VISUAL ROBOT GUIDANCE IN TIME-VARYING ENVIRONMENT USING QUADTREE DATA STRUCTURE AND PARALLEL PROCESSING.

A Thesis Presented to
The Faculty of the College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
ANIL R. BOHORA
June, 1989.
# TABLE OF CONTENTS

CHAPTER 1  
INTRODUCTION ........................................  1  

CHAPTER 2  
STATEMENT OF PROBLEM .................................  9  

CHAPTER 3  
DEVELOPMENT AND IMPLEMENTATION OF VISION  
ALGORITHMS FOR PREPROCESSING THE IMAGE ........  15  
3.1 Description of the Vision Algorithms ............  16  
3.2 Hypercube Machine Implementation of the Low  
Level Vision Algorithms ...............................  27  

CHAPTER 4  
DEVELOPMENT OF PATH PLANNING ALGORITHM USING  
QUADTREE DATASTRUCTURE ............................  35  
4.1 Overview of Quadtrees ...............................  38  
4.2 Neighbor Finding Algorithm .........................  46  
4.3 Path Planning Algorithm ............................  49  
4.4 Development of the Algorithms .....................  52  
4.5 Algorithm for Converting a Quadtree to an Array  64  

CHAPTER 5  
DEVELOPMENT AND IMPLEMENTATION OF THE VISUAL  
ROBOT GUIDANCE ALGORITHM ............................  70  
5.1 Equation for Collision Free Path ....................  71  
5.2 Equations for Moving Objects .......................  72  
5.3 Algorithm to Find Collision Points .................  73
5.4 Implementation to Find Collision Points ....... 77

CHAPTER 6

IMPLEMENTATION AND EXPERIMENTAL RESULTS ....... 84

6.1 Description of the Path Planning Program ....... 86

6.2 Experimental Results .................................. 90

CHAPTER 7

CONCLUSIONS AND DISCUSSION .......................... 97

REFERENCES .................................................. 99

APPENDIX A ................................................. 105
CHAPTER 1

INTRODUCTION

Robot navigation is a process of finding a collision-free path for a robot and guiding it through obstacles in a stationary or time varying environment. This process plays an important role in many applications of robotics and automation. Human beings demonstrate this ability with great skill. When driving a car, for example, numerous obstacles exist. These objects may be moving or they may be stationary. Despite the complex and varied nature of these obstacles, they are avoided easily even by relatively inexperienced drivers.

The path planning can be divided into two categories. The first is robot path planning and the second is manipulator path planning. Robot path planning in dynamic environment differs from the manipulator path planning in many aspects. This is because the manipulators follow the same path in their operations, while robots may follow different paths at different time instances. In addition, a robot does not have all the necessary information about its environment, and it has to find its path when it is moving along the certain direction. It obtains the visual description of the time varying environment from a sequence of successive picture frames and based upon this information it determines its path.
The robot path planning can be further divided into two categories: global path planning and local visual guidance. In global path planning the potential route from start to the end point is known. This route is calculated and stored in the memory before hand based on some a priori knowledge about the environment. Therefore, this type of path planning fails in the case of the robot moving in an unknown dynamic environment. The local visual guidance considers only the immediate environment. Using that knowledge the robot finds its immediate path avoiding the obstacles and trying to reach to the final goal position. This research addresses the problem of determining how a robot should move in order to avoid stationary or moving objects by using successive picture frames.

Considerable amount of research efforts has been devoted to study the robot path planning. For example, use of the ultrasonic range finders have been proposed to avoid collision with unexpected obstacles by means of detection and mapping [1-2]. Ultrasonic range measurements, however, suffer from some fundamental drawbacks, which limit the usefulness of these devices in mapping or in any other task requiring high accuracy. These drawbacks are inherent in almost all of the ultrasonic range finders. This is because the amount of reflected sound energy depends strongly on the surface structure of the obstacles. To obtain highly diffusive
reflection from an obstacle, the size of the irregularities on the reflecting surface should be comparable to the wavelength of the incident soundwaves. This requirement may not be met in the real environments which comprise mostly smooth surfaces such as walls, polished wood, metal surfaces, etc.

Hasegawa and Terasaki [3] have proposed a collision avoidance based on divide and conquer approach by using space characterization and intermediate goals for manipulators. This algorithm can not be directly applied to the moving robots since the robots usually follow different paths at different time in contrast to the manipulators which follow the same path repeatedly.

Hwang and Ahuja [4] have used potential field representation for path planning. The rough paths are found based only on topological information which is accomplished by assigning each obstacle an artificial potential similar to the electrostatic potential. This is used to prevent the moving objects from colliding with the obstacles by locating minimum potential valleys. The paths defined by the minimum potential valleys are modified to obtain an optimal collision free path. This algorithm takes considerable computational time (in the range of five minutes) to compute the potential functions, which are used extensively in all phases of the algorithm. In addition, a more efficient way to compute the
best candidate path can improve the performance of the method.

Filani and Mutch [5] have proposed a visual robot guidance method in a dynamic environment utilizing image sequences from two cameras. Stereoscopic motion analysis is performed first and then the path planning is achieved subsequently. However, as the authors conclude, the speed of this implementation is limited by the image analysis techniques rather than the guidance calculations. Locating the object in each image and determining its features such as centroid and displacement require considerably more time than analyzing the motion.

Kambhampati and Davis [6] have used quadtree data structure for path planning for mobile robot. The criterion they have used for selecting the best path is the minimum distance criterion. However, in the dynamic environment shortest distance is not necessarily the best path for all the cases. Because although the robot selects the minimum distance path using the knowledge of present picture frame, this path may be blocked by a moving obstacle in the next picture frame. Therefore, the robot has to calculate the minimum distance path again.

Luh and Klaasen[21] have described a 3-D vision system for on line operation of an industrial robot to avoid collision. The bounding box of each obstacle is constructed
for collision avoidance. Wong and Fu [22] have used a hierarchical orthogonal space approach to 3-D path planning in environment clustered with obstacles. 3-D collision-free path planning is performed in the three orthogonal 2-D projections of the space. Collision checking is done in each of the three 2-D subspaces using primitive path segments. A hierarchical-path search method is used to speed up the search process.

The existing path planning techniques do not consider motion parameters. Here we describe a time efficient motion path planning in dynamic environment using quadtree data structure and parallel processing techniques. A sequence of images of the top view of the dynamic environment is used for path planning. The motion analysis is performed on each of the image in sequence. Results of the motion analysis are used for selecting the best path as the shortest distance nearer to the obstacle which is moving away from the camera or from a reference point. Since the described path planning algorithm demands a preprocessed image as an input data, the raw image is preprocessed using histogram-based segmentation, noise cleaning, and motion analysis algorithms. To reduce the overall computation time, most of the preprocessing is performed on a 16-node hypercube concurrent processor. To cope with massive data search in path planning, quadtree data representation of the segmented image is formed for efficient
Since the proposed method requires the low level motion analysis of the obstacles we first describe the existing motion analysis techniques. Conventional motion analysis methods can be divided into two broad categories. In the first category the methods use stereo images for determining the motion of an object [7-10]. In these methods the object motion is determined from a sequence of stereo images by examining three dimensional features, establishing correspondence between these features, and finally computing the rigid motion parameters. Three-dimensional features are extracted from the depth map of the scene. The use of three-dimensional features provide sufficient information to find the rotation and translation components of motion via solving linear equations. However, large amount of computation is required by these methods.

In second category the methods use a sequence of two-dimensional images of scenes [11-13]. These methods represent movements of obstacles in three dimensional space from a sequence of two dimensional images. They determine the objects' movements towards or away from the camera as well as left/right and up/down movement in the image plane. The motion information is used as a part of the hierarchical matching process that determines the correspondence between the objects in different images. In this research, the
results obtained by the algorithm from the second category are used to analyze motion because we have used sequence of two-dimensional images.

After preprocessing, the merged quadtree of the segmented image is formed which is then used to search the collision free path for the robot motion.

The visual guidance is achieved by solving the equation of path and the equations of the motions of the moving objects. The collision points and the time instances of the collision are found. The previously found path is modified only at the time instant prior to the expected collision. This process is implemented on 16-node concurrent processor. The related research work is discussed below.

Fujimura and Samet [23] have presented an approach for path planning among moving obstacles using the concept of accessibility. A point is said to be accessible from a robot's initial location if there exists a direction of motion such that robot meets that point without a prior interception by any other obstacle movements. The concept of accessibility can be used effectively in planning a path only for the obstacles whose motions are piecewise linear.

Kehtarnavaz and Li [24] have introduced a framework for estimating a collision-free path in a navigational environment consisting of moving obstacles. An autoregressive
model is used to predict the future positions of the moving obstacles. A massive amount of data processing is necessary to predict the exact future position of the obstacle.

Zhu[25] has used structural pyramids for representing and locating moving obstacles in visual guidance and navigation. The obstacles represented and located in terms of the grid cells in structural pyramids. Orthogonal projections are applied to calculate the velocity of the moving objects and to predict the possible collision.

In this research we have found the equation of path and equation of moving objects using Lagrange's interpolation formula. These equations are solved simultaneously to find the points of collision. The velocity information is used to find the expected time instances of the collision.

The remainder of this thesis is organized as follows. Chapter 2 describes the statement of the problem. Chapter 3 presents the development and implementation of low level vision algorithms for preprocessing the image. Chapter 4 is devoted to the development and implementation of path planning algorithm using quadtree data structure. Chapter 5 describes the visual guidance algorithm. Experimental setup and results are described in Chapter 6. Finally, Chapter 7 provides the conclusions and the further research topics.
This research addresses the problem of finding a path for robot in order to avoid an obstacle, which itself may be moving. A time efficient robot path planning in dynamic environment is achieved using quadtree data structure and parallel processing. We have used sequence of the top views of the dynamic environment for path planning. As the long term conventional path planning does not involve any motion analysis it cannot be used directly in dynamic environment. We have used the result of motion analysis for selecting the best path. Because of the use of the motion information for the path selection, the probability of blocking the path by an obstacle in dynamic environment is reduced. The development and implementation of path planning algorithm demand a preprocessed image as an input data. The raw image is preprocessed using histogram-based segmentation, noise cleaning, and low level motion analysis algorithms. To reduce the overall computation time, the preprocessing is performed on a hypercube 16-node processor. The path planning algorithm always require a massive data search. The search time can be reduced by selecting proper data representation. Here, quadtree data representation is used for efficient search algorithm.
For the underlying motion analysis, the image sequence is first segmented using several low-level vision operations which require only repetitive local two-dimensional window convolutions on every pixel of the image. These local operations involve only pixels which are in the neighborhood of the one being processed. This iterative characteristic and the data locality feature make them particularly well-suited for the parallel processor implementation. In such implementation, each processor can be assigned a part of the whole image to perform the local operations independently from other processors except possibly at the subimage boundaries. Interaction between processors are, therefore, limited to exchanging only boundary pixel values. Hence, if the communication aspects are properly handled, significant reduction in processing time can be achieved.

In order to improve the performance of the proposed method, all the preprocessing operations are implemented on a 16-node hypercube parallel processor (AMETEK S-14) by exploiting its network-embedding feature. By parallelizing these basic vision algorithms with the use of hypercube concurrent processor, considerable speedup is achieved.

After preprocessing, a merged quadtree is formed to represent a segmented image. In this representation the uniform square image regions are assigned as the nodes of the quadtree in a hierarchical structure. Given a start and an
end point of the robot's motion path, the corresponding leaf
nodes are found in the merged quadtree. Then, the four
neighbors of the start point in all four directions in the
spatial plane are determined using the start node of the tree.
The neighbors found have size equal to or greater than that
of the robot. This criterion reduces the search time, because
the amount of data to be processed reduces. But in the cases
as shown in Figure 2.1 the space near the obstacle cannot be
used for robot motion. If a neighbor with these
characteristics is not found, then it is decided that there
is no path available in that direction. For efficient
implementation of this search, a set of stacks is used to
contain neighbor and path information. All nonobstacle
neighbors are pushed onto the stack in the decreasing order
of their nearness to the obstacles. The neighbor node which
is farthest from the obstacle is popped from the stack and is
checked for terminating the process. If the end condition is
not satisfied or the neighbor stack becomes empty, it is
concluded that there is no free path available and the robot
is trapped. If an end condition is met, the corresponding
path is accepted as the first available path. However, if
the neighbor stack still contains some nodes, then there may
be alternative paths available. The above procedure is
repeated until the neighbor stack becomes empty. All the
determined paths are stored on different stacks. The best
possible path is then selected based on the motion analysis
results.

The visual guidance is achieved by solving the equation of path and the equations of motion for the moving objects. To find these equations the Lagrange's interpolation formula is used. The robot will follow the same collision free path unless and until it expects the collision with the moving object. The top view is again processed to find the new collision free path only prior to the expected collision. The flowchart of this process is shown in Figure 2.2.
Figure 2.1 Image plane decomposition for path planning
Take the top view & preprocess it

Plan the robot path using top view

Planned path is stored on stack

Pop the co-ordi. of the next step from the stack

Using Lagrange's interpolation formula find the equation of path and equations of moving objects to check if there will be collision

will there be a collision?

Y

N

Take the next step

Figure 2.2 Flowchart of the algorithm
The development and implementation of high level vision algorithms require a preprocessed image as an input data. For this purpose different low level vision algorithms are applied to the raw image to extract the necessary picture domain features. This is achieved by the histogram-based segmentation, noise cleaning, and low level motion analysis algorithms.

Most of the low level vision operations usually require massive amount of computations. However, these computations often involve only point or local operations. For point operations, the output gray level at a pixel depends solely on the input gray level at the same pixel. Image thresholding, for example, is one of such operations. As for local operations, the output gray level at a pixel depends only on the input gray levels in a neighborhood of that pixel. Global information is not needed in either point or local neighborhood operations. Therefore, this data locality feature makes the low level vision algorithms particularly well suited for parallel processor implementation.
The detailed description and parallel implementation of low level vision algorithms used for preprocessing the raw image are presented in this chapter. These operations are histogram generation, thresholding-based image segmentation, and noise cleaning. Several issues involved in parallel algorithm development, such as the efficient image decomposition, suitable topology selection, and proper internode communications are also discussed.

3.1 Description of Vision Algorithms

A brief description of the above vision algorithms is given in the following subsections. Their ring topology implementation is also described.

3.1.1 Histogramming and Thresholding

A histogram is the frequency distribution of the gray levels in an image. It is often used to convert a gray level image into a binary image by means of thresholding. If the gray level of an image is in the range from 0 through p, then the histogram value of gray level p is given as \( h(p) \), where \( h(p) \) denotes the number of pixels corresponding to the gray level p in that image.

In its sequential implementation, histogram generation is also considered as point operation. However, since different parts of the image reside in different
processors, parallel implementation would require global information in order to sum the partial histograms generated by each of the N processors. Therefore, it can no longer be considered as point operation. In this case, it is classified as statistics computation. In parallel processing environment, proper coordination among processors is particularly important in order to correctly condense the histogram into the node 0 within a reasonable period of time.

Segmentation process is performed by converting the raw image \( G \) into a binary or bilevel form \( B \) which is then used for the motion analysis and path planning. This conversion is carried out by comparing each of the image pixel with some predetermined threshold values \((T_1 \text{ and } T_2)\). This process is defined as

\[
B(i,j) = \begin{cases} 
1 & \text{ if } T_1 < G(i,j) < T_2 \\
0 & \text{ otherwise}
\end{cases}
\]

Noise in the raw image is cleaned up considerably by this thresholding process. The binary image is then coded in such a way that object, background and path have different uniform gray level values (Figure 3.1).
Figure 3.1 Segmented image representation
3.1.2 Noise Cleaning

In the thresholded image, there are still some cells that are misclassified due to noise and image imperfections. Unless the signal-to-noise ratio is poor, almost all of these errors will be isolated, except near the object boundaries. This is called salt-and-pepper noise. These errors can be removed by noting that a particular cell is in a complementary neighborhood.

One can perhaps avoid making this kind of error by considering the gray levels at neighboring points before thresholding. If a particular picture cell has a gray level that is higher than the highest of the gray levels of its neighbors, it may have been unduly affected by noise. It should then be replaced by the highest of the gray levels of the neighbors. A similar filtering operation can be applied if the gray level is lower than the lowest gray level of the neighbors. The gray levels of the picture cells on the boundary between the objects and the background will not be affected by this operation. In addition, this algorithm puts considerable overhead on the thresholding algorithm.

Other filtering operation such as averaging of neighboring cells can be used to reduce noise. We have used this method of noise cleaning in this research. Any remaining isolated noisy pixels in the image are filtered with a pair
of noise cleaning templates as shown in Figure 3.2. This operation tends to smear out sharp boundaries and thus reduces spatial resolution. As a result, it exacerbates the problems encountered at boundaries. However, it should be noted that, because of the 3 X 3 window size of the templates, only completely isolated noisy pixels can be discarded. Any remaining two or more pixels wide isolated noisy segments would not be eliminated unless the template sizes are increased.

3.1.3 Low-level Motion Analysis

A. Motion Detection

The goal of the low level vision system under discussion is to develop a method that helps to predict the position of an object in the next image using motion information. The segmented image is used as an input data for motion detection algorithm.

As described above, in the segmented image object regions corresponding to, obstacles, background, and path have different gray level values as shown in Figure 3.1. First, using this image, areas of all the objects are calculated as follows:

\[ A = \sum_{i} \sum_{j} b(i,j) \quad \ldots \quad 3.1 \]

where \( b(i,j) \) is the binary image function.
Figure 3.2 Noise cleaning templates
Next the object position is determined in the image plane. Since the object is usually not just a single point, we chose the center of area as the point representing the object position in the image. The center of area is the center of mass of a figure of the same shape with constant mass per unit area. The center of mass, in turn, is that point where all the mass of the object could be concentrated without changing the first moment of the object about any axis. In the 2-D case the co-ordinates of the center of area are

\[
\bar{i} = \frac{\sum i \cdot b(i, j)}{A} \quad \text{... 3.2}
\]

\[
\bar{j} = \frac{\sum j \cdot b(i, j)}{A} \quad \text{... 3.3}
\]

The centroid calculations are performed for all the objects present in the image. The \( i \) and \( j \) displacement of an object between frames is calculated from the centroid of the visible portion of the object. As shown in the Figure 3.3 the centroid \( C_1(i_1, j_1) \) in the first image frame is calculated and stored. After the processing of the next picture frame centroid \( C_2(i_2, j_2) \) is found. This information is used to find the motion of the object. However, this centroid computation will be inaccurate for the partially occluded objects.
Figure 3.3 Example of motion analysis for path selection
Tracking individual features such as neighborhood information is necessary in such cases.

Next, the relative depth information is determined from the segmented image sequence. The depth measurement provides enough information to find which object is closest to and which object is farthest from a reference; i.e., camera. To find accurate depth information, generally stereo motion analysis is required. If the objects are moving towards or away from the camera, the sequence of 2-D images can also provide the relative depth information. This is because, if an object is approaching the camera, its image area increases due to perspective and vice versa. In this research, movement in depth is defined as the ratio of the area of the object in the current picture frame to the area of the same object in the previous picture frame.

\[
\text{depth} = \frac{\text{area in new frame}}{\text{area in old frame}}
\]

This ratio provides the following motion information:

If \( \text{depth} = 1 \); object is stationary or moving in horizontal or vertical direction,

\( \text{depth} > 1 \); object is coming towards the camera,

\( \text{depth} < 1 \); object is going away from the camera.

The mathematical background for this is shown in the Figure 3.4. As shown in the figure, if the position of the camera
Figure 3.4 Mathematical background for finding relative depth

\[
\frac{i_1}{d_i} = \frac{O}{do_1} \quad \text{and} \quad O = \frac{do_1 \times i_1}{d_i}
\]

\[
\frac{i_2}{d_i} = \frac{O}{do_2} \quad \text{and} \quad O = \frac{do_2 \times i_2}{d_i}
\]

\[
\frac{do_1 \times i_1}{d_i} = \frac{do_2 \times i_2}{d_i}
\]

\[
\frac{i_1}{i_2} = \frac{do_2}{do_1}
\]

\[
\text{depth} = \frac{\text{area occupied by the object in new frame}}{\text{area occupied by the object in old frame}}
\]
is fixed and if the object has moved towards the camera then the area occupied by that object in the image increases and the vice versa is also true.

If the object is having a rotational and/or translational motion then the depth information cannot be extracted from a 2-D picture sequence using the above mentioned algorithm. In that case, stereo motion analysis is required.

B. Matching Objects Between Scenes

In general, object matching is difficult to achieve precisely. A flexible matching process has been accomplished by incorporating several matching methods. The proposed matching process involves two levels of matching. The highest level of matching depends on the velocity of the object (a global property) and the lowest level depends on the relationship between neighbors (local feature). Thus, matching is arranged in levels starting with global feature and going down to the local features to achieve a proper match. This flexible matching process can fail by not finding objects.

Motion matching normally proceeds by tackling each expectation in turn and searching through the list of objects in the scene for all objects whose centroid and neighbor satisfy the expectation. First, depending on the velocity
information of the object, the difference between the centroids is calculated using successive picture frames. If the difference is less than the provided threshold, the objects are assumed to be matched. If this method fails to match the objects, then the neighborhood information is used. The neighborhood information provides the information from the previous picture frame about objects' relative positions. It is assumed that the same objects remain neighbors, if both objects are moving in the same direction. If the objects satisfy this criterion then it is assumed that the objects are matched. After the matching is finished, the extracted information about the objects' motion is stored for the further use.

3.2 Hypercube Machine Implementation of the Low-level Vision Algorithms:

The above discussed preprocessing algorithms are implemented on 16-node hypercube processor. An extensive research work in this area has been done by Celenk and Lim [26-27]. In this research we have applied the above discussed algorithms to multiple number of objects.

3.2.1 Topology Selection

Matching parallel algorithm with the interconnection network structure is often a difficult task. Before selecting
any particular topology to solve a given problem, it is important to know the data flow requirement between processors for performing specified task on the input data. Different topology used leads to different algorithm performance. If the topology employed well matches the underlying data structure, efficiency of the concurrent program can be significantly improved. This improvement is a result of the reduction of the interprocessor communications.

Because the whole image is subdivided among the processors, internode communication is inevitable in image processing task by the array processors. For local image operations, intercommunication occurs whenever operations involve some boundary data points that are stored in different processors. Operations can be carried out only if all the needed information directed from the various processors has been received. Therefore, topology that would permit fast data transfer among processors is the most suitable network for the local neighborhood operations.

Depending upon the selected topology employed, a subimage can either be in the form of $[M/N] \times M$ strip for the ring topology or $[M/N]^{1/2} \times [M/N]^{1/2}$ square matrix for the nearest neighbor topology as shown in Figure 3.5 for 512x512 image. In this work we have selected ring topology, because downloading a subimage in the form of a strip can be carried out with a
Figure 3.5 Image partitioning in ring topology
single download macro command in this configuration.

3.2.2 Ring Topology Implementation

Parallel processor implementation of the various vision algorithms are divided into the stages as shown in the block diagram of Figure 3.6.

The entire $M \times M$ image is subsequently partitioned into $N$ distinct subimages. Each subimage block is then downloaded or assigned to the available processor according to the mapping. Image processing tasks are performed concurrently in each node upon the receiving of the subimage. Output image of each stage is used as an input image for another cycle of computation in the next stage. Histogram of the raw image is generated to convert the gray level image into binary form by means of thresholding. Then, any remaining isolated noisy points in the binary image are filtered with a pair of noise cleaning templates.

After receiving the corresponding subimage from the host, all window operations are then performed locally in each node. Instead of waiting for the required boundary data to be sent by the neighbors, all necessary data items are gathered before engaging in the mask operation in a particular processing stage. Therefore, all required border strips are first copied from the neighbors prior to the processing stage. For the ring implementation, this involves only interaction of
Figure 3.6 Block diagram for preprocessing
neighbors on the left and on the right of a specific node, which makes the interprocessor communications straightforward.

The amount of data items that need to be transferred from the neighbors is dependent on the window size used by the algorithm. As presented in the previous section, all algorithms implemented in this research employ only 3 X 3 masks.

In the case of a 512 X 512 image and a 16-node hypercube processor, each node receives the corresponding 32 X 512 image portion downloaded by the host. By declaring the subimage array in the child program to be of size 34 X 512, the received subimage is then placed in the center of this larger array, which leaves the first and the last rows of the array free. This scheme facilitates storing of the boundary image pixels coming from both the left and the right neighbors. For the 3 X 3 window computations, the interprocessor communications begin by having all processors, except the node 0, sending their first image rows (rows 1) to their corresponding destination nodes on the left. Similarly, each node also receives a complementary row of data items from the right neighbor. This row of data is then appended to the end (row 33) of the subimage array. Since the node 0 is excluded from this operation, its corresponding left neighbor, which is node 15, does not receive any data from it. This is also true for the case when the last rows (rows 32)
of the subimages are transmitted to the right neighbors. In such a case, the node 0 does not receive any data from the node 15. At nodes other than node 0, the data rows obtained from the left neighbors are stored into the top rows (rows 0) of the subimage arrays. After all these communications, each subimage has been expanded to a size of 34 X 512 with the inclusion of the received boundary data rows.

Since the intercommunications involve only the immediate left and right neighbors, it is realized that all local window convolution tasks with varying mask size can be easily implemented using the ring topology. However, since processor intercommunications are also dependant on the amount of data being transferred, longer communication time is expected for transferring larger subimage blocks.

With the ring network, histogram generation requires processing time in the magnitude of the order N, since all partial results have to be added up in the node 0. A slight improvement can be achieved by breaking the ring into two halves and accumulating the partial results from the left and right neighbors at the same time. This method, however, still requires N/2 processing steps. All partial histograms generated in the non-corner nodes are shifted from each node towards the node 0 and are accumulated to form the total histogram.
This preprocessed image is then stored for the use of high level vision algorithms.
CHAPTER 4

DEVELOPMENT OF PATH PLANNING ALGORITHM
USING QUADTREE DATA STRUCTURE

The preprocessed image is used as an input image for the path finding algorithms. First the merged quadtree is formed, which is then searched to find the best available path. All the necessary algorithms developed for this process are discussed in detail in this chapter.

In the preprocessed image, each region is represented by different gray levels. Region representation is of prime importance because it contains the amount of data to be processed efficiently by the algorithm. As the amount of data to be processed reduces, considerable saving in time can be achieved. Therefore, proper selection of region representation is important for the efficient implementation of the algorithm. There is a variety of approaches to representing regions, based on their boundaries or their skeletons. Some of these techniques are reviewed here.

Any region boundary can be specified, relative to a given starting point, as a sequence of unit vectors in the principal directions. We can represent the directions by numbers. For example, as shown in Figure 4.1(a), four directions can be numbered as 0, 1, 2, 3 using 4-connectivity of
the segments. This type of boundary representation is called "chain code" representation [30]. Generalized chain codes, involving more than four directions, can also be used. Chain codes provide a compact region representation, which can be used to detect features of the region boundary such as corners or concavities. On the other hand, it is rather difficult to determine properties such as elongatedness from a chain code. It is also difficult to perform operations such as union, search, and interconnection on regions represented by chain codes.

Another class of region representations involves the use of various types of maximal blocks that are contained in a given region. For example, we can represent a region R as a link list of the runs of pixels, in which R meets the successive rows of the array. Here each block is a 1 X m rectangle, where m is the run length. The runs are the largest such blocks that R contains, and R is determined by specifying the initial points and lengths of the runs. This representation is called "medial axis transformation" [29-30]. Medial axis for some simple regions are shown in Figure 4.1(b). It is somewhat less compact than chain code representation, but it has advantage in performing union and intersection operations or detecting properties such as elongatedness. This representation is also not suitable for search algorithms.
Figure 4.1 Different forms of region representations
There has been recent interest in quadtree region representation based on successive subdivision of the image array into quadrants. In this approach, a quadtree represents square image regions by hierarchically partitioning the given image into quadrants and subquadrants until all sub-quadrants are uniform with respect to a predicate. This method of region representation was first proposed by Kilinger[20]. It is relatively compact and also well suited to operations such as union and intersection for detecting various region properties and for hierarchical search for finding path. Therefore, in this research we have selected quadtree data structure for region representation.

4.1 Overview of Quadtrees

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 bases:

1. The type of data that they represent.
2. The principle guiding mechanism used in the decomposition process.

They are used for representing points, regions, curves, surfaces, and volumes. The decomposition may be into equal parts on each level, or may be governed by the input. The resolution of the decomposition may be fixed, or it may be
governed by the properties of the input data.

The most well known and commonly used quadtree approach to region representation, termed as a region quadtree, is based on the successive subdivision of the image array into four equal-sized quadrants [14-17]. If the array does not consist entirely of 1's or entirely of 0's, it is then subdivided into quadrants, subquadrants, etc. until blocks that consist entirely of 1's or entirely of 0's are obtained as shown in Figure 4.2. Thus the region quadtree can be characterized as a variable resolution data structure. For example, consider the region shown in Figure 4.3(a) which is represented by the $2^3 \times 2^3$ binary array in Figure 4.3(b). It can be easily observed that the 1's correspond to pixels that are in a region of interest and the 0's correspond to pixels that are outside the region. The resulting blocks for the array of Figure 4.3(b) are shown in Figure 4.3(c). This process is represented by a tree of degree four so that each nonleaf node has four sons. The root node corresponds to the entire array. Each son of a particular 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 those blocks for which no further subdivision is necessary. A leaf node is said to be BLACK or WHITE, depending on whether its corresponding block is object or free path. All nonleaf nodes are considered to be GRAY. The
Figure 4.2 Quadtree representation
quadtree representation for Figure 4.3(c) is shown in Figure 4.3(d).

At this point it is appropriate to define a few terms. We use the term image to refer to the original 512x512 array of pixels. If its elements are either BLACK or WHITE then it is said to be binary. If all gray level values are possible for a pixel, then the image is said to be a gray-scale image. In our discussion we are primarily concerned with the binary images. The border of the image is the outer boundary of the square corresponding to the array. Two pixels are said to be four adjacent if they are adjacent to each other in the horizontal or vertical direction. If the concept of adjacency also includes adjacency at a corner; i.e., diagonal adjecencies, then the pixels are said to be eight adjacent. A pixel is said to have four edges, each of which is of unit length. The boundary of a BLACK region consists of the set of edges of its constituent pixels that also serve as edge of WHITE pixels. Similar definitions can be formulated in terms of blocks. For example, two disjoint blocks P and Q are said to be four adjacent if there exists a pixel p in P and a pixel q in Q such that p and q are four adjacent. Eight adjacency for blocks is defined analogously.

For a quadtree corresponding to a $2^n \times 2^n$ array, we say that the root is at level n and if the representation includes individual pixels, there will be a maximum of n
Figure 4.3 Example of building a quadtree
Figure 4.3 (cont.) Example of building a quadtree
Figure 4.3 (cont.) Example of building a quadtree

(c)
Figure 4.3 (cont.) Example of building a quadtree
levels in the tree.

4.2 Neighbor Finding Algorithm

The advantage of the quadtree representation is that many basic operations can be implemented as tree traversals. The difference among the operations is in the nature of the computation that is performed at the node. Often these computations involve the examination of nodes whose corresponding blocks are adjacent to the block corresponding to the node being processed. We shall speak of these adjacent nodes as neighbors. However, adjacency in spatial domain does not imply that any simple relationship exists among the nodes in the quadtree.

Each node of a quadtree corresponds to a block in the original image. We use the terms block and node interchangeably. The term that will be used depends on whether we are referring to a raster representation or to a quadtree representation. Let the four sides of a node's block be called as N, E, S, and W sides. The four subquadrants of a block are labeled NW, NE, SW, and SE with the obvious meaning. Given the nodes P and Q of two nonoverlapping blocks, and a direction D, we define a predicate Adjacent such that Adjacent(P, Q, D) is true if there exist two pixels p and q, contained in P and Q, such that either q is adjacent to side D of p or P is adjacent to side D of q. In such a case, nodes
P and Q are considered to be neighbors. Note that the adjacent relation also holds for nonterminal (i.e., GRAY) as well as terminal (i.e., leaf) nodes.

The neighbor relation does not have a functional definition in a mathematical sense. The problem is that given a node P and a direction D, there may be more than one node which is adjacent to P in accordance with different criteria. For example, nodes 38, 40, K, and D are all western neighbors of node N in Figure 4.3. This means that in order to specify a neighbor, more precise information is necessary to distinguish between neighbors that are adjacent to the entire side of a node (e.g., B is northern neighbor of J) and those that are only adjacent to a segment of a node's side (e.g., 37 is one of the eastern neighbors of J). An alternative criterion for the selection of the neighbor is that in the former case we are interested in determining a node Q such that its corresponding block is the smallest block of size greater than or equal to the block corresponding to P, whereas in the latter case we specify the neighbor in greater detail by indicating the corner of P to which Q must be adjacent.

In the remaining discussion the term "P is neighbor of Q" will imply that P is node of size greater than or equal to Q. A block that is not adjacent to a border of the image has a minimum of four neighbors. This can be seen by observing that a node can not be adjacent to two nodes of
greater size on opposite sides. This is because of the reason that a split of block creates four subblocks of equal size.

As mentioned above, most operations on quadtrees can be implemented as tree traversals by examining the neighbors of a selected node in the quadtree. This operation should be performed by locating neighbors independent of both position and size of the node. Moreover this search should not include any additional links to adjacent nodes. In other words, it is desired to use only the structure of the tree and no pointers in excess of the four links from a node to its four sons and one link to its father [18-19].

In locating adjacent neighbors in the horizontal or vertical directions, the basic idea is to ascend the quadtree until a common ancestor with the neighbor is located, and then descend down the quadtree in search of the neighboring node. It is obvious that we can always ascend as far as root of the quadtree and then start our descend. However, our goal is to find the nearest common ancestor, as this minimizes the number of nodes that must be traversed.

Suppose, for example, that we want to find the western neighbor of the node N in Figure 4.3(d). The nearest common ancestor is the first ancestor node of which N is not a western descendant. Next, we refer the path used to locate the nearest common ancestor, except that we make mirror image
moves about an axis formed by the common boundary between the nodes. In the case of a western neighbor, the mirror images of NW and SW are NE and SE, respectively. Therefore, the western neighbor of node N in Figure 4.3(d) is node K. It is located by ascending the quadtree until the nearest common ancestor A has been located. This requires going through a NW link to reach node E, and a SE link to reach node A. Node K is subsequently located by backtracking along the previous path with the appropriate mirror-image moves; i.e., by following a SW link to reach node D, and a NE link to reach node K.

Neighbors in the horizontal or vertical directions need not correspond to blocks of the same size. If the neighbor is larger, then only part of the path from the nearest common ancestor is retraced. Otherwise, the neighbor corresponding to a block of equal size and pointer to an appropriate BLACK, WHITE, or GRAY node is returned. If there is no neighbor; i.e., the node whose neighbor is being sought, is adjacent to the border of the image in the specified direction, then NULL pointer is returned.

4.3 Path Planning Algorithm

Conventional path planning algorithms can be divided broadly into two categories. In the first category, the methods make trivial changes to the representation of the
image map before planning a path. The regular grid search and vertex graph method fall in this category. Although these methods keep the representational cost to a minimum, their applicability to mobile robot navigation is limited. For example, the regular grid search is too local and its path planning cost increases with grid size rather than with the number of obstacles present. Further, both regular grid search and vertex graph methods generate paths which clip obstacle corners.

The methods in the second category make elaborate changes in representation to convert to a form, which is easier to analyze before planning the path. Free space methods, medial axis transform methods, vornoi methods, etc., fall in this category. A potential practical shortcoming of such methods for mobile robot navigation is that the path planning cost is still very high because of the representation conversion process involved.

Although the above two categories by no means exhaust the existing methods, they point out that what mobile robots need may be some compromise between these two categories. In this regard, we have developed an algorithm for robot path planning based on the quadtree representation of the robot's immediate environment. If there are large areas of free space or obstacles, then those areas can be represented by a few large blocks in the quadtree and can be
dealt with as units by the planning algorithm.

Compared to the first category of path planning algorithm mentioned, such as grid search method, the path planning cost for quadtree based search will be substantially lower because the number of nodes to be searched in the quadtree approach is considerably smaller. The search in quadtree has to deal with \( n \) nodes, instead of the \( n^2 \) grid points as in the case of a grid search. This results in a substantial reduction in the search space. Similarly the local bound behavior of the first category algorithms is absent in this approach, because the nodes are much larger than single pixels and it is straightforward to determine the nearness of the nodes to the obstacles. Moreover, a hierarchy of different levels of description of the space that is available with quadtrees provides a search strategy so that a path close to obstacles is sought only when necessary. Corner clipping of inflexible paths are eliminated by considering only neighbors in the horizontal and vertical directions.

Unlike methods of the second category that involve a costly change of representation, the proposed approach has a very small representation overhead. The representation algorithm using quadtree is of complexity \( O(n) \), whereas many methods of the second category have a representational cost
that is far higher.

Thus quadtree based path planning is a good compromise between free space algorithms and grid search type schemes. In addition, the path produced by the quad tree algorithm, although not optimal, is acceptable path which can be computed relatively quickly. Apart from this, the hierarchical nature of the representation gives many advantages in path planning. For example, we can easily constrain the path to satisfy certain conditions such as specification of minimal clearance of the path. We can also make the search staged; i.e., plan a path at a coarser level and subsequently refine it as needed. This reduces cost of the planning considerably.

4.4 Development of the Algorithms

4.4.1 Building a Quadtree

The input image of a $2^n \times 2^n$ array of unit square pixels is first converted to a binary form by means of segmentation. The quadtree representation of the resultant binary image is then obtained based on successive subdivision of the array into quadrants. In essence, we repeatedly subdivide the array into quadrants, subquadrants, and so on until we obtain blocks which consist entirely of either 1's or 0's. This process is represented by a tree of degree four in which the root node represents the entire array, the four
sons of the root node represent the quadrants, and the terminal nodes correspond to those blocks of the array for which no further subdivision is necessary. In general, BLACK and WHITE nodes represent blocks consisting entirely of 1's and 0's respectively. GRAY nodes denote nonterminal nodes, which have four sons, each corresponding to a subquadrant. If the tree is computed in a top-down fashion by examining the entire picture, then its quadrants, then their quadrants etc., this process may require excessive computational effort, since some parts of the picture that contain finely divided mixtures of 0's and 1's will be examined repeatedly. As an alternative, we can build the quadtree in bottom-up fashion by scanning the picture in a suitable order. This is analogous to a postorder tree traversal. For example, the pixels in the binary array of Figure 4.4 are labeled in the order that they have been examined. If the array is denoted by A, we can see that A[1,1] is examined first followed by A[1,2], A[2,1], A[2,2], and so on. However, a node is only created if it is maximal, in other words, if it is not used in any further merge operations.

Each node in the quadtree is stored as a structure containing seven fields. A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. The individual fields of the structure may themselves be the type
<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>5</th>
<th>6</th>
<th>17</th>
<th>18</th>
<th>21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>19</td>
<td>20</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>25</td>
<td>26</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>15</td>
<td>16</td>
<td>27</td>
<td>28</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>34</td>
<td>37</td>
<td>38</td>
<td>49</td>
<td>50</td>
<td>53</td>
<td>54</td>
</tr>
<tr>
<td>35</td>
<td>36</td>
<td>39</td>
<td>40</td>
<td>51</td>
<td>52</td>
<td>55</td>
<td>56</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>45</td>
<td>46</td>
<td>57</td>
<td>58</td>
<td>61</td>
<td>62</td>
</tr>
<tr>
<td>43</td>
<td>44</td>
<td>47</td>
<td>48</td>
<td>59</td>
<td>60</td>
<td>63</td>
<td>64</td>
</tr>
</tbody>
</table>

Figure 4.4  Image decomposition schematic
structure. This is known as recursive type of structure. The recursive type of structure is suitable for building a tree like structures. Dynamic memory allocation is also possible and excessive amount of data can be handled without having any static memory problems. Therefore, in this research we have used recursive type of structure for data manipulation. The first five fields of the structure contain pointers to the node's father and its four sons, labeled NW, NE, SE, and SW. The sixth field, named NODETYPE, describes the contents; i.e., WHITE, BLACK, or GRAY, of the block which the node represents. The seventh field, named FLAG, describes the start and the end node. This field is also used in path finding algorithm by containing the path information about that node as being a part of the total path or not.

4.4.2 Development of Neighbor Finding Algorithm

A pair of nodes is regarded as neighbors, if their quadrants have an entire side common. If only the corners are common then those nodes are not regarded as neighbors. The neighboring quadrants are path-wise connected. As mentioned previously each node is stored as a record containing seven fields. If P is a node and I is a quadrant, then these fields are referenced as FATHER(P) and SON(P,I), respectively. We can determine the specific quadrant in which a node, say P, lies relative to its father by use of the function SONTYPE(P), which has a value of I if
SON(FATHER(P),I) = P. As shown in Figure 4.5, the relative position of node P w.r.t. its father Q is found using the function SONTYPE(P).

Let the four sides of a block be called as its N, E, S, and W sides. They are also termed as boundaries and we may also refer to them as if they are directions. We define the following predicates and operations involving a block's quadrant and its boundaries. ADJ(B,I) is true if and only if quadrant I is adjacent to boundary B of the block; e.g., ADJ(W,SW) is true. REFLECT(B,I) yields the SONTYPE value of the block of equal size that is adjacent to side B of a block having SONTYPE value I. Figure 4.6 shows the relationship between the quadrants of a node and its boundaries. As shown in the figure REFLECT(N,SW) = NW, because NW is a neighbor of SW in N direction.

For a quadtree corresponding to a $2^n \times 2^n$ array we say that the root is at level n, and that the node at level i is at a distance of n-i from the root of the tree. In other words, for a node at level i, we must ascend n-i FATHER links to reach the root of the tree. Note that the farthest node from the root of the tree is at a level greater than or equal to 0. A node at level n corresponds to a single pixel in the image. Also, we say that a node is of size $2^s$ if it is found at level s in the tree.
Let \( I = 1 \) \( \text{SON}(Q, I) = A \):

Let \( I = 2 \) \( \text{SON}(Q, 2) = P = P \)

\[
\text{SONTYPE}(P) = \text{SON}(\text{FATHER}(P), 2) = P
\]

\( I = 2 \) \( P \) lies in second quadrant relative to its father \( Q \)

\[
\text{SONTYPE}(P) = \text{SON}(\text{FATHER}(P), I)
\]

\( = \text{SON}(Q, I) \)

Let \( I = 1 \) \( \text{SON}(Q, 1) = A \neq P \)

Let \( I = 2 \) \( \text{SON}(Q, 2) = P = P \)

\[
\text{SONTYPE}(P) = \text{SON}(\text{FATHER}(P), 2) = P
\]

\( I = 2 \) \( P \) lies in second quadrant relative to its father \( Q \)

Figure 4.5 Finding quadrant of a node relative to its father using the function \( \text{SONTYPE}(P) \)
Figure 4.6  Relationship between a block’s four quadrants
and its boundaries, where S represents the direction
and Q represents the quadrant.

<table>
<thead>
<tr>
<th></th>
<th>NW</th>
<th>NE</th>
<th>SW</th>
<th>SE</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>S</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>W</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>NW</th>
<th>NE</th>
<th>SW</th>
<th>SE</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>SW</td>
<td>SE</td>
<td>NW</td>
<td>NE</td>
</tr>
<tr>
<td>E</td>
<td>NE</td>
<td>NW</td>
<td>SE</td>
<td>SW</td>
</tr>
<tr>
<td>S</td>
<td>SW</td>
<td>SE</td>
<td>NW</td>
<td>NE</td>
</tr>
<tr>
<td>W</td>
<td>NE</td>
<td>NW</td>
<td>SE</td>
<td>SW</td>
</tr>
</tbody>
</table>
Given a node corresponding to a specific block in the image, its neighbor of equal or greater size in the horizontal or vertical direction is determined by locating a common ancestor. Next we retrace the path while making mirror image moves about an axis formed by the common boundary between the blocks associated with the two nodes. If the neighboring node does not exist, then we return a GRAY node of equal size if possible. Otherwise if the node is adjacent to the border of the image, NULL is returned since there is no neighbor in the specified direction. When a node does not have a neighboring terminal node of equal or greater size, returning a GRAY node of equal size is reasonable because the given node whose neighbor is being sought has more than one neighboring terminal node in the given direction. The process of locating eastern neighbor of node A is shown in Figure 4.7.

4.4.3 Development of Path Planning Algorithm

Given a start and an end point we first determine the corresponding quadtree leaf nodes S(for start node) and F(for end node). We change the FLAG fields of the structure representing those nodes. Then we find the four neighbors, for the start node, in all four directions. All non obstacle nodes are pushed onto the stack, named psn, in the decreasing order of their nearness to the obstacle. Then the neighbor node which is farthest from the obstacle is popped from the psn stack and it is checked for the end condition. If the end
(a) Block Representation

FIGURE 4.7 Process of locating eastern neighbor of node A
FIGURE 4.7(cont.) Process of locating eastern neighbor of node A
condition is met, the procedure is terminated. Otherwise that node is pushed onto the stack named psp[pathno], where initially pathno is 1. This procedure is repeated until the end condition is satisfied or the psp stack becomes empty. This means that there is no free path available and robot is trapped.

If the end condition is achieved for the first time while the psp stack is not empty, this means that there may be alternative paths available. Then the above procedure is repeated until the psp stack becomes empty. All detected paths are stored in psp[pathno] stacks, where pathno is increasing variable. The structure of the stack and the stack are shown in Figure 4.8.

The best possible path is selected based on the nearness to the obstacle, and the shortest distance criteria. The nearness of the node to the obstacle is checked using the motion analysis results and the following property of the quadtree: If the quadtree node, which is the part of the path, is nearer to the obstacle nodes in the quadtree, then corresponding path is nearer to the obstacle. The motion analysis results guides to select the path which is nearer to the object moving away from the camera. This helps to prevent the path getting blocked by a moving obstacle. The length of the path is then calculated using the number of nodes traveled
<table>
<thead>
<tr>
<th>Co-ordinates of start point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Co-ordinates of step 1</td>
</tr>
<tr>
<td>Co-ordinates of step 2</td>
</tr>
<tr>
<td>Co-ordinates of step 3</td>
</tr>
<tr>
<td>Co-ordinates of step 4</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Co-ordinates of end point</td>
</tr>
</tbody>
</table>

Figure 4.8 The structure of stack used to store the path information
by that path.

Finally, the backtracking of the path is achieved by the popping out the values of the nodes from the stack of the best path. The gray level values of nodes popped out are changed so that we can observe that path on the screen of the display device. The detail flow chart of this process is shown in Figure 4.9.

4.5 Algorithm for Converting a Quadtree to an Array

A method is needed to convert a quadtree representation into a raster image form. Most of the algorithms traverse the quadtree by rows and visit each quadtree node once for each row that intersects it. Some of the algorithms are top-down type approaches and others are bottom-up type schemes. The bottom-up algorithms visit adjacent blocks by use of neighbor finding techniques, whereas the top-down methods start at the root, each time visiting a node. The bottom-up methods are more suitable for the high resolution images since the number of nodes that must be visited in locating neighbors is smaller than the number of iterations starting from the root. All the algorithms have execution times that depend only on the number of blocks in the image and not on their particular configurations.
Read the input image from VICOM

Process the i/p image to produce segmented binary image

Build the merged quadtree

Take the start and end points for the robot motion and find the respective nodes in quadtree

Find four neighbors of start node

Are nodes satisfying required criteria?

Y

A

N

Don't store on psp

C

D

Figure 4.9 Flowchart of the algorithm
Store them on the psp stack with decreasing order of nearness to the obstacle

Is stack empty?

Y

Trapped

N

Pop the first element from psp & push that on psn[pathno] stack

Is end condition achieved?

N

start node=new popped node

C

Y

Path found is true

B

Figure 4.9 (cont.) Flowchart of the algorithm
Figure 4.9(cont.) Flowchart of the algorithm
The method we have used to generate a 512*512 array corresponding to a quadtree is a top-down approach. We start to traverse the tree from the root using depth first search technique, also known as an in "order tree traversal". In this type of traversal, we go left until a terminal node is reached. After reaching to the terminal node, depending on the level of the node and the starting address of the rows and the columns, we fill the array with the nodetype value (i.e., 1 or 0) of this node. Then we go back up one level, go right, and then go left until a terminal node is encountered. The example of this process is shown in Figure 4.10. This procedure is repeated until all the terminal nodes in the quadtree are examined. The end result of this procedure provides the array representation of the quadtree in which the path for the robot is also shown with differentiative gray level value.
Figure 4.10 The depth first search technique
CHAPTER 5

DEVELOPMENT AND IMPLEMENTATION OF THE VISUAL ROBOT GUIDANCE ALGORITHM

As discussed in previous chapter the collision free robot path is found using the top view of the dynamic environment. The robot will follow this collision free path unless and until it expects a collision with the moving object. The top view is again processed to find the new collision free path prior to the expected collision. This is the problem of visual guidance.

The visual guidance can be achieved mainly by two approaches. The first approach is to process the top view again after every step of the robot, to find the new collision free path. The previously discussed collision free path finding algorithm is used for each picture frame. The second approach is to find the expected collision points, and process the top view again only prior to the expected collision. In the second approach considerable amount of processing time can be saved. This is because in this approach the top view is processed less frequently as compared to the first approach. As the critical problem for the robot visual guidance in dynamic environment is high speed on line information processing, we have decided to choose second approach.
In this research we have solved the equations of motion and equation of robot simultaneously to find the points of collision. Zhu[25] has also used velocity information and equation of motion of objects to find the collision points, but he has not found the equation of collision free path. Instead of that he has used information from structural pyramid to check the collision of the moving object with the robot. To find points of collision and time instances of collision, we need to know the equation of collision free path, the equations of motion of moving objects and their respective velocities.

5.1 Equation for Collision Free Path

The collision free path planning algorithm gives the critical points forming the path. Here the critical points are assumed to be the centroids of the blocks forming the collision free path. We are interested in finding the equation of the path consisting of these points. This is a problem of polynomial interpolation which may be stated as follows. Let us assume that we have a set of N points \((x_1, y_1), (x_2, y_2), \ldots , (x_n, y_n)\) and we want to find a polynomial \(p(x)\) satisfying all N points simultaneously. It can be proved [32] that the polynomial of degree \(N-1\) is completely determined by \(N\) points. This polynomial of degree \(N-1\) as the equation of collision free path must satisfy the following conditions.
The solution to the interpolation problem is given by Lagrange's interpolation formula as shown below.

\[ p(x) = \sum_{j=0}^{N} y_j \frac{x - x_i}{x_j - x_i} \] ...

...5.1

It is clear that the degree of the polynomial increases linearly with the number of points \( N \). The numerical example of Lagrange's polynomial is discussed below:

Let \( N=3 \), \( (x_1,y_1) = (1,3); (x_2,y_2) = (2,7); (x_3,y_3) = (3,13) \)

\[ p(x) = \frac{x-2}{1-2} + \frac{x-3}{1-3} + \frac{x-1}{2-1} + \frac{x-3}{2-3} + \frac{x-1}{3-1} + \frac{x-2}{3-2} \]

\[ p(x) = x^2 + x + 1 \]

5.2 Equations for Moving Objects

The above discussed Lagrange's interpolation formula can also be used to find the equations of motion for the moving objects. The critical points satisfying the motion are used for this purpose. This will give nonlinear equations of motion for the moving objects of the form

\[ y = ax^n + bx^{n-1} + \ldots + d \] ...

...5.2

In our particular case the two successive picture frames are processed to find the equation of moving objects.
The position of the centroid of the object in frame 1 and the position of the centroid of the object in frame 2 are used to find the equation of motion of the moving object. This will give the linear equation of motion for the moving object of the form

\[ y = mx + c \quad \ldots 5.3 \]

### 5.3 Algorithm to Find Collision Points

To find the collision points and the time instances at which collision will take place, velocities of robot and moving objects in 2-D picture frame should be known. In 2-D plane velocity can be given by a vector

\[ v(t) = (v_x(t), v_y(t)) \quad \ldots 5.4 \]

\[ v(t) = \frac{dS(x,y,t)}{dt} \quad \ldots 5.5 \]

where \( S(x,y,t) = (x(t), y(t)) \)

The direction of \( v(t) \) is given by

\[ \theta_v = \tan^{-1}[\frac{v_y(t)}{v_x(t)}] \quad \ldots 5.6 \]

and the magnitude of velocity \( v(t) \) is

\[ V(t) = \sqrt{v_x(t)^2 + v_y(t)^2} \quad \ldots 5.7 \]

In discrete case, the above quantities are calculated by using two successive frames of the image.
where \( (x(t), y(t)) \) and \( (x(t-1), y(t-1)) \) are two corresponding points at two time instances \( t \) and \( t-1 \) respectively and \( \Delta t \) is difference between two time instances.

In this research we have used positions of centroid of the object in two successive picture frames to find its velocity. Let \( (x_1, y_1) \) and \( (x_2, y_2) \) be the positions of centroids of the object in two successive picture frames. Then substituting \( x(t) = x_2 \) and \( x(t-1) = x_1; y(t) = y_2 \) and \( y(t-1) = y_1 \) in the Equation 5.8 and Equation 5.9 we can get \( v_x(t) \) and \( v_y(t) \) for the object. Similarly, the centroid of the robot is used to find robots velocity.

To find if the object will collide with the robot, first the equation of the collision free path and the equation of the moving object are solved simultaneously without considering the velocity. Let the point of intersection be \( (x(t), y(t)) \) as shown in Figure 5.1. Now using this information and velocity information about the robot and the object, the time taken by the robot to reach at collision point \( t_{rob} \) and the time taken by the object to reach at the collision point \( t_{obj} \) are calculated using the following:

\[
\begin{align*}
v_x(t) &= \frac{x(t) - x(t-1)}{\Delta t} \quad \ldots 5.8 \\
v_y(t) &= \frac{y(t) - y(t-1)}{\Delta t} \quad \ldots 5.9
\end{align*}
\]
Figure 5.1
equations:

\[ v(t)^2 = v_x(t)^2 + v_y(t)^2 \quad \ldots 5.10 \]

\[ v(t)^2 = \frac{[x(t) - x(t-1)]^2}{(\Delta t)^2} + \frac{[y(t) - y(t-1)]^2}{(\Delta t)^2} \quad \ldots 5.11 \]

\[ (\Delta t)^2 = \frac{[x(t) - x(t-1)]^2 + [y(t) - y(t-1)]^2}{v(t)^2} \quad \ldots 5.12 \]

\[ t_{rob} = \frac{x(t) - x_{rob}(t-1)}{v_{xrob}(t)} \quad \ldots 5.13 \]

\[ t_{obj} = \frac{x(t) - x_{obj}(t-1)}{v_{xobj}(t)} \quad \ldots 5.14 \]

where

\[ x_{rob}(t-1) \] is x coordinate of robot in previous picture frame
\[ x_{obj}(t-1) \] is x coordinate of object in previous picture frame

If \( t_{rob} = t_{obj} \) then the collision between the robot and the moving object will take place otherwise there will be no collision. This is because \( t_{rob} = t_{obj} \) suggest that both the robot and the object will be at the collision point at the same time instance to make a collision.
If the algorithm found that there will be a collision between robot and the moving object at a particular time instant then the top view of the environment is again processed to find the new collision free path prior to that time instant.

5.4 Implementation of the Algorithm to Find Collision Point

The above discussed algorithm is implemented on the AMETEK S-14 parallel processor because of its problem regularity. To find the collision points with the moving objects and the robot the equations of motion of the moving objects have to be solved simultaneously with the equation of the robot path from \(-\infty\) to \(+\infty\). Generally the collision free robot path equation is of the order of 15 to 20 because of its complex nature. To solve the equation of degree 20 simultaneously with other equations in the range of \(-\infty\) to \(+\infty\) on a single processor is very slow and time consuming in real time situations. So some other approach has to be used.

In this research our image frames are bound from \(x = 1\) to 512 and \(y = 1\) to 512 so we are not interested from \(-\infty\) to \(+\infty\) range. We are only interested in the range from 1 to 512 in both \(x\) and \(y\) directions.

As it is possible to solve equations in different boundary limits simultaneously and separately without any sequential dependency, this algorithm is well suited for
parallel processing. In the case of 16-node hypercube processor and the range \( x = 1 \) to 512 and \( y = 1 \) to 512 each processor can work concurrently in the range \( x = 1 \) to 512 and \( y = 1 \) to 32, \( y = 33 \) to 64 ... upto \( y = 480 \) to 512 as shown in the Figure 5.5. Each child has the equation of collision free robot path and equations of motions of all the moving objects. Each child solves these equations for finding the collision points at the same time in the above specified boundaries. Finally all the found collision points from each child is downloaded to the host processor. This algorithm is general enough to handle all possible situations. For example, the situations like one object colliding more than one times with the robot in the boundaries of the single child or one object colliding more than one times with the robot in the boundary conditions of different processors as shown in Figure 5.2, 5.3 and both object and robot having nonlinear motion and colliding with each other as shown in Figure 5.4 can be handled without any problem.

After applying the above discussed algorithm to the case shown in Figure 5.5 the 0th, 1st, 2nd and 3rd processors found that there is no collision in their respective boundary limits. Processor 4 found that there is one collision with object \( O_1 \). Processor 5 found that there are two collisions with object \( O_2 \) in its boundary limits. Similarly processors 8 and 11 found one collision each in their respective
Figure 5.2
EQUATIONS OF PATH & MOVING OBJECTS

Figure 5.3

Row Values

Column Values

1

2

3
EQUATIONS OF PATH & MOVING OBJECTS

Figure 5.4

ROW VALUES

COLUMN VALUES

1
2
Figure 5.5 Example for visual guidance algorithm
boundaries. These processors download the co-ordinates of these points of collision and the rest of the processors do not download any information.
CHAPTER 6

IMPLEMENTATION AND EXPERIMENTAL RESULTS

The high level vision algorithms presented in this research have been implemented on a DEC VAX 11/750 mini-computer. Parallel implementation of the low level vision algorithms for preprocessing the raw image has been realized on the AMETEK S-14, a 16 node hypercube system [28] as described in the previous chapter. S-14 is a loosely coupled message passing architecture and incorporates a minimum of 16 and maximum of 256 processors at a peak performance of 15 Mflops. Each processor has 1 Mbytes of local memory. Image acquisition, digitization, and display have been performed through the VICOM image processor, which is interfaced with an RCA camera. The VAX 11/750 hosts the operation of these two systems under the Unix 4.3 BSD operating system. The overall experimental setup is shown in the block diagram in Figure 6.1.

A concurrent program, which consists of a parent and a child program, was written for ring topology. These programs were developed using the user topology interface routines. All aspects of the processor coordinations and communications were programmed. The parallel program is designed based on the network structure and coded in the C
Figure 6.1 Experimental set-up
programming language with the aid of the ADE (Ametek Development Environment) software tools.

6.1. Description of the Path Planning Program

The programs for implementing the previously discussed algorithms are written in C. In terms of C language quadtree is defined as a collection of structures. A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Structure is an advanced type of link list. To build a quadtree we have used a structure of the form shown bellow:

```
struct TREE
{
    struct TREE *root;
    struct TREE *nw;
    struct TREE *ne;
    struct TREE *sw;
    struct TREE *se;
    int nodetype;
    int row;
    int column;
    int flag;
}
```

Here the definition of the structure itself is recursive. A structure is defined within a structure. The
first five fields of the structure are the pointers connecting all structures together to form a quadtree. The root link connects the structure in question to its father and forms four children or four links (NW, NE, SW, SE) so that quadtree can grow in a well regulated manner. The last four fields are the information fields and they hold the information which is necessary for manipulation and traversal of the quadtree.

The different functions that have been developed in this research to build a quadtree are as follows:

1. Initialize_a_tree
2. Calculate
3. Add_to_tree
4. Merge_a_tree
5. Print_a_tree

A short description of these functions is given below.

Function "Initialize_a_tree":

We have used dynamic memory allocation to utilize the available memory efficiently. The memory is allocated to the particular node in a quadtree only when that node is actually formed. When the function initialize-a-tree is called in the program, it creates that node and allocates the memory for that node. It also initializes all the fields of the newly formed structure node to the values passed by the calling program. This function finally returns with the
initialized structure; i.e., newtree to the calling function.

Function "Calculate":

This function finds the information about the father and the position of the current node with respect to the tree. This function requires level of the node and the number of the node as input. The parameters found using this function are used by add_to_tree function.

Function "Add_to_tree":

This function properly links the newly initialized structure to build a tree. It uses the information calculated by the function calculate about the father and position of the node. Using this information it links the structure to the partial quadtree to build the final quadtree. The function adds one node at a time in the quadtree. It returns a partially built quadtree to the calling program for further calculations. It uses the same partially built quadtree in the next iteration to add a new node.

Function "Merge_a_tree":

This function is called at the end after whole quadtree is build. It takes the unmerged quadtree as an input and returns the merged quadtree as an output. It starts from the lowest level of the quadtree. It checks the node type fields of the four children of each father. If all the four
children are of the same type; i.e., all of them are type 0 or type 1, then this function merges them and only father node is left. It repeats this procedure for all levels to produce the final merged tree.

Function "Print_a_tree":

This function is used to print the whole merged quadtree for the illustration purposes. It uses the merged quadtree as its input and gives a file which contains the information about the positions of the nodes in the quadtree. To print the quadtree, the merged quadtree is traversed in the depth first manner.

Function "Path_finding":

This function takes the start point, the end point and the merged quadtree as an input and it gives the intermediate shortest obstacle free path as an output. For efficient implementation of this algorithm, a set of stack is used to store the necessary information. For this purpose, functions push, pop, set_stack, empty_stack are defined in their conventional way.

Using the start and the end point of the robot's motion path, the corresponding leaf nodes are found in the merged quadtree. Then using the start node of the tree, its four neighbors in all four directions in the spatial plane are determined using the function "Find_neighbor". The
neighbors found have size equal to or greater than that of the robot. If a neighbor with these characteristics is not found then it is decided that there is no path available in that direction. All the nonobstacle neighbors are then pushed on the stack named psn. The neighbor node in the direction of the end point is popped from the psn stack and is checked for terminating the process. If the end condition is not satisfied and the psn stack become empty, then it is concluded that there is no free path available and the robot is trapped. If the end condition is achieved, the corresponding path is accepted as the first available path. If the end condition is not satisfied, that node is pushed on the stack psp[pathno] where initially the pathno is 1 and its value is incremented as the number of available paths increases. The best path is printed by poping its stack.

6.2. Experimental Results

The presented method is tested in a time-varying environment. An experiment is performed on a sequence of two images of the top view of the time-varying scene. For the top image the plane of the ground is not exactly perpendicular to the optical axis of the camera, rather there is about 20 degree angle between the image plane and the camera optical axis. The input images have gray level values from 0 to 255. Various complex shaped objects have been used to create a real
world situation.

Figure 6.2(a) shows the 512x512 top view image. It contains five obstacles, a background and a path. This is the original unpreprocessed image. Figure 6.2(b) shows the preprocessed (i.e., segmented and noise cleaned) image in which the background and objects have given different gray level values. This image is used as an input for the high level vision algorithms. The collision free path found in this image is shown in Figure 6.2(c). The end point of the path is shown with zero gray level value and each block forming a path is depicted with different gray level values with difference of 7.

The results of visual guidance algorithm is shown in Figure 6.3. The equation of path shown in Figure 6.2(c) is found using Lagrange's interpolation formula and the critical points given in Table 6.1(a). The resultant equation is plotted as shown in the Figure 6.3. Similarly the equation of motion of the object is found using the critical points given in Table 6.1(b). The point of collision is found as ( , ). As there is an expected collision, the top view of the dynamic scene is processed again.

The same experiment is repeated for the new scene in which center object is moved. The Figure 6.4(a) shows the new scene. The preprocessed image of the new scene is given in Figure 6.4(b). Since the previously found path is blocked
Figure 6.2 Results of path planning in the top view of an experimental scene: a(top), b(center), c(bottom).
Table 6.1 Row and column numbers of the critical points
(a) for robot motion  (b) for object motion

<table>
<thead>
<tr>
<th>COL #</th>
<th>ROW #</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>256</td>
</tr>
<tr>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>160</td>
<td>256</td>
</tr>
<tr>
<td>192</td>
<td>256</td>
</tr>
<tr>
<td>192</td>
<td>288</td>
</tr>
<tr>
<td>192</td>
<td>320</td>
</tr>
<tr>
<td>192</td>
<td>352</td>
</tr>
<tr>
<td>224</td>
<td>352</td>
</tr>
<tr>
<td>256</td>
<td>352</td>
</tr>
<tr>
<td>288</td>
<td>352</td>
</tr>
<tr>
<td>320</td>
<td>352</td>
</tr>
<tr>
<td>352</td>
<td>352</td>
</tr>
<tr>
<td>352</td>
<td>352</td>
</tr>
<tr>
<td>384</td>
<td>352</td>
</tr>
<tr>
<td>384</td>
<td>320</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COL #</th>
<th>ROW #</th>
</tr>
</thead>
<tbody>
<tr>
<td>224</td>
<td>382</td>
</tr>
<tr>
<td>256</td>
<td>278</td>
</tr>
</tbody>
</table>

(b)
Figure 6.3
as a result of this motion, the new path planning is performed using the extracted motion information from the image sequence. The resultant new path is shown in Figure 6.4(c).
Figure 6.4 Results of path planning for the scene of Figure 6.2 after the center object is moved a(top), b(center), c(bottom).
CHAPTER 7

CONCLUSIONS AND DISCUSSION

In this research we have presented a time efficient algorithm for robot path planning in dynamic environment using quadtree data structure and parallel processing. Initially the raw image is preprocessed using histogram-based segmentation, noise cleaning, and low level motion analysis algorithms. The preprocessed image is used to build a merged quadtree, which is then searched to find the neighbor information. This neighbor information is used to find obstacle free path between the start and the end point. As long term conventional path planning does not involve any motion information and path verification, they cannot be used directly in dynamic environment. In this research the results of motion analysis are used for selection of the best path. Because of the use of the motion analysis results for the path selection, the probability of blocking the path by the obstacle in dynamic environment is prevented.

Visual guidance is achieved by solving the equation of the path and the equations of the motion of the obstacles. The points of expected collision and the time instances of the collision are found using velocity information. The top view of the image is processed again only at the time instance
prior to the expected collision.

Image processing and visual guidance are performed on a 16-node attached hypercube array processor by exploiting the inherent parallelism, problem regularity, and data locality of the low-level vision operators in a well structured problem domain. This improves the overall computation efficiency of the algorithm. Passing the recursive structure from processor-to-processor and processor-to-host is the major problem encountered in constructing the quadtree in the hypercube attached parallel processor [31]. This remains as our further research effort.
REFERENCES


14. H.Samet, "An algorithm for converting rasters to quad

15. H.Samet, "Region representation : Quadtrees from binary
pp. 88-93.

16. H.Samet and C.A.Shaffer, "Optimal quadtree construction
algorithm," Computer Graphics and Image Processing, vol.37,
1987, pp. 402-419.

17. H.Samet, "The quadtree and related hierarchical data
187-260.

18. H.Samet, "Neighbor finding techniques for image
represented by quadtrees," Computer Graphics and Image

19. G.M.Hunter and K.Steiglitz, "Operations on images using


25. Q. Zhu, "Structural pyramids for representing and locating


The order of the program listings is as follows:

1. Sequential preprocessing program
2. Concurrent preprocessing program in ring topology
3. High level vision algorithms program
4. Visual Guidance Program
Program : Preprocessing program using sequential algorithms

Developed By : ANIL R. BOHORA

#include <stdio.h>
unsigned char data[512][512],binary[512][512];
int y[500];
FILE *fp;

main()
{
    int i,j,k,l,m,x;
    int min,max;
    int th[10],objects;
    int istart,istop,jstart,jstop,mstart,mstop;
    int sum[100],area[10],xbar[10],ybar[10];
    int countx[10],county[10];
    int neighbour[20][20][20],found;
    char infile[25],outfile[25],histfile[25],resultfile[25];

    printf("Enter the input image filename : ");
    scanf("%s",infile);
    printf("Enter the output image filename : ");
    scanf("%s",outfile);
printf("Enter the histogram filename : ");
scanf("%s",histfile);

printf("Enter the result filename : ");
scanf("%s",resultfile);

/*-----------------------------------------------------
 *         Copying file into an array named data
 *----------------------------------------------------*/
fp=fopen(infile,"r");
fread(data,512*512,1,fp);
fclose(fp);
printf("\narray scanned \n\n");
/*-----------------------------------------------------*/
/* Finding minimum and maximum grey level values    */
/*-----------------------------------------------------*/
min=0;
max=0;
for(i=0;i<=511;i++)
{
    for(j=0;j<=511;j++)
    {
        if(data[i][j]<min)
            min=data[i][j];
        if(data[i][j]>max)
            max=data[i][j];
    }
printf("%s\n", "\n"");

/*-------------------------------------------------*/
/* Counting the pixels for histogram */
/* and printing counted values */
/*-------------------------------------------------*/

for(i=0;i<=max;i++)
    y[i]=0;

for(i=0;i<=max;i++)
{
    for(j=0;j<=max;j++)
    {
        k=data[i][j];
        y[k]++;
    }
}

fp=fopen(histfile,"w");
for(i=0;i<=max;i++)
{
    fprintf(fp,"y[%2d]= %9d",i,y[i]);
    fprintf(fp,"\n");
}

fclose(fp);

printf("\t Max. number of objects allowed=10\n\n");
printf("\t Enter the number of objects = ");
scanf("%d",&objects);
printf("\t Enter the threshold values \n");

for(i=1;i<=objects;i++)
{
    printf("th[%d]=",i);
    scanf("%d",&th[i]);
}
/*-----------------------------------------------------*/
/* Finding binary image                           */
/*-----------------------------------------------------*/
for(i=0;i<=511;i++)
{
    for(j=0;j<=511;j++)
    {
        for(m=1;m<=objects;m++)
        {
            if(data[i][j]<=th[m]&&data[i][j]>th[m-1])

                binary[i][j]=(20*m);
            else
                continue;
        }
    }
}
printf("binary image created\n");

/*----------------------------------------------------
/*       Cleaning noise       */
/*----------------------------------------------------*/
for(i=0;i<=99;i++)
    sum[i]=0;
x=0;
istart=0;
istop=2;
for(k=0;k<=509;k++)
{
    jstart=0;
jstop=2;
for(l=0;l<=509;l++)
{
    x=binary[istart][jstart];
    for(i=istart;i<=istop;i++)
    {
        for(j=jstart;j<=jstop;j++)
        {
            if(binary[i][j]==x)
                sum[x]=sum[x]+1;
        }
    }
}
if(sum[x]==8)
    for(i=istart;i<=istop;i++)
    {
        for(j=jstart;j<=jstop;j++)
        {
            if(binary[i][j]!=x)
                binary[i][j]=x;
        }
    }
if(sum[x]!=8)
    sum[x]=0;
    jstart=jstart+1;
    jstop=jstop+1;
}
istart=istart+1;
istop=istop+1;

printf("\tnoise cleaned\n");

/*-----------------------------------------------------
*/
/* Storing binary image into an file */
/*-----------------------------------------------------*/
fp=fopen(outfile,"w");
fwrite(binary,512*512,1,fp);
fclose(fp);
printf("\tfile copyed\n");
for(i=1;i<=objects;i++)
{
    area[i]=0;
    xbar[i]=0;
    ybar[i]=0;
    countx[i]=0;
    county[i]=0;
}
for(i=0;i<=511;i++)
{
    for(j=0;j<=511;j++)
    {
        for(m=1;m<=objects;m++)
        {
            if((20*m)==binary[i][j])
            {
                countx[m]=countx[m]+i;
                county[m]=county[m]+j;
                area[m]=area[m]+1;
            }
        }
    }
}
for(m=1;m<=objects;m++)
{
    xbar[m]=countx[m]/area[m];
    ybar[m]=county[m]/area[m];
    printf("(x[%d],y[%d])=(%d,%d)\n",m,m,xbar[m],ybar[m]);
    printf("area[%d]=%5d\n",m,area[m]);
}
/*------------------------------------------------------*/
/* Finding neighbours using partitioning of the image */
/*------------------------------------------------------*/

for(k=0;k<=19;k++)
{
    for(l=0;l<=19;l++)
    {
        for(m=0;m<=19;m++)
        {
            neighbour[k][l][m]=999;
        }
    }
}
x=0;
mstart=0;
mstop=0;
found=0;
istart=0;
istop=31;
fp=fopen(resultfile,"w");
for(k=0;k<=15;k++)
{
    jstart=0;
    jstop=31;
    for(l=0;l<=15;l++)
    {
        for(i=istart;i<=istop;i++)
        {
            for(j=jstart;j<=jstop;j++)
            {
                for(m=mstart;m<=mstop;m++)
                {
                    if(neighbour[k][l][m]==binary[i][j])
                    {
                        found=1000;
                    }
                    else continue;
                }
                if(found!=1000)
                {
                    x=x+l;
                    neighbour[k][l][x]=binary[i][j];
                    fprintf(fp,"neighbour[%d][%d][%d]=%d \n",k,l,x,neighbour[k][l][x]);
                }
            }
        }
    }
}
mstop=x;
}
else continue;
}
}
jstart=jstart+32;
jstop=jstop=32;
x=x+1;
mstart=x;
mstop=x;
found=0;
}

istart=istart+32;
istop=istop+32;
}
close(fp);

/*-------------------------------------------------*/
/* Putting result in a file */
/*-------------------------------------------------*/
fp=fopen(resultfile,"a");
for(m=1;m<=objects;m++)
{
    fprintf(fp,"area[%d]=%d\n",m,area[m]);
    fprintf(fp,"x[%d]=%d\ny[%d]=%d\n",m,xbar[m],m,
            ybar[m]);
fclose(fp);
Program: Preprocessing program for parent using RING topology

Developed by: ANIL R. BOHORA

#include <stdio.h>
#include <math.h>
#include "ringp.h"

static unsigned char image[512][512];
static unsigned char clean[512][512];

int objects, found;
int hist[256], th[10];
int area[10], xbar[10], ybar[10], countx[10], county[10];
int sum[100], neighbour[20][20][20];
int i, j, k, l, m, x;
int istart, istop, jstart, jstop, mstart, mstop;
FILE *fp, *fp1, *fp2, *fp3;
/* Query for input, output, histogram and result file names */

printf("Enter input image filename : ");
scanf("%s",infile);

printf("Enter the output image filename : ");
scanf("%s",outfile);

printf("Enter the histogram filename : ");
scanf("%s",histfile);

printf("Enter the result filename : ");
scanf("%s",resultfile);

/* Opening input file for reading into in image array */
if((fp1=fopen(infile,"r")) == NULL)
{
    printf(stderr,"Error opening %s for input \n",infile);
    exit(1);
}

fread(image,512*512,1,fp1);
fclose(fp1);

/* Ring topology initialization */
ring(SAME,"preproc.exe");

/*-----------------------------------------------
 * Downloading subimages
 -----------------------------------------------*/
dnldac(image, 32*512);

/*-----------------------------------------------
 * Receive accumulated histogram from corner node
 -----------------------------------------------*/
c_rcvai(hist,256);
if((fp2=fopen(histfile,"w"))!=NULL)
{
    printf(stderr,"Error opening historing for output \n");
    exit(1);
}
for(i=0;i<256;++i)
    fprintf(fp2,"y[%d] = %d \n",i,hist[i]);
fclose(fp2);

/*-----------------------------------------------
 * Read and broadcast thresholdvalues based on the
 * returned histogram
 -----------------------------------------------*/
printf("Enter the number of objects : ");
scanf("%d",&objects);
printf("Enter the threshold values : \n ");
for(i=1;i<=objects;i++)
{
    printf("th[%d]= ",i);
    scanf("%d",&th[i]);
}
sndi(objects,ALL);
sndai(&th[1],objects,ALL);
/*----------------------------------------------------
* Receive the preprocessed image from the children
-----------------------------------------------------*/
upldac(clean,32*512);
if((fp3=fopen(outfile,"w"))==NULL)
{
    printf(stderr,"Error opening bimagel for outputput \n");
    exit(1);
}
fwrite(clean,512*512,1,fp3);
fclose(fp3);
/*----------------------------------------------------
* Finding area & centroid of the objects
-----------------------------------------------------*/
for(i=1;i<=objects;i++)
{
    area[i]= xbar[i]= ybar[i]= 0;
    countx[i]= county[i]= 0;
for(i=0;i<=511;i++)
{
    for(j=0;j<=511;j++)
    {
        for(m=1;m<=objects;m++)
        {
            if((20*m)==clean[i][j])
            {
                countx[m]=countx[m]+i;
                county[m]=county[m]+j;
                area[m]=area[m]+1;
            }
        }
    }
}

for(m=1;m<=objects;m++)
{
    xbar[m]=countx[m]/area[m];
    ybar[m]=county[m]/area[m];
}

/*---------------------------------------------------
* Finding neighbours using partitioning of the image
*----------------------------------------------------*/
for(k=0;k<=19;k++)
{

for(l=0;l<=19;l++)
{
    for(m=0;m<=19;m++)
    {
        neighbour[k][l][m]=999;
    }
}

x=0;
mstart=0;
mstop=0;
found=0;
istart=0;
istop=31;
fp=fopen(resultfile,"w");
for(k=0;k<=15;k++)
{
    jstart=0;
    jstop=31;
    for(l=0;l<=15;l++)
    {
        for(i=istart;i<=istop;i++)
        {
            for(j=jstart;j<=jstop;j++)
            {
                for(m=mstart;m<=mstop;m++)
if (neighbour[k][l][m] == clean[i][j])
{
  found = 1000;
}
else continue;

if (found != 1000)
{
  x = x + 1;
  neighbour[k][l][x] = clean[i][j];
  fprintf(fp, "neighbour[%d][%d][%d] = %d\n", k, l, x, neighbour[k][l][x]);
  mstop = x;
}
else continue;

jstart = jstart + 32;
jstop = jstop = 32;
x = x + 1;
mstart = x;
mstop = x;
found = 0;
istart=istart+32;
istop=istop+32;
}
fclose(fp);

*/---------------------------------------------------------------------------------------------------
* Putting result in a file
---------------------------------------------------------------------------------------------------*/

fp=fopen(resultfile,"a");
for(m=1;m<=objects;m++)
{
    fprintf(fp,"area[%d]=%d\n",m,area[m]);
    fprintf(fp,"x[%d]=%d\ny[%d]=%d\n",m,xbar[m],m,ybar[m]);
}
fclose(fp);
Program:- Preprocessing child program
for RING topology

Developed by:- ANIL R. BOHORA

#include "ringc.h"

static char pro[32][512], bin[35][512];

user_main()
{
    register int i, j, k, l, m, count;
    int th[10], objects;
    int hist[256], hist1[256], histr[256], sum[256];
    int x, istart, istop, jstart, jstop;

    dnldac(pro, 32*512);
    for (j=0; j<256; ++j)
        hist[j] = 0;

    for (i=0; i<32; ++i)
        for (j=0; j<512; ++j)
            hist[pro[i][j]] = ++hist[pro[i][j]];
for(count=0;count<8;++count)
{
    if(childnum==0)
    {
        rcvai(histl,256,LEFT);
        rcvai(histr,256,RIGHT);
        if(count<7)
        {
            for(i=0;i<256;++i)
                hist[i]=hist[i]+histl[i]+histr[i];
        }
    }
    else
    {
        for(i=0;i<256;++i)
            hist[i]=hist[i]+histl[i];
    }
}
if(childnum<8 && childnum!=0)
{
    sndai(hist,256,LEFT);
    rcvai(hist,256,RIGHT);
}
if(childnum > 8)
{
    sndai(hist,256,RIGHT);
rcvai(hist, 256, LEFT);
}
if(childnum==8)
{
    sndai(hist, 256, RIGHT);
    sndai(hist, 256, LEFT);
}

/*------------------------------------------------------------------------------
 * Send histogram to host from corner node
 *-----------------------------------------------------------------------------*/
c_sndai(hist, 256);

/*------------------------------------------------------------------------------
 * Receive no. of objects & threshold values from parent
 *-----------------------------------------------------------------------------*/
rcvi(objects, PAR);
rcvai(&th[1], objects, PAR);

/*------------------------------------------------------------------------------
 * Thresholding
 *-----------------------------------------------------------------------------*/
for(i=0; i<32; ++i)
{
    for(j=0; j<512; ++j)
    {
        for(m=1; m<=objects; m++)
        {

}}
if(pro[i][j]<=th[m] && pro[i][j] > th[m-1])
    bin[i][j]=(20*m);
else
    continue;

/*---------------------------
   * Noise cleaning

-----------------------------------*/
for(i=0;i<=99;i++)
    sum[i]=0;
x=0;
istart=0;
istop=2;
for(k=0;k<32;++k)
{
    jstart=0;
    jstop=2;
    for(l=0;l<510;++l)
    {
        x=bin[istart][jstart];
        for(i=istart;i<=istop;i++)
        {
            for(j=jstart;j<=jstop;j++)
            {
if(bin[i][j]==x)
    sum[x]=sum[x]+1;

}

if(sum[x]==8)
    for(i=istart;i<=istop;i++)
    {
        for(j=jstart;j<=jstop;j++)
        {
            if(bin[i][j]!=x)
                bin[i][j]=x;
        }
    }

    if(sum[x]!=8)
        sum[x]=0;
    jstart=jstart+1;
    jstop=jstop+1;

}

istart=istart+1;
istop=istop+1;

/
* Upload preprocessed image to host
* ---------------*/
upldac(bin,32*512);
/*-----------------------------------------------
* Program :- Program for building a merged quadtree
* from the preprocessed image and searching
* it for path planning.
*
* Developed By :- ANIL R. BOHORA
*-----------------------------------------------*/

#include <stdio.h>
#include <math.h>
#define slimit 1000
#define true 1
#define false 0

/* declaring the structure */
struct TREE
{
    struct TREE *root;
    struct TREE *nw;
    struct TREE *ne;
    struct TREE *sw;
    struct TREE *se;
    int nodetype;
    int flag;
    int row;
    int col;
};
struct stack
{
    struct TREE *stackarray[slimit];
    int top;
};

struct TREE *root,*t,*tras,*rt;
struct TREE *pvalue,*value,*qp;

FILE *fp,*fd;

double sqrt();
double pow();
double size,nesize,onesize;
double four,two,qx,bx,pp;

char path[128];
char d;

unsigned char data[512][512],binary[512][512];
int *rstrt,*cstrt;
int *rstrtl,*cstrtl1;
int *rrow[9],*ccol[9];

int type,length,rown0,colno;
int father[50], fa[50], flevel, level, x, iposition;
int i, j, jj, ii, iii, lf, kf, ff, y, yy;
int pint;
int ir, is, ia;
int pathno, joy;
int trapped, freepath;
int n, n1, newn, newn1, q, qn, qne, llevel;
int fx;

/*---------------------------------------------------------
    *                Initializing a tree
    *
    *---------------------------------------------------------*/

struct TREE *initialize(root)
struct    TREE *root;
{
    struct TREE *newtree;

    newtree=(struct TREE *)malloc(sizeof(struct TREE));

    newtree->row=rowno;
    newtree->col=colno;
    newtree->nodetype=type;
    newtree->flag=length;
    newtree->root=root;
newtree->nw=NULL;
newtree->ne=NULL;
newtree->sw=NULL;
newtree->se=NULL;

if (newtree->root == NULL)
    newtree->root = newtree;
return (newtree);
}

/*************************************************************/
/*
*    Adding an element to a tree
*
*    i/p = father[i], *t, iposition, type, length, level
*    o/p = void
*
*************************************************************/

add_to_tree(t, level)

struct TREE *t;
int level;
{
    struct TREE *position;
t=t;

for(i=1; i<level; i++)
{
    fa[i] = father[i]%4;
    if(fa[i] == 1) t = t->nw;
    else if(fa[i] == 2) t = t->ne;
    else if(fa[i] == 3) t = t->sw;
    else if(fa[i] == 0) t = t->se;
}

switch(iposition)
{
    case 1:
        position = t->nw;
        break;
    case 2:
        position = t->ne;
        break;
    case 3:
        position = t->sw;
        break;
    case 4:
        position = t->se;
        break;
}
if (position != NULL)
    add_to_tree(position, level);
else
{
    if (iposition == 1)
    {
        t->nw = initialize(t);
        position = t->nw;
    }
    else if (iposition == 2)
    {
        t->ne = initialize(t);
        position = t->ne;
    }
    else if (iposition == 3)
    {
        t->sw = initialize(t);
        position = t->sw;
    }
    else if (iposition == 4)
    {
        t->se = initialize(t);
        position = t->se;
    }
}
Calculates the parameters required by function add_to_tree
i/p = x, level
o/p = iposition, father[i]

calculate()
{
    flevel = level - 1;
    father[flevel + 1] = x;
    if (flevel != 0)
    {
        for (i=flevel; i>0; i--)
        {
            if((father[i +1] %4)!=0)
            {
                father[i] = father[i +1] /4 + 1;
            }
            else
            {
                if(father[i +1] >4)
                    father[i]=father[i+1]/4;
                else
                    father[i] = 1;
            }...
        }...
    }...
}...
if (level != 0) {
    iposition = x%4;
    if (iposition == 0)
        iposition = 4;
}

/*----------------------------------------------------------
*             Function to print the tree elements
*----------------------------------------------------------*/
print_tree(tree, path)
struct TREE *tree;
char *path;
{
    char node[5];

    if (tree == NULL) {
        return;
    }
else if (tree == tree->root->nw) {
    strcpy(node,".nw");
}
else if (tree == tree->root->ne) {
    strcpy(node,".ne");
}
else if (tree == tree->root->sw) {
    strcpy(node,".sw");
}
else if (tree == tree->root->se) {
    strcpy(node,".se");
}
else if (tree == tree->root) {
    if (strlen(path) == 0) strcpy(path,"root");
    strcpy(node,"");
}
else {
    printf("\007\007We have a bad tree!!!\n");
    printf("We are below %s\n",path);
    return;
}
strcat(path,node);

fprintf(fp,"%50s",path);
fprintf(fp," (%2d),%2d,r=%d,c=%d\n",tree->nodetype,
        tree->flag, tree->row, tree->col);
print_tree(tree->nw,path);
print_tree(tree->ne,path);
print_tree(tree->sw,path);
print_tree(tree->se,path);

while(path[strlen(path)-1] != '.' && strlen(path) != 0) {
    path[strlen(path) - 1] = '\0';
}
if (strlen(path) != 0 )
    path[strlen(path) - 1] = '\0';
return;
}

/*-----------------------------------------------------------
Function for merging the quadtree
i/p =*t
result = *t
----------------------------------------------------------*/

merge()
{
    int one, zero, lm, ml, temp, ifo, ikk;
    double four, fo, xle;
int check[200];

for(ml=7; ml>=1; ml--)
{
    x=0;
    level=ml;
    four=4;
    xle=level-1;
    fo=pow(four, xle);
    ifo=(int) fo;
    for(j=1; j<=ifo; j++)
    {
        for(ii=0; ii<=100; ii=ii+20)
            check[ii]=0;
        zero=0;
        one=0;
        for(ikk=1; ikk<=4; ikk++)
        {
            x=x+1;
            calculate();
            t=root;
            for(lm=1; lm<level; lm++)
            {
                fa[lm]=father[lm] % 4;
                if(fa[lm]==1)t=t->nw;
                else if(fa[lm]==2)t=t->ne;
            }
        }
    }
}
else if(fa[lm]==3)t=t->sw;
else if(fa[lm]==0)t=t->se;
}

switch(iposition)
{
    case 1:
        t = t->nw;
        break;
    case 2:
        t = t->ne;
        break;
    case 3:
        t = t->sw;
        break;
    case 4:
        t = t->se;
        break;
}

    temp=t->nodetype;
    if(temp==80)
        zero=zero+l;
    else if(temp!=2)
        check[temp]=check[temp]+1;
else
    t=t->root;
}
for(ii=0;ii<=100;ii=ii+20)
{
    if(check[ii]==4)
    {
        one=4;
        temp=ii;
    }
}
if((one==4)||(ml>4)&&(zero==4))
{
    t=t->root;
    t->nodetype=temp;
    t->nw=NULL;
    t->ne=NULL;
    t->sw=NULL;
    t->se=NULL;
}
else
    t=t->root;
}
}
/*----------------------------------------------------------*/

Functions for finding neighbors
i/p = BB,II
o/p = ir = {nw, ne, sw, se}

-----------------------------------------------------------*/

Reflect(BB,II)

int II;
char BB;
{
    switch(II)
    {
    case 1:
        if((BB == 'N')||(BB == 'S')) ir = 3;
        else if ((BB == 'E')||(BB == 'W')) ir = 2;
        break;

    case 2:
        if((BB == 'N')||(BB == 'S')) ir = 4;
        else if ((BB == 'E')||(BB == 'W')) ir = 1;
        break;

    case 3:
        if((BB == 'N')||(BB == 'S')) ir = 1;
        else if ((BB == 'E')||(BB == 'W')) ir = 4;
        break;
case 4:
    if((BB == 'N')||(BB == 'S')) ir = 2;
    else if ((BB == 'E')||(BB == 'W')) ir = 3;
    break;
}
return(ir);
}/**------------------------------------------
i/p = *pq
o/p = is=(nw,ne,sw,se)=I
---------------------------------------------*/

Sontype(pq)
struct TREE *pq;
{
    struct TREE *tt;
tt=pq->root;
    if(tt->nw==pq) is=1;
    else if(tt->ne==pq) is=2;
    else if(tt->sw==pq) is=3;
    else if(tt->se==pq) is=4;
    else is=0;

    return(is);
}
Adj(B,I)
int I;
char B;
{
  switch(B)
  {
  case 'W':
    if(I == 1 || I == 3)
      ia = true;
    else ia=false;
    break;
  case 'N':
    if(I == 1|| I == 2)
      ia = true;
    else ia=false;
    break;
  case 'E':
    if(I == 2 || I == 4)
ia = true;
else ia=false;
break;

case 'S':
    if(I == 3 || I == 4)
        ia = true;
    else ia=false;
    break;

default:
    ia = false;
    break;
}

return(ia);
}

/*-----------------------------------------------

Function to find the neighbors
i/p :- *ppq,d
o/p :- *qp
-----------------------------------------------*/

struct TREE *neighbour(ppq,d)
struct TREE *ppq;
{
int IR;
if(((ppq->root)!=NULL)&&(Adj(d,Sontype(ppq))!=false))
{
    qp=neighbour(ppq->root,d);
}
else
    qp=ppq->root;

if((qp!=NULL) && (qp->nodetype==2))
{
    IR=Reflect(d,Sontype(ppq));
    switch(IR)
    {
    case 1:
        qp=qp->nw;
        break;
    case 2:
        qp=qp->ne;
        break;
    case 3:
        qp=qp->sw;
        break;
    case 4:
        qp=qp->se;
        break;
    }
return (qp);
}
else
return (qp);

/*-----------------------------------------------

i/p = qn,qne,n1,onesize
o/p = rstart[],cstart[]
-----------------------------------------------*/

fexecute()
{
for(lf=0;lf<=l;lf++)
{
    n=qn;
    newn=qne;
    for(kf=0;kf<=l;kf++)
    {
        rstrt[q]=n1;
        cstrt[q]=n;
        q++;
        n=newn;
        newn=newn+onesize;
    }
    n1=newn1;
Functions to implement stack

set_stack(ps)
struct stack *ps;
{
    ps->top= (-1);
}

push(ps)
struct stack *ps;
if(ps->top==(slimit-1))
    printf("stack overflow\n");
else
{
    ps->top= (ps->top) + 1;
    ps->stackarray[ps->top]=value;
}
}

/

 pop(ps)
 struct stack *ps;
 {
 if(empty(&ps))
    printf("stack underflow\n");
 else
 {
     pvalue=ps->stackarray[ps->top];
     ps->top=(ps->top) - 1;
 }
 }
Function to convert tree into a raster

i/p = *tras
o/p = *tras

tree_to_raster()
{
    int ik, rstart, cstart, rstop, cstop, len, lm, ml, ifo, ikk, jkk;
    double four, fo, xle;

    for(ml=1; ml<=7; ml++)
    {
        x=0;
        printf("ML=%d\n", ml);
        switch(ml)
        {
            case 1:
                len=256;
                break;
            case 2:
                len=128;
                break;
            case 3:
                len=64;
                break;
        }
case 4 :
    len=32;
    break;
case 5 :
    len=16;
    break;
case 6 :
    len=8;
    break;
case 7 :
    len=4;
    break;
}
level=ml;
four=4;
xle=level-1;
fo=pow(four,xle);
ifo=(int)fo;
for(j=1;j<=ifo;j++)
{
    for(ik=1;ik<=4;ik++)
    {
        x=x+1;
calculate();
        tras=t;
        for(lm=1;lm<level;lm++)
if (tras != NULL)
{
    fa[lm] = father[lm] % 4;
    if (fa[lm] == 1) tras = tras->nw;
    else if (fa[lm] == 2) tras = tras->ne;
    else if (fa[lm] == 3) tras = tras->sw;
    else if (fa[lm] == 0) tras = tras->se;
}

if (tras != NULL)
{
    switch (iposition)
    {
    case 1:
        tras = tras->nw;
        break;
    case 2:
        tras = tras->ne;
        break;
    case 3:
        tras = tras->sw;
        break;
    case 4:
        tras = tras->se;
break;

if((tras!=NULL)&&(tras->nodetype!=2))
{
    rstart=tras->row;
cstart=tras->col;
rstop=rstart+len;
cstop=cstart+len;
for(ikk=rstart;ikk<rstop;ikk++)
{
    for(jkk=cstart;jkk<cstop;jkk++)
    {
        binary[ikk][jkk]=tras->nodetype;
    }
}
}
}
Function to find the collision free path

flag=225=finish
222=visited
111=start

nodetype=10-@@=path

find_path()
{
    struct stack psn, psp, ptr[10];

    pathno=1;

    set_stack(&psn);
    set_stack(&ptr[1]);

    push(&ptr[1]);
    joy=13;
    trapped=false;
    freepath=false;
    while((trapped!=true) && (freepath!=true))
    {
        pvalue->flag=222;
    }
for(jj=4;jj>=1;jj--)
{
    switch(jj)
    {
    case 1:
        value=neighbour(pvalue,'E');
        break;
    case 2:
        value=neighbour(pvalue,'S');
        break;
    case 3:
        value=neighbour(pvalue,'N');
        break;
    case 4:
        value=neighbour(pvalue,'W');
        break;
    }
    if((value!=NULL)&&(value->flag==225))
    {
        freepath=true;
        push(&ptr[l]);
    }
    else
    if((value!=NULL)&&(value->nodetype==80)&&(value->flag!=222))
        push(&psn);
}
if(empty(&psn)&&(freepath!=true))
  trapped=true;
else if(freepath!=true)
{
  pop(&psn);
  value=pvalue;
  push(&ptr[l]);
  if(pvalue->flag==225)
    freepath=true;
}
}

if(trapped!=true)
{
  printf("came in path printing loop\n");
  while(!empty(&ptr[l]))
  {
    pop(&ptr[l]);
    pvalue->nodetype=joy;
    joy=joy+4;
  }
}
else
{
  printf("trapped !!! help !!!\n ");
main()
{
char infile[25], outfile[25], resultfile[25];

rowno = colno = length = 0;
type = 2;
root = initialize(NULL);

rstrt = ((int*) malloc(65536, sizeof(int)));
cstrt = ((int*) malloc(65536, sizeof(int)));
rstrt1 = ((int*) malloc(65536, sizeof(int)));
cstrt1 = ((int*) malloc(65536, sizeof(int)));

printf("Enter the input image filename : ");
scanf("%s", infile);
printf("Enter the output image filename : ");
scanf("%s", outfile);

}
printf("Enter the resultant tree filename : ");
scanf("%s",resultfile);

fp=fopen(resultfile,"w");
fd=fopen(infile,"r");

/*-----------------------------*/

fread(data,512*512,1,fd);
fclose(fd);
printf("array copyed\n");

size=512*512;
nsize=512;
nl=0;
q=1;
qn=0;
qne=newn1=onesize=nsize/2;
exit();

rrow[1]=((int*)calloc(4,sizeof(int)));
ccol[1]=((int*)calloc(4,sizeof(int)));
for(y=1;y<=4;y++)
{
    rrow[1][y]=rstrtl[y]=rstr[y];
    ccol[1][y]=cstrtl[y]=cstr[y];
for(level=2; level<=7; level++)
{
    printf("level=%d
", level);
    q = 1;
    four = 4;
    qx = pow(four, (double)level);
    two = 2;
    bx = pow(two, (double)level);
    onesize = nesize/(int)bx;

    for(ff = 1; ff<=(int)qx; ff++)
    {
        n1 = rstrtl[ff];
        newn1 = n1 + onesize;
        qn = cstrtl[ff];
        qne = qn + onesize;
        fexecute();
    }
}

rrow[level] = ((int*)malloc((int)qx*sizeof(int)));
col[level] = ((int*)malloc((int)qx*sizeof(int)));
for(yy = 1; yy<=(int)qx; yy++)
{
    rrow[level][yy] = rstrt[yy];
    col[level][yy] = cstrt[yy];
}
fx=(int)qx*4;
if(llevel!=7)
{
    for(y=0;y<=fx-1;y++)
    {
        rstrtl[y]=rstrt[y];
        cstrtl[y]=cstrt[y];
    }
}
/*--------------------------------------
   adding one level
   above the last level
--------------------------------------*/
printf("creating partial tree \n");

four=4;
for(ii=1;ii<=6;ii++)
{
    printf("II=%d\n",ii);
    pp=pow(four,(double)ii);
    ppint=(int)pp;
    for(j=1;j<=ppint;j++)
    {
        level=ii;
        x=j;
type=2;
length=0;
rowno=rrow[ii][j];
colno=ccol[ii][j];

calculate();

root=root;

add_to_tree(root,level);
}
}

/*---------------------------------------------------------
  adding last level to the tree
----------------------------------------------*/

printf("adding last level to the tree\n ");

x=0;
for(iii=1;iii<=16384;iii++)
{
    level=7;

    rowno=rstrt[iii];
colno=cstrt[iii];
\texttt{x=x+1;}

\texttt{type=(int)\text{data}[rstrt[iii]][cstrt[iii]];}

\texttt{length=0; calculate(); add\_to\_tree(root,level);} 

\texttt{\} 

\texttt{\} 

\texttt{/*----------------------------------------------*/}

\texttt{printf("going into merge\n");}

\texttt{t=root; merge(); root = tras = rt = t;}

\texttt{\} 

\texttt{\} 

\texttt{\} 

\texttt{\} 

\texttt{printf("going for finding path\n");}

\texttt{rt->root->sw->nw->ne->ne->flag=111; pvalue = value = rt->root->sw->nw->ne->ne; }
rt->root->se->ne->sw->nw->flag=225;

find_path();

/*---------------------------------------------------------------*/

printf("writing a tree into a file \n");

strcpy(path,"root");
print_tree(t->root,path);

/*---------------------------------------------------------------*/

printf("converting tree into a outfile \n");

tree_to_raster();

fd=fopen(outfile,"w");
fwrite(binary,512*512,1,fd);
fclose(fd);

printf("end\n");
}