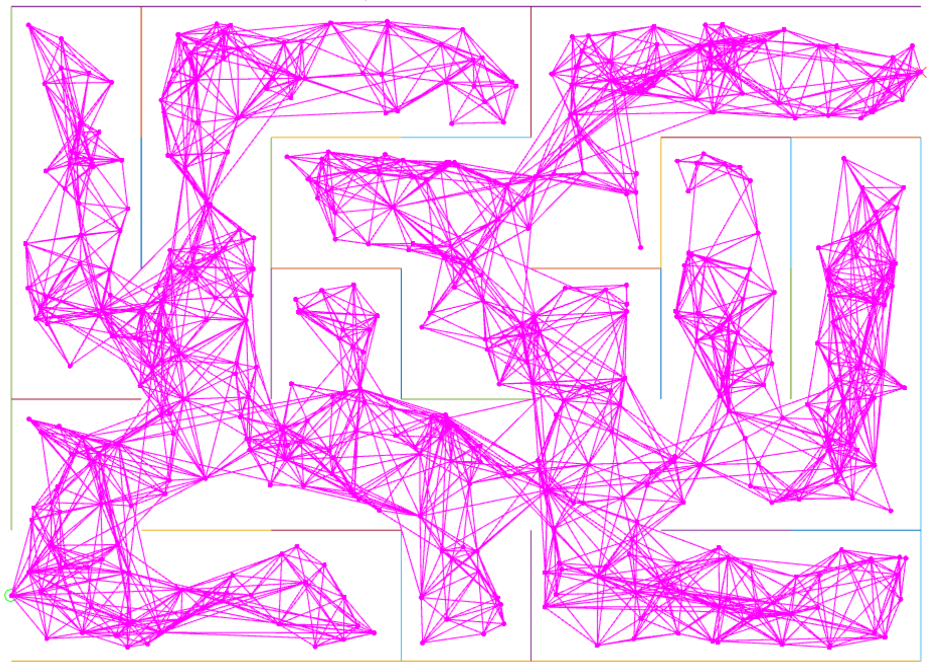
Q1 - 5 X 7 Maze PRM

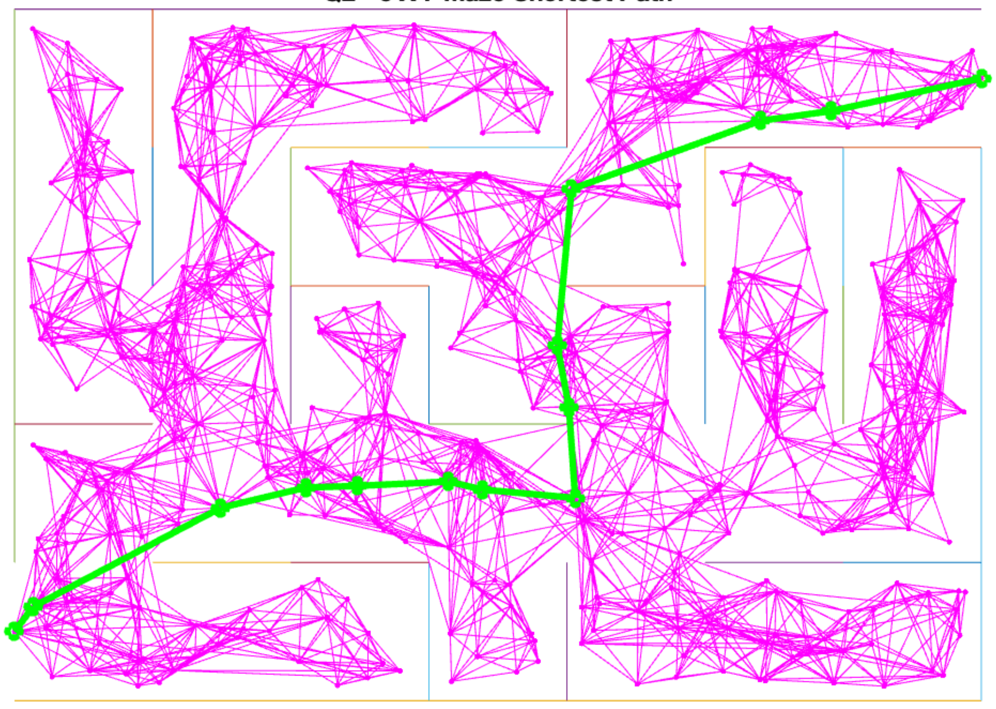


Nothing too interesting here. We do see some edges that go very close to walls because there is no restriction on how close an edge can be as long as it does not pass through. The walls are 2D so they are clearly not accounting for the area the walls takes up which likely means this robot in reality would run into the walls.

We attempt to connect each node with its 8 nearest neighbors and generate nodes until there are 500 valid nodes in the maze. We find that generally around 20% of generated nodes are invalid.

We also profile the script and find that edge checking is a substantial time sink, but the `MinDist2Edges` is taking up the largest amount of time. This runs counter to our usual assumption that sampling is fast and edge checking is slow. This will be addressed in the optimized solution.

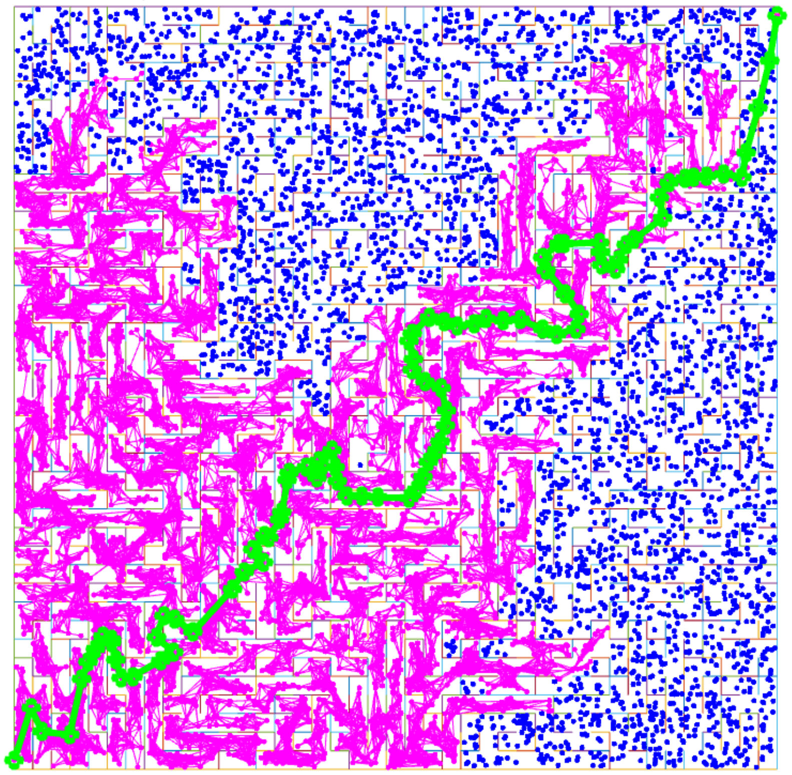
Q2 - 5 X 7 Maze Shortest Path



In question 2 we use A* with a euclidean distance heuristic. This is clearly admissible since the shortest path to the goal is a straight line. We use a linear time priority queue instead of logarithmic for simplicity of implementation. Profiling the script we find that the runtime is still dominated by edge checks so there is no need to implement a more complex priority queue. For our dead set, parent map, and cost-to-come map, we use constant time set and lookup data structures.

Searching for neighbors is inefficient in this scheme because it requires doing a scan over all edges and selecting those where the current node is one of the endpoints. We should have used an adjacency matrix for constant time neighbor lookup.

Q3 - 41 X 41 Maze solved in 8.812794 seconds

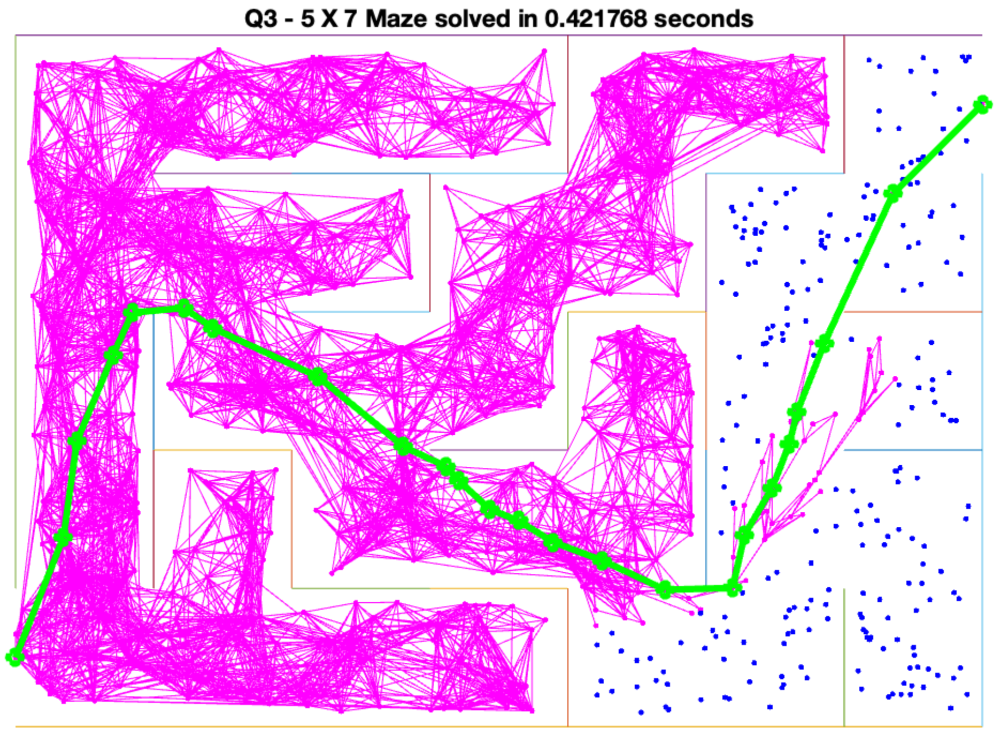


For solving a 41x41 maze we use 8000 valid nodes which seems to relatively consistently find a solution. It generally takes between 10 and 15 seconds to find a solution. This case showed off a number of optimizations well though so I selected it even though it has a lower than normal completion time. Here, magenta nodes are those that A* explored, blue remain unexplored, and collision checked edges are in magenta. We see that unexplored nodes do not have collision checked edges because we opted for lazy collision checking.

The changes from the original implementation are:

1. Lazy collision checking. Only check if edges collide with walls when A* is exploring that path.
2. Use an occupancy map at 0.1m resolution instead of checking distance to walls. This converts an expensive O(E) task to a O(1) task.
3. Use an adjacency list instead of an edges list. With an edges list we need to search through all edges to find the neighbors, but with an adjacency list it is an O(1) lookup. We still construct an edges list when A* is running, but it only includes edges that we explored.

These changes address the three issues we identified when profiling the script in previous sections. The number of edge collision checks is reduced by using a lazy strategy. We no longer use `MinDist2Edges` opting instead for a constant time lookup occupancy map. We only need to iterate over the edges once now instead of every time we add a new node. Finally, neighbor lookups use a O(1) call to the adjacency list instead of scanning over all edges.



Just for fun, here is the optimized version solving a small maze. You can clearly see that once A* finds an exploitable path, the number of edge collision checks goes way down.

Code:

```
% ======
% 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; %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-------
% Steps:
```

```
% 1. Generate sample
% 2. Check for sample validity. If not valid, go back to 1.
% 3. For each of the k closest neighbors
% 3.1 Check if line between new point and neighbor intersects with an edge
% 3.2 If no intersection, add the edge to the edges list
c = 1;
r = 0;  % Keeping track of the number of rejected milestones
a = 0;  % Keeping track of the number of accepted milestones
k = 8;  % Use the top k neighbors for connection attempts
while a < nS
   % Draw a sample uniformly x in [0.5, col+0.5] y in [0.5, row+0.5]
   x = 0.5 + rand(1) * col;
   y = 0.5 + rand(1) * row;
   new_point = [x y];
   c = c+1;
   % Check for validitiy of the point using MinDist2Edges
   d = MinDist2Edges(new_point, map);
   if d <= 0.1
       % Then we reject the milestone for being too close to an edge
       r = r+1;
       continue
   end
   % Then we accept the new milestone and need to check for edges
   a = a+1;
   milestones = [milestones; new_point];
   % Compute a vector where each element is a squared distance to the
   % corresponding already existing milestone
   diff = milestones - new_point;
   squared_distances = sum(diff.^2, 2);
   % Then we get the top k neighbors
   [D, I] = mink(squared_distances, k);
   for i_index = 1:size(I)
       m_index = I(i_index);
       con_point = milestones(m_index, :);
       % Now we have our proposed point and a point to attempt to connect
       % it to. All we need to do is check if there is a wall collision
       % for the edge that connects them.
       [ inCollision, edge ] = CheckCollision(new_point, con_point, map);
       if ~inCollision
           % Then we are free to add the edge
           % The graph is undirected so we could add two for going both
           % ways, but we will assume we can just add one and further code
           % will account for this
           edges = [edges; new_point con_point];
       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-------
% We deal in node indices to avoid comparing float vectors
% Steps:
% 1. Initialize variables
% 1.1 Create the priority queue and initialize with the start milestone
% 1.2 Initialize the dead hash map that we will be using to check whether
%     we should expand a node
% 1.3 Initialize the parent map that we will use to recover the path
pq = MinPriorityQueue();
pq = pq.add(calcHeuristic(start, finish), [1, -1]);  % The start node is at index 1
with parent -1
dead_set = false(1, size(milestones, 1));
% So dead_set(n) being true means that node has already been visited
parent_map = ones(1, size(milestones, 1)) * -1;
% parent_map(n) returns the parent of milestone index n
cost_to_come_map = ones(1, size(milestones, 1)) * -1;
cost_to_come_map(1) = 0;
% We initialize all cost-to-come values to -1 and then set the start to
% have a cost-to-come of 0
while ~pq.isEmpty()
   % Pop the lowest priority node
   [priority, cur_node_info, pq] = pq.pop();
   cur_node_index = cur_node_info(1);
   cur_node_parent_index = cur_node_info(2);
   cur_node = milestones(cur_node_index, :);
   if cur_node_parent_index >= 1
       parent_node = milestones(cur_node_parent_index, :);
```

```
    else
        parent_node = [-1, -1];
    end
    % disp(["Processing node" cur_node_index cur_node]);
    % Then we check if this node has been processed. Since the heuristic is
    % consistent, if the node has been processed we can skip it
    if dead_set(cur_node_index)
        continue
    end
    % Now we have processed it so we can set it to dead
    dead_set(cur_node_index) = true;
    if cur_node_index ~= 1
        % We can also now update the parent since we know this is the best
        % route to the current node
        parent_map(cur_node_index) = cur_node_parent_index;
        % At this point we can also update the cost to come
        cost_to_come_map(cur_node_index) = cost_to_come_map(cur_node_parent_index) +
getEdgeCost(parent_node, cur_node);
    end
    if cur_node_index == 2
        % The finish is always the second element of the milestone list
        break;
    end
    % Otherwise we need to iterate over the neighbors
    neighbors = getNeighbors(cur_node, edges);
    for n_index = 1:size(neighbors, 1)
        % For each neighbor, we now compute the node info and the estimated
        % total cost and add it into the priority queue
        neighbor = neighbors(n_index, :);
        neighbor_index = getNodeIndex(neighbor, milestones);
        % We could check if the neighbor is dead and skip it if it is
        % Create a new node info. The current node is now the parent
        node_info = [neighbor_index, cur_node_index];
        % Then we need to get the priority
        cost_to_come = cost_to_come_map(cur_node_index) + getEdgeCost(cur_node,
neighbor);
        estimated_total_cost = cost_to_come + calcHeuristic(neighbor, finish);
        pq = pq.add(estimated_total_cost, [neighbor_index, cur_node_index]);
    end
end
if parent_map(2) == -1
    % Then there was no path found
    error("No path found")
end
% Otherwise, we can trace back through the parents to find the path
parent_index = 2;
while parent_index ~= -1
    spath = [parent_index spath];
```

```
   parent_index = parent_map(parent_index);
end
% disp(spath)
% ------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------
% Changes:
% 1. Lazy collision checking. Only check if edges collide with walls when
% A* is exploring that path.
% 2. Use an occupation map instead of checking distance to walls. This
% converts an expensive O(E) task to a O(1) task.
% 3. Use an adjacency list instead of an edges list. With an edges list we
% need to search through all edges to find the neighbors, but with an
% adjacency list it is an O(1) lookup. We still construct an edges list
% when A* is running, but it only includes edges that we explored.
```

```
nS = 8000;  % We use this to be the number of valid nodes so that we can pre-allocate
our adjacency list
% An adjacency map will be efficient here as the number of neighbors is
% limited to be small
adjacency_list = cell(nS, 1);
% An occupancy map will allow us to do efficient checking for being within
% the allowed area
occ_map = build_occ_map(col, row, map);
c = 1;
r = 0;  % Keeping track of the number of rejected milestones
a = size(milestones, 1);  % Keeping track of the number of accepted milestones
k = 12;  % Use the top k neighbors for connection attempts
while a < nS
   % Draw a sample uniformly x in [0.5, col+0.5] y in [0.5, row+0.5]
   new_point = drawPoint(col, row);
   c = c+1;
   % Check that the drawn point is not already occupied
   occ_x = floor(new_point(1) * 10);
   occ_y = floor(new_point(2) * 10);
   occ = occ_map(occ_x, occ_y);
   if occ
       % Then we reject the milestone for being too close to an edge
       r = r+1;
       continue
   end
   % Then we accept the new milestone and need to check for edges
   a = a+1;
   new_point_index = size(milestones, 1) + 1;
   I = getKClosest(new_point, milestones, k);
   % This does collision checking and adds the edges
   % Except that we do lazy collision checking now so it actually just
   % passes all edges through
   adjacency_list = update_adjacency(adjacency_list, I, new_point_index, milestones,
map);
   % Append to the milestones afterward to avoid connecting the node to
   % itself
   milestones = update_milestones(milestones, new_point);
end
toc;
spath = [];
pq = MinPriorityQueue();
pq = pq.add(calcHeuristic(start, finish), [1, -1]);  % The start node is at index 1
with parent -1
dead_set = false(1, size(milestones, 1));
% So dead_set(n) being true means that node has already been visited
parent_map = ones(1, size(milestones, 1)) * -1;
% parent_map(n) returns the parent of milestone index n
cost_to_come_map = ones(1, size(milestones, 1)) * -1;
```

```
cost_to_come_map(1) = 0;
% We initialize all cost-to-come values to -1 and then set the start to
% have a cost-to-come of 0
while ~pq.isEmpty()
   % Pop the lowest priority node
   [priority, cur_node_info, pq] = pq.pop();
   cur_node_index = cur_node_info(1);
   cur_node_parent_index = cur_node_info(2);
   cur_node = milestones(cur_node_index, :);
   if cur_node_parent_index >= 1
       parent_node = milestones(cur_node_parent_index, :);
       % Check for collision
       [ inCollision, edge ] = CheckCollision(cur_node, parent_node, map);
       if inCollision
           continue
       end
       % Add the new edge for visualization
       % We only visualize the ones we checked and found no collision
       edges = [edges; parent_node cur_node];
   else
       parent_node = [-1, -1];
   end
   % Then we check if this node has been processed. Since the heuristic is
   % consistent, if the node has been processed we can skip it
   if dead_set(cur_node_index)
       continue
   end
   % Now we have processed it so we can set it to dead
   dead_set(cur_node_index) = true;
   if cur_node_index ~= 1
       % We can also now update the parent since we know this is the best
       % route to the current node
       parent_map(cur_node_index) = cur_node_parent_index;
       % At this point we can also update the cost to come
       cost_to_come_map(cur_node_index) = cost_to_come_map(cur_node_parent_index) +
getEdgeCost(parent_node, cur_node);
   end
   if cur_node_index == 2
       % The finish is always the second element of the milestone list
       disp("Found finish")
       break;
   end
   % Otherwise we need to iterate over the neighbors
   % OLD
   % neighbors = getNeighbors(cur_node, edges);
   % NEW
   neighbor_indices = getNeighborsAdjacency(cur_node_index, adjacency_list);
   for n_index = 1:size(neighbor_indices, 2)
```

```
        % For each neighbor, we now compute the node info and the estimated
        % total cost and add it into the priority queue
        % OLD
        % neighbor = neighbors(n_index, :);
        % neighbor_index = getNodeIndex(neighbor, milestones);
        % NEW
        neighbor_index = neighbor_indices(n_index);
        neighbor = milestones(neighbor_index, :);
        % We could check if the neighbor is dead and skip it if it is
        if dead_set(neighbor_index)
            continue
        end
        % Create a new node info. The current node is now the parent
        node_info = [neighbor_index, cur_node_index];
        % Then we need to get the priority
        cost_to_come = cost_to_come_map(cur_node_index) + getEdgeCost(cur_node,
neighbor);
        estimated_total_cost = cost_to_come + calcHeuristic(neighbor, finish);
        pq = pq.add(estimated_total_cost, [neighbor_index, cur_node_index]);
    end
end
if parent_map(2) == -1
    % Then there was no path found
    error("No path found")
end
% Otherwise, we can trace back through the parents to find the path
parent_index = 2;
while parent_index ~= -1
    spath = [parent_index spath];
    parent_index = parent_map(parent_index);
end
% ------end of your optimized algorithm-------
toc;
dt = toc;
% Get the indices of all nodes in the dead set and not in the dead set
dead_indices = find(dead_set);
alive_indices = find(~dead_set);
figure(2); hold on;
% plot(milestones(:,1),milestones(:,2),'m.');
plot(milestones(dead_indices,1),milestones(dead_indices,2),'m.');
plot(milestones(alive_indices,1),milestones(alive_indices,2),'b.');
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
% ------functions-------
function h = calcHeuristic(cur_point, goal_point)
   h = sqrt(sum((cur_point - goal_point) .^ 2));
end
function neighbors = getNeighbors(point, edges)
   neighbors = [];
   for edge_index = 1:size(edges, 1)
       % Check if one of the endpoints is the point
       edge = edges(edge_index, :);
       if point(1) == edge(1) && point(2) == edge(2)
           neighbors = [neighbors; edge(3) edge(4)];
       end
       if point(1) == edge(3) && point(2) == edge(4)
           neighbors = [neighbors; edge(1) edge(2)];
       end
   end
end
function neighbor_indices = getNeighborsAdjacency(point_index, adjacency_list)
   neighbor_indices = adjacency_list{point_index};
end
function index = getNodeIndex(point, milestones)
   for node_index = 1:size(milestones, 1)
       milestone = milestones(node_index, :);
       if point(1) == milestone(1) && point(2) == milestone(2)
           index = node_index;
           return
       end
   end
   index = -1;
end
function cost = getEdgeCost(e1, e2)
   cost = sqrt(sum((e1 - e2) .^ 2));
end
function point = drawPoint(col, row)
   x = 0.5 + rand(1) * col;
   y = 0.5 + rand(1) * row;
   point = [x y];
end
function k_closest_indices = getKClosest(point, milestones, k)
   % Compute a vector where each element is a squared distance to the
   % corresponding already existing milestone
   diff = milestones - point;
   squared_distances = sum(diff.^2, 2);
   % Then we get the top k neighbors
   [D, k_closest_indices] = mink(squared_distances, k);
end
function adjacency_list = update_adjacency(adjacency_list, neighbor_indices,
new_point_index, milestones, map)
   for i_index = 1:size(neighbor_indices)
       m_index = neighbor_indices(i_index);
       con_point = milestones(m_index, :);
       % Now we have our proposed point and a point to attempt to connect
       % it to. All we need to do is check if there is a wall collision
       % for the edge that connects them.
       % [ inCollision, edge ] = CheckCollision(new_point, con_point, map);
       inCollision = false;
       if ~inCollision
           % Then we are free to add the edge
           % The graph is undirected so we could add two for going both
           % ways, but we will assume we can just add one and further code
           % will account for this
           % edges = [edges; new_point con_point];
           % And we also update the adjacency list to reflect the new edge
           adjacency_list{new_point_index} = [adjacency_list{new_point_index}
m_index];
           adjacency_list{m_index} = [adjacency_list{m_index} new_point_index];
       end
   end
end
function milestones = update_milestones(milestones, new_point)
   milestones = [milestones; new_point];
end
function edges = build_edges(adjacency_list, milestones)
   edges = [];
   for i = 1:size(milestones, 1)
       point = milestones(i, :);
       neighbor_indices = adjacency_list{i};
       for j = 1:size(neighbor_indices, 2)
           neighbor_index = neighbor_indices(j);
           n_point = milestones(neighbor_index, :);
           edges = [edges; point n_point];
       end
   end
end
function occ_map = build_occ_map(col, row, map)
   occ_map = false(10*col, 10*row);
   for edge_ind = 1:size(map, 1)
       edge = map(edge_ind, :);
       x1 = edge(1) * 10;
       x2 = edge(3) * 10;
       y1 = edge(2) * 10;
```

```
        y2 = edge(4) * 10;
        if x1 == x2
            % This is a vertical
            x = x1;
            for y=(y1-1):(y2)
                occ_map(x-1, y) = true;
                occ_map(x, y) = true;
            end
        elseif y1 == y2
            % This is a horizontal
            y = y1;
            for x=(x1-1):(x2)
                occ_map(x, y-1) = true;
                occ_map(x, y) = true;
            end
        else
            error("Edge is neither vertical nor horizontal");
        end
    end
end
```

Priority Queue Implementation

```
classdef MinPriorityQueue
    properties (Access = private)
        priorities % An array to store priorities
        elements % A cell array to store elements
    end

    methods
        function obj = MinPriorityQueue()
            % Constructor to create a new min priority queue
            obj.priorities = []; % Initialize as an empty array
            obj.elements = {}; % Initialize as an empty cell array
        end

        function isEmpty = isEmpty(obj)
            % Checks if the priority queue is empty
            isEmpty = isempty(obj.priorities);
        end

        function obj = add(obj, priority, element)
            % Adds a member to the priority queue
            obj.priorities = [obj.priorities; priority]; % Add priority to the
priorities array
            obj.elements = [obj.elements; {element}]; % Add element to the elements
cell array
        end
        function [priority, element, obj] = pop(obj)
            % Pops and returns the element with the lowest priority in the queue
            if isempty(obj.priorities)
                error('Priority queue is empty');
            end
            [priority, I] = min(obj.priorities); % Find the min priority and its index
            element = obj.elements{I}; % Retrieve the element with the lowest priority
            obj.priorities(I) = []; % Remove the priority from the array
            obj.elements(I) = []; % Remove the element from the cell array
        end
    end
end
```