|  |
| --- |
| Drexel University |
| Mobile Robots |
| Final Report |
|  |
|  |
| **3/17/2011** |

**Robert Borzellieri**

**Jasen Caroll**

**Swati Maini**

|  |
| --- |
|  |

**Abstract**

As students in a robotics course at Drexel University, a robot is needed to successfully move to a goal location from a starting location. This needs to be done while avoiding walls, other obstacles and most importantly not striking any moving doors that may be in the way. To begin this will be done using two different sets of programs. One implementation will use A\* while the other will use potential field and waypoint navigation. A few days before the demo it was concluded that there was not enough time to finish A\* and that too much needed to be changed. As a result Potential Field navigation was used for the demo. Though the performance was not what was hoped for, we did manage to make it to two out of the six goals on the easiest map. Considering the fact that a vital portion of the code had to be rewritten only minutes before the demo, things went rather smoothly.

Table of Contents

[Introduction: 4](#_Toc288155699)

[Problem Statement: 5](#_Toc288155700)

[Methodology: 5](#_Toc288155701)

[Simulation Results: 6](#_Toc288155702)

[Lessons Learned: 6](#_Toc288155703)

[Conclusions: 6](#_Toc288155704)

[Appendix: A1](#_Toc288155705)

# Introduction:

Unreal Tournament 2004 packaged with USARSim will be used to simulate a robot and understand its movements and programming algorithms. Unreal Tournament is simply used to provide physics and other real life properties to the robot realm. The USARSim package was developed for the simulation of urban search and rescue robots. For the scope of this class USARSim is simply used to understand how to move and control a robot in a virtual world. This method is much more practical than using real life robots in the sense that it is certainly cheaper and it is much safer in the fact that a virtual robot cannot be broken – granted the computer you are using doesn’t malfunction somehow. The type of robot that will be used is a P2AT which is a four wheel drive all-terrain robot. It is equipped with a camera, sonar sensors, a laser scanner, INS, odometry sensor and a RFID sensor.

The Final for this class will then be showing that one can successfully move the robot to a desired location using what has been learned during the course of this semester. The final will consists of three different maps. Each one will grow in difficulty, with the more difficult maps leading to more points toward your overall score. The grading breakdown is as follows…

* γ = 100%, is a reward for the percentage of goals reached
* g = # of goals reached and G = total # of goals
* σ = 20%, a penalty for the # of global sensors
* s = # of global sensors
* λ = 20%, penalty for local sensors
* l = # of local sensors
* µ = 12%, penalty for hitting doors
* m = # of doors hit and M = total # of doors
* ω = 20%, penalty for hitting obstacles
* o = # of obstacles hit and O = total # of obstacles
* φ = 10%, penalty for other failures such as flipped or immobilized robot

This grading scale helps enforce how local sensors are cheaper than global sensors by allowing us one “free” local sensor. Practice maps have been provided for testing of the robot as well as getting a general feel for how the final could look and will be graded.

# Problem Statement:

In this final project we were asked to design a program in MATLAB to navigate a P2AT robot through an obstacle map. This was all done in USAR Sim which is a program that utilizes Unreal Tournament 2004 for the virtual environment. We were given the option of choosing between an easy, medium or hard map for the final demonstration. Leniency was given if we chose to run the robot program on the harder maps. Easy maps had less geometrical complexity compared to the harder maps, but they guaranteed a higher chance of success. It was a higher probability that the P2AT robot would make it through the goals from the start point. Harder maps also had more obstacles such as doors (that opened and closed at various intervals), static barrels, and increased number of hallways.

# Methodology:

Two different methods were chosen with the best being the one that would be used on testing day. As for the first method, a version of A\* would be implemented that contained a version of Tangent BUG as an obstacle avoidance approach. The A\* was rather simplistic in that the robot was fed the back list from a slightly modified version of the A\* algorithm that was built for a homework assignment. The robot then had a pre-decided path to move to. However, A\* takes a .txt file of 0’s and 1’s and the robot was working in Unreal Tournament. To get the .txt file the UT map was opened as well as a blank .txt file. It was found that the number of tiles in the UT map denoted how big the map was, in that 1 tile was one square meter. Knowing this the .txt files were then built to be the same size, with a 0 representing a free square and a 1 representing a square with a wall in it. This proved to be a very unreliable method once the 3rd final map was introduced. Next the coordinates had to be changed from the A\* text file coordinates to the world coordinates of USARsim. Finally TBUG was implemented to avoid obstacles and a simple pause was introduced to avoid hitting doors after which the robot would just rush past the door once it opened.

As for the second approach, Potential Field Navigation, a rather complex and robust potential field was set up to command the robot. A potential field is a method of controlling a robot by setting the goal as a lower position than the starting point, the robot is then pulled to the goal by taking the gradient of this slope. Walls can be avoided by setting them to a relatively infinite value of the field, this would help the robot stay away from walls. As for the obstacles and doors, active line fitting was implemented. If the data returned a circle the object was known to be a barrel obstacle and added it to the potential field. If the data returned a line, and we didn’t know the line already existed, it was deemed a door and the robot was told to pause until the door opened.

It was determined that A\* was not robust enough to compete with the Potential Field algorithm and was not chosen for the demo. Not only did A\* rely too heavily on a pre-determined .txt file but the implementation of the obstacle and door avoidance was not very consistent in how the robot would communicate.

# Simulation Results:

The robot was run multiple times; the total number is unknown, most likely around eight. The first time the robot got to the first and second goals and then was immediately stuck in a local minimum at the second goal. After fixing this, the second, third and fourth time the robot was run for some reason headed in the wrong direction after the second goal, it turns out there was a positive number where a negative number should have been. However, this was unknown until the fourth run, there was however a problem with the obstacle that it was going to. It would stop just millimeters from the obstacle and it turned out that because the obstacle lay in a path that was deleted from the potential function due to the door opening, the obstacle was never recognized by the mapping function, thus the robot saw it as a door and waited for it to open. After heading in the correct direction for trial six the robot was then stuck at the door because half of it was at the wall. This proved to be the failing point of our robot and no progress was made in the following trials.

# Lessons Learned:

Potential functions can be very finicky. The data found from the mapping function should have then been compared to the data hard coded from the map, thus eliminating obstacles where there is actually an open door. This could be applied to our attempts at A\* as well considering it mapped already existing walls into new obstacles every time it looked around. Also, the frequency of this should have been turned down so that the robot was less likely to map the door into a new obstacle when it was closed. A lot of data was thrown out that could have potentially been used. Bounding boxes should be as small as possible for potential field navigation. As for A\* the map resolution in the .txt files should have been a lot higher than one block equaling one meter by one meter in the UT map.

# Conclusions:

It’s a common thought that if one more week was given the robots could have performed as desired. With that said the actual results of the robot were not significantly disappointing with the expectations the team had going into this demonstration. The robot using potential field navigation ended up hitting two out of six goals and most likely would have hit considerably more if it could have made it through the door.

# Appendix:

Potential Field Navigation

%% Potential Functions Final Project

% Requires Find\_obsticals.m and Key\_PotField\_RAB75.m

if exist('ROB','var')

shutdownRobot(ROB)

end

clear

clc

start = [-5 0];

goals = [5 5;0 0]'; %column-vector list of goals

turnConst = 80; %speed constant for rotational motion

moveConst = 40; %speed constant for linear motion

gI = 1; %The gI variable denotes the current goal index

%in the list of indeces. ie if you're moving toward

%goal 1, gI = 1 and the goal is goals(:,gI)

cd('/Users/robertborzellieri/Documents/MATLAB/Robotics');

ROB = initializeRobot('Borzellieri', 'P2AT', [start 1.8],[0 0 pi]);

pause(3)

insRdgs = getINSReadings(ROB);

robPose = [insRdgs.Position; insRdgs.Orientation];

theta = robPose(6);

pos = robPose(1:2);

pause(1)

List\_of\_circles=[];

List\_of\_Circles=[];

List\_of\_Circles\_REAL=[];

while(norm(goals(:,end) - pos) > .5 || gI ~= size(goals,2))

% Get Data -- Scans

[Obstical\_Cell] = Find\_Obsticals( ROB );

% Get Data -- Global Position

insRdgs = getINSReadings(ROB);

robPose = [insRdgs.Position; insRdgs.Orientation];

theta = robPose(6);

pos = robPose(1:2);

if(norm(pos - goals(:,gI)) < .5)

gI = gI + 1;

disp('I AM A MACHINE')

figure

end

% Rotation Matrix Global = R \* Local

R=[ cos(theta) sin(theta) 0 robPose(2);...

-sin(theta) cos(theta) 0 robPose(1);...

0 0 1 0;...

0 0 0 1];

% Builting Map for Potential Functions

for i = 1:(size(Obstical\_Cell,1)-1)

switch Obstical\_Cell{i+1,1}

case 'Line'

% Computes the List of Lines

case 'Circle'

% Computes the List of Circles

for MINIMIZE\_1=1

Circle\_Rotated=R\*[Obstical\_Cell{i+1,2}(:,1);...

Obstical\_Cell{i+1,2}(:,2);...

0;...

1];

List\_of\_circles{size(List\_of\_circles,1)+1,1}=[Circle\_Rotated(1)...

Circle\_Rotated(2) Obstical\_Cell{i+1,2}(:,3)];

% Fix List of Circles

for qwerty=1:size(List\_of\_circles,1)

List\_of\_circles1(qwerty,:)=List\_of\_circles{qwerty,1};

end

tol = .25; K = 1;

clusterInd = zeros(length(List\_of\_circles1),2);

clusterInd(1,1) = 1;

for asdf = 2:size(List\_of\_circles1,1)

d = sqrt((List\_of\_circles1(asdf,1)-List\_of\_circles1(asdf-1,1))^2 +...

(List\_of\_circles1(asdf,2)-List\_of\_circles1(asdf-1,2))^2);

if d > tol

% start new cluster

clusterInd(K,2) = asdf-1;

K = K + 1;

clusterInd(K,1) = asdf;

end

end

clusterInd = clusterInd(clusterInd(:,1)>0,:);

clusterInd(end,2) = size(List\_of\_circles1,1);

for j = 1:size(clusterInd,1)

if clusterInd(j,1)==clusterInd(j,2)

clusterInd(j,1)=0;

clusterInd(j,2)=0;

end

end

clusterInd = clusterInd(clusterInd(:,1)>0,:);

K=size(clusterInd,1);

if K==0

else

for k=1:K

List\_of\_Circles\_REAL{k,1}=List\_of\_circles1(clusterInd(k,1),:);

end

end

clear Circle\_Rotated

end

clear Circle\_Rotated

otherwise

disp('error')

end

end

clear Obstical\_Cell

% Potential Functions

% desired heading

if pos(1)/goals(:,gI) >= 0

% disp('goal')

desvH = Key\_PotField\_With\_List\_RAB75(pos,goals(:,gI),List\_of\_Lines,List\_of\_Circles\_REAL);

else

% disp('door')

desvH = Key\_PotField\_With\_List\_RAB75(pos,[5\*goals(1,gI)/abs(goals(1,gI)) 0]',List\_of\_Lines,List\_of\_Circles\_REAL);

end

for zxcv=1:2

if ~isfinite(desvH(zxcv))

desvH(zxcv)=10^5;

end

end

desvH = desvH./norm(desvH);

% actual heading

vH = [cos(theta);sin(theta)];

% Calculate forward and rotational velocity.

omega = turnConst\*desvH'\*[0 -1;1 0]\*vH;

V = moveConst\*desvH'\*vH \* (abs(desvH'\*vH) > .7);

% Stop for Door.

lrf = getLaserSensorReadings(ROB);

if min(min(lrf.Scans))<0.5

omega = 0;

V = 0;

end

% Send the command to the robot

sendDriveCommand(ROB,V+[omega,-omega]);

% Update position information.

INS = getINSReadings(ROB);

pos = INS.Position(1:2);

theta = INS.Orientation(3);

end

disp('I AM A MACHINE')

figure

sendDriveCommand(ROB,[0,0]);

A\* with active mapping

%% AStar for Final Project

if exist('rob', 'var')

shutdownRobot(rob);

end

clear

clc

map = load('FinalMap2.txt');

nLines = size(map,1);

center = nLines/2+0.5;

start=[5 5];

astart = [center+(start(1)) center-(start(2))];

qgoal = [-5 -5];

agoal = [center+qgoal(1) center-qgoal(2)];

turnConst = 80; %movement constants

moveConst = 40;

% wall-following numbers - used for obstacle avoidance

follow = 0; %bug knows when it is/is not following an obstacle

perim = [0 0]; %first is total perimiter; seconcd is perimiter since last closest point to the goal

qhit = [0 0]'; %so the bug knows when it has circled the obstacle

qleave = [0 0]';% closest point to goal around the obstacle

hug = 0.75; %stay about 1m away from wall

rightleft = 1; %1=hang a left at an obstacle; -1=turn right

closest = 0; %stores closest distance to goal around the perimeter of an obstacle

pts = zeros(2,0);

m = .75;

% initialize robot

rob = initializeRobot('rob','P2AT',[start 1.8],[0 0 0]);

pause(2);

% start the loop

INS = getINSReadings(rob);

robPose = [INS.Position; INS.Orientation];

theta = robPose(6);

pos = INS.Position(1:2)';

orient = INS.Orientation(3);

lastpos = pos;

v = [0 0];

vall = AStar\_final(astart, agoal, map);

flag2 = 0;

i = 1;

%%

while i < size(vall,1)

[Obstical\_Cell] = Find\_Obsticals( rob );

pilar\_added = 0;

for w = 1:(size(Obstical\_Cell,1)-1)

switch Obstical\_Cell{w+1,1}

case 'Line'

% dont care

case 'Circle'

pilar\_added = pilar\_added + 1;

%changing the map is here

x1 = Obstical\_Cell{w+1,2}(:,1);

y1 = Obstical\_Cell{w+1,2}(:,2);

R = [cos(theta) sin(theta) 0 robPose(2);

-sin(theta) cos(theta) 0 robPose(1);

0 0 1 0

0 0 0 1];

Circle\_Rotated=R\*[x1 y1 0 1]';

radius = Obstical\_Cell{w+1,2}(:,3);

radius = ceil(radius);

x1 = round(x1);

y1 = round(y1);

z = [center+(x1) center-(y1)];

for k = -radius:radius

for j = -radius:radius

map(z(1)+k,z(2)+j) = 1;

end

end

otherwise

disp('error')

end

end

if pilar\_added>0

vall = AStar\_final(astart, agoal, map);

pilar\_added = 0;

pause(1)

end

lrf = getLaserSensorReadings(rob);

data = lrf.Scans;

x = vall(i,1);

y = vall(i,2);

q = zeros(1,2);

q(1) = x-center;

q(2) = center-y;

if norm(pos-q) < 0.1

i = i+1;

elseif data(90) > hug

v = (q-pos)/norm(q-pos);

end

vH = [cos(orient);sin(orient)];

omega = turnConst\*v\*[0 -1;1 0]\*vH;

V = moveConst\*v\*vH;

sendDriveCommand(rob,V+[omega,-omega]);

lastpos = pos;

INS = getINSReadings(rob);

pos = INS.Position(1:2)';

orient = INS.Orientation(3);

end

disp('You have reached your goal!')

shutdownRobot(rob);

Find\_Obstacles

function [Obstical\_Cell] = Find\_Obsticals( ROB )

% Find\_Obsticals Takes in Robot, Exports information about local world.

lrf = getLaserSensorReadings(ROB);

data = lrf.Scans;

angle=[-179:2:179]'+90; DUMMY2=1;

% Eliminate Out of Range Data

for DUMMY=1:numel(data)

if data(DUMMY,1)>=2.8

else

NewData(DUMMY2,1)=data(DUMMY,1);

NewData(DUMMY2,2)=(pi/180).\*angle(DUMMY,1);

DUMMY2=DUMMY2+1;

end

end

if exist('NewData','var')

else

NewData=[0 0];

end

% Performs Line and Circle Fitting and saves data to output cell

for MINIMIZE=1

for CLUSTER=1

Xdata = NewData(:,1).\*cos(NewData(:,2));

Ydata = NewData(:,1).\*sin(NewData(:,2));

% We will first separate the data set into M different clusters. We will

% then try to fit circles to each individual cluster

tol = .25; % if two consecutive points are more than 100 mm (10 cm) apart, start a new cluster

K = 1; % initialize the number of the clusters

clusterInd = zeros(length(Xdata),2); % initialize the array that will contain the start and end indices for each cluster

% the indices are to index into Xdata and Ydata

% since we don't know how large M will get, we will

% initialize clusterInd to have the same rows as Xdata

clusterInd(1,1) = 1; % setting the start index for the first cluster to be 1

for i = 2:size(NewData,1)

d = sqrt((Xdata(i)-Xdata(i-1))^2 + (Ydata(i)-Ydata(i-1))^2);

if d > tol

% start new cluster

clusterInd(K,2) = i-1;

K = K + 1;

clusterInd(K,1) = i;

end

end

clusterInd = clusterInd(clusterInd(:,1)>0,:); % this clips off all the excess rows that only contains 0s

clusterInd(end,2) = size(NewData,1); % this sets the last index to be the last element in data

for j = 1:size(clusterInd,1)

if clusterInd(j,1)==clusterInd(j,2)

clusterInd(j,1)=0;

clusterInd(j,2)=0;

end

end

clusterInd = clusterInd(clusterInd(:,1)>0,:); % this clips off all the excess rows that only contains 0s

K=size(clusterInd,1);

end

for CIRCLE\_FIT=1

% Now that we have the clusters, let's fit the circles

circParam = zeros(K,3); % initialize the vector to store the Xc, Yc, and R for each circle

lse\_Circle = zeros(K,1);% initialize the vector to store the error of the K line fits

% main line fitting code begins here

for i = 1:K

xBar = sum(Xdata(clusterInd(i,1):clusterInd(i,2)))/(clusterInd(i,2)-clusterInd(i,1)+1);

yBar = sum(Ydata(clusterInd(i,1):clusterInd(i,2)))/(clusterInd(i,2)-clusterInd(i,1)+1);

u = Xdata(clusterInd(i,1):clusterInd(i,2)) - xBar;

v = Ydata(clusterInd(i,1):clusterInd(i,2)) - yBar;

Suu = sum(u.\*u);

Svv = sum(v.\*v);

Suv = sum(u.\*v);

Suuu = sum(u.\*u.\*u);

Svvv = sum(v.\*v.\*v);

Suvv = sum(u.\*v.\*v);

Svuu = sum(v.\*u.\*u);

S = [Suu Suv; Suv Svv];

b = 0.5\*[Suuu + Suvv; Svvv + Svuu];

circParam(i,1:2) = S\b;

alpha = circParam(i,1)^2 + circParam(i,2)^2 + (Suu + Svv)/(clusterInd(i,2)-clusterInd(i,1)+1);

circParam(i,3) = sqrt(alpha);

% converts uc vc back into xc, yc

circParam(i,1) = circParam(i,1) + xBar;

circParam(i,2) = circParam(i,2) + yBar;

Circle\_Err=eig(S);

lse\_Circle(i)=Circle\_Err(2);

end

end

for LINE\_FIT=1

% Now that we have the clusters, let's fit the lines

lineCoeff = zeros(K,3); % initialize the vector to store the coefficients for the K line equations

lse\_Line = zeros(K,1); % initialize the vector to store the error of the K line fits

% main line fitting code begins here

for i = 1:K

M = [Xdata(clusterInd(i,1):clusterInd(i,2)), Ydata(clusterInd(i,1):clusterInd(i,2)), ones(clusterInd(i,2)-clusterInd(i,1)+1,1)];

[U, S, V] = svd(M);

lineCoeff(i,:) = V(:,end)';

[a b]=size(S);

if a < 3

Obstical\_Cell{i+1,1}='Not Line';

else

lse\_Line(i) = S(3,3);

end

end

end

for CELL\_SETUP=1

Obstical\_Cell = cell(K+1, 6);

Obstical\_Cell{1,1}='Line or Circle';

Obstical\_Cell{1,2}='Circle Parameters';

Obstical\_Cell{1,3}='Circle Error';

Obstical\_Cell{1,4}='Line Parameters';

Obstical\_Cell{1,5}='Line Error';

Obstical\_Cell{1,6}='Real Data';

for i=1:K

Obstical\_Cell{i+1,2}=circParam(i,:);

Obstical\_Cell{i+1,3}=lse\_Circle(i);

Obstical\_Cell{i+1,4}=lineCoeff(i,:);

Obstical\_Cell{i+1,5}=lse\_Line(i);

Obstical\_Cell{i+1,6}=NewData(clusterInd(i,1):clusterInd(i,2),:);

end

end

end

% Determine Circle or Line

for i = 1:K

if Obstical\_Cell{i+1,2}(:,3)<=20

Obstical\_Cell{i+1,1}='Circle';

end

if Obstical\_Cell{i+1,2}(:,3)>=1

Obstical\_Cell{i+1,1}='Line';

end

if abs(Obstical\_Cell{i+1,2}(:,1))>=1

Obstical\_Cell{i+1,1}='Line';

end

if Obstical\_Cell{i+1,5}<=0.015

Obstical\_Cell{i+1,1}='Line';

end

if Obstical\_Cell{i+1,3} > 10

Obstical\_Cell{i+1,1}='Line';

end

end

end

Modified version of the Key for PotField

%% Function A4PotField\_user

% Function takes in arguments pos, and goal, being position and goal

% point, respectively. It returns a column vector denoting desired heading

% of the robot.

function desvH = Key\_PotField\_With\_List\_RAB75(pos,goal,List\_of\_Lines,List\_of\_Circles,bob)

% If variable bob exists, do a 3D plot of the nav field

if (nargin == 5)

figure(2)

Xpts = linspace(-10,10,100);

Ypts = linspace(-10,10,100);

phiVis = zeros(100,100);

for iX = 1:length(Xpts)

for iY = 1:length(Ypts)

phiVis(iX,iY) = getPotential([Xpts(iX);Ypts(iY)],goal,List\_of\_Lines,List\_of\_Circles);

end

end

surf(Ypts,Xpts,real(phiVis'./norm(phiVis)));%,'edgealpha',0)

shading interp

end

% take a discrete gradient using a helper function to give numeric values

% for the potential field.

val = getPotential(pos,goal,List\_of\_Lines,List\_of\_Circles);

valdx = getPotential(pos + [.01;0],goal,List\_of\_Lines,List\_of\_Circles);

valdy = getPotential(pos + [0;.01],goal,List\_of\_Lines,List\_of\_Circles);

desvH = -([valdx;valdy] - val)/.01;

% desvH=val;

end

%% getPotential

% Takes in position and goal points, and returns a value denoting the

% magnitude of the potential field at that point. val is normailized and

% will always be between 0 and 1.

function val = getPotential(x,goal,List\_of\_Lines,List\_of\_Circles)

% Repulsive Term for Lines

for K=1:size(List\_of\_Lines,1)

x\_min=List\_of\_Lines{K,2}(1,1);

x\_max=List\_of\_Lines{K,2}(end,1);

y\_min=List\_of\_Lines{K,2}(1,2);

y\_max=List\_of\_Lines{K,2}(end,2);

obst=List\_of\_Lines{K,1};

if x(1) >= x\_min && x(1) <= x\_max

B\_s(K) = sum(repmat(x,1,size(obst,1)).\*obst(:,1:2)',1) - obst(3);

else

if x(2) >= y\_min && x(2) <= y\_max

B\_s(K) = sum(repmat(x,1,size(obst,1)).\*obst(:,1:2)',1) - obst(3);

else

B\_s(K)=1;

end

end

end

B\_Lines = prod(B\_s)^(1/4);

% Repulsive Term for Circles

for I=1:size(List\_of\_Circles,1)

obst(I,:)=List\_of\_Circles{I,1}';

end

obst=obst';

B\_Circles = prod(abs(sum((repmat(x,1,size(obst,2)) - obst(1:2,:)).^2,1).^.5 - obst(3,:)))^.125;

% attractive term

G = norm(x - goal);

% normalized value for potential

val = real((G)/(B\_Lines.\*B\_Circles.\*0.5 + G\*0.5));

end

A\* without mapping

%% Function AStar\_jcc329()

% map- matrix full of zeros denoting free space and ones denoting

% obstacles.

% start- start point (row vector)

% qgoal- end point (row vector)

% path- path from start to goal

% Takes in a map, start point, and goal point and returns a path from the

% start to the goal via the Wavefront planner algorithm.

function path = AStar\_final(start, qgoal,map)

% The hueristic function will be a function for ease of calculation.

% Pass it a point and it returns the heuristic at that point. (for 8-point

% adjancy)

heur = @(q) abs(qgoal(1) - q(1))+abs(qgoal(2) - q(2));

% List of neighbor offsets. Add a row to a point to get the point's

% neighbor.

nei = [-1 0;0 -1;0 1;1 0]; %4-point

%Open list stores [pointX, pointY, (distance from start), (hueristic to goal + distance from start)]

open = [start(1) start(2) 0 heur([start(1) start(2)])];

%Negate the map

map = -map;

%size of the map for reference in back function

[mapn mapm] = size(map);

% The back function will be a cell. Each entry in the cell corresponds to

% an entry in the map. Each entry will store either nothing (ie the point

% is not in the closed list) or the previous point with the distance from

% the start.

% Closed list isn't actually used- it's easier to test the back function

% for existence. ie if isempty(back{1,1}) is true, that point has never

% been expanded.

% If you do not know how to use cells, you can either consult MATLAB help

% or substitute for your own solution. Essentially, each element in a cell

% is its own matrix.

back = cell(mapn,mapm);

back{start(1),start(2)} = -1; %flag the path start

while(~isempty(open)) %main loop

%sort the open list in accordance with the actual distance to point

%plus the heuristic of the point (column 4 of the list)

open = sortrows(open,4);

%There is a possibilty of repeats in the open list. Since the list is

%sorted, we can remove the lesser-performing repeats with these lines.

[~, I, ~] = unique(open(:,1:2),'rows','first');

I = sort(I);

open = open(I,:); %Now the open list is sorted and unique.

imagesc(map) % use these two lines to view the progression of the

drawnow; % algorithm

%% The rest of the loop is on you.

% Pop the first point off the open list

[p, open] = popOffStack(open);

x = [p(1) p(2)];

%Check to see if point x is the goal. If so, break.

if x == qgoal

break

end

% Assign this point in the map as its distance from the start point.

map(x(1),x(2)) = p(3);

% Expand current point x by adding it's valid neighbors to the open

% list and back path. So, for each neighbor,

for i=1:4

n = x+nei(i,:);

% Check to make sure the neighbor exists and is not an obstacle

if ~(all(n >= 1) && n(1) <= mapn && n(2) <= mapm)

continue;

end

if map(n(1),n(2)) == -1

continue;

end

% Check to see if this neighbor has been added to the closed list.

% (Does this point have an entry in the back path?)

% If the point is not in the closed list, add it to Both the open

% list and the back path (which doubles as the closed list.

if isempty(back{n(1),n(2)})

dist = p(3)+1; % new distance

h = heur([n(1) n(2)]); % cost

temp = [n(1) n(2) dist dist+h]; % new stack

open = addToStack(temp, open); % add new stack

back{n(1),n(2)} = x; % add to back

end

end

end

%% We're done the algorithm. Now we back-search for the start from the

% goal.

if(isempty(back{qgoal(1),qgoal(2)})) %the goal has no back path; FAIL

disp('Unreachable');

path = -1;

else

%The goal has a backpath; therefore, we can get to the start from the

%goal.

% Check the goal and add it's back path to a temporary variable. Repeat

% process for that point and consecutive points until the start is

% found.

tmp = qgoal;

while(back{tmp(end,1),tmp(end,2)} ~= -1)

tmp(end+1,:) = back{tmp(end,1),tmp(end,2)}([1,2]);

end

%path is the tmp variable backwards

path = tmp(end:-1:1,:);

%Plot the path.

imagesc(map); colormap jet;

hold on;

plot(path(:,2),path(:,1),'k','LineWidth',3);

hold off;

end

end