Analysis of Data Structures for VLSI Layout Tools

Sandeep Kumar Dey
Reg. No. - CS0708

Supervisor: Professor Susmita Sur Kolay

July 16, 2009

Analysis of Data Structures for VLSI Layout Tools

Report Submitted

by

Sandeep Kumar Dey

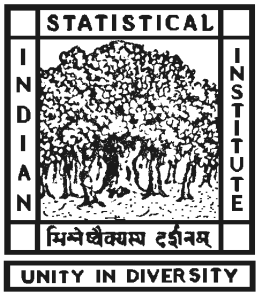
(Roll No: MTC0708)

As a Partial Fulfillment of Master of Technology (2007-2009)

in Computer Science

Under the Guidance of

Professor Susmita Sur Kolay

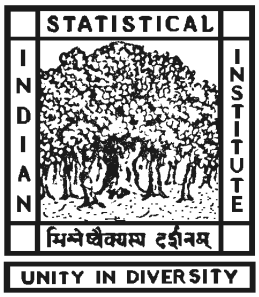


Indian Statistical Institute, Kolkata

203 B. T. Road,

Kolkata - 700108

July 2009



Indian Statistical Institute

Kolkata-700 108

CERTIFICATE

This is to certify that the thesis entitled "**Analysis of Data Structures for VLSI Layout Tools**" is submitted in the partial fulfilment of the degree of M. Tech. in Computer Science at Indian Statistical Institute, Kolkata. It is fully adequate, in scope and quality as a dissertation for the required degree.

The thesis is a faithfully record of bonafide research work carried out by Sandeep Kumar Dey under my supersion and guidance. It is further certified that no part this thesis has been submitted to any other university or institute for the award of any degree or diploma.

Professor Susmita Sur Kolay
(Supervisor)

Countersigned
(External Examiner)
Date:

Acknowledgement

I feel privileged and take this opportunity to express my sincere gratitude to the supervisor of this study, Prof. Susmita Sur Kolay. Her command over the area of my work has been of great help for my analysis. She not only suggested directions towards the solution of the problem but also helped me in all aspects including the preparation of this manuscript and have also given me full freedom to think and work independently. This work has been possible only because of her continuous suggestions, inspiration, motivation and full freedom given to me to incorporate my ideas. I also take this opportunity to thank all my teachers who have taught me in my M. Tech. course. I am specially thankful to my teachers Dr. Arijit Bishnu, Prof. Subhas Nandy and Dr. Sandip Das for teaching me graph algorithms and geometric algorithms which were very essential part of my dissertation. I am also thankful to Subhasis-da, Debasis, Pritha-di, Arindam-da, Subhabrata, Sanjay, Santanu, Somindu, Aritra, Nargis, Kali, Mrinmoy, Pulak, Swarup and Chiranjit for motivating me in my downtimes. I am specially thankful to Somindu, Aritra and Mrinmoy for providing me resources like food, coffee whenever I felt hungry and laptop for unlimited time when I needed it most. This helped me carry on my work uninterruptedly and conveniently. My special thanks to my friends Subhasis-da, Subhabrata, Sanjay and Santanu for their fruitful discussions on different algorithms and codes throughout the thought process. And last but never the least, I thank my parents, sister Papri, and my wife Ruchira for their endless support.

Place : Kolkata
Date : Sandeep Kumar Dey

Abstract

An efficient layout data-structure is of great importance to design rule checking algorithms, compaction algorithms, gridless area routing algorithms and many more layout tools. The atomic operations for most of the layout tools are point searching, area searching, neighbour searching, node searching and shadow searching. Nowadays, the trapezoidal corner stitching structure, whose point searching is $O(N^{1/2})$, is the most popular data structure used for the layout tools. The point searching is $O(logN)$ for the other popular tree based data-structures such as KD trees and Quad trees, which is natural due to their construction. But tree based data structures does not provide a fascinating solution to other atomic operations such as area searching and neighbour searching. And the trapezoidal corner stitching structure wins over tree based structures in the area searching and neighbour searching operations.
In this thesis, we propose a new data-structure for the atomic operations of layout tools, which combines the trapezoidal corner stitching structure with a KD tree based structure. We call this structure TBT corner stitching (Tree based Trapezoidal corner stitching). The point searching is $O(logN)$, area and neighbour searching is $O(n)$, where N is total number of tiles in the layout and n is the number of tiles intersecting the querry area or the number of neighbouring tiles of the query tiles. We will also compare the complexity of point searching, area searching and neighbour searching algorithms of tree based as well as corner stitch based layout data-structures with our data-structure.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

VLSI designers have a number of layout tools at their disposal now a days. They have a graphical editor to enter and modify their layouts, a design rule checker to verify the geometrical correctness of their layouts, an extractor to permit simulating and verifying the timing of their layouts, and a postprocessor to create the required fabrication masks from symbolic layouts. Some layout tools also have a compactor to quickly generate design rule correct layouts, and placement and routing tools to generate floorplans and interconnecting wires. Traditionally, each of these layout tools operates in its own environment, employing dedicated layout data-structures and often a dedicated user interface. If these tools are integrated together, it is usually on the surface, namely the user sees what appears to be a common interface. Although to the user such a tool system provides the functionality and integration he needs, to the CAD developer it is difficult to maintain and to build upon for the future.

What is needed is an integrated layout system which provides complete functionality, is tightly integrated, is fast and efficient, and is easy to build upon. This requires the entire system to rely on a common layout database. Moreover, the database should be based on a data structure which supports efficient and *fast access* and *search operations* for all the tools in the system.

This thesis presents such a data structure namely, tree based trapezoidal corner stitching(TBT corner stitching) suitable for representing 45^o layout and performs efficiently the point, area and the neighbour search operations. We could have used tree based data structures such as KD trees, Quad trees but they lack ability to do efficient area and neighbour searching when compared to corner stitching structures. If we consider only corner stitching structures, they lack ability to do efficient point searching when compared to tree based data structures.

The desired layout data structure should lend itself to optimal access and update algorithms as needed by a layout editor, design rule checker, compactor, and extractor. A layout editor requires access operations such as selecting layout objects at a point (point search), selecting all layout objects in a rectangular area (area search), and selecting all layout objects electrically connected to an object (node search). Furthermore, an editor requires update operations such as inserting and deleting

objects in the database. Since a layout editor is an interactive tool, all these operations must be fast. Frequently used "batch" tools, such as design rule checking, compaction, and extraction, must also have fast database operations. A design rule checker requires finding the context of layout objects (neighbor search) and searching the possibility of design rule violations near objects (area search). A compactor also considers design rules and the context of layout objects (neighbour search). It also must find the closest interacting (in the sense of design rules) objects to each object (unbounded area search or shadow search). An extractor requires identifying circuit nodes in a layout (node search) and calculating circuit perimeter (neighbor search) and area (area search) parasitics. Note that the most used operations for all tools are neighbor and area searching. Therefore, these search operations were given the highest priority when choosing the best data structure. A typical collection of layout tools is shown in Figure 1.1.

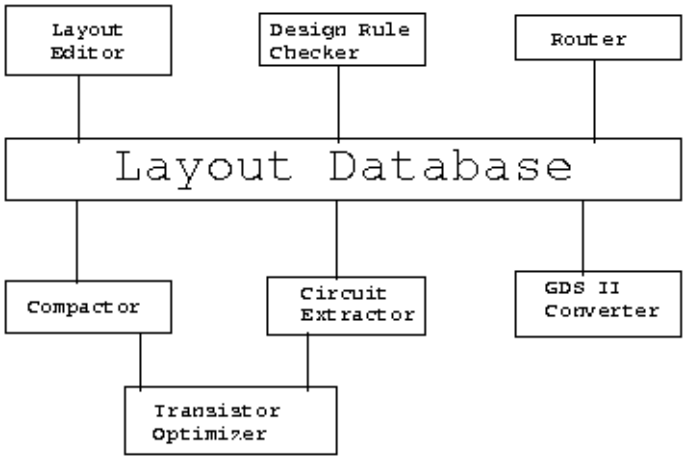


Figure 1.1: *A*ll the layout tools shown directly access a common layout database

The rest of the thesis is organized as follows. In section 2, we provide details of corner stitching datastructures ([Ous84]), ([JKOT84]), ([BM97]), ([DMH90]), ([SF93]), ([She93]) and the algorihms for point searching, area searching and neighbour searching with their time complexities. In section 3, we provide details of tree based data structures ([Ked82]), ([HF90]), ([GGLW96]), ([NR86]), ([MB99]) and the algorihms for point searching, area searching and neighbour searching with their time complexities. In section 4 we will discuss Partial bin based corner stitching ([ZYX00]) and the algorihms for point searching, area searching and neighbour searching with their time complexities. In section 5, we will discuss our TBT corner stitching datastructure and the algorihms for point searching, area searching and neighbour searching with their time complexities. In section 6, we will compare the time complexities of the different data structures as mentioned in the previous sections. In section 7, we conclude the thesis and focus on its future capabilities.

Chapter 2

Corner Stitching

In this chapter we will discuss the corner stitching data structures for layouts having rectilinear tiles as well as layouts having trapezoidal tiles. We will discuss the algorithms for point searching, area searching and neighbour searching.

2.1 The Rectilinear Corner Stitch

The Rectilinear Corner Stitching is a data-structure for representing rectangular regions in a plane. It was originally developed by Ousterhout ([JKOT84]), ([Ous84]) as a data-structure for CAD of VLSI layouts. Corner Stitching represents regions by simply associating values with areas, and linking these areas only to their neighborhood. This allows many algorithms to operate on a region in time typically linear with respect to the area of the region.
In CAD of VLSI layouts the most important interactions occur between mask features lying in close proximity. One of the most computationally demanding aspects of layout CAD is the verification of mask geometries; minimum widths and separations dominate design rule checking, and abutment and overlap of geometries dominate the extraction of circuit descriptions. Similarly in constructive operations, such as compaction and channel finding, examination of neighborhoods plays a key role. Corner Stitching, by linking regions to their neighborhood, is able to provide cost-effective operations suited to the needs of VLSI CAD.
Corner stitching is simple, provides a variety of efficient searching operations, and allows the database to be modified quickly. There are three important properties of a corner-stitched plane, illustrated in Figures 2.1, 2.2, and 2.3:
Coverage: Each point in the x-y plane is contained in exactly one tile (Figure 2.1). Empty space is represented as well as the area covered with material.
Strips: Material of the same type is represented with horizontal strips (Figure 2.2). Areas of the same type of material are represented with horizontal strips that are as wide as possible, then as tall as possible. In each of the figures the tile structure on the left is illegal and is converted into the tile structure on the right. The strip structure provides a canonical form for the database and prevents it from fracturing

into a large number of small tiles.
Stitches: The records describing the tile structure are linked together in the database using four links per tile, called stitches. The links point to neighboring tiles at two of the tile's four corners (Figure 2.3).

- In the upper right corner, one stitch points to the top most neighboring tile on the right (tr stitch) and one to the right most tile whose bottom edge contains the upper right corner (rt stitch).
- In the lower left corner, one stitch points to the bottom most neighboring tile on the left (bl stitch) and one to the left most tile whose top edge contains the lower left corner (lb stitch).

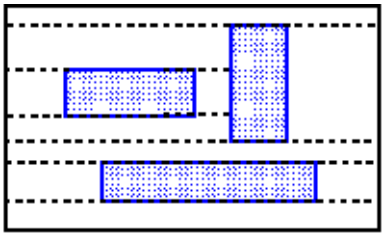


Figure 2.1: *Every point in a corner-stitched plane is contained in exactly one tile. In this case there are three solid tiles, and the rest of the plane is covered by space tiles (dotted lines). The space tiles on the sides extend to infinity. In general, a plane may contain many different types of tiles.*

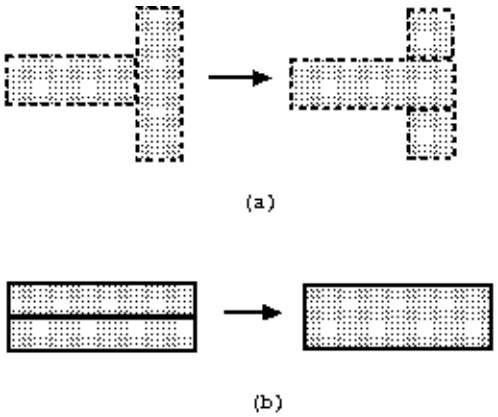


Figure 2.2: *In (a) it is illegal for two tiles of the same type to share a vertical edge. In (b) the two tiles must be merged together since they have exactly the same horizontal span.*

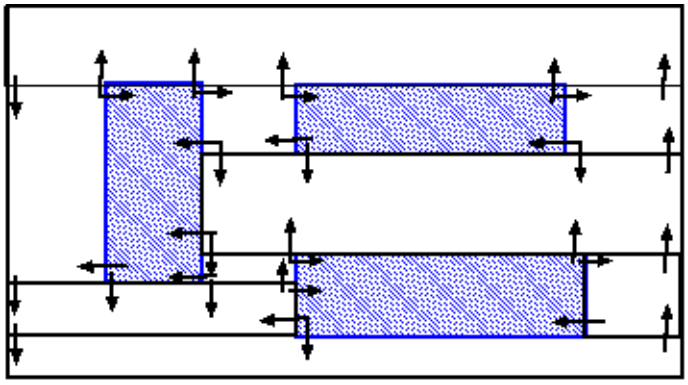


Figure 2.3: *The record describing each tile contains four pointers to other tile records. The pointers are called corner stitches, since they point to neighboring tiles at the lower-left and upper-right corners. The corner stitches provide a form of two-dimensional sorting. They permit a variety of geometrical operations to be performed efficiently, such as searching an area or finding all the neighboring tiles on one side of a given tile.*

2.1.1 Point Searching Algorithm

Point searching is used to locate the tile containing a given point P. The algorithm iterates in y and then x, starting from any given tile in the plane as follows:

1. Move upwards using rt stitches or downwards using lb stitches until a tile is found whose vertical range contains y_p. The vertical range of a tile includes the tile's bottom up to but not including the tile's top.

2. If the tile contains the point P , then stop. Otherwise move left using the bl stitch if, along the line $y = y_p$ P is to the left of the tile, or move right using the tr stitch if P is to the right of the tile.

3. Go to step 1

Fig. 2.4 shows an example of a point search. If the tiles in a tile plane are of relatively uniform size, the number of tiles intersecting a horizontal or vertical line is approximately $\sqrt{N}$. Therefore, the complexity of the point search algorithm is O($\sqrt{N}$) where N is the total number of tiles in the plane. The worst-case complexity is N when all tiles line up in a row or column. In practice, the point search performs better than $\sqrt{N}$ (or N) if one can start from a tile close to the point.

2.1.2 Area Searching Algorithm

Area searching is used to find all tiles overlapping (with $surfacearea > 0$) a given area, where the area is a rectangle. The algorithm is recursive and resembles a depth first search algorithm. The area search algorithm is pivotal upon tile left edges. Tiles overlapping the area, but whose left edge does not extend into the area are called

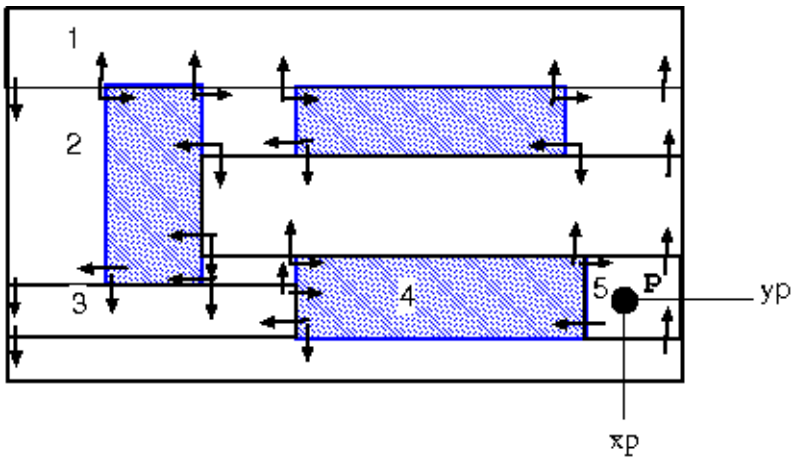


Figure 2.4: *Example point search showing the tile enumeration order and traversal of tile stitches.*

seed tiles. Seed tiles always lie along the left hand side of the area and are used to begin the recursive area search. Tiles overlapping the area whose left edge extends into or overlaps the area are reported during the search from a seed tile. In general, files are visited from left to right and from top to bottom (see Figure 2.5). When a tile is reported, all neighbour tiles along its right edge and overlapping the area are visited. Each visited tile is reported recursively if the portion of its left edge lying within the area is above the reported tile's bottom. The algorithm operates as follows:

1. For each seed tile, ordered from top to bottom along the left edge(s) of the area, do steps 2 and 3.

2. Report the tile and find all right neighbor tiles via an edge walk along the tile's right edge that a) overlap the area and b) do not extend below the reported tile within the area.

3. For each tile found in step 2, perform step 2 and 3 recursively.

Figure 2.5 shows an example of a area search. The complexity of enumerating tiles in an area is $O(n)$, where n is the number of tiles overlapping the area. Before an area search can begin, the first seed tile must be found using a point search and seed tile search. Therefore, the total complexity of a general area search is $O(\sqrt{N} + n)$.

2.1.3 Neighbour Searching Algorithm

Neighbour searching is used to find all tiles having overlapping edges with the query tile. The neighbour search algorithm is pivotal upon the four pointers(rt, tr, lb, bl) of the query tile. So there are four *seed tiles*. The algorithm operates as follows:

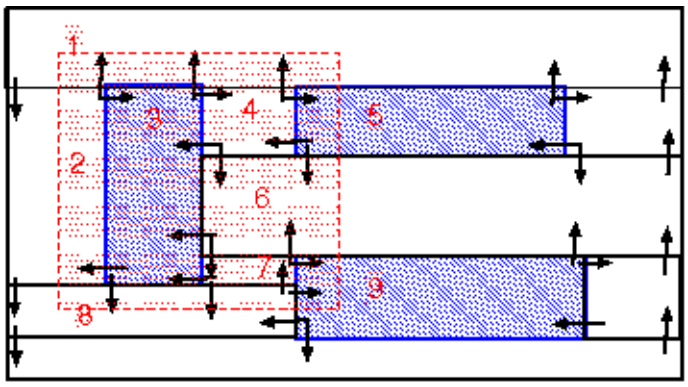


Figure 2.5: *Example area search showing the tile enumeration order and traversal of tile stitches.*

1. For seed tile rt, follow the bl pointer and report the tile if the bottom edge of the tile is ovelapping with the top edge of the query tile. Otherwise stop.
2. For seed tile tr, follow the lb pointer and report the tile if the left edge of the tile is ovelapping with the right edge of the query tile. Otherwise stop.
3. For seed tile lb, follow the tr pointer and report the tile if the top edge of the tile is ovelapping with the bottom edge of the query tile. Otherwise stop.
4. For seed tile bl, follow the rt pointer and report the tile if the right edge of the tile is ovelapping with the left edge of the query tile. Otherwise stop.

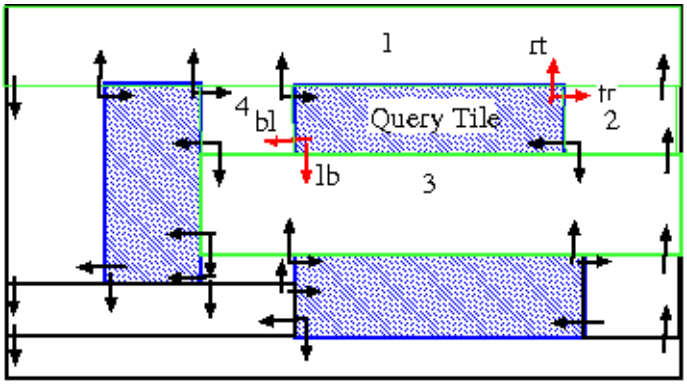


Figure 2.6: *Example neighbour search showing the tile enumeration order and traversal of tile stitches.*

Figure 2.6 shows an example of a neighbour search. The complexity of enumerating tiles is $O(n)$, where n is the number of neighbour tiles. Before an neighbour search can begin, the first seed tile must be found using a point search and seed tile search. Therefore, the total complexity of a general area search is $O(\sqrt{N} + n)$.

2.2 The Trapezoidal Corner Stitch

The primary deficiency of rectangular comer stitching is that it only works with rectangles. To remedy this, trapezoidal corner stitching ([DMH90]) was introduced so that corner stitching could be efficiently applied for $non-90^o$ layout. All tiles in Tailor are trapezoids whose top and bottom edges are parallel with the x axis and whose left and right edges have an angle of $45^o, 90^o, or 135^o$ with the x axis. One of nine different shapes of trapezoids are possible as shown in Figure 2.7. Each tile is of a particular material type. Tile planes in trapezoidal corner stitching maintain similar properties to rectangular corner stitching.

Coverage: Each point in the x-y plane is contained in exactly one tile. Empty space is represented as well as the area covered with material.
Strips: Material of the same type is represented with horizontal strips. Areas of the same type of material are represented with horizontal strips that are as wide as possible, then as tall as possible. In each of the figures the tile structure on the left is illegal and is converted into the tile structure on the right. The strip structure provdes a canonical form for the database and prevents it from fracturing into a large number of small tiles.
Stitches: The records describing the tile structure are linked together in the database using four links per tile, called stitches. The links point to neighboring tiles at two of the tile's four corners (Figure 2.8).

- In the upper right corner, one stitch points to the top most neighboring tile on the right (tr stitch) and one to the right most tile whose bottom edge contains the upper right corner (rt stitch).

- In the lower left corner, one stitch points to the bottom most neighboring tile on the left (bl stitch) and one to the left most tile whose top edge contains the lower left corner (lb stitch).

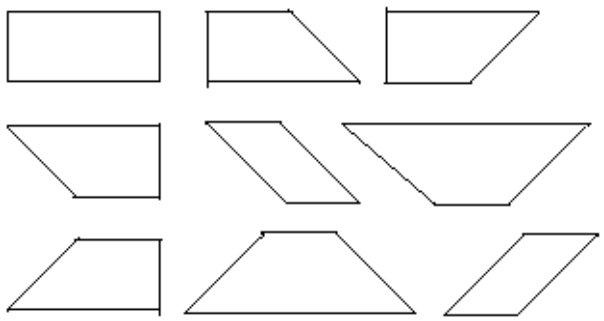


Figure 2.7: *Nine trapezoid shapes are possible for tiles representing* 45^o *layout.*

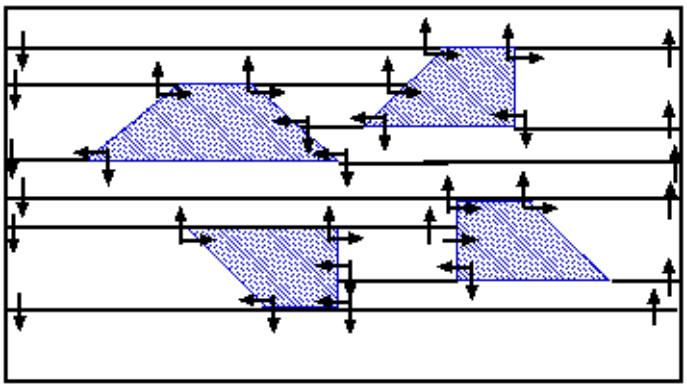


Figure 2.8: *Example of trapezoidal corner stitching in a layout plane.*

2.2.1 Point Searching Algorithm

Point searching is used to locate the tile containing a given point P. The algorithm iterates in y and then x, starting from any given tile in the plane as follows:

1. Move upwards using rt stitches or downwards using lb stitches until a tile is found whose vertical range contains y_p. The vertical range of a tile includes the tile's bottom up to but not including the tile's top.

2. If the tile contains the point P , then stop. Otherwise move left using the bl stitch if, along the line $y = y_p$ P is to the left of the tile, or move right using the tr stitch if P is to the right of the tile.

3. Go to step 1

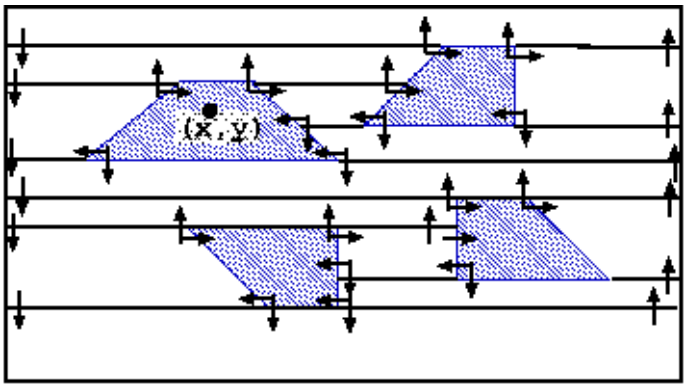


Figure 2.9: *Example point search showing the tile enumeration order and traversal of tile stitches.*

Fig. 2.9 shows an example of a point search. If the tiles in a tile plane are of relatively uniform size, the number of tiles intersecting a horizontal or vertical line

is approximately $\sqrt{N}$. Therefore, the complexity of the point search algorithm is O($\sqrt{N}$) where N is the total number of tiles in the plane. The worst-case complexity is N when all tiles line up in a row or column. In practice, the point search performs better than $\sqrt{N}$ (or N) if one can start from a tile close to the point.

2.2.2 Area Searching Algorithm

Area searching is used to find all tiles overlapping (with $surfacearea > 0$) a given area, where the area is a rectangle. The algorithm is recursive and resembles a depth first search algorithm. The area search algorithm is pivotal upon tile left edges. Tiles overlapping the area, but whose left edge does not extend into the area are called *seed tiles*. Seed tiles always lie along the left hand side of the area and are used to begin the recursive area search. Tiles overlapping the area whose left edge extends into or overlaps the area are reported during the search from a seed tile. In general, files are visited from left to right and from top to bottom (see Figure 2.10). When a tile is reported, all neighbour tiles along its right edge and overlapping the area are visited. Each visited tile is reported recursively if the portion of its left edge lying within the area is above the reported tile's bottom. The algorithm operates as follows:

1. For each seed tile, ordered from top to bottom along the left edge(s) of the area, do steps 2 and 3.

2. Report the tile and find all right neighbor tiles via an edge walk along the tile's right edge that a) overlap the area and b) do not extend below the reported tile within the area.

3. For each tile found in step 2, perform step 2 and 3 recursively.

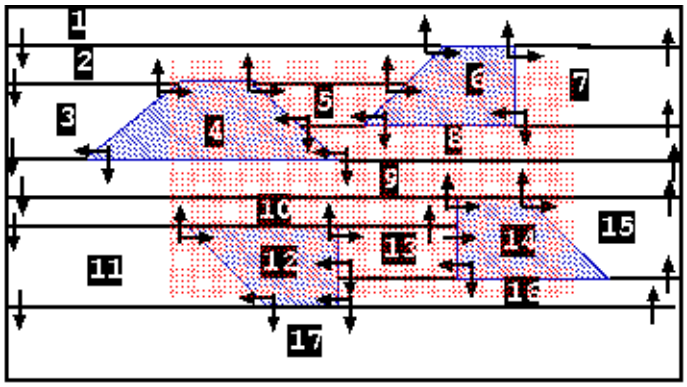


Figure 2.10: *Example area search showing the tile enumeration order and traversal of tile stitches.*

Figure 2.10 shows an example of a area search. The complexity of enumerating tiles in an area is $O(n)$, where n is the number of tiles overlapping the area. Before an

area search can begin, the first seed tile must be found using a point search and seed tile search. Therefore, the total complexity of a general area search is $O(\sqrt{N} + n)$.

2.2.3 Neighbour Searching Algorithm

Neighbour searching is used to find all tiles having overlapping edges with the query tile. The neighbour search algorithm is pivotal upon the four pointers(rt, tr, lb, bl) of the query tile. So there are four *seed tiles*. The algorithm operates as follows:

1. For seed tile rt, follow the bl pointer and report the tile if the bottom edge of the tile is ovelapping with the top edge of the query tile. Otherwise stop.

2. For seed tile tr, follow the lb pointer and report the tile if the left edge of the tile is ovelapping with the right edge of the query tile. Otherwise stop.

3. For seed tile lb, follow the tr pointer and report the tile if the top edge of the tile is ovelapping with the bottom edge of the query tile. Otherwise stop.

4. For seed tile bl, follow the rt pointer and report the tile if the right edge of the tile is ovelapping with the left edge of the query tile. Otherwise stop.

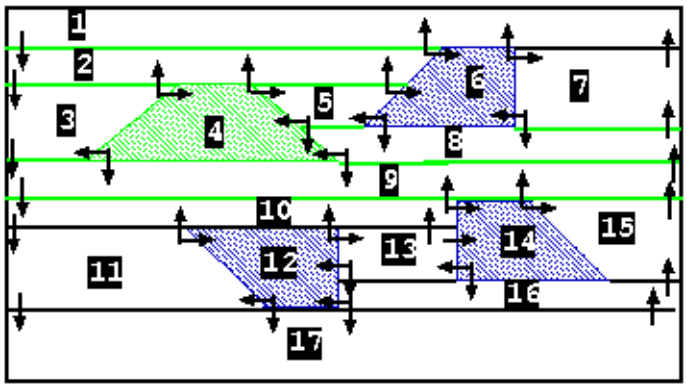


Figure 2.11: *Example neighbour search showing the tile enumeration order and traversal of tile stitches.*

Figure 2.11 shows an example of a neighbour search. The complexity of enumerating tiles is $O(n)$, where n is the number of neighbour tiles. Before an neighbour search can begin, the first seed tile must be found using a point search and seed tile search. Therefore, the total complexity of a general area search is $O(\sqrt{N} + n)$.

Chapter 3

Tree Based Data Structures

In this chapter we will discuss the tree based data-structures for layouts having rectilinear tiles as well as layouts having trapezoidal tiles. We will discuss the algorithms for point searching, area searching and neighbour searching.

3.1 The Quad Tree

The quad tree([Ked82]) can be generalized to organize any two dimensional collection of objects. We start with a big rectangle that contains all the objects. This rectangle is the root of the quad tree. That rectangle is divided to four equal subrectangles by dividing each of its sides into two (See Figure 3.1). The four rectangles are the sons of the original rectangle. In turn, each of these rectangles is divided into four, and so on. Now each node in the tree has a list of objects that resides in that node.

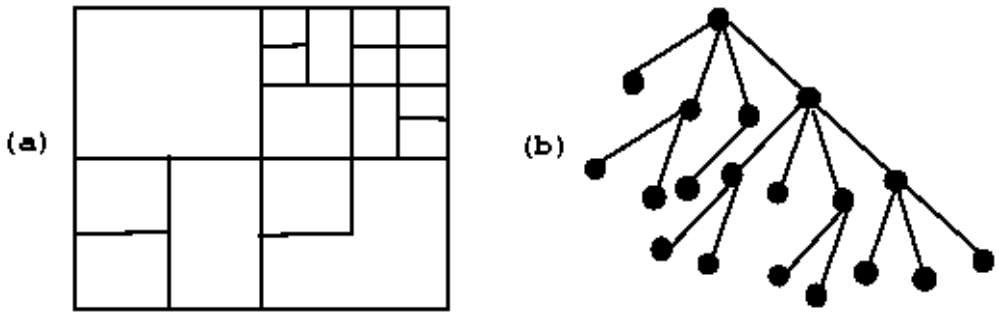


Figure 3.1: *a) Two dimensional representation of a quad tree b) The corresponding tree structure*

An object is put into a node if and only if it is inside the rectangle that corresponds to that node but is not inside any of its sons. Since all the objects are enclosed in the first rectangle, each object resides in a node. This node is uniquely determined by the above rule. Since most rectangles in IC design tend to be small and uniformly distributed, most of the rectangles will be at the leaves of the quad tree. Theretore, the expected depth of the tree is 0(log N). One more problem is left:

Many small objects will be at the upper nodes of the quad tree, namely, those that intersect the (rectangle) division lines. Therefore, if a window intersects a division line of an upper node, all the objects that also intersect a division line of that node have to be checked. Even the object that are far away from the window have to be checked (See Figure 3.2).

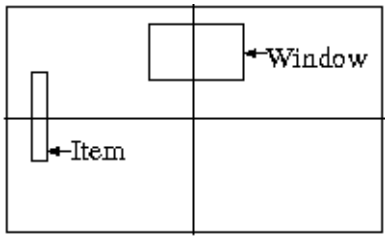


Figure 3.2: *Tiles Intersecting Division lines*

In order to reduce the search effort, the items are not put in a list at the quad-tree node. Instead, they are put into a binary tree that divides either the X dividing line or the Y dividing line. The binary tree represents a successive bisection of the line segment. An object is put into a node if a dividing point is inside that node. For example, In Figure 3.3 it is shown that how a line segment is being divided and the corresponding binary tree.

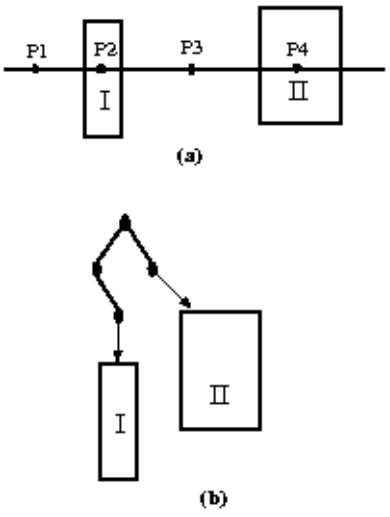


Figure 3.3: *(a) Division of a line segment, (b) The corresponding binary tree*

There are different variations of Quad trees such as Hinted Quad tree ([GGLW96]), Multiple Storage Quad trees ([HF90]), Dual Quad trees ([NR86]). The underlying algorithms has a common structure with some modifications to achieve better performance on different layout tool objectives such as compaction or design rule verification. We will only discuss the algorithms for the simple Quad tree.

3.1.1 Point Searching Algorithm

We need to traverse the quad tree by checking whether the x value of the query point lies left or right with respect to the bisector lines and y value lies above or below the bisector lines. When we reach a leaf node of the quad tree, we check in the list of tiles that the query point lies in a tile or not. In the worst case we may have to traverse the height of the tree. Since the height of the tree is $O(logN)$, the complexity of the point search is also $O(logN)$.

3.1.2 Area Searching Algorithm

REGIONSEARCH (qptr, region) qptr points to the root of the tree. region is an array containing 4 coordinates of a rectangle.

1. If $(qptr == NULL)$ then return.
2. If this is a leaf of tree nodes then scan through the tile list and report the tiles that intersect with the given region. After checking all tiles in the list, return.
3. If a given region intersects the boundaries of the first child, then $REGIONSEARCH(qptr- > first, region)$
4. If a given region intersects the boundaries of the second child, then $REGIONSEARCH(qptr- > second, region)$
5. If a given region intersects the boundaries of the third child, then $REGIONSEARCH(qptr- > third, region)$
6. If a given region intersects the boundaries of the fourth child, then $REGIONSEARCH(qptr- > fourth, region)$

End REGIONSEARCH

3.1.3 Neighbour Searching Algorithm

To find the neighbours for a tile pointed by *querytile* with distance being the distance to expand tile's extent in $+X, +Y, -X, -Y$ to form the search window, we use the following algorithm. NEIGHBOURSEARCH(querytile, distance)

1. Expand the query tile by the given distance along all the four edges of the query tile.
2. Store the coordinates of the generated window in the array called $region$.
3. call REGIONSEARCH(qptr, region) Where qptr points to the root of the quad tree.

End NEIGHBOURSEARCH.

Chapter 4

Partial Bin Based Hierarchical Corner Stitching

4.1 Bin Layer

The size and shape of the bin layer are the same with the corner stitching layer. All tiles in the corner stitching layer are projected to the bin layer, as shown in Figure 4.1. An imaginary square grid divides the area into mxn bins, which are managed by a two-dimensional array. The bins are indexed by its position at x and y directions, such as Bin(1,3) or Bin(5,2). Given the coordinates of a point and the size of bins, we can easily determine the indexes of the bin that contains the point. If we directly add the bin-based structure onto the corner stitching layer, which means that all of the tiles intersecting a particular bin are linked together and stored in that bin, then too much data redundancy would occur. So we simplify the bin layer by cutting out the superfluous, and get the new hierarchical PB corner stitching structure, as shown in Figure 4.2. A tile is only kept in the bin where the lower left corner of the tile's projection locates. In Figure 4.1, the solid tile's projection on the bin layer covers two bins: Bin(3,2), Bin(4,2), but only one of them Bin(3,2) keeps a pointer to the tile. Also, when a bin covers more than one projection's lower left corner, as Bin(3,1) in the figure, only one tile is kept in the bin. This idea is based on the fact that corner stitching is efficient at neighbor searching. When performing point searching, first a tile near the given point is obtained from the bin layer, and then that tile would be used as a starting tile to search in the corner stitching layer. Since the searching in the corner stitching layer is started near the target tile, the scope of the searching is limited; thus the searching time would not be decreased.

Figure 4.1

In hierarchical PB corner stitching, bin layer keeps the global position information of some tiles, which greatly improves the speed of obtaining tiles from coordinates, namely point searching. In PB corner stitching, each bin keeps no more than one pointer, and the searching within a bin is via point searching in corner stitching layer,

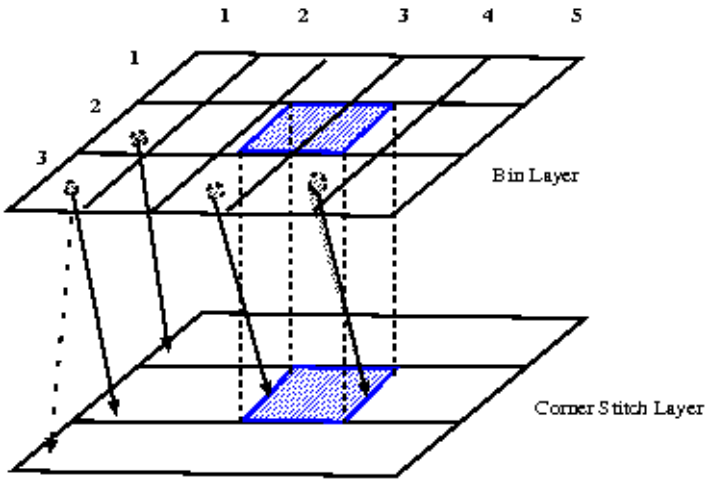


Figure 4.1: *The configuration of PB Corner Stitching.*

which is faster than searching in linked lists.Compared to traditional corner stitching, the memory requirement of PB corner stitching structure increases r^2 times, where r^2 is the number of bins.

Figure 4.2

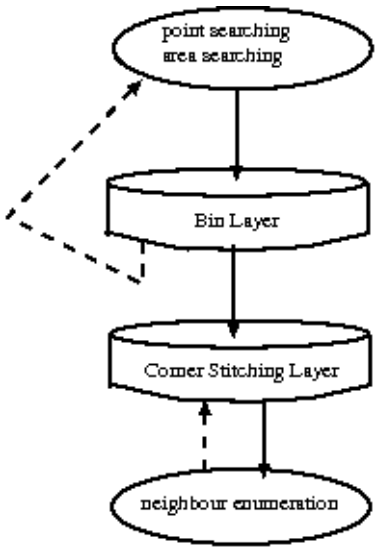


Figure 4.2: *The operations of PB Corner Stitching.*

4.2 Point Searching Algorithm

Point searching is used to locate the tile containing a given point P. The algorithm first obtains a tile near the point from bin layer, then switches to the corner stitching layer, starts with that tile, and iterates in y and then x until the target tile is found. The algorithm goes as follows:

1. Given the coordinates of $P : X_p, Y_p$, and the size of bin:S, figure out the bin that P lies in, called bin(a,b), where $a = \lfloor X_p/S \rfloor, b = \lfloor Y_p/S \rfloor$.
2. If bin(a,b) is not empty, then set the starting tile to bin(a,b), go to step 3. Otherwise, search all the bins that lie to the lower left of bin(a,b), until a bin that is not empty, or bin(a1,b1) is found, where $O <= a1 <= a, O <= b1 <= b$, then set the starting tile to bin(a1,b1)
3. Move upwards using rt stitches or downwards using lb stitches until a tile is found whose vertical range contains Y_p.
4. If the tile contains P, then stop. Otherwise move left using the bl stitches, or move right using the tr stitches.
5. Go to step 3.

Suppose the bins are much larger than the average tile size, then the speed of the point searching depends on step 3 to step 5. Which is traditional corner stitching point searching in the bin's range. Thus the complexity of the point searching is $O(N^{1/2}/r)$, where N is the total number of tiles and r^2 is the total number of bins.

4.3 Area Searching/Enumeration

Area enumeration is used to report all tiles in a tile plane by giving a bounding box as the area. The area enumerate complexity assumes the tile covers the lower left corner of the area has already been found. Since the searching area is limited, the point searching used in area enumeration is the traditional corner stitching point searching. Area enumeration is O(n), where n is the number of tiles in the area.

4.4 Neighbour Searching/Enumeration

Neighbour searching/enumeration is used to report all tiles which have a overlapping edge with the query tile. The neighbour enumerate complexity assumes the tile covers the lower left corner of the tile has already been found. Since the searching neighbours is limited, the point searching used in neighbour enumeration is the traditional corner stitching point searching. Neighbour enumeration is O(n), where n is the number of neighbour tiles.

Chapter 5

Tree Based Trapezoidal Corner Stitching

As mentioned in 1 chapter, the atomic operations point searching, area searching and the neighbour searching algorithms are of prime importance in the layout tools. The partial bin based corner stitching also has some limitations. The number of bins would affect the efficiency of PB corner stitching. If the bin number is too small, and then PB corner stitching tends to resemble traditional corner stitching, the improvement of efficiency is limited. If the bin number is too large, then a tile may cover several bins. It might be slower to find a nearby tile to a point in the bin layer, thus affect the speed of point searching. Also, if there are too many bins, the memory requirement would increase too much. One question that arises from the discussions in the previous sections is the following:

> *Can we further extend the trapezoidal corner stitching to make it more efficient for point searching algorithm? What would be the added extra layer?*

Our goal here is to analyze the performance of the point searching, area searching and the neighbour searching algorithms using newly introduced data-structure Tree Based Trapezoidal Corner Stitch (see Figure 5.1).

5.1 Implementation Details of the Data-Structure

Our layout data-structure consists of two layers namely, a tree layer and a corner-stitch layer. We have shown a tree layer in Figure 5.3 for an example layout shown in Figure 5.2.

5.1.1 Cornerstitch Layer

We have maintained an array of structures. Each element of the array which is a structure contains the detailed information of a layout tile. The details are already

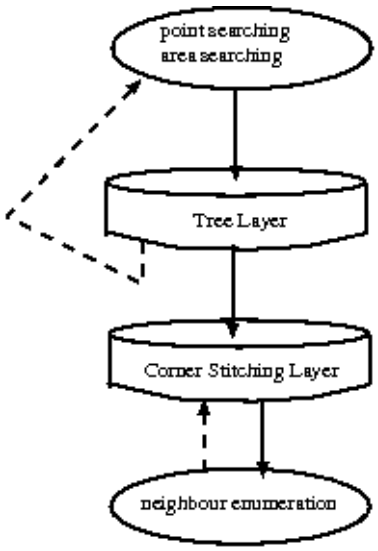


Figure 5.1: *The operations of TB Trapezoidal Corner Stitching.*

explained in section 2.2 of chapter 2. Besides that for each tile the information of the leftmost x and the rightmost x information is also stored in the structure.

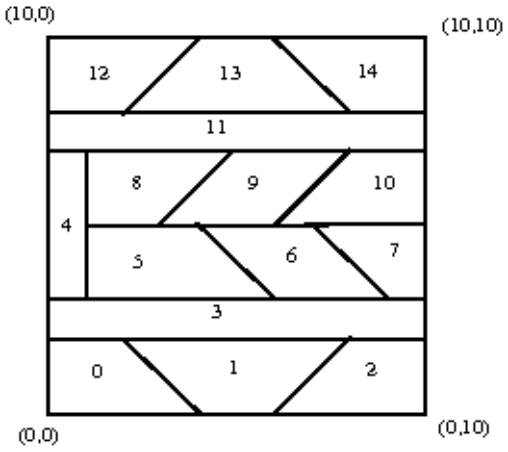


Figure 5.2: *An Example Layout containg all types of trapezoidal tiles*

5.1.2 Tree Layer

Each node in the tree layer (see Figure 5.3) consists of array of indices of the layout tiles and pointers to its left and right child. The odd level nodes also contains the median x value of all the tile's left-most x value in the node and the even level contains the median y value of all the tile's top-most y value in the node. When creating nodes of a even levels, a partition of the tile indices is done based on the

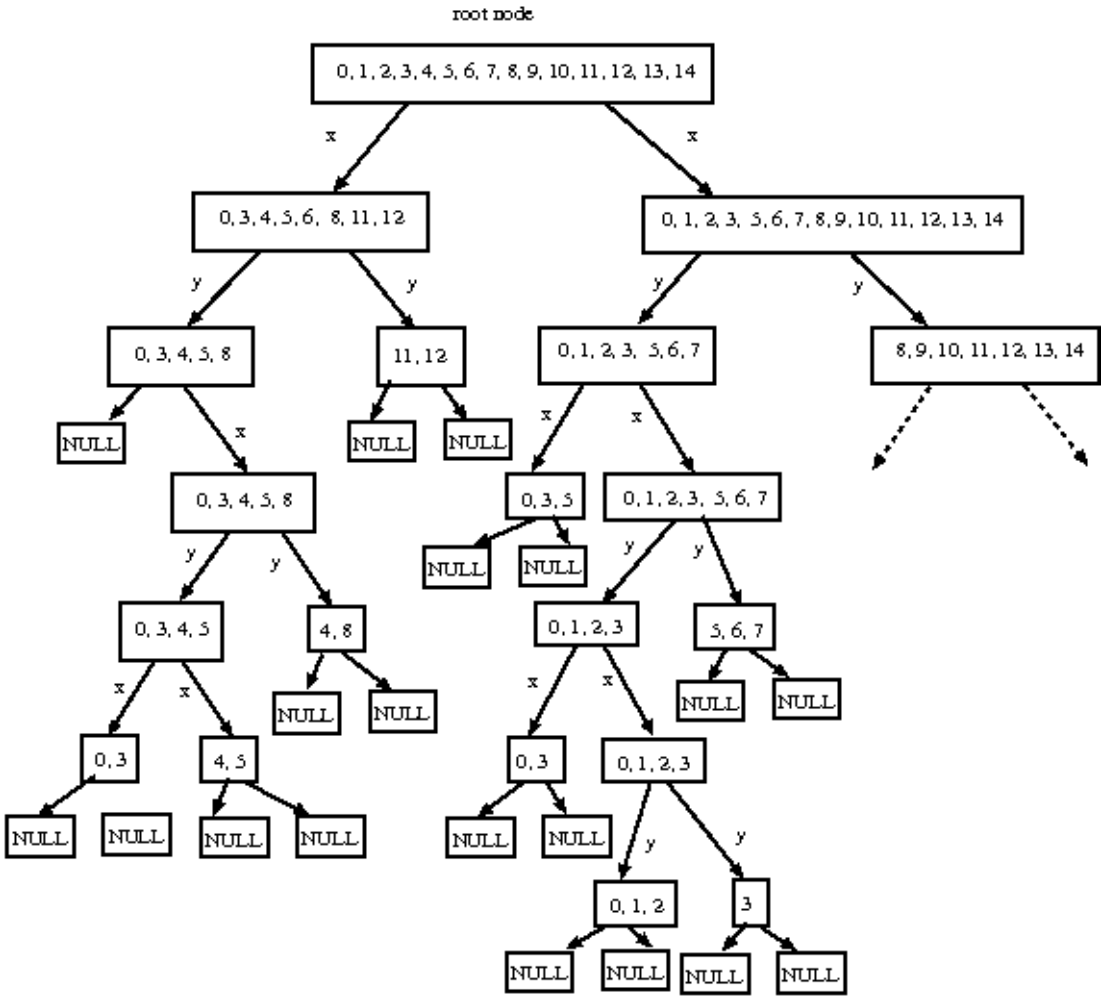


Figure 5.3: *The tree structure for the Example layout in Figure 5.2*

median x value. The tiles which have their left-most x value less median x value are put in the left child and the rest are put in the right child. And the tiles which are in the left child are checked if their right-most x value is greater than the median x value. If it is so then those tile indices are also copied to the right child. Similarly, When creating nodes of a odd levels, a partition of the tile indices is done based on the median y value. The tiles which have their top-most y value less median y value are put in the left child and the rest are put in the right child. And the tiles which are in the left child are checked if their bottom-most y value is lesser than the median y value. If it is so then those tile indices are also copied to the right child. The leaf nodes contains a threshold number of tile indices which is a constant. This is done for a simpler implementation of the code. Through the tile indices stored in a tree node one can easily reach the tile in the cornerstitch layer and extract the detailed information (see Figure 5.4). The height of the tree is $O(logN)$ where N is

the number of tiles in the layout. Space required is $O(NlogN)$.

5.2 Algorithms Implemented

In this section we will discuss the point searching, area searching and the neighbour searching algorithms for our data-structure which are of primary importance for a layout tool.

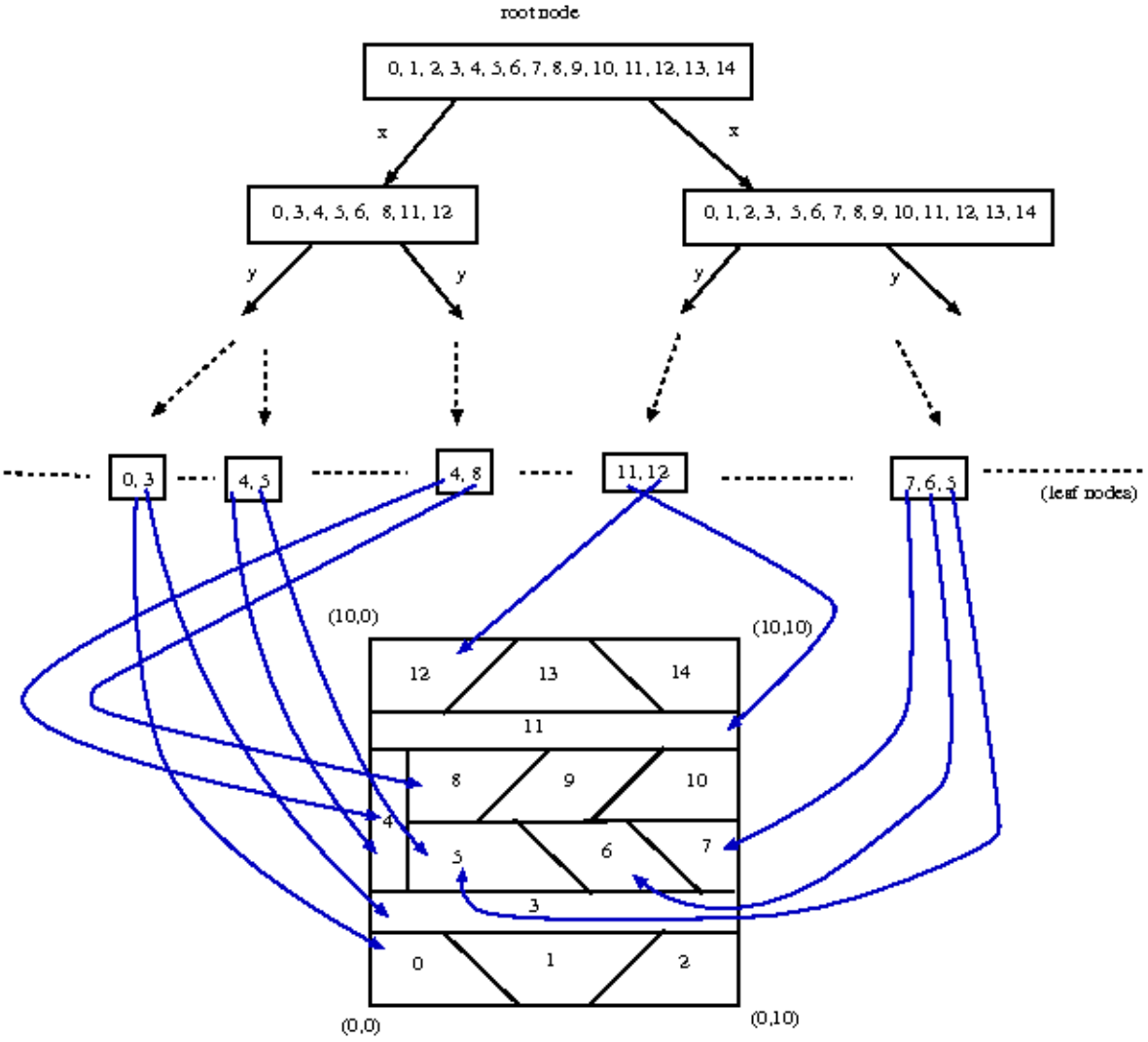


Figure 5.4: *Connection between the tree structure and the corstitched tiles*

5.2.1 Point Searching

Point searching is used to locate the tile containing a given point P(x,y). We use our tree layer to find the tile index for the point and then report the details of the tile from the corner stitch layer. The algorithm iterates in x and then y alternately depending on the level of the tree, starting from the root node of the tree layer.

1. while a leaf node is not reached do step 2 and step 3.

2. If the level is odd then check
   if $x < medianx$ of the node then go to left child of current node.
   else go to right child of the current node.

3. If the level is even then check
   if $y < mediany$ of the node then go to left child of current node.
   else go to right child of the current node.

4. check in the list of tiles in the leaf node.
   If a tile is found that contains the query point
   then report the tile information from corner stitch layer.
   else report point does not lie in the layout.

5.2.2 Area Searching

Area searching is used to find all tiles overlapping (with $surfacearea > 0$) a given area, where the area is a rectangle. The algorithm is recursive and resembles a depth first search algorithm. The area search algorithm is pivotal upon tile left edges. Tiles overlapping the area, but whose left edge does not extend into the area are called *seed tiles*. Seed tiles always lie along the left hand side of the area and are used to begin the recursive area search. Tiles overlapping the area whose left edge extends into or overlaps the area are reported during the search from a seed tile. In general, files are visited from left to right and from top to bottom (see Figure 2.5). When a tile is reported, all neighbour tiles along its right edge and overlapping the area are visited. Each visited tile is reported recursively if the portion of its left edge lying within the area is above the reported tile's bottom. The algorithm operates as follows:

1. The index of the first seed tile is found from the tree layer by point search algorithm described in section 5.2.1.

2. For each seed tile, ordered from top to bottom along the left edge(s) of the area, do steps 2 and 3.

3. Report the tile and find all right neighbor tiles via an edge walk along the tile's right edge that a) overlap the area and b) do not extend below the reported tile within the area.

4. For each tile found in step 2, perform step 2 and 3 recursively.

The complexity of enumerating tiles in an area is $O(n)$, where n is the number of tiles overlapping the area. Before an area search can begin, the first seed tile must be found using a point search and seed tile search. Therefore, the total complexity of a general area search is $O(logN + n)$.

5.2.3 Neighbour Searching

Neighbour searching is used to find all tiles having overlapping edges with the query tile. The neighbour search algorithm is pivotal upon the four pointers(rt, tr, lb, bl) of the query tile. So there are four *seed tiles*. The algorithm operates as follows:

1. For seed tile rt, follow the bl pointer and report the tile if the bottom edge of the tile is ovelapping with the top edge of the query tile. Otherwise stop.

2. For seed tile tr, follow the lb pointer and report the tile if the left edge of the tile is ovelapping with the right edge of the query tile. Otherwise stop.

3. For seed tile lb, follow the tr pointer and report the tile if the top edge of the tile is ovelapping with the bottom edge of the query tile. Otherwise stop.

4. For seed tile bl, follow the rt pointer and report the tile if the right edge of the tile is ovelapping with the left edge of the query tile. Otherwise stop.

If the query tile itself is not given it can be retrieved by our point search algorithm presented in section 5.2.1. The complexity of enumerating tiles is $O(n)$, where n is the number of neighbour tiles. Before an neighbour search can begin, the first seed tile must be found using a point search and seed tile search. Therefore, the total complexity of a general area search is $O(logN + n)$.

5.2.4 Insertion of a Tile

The first step is to run the area search algorithm to check that there are no existing solid tiles in the desired area of the new tile. The second step is to insert the tile in to the data-structure which involves spliting and merging of the vacant tiles. The insertion algorithm is as follows:

1. Find the space tile containing the top edge of the area to be occupied by the new tile.

2. Split the top space tile in to a piece entirely above the new tile and a piece overlapping the new tile. Update corner stitches in the tiles adjoing the new tile.

3. Find the space tile containing the bottom edge of the new tile, split it in the same manner and update the corner stitches around it.

4. Traverse along the left edge of the new tile, each tile along this edge must be a vacant tile that spans the entire width of the new tile. Split the space tile in to piece entirely to the left of the new tile, a piece entirely to the right of the new tile, and a piece entirely with in the new tile. Now Merge the center space tile with the solid tile. Each spliting or merging requires updation of the stitches of the adjoining tiles.

5. Delete the older vacant tile index from the tree layer.

6. For each tile generated in the corner stitch layer we enter the index of the new tile in the tree layer. This is done by searching from the root checking with the median x or the median y values depending upon the level (similar to point search in the tree layer) till a leaf node is reached.

7. If the threshold value exceeds in the leaf, split the node to the next level by creating a left and a right child based on the current level and the median x or the median y value

5.2.5 Deletion of a Tile

The deletion is complicated by the need of split and merge space tiles so as to maintain the horizontal strip representation. We present the algorithm that works in the clockwise direction around the tile being deleted, which is referred as *dead tile*.

1. Change the type of the dead tile to vacant.

2. Use neighbour finding algorithm to search from top to bottom through all the tiles that adjoin the right edge of the dead tile.

3. For each vacant tile found in step 2, split either the neighbour or the dead tile so that the two tiles have the same vertical span, then merge the tiles together horizontally.

4. When the bottom edge of the dead tile is reached, scan upwards along the left edge of the original dead tile to find all space tiles that are left neighbours of the original dead tile.

5. For each vacant tile found in step 4, merge the vacant tile with the adjoining remains of the original dead tile. Do this by repeating steps 2 and 3, treating the current vacant tile as the dead tile in the step 2 and 3.

6. It is also necessary to do vertical merging in step 5. After each horizontal merging in step 5, check to see if result tile can be merged with the tiles just above or below it, and merge if possible.

7. Delete the dead tile index from the tree layer.

8. If the tiles are merged during deletion operation in the corner stitch layer then delete the indices of the older tiles and insert the index of the new merged tile.

9. If the threshold value exceeds in the leaf, split the node to the next level by creating a left and a right child based on the current level and the median x or the median y value.

5.3 Work flow of the Code

The code is written using C on Linux platform. There are multiple C files for different functions. The main function is written in *LayoutAtomicOperations.c*. The trapezoidal corner stitch layer is generated by *TrapezoidalCornerStichConstruct.c* using the input layout file named *InputLayoutFile.txt* and the tree layer is generated by *TreeLayerConstruct.c* using the generated trapezoidal corner stitched array of structures.

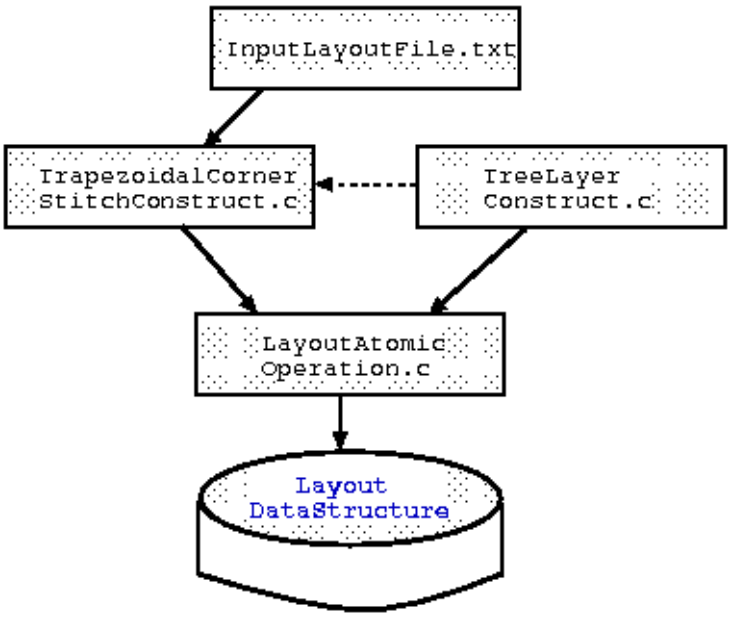


Figure 5.5: *Layout Data-Structures Construction*

The *pointSearch.c*, *areaSearch.c* and *neighbourSearch.c* files are used to output tiles given a query point, coordinates of query rectangle and coordinates of the query tile respectively. The tile information are stored in files named *pointTile.txt*, *areaTile.txt* and *neighbourTile.txt* for point searching, area searching and neighbour searching respectively. Internally, the *areaSearch.c* and *neighbourSearch.c* calls *pointSearch.c* to find the seed tile.

Sample content of file *InputLayoutFile.txt*:
(x1,y1), (x2,y2), (x3,y3), (x4,y4) type name

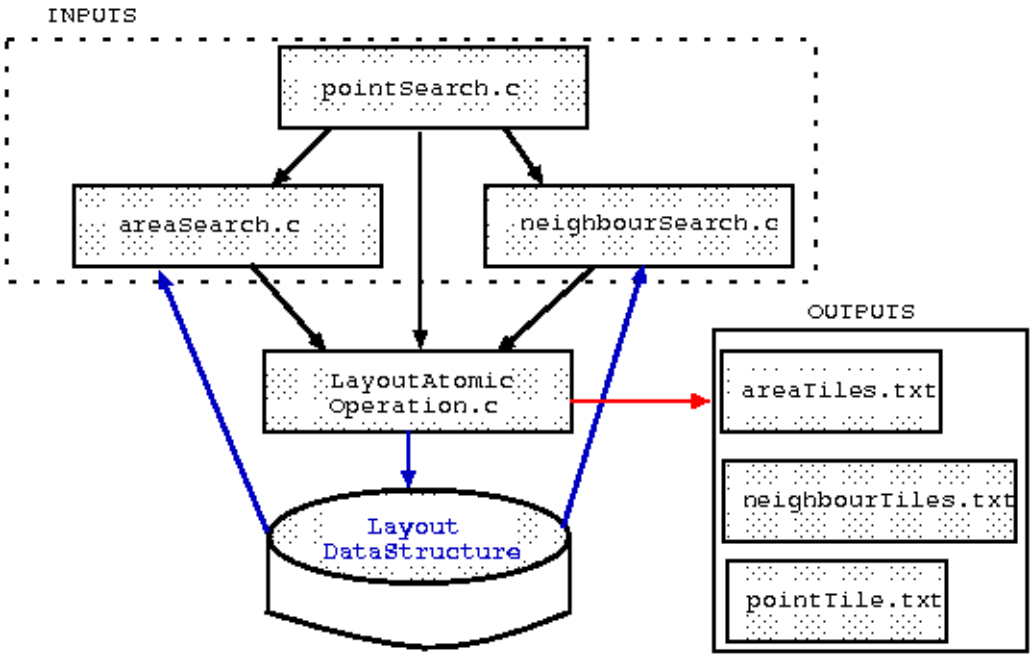


Figure 5.6: *Point search, Area search and Neighbour search programs interacting with the tree based trapezoidal data-structure*

The type parameter indicates the shape of the tile which is among the nine tiles as shown in Figure 2.7. The name parameter indicates type of tile that can be solid (diffusion, polysilicon, metal), vacant. Instructions for running the code:

1. make
2. $./laos < inputgen >< trx >< try >< pSearchKD >< pSearchCS >< aSearch >< nSearch >$

Meaning of the different parameters:

1. inputgen can take 1 or 0 value, 1 mean need to generate *InputLayoutFile.txt*, 0 means *InputLayoutFile.txt* already exists.
   if inputgen = 0, then trx = try = 0. if inputgen = 1, then trx = try = multiple of 10, (top right vertex of the bounding box of layout)
2. For point searching
   $pSearchKD = 0/1$: OFF/ON
   $pSearchCS = 0/1$: OFF/ON
3. For Area searching $aSearch = 1/2$
   1 means it uses CS internally and 2 means it uses KD internally.

4. For Neighbour searching $nSearch = 1/2$
   1 means it uses CS internally and 2 means it uses KD internally.

5. Example runs:
   ./laos 1 10 10 0 0 0 0 generates only *InputLayoutFile.txt* file
   ./laos 0 0 0 1 0 0 0 doing point search using tree layer
   ./laos 0 0 0 0 1 0 0 doing point search using corner stitch layer
   ./laos 0 0 0 0 0 1 0 doing area search
   ./laos 0 0 0 0 0 0 1 doing neighbour search

Chapter 6

Comparative Study of Different Layout Data Structures

In this chapter we will compare the time and the space complexities of the point searching, area searching and neighbour searching algorithms performed using different data-structures

6.1 Time Complexity for Different Data-Structures

In the Table 6.1 we have give the time complexities of the point searching, area searching and neighbour searching algorithms performed using different data-structures The tree based trapezoidal corner stitching (TBTCS) wins over traditional corner

Table 6.1: Comparion of Time Complexities

Operation	Linked List	Quad Tree	4-D Tree	Corner Stitching	PBCS	TBTCS
Point Search	$O(N)$	$O(logN + T)$	$O(logN)$	$O(N^{1/2})$	$O(N^{1/2}/r)$	$O(logN)$
Area Search	$O(N)$	$O(T + n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Neighbour Search	$O(N)$	$O(logN + T)$	$O(logN)$	1	1	1

stitching and other tree based data-structures. The partial bin based corner stitching has a competitive time complexity, but it depends on the number of bin. And the estimation of the number of bins required for a layout is a difficult problem to answer because if the number of bins is too small, and then PB corner stitching tends to resemble traditional corner stitching, the improvement of efficiency is limited. If the bin number is too large, then a tile may cover several bins. It might be slower to find a nearby tile to a point in the bin layer, thus affect the speed of point searching.

6.2 Space Complexity for Different Data-Structures

In the Table 6.2 we have give the space complexities of different data-structures Our

Table 6.2: Comparion of Space Complexities (in Bytes)

Linked List	Quad Tree	4-D Tree	Corner Stitching	PBCS	TBTCS
$12N$	$2N^{3/2} + 4(N-2)$	$2NlogN + 4(N-2)$	$32N$	$32N + 8r^2$	$36N + 2NlogN + 4(N-2)$

data structure TBTCS requires more space than all other data-structures, but now a days memory is it is a good deal to have better time complexity sacrificing some more space.

Chapter 7

Conclusion

In this dissertation work we have described the different data-structures used in the layout tools such as Corner Sitching and Quad trees. Both variations have their advantages and disadvantages. So we have proposed the new data-structure namely Tree Based Trapezoidal Corner Stitching. It takes more space as compared to other mentioned data-structures and the insertion and deletion operations also have more complexity. But the most important atomic operations for the layout tools that is the point searching, area searching and the neighbour searching are improvised when compared to other tree based data-structures and corner stitched data-structures.

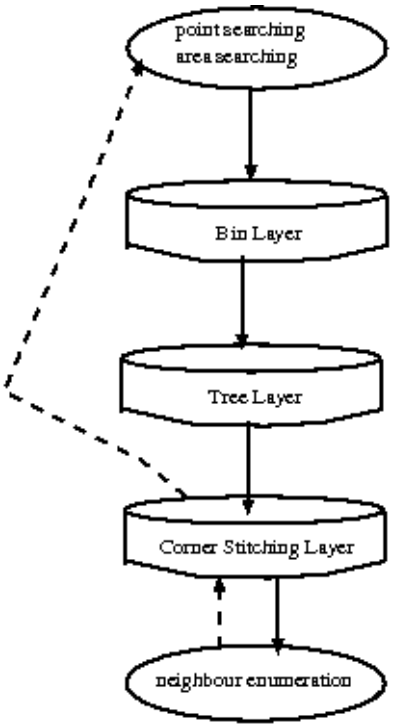


Figure 7.1: *The operations of 3 - Layer Trapezoidal Corner Stitching.*

We have also given a thought of a future work by introducing more layers to the data-structure which may increase the storage requirement but the time complexity for the atomic operations may drastically come down. In Figure 7.1 we have shown a 3-layer trapezoidal corner stitching which consists of a bin layer, a tree layer and a corner stitched layer. We have not done the complexity analysis for the algorithms based on this data-structure.

Bibliography

[BM97] G. Blust and Dinesh P. Mehta. Corner stitching for simple rectilinear shapes. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 16(2):186–198, 1997.

[DMH90] Michiel Smulders David Marple and Henk Hegen. Tailor: A layout system based on trapezoidal comer stitching. *IEEE Transactions on Computer-Aided Design*, 9(1):66–90, 1990.

[GGLW96] Donald S . Fussell Glenn G. Lai and D. F. Wong. Hinted quad trees for vlsi geometry drc based on efficient searching for neighbors. *The American Mathematical Monthly*, 15(3):317–324, March 1996.

[HF90] Pei-Yung Hsiao and Wu-Shiung Feng. Using a multiple storage quad tree on a hierarchical vlsi compaction scheme. *IEEE Transactions on Computer-Aided Design*, 9(5):522–536, 1990.

[JKOT84] Robert N. Mayo Walter S. Scott John K. Ousterhout, Gordon T. Hamachi and George S. Taylor. Magic: A vlsi layout system. *21st Design Automation Conference*, pages 152–159, 1984.

[Ked82] Gershon Kedem. The quad-cif tree: A data structure for hierarchical on-line algorithms. *19th Design Automation Conference*, pages 352–357, 1982.

[MB99] M. Overmars O. Schwarzkopf M. Berg, M. Kreveld. *Computational Geometry Algorithms and Applications*. Springer, 1999.

[NR86] S. K. Nandy and I. V. Ramakrishnan. Dual quadtree representation for vlsi designs. *23rd Design Automation Conference*, pages 663–666, 1986.

[Ous84] John K. Ousterhout. Corner stitching: a data structuring technique for vlsi layout tools. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 3(1):87–91, 1984.

[SF93] Carlo H. Scquin and H. Faqanha. Corner-stitched tiles with curved boundaries. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 12(1):47–58, 1993.

[She93] Naveed A. Sherwani. *Algorithms for VLSI Physical Design Automation.* Kluwer Academic Publishers, 1993.

[ZYX00] C. Yici Z. Yan, W. Baohua and H. Xianlong. Area routing oriented hierarchical corner stitching with partial bin. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 16(2):105–110, Spring 2000.