4 CONCLUSIONS AND FURTHER REMARKS

We have, in this paper, presented the basic algorithm which has been implemented by the program enclosed. This algorithm uses the framed quadtree method as describe earlier to find the conditional shortest path in a known 2D environment. This algorithm in attempting to optimize the time and space taken to compute the conditional shortest path is held back by several limitations.

First, the path is usually not the absolute, but conditional, shortest path. This is due to the 'coarseness' of the path found by moving from frame to frame, and is exceptionally visible in environments with a lot of small scattered obstacles. However, in large open areas, this algorithm is extremely efficient and becomes a lot more accurate. Second, in small environments, this algorithm is just as slow as, if not slower, than the regular grid based approach. This however, has been remedied in this program. We implement the grid based approach for quads less than or equal to size four and the framed-quad approach to anything larger. Third, because of the present means of implementing the algorithm, the path found may not be the most desirable. The program does not have a second criterion to base the choosing of the path on if there exists two distinct paths with equal length.

The foundation has been set for further studies concerning the implementation of other framed-tree models. This project may be expanded to deal with 3D environments using structure known as framed-octrees (very similar to framed-quadtrees) and then taken one step further to unknown environments. At this point, possibilities for further research seem endless.

3.3.4 Side Four Scheduling

Side four scheduling is a little more complicated because domination of entry points is not as easily computed. This, along with the fact that each entry point on side one (even when unable to propagate to a point directly above it) may propagate to many cells on side four, make side four propagation perhaps the hardest of the four (figure 3.8). This problem is partially solved by scheduling only a subset of points after a specific time interval (once). This interval is determined through a worst case scenario in which an entry point has to propagate from one corner of the f-quad to the other corner. Regular Voronoi domains are assigned to this subset of entry points using the same formulas used in side two propagation.

Again, an overlap in the domains was in order. This may lead to redundancy and a loss of speed (slight) but will allow us to guarantee a correct conditional shortest path.

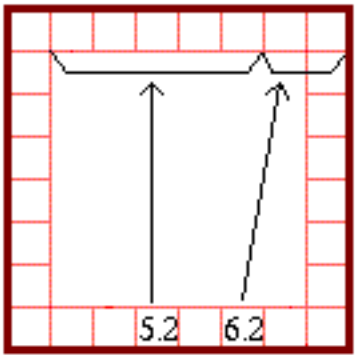


figure 3.8

3.3.5 Assigning Values

After determining the Voronoi domains of unblocked entry points, we may then assign appropriate values to the border cells or sides two, three, and four. This is done by traversing the entry point list and finding unblocked entry points with the potential to propagate within the given waveval. The value which it would assign is calculated and compared to the current value of the destination cell. If the new value is smaller, it is assigned to the cell and a pointer within the entry point structure (origin) is set to point to the cell from which it propagated. The interval is then incremented to represent the current propagating potential. This method of actually shrinking the domain of each entry point vastly reduces the space as well as computational time required to assign values to cells.

3.4 Finding the Shortest Path

After the goal quad has been reached (if it can), a fine path through specific cells and quads may be found by simply tracing the path back from the goal to the robot through the ‘origin’ pointers. This, we assert, is the shortest conditional path from the robot to the goal at this time, based on this framed quadtree algorithm.

or equal to that of the closer point. This, however, is evident upon inspection and will be accepted at face value.

After the initial points have been blocked, Voronoi domains are determined using the equation for a hyperbola with a and b being the focus points. This equation is a general fourth order equation and was solved with xmaple. The resulting equation was long and messy, but works quite well in determining the border cell which divides side two into the regions to be dominated by each of the two foci being considered. This procedure of assigning a region to each of the unblocked entry points is repeated from right to left.

The initial Voronoi scheduling algorithm set out by Sczcerba called for the domains to be split into two distinct regions, one including and above the division point and one below this point. In implementing the algorithm, however, we found it necessary for the division point to be included in both intervals. Although this means that the point will be checked twice by the entry points, it does ensure that no incorrect values are assigned due to round off errors.

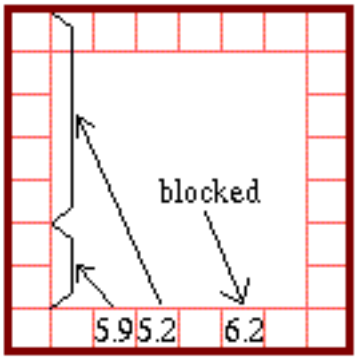


figure 3.6

3.3.3 Side Three Scheduling

Side three scheduling is merely a mirror image of side one scheduling, and implementation, with several minor modifications, is trivial (Figure 3.7).

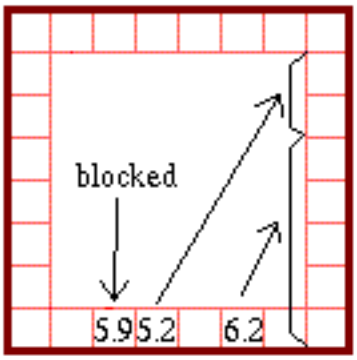


figure 3.7

may be implemented in linear time, greatly improves on the efficiency of regular quadtree propagating methods. This scheme is thus both accurate and efficient.

Entry points for a given f-quad are put into a linked list as they enter from border quads. These are the only cells which are allowed to propagate within an f-quad. All other cells are assigned values from within the f-quad and are dominated by at least one other cell (the one from which it came). During each iteration, the entry points are considered for internal propagation within each f-quad. This task is broken into four cases, one for each of the four sides to which the entry point may propagate. As there may be multiple entry points on each side, all coming in at different times and with different path grid values, a scheduling algorithm must be used to deal efficiently with the assignment of values. The algorithm is then repeated for entry points entering on each of the four sides.

In implementing this side scheduling algorithm, we used a control function (propogate_to_same_quad) which called the side one, two, and three scheduling functions four times, once for entry points on each of the four sides. Once the Voronoi domains were assigned to the entry points, the functions which assigned values to the appropriate cells were called (once again four times). Side four propagation was only performed once for each set of entry points entering on each side. This procedure is explained in detail below.

3.3.1 Side One Scheduling

Perhaps the easiest and most straightforward of the four cases, side one scheduling, is based on regular distance transform calculations. The waveval (iteration number) is used to calculate the distance which the entry point may propagate in either direction. Once this is determined, deciding whether or not to propagate is a trivial task.

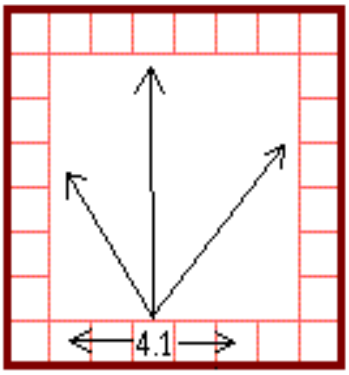


figure 3.5

3.3.2 Side Two Scheduling

Side two scheduling is a little trickier than side one scheduling. For the case of side two scheduling, every border cell may attempt to propagate to more than one side two cell (figure 3.6). A more complex scheduling algorithm is thus in order. Since the entry points arrive at different times and in no particular order, they must first be sorted. This came down to inserting an entry in the appropriate spot in a linked list, an easy task. The first step in scheduling these entry points is to remove all dominated entry points. It may be proved that if two entry points enter with two distinct path grid values then the point further from side two is dominated if its value is greater than

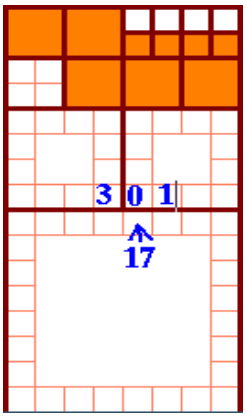
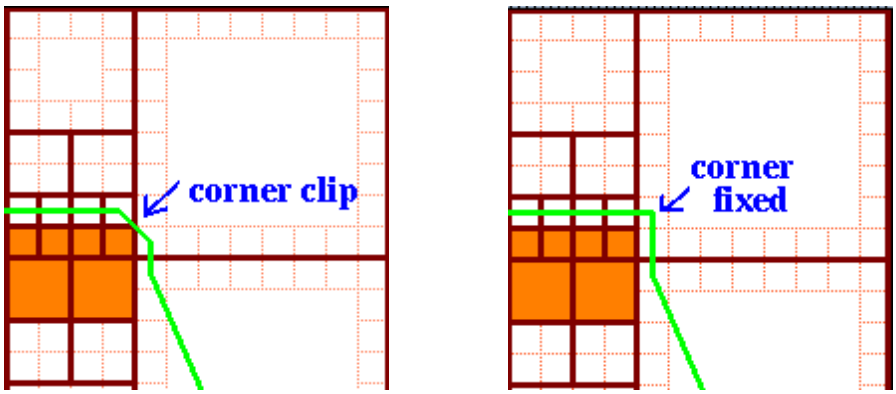


figure 3.3

3.2.4 Corner Clipping

This brought about the problem of corner clipping as shown in figure 3.4a. This path is not one that the robot would physical be able to execute. If the robot were to follow this path half of it would run into the obstacle. To alleviate this problem it was necessary to first check if the origin frame was a corner frame. Then we put in a check to determine if the neighbors frame was a corner frame. If both these conditions were met it was then determined if these two quads shared a



a. *b.*

figure 3.4

neighbor which was an obstacle. If it was then the diagonal propagation was not allowed to continue. With these checks in place the correct path, shown in figure 3.4b was found.

3.3 Propagation Within a Given Quad

A feature of this wave propagation algorithm is the method for propagating the wave within a particular f-quad. The wave is only propagated through border cells (or frames) of f-quads in the wave frontier table. Propagating to only these border cells, which fully define the entry and exit points of the robot from that particular f-quad, drastically reduces the calculations required to compute wave transforms for every cell within the f-quad. In fact, this method, which

mixed type. Special checks were put in to deal with cases where this quad was an obstacle. To find the index of the smaller quad we wrote a function called find_smaller which returns the quad that a particular frame is adjacent to and the index in this new quad. The method used is depicted in figure 3.2. First, by using the index value in the origin quad it is determined which child of the new neighbor to take. In this case since the frame index, 4, is greater than size/2 it can be determined that the northeast child is to be taken and the index is then decremented by size/2. Then the new neighbor quad is set to point at this northeast child. At this point the index has a value of 4-4 = 0. Since this is less than half the new quad size it is determined that the northwest neighbor must be taken and nothing is done to the index value. The new quad is then set to equal

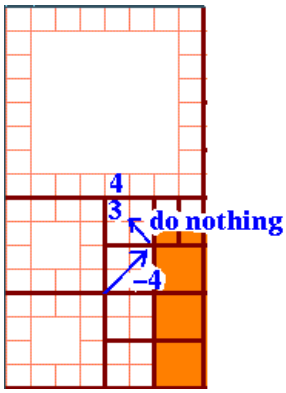


figure 3.2

to its northwest child and it is found that its type is free so this loop stops. The new index is then determined by subtracting the index from new quad size*3-3. As with determining the larger neighbors index simple equations are used to determine the relative index on each other side as compared to the smallest index on that side.

3.2.3 Propagating Diagonally

The next problem encountered was determining the diagonal quads that a particular frame could propagate to in other quads. In the case where the new index found was not a corner it was a simple matter of incrementing or decrementing this index and then just assigning this a value equal to the value of the origin frame plus the square root of 2. However, if the new found index was a corner then it was sometimes necessary to find a neighbor of this new quad. For example, in figure 3.3, the frame labeled 17 would be able to propagate to the frame labeled 3. However, to get the index and quad of this frame it is first necessary to use find_smaller going north to get the frame labeled 0 and then find_larger_or_eq going west on this new frame and quad to finally arrive at this frame labeled 3. Using the values we assigned for the directions (south =0, east=1, north=2, west=3) It was found that only 2 cases needed to be tested. If the new frame index was equal to size*(direction+1) mod 4 - (direction +1)mod 4, then a diagonal propagation could not occur in the same quad less one index and so a neighbor in (direction +2)mod 4 must be found. If the new frame index was equal to (direction +1)mod 4 then propagation in the same quad plus one index could not occur and the neighbor of this new quad in direction+1 must be found.

of these is in determining the frame of an adjacent quad which is adjacent to a given frame and a given quad. This involves knowing the size and type of the quad containing the origin and destination frames. Both of these bits of information were stored directly in the quadtree structure; the neighbors of a given quad were determined by the algorithms given by Samet and the type of quad was determined when the quadtree structure was created. This problem is then broken down into two parts. The first is determining the frame if the neighboring quad is larger or equal and the second is in this determination if the neighbor is smaller.

3.2.1 Propagating to Larger or Equal Sized Quads

To determine the frame index of a larger quad given the origin quad and the frame number we wrote a function find_larger_or_eq. This function takes the direction in which the origin quad is propagating, the neighboring quad in that direction, and the origin quad and returns the index of the frame in this larger or equal sized quad. This is done with a simple loop testing the sontype of the origin quad and incrementing the origin frame index accordingly and then setting the origin quad equal to its parent node. For example in figure 3.1 if the frame labeled 1 is able to propagate during this iteration of waveval, its southern neighbor is found by following its neighbors pointer contained in the quadtree structure. Next using a Samet function, sontype, it is found that the origin quad is a southeast child. So the index is incremented by the origin quad's size, which is four. Then the origin quad is set to its parent quad. After this it is found that this parent quad is the same size as the destination quad so this loop is finished. The index of the frame of the

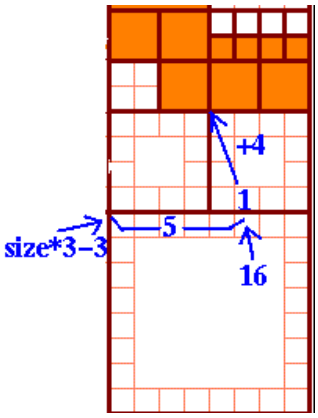


figure 3.1

southern neighbor is then found by taking its size*3 -3 and subtracting this new index to get 16, which is the index of this point. When moving in directions other than south a simple calculation was performed on the frame index because the frame indexes on these sides do not run from zero to size-1. It is an easy task to subtract the index of the lower corner from the given index to get a value from zero to size -1. From there all cases are parallel to the one given above.

3.2.2 Propagating to Smaller Quads

To determine the frame index of a smaller quad is a bit more difficult. The first problem is that the neighbors stored in the quadtree structure are greater than or equal in size to the origin quad. Therefore it is possible that the destination quad returned by the neighbor pointer is of

to have definite steps and user interface between parts of the program no variables may be passed to these functions as they must be called from the mouse or keyboard function. As a result we needed to define several global variables which made the program somewhat space inefficient.

2.3 The Bitmap

Having tackled the problems in learning OpenGL and GLUT the next task was to read in the bitmap. This bitmap was not a standard bitmap but was one output by a program written by Kristin Neustadt. We soon found that the image she output was turned sideways and flipped diagonally. Once we realized this the reading in and plotting of the bitmap was a simple task.

3 IMPLEMENTING THE ALGORITHM

3.1 Dividing Environment into Quadtrees

The quadtree is an approach to image representation based on repeatedly dividing the environment (or image) into quadrants. The process of subdividing a quadrant is repeated so long as the minimum resolution (in this case determined by the size of the robot) is not reached and there are both obstacle and free space in the quadrant being considered. Theoretically, this is a simple procedure, however, programming implementation was a little more difficult.

The environment used in this project was generated using Kristin Neustadt's program "robot" and saved as a bitmap file. A global pointer to a pointer of integers was assigned to point to the two dimensional array which was calloc'ed and used to store the scanned bitmap. Several methods for dividing the quadtree from this bitmap were tried and found to be inefficient and sometimes inaccurate.

The first method we attempted involved reading in the bitmap in a pseudo-random fashion, with flags keeping track of what object existed in a particular quad. A quad would be divided every time an obstacle and free space were detected in it. This method was very inefficient and lacked much insight. It was soon realized that every bit had to be eventually scanned (for a resolution of one bit, which we used throughout this project). Next, we tried creating an entire quadtree structure into which we read the bitmap and eliminated leaf nodes as required. This however, proved to be a strain on space and time. We proceeded to try to scan and create the quadtree structure from bottom up, scanning each node and then merging it. This too proved too great a task even for recursion.

The final method used seems to be that which provide for greatest efficiency as well as accuracy. In this method, we start at the head node and create Northwest nodes till the minimum resolution is reached. We then scan the bitmap and determine if the leafs are needed. If they are of the same kind, then they may be eliminated and their parent assigned the appropriate type. We then move up the structure into a quad which has children which have not been scanned, and proceed in the same way as before. This method lends itself well to recursion and improves on prior methods which are either inaccurate, slow, or space consuming.

3.2 Propagation to Other Quads

In using framed quadtrees to propagate the path planning wave, several problems arise in determining which other quads a particular frame is able to propagate into. This appears a simple task in theory, however, there are many subtle peculiarities to deal with in programming. The first

```
  double enter_time[4];
}wft_t;
```

The Wave Frontier Table is a linked list of quads that are able to propagate at a given time. It contains a pointer to the head of a list of entry points for each side, *entry[4]. It also contains the enter_time for each side which is the value of waveval at the time the first entry point came into a side. This time is needed in side 4 propagation. A quad is first put into the wft when the first entry point arrives. It is removed when none of its frames are able to propagate any further.

2.1.4 Entry Points

The entry point structure was simply:

```
typedef struct entry_s {
  int index;
  int interval1[2];
  int interval2[2];
  int interval3[2];
  int interval4[2];
  struct entry_s *next;
  struct entry_s *prev;
} entry_t;
```

This linked list of entry points store all available entry points on a given side of a quad. Index refers to the index of the cell in which the robot may potentially enter a quad. The intervals are used in the internal side propagation to keep track of what intervals the point may propagate to and are not dominated on. These are initially assigned a value of -88 and later assigned index values as necessary.

2.1 Graphics Packages: OpenGL and GLUT

2.2.1 OpenGL

The graphics package used for this project was openGL. We found it no harder or easier to program than other graphics packages we have used. However, it seems that it will make things simpler when this project is implemented in three dimensions and also seems to work a bit faster than SRGP.

2.2.2 GLUT

The biggest problem with using OpenGL is that it does not come with its own windowing system. GLUT (OpenGL Utility Toolkit) is a windowing package that was designed to interface with OpenGL. However, we found it had some rather annoying quirks that lead to inefficiency and failure. GLUT will not execute any drawing functions until glutMainLoop is called. Once this function is called it calls all the callback functions that have been defined. These callback functions include such things as a mouse function, keyboard function, normal display functions, etc... However the variables that these functions receive are predetermined. So if a programmer wishes

```
  frame_t *frames;
  int ulx, uly;
}quadr_t;
```

Where size holds the length of one edge of the quad and type holds the type of the quad. Valid types are FREE, OBS, or MIXED, to define all free, all obstacles, or some of both respectively. Mother and children point to the quad's parent and four children as defined by the quadtree. We used the algorithms given by Samet to find the neighbors and the mother and children were assigned as the quadtree structure was built. *frames is later allocated space as a one dimensional array for quads of type FREE and ulx and uly are the coordinates of the upper left corner of the quad. We defined the directions and index values for the frames in this manner:

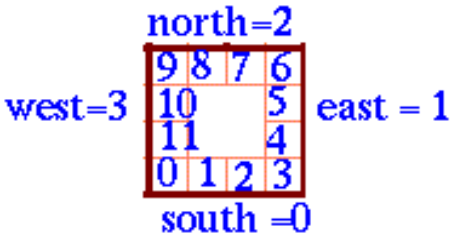


figure 2.1

When dealing with children, northwest was 0, northeast was 1, southeast was 2, and southwest was 3.

2.1.2 Frames

Our structure for the frames held these values:

```
typedef struct frame_s {
  int index;
  struct frame_s *origin;
  struct quadr_s *owner;
  double value;
} frame_t;
```

The index was defined as above. Origin is a pointer to the frame that this frame got its value from. Owner is a pointer to the quad the frame is in. Value is the distance assigned to this frame from the robot and index is the index of this frame in the array of frames in the owner quad. This was necessary to compute its location when tracing the path back from the goal to the robot.

2.1.3 Wave Frontier Table

The wave frontier table was:

```
typedef struct wft_s {
  quadr_t *quad;
  entry_t *entry[4];
  struct wft_s *next;
```

1 INTRODUCTION

1.1 An Overview

Planning collision free paths of shortest distances through known obstacle-scattered environments has been studied intensively. As a result, the algorithms which exist to find shortest paths have evolved considerably recently. This project is an attempt to implement a relatively new algorithm based on a framed-quadtree in order to find the conditional shortest path through a known 2D environment. The framed-quadtree method utilizes a combination of regular grid-based as well as quadtree methods. This algorithm may be expanded to deal with 3D and eventually with unknown environments.

1.2 Path Planning

The main loop to the path planning part of our program is relatively simple. It simply takes the first entry on the wave frontier table or wft (which initially contains only the robot quad) and calls the function which deals with propagating within a quad (if the entry is the robot quad, these values are pre-assigned). It then calls the function to propagate to other quads. This function adds and removes quads from the wave frontier table as need be and assigns entry points. The current wave table entry is then incremented and it returns to the inner loop. This continues until the end of the wft is reached. At this point the current wft entry is set to point back to the head entry and the loop repeats with an incremented iteration number or waveval. This goes on either until the wft is empty, which signifies that the goal could not be reached from the robots current position, or until the goal quad has been on and removed from the wave frontier table. In the former case a message is displayed defining the futility of the robot's quest. In the latter case the index in the robot quad is found which contains the shortest path back to the robot and this path is drawn. If the robot and goal are in the same quad, a line is simply drawn from the robot straight to the goal and the path planning loop is never entered.

2 PRELIMINARIES

2.1 Basic structures

The structures set up to store the information seem to define a large part of the program and algorithm and as such must be noted and understood. The following are the important structures used:

2.1.1 The Quad

Each quad structure was defined as thus:

```
typedef struct quadr_s{
  int size, type;
  struct quadr_s *mother, *children[4];
  struct quadr_s *neighbors[4];
```

Abstract

The motion planning problem is of central importance to the fields of robotics, spatial planning, and automated design. In robotics, we are concerned in the automatic synthesis of robot motions, given specifications of tasks and geometric models of the robot and the obstacles. The Mover's problem is to find a continuous, collision free path for a moving object through an environment containing obstacles.

In this project, we implement an algorithm which finds the conditional shortest path, a collision-free path of shortest distance based on known information on an obstacle-scattered environment at a given time. This method utilizes a circular path planning wave and is based on a revolutionary data structure, the framed quadtree, which improves upon existing square-grid and quadtree-based techniques. This algorithm works in linear time and guarantees a conditional shortest path in any known 2-D environment.

Keywords: *Framed-quadtrees, Quadtrees, Shortest Paths, Voronoi Domain, Wave Frontier Table*

Simulation of Euclidean Shortest Path Planning Algorithms Based on the Framed-Quadtree Data Structure

Desney S. Tan *James T. Herro III* *Robert J. Szczerba*

Technical Report: #95-26

October 1995

Department of Computer Science and Engineering

University of Notre Dame

Notre Dame, Indiana 46556

Corresponding Authors:

Desney S. Tan and James T. Herro

Department of Computer Science and Engineering

Fitzpatrick Hall of Engineering

University of Notre Dame

Notre Dame, Indiana 46556