Assignment 1 - Motion Planning

Name: Sugumar Prabhakaran (id#: 994126815)

Course: ROB521 - Mobile Robotics

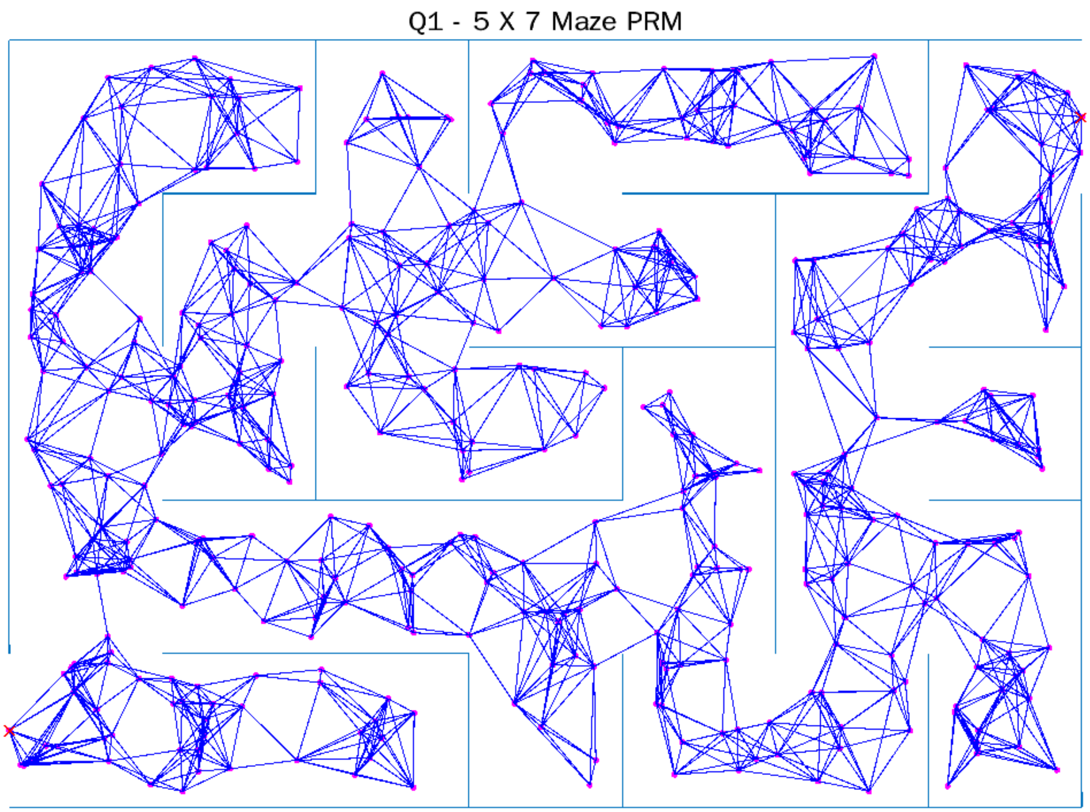
Instructor: Dr. Steven Waslander

Due Date: 15 Feb 2022

Results and Analysis

Question 1 - Implement PRM Algorithm

Below is the probabilistic road map (PRM) graph generated for this question using a K-nearest-neighbors (KNN) value of K= 8. The graph was generated in approximately 0.44 seconds.

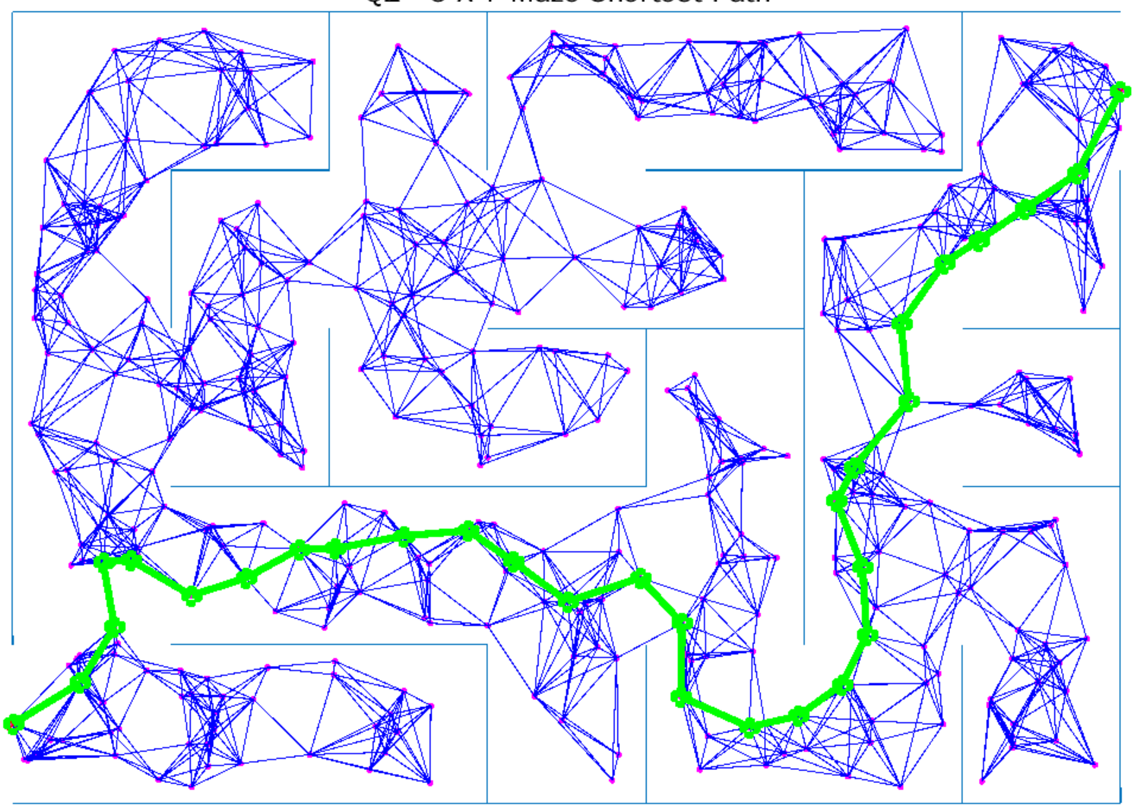


K values below 8 did not always generate a completely connected graph between the start and finish vertices. The points were generated randomly and collisions were checked for the potential edge between all vertices and their nearest neighbors (K=8). Overall, the performance of this method was quite reasonable for the 5x7 maze despite the significant number of collision checks.

Question 2 - Shortest Path Using Dijkstra's or A* Algorithm

For this section, I implemented Dijkstra's algorithm with successful results. The algorithm was able to find the shortest path in less than 1.2 seconds. The results can be seen below in the next image.

Q2 - 5 X 7 Maze Shortest Path

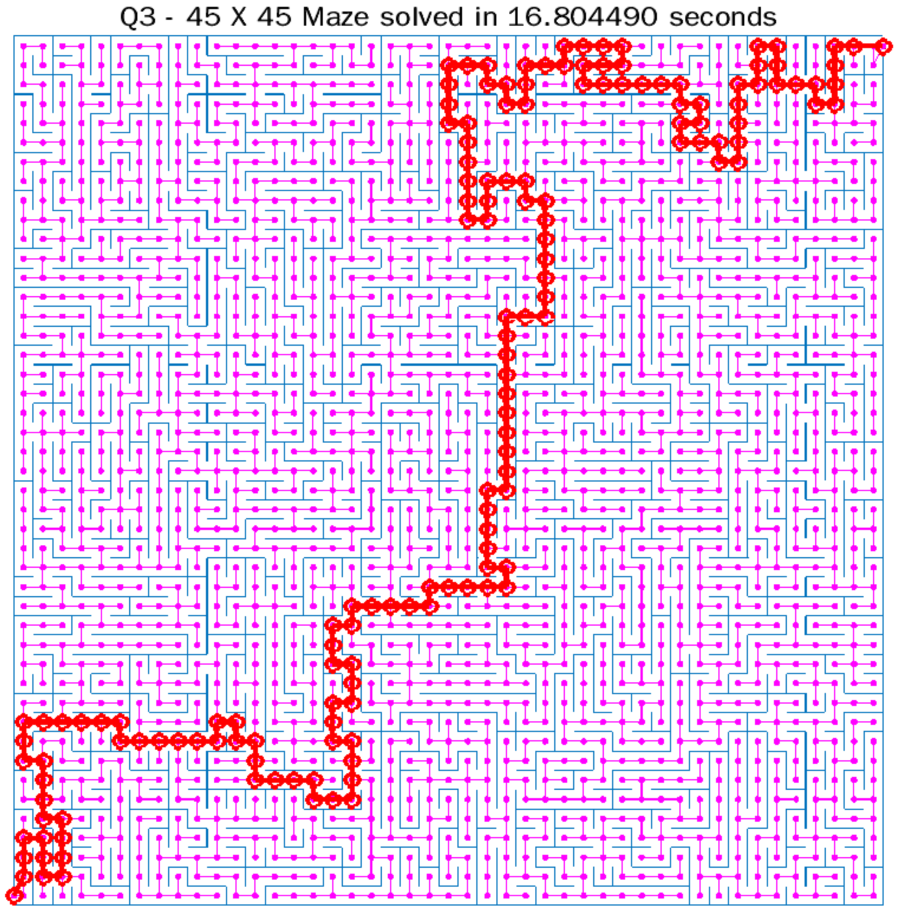


Question 3 - Optimize Sampling, Connections or Collisions to Improve Efficiency

When running the same above code for larger graphs, we are unable to reliably obtain a solution for graphs larger than 12 x 12. The largest successful solution was for a 20x20 maze that took over 120 seconds. As a result, several areas were looked at to improve efficiency: (1) collision checking, (2) searching algorithm, (3) obtaining neighbors, and (4) sampling.

For collision checking, the cross-product method between two edges was used but had negligible improvement in performance to the end point coordinate min-max method (CheckCollision function). Similarly, the A* algorithm was implemented but did not produce any improvement in time. Implementing a KD-tree was considered but was not necessary after implementing a uniform grid sampling. For smaller grids, a density of 16 points per unit square was used and for larger grids (greater than 10x10), the density was 1 point per unit square.

The most significant improvement in performance was obtained by generating milestones using a uniform grid of points. The resulting 45 x 45 maze was solved in under 17 seconds:



Appendix - Source Code

The below source code is broken up into five sections:

1. **Main Code Implementation**
2. **Question 1 Main Functions**
3. **Question 2 Main Functions**
4. **Question 3 Main Functions**
5. **Helper Functions**

```
% 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;
show_maze(map,row,col,h); % Draws the maze
drawnow;
```

Main Code Implementation

```
% ======================================================
% 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 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;                      % # of samples to try for milestone creation
milestones = [start; finish];  % each row is point [x y] in feasible space
%edges = [];                   % each row is edge of form: [x1 y1 x2 y2]

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

%1. Create milestones with rand. sampling and discard pts close to edges
milestones = GenerateMilestones(nS, row, col, start, finish, map);

%2. Implement PRM using KNN method and collision checking to generate graph
K = 8;
edges = GeneratePRM(milestones, K, map);


% ------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','blue')%row:[x1 x2 y1 y2]
end
str = sprintf('Q1 - %d X %d Maze PRM', row, col);
plot(start(1), start(2),'rx', 'LineWidth',1)
plot(finish(1), finish(2),'rx','LineWidth',1)
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;
%pause(5);
% Variable to store shortest path
%spath = []; shortest path, stored as a milestone row index sequence

% ------insert your shortest path finding algorithm here-------

% 1. Implement Dijkstra's algorithm function
```

```
[spath, visited] = Dijkstra(edges, milestones);

% 2. Backtrack from finish node using visited array to update shortest path
spath = BackTrack(spath, visited, milestones);

% ------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)
clear; clc;

row = 45;
col = 45;
map = maze(row,col);
start = [0.5, 1.0];
finish = [col+0.5, row];
milestones = [start; finish];  % each row is point [x y] in feasible space
edges = [];  % each row is 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------
size = 'large';
% 1. Generate graph using Probabilistic Road Map (PRM)
%milestones = GenerateMilestones(nS, row, col, start, finish, map);
```

```
%edges = GeneratePRM(milestones, K, map);
milestones = GenerateGridSamples(row, col, start, finish, map);
edges = GeneratePRMFromGrid(milestones, map, size);

% 3. Implement A* algorithm function
[spath2, visited2] = AStar(edges, milestones);

% 4. Backtrack from finish node using visited array to update shortest path%
spath2 = BackTrack(spath2, visited2, milestones);


% ------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

% plot A* path
if (~isempty(spath2))
    for i=1:length(spath2)-1
        plot(milestones(spath2(i:i+1),1), milestones(spath2(i:i+1),2), ...
             'ro-', 'LineWidth',2);
    end
end
str = sprintf('Q3 - %d X %d Maze solved in %f seconds', row, col, dt);
title(str);

print -dpng assignment1_q3.png
```

Question 1 - Main Functions

```
function [milestones] = GenerateMilestones(nS, row, col, start, finish, map)
% GENERATEMILESTONES - Generate random samples and discard points that are
%                       too close to edges.
% inputs: nS (scalar)    - Number of samples to try. (Ex. 500 points)
%        row (scalar)    - Number of rows for the maze
%        col (scalar)    - Number of columns for the maze
%      start (2x1 array) - start point in form: [xs ys];
%     finish (2x1 array) - end point of maze in form: [xf yf];
%        map (Nx4 array) - array of all edges, each row form: [x1 y1 x2 y2]
% output: milestones (Mx2 array) - array of all points: [x1 y1; x2 y2; ...]

% generate milestones within bound:  X: (0.5, col+0.5), Y: (0.5, row+0.5)
points = [rand(nS,1)*col + 0.5, rand(nS,1)*row + 0.5]; % 500 random points
pts_bool_filter = MinDist2Edges(points, map)>0.1;  % filter pts near edge
milestones = [start; points(pts_bool_filter, :); finish]; % update milest.
end

function [edges] = GeneratePRM(milestones, K, map)
% GENERATEPRM - Iterates through K neighbors for each vertex, checks for
%               collision with an edge between them, connects them in graph
```

```
% inputs: milestones (Nx2 array) - all possible vertexes
%                     K (scalar) - number of neighbors to uses
%                map (Mx4 array) - all edges of maze with end pts in a row
% outputs:     edges (Rx4 array) - all valid edges connecting vertices

% 1. set parameters
edges = [];
N = length(milestones);     % # of milestones

% 2. implement KNN Algorithm and collision checking
for i=1:N                   % iterate through 1:N milestones
    % find K neighbors for milestone i
    neighbors = KNN(milestones, milestones(i,:), K);

    for j=1:K               % iterate through 1:K neighbors
        % check if edge between milestone i and neighbor j collide
        [Collision, ~] = CheckCollision(milestones(i,:), neighbors(j,:), map);

        if (Collision == 0) % if no collision
            % edge can be written in 2 ways (mile.,nbor.) & (nbor., mile.)
            edge = [milestones(i,1),milestones(i,2), ...
                    neighbors(j,1), neighbors(j,2)];
            edge2 =[neighbors(j,1), neighbors(j,2), ...
                    milestones(i,1),milestones(i,2)];
            if (isempty(edges))
                edges = [edge];
            % check if either form of edge is already in 'edges' array
            elseif ((~ismember(edge, edges, 'rows')) && ...
                    (~ismember(edge2, edges, 'rows')))
                % if edge is not in 'edges' add it to the array
                edges = [edges; edge];
            end
        end
    end
end
end
```

Question 2 - Main Functions

```
function [spath, visited] = Dijkstra(edges, milestones)
% DIJKSTRA - Find shortest path from start node to finish node
% inputs: edges (Mx4 array) - each 1x4 row is edge: [x1 y1 x2 y2];
%         milestones (Nx2 array) - each row is a point of form: [xi yi];
%         spath (1xJ array) - contains indices of nodes on shortest path in
%                             milestones array in order of start to finish
% outputs: spath (1x1 array) -index of finish node if alg 'success'
%          visited (Nx5 array) - row form: [xi yi cost parent_x parent_y];

% 1. Create and initialize variables, arrays
start = milestones(1,:);      % start is first node in milestones
finish = milestones(end, :);  % finish is last node in milestones
queue = [start, 0];    % pri. queue of unvisited nodes + cost-to-come (c2c)
visited = [start, 0, start;]; % list of visited nodes + c2c + parent node
```

```
% 2. Implement Dijkstra's Algorithm
while ~isempty(queue)       % run as long as nodes remaining in queue

    % A. Get First Node
    x = queue(1,1:2);       % new node 'x' first queue elm first 2 cols
    %scatter(x(1), x(2), 'co', 'filled')
    x_cost = queue(1, 3);   % cost-to-come of new node 'x'
    queue = queue(2:end,:); % pop vertex 'x' from queue

    % B. Termination Condition: if finish node reached
    if (finish == x)
        [~, idx] = ismember(finish, milestones, 'rows'); % store finish idx
        spath = [idx];
        disp('Success')
        return
    end

    % C. find all reachable neighbors of 'x'
    neighbors = FindNeighbors(edges, x);    % rtn neighboring nodes array
    [J,~] = size(neighbors);                % total J neighbors

    % D. Iterate through each neighbor
    for j=1:J                               % iterate thru each neighbor
        x_prime = neighbors(j,:);           % set neighbor node x'
        %scatter(x_prime(1),x_prime(2), 'ro', 'filled')

    % E. For neighbors not visited, update Queue and Visited arrays
        if ~(ismember(x_prime, visited(:,1:2), 'rows'))   % if not visited
            cost_come = x_cost + EuclideanDist(x_prime, x); % x' c2c
            visited(end+1,:) = [x_prime, cost_come, x];   % mark as visited
            queue = InsertQueue(queue, x_prime, cost_come); % add to queue
        end
    end
end

spath = [];
disp('Unsucessful - Did not find a path')
end

function [spath] = BackTrack(spath, visited, milestones)
% BACKTRACK - backtrack from finish vertex to start vertex through
%              respective parent nodes until the parent node = current node
% input: spath (1x1 array)      - contains index of finish node
%        visited (Nx5 array)    - contains data on parent nodes in col 4:5
%        milestones (Nx2 array) - contains all milestone vertices in graph
% output: spath (1xM array)     - contains updated indexes of milestones in
%                                 order start to finish on shortest path

% initialize current and parent nodes
cur_node = milestones(spath(end), :);    % set current node to finish
[~, cur_idx_vis] = ismember(cur_node, visited(:,1:2),'rows');%parent idx in visited
parent_node = visited(cur_idx_vis,4:5);          % set parent node
```

```
% loop through starting at finish node and add parent nodes to spath
% loop until parent = cur
while (~(parent_node(1) == cur_node(1)) || ~(parent_node(2) == cur_node(2)))
    % get parent idx from milestone
    [~, parent_idx_mil] = ismember(parent_node, milestones, 'rows');
    spath = [parent_idx_mil; spath];          % add parent node to spath
    cur_node = parent_node;               % set parent node to current node
    [~, parent_idx_vis] = ismember(cur_node, visited(:,1:2),'rows'); % new parent id
    parent_node = visited(parent_idx_vis,4:5);      % set new parent node
end
end
```

Question 3 - Main Functions

```
function [milestones] = GenerateGridSamples(row, col, start, finish, map)
% GENERATEGRIDSAMPLES - take grid size and generate milestones in grid
% inputs: row (scalar)    - number of rows of grid
%         col (scalar)    - number of cols of grid
%       start (2x1 array) - start point coordinates [xs, ys];
%      finish (2x1 array) - end point cooridnates [xf, yf];
%         map (Mx4 array) - all edges of maze with end pts in a row
% output: milestones (Nx2 array) - all generated vertices in the grid

% Note. maze bounds: bottom left = [0.5 0.5], top right = [col +.5, row+.5]

points = [start];

% smallest grids, generate 9 points per square
if (row*col <= 100)
   x_coord = linspace(0.75, col+0.25, col*4-1);
   y_coord = linspace(0.75, row+0.25, row*4-1);

   for i=1:length(x_coord)
        for j =1:length(y_coord)
            if ((mod(x_coord(i)-0.5,1)<0.1) || (mod(y_coord(j)-0.5,1)<0.1))
                dist = MinDist2Edges([x_coord(i), y_coord(j)], map);
                if dist <0.1
                    continue
                else
                    points = [points; x_coord(i), y_coord(j)];
                end
            else
                points = [points; x_coord(i), y_coord(j)];
            end
        end
   end

else
   x_coord = 1:col;
   y_coord = 1:row;

   for i=1:length(x_coord)
        for j =1:length(y_coord)
            points = [points; x_coord(i), y_coord(j)];
```

```
        end
    end
end

milestones = [points; finish];
end

function [edges] = GeneratePRMFromGrid(milestones, map, size)
% GENERATEPRMFROMGRID - Given a uniform-grid of milestones, build a graph
% inputs: milestones (Nx2 array) - each row is a point of form: [xi yi];
%                map (Mx4 array) - all edges of maze with end pts in a row
%                  size (string) - 'small' or large to compensate for dist
%                                  between the points
% outputs: edges (Mx4 array) - each 1x4 row is edge: [x1 y1 x2 y2];

% for small grid: 8 directions (neighbors) for vertex to traverse:
% up:          (x, y+0.25),         down:       (x, y-0.25)
% right:       (x+0.25, y),         left:       (x-0.25, y)
% topright:    (x+0.25, y+0.25),    topleft:    (x-0.25, y+0.25)
% bottomright: (x+0.25, y-0.25),    bottomleft: (x-0.25, y-0.25)

% initialize parameters
edges = [0 0 0 0];
N = length(milestones);

% iterate through all vertices
for i=1:N
    vertex = milestones(i,:);           % set vertex
    x = vertex(1); y = vertex(2);       % x and y coords for vertex
    % neighbors in clockwise order starting with directly above
    if (size == 'small')                % for graph <= 10x10
        neighbors = [x, y+0.25; x+0.25, y+0.25; x+0.25, y; ...
                     x+0.25, y-0.25; x, y-0.25; x-0.25, y-0.25; ...
                     x-0.25, y; x-0.25, y+0.25];
    else                                % if graph is larger than 10x10
        if ((i==1) || (i==N)) % start, finish pts are closer than 1 unit
            neighbors = KNN(milestones, vertex, 3);
        else                  % rest of the points are 1 unit away
            neighbors = [x, y+1; x+1, y+1; x+1, y; x+1, y-1; ...
             x, y-1; x-1, y-1; x-1, y; x-1, y+1];
        end
    end
    % iterate through each of the neighbors for a vertex
    % if no collision, add it as an edge to build the graph
    for j=1:length(neighbors)
        if (ismember(neighbors(j,:),milestones,'rows'))
            [inCollision, ~] = CheckCollision(vertex, neighbors(j,:), map);
            if ((inCollision==0) && ...
                (~ismember([vertex, neighbors(j,:)],edges, 'rows')) && ...
                (~ismember([neighbors(j,:), vertex],edges, 'rows')))
                edges = [edges; vertex, neighbors(j,:)];
            end
        end
    end
```

```
end
end

function [spath, visited] = AStar(edges, milestones)
% ASTAR (A*) - Find shortest path from start node to finish node
% inputs: edges (Mx4 array) - each 1x4 row is edge: [x1 y1 x2 y2];
%         milestones (Nx2 array) - each row is a point of form: [xi yi];
%         spath (1xJ array) - contains indices of nodes on shortest path in
%                             milestones array in order of start to finish
% outputs: spath (1x1 array) -index of finish node if alg 'success'
%          visited (Nx5 array) - row form: [xi yi cost parent_x parent_y];

% 1. Create and initialize variables, arrays
start = milestones(1,:);      % start is first node in milestones
finish = milestones(end, :);  % finish is last node in milestones
cost_to_go = EuclideanDist(finish, start);
queue = [start, 0+cost_to_go];    % pri. queue of unvisited nodes + cost-to-come (c2c)
visited = [start, 0, start;]; % list of visited nodes + c2c + parent node


% 2. Implement A* Algorithm
while ~isempty(queue)       % run as long as nodes remaining in queue

    % A. Get First Node
    x = queue(1,1:2);       % new node 'x' first queue elm first 2 cols
    x_cost = queue(1, 3);   % cost-to-come of new node 'x'
    queue = queue(2:end,:); % pop vertex 'x' from queue

    % B. Termination Condition: if finish node reached
    if (finish == x)
        spath = find(milestones==finish,1);   % store index of finish node
        disp('Success')
        return
    end

    % C. find all reachable neighbors of 'x'
    neighbors = FindNeighbors(edges, x);    % rtn neighboring nodes array
    [J,~] = size(neighbors);                % total J neighbors

    % D. Iterate through each neighbor
    for j=1:J                               % iterate thru each neighbor
        x_prime = neighbors(j,:);           % set neighbor node x'

    % E. For neighbors not visited, update Queue and Visited arrays
        if ~(ismember(x_prime, visited(:,1:2), 'rows'))   % if not visited
            cost = x_cost + EuclideanDist(x_prime, x) + ...
                   EuclideanDist(finish, x); % x' c2c
            visited(end+1,:) = [x_prime, cost, x];   % mark as visited
            queue = InsertQueue(queue, x_prime, cost); % add to queue
        end
    end
end

spath = [];
```

```
disp('Unsucessful - Did not find a path')
end
```

Helper Functions

These helper functions were required in addition to those provided with the problem to feed into the main functions in the previous section.

```
function dist = EuclideanDist(P1, P2)
% EUCLIDEANDIST - Calculate euclidean distance (l2 norm) for 2D points
% inputs: P1 (2x1 array) - Point 1 of form: [P1_x, P1_y]
%         P2 (2x1 array) - Point 2 of form: [P2_x, P2_y]
% output: dist (scalar)  - distance between P1, P2

dist = sqrt( ( P2(1)-P1(1) )^2 + ( P2(2)-P1(2) )^2 );
end

function [neighbors] = KNN(milestones, point, K)
% KNN - Implement K-Nearest-Neighbors alg. to find K neighbors of a point
% inputs: milestones (Nx2 array) - 2D vertices: [X1 Y1; X2 Y2; ... XN YN;]
%         point (2x1 array)      - current vertex Ex. [x, y]
%         K (scalar)             - number of neighbors to find
% output: neighbors (Kx2 array)  - K nearest neighbors sorted by distance
%                                  [XN1, YN1; XN2, YN2;... XNK, YNK;]

x = point(1); y = point(2);       % x, y coordinates of current node

% array based dist calculation rtning index of distances sorted in ascend.
[~, idx_array] = sort(sqrt((milestones(:,1) - x).^2 + ...
                          (milestones(:, 2) - y).^2));
% re-sort milestones using sorted distance idx
sorted_points_array = milestones(idx_array, :);

% select only first K neighbors. Note, 1st neighbor is vertex itself
neighbors = sorted_points_array(2:K+1, :);
end

function [neighbors] = FindNeighbors(edges, point)
% FINDEDGES - Given a point, find all edges connected to that point
% inputs: edges (Nx4 array) - each 1x4 row is edge: [x1 y1 x2 y2];
%         point (2x1 array) - milestone point of form: [P_x, P_y]
% output: neighbors (Kx2 array) - K rows of neighbors: [xj yj;] to point P

neighbors = [];                    % initialize array to store neighbors
[N,~] = size(edges);               % N edges

for i=1:N                          % iterate through all N edges in graph

    % point could match start vertex of edge, end vertex of edge or neither
    if point == edges(i,1:2)       % point matches start vertex
        neighbor = edges(i,3:4);   % neighbor is opposite vertex
    elseif point == edges(i,3:4)   % point matches end vertex
        neighbor = edges(i,1:2);   % neighbor is opposite vertex
    else
```

```
        continue                    % no matches, skip to next edge
    end

    % if neighbor is not in neighbor list, add it to the list
    if isempty(neighbors)
        neighbors = [neighbor];
    elseif ~(ismember(neighbor,neighbors, 'rows'))
        neighbors = [neighbors; neighbor];
    end
end
end

function [updated_queue] = InsertQueue(queue, vertex, cost)
% INSERTQUEUE - Insert vertex into  queue in ascend. order of cost-to-come
% inputs: queue (Nx5 array) - of row form: [Vi_x, Vi_y, 3.21, Vj_x, Vj_y]
%         vertex (2x1 array) - point of form: [Vi_x, Vi_y];
%         cost (scalar) - cost-to-come from start to vertex 'Vi'
% output: updated_queue (Nx3 array) - with vertex 'Vi' inserted

if (isempty(queue))     % if queue is empty, insert into empty array
    updated_queue = [vertex, cost];
    return

else                    % else, find id of queue elm with cost < Vi cost
    idx = find(queue(:,3) < cost, 1, 'last');

    if (isempty(idx))   % if all elms cost > Vi cost, insert Vi at front
        updated_queue = [[vertex, cost]; queue(:, :)];
        return
    else                % otherwise, insert Vi right after idx
        updated_queue = [queue(1:idx,:); [vertex, cost]; ...
                         queue(idx+1:end, :)];
        return
    end
end
end
```