ROB521: Mobile Robotics and Perception
Assignment #1: PRM for Maze Solving

Question 1: Construct a PRM connecting start and finish

The first question consists of constructing a graph that connects the start and finish nodes using 500 random samples. The MinDist2Edges function was used to determine the minimum distances between the samples and walls of the maze. Only those that were more than 0.1 units away from the walls were defined as valid milestones, which overall eliminated samples that weren't in feasible space.

The next step was to loop over each of the milestones to obtain its nearest neighbours. The pdist2 function was used to obtain 8 nearest neighbours, a number that was determined after a number of trials. For each of these 8 neighbours, the Check Collision function was used to check if the edge between the milestone and its neighbour is crossing any of the maze's walls. If not crossing the walls, the edge was valid.

Figure 1 shows the PRM graph that was created using the described process on a 5 by 7 maze.

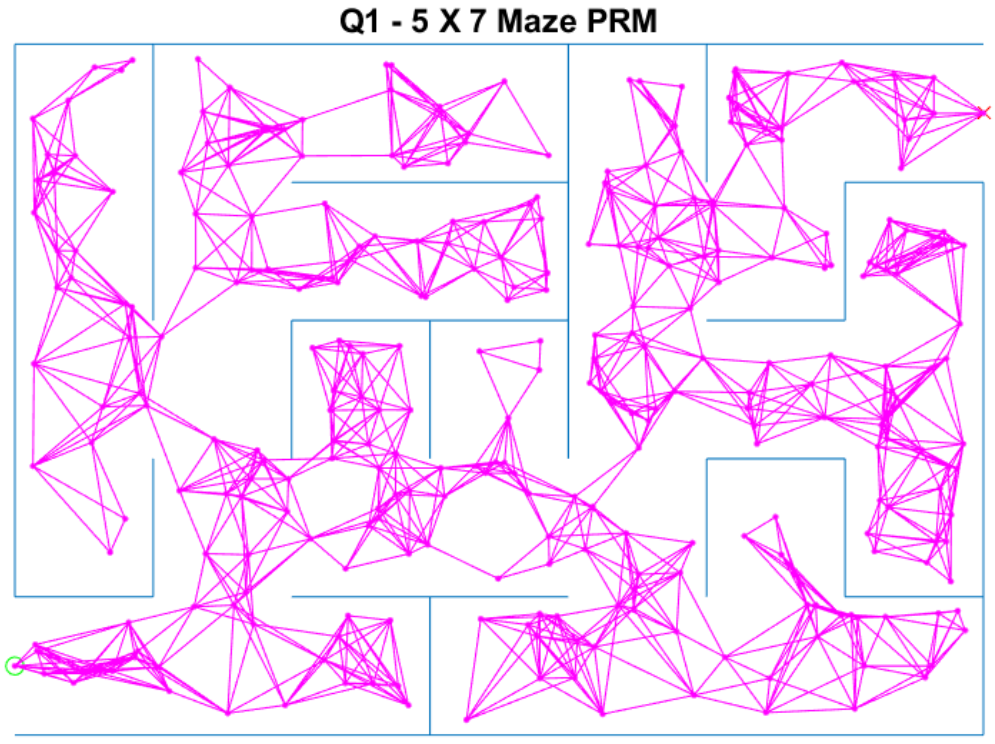

Figure 1: PRM graph on a 5 by 7 maze

Observations

We can see that the graph reaches all the paths in the maze quite well. However, it's visible that there are clusters of edges in various locations. If the randomly sampled milestones were close to each other, they formed edges due to the nearest neighbour's connection strategy. However, not

all of them are necessary, so it is possible to eliminate some edges that intersect such that the search algorithm in Question 2 has fewer edges to loop over. We could also reduce the number of nearest neighbours being chosen to eliminate clusters of edges. However, this sometimes resulted in a disconnected graph, which meant there was no path from start to finish. The disconnected graphs were due to not enough neighbours being identified for more edges to form between distant milestones, along with edges being eliminated if they were intersecting with walls.

Question 2: Construct a PRM connecting start and finish

Building on top question 1, the goal of question 2 was to implement an optimal graph search algorithm from scratch to find the shortest path from start to finish.

In particular, the A* algorithm was used and the shortest path for the PRM graph from Figure 1 is displayed in Figure 2.

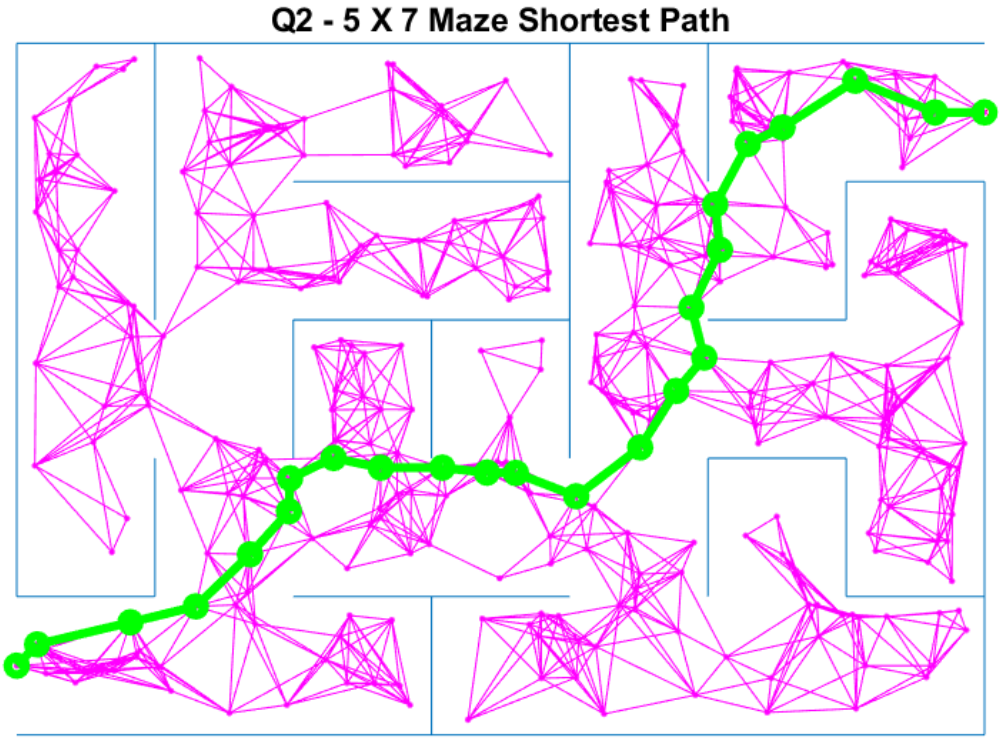

Figure 2: Shortest path obtained using the A algorithm on the PRM graph from Figure 1*

General steps taken for pathfinding (A* algorithm and backtracking):

- At the start of the algorithms, 2 matrices are defined: (1) an empty closed states matrix and (2) an open states matrix that holds the start milestone. Both these matrices will

contain the milestone, g (cost-to-come), f (the sum of the cost-to-come and the h, heuristic cost-to-go), and the parent of the milestone.

- The A* algorithm begins with a while loop that runs until there are no more open states left. Inside the while loop, the milestone with lowest total f cost from the open states is set as current.
- If current is the finish milestone, it can be added to the closed states and the search is complete. If it is not the finish milestone, current is removed from open states and added to the closed states.
- There is a for loop that iterates over all the edges in the graph to get the current's neighbours.
- Once a neighbour is found, if it has already been explored and is in closed states, the next edge is checked and so on.
- If the neighbour was not in the closed states, it will need to be added to or updated in the open states. To do so, the g (cost-to-come), h (heuristic cost-to-go), and f (the sum of g and h) is calculated for the neighbour.
- If the neighbour was already in open states, it was checked if the stored cost-to-come is higher or lower than the new calculated cost-to-come. If lower, then the process continued to the next neighbours. If higher, the costs and parent were updated in open states.
- If the neighbour was not in open states, it was added to open states with the appropriate information
- The path planning finishes by backtracking to obtain the list of milestones in row index sequence that form the shortest path from start to finish

Observations

While designing the algorithm, it was important to ensure sufficient information was stored in the closed and open states for the shortest path to be found. Both these matrices contained the milestone, g (cost-to-come), f (the sum of the cost-to-come and the h, heuristic cost-to-go), and the parent of the milestone for backtracking.

Relating back to the observations in Question 1, when there were disconnected graphs, no path could be found with the algorithm; therefore, the number of neighbours parameter was modified. Using random sampling and the nearest neighbours edge connection approach, it was clearly visible the graph was able to explore all parts of the maze quite well and this helped ensure that the shortest path could be found between the start and finish milestones.

After some experiments, the appropriate combination of cost-to-go and heuristic cost was determined. The cost–to-come was calculated as the cumulated Euclidean distance from the start to any milestone and the heuristic cost-to-go was calculated as the Manhattan distance from a

milestone to the finish milestone. Since the start milestone was on the bottom left corner and the finish milestone was in the top right corner, exploring milestones with a lower Manhattan cost (and a lower cost-to-go) was important to quickly find the shortest path.

Question 3: Construct a PRM connecting start and finish

The goal of Question 3 was to modify the milestone generation, edge connection, collision detection, and/or shortest path algorithm to reduce runtime. The modified code was to find the shortest path from start to the goal in under 20 seconds for mazes larger than 40 by 40.

Figures 3 and 4 show examples of 41 by 41 sized mazes being solved in under 20 seconds on my computer. Figure 5 shows a 45 by 45 maze solved in under 20 seconds.

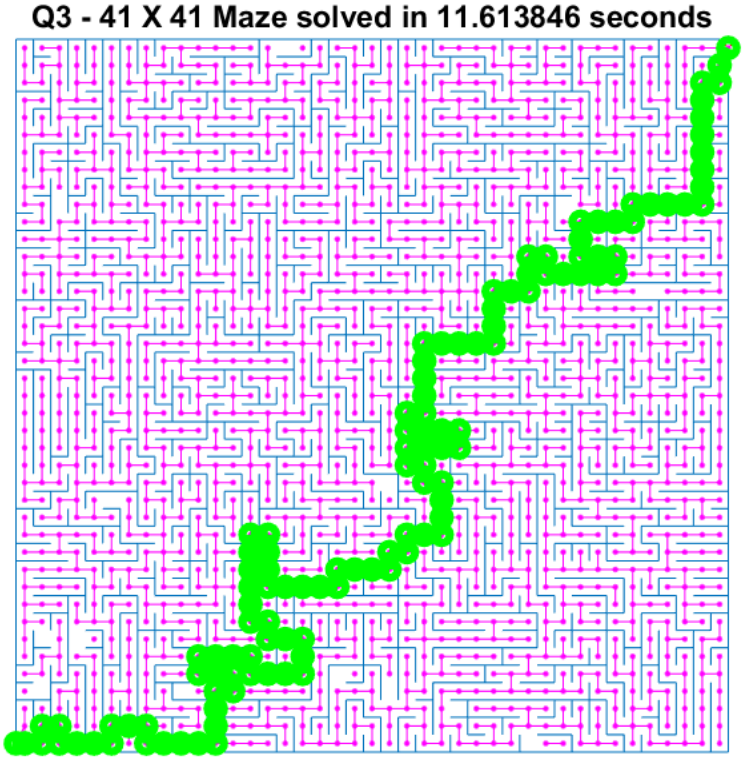

Figure 3: 41 by 41 maze solved in 11.6 seconds

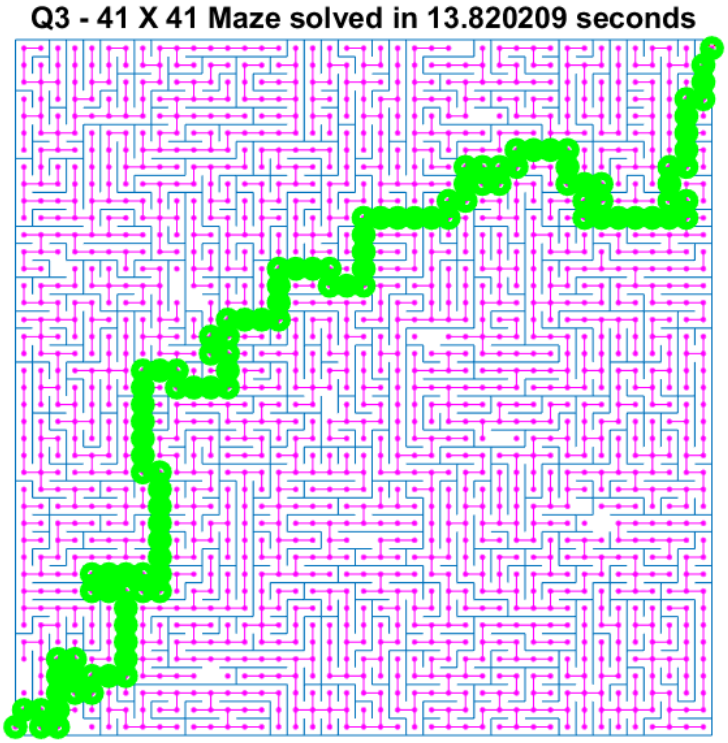

Figure 4: 41 by 41 maze solved in 13.8 seconds

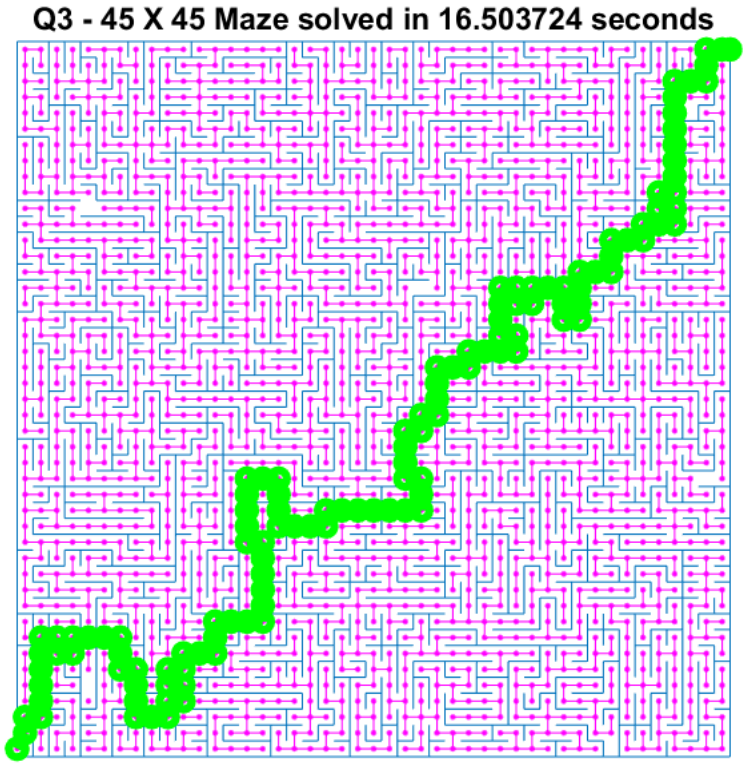

Figure 5: 45 by 45 maze solved in 16.5 seconds

Observations

To achieve the required specifications, instead of sampling randomly with points defined using decimal places, a grid of integer resolution was created to sample from. This ensured that the milestones were placed right in the middle of corridors in the maze. By sampling more milestones in correspondence with the size of the maze, it creates a graph that covers almost every path in the maze. An approach as such was used to ensure that every path was covered in the large maze since the shortest path can be anywhere. Moreover, to reduce runtime, duplicate edges were removed. Since the milestones are so close to each other, the number of neighbours to attempt for edge connection was also reduced and prevented duplicates.

Within the A* algorithm, modifications were made to ensure that various loops were not unnecessarily running after their purpose was achieved to further decrease runtime.

After running multiple times, it was observed that based on the shape of the path, the runtimes would vary by small amounts depending on the number of milestones explored. In the future, other techniques like lazy collision checking could be explored to further reduce runtime.

```
% ======
% ROB521_assignment1.m
% ======
%
% This assignment will introduce you to the idea of motion planning for
% holonomic robots that can move in any direction and change direction of
% motion instantaneously.  Although unrealistic, it can work quite well for
% complex large scale planning.  You will generate mazes to plan through
% and employ the PRM algorithm presented in lecture as well as any
% variations you can invent in the later sections.
%
% There are three questions to complete (5 marks each):
%
%    Question 1: implement the PRM algorithm to construct a graph
%    connecting start to finish nodes.
%    Question 2: find the shortest path over the graph by implementing the
%    Dijkstra's or A* algorithm.
%    Question 3: identify sampling, connection or collision checking
%    strategies that can reduce runtime for mazes.
%
% Fill in the required sections of this script with your code, run it to
% generate the requested plots, then paste the plots into a short report
% that includes a few comments about what you've observed.  Append your
% version of this script to the report.  Hand in the report as a PDF file.
%
% requires: basic Matlab,
%
% S L Waslander, January 2022
%
clear; close all; clc;

% set random seed for repeatability if desired
% rng(1);

% ==========================
% Maze Generation
% ==========================
%
% The maze function returns a map object with all of the edges in the maze.
% Each row of the map structure draws a single line of the maze.  The
% function returns the lines with coordinates [x1 y1 x2 y2].
% Bottom left corner of maze is [0.5 0.5],
% Top right corner is [col+0.5 row+0.5]
%

row = 5; % Maze rows
col = 7; % Maze columns
map = maze(row,col); % Creates the maze
start = [0.5, 1.0]; % Start at the bottom left
finish = [col+0.5, row]; % Finish at the top right
```

```
h = figure(1);clf; hold on;
plot(start(1), start(2),'go')
plot(finish(1), finish(2),'rx')
show_maze(map,row,col,h); % Draws the maze
drawnow;

% ======================================================
% Question 1: construct a PRM connecting start and finish
% ======================================================
%
% Using 500 samples, construct a PRM graph whose milestones stay at least
% 0.1 units away from all walls, using the MinDist2Edges function provided for
% collision detection.  Use a nearest neighbour connection strategy and the
% CheckCollision function provided for collision checking, and find an
% appropriate number of connections to ensure a connection from start to
% finish with high probability.


% variables to store PRM components
nS = 500;  % number of samples to try for milestone creation
milestones = [start; finish];  % each row is a point [x y] in feasible space
edges = [];  % each row is should be an edge of the form [x1 y1 x2 y2]

disp("Time to create PRM graph")
tic;
% ------insert your PRM generation code here-------

% Obtain 500 random sample points within the grid range specified
candidateMileStones = 0.5 + (col+0.5 - 0.5)*rand(nS, 2);

% Obtain the minimum distances to the edges for the points and only keep
% the points that have a distance of at least 0.1 units from all the walls
% as milestones. Add those valid milestones to the matrix containing all
% the milestones in feasible space.
dist = MinDist2Edges(candidateMileStones, map)';
idx = find(dist >= 0.1);
validMileStones = candidateMileStones(idx, :);
milestones = [milestones; validMileStones];

% Loop through each of the milestones to obtain its nearest neighbours
for i=1:length(milestones)
    [D,I] = pdist2(milestones, milestones(i, :), 'euclidean','Smallest', 9);

    % Loop through each nearest neighbour and check if the edge between the
    % milestone and its neighbour is crossing a wall. If not, then add to
    % the matrix of edges.
    for j = 2:length(I)
        [inCollision, edge] = CheckCollision(milestones(i, :), milestones(I(j), :), ↙
map);
```

```
        if inCollision == 0
            edges = [edges; milestones(i, :) milestones(I(j), :)];
        end
    end
end


% ------end of your PRM generation code -------
toc;

figure(1);
plot(milestones(:,1),milestones(:,2),'m.');
if (~isempty(edges))
    line(edges(:,1:2:3)', edges(:,2:2:4)','Color','magenta') % line uses [x1 x2 y1 y2]
end
str = sprintf('Q1 - %d X %d Maze PRM', row, col);
title(str);
drawnow;

print -dpng assignment1_q1.png


% =================================================================
% Question 2: Find the shortest path over the PRM graph
% =================================================================
%
% Using an optimal graph search method (Dijkstra's or A*) , find the
% shortest path across the graph generated.  Please code your own
% implementation instead of using any built in functions.

disp('Time to find shortest path');
tic;

% Variable to store shortest path
spath = []; % shortest path, stored as a milestone row index sequence


% ------insert your shortest path finding algorithm here-------
% Matrix to hold the closed states from the Astar algorithm
closedStates = [];

% Matrix to hold the open states from the Astar algorithm
% First 2 columns hold the milestone, next 2 columns hold g (cost-to-come)
% and f (which is the sum of the cost-to-come and the heuristic cost-to-go)
% Last 2 columns hold the parent of the milestone
openStates = [start 0 norm(start - finish) 0 0];

% Set variable to not solved
solved = 0;
```

```
% Keep looping until there are no more open states left
while(~isempty(openStates))

    % Find the open state with the lowest total f cost and set as current
    [P, I] = min(openStates(:, 4));
    current = openStates(I,:);

    % Check if current is the finish milestone. If so, add current to the
    % closed states and break out of the loop
    if isequal(current(1:2), finish)
        closedStates = [closedStates; current];
        solved = 1;
        break;
    end

    % Remove current from open states and add to closed states
    openStates(I, :) = [];
    closedStates = [closedStates; current];

    % Loop over all the edges to get the current milestone's neighbours
    for m = 1:size(edges, 1)
        % Determining which edges are corresponding to the current and
        % which milestone from the edges is the neighbour
        if isequal(edges(m, 1:2), current(1:2)) || isequal(edges(m, 3:4), current(1:2))
            n = [];
            if isequal(edges(m, 1:2), current(1:2))
                n = edges(m, 3:4);
            else
                n = edges(m, 1:2);
            end
            if isequal(n, current(1:2))
                continue
            end

            % If neighbour already in closed states, break out of for loop
            % and continue to the checking next edges
            skipClosed = 0;
            for c = 1:size(closedStates, 1)
                if isequal(closedStates(c, 1:2), n)
                    skipClosed = 1;
                    break;
                end
            end

            % If neighbour not in the closed states, then it will need to
            % be added to the open states or updated in the open states
            if(skipClosed == 0)
                % Define the g (cost-to-come), h (heuristic cost-to-go),
                % and f (the sum of g and h)
                cost_to_come = current(3) + norm(current(1:2) - n);
```

```
                heuristic_cost_to_go = abs(n(1)-finish(1)) + abs(n(2)-finish(2));
                F = cost_to_come + heuristic_cost_to_go;

                % Check if neighbour is in open states already
                skip = 0;
                if(~isempty(openStates))
                    for o = 1:size(openStates, 1)
                        if isequal(openStates(o, 1:2), n)
                            skip = 1;
                            % If already in open states, check if the
                            % stored cost-to-come is higher or lower than
                            % the new calculated cost-to-come. If lower,
                            % then continue. If higher, update the costs in
                            % open states
                            if openStates(o, 3) < cost_to_come
                                continue
                            else
                                openStates(o, 3:6) = [cost_to_come F current(1:2)];
                            end
                            break;
                        end
                    end
                end
                % If neighbour not in open states, add neighbour to open
                % states with appropriate information
                if skip == 0
                    openStates = [openStates; n cost_to_come F current(1:2)];
                end
            end
        end
    end
end


spath2 = [];

if solved == 1
    % Back track to obtain the list of milstones that form the shortest path
    % from start to finish
    spath2 = [closedStates(size(closedStates, 1), :)];
    while ~isequal(spath2(1, 1:2), start)
        parent = spath2(1, 5:6);
        for c = 1:size(closedStates, 1)
            if isequal(closedStates(c, 1:2), parent)
                spath2 = [closedStates(c, :); spath2];
            end
        end
    end
    % Obtain the milestone row index sequence
    for s = 1:size(spath2, 1)
```

```
        for m = 1:size(milestones, 1)
            if isequal(milestones(m, :), spath2(s, 1:2))
                spath2(s, 1:2);
                spath = [spath; m];
            end
        end
    end
end

% ------end of shortest path finding algorithm-------
toc;

% plot the shortest path
figure(1);
for i=1:length(spath)-1
    plot(milestones(spath(i:i+1),1),milestones(spath(i:i+1),2),  'go-', 'LineWidth',3);
end
str = sprintf('Q2 - %d X %d Maze Shortest Path', row, col);
title(str);
drawnow;

print -dpng assingment1_q2.png


% ========================================================================
% Question 3: find a faster way
% ========================================================================
%
% Modify your milestone generation, edge connection, collision detection
% and/or shortest path methods to reduce runtime.  What is the largest maze
% for which you can find a shortest path from start to goal in under 20
% seconds on your computer? (Anything larger than 40x40 will suffice for
% full marks)


row = 41;
col = 41;
map = maze(row,col);
start = [0.5, 1.0];
finish = [col+0.5, row];
milestones = [start; finish];  % each row is a point [x y] in feasible space
edges = [];  % each row is should be an edge of the form [x1 y1 x2 y2]

h = figure(2);clf; hold on;
plot(start(1), start(2),'go')
plot(finish(1), finish(2),'rx')
show_maze(map,row,col,h); % Draws the maze
drawnow;

fprintf("Attempting large %d X %d maze... \n", row, col);
```

```
tic;
% ------insert your optimized algorithm here------

nS = 1750;  % number of samples to try for milestone creation

% Implement a grid with integer values instead of continous milestones
x = 0:1:(col+0.5);
y = 0:1:(row+0.5);
[X,Y] = meshgrid(x,y);
points = [X(:), Y(:)];

% Obtain random sample points within the grid specified. Add milestones to
% the matrix containing all the milestones in feasible space.
index = randsample(length(points), nS);
candidateMileStones = points(index,:);
milestones = [milestones; candidateMileStones];

% Loop through each of the milestones to obtain its nearest neighbours
for i=1:length(milestones)
    [D,I] = pdist2(milestones, milestones(i, :), 'euclidean','Smallest', 5);

    % Loop through each nearest neighbour and check if the edge between the
    % milestone and its neighbour is crossing an edge. If not, then add to
    % the matrix of edges.
    for j = 2:length(I)
        [inCollision, edge] = CheckCollision(milestones(i, :), milestones(I(j), :), ↙
map);
        if inCollision == 0
            edges = [edges; milestones(i, :) milestones(I(j), :)];
        end
    end
end

% Ordering the edges and removing duplicates
for k = 1:size(edges, 1)
    if edges(k, 1) > edges(k, 3)
        one = edges(k, 1:2);
        two = edges(k, 3:4);
        edges(k, :) = [two one];
    end
end
edges = unique(edges,'rows');

% Variable to store shortest path
spath = []; % shortest path, stored as a milestone row index sequence

% Matrix to hold the closed states from the Astar algorithm
closedStates = [];

% Matrix to hold the open states from the Astar algorithm
```

```
% First 2 columns hold the milestone, next 2 columns hold g (cost-to-come)
% and f (which is the sum of the cost-to-come and the heuristic cost-to-go)
% Last 2 columns hold the parent of the milestone
openStates = [start 0 norm(start - finish) 0 0];

% Set variable to not solved
solved = 0;

% Keep looping until there are no more open states left
while(~isempty(openStates))
    % Find the open state with the lowest total f cost and set as current
    [P, I] = min(openStates(:, 4));
    current = openStates(I,:);

    % Check if current is the finish milestone. If so, add current to the
    % closed states and break out of the loop
    if isequal(current(1:2), finish)
        disp('FOUND FINISH MILESTONE')
        closedStates = [closedStates; current];
        solved = 1;
        break;
    end

    % Remove current from open states and add to closed states
    openStates(I, :) = [];
    closedStates = [closedStates; current];

    % Loop over all the edges to get the current milestone's neighbours
    for m = 1:size(edges, 1)
        % Determining which edges are corresponding to the current and
        % which milestone from the edges is the neighbour
        if isequal(edges(m, 1:2), current(1:2)) || isequal(edges(m, 3:4), current(1:2))
            n = [];
            if isequal(edges(m, 1:2), current(1:2))
                n = edges(m, 3:4);
            else
                n = edges(m, 1:2);
            end
            if isequal(n, current(1:2))
                continue
            end

            % If neighbour already in closed states, break out of for loop
            % and continue to the checking next edges
            skipClosed = 0;
            for c = 1:size(closedStates, 1)
                if isequal(closedStates(c, 1:2), n)

                    skipClosed = 1;
                    break;
```

```
            end
        end

        % If neighbour not in the closed states, then it will need to
        % be added to the open states or updated in the open states
        if(skipClosed == 0)
            % Define the g (cost-to-come), h (heuristic cost-to-go),
            % and f (the sum of g and h)
            cost_to_come = current(3) + norm(current(1:2) - n);
            heuristic_cost_to_go = abs(n(1)-finish(1)) + abs(n(2)-finish(2));
            F = cost_to_come + heuristic_cost_to_go;

            % Check if neighbour is in open states already
            skip = 0;
            if(~isempty(openStates))
                for o = 1:size(openStates, 1)
                    if isequal(openStates(o, 1:2), n)
                        skip = 1;
                        % If already in open states, check if the
                        % stored cost-to-come is higher or lower than
                        % the new calculated cost-to-come. If lower,
                        % then continue. If higher, update the costs in
                        % open states
                        if openStates(o, 3) < cost_to_come
                            continue
                        else
                            openStates(o, 3:6) = [cost_to_come F current(1:2)];
                        end
                        break;
                    end
                end
            end
            % If neighbour not in open states, add neighbour to open
            % states with appropriate information
            if skip == 0
                openStates = [openStates; n cost_to_come F current(1:2)];
            end
        end
    end
end

spath2 = [];

if solved == 1
    % Back track to obtain the list of milstones that form the shortest path
    % from start to finish
    spath2 = [closedStates(size(closedStates, 1), :)];
    while ~isequal(spath2(1, 1:2), start)
```

```
        parent = spath2(1, 5:6);
        for c = 1:size(closedStates, 1)
            if isequal(closedStates(c, 1:2), parent)
                spath2 = [closedStates(c, :); spath2];
            end
        end
    end
    % Obtain the milestone row index sequence
    for s = 1:size(spath2, 1)
        for m = 1:size(milestones, 1)
            if isequal(milestones(m, :), spath2(s, 1:2))
                spath2(s, 1:2);
                spath = [spath; m];
            end
        end
    end
end

% ------end of your optimized algorithm-------
dt = toc;

figure(2); hold on;
plot(milestones(:,1),milestones(:,2), 'm.');
if (~isempty(edges))
    line(edges(:,1:2:3)', edges(:,2:2:4)', 'Color','magenta')
end
if (~isempty(spath))
    for i=1:length(spath)-1
        plot(milestones(spath(i:i+1),1),milestones(spath(i:i+1),2),  'go-', 'LineWidth',↙
3);
    end
end
str = sprintf('Q3 - %d X %d Maze solved in %f seconds', row, col, dt);
title(str);

print -dpng assignment1_q3.png
```