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)

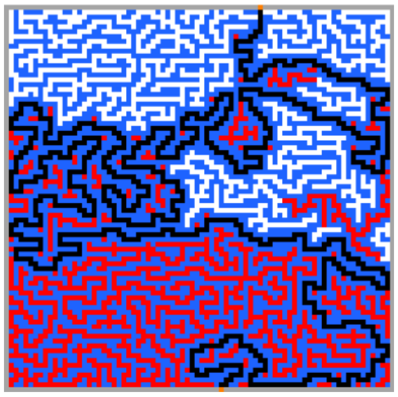


Figure 1. Sample output

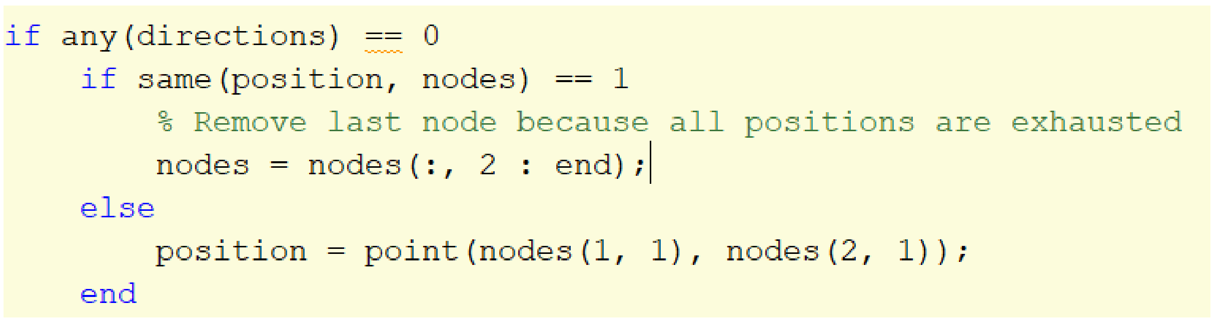
## Maze Generation Overview

This program takes in a user input to generate a maze using a custom version Breadth-first search algorithm. The user inputs a size *n* and difficulty *d* which is used to create a maze of *nxn* size. The `main` script controls the rest of this program, so this is the only script that needs to be called in order to generate a maze. Descriptions of the created functions are listed below.

## Code Modifications

*move.m*

Inputs: *maze, position, nodes, difficulty*
Outputs: *maze, position, nodes*
This function takes a maze and generates one new point along its current path. This function is recursively called within the main function until there are zero nodes. This function first calls `validateMove` to get a list of possible directions the function can go. If there are 0 possible directions and the current position is a node, the node is removed and the current position is set to the first node on the list. If there are 0 possible directions and the current position is *not* a node, the current position is set to the first node on the list. If there are 1, 2, 3, or 4 possible directions, the function does a couple of things. It first finds a random direction using the weighting described in the movement section of this README. It then finds the position of the previous position (whatever point was created just before this one). It then calls the `checkNodes` function. If it returns `true`, then add the current point to the `node` list. Regardless, the `currentPoint` is set equal to the `futurePoint` and that point in the maze is set to 1. *maze*, *pos*, and *nodes* are all returned in order to make this function easy to be called recursively.



```
if any(directions) == 0
    if same(position, nodes) == 1
        % Remove last node because all positions are exhausted
        nodes = nodes(:, 2 : end);
    else
        position = point(nodes(1, 1), nodes(2, 1));
    end
```

**Figure 1** move.m

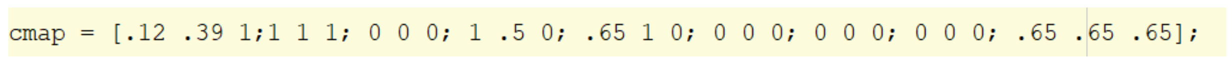
Modify Reason

1. In BFS maze generation, this function explores one path first, stores the coordinates into nodes. Once there is no possible move direction, the next position is set to the first nodes in the list. If this position already exists in the nodes list, the current position is move to the next nodes by deleting the first nodes. Because of changing

search strategy and new method to set position, the same function is necessary to modify in `point.m`.

*dispMaze.m*

Inputs: *maze*
Outputs: *None* (Note: displays maze)
This function formats the matrix to be graphed via heatmap. This is a two dimensional graph in which there are different colors based on the value in the matrix. That way, the walls can be grey, paths blue, and start/end orange.



```
cmap = [.12 .39 1;1 1 1; 0 0 0; 1 .5 0; .65 1 0; 0 0 0; 0 0 0; 0 0 0; .65 .65 .65];
```

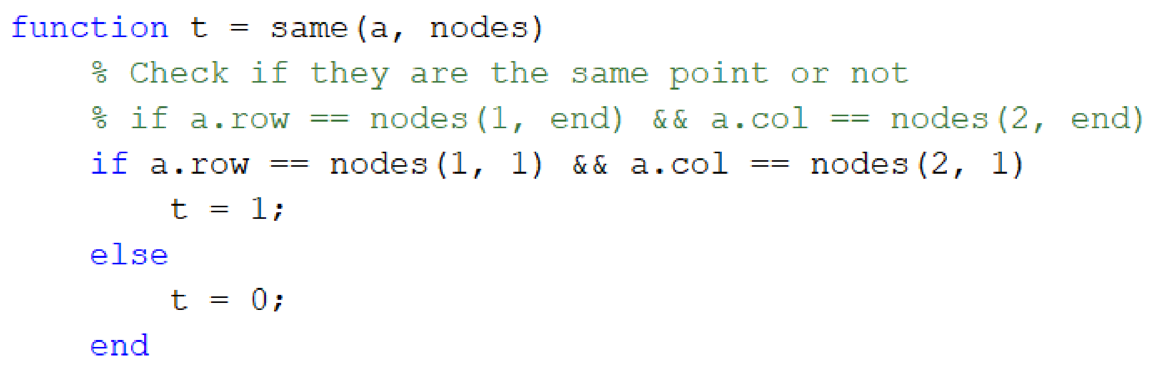
**Figure 2** dispMaze.m

Modify Reason

1. Cmap matrix control the different colors in maze. It has 9 rows, 3 columns, each row represents a color. First row represents wall, the second is path, the fourth is start point, the fifth is end point, and the last represent border. After changing the cmap(6, 1) from 0 to one, this row shows red color and the default triplet three zero represent black. As a result, the value of maze pathway is set to 2 and the value of expand is set to 5.

*point.m*

Point class
Properties: *row, col*
Methods: *point(a, b), adjust(maze, position, value), mazeValue(maze, position, a, b), same(a, nodes)*
The purpose of this class is to prevent the need for x and y components for all points throughout the maze. These functions are mostly one liners whos purpose is to make adjustments to the maze and positions simpler.



```
function t = same(a, nodes)
    % Check if they are the same point or not
    % if a.row == nodes(1, end) && a.col == nodes(2, end)
    if a.row == nodes(1, 1) && a.col == nodes(2, 1)
        t = 1;
    else
        t = 0;
    end
```

**Figure 3** point.m

Modify Reason

1. Same function is used for comparing a point with a special position recorded in nodes list. In DFS maze generation, same function uses a point “a” to compare with the last nodes. Because of the BFS, this function needs to used the first nodes to confirm weather this nodes and a point are the same.

**A Star Maze Solver Overview**

This program takes in a maze matrix to generate a maze and uses A* algorithm to get a maze pathway. It is runnable by calling the function with name ‘AStarMazeSolver(maze)’, the input of function is a ‘maze’ generated by the maze generator. The `AStarMazeSolver` function controls the rest of this program, so this is the only script that needs to be called in order to generate a maze pathway. Descriptions of the created functions are listed below.

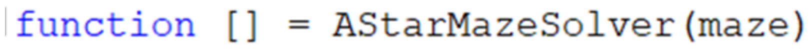
**Code Descriptions and Modifications**

*AStarMazeSolver.m*

Inputs: *maze*
This is the main function of the maze solver program. This is the also the main file that contains the algorithm. It uses `dispMaze` function to draw a colorful maze. Then, it uses traversal to identify the start and target points and create the OBSTACLE array. Next, create QUEUE inserted with the start point. The search part follows by calling `expand function`. Lastly, `result` function is called to format the maze pathway to a viewable format.

Modify Reason

1. Call the function

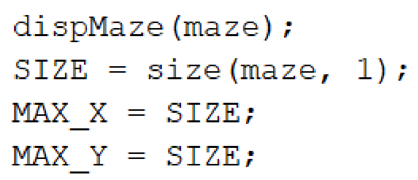


```
function [] = AStarMazeSolver(maze)
```

**Figure 4** AStarMazeSolver.m

(1) This function needs to run by calling the function with name ‘AStarMazeSolver(maze)’, the input of function is a ‘maze’ generated by the maze generator

2. Display maze and valuation

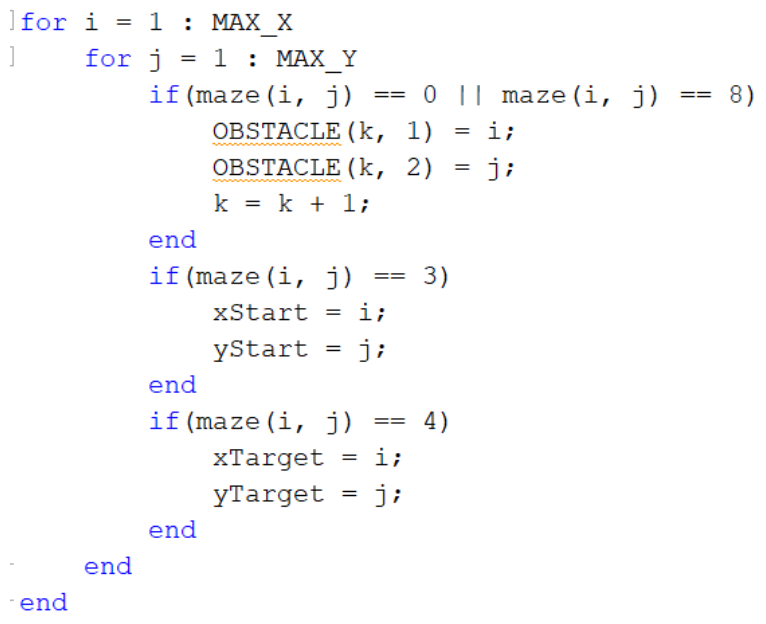


```
dispMaze(maze);
SIZE = size(maze, 1);
MAX_X = SIZE;
MAX_Y = SIZE;
```

**Figure 5** AStarMazeSolver.m

(1) Because the `problem.m` is useless and abandoned, A Start maze solver needs to load and draw a maze directly and valuate for those variables who are assigned in `problem.m`

3. Use traversal to create OBSTACLE, start and end points

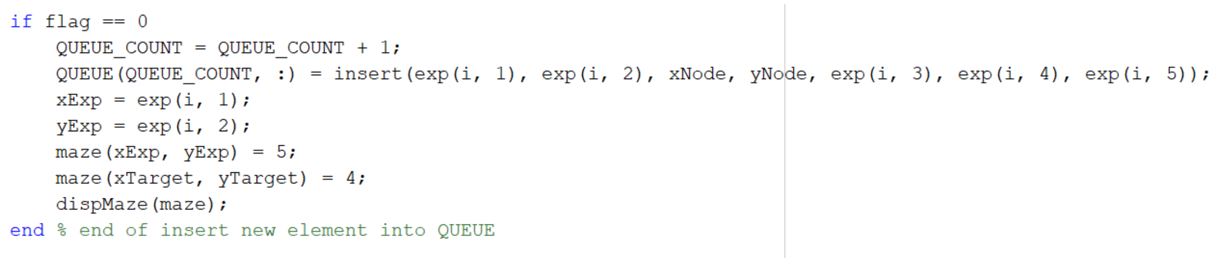


```
for i = 1 : MAX_X
    for j = 1 : MAX_Y
        if(maze(i, j) == 0 || maze(i, j) == 8)
            OBSTACLE(k, 1) = i;
            OBSTACLE(k, 2) = j;
            k = k + 1;
        end
        if(maze(i, j) == 3)
            xStart = i;
            yStart = j;
        end
        if(maze(i, j) == 4)
            xTarget = i;
            yTarget = j;
        end
    end
end
```

**Figure 6** AStarMazeSolver.m

(1) This part uses traversal to create OBSTACLE, start and end points.
(2) The OBSTACLE includes both wall and border, hence it searches all the maze value, if the value equals to 0 or 8, this point is recorded in OBSTACLE
(3) Because of the lost of `problem.m`, if the value equals to 3 or 4, this point is recorded as start or target respectively.

4. Display expand path

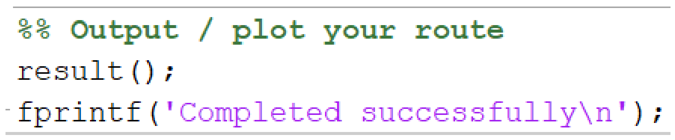


```
if flag == 0
    QUEUE_COUNT = QUEUE_COUNT + 1;
    QUEUE(QUEUE_COUNT, :) = insert(exp(i, 1), exp(i, 2), xNode, yNode, exp(i, 3), exp(i, 4), exp(i, 5));
    xExp = exp(i, 1);
    yExp = exp(i, 2);
    maze(xExp, yExp) = 5;
    maze(xTarget, yTarget) = 4;
    dispMaze(maze);
end % end of insert new element into QUEUE
```

**Figure 7** AStarMazeSolver.m

(1) This part assigns the value of each expand point to 5, and because this assignment also influences the target, so it is ought to revaluate the target to 4, then display the maze to show each expand step in screen.

5. Successful output



```
%% Output / plot your route
result();
fprintf('Completed successfully\n');
```

**Figure 8** AStarMazeSolver.m

(1) After calling the `result` function, this program prints “Completed successfully” to show it worked without any error.

*dispMaze.m*

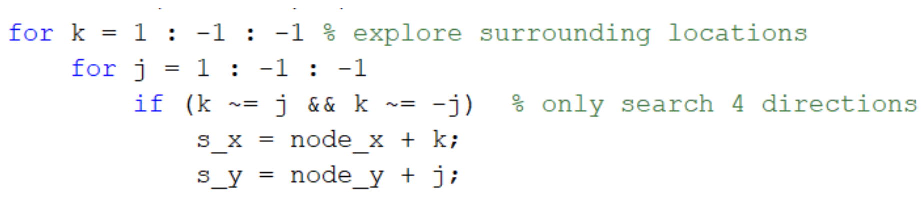
Inputs: *maze*
Outputs: *None* (Note: displays maze)
This function formats the matrix to be graphed via heatmap. This is a two dimensional graph in which there are different colors based on the value in the matrix. That way, the walls can be grey, paths blue, and start/end orange.

Modify Reason

1. Copy it from BFSMazeGenerator

*expand.m*

Inputs: *node_x, node_y, gn, xTarget, yTarget, OBSTACLE, MAX_X, MAX_Y*
This function takes a node and returns the expanded list of successors (child nodes), with the calculated fn values.



```
for k = 1 : -1 : -1 % explore surrounding locations
    for j = 1 : -1 : -1
        if (k ~= j && k ~= -j)  % only search 4 directions
            s_x = node_x + k;
            s_y = node_y + j;
```

**Figure 9** expand.m

Modify Reason

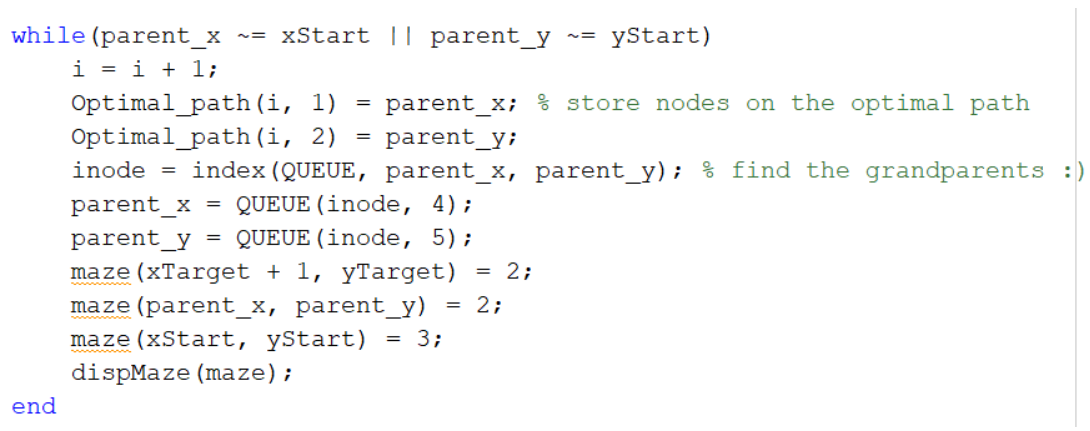
1. This function explores surrounding location by using k, j to traversal the nearest position. Because this maze solver only allows to explore the up, down , left, right direction, hence k cannot equals to j or -j, for in that case this maze can explore by going diagonally

*result.m*

This function formats the Optimal path, which starting from the last node, backtrack to its parent nodes until it reaches the start node.

Modify Reason

1. Display the maze pathway

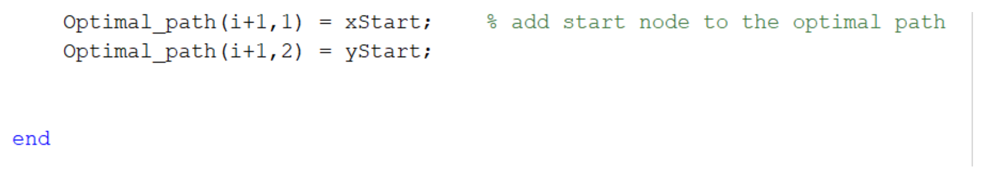


```
while(parent_x ~= xStart || parent_y ~= yStart)
    i = i + 1;
    Optimal_path(i, 1) = parent_x; % store nodes on the optimal path
    Optimal_path(i, 2) = parent_y;
    inode = index(QUEUE, parent_x, parent_y); % find the grandparents :)
    parent_x = QUEUE(inode, 4);
    parent_y = QUEUE(inode, 5);
    maze(xTarget + 1, yTarget) = 2;
    maze(parent_x, parent_y) = 2;
    maze(xStart, yStart) = 3;
    dispMaze(maze);
end
```

**Figure 10** result.m

(1) This part assigns the value of each pathway point to 2, and because this assignment also ignores the point below the target and influences the target point, so it is ought to assign the first point to 2 and the start point to 3, then display the maze to show each expand step in screen.

2. Delete the plot codes



```
        Optimal_path(i+1,1) = xStart;     % add start node to the optimal path
        Optimal_path(i+1,2) = yStart;


    end
```

**Figure 11** result.m

(1) In the previous `result.m`, there exist some codes about plotting the coordinates and the maze pathway line. But in this script, there is no needs to use plot method.