

Portland State University

PDXScholar

Dissertations and Theses

Dissertations and Theses

1986

Quadtree-based processing of digital images

Ramin Naderi

Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds



Part of the [Electrical and Computer Engineering Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Naderi, Ramin, "Quadtree-based processing of digital images" (1986). *Dissertations and Theses*. Paper 3590.

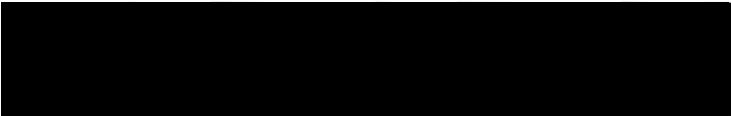
<https://doi.org/10.15760/etd.5474>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

AN ABSTRACT OF THE THESIS OF Ramin Naderi for the Master of
Science in Engineering: Electrical and Computer

Title: Quadtree-Based Processing of Digital Images

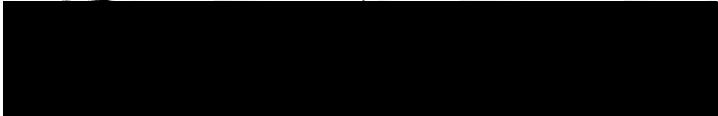
APPROVED BY MEMBERS OF THE THESIS COMMITTEE:



Dr. F. Badi'i, Chairman



Dr. C. Wu



Dr. M. Perkowski

Region growing is the first problem considered. It is assumed that the image is available only in quadtree form. An algorithm that detects all connected, uniform regions of the image is introduced. Quadtree representation of the image is used as input to the algorithm. The procedure assumes no knowledge of the pixel representation of the picture.

Next, computation of orthogonal transforms based on quadtree representation of the image is explored. A

parallel architecture for computation of such transforms is presented. A new type of content addressable memory is utilized in the system. The architecture requires N processors for parallel computation of N point image. Steps needed for this process are $\log_2 N$. The architecture is adaptable for computation of different families of orthogonal transforms.

QUADTREE-BASED PROCESSING OF DIGITAL IMAGES

by

RAMIN NADERI

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
Engineering: Electrical and Computer

Portland State University

1986

TO THE OFFICE OF GRADUATE STUDIES AND RESEARCH:

The members of the Committee approve the thesis of
Ramin Naderi presented June 11, 1986.

[REDACTED]
Dr. F. Badi'i, Chairman

[REDACTED]
Dr. ~~Q.~~ Wu

[REDACTED]
Dr. M. Berkowski

APPROVED:

[REDACTED]
Dr. ~~Peter Frick~~, Head, Department of Electrical Engineering

[REDACTED]
Dr. Bernard Ross, Dean of Graduate Studies and Research

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor Dr. Badi'i who guided me through this thesis project by giving me valuable suggestions and ideas. I would also like to thank Dr. Ghafarzadeh for his helpful suggestions during the development of my thesis.

Special thanks are due my parents for their support and encouragement throughout my education.

TABLE OF CONTENTS

	PAGE
ACKNOWLEDGEMENTS	iii
LIST OF FIGURES	v
CHAPTER	
I INTRODUCTION	1
II REGION-GROWING USING THE QUADTREE	4
Background: Quadtrees	4
Region-Growing	7
Algorithm	9
Results	39
III COMPUTATION OF ORTHOGONAL TRANSFORMS USING QUADTREES	41
Background	41
Region Search Memory	42
Hardware Configuration	44
Processing Elements	44
Memory	46
Interconnection Network	49
System Operation	53
Conclusion	54
REFERENCES	56
APPENDIX	58

LIST OF FIGURES

FIGURE	PAGE
1. An Image, Its Maximal Blocks, and the Corresponding Quadtree	6
2. Intermediate Trees in the Process of Obtaining a Quadtree	8
3. Structure Chart of the Design	10
4. Examination of a Quadtree	17
5. Copying the Path of a Node from One List to Another	22
6. Conversion of the Path on the Quadtree to the Corresponding I,J in the Maximal Block Diagram	29
7. A Node on the Maximal Block Diagram and Its Neighbors	32
8. Adjustment of the Neighbors of a Node on the Maximal Block Diagram	36
9. The Region Growing Operation	40
10. A Flow Graph for Computation of an 8-Point DFT	43
11. A Hardware System for Computation of Orthogonal Transforms	45
12. A Node on a Quadtree and Its Associated Properties	47

13.	A PE Configuration	48
14.	A PEM Configuration	50
15.	A Comparator Element	51
16.	A Greater-than Block	52

CHAPTER I

INTRODUCTION

Image representation plays an important role in image processing applications, which usually contain a huge amount of data. An image is a two-dimensional array of points, and each point contains information (eg: color). A 1024 by 1024 pixel image occupies 1 mega byte of space in the main memory. In actual circumstances 2 to 3 mega bytes of space are needed to facilitate the various image processing tasks. Large amounts of secondary memory are also required to hold various data sets.

There are, in general, two types of data compression techniques in image processing. One approach is based on elimination of redundant data from the original picture. Other techniques rely on higher levels of processing such as interpretations, generations, inductions and deduction procedures [1, 2]. One of the popular techniques of data representation that has received a considerable amount of attention in recent years is the quadtree data structure. This has led to the development of various techniques for performing conversions and operations on the quadtree.

Klinger and Dyer [3] provide a good bibliography of the history of quadtrees. Their paper reports experiments on

the degree of compaction of picture representation which may be achieved with tree encoding. Their experiments show that tree encoding can produce memory savings. Pavlidis [15] reports on the approximation of pictures by quadrees. Horowitz and Pavidis [16] show how to segment a picture using traversal of a quadtree. They segment the picture by polygonal boundaries. Tanimoto [17] discusses distortions which may occur in quadrees for pictures. Tanimoto [18, p. 27] observes that quadtree representation is particularly convenient for scaling a picture by powers of two. Quadrees are also useful in graphics and animation applications [19, 20] which are oriented toward construction of images from polygons and superpositions of images. Encoded pictures are useful for display, especially if encoding lends itself to processing.

In this thesis, two different operations on the quadtree are presented.

A review of the quadtree data structure and its utilization in image processing is presented in Chapter II. Region growing in the context of digital pictures is also discussed. An algorithm which takes the quadtree data structure as input and detects uniform connected regions of the image is presented. This algorithm is useful because it is independent of the original image. It takes the quadtree representation of the image and performs the region growing

operation. This algorithm alleviates the need for reconstruction of the original image by performing region growing in the quadtree structure.

In the last section of Chapter II some experimental results are presented. The actual code written to implement the algorithms is included in Appendix A.

In Chapter III, orthogonal transforms are used to convert the signal to another form which is more desirable to operate on [9, 11]. Transforms of any kind generally involve a large number of operations. Hence, single processor digital computers do not lend themselves well to this application, and special purpose parallel processors must be designed which reduce the computation time by performing the required operations in parallel instead of in sequence [10, 14].

Basic orthogonal transforms and their application in the field of image processing are discussed. Families of orthogonal transforms are reviewed. A VLSI architecture for computation of various orthogonal transforms is introduced. The architecture again assumes that the input image is stored in a quadtree format. The system is designed so that it is slice expandable. The PE interconnection network, which is the subject of another report, is intended to be a reconfigurable one to permit implementation of different types of transforms.

CHAPTER II

REGION GROWING USING THE QUADTREE

BACKGROUND: QUADTREES

Image representation is considered very important in the field of image processing. Recently there has been considerable interest in the use of quadtrees for image processing. This has led to the development of algorithms for performing image processing tasks as well as conversions between the quadtree and other representations. In this chapter a region growing algorithm is developed which uses quadtree representation of the image as input.

In this thesis quadtree representation of binary arrays [3, 4] is used. The quadtree structure is useful because it provides hierarchical representation and facilitates operations. A quadtree for representing an image is a tree in which successively deeper levels represent successively finer subdivisions of image areas. Leaves of a quadtree represent areas of the image. Each leaf is labeled with the property (eg: color) of the area of the image it represents. Each node is associated with a square region of the image. An algorithm for generating a quadtree from the two dimensional binary array is discussed below.

It is assumed that the image is a 2^n by 2^n array of binary pixels. The quadtree is an approach to image representation based on successive subdivision of the array into quadrants. In essence, the array is repeatedly subdivided into quadrants until blocks are obtained (possibly single pixels) which consist of entirely the same type of pixels. This process is represented by a tree of out degree 4 in which a root node represents the entire array, the four sons of the root node represent the quadrants, and terminal nodes correspond to those blocks of the array for which no further subdivision is necessary. For example, Figure 1b is a block description of the region in Figure 1a, while Figure 1c is the corresponding quadtree. In general, black and white square nodes represent blocks consisting entirely of 1's and 0's respectively. Circular nodes are termed gray.

Each node in the quadtree is stored as a structure containing seven fields. Of these seven, five are pointer fields which are the node's father and its four sons, labeled NW, NE, SW, SE. The sixth field is the node type which describes the contents of the image block which the node represents, i.e., white, black, or gray. The seventh is the field describing the region the node belongs to.

The quadtree construction algorithm examines each pixel in the binary array only once. A node is created if it is

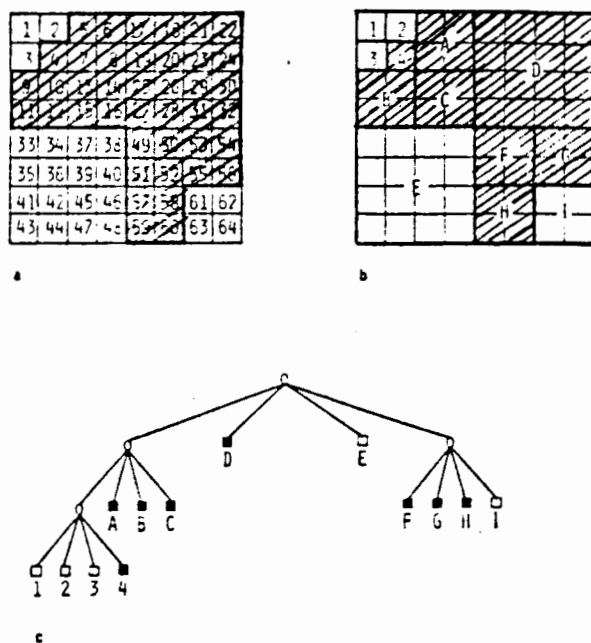


Figure 1. An image, its maximal blocks, and the corresponding quadtree.

Blocks in the image are shaded: (a) sample image,
(b) block decomposition of the image in (a),
(c) quadtree representation of the block in (b).

maximal, that is, if it cannot participate in any other merges. For example, Figure 2a shows the partial quadtree resulting from pixels 1, 2, 3, and 4 of Figure 2a. Note that since not all pixels are of the same type, their node cannot participate in any further merges. In contrast, pixels 5, 6, 7, and 8 of Figure 1a are of the same type, and thus they will be represented by node A in the final quadtree. No nodes are ever constructed corresponding to these pixels. As a final example of a merge, the Figure 1c quadtree shows that pixels 17-32 are ultimately represented by node D. However, the node corresponding to those pixels is only created once; its remaining brothers have been processed (i.e., pixels 33-48 and 49-64). This is in contrast with the gray node corresponding to pixels 1-16, which was created as soon as it was determined that its four sons are not all white or all black. The algorithm used for construction of the quadtree is implemented as a part of the program in Appendix A.

REGION GROWING

Region growing is an approach to detect connected regions composed of same-type pixels. A typical region-growing process includes assigning distinct labels to distinct regions detected. A pixel is given a label. Then all neighboring pixels are examined. If at least one

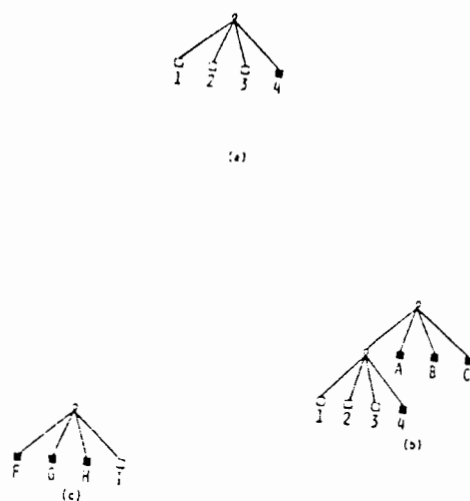


Figure 2. Intermediate trees in the process of obtaining a quadtree.

neighboring pixel meets the criteria for inclusion in the same region, the region grows. Otherwise growth of the region is terminated and new regions are defined [5, 6]. The process terminates whenever all pixels of the image have been assigned to a region. Of particular interest is the assignment of labels to regions consisting of pixels having the same property using the quadtree structure as input.

ALGORITHM

In this section the algorithms needed for implementation of region-growing are presented. In these algorithms an eight-neighbor check technique is used to find the neighbors. A top-down design technique was used to design this algorithm in which the procedure was subdivided into smaller sub-algorithms in order to accomplish the task.

Before the algorithm is explained in detail, an overview of task accomplishment is presented. Figure 3 shows the diagram representing the design of the algorithm. The algorithm is invoked by calling the label-region. It uses the find-depth routine to find the depth of the tree for size adjustment. As long as there are unlabeled nodes on the tree, the label-region gives them a label and calls the get-neighbor routine to find the neighbors of the node in question. The get-neighbor routine calls the IJ to convert the path of the node in question to the

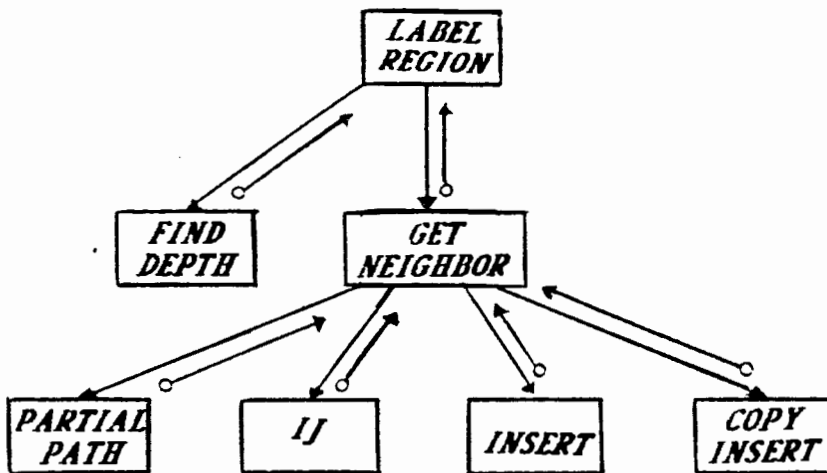


Figure 3. Structure chart of the design.

corresponding block in the maximal block diagram. Next, get-neighbor calls partial-path to find the neighbors of the same size as the node in question using the maximal block diagram. Then it uses the insert and copy insert routines to make adjustment for neighbors with smaller sizes.

The main algorithm is label-region, which takes the root of the quadtree as its argument. There are two lists used in this algorithm which are dynamically allocated. Each of them is a structure with two fields. One is a pointer to the path structure which holds the path of the particular node on the tree, and the other is a pointer to the next element in the list. Nodes in the open list are generally nodes whose property is different from the current node property being processed. These nodes are saved in the open list for further examination. Nodes in the closed list are nodes which have the same property as the current node property being processed, so they are saved in the closed list for further examination. These two lists are the base of this algorithm.

The algorithm first starts by finding the depth of the quadtree. The routine written for this purpose is find-depth (which will be presented later), algorithm A. The depth information is needed for later use. After finding the depth of the quadtree, the smallest SW node on the quadtree is placed in the open list. (This is done

arbitrarily; any node could have been chosen). A while-loop exists, which runs as long as there is a node in the open list and terminates the algorithm when all nodes are gone from the open list. Obviously, taking the smallest SW node on the quadtree as explained above, a node exists in the open list to start with. The first node is moved from the open list to the closed list for further examination. Algorithm C, copy-path, is used to accomplish this movement.

Get-neighbor is an algorithm which finds the neighbors of the node in process; it will be presented and discussed in detail in algorithm D. After finding the neighbors in the form of a list similar to the open and closed lists which is returned by the get-neighbor routine (called neighbor-list), the neighbors are checked one by one to see whether they have already been assigned a label. The node in the neighbor list matches one of the following conditions: (a) it has already received a label, in which case the algorithm moves on to the next neighbor; (b) it is the same type as the node being processed, in which case it receives the current label and is transferred into the closed list; or (c) it is not the same type as the node being processed, in which case it is transferred into the open list.

Algorithm C is used for copying nodes from one list to another, which is used throughout the algorithm. The

process of finding neighbors for the nodes in the closed list continues as long as there are nodes in this list. This causes the current region to grow until all the nodes with the current property are found and assigned to the current region. Then another node is taken from the open list and checked to see whether or not it has a label. If it does, it has already been processed. If it does not, it is put in the closed list, and the process is then repeated with a different region number. The algorithm terminates when the open list is empty, i.e., when all the nodes are labeled. Note: In order to find a node on the tree to check its property, the find-node routine is used. This routine is explained in algorithm B. The nodes involved were created dynamically by the algorithm and its sub-algorithms in C language. This dynamic creation frees the memories in use whenever a node is processed and no longer needed, which in turn, decreases the amount of storage requirements.

```

label_region(root of the quadtree)
{
  find_depth(root);
  put an arbitrary node into the open list;
  region_number=1;
  while(there are nodes in open list)
  {
    transfer a node from open list to close list;
    while(there are nodes in closed list)
    {
      take a node and find neighbors;
      for(all neighbors)
      {
        if(neighbor has a label)
          process next neighbor;

        if(neighbor has the same property)
        {
          label it with current region_number;
          put it into closed list for further processing;
        }/*if*/
        else
          put it into open list for further processing;
      }/*for*/
      region_number++;
    }/*while*/
  }/*while*/
}/*end label_region*/

```

Algorithm A is designed to find the deepest level existing on the quadtree, i.e., the smallest node on the tree will have the deepest level. The algorithm takes the root of the quadtree as its argument and uses a recursive method to find the deepest level. Starting with the root, the algorithm recursively calls itself. In each iteration, a son is taken as the root node. The process continues until all sons have been accounted for.

Upon returning at the top level (first time function was called), the array termed length will contain the results of the search of four sons. The largest one is chosen. Figure 4a shows an example of a quadtree. Its deepest level is five, and it is obtained through the NW son of the root. Levels are shown on the quadtree for better understanding.

Algorithm A

```
find_depth(root)
{
  if (tree has no sons)
    return (depth=1);

  else
  {
    for (all sons)
      length[son]=find_depth(son as root); /*recursive*/
    } /*else*/
  return (depth=son with largest length+1);
} /*end find_depth)
```

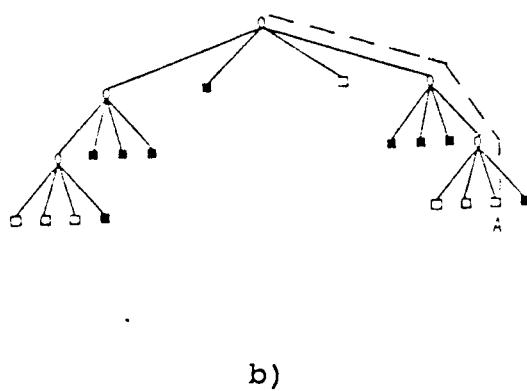
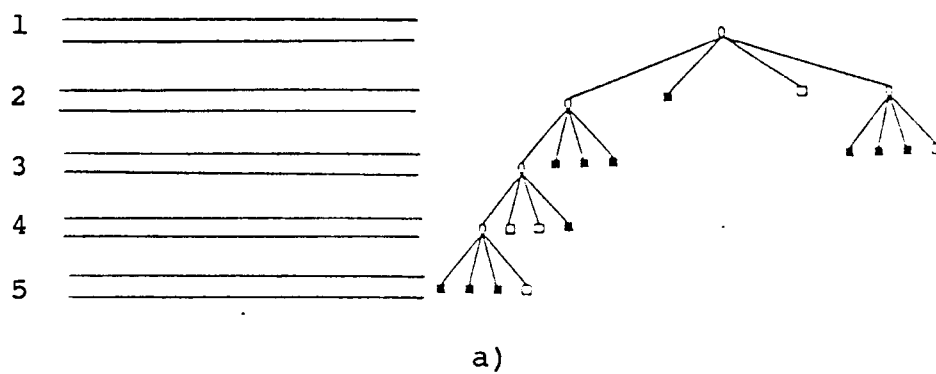


Figure 4. Examination of a quadtree.

Algorithm B is the find-node algorithm. Arguments passed to this routine are paths to a node on the quadtree and the root of the quadtree. Depending on the value of the path at each stage, it will take the pointer to that particular one to reach the desired node. The example in Figure 4b shows how a particular node is reached having the path to that node. If stages on the path are more than what actually exist on the quadtree, the rest of the stages are ignored. (This will be clarified when a discussion on the get-neighbor algorithm is presented.) For example, using the Figure 4b quadtree, the path to a particular node may be 3, 3, 2. In this instance, starting from the root, the first pointer used in the node structure is 3, followed by pointer 3 again, followed by the second pointer, the latter of which will result in node A of Figure 4b. This route is illustrated in Figure 4b.

Algorithm B

```
find_node(path to the node,root)
{
  while(path is not exhausted)
    go to the son indicated by the path; /*start at root*/

  return(a pointer,pointing to node in question);
}/*end find_node*/
```

Algorithm C is the copy-path algorithm. Since each node on the quadtree is referred to by the path to that node, for the purpose of copying a node from one list to another, the path of the node from one list is simply copied to another. Later, that node in the new list can simply be referred to by its path. A simple example of this is shown in Figure 5 in which both a source and destination list are shown. The path shown with a dotted circle in Figure 5a is copied to the place marked by a dotted circle in Figure 5b. In order to do this, each stage is copied from the first stage of the source path to the destination path, so long as there are items on the source path.

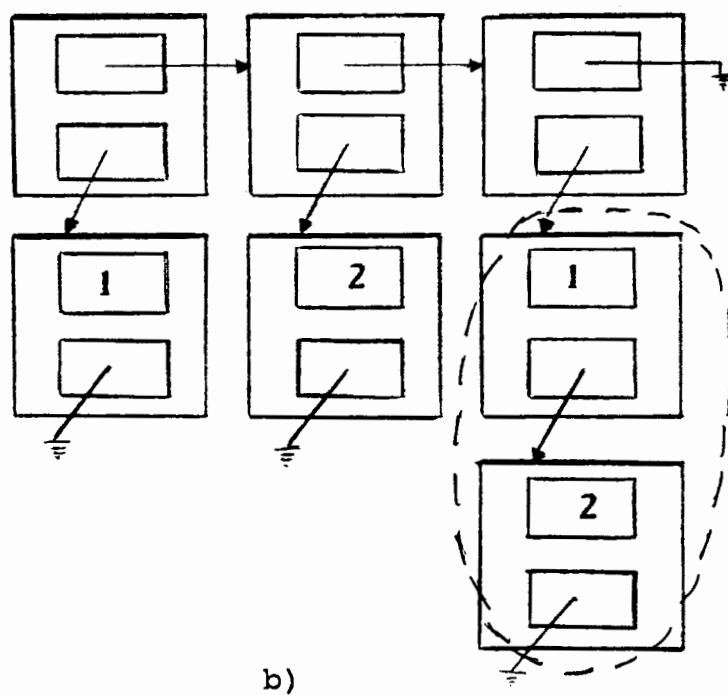
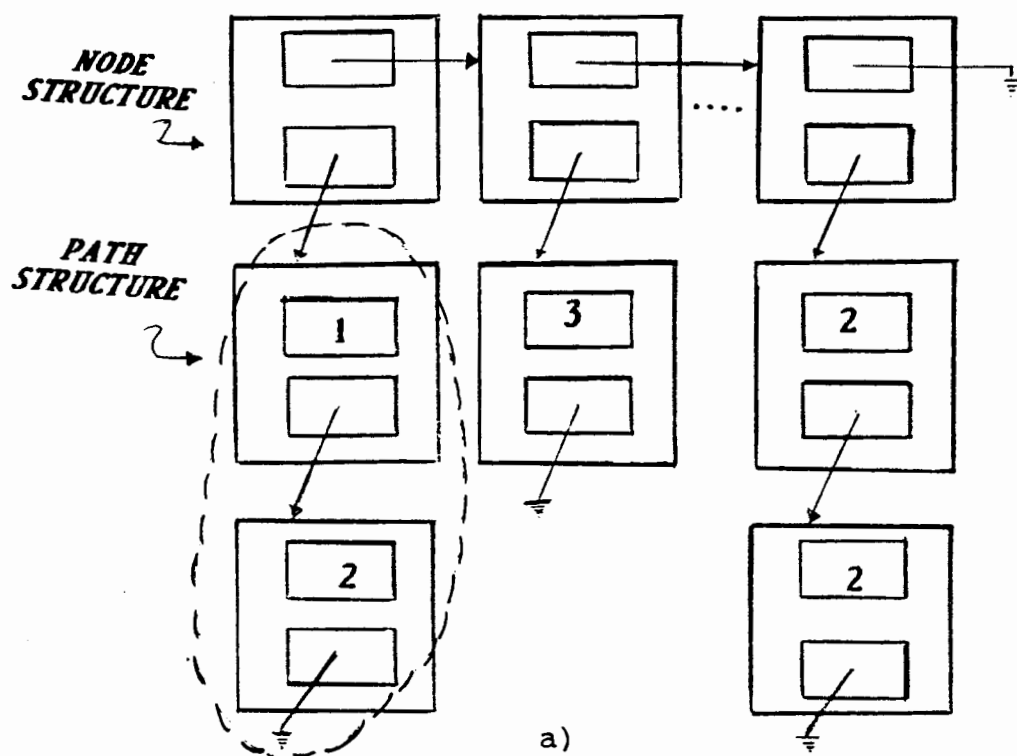


Figure 5. Copying the path of a node from one list to another.

Algorithm C

```
copy_path(destination,source)
{
  while(source path is not exhusted)
    add current value in source path to destination path;
}/*end copy_path*/
```

Algorithm D is the get-neighbor routine. Having a path which corresponds to a particular node on the quadtree as its argument, this routine will find the neighbors for that node. This algorithm is broken down into four smaller algorithms for better implementation. These sub-algorithms will be presented later in this discussion.

Since each node on the quadtree represents a block in the maximal blocks diagram, each node on the quadtree can be matched to an equal representation on the maximal blocks diagram. When this routine is invoked, a pointer to a path is passed to it. This path, as discussed before, corresponds to a node on the quadtree. Routine IJ (as will be discussed in more detail in algorithm D.1) converts the corresponding path to equivalent representation in the maximal blocks diagram, consisting of a row and a column (I, J). From there, the neighbors of the node can be found in the maximal blocks diagram by using the eight-neighbor check technique, in which blocks immediately around the block in question are examined. This is explained in detail in the partial-path algorithm (Algorithm D.2). It should be clear by now that nodes in quadtrees or the maximal blocks diagram are of different sizes. Some adjustment is thus needed to resolve this problem. When the partial-path routine is invoked to find the neighbors of a particular node, it only finds neighbors that are of the same size as

the node in question (hence the name partial path). The algorithms insert and copy-insert (D2, D3) are used to take care of this problem. Note that if all the neighbors are of the same size, partial-path will return neighbors which do not need any alteration. But if some of the neighboring nodes are smaller, each node found by partial-path will be further broken down into more nodes and adjustment to their paths will be made, using Algorithm D.1, D.2). Examples that will follow the sub-D algorithms should resolve questions which have likely arisen here.

Algorithm D

```

get_neighbor(node in question)
{
    /*find I,J,path_length,depending on the node*/
    path_length=IJ(I,J,node in question);
    /*find partial paths of the neighbors*/
    top_neighbor=partial_path(I,J,path_length,indicator);
    NOX=MAX_DEPTH-path_length;
    if(node is one of smallest nodes on the quadtree)
        return(neighbors found);/*no adjustment needed*/

    else
    {
        for(all existing neighbors)
        {
            switch(neighbor)
            {
                case 1: insert(3,NOX,neighbor);
                        break;

                case 2: copy_insert(2,3,NOX,neighbor);
                        break;

                case 3: insert(2,NOX,neighbor);
                        break;

                case 4: copy_insert(1,3,NOX,neighbor);
                        break;

                case 5: copy_insert(0,2,NOX,neighbor);
                        break;

                case 6: insert(1,NOX,neighbor);
                        break;

                case 7: copy_insert(0,1,NOX,neighbor);
                        break;
            }
        }
    }
}

```

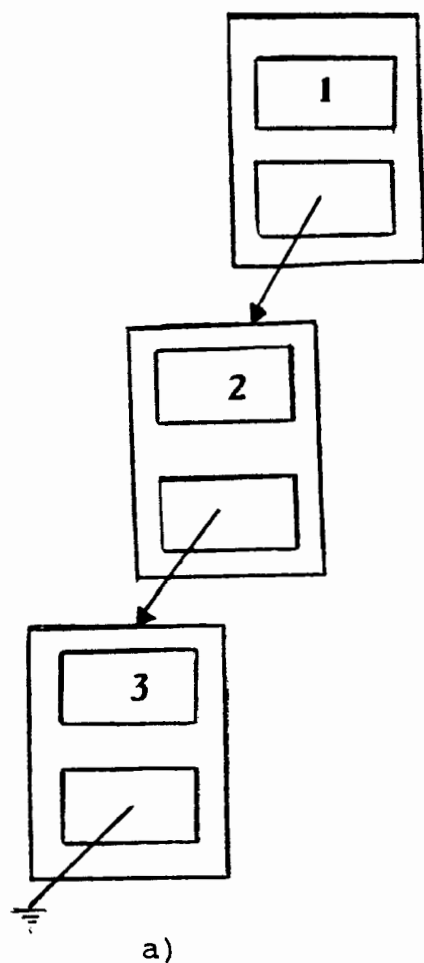
```
        case 8: insert(0,NOX,neighbor);
                break;

        }/*switch*/
    }/*for*/
}/*else*/
return(head of neighbor list);
}/*end get_neighbor*/
```

Algorithm D.1 is the IJ routine. It converts the node on the quadtree specified by the path passed to it to corresponding I and J, which will address the row and column of the equivalent block in the maximal block diagram. In each stage, that portion of the path is broken into an I and a J element. Then all the I and J elements obtained from all the stages are put together to form the resulting I and J. This routine will also return the level of the path in question to the calling routine. The example given in Figure 6 demonstrates how paths 1, 2, and 3 of the node on the quadtree are converted to corresponding I and J in the maximal blocks diagrams, 3 and 5 respectively.

Algorithm D.1(IJ)

```
IJ(I,J,path)
{
  path_length=0;
  while(path is not exhausted)
  {
    add second bit of current path value to I;
    add first bit of current path value to J;
    path_length++;
  }/*while*/
  return(path_length);
}/*end IJ*/
```



PATH = 1, 2, 3

b)

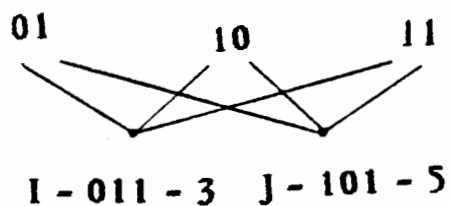


Figure 6. Conversion of the path on the quadtree to the corresponding I,J in the maximal block diagram.

Algorithm D.2 is the partial-path routine. It finds the existing neighbors of the node in question. The row and column of the node in the maximal blocks diagram and an array termed indicator which will indicate the existing neighbors for later use and the length of the path of the node in question are the arguments to this routine. It will find the neighbors of the same size as the node in question and will return a list of neighbors which are the paths to those neighbors on the quadtree to the calling routine. Note that the node in question is passed to this routine as row and column of a block in the maximal blocks diagram, but neighbors are returned as paths of the nodes on the quadtree. For example, to find the neighbors of C in Figure 7a, the list of neighbors is obtained as shown in Figure 7b. As stated earlier, before partial-path will find the neighbors of the same size and larger sizes, the problem of smaller nodes must be resolved. In looking at the first neighbor in the 7b list, four possibilities (1, 2, 3, 4) exist. The only neighbor is 4, but partial-path cannot make that distinction, so an adjustment must be made. This is done, and the adjustment is explained in Algorithms D.1 and D.2, insert and copy-insert respectively.

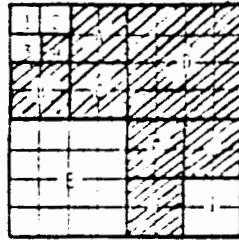
Algorithm D.2(partial_path)

partial_path(I,J,path_length,indicator)

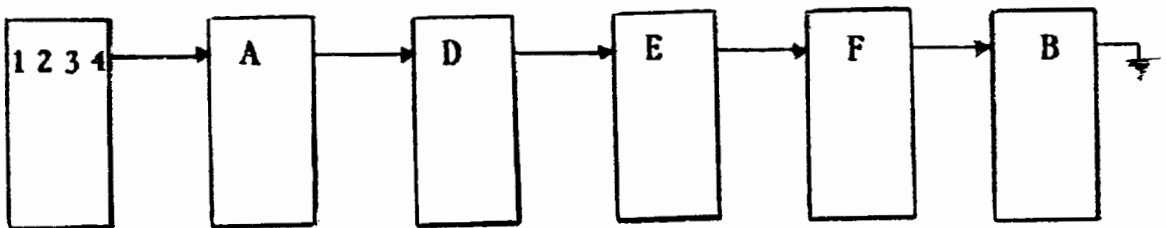
```
{
  for(I-1 to I+1)
  {
    if(I does not exist)
      process next I;

    else
    {
      for(J-1 to J+1)
      {
        if(J does not exist)
          process next J;

        else
        {
          creat a neighbor node and add it to neighbor list;
          depending of I and J make it's path;
        }/*else*/
      }/*for*/
    }/*else*/
  }/*for*/
  return(head of neighbor list);
}/*end partial_path*/
```

a)



b)

Figure 7. A node on the maximal block diagram and its neighbors.

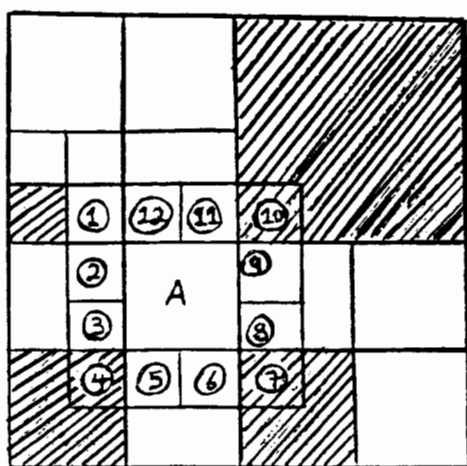
Algorithm D.3 is the insert routine. As was discussed for the previous algorithm, a problem arises whenever there are nodes smaller than the node for which neighbors are being sought. The problem is in the selection of the right neighbor from the group of possible neighbors. Since each node has only a maximum of eight neighbors of its own size, the problem may be narrowed down to a maximum of eight different cases only, which are taken care of by adding the right number to the end of their path. This routine is invoked by the get-neighbor routine. Whenever it is invoked, this routine will add the number passed to it and repeat it an appropriate number of times to the end of the path indicated by ptr, also passed to this routine. This causes the right neighbor to be accessed.

A look at the neighbors of A in Figure 8a shows that there are nodes which are smaller than A. When the neighbors of A are returned from partial-path in get-neighbor, there are eight possible neighbors of the same size as A. Neighbors 1, 4, 7, and 10 can be obtained by simply adding the right number to the end of the neighbor that represents them in the neighbor list. In the case of 1, 3 must be added; in the case of 7, 0 must be added; and so on. Figure 8c shows the path of Figure 8b after adding a number to it. Note that the number of times the number is added to the end of the path is dependent upon the variable

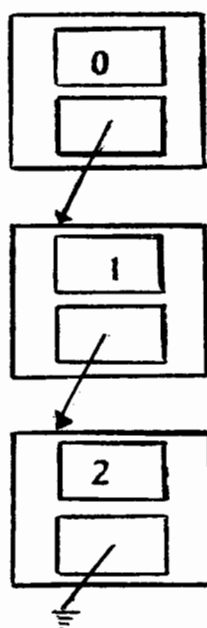
repeat which is passed to this routine. Repeat is the length of the path difference between the current node in question and the smallest node on the quadtree. So, if the node in question is one size larger than the neighbor, the number is added once to the end. As the difference grows, the number is added more times.

Algorithm D.3(insert)

```
insert(number,repeat,path)
{
  for(repeat times)
    add number to end of path;
}/*end insert*/
```



a)



b)

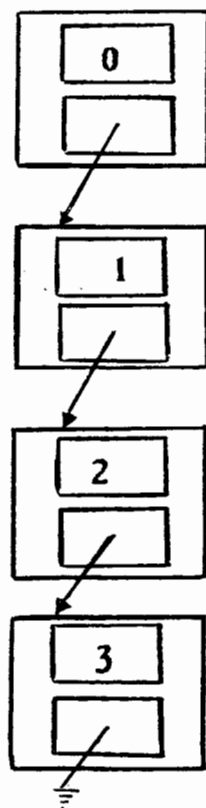


Figure 8. Adjustment of the neighbors of a node on the maximal block diagram.

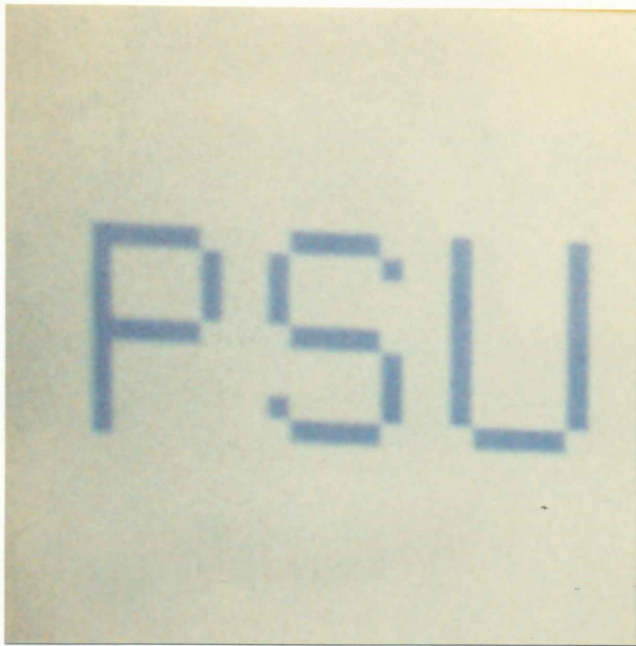
D.4 is the copy-insert routine. In the description of the last algorithm, part of the problem which arose when smaller nodes than the node in question were detected was solved. In the case of 1, 4, 7, and 10 in Figure 8, the appropriate number was added to the end of the path representing that neighbor to solve the problem. Copy-insert takes care of the rest of the neighbors. In finding neighbors for A, 2 and 3 are considered as a part of a big node. Thus, their path must be copied 2 to the power of NOX times; the combinations of number1 and number2 must be added to the end of their path. To accomplish the right nodes, for 2, 1 must be added because the first son is wanted; for 3, 3 must be added to generate the third son, to the end of the path that represents them. Some approach has to be taken towards 5,6 and 8,9 and 11,12. Note that NOX, which is passed to this routine, is the difference in path length between the node in question and the smallest node on the quadtree. So if the node in question is one size larger than the neighbor, 2 to the power of one neighbors are created, and the combination of number1 and number2 is added to the end of their path. As the difference grows, the number of neighbors increases, as does the combination of number1 and number2, which results in a larger path length.

Algorithm D.4(copy_insert)

```
copy_insert(number1,number2,NOX,neighbor)
{
  repeat=pow(2,NOX);
  for(repeat times)
  {
    creat a neighbor node;
    copy path of the current neighbor to created neighbor;
  }/*for*/
  insert(all possible combinations of number1 and number2
         to the end of created neighbor nodes);
}/*end copy_insert*/
```

RESULT

All the algorithms were implemented in C language. These algorithms were examined on a number of test patterns. Theoretical expectations were strongly supported by the experimental results. At the end, the quadtree which was performed on was displayed again. Different images were used to test the validity of the algorithms. It performed rather well. To demonstrate, using Figure 9a as the initial image, after performing the region-growing, Figure 9b was produced in which regions labeled with different colors are evident.



a. An image before region growing.



b. An image after region growing.

Figure 9. The region growing operation.

CHAPTER III

COMPUTATION OF ORTHOGONAL TRANSFORMS USING QUADTREES

BACKGROUND

There has been growing interest regarding the study of orthogonal transforms in the area of digital signal processing [7-11]. Research efforts and related applications of such transforms include image processing, speech processing, pattern recognition and many more. Many picture processing applications require the use of two dimensional signal processing techniques. This is the case in x-ray enhancement, the enhancement and analysis of aerial photographs for detection of forest fires or crop damage, the analysis of satellite weather photos, and the enhancement of television transmission from lunar and deep-space probes. Seismic data analysis as required in oil exploration, earthquake measurements and nuclear test monitoring also utilize multidimensional signal processing techniques [7-9, 12, 13]. The Fourier and the Walsh-Hadamard transforms are examples of such orthogonal transforms.

Algorithms have been developed for fast computation of such transforms, for instance the Fast Fourier Transform, or

FFT, for the Fourier Transform. These algorithms reduce the computation time of these transforms by several orders of magnitude. Figure 10 shows the flow graph for computation of an eight-point discrete Fourier transform.

Computation of orthogonal transforms generally involves a large number of operations. Hence, single processor computers are not very suitable for this application. Special purpose parallel processors must be designed which reduce the computation time by performing the required operations in parallel instead of in sequence [10-14].

As discussed in previous chapters, quadtrees are attractive data base structures for the storage of imaging information. In this chapter a VLSI structure for parallel computation of orthogonal transforms based on quadtrees, is presented.

REGION SEARCH MEMORY (RSM)

As was explained in previous chapters, the quadtree is an image representation technique. All the points in a two dimensional image are included in the nodes of the quadtree. In order to compute any kind of transform there is a need for extraction of the property of the points from their nodes. In order to do this a special type of CAM called Region Search Memory (RSM) is designed. In RSM, given the coordinates of the point, the point will be compared to end

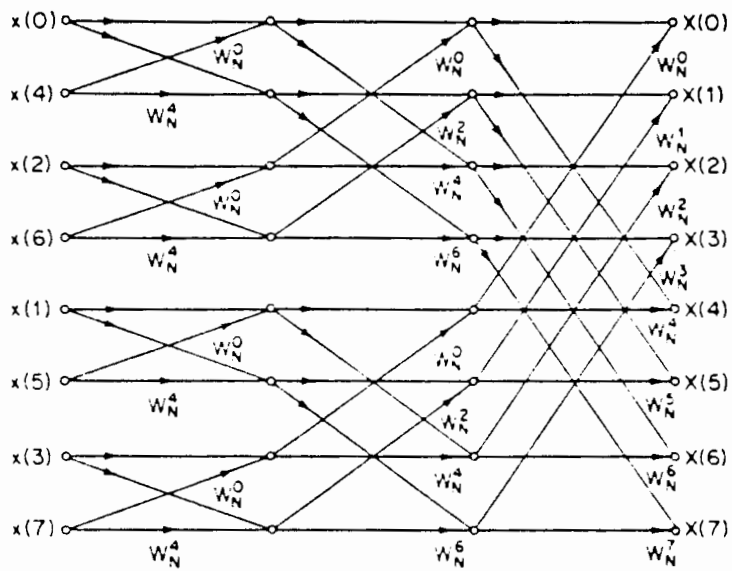


Figure 10. A flow graph for computation of an 8-point DFT.

points of all the quadtree nodes simultaneously. Upon finding the node that the point belongs to, its property is extracted. Then the property of the point is used for computation of the transform. RSM is shown in Figure 14.

HARDWARE CONFIGURATION

This project utilizes a two-dimensional array which represents the image. The assumption is that the image is available in quadtree form. The properties associated with each node of the quadtree are indicated in Figure 12. The control unit in this system coordinates the communication between the processors via an interconnection network, controls the operations performed by the processors, and controls the local memories. Local memories (PEM's) are region search memories (RSM). Processing elements (PE's) are used to perform the actual computation of orthogonal transforms. An interconnection network is used for routing the data between the processors. There is a requirement of N processors and N local memories for an N -point transform. The connection of all the units explained above is shown in Figure 11.

PROCESSING ELEMENTS

In the proposed architecture, N PE's are needed to perform an N -point transform [12-14]. The internal

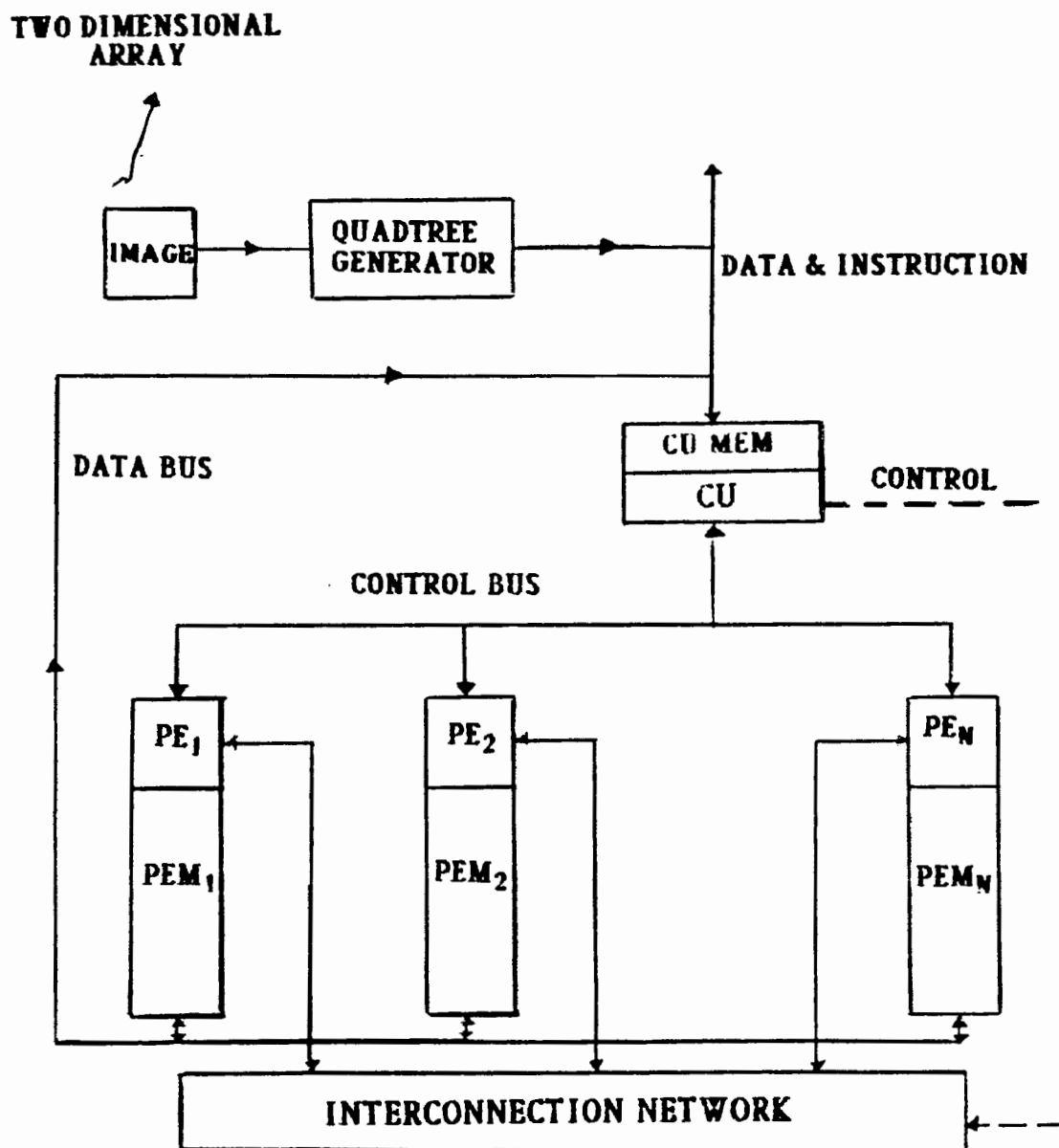


Figure 11. A hardware system for computation of orthogonal transforms.

configuration of each processor is shown in Figure 13a. This CPU consists of an ALU unit, a register called R1, and two multiplexers. The ALU is capable of performing subtraction, addition, multiplication, or of letting the data go through unchanged. The operation of ALU is selected by S0 and S1, as indicated in the table contained in Figure 13b.

Inputs to the ALU can come from four different sources. They can come from a different processor, from register R1, from the memory associated with the particular processor, or from the control unit. Selection of the multiplexers' inputs is made by control bits shown in the table contained in Figures 13c and 13d. The result of the ALU operation is stored in R1.

MEMORY

There are N PEM's associated with N PE's in the system. The configuration of each PEM is shown in Figure 14.

In order to compute the transforms we need to find the points on the butterfly arrangement. There is a need to search for the points on the quadtree. The properties of the points must be extracted from regions of the quadtree. For purposes of this project, a Region Search Memory (RSM) is designed which will find the end points of the node in which the needed point is located. With X,Y as the

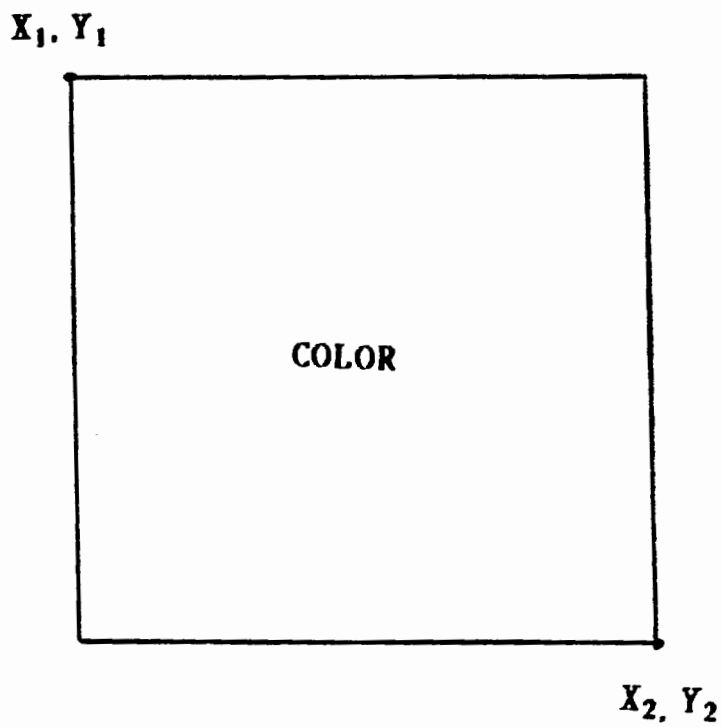
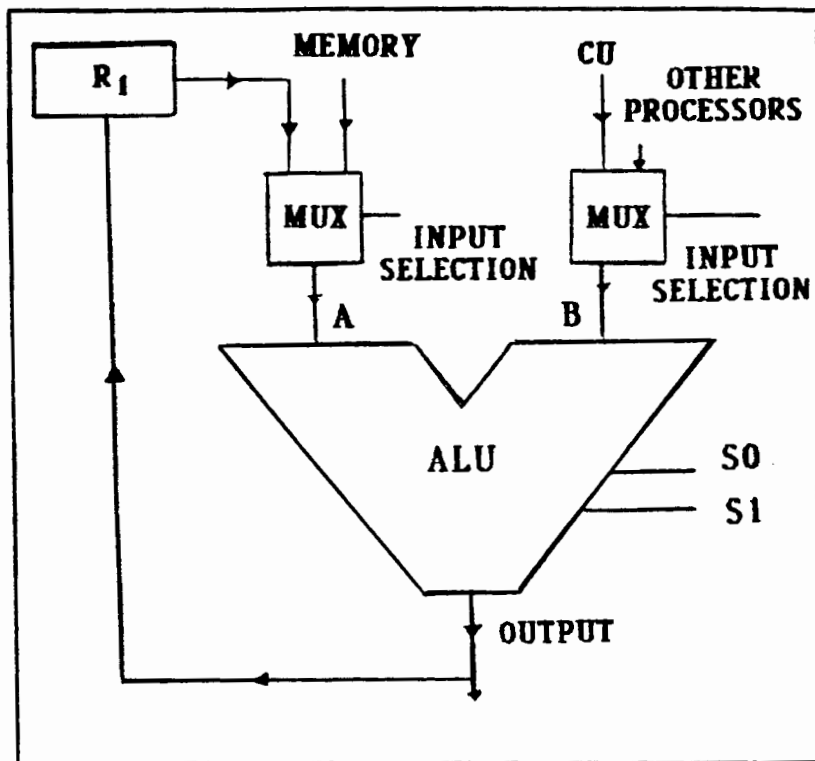


Figure 12. A node on a quadtree and its associated properties.



a)

INPUT SELECTION A	
0	MEMORY
1	R ₁

c)

INPUT SELECTION B	
0	CU
1	PROCESSORS

d)

FUNCT SELECTION		
S0	S1	OUTPUT
0	0	A+B
0	1	A-B
1	0	A
1	1	M*A

b)

Figure 13. A PE configuration.

coordinates of the point in question, the quadtree nodes are checked by RSM to find a node for which X, Y is greater than X_1, Y_1 respectively and X, Y is smaller than X_2, Y_2 respectively. (X_1, Y_1 are the left upper and X_2, Y_2 are the right lower end points of the nodes on the quadtree.) This is accomplished by use of an array of comparators, shown in Figure 14. Each element in this array is shown in Figure 15. Upon the satisfaction of all the conditions shown in Figure 15, the tag for that particular node is turned on. The greater block in Figure 15 is shown in detail in Figure 16. The less-than blocks are exactly like the greater-thans, except that the X and Y locations are reversed. So when the tag for a particular node is on, it is possible to readily identify which node the point belongs to. All the points in that node should have the same color property. The color property is then extracted for further processing. (Note: EQUAL in Figure 15 is equivalent to a circuit in which the output is high when X and Y bits are equal and the output is low otherwise.)

INTERCONNECTION NETWORK

This block takes care of the routing of data between different processors. It is controlled by CU, but is not reviewed in detail in this thesis. Its task is to have the necessary routes available for the transfer of data from the

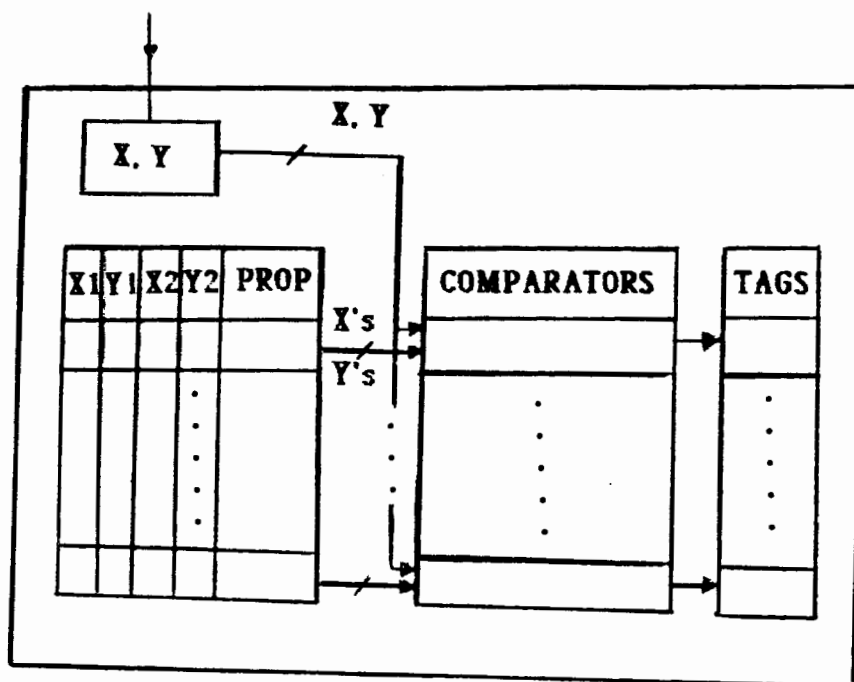


Figure 14. A PEM configuration.

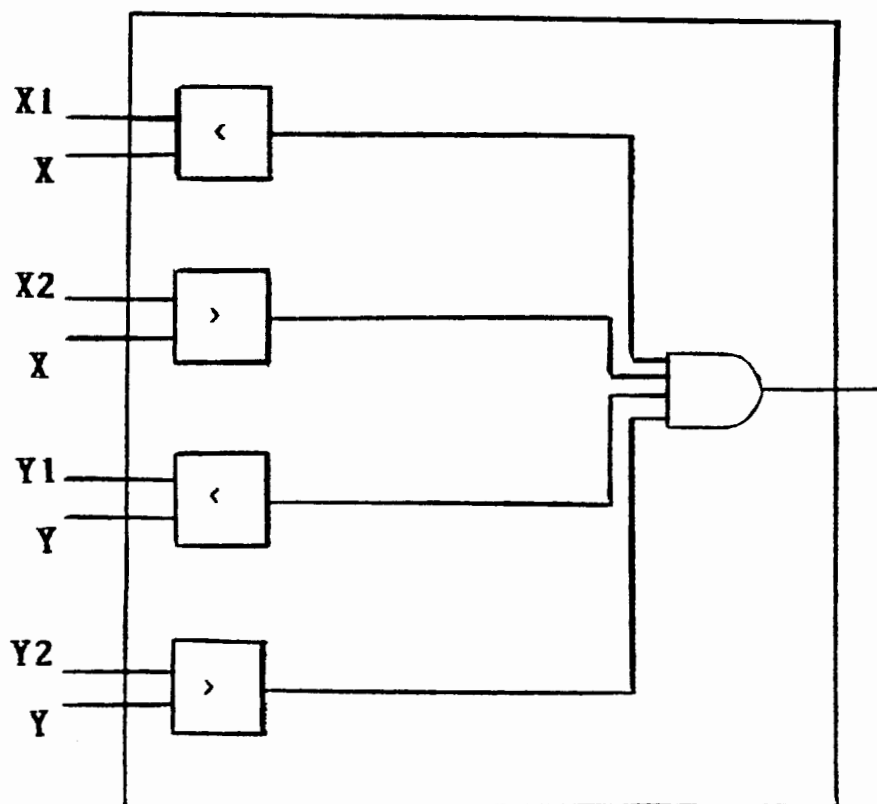


Figure 15. A comparator element.

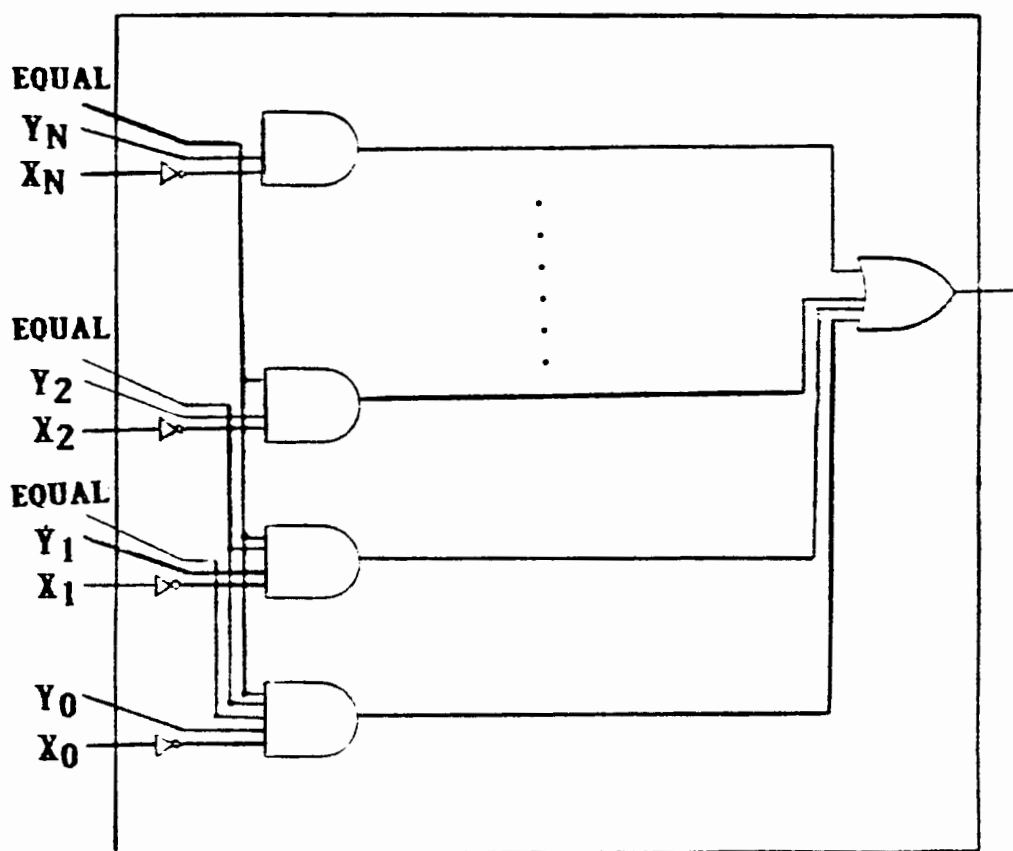


Figure 16. A greater-than block.

source processors to the destination processors. To accomplish the above task, the CU unit sends appropriate signals to the interconnection network to facilitate the routing needs.

SYSTEM OPERATION

The quadtree generated consists of an array of nodes which are identified by their left upper and right lower corners. Each node also has a property (for example, color) associated with it. These characteristics, produced by the quadtree generator, are distributed to all the PEM's via a data bus. So, whenever the property of i th point is needed, the i th processor is addressed. Then using RSM memory, the appropriate node is found and the property of the point is extracted. To compute the desired orthogonal transform, the program to accomplish the task is stored in CU memory. Then each instruction is passed to all the processors and processed in parallel. The number of operations performed in each processor is equal to $\log_2 N$, for an N -point image.

The first operation is to extract the properties of the points from the appropriate nodes, by use of RSM's, and to store them in Rls via ALU's in the appropriate processors.

From the second operation to the $(\log_2 N)$ th operation, the data necessary for operations come from the registers of different processors or from the CU. In Figure 13a, in the

CPU configuration, there are two inputs for the ALU, termed A and B. Input B, when needed, comes from a different processor or CU. Input A can come from two different places: register R1 or RSM memory. RSM memory is selected only once, when properties of the points are needed. CU is selected whenever a multiplier is needed for R1. The multiplication activity is then performed, and the result is stored back in R1. R1 is used throughout the computation. Whenever data are needed from a different processor for an operation, they are routed from one processor to another, through the interconnection network, by CU, sending the appropriate control signals to the interconnection network and the appropriate processors.

As each operation is performed in the processors, the result is put back in the R1 of the appropriate processors. After the $(\log_2 N)$ th operation, processor 1 will have the result of the transformation of the first point, processor 2 the result of the second point, and so on.

CONCLUSION

A hardware system for computation of orthogonal transforms is developed. The computation is based on the quadtree representation of the image. The hardware performs independently from the original image. Steps needed for computation of the orthogonal transform are $\log_2 N$. Since

parallel processing is incorporated into this system, the computation is performed extremely fast. This system is best suited for applications where a high degree of speed is sought in the computation of orthogonal transforms.

REFERENCES

1. H. Freeman, "On the encoding of arbitrary geometric configurations," IRE Trans. Electron. Comput., vol. EC-10, pp. 260-268.
2. S. Murakami, M. Okada, and T. Tsuchiya, "A study on a method of line drawings," Trans. IECE Japan, vol. J59-D, Feb 1976, pp. 117-124.
3. A. Kilinger and C. R. Dyer, "Experiments in picture representation using regular decomposition," Computer Graphics Image Processing, vol. 5, 1986, pp. 68-105.
4. G. M. Hunter and K. Steiglitz, "Operations on images using quadrees," IEEE Trans. Pattern Analysis Machine Intelligence, vol. PAMI-1, 1979, pp. 145-153.
5. W. E. Snyder and C. D. Savage, "control addressable read/write memories for image analysis," IEEE Trans. Computers, no. 10, Oct 1982, pp. 963-968.
6. W. E. Snyder and A. Cowart, "An iterative approach to region growing using associative memories," IEEE Trans. Pattern Analysis and Machine Intelligence, vol. PAMI-5, no. 3, May 1983, pp. 349-352.
7. L. R. and C. M. Rabiner, Digital Signal Processing, New York: IEEE press, 1972.
8. Special issue on digital picture processing, IEEE 60, July 1972.
9. Special issue on two dimensional digital signal processing. IEEE Trans. Computers, vol. C-21, July 1972.
10. H. D. Wisher, "Designing special purpose image processor," Computer Design, vol. 11, 1972, pp. 71-76.
11. J. A. Glassman, "A generalization of fast Fourier transform," IEEE Trans. Computer, vol. C-19, 1970, pp. 105-116.

12. L. S. Davis, "Computer architecture for image processing," Picture Data Descrip. Management Conf., Aug 27-28, 1980, pp. 249-254.
13. F. A. Briggs, K. S. Fu, K. Hwang, and J. Patel, "PM: A configurable multiprocessor system for pattern recognition and image processing," Proc. NCC, AFIPS, June 1979, pp. 255-256.
14. Kai Hwang and Faye A. Briggs, "Computer architecture and parallel processing," McGraw-Hill, Inc., 1984.
15. T. Pavlidis, "The use of algorithms of piecewise approximations for picture processing applications," Assoc. Comput. Mach. Trans. Math. Software, vol. 2, pp. 305-321, Dec. 1976.
16. S. L. Horowitz and T. Pavlidis, "Picture segmentation by a tree traversal algorithm," J. Assoc. Comput. Mach., vol. 23, pp. 368-388, April 1976.
17. S. L. Tanimoto, "Pictorial feature distortion in a pyramid," Comput. Graphics and Image Processing, vol. 5, pp. 333-352, 1976.
18. "A pyramid model for binary picture complexity," in Proc. IEEE Comput. Soc. Conf. Pattern Recognition and Image Processing, Rensselaer Polytechnic Institute, NY, June 6-8, 1977, pp. 25-28.
19. G. M. Hunter, "Computer animation survey," Comput. and Graphics, vol. 2, pp. 225-229, 1977.
20. N. Negroponte, "Raster scan approaches to computer graphics," Comput. and Graphics, vol. 2, pp. 174-193, 1977.

APPENDIX A

A COMPUTER PROGRAM FOR IMPLEMENTATION OF THE
ALGORITHM DEVELOPED FOR REGION GROWING

```

/*****
/*This is th IJ routine.Having the path of */
/*the particular place it will find the */
/*corresponding I and J. */
/*Inputs are pointers to I and J and a */
/*pointer to the particular path. */
*****/

#include "vars.h"

IJ(pI,pJ,ptr)
int *pI; /*pointer to I*/
int *pJ; /*pointer to J*/
PATH *ptr;
{
    int temp =0;
    int path_length=0;

    while(ptr != NULL)
    {
        *pI <= 1;

        temp=ptr->num & 2; /*to get the second bit*/
        temp >>= 1;
        *pI |= temp;

        *pJ <=1;

        temp=ptr->num & 1; /*to get first bit*/
        *pJ |= temp;

        path_length++;

        ptr=ptr->front;
    }/*while*/

    return(path_length);
}/*end*/

/*****
/*This is the construct routine. It will construct */
/*the quad tree using a recursive approach. */
*****/

#include "ext.h"
#include <math.h>

#define NW 0
#define NE 1
#define SW 2
#define SE 3

NODE *construct(level,x,y,color)
int level; /*2 to power level by 2 to power level*/
int x; /*horizontal size of the picture*/
int y; /*vertical size of the picture*/

```

```

int *color; /*pointer to type*/
{
NODE *construct();
NODE *Q,*R;
NODE *pointer[4]; /*keeps track of pointers*/
int type[4]; /*keeps track of types*/
int i,j; /*index*/
char *calloc();

if(level == 0) /*process the pixel*/
{
*color = A[x-1][y-1]; /*color of the picture*/
return(NULL); /*return pointer NULL*/
} /*if*/

else
{
level --;

pointer[NW]=construct(level,x- ((int) pow(2.0,(double) level)))
, (y- ((int) pow(2.0,(double) level))), &type[NW]);

pointer[NE]=construct(level,x- ((int) pow(2.0,(double) level)))
,y, &type[NE]);

pointer[SW]=construct(level,x,(y- ((int) pow(2.0,(double) level)))
,&type[SW]);

pointer[SE]=construct(level,x,y,&type[SE]);

if((type[NW] != 2) &&
type[NW] == type[NE] &&
type[NW] == type[SW] &&
type[NW] == type[SE]) /*not gray*/
{
*color = type[NW];
return(pointer[NW]);
} /*if*/

else /*begin a non terminal GRAY node*/
{
Q= (NODE *) calloc(1,sizeof(NODE));
gray_nodes++;
for(i=0;i<4;i++)
{
if(type[i] == 2) /*if gray*/
/*link it to father node*/
{
Q->son[i] = pointer[i];
pointer[i]->father = Q;
} /*if*/

else
{
R= (NODE *) calloc(1,sizeof(NODE));
if(type[i] == 0) /*it is a black node*/
black_nodes++;
else /*it is a white node*/
white_nodes++;
}
}
}
}

```

```

        R->node_type = type[i];
        R->region=0;      /*region set to zero*/

        for(j=0; j<4;j++)
            R->son[j]=NULL;

        Q->son[i] =R;
        R->father =Q;

    }/*else*/

}/*for*/

Q->node_type =2;      /*set it to gray*/
*color =2;          /*set it to gray*/
return(Q);

}/*else*/

}/*else*/

}/*end*/

/*****
/*This is the copy_insert routine.It will copy */
/*the path two to the power of NOX times.Then it */
/*will insert all the combinations of number1 and*/
/*number2 to the end of the paths.          */
*****/

#include "vars.h"
#include <math.h>
#define Ntype calloc(1,sizeof(NEY))
#define Ptype calloc(1,sizeof(PATH))

copy_insert(number1,number2,NOX,ptr_neighbor)
int number1;
int number2;
int NOX;    /*number of x*/
NEY *ptr_neighbor;
{
    int i,j,k;
    int repeat;
    int count,til;
    char *calloc();

    NEY *temp_neighbor,*middle_neighbor,*last_neighbor;

    repeat = (int) pow(2.0,(double) NOX);
    temp_neighbor=ptr_neighbor;
    while(temp_neighbor->next != NULL)
        temp_neighbor=temp_neighbor->next;
    middle_neighbor=temp_neighbor;

    for(i=0;i<repeat;i++)
    {
        last_neighbor= (NEY *) Ntype;
        middle_neighbor->next=last_neighbor;
        middle_neighbor=last_neighbor;
    }
}

```

```

    last_neighbor->next = NULL;
    Ncopy_path(middle_neighbor,ptr_neighbor->pointer);
}/*for*/

count=NOX-1;

for(i=1;i<=NOX;i++)
{
    middle_neighbor=temp_neighbor->next;

    repeat= ((int) pow(2.0,(double) i))/2;

    if(count == 0)
        til=1;
    else
        til= (int) pow(2.0,(double) count);

    count--;

    for(k=til;k>0;k--)
    {
        for(j=1;j<=repeat;j++)
        {
            insert(number1,1,middle_neighbor->pointer);
            middle_neighbor=middle_neighbor->next;
        }/*for*/

        for(j=1;j<=repeat;j++)
        {
            insert(number2,1,middle_neighbor->pointer);
            middle_neighbor=middle_neighbor->next;
        }/*for*/

    }/*for*/

}/*for*/

}/*end*/

/*****
/*This is the copy_path routine.It will copy the path */
/*of the specified source to the path of the specified*/
/*destination.                                         */
*****/

#include "vars.h"
#define Ptype calloc(1,sizeof(PATH))

Ccopy_path(to,from) /*copy source path to destination path*/
Clist *to;
PATH *from;
{
    PATH *dest;
    char *calloc();

    if(from != NULL)
    {
        to->pointer=(PATH *) Ptype;
        dest=to->pointer;
    }

```

```

dest->front=NULL;

dest->num = from->num; /*copy one number*/

from=from->front;

}/*if*/

while(from != NULL) /*if still item in source path*/
{
    dest->front=(PATH *) Ptype;
    dest=dest->front;
    dest->front=NULL;

    dest->num=from->num; /*copy one number*/

    from=from->front;
}/*while*/

}/*end*/

Ocopy_path(to,from) /*copy source path to destination path*/
Olist *to;
PATH *from;
{
    PATH *dest;

    if(from != NULL)
    {
        to->pointer=(PATH *) Ptype;
        dest=to->pointer;
        dest->front=NULL;

        dest->num = from->num; /*copy one number*/

        from=from->front;

    }/*if*/

    while(from != NULL) /*if still item in source path*/
    {
        dest->front=(PATH *) Ptype;
        dest=dest->front;
        dest->front=NULL;

        dest->num=from->num; /*copy one number*/

        from=from->front;
    }/*while*/

}/*end*/

Ncopy_path(to,from) /*copy source path to destination path*/

```



```

NEY *to;
PATH *from;
{
    PATH *dest;

    if(from != NULL)
    {
        to->pointer=(PATH *) Ptype;
        dest=to->pointer;
        dest->front=NULL;

        dest->num = from->num; /*copy one number*/

        from=from->front;

    }/*if*/

    while(from != NULL) /*if still item in source path*/
    {
        dest->front=(PATH *) Ptype;
        dest=dest->front;
        dest->front=NULL;

        dest->num=from->num; /*copy one number*/

        from=from->front;
    }/*while*/

}/*end*/

/*****/
/*This is the find_depth routine. It will find*/
/*the depth of the quad tree and returns it. */
/*****/

#include "vars.h"
#define NW 0
#define NE 1
#define SW 2
#define SE 3

find_depth(head)
NODE *head;
{
    int length[4];
    int temp=0;

    if(head == NULL) /*no more sons*/
        return(0);

    else
    {
        length[NW]= find_depth(head->son[NW]);
        length[NE]= find_depth(head->son[NE]);
        length[SW]= find_depth(head->son[SW]);
        length[SE]= find_depth(head->son[SE]);
    }/*else*/

    /*find maximum between four paths*/
    temp=length[NW];

```

```

if(length[NE] > temp)
    temp=length[NE];

if(length[SW] > temp)
    temp=length[SW];

if(length[SE] > temp)
    temp=length[SE];

return(temp+1);

}/*end*/

/*****
/*This is the find_node routine. It will find the */
/*the node in the quad tree, specified by the path */
/*passed to the this routine. */
*****/

#include "vars.h"

NODE *find_node(path_ptr,head)
PATH *path_ptr; /*path to the particular node*/
NODE *head; /*root of the tree*/
{
    NODE *node_in_process;
    PATH *last_ptr;

    node_in_process=head;

    while(path_ptr != NULL)
    {
        if(node_in_process->son[path_ptr->num] == NULL)
        {
            last_ptr->front=NULL;
            break;
        }

        node_in_process=node_in_process->son[path_ptr->num];

        last_ptr=path_ptr;
        path_ptr=path_ptr->front;
    }/*while*/

    return(node_in_process); /*return the node found*/
}/*end*/

/*****
/*This is the get_neighbor routine. It will find the */
/*neighbors of the particular node specified by the ptr */
/*passed to this routine. It will return a pointer which */
/*is the head of a list which are the neighbors of the */
/*particular node. */
/*All the neighbors are found by calling the partialpath */
/*routine and then if the node is bigger than smallest */

```

```

/*existing node on the tree it will make the necessary */
/*adjustment,by calling the insert or copy_insert */
/*****
#include "vars.h"
#define Ntype calloc(1,sizeof(NEY))
#define Ptype calloc(1,sizeof(PATH))

NEY *get_neighbor(ptr)
PATH *ptr;
{
int indicator[9]; /*determines the status of the paths*/
int I,J; /*will have location of path*/
int NOX;
int path_length,IJ();
int path_num;
NEY *top_neighbor,*partial_path();
NEY *ptr_neighbor;
NEY *temp_neighbor;
char *calloc();
I=J=0;

path_length=IJ(&I,&J,ptr); /*find I and J length of path*/

/*find partial paths*/
top_neighbor=partial_path(I,J,path_length,indicator);

NOX = MAX_DEPTH - path_length;

if(NOX == 0)
return(top_neighbor); /*no adjustment needed*/

else
{
path_num=1;
for(ptr_neighbor=top_neighbor;path_num<=8;path_num++)
{
switch(path_num)
{
case 1 :
if(indicator[path_num])
{
insert(3,NOX,ptr_neighbor->pointer);
ptr_neighbor=ptr_neighbor->next;
}

break;

case 2 :
if(indicator[path_num])
{
copy_insert(2,3,NOX,ptr_neighbor);

if(ptr_neighbor == top_neighbor)
top_neighbor=top_neighbor->next;

else
{

```

```

        temp_neighbor=top_neighbor;
        while(temp_neighbor->next != ptr_neighbor)
            temp_neighbor=temp_neighbor->next;
        temp_neighbor->next=ptr_neighbor->next;
    }

    ptr_neighbor=ptr_neighbor->next;
}

break;

case 3 :
    if(indicator[path_num])
    {
        insert(2,NOX,ptr_neighbor->pointer);
        ptr_neighbor=ptr_neighbor->next;
    }

    break;

case 4 :
    if(indicator[path_num])
    {
        copy_insert(1,3,NOX,ptr_neighbor);

        if(ptr_neighbor == top_neighbor)
            top_neighbor=top_neighbor->next;

        else
        {
            temp_neighbor=top_neighbor;
            while(temp_neighbor->next != ptr_neighbor)
                temp_neighbor=temp_neighbor->next;
            temp_neighbor->next=ptr_neighbor->next;
        }

        ptr_neighbor=ptr_neighbor->next;
    }

    break;

case 5 :
    if(indicator[path_num])
    {
        copy_insert(0,2,NOX,ptr_neighbor);

        if(ptr_neighbor == top_neighbor)
            top_neighbor=top_neighbor->next;

        else
        {
            temp_neighbor=top_neighbor;
            while(temp_neighbor->next != ptr_neighbor)
                temp_neighbor=temp_neighbor->next;
            temp_neighbor->next=ptr_neighbor->next;
        }

        ptr_neighbor=ptr_neighbor->next;
    }

```

```

        break;

case 6 :
    if(indicator[path_num])
    {
        insert(1,NOX,ptr_neighbor->pointer);
        ptr_neighbor=ptr_neighbor->next;
    }

    break;

case 7 :
    if(indicator[path_num])
    {
        copy_insert(0,1,NOX,ptr_neighbor);

        if(ptr_neighbor == top_neighbor)
            top_neighbor=top_neighbor->next;

        else
        {
            temp_neighbor=top_neighbor;
            while(temp_neighbor->next != ptr_neighbor)
                temp_neighbor=temp_neighbor->next;
            temp_neighbor->next=ptr_neighbor->next;
        }

        ptr_neighbor=ptr_neighbor->next;
    }

    break;

case 8 :
    if(indicator[path_num])
    {
        insert(0,NOX,ptr_neighbor->pointer);
        ptr_neighbor=ptr_neighbor->next;
    }

    break;

    }/*switch*/

}/*for*/

}/*else*/

return(top_neighbor);    /*return list of neighbors*/

}/*end*/

/*****
/*This is the insert routine. It will insert the
/*number which is passed to this routine to the end*/
/*of the path repeat number of times which is also */
/*passed to this routine.                               */

```

```

/*****/

#include "vars.h"
#define Ptype calloc(1,sizeof(PATH))

insert(number,repeat,ptr)
int number; /*number to insert*/
int repeat;
PATH *ptr;
{
int i; /*index*/
char *calloc();

while(ptr->front != NULL)
    ptr=ptr->front;

for(i=0;i<repeat;i++)
{
    ptr->front= (PATH *) Ptype;
    ptr=ptr->front;
    ptr->num=number;
    ptr->front=NULL;
}/*for*/

}/*end*/

/*****/
/*This is the load_picture routine.It will load */
/*the picture pixel by pixel into a two dimensional*/
/*array of 256 by 256. */
/*****/

#include "ext.h"

/*load the picture into an array*/

load_picture()
{
char *ptr,*terminal_check();
int i,j,k; /*index*/
int color;

ptr=terminal_check(); /*which display in use*/

SETUP(ptr,0); /*set up the display*/

CHSIZ(8,8);

LCMAP(0,255,255,255); /*white*/
LCMAP(1,40,40,255); /*blue*/
LCMAP(2,255,0,132); /*pink*/
LCMAP(3,255,206,0); /*brown*/
LCMAP(4,255,255,255); /*white*/
LCMAP(5,0,0,0); /*black*/

STCOL(5); /*color is black*/
CLEAR(); /*set all pixels to black*/

STCOL(0);

```

```

POLYS();
POLYV(0,300);
POLYV(255,300);
POLYV(255,555);
POLYV(0,555);
POLYC();
POLYF();

STCOL(1);
MOVPL(30,335);
TEXT("PSU");
/*load up the array*/
printf("loading the picture into array\n");

for(j=555,k=0;j>=300;j--,k++)
    for(i=0;i<256;i++)
    {
        MOVPL(i,j);
        A[k][i] = RPIXEL();
    }

}/*end*/

/*****
/*This is the label_region routine.Givining it the */
/*root of the quad tree it will label all regions */
/*calling appropriate routines. */
*****/

#include "vars.h"

#define Otype calloc(1,sizeof(Olist))
#define Ctype calloc(1,sizeof(Clist))
#define Ptype calloc(1,sizeof(PATH))
#define Ntype calloc(1,sizeof(NEY))

label_region(head) /*root*/
NODE *head;
{
    NODE *node_in_process,*find_node();
    PATH *path_ptr;
    Olist *temp_open;
    Clist *temp_close;
    NEY *top_neighbor,*temp_neighbor,*get_neighbor();
    char *calloc();
    printf("\nlabeling the regions\n");

    begin=tail=open_ptr=NULL; /*open list pointers*/
    first=last=close_ptr=NULL; /*close list pointers*/

    /*finds the depth of the tree,root is not included*/
    MAX_DEPTH = find_depth(head)-1;
    printf("Maximum depth is %d\n",MAX_DEPTH);

    region_number=1;

    /*put first item in the open list*/
    begin=(Olist *) Otype;
    tail=begin;

```

```

open_ptr=begin;

begin->pointer=(PATH *) Ptype;
path_ptr=begin->pointer;

node_in_process=head; /*root*/

if(node_in_process->son[0] != NULL)
{
    path_ptr->num=0;
    path_ptr->front=NULL;

    node_in_process=node_in_process->son[0];
}/*if*/

while(node_in_process->son[0] != NULL)
{
    path_ptr->front=(PATH *) Ptype;
    path_ptr=path_ptr->front;
    path_ptr->num=0;
    path_ptr->front=NULL;

    node_in_process=node_in_process->son[0];
}/*while*/

while(open_ptr != NULL)
{
    /*finds the node on the quad tree*/
    node_in_process=find_node(open_ptr->pointer,head);
    if(node_in_process->region != 0)
    {
        /*get rid of processed node in open list*/
        temp_open=open_ptr;
        open_ptr=open_ptr->next; /*next element in open list*/
        cfree((char *) temp_open); /*free the node*/

        continue;
    }/*if*/

    /*put item in close list*/
    first=(Clist *) Ctype;
    last=first;
    close_ptr=first;

    first->next=NULL;
    /*copy path in open list to path in close list*/
    Ccopy_path(close_ptr,open_ptr->pointer);

    /*get rid of processed node in open list*/
    temp_open=open_ptr;
    open_ptr=open_ptr->next;
    cfree((char *) temp_open);

    /*give node a region number*/
    node_in_process->region=region_number;

    current_type=node_in_process->node_type;

    while(close_ptr != NULL)
    {

```



```

/*return neighbors*/
top_neighbor=get_neighbor(close_ptr->pointer);
while(top_neighbor != NULL)
{
    node_in_process=find_node(top_neighbor->pointer,head);
    if(node_in_process -> region != 0)
    {
        /*get rid of processed node in neighbor list*/
        temp_neighbor=top_neighbor;
        top_neighbor=top_neighbor->next;
        cfree((char *) temp_neighbor);

        continue;
    }/*if*/

    if(node_in_process->node_type == current_type)
    {
        node_in_process->region=region_number;
        put_in_close_list(top_neighbor->pointer);
    }/*if*/

    else
        put_in_open_list(top_neighbor->pointer);

    /*get rid of processed node in neighbor list*/
    temp_neighbor=top_neighbor;
    top_neighbor=top_neighbor->next;
    cfree((char *) temp_neighbor);

}/*while*/

/*get rid of processed node in close list*/
temp_close=close_ptr;
close_ptr=close_ptr->next;
cfree((char *) temp_close);

}/*while*/

region_number++;
first=last=close_ptr=NULL; /*initilize the close list*/

}/*while*/

}/*end*/

/*****
/*This is the main routine.It will call load_picture*/
/*to load the picture into an array. Then it will */
/*call the quad tree routine to make a quad tree out*/
/*of the array.Next it will call the label_region */
/*routine to label the regions.And finally it will */
/*call the quad to picture routine to draw the */
/*the picture which is labeled. */
*****/

#include "ext.h"
#define level 8

/*      assumption      */
/*black is represented by 0*/

```

```

/*white is represented by 1*/
/*gray is represented by 2*/

main()
{
    NODE *head,*quad_tree();

    white_nodes=0; /*number of white nodes created*/
    black_nodes=0; /*number of black nodes created*/
    gray_nodes=0; /*numbewr of gray nodes created*/

    load_picture(); /*load picture into an array*/

    /*call routine quad tree, to built the tree*/
    head=quad_tree(level);

    printf("\nnodes created\n");
    printf("-----\n");
    printf("gray_nodes %d\n",gray_nodes);
    printf("black_nodes %d\n",black_nodes);
    printf("white_nodes %d\n",white_nodes);

    label_region(head); /*region labeling*/

    quad_to_picture(head); /*draw picture using quad tree*/

    FINIS();

}/*end*/

/*****
/*This routine will actually draws the picture at the */
/*end when the picture is labeled. */
*****/
#include "ext.h"

#define NW 0
#define NE 1
#define SW 2
#define SE 3

make_picture(head,x,y,size)
NODE *head; /*head of tree or subtree*/
int x; /*x coordinate */
int y; /*y coordinate*/
int size; /*size of picture*/
{
    int j; /*index*/

    if(head->node_type != 2) /*if white or black draw it,not gray*/
    {
        STCOL(head->region);

        for(j=y;j<y+size;j++)
        { MOVPl(x,j);
          RLFILL(size);
        }/*for*/
    }
}

```

```

        return;          /* return done drawing*/
    }/*if*/
else
    {
        make_picture(head->son[NW],x,y+size/2,size/2);
        make_picture(head->son[NE],x+size/2,y+size/2,size/2);
        make_picture(head->son[SW],x,y,size/2);
        make_picture(head->son[SE],x+size/2,y,size/2);
    }/*else*/
}/*end*/

/*****
/*This is the partial_path routine.It will find the */
/*neighbors of the paarticular node and will return */
/*the a list of all the neighbors.It will find the */
/*neighbors depending on the size of the node in que-*/

/*stion,that is ,size of the neighbors are the same */
/*size as the node which we seek neighbors for. */
*****/

#include "vars.h"
#include <math.h>
#define Ntype calloc(1,sizeof(NEY))
#define Ptype calloc(1,sizeof(PATH))
#define TRUE 1
#define FALSE 0

NEY *partial_path(I,J,path_length,indicator)
int I;
int J;
int path_length;
int indicator[];
{
    int Nflag=0; /*tells you if it is first time in*/
    int newI,newJ;
    int temp=0;
    int index=1;
    int nol;
    PATH *pptr;
    NEY *first_neighbor,*middle_neighbor,*last_neighbor;
    char *calloc();

    first_neighbor= (NEY *) Ntype;
    last_neighbor=first_neighbor;
    middle_neighbor=first_neighbor;
    middle_neighbor->next=NULL;

    for(newI=I-1;newI <= I+1;newI++)
    {
        if(newI < 0 || newI >= ((int) pow(2.0,(double) path_length)))
        {
            indicator[index++]=FALSE;
            indicator[index++]=FALSE;
            indicator[index++]=FALSE;

```

```

    continue;
}/*if*/

for(newJ=J-1;newJ <= J+1;newJ++)
{
    if(newJ < 0 || newJ >= ((int) pow(2.0,(double) path_length))
    {
        indicator[index++]=FALSE;

        continue;
    }/*if*/

    if(newI == I && newJ == J)
        continue;

    if(Nflag)    /*if not first time here*/
    {
        last_neighbor= (NEY *) Ntype;
        middle_neighbor->next=last_neighbor;
        middle_neighbor=last_neighbor;
        middle_neighbor->next=NULL;
    }/*if*/
    Nflag=1;

    middle_neighbor->pointer= (PATH *) Ptype;
    pptr=middle_neighbor->pointer;
    pptr->num=0;
    pptr->front=NULL;

    temp= newI >> path_length-1;
    temp &= 1;
    pptr->num |= temp;
    pptr->num <= 1;

    temp=newJ >> path_length-1;
    temp &= 1;
    pptr->num |= temp;

    for(nol=path_length-2;nol >= 0;nol--)
    {
        pptr->front=(PATH *) Ptype;
        pptr=pptr->front;
        pptr->num=0;
        pptr->front=NULL;

        temp= newI >> nol;
        temp &= 1;    /*only first bit*/
        pptr->num |= temp;
        pptr->num <= 1;

        temp=newJ >> nol;
        temp &= 1;    /*only first bit*/
        pptr->num |= temp;

    }/*for*/    /*end of first path*/

```

```

        indicator[index++]=TRUE;

    }/*for*/

    }/*for*/

    return(first_neighbor);

}/*end*/

/*****
/*This is the routine put_in_close_list.It will put */
/*the specified item in the close list.          */
*****/

#include "vars.h"
#define Ctype calloc(1,sizeof(Clist))

put_in_close_list(ptr)
PATH *ptr;
{
    char *calloc();
    if(close_ptr == NULL)    /*nothing in list*/
    {
        first=(Clist *) Ctype;
        last=first;
        close_ptr=first;
        first->next=NULL;

        Ccopy_path(first,ptr); /*put path in close list*/

    }/*if*/

    else    /*list not empty*/
    {
        last=(Clist *) Ctype;
        first->next=last;
        first=last;
        first->next=NULL;

        Ccopy_path(first,ptr); /*put path in close list*/

    }/*else*/

}/*end*/

/*****
/*This is put_in_open_list routine.It will put the */
/*the specified item in the open list.          */
*****/

#include "vars.h"
#define Otype calloc(1,sizeof(Olist))

put_in_open_list(ptr)
PATH *ptr;

```

```

(
char *calloc();
if(open_ptr == NULL) /*nothing in list*/
{
begin=(Olist *) Otype;
tail=begin;
open_ptr=begin;
begin->next=NULL;

Ocopy_path(begin,ptr); /*put path in open list*/

}/*if*/

else /*list not empty*/
{
tail=(Olist *) Otype;
begin->next=tail;
begin=tail;
begin->next=NULL;

Ocopy_path(begin,ptr); /*put path in open list*/

}/*else*/

}/*end*/

/*****
/*This is the quad_to_picture routine.It will make */
/*the picture by calling the make_picture routin. */
*****/

#include "ext.h"

quad_to_picture(head)
NODE *head;
{
int x=0; /*left corner of screen*/
int y=0; /*left corner of screen*/
int size=256; /*size of picture*/
printf("making the picture from quad tree\n");

make_picture(head,x,y,size);

}/*end*/

/*****
/*This is the quad_tree routine.It will make the quad */
/*tree out of the two dimensional array.It will call */
/*the routine construct in order to do that. */
*****/

#include "ext.h"
#include <math.h>

NODE *quad_tree(level)
int level;
{
int type; /*returns the node type*/
NODE *Q,*pointer,*construct();

```

```

int i;    /*index*/
char *calloc();
printf("making the quad tree from the array\n");

pointer= construct(level,(int) (pow(2.0,(double) level))
                  ,(int) (pow(2.0,(double) level)),&type);

if(type == 2) /*node is gray*/
{
    pointer->father = NULL;
    return(pointer); /*tree built,pointer points to root*/
}/*if*/

else /*entire image is black or white*/
{
    if(type == 0) /*entire picture is black*/
        black_nodes++;

    else /*entire picture is white*/
        white_nodes++;

    Q= (NODE *) calloc(1,sizeof(NODE));

    Q->node_type=type;
    Q->region=0; /*region set to zero*/

    Q->father=NULL;

    for(i=0;i<4;i++)
        Q->son[i]=NULL;

    return(Q);
}/*else*/

}/*end*/

/*****
/*This is the terminal_check routine. It will find*
/*which terminal you are sitting by.          */
*****/

#include <stdio.h>

char *terminal_check( )

{
    char *ttyname( );
    char *pointer;

    pointer=ttyname(1); /*name of the terminal*/

    /*sitting by /dev/ttyh8/ */
    if(!(strcmp(pointer,"/dev/ttyh8")))
        return("/dev/ttyh9"); /*path of display is /dev/ttyh9*/

    /*sitting by /dev/ttyha*/
    else if(!(strcmp(pointer,"/dev/ttyha")))

```

```

    return("/dev/ttyhb"); /*path of display is /dev/ttyhb*/

/*sitting by /dev/ttyhc*/
else if(!(strcmp(pointer,"/dev/ttyhc")))
    return("/dev/ttyhd"); /*path of display is /dev/ttyhd*/

else /*sitting by a terminal not connected to a display*/
    return(NULL);
}/*end*/

/*****
/*This is the routine vars.c .It contains the */
/*the variables globally used by the program. */
*****/

#include <stdio.h>
#define hsize 256      /*horizontal size*/
#define vsize 256      /*vertical size*/

struct node
{
    struct node *son[4];
    struct node *father;
    int region; /*type of region*/
    int node_type; /*black,white,gray*/
};

typedef struct node NODE;

/*keeps track of the number of white nodes built*/
int white_nodes;

/*keeps track of the number of black nodes built*/
int black_nodes;

/*keeps track of the number of gray nodes created*/
int gray_nodes;

int A[vsize][hsize];

struct path
{
    int num; /*path*/
    struct path *front;
};

typedef struct path PATH;

struct open
{
    PATH *pointer; /*pointer to path*/
    struct open *next; /*next element*/
};

typedef struct open Olist;
Olist *begin,*tail,*open_ptr;

struct close
{
    PATH *pointer; /*pointer to path*/
};

```



```

        struct close *next; /*next element*/
    };
typedef struct close Clist;
Clist *first,*last,*close_ptr;

int current_type; /*holds property of region*/

int region_number;

struct neighbors
{
    PATH *pointer; /*pointer to path*/
    struct neighbors *next;
};
typedef struct neighbors NEY;

int MAX_DEPTH;
/*****
/*This is the routine ext.h. It contains*/
/*the variables indicated. They are used*/
/*by the routines which include ext.h. */
*****/

#include <stdio.h>
#define hsize 256      /*horizontal size*/
#define vsize 256      /*vertical size*/

struct node
{
    struct node *son[4];
    struct node *father;
    int region; /*region property*/
    int node_type; /*black,white,gray*/
};
typedef struct node NODE;

/*keeps track of the number of white nodes built*/
extern int white_nodes;

/*keeps track of the number of black nodes built*/
extern int black_nodes;

/*keeps track of the number of gray nodes created*/
extern int gray_nodes;

extern int A[vsize][hsize];
/*****
/*This is the routine vars.h. It contains*/
/*the variables used by the routines that*/
/*include vars.h. */
*****/

#include <stdio.h>
#define hsize 256      /*horizontal size*/
#define vsize 256      /*vertical size*/

struct node
{
    struct node *son[4];
    struct node *father;

```

```

        int region; /*type of region*/
        int node_type; /*black,white,gray*/
    };
typedef struct node NODE;

/*keeps track of the number of white nodes built*/
extern int white_nodes;

/*keeps track of the number of black nodes built*/
extern int black_nodes;

/*keeps track of the number of gray nodes created*/
extern int gray_nodes;

extern int A[vsize][hsize];

struct path
{
    int num; /*path*/
    struct path *front;
};
typedef struct path PATH;

struct open
{
    PATH *pointer; /*pointer to path*/
    struct open *next; /*next element*/
};
typedef struct open Olist;
extern Olist *begin,*tail,*open_ptr;

struct close
{
    PATH *pointer; /*pointer to path*/
    struct close *next; /*next element*/
};
typedef struct close Clist;
extern Clist *first,*last,*close_ptr;

extern int current_type; /*holds property of region*/

extern int region_number;

struct neighbors
{
    PATH *pointer; /*pointer to path*/
    struct neighbors *next;
};
typedef struct neighbors NEY;

extern int MAX_DEPTH;

```