To the Graduate Council:

I am submitting herewith a thesis written by George Joseph Blust entitled "Corner Stitching for L-shaped Tiles." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

D P Mehta

Dr. Dinesh Mehta, Major Professor

We have read this thesis
and recommend its acceptance:

Bern Ulmlich

A Pujol Jr

Accepted for the Council:

Lou Winkel

Associate Vice Chancellor and
Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at the University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature George J Blust

Date April 23, 1994

Corner Stitching for L-shaped Tiles

A Thesis

Presented for the Master of Science

Degree

The University of Tennessee, Knoxville

George Joseph Blust

May, 1994

Dedication

For my mother.

Acknowledgments

I would like to thank my advisor, Dr. Dinesh Mehta, for his help and leadership in the development of this thesis. I would also like to thank Dr. Bruce Whitehead and Dr. Al Pujol for serving on my thesis committee.

Abstract

Corner Stitching is a data structure for storing objects in interactive VLSI layout editors. This thesis extends corner stitching for rectangular objects proposed by Ousterhout to L-shaped objects. The addition of L-shaped objects provides added flexibility to handle more than rectangular objects and represent L-shaped channels in a layout. There is also a significant memory savings while the run times required for operations is still within an acceptable range for a human designer.

TABLE OF CONTENTS

CHAPTER	PAGE
I Introduction	1
II Operations and Existing Data Structures	4
2.1 Atomic Operations	4
2.2 Other Operations	5
2.3 Linked Lists	6
2.4 Bin-Based	7
2.5 Neighbor Pointers	7
III Rectangular Corner Stitching (RCS)	9
IV L-shaped Corner Stitching (LCS)	12
4.1 Introduction	12
4.2 Representation	13
4.3 Space Comparison	17
4.4 Neighbor Find	17
4.5 Point Find	19
4.6 Area Search	25
4.5 Insert	28
4.7 Delete	32
4.8 Directed Area Enumeration	35

V Experimental Results And Conclusions 38

5.1 Experimental Results 38

5.2 Conclusions 42

List of References 44

VITA 46

LIST OF FIGURES

Figure 2-1: Linked List Example 6

Figure 2-2: Bin Based Example 7

Figure 2-3: Neighbor Pointers Example 8

Figure 3-1: Example of RCS Tile 10

Figure 4-1: Example LCS Layout 12

Figure 4-2: Construction of an L Tile 14

Figure 4-3: L Types 15

Figure 4-4: Ambiguity in Stitches 16

Figure 4-5: Neighbor Find Example 19

Figure 4-6: Point Find Example 21

Figure 4-7: Point Find Proof 24

Figure 4-8: Area Search Example 27

Figure 4-9: Insert Example 31

Figure 4-10: Delete Example 33

Figure 4-11: Area Enumerate Example 36

Figure 5-1: Time per Insert Operation 41

Figure 5-2: Time per Delete Operation 41

LIST OF TABLES

Table 4-1 Space requirements of tiles in RCS and LCS . 17

Table 5-1 Layout A (10000 tiles) 40

Table 5-2 Layout B (3001 tiles) 40

Table 5-3 Layout C (907 tiles) 40

Table 5-4 Layout D (387 tiles) 40

Chapter I

Introduction

Layout editors are used by human designers to create and modify VLSI (Very Large Scale Integration) layouts. Each VLSI layout is usually represented by several layers of geometric elements. Each layer represents a different layer in an integrated circuit: diffusion, polysilicon, and metal. The elements on a layer are normally limited to manhattan features, rectangles and squares. Elements on the same layer do not overlap. The layout editor must allow the designer to modify the layout quickly and efficiently. VLSI chips are becoming larger and more complex. The new Intel Pentium chip has 3.1 million transistors on it. Each transistor is represented by a number of elements. So the total number of elements needed to represent all the transistors and interconnecting wires is some multiple of the number of transistors. As the number of elements to be handled by the editor increases, the workload that must be handled increases.

The layout editor relies on an underlying data structure. This data structure must allow fast modifications to minimize wasted time for the human designer. The higher number of graphical elements in new layouts requires the data structure to have a low time complexity for large number of elements. A data structure's *time complexity* is

defined as the number of nodes the data structure must examine to perform the desired operation for a layout with N nodes. The data structure must be space efficient since the number of nodes is large and the primary memory on workstations is limited. If the layout requires more primary memory than the workstation has then some of the layout must be placed on secondary storage. This will have an adverse affect on an editor's speed. The data structure determines the efficiency of the layout editor. A good graphical interface cannot make up for short comings in the data structure used to store the layout.

Corner stitching is a data structure proposed by Ousterhout [5] for representing rectangular tiles in interactive VLSI layout editing systems. A *tile* in this data structure is a rectangle that represents a rectangular area in a layout layer. There are two types of tiles. A *solid* tile can, depending on the layer, represent an area that is to contain diffusion, polysilicon, or metal. *Vacant* tiles are the area of a layer not occupied by a solid tile. It is superior to other data structures for layout editors such as linked lists, bins, and neighbor pointers because it allows fast localized algorithms for operations such as searching, deletion, creation, and compaction.

This thesis extends the corner stitching data structure so that, in addition to representing rectangular tiles, it also represents L-shaped tiles. This allows the movement

towards algorithms that operate on non-rectangular (but rectilinear) tiles. For example, L-shaped tiles have been used to obtain better solutions in floorplanning [9]; non-rectangular tiles are also useful in mixed place-and-route systems, where they are place in a "sea" of cells [7]. L-shaped channels have been considered as a means to eliminate cyclic channel-routing precedence constraints in non-slicing floorplans [1, 2]. This approach requires less memory than the alternative approach of partitioning each L-shaped block into two rectangular tiles. It does, however, retain the advantages of corner stitching.

Chapter 2 discusses operations needed for an interactive layout editor. Other previous data structures are also discussed. Chapter 3 presents the corner stitching data structure that Ousterhout proposed for rectangular tiles. Chapter 4 presents the corner stitching data structure for L-shaped tiles. Chapter 5 contains the results of experimental space and time comparisons between the Ousterhout's corner stitching data structure and the extension presented in this thesis.

Chapter II

Operations and Existing Data Structures

This chapter presents the operations an interactive editor must provide. Several previous data structures that have been used for editors are presented.

2.1 Atomic Operations

Atomic operations are a set of basic operations that give a designer the ability to fully manipulate a layout. More complex operations such as compaction rely on these simpler operations. A good data structure for interactive editing systems must be able to perform the following atomic operations quickly.

1. *Point find* - Given a point *(x,y)* determine the tile that contains that point.
2. *Neighbor find* - For a given tile find all tiles that have a side in common with the given tile.
3. *Block visibility* - For a given tile find all tiles that there exists a imaginary vertical or horizontal line between the tile and the given tile that passes only though vacant space. The set of found tiles are said to be "visible" from the original tile.
4. *Area search* - Given a rectangular area, does that area

contain any solid tiles.

5. *Directed area enumeration* - Given a rectangular area and a starting side, number each tile such that $i < j$ if an imaginary line extending perpendicularly from the starting side encounters T_i before encountering T_j.

6. *Block insertion* - Insert a given tile into the layout.

7. *Block deletion* - Delete the selected tile from the layout.

2.2 Other Operations

More complex operations are listed for completeness. These operations depend on the atomic operations defined in the previous section.

1. *Plowing* - Given an rectangular area A and a direction d, remove all solid tiles T_i from A by shifting each T_i in the direction d while preserving relative positioning between tiles.

2. *Compaction* - Compaction removes all unnecessary space between solid tiles. If the compaction is along the x-axis or y-axis then the compaction is a 1-dimensional compaction. If compaction is done along both x- and y-directions then it is called a 2-dimensional compaction.

3. *Channel Generation* - Channel generation creates vacant tiles for data structures that do not store vacant space.

2.3 Linked Lists

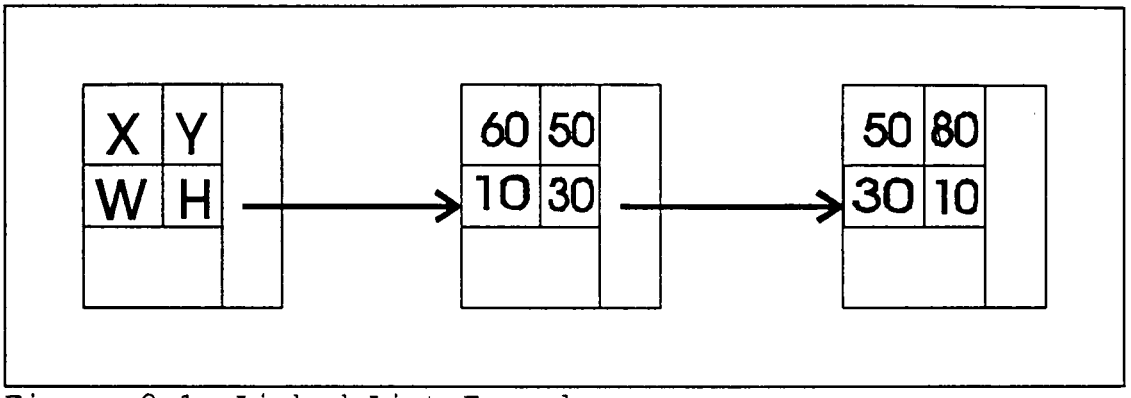


Figure 2-1: Linked List Example

The simplest data structure for representing the tiles in a layout is the linked list. Each node in the linked list represents a tile. Each node contains the tile's coordinates and a pointer to the next node in the list. Figure 2-1 shows an example linked list. The first node in the list gives the names of field names for each node. Since tiles are not stored in any particular order, operations such as point find may require that the entire linked list be searched. This data structure works well for layouts with few tiles in them or hierarchical systems with a few tiles on each level. Performance degrades when the number of tiles in the list grows large. Linked lists do not store vacant space. Therefore generating a channel for routing is a very complex operation.

2.4 Bin-Based

In a bin-based data structure a virtual grid is placed over the layout. Each square of the grid is a bin. A tile is placed in a bin if it intersects the square on the

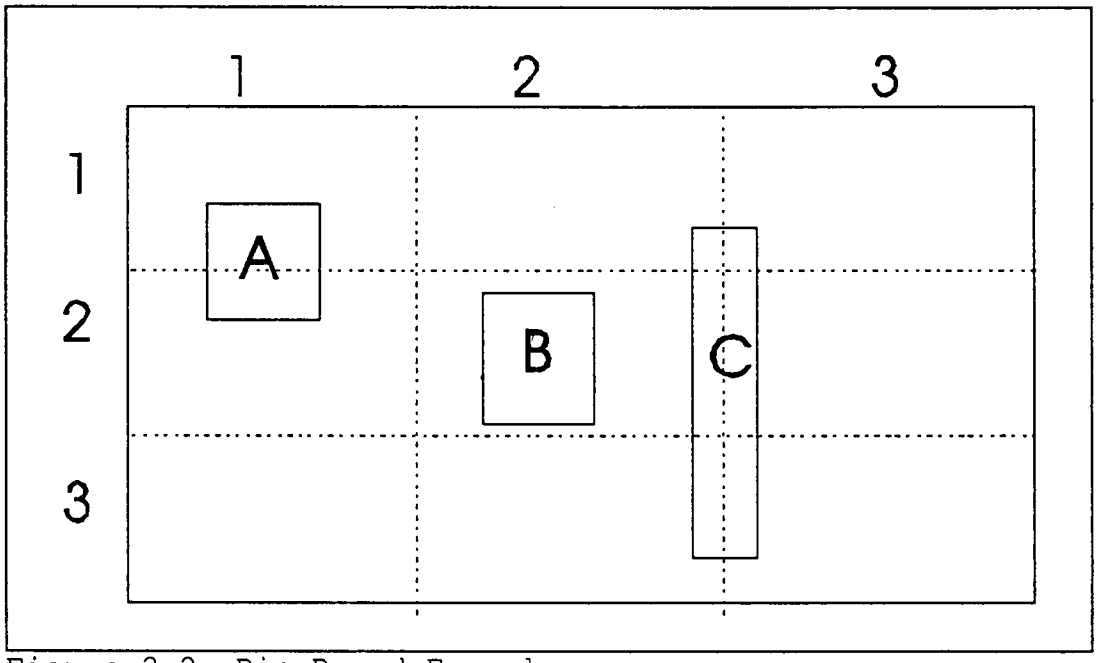


Figure 2-2: Bin Based Example

virtual grid corresponding to the bin. A tile can be in more than one bin. For example, in Figure 2-2 tile C is in 6 bins. All tiles in a bin are stored in a linked list.

Performance degrades when all tiles fall in the same bin or tiles intersect a large number of bins. Bin-based data structures do not store vacant space.

2.5 Neighbor Pointers

In the neighbor pointers data structure, each block in the layout is represented by a node that contains its

dimensions and pointers to all of its "visible" neighbors. Neighbors are defined as the tiles returned by the block visibility operation of Section 2.1. An example layout is shown in Figure 2-3. Neighbor pointers perform well on operations like compaction. This is because each node contains pointers to all blocks that would be affected when the block represented by the node is moved.

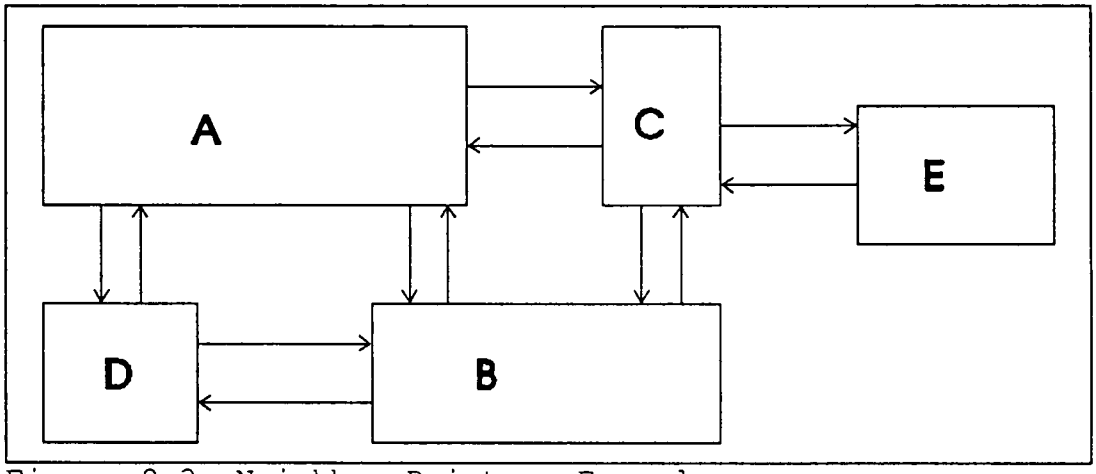


Figure 2-3: Neighbor Pointers Example

The data structure does have some disadvantages. One is that a simple modification to the structure usually requires all pointers in the structure to be recomputed. An example would be if a tile were moved to a new location. At the new location the tile may be visible to a new set of tiles and no longer visible to its previous neighbors. It may also come between two tiles thus the pointers between the two tiles must be deleted. Another is that vacant space is not represented. Major modifications to the structure would be needed to help generate channels.

Chapter III

Rectangular Corner Stitching (RCS)

The corner-stitching data structure for rectangular tiles was proposed by Ousterhout in [5]. This chapter describes how tiles are represented in the data structure. The algorithm for point find (Section 2.1) is also given as an example of how corner stitches are used.

The corner stitching data structure is unique in that it stores vacant space explicitly. This does increase the space required for a layout but generating channels for routing becomes much simpler. Vacant tiles are formed by extending horizontal lines from each of the four outer corners of a solid tile until they encounter another solid tile or a boundary of the layout region (Figure 3-1). In Figure 3-1, solid tiles are shaded, while vacant rectangles are left unshaded. This convention will be maintained throughout this thesis. Tiles contain their south and west edges, but not their north or east edges, so every point in the layout is in exactly one tile. The set of vacant tiles obtained is unique for a given set of tiles. The partitioning scheme ensures that vacant tiles are horizontally maximal; i.e., no two vacant tiles share a vertical side.

Each tile has associated with it the coordinates of its bottom left corner x_1 and y_1, and four pointers N, E, W, and

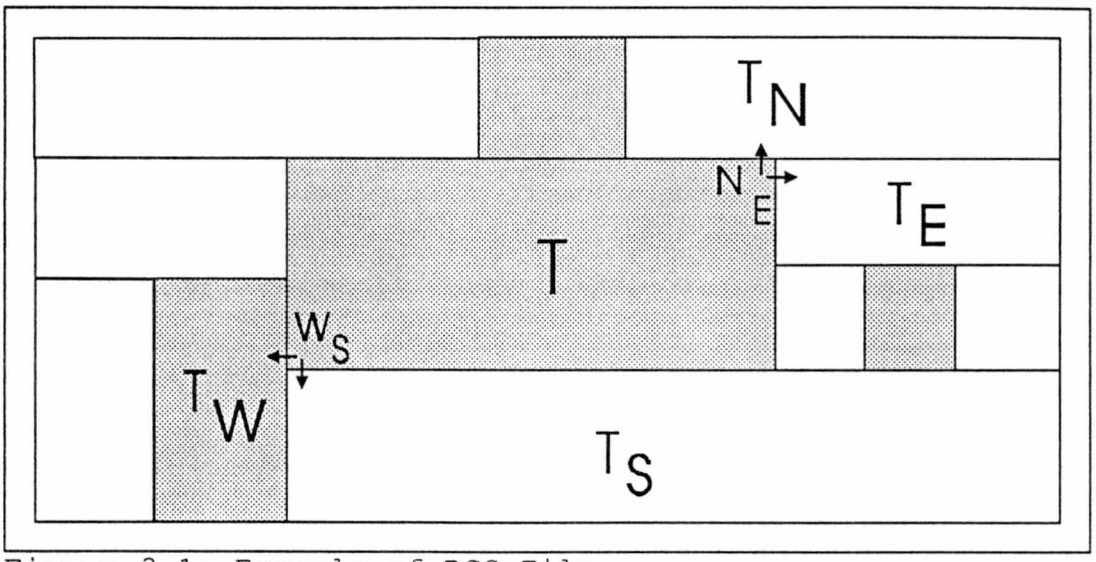


Figure 3-1: Example of RCS Tile

S as shown in Figure 3-1. N (respectively, *E*, *W*, and *S*) points to T_N (respectively, T_E, T_W, and T_S) the rightmost (respectively, topmost, bottommost, leftmost) tile neighboring its northern (respectively, eastern, western, and southern) boundary. The *x* and *y*-coordinates of the top right corner of *T* are identical to $T_E.x_1$ and $T_N.y_1$, respectively, and can easily be obtained by using *T.N* and *T.E*. This is why the coordinates of the top right corner of the tile are not stored in the node representing the tile.

Operations are performed by using the corner stitches. The algorithm for Point Find is presented as an example of how corner stitches are used.

```
TileType PointFind(tile, x, y)
{
1)   Move up or down, using north and south pointers, until
     a tile is found whose y_range contains the desired
     point.
2)   Move left or right, using east and west pointers, until
     a tile is found whose x_range contains the desired
```

point.
3) Since the horizontal motion may introduce a vertical misalignment, steps 1) and 2) may have to be iterated several time to locate the tile containing the point.
}

The other operations are in [5].

This data structure represents a significant improvement over traditional data structures presented in Chapter II. It is faster in practice. This is so because it allows fast, localized algorithms for the atomic operations. A comparison of the corner stitching data structure and traditional data structures may be found in [3]. However, the corner stitching data structure has two disadvantages relative to previous data structures: first, it generally requires more memory; and second, it is harder to implement.

Chapter IV

L-shaped Corner Stitching (LCS)

This chapter presents the extension to RCS used to store L-shaped tiles. The representation of tiles and the algorithms for the atomic operations are given.

4.1 Introduction

The LCS data structure is used to store solid L-shaped and rectangular tiles. Like the RCS data structures, it also stores vacant space explicitly. Horizontally maximal vacant tiles are formed by extending horizontal lines from some corners of solid tiles so that:

1. All vacant tiles are either rectangles or L-shapes.
2. No two vacant tiles in the layout can be merged to form a vacant rectangle or L-shaped tile.

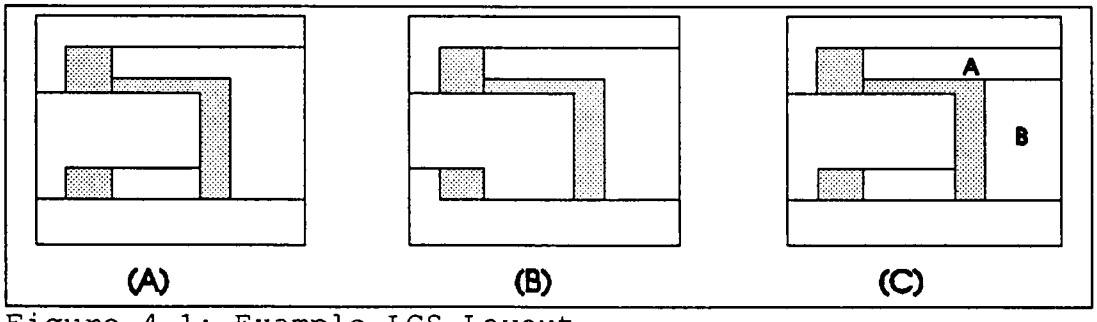


Figure 4-1: Example LCS Layout

Figure 4-1 shows three possible configurations for the same set of solid tiles. Layouts A and B are both valid, but

they are not identical. This shows that, unlike RCS, the LCS data structure does not yield a unique partition for a given set of solid tiles. Notice, also, that layout C is invalid because the vacant tiles A and B can be merged into a vacant L-shaped tile.

4.2 Representation

There exist four possible types of L-shapes. The L-shapes are numbered according to the quadrant represented by the two lines meeting at the single concave corner of the L-shape. For example, in Figure 4-3 (A), the lines represent the first quadrant, so the L-shape type is L1. An L-shape is uniquely defined by (i) its type, (ii) three x-coordinates, X_{min}, X_{mid}, and X_{max}, and (iii) three y-coordinates, Y_{min}, Y_{mid}, and Y_{max}. The pointers required for an L-shape type are obtained as follows:

(1) The L-shape is partitioned into two rectangles R_1, and R_2 by drawing a horizontal line from its concave corner to its longest vertical side (Figure 4-2). These two rectangles are the same rectangles that would be used in RCS to represent the L-shaped tile. (2) Pointers from R_1 to R_2 or R_2 to R_1 are eliminated; pointers that do not correspond to a corner of the L-shape are also eliminated. The eliminated pointers are represented by the broken arrows in Figure 4-2. (3) The remaining pointers (represented by the solid arrows) are named according to the following convention: if there is

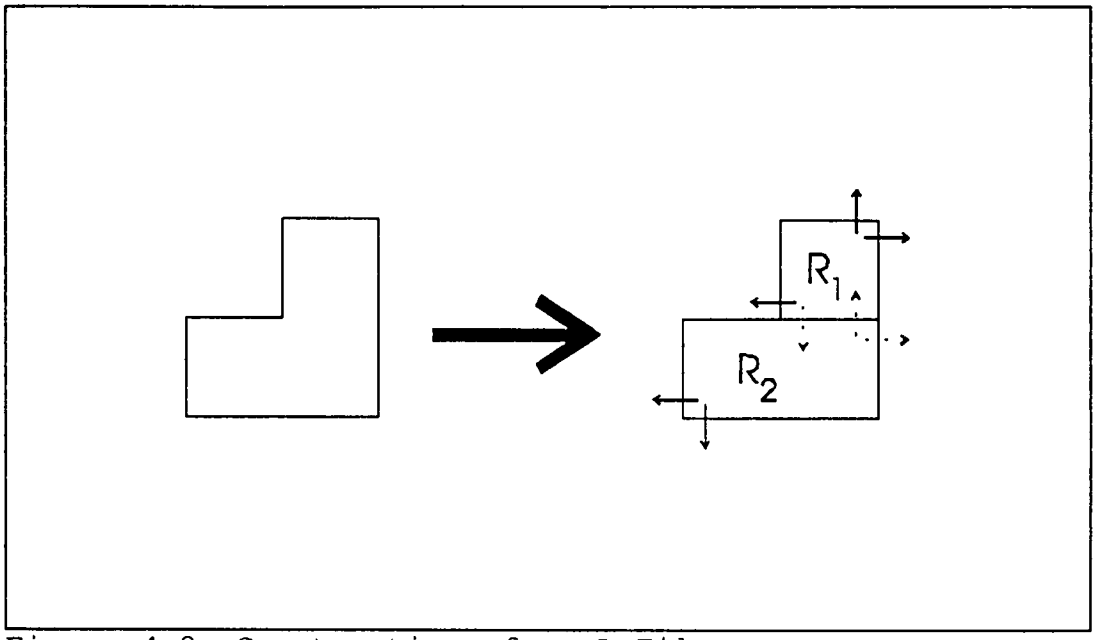


Figure 4-2: Construction of an L Tile

exactly one pointer in a given direction (e.g., E in Figure 4-3 (B)) then that pointer retains its name from RCS. If there is a pointer corresponding to the concave corner of the L-shape, it takes on the name of the quadrant to which it points(e.g., the NW pointer in Figure 4-3 (B)). If there are two pointers in a given direction (excluding concave-corner pointers), then one retains its RCS name while the other receives the subscript s, (e.g., there are two pointers in the eastern direction in Figure 4-3 (A); one pointer is called E, the other E_s). The pointer without the subscript is said to be the primary pointer and the other pointer is a secondary pointer. The choice of primary and secondary pointers is not arbitrary and the explanation is given in the neighbor find operation.

As with the RCS data structure, we do not require all six coordinates that define an L-shape to be explicitly

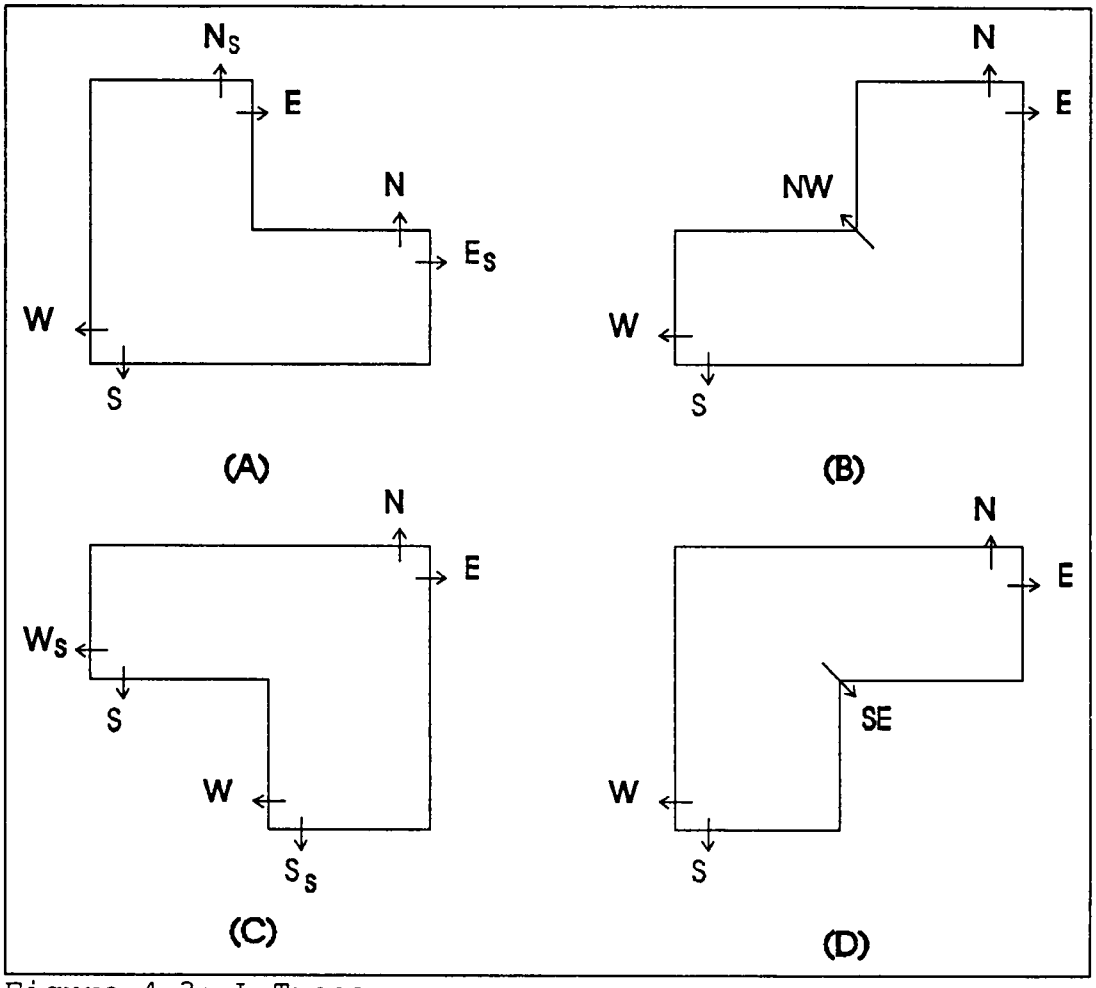


Figure 4-3: L Types

stored in the node representing it in the LCS data structure. The coordinates that are to be explicitly stored by an L-shape are obtained as follows: each S pointer has a y-coordinate associated with it, and each W pointer has an x-coordinate associated with it. Combined pointers also have x (y) coordinates associated with them if they have S (W) components. Thus, an L1 tile explicitly stores X_{min} and Y_{min}

corresponding to W and S, respectively; an L2 tile stores X_{min}, Y_{min}, X_{mid}, corresponding to W, S, and NW, respectively; an L3 tile stores X_{min}, X_{mid}, Y_{min}, and Y_{mid}, corresponding to W, W_s, S, and S_s, respectively; an L4 tile stores X_{min}, Y_{min}, and Y_{mid}, corresponding to W, S, and SE, respectively. A rectangle (R) stores X_{min} and Y_{min} corresponding to W and S. However, unlike the RCS data structure, the remaining coordinates defining a tile cannot be determined solely from the N and E pointers. This is due to the ambiguity illustrated in Figure 4-4. Tile *U* is the tile pointed to by *T*.N. In Figure 4-4 (A) $Y_{max}(T) = Y_{mid}(U)$, while in Figure 4-4 (B) $Y_{max}(T) = Y_{min}(U)$. Additional information is required to choose the correct coordinate of *U*. This is done by associating a bit with each N pointer,E pointer or combined pointer with a N or E component. For an N (E) pointer or

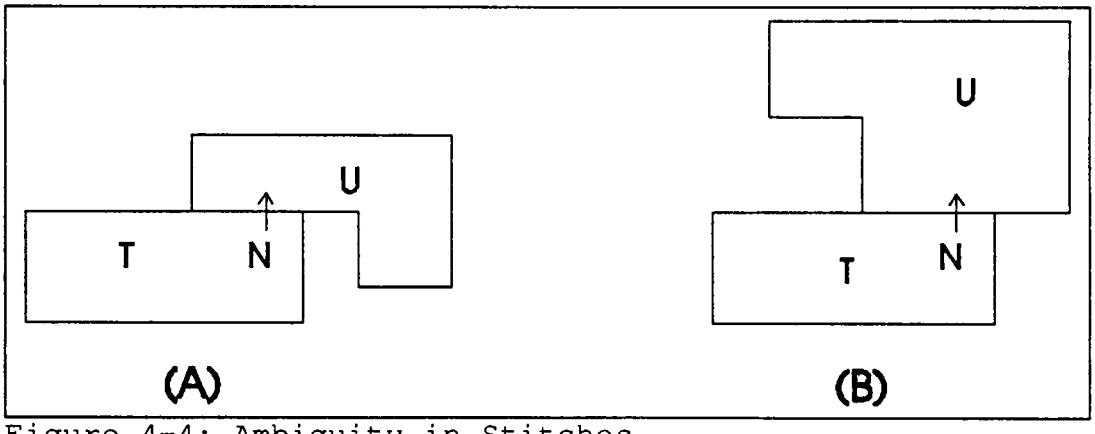


Figure 4-4: Ambiguity in Stitches

combined pointer component, this bit indicates whether the pointer to the tile enters it from its lowest (leftmost) edge or from its middle edge. An *edge* is defined as a single

line segment of the perimeter of the tile. Tile *U* has two south edges along its south side. A tile's south *side* contains all its south edges.

4.3 Space Comparison

Using the definitions of the RCS and LCS data structures we obtain Table 4-1. which is a table describing the contents of L-shapes (L1, L2, L3, L4) and rectangles (R) in LCS and rectangles in RCS. The actual memory requirements of nodes in bytes (last column of the table) are obtained by assuming that pointers and coordinates, each, require 4 bytes of storage, and by placing all remaining bits into a single byte. Note that the space required by any L-shape is less that the space required by the corresponding two rectangles in RCS and that the space required by a rectangle in LCS is equal to the space required by a rectangle in RCS.

Table 4-1 Space requirements of tiles in RCS and LCS

Tile type	Number of Coordinates	Number of Pointers	Number of N/E bits	Solid/ Vacant tile bit	R/L bit	L type bits	Number of bytes
L1	2	6	4	1	1	2	33
L2	3	5	3	1	1	2	33
L3	4	6	2	1	1	2	41
L4	3	5	3	1	1	2	33
R in LCS	2	4	2	1	1	0	25
R in RCS	2	4	0	1	0	0	25

4.4 Neighbor Find

The neighbor find operation finds all tiles that touch a given side of a tile T. To find all neighbors of the south

side of T, the algorithm travels T's south pointer. Next it follows east pointers until the X-coordinate of the east pointer is to the right of $T.X_{max}$ of the original tile. Since an L-shaped tile may have two east pointers, the algorithm must specify with one to follow. The first move through an east pointer could be through either of the two east pointers. All subsequent moves traverse the upper of the two east pointers. We have defined the primary east pointer of a tile to be the uppermost east pointer of the tile. The algorithm then simply follows primary east pointers until all neighbors have been found. All tiles traveled through are south neighbors of T. Finding neighbors of the other sides is symmetric.

```
Neighbor_Find_South(tile)
{
For each south edge of tile
     Do a Neighbor_Find_South_Edge(southern pointer of edge,
          x-coordinate of right end point of edge,
          y-coordinate of southern pointer)
}

Neighbor_Find_South_Edge(tile, stop_x, enter_y)
{
if(enter_y not equal to Ymin(tile))
     Output tile;
     follow tile's East_Sec pointer;
while(x-coordinate of current tile's East pointer is less
than stop_x)
     Output current tile;
     follow tile's East pointer;
Output current tile;
}
```

In Figure 4-5, the south neighbors of tile *T* are being found. First the algorithm moves south to tile *A*. It then

checks to see if T's south pointer entered A from the top north edge L or the bottom north edge M. Since T entered A through M, the algorithm chooses $A.E_s$ to B. Now the

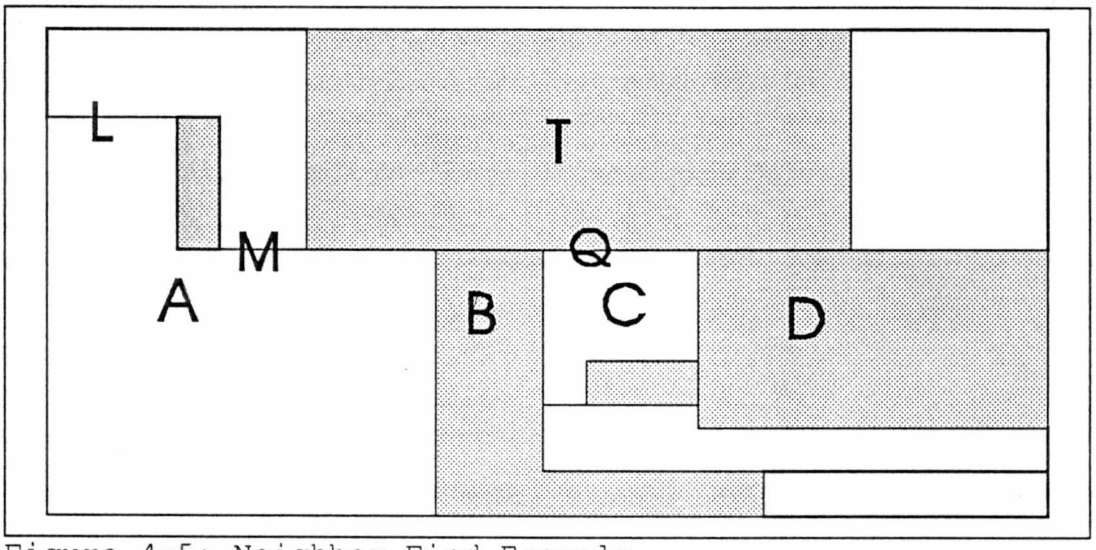


Figure 4-5: Neighbor Find Example

algorithm simply follows B, C, and D's primary east pointers until the x-coordinate of D's east pointer is larger than the right end point of T's south edge Q.

4.5 Point Find

Point find searches the data structure for the tile containing the point (x,y) starting from any given tile. The algorithm is similar to the point find algorithm for RCS except that when the LCS algorithm changes direction it may have a choice of two pointers in the direction it wants to travel. The LCS algorithm chooses the pointer whose *subrange* is closer to the desired point. A subrange of a pointer is the range of values of the edge through which the pointer

extends. For example, the subrange of the N_s pointer in Figure 4-3 (A) is $[X_{min}, X_{mid})$ while the subrange of N is $[X_{mid}, X_{max})$.

If a layout has tiles of relatively uniform size, the number of tiles intersecting a horizontal or vertical line is approximately $N^{1/2}$. Like the time complexity for RCS, the complexity for LCS is also $O(N^{1/2})$. The worst case is O(N). This occurs when all tiles are in a single column or row.

```
TileType PointFind(tile, x, y)
{
current = tile
while ((x,y) not in current)
      {
      1. current = tile pointed to by east or west
         pointer in the direction of x whose y_subrange is
         closer to y.
      2. while (the x-coordinate of the primary pointer does
                not cross x)
           current = the tile pointed to by the primary east
                     or west pointer in the direction chosen
                     in step 1.
      3. if((x,y) not contained in current)
           current = the tile pointed to by the northern or
                     southern pointer in the direction of y
                     whose x_subrange contains x.
      4. while(the y-coordinate of the primary pointer does
                not cross y)
           current = the tile pointed to by the primary north
                     or south pointer in the direction chosen
                     in step 3.
      }
return current
}
```

Given the example layout in Figure 4-6, the algorithm searches for the point represented by the intersection of the dashed lines. If the algorithm were to start at tile 1, it would follow tile 1's NW pointer to reach tile 2 because the NW's y_subrange is closer to y. Then the path of tiles 2

through 5 is obtained by following the primary west pointers of each tile. When the algorithm reaches tile 5, it finds that x is contained in the x_subrange associated with the N_s pointer so the algorithm travels it to reach tile 6. After reaching tile 6, the algorithm follows the primary north pointers to tile 9. Notice that the algorithm travels from 6 to 8 through 7, when it would have taken less steps to go

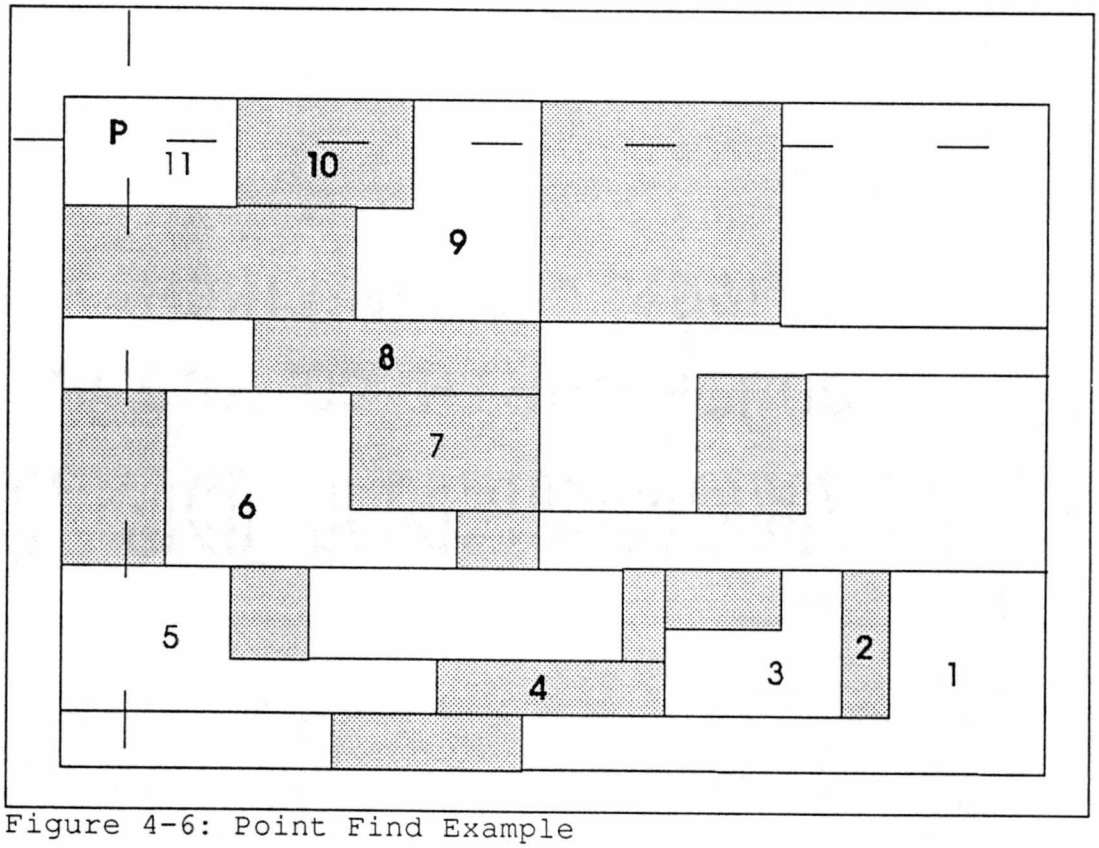


Figure 4-6: Point Find Example

directly to 8. This is because the algorithm only evaluates which pointer would be better when the algorithm changes direction. The algorithm travels primary pointers only, saving the time to do the evaluation every time. After reaching tile 9, the NW pointer is followed to tile 10.

After traveling 10's West pointer to tile 11 the algorithm stops.

When the algorithm crosses an axis (the vertical and horizontal dashed lines in Figure 4-6) it changes directions. When the algorithm changes direction, it follows the pointer whose subrange contains the dashed line that was crossed. So if the found tile has only one pointer in the direction of P, that pointer is chosen. However, if the tile has more than one pointer in the direction of P,(i.e. it is a L-shaped tile), the L shape is visualized as its two component rectangles and the pointer corresponding to the component rectangle that contains the axis is chosen.

Lemma 1

(a) Traveling south from a tile T on the positive portion of the y axis will stop at a tile whose south pointer is to the left of the y axis.

(b) Traveling north from a tile T on the negative y axis will stop at a tile whose north pointer is to the right of the y axis.

(c) Traveling west from a tile on the positive x axis will stop at a tile whose west pointer is below the x axis.

(d) Traveling east from a tile on the negative x axis will stop at a tile whose east pointer is above the x axis.

Proof

Since T lies on the y-axis, the south pointer of T whose x-subrange contains 0 is followed. Since the south pointer will lie at the leftmost point of the x-subrange, its x-coordinate must be less than or equal to 0. In subsequent steps, primary south pointers are followed until the x-axis is reached. Suppose that we are currently at tile P and that P's primary south pointer points to Q. From the definition of corner stitching pointers, the x-coordinate of Q's primary south pointer is less than or equal to that of P's primary south pointer. By transitivity, the x-coordinate of primary south point of the last tile in the sequence is less than or equal to that of T which is less than or equal to 0. The proofs for (b), (c), and (d) are similar.

Theorem 1 Point Find Algorithm converges

Proof

Steps 1 and 2 of iteration i of the while loop in Algorithm Point find ensure the current tile lies on the y-axis. Without loss of generality, assume that this tile lies on the negative y-axis. Let this tile be n_1. From Lemma 1 (b), steps 3 and 4 of the algorithm traverse a sequence of tiles $n_1..n_i$ such that n_i lies on the positive x-axis. Next, from lemma 1 (c), steps 1 and 2 on the $(i+1)$th iteration of the while loop traverse a sequence of tiles $w_1..w_j$, where w_j lies on the negative y-axis (Note $w_1 = n_i$). To prove that the

algorithm converges, we must show that w_j lies above n_1 on the y-axis. This would imply that the tile reached after steps 1 and 2 of iteration i+1 is above the tile reached at the same point in the previous iteration. Since there are a finite number of tiles above n_1 on the negative y-axis, the algorithm is guaranteed to converge.

Prove w_j lies above n_1:

To prove w_j lies above n_1, it is sufficient to show w_j can not reach n_1 and therefore any tile below n_1. Given a sequence n_1 to w_j, for w_j to be n_1, w_{j-1} must lie along an east edge of n_1. There exists two cases where this can happen. Case A is where n_2 is w_{j-1} and case B is where n_2 is above w_{j-1}.

In case A:

See Figure 4-7 (A), n_2 is below n_3 and to the west of w_{j-2}.

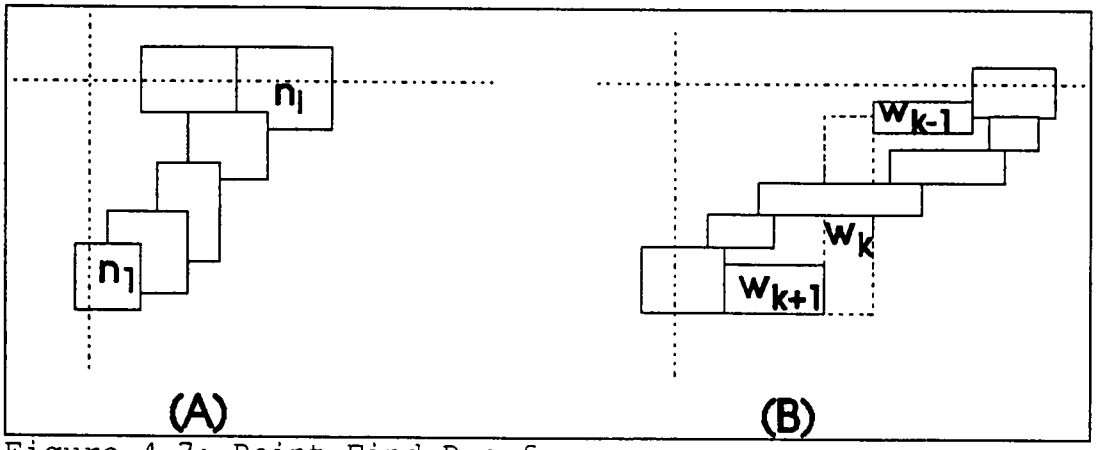


Figure 4-7: Point Find Proof

The same two cases still exist. therefore case B must exist at some point n the sequence or n_2 to n_1 are all L3 shapes. However the west pointer chosen would not be the one that points to n_{1-1}. Therefore case A does not lead to sequence where n_1 is w_j.

In case B:

See Figure 4-7 (B), there is a tile w_k where w_{k-1} is above the sequence n and w_{k+1} is below the sequence n. Since n is continuous and all points only exists in only one tile w_k can not exist.

Since w_j can not be n_1 and w_j can not be below n_1, w_j must be above n_1.

4.6 Area Search

Area search is used to see if a given area is free of any solid tiles. The algorithm relies on the fact that vacant tiles are horizontally maximal; i.e. the east and west neighbors of a vacant tile must be solid. For a given area A, the algorithm begins by moving to the tile Q_0 containing the north west corner of A. If Q_0 is solid, Area Search returns FALSE. Next it sees if the north east corner of A is contained in Q_0 and returns FALSE if it is not. Since Q_0 is a vacant tile, its east neighbors must be solid. Therefore if A includes some amount of area east of Q_0, A must include a solid tile. If Q_0 does not contain the south

edge of A being searched then the east and west edges of A must both leave Q_0 through the same edge otherwise the search fails. The algorithm moves down A's west edge until it reaches A's southwest corner. Let $Q_0..Q_k$ be the tiles from top to bottom that lie on A's west edge. The algorithm moves from Q_i to Q_{i+1} by following the south pointer of Q_i whose subrange contains A's west edge. This could lead to a tile that no longer contains A's west edge. To get back to the tile beneath Q_i that contains A's west edge (Q_{i+1}), we move eastward along Q_i's south edge until we reach Q_{i+1}. This is accomplished by a process that is similar to Neighbor find.

For the algorithm to continue, each Q_i $0<=i<k$ must satisfy the following properties:

1) Q_i must be vacant.

2) Q_i must contain the east edge of A.

3) If Q_i is an L-shape both the east edge and west edge of A must be on the same side of X_{mid} of Q_i.

If any of these is not satisfied, the algorithm returns FALSE and stops. If Q_k does not contain the south east corner of A, the algorithm returns FALSE and stops. Otherwise, the algorithm returns TRUE.

```
AreaSearch(A)
{
1)  Move to the tile Q0 that contains the north west corner
    of A.
2)  Set i=0.
3)  If Q0 does not contain the north east corner of A
    return FALSE.
2)  While Qi does not contain the south west corner of A do
```

a) **If** Q_i is solid **return** FALSE.
b) **If** east edge of A is outside Q_i **return** FALSE.
c) **If** Q_i is an L shape and the vertical middle edge of Q_i is between the east and west edge of *A* then **return** FALSE.
d) Move to the next tile Q_{i+1} by leaving Q_i through Q_i's south edge that crosses *A*'s west edge and following east pointers until the tile that contains *A*'s west edge is reached.
e) Increment *i*.

3) **If** Q_i does not contain *A*'s south east corner **return** FALSE.
4) **Return** TRUE.
}

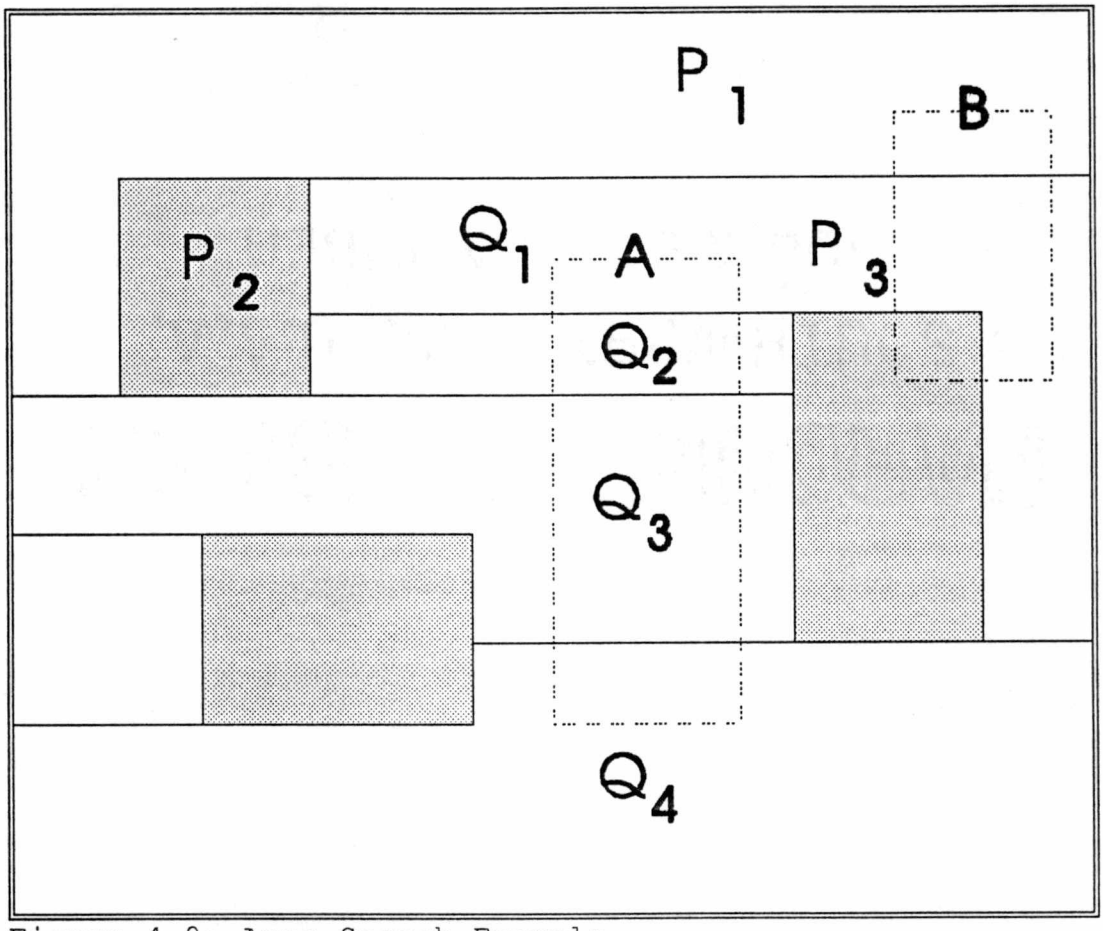


Figure 4-8: Area Search Example

Two example area searches are shown in Figure 4-8. Area *A* returns TRUE. It first moves to Q_1. Since Q_1 contains *A*'s north east corner the algorithm checks if the east and west

edge of A leave Q_1 though the same south edge. Next the algorithm moves to Q_2 and finds that the east edge of A is to the left of Q_2's east edge. Next Q_3 is checked and finally Q_4 is checked. Since Q_4 contains the south east corner of A the search returns TRUE. Area B is a failed search. The algorithm moves to P_1. Since the east edge of B is to the east of the east edge of P_1 and both of B's vertical edges leave P_1 through the same edge the algorithm continues. The algorithm reaches P_3 in a similar manner as neighbor find by traversing P_1's SE pointer to P_2 and P_2's E pointer to P_3. B's east edge is to the left of P_3's east edge but B's west edge is to the left of P_3's middle vertical edge and B's east edge is to the right of P_3's middle vertical edge. The search then returns FALSE.

4.5 Insert

The insert operation inserts a solid tile into the layout. It accomplishes this by a sequence of split and merge operations. The split operation breaks a tile into two tiles along either a vertical or a horizontal line. The merge operation combines two tiles to form a rectangular or L-shaped tile. A merge will also combine 2 L-shaped tiles if the merge can result in 1 L-shaped tile and 1 rectangular tile, since it takes less memory to store 1 L-shape and 1 rectangle than 2 L-shaped tiles.

An L-shaped tile is inserted into the layout by first

individually inserting the two component rectangles that form the L-shape and then merging the two rectangles. The remainder of the section discusses the insertion of a rectangle into the layout.

```
Insert_Block(x1, y1, x2, y2)
{
1)   If (!AreaSearch(x1, y1, x2, y2)) return.
2)   Let i=0.
     Split Q_i, the tile containing the north edge of the
     rectangle to be inserted, into two tiles along the line
     y=y2. Let T be the upper tile and Q_i be the lower tile.
3)   While (Q_i does not contain the south edge of the
     rectangle to be inserted)
       a)  Split the tile along x1, let L_i be the new tile on
           the left of x1.
       b)  Split the tile Q_i along x2, let R_i be the new tile
           on the right of x2.
       c)  Try to merge R_i-1 and L_i-1 with neighboring tiles.
       d)  Merge Q_i and Q_i-1.
       e)  i++.
       f)  Let Q_i be the tile beneath Q_i-1.
4)   Split Q_i along y1 and let B be the new tile below y1.
     Split Q_i into R_i, Q_i, L_i by the lines x=x1 and x=x2.
5)   Merge Q_i and Q_i-1.
6)   Merge R_i and L_i, R_i-1, L_i-1 with neighboring tiles.
7)   Merge T and B with any surrounding tiles.
}
```

Figure 4-9 shows the various steps involved in the insertion of a rectangle Q (represented by dashed lines in Figure 4-9 (A)) into the layout. The coordinates of the bottom left and top right corners of the rectangle area (*x1*, *y1*) and (*x2*, *y2*), respectively. First, the area that will be occupied by the rectangle is checked to ensure that there are no solid tiles already present in that area. This is accomplished by calling the Area Search operation.

Next, the tile Q_0 containing the north edge of Q is

split by the horizontal line $y=y2$ into two tiles: T above the split-line and Q_0 below the split-line. Next, Q_0 is split to form L_0, Q_0, R_0. Next Q_1 is split to form R_1, L_1, Q_1. Now R_0 and L_0 are check to see if they can be merged. L_0 merges with T to form an L-shaped tile named T. Q_1 and Q_0 are also merged to form Q_0. Q_2 is split to form R_2, L_2, and Q_2. L_1 is merged with T and R_1 is merged with R_2. Q_2 is also merged with Q_0. Now Q_3 contains the new tile's south edge so Step 3 is done. Q_3 is split to form B and Q_3. Q_3 is split to form R_3, L_3, and Q_3. Q_3 is merged with Q_0 to form the new tile. L_2 and L_3 are merged. R_1 and R_3 are merged. Lastly T and B are checked to see if they can be merged. B is merged to form an L-shape. The new tile Q is now in the layout. Notice that intermediate layouts, Figure 4-9 (B) through (E) are invalid layouts. However the final layout is valid. For a layout to be valid it must satisfy these properties:

1. All vacant tiles must be horizontally maximal.
2. All tiles must be rectangles or L-shapes.
3. There should not exist any pair of vacant tiles V_1 and V_2, such that V_1 and V_2 can be merged to form a rectangle or L-shape.

Theorem 2 The insert operations results in a valid layout.

Proof

The Insert operation only affects the set S of vacant tiles consisting $Q_0, .., Q_k$ and their neighboring vacant tiles. All other vacant tiles in the layout are unchanged

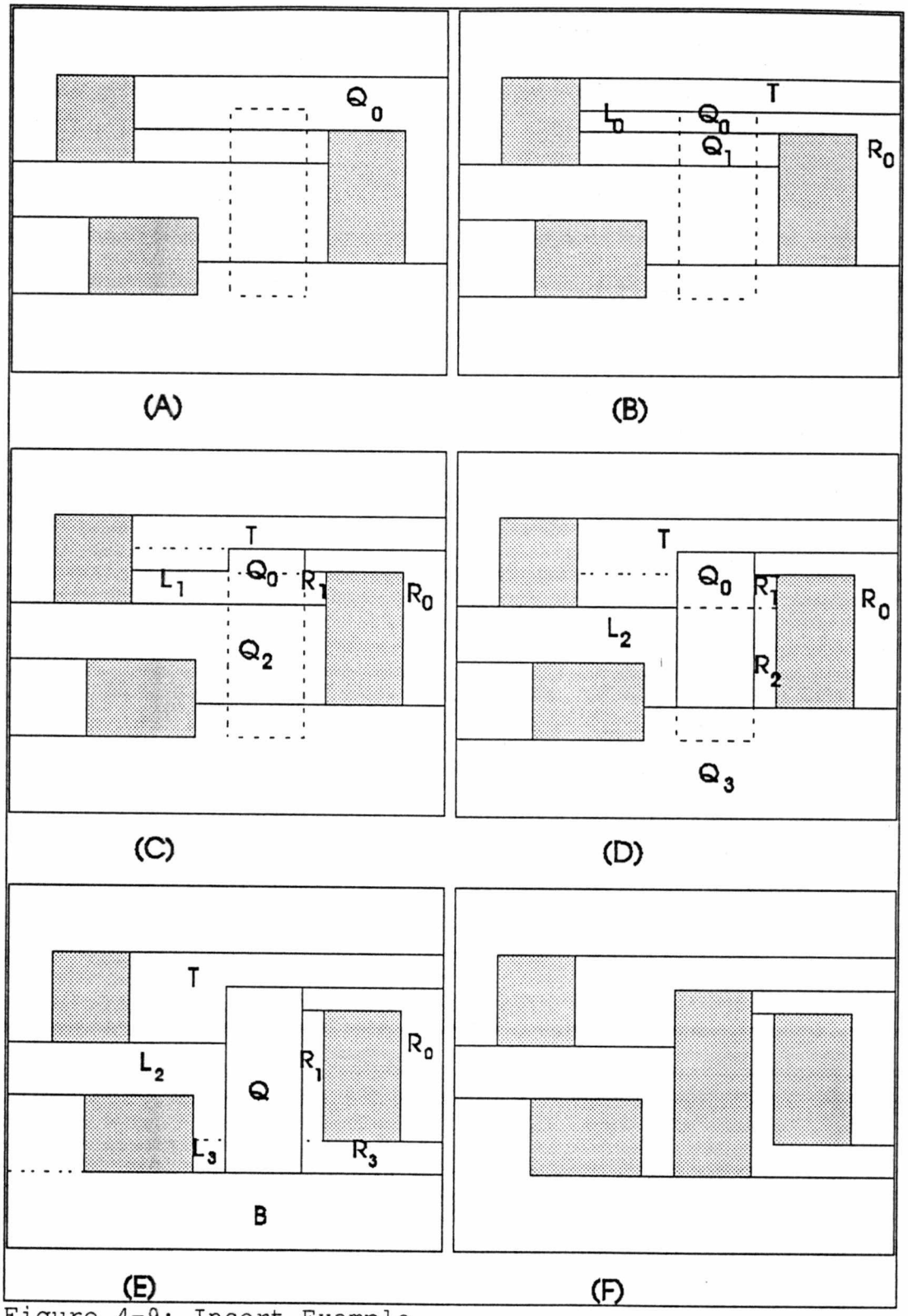


Figure 4-9: Insert Example

and continue to satisfy the properties for a valid layout.

In the remainder of the proof, we will show that the set S' of vacant tiles obtained from set S as a result of the insertion operation satisfies the properties listed above.

1. The only way for the maximality property to be violated is by splitting a vacant tile with a vertical split line. This is only done to tiles $Q_0, .., Q_k$. Furthermore, each vertical split line corresponds to a boundary between a vacant tile and the solid tile being inserted. This does not violate maximality.
2. Splitting an L-shaped or rectangular tile results solely in rectangular and L-shaped tiles. Tiles area only merged if they result in rectangular and L-shaped tiles. Thus, property 2 is satisfied.
3. The systematic mergability check applied to vacant tiles $L_0, .., L_l$, $R_0, .., R_l$, T, and B by the Insert algorithm ensure that the resulting set of tiles S' consists of tiles that cannot be merged with other tiles.

4.7 Delete

Delete removes a solid tile from the layout. The layout after the operation must be valid. All space tiles must be horizontally maximal and can not be merged with a north or south neighbor. An example delete is given in Figure 4-10.

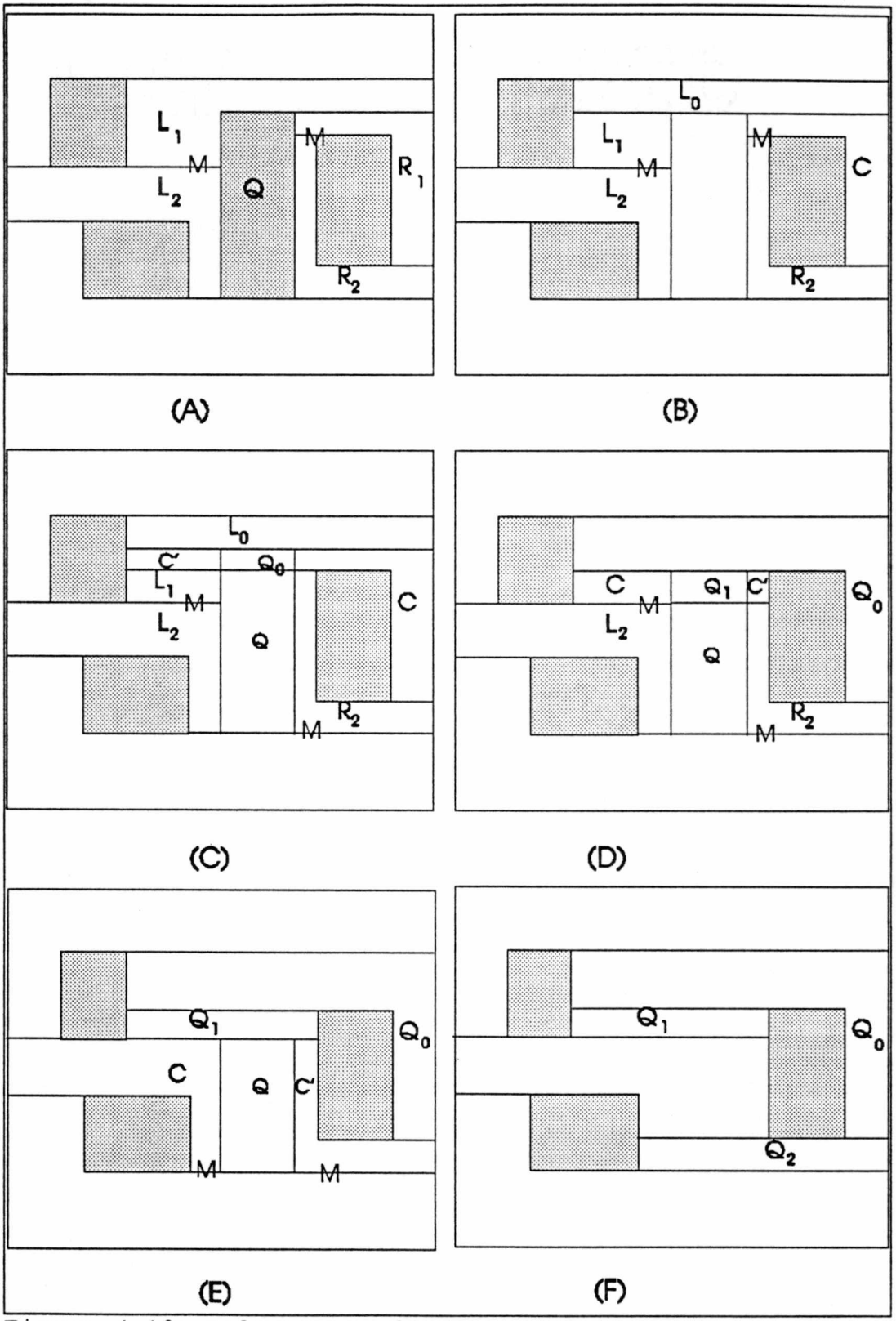


Figure 4-10: Delete Example

```
Delete(tile Q)
{
1)   Create a list of all right and left neighbor R1..Rm and
     L1..Ln.
2)   Split L1 and R1 at Ymax(Q) to get R0 and L0.
3)   Split Rm and Ln at Ymin(Q) to get Rm+1 and Ln+1.
4)   Choose tile C from R and L whose y-coordinate of the
     south edge that touches Q is maximum.
     i=0;
5)   While C
     a)   Split Q and the matching tile C' in the other list
          to obtain C and Q1.
     b)   Merge Q1 and C and C'.
     c)   Merge Q1 north.
     d)   Choose tile C from R and L whose y-coordinate of
          the south edge that touches Q is maximum.
     e)   Increment i;
6)   Check to see if R0, L0, Rm+1, and Ln+1 should be merged.
}
```

In Figure 4-10 (A), tile Q is to be deleted. First tile Q's left and right neighbors are put into a list. Next L_1 is split at $Y_{max}(Q)$. Next R_1 is chosen as C since R_1's M edge is above L_1's M edge. Q and L_1 are split at the Y-coordinate of R_1's M edge (Figure 4-10 (C)). C, C' and Q_0 are then merged. Then Q_0 is merged north to L_1. L_1 is chosen next and Q and R_2 are split at L_1's M edge (Figure 4-10 (D)). C, Q_1, and C' are merged to form Q_1. Next L_2 is chosen as C. However R_2 could also have been chosen since L_2's M edge has the same y-coordinate as R_2's M edge. Q and R_2 are split at L_2's M edge but this does not change the layout since Q and R_2 already have an edge there. Next C, Q, and C' are merged. To merge the three tiles into a valid layout, all three tiles must be split at $Y_{mid}(C')$ so that an L-shape and a rectangle are formed by the merge. Lastly L_0 is checked to see if it can be merged with an adjacent tile.

4.8 Directed Area Enumeration

This operation orders all tiles that intersect a given area. This ordering has a direction such as left-to-right. For this ordering a tile T appears in the ordering only after all tiles that are to the left of T have been placed in the ordering. This ordering is done so that some operation may be performed on the tile. An example operation would be to print all tiles that have been enumerated.

```
Enumerate_Area_And_Print(A)
{
1)   Use the pointfind algorithm to find Q0 that contains
     the upper left corner of A.
2)   Search along the west side of A.
3)   For each tile Qi found in steps 1 and 2
     a)   If Qi is a L3 and the middle vertical edge of Qi
          intersects A skip b) else Print the tile Qi.
     b)   For tile Qi call EvaluateEast(Qi).
}
PrintAndProceed(Q)
{
1)   Print the tile Q.
2)   For tile Q call EvaluateEast(Q).
}
EvaluateEast(Q)
{
1)   If the right side of Q is outside A then return.
2)   For each right edge Ej of Qi inside A
     a)   Locate all tiles in A touching Ej.
     b)   For each tile Pk found in the previous step do
          1)   If any of the following four conditions are
               true
               i)   If Pk is a L3 and Pk's W pointer points
                    to Ej.
               ii)  If Pk is a L3 and the south edge of A
                    cuts Ej above the middle horizontal edge
                    of Pk or Pk's middle vertical edge is
                    outside A.
               iii) If Pk is not a L3 and Pk's W pointer
                    points to Ej.
               iv)  If Pk is not a L3 and the south edge of
                    A cuts Ej.
               Call PrintAndProceed(Pk).
}
```

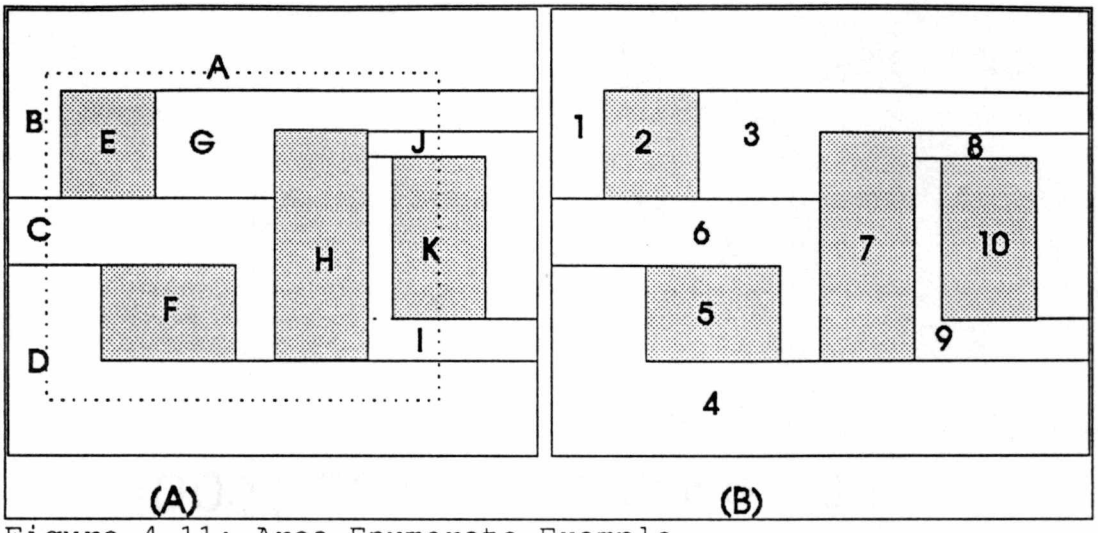


Figure 4-11: Area Enumerate Example

An example of Area Enumerate is given in Figure 4-11. Step 1 of the algorithm moves to tile *B*. A search along the west edge of the area *A*, puts *B*, *C*, and *D* on the list. First tile *B* is examined. Since tile *B* is not an L3, it is printed and given the number 1. Next all tiles along its one east edge in *A* are put on a list. This is only tile *E*. Condition iii) is meet since *E*'s W pointer points to the edge of *B* that is being enumerated PrintAndProceed is called with *E*. *E* is printed and given the number 2. Next the east edge of *E* is searched and its neighbor *G* is put on a list. Condition iii) is meet by *G* so PrintAndProceed is called with *G*. *G* is printed and given the number 3. Next *G*'s east neighbor is put on a list. Since *H*'s W pointer does not point to *G*'s east edge. The recursion now drops to the first level. Since *C* is an L3 and its vertical middle edge is in *A* the algorithm skips step b). Next *D* is examined. *D* is printed

and given the number 4. *D*'s east neighbor *F* is found. Since *F*'s W pointer points to *D*'s east edge PrintAndProceed is called with *F*. *F* is printed and given the number 5. Next *F*'s east neighbor *C* is examined. Since *C*'s W pointer points to *F*'s east edge and *C* is an L3 condition i) is meet and PrintAndProceed is called with *C*. *C* is printed and given the number 6. Next *C*'s east neighbor *H* is examined. *H*'s W pointer points to *C*'s east edge. Therefore PrintAndProceed is called with *H*. *H* is given the number 7. Next the neighbors of *H* are found. First *J* is examined and since its middle vertical edge is outside *A* PrintAndProceed is called with *J* and *J* is given number 8. Lastly *I* and *K* are enumerated.

Chapter V

Experimental Results

And Conclusions

This chapter presents the experimental comparison of LCS and RCS. The results are given in tables. Lastly conclusions from the results are presented along with some areas of further research.

5.1 Experimental Results

The LCS and RCS data structures were implemented in ANSI C to provide maximum portability between platforms. The results of this section were obtained on a SUN SPARC station 10.

Four layouts were used to compare the data structures. Each layout consisted solely of rectangular tiles. This favors RCS because the overhead of breaking solid L-shaped tiles into rectangles and maintaining these rectangles so that they can be distinguished form other rectangles is avoided. Thus, the results presented here represent a pessimistic evaluation of the LCS data structure.

Layout A is a randomly generated layout with 10,000 tiles. A tile is a rectangle between 25 and 150 units in height and width. Each tile has a 75% chance of touching another tile. Layout B is another randomly generated layout

with 3,001 tiles. A solid tile is between 25 and 150 units in width and between 100 and 250 units in height. Each tile has a 75% chance of touching another tile. Layouts C and D are from an actual circuit provided by Professor Naveed Sherwani of Western Michigan University. Each layout represents a different layer in the circuit. Layout C contains 907 solid tiles. Layout D has 387 tiles.

For each layout, the experiment consisted of inserting all tiles into the each data structure, and then deleting all the tiles. The times required for inserting and deleting the layout were recorded. The memory required for each layout was also recorded. Since insert and delete call operations point find, neighbor find, and area search, these operations are not separately evaluated. The results of our experiments are tabulated in Tables 5-1, 5-2, 5-3, and 5-4 corresponding to the four layouts. All times are measured in cpu seconds. Memory is measured in bytes. In the comparison row, positive percentages mean that LCS used less memory or ran faster than RCS. A negative percentage means LCS used more memory or ran slower. Thus positive percentages signify better performance by LCS.

The results show that LCS consistently saves 16-18% in space relative to RCS. As expected the run times of LCS are generally slower, though there are instances where it performs better than RCS. The slower runtimes for LCS can be

Table 5-1 Layout A (10000 tiles)

	Space	Insert Time	Delete Time
RCS	949228	183.08	331.64
LCS	770348	207.11	290.47
Comparison	18.8%	-13.1%	12.4%

Table 5-2 Layout B (3001 tiles)

	Space	Insert Time	Delete Time
RCS	270088	18.48	23.53
LCS	219468	21.48	23.26
Comparison	18.7%	-16.23%	1.1%

Table 5-3 Layout C (907 tiles)

	Space	Insert Time	Delete Time
RCS	61936	1.08	1.09
LCS	52220	1.85	2.05
Comparison	15.7%	-71.3%	-88.1%

Table 5-4 Layout D (387 tiles)

	Space	Insert Time	Delete Time
RCS	28616	.29	.31
LCS	23928	.67	.72
Comparison	16.4%	-131.0%	-132.3%

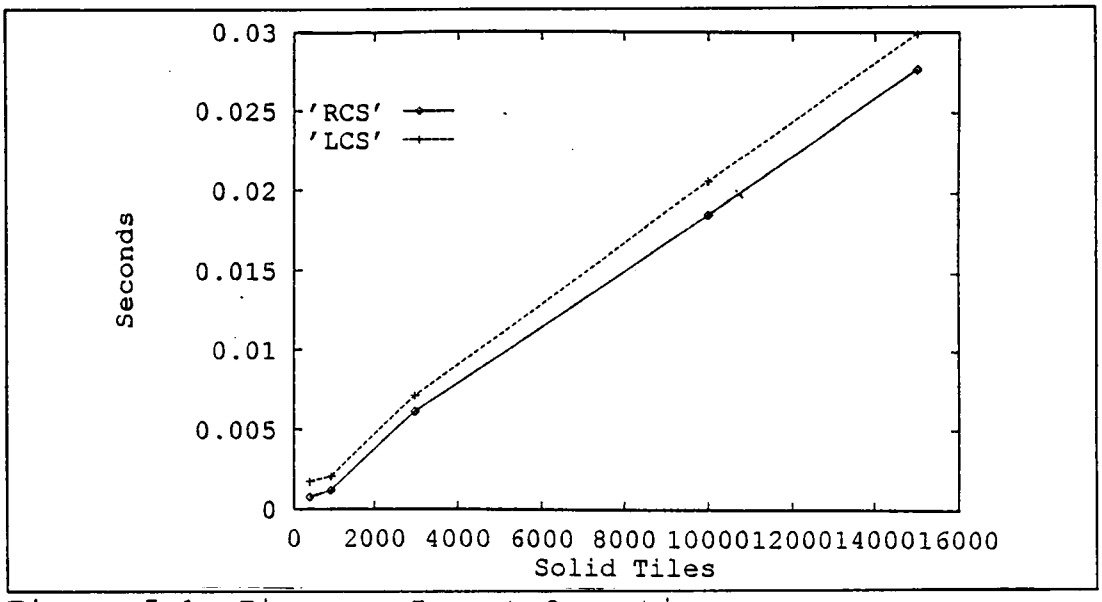


Figure 5-1: Time per Insert Operation

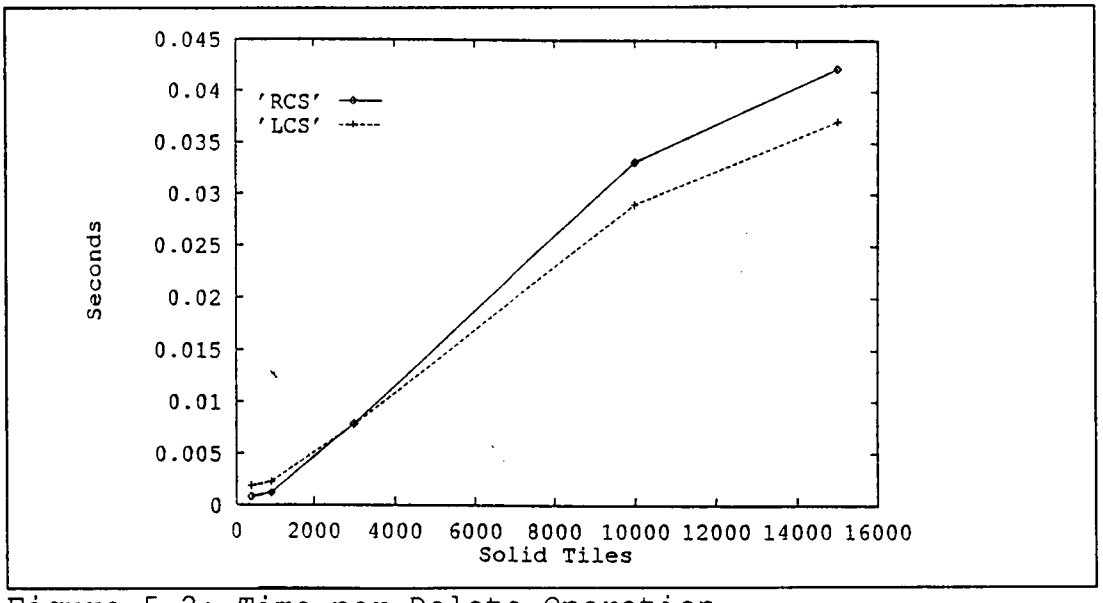


Figure 5-2: Time per Delete Operation

attributed to the added complexity in the implementation of its operations. Figures 5-1 and 5-2 plot the average time per insert or delete.

5.2 Conclusions

LCS has the added ability to handle L-shaped tiles. LCS saves a significant amount of memory compared to the space requirements of RCS. As the number of tiles in a layout increases the LCS data structure's run times approaches the run times of RCS. LCS is useful even if the design still uses only solid rectangles because LCS divides the vacant space more efficiently because LCS uses L-shaped tiles. LCS memory savings would be even greater if solid L-shapes were used. It can be concluded that LCS successfully extends RCS to handle L-shaped tiles. An area of further research would be to extend L-shaped tiles to have trapezoidal and or curved edges in [3, 6]. Also since the division of vacant space in not unique, a method for optimizing this division would help yield even better space savings.

LIST OF REFERENCES

List of References

[1] Cai, Y. and Wong, D. F. "On minimizing the number of L-shaped channels in building-block layout." IEEE Transactions on Computer-aided Design, vol 12, no. 6, pp 757-769, 1993.

[2] Dai, W. M., Asano, T. and Kuh, E. "Routing region definition and ordering scheme for building-block layout." IEEE Transactions on Computer-aided Design, vol. 4, pp 189-197, 1985.

[3] Marple, David, and Smulders, Michiel, and Hegen, Henk. "Tailor: A Layout System Based on Trapezoidal Corner Stitching." IEEE Transactions on Computer-Aided Design Vol. 9, No. 1, 66-90, January 1990.

[4] Mehta, Dinesh. "Estimating the Storage Requirements of the Rectangular and L-shaped Corner Stitching Data Structures." Fourth Great Lakes Symposium on VLSI. pp 33-37. March 1994.

[5] Ousterhout, John K. "Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools." IEEE Transactions on Computer-Aided Design, Vol. 3, No. 1, pp 87-100, January 1984.

[6] Sequin, Carlo H, and Facanha, Helosia da Silva. "Corner-Stitched Tiles with Curved Boundaries." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 12, No. 1, pp 47-58, January 1993.

[7] Shanbhag, A. G., Danda, S. R. and Sherwani, N. "Floorplanning for Mixed Macro Block and Standard Cell Designs." Proceedings of the Fourth Great Lakes Symposium on VLSI. pp 26-29, March 1994.

[8] Sherwani, Naveed. Algorithms for VLSI Physical Design Automation. Norwell, Massachusetts: Kluwer academic Publishers, 1993.

[9] Wang, T. and Wong, D. F. "An optimal algorithm for floorplan area optimization," Proceedings of the 27th Design Automation Conference, pp. 180-186, 1990.

VITA

George Joseph Blust was born on August 7, 1969 in Honolulu, Hawaii. He graduated from Mohawk High School, Sycamore, Ohio in 1988. He received his Bachelor of Science majoring in Computer Science and Math and minoring in Accounting from Ohio Northern University in 1992. In May 1994 he received his Master of Science degree in Computer Science from the University of Tennessee Space Institute.