

---

```

% =====
% 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-----

% random sampling 500 milestones
for i = 1:nS
    % for x coordinate
    ran = rand;
    pt_x = (col - 0.2) * ran + 0.6;
    % for y coordinate
    ran = rand;
    pt_y = (row - 0.2) * ran + 0.6;
    pt = [pt_x pt_y];
    % ensure each milestone is at least 0.1 units away from maze walls
    if (MinDist2Edges(pt, map) > 0.1)
        milestones = [milestones; pt];
    end
end

% ensure each milestone is unique
milestones = unique(milestones, 'rows', 'stable');

% nearest neighbour connection
% compute nearest neighbour distance
% (any milestone within this distance is a neighbour)
neigh_dist = row / 4;
if (col > row)
    neigh_dist = col / 4;
end
% find the nearest neighbours for each milestone

```

---

---

```

for j = 1:size(milestones)
    curr_pt = milestones(j, :);
    dist_xy = abs(milestones - curr_pt);
    dist_tot = sum(dist_xy, 2);
    indices = find(dist_tot < neigh_dist);
    indices = indices(indices > j);
    % for each nearest neighbour, compute an edge
    for k = 1:size(indices)
        curr_ind = indices(k);
        neigh_pt = milestones(curr_ind, :);
        edges = [edges; curr_pt neigh_pt];
        edges = [edges; neigh_pt curr_pt];
    end
end

% check edges for collision with walls
edges_removed = edges;
remove_ind = [];
for e = 1:size(edges)
    curr_edge1 = edges(e, 1:2);
    curr_edge2 = edges(e, 3:4);
    % compute all edge indices that contain colliding edges
    if (CheckCollision(curr_edge1, curr_edge2, map) == 1) % if collision
        remove_ind = [remove_ind; e];
    end
end
% remove colliding edges
for r = size(remove_ind):-1:1 % start from the end of the indices list
    curr_remove = remove_ind(r);
    edges_removed(curr_remove, :) = []; % remove
end
% set edges as the set with all colliding edges removed
edges = edges_removed;

% -----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-----

% A* algorithm is implemented as a function
% (functions are written at the end of the code)
spath = A_star(1, 2, milestones, edges);

% -----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 = 42;
col = 42;
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-----

% uniform sampling (with distance of 1 unit in each direction)
dist_uni = 1;

for i = 1:col
    % for x coordinate
    % start from x = 0.5, and want center so +0.5
    pt_x = (0.5 + 0.5) + dist_uni * (i - 1);
    % for y coordinate
    for j = 1:row
        % start from y = 0.5, and want center so +0.5
        pt_y = (0.5 + 0.5) + dist_uni * (j - 1);
        pt = [pt_x pt_y];
        % ensure each milestone is at least 0.1 units away from maze walls
        if (MinDist2Edges(pt, map) > 0.1)
            milestones = [milestones;pt];
        end
    end
end

% ensure each milestone is unique
milestones = unique(milestones, 'rows', 'stable');

% nearest neighbour connection
neigh_dist = 2; % nearest neighbour distance
% find the nearest neighbours for each milestone
for j = 1:size(milestones)
    curr_pt = milestones(j, :);
    dist_xy = abs(milestones - curr_pt);
    dist_tot = sum(dist_xy, 2);
    indices = find(dist_tot < neigh_dist);
    indices = indices(indices > j);
    % for each nearest neighbour, compute an edge
    for k = 1:size(indices)
        curr_ind = indices(k);
        neigh_pt = milestones(curr_ind, :);
        edges = [edges; curr_pt neigh_pt];
        edges = [edges; neigh_pt curr_pt];
    end
end

% check edges for collision with walls
edges_removed = edges;

```

---

---

```

remove_ind = [];
for e = 1:size(edges)
    curr_edge1 = edges(e, 1:2);
    curr_edge2 = edges(e, 3:4);
    % compute all edge indices that contain colliding edges
    if (CheckCollision(curr_edge1, curr_edge2, map) == 1) % if collision
        remove_ind = [remove_ind; e];
    end
end
% remove colliding edges
for r = size(remove_ind):-1:1 % start from the end of the indices list
    curr_remove = remove_ind(r);
    edges_removed(curr_remove, :) = []; % remove
end
% set edges as the set with all colliding edges removed
edges = edges_removed;

% find the shortest path, stored as a milestone row index sequence
spath = A_star(1, 2, milestones, edges);

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

figure(2); hold on;
plot(milestones(:,1),milestones(:,2),'m.');
```

if (~isempty(edges))

```

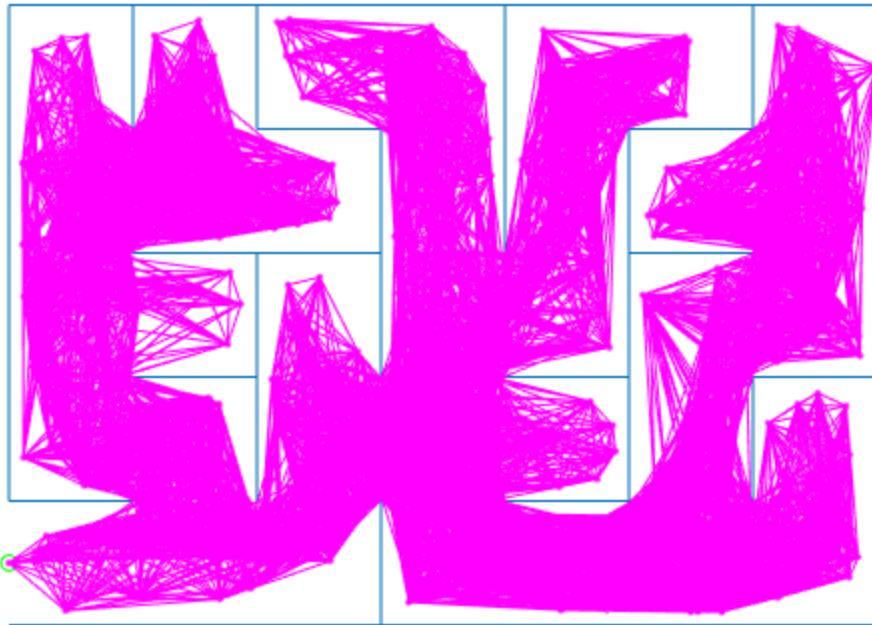
    line(edges(:,1:2:3)', edges(:,2:2:4)', 'Color', 'magenta');
end
if (~isempty(spath))
    for i=1:length(spath)-1
        plot(milestones(spath(i:i+1),1),milestones(spath(i:i
+1),2), 'go-', 'LineWidth',3);
    end
end
str = sprintf('Q3 - %d X %d Maze solved in %f seconds', row, col, dt);
title(str);

print -dpng assignment1_q3.png

Time to create PRM graph
Elapsed time is 1.602603 seconds.
Time to find shortest path
```

---

Q1 - 5 X 7 Maze PRM



## Helper Functions

compute a list of milestone indices constructing the shortest path input "precedent" is a column vector of the precedent milestone for each milestone of the corresponding vector index

```
function path_tot = construct_path(precedent, current, start)
    path_tot = [current];
    while current ~= start
        current = precedent(current);
        path_tot = [current path_tot];
    end
end
```

% the heuristic function (using the Manhattan distance)

```
function hScore = heuristic(state, goal, milestones)
    x1 = milestones(state, 1);
    y1 = milestones(state, 2);
    x2 = milestones(goal, 1);
    y2 = milestones(goal, 2);
    hScore = abs(x2 - x1) + abs(y2 - y1); % Manhattan distance
end
```

% the edge weight of each edge (using the edge distance in units)

```
function edge_w = edge_weight(state, neigh, milestones)
    x1 = milestones(state, 1);
    y1 = milestones(state, 2);
```

---

```

    x2 = milestones(neigh, 1);
    y2 = milestones(neigh, 2);
    edge_w = sqrt((x2 - x1)^2 + (y2 - y1)^2);
end

% compute the minimum fScore among only the currently available
% states/milestone indices
function curr_state = min_fScore(states, fScore)
    curr_fScore = [];
    % compute a list of fScore for all available states
    for i = 1:size(states, 2)
        curr_fScore = [curr_fScore fScore(states(i))];
    end
    % find the minimum fScore among the list computed
    ind = find(fScore == min(curr_fScore));
    % find the corresponding state(s) of the minimum fScore
    % (return only the first state of the corresponding minimum fScore
    % if multiple are found)
    for j = 1:size(ind)
        if (ismember(ind(j), states) == 1)
            curr_state = ind(j);
            return;
        end
    end
end

% compute the neighbours (which have an edge) of the current milestone
function neigh_list = neighbours(curr_mile, edges, milestones)
    neigh_list = [];
    % find indices in the edge list that has the current milestone
    % as a side of the edge
    edge_ind = find(edges(:, 1) == curr_mile(1) & edges(:, 2) ==
curr_mile(2));
    % for each edge index, find the corresponding milestone from the
    % milestones list (i.e., the other side of the edge)
    for e = 1:size(edge_ind, 1)
        neigh_mile = edges(edge_ind(e), 3:4); % neighbour milestone
        % find a milestone in the milestones list with the same
        % x and y coordinates of the neighbour milestone
        x_ind = find(milestones(:, 1) == neigh_mile(1)); % x coordinate
        y_ind = find(milestones(:, 2) == neigh_mile(2)); % y coordinate
        % compute the milestone that exist both in the x coordinate
        % and y coordinate indices list
        [neigh, ~] = intersect(x_ind, y_ind);
        neigh_list = [neigh_list neigh];
    end
end

% implement the A-star algorithm to find the shortest path
function sPath = A_star(start, goal, milestones, edges)
    states = [start]; % to store milestone indices/states
    % list of precedent milestone of each corresponding milestone index
    precedent = zeros(size(milestones, 1), 1);
    gScore = inf(size(milestones, 1), 1); % gScore for milestones

```

---



---

```

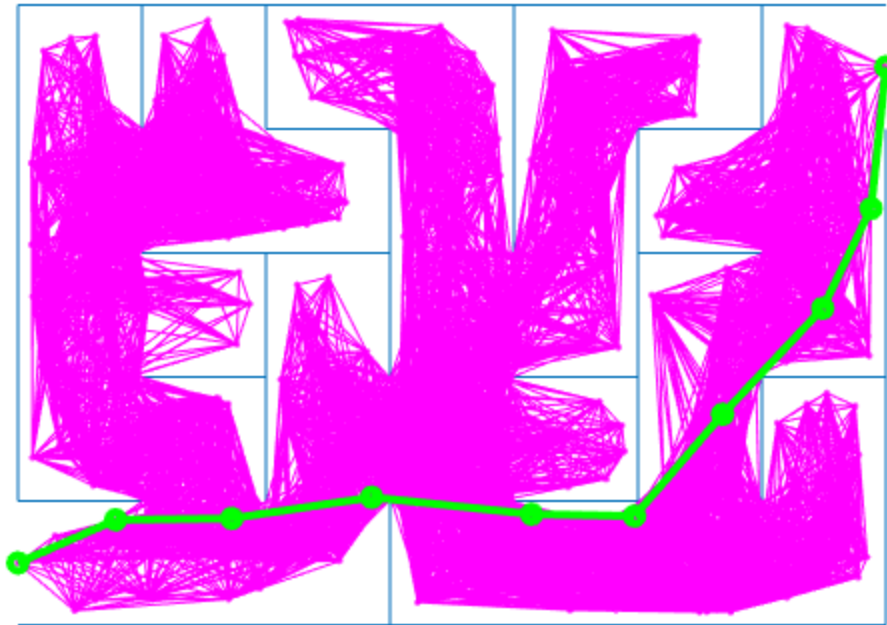
gScore(start) = 0;
% fScore for milestones (g + h scores)
fScore = inf(size(milestones, 1), 1);
fScore(start) = heuristic(start, goal, milestones);
% while there are states in the list
while (~isempty(states))
    % pick the state with min fScore (from list of available states)
    curr_state = min_fScore(states, fScore);
    % finish if current state is the goal state
    if (curr_state == goal)
        sPath = construct_path(precedent, curr_state, start);
        return; % return after path found
    end
    % remove the current state from the list of available states
    states = states(states~=curr_state);
    % compute list of neighbour states (i.e. have an edge with current)
    curr_mile = milestones(curr_state, :);
    neigh_list = neighbours(curr_mile, edges, milestones);
    % for each neighbour of current state
    for n = 1:size(neigh_list, 2)
        neigh = neigh_list(n); % neighbour state
        % compute the tentative gScore (current + new edge score)
        gScore_ten = gScore(curr_state) + edge_weight(curr_state, neigh,
milestones);
        % if tentative gScore < than the neighbour gScore
        if gScore_ten < gScore(neigh)
            % set current state as the precedent of current neighbour
            precedent(neigh) = curr_state;
            % set tentative gScore as its gScore
            gScore(neigh) = gScore_ten;
            % set its fScore (gScore + heuristic)
            fScore(neigh) = gScore_ten + heuristic(neigh, goal,
milestones);

            % add the current neighbour to the list of available states
            % if it is not already in
            if ismember(neigh, states) == 0
                states = [states neigh];
            end
        end
    end
end
end
sPath = [];
end

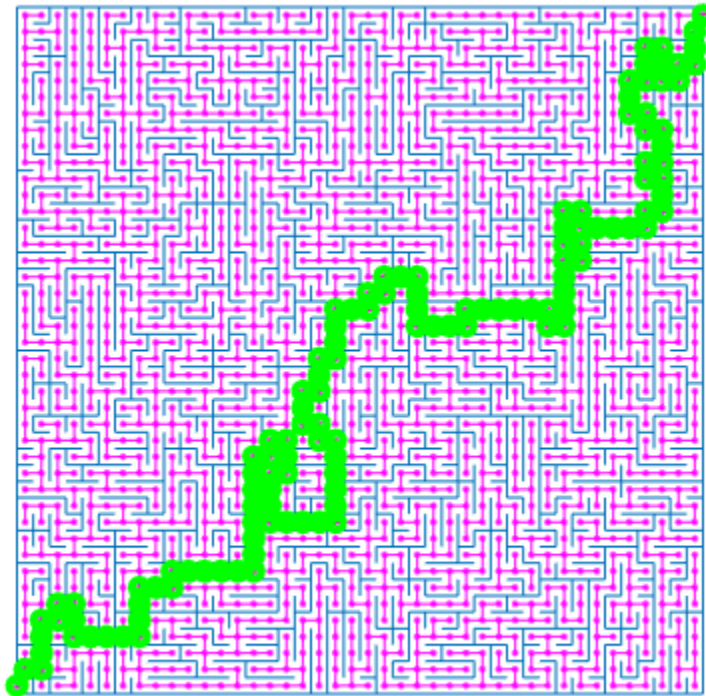
```

*Elapsed time is 18.124164 seconds.*  
*Attempting large 42 X 42 maze...*

**Q2 - 5 X 7 Maze Shortest Path**



**Q3 - 42 X 42 Maze solved in 9.514541 seconds**



---

*Published with MATLAB® R2021b*