पुस्तकालय - प्रतिलिपि

[illegible] 3/2/97

A PROJECT REPORT

REPRESENTATION OF SPATIAL DATA USING QUADTREES

Submitted in partial fulfilment of the requirements
for the award of the Degree of

MASTER OF TECHNOLOGY
in
COMPTUER SCIENCE

by
B. RAMANA

Under the Guidence of
Prof. P.C.Saxena

SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110067
JAN. 1997

1430

A PROJECT REPORT

REPRESENTATION OF SPATIAL DATA USING QUADTREES

Submitted in partial fulfilment of the requirements
for the award of the Degree of

MASTER OF TECHNOLOGY
in
COMPTUER SCIENCE

by
B. RAMANA

Under the Guidence of
Prof. P.C.Saxena 77p+ tables

SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110067
JAN. 1997

CERTIFICATE

This is to certify that the dissertation entitled REPRESENTATION OF SPATIAL DATA USING QUADTRESS by B.Ramana in partial fulfilment of the Master of Technology in Computer Science, Jawaharlal Nehru University, New Delhi is a record of bonafide work carried out by him under my guidence.

The result emboidied in this dissertation have not been submitted to any other university or institute for the award of any degree or diploma

[illegible]

Prof. G.V.Singh
Dean
School of Computer & Systems Sciences
Jawharlal Nehru University, New Delhi

[illegible] 6·1·97

Prof. P.C.Saxena (Supervisor)
School of Computer & Systems Sciences
Jawarhal Nehru University, New Delhi

ACKNOWLEDGEMENTS

I am deeply indebted to Prof. P.C. SAXENA for having me an opportunity to work with him. His valuable advice at crucial stages and above all his infinite patience have been chiefly responsible for everything that was done in this project.

I would also like to thank him for enabling me to work in the exciting area of representation of spatial data. I would also like to thank him for allowing me to use the various resources of the department.

I would like to thank to our Prof. C.P. KATTI for his encouragement in doing this project.

I would like to express my thanks to the faculty of school of Computer Science. I would like to thank my friends who adviced me on various matters.

ABSTRACT

Spatial data consists of points, lines, regions, surfaces and volumes. The representation of such data is becoming important in applications suchas computer graphics, CAD, GIS, image processing, pattern recognition and other areas.

Many of the data structures currently used to represent spatial data are hierarchical. They are based on the principle of recursive decomposition (similar to divide and conquer methods). One such data structure is the QUADTREE. Hierarchical datastructures are useful, because of their ability to focus on the interesting subsets of the data. This focussing results in an efficient representation and in improved execution times. Thus they are particular convenient for performing set operarions. These hierarchical data structures are attractive because of their conceptual clarity and ease of implementation.

The most studied quadtree approach to region representation called a region quadtree, is based on the successive subdivion of a bounded image into four equal sized quadrants. This subdivision continues till the blocks contains completely either black or white pixels. This region quadtree is more useful for simple polygons i.e. polygons does not contain holes and intersecting edges. Insertion, deletion, point location and overlay operations on region quadtree are discussed and implemented.

For overcoming the deficiency of region quadtree PM (polygonal map) quadtrees are developed. The PM quadtree family represents regions by only specifying their boundaries without mentioning about their interiors. Based on type of PM quadtree (PM1, PM2, PM3 quadtrees) the polygonal map is repeatedly subdivided into four equal sized square quadrants until the blocks satisfies the stated condition. The PM3 quadtree is more useful for the regions which contains holes. Insertion, deletion, overlay and point location problems are discussed and implemented.

INDEX

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION :

Spatial data consists of points, lines, polygons, regions, surfaces and volumes. The representation of such data is increasingly important in many application areas. Once an application has been specified, it is common for the spatial data types to be more precise. Take an example from Geographical information systems (GIS). In such a case line data are differentiated on the basis of whether the lines are isolated (e.g. earthquake faults), elements of tree like structures (e.g. rivers and their tributaries) or elements of networks (e.g. rail and highway systems). Similarly region data are often in the form of polygons that are isolated (e.g. lakes), adjacent (e.g. nations) or nested (e.g. contours). There are numerous hierarchical data structuring techniques in use for representing spatial data. One commonly used technique is based on recursive decomposition is quadtree.

The term quadtree is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space. They can be differentiated on the following criteria [SAMET8].

1. The type of data they are used to represent.

2. The principle guiding the decomposition process.

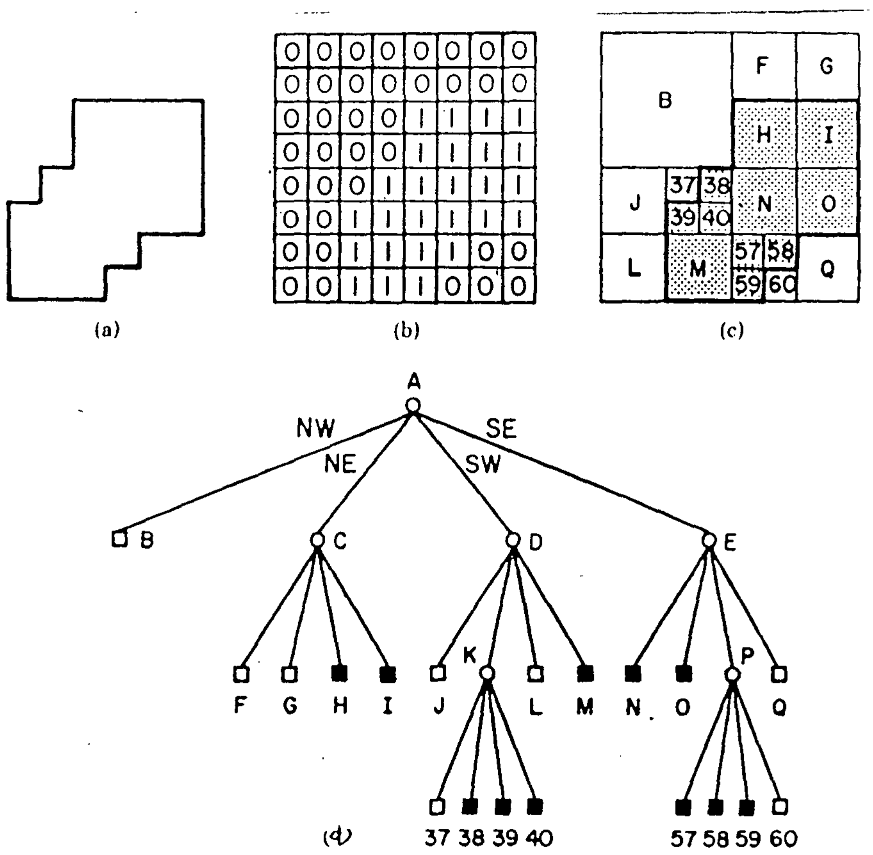
3. The resolution (variable or not).

Currently they are used for point data, areas, curves, surfaces and volumes. The decomposition may be into equal parts on each level (i.e. regular polygons and regular decomposition) or it may be governed by the input. The resolution of the decomposition (i.e., the number of times that the decomposition process is applied) may be fixed beforehand, or it may be governed by the input data. Note that for some applications we can also differentiate the data structures on the basis of whether they specify the boundaries of regions (i.e. curves and surfaces) or organise their interiors (i.e. areas and volumes).

The first example of a quadtree representation of data is concerned with representation of two dimensional binary data. The most studied quadtree approach to region representation called region quadtree (often termed a quadtree) is based on successive sub division of a bounded image array into four equal sized quadrants. If the array does not cover the entirely of 1s or 0s (i.e. region does not cover the entire array) it is sub divided into further quadrants, sub quadrants and so on until blocks are obtained that consists entirely of 0s or entirely of 1s. Thus each block is entirely contained in the region or entirely disjoint from it. The region quadtree can be characterized as a variable resolution data structure.

As an example of region quadtree consider the region shown in figure 1a of next page is represented by $2^3 * 2^3$ binary array in figure 1.b (next page). Observe that 1s correspond to picture elements (pixels) in the region and the 0s correspond to picture elements outside region. The resulting blocks for the array of figure 1.b are shown in figure 1.c of next page. This is represented by a tree of degree 4 (i.e. each non leaf node has four sons).

In the tree representation, the root node corresponds to the entire array. Each son of a node represents a quadrant (labeled in order NW, NE, SW, SE) of the region represented by that node. The leaf nodes of the tree correspond to these blocks for which to further subdivision is necessary. A leaf node is said to be black or white depending on whether its corresponding block is entirely inside (i.e., it contains entirely 1s) or entirely outside the represented region (i.e. it contains only 0s). All non leaf nodes are said to be gray (i.e. it's block contains 0s and 1s). Given a $2^n * 2^n$ image, the root node is said to be at level n while a node at level 0 corresponds to a single pixel in the image. The region quadtree representation for figure 1.a is shown in figure 1.d (next page). The leaf nodes are labeled with numbers and the nonleaf nodes are labeled with letters, The levels of the tree are also marked.



(a)
(b)
(c)
B
F
G
H
I
J
37
38
39
40
N
O
L
M
57
58
59
60
Q
A
NW
NE
SW
SE
C
D
E
K
P
F G H I J L M N O Q
37 38 39 40
57 58 59 60
(d)

FIG.1

At this point, it is appropriate to justify the use of a quadtree decomposition into squares. Of course there are many methods of planer decomposition. Squares are used because the resulting decomposition satisfies the following two properties [SAMET9].

1. It yields a partition that is an infinitely repetitive pattern so that it can be used for image of any size.

2. It yields partition that is infinitely decomposable into increasingly finer patterns (i.e. higher resolution).

A quadtree like decomposition into four equilateral triangles also satisfies these criteria. However, unlike the decomposition into squares, it does not be mapped into each other by translations of the plane that do not involve rotation or reflection. In contrast, a decomposition into hexagons has a uniform orientation, but it does not satisfy property2.

The prime motivation for the development of the quadtree is the desire to reduce the amount of space necessary to store data through the use of aggregation of homogeneous blocks. However a quadtree implementation does have overhead in terms of the non leaf nodes. For an image with B and W black and white nodes respectively, 4*((B+W)/3 nodes required. In contrast, a binary array representation of a 2^n * 2^n image requires only 2^{2n} bits. However this quantity grows quite quickly. Further more if the amount of aggregation is minimal, the quadtree is not very efficient.

1.2 VARIANTS OF QUADTREES :

The array is the most frequently used data structure for representing the images. For large images, however the amount of storage required is often deemed excessive, in a raster representation (i.e. a list of image rows). The image is processed one rows into one dimensional blocks of identically valued pixels (termed a run representation or runlength coding). The image is then represented as a list of such runs.

The region quadtree is a member of a class of representations characterized as being collection of maximal blocks, each of which is contained in a given region and when union is the entire image. In this case blocks are restricted to 1* m rectangles. A more general representation treats the region as a union of maximal square blocks (or blocks of any other desired shape) that may possibly overlap. Usually the blocks are specified by their centers and radii. This representation is called MAT (medial axis transformation). Of course other approaches are also possible (i.e. rectangular coding etc.).

The region quadtree is a variant of on the maximal block representation. It requires the block to be disjoint and have standard sizes (i.e. sides of lengths that are powers of two) and standard locations. The motivation for it's development was a desire the obtain a systematic way to transform the data intro region quadtree, a criterion must be chosen for deciding that an image is homogeneous (i.e. uniform).

The region quadtree is an example of a region representation based on a description of its' interior. One of the variant is representation based on exteriors i.e. boundaries of the regions. This is done in the more general context of hierarchical data structures for complex polygons (polygons containing holes). The emphasis is on regions having linear boundaries (i.e. specified by lines). The data are usually in the form of network of adjacent polygons resulting in a subdivision of space termed a polygonal map. The PM (polygonal map) quadtree is a term used to describe collectively a number of related quadtree like data structures devised by Samet and Webber [SAMET9], Nelson and Samet to overcome some of the problems associated with some other data structures. These some other data structures are based on boundary representation named edge quadtree, line quadtree, and MX quadtree. The disadvantage of the above quadtree data structures are, they are sensitive to shift and rotation.

The PM quadtrees are vertex based and edge based. Their implementations make use of the same basic data structures.

All are built by applying the principle of repeatedly breaking up the collection of vertices and edges (forming the polygonal map) until obtaining a subset that is sufficiently simple that it can be organised by some other data structure. This is achieved by successively

weakning the definition of what contributes a permissible leaf node. The goal is to avoid data degradation when fragments of line segments are subsequently recombined. The result is an exact representation of the lines, but not an approximation.

1.3 OPERATIONS ON IMAGES :

Operations on images is very important aspect in many applications. Without some operations in mind, data structures developed may not appear worthy. Many operations can be performed on images using quadtrees. Some of the operations are point location, overlay, intersection, shifting, rotation and zooming. The region quadtrees and PM quadtrees are useful in determination of the identity of the region i.e., in which a point lies (known as point location problem) and overlay of two maps (i.e. union of two maps). These two operations are discussed and implemented in this project.

1.4 OVERVIEW OF THE PROJECT :

This book is presented as follows. Chapter 1 is introduction. Chapter 2 describes the various types of spatial data structures. Chapter 3 is mainly concerned with the design of the project. Chapter 4 deals with implementation details of project. Chapter 5 is concerned with comparison and analysis of various spatial data structures. Chapter 6 is the conclusions of project, then it follows with bibliography and then follows with appendices. Appendix_A describes about description of the notations of algorithms and Appendix_B shows the results (output) of the project.

CHAPTER 2

VARIOUS SPATIAL DATA STRUCTURES

In this chapter the various spatial data representation techniques are going to be discussed. Generally there are two approaches to the representation of regions. Those that specify the borders of a region and those that organise the interior of a region. This corresponds to either storing region identification information only on the region's border or also storing it on parts of the region's interior. The following techniques are based on the above both approaches.

2.1 ARRAY :

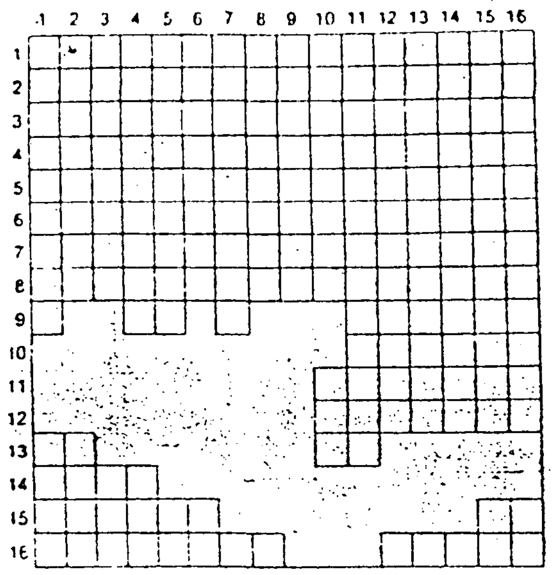
When each cell has a unique value it takes a total of n rows * m columns. For example a grid having 2^n rows and 2^m columns having 2^{m+n} bits (for binary images) required to represent the image. This type of data structure is useful when image consists of chess-board pattern. For the image shown in figure 2.a of next page, it takes the 256 (cells) numbers for complete representation [BURROWS 6].

2.2. CHAIN CODES :

The boundary of the region can be given in terms of origin and a sequence of unit vectors in the cardinal directions. These directions can be numbered (east=0, north=1, west=2, south=3). For example, for the figure shown in 2.a (next page), if we start at cell row = 10, column = 1, the boundary of the region is coded clockwise by

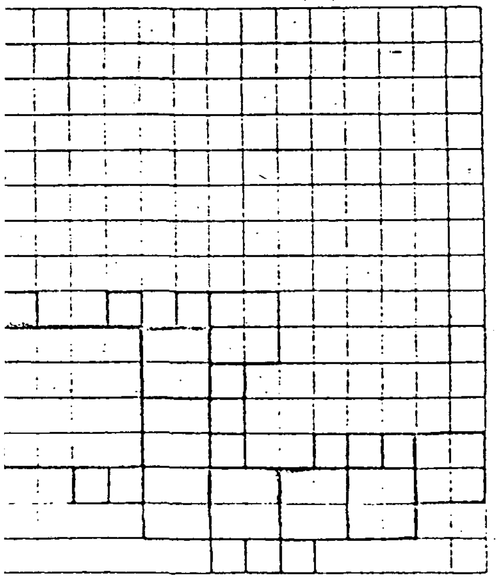
0, 1, 0 (2), 3, 0(2), 1, 0, 3, 0, 1, 0(3), 3(2), 2, 3(3), 0(2), 1, 0(5), 3(2), 2(2), 3, 2(3), 3, 2(3), 3, 2(3), 1, 2(2), 1, (2), 1, 2(2), 1, 2(2), 1 (3).

Where the number of steps in each direction is given by the number in brackets. Chain codes provide a very compact way of storing a region representation and they allow certain

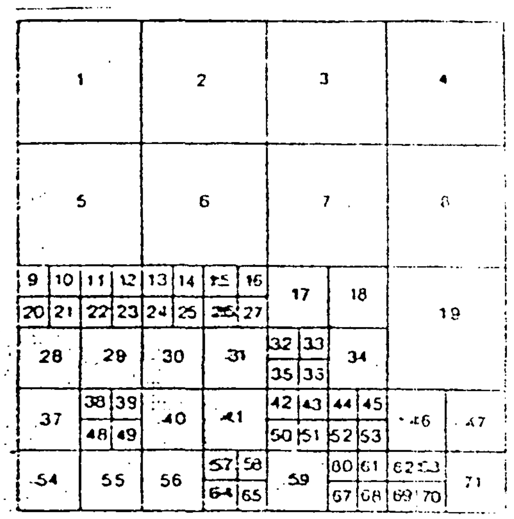


REGION

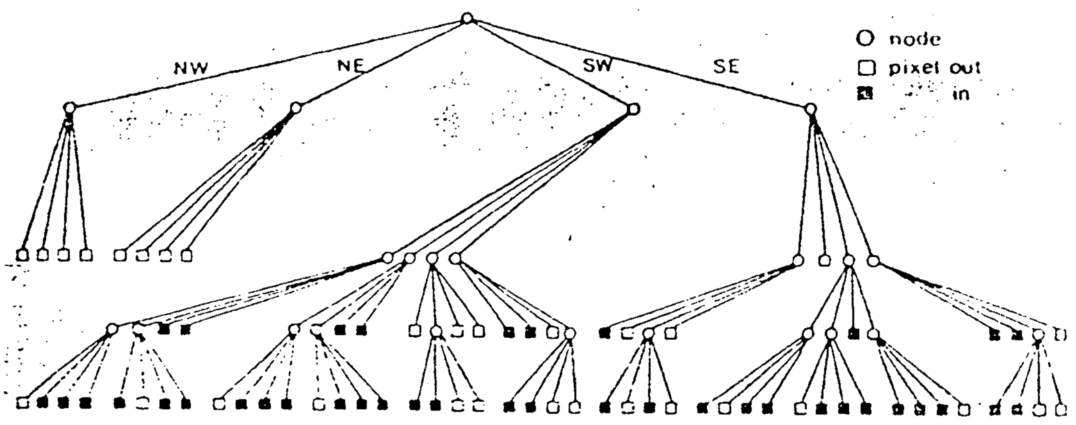
(a)



BLOCK CODES



(c)



(d)

FIG 2

operations such as estimation of areas and perimeters or detection of short turns and concativites to be carried out easily. On the other hand, overlay operations such as union and intersection are difficult to perform without returning to a full grid representation. Another disadvantage is redundancy introduced because all boundaries between regions must be stored twice [burrows 6].

2.3 RUN LENGTH CODES :

Runlength codes allow the points in each mapping unit to be stored per row in terms from lift to right of a begin cell and an end cell. For the area shown in figure 2.a of previous page the codes would be as follows.

Row 9:	2, 3	6, 6	8, 10
Row : 10	1, 10		
Row : 11	1, 9		
Row : 12	1, 9		
Row : 13	3, 9	12, 16	
Row : 14	5, 16		
Row : 15	7, 14		
Row : 16	9, 11		

In this example 69 cells have been completely coded by 29 numbers. thereby effecting a considerable reduction in the space needed to store the data.

Clearly run length coding is a considerable improvement in storage requirements over conventional methods. On the other hand, too much data compression may lead to increased processing requirements during cartographic processing and manipulation. Run length codes are also useful in reducing the volume of data that need to be input to a simple raster data base [BURROWS 6].

2.4 BLOCK CODES :

The idea of run length codes can be extended to two dimensions by using square blocks to file the area to be mapped. Fig 2.b (previous page) shows how this can be done for the map of figure (previous). The data structure consists of just three numbers, the origin (the center or bottom left) and radius of each square. This is called the medial axis transformation (MAT). The region shown can be stored by 17 unit square +9 four squares +1 sixteen square. Given that two coordinates are needed for each square, then the region can be stored using 57 numbers (54 for coordinates and 3 for cell sizes). Clearly the larger the square that can be fitted in any given region and the boundary is simple, then the block coding becomes more efficient. Both run length and block codes are clearly most efficient for large simple shapes and least so far small complicated areas that are only a few times larger than the basic cell. Mat has advantage of performing union and intersection of regions and detecting the properties such as elongation.

2.5 QUADTRESS :

2.5.1. REGION QUADTREE :

This method of more compact representation is based on successive subdivision of $2^n * 2^n$ array into quadrants. A region is filed by subdividing the array step by step into quadrants and noting which quadrants are wholly contained with the region. The lowest limit of division is the single pixel. Figure 2.c. (previous page) shows successive division of one region into quadrant blocks. This entire array of $2^n * 2^n$ is the root node of the tree and height is at most n levels. Each node has four branches, respectively the NW, NE, SW, SE quadrants. Leaf nodes composed to those quadrants for which no further subdivision is necessary. Each node in the quadtree can be represented by two bits, which define whether a node is GRAY or BLACK or WHITE [BURROWS 6].

2.5.2. POINT QUADTREE :

The point quadtree, invented by FINKEL AND BENTLEY is a marriage of the grid method and the binary search tree that results in a tree-like directory with non-uniform sized cells containing one element apiece. The point quadtree shown in figure 3 of next page is implemented as a multidimensional generalisation of a binary search tree. It is referred to as a point quadtree where confusion with a region quadtree is possible. Point quadtree shape is highly dependent on the order in which the points are added to it [SAMET 8].

2.5.3. MX QUADTREE :

The MX quadtree is organised in a similar way to the region quadtree. The difference is that the leaf nodes are BLACK or WHITE corresponding to the presence or absence, respectively, of a data point in the appropriate position in the matrix. The MX quadrants, each data point as if it is a black pixel in region quadtree. An alternative characterisation of MX quadtree is to think of the data points as

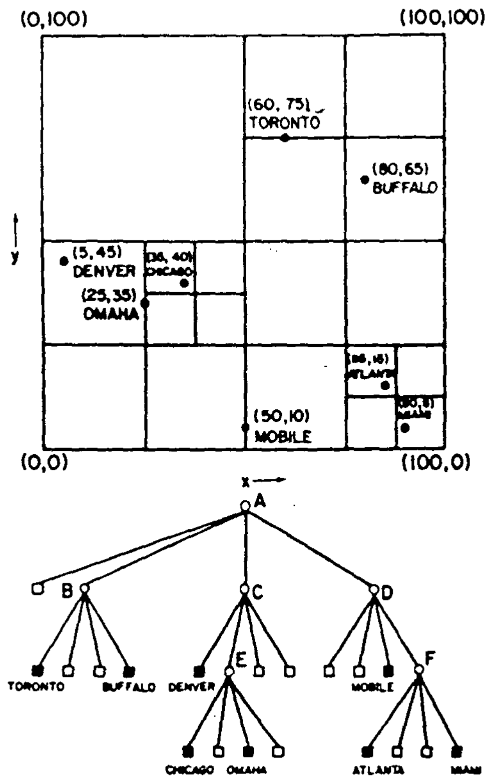


FIG 4 *PR quadtree and the records it represents*

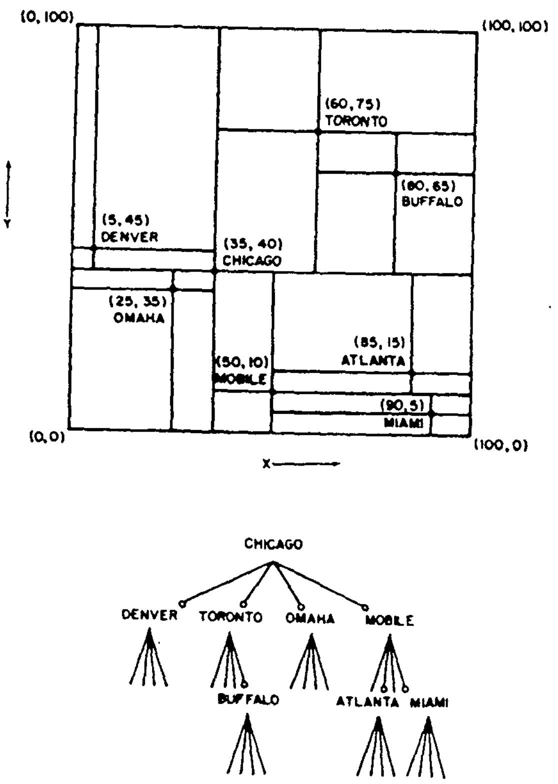


FIG 3 *A point quadtree and the records it represents*

non-elements in a square matrix, although the term MX quadtree would probably more appropriate. The MX quadtree is shown in figure 8 (in the following pages)

[SAMET 8].

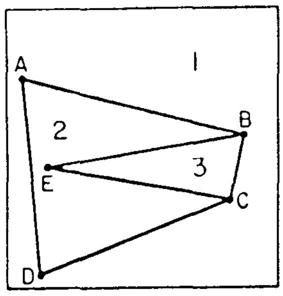
2.5.4. PR QUADTREE :

The MX quadtree is an adequate representation for points as long as the domain of the data points is discrete and finite. If this is not the case, the data points can not be represented since the minimum separation between the data points is unkown. This leads to an alternative adaptation of the region quadtree to point data that associates data points (that need not be discrete) with quadrants. We call it as a PR quadtree (P for point and R for region) although again the term PR quadtree is probably more appropriate. The PR quadtree is organised in the same way as the region quadtree. The difference is that leaf nodes are either empty (i.e., white) or contain a data point (i.e., black) and its coordinates. A quadrant consists of, at most one data point. The PR quadtree representation can also be adapted to a region that consists of a collection of polygons (termed polygonal map) [SAMET 8]. The PR quadtree is shown in figure 4 of previous page.

2.5.5. EDGE QUADTREE

The edge quadtree is based on a refinement of the MX quadtree first suggested by WARNOCK. Although WARNOCK did not make use of a tree structure, he observed that the number of squares in the decomposition could be reduced by terminating the subdivision whenever the square contains at most one line of the polygon or the intersection of two polygons. In the edge quadtree, a region containing a linear feature, or part there of, it subdivided into four squares repeatedly until a square is obtained that contains a single curve that can be approximated by a single straight line. If an edge terminates within a node, a special flag is set, and the intercept denotes the point at which the edge terminates. Applying this decomposition process leads to quadtress in which long edges are represented by large leaf nodes or a sequence of large leaf nodes. Small leaf nodes are required in the vicinity of corners or intersecting edges. Of course, many leaf nodes will contain no edge information at all [SAMET 9].

FIG. 5 Sample polygonal map.



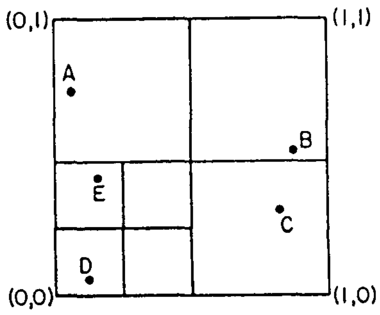
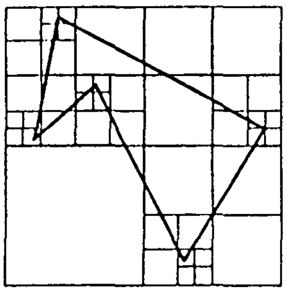


Fig. 6 PR quadtree correspon-ding to the vertices A, B, C, D, and E, of polygonal map of Figure

FIG 7 Sample edge quadtree.



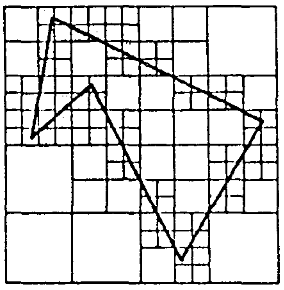


Fig. 8 MX quadtree corresponding to

2.5.6. PM QUADTREE:

The PM quadtree is a term used to describe collectively a number of related quadtree like data structures devised by SAMET and WEBBER, SAMET and NELSON to overcome some of the problems associated with the edge quadtree, and the MX quadtree for representing the polygonal maps. There are three variants of the PM quadtrees named PM1, PM2, PM3. These quadtrees are based upon either vertex or edge. Their implementation make use of the same basic data structure. All are built by applying the principle of repeatedly breaking up the collection of vertices and edges (forming polygonal map) until obtaining a subset that is sufficiently simple that it can be organised by some other data structure. This is achieved by successively weakening the definition of what constitutes a permissible leaf node, thereby enabling more information to be stored at each leaf node [SAMET 9].

2.5.6.1. PM1 QUADTREE :

The PM1 quadtree is organised in a similar way to the region and PR quadtrees. A region is repeatedly subdivided into four equal sized quadrants until we obtained blocks that do not contain more than one line. To deal with lines that intersect other lines, we say that if a block contains a point, say p, then we permit it to contain more than one line provided that p is an end point of each of the lines it contains. A block can never contain more than one end point. The definition of a PM1 quadtree can be restated as satisfying the following condition [SAMET 9].

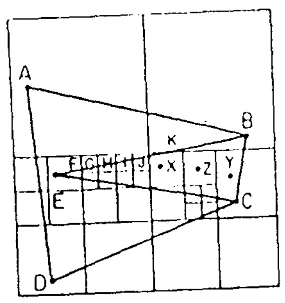
1. At most, one vertex can lie in a region represented by quadtree leaf node.

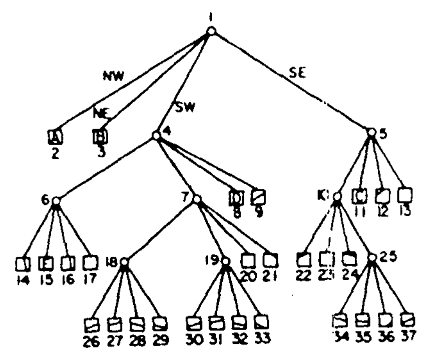
2. If a quadtree leaf node's region contain a vertex, it can contain no of edge that does not include that vertex.

3. If a quadtree leaf node's region contain no vertices, it can contain, at most one of edge.

4. Each region's quadtree leaf node is maximal.

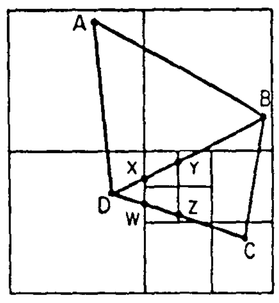
This definition of PM1 quadtree is that of a PR quadtree. The difference is that we are representing edges here rather than points. This effects the action to be taken when the decomposition induced by the PM1 quadtree results in a vertex that lies on the border of a quadtree node. We could move the vertices so that this does not happen, but generally this requires global knowl





Tree representation of the PM_1

FIG.9 PM1 QUADTREE



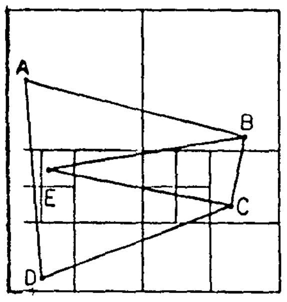


FIG10 PM2 QUADTREE

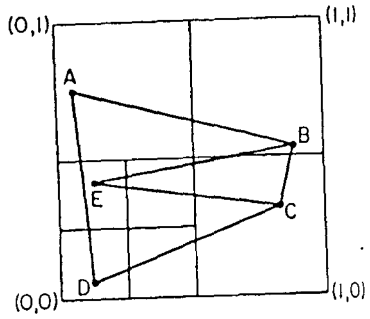


FIG11 PM3 QUADTREE

edge about the maximum depth of the quadtree point to its construction.

2.5.6.2. PM2 QUADTREE :

As the vertex of the PM1 quadtree moves closer to the quadrant boundary on its right, as shown in figure 10 (previous page), the minimum separation between ql edges becomes smaller and smaller thereby resulting in the growth of depth to unacceptable values. To remedy the deficiency (number 3 condition of the PM1 quadtree) it is necessary to determine when it dominates the cost of storing a polygonal map. The analysis leads to replace the condition 3 of PM1 quadtree definition with condition 3' defined below [SAMET 9].

3' If a quadtree leaf node's region contains no vertices, then it can contain only ql edges that meet at a common vertex exterior to the region.

The quadtree built from the above condition is termed PM2 quadtree.

2.5.6.3. PM3 QUADTREE :

The PM2 quadtree is less sensitive to shift and rotation of the polygonal map than the PM1 quadtree. This is achieved by removing the contribution of condition 3 of the PM1 quadtree definition. The next step is to remove the contribution 2 as well. This brings as back full circle to the PR quadtree, in that only condition 1 and 4 of the PM1 quadtree definition satisfied. The result is termed a PM3 quadtree and is characterised by having at most one vertex in a region represented by a quadtree leaf node. It should be clear that the number of quadtree nodes in the PR quadtree for the vertices of a polygonal map is equal to the number of quadtree nodes in the PM3 quadtree will be much greater than the amount of information stored in the quadtree leaf node of a PR quadtree [SAMET]. This is shown in figure 11 of previous page.

CHAPTER 3

DESIGN OF THE PROJECT

This chapter is mainly concerned with the design of the project. In this the most important quadtree data structures such as region quadtree and PM3 quadtree are going to be discussed. Brief discussion of each quadtree and design of algorithms are discussed in a detailed way. This chapter is broadly divided into two parts. One is region quadtree and it's details and other part is PM3 quadtree. For both types of quadtrees insertion (building the quadtree), deletion of image from the quadtree are the prime concern of the project. Not only insertion and deletion of quadtrees, operations on images using quadtrees are also discussed in great detail. The operations discussed are point location and overlay. Algorithms for insertion, deletion, point location and overlay for both quadtrees are devised and discussed.

3.1. REGION QUADTREE :

As already mentioned, the planer region is recursively subdivide into four equal parts of square type until each part contain data that is sufficiently simple so that it can be organised by some other data structure. The most studied quadtree is the region quadtree is based on the subdivision of the array into four equal sized quadrants. It the array does not contain entirely of 1s and entirely of 0s it is subdivided into quadrants and subquadrants until we obtain blocks (possibly single pixels) that consists entirely of 1s or entirely of 0s i.e., each block is entirely contained in the region or entirely disjoint from it. The algorithms are written by using pseudo algol. This particularly important because it gives maximum information with minimum code.

3.1.1. ALGORITHM FOR INSERTION :

. This algorithm builds a quadtree from raster representation.

pointer node procedure BUILD_REGION_QUADTREE (minx, miny, maxx, maxy);

/* This algorithm constructs the quadtrees from the existent image. It is assumed that image is already exists and algorithm using this image for building quadtree. Node is a pointer variable pointing to a structure consists of 9 fields. First four fields are four sons of it's node. These are also pointers of type node. Fifth field is type integer specifying it's state i.e. whether black, white or gray. Leaf nodes may contain either black nodes or white nodes based on whether they contain image or devoid of image. Intermediate nodes are of gray type. Sixth field is integer type specifying corresponding lower left corner of x_ coordinate of block and seventh field is the lower left corner y_coordinate of block. Eighth field is integer type specifying the length of the block. Ninth field is also integer type specifying the id of the node. This last field is particularly important in point location problem. This procedure calls the another procedure named NODE_ON which takes block's left_top and bottom_right corner coordinates values and return an integer which states that whether decomposition is necessary or not. Minx, miny, maxx, maxy, MAXX, MAXY, k, i are integers representing the coordinate values. r is a pointer variable of type node. This procedure takes the four integer values as input corresponding to the image top left and bottom right corners and returns a pointer of type node which is the address of the root node of quadtree */

```
value integer minx, miny, maxx, maxy;
BEGIN
k< - NODE_ON (minx, miny, maxx, maxy);
    r = new node ;
    if (k = 0)
      BEGIN
        son (r, 11)<-son (r, 12)<- son (r, 13)<-son (r, 14)< - NULL;
          state (r) < - WHITE;
    minix(r)<-minx;
```

```
        miniy(r)<-miny;
        side(r)<-maxx-minx;
        id(r)<-num;
        return r;
    END
    if(k=1)
      BEGIN
        son(r,11)<-son(r,12)<-son(r,13)<-son(r,14)<-NULL;
        state(r)<-BLACK;
        minix(r)<-minx;
        miniy(r)<-miny;
        side(r)<-maxx-minx;
        id(r)<-num;
        return r;
    END
        i<-maxx-minx;
        i<-i/2;
    son(r,11)<-BUILD_REGION_QUADTREE(minx,miny,minx+i,miny+i);
    son(r,12)<-BUILD_REGION_QUADTREE(minx+i,miny,minxx,miny+i);
    son(r,13)<-BUILD_REGION_QUADTREE(mix,miny+i,minx+i,minxy);
    son(r,14)<-BUILD_REGION_QUADTREE(mix+i,miny+i,maxx,maxy);
    return r;
    END
```

Int procedure NODE_ON (xo,yo,x1,y1)

/* This procedure takes the four integer values x0,y0,x1,y1. This algorithm scans the image pixel by pixel and row by row. This checks whether given block (left top is x0,y0 and right bottom is x1,y1) contains only 0s or 1s or both. Here 0s represent off pixels and 1s represent on pixels. It returns an integer value of 0 if block contains only white pixels (off pixels), 1 if the block contain only black pixels (on pixels) otherwise it returns the value of 2 stating that image contains both*/

```
        value integer x0,y0,x1,y1;
```

```
BEGIN
sum<-0;
for each x0 to x1
  BEGIN
    for each y0 to y1
      BEGIN
        value<-get_pixel();
            sum<-sum+value;
        END
END
if(sum=0)
return 0;
else if(sum=((x1-x0)*(y1-y0))
return 1;
else
return 2;
END
```

3.1.2. ALGORITHM DELETION:

Pointer node DELETE_REGION_QUADTREE(root1, root2);

/* This algorithm deletes the image. It is assumed that the image to be deleted is exists and there is no chance to deleting the inexistant image. The procedure SPLIT_NODE splits the existing image block four sub blocks of the image. The algorithm for this procedure can be found in INSERTION algorithm of PM3 quadtree.*/

```
value pointer node root1;
value pointer node root2;
BEGIN
```

```
if ((state (root) is WHITE and state (root2) is (WHITE) or (state(root1)
is BLACK and state (root2) is (WHITE) or (state(root1) is GRAY
and state (root2) is WHITE))

        return root1;

        if (state (root1) is BLACK and state (root2) is BLACK)

        BEGIN

        state (root1)<-NUL;

        return root1;

        END

        if (state(root1) is BLACK and state(root2) is GRAY)

        BEGIN

        SPLIT_NODE (root1);

  son(root1,11)<-DELETE_REGION_QUADTREE(son(root1,11), son(root2,11));

  son(root1,12)<-DELETE_REGION_QUADTREE(son(root1,12), son(root2,12));

  son(root1,13)<-DELETE_REGION_QUADTREE(son(root1,13), son(root2,13));

  son(root1,14)<-DELETE_REGION_QUADTREE(son(root1,14), son(root2,14));

        return root1;

        END

  if (state(root1) is GRAY and state (root2) is GRAY)

        BEGIN

  son(root1,11)<-DELETE_REGION_QUADTREE(son(root1,11), son(root2,11));

  son(root1,12)<-DELETE_REGION_QUADTREE(son(root1,12), son(root2,12));

  son(root1,13)<-DELETE_REGION_QUADTREE(son(root1,13), son(root2,13));

  son(root1,14)<-DELETE_REGION_QUADTREE(son(root1,14), son(root2,14));

        return root1;

        END

  if ((state(root1) is WHITE and state (root2) is BLACK) or
(state(root1) is WHITE and state(root2) is GRAY) or (state(root1) is
GRAY and state(root2) is BLACK))

        print("Deletion of inexistant image area");

        END
```

TH-6372

3.1.3. ALGORITHM POINT LOCATION :

Probably the simplest task to perform on image is to determine the color of a given pixel. An equivalent statement of this task is to determine the block associated with a given point. In traditional array representation this is achieved by exactly one array access. In region quadtree this requires searching the quadtree structure.

```
procedure POINT_LOCATE_REGION_QUADTREE(o,p);

/* This procedure locates the block of the image in which a given
point resides. This algorithm calls the another algorithm
CREATE_QUADTREE which develops a code for finding the required block. This code eliminates the comparison with x and y
coordinates. The code is available in the array a[]. K is an index
variable and it is initialised to 0 */

value pointer node p;
value pointer point c;
    BEGIN
    CREATE_QUADCODE(c);
    if(state(p) is WHITE)
    BEGIN
    print("empty root node");
    return;
    END
if(state(p) is BLACK)
    BEGIN
    print("ID of the region");/*p->id*/
    return;
    END
while (p is not equal to NULL)
    BEGIN
    if (a[K] is 0)
        p<-son(p,11);
    if(a[K] is 1]
        p<-son(p,12);
    if(a[k] is 2)
```

```
        p<-son(p,13);
    if(a[K] is 3)
        p<-son(p,14);
    k<-k+1;
    if(state(p) is not equal to GRAY)
        BEGIN
        if (state(p) is WHITE)
        print ("empty node, no region is represented");
          return;
        END
    if(state(p) is BLACK)
        BEGIN
        print("id of the region");
        return;
        END
    END
```

procedure CREATE_QUADCODE(b);

/* This algorithm creates a code for traversing to the desired block in which the given point resides. The algorithm calls the algorithm CODE which actually gives the code number. s,y,k,i,X,Y,MAX,MIN are integers.*/

value pointer point b;

```
    BEGIN
        X<-0;
        Y<-0;
        k<-(MAX-MIN)/2;
        x<-xco(b);
        y<-yco(b);
        for i is 1 to 8 do
            BEGIN
        A[I]<-CODE(X,Y,x,y);
        k<-(k+1)/2;
        if(a[i] is 1)
            BEGIN
```

```
                X<-X-k;
                Y<-Y-k;
                END
            if(a[i] is 2)
                BEGIN
                X<-X+k;
                Y<-Y-k;
                END
            if(a[i] is 3)
                BEGIN
                X<-X-k;
                Y<-Y+k;
                END
            if(a[i] is 4)
                BEGIN
                X<-X+k;
                Y<-Y+k;
                END
            END
END
int procedure CODE(p,q,r,s);
/* This algorithm calculates the code and returns the same to the
procedure CREATE_QUADCODE*/
        value integer p,q,r,s;
BEGIN
        if(r>p)
        BEGIN
            if(s>q);
            return 4;
            else
            return 2;
        END
        else
        BEGIN
            if(s>q)
            return 3;
            else
            return 1;
        END
    END
```

3.1.4 ALGORITHM OVERLAY :

This algorithm takes the two images in the forms of quadtrees and then merges them as a single image in the form of quadtree.

recursive procedure pointer node OVERLAY (root1, root2);

/* This computes the overlay i.e., merges the two trees or union of the two trees and returns resultant quadtree root node address. This algorithm recursively calls itself when two node states are of type GRAY. This algorithm calls the another simple algorithm COPY, which takes the two arguments and copies the first argument into second argument.*/

```
value pointer node root1,root2;
BEGIN
    pointer node r;
    if(((state(root1) is WHITE) and (state(root2) is WHITE)) or
    ((state(root1) is BLACK) and (state(root2) is WHITE)) or
    ((state(root1) is BLACK) and (state(root2) is GRAY)) or
    ((state(root1) is GRAY) and (state(root2) is WHITE))
        BEGIN
            COPY(root1,r);
            return r;
        END
    if(((state(root1) is WHITE) and (state(root2) is BLACK)) or
    ((state(root1) is WHITE) and (state(root2) is GRAY)) or
    ((state(root1) is WHITE) and (state(root2) is BLACK))
        BEGIN
            COPY(root2,r);
            return r;
        END
    if((state(root1) is GRAY) and (state(root2) is GRAY))
        BEGIN
        r<-new node();
        son(root1,11)<-OVERLAY(son(root1,11),son(root2,11));
        son(root1,12)<-OVERLAY(son(root1,12),son(root2,12));
        son(root1,13)<-OVERLAY(son(root1,13),son(root2,13));
        son(root1,14)<-OVERLAY(son(root1,14),son(root2,14));
        return r;
        END
END
```

3.2 PM3 QUADTREE

Since PM quadtree is used to implement polygonal maps, it's basic entities are vertices and edges. Each vertex is represented as a record of type point which has two fields called XCOORD and YCOORD that correspond to the x and y coordinates, respectively, of the point. They can be of type real or integer depending on implementation considerations such as floating precision. An edge is implemented as a record of type line with four fields, p1, p2, left and right. p1, p2 contain pointers to the records containing the edge's vertices. Left and right are pointers to structures that identify the regions which are on the sides of the edge. We shall use the convention that left and right are with respect to a view of the edge that treats the vertex closest to the origin are marked as being associated with regions 1 and 2, respectively. Each PM node is a collection of q_edges which is organized according to the variant being implemented (i.e., PM1,PM2,PM3) and is represented as a record of type node containing seven fields. The first four fields contain pointers to the node's four sons corresponding to the directions (i.e., quadrants NW, NE, SW,SE. If p is a pointer to a node and 1 is quadrant, then these fields are referred as son (p, i). Then fifth field is NODETYPE(STATE) indicates whether the node is a terminal node(LEAF) or a non terminal node (GRAY) node. The SQUARE field is a pointer to a record of type square which indicates the size of the block corresponding to the node. It is defined for both LEAF and GRAY nodes. It has two fields, CENTER and LEN. Center points to a record of type point which contains the x andy coordinates of the center of the square. LEN contains the length of side of the square which is the block corresponding to the node in the PM quadtree. DICTIONARY is the last field and it is a pointer to a data structure that represents the set of q_edges that are associated with the node. Initially, the universe is empty and consists of no edges or vertices. It is represented by a tree of one LEAF node whose DICTIONARY field points to the empty set.

In the implementation given here the set of q_edges for each LEAF node is a linked list whose elements are records of type edgelist containing two fields DATA and NEXT. DATA points to a record of type line corresponding to the edge of which the q_edges. Although the set of q edges is implemented as a list here, it really should be implemented by a data structure that supports the efficient execution of the delete, insert, set union, and set difference operations. However, a linked list is usually sufficient since in our empirical tests, the list is enough. In case of PM3 quadtree we would want to have seven subsets corresponding to the vertex and the six combinations of sides. These subsets have been approached by D_VERTEX and D_SIDE, respectively. The set of q_edges corresponding to GRAY node is said to be empty. Note that all of the q_edges point to the same line record. The following algorithm was developed by SAMET and WEBBER[SAMET 9].

3.2.1 INSERTION ALGORITHM :

An edge is inserted into a PM3 quadtree by traversing the tree in preorder and successively clipping it (using CLIP_LINES) against the blocks corresponding to the nodes. Clipping is important because it enables us to avoid looking at areas where the edge cannot be inserted. This process is controlled by procedure PMINSERT which actually inserts a list of edges. If the edge can be inserted into the node, say p, then PMINSERT does so and exits. Otherwise, a list, say L, is formed containing the edge and any q_edges already present in the node, p is split and PMINSERT is recursively invoked to attempt to insert the elements of L in the four sons of P. PMINSERT uses PM3_CHECK to determine if the criteria of the appropriate PM quadtree is satisfied. Isolated vertices pose no problems and are handled by PM3-CHECK. The implementation given below assumes that whenever an edge or an isolated vertex is inserted into the PM quadtree it is not already there or does not intersect an existing edge. However, an endpoint of the edge may intersect an existing vertex as long as it is not an isolated vertex. Procedure CLIP_SQUARE is predicate that indicates if an edge crosses a square. Similarly, procedure PT_IN_SQUARE is a predicate that indicates if a vertex lies in a square. They are responsible for enforcing the conventions with respect to vertices and edges that lie on the boundaries of blocks. Their code is not given here. Equality between records corresponding to vertices is tested by use of the "=" symbol which requires that its two operands be of the same type (i.e., pointers to records of type point). This algorithm is developed by Samet and Webber [SAMET 9].

```
recursive procedure PMINSERT(P,R);

/* Insert the list of edges pointed at by p in the PM3 quadtree rooted
at r. This calls the procedure PM3_CHECK. */

BEGIN

        value pointer edgelist P;

        value pointer node R;

        pointer edgelist L;

        quadrant I;

        L<-CLIP_LINES(P,SQUARE(R))K;

        if (empty(L) then return; /*NO new edges belong in the quad
        rants*/

                if LEAF(R) then /*A terminal node*/

        BEGIN

        L<-MERGE_LISTS(L,DICTIONARY(R));

        if PME_CHECK(L,SQUARE(R)) then

                        BEGIN

                        DICTIONARY(R)<-L;

                        return;

                        END

        else SPLIT_PM3_NODE(R);

        END

        for I in {NW,NE,SW,SE} do PMINSERT(L,SON(R,I));

        END

        recursive edgelist procedure  CLIP_LINES(L,R);

        /* Collect all of the edges in the list of edges pointed by P that
        intersect square pointer at by R. ADD_TO_LIST(X,Y) adds
        element X to the list S and returns a pointer to the resulting
        list. */

        BEGIN

        value pointer edgelist L;

        value pointer square R;
```

```
return(if(empty(L) then NIL
        else if CLIP_SQUARE(DATA(L)R), then
        ADD_TO_LIST(DATA(L), CLIP_LINES(NEXT(L),R))
        else CLIP_LINES(NEXT(L), R));
        END
```

recursive Boolean procedure PM3_CHECK(L,S);

/* Determine if the square pointed by S and the list of edges pointed at by L form a legal PM3 quadtree node. In order to allow an isolated vertex to coexist in a leaf node along with edges that do not intersect it, INF is used to represent a factious point at (infinite, infinite) and serves as the shared vertex in the call to **SHARE_PM3_VERTED. */**

```
BEGIN
value pointer edgelist L;
value pointer square S;
return (if(empty(L) then true
        else if P1(DATA(L)) = P2(DATA(L)) then
                SHARE_PM3_VERTEX(INF,NEXT(L),S)
        else if PT_IN_SQUARE(P1(DATA(L),S) and
                PT_IN_SQUARE(P2(DATA(L),S) then false
        else if PT_IN_SQUARE(P1(DATA(L),S) then
                SHARE_PM3—IN_VERTEX(P1(DATA(L),S), NEXT (L),S)
        else if PT_IN_SQUARE(P2(DATA(L),S) then
                SHARE_PM3_VERTEX(P2(DATA(L)),NEXT(L),S)
        else PM3_CHECK(NEXT(L),S));
        END;
```

recursive Boolean procedure SHARE_PM3_VERTEX(P,L,S);

/* The vertex pointed at by P is the shared vertex in a PM3 quadtree. It is inside the square pointed at by S. Determine if all the edges in the list of edges pointed at by L either share P and do not have their other vertex within S, or do not have either of their vertices in S*/

```
BEGIN
        value pointer point P;
        value pointer  edgelist L;
        value pointer square S;
```

```
return (if(empty(L) then true
        else if P=P1(DATA(L)) then
                not (PT_IN_SQUARE(P2(DATA(L))S,)) and
                SHARE_PM3_VERTEX(P,NEXT(L),S)
        else if P=P2(DATA(L)) then
                not (PT_IN_SQUARE(P1(DATA(L)),S)) and
                SHARE_PM3_VERTEX(P,NEXT(L),S)
        else not (PT_IN_SQUARE(P1(DATA(L)),S) and
                not(PT_IN_SQUARE(P2(DATA(L)),S)) and
                SHARE_PM3_VERTEX(P,NEXT(L),S));
        END;
        procedure SPLIT_PM3_NODE(P);
/* Add four sons to the node pointed at by P and change P to be of type GRAY*/
BEGIN
        value pointer node P;
        quadrant I, J;
        pointer node Q;
        pointer square S;

/*XF and YF contain multiplicative factors to aid in the location of
the centers of the quadrant sons while descending the tree */
preload real array XF[NW,NE,SW,SE] with -0.25,0.25,-0.25, 0.25;
preload real array YF[NW,NE,SW,SE] with -0.25,0.25,-0.25, 0.25;
BEGIN

        Q<-create(node);
        SON(P,I)<-Q;
        for J in {NW,NE,SW,SE} do SON(Q,J)<-NIL;
        STATE (Q)<-LEAF;
        S<-create(square);
        SQUARE(Q)<-S;
        CENTER(S)<-create(point);
        XCOORD(CENTER(S)<-XCOORD(CENTER(SQUARE(P))) +
                              XF[I]*LEN(SQUARE(P));
        YCOORD(CENTER(S)<-YCOORD(CENTER(SQUARE(P))) +
                              YF[I]*LEN(SQUARE(P));
        LEN(S)<-0.5* LEN(SQUARE(P))
    END
    DICTIONARY(P)<-NIL;
    STATE(P)<-GRAY;
    END;
```

3.2.2. DELETION ALGORITHM :

An edge is deleted from a PM quadtree by using a process whose control structure is identical to that used in the insertion of an edge. Again, the tree is traversed in preorder and the edge is successively clipped (using CLIP_LINES) against the blocks corresponding to the nodes. This process is controlled by procedure PMDELETE which actually deletes a list of edges. At each LEAF node, the DICTIONARY field is updated to show the elimination of the edge (or edges). Once all four sons of GRAY node have been processed, an attempt is made to merge the four sons by the use of procedures POSSIBLE_PM3_MERGE and TRYTOMERGE_PM3 to check if the criteria of the appropriate PM quadtree are satisfied. These procedures made use of PM_CHECK. This algorithm is also developed by Samet and Webber [SAMET 9].

```
recursive procedure PMDELETE(P,R);

    /* Delete the list of edges pointed by P from the PM quadtree
rooted at R. This calls to procedure POSSIBLE_PME_MERGE and
TRYTOMERGE_PM3. */

BEGIN
    value pointer edgelist P;
    value pointer node R;
    pointer edgelist L;
    quadrant I;

    L<-CLIP_LINES(P,SQUARE(R));
    if empty(L) then return        ; /* none of the edges are in the quadrant*/
    if GRAY then
        BEGIN
            for I in {NW,NE,SW,SE} do PMDELETE(L,SON(R,IO));
                if POSSIBLE_PM_MERGE(R) then
        BEGIN
          L<-NIL;
          if TRYTOMERGE_PM(R,R,L) then
                BEGIN
                 DICTIONARY9R)<-L;
                      STATE(R)<-LEAF;
                   END
                END
        END
elseDICTIONARY9R)<-SET_DIFFERENCE(DICTIONARY(R),L);
        END;
```

Boolean procedure POSSIBLE_PM3_MERGE(P);

/* Determine if an attempt should be made to merge the four sons of the PM3 quadtree. Such a merger is only feasible if all four sons of a GRAY node are LEAF nodes */

```
BEGIN
  value pointer node P;
  return (LEAF(SON(P,NW)) and LEAF(SON(P,NE)) and
        LEAF(SON(P,SW)) and LEAF(SON(P,SE));
END;
```

Boolean procedure TRYTOMERGE_PME(P,R,L)P;

/*   Determine if the four sons of the PM3 quadtree rooted at node P can be merged. Variable L is used to collect all of the edges that are present in the subtrees. Note that there is no need for parameter R, and the procedure is not recursive. The call to PM3_CHECK is necessary for checking legality of quadtree */

```
        BEGIN
        value pointer node P, R;
        reference pointer edgelist L;
        quadrant I;
        for I in {NW,NE,SW,SE} do
        L<-SET_UNION(L,DICTIONARY(SON(P,I)));
        return(PM_CHECK)L,SQUARE(P)));
 END;
```

3.2.3. POINT LOCATION ALGORITHM :

Point location is an operation on image using quadtrees. This operation is starts with finding corresponding node from quadtree. Point location in PM3 quadtree is accomplished by finding closest q_edge(part of the original edge) with respect to the 7 classes of q_edges. For finding the closest edge two cases arise. One is the block(node) in which point lies, consists the vertex and other case is the block does not contain vertex. In case block contain vertex, then find the nearest edge among the edges that passes through vertex. Then find the line joining the vertex to given point. If this line intersecting any of the q_edges other than edges those closes at vertex. If no intersection occurs, the nearest q_edges is the q_edge that passes through the vertex. Then finds the region on which side of this q_edge the point lies. This is accomplished by finding intersection point (offset point) and then compare this point to the given point. Then finds the region. In case of the line joining the given point and vertex intersects the any of the 6 classes of q_edges that causes the intersection. Then finds the nearest q_edge among the intersection edges. After finding the nearest q_edge the task is simplified by finding on which side of the q_edge the point lies. This is done by finding the offset point and compare this point with the given point. In case of the block does not contain vertex vertex the task is entirely different. In this case find the q_edge among all the 6 classes of q_edges. This task is done by finding the offset from the given point to each of the q_edge. The closest q_edge is the q_edge which have the shortest distance from the point. After finding the nearest q_edge then task is simplified to finding on which side the point resides. This is also accomplished by find the offset intersection point and then compare it to the point and then finds the region [SAMET 9].

```
procedure POINT_LOCATION (C,P);

/* This algorithm locates the region of the given point */

value pointer point C;

value pointer node P;

BEGIN

 if D_VERTEX(DICTIONARY(P)) is not equal to NULL

     BEGIN
```

```
        FIND_NEAREST_VERTEX_EDGE()][
        if line joining the vertex and point c is intersected with q_edges
           BEGIN
               COLLECT_ALL_LINES_THAT_INTERSECTED();
               FIND_THE_NEAREST_LINE_TOPOING();
               FIND_REGION();
           END
        else
               FIND_REGION()'
        END
else
        BEGIN
         FIND_THE_NEAREST_EDGE();
         FIND_REGION();
        END

END;
```

3.2.4. OVERLAY ALGORITHM :

This algorithm computes the overlaying of two PM3 quadtrees. The overlay algorithm can be decomposed into four procedures: **OVERLAY, MERGE, CAN_MERGE** and **QUARTER.** The code for some of them is presented below using a pseudo **ALGOL** notation in order to provide a maximum amount of information in a minimum amount of space. Procedure **OVERLAY** takes two PM3 quadtrees as parameters. It traverses the two quadtrees in parallel. When one tree is leaf and the other tree is not, the leaf is split into a node with four sons, each of which are leaf nodes (and correspond to a description of the same region as the corresponding sons. When both quadtrees are leaf nodes, the dictionaries of q_edges in each of them are merged to form a leaf in the output tree. The dictionaries are accessed by the **D_VERTEX** and **D_SIDE** fields. **D_VERTEX** refers to the dictionaries which are accessed with the aid of dictionary indices **{NW,NE,NS,SE,SW,EW}.**

Procedure **MERGE** produces the subtree the results from merging two leaf nodes (from a pair of PM3 quadtrees) depending on whether or not the q_edges involved intersect. Recall that the information about q_edges that is stored in the leaf nodes is ordered with respect to various intercepts (either a vertex of a side of the block). Thus the merger of this information is simply the merger of the corresponding trees. The routine that performs the actual merging is termed **D_MERGE** and is not given here. The worst_case execution time of **MERGE** is proportional to the number of nodes merged plus the cost of executing the procedures : **CAN_MERGE, and QUARTER.**

The coding of procedure **MERGE** uses **WHICHEVER_HAD_D_VERTEX,** which returns **NIL** if neither leaf contains a vertex and otherwise returns the dictionary connected to the vertex. Note that the function **CAN_MERGE** has a side effect of removing redundant references to the same vertex (i.e., with the same x and y coordinates). The information in two dictionaries is merged to form a new dictionary by the function **D_MERGE.**

Procedure **CAN_MERGE** determine whether a pair of leaf nodes of PM3 quadtrees can be mered. In order to be mergible, the q_edges in the two leaf nodes cannot intersect and if there is a vertex in both of the leaf nodes, then it must have the same x and y coordinate values. Since the checking of intersection (done by the procedure **LINES_INTERSECT)** can take advantage of the q_edges, the execution time of **CAN_MERGE** is proportional to the number of q_edes in its leaf parameters.

The final procedure to consider is **QUARTER,** which takes a leaf as a parameter and returns a subtree containing four leaves that represents the same map. This procedure involves visiting each q_edge in its leaf parameter and determine which parts of it will lie in which sons of the new subtree. Its execution time is proportional to the number of q_edges processed. Its code is similar to **SPLIT_NODE** of the insertion algorithm. This algorithm is developed by Samet and Webber **[SAMET 9].**

```
procedure OVERLAY(SUBTREE1, SUBTREE2);
/* compute the overlay of the quadtrees SUBTREE1, and SUBTREE2 */
BEGIN
      value pointer quadtree SUBTREE1, SUBTREE2;
      pointer quadtree QTD,THE_SUBTREE,TREE_TO_RETURN;
      quadrant X;
      if IS_LEAF(SUBTREE1) and IS_LEAF(SUBTREE2) then
      return(MERGE(SUBTREE1, SUBTREE2);
      else if IS_LEAF(SUBTREE1) or IS_LEAF(SUBTREE2) then
      BEGIN
                                                        QTD<-
QUARTER(WHICHEVER_WAS_LEAF(SUBTREE1,SUBTREE2));
                                                THE_SUBTRE<-
WHICHEVER_WAS_NOT_LEAF(SUBTREE1,SUBTREE2);
            TREE_TO_RETURN<-NEW_NODE();
      foreach X in {NW,NE,SW,SE} do
      SON(TREE_TO_RETURN,X)<- OVERLAY(SON(qTD,X),SON(THE_SU
BTREE,X));
            return(TREE_TO_RETURN);
      END;
else
      BEGIN
       TREE_TO_RETURN<-NEW_NODE();
      for each X in {NW,NE,SW,SE} do
        SON(TREE_TO_RETURN,X)<-OVERLAY)
                  SON(SUBTREE1,X), SON(SUBTREE2,X));
        return (TREE_TO_RETURN);
      END;
END;
```

```
procedure MERGE(LEAF1, LEAF2);
/* Perform the overlay algorithm on the simple case where both
quadtrees, LEAF1,and LEAF2, are leaf nodes. */
BEGIN
      value pointer quadtree LEAF1, LEAF2;
      pointer quadtree LEAF_TO_RETURN, QT1, QT2;
```

```
        quadrant Q;
        dictionary_index X;
        if not CNA_MERGE(LEAF1, LEAF2) then
            BEGIN
            LEAF_TO_RETURN<-NEW_NODE();
            QT1<-QUARTER(LEAF1);
            QT2<-QUARTER(LEAF2);
            foreach Q in {NW,NE,SW,SE} do
            SON(LEAF_TO_RETURN,Q)<-MERGE(SON(QT1,Q),SON(QT2,Q));
                return (LEAF_TO_RETURN);
        END
    else
        BEGIN
        LEAF_TO_RETURN<-NEW_NODE();
        D_VERTEX(LEAF_TO_RETURN)<-
        WHICHEVER_HAD_D_VERTEX(LEAF1, LEAF2);
        foreach X in {NW,NW,NS,SE,SW,EW} do
            D_SIDE(LEAF_TO_RETURN,X)<- D_MERGE(D_SIDE(LEAF1,X),D
    _SIDE(LEAF2,X));
        return(LEAF_TO_RETURN);
        END;
        END;
```

Boolean procedure CAN_MERGE(LEAF1,LEAF2);

/* Returns true if and only if the merger of the leaf nodes, **LEAF1**, and **LEAF2**, would not create any new vertices. Note that in the case that neither leaf node contains a vertex, it is possible for one intersection to occur and yet the nodes would still be mergible. The counter, **N**, records the number of known vertices in the pair of nodes. If this counter is zero, then **LINES_INTERSECT**, upon noticing that exactly one intersection occurs, has the side effect of incrementing N and updating the **D_VERTEX** field of its last parameter, which is always **LEAF1**. Of course, if more than one intersection occurs, then **LINES_INTERSECT** will cause **CAN_MERGE** to return false. */

```
        BEGIN

        reference pointer quadtree LEAF1, LEAF2 ;
        dictionary_index X,Y ;
        dictionary THE_VERTEX;
        integer N;
        N<-0
        if HAS_VERTEX(LEAF1) and HAS_VERTEX(LEAF2) then
          if SAME_XY_VERTEX(LEAF1, LEAF2) then
              BEGIN
                                                   D_VERTEX(LEAF1<-
        D_MERGE(D_VERTEX(LEAF),D_VERTEX(LEAF2));
              D_VERTEX(LEAF2)<-nil;
               END;
        else return (false);
        if HAS_VERTEX(LEAF1) or HAS_VERTEX(LEAF2) then
              BEGIN
              N<-1
                                                      THE_VERTEX<-
    D_VERTEX(WHICHEVER_HAD_VERTEX(LEAF1, LEAF2);
        for each X in {NE,NW,NS,SW,SE,EW} do
          if LINES_INTERSECT(THE-VERTEX,D_SIDE(LEAF1,X),N,LEAF1)
            or LINES_INTERSECT(THE_VERTEX,D_SIDE(LEAF2,X),N,LEAF1)
            then return(false);
              END;
    foreach X in {NE,NW,NS,SW,SE,EW} do
       BEGIN
         for each Y in {NE,NW,NS,SW,SE,EW} do
            BEGIN
             if LINES_INTERSECT(D_SIDE(LEAF1,X),D_SIDE(LEAF2,Y),N,LEAF1)
           then return (false);
         END;
       END;
     return (true);
    END :
```

CHAPTER 4

IMPLEMENTATION OF THE PROJECT

This implementation part can be broadly divided into two parts. One is implementation of region quadtree and another is implementation of PM3 quadtree. The implementation of insertion, deletion, point, location, and overlay operations are done on both types of quadtrees.

ENVIRONMENT :

The project is implemented on computer system 80386/ 80486 (LAN connected) with operating system MS-DOS version 5.0. This was coded in "C" language. Compiler used was Borland C/C++ compiler.

'C' is a general purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators. "C" provides a variety of data types. The fundamental types are characters, integers and floating-point numbers of several sizes. In addition, there is a hierarchy of derived data types created with pointers, arrays, structures, and unions. Expressions are formed from operators and operands. Any expression, including an assignment or a function call, can be a statement. Pointers provide for machine independent address arithmetic.

COMPILER USED :

The compiler used to do the project work is the Borland C/C++ compiler, version 3.1. Borland C compiler is a full implementation of the language, a language known for its efficiency, economy, and portability.

HEADER FUNCTIONS USED :

In the implementation of quadtrees, the following header functions were used.

```
#include<studio.h>

#include<conio.h>

#include<graphics.h>

#include<math.h>

#include<stdlib.h>

#include<iostream.h>

#include<mem.h>

#include<alloc.h>
```

4.1 DATA STRUCTURES USED :

The following structures are defined and heavily used. Their definitions and descriptions are mentioned below.

4.1.1.STRUCTURE NODE FOR REGION QUADTREE : This structure is used region quadtree. This has 9 fields four of which are sons, fifth is state of node i.e. whether it is a black or white or gray, sixth is identification number, seventh is length of side of the block, eighth and ninth is the x and y coordinates of left bottom of block.

```
struct node {
        struct node *11,*12,*13,*14;
        int state;
        int id;
        int side;
        int minix, miniy;
                };

        typedef struct node node;
```

4.1.2. STRUCTURE FOR POINT : This is used for both types of quadtrees and consists x and y coordinates.

```
        struct point {
        int xco ;
        int yco;
        };

typedef struct point point;
```

4.1.3. STRUCTURE FOR LINE : This structure is consists of four fields. First two fields are pointers to end points of line. And last two fields are specifying the regions on both sides of line.

```
        struct line {
                point *p1, *p2;
                int left;
                int right;
              };
        typedef struct line line ;
```

4.1.4. STRUCTURE FOR SQUARE : This structure is used in PM3 quadtree. This specifies the block center coordinates and length of the block.

```
struct square {
        point *center;
        int len;
        };

typedef struct square square;
```

4.1.5. STRUCTURE FOR EDGELIST : This structure is heavily used in PM3 quadtree. This consists of two fields, one is data field and next is pointer to next edgelist.

```
struct edgelist {
        line *data;
        struct edgelist *next;
        };

typedef struct edgelist edgelist;
```

4.1.5. STRUCTURE FOR DICTIONARY : This structure is used if PM3 quadtree for accessing the edgelists. The edges having vertex are placed in dvertex field and other edges are placed in dside field.

```
struct dictionary {

        edgelist *dvertex;

        struct dside *ds;

        };

        typedef struct dictionary dictionary;
```

4.1.6. STRUCTURE FOR DSIDE : This structure is used in PM3 quadtree for accessing the edges those have no vertex.

```
            struct dside {
            edgelist *d1;
            edgelist *d2;
            edgelist *d3;
            edgelist *d4;
            edgelist *d5;
            edgelist *d6;
             };
typedef struct dside dside ;
```

4.1.7. STRUCTURE FOR NODE : This structure is used in PM3 quadtree. This description is mentioned in detail in PM3 quadtree.

```
struct node {
            struct node *11,*12,*13,*14;
            int state;
            square *sqr;
            dictionary *d;
            };
typedef struct node node;
```

4.2 REGION QUADTREE :

In this region quadtree insertion, deletion, point location and overlay implementation details are discussed.

4.2.1 IMPLEMENTATION OF INSERTION ALGORITHM :

node *create_region_quadtree(void)

```
    {
    /* This function builds the quadtree from the raster image and
returns a pointer of type node, which is root node address. This
function calls the build_tree() function. */
    }
```

```
node *build_tree(int x0, int y0, x1, int y1)

{

/* This function takes the left top and right_bottom coordinates of
the image block and returns a pointer of type node. This function
calls another function node_on(). */

}

int node_on(int x0,inty0, int x1, inty1)

{

/* This routine checks whether the block of the image contain en-
tirely 1s or entirely 0s or both. According to image position in block
it returns 0 if the block contain entirely 0s (i.e., off pixels) or it re-
turns 1 if the block of image contain entirely 1s(i.e., on pixels) or it
returns 2 if block of image contain both off pixels and on pixels.*/

}
```

4.2.2. IMPLEMENTATION OF DELETION ALGORITHM :

```
node *delete_region_quadtree(node*p, node*q)

{

/* This function takes the two pointer variables of type node. One is
the source tree root node and another is destination tree root node
address which is to be deleted. This function returns a node which
is the resultant tree root node address. This function recursively
calls itself, when two node states of the input trees are of type GRAY
i.e., 2. It also calls the function split_node(), if the source tree has
the state of 1(BLACK and destination tree has the state or 2(GRAY).
If p state is o(WHITE) and q state is 1(BLACK) or p state is 0 and
q state is 2 or p state is 2 and q state is 1 the deletion algorithm
prints the message "illegal deletion". This is because of assump-
tion that the image to be deleted is there and exists.*/

}
```

```
node *split_node(node *p)

{

/* This function splits the image block into four sub-blocks without
change in image. It simply adds four sons to the given node and
changes its state to 2 (GRAY). */

}
```

4.2.3. IMPLEMENTATION OF POINT LOCATION ALGORITHM :

```
void point_locate_region_quadtree(node *p, point *c)

{

/* This function takes the arguments of two pointer variables. One
is of type node and another of type point. Node p is root node of
the tree of the image. This function calls the another function
create_quadcode() which returns the code for accessing the re-
quired node. After getting the required node then task is to find the
region by printing the id of the node.*/

}


void create_quadcode(point *p)

        {

/* This function takes the argument as pointer variable of type point
and then determines the code for finding the block from total im-
age. This function calls the function code () which actually calcu-
lates the code. This code is stored in global array which is ac-
cessed by parent function. */

}

int code (int p, int q, int r, int s)

{

/* This function calculates the code according to the grid size and
returns 0,1,2,3, values */

}
```

4.2.4. IMPLEMENTATION OF OVERLAY ALGORITHM :

node *overlay_region_quadtree(node *p, node*q)

{

/* This function takes the two arguments as pointer variables of type node and returns a pointer of type node which is the root node address of resultant quadtree after overlaying. The two input variables are actually two subtree root node addresses. In this case it is assumed that two images are not intersected. This function recursively calls itself when two states of input nodes are of type GRAY. Other wise it calls the function copy () which copies the first argument into second argument and returns the same which is the resultant quadtree root node */

}

4.3 PM3 QUADTREE :

The following section describes the implementation details of the PM3 quadtrees i.e. insertion, deletion, point location and overlay.

4.3.1. IMPLEMENTATION OF INSERTION ALGORITHM :

void insert_pm3_quadtree(edgelist *p, node *r)

{

/* This routine inserts the set of edges into the quadtree. This function takes the two arguments and returns null. The first argument is the pointer variable of type edgelist and second is the pointer variable of type node. The first argument is the address of the first edge of the set of edges. This is a linked list. The second argument is the root node address of the quadtree. This function calls the clip_lines(), which actually clips the set of edges against the given node boundaries and it returns the edges that are clipped. Then it calls the another function mergelists() which actually merges the two given lists and returns the resultant list. The function pm3_check() checks whether the block contain the set of edges forms a legal quadtree node or not. This function also calls the place_edges(), split_pm3_node() functions.*/

}

edgelist *cliplines(edgelist *l, square *r)

{

/* This function actually clips the lines against the square spefied by the second argument. The lines are submitted through first argument. This function calls another function clip_square() which checks whether the line is clipped or not. This is recursively called by itself when situation arises.*/

}

int pme_check)edgelist *l, square *s)

{

/* This function verified the set of edges placed in square are formed a legal node or not. It return 0 if it fails or 1 if it success. This function calls the pt_in_square() and share_pm3_vertex().*/

}

int share_pm3_vertex(point *P, edgelist *l, square *s)

{

/* This function determines, the vertex pointed by p is the shared vertex or not. Determine if all of the edges in the list of edges pointed by L either share p and do not have their other vertex within s or do not have either of their vertices in s. */

}

void split_pm3_node(node *p)

{

/* This function takes the argument a node and adds four sons and changes the state of this node into of types GRAY and returns the same.*/

}

edgelist *merge_lists(edgelist *a, dictionary *d)

{

/* This function takes the input arguments, one is edgelist and other is the dictionary. It merges the given edgelist to the edgelists of dictionary and returns an edgelist of resultant. This function calls setunion ().*/

}

```
void place_edges(edgelist *l, node *r)

{

/* This function places the set of edges which is first argument of
the function in the given node into appropriate places of 7 classes
*/

}
```

4.3.2. IMPLEMENTATION OF DELETION ALGORITHM :

```
node *delete_pm3_quadtree(edgelist *p, node *r)

{

/* This function deletes the list of edges pointed by p from pm3
quadtree rooted at r. This calls the functions possible_pm3_merge(),
try_to_merge(). This function also calls the clipliness(),
set_difference.*/

}


int possible_pm3_merge(node *p)

{

/* Determine if an attempt should be made to merge the four sons
of the pm3 quadtree. Such a merge is only feasible if all four sons
of gray node are leaf node. It return 1 if the merging is possible or
0 if merging is not possible. */

}


int try_to_merge(node *p, node *r, edgelist *c)

{

/* This function determines if the four sons of the pm3 quadtree
rooted at node p can be merged. Variable L is used to collect all of
the edges that are present in the subtrees. Note that there is no
need to the parameter r but it is used for generic. This functions
calls the another function setunion(). */

}
```

4.3.3. IMPLEMENTATION OF POINT LOCATION ALGORITHM :

```
void point_location_pm3_quadtree(point *p, node *n)

{

/* Before computing the region in which the point lies, find the node
(block of image which consists set of lines) from the given quadtree.
This node can be found by using the create_quadtree() and code()
functions of the previous topic of quadtree. This function computes
the region in which a given point lies. This function calls the
find_vertex(), nearest_vertex_edge(), intersect(), setunion(),
point_loc(), locate() functions. */

}


point *find_vertex(edgelist *a, square *s)

{

/* This function finds the vertex from the set of lines within the block
of square s. This returns the vertex point to the parent function */

}

edgelist *nearest_vertex_edge(point *p, edgelist *e, square *s)

{

/* This function computes the nearest edge among the number of
lines having vertex with the given point. This uses the header func-
tion <math.h> for finding the angles, there by it finds nearest vertex
edge */

}


edgelist *intersect(edgelist*z, node *n)

{

/* This function calculates the edges that are intersected with the
any of the edges that are placed in the given node and return these
edges that are intersected. This calls the function
find_intersection_point(). */

}
```

```
point *find_intersect_point(edgelist *a, edgelist *b)

{

/* This function computes the intersection of two edges which are
supplied as input and returns the intersection point to the parent
function. */

}


void locate (point *p, edgelist *e)

{

/* This function computes region of the given point by finding on
which side of the edge it lies. This function calls the offset_point().
*/

}

point *offset-Point(edgelist *e, point *p)

{

/* This function computes the offset (perpendicular) point for the
given point to the edgelist and returns the same to the parent func-
tion */

}


void point_locate(point *p, edgelist *a)

{

/* This function computes the nearest edge among the number of
lines and then finds on which side of the line and point lies. This
function calls another function offset_distance(0. */

}


int offset_distance(edgelist *a, point *p)

{

/* This function computes the offset distance between the given
edge to the given point and then returns the same to the original function. */

}
```

edgelist *setunion (edgelist *a, dictionary *z)

{

/* This function collects all the edges from the dictionary i.e., from 7 classes (dvertex and dside). The first argument to dummy argument which has the zero value. This is using for generic facility.*/

}

4.2.4. IMPLEMENTATION OF OVERLAY ALGORITHM :

node *overlay_pm3_quadtree(node *a, node *b)

{

/* The above function takes the two arguments as input. These two arguments are actually two quadtree (subtrees) addresses. This function merges or overlaying the two trees into one tree (as a single image) and returns the same. This function calls the another functions merge(), quarter(). This function also calls itself when both node states are of type GRATY. */

}

node *merge (node *a, node *b)

{

/* This function actually merges the two given input nodes which are two quadtree root node addresses and returns resultant quadtree root node. Merging the two nodes means merging the edgelists of two node dectionaries. That is each class of dictionary into corresponding each class of 7 classes in the dictionary. This function calls another function which actually returns an integer which specifies whether merging can be possible or not. This function also calls quarter function and also calls d-merge(). */

}

int can_merge(node *a, node *b)

{

/* This function determines the possibility of merging and returns 1 if the merging can possible or 0 if merging can not possible. This function calls lines_intersect() and samexy() functions. */

}

```
int lines_intersect(edgelist *v, edgelist *d, node *p)

{

/* This function calculates the intersection of two lines and also finds whether
the two lines are intersected or not. If single intersection occur, the vertex is
created and placed in the node p. Each line in each edgelist is tested thor-
oughly. If there is an intersection it returns 1, or if no intersection it returns
0. */

}


node *quarter (node *p)

{

/* This function takes input as a single argument which is a pointer variable
of type node and this node is splinted into four equal blocks without chang-
ing the image i.e., adding the four sons to the p and changing the p state to
non leaf and returns the address of this father to the original function. */

}


edgelist *d_merge(edgelist *a, edgelist *b0

{

/* This function takes two arguments as two edgelists and merges them into
a single edgelist and returns that edgelist to the function from which it is
called. */

}


int samexy (node *a, node *b}

{

/* This function takes the input as two nodes and collects the edgelists from
dvertex of dictionary and then tests whether the two dvertices have the same
vertex or different vertex. If the vertex of two nodes are same it returns 1 or
it returns 0 otherwise. */

}
```

CHAPTER 5

COMPARISON AND ANALYSIS :

5.1 COMPARISON :

In previous chapters various data structures for representing spatial data were discussed. Among the various data structures we have chosen region quadtree and PM3 quadtree because these have many advantages over other structures. So this chapter of comparison and analysis is very important for justification of the benefits. In this chapter the comparisons are to be made to data structures, that represent interiors of the image and the data structures that specify only exteriors i.e., boundaries of the data structures. The region can be represented only exteriors i.e., boundaries of the data structures. The region can be represented either by chain codes, runlength codes, block codes and region quadtrees. Boundaries of image can be represented by MX quadtree, edge quadtree, PM1 quadtree, PM2 quadtree and PM3 quadtrees. So comparisons can be made between the data structures of region representation and boundary representation.

5.1.1. COMPARISON BETWEEN DATA STRUCTURES OF INTERIOR REPRESENTATION :

For comparing the various methods, some form of image is necessary. This image is shown in figure 2 of chapter 2. For representing that region each data structure requires some amount of memory (space complexity). So we are comparing between among chain codes, runlength codes, block codes and region quadtree. Table 7 of following pages shown that for representing that image it requires 256 numbers (cells) if array method is used. But it requires 64 numbers if chain codes, and it requires 29 numbers if runlength codes are used. In case block codes 57 numbers are required. Lastly our most studied region quadtree requires only 6 numbers. So region qudtree is superior than all.

5.1.2. COMPARISON BETWEEN DATA STRUCTURES OF EXTERIOR REPRESENTATION :

For comparing the various quadtrees of exterior nature of representation 3 cartographic maps are taken. One is road line map, another is cityline map and last is powerline map [SAMET 9]. The power line map shows the path of the main power line. The cityline map indicates the border of the local municipality. The road line map is our most complicated map, which details a part of the local roadway work. Table 7 of next page contains the numbers of vertices and edges in each of these maps. Note that all of these maps consists of line segments whose vertices rest on 512*512 grid at level 9.

As mentioned in previous chapter neither MX quadtree nor the edge quadtree is really an appropriate representation for polygonal maps, since they only correspond to an approximation of the map, where as the variants of the PM quadtree represent the maps exactly. Nevertheless in practice, for the MX quadtree is natural to consider the approximation that results from representing line segments with same accuracy as grid. In the 512*512 image, that we are considering, this means that the MX quadtree is built by truncating the decomposition at depth 9. Similarly the edge quadtree is also constructed by translating the decomposition at depth 9. Tables 1 to 6 summaries the storage requirements of the various quadtree methods of representing the maps. As we observed before, the PM1 quadtree will always be the largest of the PM quadtrees, i.e., it will require the most nodes. Therefore, let us consider how it compares with two alternative approaches, the MX and edge quadtree given in tables 2 and 3 respectively. Table 4 contain data for the different PM quadtrees. Table 5 shows breakdown the leaf count in terms of the different type of nodes on all as gives the average number of q_edges for each node type (in parenthes) where it is relevant.

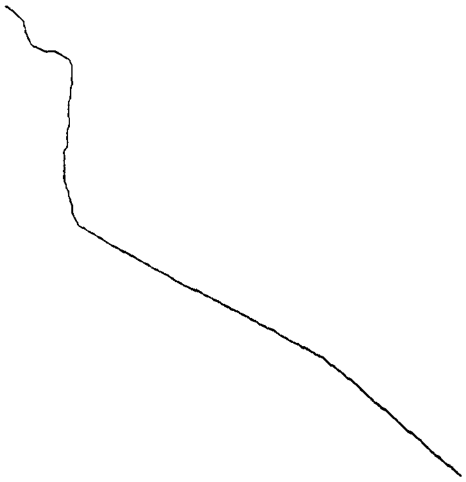


Fig. 12. The powerline map.

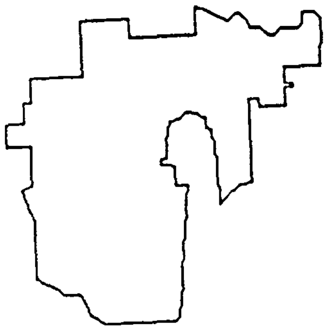


Fig. 13. The cityline map.

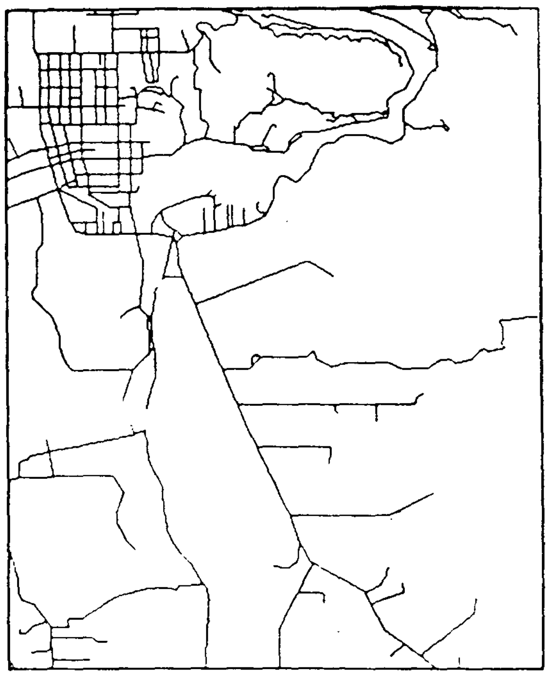


FIG NO: 14 ROADLINE MAP

The MX quadtree (see table 2) has the worst case performance. In all of our examples the MX quadtree is larger than the PM1 quadtree (see table 4) by at least a fraction of 9. More generally, we could expect the size of the MX quadtree for a polygonal map to roughly as large as the product if the average line and number of nodes in the corresponding PM1 quadtree. The edge quadtree can be seen to a definite improvement over MX quadtree corresponding the trivial maps like powerline and cityline, we seen the edge quadtree is the 3 times as large as the PM1 quadtree.

The roadline map, which is the most complex map that we have examined to date is also 3 times the PM1 quadtree. This is first surprising, since we observe that the maximum depth at this quadtree is considerably greater than that required by the digitization grid. In this digitization grid, it requires some nodes at depth of 13. In essence, the average depth of a vertex node is again between 6 and 7 for PM1 quadtree while it is 9 for the edge quadtree.

The above discussion leads us to conclude that the PM1 is improvement over earlier approaches to handling real data. We have seen that the PM1 quadtree has the desirable property of reducing the average depth at which the dominant node type is located. The PM2 and PM3 quadtrees are attempts to further reduce the maximum depth of nodes in a PM1 quadtree. The PM2 quadtree has the effect of reducing the maximum depth (see table 6 of previous) by a virtue of compact treatment of the case when close edges that radiate from the same vertex lie in a different node from the vertex. When comparing the depth of the PM1 quadtree column with the data of the PM2 quadtree columns of table 4, we observe no change in the powerline map, since it is composed of only obtuse angles.

Comparing the PM3 quadtree with the PM1, PM2 quadtrees also shows no change in the number of nodes, when used on the powerline map. This is because, the powerline map contain no edges that pass closely to vertices, other than their end points. This is because the powerline map contain no edges that pass closely to vertices other than their end points. This situation occurs a bit more frequently in the cityline map resulting 12 percent reduction in size. Note that the existence of such situation implies that vertex nodes will be slightly closer to the root in the PM3 quadtree instead of PM1 quadtree leads to 19 percent

Table I. Size of the Maps

Map	No. of vertices	No. of edges
Powerline	15	14
Cityline	64	64
Roadline	684	764

Table II. Size of the MX Quadtrees

Map	Depth	Leaves	BLACK nodes	WHITE nodes
Powerline	9	1594	521	1073
Cityline	9	2335	782	1553
Roadline	9	19513	7055	12458

Table III. Size of the Edge Quadtrees

Map	Depth	Leaves	Vertex nodes	Line nodes	WHITE nodes
Powerline	9	211	15	68	128
Cityline	9	730	64	219	447
Roadline	9	6658	684	2431	3543

Table IV. Size of the PM_1, PM_2, and PM_3 Quadtrees

	Depth			Leaves			Q-edges		
Map	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3
Powerline	7	7	7	61	61	61	38	38	38
Cityline	9	8	8	214	208	187	178	176	168
Roadline	13	9	9	2125	1960	1714	2144	2096	1976

Table V. Breakdown of Information in Table IV by Node Type

	Vertex nodes (average q-edges)			Line nodes (average q-edges)			WHITE nodes		
Map	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3
Powerline	15 (1.9)	15 (1.9)	15 (1.9)	10	10 (1.0)	10 (1.0)	36	36	36
Cityline	64 (2.0)	64 (2.0)	64 (2.1)	50	47 (1.0)	33 (1.0)	100	97	90
Roadline	684 (2.2)	684 (2.2)	684 (2.3)	618	515 (1.1)	360 (1.1)	823	761	670

Table VI. Distribution of Node Types by Depth for PM Quadtrees in the Roadline Map

	Vertex nodes			Line nodes			WHITE nodes		
Depth	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	4	4	4
3	2	2	2	0	0	0	8	8	8
4	10	10	12	4	6	7	38	38	38
5	75	75	82	34	35	29	109	105	101
6	180	180	192	132	127	120	218	212	199
7	224	224	232	204	198	139	237	222	195
8	158	158	135	156	126	59	162	142	104
9	35	35	29	48	23	6	40	30	21
10	0			13			2		
11	0			14			3		
12	0			10			1		
13	0			3			1		

Table 7

Type	Nodes	q_edges	Dictionaries
PM1	28	31	24
PM2	16	22	13
PM3	7	17	9

Comparison of various PM quadtrees.

Table 8

Data structure	Numbers required
Array	69
Chain codes	64
Runlength code	29
Block codes	57
Quadtrees	6

Comparison of various data structures of interior region representation.

reduction in size. This is due to the tendency for vertex nodes to occur closer to the root in the PM3 quadtree than the PM1 quadtree.

From the above we see that although the difference among the different PM quadtrees can be drastic in principle, for typical cartographic data, this difference in the number of nodes in the various PM quadtrees for a particular map is less pronounced. Thus, for cartographic data the choice among the different PM quadtrees is dictated more by the problems of implementation rather than by the need to conserve space. However, it should be noted that cartographic data is rather special in that it generally consists of sequence of short line segments meeting at obtuse angles. Since the lengths of the line segments are often shorter than the distance between the features that the line segments are representing, this yields data tends to bring out the best in each of the types of PM quadtrees with the result that there is little difference between them. For data that is not this simple, then benefits of the PM2 and PM3 quadtrees over PM1 quadtree should be more pronounced. The main motivation for the development of the PM qudtree data structure is that its size is relatively invariant to shifting and rotation.

5.2 ANALYSIS :

5.2.1. REGION QUADTREE :

The prime motivation for the development of the quadtree is the desire to reduce the amount of space necessary to store data through the use of aggregation of homogeneous blocks. However a quadtree implementation does have overhead in terms of the non leaf nodes. For an image with B and W black and white nodes respectively, 4*(B+W)/3 nodes required. In contrast, a binary array representation of a $2^n * 2^n$ image requires only 2^{2n} bits. However this quantity grow quite quickly. Further more if the amount of aggregation is minimal, the quadtree is not very efficient. This is due to [SAMET 8].

The worst case for a quadtree of a given depth in terms of storage requirements occurs when the region depth corresponds to a chessboard pattern. The amount of space required is obviously a function of the resolution (i.e. the number of levels in the grid within which it is embedded). As a single example of placing a square of size $2^m * 2^m$ at any position in a $2^n * 2^n$ image requires an average of $0(2^{m+n} + n - m)$ quadtree nodes. An alternative characterization of this result is that the amount of space necessary is $o(p+n)$ where p is the perimeter (in pixel width) of the block.

Dyer's $0(p+n)$ result for a square image is merely an instance of the following theorem developed by Hunter and Steglitz who obtained the same result for simple polygons (i.e. polygons with non intersecting edges and without holes) [SAMET].

THEOREM : The quadtree corresponding to a polygon with perimeter p embedded in an image of $2^n * 2^n$ has a maximum of $24*n - 19 + 24*p$ (i.e., $0(p+n)$ nodes.

COROLLARY : The maximum number of nodes in a quadtree corresponding to an image is directly proportional to the resolution of the image.

The significance of corollary is that when using quadtrees, increasing the image resolution leads to a linear growth in the number of nodes. This is in contrast to the binary array representation where doubling the resolution leads to a quadrupling of number pixels.

Since in most practical cases the perimeter, p, dominates the resolution, n, the results of theoremare usually interprets as stating that the number of nodes in a quadtree is proportional to the perimeter of the region contained therein. The complexity measures discussed above do not explicitly reflect the fact that the amount of space occupied by a quadtree corresponding to a region is extremely sensitive to its orientation. For example, in DYER experiment, the number of nodes required for the arbitrary placement of a square of size $2^m * 2^m$ at any position in a $2^n * 2^n$ image ranged between $4*(n-m) + 1$ and $4*p + 156*(n-m) - 27$, with average being $0(p+n-m)$. Clearly shifting the image within the space in which it is embedded can reduce the total number of nodes. The problems of finding the optimal position for a quadtree can be decomposed into two parts. First, we must determine the optimal grid, resolution and second, the partition points.

5.2.2. PM QUADTREES :

The main goal in specifying this data structure is derive a reasonably compact representation that satisfies the following three criteria.

1. It stores polygonal maps without information loss (i.e. it does not suffer a loss of accuracy resulting from digitization).

2. It is not overly sensitive to the positioning of the map (i.e., shift and rotation operations do not drastically change the storage requirements of the map.)

3. It can be efficiently manipulated.

To meet these goals we develop three closely related quadtree structures, PM1, PM2, PM3. Our approach is to find a decomposition criteria that corresponds to the principle of repeatedly breaking up the collection of vertices and edges (forming the polygonal map) until obtaining a subset that is sufficiently simple so that it can be organised by some other data structure. We find this decomposition criteria by recursively weakening the definition that constitutes a permissible leaf node. Thus a permissible PM1 quadtree leaf node is also a permissible PM2 leaf node and likewise a permissible PM3 quadtree leaf node.

The depth of the PM1 quadtree is maximum of the following.

D1 = 1 + log 2(root 2/d min w)

D2 = 1 + log 2(root 2/d min ev).

D3 = 1 + log 2(root 2/d min ee).

Where d min W is the minimum separation distance between vertex and vertex.

Where d min ev is the separation distance between edge and vertex.

Where d min ev is the separation distance between edge and edge.

Plus the depth of the dictionary structure.

$$A1 = \log 2 V + 1$$

Where V is the number of vertices.

But the depth of the PM3 quadtree is D1 + A1.

CHAPTER 6

CONCLUSION

The approach taken to the development of a quadtree like data structures for storing polygonal maps is interruptive. This is started with the region quadtree, and then proceed to development of PM quadtrees. The final formulation, PM3 quadtree, uses same decomposition rate as the PR quadtree, but it stores considerably more information in the terminal nodes. The PM quadtree enables storing polygonal maps without information loss. Since isolated vertices pose no problems, the PM quadtree can be used to represent points, lines and regions. The workdone in this project is implementation of insertion, deletion, point location and overlay for both region and PM3 quadtrees.

Some future work includes the development of algorithms for other operations i.e., shifting and rotation. In implementation aspect of view window based user interface can be developed for this project.

BIBLIOGRAPHY

1. Brian Kernighan and Dennis Ritchie. C programming language, second edition, Tata Mc graw Hill, Prentice Hall of India, 1993.

2. Gottfried. Programming with C, schaum series, Tata Mc Graw Hill, 1991.

3. Harrington. S, Computer Graphics, A programming approach, Tata Mc Graw Hill, 1987.

4. Herbert. F and Gofferdo. G. Pieroni, Map data processing, Academic press, 1980.

5. Tremble and Sorenson. Data structures and their applications, Tata Mc Graw Hill, 1991.

REFERENCES

6. Burrows, Geographic information Systems for land resources assessment.

7. Hanan Samet. Applications of spatial data structures, Addison Wesley, 1994.

8. Hanan Samet. The design and analysis of spatial data struc tures, Addison Wesley, 1994.

9. Hanan Samet and Robert Webber, Storing a collection of polygons using quadtrees. ACM transactions on graphics, Vol4, No. 3 July, 1985, pages 182-222.

APPENDIX A

DESCRIPTION OF PSEUDO-CODE LANGUAGE

The algorithms are given using pseudo-code. This pseudo-code is a variant of the **ALGOL** programming language, which has a data structuring facility that incorporates pointers and record structures. It make heavy use of recursion. This language has similarities to **C, PASCAL and ALGOL** W. All reserved words are designed in boldface. This also includes the names of predefined record structures.

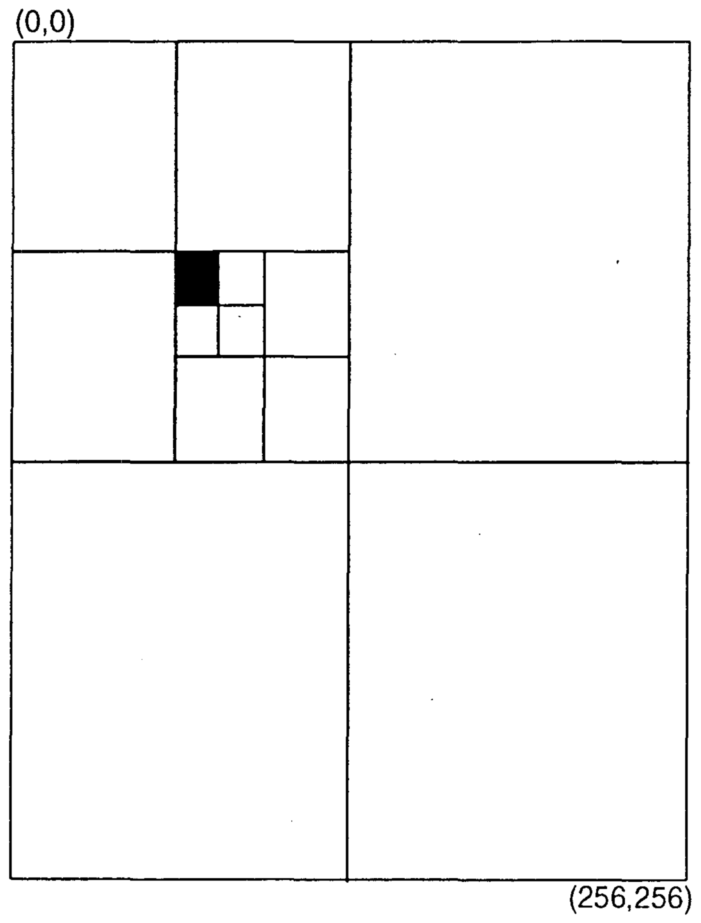
A program is defined as a collection of functions and procedures. Functions are distinguished from procedures by the fact that they return values via the use of the **return** construct. Also the function header declaration specifies the types of the data that it returns as its value. In the following the term **procedure** is used to refer to both procedures and functions.

All formal parameters to procedures must be declared along with the manner in which they have been transmitted **(value or reference).** All local variables are declared. Global variables are declared and are specified by use of the reserved word **global** before the declaration with the reserved **preload** and appending the reserved word with and the initialisation values to the declaration.

A procedure is a block of statements separated by semicolons. A block is delimited by **begin** and **end.** An if-then-else constitutes a statement. Blocks must be used after a then and an else when there is more than one statement in this position. Pointer to a record structures are declared by use of the reserved word pointer, which is followed by the name of the type of the record structure. Short-circuit evaluation techniques are used for Boolean operators that combine relational expressions. In other words, as soon as any parts of a Boolean or (and) are ue(false), the remaining parts are not tested. There is a rich set of types. These types are defined as they are used. In particular, enumerated types in the sense of PASCAL are used heavily. For example, the type **quadrant** is an enumerated type whose values are **NW, NE, SW, SE.**

APPENDIX B
(RESULTS)

BUILDING A QUADTREE FROM IMAGE (INSERTION)



2 2 0 0 0 2 2 1 0 0 0 0 0 0 0 0

(PREORDER TRAVERSAL OF STATES)

Give the id of object 1

Give the number of points of polygons : 4

Give the x value of point 1 : 64

Give the y value of point 1 : 64

Give the x value of point 2 : 79

Give the y value of point 2 : 64

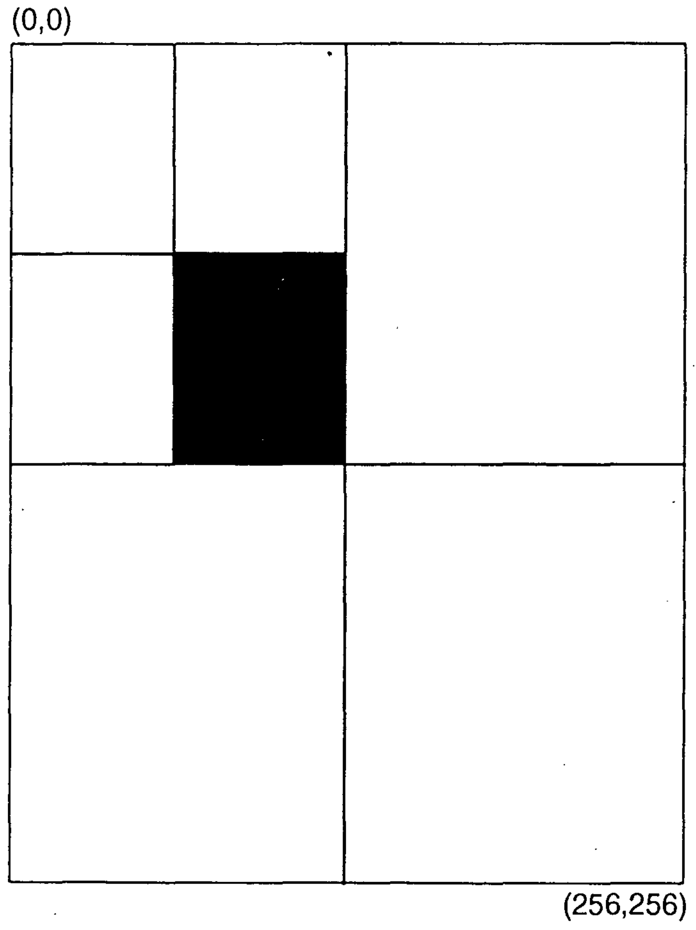
Give the x value of point 3 : 79

Give the y value of point 3 : 79

Give the x value of point 4 : 64

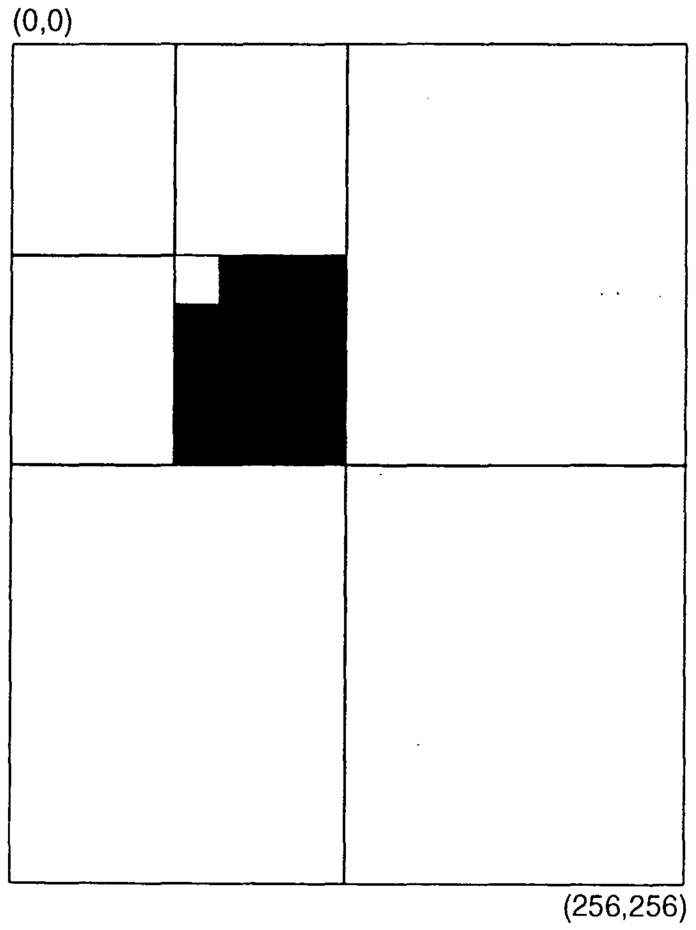
Give the y value of point 4 : 79

BUILDING A QUADTREE (INSERTION)



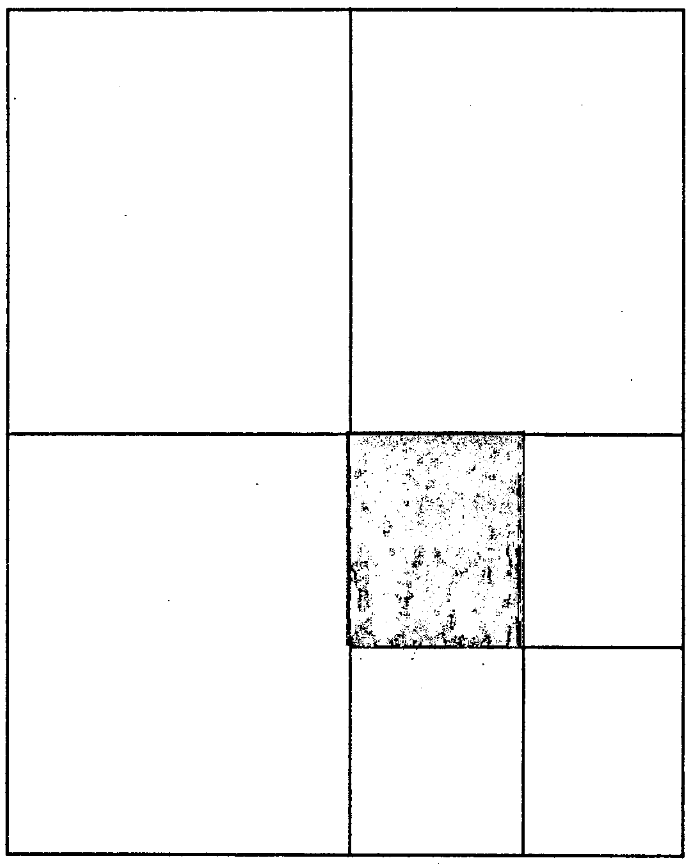
```
2    2    0    0    0    1    0    0    0

20 20 256 25 1 2

Give the id of object 1

Give the number of points of polygons : 4

Give the x value of point 1 : 64

Give the y value of point 1 : 64

Give the x value of point 2 : 127

Give the y value of point 2 : 64

Give the x value of point 3 : 127

Give the y value of point 3 : 127

Give the x value of point 4 : 64

Give the y value of point 4 : 127
```

DELETION OF IMAGE (QUADTREE) (AFTER DELETION)



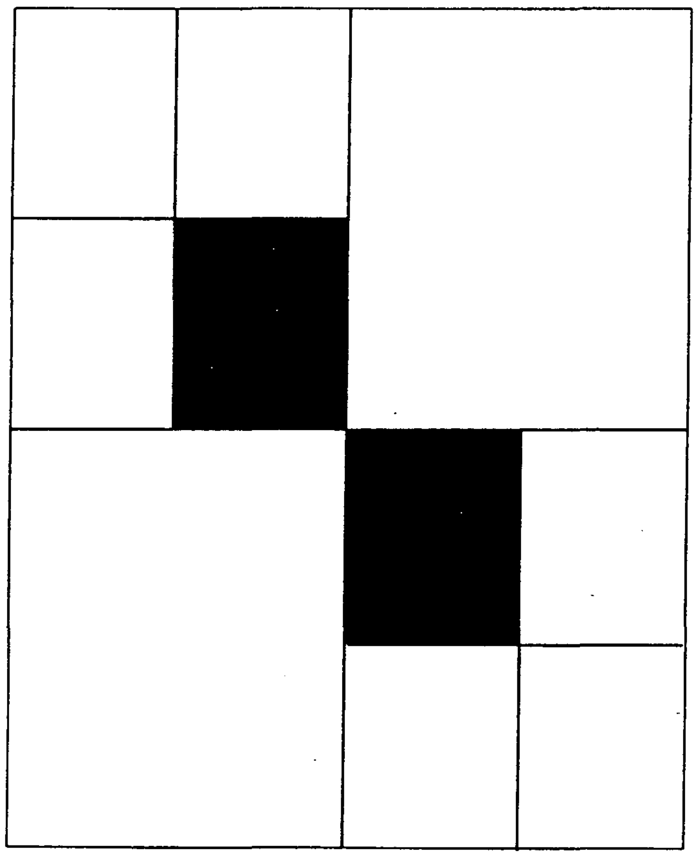
2 2 0 0 0 2 2 0 1 1 1 1 1 1 0 0 0

IMAGE TO BE OVERLAYED

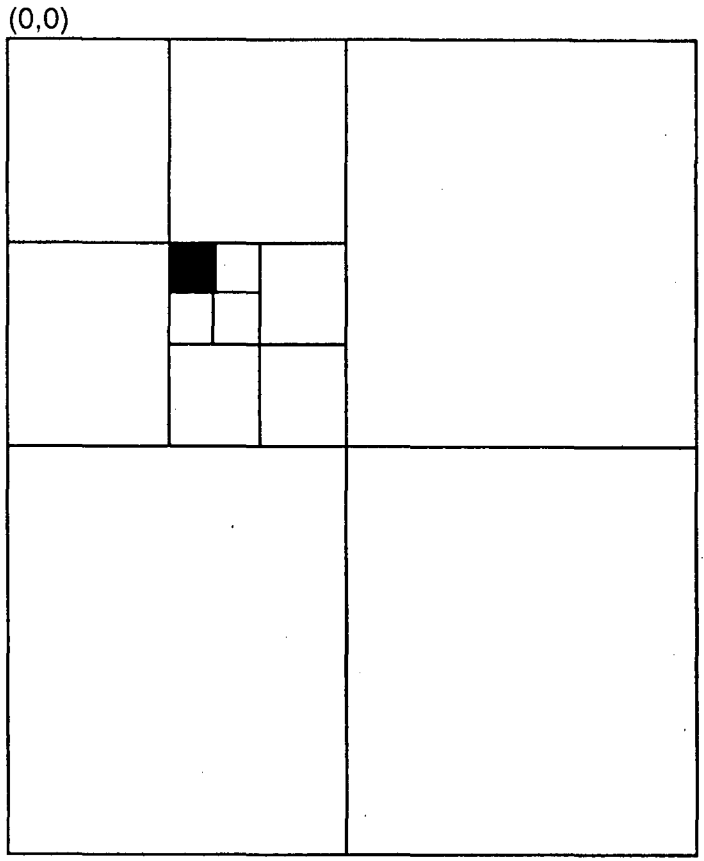


```
2  0  0  0  2  1  0  0  0

Give the id of object 2

Give the number of points of polygons : 4

Give the x value of point 1 : 128

Give the y value of point 1 : 128

Give the x value of point 2 : 191

Give the y value of point 2 : 128

Give the x value of point 3 : 191

Give the y value of point 3 : 191

Give the x value of point 4 : 128

Give the y value of point 4 : 191
```

OVERLAYING OF TWO IMAGES (AFTER OVERLAY)



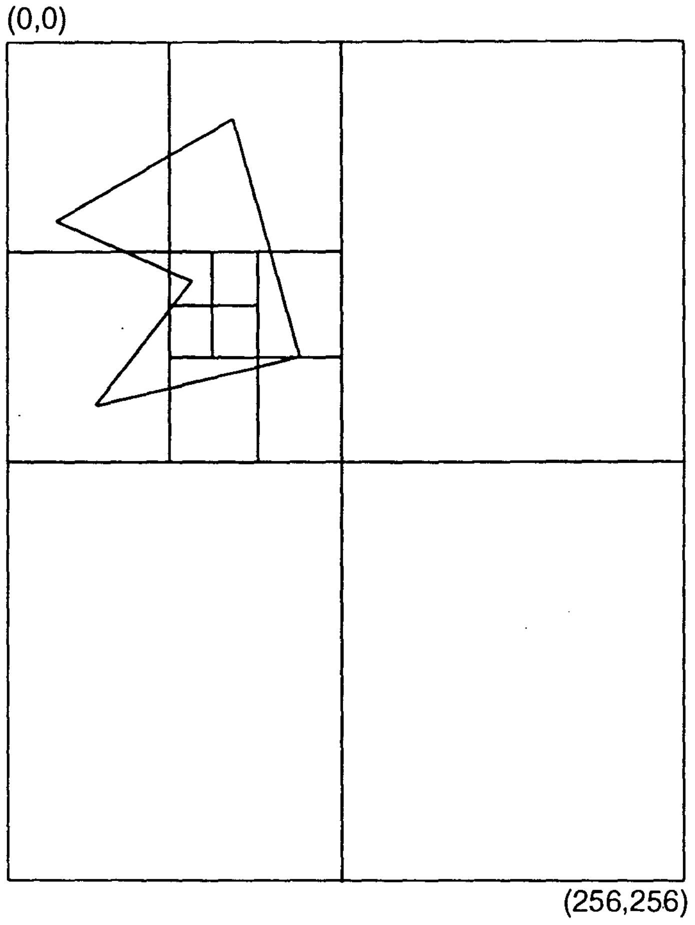
2 2 0 0 0 1 0 0 2 1 0 0 0



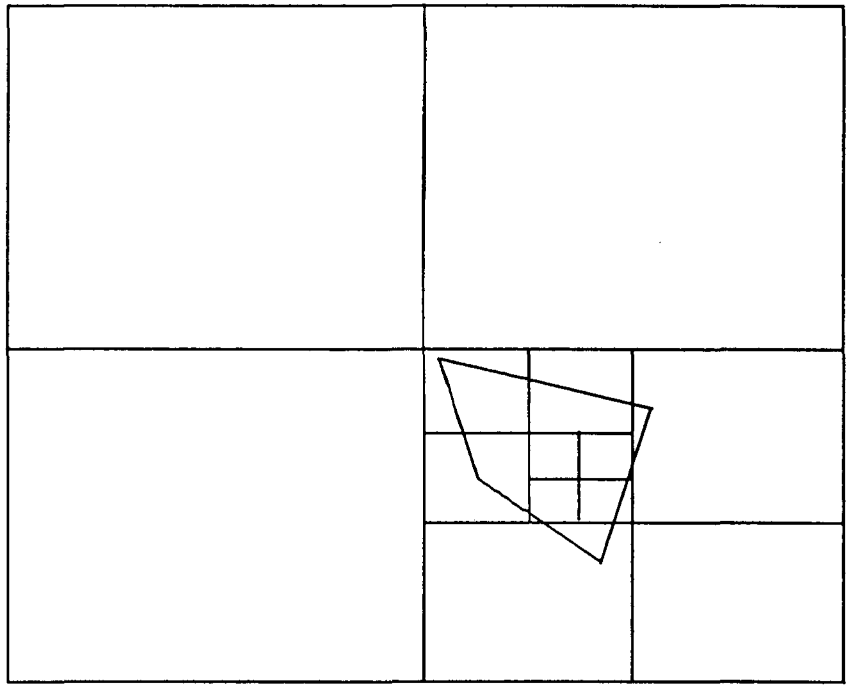
POINT LOCATION

```
2 2 0 0 0 2 2 1 0 0 0 0 0 0 0 0 0
     give x value and y value
     70 70
     Id is 1
```

PM3 QUADTREE (INSERTION)

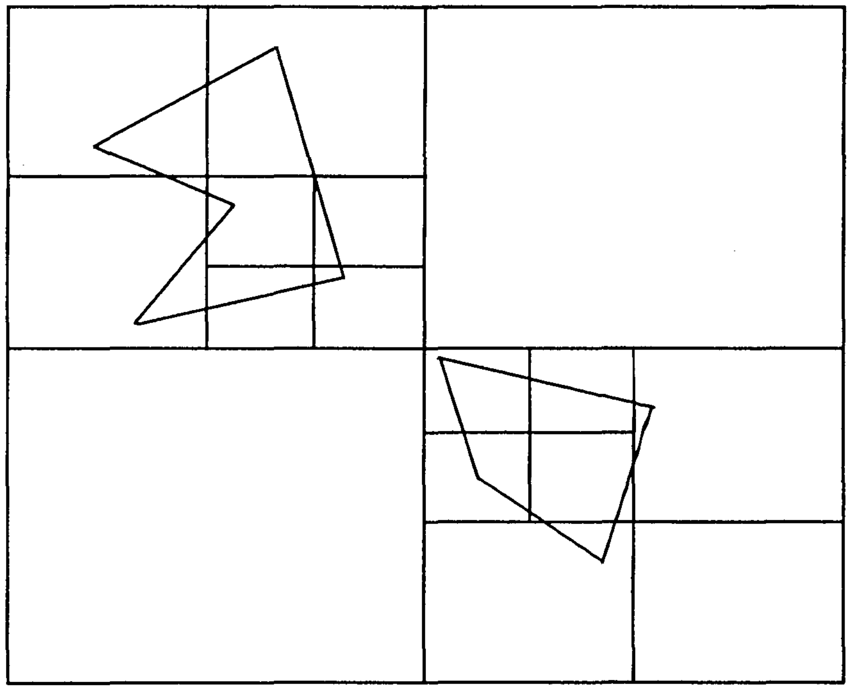


```
2 2 0 0 0 0 2 0 0 0 0 0 0 0

Give the number of lines of nonintersecting nature.

5

Give the values of x1, y1, x2, y2, left, right of line 20 50 100 30 1 2

Give the values of x1, y1, x2, y2, left, right of line 200 30 120 100 2 1

Give the values of x1, y1, x2, y2, left, right of line 120 100 40 120 2

Give the values of x1, y1, x2, y2, left, right of line 40 120 80 80 1

Give the values of x1, y1, x2, y2, left, right of line 80 80 20 50 2 1
```



PM3 QUADTREE (INSERTION)

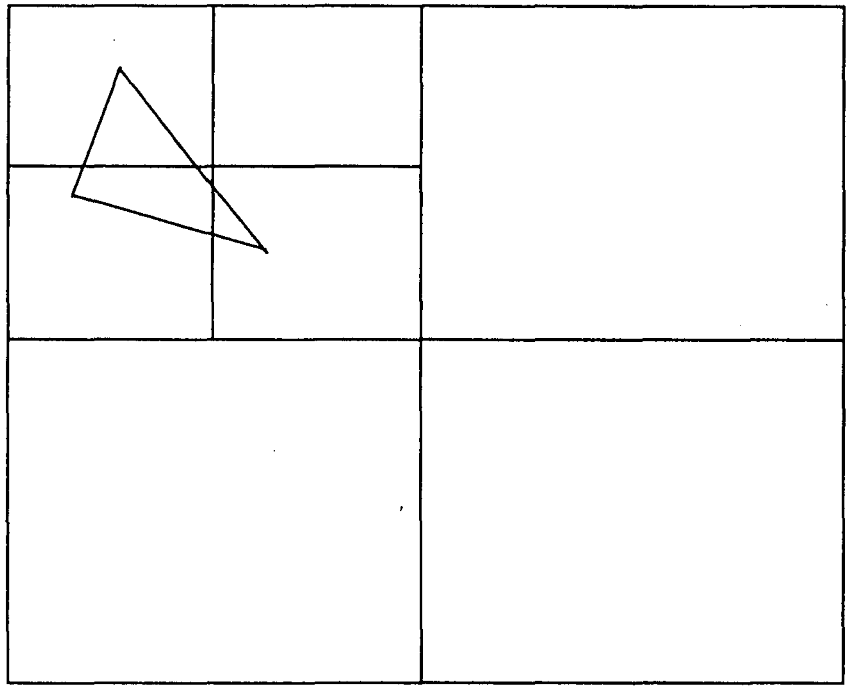
Give the number of lines of nonintersecting nature 4
Give the values of x1, y1, x2, y2, left, right of line 130 130 200 140 1 2
Give the values of x1, y1, x2, y2, left, right of line 200 140 180 200 2 1
Give the values of x1, y1, x2, y2, left, right of line 180 200 140 180 2 1
Give the values of x1, y1, x2, y2, left, right of line 140 180 130 130 1 2



OVERLAY OF PM3 QUADTREES

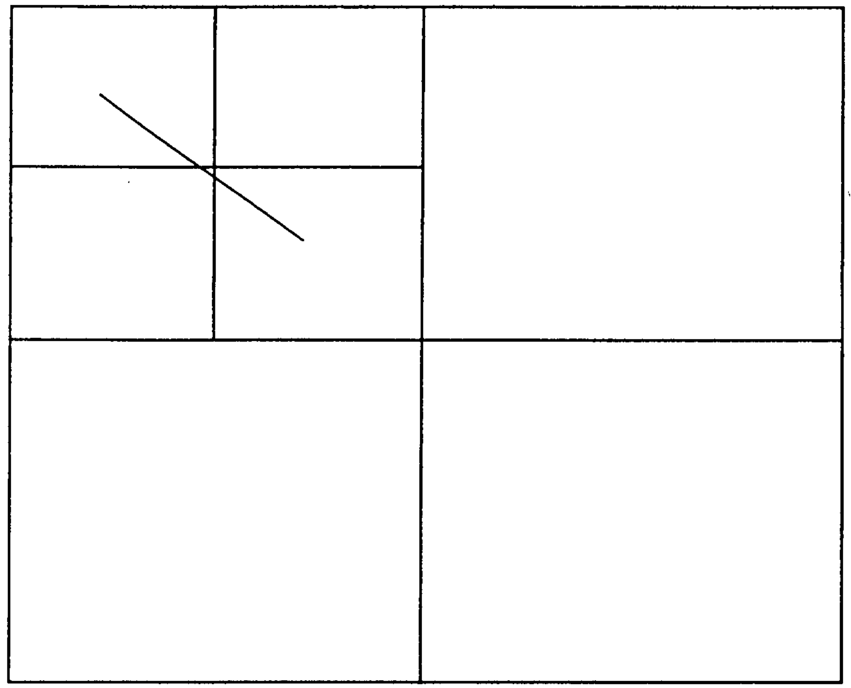
2 2 0 0 0 2 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0

PM3 QUADTREE DELETION (ORIGINAL IMAGE)

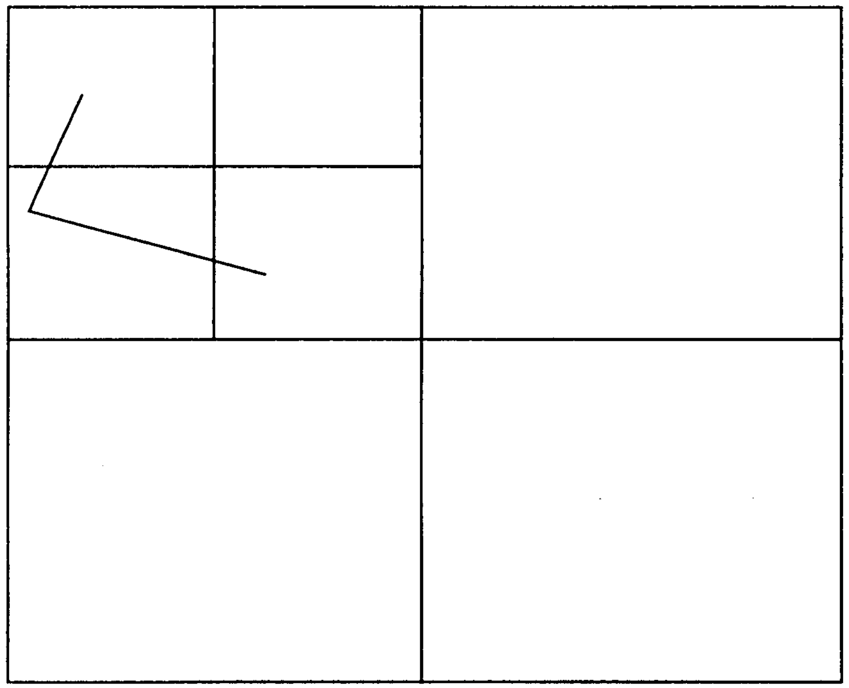


```
2  2  0  0  0  0  0  0  0

Give the x1, y1, x2, y2 region left and right values 20 30 100 100 1 2
Give the x1, y1, x2, y2 region left and right values 100 100 10 80 2 1
Give the x1, y1, x2, y2 region left and right values 20 30 10 80 2 1
```

PM3 QUADTREE (IMAGE TO BE DELETED)

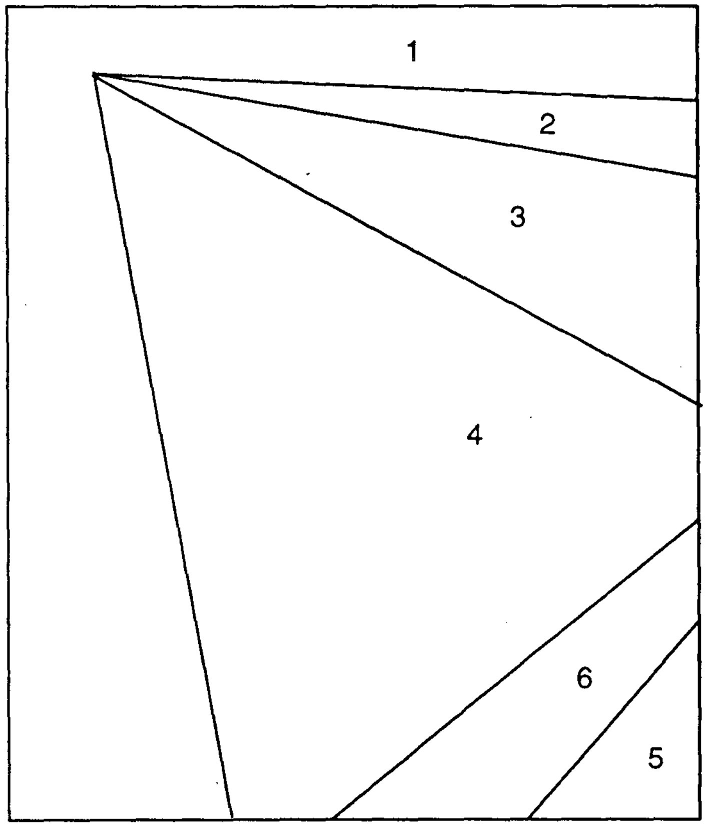


give the x1, y1, x2, y2 values of deleting line
20 30 100 100



POINT QUADTREE (IMAGE AFTER DLELETION)

POINT LOCATION IN PM3 QUADTREE



```
Give the number of lines of nonintersection nature 6
Give the values of x1, y1, x2, y2 left and right region of line 20 20 256 25 1 2
Give the values of x1, y1, x2, y2 left and right region of line 20 20 256 50 2 3
Give the values of x1, y1, x2, y2 left and right region of line 20 20 256 100 3 4
Give the values of x1, y1, x2, y2 left and right region of line 20 20 50 256 1 4

Give the values of x1, y1, x2, y2 left and right region of line 256 150 100 256 4 6
Give the values of x1, y1, x2, y2 left and right region of line 256 200 200 256 5 6 6
        give value of x and y 250 250
        Region is 5
```