ROB521 A1 - PRM Maze Solver

Ethan Rajah

February 28, 2025

1 Question 1

The first question involves the implementation of a PRM algorithm to solve a maze. The maze is represented as a 2D grid with obstacles and the goal is to construct a graph of nodes and edges that connects the start and end points of the maze. The graph construction is limited to a maximum of 500 samples and the number of edges per node is tuned as a hyperparameter. The PRM algorithm is implemented by uniformly sampling 500 points in the maze, filtering out points that are within 0.1 units of obstacles using the MinDist2Edges function, and connecting each remaining point (now milestone) to its k-nearest neighbors. The k-nearest neighbors are found using a squared distance metric and the edges are added only if they do not intersect with any obstacles (found using the CheckCollision function). I found that k=8 was the optimal number of neighbors to connect each node to for the 5x7 maze as it resulted in a good balance between computational efficiency and high probability in generating a path. A sample graph is shown below in **Figure 1**.

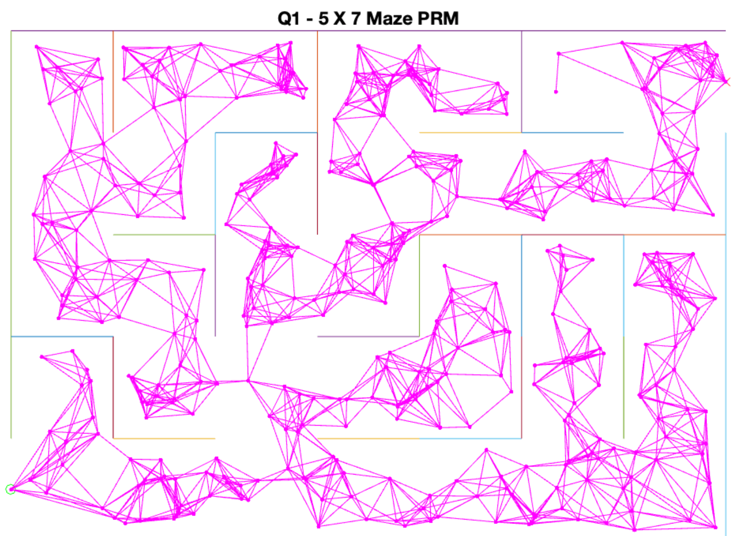

Figure 1: PRM graph generation example with k=8

2 Question 2

The second question involves the implementation of a path planning algorithm to find the shortest path using the PRM graph generated in the previous question. I chose to implement A* because of its efficiency and optimality in finding the shortest path. A* involves a cost metric that consists of a cost-to-come and a heurisitic cost-to-go. I chose to use the Euclidean distance between two nodes to assign edge costs for computing the cost-to-come. I also chose to use the Manhattan distance heurisitic for computing the lower bound cost estimate for going from the current node to the goal node. These metrics are commonly used in path planning algorithm and are efficient to compute for the 2D maze. The A* implementation involves defining a priority queue to store the nodes to be expanded along with their cost-to-come and total cost. The algorithm iteratively expands the node with the lowest total cost until the goal node is reached. When a node that has not been visited before is reached, the cost-to-come and total cost using the heurisitic are computed and the node is added to the priority queue. If a node that has been visited before is reached, the cost-to-come is updated if the new cost-to-come is lower than the previous cost-to-come. This ensures that the algorithm finds the optimal path. When the goal node is reached, any node in the priority queue with a cost-to-come greater than or equal to the cost-to-come of the goal node is pruned. The final path is then reconstructed by backtracking from the goal node to the start node. The parent node of each node is stored in a parent array during the search to facilitate the backtracking procedure. The path is shown below in **Figure 2**, which took on average 0.05s to complete.

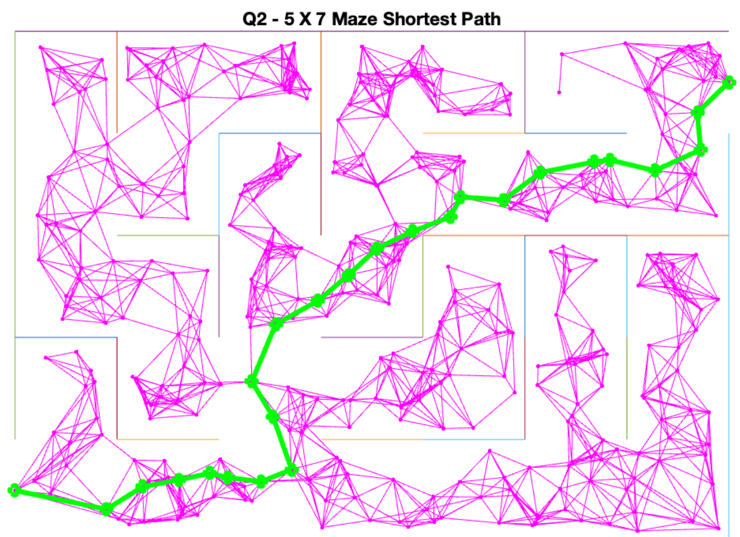

Figure 2: A* path planning example

3 Question 3

The third question involves solving large mazes using an optimized version of the algorithms implemented so far. I struggled to get 40x40 mazes completed within 20s, even with optimizations presented in class. In my best attempt, I used a non-uniform Gaussian sampling technique, specifically bridge sampling as its known to be useful for sampling small corners and narrow passages. The key idea with this approach is to sample two points within a Gaussian distribution and save the midpoint of the two points as a milestone if both points are in collision space, thus allowing for more efficient sampling in narrow passages, which is common in large mazes. With this, I also implemented a lazy collision checking approach to reduce the number of collision checks between nodes. This involves only checking for edge collisions when an edge is being considered during the A* search, rather than during the graph construction process. This is known to provide up to 10x speedups in path planning algorithms, as stated in lecture. Despite these optimizations, I was only able to complete 25x25 mazes within 9s on average, and 40x40 mazes within 35s. With 25x25 mazes, I found that k=15 and 2500 samples were optimal for consistently finding a path while also aiming to minimize computational time. However, with 40x40 mazes, I found that k=20 and 6000 samples were required to have about a 75% success rate in finding a path. While further optimizations could be made, such as using KD-trees for nearest neighbor searches, I decided to try a simpler uniform sampling approach where the maze is divided into a grid and a point is sampled from each grid cell. This approach was able to complete 45x45 mazes within 20s on average, when using lazy collision checking and k=4 (in this case the number of samples is based on the size of the maze). An example path is shown below in **Figure 3**.

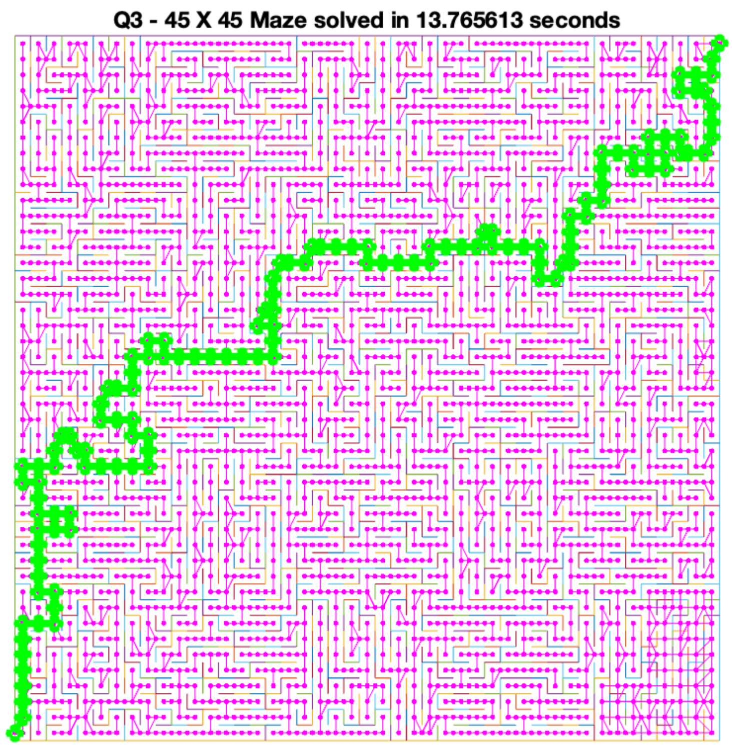

Figure 3: Optimized PRM and A* path planning example

Note that since lazy collision checking is used, the plot shows edges that intersect with obstacles. This is because if an edge is not explored during the A* search, it is not checked for collisions and thus is not removed from the graph. This can be seen in **Figure 3** in the top left and bottom right corners, where there is a higher density of edges resulting from violating edges that were not checked for collisions as A* did not explore them.

4 MATLAB Code

The MATLAB code for these implementations are provided here, as well as are attached in the submission.

```
% ======
% 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-------

% PRM algorithm: sample a point, check if it's at least 0.1 units
```

```
away from walls (collision check), if good, add to milestones
% then connect milestone to k nearest neighbors for all edges that are not in collision. k is a tuned hyperparameter
k = 8;
% Sample 500 points
x_pts = rand(nS,1) * col + 0.5;
y_pts = rand(nS,1) * row + 0.5;
samples = [x_pts, y_pts];
% Check if points are at least 0.1 units away from walls. min_distances is 1x500 vector of minimum distances to walls for each point
min_distances = MinDist2Edges(samples, map);
% Get indices of valid samples that are at least 0.1 units away from walls
valid_sample_idxs = find(min_distances > 0.1);
% Save the valid index rows of sample points to milestones
milestones = [milestones; samples(valid_sample_idxs,:)];
% Connect each milestone to k nearest neighbors
for i = 1:length(milestones)
    % Compute euclidean distances between all milestones to get k nearest neighbors
    distances = sqrt(sum((milestones - milestones(i,:)).^2,2));
    % Sort and get k nearest indices, excluding the point itself
    [~, idx] = sort(distances);
    nearest = idx(2:k+1);
    % Check if potential edge is in collision with any walls
    for j = 1:length(nearest)
        if ~CheckCollision(milestones(i, :), milestones(nearest(j), :), map)
            edges = [edges; milestones(i,:), milestones(nearest(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-------

% A* algorithm: use a cost to come and heuristic cost to go to find the shortest path.
% Cost to come is the sum of the edge costs from the start to the current node.
% Heuristic cost to go is the manhattan distance from the current node to the goal.
edge_costs = sqrt(sum((edges(:,1:2) - edges(:,3:4)).^2,2));
% Compute the heuristic cost to go for each milestone
heuristic_cost = sum(abs(milestones - finish),2);
% Initialize priority queue of tuples (node_idx, cost_to_come, total_cost) - 3 columns
pq = [1, 0, heuristic_cost(1)];
% Initialize visited set - 1 if visited, 0 if not
visited = zeros(length(milestones),1);
visited(1) = 1;
% Initialize array to store (parent_idx, cost_to_come) for each node
parent = inf * ones(length(milestones),2);

while ~isempty(pq)
    % Select the node with the minimum total cost from the priority queue
    [~, idx] = min(pq(:,3));
    node_idx = pq(idx,1);
    node_x = milestones(node_idx,1);
    node_y = milestones(node_idx,2);
    cost_to_come = pq(idx,2);
    total_cost = pq(idx,3);
    % Remove the node from the priority queue
    pq(idx,:) = [];
    % Check if the node is the goal
    if node_idx == 2
        % Goal reached, prune the priority queue based on if node cost is greater than goal cost
```

```
        pq = pq(pq(:,3) < total_cost,:);
    end
    % Get the neighbor indices of the current node
    neighbors_out = find(edges(:,1) == node_x & edges(:,2) == node_y);
    neighbors_in = find(edges(:,3) == node_x & edges(:,4) == node_y);
    neighbors = [neighbors_out; neighbors_in];
    for i = 1:length(neighbors)
        if ismember(neighbors(i), neighbors_out)
            neighbor_x = edges(neighbors(i),3);
            neighbor_y = edges(neighbors(i),4);
        else
            neighbor_x = edges(neighbors(i),1);
            neighbor_y = edges(neighbors(i),2);
        end
        % Get neighbor index in milestones
        neighbor_idx = find(milestones(:,1) == neighbor_x & milestones(:,2) == neighbor_y);
        % Check if neighbor has been visited
        if visited(neighbor_idx) == 0
            % Add neighbor to visited set
            visited(neighbor_idx) = 1;
            % Compute the cost to come for the neighbor
            new_cost_to_come = cost_to_come + edge_costs(neighbors(i));
            % Compute the total cost for the neighbor
            new_total_cost = new_cost_to_come + heuristic_cost(neighbor_idx);
            % Add the neighbor to the priority queue
            pq = [pq; neighbor_idx, new_cost_to_come, new_total_cost];
            % Update the parent array with the new parent, cost to come
            % This runs if the neighbor node has not been visited, so its previous cost to come is inf
            parent(neighbor_idx,:) = [node_idx, new_cost_to_come];
        else
            % Handle case where neighbor has been visited but new cost to come is less than previous cost to come
            if cost_to_come + edge_costs(neighbors(i)) < parent(neighbor_idx,2)
                % Add the neighbor back to the priority queue with the new cost to come
                new_cost_to_come = cost_to_come + edge_costs(neighbors(i));
                new_total_cost = new_cost_to_come + heuristic_cost(neighbor_idx);
                pq = [pq; neighbor_idx, new_cost_to_come, new_total_cost];
                % Update the parent array with the new parent, cost
```

```
                    to come
                parent(neighbor_idx,:) = [node_idx, new_cost_to_come];
            end
        end
    end
end

% Reconstruct the shortest path from the parent array
spath = [2];
while spath(1) ~= 1
    if spath(1) == inf
        disp("No path found. Please rerun the script.");
        break;
    end
    spath = [parent(spath(1),1), spath];
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 = 45;
col = 45;
```

```
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]
spath = [];

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;
% Save maze plot prior to PRM generation
print -dpng assignment1_q3_maze.png

fprintf("Attempting large %d X %d maze... \n", row, col);
tic;
% ------insert your optimized algorithm here------

% nS = 6000; % 2500 best for 25x25 maze, Gaussian method
% k = 20; % 15 best for 25x25 maze, Gaussian method
% sigma = 0.5;
% % Lavalle Gaussian Sampling
% x = (randi([4, 4*col], nS, 1))/4;
% y = (randi([4, 4*row], nS, 1))/4;
% initial_samples = [x, y];
% x_gauss = x + sigma * randn(nS, 1);
% y_gauss = y + sigma * randn(nS, 1);
% % Clip samples to be within maze bounds
% x_gauss = max(min(x_gauss, col + 0.5), 0.5);
% y_gauss = max(min(y_gauss, row + 0.5), 0.5);
% gauss_samples = [x_gauss, y_gauss];
% % Calculate midpoints
% midpoints = [(x + x_gauss)/2, (y + y_gauss)/2];
% % Initialize samples array
% all_samples = [initial_samples; gauss_samples];
% % Compute distances
% min_distances_all = MinDist2Edges([initial_samples; gauss_samples;
     midpoints], map);
% min_distances_initial = min_distances_all(1:nS);
% min_distances_gauss = min_distances_all(nS+1:2*nS);
% min_distances_midpoint = min_distances_all(2*nS+1:end);
% valid_samples = [];
% for i = 1:nS
%     % Skip if samples already exist
%     if ~isempty(valid_samples)
%         if ismember(initial_samples(i,:), valid_samples, 'rows')
    || ...
%             ismember(gauss_samples(i,:), valid_samples, 'rows')
```

```
%               continue;
%           end
%       end
%       % Bridge sampling - if pair of uniform and gaussian samples are both in collision, add midpoint, otherwise only add one of the two
%       if min_distances_initial(i) > 0.1 && min_distances_gauss(i) > 0.1 && min_distances_midpoint(i) > 0.1
%           valid_samples = [valid_samples; midpoints(i,:)];
%       elseif min_distances_initial(i) > 0.1 && min_distances_gauss(i) > 0.1
%           valid_samples = [valid_samples; initial_samples(i,:)];
%       elseif min_distances_initial(i) > 0.1
%           valid_samples = [valid_samples; initial_samples(i,:)];
%       elseif min_distances_gauss(i) > 0.1
%           valid_samples = [valid_samples; gauss_samples(i,:)];
%       elseif min_distances_midpoint(i) > 0.1
%           valid_samples = [valid_samples; midpoints(i,:)];
%       end
% end
% Save the valid index rows of sample points to milestones
% milestones = [milestones; valid_samples];
% % Connect each milestone to k nearest neighbors
% for i = 1:length(milestones)
%     % Compute euclidean distances between all milestones to get k nearest neighbors
%     distances = sqrt(sum((milestones - milestones(i,:)).^2,2));
%     % Sort and get k nearest indices, excluding the point itself
%     [~, idx] = sort(distances);
%     nearest = idx(2:k+1);
%     % Lazy collision check: only check in A* algorithm if edge is in collision
%     for j = 1:length(nearest)
%         edges = [edges; milestones(i,:), milestones(nearest(j),:)];
%     end
% end

% Grid Approach: divide maze into grid cells using uniform sampling and define points for each cell
x_pts = 0.5:0.5:col+0.5;
y_pts = 0.5:0.5:row+0.5;
% Sample points for each cell
samples = [];
for i = 1:length(x_pts)
    for j = 1:length(y_pts)
        % Only save every other point to reduce number of samples
        if mod(j,2) == 0
            samples = [samples; x_pts(i), y_pts(j)];
        end
    end
```

```
end
% Check distances
min_distances = MinDist2Edges(samples, map);
% Get indices of valid samples that are at least 0.1 units away from walls
valid_sample_idxs = find(min_distances > 0.1);
% Save the valid index rows of sample points to milestones
milestones = [milestones; samples(valid_sample_idxs,:)];
% Connect each milestone to k nearest neighbors
k = 4;
for i = 1:length(milestones)
    % Compute euclidean distances between all milestones to get k nearest neighbors
    distances = sqrt(sum((milestones - milestones(i,:)).^2,2));
    % Sort and get k nearest indices, excluding the point itself
    [~, idx] = sort(distances);
    nearest = idx(2:k+1);
    % Lazy collision check: only check in A* algorithm if edge is in collision
    for j = 1:length(nearest)
        edges = [edges; milestones(i,:), milestones(nearest(j),:)];
    end
end

edge_costs = sqrt(sum((edges(:,1:2) - edges(:,3:4)).^2,2));
% Compute the heuristic cost to go for each milestone
heuristic_cost = sum(abs(milestones - finish),2);
% Initialize priority queue of tuples (node_idx, cost_to_come, total_cost) - 3 columns
pq = [1, 0, heuristic_cost(1)];
% Initialize visited set - 1 if visited, 0 if not
visited = zeros(length(milestones),1);
visited(1) = 1;
% Initialize array to store (parent_idx, cost_to_come) for each node
parent = inf * ones(length(milestones),2);
% Initialize edges to remove mask for lazy collision checking
invalid_edges = false(size(edges,1), 1);

while ~isempty(pq)
    % Select the node with the minimum total cost from the priority queue
    [~, idx] = min(pq(:,3));
    node_idx = pq(idx,1);
    node_x = milestones(node_idx,1);
    node_y = milestones(node_idx,2);
    cost_to_come = pq(idx,2);
    total_cost = pq(idx,3);
    % Remove the node from the priority queue
    pq(idx,:) = [];
    % Check if the node is the goal
    if node_idx == 2
```

```matlab
        % Goal reached, prune the priority queue based on if node cost is greater than goal cost
        pq = pq(pq(:,3) < total_cost,:);
    end
    % Get the neighbor indices of the current node
    neighbors_out = find(edges(:,1) == node_x & edges(:,2) == node_y);
    neighbors_in = find(edges(:,3) == node_x & edges(:,4) == node_y);
    neighbors = [neighbors_out; neighbors_in];
    for i = 1:length(neighbors)
        edge_idx = neighbors(i);
        if ismember(neighbors(i), neighbors_out)
            neighbor_x = edges(neighbors(i),3);
            neighbor_y = edges(neighbors(i),4);
        else
            neighbor_x = edges(neighbors(i),1);
            neighbor_y = edges(neighbors(i),2);
        end
        % % Check if edge is in collision
        [inCollision, ~] = CheckCollision([node_x, node_y], [neighbor_x, neighbor_y], map);
        if inCollision
            invalid_edges(edge_idx) = true;
            continue;
        end
        % Get neighbor index in milestones
        neighbor_idx = find(milestones(:,1) == neighbor_x & milestones(:,2) == neighbor_y);
        neighbor_idx = neighbor_idx(1); % In case of duplicate points
        % Check if neighbor has been visited
        if visited(neighbor_idx) == 0
            % Add neighbor to visited set
            visited(neighbor_idx) = 1;
            % Compute the cost to come for the neighbor
            new_cost_to_come = cost_to_come + edge_costs(neighbors(i));
            % Compute the total cost for the neighbor
            new_total_cost = new_cost_to_come + heuristic_cost(neighbor_idx);
            % Add the neighbor to the priority queue
            pq = [pq; neighbor_idx, new_cost_to_come, new_total_cost];
            % Update the parent array with the new parent, cost to come
            % This runs if the neighbor node has not been visited, so its previous cost to come is inf
            parent(neighbor_idx,:) = [node_idx, new_cost_to_come];
        else
            % Handle case where neighbor has been visited but new
```

```
                cost to come is less than previous cost to come
            if cost_to_come + edge_costs(neighbors(i)) < parent(neighbor_idx,2)
                % Add the neighbor back to the priority queue with the new cost to come
                new_cost_to_come = cost_to_come + edge_costs(neighbors(i));
                new_total_cost = new_cost_to_come + heuristic_cost(neighbor_idx);
                pq = [pq; neighbor_idx, new_cost_to_come, new_total_cost];
                % Update the parent array with the new parent, cost to come
                parent(neighbor_idx,:) = [node_idx, new_cost_to_come];
            end
        end
    end
end

% Remove edges that are in collision using mask
edges = edges(~invalid_edges,:);

% Reconstruct the shortest path from the parent array
spath = [2];
while spath(1) ~= 1
    if spath(1) == inf
        disp(size(visited));
        disp(size(visited(visited == 1)));
        disp("No path found. Please rerun the script.");
        break;
    end
    spath = [parent(spath(1),1), spath];
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
```