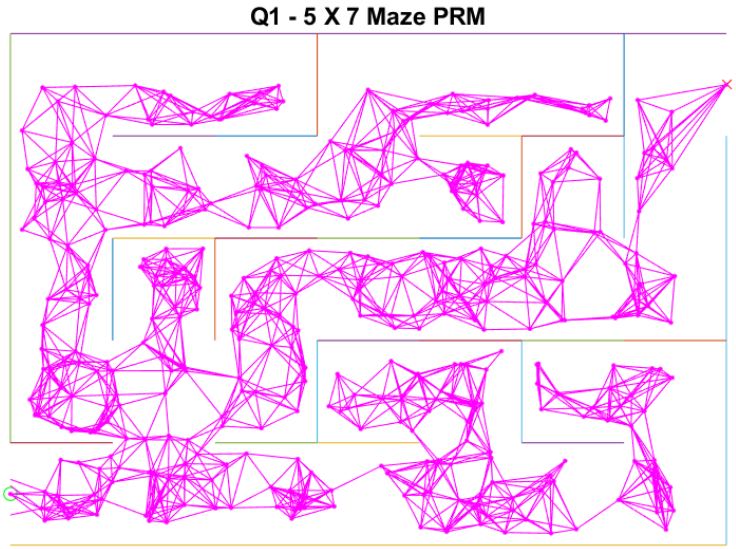
ROB521 Assignment 1: PRM and A* Planning

Andrew Jairam
1006941972
andrew.jairam@mail.utoronto.ca

Question 1: Mapping on a 5x7 Grid



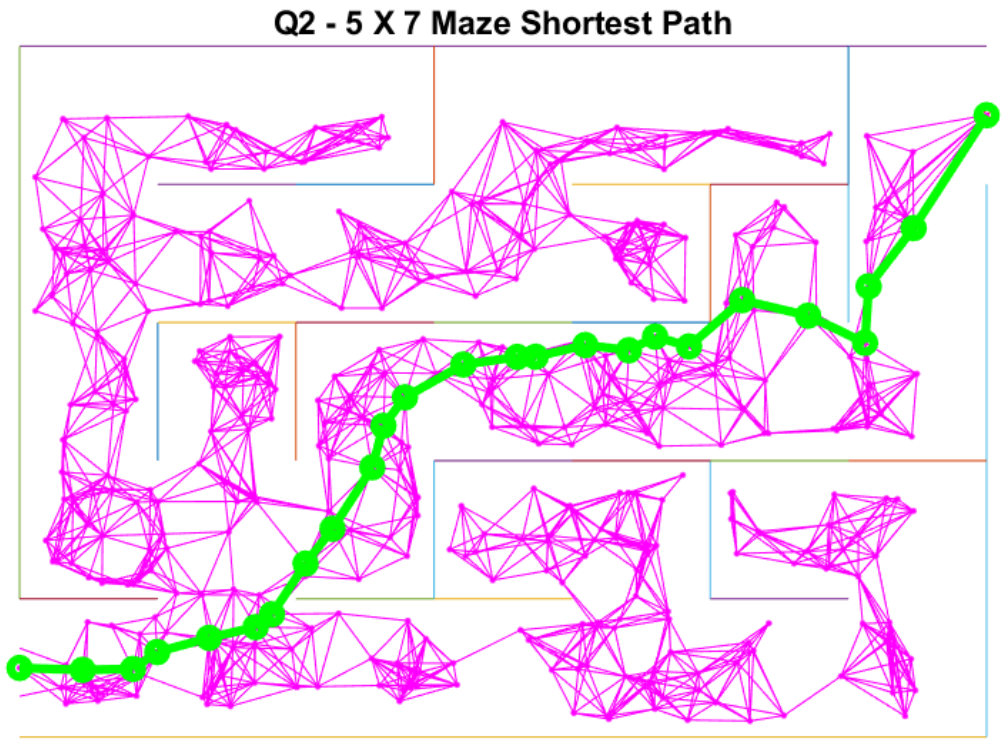
For this part, simple random sampling is used with a k = 10 nearest neighbour strategy to connect each node to 10 of its closest neighbours. Point generation stops when 500 valid points are placed onto the graph. Since there is a lot of open space, a highly visible graph is constructed. Graph generation is very quick, taking 0.225 seconds to place 500 valid points.

While successful, trying to randomly sample on the larger maps will not work: despite using many samples, the graph generated will rarely be complete, and a path from start to goal will not be formed. Additionally, as presented in lecture, the MinDist2Edges collision checking is very expensive, so this method would also be slow. A better sampling strategy should be the main optimization in the better algorithm since A* search is already the fastest search we studied.

```
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-------
% % Generate nS samples -> loop
countSamples = 0;
while countSamples < nS
    % Use rand to get [0, 1] sample, then get x, y coords multiplying by row,
col
    sample = [rand*col, rand*row];
    % Check min dist to edge more than 0.1, and if it is, add to milestones and
increment count
    if MinDist2Edges(sample, map) > 0.1
        milestones = [milestones; sample];
        countSamples = countSamples + 1;
    end
```

```
end

% Have our nS samples, connect them using k-NN
k = 10; % Number of nearest neighbours to connect: TUNABLE
for i = 1:length(milestones)
    % Get distance to all other milestones from this milestone: use euclidean
summed along row dim (dim 2 in matlab)
    distances = sqrt(sum((milestones - milestones(i,:)).^2, 2));
    % Then sort the distances -> use sort? -> get k nearest indices
    [sortedValues, idx] = sort(distances);
    % Bugfix: skip first idx, because itll be the same point (which will have
distance 0)
    kNearestIndices = idx(2:k+1);
    % Check for collisions, add to edges if no collision. CheckCollision = 1 if
collision, 0 no collision
    for j = 1:length(kNearestIndices)
        if CheckCollision(milestones(i,:), milestones(idx(j),:), map) == 0
            edges = [edges; milestones(i,:), milestones(idx(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: Optimal Search on a 5x7 Grid



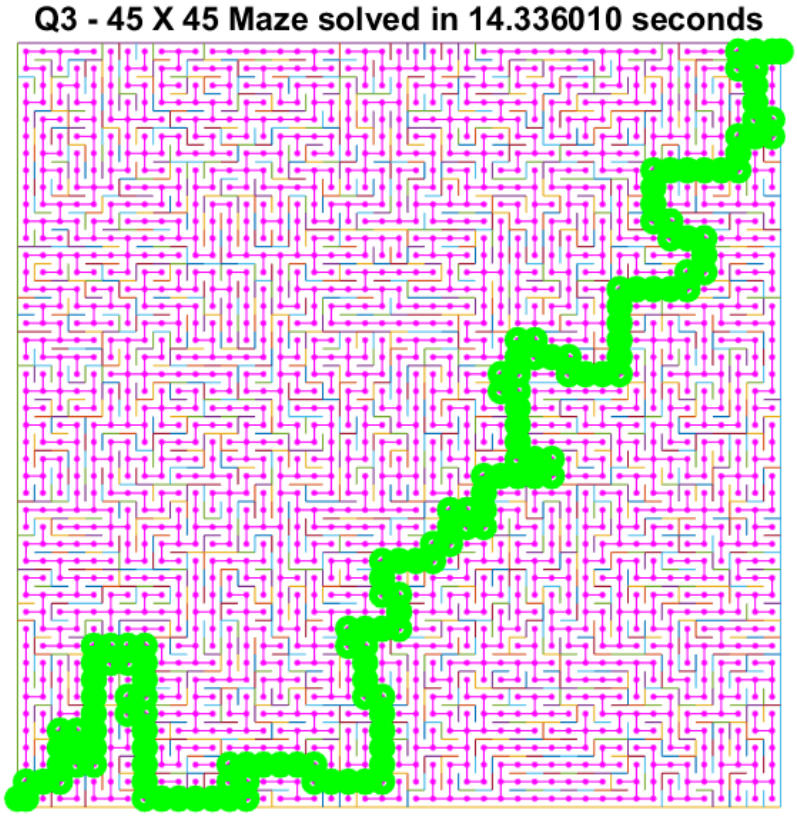
To conduct optimal search, we use A* search with Euclidean distance to the goal heuristic. This is an admissible heuristic since the robot can move freely in any direction. This method finds the shortest path on this small graph in 1.61 seconds, exploring less than 300 of the 500 dead nodes. Time is dominated by queue sorting, which takes linear time at each iteration. Potential optimizations for the next part include improving the queue mechanism by using a K-D tree (logarithmic time) or an adjacency matrix (O(1) time) to trade memory for runtime complexity, or using lazy collision checking to only check paths when we get to nodes, which would eliminate the need to check many of the collisions.

```
% ================================================================
% 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-------
% Heuristic function for A* search: use euclidean distance w/o obstacles
heuristic = @(nodeIdx) sqrt(sum((milestones(nodeIdx, :) - finish).^2, 2));
% NOTE: start node is milestones(1) (or start), finish node is milestones(2)
(or finish)

% Keep queue of alive nodes, and set of dead nodes
% FORMAT: [nodeIndex, costToCome, costToCome + heuristic, parentIndex].
Separate by ;
alive = [1, 0, heuristic(1), NaN]; % only alive is start node at start
dead = [];

iterationCounter = 0;
while ~isempty(alive)
    if mod(iterationCounter, 100) == 0
        fprintf("Iteration %d, alive nodes: %d, dead nodes: %d\n",
iterationCounter, size(alive, 1), size(dead, 1));
    end
    % Sort the alive nodes by cost to come + heuristic (index 3 of rows: use
sortwors)
    alive = sortrows(alive, 3);
    % Pop the first node from alive
    currentNode = alive(1, :);
    alive(1, :) = [];

    % IF currentNode is finish node, recover the path and break out of while
loop
    if currentNode(1) == 2  % finish index in milestones is 2
        spath = [currentNode(1)];
        parent = currentNode(4);
        while ~isnan(parent)
            % stack in reverse order
            spath = [parent; spath];
            % Find next parent from dead
            parent = dead(dead(:,1) == parent, 4);
        end
        break;
    end

    % Add currentNode to dead
    dead = [dead; currentNode];
```

```
    % Get neigbours of currentNode
    neighbours = [];
    for i = 1:size(edges, 1)
        % Edge format: [x1 y1 x2 y2], so want to check if either x1 y1 or x2 y2
is currentNode
        if isequal(edges(i, 1:2), milestones(currentNode(1), :))
            neighbourIdx = find(ismember(milestones, edges(i, 3:4), 'rows'));
            neighbours = [neighbours; neighbourIdx];
        elseif isequal(edges(i, 3:4), milestones(currentNode(1), :))
            neighbourIdx = find(ismember(milestones, edges(i, 1:2), 'rows'));
            neighbours = [neighbours; neighbourIdx];
        end
    end
    % Iterate through neighbours, add to alive if not in dead
    for i = 1:length(neighbours)
        neighbour = neighbours(i);
        % Check if neighbour is in dead
        if ismember(neighbour, dead(:, 1))
            continue
        end

        % Compute cost-to-come for neighbour: euclidean distance from
currentNode to neighbour?
        neighbourCostToCome = currentNode(2) + sqrt(sum((milestones(neighbour,
:) - milestones(currentNode(1), :)).^2, 2));

        % if node is in alive, check if new cost is lower, if so update. Else,
add to alive normally
        if ismember(neighbour, alive(:, 1))
            idx = find(alive(:, 1) == neighbour);
            if neighbourCostToCome < alive(idx, 2)
                alive(idx, 2) = neighbourCostToCome;
                alive(idx, 3) = neighbourCostToCome + heuristic(neighbour);
                alive(idx, 4) = currentNode(1);
            end
        else
            % if node is not in alive, add to alive
            alive = [alive; neighbour, neighbourCostToCome, neighbourCostToCome
+ heuristic(neighbour), currentNode(1)];
        end
    end
    iterationCounter = iterationCounter + 1;
end
```

Question 3: Optimized Mapping and Planning



The main problem of the last approach is that points are not sampled so that a complete path can even be found. Better sampling strategies presented in lecture were attempted (Gaussian/Bridge sampling to make samples between walls) but the connectivity problem persisted. A lot of points were attempted, which was also expensive to do, making the graph generation require substantial time.

Instead, the geometry of this maze can be exploited: walls are uniformly spaced at the resolution chosen. If the map is size 40x40, the walls will separate the space so that passageways are small 1xN segments. Thus, if we sample 40 points at linearly spaced intervals per row, we will get 1600 collision-free points that span throughout the graph, formulating planning for this maze as a grid search problem. There are a couple of benefits to this approach: mainly, collision checking does not have to be carried out at all in sampling. This also solves the earlier problem of sparse sub-graphs being generated, as one point will be guaranteed to be generated per 1x1 segment of the graph. To make edges, nodes can be connected if the distance between the nodes is less or equal to 1. k-nearest neighbours with $k = 8$ can be used to do this, since at most, each node would have eight neighbours: 4 in diagonal directions, and 4 in cartesian directions.

No further improvements to A* were needed, other than using the Manhattan distance to the goal heuristic for a tighter lower bound, making A* search faster. Manhattan distance is only an admissible heuristic because the larger map walls don't allow diagonal

connections, even though diagonal connections are checked for. On the 5x7 map, diagonal connections would be allowed since the walls don't separate the space the same, allowing more free space/visibility, and Manhattan distance would not be admissible. In this case, we should return to using the Euclidean distance to the goal as the heuristic. Thus, the generated graph must be carefully analyzed in advance to see which heuristic should be used.

```
% ================================================================
% 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]

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------

% GRID APPROACH: We can exploit the geometry of the maze to discretize the
space: this ensures only one point per grid cell, & all space is covered
rows = linspace(1, row, row);
cols = linspace(1, col, col);
% Create milestones: each row + each col to cover all space -> double loop. No
need to check for collisions due to map geometry
for i = 1:length(rows)
    for j = 1:length(cols)
```

```
        milestones = [milestones; [cols(j), rows(i)]];
    end
end

% No more kNN: only make edges to nodes that are adjacent. check the first 8
closest, because a node would have at most 8 neighbours
for i = 1:length(milestones)
    % Get distance to all other milestones from this milestone: use euclidean
summed along row dim (dim 2 in matlab)
    distances = sqrt(sum((milestones - milestones(i,:)).^2, 2));
    % Then sort the distances -> use sort? -> get k nearest indices
    [sortedValues, idx] = sort(distances);
    kNearestIndices = idx(2:9);
    % check if distance is 1, if so, add to edges
    for j = 1:length(kNearestIndices)
        if distances(kNearestIndices(j)) <= 1
            % check collision
            if CheckCollision(milestones(i,:),
milestones(kNearestIndices(j),:), map) == 0
                edges = [edges; milestones(i,:),
milestones(kNearestIndices(j),:)];
            end
        end
    end
end
disp(edges(1, :))

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


% ------insert your shortest path finding algorithm here-------
% We can now use manhattan for a tighter bound
heuristic = @(nodeIdx) sum(abs(milestones(nodeIdx, :) - finish), 2);
% NOTE: start node is milestones(1) (or start), finish node is milestones(2)
(or finish)

% Keep queue of alive nodes, and set of dead nodes
% FORMAT: [nodeIndex, costToCome, costToCome + heuristic, parentIndex].
Separate by ;
alive = [1, 0, heuristic(1), NaN]; % only alive is start node at start
dead = [];

iterationCounter = 0;
```

```
% A* SEARCH HERE

while ~isempty(alive)
    if mod(iterationCounter, 100) == 0
        fprintf("Iteration %d, alive nodes: %d, dead nodes: %d\n",
iterationCounter, size(alive, 1), size(dead, 1));
    end
    % Sort the alive nodes by cost to come + heuristic (index 3 of rows: use
sortwors)
    alive = sortrows(alive, 3);
    % Pop the first node from alive
    currentNode = alive(1, :);
    alive(1, :) = [];

    % IF currentNode is finish node, recover the path and break out of while
loop
    if currentNode(1) == 2  % finish index in milestones is 2
        spath = [currentNode(1)];
        parent = currentNode(4);
        while ~isnan(parent)
            % stack in reverse order
            spath = [parent; spath];
            % Find next parent from dead
            parent = dead(dead(:,1) == parent, 4);
        end
        break;
    end

    % Add currentNode to dead
    dead = [dead; currentNode];
    % Get neigbours of currentNode
    neighbours = [];
    for i = 1:size(edges, 1)
        % Edge format: [x1 y1 x2 y2], so want to check if either x1 y1 or x2 y2
is currentNode
        if isequal(edges(i, 1:2), milestones(currentNode(1), :))
            neighbourIdx = find(ismember(milestones, edges(i, 3:4), 'rows'));
            neighbours = [neighbours; neighbourIdx];
        elseif isequal(edges(i, 3:4), milestones(currentNode(1), :))
            neighbourIdx = find(ismember(milestones, edges(i, 1:2), 'rows'));
            neighbours = [neighbours; neighbourIdx];
        end
    end
    % Iterate through neighbours, add to alive if not in dead
    for i = 1:length(neighbours)
```

```
        neighbour = neighbours(i);
        % Check if neighbour is in dead
        if ismember(neighbour, dead(:, 1))
            continue
        end

        % Compute cost-to-come for neighbour: euclidean distance from
currentNode to neighbour?
        neighbourCostToCome = currentNode(2) + sqrt(sum((milestones(neighbour,
:) - milestones(currentNode(1), :)).^2, 2));

        % if node is in alive, check if new cost is lower, if so update. Else,
add to alive normally
        if ismember(neighbour, alive(:, 1))
            idx = find(alive(:, 1) == neighbour);
            if neighbourCostToCome < alive(idx, 2)
                alive(idx, 2) = neighbourCostToCome;
                alive(idx, 3) = neighbourCostToCome + heuristic(neighbour);
                alive(idx, 4) = currentNode(1);
            end
        else
            % if node is not in alive, add to alive
            alive = [alive; neighbour, neighbourCostToCome, neighbourCostToCome
+ heuristic(neighbour), currentNode(1)];
        end
    end
    iterationCounter = iterationCounter + 1;
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
else
```

```
    disp('No path found');
end
str = sprintf('Q3 - %d X %d Maze solved in %f seconds', row, col, dt);
title(str);

print -dpng assignment1_q3.png
```