;
$N_l := N_l - k$;
end if
if $N_l > 0$ then
find_region(N_l, s)
d_assign_current(forward_list, n_forwardpt,Xarr,Yarr,s)
end if
Update ***prev_assgn*** to include all assigned nodes with unassigned successors
$xmin := \min\{Z_x(v) : v \in prev\_assgn\}$
$ymin := l + 1$
Update $forward\_list$ and $n\_forwardpt$, adding unused grid points within (s, s) and removing any grid point (x, y) with $x < xmin$ or $y < ymin$

M3 Print s.

M4 (* Map edges to complete the drawing*)

The region that would contain the vertices from a level l, lies on and above the row $y = l$, and on and to the right of column $x = xmin$. Here the value of $xmin$ is defined by the lowest column that contains a node in *prev_assgn*. As there is no node $u \in prev\_assgn$ with $Z_x(u) < xmin$, no assignments for vertices in the current and the forthcoming levels shall be in columns to the left of column $x = xmin$.

The predecessor region (that contains all the nodes in *prev_assgn*) is not divided into subregions that was done for the complete binary tree. This is because the distribution of

the nodes that have successors in the current level is not as uniform as was for complete binary tree. In addition to this, the successors of a node in *prev_assgn*, belonging to the current level, and having in-degree > 1, may not be put on the same column, unless all its predecessors are placed on or before that column. Similarly, the successors with indegree > 1 may not be put in the immediate next row. **Algorithm 1** is modified here, and we make lists of current level nodes that could be allocated to a grid point in column or row. This is done by ***build_arr*** function.

FUNCTION 1: PREPARES LIST OF CURRENT LEVEL NODES FOR EACH ROW/COLUMN

build_arr$(prev\_assgn, Xarr, Yarr)$

Input: *prev_assgn*, list of assigned nodes from previous levels

Output: column-wise($Xarr$) and row-wise($Yarr$) lists of current level nodes

B1 for each column $x = xmin$ to s do
  for each node $u \in prev\_assgn$ assigned in the column x do
    Scan successors v of u and put one having all its predecessors assigned to grid points in columns on or before column x into $Xarr[x]$, along with $min\_y = \max\{Z_y(pred(v))\} + 1$
  end for
 end for

B2 for each row $y = ymin$ to s do
  for each node $u \in prev\_assgn$ assigned in the row $y - 1$ do
    Scan successors v of u and put one having all its predecessors assigned to grid points in rows on or before row $y - 1$ into $Yarr[y]$
  end for
 end for

B3 Return

The ***build_arr*** function scans through the successors of the nodes in *prev_assgn* array, and those having all its predecessors placed but is itself unassigned, is put into $Xarr$ or $Yarr$. $Xarr$ is a list with column-wise partitioning, one for each column from $x = xmin$ to $x = s$, that contains some node in *prev_assgn*. Similarly, $Yarr$ is a list with row-wise partitioning, one for each row from $y = l$ to $y = s$. Due to topological sorting of the DAG, only the nodes of the current level, can have all its predecessors placed, therefore the successors put into $Xarr$ or $Yarr$ belong to the current level. Also, the partition of $Xarr$ or the column which contains a node is the one corresponding to the column of a predecessor having largest abscissa value for its assignment. A partition of $Yarr$ contains the nodes having all its predecessors placed below the corresponding row, and atleast one predecessor placed in the immediate previous row.

It is to be noted that a node present in a partition of *Xarr* or *Yarr* corresponding to column $x = x'$ or to row $y = y'$ respectively can be placed on any columns $x \geq x'$ or any rows $y \geq y'$.

Before, finding out the potential region for assignment of vertices in the level l, the usable (with abscissa $\geq xmin$ and ordinate $\geq l$) forwarded grid points lying within the last achieved square bound are tried out to assign as many nodes as permitted under the constraints.

The forwarded grid points can exist deep into the square region defined by the achieved square bound s. This is unlike the case of dominance drawing for a complete binary tree, where they are located only on the final boundary of the achieved square bound.

The function ***f_assign_current*** does the assignment for the grid points in the *forward_list*. We still fill the grid points in the lowest row, if there is some node that could be allocated to them under dominance constraints. The array *Xarr* is used for this purpose as all the forwarded points lie on some column from $x = xmin$ to $x = s$. But, here we also require to check if the node that is allowed to be placed on the column containing the forwarded grid point, does not violate the dominance constraint for the y-coordinates. For this, we also keep an associated variable with each node, *min_y*, that specifies the minimum ordinate value that the grid point must satisfy. By the constraints, *min_y* is the lowest row $y = y'$, such that all the predecessor nodes are placed on rows $y < y'$.

FUNCTION 2 : DOES ASSIGNMENT INTO GRID POINTS OF FORWARD_LIST

f_assign_current$(forward\_list, Xarr)$
Input: lists of forwarded grid points $forward\_list$, and column-wise nodes $Xarr$
Output: assigns into some forwarded points, returns number of points assigned
FA1 Sort $forward\_list$ in ascending order w.r.t ordinate values,
followed by ascending order w.r.t abscissa values;
$k := 0$;
FA2 for each grid point (x', y') in $forward\_list$ do
assign (x', y') to an unassigned node from $Xarr[i](i \leq x')$ with $min\_y \leq y'$
(preferably from column $i = x'$)
if (x', y') is assigned then
$k := k + 1$;
end if
end for
FA3 Return k;

The function ***find_region*** is similar to the one described for a complete binary tree. It finds the potential region for the placement of the yet unplaced nodes of the current level l, by counting as many grid points as there are unassigned nodes in the level l. Unlike, for complete binary tree, the potential region found may not be sufficient to place all the nodes. This is an outcome of the random degrees and edge connections, as the nodes may be pushed towards the higher columns and rows to satisfy the constraints for each successorpredecessor pair. Yet, it is only during the assignment into this region that we find the grid points that do not have any unassigned node left to be assigned to them.

FUNCTION 3: FINDS POTENTIAL REGION FOR THE TOPOLOGICAL LEVEL l	
find_region(N_l, s)	
Input:	N_l, the number of vertices in level l
Output:	s, the minimum square bound
F1	Repeat while $N_l > 0$
	$s := s + 1$;
	Count (*count*) the grid points on the new boundary line with abscissa $\geq xmin$ on and above row $y = l$
	$N_l := N_l - count$;
F2	$(last\_x, last\_y) :=$ coordinates of the last grid point counted in Step **F1**
	Return

Therefore, we have another version of the ***assign_current*** function called ***d_assign_current***, that may have to dynamically decide the region to completely assign all nodes of the level, in some cases.

In ***d_assign_current***, we bring in some modifications to cater our needs for dominance drawing of a directed acyclic graph.

Firstly, the current region is not sub-divided in the same fashion as was done in ***assign_current*** (for complete binary trees), as we cannot assure that all the grid points in the region could be used. Rather here we assign the grid points in a row-wise fashion, starting from the lowest row, to maximize usage of lower grid points that may help achieving less increments on s. Only for sake of convenience we divide the rows of the region into

subregion A containing grid points with $s(l-1) < x < s(l), l \leq y \leq s(l-1)$ (here, $s(l-1)$ is the square bound achieved for level $l-1$, and $s(l)$ the currently known square bound for level l), and

subregion B containing the grid points on higher rows, i.e points with $xmin \leq x \leq$

$s(l), s(l-1) < y \leq s(l)$.

The rows in *subregion* A being lower than rows having some predecessor nodes assigned, the array $Yarr$ is used to fill grid points in them. For grid points in the rows of *subregion* B, ordinate value is not a constraint anymore, therefore only $Xarr$ is used to select the unassigned node to be placed.

FUNCTION 4: COMPLETES ASSIGNMENT FOR ALL UNASSIGNED NODES IN LEVEL l

d_assign_current$(forward\_list, n\_forwardpt, Xarr, Yarr, s)$

Input: $forward\_list$, lists of current level nodes $Xarr, Yarr$

Output: Completed dominance constrained placement for the level l

DA1 Prefer assignments in grid points on lowest row under the following rules (as applicable):

i. For each grid point (x', y') on rows $y = y'$ of subregion A, assign an unassigned node from $Yarr[i](i \leq y')$, prefering one from $Yarr[y']$ with highest out-degree.
if no unassigned node found then
go to Step **DA2**
end if

ii. For each grid point (x', y') on rows $y = y'$ of subregion B, assign an unassigned node from $Xarr[i](i \leq x')$, prefering one from $Xarr[x']$ with highest out-degree.
if no unassigned node found then
go to Step **DA2**
end if

DA2 Include (x', y') in $forward\_list, n\_forwardpt := n\_forwardpt + 1$;
if $(last\_x, last\_y) = (s, s)$ then
$s := s + 1$;
$(last\_x, last\_y) := (s, ymin)$;
else
$(last\_x, last\_y) :=$ coordinates of the uncounted grid point next to $(last\_x, last\_y)$
end if
go to Step **DA1**

Secondly, if any grid point cannot be assigned, due to lack of unassigned nodes that could be placed there, a new grid point is included into the region. The new grid point is one that is next to the last grid point, (*last_x, last_y*), previously counted into the region. Note that the grid points on a new boundary line (formed by $x = s$ and $y = s$) are counted in ascending order of their ordinate values if abscissa $= s$. And for points with

same ordinates they are counted in ascending order of abscissa values. If (*last_x, last_y*) was the grid point (s, s), we would require incrementing s, to get the new grid point.

In case the new grid point is on a lower row than one that contains the grid point that was being assigned, its assignment is done before other grid points.

Thirdly, a node with greater out-degree is preferably placed on a lower row or column. This is done to increase the chance for its successors to be placed without requiring many increments in the square bound. Without this clause, a predecessor node placed on a higher row or later column, will have all its successors to be placed above its row or after its column, which may push the square bound higher.

Chapter 8

Implementation

8.1 Implementation of Algorithm 1 for Complete Binary tree

The input is a full complete binary tree, i.e for an N- node tree, level l has 2^l nodes (root is considered to be at level $l = 0$). The input tree is considered to be labeled from the root level to the leaf level, and at each level from left to right. Each vertex of the tree is recognized by its label. Also, being full and complete, each non-leaf vertex has two children. If the label of a vertex is i, the label of its left child is $2i$ and of its right child is $2i+1$.

Information associated with each vertex v of the complete binary tree is :

1. its label
2. assignment $Z(v) = (Z_x(v), Z_y(v))$
3. assignments for its child nodes.

For this, we define a C-*struct*ure as follows:

```
struct vertex {
        int key[3];          /*   key[0] = Z_x(v), key[1] = Z_y(v),
                                  key[2] = label(v) */
        int child1_xy[2];    /*   assignment for its left child node,
                                  initially (-1,-1) */
        int child2_xy[2];    /*   assignment for its right child node,
                                  initially (-1,-1) */
    };
```

Initial value for the assignments of the left and right child is kept at $(-1, -1)$ to indicate that they are unassigned.

To choose assignments for nodes in a level, one has to know the assignments of their parent nodes. All parent nodes exist in the previous level. With information about the child assignment stored along with a node, an array of nodes in the parent level is kept during iteration for each level.

```
struct vertex ** Xarr, ** Yarr, ** Yarr_x;
```

The size of the dynamic array is given by the number of nodes in the parent level.

The nodes in *subregion* A' of the parent region on the grid, is stored in *Yarr*, and the nodes in *subregion* B' in *Xarr*. *Yarr* is used to assign grid points in *subregion* A, *Xarr* is used for assigning into *subregion* B. *Xarr* and *Yarr_x* (a copy of *Yarr*) is used for assignment of *subregions* C and D. The preference for assigning a child node from farther end of each row (form A' to the next row in A) is obtained by sorting *Yarr* first in increasing order of y- coordinates (i.e *key[1]* values of nodes). This partitions *Yarr* into subgroups with same *key[1]* value, each subgroup represents a row in A'. Each subgroup is then sorted in increasing order of x- coordinates (i.e *key[0]* values).

The preference for assigning a child node from nearer end of each column (from B' to the same column in B, or from A' to same column in C, if required) is obtained by first sorting *Xarr/ Yarr_x* in increasing order of x- coordinates (i.e *key[0]* values). This divides *Xarr/ Yarr_x* into subgroups, each subgroup represents a column in B' / A' respectively. Each subgroup is then sorted in decreasing order of y- coordinates (i.e *key[1]* values).

Sorting of the arrays is done using *heapsort* algorithm.

There is also a possibility of changing the order within subgroups of *Xarr, Yarr_x, Yarr,* this would change the preferences for the assignment into grid points from a row in A, and from a column in B, C and D. For smaller values of level, a change of order within the subgroups of *Xarr*, may not allow completion of assignment within the potential region found for these levels. With a small adjustment : to stick with decreasing order within a subgroup of *Xarr* for levels $l < 5$, all the 8 combinations of sorting orders within *Xarr, Yarr, Yarr_x* successfully completes the assignment within the minimum square bound found by ***find_region***.

Variations in placements created by altering combination of the sorting orders within subgroups also result into different dilation and congestion values.

Each vertex of the host graph, a 2 -dimensional planar grid, is represented by a point structure

```
struct point {
     int x, y;
 };
```

Finally, the dominance constrained assignments of the vertices of the input complete binary tree, is stored into an array indexed by the label of the vertices in complete binary tree. Each element of the array keeps the assigned grid point and the label of the associated vertex. The C-*struct* used for each element is

```
struct pos_array{
             int x, y, label;
  };
```

8.2 Implementation of Algorithm 2 for Edge mapping

The ***Algorithm 2*** (i.e the *Edge_mapper* module) for edge mapping keeps the congestion information for each grid edge, inorder to minimize the overall maximum congestion. Each grid point is a starting point for 2 edges, one is the vertical upward edge and the other is the horizontal edge in rightward direction. We keep the information for both edges associated with a single grid point, their common starting point. This is done through a C-*struct* for a grid edge

```
struct Grid_edge{
       point st_pt;           /*  the common starting grid point */
       int v_cong, h_cong;    / * congestion value for associated
                                  vertical and horizontal edges */
  };
```

A tree edge connects a node from level $l-1$ to a node in level l, therefore the path followed by its mapping on the grid would traverse through the region for level $l-1$ on the grid as well as one for level l. Hence, for mapping a tree edge originating in level $l-1$, we keep all grid edges in the region for level $l-1$ (the parent region) and also those in the region for level l (the child region). This is kept into a dynamic array of *Grid_edge* type: *P_gridedge* stores those from parent region and *C_gridedge* stores those from the child region.

```
struct Grid_edge ** P_gridedge, ** C_gridedge;
```

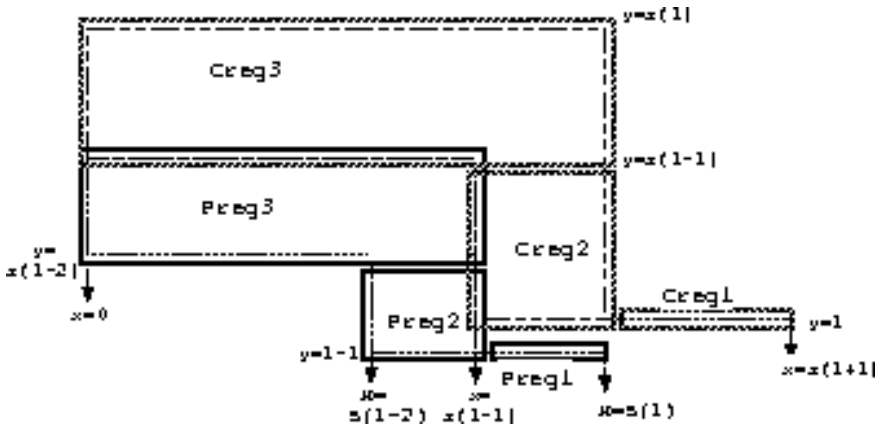

Figure 8.1: Ordering of grid points in parent and current level region

To map a tree edge $e = (u, v)$ starting from a grid point *src_pt* ($= Z(u)$, initially), we need to locate the element in the dynamic array that represents the grid edges associated with *src_pt* as its starting point. This may be required to be done from *P_gridedge* while the mapping is still in the parent region. Later on when the *src_pt* is in the child region, it is to be found from *C_gridedge*. To be able to do so, the grid edges must be stored in the following order: (refer Fig.8.1)

for **P_gridedge**:

the grid edges associated with grid points in **Preg1** given by $s(l-1) \leq x \leq s(l), y = l-1$ as *st_pt*, are kept first in increasing order of their x- values.

This is followed by those with grid points in **Preg2** given by $s(l-2) \leq x \leq s(l-1), l-1 \leq y < s(l-2)$ as *st_pt*, placed in increasing order of y- values followed by increasing order of x- values.

And then the grid edges with grid points from **Preg3** given by $0 \leq x \leq s(l-1), s(l-2) \leq y \leq s(l-1)$ as *st_pt*, placed in increasing order of y- values followed by increasing order of x- values.

Similarly for **C_gridedge** with $l-1$ replaced with l, and $l-2$ replaced with $l-1$, wherever they occur, to define **Creg1, Creg2** and **Creg3**.

The graph edges are stored using the C-*struct* given by

```
struct Graph_edge{
                point src, dest;
                int * dir_bit;
```

```
                int max_con_used, feq_max_con;
        };
```

For a tree edge $e = (u, v)$ directed from u to v, $src = Z(u)$ and $dest = Z(v)$, *pathcode(e)* mentioned in the edge mapping algorithm is implemented using int array *dir_bit*, it stores a binary string to depict the path used for mapping the tree edge (with 0 representing horizontal grid edge from the *src_pt*, 1 representing vertical edge from *src_pt*). The binary string has a number if bits equal to the dilation present for the (u, v) pair, so *dir_bit* is allocated this much memory . Dilation is calculated by

$$dilation = (dest(x) - src(x)) + (dest(y) - src(y))$$

As we wish to rip and re-route the tree edge, if it passes through a highly congested grid edge (i.e when *max_con_used = max_cong*), we store the maximum value of congestion over all grid edges through which a graph edge passes. In case, *max_con_used* value cannot be decreased, we wish to decrease the number of times the graph edge would pass through grid edges with congestion value equal to *max_cong*. So, we keep the frequency, *feq_max_con*, for a graph edge having passed through a highly congested grid edge.

8.3 Implementation of Algorithm 3 for DAGs

Directed Acyclic graphs do not have a predefined connectivity as a full complete binary tree. Rather DAGs with same number of nodes too can vary greatly in their structure. Some having large degree of parallelism have a smaller value for its maximum topological level. And some having denser structure (i.e very large number of edges) would have a scarce number of nodes in each of its topological level, making the value for its maximum topological level very close to the number of vertices in the DAG. Due to various possibilities for the structure of a DAG, we randomly generate a simple, connected, digraph using the library of datatypes in *LEDA*. The random digraph generated is then turned into an acyclic digraph using the fuction *Make_Acyclic(graph &)*, available in *LEDA*, and is also made single-sourced.

Input for the **Algorithm 3** is a topologically sorted DAG, therefore we perform topological sorting of the obtained DAG, and label the vertices with their topological number. We choose a modification of the data structure *ajacency list* for storing the topologically sorted and labeled DAG. It is stored as an array of *Graph_node* type, where each *Graph_node* type element is represented by the following C-*struct*ure,

```
struct Graph_node{
            int name, x, y;
            int outdeg, indeg;
            Graph_node ** child_v, **par_v;
  };
```

Each vertex v keeps its topological number as its *name*, the coordinates for its assignment $Z(v)$ under dominance constraint, in *x,y*. Initially, $Z(v) = (-1, -1)$, indicating v is not yet placed. It also keeps the out-degree and indegree values, along with an array of *Graph_node* pointers, *child_v*, that points to the Graph_nodes representing successor vertices of v. Similarly, it keeps an array of Graph_node pointers, *par_v*, that points to the Graph_nodes representing predecessor vertices of v.

The array representing DAG, is indexed by the *name* of the vertices, in their topological order. Initialization of the array is done reading the output of topological sorting on the DAG generated randomly by *LEDA*. The output is in a text file in the following format,

```
line 1  :     #vertices in DAG, N
line 2  :     #topological levels, lmax
line 3  :     #n1 #n2 ....  #nlmax
line 4  :     vertex 0 details   -  name    in-degree   out-degree
  ...   :     vertex 1 details   -  name    in-degree   out-degree
  ...   :           ....
line N+3:     vertex N-1 details -  name    in-degree   out-degree
line N+4:     edge details -      soucrce name      target name
  ...   :          ....
  ...   :          ....
  ...   :          ....
```

Line 3 gives the number of vertices in each topological level.

The arrays *Xarr* and *Yarr* defined for **Algorithm 3** in Chapter 7, are kept as an array of type *X1*, where each element of type *X1* has a Graph_node pointer to a vertex of the currently being assigned level and the minimum value of abscissa or ordinate (as the case may be, abscissa for *Yarr* and ordinate for *Xarr*) that its assignment should have.

```
struct X1{
              Graph_node * node;
              int min_xy;
  };
struct X1 * Xarr, * Yarr;
```

Forwarded grid points are maintained as a linked list of type *Grid_pt*, given by,

```
struct Grid_pt{
              int x,y;
              Grid_pt * next;
  };
```

To sort the forwarded grid points as is done by ***f_assign_current*** for trying out the assignments into them, we develop a variation of *heapsort* to work on the linked list. We may even do this by copying the linked list into an array of *point* type, do the sorting using heapsort and then copy the sorted order of points into a linked list representing the forwarded points again.

Chapter 9

Experimental Results

9.1 Square bound for complete binary tree

For a full complete binary tree of height l, which is also the number of levels in the tree (assumption root is at level 0), the number of vertices N is given by, $2^{l+1}-1$. The theoretical value, refer section 4.2, for the square bound $s(l)$ for a complete binary tree of height l lies close to $\lceil 2^{(l+1)/2} \rceil$ and is less than $\lceil 2^{(l+2)/2} \rceil$. And for the number of increments, $\Delta s(l)$, required over the square bound $s(l-1)$ achieved for a complete binary tree of height $l-1$ to get $s(l)$, it is $\lceil 2^{l/2} * (2^{1/2}-1) \rceil$.

Table 9.1 gives the comparison of the theoretical value of square bound s and the value of the square bound achieved by implementation of Algorithm 1 in C language.

Note:

a. The $\Delta s(l)$ value differs by one from the theoretical value $\lceil 2^{l/2} * (2^{1/2}-1) \rceil$, so $\Delta s(l) \approx 2^{l/2} * (2^{1/2}-1)$.

b. The square bound $s(l) \geq \lceil 2^{(l+1)/2} \rceil$ but is certainly $\ll \lceil 2^{(l+2)/2} \rceil$.

9.2 Execution time for Algorithm 1

The **Algorithm 1** discussed in Section 4.1 was implemented in C, using the data structures mentioned in Section 8.1, and the execution time on Intel Core 2 Duo machine running at 1.5 GHz with RAM size 1 GB for complete binary trees of the following height l is summarized in the Table 9.2.
The graphical comparison of the CPU time with $1000 * \frac{N^{3/2}}{(2^{21}-1)^{3/2}}$, is shown in Figure 9.1.

Level l	# vertices N	$\Delta s(l)$	$s(l)$	$\lceil 2^{l/2} * (2^{1/2} - 1) \rceil$	$\lceil 2^{(l+1)/2} \rceil$	$\lceil 2^{(l+2)/2} \rceil$
0	1	0	0	1	2	2
1	3	1	1	1	2	3
2	7	2	3	1	3	4
3	15	1	4	2	4	6
4	31	2	6	2	6	8
5	63	3	9	3	8	12
6	127	4	13	4	12	16
7	255	5	18	5	16	23
8	511	7	25	7	23	32
9	1023	10	35	10	32	46
10	2047	14	49	14	46	64
11	4095	19	68	19	64	91
12	8191	27	95	27	91	128
13	$2^{14} - 1$	38	133	38	128	182
14	$2^{15} - 1$	53	186	54	182	256
15	$2^{16} - 1$	76	262	75	256	363
16	$2^{17} - 1$	106	368	107	363	512
17	$2^{18} - 1$	151	519	150	512	725
18	$2^{19} - 1$	212	731	213	725	1024
19	$2^{20} - 1$	301	1032	300	1024	1449
20	$2^{21} - 1$	424	1456	425	1449	2048
21	$2^{22} - 1$	601	2057	600	2048	2897
22	$2^{23} - 1$	849	2906	849	2897	4096
23	$2^{24} - 1$	1200	4106	1200	4096	5793
24	$2^{25} - 1$	1697	5803	1697	5793	8192

Table 9.1: Comparison of Square bound

level l	# vertices N	CPU Time (sec)
0	1	0.002
1	3	0.002
2	7	0.002
3	15	0.002
4	31	0.002
5	63	0.002
6	127	0.007
7	255	0.003
8	511	0.004
9	1023	0.006
10	2047	0.014
11	4095	0.058
12	8191	0.1
13	$2^{14} - 1$	0.265
14	$2^{15} - 1$	0.591
15	$2^{16} - 1$	1.560
16	$2^{17} - 1$	4.662
17	$2^{18} - 1$	14.269
18	$2^{19} - 1$	57.302
19	$2^{20} - 1$	525.779
20	$2^{21} - 1$	773.902

Table 9.2: Execution Time on FC6 running at 1.5GHz

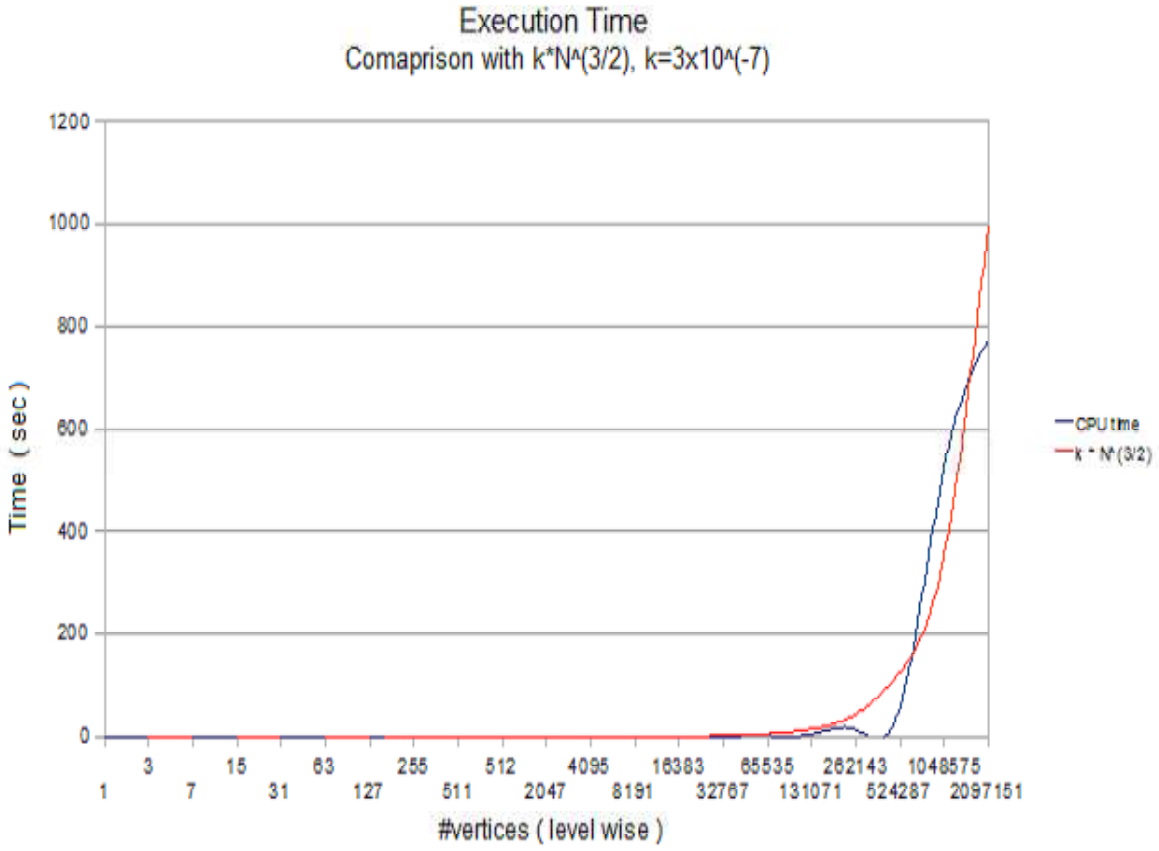

Figure 9.1: Execution Time compared with $N^{3/2}$

9.3 Maximum Congestion obtained by edge-mapping

As mentioned in the implementation section 8.1, we could have 8 different placements for the complete binary tree, all fulfilling the dominance constraints. These are created by changing the sorting order within the subgroups of *Xarr, Yarr* and *Yarr_x* (which store the nodes from B', A' and A' respectively). We associate type numbers, given in Table 9.3, with each combination of sorting order.

Type	Yarr	Xarr	Yarr_x
I	decreasing order of x- coord.	decreasing order of y- coord.	decreasing order of y- coord.
II	increasing order of x- coord.	decreasing order of y- coord.	decreasing order of y- coord.
III	decreasing order of x- coord.	increasing order of y- coord.	decreasing order of y- coord.
IV	decreasing order of x- coord.	decreasing order of y- coord.	increasing order of y- coord.
V	decreasing order of x- coord.	increasing order of y- coord.	increasing order of y- coord.
VI	increasing order of x- coord.	increasing order of y- coord.	decreasing order of y- coord.
VII	increasing order of x- coord.	decreasing order of y- coord.	increasing order of y- coord.
VIII	increasing order of x- coord.	increasing order of y- coord.	increasing order of y- coord.

Table 9.3: Possible combinations of sorting order in *Xarr, Yarr, Yarr_x*

The maximum value of the congestion obtained in these cases differ by a small amount, all satisfying the lower bound on the maximum congestion calculated theoretically $\approx \Delta s(l)$, refer Section 5.1. Refer Table 9.4 and Figures 9.2, 9.3.

level l	$N^{1/2}$	Theoretical $min^m \approx \Delta s(l)$	Type I	Type II	Type III	Type IV	Type V	TypeVI	Type VII	Type VIII
1	1.732	1	1	1	1	1	1	1	1	1
2	2.646	2	2	2	2	2	2	2	2	2
3	3.873	1	2	2	2	2	2	2	2	2
4	5.568	2	3	3	3	3	3	3	3	3
5	7.937	3	4	4	4	4	4	4	4	4
6	11.269	4	6	6	6	6	6	6	6	6
7	15.269	5	8	8	8	8	8	8	8	8
8	22.605	7	12	12	11	12	11	10	11	10
9	31.984	10	18	16	15	17	15	15	16	15
10	45.244	14	26	21	27	27	28	27	22	28
11	63.992	19	37	31	36	39	38	36	31	36
12	90.504	27	53	48	54	55	54	50	48	54
13	127.996	38	79	69	71	78	68	72	69	72
14	181.017	53	114	102	107	111	107	105	102	105
15	255.998	76	162	135	159	168		160	135	160

Table 9.4: Maximum Congestion values for edge mapping over the different placements

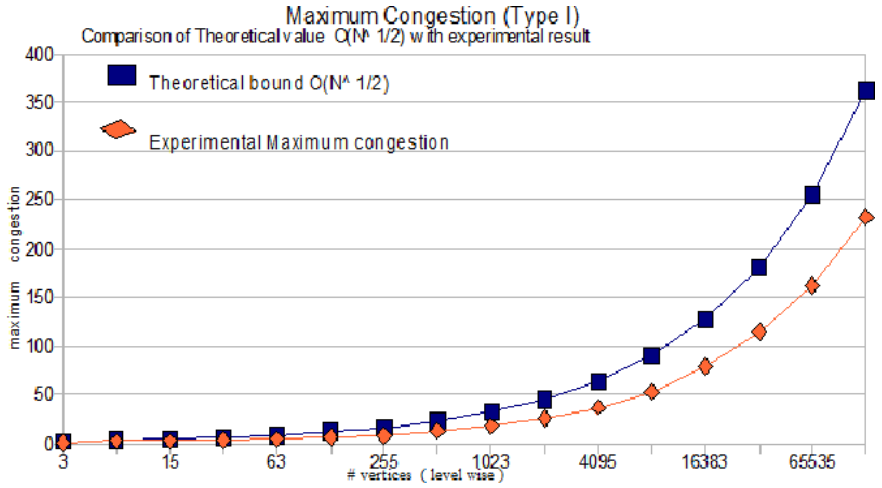

Figure 9.2: Comparison of maximum congestion (for *Type I*) values with $N^{1/2}$

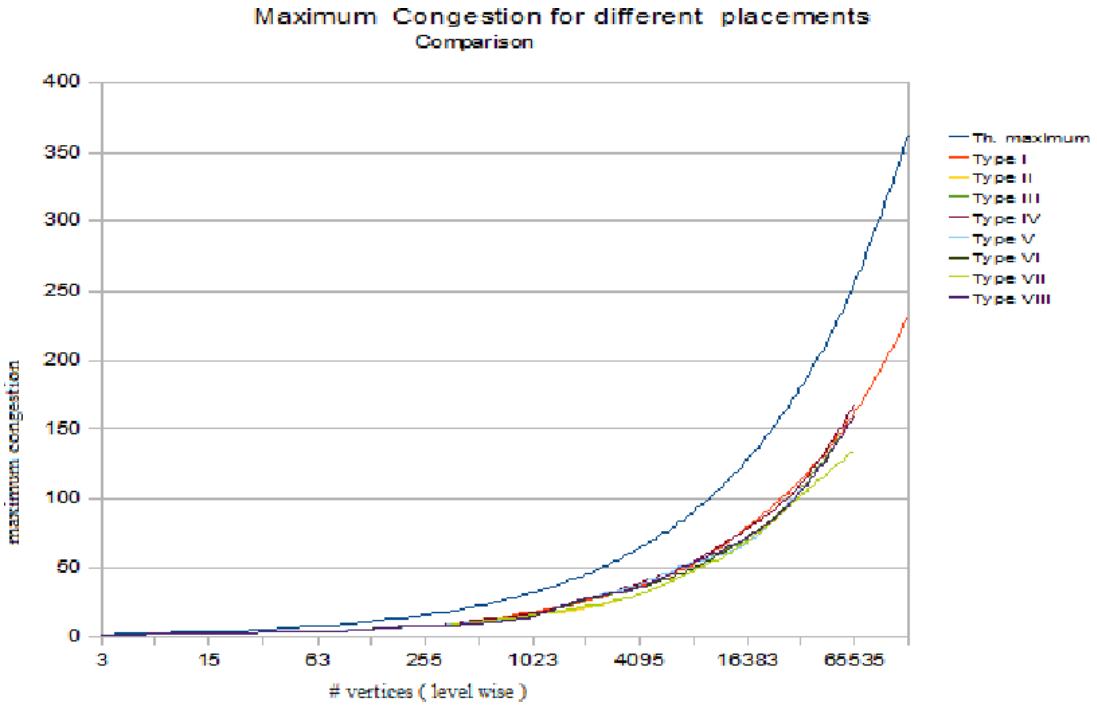

Figure 9.3: Maximum Congestion for all *Types* of Placements compared with $N^{1/2}$

9.4 Maximum dilation for the placements

Maximum dilation values differ from one placement order to another. The dilation analysis for the placement, refer Section 4.5 equation 4.22, gives an upper bound on maximum dilation. In Table 9.5 this upper bound is tabulated under column heading **Th.** max^m, along with experimentally attained values. In Figure 9.4 the graphical comparison is shown.

level l	$N^{1/2}$	Th. max^m	Type I	Type II	Type III	Type IV	Type V	Type VI	Type VII	Type VIII
1	1.732	2.293	2	2	2	2	2	2	2	2
2	2.646	4.586	3	3	3	3	3	3	3	3
3	3.873	3.293	2	2	2	2	2	2	2	2
4	5.568	5.586	6	6	6	6	6	6	6	6
5	7.937	8.879	8	8	8	8	7	7	8	6
6	11.269	13.172	13	11	13	11	11	11	11	10
7	15.969	18.465	16	16	16	16	16	16	13	13
8	22.605	27.051	27	24	27	24	22	24	24	20
9	31.984	39.93	40	35	40	35	35	35	33	30
10	45.244	58.102	55	52	55	52	52	52	46	46
11	63.992	82.567	81	74	81	81	81	74	68	68
12	90.504	118.911	115	106	115	112	106	106	106	106
13	127.996	170.134	173	156	173	168	168	156	150	150
14	181.017	241.529	247	225	247	236	236	225	218	218
15	255.998	346.268	349	322	349	341	341	322	308	308

Table 9.5: Maximum Dilation values for different *Types* of Placement

Note:

a. All placement orders, *Type I to VIII* satisfy the lower bound on maximum congestion. Also, they achieve maximum congestion of order $O(N^{1/2})$ (eg. $0.6 * N^{1/2}$ for *Type I*), which is an optimal in presence of essential congestion. So, ***Edge_mapper*** module optimally minimizes the maximum congestion for area-efficient dominance drawing of complete binary tree.

b. Following *Types* of placement achieves same maximum dilation values:

i. *Type I* and *Type III*

ii. *Type II* and *Type VI*

iii. *Type IV* and *Type V*

iv. *Type VII* and *Type VIII*

c. *Type II* and *Type VII* attains least value for maximum congestion of all *Types*. Of which *Type VII* also achieves the least maximum dilation value. So, *Type VII* is the best placement order.

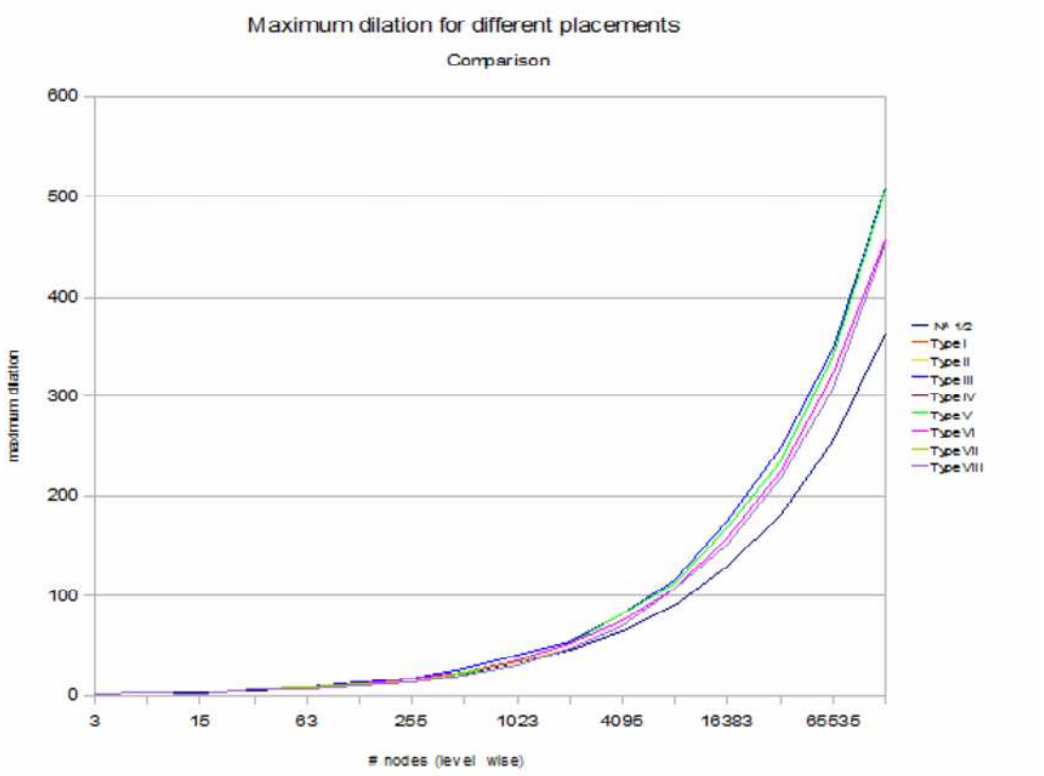

Figure 9.4: Comparison of maximum dilation for different placements with $N^{1/2}$

9.5 Comparison of Algorithm 1 with its Improvement

Here we compare the square bounds, shown in table 9.6, achieved by the **Algorithm 1** with that achieved by implementation f the **Improvement** mentioned in section 6. We also find *utilization ratio*, the ratio between the number of nodes in the tree and the number of grid points falling within the square area defined by the square bound. Figure 9.5 gives the comparison between the *utilization ratio* for *Algorithm 1* with that of its offline *improvement*.

Note:

a. The *utilization ratio* increases with the number of vertices N in the complete binary tree. This indicates an optimal usage of the area on grid for the placement of complete binary tree under domimnance constraints.

b. *Algorithm 1*, though does not intend to achieve a global minimum square bound for an N- node complete binary tree, it does not incur a heavy loss in the utilization ratio. In fact, the global minimum square bound achieved by its offline improvement

level l	# vertices N	$s(l)$ <Improvement>	$s(l)$ <Algo. 1>	Util. ratio <Impr.>	Util. ratio <Algo. 1>
0	1	0	0	1	1
1	3	1	1	0.75	0.75
2	7	3	3	0.438	0.438
3	15	4	4	0.6	0.6
4	31	6	6	0.633	0.633
5	63	9	9	0.63	0.63
6	127	12	13	0.751	0.648
7	255	17	18	0.787	0.706
8	511	24	25	0.818	0.756
9	1023	34	35	0.835	0.789
10	2047	47	49	0.888	0.819
11	4095	66	68	0.912	0.86
12	8191	93	95	0.927	0.889
13	$2^{14}-1$	131	133	0.94	0.912
14	$2^{15}-1$	184	186	0.957	0.937
15	$2^{16}-1$	259	262	0.969	0.947
16	$2^{17}-1$	365	368	0.978	0.963
17	$2^{18}-1$	516	519	0.981	0.969
18	$2^{19}-1$	728	731	0.987	0.978
19	$2^{20}-1$	1028	1032	0.99	0.983
20	$2^{21}-1$	1452	1456	0.993	0.988
21	$2^{22}-1$	2053	2057	0.994	0.99
22	$2^{23}-1$	2901	2906	0.996	0.993
23	$2^{24}-1$	4101	4106	0.997	0.995
24	$2^{25}-1$	5798	5803	0.998	0.996

Table 9.6: Comparison of *Algorithm 1* with its *Improvement*

is approximately equal to one achieved by itself.

9.6 Results from Extension to DAGs

The **Algorithm 3** mentioned in the chapter 7, was also implemented in C-language, and tested on a number of randomly generated single source connected directed acyclic graphs. The structure for DAGs can vary even for same number of vertices, we give the obtained square bound for a number of DAGs in Table 9.7.

Note:

a. The number of topological levels l_{max} present in the DAG (obtained by topological sorting) determines the lower bound on the square bound. This follows straight from the *dominance constraint* on the y- coordinates of the successor vertices, once its predecessor is assigned.

The dominance drawings for the randomly generated DAGs, as shown in the Table 9.7, obtains a square bound nearly equal to l_{max}. This may not be true in general.

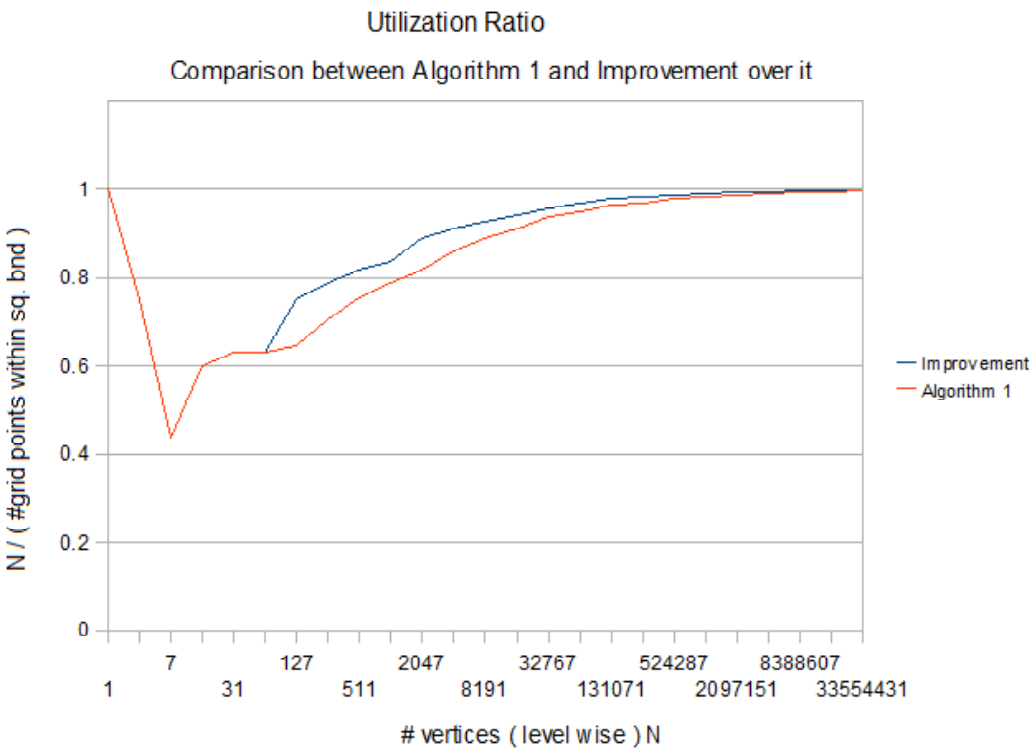

Figure 9.5: Utilization ratio for *Algorithm 1* and its *improvement*

b. The region on the grid that consists the assigned vertices of a DAG, is not evenly used. More often, the utilization of the grid points along the right diagonal is greater. Also, the lower rows and lower columns (defined by low value for k in $x = k$ or $y = k$) become disqualified for placement of the vertices in forthcoming levels of the DAG, very soon. So, the area used for the dominance drawing of DAGs on a 2-dimensional grid is in general under utilized. Perhaps, the congestion during edge mapping would be lower.

# vertices N	# edges	# topological levels	Square bound s
25	79	22	21
25	35	10	10
25	36	8	8
30	61	30	23
35	56	15	16
35	67	16	20
35	595	35	34
35	49	17	16
35	61	24	23
40	77	21	20
40	84	17	16
40	79	27	26
50	470	48	47
50	477	47	46
50	76	17	16
51	80	18	19
51	78	17	17
51	85	11	12
51	95	20	20
101	164	28	30
101	185	35	35
101	188	32	33
101	211	41	44
201	364	60	62

Table 9.7: Square bound for randomly generated DAGs with given N

Chapter 10

Comaprison with Previous Works

Before coparing our work with the previousy conducted ones, it is importatnt to note that the dominance constraints as defined by us, is more restrictive in both x- and y-coordinates. Though a strictly upward drawing has a similar constraint on y- coordinate, there is no constraint put on x- coordinate. As we do not preserve the left to right order of siblings during the placement, and also allow non-planarity into our drawings, we have obtained a lower area $O(N)$ than the lower bound of $\Omega(NlogN)$ specified in [5].

The paper [2] also achieves an $O(N)$ area for planar upward tree drawings. Their drawing is a polyline grid drawing. For an orthogonal grid drawing they have obtained $O(NloglogN)$ lower bound on area in the same paper. Our drawing has been an orthogonal grid drawing and we have obtained a lower area for a more restricted problem. But, our solution is non-planar and we have not included the calculation for bends into the area, i.e bends have not been assigned separate grid points.

Having permitted a non-planar drawing for our purpose, we successfully construct dominance drawings for all acyclic digraphs, while [4] proves inexistence of planar upward drawings for some single source DAGs that do not satify *Thomassen*'s criteria.

The utilization ratio (reciprocal of expansion ratio), for our drawings show that we have obtained better area than achieved in [3]. The area efficiency in our case, is on the cost of congestion and crossings. for the same reason we have a lower area than the $\Omega(NlogN)$ lower bound for non-planar graph layouts of planar graphs in VLSI [9].

Chapter 11

Conclusion

As mentioned earlier, dominance constrained placement of the modules on the VLSI chips, enables the clock wires to be routed in parallel to the signal wires. As said before, we had modified the usual definition of dominance drawing to become more restrictive in having the y- coordinates of the successors to be strictly greater than that of its predecessors. The purpose behind this was to ensure that the clock arrival times for predecessors always precede the clock arrival times for the successors. This implementation of the data dependence into the placements would imply 'no time skew' between the dependent modules.

During the period for dissertation work, an analysis of the area requirement for placement of the modules, assuming a complete binary tree represents the data dependence among the modules, has been conducted. Along with this, the dilation and congestion analysis for the edge mapping has also been presented. Our main objective throughout had been to minimize the area, or the square bound for the dominance constrained placement. To achieve minimum area under these constraints on placement, we developed a new polynmial time algorithm.

The proposed algorithm achieves an area-efficient, $O(N)$, dominance constrained drawing for an N- node complete binary tree, in $O(N^{3/2})$ time. The square bound s, maximum dilation and the maximum congestion attained by or present in the resulting drawing has been of order $O(N^{1/2})$. To minimize the area of the dominance constrained placement, we have concentrated mainly on the area minimization during placement. But in VLSI chips, congestion and bends also contribute to increase the VLSI chip area. Higher congestion would increase the channel width during the VLSI routing phase, and bends would increase the number of vias, thereby affecting the overall area.

As a secondary objective to minimize congestion, we have proposed an edge mapping

algorithm that minimizes the maximum congestion as much possible for the obtained minimal area dominance constrained placement. Later on, we would want to study the problem of dominance constrained drawing with bounded congestion, to calculate its effects on the placement area, i.e on the square bound. Having such additional constraint, we would have to solve the problem of minimal area placement and bounded congestion edge mapping (or routing) in tandem. We would also make an analysis of the area requirement of our drawing with bends analysis.

As the VLSI modules are more generally covered by directed acyclic graphs, we have exteneded our proposed algorithm for minimal area dominance constrained drawing to DAGs. The trivial lower bound on the square bound for the dominance constrained drawing of DAGs was attained for some of the random DAGs on which we tested our algorithm. Yet, it is difficult to conclude definitely on the area requirement for DAGs at this moment. This requires more thorough study. We would like to collect a pool of DAGs, with different kinds of connectivity, and test our algorithm on both dense and sparsely connected ones. Perhaps, with some more time to spend, we would be in a position to say more about the area requirement for dominance constrained placement for DAGs.

Bibliography

[1] Ashim Garg and Adrian Rusu,"Area-efficient planar straight-line drawings of outerplanar graphs", Discrete Applied Mathematics, Vol. 155, No. 9, pp. 1116—1140, 2007.

[2] A. Garg, M. T. Goodrich and R. Tamassia, "Planar Upward Tree Drawings With Optimal Area," International Journal of Computational Geometry & Applications, 1995.

[3] C. O. Shields, Jr. I. H. Sudborough,"Area Efficient Layouts of Binary Trees on One, Two Layers," Parallel and Distributed Computing and Systems, 2001.

[4] M. D. Hutton and A. Lubiw, "Upward Planar Drawing of Single Source Acyclic Digraphs," In Proc. 2^{nd} ACM–SIAM Symposium Discrete Algorithms, pages 203—211, 1991.

[5] P. Crescenzi, G. Di Battista and A. Piperno, "A note on Optimal area algorithms for upward drawings of binary trees," Computational Geometry Theory Applications, 2: 187—200, 1992.

[6] C. Thomassen, "Planar acyclic oriented graphs," Order 5, pp. 349—361, 1989.

[7] A. L. Fisher and H. T. Kung, "Synchronizing large systolic arrays," Proceedings of SPIE, pages 44—52, May 1982.

[8] S. Dhar, M. A. Franklin and D.F. Wong, "Reduction of clock delays in vlsi structures," Proceedings of IEEE International Conference on Computer Design, pages 778—783, October 1994.

[9] F. T. Leighton "New Lower bound techniquea for VLSI," Mathematical Systems Theory, 17: 47—70, 1984.

[10] "Handbook of Computational Geometry," J. -R. Sack and J. Urrutia, Elsevier 2000.

[11] "Algorithms for VLSI Physical Design Automation," 3^{rd} Edition, Naveed Sherwani.

[12] "Introduction to Algorithms," 2^{nd} Edition, T. H Coreman, C. E. Leiserson, R. L. Rivest and C. Stein,Prentice Hall of India, 2004.