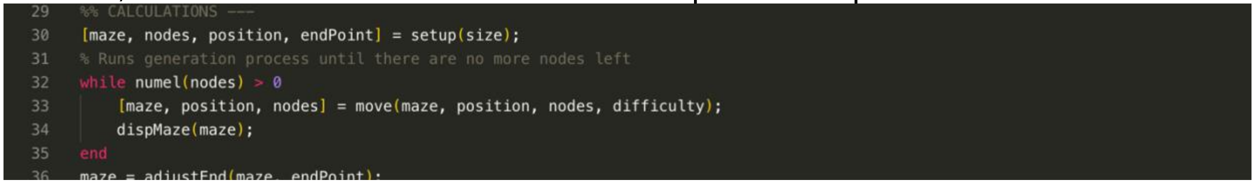
This part is based on the maze generator demo (MazeGeneration-master). The maze generator is a project written by some student using Matlab. He has adopted a tree search approach to randomly generate a maze with user-defined size and difficulty. ***(15 marks)***

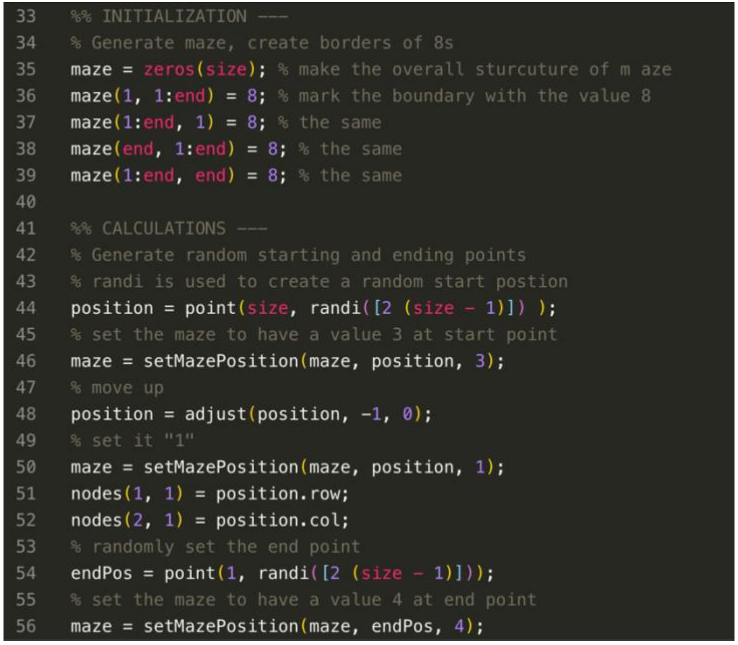
(a) Read the MazeGeneration-master code, identify which line(s) of code is used to implement the tree search approach, explain the logic and the data structure used by the student to implement the tree search. (3 marks)

1.`main.m`: Lines 32 – 35 The program explores the path within the while loop. For each iteration, the function move decides a node to expand and updates the new maze.



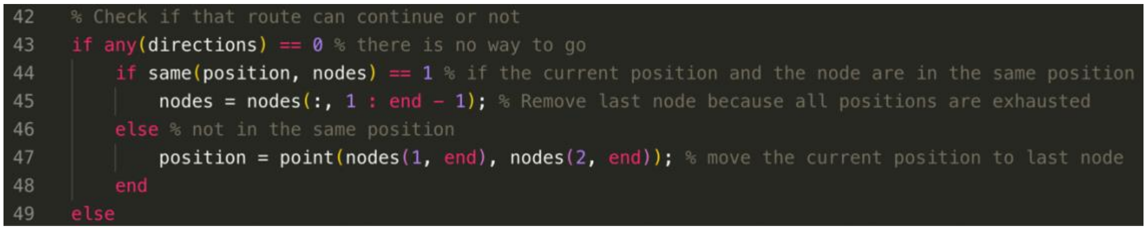
```
29  %% CALCULATIONS ---
30  [maze, nodes, position, endPoint] = setup(size);
31  % Runs generation process until there are no more nodes left
32  while numel(nodes) > 0
33      [maze, position, nodes] = move(maze, position, nodes, difficulty);
34      dispMaze(maze);
35  end
36  maze = adjustEnd(maze, endPoint);
```

2.`setup.m`: Lines 33 – 56 Set up the maze (2D-matrix with values) to store the boundary, start point (randomly set) and end point (randomly set). It is an overall structure of the maze for later operations. Also, this is the initialization of the maze.



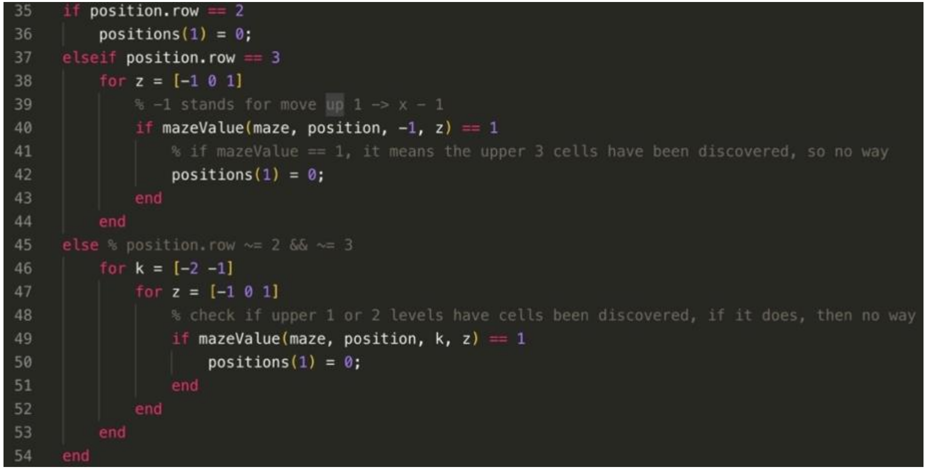
```
33  %% INITIALIZATION ---
34  % Generate maze, create borders of 8s
35  maze = zeros(size); % make the overall sturcuture of m aze
36  maze(1, 1:end) = 8; % mark the boundary with the value 8
37  maze(1:end, 1) = 8; % the same
38  maze(end, 1:end) = 8; % the same
39  maze(1:end, end) = 8; % the same
40
41  %% CALCULATIONS ---
42  % Generate random starting and ending points
43  % randi is used to create a random start postion
44  position = point(size, randi([2 (size - 1)]) );
45  % set the maze to have a value 3 at start point
46  maze = setMazePosition(maze, position, 3);
47  % move up
48  position = adjust(position, -1, 0);
49  % set it "1"
50  maze = setMazePosition(maze, position, 1);
51  nodes(1, 1) = position.row;
52  nodes(2, 1) = position.col;
53  % randomly set the end point
54  endPos = point(1, randi([2 (size - 1)]));
55  % set the maze to have a value 4 at end point
56  maze = setMazePosition(maze, endPos, 4);
```

3.`move.m`: Lines 42 – 49 Handle the array(matrix). Whenever there is no way to go for the current position, it checks ‘nodes’ and decide to go back or to remove the last inflection node. It implements the nodes as stack data-structure. Because of the implementation of stack, only the last element which is (nodes (1, end), nodes (2, end)) each step will be handled. By doing so, early data is protected, and the search tree always traces back to the nearest inflection node.



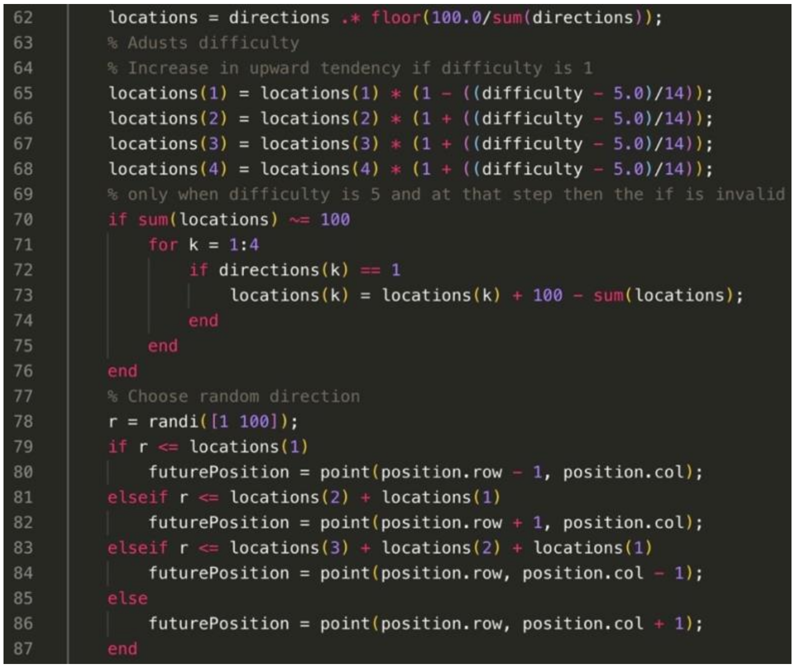
```
42  % Check if that route can continue or not
43  if any(directions) == 0 % there is no way to go
44      if same(position, nodes) == 1 % if the current position and the node are in the same position
45          nodes = nodes(:, 1 : end - 1); % Remove last node because all positions are exhausted
46      else % not in the same position
47          position = point(nodes(1, end), nodes(2, end)); % move the current position to last node
48      end
49  else
```

4.`validateMove.m`: Lines 35 – 54 Check if one direction is valid. This is the step for expansion of search tree. The basic idea is to check if the upper 3 cells of the current position is a path, which is the value "1" in maze. If not, then this direction is valid to expand, otherwise, invalid.



```
35  if position.row == 2
36      positions(1) = 0;
37  elseif position.row == 3
38      for z = [-1 0 1]
39          % -1 stands for move up 1 -> x - 1
40          if mazeValue(maze, position, -1, z) == 1
41              % if mazeValue == 1, it means the upper 3 cells have been discovered, so no way
42              positions(1) = 0;
43          end
44      end
45  else % position.row ~= 2 && ~= 3
46      for k = [-2 -1]
47          for z = [-1 0 1]
48              % check if upper 1 or 2 levels have cells been discovered, if it does, then no way
49              if mazeValue(maze, position, k, z) == 1
50                  positions(1) = 0;
51              end
52          end
53      end
54  end
```

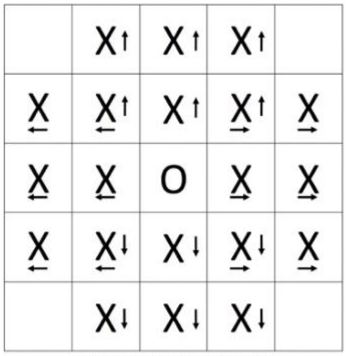
5.`move.m`: Lines 62 – 87 Based on the input difficulty, randomly choose a direction. The higher the difficulty is, the lower the tendency to go upwards. After deciding possibilities of four directions, a random direction will be chosen by generating a random number between 1 to 100.



```
62      locations = directions .* floor(100.0/sum(directions));
63      % Adusts difficulty
64      % Increase in upward tendency if difficulty is 1
65      locations(1) = locations(1) * (1 - ((difficulty - 5.0)/14));
66      locations(2) = locations(2) * (1 + ((difficulty - 5.0)/14));
67      locations(3) = locations(3) * (1 + ((difficulty - 5.0)/14));
68      locations(4) = locations(4) * (1 + ((difficulty - 5.0)/14));
69      % only when difficulty is 5 and at that step then the if is invalid
70      if sum(locations) ~= 100
71          for k = 1:4
72              if directions(k) == 1
73                  locations(k) = locations(k) + 100 - sum(locations);
74              end
75          end
76      end
77      % Choose random direction
78      r = randi([1 100]);
79      if r <= locations(1)
80          futurePosition = point(position.row - 1, position.col);
81      elseif r <= locations(2) + locations(1)
82          futurePosition = point(position.row + 1, position.col);
83      elseif r <= locations(3) + locations(2) + locations(1)
84          futurePosition = point(position.row, position.col - 1);
85      else
86          futurePosition = point(position.row, position.col + 1);
87      end
```

The implementation of the tree search is comprised of two parts which are data storage (related to data structure) and decision-making or the mode of searching.

The **logic** of tree search is to set up a 2D matrix at the beginning by filling in boundary (8), start position (3) at the bottom, end position (4) at the top. After initialization, a node travels from the start position to next position, set a value “1” of that position which stands for a path, so that other nodes are unable to expand to this node in further movement due to the function validateMove (), which checks if one direction is valid or visited (as showing on the right). Each time, the path chooses a random direction but not totally irregular which is based on difficulty referred to move.m (lines 62 – 87). The difficulty input by user influences the tendency of a path to travel upwards. The higher the difficulty is, the lower tendency for the path to travel upwards. Once, there is no way to travel for a path, the current position will go back to the last inflection point which is recorded by the array(matrix) ‘**nodes’.** The array(matrix) ‘**nodes**’ is used to record node which has more than one direction to go so that it can expand the paths when the current position has no way to go. After moving the current position to the last node in ‘**nodes**’, four directions will be checked if there is valid direction to go for the node. After checking an inflection point in ‘**nodes**’, if the current position is still the coordinate of that node, then delete this node in ‘**nodes**’ which means this node has been checked and no way to travel and move the current position to last which is the last element in ‘**nodes**’ and check again.



validateMove (Ryan Schwartz)

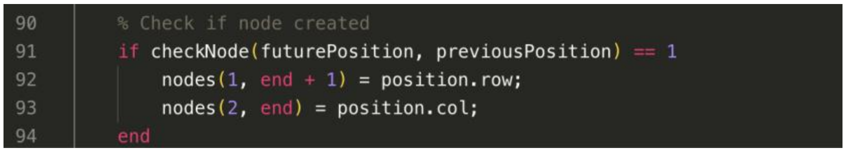
The **data structure** for the maze is 2D-array(matrix) and for ‘**nodes**’ is **stack**, **implemented by array(matrix)**. The 2D-array stores values including 0, 1, 3, 4, 8 which stands for different meaning within the maze. Element can be accessing by searching the index which is also the coordinates of that position in the maze. All operations are directly implemented on the maze including targeting a position as the boundary or valid path and checking if a direction is valid to go. As for ‘**nodes**’, it contains the horizontal and vertical coordinates of an inflection point. It is operated only at the end of the array(matrix) and data is first-in and first-out which makes the current position always go back to the nearest inflection point.

(b) Identify the logic problem of this maze generator if there is any. (1 marks)

In the original “`move.m`” file, after the direction is decided randomly but base on the difficulty, **a previous position** of **current position** which is the current inflection node in ‘**nodes’** is found. However, **the previous position** is found in a default sequence. First is up, second down, next left, and last right. **This is where causes the logic problem**. It is because, **previousPosition** is decided in default sequence. It is not exact a previous position for **current node**. The program(maze-generator) is unable to truly find **previousPosition**.

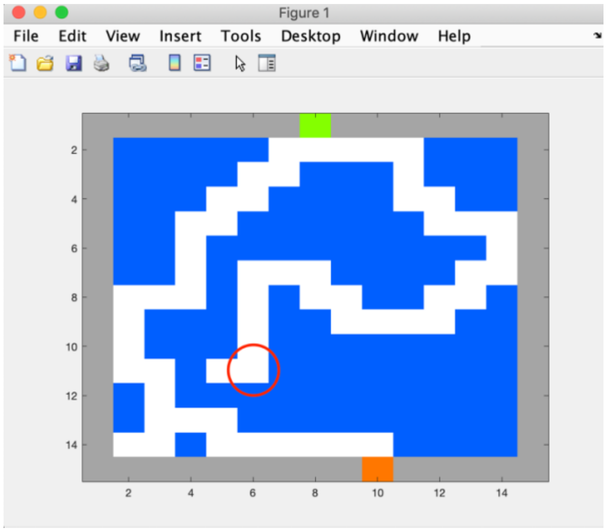
After that, the ‘if-statement’ in line 90~94 which contains a function ‘**`checkNode()`**’

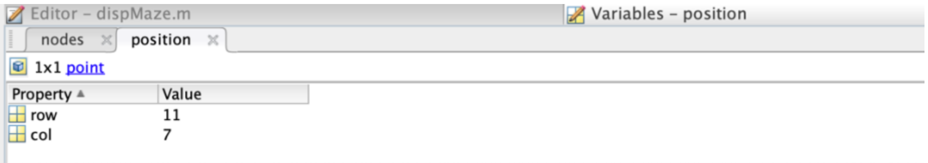
checks if the **furtherPosition** (the expanded position) has totally different 'row' and 'col' with **previousPosition.** In normal conditions, it is used to check when exploring the path for the first time not including the case for tracing back, which means, a node would be added to '**nodes'** more than once.



```
        % Check if node created
        if checkNode(futurePosition, previousPosition) == 1
            nodes(1, end + 1) = position.row;
            nodes(2, end) = position.col;
        end
```

For example, right now the inflection point in nodes is the red point below with coordinate (11, 6). Left is an exploring path which has no way to go, so it returns back to (11, 6).

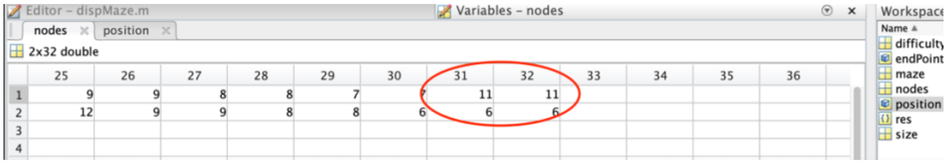




1x1 point

Property	Value
row	11
col	7

Due to the algorithm of finding previous position, (10, 6) is the **previousPosition. However,** the current position is (11, 7). Because they have different 'col' and 'row', the inflection point, (11, 6), is regarded as another inflection point and is added to 'nodes' once again.



2x32 double

	25	26	27	28	29	30	31	32	33	34	35	36
1	9	9	8	8	7	7	11	11				
2	12	9	9	8	8	6	6	6				
3												
4												

Because of that, when finishing exploring, this node (11, 6) will be examined twice which is not necessary.

To conclude, there are two main continuous logic problems, which are **deciding**

previousPosition in a sequence instead of finding the direction of the path and **adding nodes twice** (in certain condition), though in the example, **previousPosition** is found correctly. The later problem occasionally happens when tracing back to an inflection node and mistakenly target it as another inflection node due to ‘**`checkNode()`**’.

(c) Write a maze solver using A* algorithm. (5 marks)

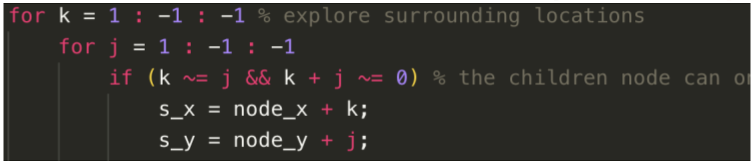
i) The solver needs to be called by command ‘**AStarMazeSolver(maze**)’ within the Matlab command window, with the assumption that the ‘maze’ has already been generated by the maze generator.

ii) In the report, show what changes you have made and explain why you make these changes. You can use a screenshot to demonstrate your code verification.

Changes on **AStarMazeSolver** comparing to **the original AStarSolver**:

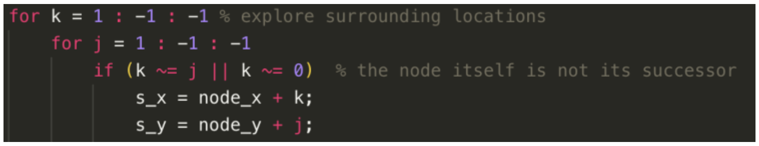
1. Removing the function “`problem()`”. Because in AStarMazeSolver, we have an argument which is the maze matrix with values. Besides the size of maze and difficulty input by user, the borders, walls, start point and target have already stored in maze. Additionally, coordinates system is not necessary to draw in output since the AStarMazeSolver is used to find the path.

2. Moving diagonally is prohibited in `AStar-MazeSolver/expand.m`. It is because in MazeGeneration-master, path can only change direction at 90 degrees. Therefore, the condition for ‘if-statement’ becomes “`if (k ~= j && k + j ~= 0)`”.



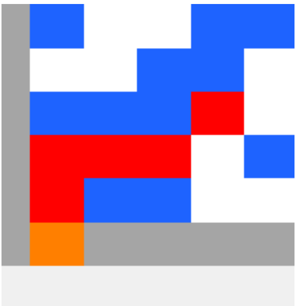
```
for k = 1 : -1 : -1 % explore surrounding locations
    for j = 1 : -1 : -1
        if (k ~= j && k + j ~= 0) % the children node can o
            s_x = node_x + k;
            s_y = node_y + j;
```

However, in original `AStar,` it is allowed because in coordinate system without restriction, moving diagonally is achieved by ‘if-statement’: “`if (k ~= j || k ~= 0)`”.

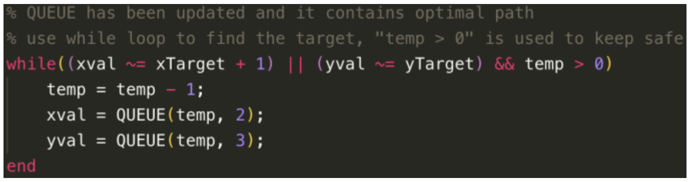


```
for k = 1 : -1 : -1 % explore surrounding locations
    for j = 1 : -1 : -1
        if (k ~= j || k ~= 0)  % the node itself is not its successor
            s_x = node_x + k;
            s_y = node_y + j;
```

Otherwise, it may explore the path like below.

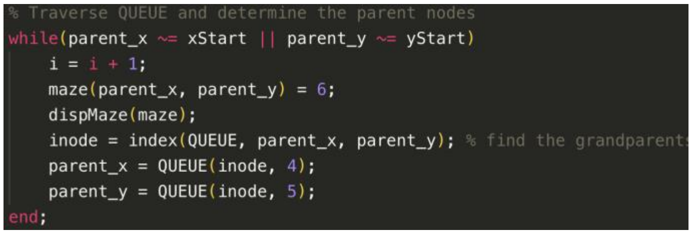


3. Based on the demo during the lecture, when marking the final optimal path, the path is marked from the target to the start point which differs from AStarSolver. It is because, when implementing ‘**QUEUE’**, the coordinates of a node and coordinates of its parent node are stored in it. Therefore, when tracing the path, it will start from the end to start, from child to parent, which makes the final optimal path inverse. To implement that, the array “Optimal_path” in the original AStarSolver could be omitted and set the value of each node, trace its parent node and do it iteratively until the start point is found.



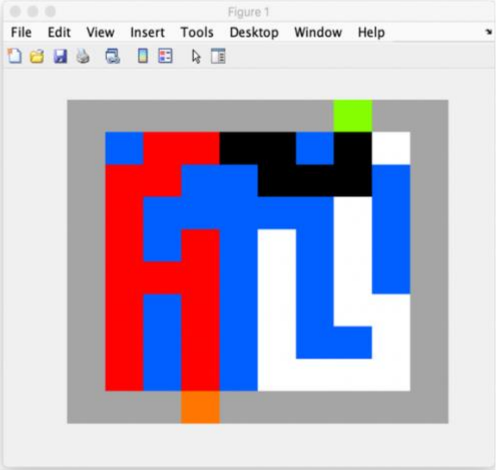
```
% QUEUE has been updated and it contains optimal path
% use while loop to find the target, "temp > 0" is used to keep safe
while((xval ~= xTarget + 1) || (yval ~= yTarget) && temp > 0)
    temp = temp - 1;
    xval = QUEUE(temp, 2);
    yval = QUEUE(temp, 3);
end
```

`AStar-MazeSolver/result.m`: finding the target(end)



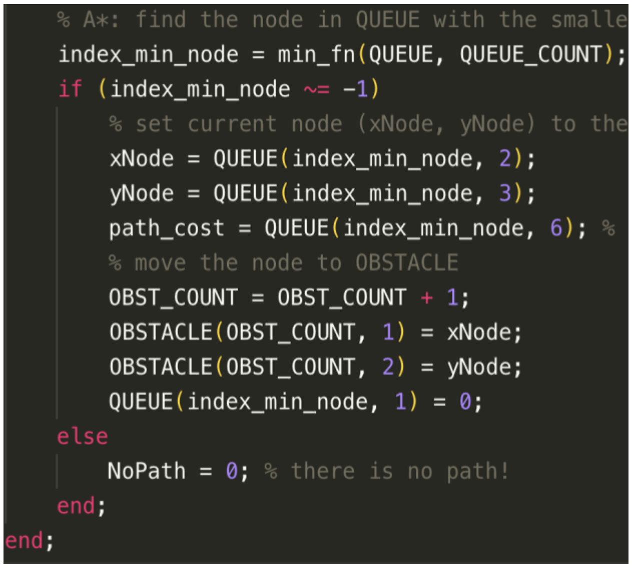
```
% Traverse QUEUE and determine the parent nodes
while(parent_x ~= xStart || parent_y ~= yStart)
    i = i + 1;
    maze(parent_x, parent_y) = 6;
    dispMaze(maze);
    inode = index(QUEUE, parent_x, parent_y); % find the grandparents
    parent_x = QUEUE(inode, 4);
    parent_y = QUEUE(inode, 5);
end;
```

`AStar-MazeSolve/result.m`: finding the start point

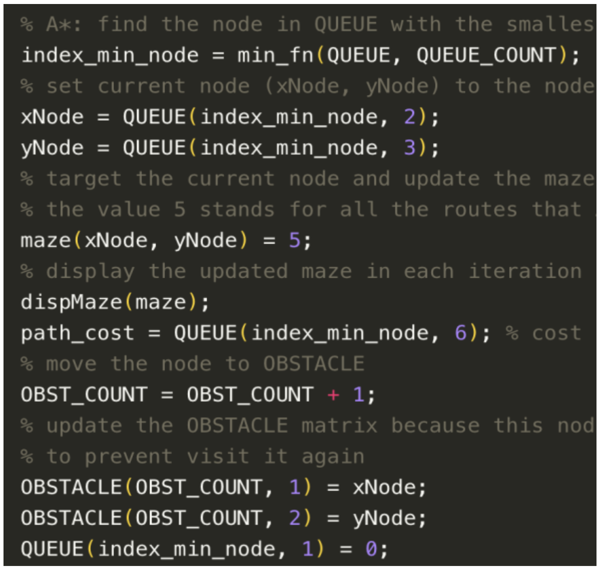


optimal path (in black) traces back from the end point to the start point

4. In MazeGeneration-master, a valid path from the end to the start always exists which is achieved in “`adjustEnd.m`”. Therefore, the case for no valid path and prompt in “`result.m`” could be omitted. On this basis, the program(maze-solver) have been tested for a large number of times and it works successfully.



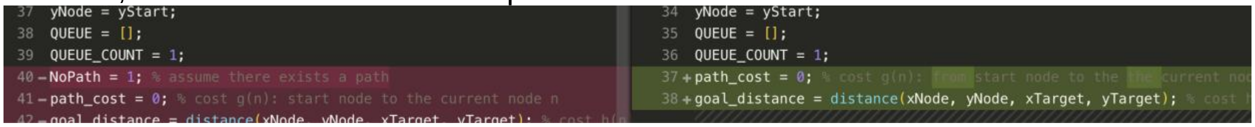
```
    % A*: find the node in QUEUE with the smalle
    index_min_node = min_fn(QUEUE, QUEUE_COUNT);
    if (index_min_node ~= -1)
        % set current node (xNode, yNode) to the
        xNode = QUEUE(index_min_node, 2);
        yNode = QUEUE(index_min_node, 3);
        path_cost = QUEUE(index_min_node, 6); %
        % move the node to OBSTACLE
        OBST_COUNT = OBST_COUNT + 1;
        OBSTACLE(OBST_COUNT, 1) = xNode;
        OBSTACLE(OBST_COUNT, 2) = yNode;
        QUEUE(index_min_node, 1) = 0;
    else
        NoPath = 0; % there is no path!
    end;
end;
```



```
% A*: find the node in QUEUE with the smalles
index_min_node = min_fn(QUEUE, QUEUE_COUNT);
% set current node (xNode, yNode) to the node
xNode = QUEUE(index_min_node, 2);
yNode = QUEUE(index_min_node, 3);
% target the current node and update the maze
% the value 5 stands for all the routes that
maze(xNode, yNode) = 5;
% display the updated maze in each iteration
dispMaze(maze);
path_cost = QUEUE(index_min_node, 6); % cost
% move the node to OBSTACLE
OBST_COUNT = OBST_COUNT + 1;
% update the OBSTACLE matrix because this nod
% to prevent visit it again
OBSTACLE(OBST_COUNT, 1) = xNode;
OBSTACLE(OBST_COUNT, 2) = yNode;
QUEUE(index_min_node, 1) = 0;
```

The left one is original “A_Star.m” which contains a case for not existing path
The right one is “AStarMazeSolver.m" which omits this case (the ‘if-statement’)

Besides, indicator variables like ‘Nopath’ have been removed as well.



```
  yNode = yStart;
  QUEUE = [];
  QUEUE_COUNT = 1;
- NoPath = 1; % assume there exists a path
- path_cost = 0; % cost g(n): start node to the current node n
- goal_distance = distance(xNode, yNode, xTarget, yTarget); % cost h(n
```

```
  yNode = yStart;
  QUEUE = [];
  QUEUE_COUNT = 1;
+ path_cost = 0; % cost g(n): from start node to the the current noc
+ goal_distance = distance(xNode, yNode, xTarget, yTarget); % cost h
```

iii) The maze solver should be able to solve any maze generated by the maze generator

iv) Your code need display all the routes that A* has processed with RED color.

v) Your maze solver should be about to display the final solution with BLACK color.

vi)

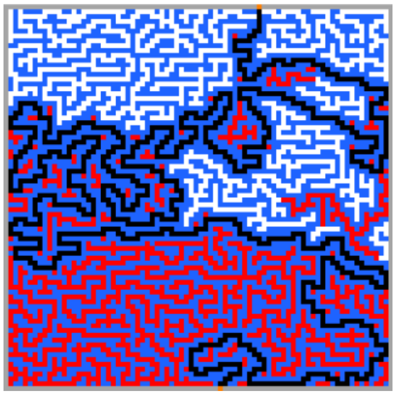
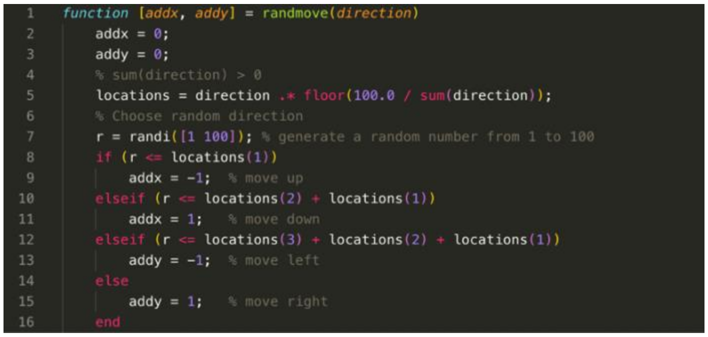


Figure 1. Sample output

(d) Based on the previous AStarMazeSolver, implement a maze solver using the DFS algorithm. Similar to question (c), the solver needs by called and ran by command '**DFSMazeSolver(maze)**'. (3 marks)

Changes on **DFSMazeSolver** comparing to **AStarSolver**:

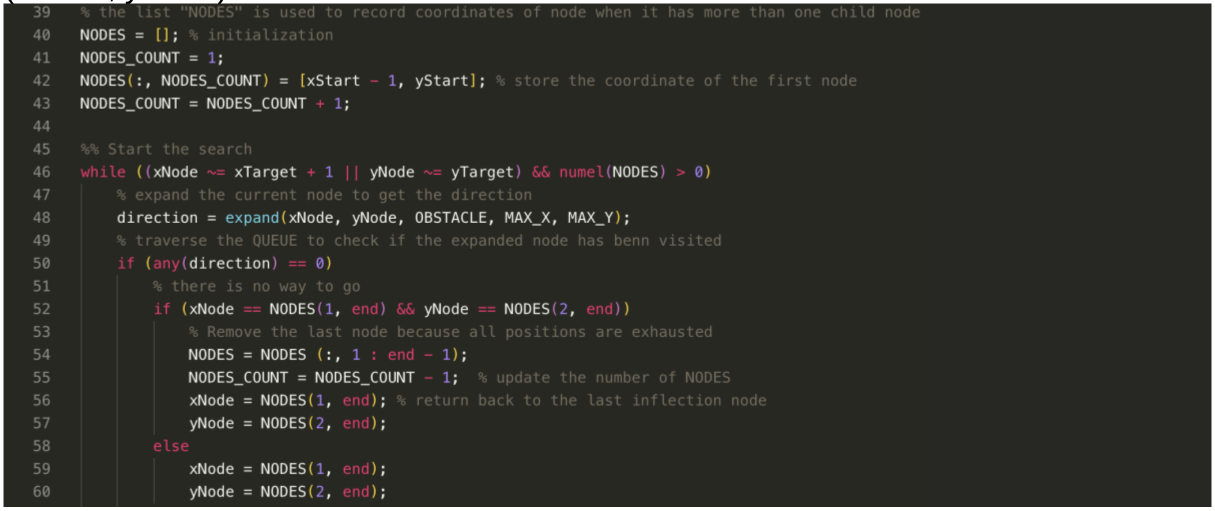
1. DFSMazeSolver takes no account of cost of the path (gn) discovered so far and heuristic search (hn). It is totally **blind search** without knowledge such as straight-line distance. Whenever the current position has more than one direction to expand, it will randomly choose one which is implemented in `DFS-MazeSolver/randmove.m`. **Randomly choosing a direction** imitates the maze-generator in `MazeGeneration-master/move.m`.



```
function [addx, addy] = randmove(direction)
    addx = 0;
    addy = 0;
    % sum(direction) > 0
    locations = direction .* floor(100.0 / sum(direction));
    % Choose random direction
    r = randi([1 100]); % generate a random number from 1 to 100
    if (r <= locations(1))
        addx = -1;  % move up
    elseif (r <= locations(2) + locations(1))
        addx = 1;   % move down
    elseif (r <= locations(3) + locations(2) + locations(1))
        addy = -1;  % move left
    else
        addy = 1;   % move right
    end
```

`DFS-MazeSolver/randomove.m`

2. Because searching for the optimal path is not the task for DFS, it only searches for the end point. Thus, it is implemented like maze-generator. Each time, the current position meets an inflection node and the coordinate of this inflection node is stored in an array '**NODES** just like maze-generator does. When there is no way to go for current node which is (xNode, yNode), it returns back to the nearest node in '**NODES'.** If there is no way to go and the current node (xNode, yNode) is the last element in '**NODES'**, then the last element will be removed from the array the current node (xNode, yNode) returns to the last inflection node.



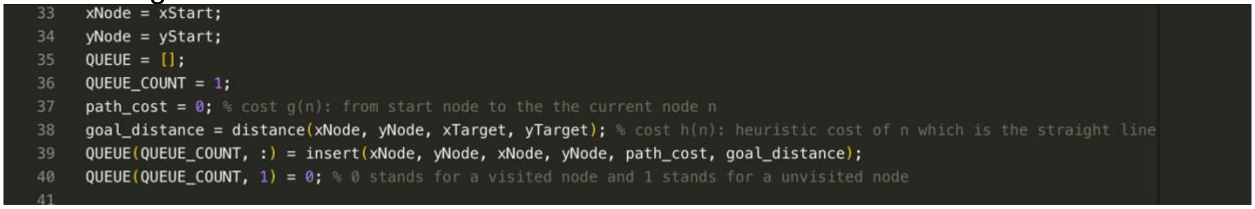
```
% the list "NODES" is used to record coordinates of node when it has more than one child node
NODES = []; % initialization
NODES_COUNT = 1;
NODES(:, NODES_COUNT) = [xStart - 1, yStart]; % store the coordinate of the first node
NODES_COUNT = NODES_COUNT + 1;

%% Start the search
while ((xNode ~= xTarget + 1 || yNode ~= yTarget) && numel(NODES) > 0)
    % expand the current node to get the direction
    direction = expand(xNode, yNode, OBSTACLE, MAX_X, MAX_Y);
    % traverse the QUEUE to check if the expanded node has benn visited
    if (any(direction) == 0)
        % there is no way to go
        if (xNode == NODES(1, end) && yNode == NODES(2, end))
            % Remove the last node because all positions are exhausted
            NODES = NODES (:, 1 : end - 1);
            NODES_COUNT = NODES_COUNT - 1;  % update the number of NODES
            xNode = NODES(1, end); % return back to the last inflection node
            yNode = NODES(2, end);
        else
            xNode = NODES(1, end);
            yNode = NODES(2, end);
```

`DFS-MazeSolver/DFSMazeSolver.m`

(e) Based on the previous AStarMazeSolver, implement a maze solver using the Greedy search algorithm. Similar to question (c), the solver needs by called by command '**GreedyMazeSolver(maze)**'. (1 marks)

Changes on **GreedyMazeSolver** comparing to **AStarSolver**:

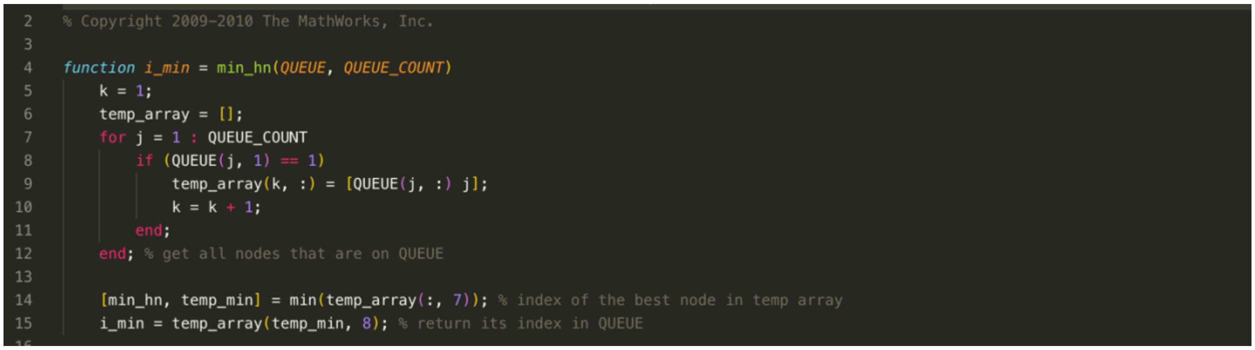
1. GreedyMazeSolver takes no account of the evaluation of the actual path cost so far. In another way, GreedyMazeSolver only considers nodes with minimum goal distance or straight-line distance.



```
xNode = xStart;
yNode = yStart;
QUEUE = [];
QUEUE_COUNT = 1;
path_cost = 0; % cost g(n): from start node to the the current node n
goal_distance = distance(xNode, yNode, xTarget, yTarget); % cost h(n): heuristic cost of n which is the straight line
QUEUE(QUEUE_COUNT, :) = insert(xNode, yNode, xNode, yNode, path_cost, goal_distance);
QUEUE(QUEUE_COUNT, 1) = 0; % 0 stands for a visited node and 1 stands for a unvisited node
```

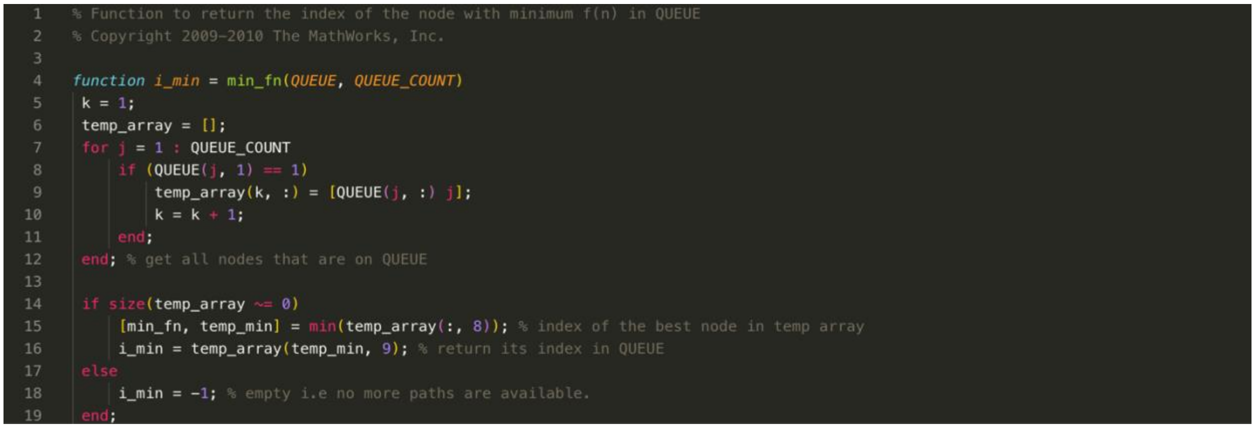
Greedy-MazeSolver/GreedyMazeSolver.m

The data-structure for '**QUEUE'** is [0/1, xValue, yValue, xParent, yParent, g(n), h(n)]. In GreedyMazeSolver, min_hn.m searches for the index of node which has the smallest hn (straight-line distance). While in AStar, min_fn.m searches for the index of node which has the smallest fn (cost of path plus straight-line distance).



```
% Copyright 2009-2010 The MathWorks, Inc.

function i_min = min_hn(QUEUE, QUEUE_COUNT)
    k = 1;
    temp_array = [];
    for j = 1 : QUEUE_COUNT
        if (QUEUE(j, 1) == 1)
            temp_array(k, :) = [QUEUE(j, :) j];
            k = k + 1;
        end;
    end; % get all nodes that are on QUEUE

    [min_hn, temp_min] = min(temp_array(:, 7)); % index of the best node in temp array
    i_min = temp_array(temp_min, 8); % return its index in QUEUE
```

GreedyMazeSolver/min_hn.m



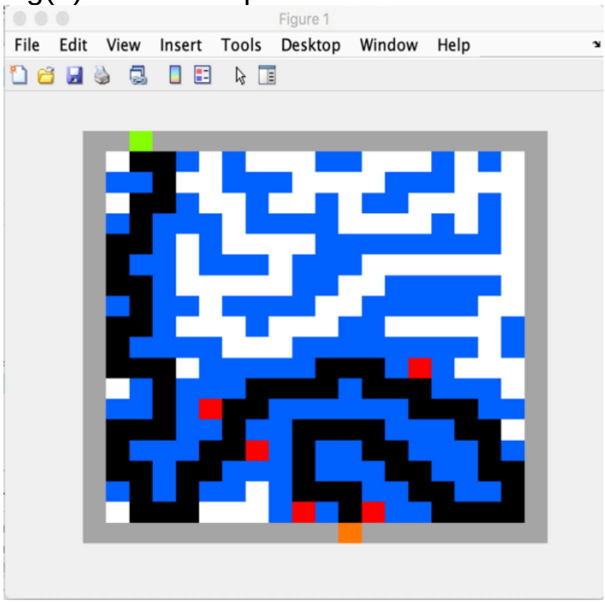
```
% Function to return the index of the node with minimum f(n) in QUEUE
% Copyright 2009-2010 The MathWorks, Inc.

function i_min = min_fn(QUEUE, QUEUE_COUNT)
 k = 1;
 temp_array = [];
 for j = 1 : QUEUE_COUNT
     if (QUEUE(j, 1) == 1)
         temp_array(k, :) = [QUEUE(j, :) j];
         k = k + 1;
     end;
 end; % get all nodes that are on QUEUE

 if size(temp_array ~= 0)
     [min_fn, temp_min] = min(temp_array(:, 8)); % index of the best node in temp array
     i_min = temp_array(temp_min, 9); % return its index in QUEUE
 else
     i_min = -1; % empty i.e no more paths are available.
 end;
```

AStar/min_fn.m

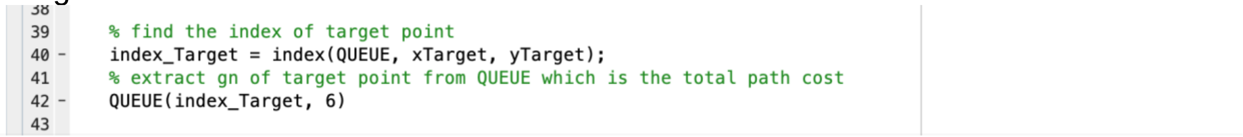
(f) You are required to use the Matlab basics from the first lab session to show the evaluation results of the three searching methods you've implemented in (c), (d) and (e) (hint: bar/plot) with respect to the '**total path cost**', '**number of nodes discovered**' and '**number of nodes expanded**'. Explain how you can extract the related information from data stored in variable '**QUEUE'.** (2 marks).

(the related codes are not shown in submitted files)

total path cost:
Regarding to three searching methods, total path cost is the cost of the path from the start point to the end point or from the end point to the start point and the path is in black. Also, the total path cost is the g(n) of the end point.



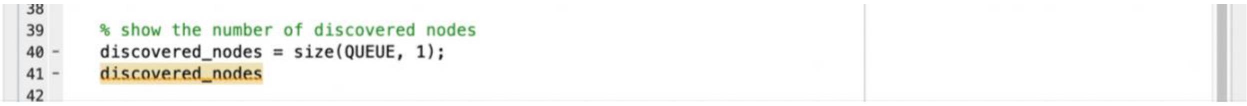
To extract the **total path cost** from data stored in variable **'QUEUE'**, we firstly find the index of the end point / target, and then extract the gn from '**QUEUE**' with the index by using the command:



```
    % find the index of target point
    index_Target = index(QUEUE, xTarget, yTarget);
    % extract gn of target point from QUEUE which is the total path cost
    QUEUE(index_Target, 6)
```

result.m

number of nodes discovered:
Each time whenever the current position meets potential nodes that could expand, it will add them into '**QUEUE'**. Thus, the **number of nodes discovered** is all the nodes stored in '**QUEUE'**.

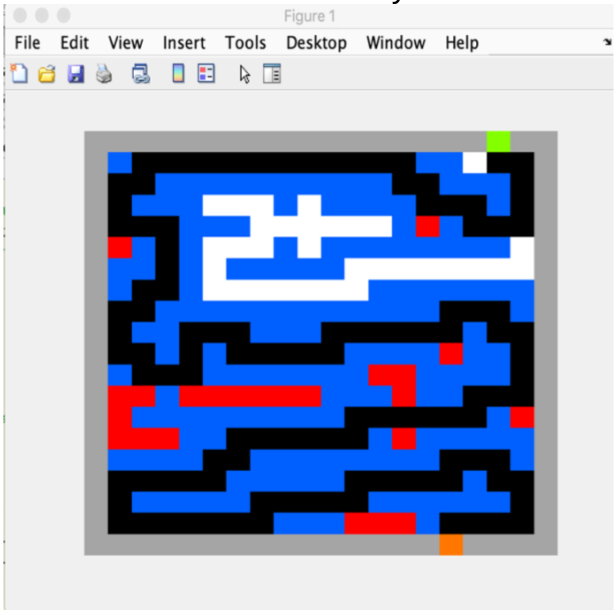


```
    % show the number of discovered nodes
    discovered_nodes = size(QUEUE, 1);
    discovered_nodes
```

As for DFSMazeSolver, the number of discovered nodes is the number of expanded nodes **plus one** due to the different algorithm comparing to AStarMazeSolver and

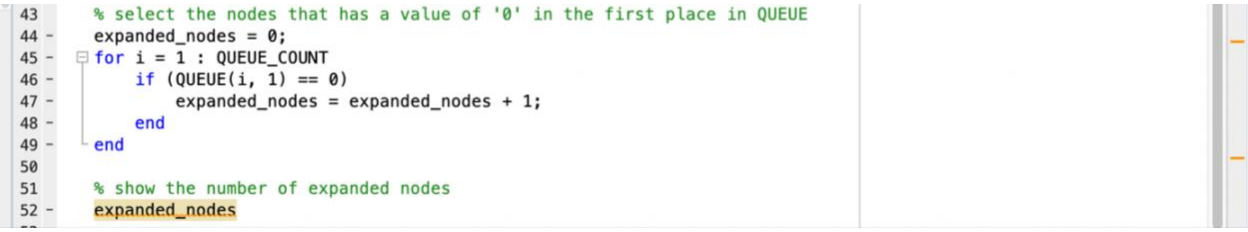
GreedyMazeSolver. DFS has no need to find the minimum path-cost or straight-line distance to decide the next movement, instead, DFS randomly chooses a direction based on my implementation. Each time, DFS only discover one node for next expansion.
The reason why the **number discovered nodes** has **one more node** (not only in DFSMazeSolver) than the **number of expanded nodes** is that the algorithm would discover the target node and add it into **'QUEUE'** as well. And when current node is on target, it has the smallest cost, or straight-line distance or it gets to the goal (in DFS) so that the loop would end.

number of nodes expanded:
It is obvious that expanded nodes are those in red and black in the maze. To extract these nodes, we can define a counter to count how many nodes are in **red**.



In **'QUEUE'**, the nodes that expanded or visited are colored in red and the first element of that tuple is 0. Thus, finding the number of expanded nodes is to count how many nodes in **'QUEUE'** has a value of '0' in first element, which is:

```
QUEUE (i, 1);
```

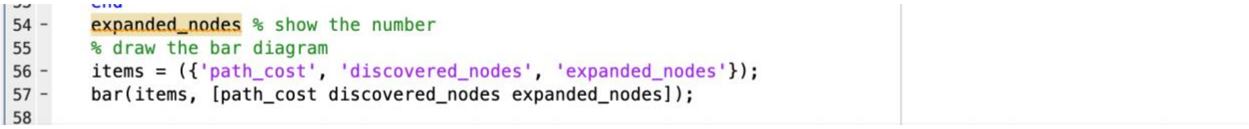


```
% select the nodes that has a value of '0' in the first place in QUEUE
expanded_nodes = 0;
for i = 1 : QUEUE_COUNT
    if (QUEUE(i, 1) == 0)
        expanded_nodes = expanded_nodes + 1;
    end
end

% show the number of expanded nodes
expanded_nodes
```

```
result.m
```

Total path cost, the number of nodes expanded, and the number of nodes discovered could be extracted from result.m by using the above command.

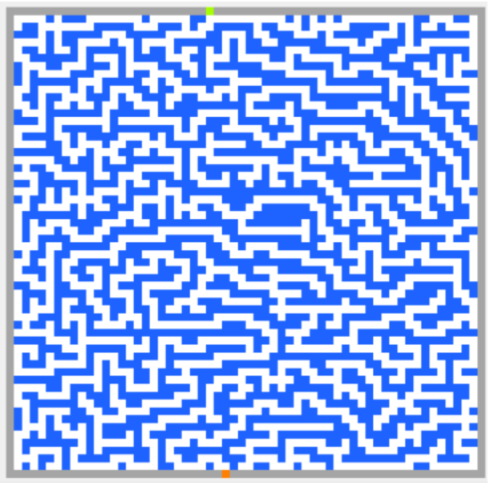
To **evaluate** results of three searching methods, we could apply the function “bar” to draw a bar diagram of three results within MATLAB.



```
    end
    expanded_nodes % show the number
    % draw the bar diagram
    items = ({'path_cost', 'discovered_nodes', 'expanded_nodes'});
    bar(items, [path_cost discovered_nodes expanded_nodes]);
```

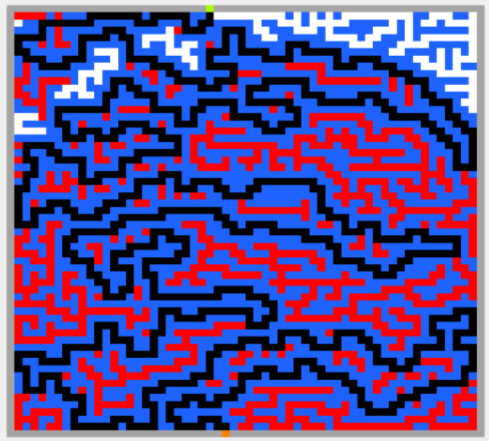
result.m

To compare results of three methods, a test maze is generated with size 60 and difficulty 8 like below.

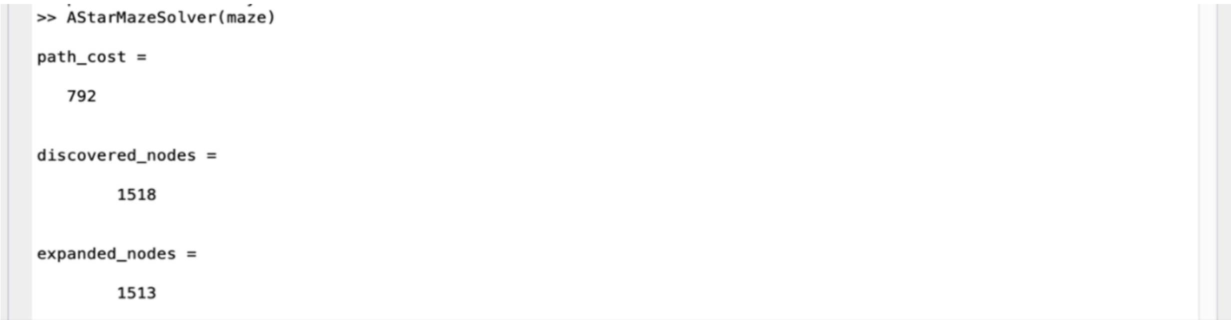


the maze for testing

To evaluate the three results of **AStarMazeSolver**, we apply it to this maze and get results.

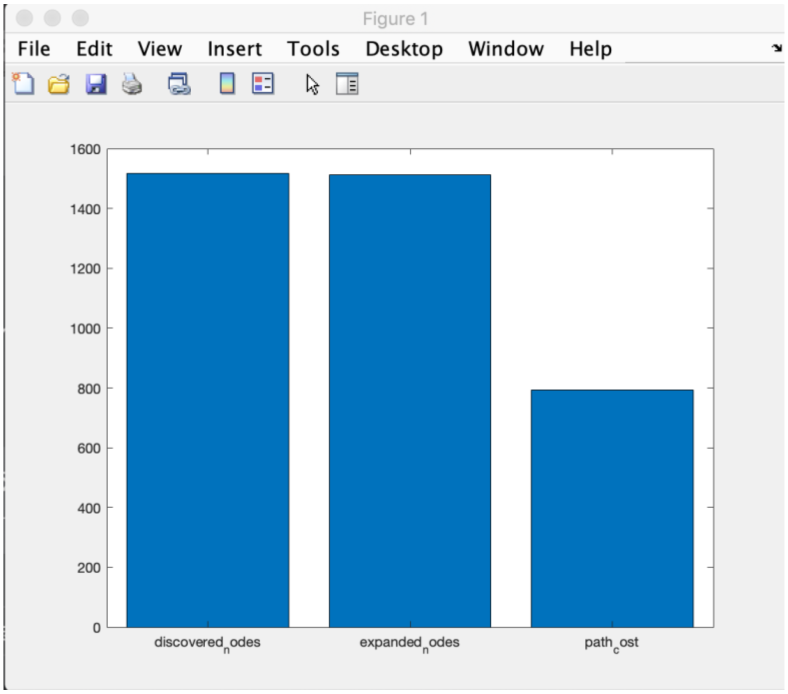


the maze for AStarMazeSolver



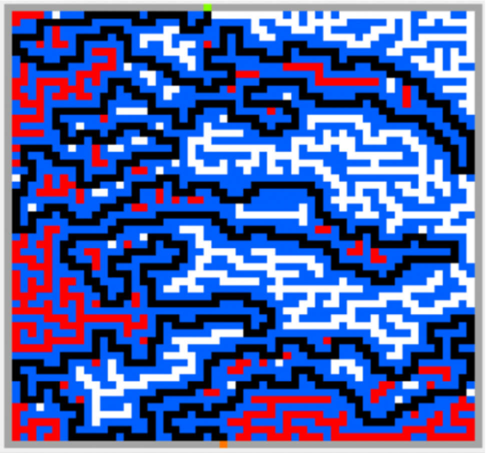
```
>> AStarMazeSolver(maze)

path_cost =

   792


discovered_nodes =

        1518


expanded_nodes =

        1513
```

the results for AStarMazeSolver

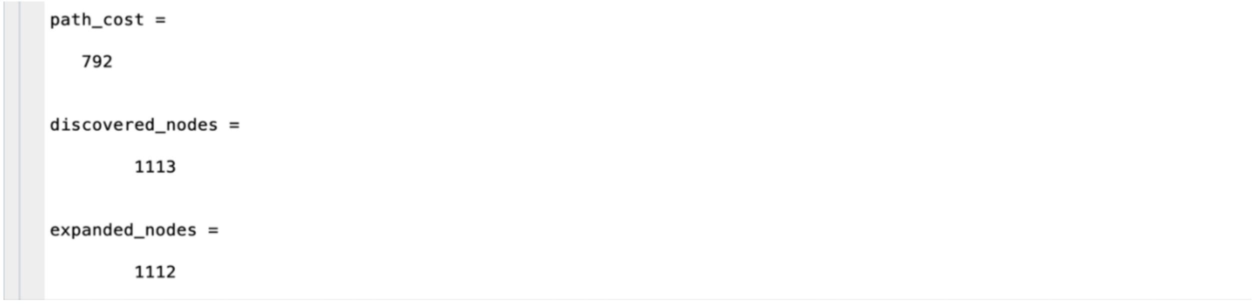


the bar diagram for AStarMazeSolver

To evaluate the three results of **DFSMazeSolver**, we apply it to this maze and get results.

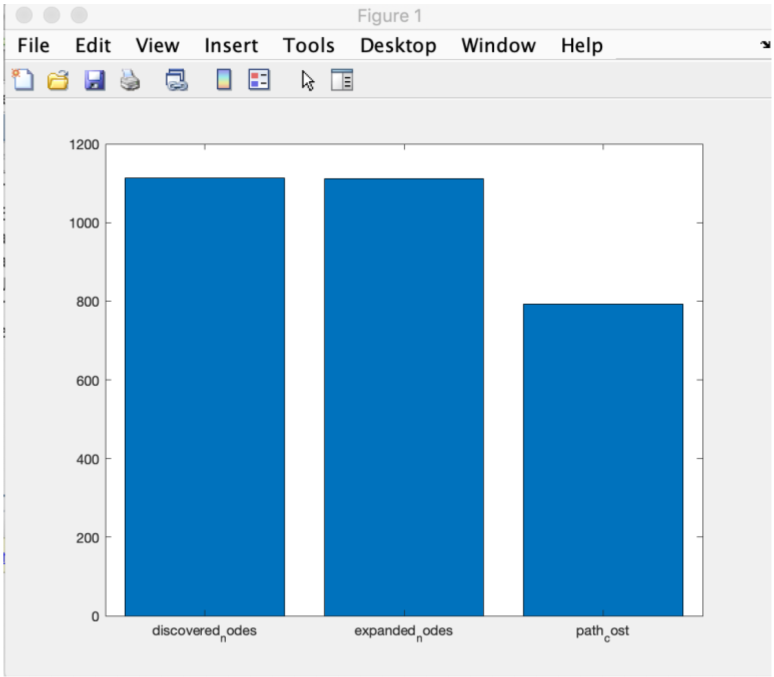


the maze for DFSMazeSolver



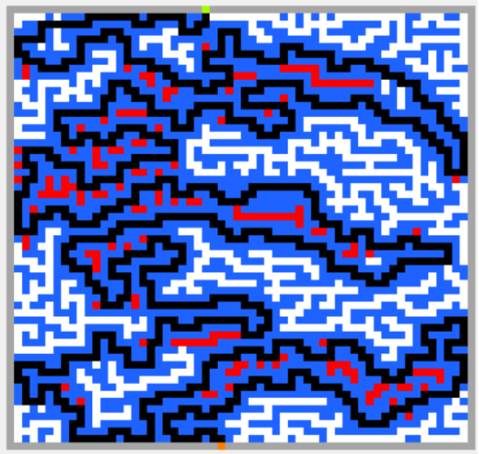
```
path_cost =

   792


discovered_nodes =

        1113


expanded_nodes =

        1112
```

the results for DFSMazeSolver

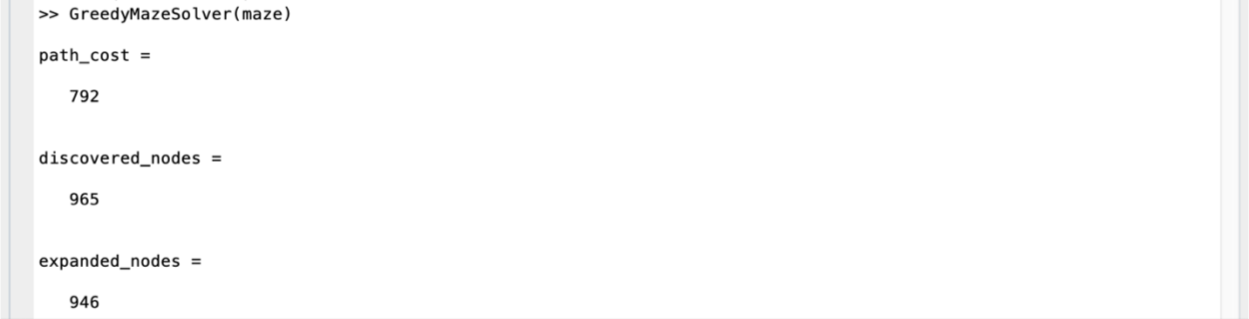


the bar diagram for DFSMazeSolver

To evaluate the three results of **GreedyMazeSolver**, we apply it to this maze and get results.

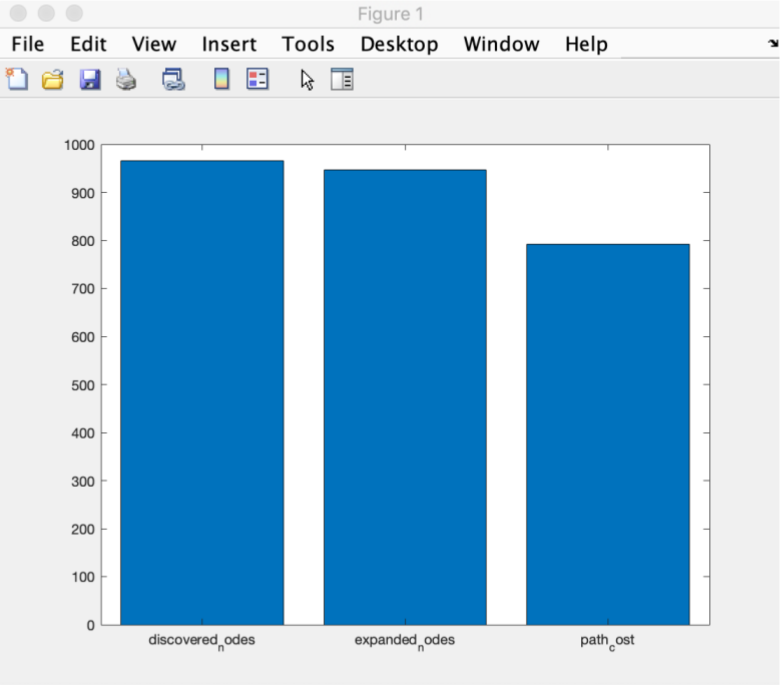


the maze for GreedyMazeSolver

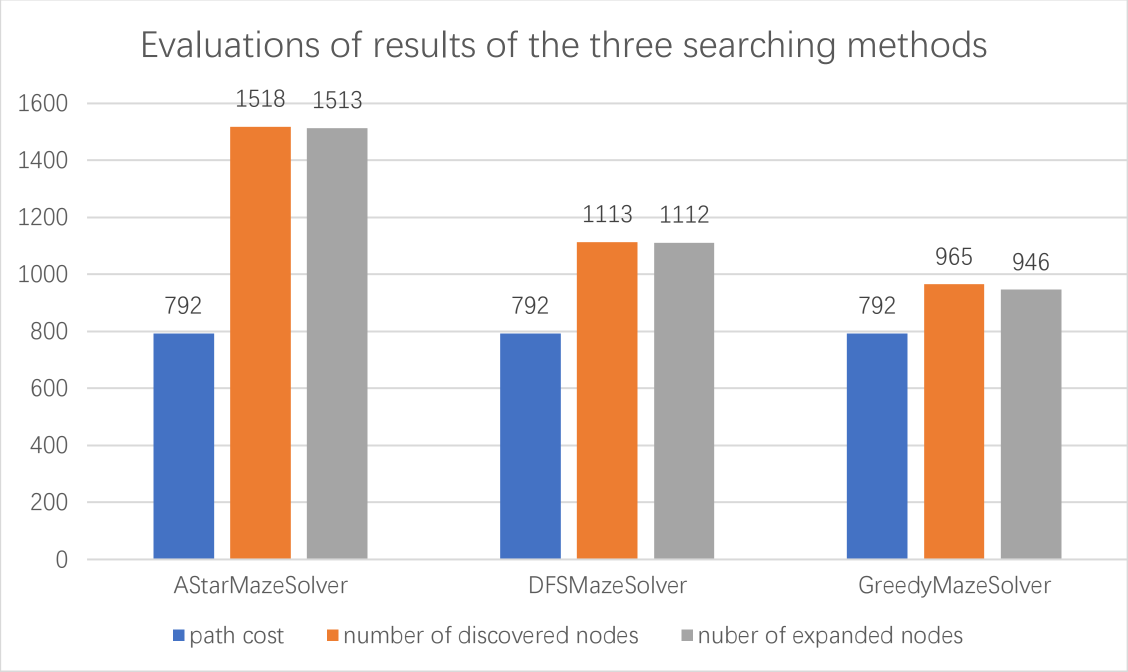


```
>> GreedyMazeSolver(maze)

path_cost =

   792


discovered_nodes =

   965


expanded_nodes =

   946
```

the results for GreedyMazeSolver



the bar diagram for GreedyMazeSolver



Explanation and analysis:
Normally, AStar is the best alogorithm to find the optimal path, Greedy algorithm is the second and DFS is the worst. However, in this maze, there is only one valid path form the end point to the start point which is also the optimal path. Because of that, once the valid path is found, the optimal path is found, which makes DFS and Greedy algorithm more efficient without considering the cost of expanded path.

Nevertheless, results might be different due to the configuration of the maze. Also, for DFSMazeSolver, results differ from each time we run it due to its choosing directions randomly.