Path Planning for the Dual-Use Detachable Mobile Manipulator

Charles W. Su

Abstract: This report describes the work done for path planning for the mobile base of the Dual-Use Detachable Mobile Manipulator $(DM)^2$ project. It is intended to be an overview of the work done thus far. In addition it will document all code done and include a users manual for the off-line path planner.

Mobile Base Path Planing

This section will describe the theory and the execution of the path planner designed for the mobile base of the $(DM)^2$. Due to the kinematics of the car-like base, it is not possible to follow any given path. To avoid this problem, the path planner currently has a resolution of two feet by two feet. This allows us to ignore the nonholonomic equations which arise from the kinematics of the base.

Wavefront Expansion Method for Path Planning

The mobile base planner assumes a two degree-of-freedom (two translations) and no rotations. The robot is assumed to be able to move in any direction at any given position or time (we will see later that this is not the case for the $(DM)^2$ mobile base). Wave front expansion is a numerical potential field technique for robot path planning. An in depth discussion of this method is available in [1], [2], and [3].

Initialization of the Workspace

The wavefront expansion method relies on the initialization of the workspace given a particular goal location. The workspace is divided up into an n-by-n grid. Each location in the grid in which an object occupies is then designated as off-limits. The goal is then assigned the value 1. Any cell which neighbors the goal cell and is not an object is given the value 2. Any cell which neighbor a cell with value 2 and has not already been assigned a value is given the value 3. This process is repeated until every cell in the grid has been assigned a value. Figure 1 shows the completed grid after the wave has been propagated. Zero represents the goal location and 99 represent the objects.

FIGURE 1. Workspace Matrix After Initalization

4	3	2	2	2	2	2	3	4	5
4	99	99	1	1	1	2	3	4	5
5	99	99	1	0	1	2	3	4	5
6	99	99	99	99	99	99	3	4	5
7	99	99	99	99	99	99	3	4	5
8	99	99	99	99	99	99	4	4	5
9	10	9	8	7	6	5	5	5	5
10	10	9	8	7	6	6	99	99	99
11	10	9	8	7	7	7	99	99	99
11	10	9	8	8	8	8	99	99	99

In a worst case situation, this algorithm will take $\lceil n/2 \rceil \times \lceil m/2 \rceil$ traversals to fill in the entire grid, where n and m are the number of grids per side. In a sparse workspace, only $\sqrt{n^2 + m^2}$ (the length of the diagonal) traversals are needed. For a grid with a cells, the complexity of the algorithm is *O(a)*. Figure 2 shows a 3-D representation of the workspace in figure 1 in which the values at each point is squared to accentuate the "slope" of the workspace toward the goal, the goal is represented by the lowest point on the graph. Figure 3 shows the 'rear' view of the same system along with a contour map of the system.

FIGURE 2. 3-D View of Example Workspace

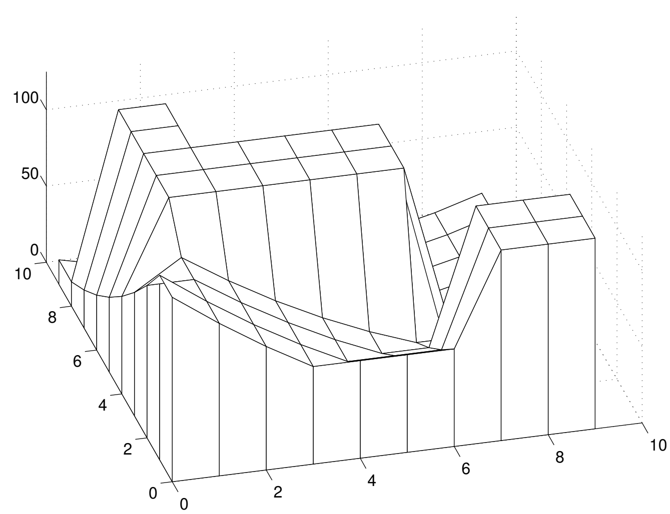


FIGURE 3. 'Rear' View of Example Workspace

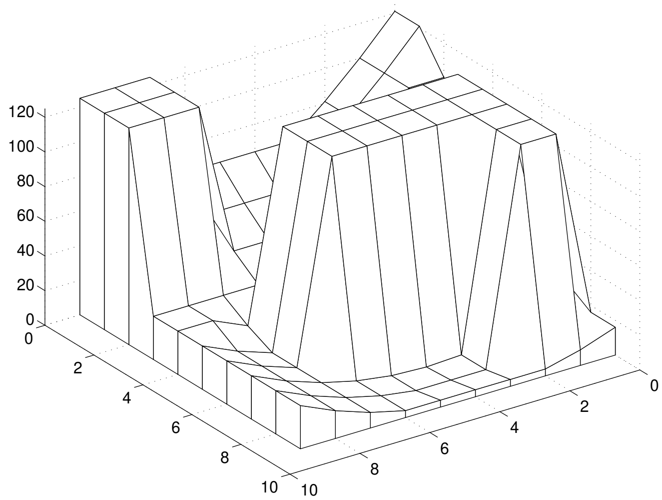
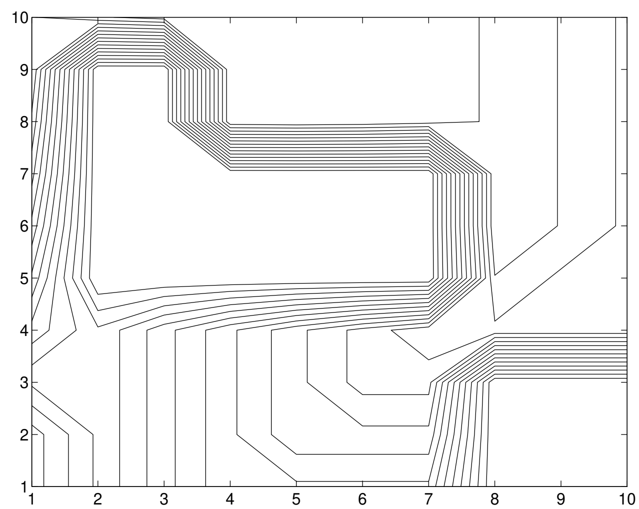


FIGURE 4. Contour of Example Workspace



Movement

Motion from any given to a give goal is trivial once the workspace has been initialized. Movement is accomplished by always moving to the neighboring cells with the lowest value. See figure 5 for some example paths in which movement is restricted to 90 degrees. (The shaded cells represent the path taken). In the case in which there is more than one neighboring cell with the lowest value, all of the cells with that value are the same distance to the goal.

FIGURE 5. Example Paths

4	3	2	2	2	2	2	3	4	5
4	99	99	1	1	1	2	3	4	5
5	99	99	1	0	1	2	3	4	5
6	99	99	99	99	99	99	3	4	5
7	99	99	99	99	99	99	3	4	5
8	99	99	99	99	99	99	4	4	5
9	10	9	8	7	6	5	5	5	5
10	10	9	8	7	6	6	99	99	99
11	10	9	8	7	7	7	99	99	99
11	10	9	8	8	8	8	99	99	99

It can be shown that the path chosen using this method will always be of minimal length. In addition, this method is local-minima free and guarantees a path from any given point to the goal provided that one exists. If we view our system as the 3-D representation in figure 2, and that the robot as a marble, then we should just watch how the ball rolls to the goal to obtain the path.

Implementation

The wavefront path planner was implemented for $(DM)^2$'s mobile base. The room was divided up into 2 foot by 2 foot cells and all objects were noted. The robot was sucessfully navigated the workspace while maintaining a safe distance from the objects.

$(DM)^2$ Mobile Base Control

The $(DM)^2$ mobile base is an approximately 2' by 3' tricycle-like robot and weights approximately. 35 kg (see figure 3). Each of the wheels is solid rubber and is 10 inches in diameter and 2.75 inches wide. The two rear wheels are spin freely on a common axis, while the front wheel turns and drives the robot.

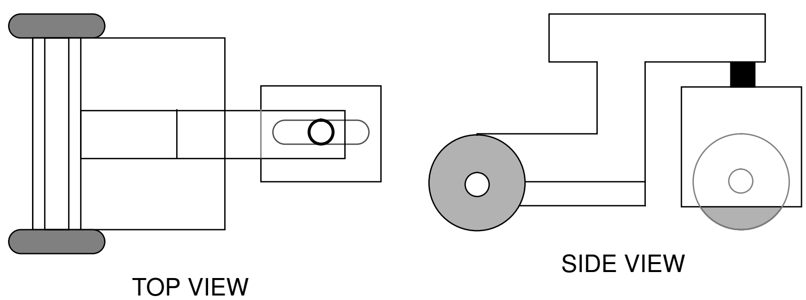
The floor surface varies from tile to low-pile carpet, which causes some difficulty in designing a controller that works on both surfaces. It take approximately two times as much force to turn the wheel on carpet as on tile. This made designing a controller difficult. In addition, the only feedback that was available was steering-angle and front wheel-position. This limited us to a dead-reckoning system for movement, and errors is position were accumulated.

The final design for the controller for the steering is PID with velocity limit. The gains for the proportional and integral term also change depending on the current error. When the error is 'big', the integral gain is set to zero to reduce integral wind-up and the proportional gain is set at a large value. Once the steering is within a user-specified region around the destination, the proportional gain is set to be very small and the integral gain is turned on. At all times, the controller is limiting the velocity of the steering angle change in order to reduce overshoot. Trajectory generation is achieved by never allowing the input to lead the output by a certain amount.

In testing, the steering controller performed acceptable on both the carpet and the tile surfaces, with reaction on the tile surface approximately 50% faster than that on the carpet.

The driving motor also utilized PID controller design, in addition friction and damping terms were included. The controller for the driving motor also check to make sure that the steering angle was with 1% of the desired angle. If this condition was not met, the driving motor was halted while the steering fixed itself. This reduced the amount of position error accumulated while the robot was running.

FIGURE 6. Mobile Base



Discussion

The path planner is a robust planner for holonomic systems. Because of the non-holonomic nature of the $(DM)^2$ base, the workspace resolution was vary coarse. The planner itself runs fast. With the proper modifications in the planner, however, it should be possible to increase resolution. The code for the path planner and a manual may be found in appendix A.

The mobile base performed acceptably given the current feedback. Performance would be greatly improved if external location feedback was available. Future improvements include tele-operation of the mobile base, added feedback, and possibly a new base.

References

1. J. Barraquand, B. Langlois, and J.C. Latombe, Numerical Potential Field Techniques for Robot Path Planning, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 22, No.2, March 1992.
2. A. Zelinsky, R.A. Jarvis, J.C. Byrne, and S. Yuta, Planning Paths of Complete Coverage of an Unstructured Environment by a Mobile Robot, Proceeding from ICAR 1993.
3. J.C. Latombe, Robot Motion Planning, Kluwer Academic Publishers, 1991.
4. F. Ben Amar, Ph. Bidaud, and F. Ben Ouezdou, On Modeling and Motion Planning of Planetary Vehicles, Proceedings of the 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems
5. Y. Yamamoto and X. Yun, Coordinating Locomotion and Manipulation of a Mobile Manipulator, X. Yun and Y. Yamamoto, Internal Dynamics of a Wheeled Mobile Robot, Proceedings of the 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems
6. C. Wang and V. Kumar, Velocity Control of Mobile Manipulators,
7. J.E. Borrow and R.M.C. Bodduluri, A General Path Planning Algorithm, Proceedings of the 27th Conference on Decision and Control, December 1988, pp. 2292-2293
8. M. Minami, N. Fujiwara, and K. Kanbara, Inverse Dynamics Calculation Method of Power Wheeled Mobile Manipulators, Proceedings of the 1993 IEEE/RSJ International Conferencs on Intelligent Robots and Systems, pp. 781-786.
9. H.A. Vasseur, F.G. Pin, J.R. Taylor, Navigation of a Car-like Mobile Robot using a Decomposition of the Environment in Convex Cells, Proceedings of the 1991 IEEE International Conference on Robotics and Automation, pp. 1496-501
10. J.B. Pomet, B. Thuilot, G. Bastin, and G. Campion, A Hybrid Strategy for the Feedback Stabilization of Nonholonomic Mobile Robots, Proceedings of the 1992 IEEE International Conferencs on Robotics and Automation, pp 129-134
11. B. d'Andrea-Novel, G. Bastin, and G. Campion, Dynamic Feedback Linearization of Nonholomic Wheeled Mobile Robots, Proceeding of the 1992 International Conference on Robotics and Automation, pp 2527-1797.
12. B. Steer and M. Larcombe, A Goal Seeking and Obstacle Avoiding Algorithm for Autonomous Mobile Robots,
13. Y. Dote, Applications of Modern Control Techniques to Motor Control, Proceedings of the IEEE, Vol. 76, No. 4, April 1988, pp 438-54
14. A. Pruski and S. Rohmer, Multivalue Coding: Applications to Autonomous Robot Path Planning with Rotations, Proceedings of the 1991 IEEE International Conferencs on Robotics and Automations, pp 694-699
15. C.C de Wit and O.J. Sordalen, Exponential Stabilization of Mobile Robots with Nonholonomic Constratins, IEEE Transactions on Automatic Control, Vol 37, No. 11, November 1992, pp. 1791-7.

Appendix A

This appendix explains the usage of the wavefront path planner. There are three files that must exist in order for the path planner to work: the executable *path*, and the two input file *const.file* and *object.file*. Currently the program does not compile on Suns, but does compile on any Decstation running AFS.

To execute the program, simply run the executable *path*. If the program crashes it may be because there is no path from the start to the goal. Figure 9 shows the output of the system given the input files in figure 7 and 8.

path.c

The source code named *path.c* contains the C code for the path planner. In addition, it contains the variable XMAX and YMAX which determine the size of the workspace.

const.file

This file contains the beginning and end points for the robot, in addition to a buffer value with which the path planner will expand all object to. The fist line contain the x and y location for the starting and ending points for the head of the robot. The second line contains the starting and ending points for the tail of the robot. The third line contain the number of cells to expand all object by in both the x and y directions. Care must be made to insure that the starting and ending points are realizable. An example *const.-file* is shown in figure 7.

FIGURE 7. Sample output

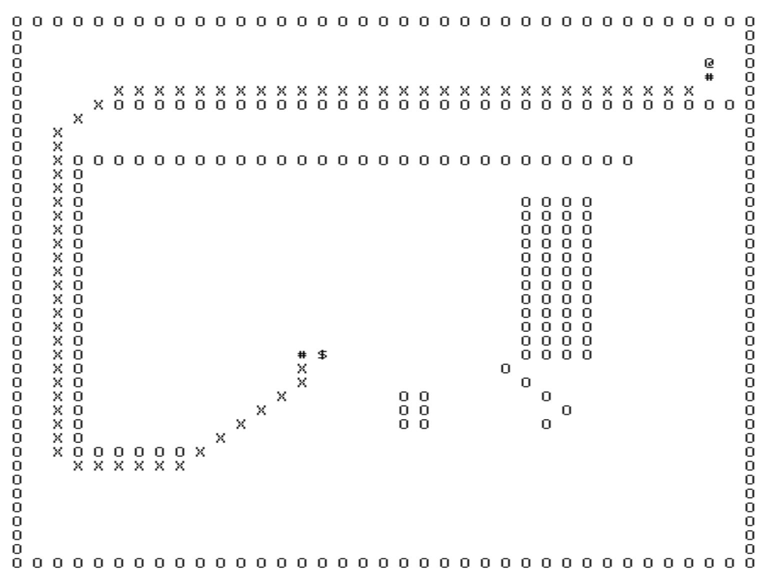


FIGURE 8. const.file

34 36 15 15
34 35 14 15
0

object.file

This file contains the locations of all of the object in the workspace. The first line contain the number of objects. Every subsequent line contains the location of one object in the form of YMIN, YMAX, XMIN, XMAX. Care must be take to insure that the MAX's are greater than the MIN's. An example *object.file* is shown in figure 8.

FIGURE 9. object.file

11
33 33 5 37
29 29 3 30
15 26 25 28
14 14 24 24
13 13 25 25
12 12 26 26
11 11 27 27
10 10 26 26
8 29 3 3
8 8 3 8
10 12 19 20

Source Code for *path.c*

```
/* path.c
 a general path planner using the wavefront propagation method
 written by Charles Su
 version 1.0
 May 6, 1994
 */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>


#define MAX_X 37 /* the size of the workspace in the x-direction */
#define MAX_Y 40 /* the size of the workspace in the y-direction */
#define FREE_VAL -999 /* the value given to an initilized but unused cell */
#define OBJECT_VAL 777 /* the value of a cell with an object in it */
#define GOAL_VAL 0 /* the value given to the goal */
#define HOME_VAL 888 /* the value given to the goal */
#define SAFETY_VAL 666 /* the value given to the buffer region around objects */


enum heading {N, NE, E, SE, S, SW, W, NW};
```

```
struct point{ /* this is the format of each cell in the map */
 int x;
 int y;
 int value;
};

struct cell{
 struct point start;
 struct point goal;
};

/* path_maker is commented out because it is not needed */
/*void path_maker(struct cell *head, struct cell *tail, struct point *path_list); */

int turn_angle (struct point *tail, struct point *current_head, struct point *desired_head);
int determine_heading (struct point *head, struct point *tail);
void find_path(int map[MAX_X][MAX_Y], struct cell *head, struct cell *tail, struct point *path_list);
struct point *palloc(int size_desired);
void make_wave (int map[MAX_X][MAX_Y], struct cell *tail);
struct cell *cell_alloc(void);
void init_cell(struct cell *cell2binit);
void copy_map(int from_map[MAX_X][MAX_Y],int to_map[MAX_X][MAX_Y]);
void make_safe (int map[MAX_X][MAX_Y], int safety);
void initialize_map (int map[MAX_X][MAX_Y]);
int get_constants (char *name, struct cell *head, struct cell *tail);
int get_objects (int map[MAX_X][MAX_Y], char *name);
void print_map (int map[MAX_X][MAX_Y]);
void save_map (int map[MAX_X][MAX_Y], char *name);


void main (){
 struct cell head, tail;
 int wave[MAX_X][MAX_Y];
 int temp[MAX_X][MAX_Y];
 int num_objects;
 int safety;
 struct point *path_list;


 safety = get_constants (“const.file”, &head, &tail);
 initialize_map (wave);
 num_objects = get_objects (wave, “object.file”);

 make_safe(wave, safety);
 make_wave(wave, &tail);

 path_list = palloc((tail.start.value) + 12);

 copy_map(wave,temp);
 find_path(wave, &head, &tail, path_list);
 /*path_maker(&head, &tail, path_list); */

 exit(0);

}

/****************
 this procedure takes in the path_list and outputs the sequence file which we use
 for input to DDMM. Currently commented out because it is not needed
 ****************/

/*void path_maker(struct cell *head, struct cell *tail, struct point *path_list){
 enum move {TURN, FOWARD};

 struct point *temp_point;
 int delta_x, delta_y;
 int turn;
 int done = 0;
 int last_move;


 struct point *current_tail;
 struct point *next_tail;
 struct point *current_head;
 struct point next_head;

 struct point last_head;
 struct point last_tail;

 /* find out if we initially need to turn
 temp_point = path_list;
 temp_point++;

 if ((temp_point->x != head->start.x) || (temp_point->y != head->start.y)){
```

```
turn = turn_angle (&tail->start, &head->start, temp_point);
printf("Turn %3d deg.\n",turn );
last_move = TURN;
}
else
last_move = FOWARD;

current_tail = path_list;
next_tail = path_list + 1;
current_head = next_tail;

/* go until the tail is at it's goal
while(!(done)){
delta_x = (next_tail->x) - (current_tail->x);
delta_y = (next_tail->y) - (current_tail->y);
next_head.x = (current_head->x) + delta_x;
next_head.y = (current_head->y) + delta_y;
last_head.x = current_head->x;
last_head.y = current_head->y;
current_tail->x = next_tail->x;
current_tail->y = next_tail->y;
last_tail.x = current_tail->x;
last_tail.y = current_tail->y;
next_tail++;
if (((next_head.x) != (next_tail->x)) ||
((next_head.y) != (next_tail->y))){
printf("Move to (%2d,%2d), ", current_tail->x, current_tail->y);
turn = turn_angle (&last_tail, &next_head, next_tail);
printf("turn %3d degrees.\n",turn );
last_move = TURN;
}
else
last_move = FOWARD;

current_head->x = next_tail->x;
current_head->y = next_tail->y;
if (((next_tail->x) == (tail->goal.x)) &&
((next_tail->y) == (tail->goal.y)))
done = 1;
}

/* check to see if we have to do a final rotation
delta_x = (next_tail->x) - (current_tail->x);
delta_y = (next_tail->y) - (current_tail->y);
next_head.x = (current_head->x) + delta_x;
next_head.y = (current_head->y) + delta_y;
if ((next_head.x != head->goal.x) || (next_head.y != head->goal.y)){
printf("Move to (%2d,%2d), ", next_tail->x, next_tail->y);
turn = turn_angle (&tail->goal, &next_head, &head->goal);
printf("turn %3d deg.\n DONE\n",turn );
}
else
printf("Move to (%2d,%2d).\n DONE.\n ", next_tail->x, next_tail->y);
}
*/

/************
this function calculates the turn angle given the current tail location
and the current and next head location
************/
int turn_angle (struct point *tail, struct point *current_head, struct point *desired_head){
int current_heading = determine_heading(current_head, tail);
int desired_heading = determine_heading(desired_head, tail);
int angle = -45*(desired_heading - current_heading);

if (angle > 180) return (360 - angle);
else
if (angle < -180) return (360 + angle);
else
return angle;
}

/*************
this function determines the heading of the robot
given the head and tail location
************/

int determine_heading (struct point *head, struct point *tail){
int dx = head->x - tail->x;
int dy = head->y - tail->y;

if ((dx == 0) && (dy == 1) ) return N;
if ((dx == 1) && (dy == 1) ) return NE;
if ((dx == 1) && (dy == 0) ) return E;
if ((dx == 1) && (dy == -1)) return SE;
```

```
 if ((dx == 0) && (dy == -1)) return S;
 if ((dx == -1) && (dy == -1)) return SW;
 if ((dx == -1) && (dy == 0) ) return W;
 if ((dx == -1) && (dy == 1) ) return NW;
}

/***********
 this function allocates memory for n points
 ***********/
struct point *palloc(int points_desired){
 return (struct point *) malloc( points_desired * sizeof(struct point));
}


/**********
 this function acutally find the path given the map, and the starting head and tail locations
 *********/
void find_path(int map[MAX_X][MAX_Y], struct cell *head, struct cell *tail, struct point *path_list){
#define PATH_VAL 555 /* value of a cell if it is on the path */
#define VISITED 444 /* value of a cell that has been visited */

 int true = 1;
 int false = 0;
 int x,y;
 int home = false;
 int x_delta, y_delta;
 int x_check, y_check;
 int x_current, y_current;
 struct point low;

 x_current = tail->start.x;
 y_current = tail->start.y;
 map[x_current][y_current] = map[tail->start.x][tail->start.y];
 tail->start.value = map[tail->start.x][tail->start.y];
 path_list->value = map[x_current][y_current];

 while(!(home)){
 low.x = x_current;
 low.y = y_current;
 low.value = map[x_current][y_current];

 path_list->x = x_current;
 path_list->y = y_current;
 path_list++;

 for (x_delta = -1; x_delta < 2; x_delta++){
 x_check = x_current + x_delta;
 for (y_delta = -1; y_delta < 2; y_delta++){
y_check = y_current + y_delta;
if ((x_check > 0) && (x_check < MAX_X) &&
 (y_check > 0) && (y_check < MAX_Y)){
 switch(map[x_check][y_check]){
 case OBJECT_VAL : break;
 case VISITED : break;
 default :
 if (map[x_check][y_check] < low.value){
low.x = x_check;
low.y = y_check;
low.value = map[low.x][low.y];
 }}}}}
 path_list->value = map[low.x][low.y];
 map[x_current][y_current] = PATH_VAL;

 for (x_delta = -1; x_delta < 2; x_delta++){
 x_check = x_current + x_delta;
 for (y_delta = -1; y_delta < 2; y_delta++){
y_check = y_current + y_delta;
if ((x_check > 0) && (x_check < MAX_X) &&
 (y_check > 0) && (y_check < MAX_Y)){
 switch(map[x_check][y_check]){
 case OBJECT_VAL : break;
 case PATH_VAL : break;
 default :
 map[x_check][y_check] = VISITED;
 }}}}

 x_current = low.x;
 y_current = low.y;

 if((x_current == tail->goal.x) && (y_current == tail->goal.y)){
 home = true;
 map[x_current][y_current] = HOME_VAL- 1;
 map[head->goal.x][head->goal.y] = GOAL_VAL;
 map[head->start.x][head->start.y] = HOME_VAL;
 map[tail->start.x][tail->start.y] = HOME_VAL+1;
```

```
 path_list->x = x_current;
 path_list->y = y_current;
 path_list->value = 0;


 }
 /* for( y = (MAX_Y - 1); y >= 0; y--){
for (x = 0; x < MAX_X; x++){
switch (map[x][y]){
case PATH_VAL: printf( " X"); break;
case OBJECT_VAL : printf( " O"); break;
case GOAL_VAL: printf( " $"); break;
case HOME_VAL: printf( " @"); break;
case (HOME_VAL-1): printf( " #"); break;
case (HOME_VAL+1): printf( " #"); break;
default : printf( " ", map[x][y]);
}
}
printf("\n");
}
printf( "\n");
system("clear"); */
 }

 for( y = (MAX_Y - 1); y >= 0; y--){
 for (x = 0; x < MAX_X; x++){
 switch (map[x][y]){
 case PATH_VAL: printf( " X"); break;
 case OBJECT_VAL : printf( " O"); break;
 case GOAL_VAL: printf( " $"); break;
 case HOME_VAL: printf( " @"); break;
 case (HOME_VAL-1): printf( " #"); break;
 case (HOME_VAL+1): printf( " #"); break;
default : printf( " ", map[x][y]);
 }
 }
 printf("\n");
 }
 printf( "\n");


}

/*************
 This function take in the initialized map and the final tail location
 and generates the map with the wave propagated thru
 ************/
void make_wave (int map[MAX_X][MAX_Y], struct cell *tail){
 int count = 1;
 int lastval = GOAL_VAL;
 int currentval = GOAL_VAL + 1;
 int x , y;
 map[tail->goal.x][tail->goal.y] = GOAL_VAL;

 while(count != 0){
 count = 0;
 for(x = 0; x < MAX_X; x++){
 for(y = 0; y < MAX_Y; y++){
if ((map[x][y] == FREE_VAL) &&
 (map[x][y] != OBJECT_VAL) &&
 ((map[x][y - 1] == lastval) ||
 (map[x][y + 1] == lastval) ||
 (map[x - 1][y] == lastval) ||
 (map[x + 1][y] == lastval))){
 map[x][y] = currentval;
 count++;
}
 }
 }
 lastval = currentval++;
 }
 tail->start.value = map[tail->start.x][tail->start.y];
}

/***********
 This function allocated room for a cell
 **********/
struct cell *cell_alloc(void){
 return (struct cell * ) malloc(sizeof(struct cell));
}

/***********
 This function initializes a cell to NULL
 **********/
void init_p(struct cell *cell2binit){
```

```
 cell2binit->start.x = NULL;
 cell2binit->start.y = NULL;
 cell2binit->start.value = NULL;
 cell2binit->goal.x = NULL;
 cell2binit->goal.y = NULL;
 cell2binit->goal.value = NULL;
}

/**********
 This function copies one map onto another
 *********/
void copy_map(int from_map[MAX_X][MAX_Y],int to_map[MAX_X][MAX_Y]){
 int x,y;

 for ( x = 0; x < MAX_X; x++)
 for ( y = 0; y < MAX_Y; y++)
 to_map[x][y] = from_map[x][y];
}

/*********
 This function expands all object by safety number of cells
 ********/
void make_safe (int map[MAX_X][MAX_Y], int safety){
 int x, y;
 int delta_x, delta_y;
 int last_val = OBJECT_VAL;
 int current_val = SAFETY_VAL;

 for (;safety--;){
 for ( x = 1; x < MAX_X; x++){
 for ( y = 1; y < MAX_Y; y++){
for ( delta_x = -1; delta_x <= 1; delta_x++){
 for ( delta_y = -1; delta_y <= 1; delta_y++){
 if ((map[x][y] == FREE_VAL) &&
(map[x + delta_x][y + delta_y] == last_val)){
 map[x][y] = current_val;
 }}}}}
 last_val = current_val--;
 }
}

/********
 This function initializes the map and puts a wall around the
 workspace
 *******/
void initialize_map(int map[MAX_X][MAX_Y]){
 int x,y;

 for (y = 0; y < MAX_Y; y++){
 for (x = 0; x < MAX_X; x++){
 map[x][y] = FREE_VAL;
 map[x][0] = OBJECT_VAL;
 map[x][MAX_Y-1] = OBJECT_VAL;
 }
 map[0][y] = OBJECT_VAL;
 map[MAX_X-1][y] = OBJECT_VAL;
 }
}

/*******
 This function gets all of the values
 from const.file
 ******/
int get_constants(char *name, struct cell *head, struct cell *tail){
 FILE *fp;
 int safety;

 init_p(head);
 init_p(tail);

 if ((fp = fopen( name, “r”)) == NULL){
 fprintf (stderr, “ERROR: can’t open %s.\n”, name);
 exit(1);
 }
 else{
 fscanf (fp, “ %d %d %d %d \n “, &head->start.x, &head->start.y, &head->goal.x, &head->goal.y);
 fscanf (fp, “ %d %d %d %d \n “, &tail->start.x, &tail->start.y, &tail->goal.x, &tail->goal.y);
 fscanf (fp, “ %d\n”, &safety);
 fclose (fp);

 if ((head->start.x < 1) || (head->start.x >= MAX_X) || (head->start.y < 1) || (head->start.y >= MAX_X)){
 fprintf(stderr, “ERROR: front start(%3d,%3d) out of bounds\n”,head->start.x,head->start.y);
 exit(1);
 }
 if ((head->goal.x < 1) || (head->goal.x >= MAX_X) || (head->goal.y < 1) || (head->goal.y >= MAX_Y)){
```

```
fprintf (stderr, "ERROR: front goal(%3d, %3d) out of bounds\n", head->goal.x, head->goal.y);
exit (1);
}
if ((tail->start.x < 1) || (tail->start.x >= MAX_X) || (tail->start.y < 1) || (tail->start.y >= MAX_X)){
fprintf(stderr, "ERROR: back start(%3d,%3d) out of bounds\n",tail->start.x,tail->start.y);
exit(1);
}
if ((tail->goal.x < 1) || (tail->goal.x >= MAX_X) || (tail->goal.y < 1) || (tail->goal.y >= MAX_Y)){
fprintf (stderr, "ERROR: back goal(%3d, %3d) out of bounds\n", tail->goal.x, tail->goal.y);
exit (1);
}
head->goal.value = GOAL_VAL;
tail->goal.value = GOAL_VAL;
}
return safety;
}


/*******
This function gets the location of all of the object from
from object.file
******/
int get_objects(int map[MAX_X][MAX_Y], char *name){
FILE *fp;
int num_objects;
int bottom, top, left, right;
int x,y,i;

if((fp = fopen (name, "r")) == NULL){
fprintf( stderr, "ERROR: can't open %s.\n", name);
exit(1);
}
else{
fscanf (fp, " %d \n ", &num_objects);
for (i = 1; i<= num_objects; i++){
fscanf( fp, " %d %d %d %d \n ", &bottom, &top, &left, &right);
for (y = bottom; y <= top; y++)
for (x = left; x <= right; x++){
if ( (x < 1) || (x > MAX_X) || (y < 1) || (y > MAX_Y))
fprintf (stderr,"WARNING: obstacle(%3d,%3d,%3d,%3d) out of bounds.\n",bottom,top,left,right);
else
map[x][y] = OBJECT_VAL;
}
}
fclose(fp);
}
return num_objects;
}

/*******
This function prints out the map for viewing
******/
void print_map(int map[MAX_X][MAX_Y]){
int x,y;

for( y = (MAX_Y - 1); y >= 0; y--){
for (x = 0; x < MAX_X; x++)
printf( "%4d ", map[x][y]);
printf( "\n");
}
printf( "\n");
}

/********
This function saves a map to a file
*******/
void save_map(int map[MAX_X][MAX_Y], char *name){
FILE *fp;
int x,y;

if ((fp = fopen( name,"w")) == NULL)
fprintf (stderr, "ERROR: can't open %s\n", name);
else{
for (y = (MAX_Y - 1); y >= 0; y--){
for (x = 0; x < MAX_X; x++)
fprintf (fp, "%4d ", map[x][y]);
fprintf (fp, "\n");
}
fprintf (fp, "\n");
fclose (fp);
}
}
```