This part is based on the maze generator demo (DFSMazeGeneration-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 DFSMazeGeneration-master code, modify the code into a maze generator following BFS tree search strategy. [5 marks]

i) The BFS maze generator needs to be successfully called by command 'main'. (2 marks)

ii) In the report, use screenshot of code show what changes you have made. Explain why you make these changes. (3 marks)

(b) Write a maze solver using A* algorithm. [10 marks]

i) The solver needs to be successfully called by command '**AStarMazeSolver(maze**)' within the Matlab command window, with the assumption that the 'maze' has already been generated by the maze generator. The maze solver should be able to solve any maze generated by the maze generator. (2 marks)

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. (4 marks)

iii) Your code need display all the routes that A* has processed with RED color. (2 marks)

iv) Your maze solver should be about to display the final solution with BLACK color. (2 marks)

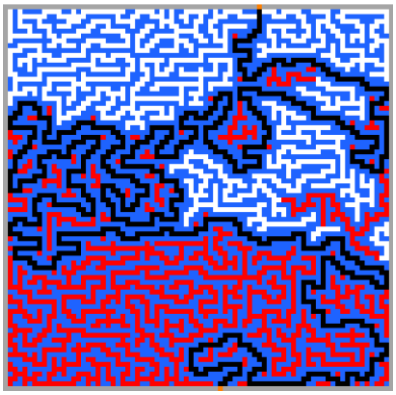

Figure 1. Sample output

FAI CW1 Report

In both files of BFS maze generator and A* maze solver, only necessary scripts and functions are contained.

(a) BFS maze generator

(ii) There are 3 changes:

1) Change 1
- Position: main.m, line 41
- Screenshot:

```
41 –     save my_BFSmaze.mat maze % change
```

- Explanation:

  Previous code "save my_DFSmaze.mat maze" saves the output as DFS maze. As the DFS maze generator has been modified into a BFS one, the output should be saved as BFS maze.

2) Change 2
- Position: move.m, line 46 and line 48
- Screenshot:

```
44 –     if same(position, nodes) == 1
45           % Remove first node because all positions are exhausted
46 –         nodes = nodes(:, 2 : end); % change
47 –     else
48 –         position = point(nodes(1, 1), nodes(2, 1)); % change
49 –     end
```

- Explanation:

  In DFS maze generator, the array/node list "nodes" that stored all the discovered nodes acts as a **last-in-first-out queue (act as a stack)**. However, in BFS maze generator, these changes are made to modify the stack into a **first-in-first-out queue**.

  The reason is that, the main difference of these 2 strategies lies in the order of node expansion. The queuing function for DFS is removing front node first and **adding its child nodes to the front of queue**, whilst that for BSF is removing front node but **adding its child nodes to the end of queue**.

  In previous code, **for DFS, the end of array "nodes" with larger index is regarded as the front of queue**. When the route cannot continue, if current position is the last node in "nodes", it should be removed first, or the current position should be moved to last node which should be further expanded. The child nodes of the first node in "nodes" should be added to the end of "nodes".

  In modified code shown above, **for BFS, the start of "nodes" with smaller index is regarded as the front of queue**. When the route cannot continue, if current position is the first node in "nodes", it should be removed first, or the current position should be moved to the first node which should be further expanded. Newly generated nodes should also be added to the end of "nodes".

3) Change 3

- Position: point.m, line 40, function "same"
- Screenshot:

```
40 -                if a.row == nodes(1, 1) && a.col == nodes(2, 1) % change
41 -                    t = 1;
```

- Explanation:

  Previous code aimed to check if the given node "a" is the same as last node in "nodes". It is now modified to check if the node "a" is the same as **first node** in "nodes".

  It is used in move.m to check if the route cannot continue and the position is the first node of "nodes". If so, first node should be removed. Both DFS and BFS add nodes to the end of "nodes", but DFS expands and removes nodes from the ends first, BFS expands and removes nodes from the start first.

(b) A* maze solver

(ii) There are 4 changes:

1) Change 1

- Position: AStarMazeSolver.m, line 20, line 25, line 30-33, line 34-37
- Screenshot:

```
19      % OBSTACLE: [X val, Y val]
20 -    [MAX_X, MAX_Y] = size(maze); % change
21 -    OBSTACLE = [];
22 -    k = 1;
23 -    for i = 1 : MAX_X
24 -        for j = 1 : MAX_Y
25 -            if(maze(i, j) == 0) % change
26 -                OBSTACLE(k, 1) = i;
27 -                OBSTACLE(k, 2) = j;
28 -                k = k + 1;
29 -            end
30 -            if(maze(i, j) == 3) % change
31 -                xStart = i - 1;
32 -                yStart = j;
33 -            end
34 -            if(maze(i, j) == 4) % change
35 -                xTarget = i + 1;
36 -                yTarget = j;
37 -            end
```

- Explanation:

  AStarMazeSolver.m is mainly based on the script A_Star.m, but modified to a **function** as "maze" that is generated by maze generator in part (a) should be passed to it as an argument.

The function "problem()" is deleted as "maze" is passed and there is no need to create a problem.

In the piece of code shown above, "MAX_X" and "MAX_Y" refer to max row and max column in "maze" respectively. And they are acquired by using function to find out the size of "maze".

By processing all values that are stored in "maze", the **walls** that are marked/numbered by **index 0** are stored into the array "OBSTACLE" and none of the successors should be on it.

The start position and end position are found using their corresponding numbers in "maze". The row and column of **start position** that is marked by **index 3** in "maze" should be stored in "xStart" and "yStart" respectively. The row and column of **end position** that is marked by **index 4** in "maze" should be stored in "xTarget" and "yTarget" respectively.

2) Change 2

- Position: expand_mazesolver.m, line 11 and line 14
- Screenshot:

```
9 –     for k = 1 : -1 : -1 % explore surrounding locations
10 –        for j = 1 : -1 : -1
11 –            if (k ~= j && k ~= -j)  % change
12 –                s_x = node_x + k;
13 –                s_y = node_y + j;
14 –                if( (s_x > 1 && s_x <= MAX_X - 1) && (s_y > 1 && s_y <= MAX_Y - 1)) % node within array bound % change
15 –                    flag = 1;
```

- Explanation:

expand_mazesolver.m is mainly based on expand.m, but the name is changed to expand_mazesolver.m to highlight the differences. When the function is called, the caller, basically the function AStarMazeSolver, will call its new name.

Previous code allows a node to move up and down diagonally. However, in case of maze solver, the path can **only go up, down, left and right without any diagonal movements**. Therefore, the condition in line 11 is modified to not only exclude the current node itself, but also prevent the node from moving to top left, top right, bottom left and bottom right corners.

In line 14, the **array bound** is modified. Unlike the previous problem, the "MAX_X" (max row) and "MAX_Y" (max column) also include the **boundaries of "maze"** which are marked by **index 8**, so the boundaries need to be excluded from the array bound as they cannot be legal path.

3) Change 3

- Position: dispMaze.m, line 27
- Screenshot:

```
27 –    cmap = [.12 .39 1;1 1 1; 0 0 0; 1 .5 0; .65 1 0; 1 0 0; 0 0 0; 0 0 0; .65 .65 .65]; % change (5: red, 6: black)
```

- Explanation:

dispMaze.m is copied from the file of maze generator to display the maze and the required output of A* maze solver.

The initialized colors for function "colormap" are modified, red for index 5 and black for index 6. They are used in result_mazesolver.m, where **index 5** is used to

display all the routes that A* has processed with **RED** color and **index 6** is to display the final solution with **BLACK** color.

4) Change 4

- Position: result_mazesolver.m, line 42 to 56
- Screenshot:

```
40          % change
41          % display all the routes that A* has processed with RED color
42 -        img = maze;
43 -        for i = 1 : 1 : QUEUE_COUNT
44 -            row = QUEUE(i, 2);
45 -            col = QUEUE(i, 3);
46 -            img(row, col) = 5;
47 -            dispMaze(img);
48 -        end
49          % display the final solution with BLACK color
50 -        PATH_COUNT = size(Optimal_path, 1);
51 -        for j = 1 : 1 : PATH_COUNT
52 -            row = Optimal_path(j, 1);
53 -            col = Optimal_path(j, 2);
54 -            img(row, col) = 6;
55 -            dispMaze(img);
56 -        end
```

- Explanation:

result_mazesolver.m is mainly based on result.m, but the name is changed to result_mazesolver.m to highlight the differences. When the function is called, the caller, basically the function AStarMazeSolver, will call its new name.

Previous code plots the target, shows the path and plot optimal path. However, to meet the requirements of maze solver, this part is modified to **display all the routes that A* has processed with RED color** and **display the final solution with BLACK color**.

First, a copy of "maze", the "img", is created to avoid changing the values that are stored in "maze" directly. By processing through the array **"QUEUE" that stored all nodes along the routes that A* has processed**, corresponding places in "img" are marked by **index 5**. In the same way, by processing through the array **"Optimal_path" that stored all nodes along the optimal path that A* has found**, corresponding places in "img" are marked by **index 6**. Finally, by calling the function "dispMaze" that has been modified in "change 3", both the maze that is generated by maze generator and the required output of A* maze solver can be displayed.