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)

Line 34 in main.

Depth first search is implemented in function 'move' to create the maze. The data structure is a graph.

The function 'move' first recursively searched for new positions until the current position has no direction to go. This step is the depth expansion in DFS, exploring as many nodes as possible on one branch. When the current position has '0 direction' (the student uses function 'validateMove' to check directions), it means the last node on this branch is being visited.

For DFS, the second step is to repeatedly trace back to the shallower nodes and expand them. In function: When the current position has 0 direction, 'move' sets the current position to the last position (the shallower node) and check whether it has any directions. If direction is greater than 0, the depth expansion will be implemented. Otherwise, the function will continue checking last positions.

When all nodes have been visited, the search is completed. The function 'move' uses a nodes-list to check whether all nodes have been visited. If a node has 0 direction, then remove the node from the list. When there is 0 node in the list, the function 'move' stops.

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

```
79          % Find previous number 1
80 —        if mazeValue(maze, position, -1, 0) == 1
81 —            previousPosition = point(position.row - 1, position.col);
82 —        elseif mazeValue(maze, position, 1, 0) == 1
83 —            previousPosition = point(position.row + 1, position.col);
84 —        elseif mazeValue(maze, position, 0, -1) == 1
85 —            previousPosition = point(position.row , position.col - 1);
86 —        else % mazeValue(maze, position, 0, 1) == 1
87 —            previousPosition = point(position.row, position.col + 1);
88 —        end
89          % Check if node created
90 —        if checkNode(futurePosition, previousPosition) == 1
91 —            nodes(1, end + 1) = position.row;
92 —            nodes(2, end) = position.col;
93 —        end
```

For the node above the Start (at the beginning), the positions of up, left, right directions are all 0, and the value of down direction is 3(the Start). Therefore, the 'previousPosition' of the node above the Start is always the node on its right according to the code. If the future Position is also on its right, this future position will not be added into the node list, which causes the logic problem.

(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.

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.

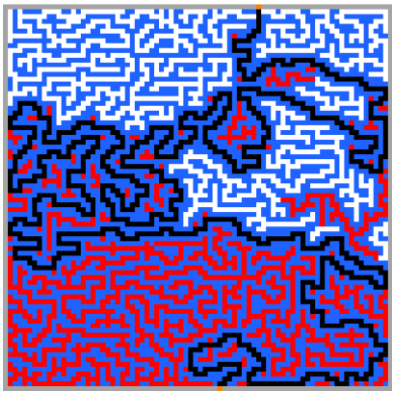

Figure 1. Sample output

First, 'clear' in main was removed to preserve the parameters in workspace.Because of the different use of 'Astar', the function 'problem' was also removed.Instead, the position of 'Start','Target' and 'Obstacles' were loaded by for-loop in main function.(In Figure 2.)

```
16      %load size,Start and Target
17 —    r=size(maze,1);
18 —    xStart=r;
19 —    for q = 1 : r
20 —      if(maze(r,q)==3)
21 —        yStart=q;
22 —      end
23 —    end
24
25 —    xTarget=1;
26 —    for p = 1 : r
27 —      if(maze(1,p)==4)
28 —        yTarget=p;
29 —      end
30 —    end
31
32      % OBSTACLE: [X val, Y val]
33 —    OBSTACLE = [];
34 —    k = 1;
35 —    for i = 1 : r
36 —        for j = 1 : r
37 —          if((maze(i, j) == 0)||(maze(i, j) == 8))%load the position of obstacles
38 —              OBSTACLE(k, 1) = i;
39 —              OBSTACLE(k, 2) = j;
40 —              k = k + 1;
41 —          end
```

Figure 2. Load Positions

Second, Manhattan distance is applied to calculate the hueristic estimate.The reason for this change is that Manhattan distance is closer to the physical truth than ‘the distance of two points’. In Figure 3.

```
4       function dist = distance(x1, y1, x2, y2)
5 —     dist =abs(x1-x2)+abs(y1-y2); % h1
```

Figuire 3. Distance

Third, in function ‘expand’, the original code can control 8 directions. However, only 4 directions (up, down, left, right) are involved in ‘AStarMazeSolver’. Therefore, the direction control parameters need to be changed. In Figure 4.

```
9 —     for k = 1 : -1 : -1 % explore surrounding locations
10 —        for j = 1 : -1 : -1
11 —            if (k ~= j && k ~= -j) % the node itself is not its successor
12 —                s_x = node_x + k;
13 —                s_y = node_y + j;
```

Figure 4. expand

Finally, in function ‘result’, original style of display is not applicative, all plot function and relative calculations were removed. Function ‘dispMaze’ is used again to display the final result. Set the routes that A* has processed to value 7 in main. (and avoid covering the Target)(in Figure 5) The color is set to red in

'dispMaze'.(in Figure 7.)

```
87      % A*: find the node in QUEUE with the smallest f(n), returned by min_fn
88 —    index_min_node = min_fn(QUEUE, QUEUE_COUNT);
89 —    if (index_min_node ~= -1)
90          % set current node (xNode, yNode) to the node with minimum f(n)
91 —        xNode = QUEUE(index_min_node, 2);
92 —        yNode = QUEUE(index_min_node, 3);
93 —        path_cost = QUEUE(index_min_node, 6); % cost g(n)
94          % move the node to OBSTACLE
95 —        OBST_COUNT = OBST_COUNT + 1;
96 —        OBSTACLE(OBST_COUNT, 1) = xNode;
97 —        OBSTACLE(OBST_COUNT, 2) = yNode;
98 —         if(maze(xNode,yNode)~=4)
99 —            maze(xNode,yNode)=7;
100 —           dispMaze(maze);
101 —       end
102 —       QUEUE(index_min_node, 1) = 0;
103 —       discover = discover+1;
104 —   else
105 —        NoPath = 0; % there is no path!
106 —   end
```

Figure 5. main (set value)

Finally, set the final solution to value 5 in result: As the route is saved from the Target to the Start, a while-loop is used to display them in inverted order. (in Figure 6.)

```
39 —            while(i>1)
40 —            maze( Optimal_path(i, 1),Optimal_path(i, 2))=5;
41 —            dispMaze(maze);
42 —            i=i-1;
43 —        end
```

Figure 6. result(set value)

```
25      % Color map
26
27 —    cmap = [.12 .39 1;1 1 1; 0 0 0; 1 .5 0;.65 1 0;0 0 0;0 0 0;1 0 0;.65 .65 .65];
28
29
```

black red

Figure 7. dispMaze

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

Compared with Astar algorithm, DFS doesn't consider about the distance. DFS only searches for directions to move. (The priority of the direction is up > left >right >down.) Therefore, all of the function and parameters related to distance should be removed.

The first step is to expand new positions. If the current position has direction to expand, the solver firstly checks whether the node has been expanded. In Figure 8.

```
63 —    if(exp_count>0)
64 —        for i = 1 : exp_count
65 —            for j = 1 : QUEUE_COUNT
66 —                if(exp(i, 1) == QUEUE(j, 2) && exp(i, 2) == QUEUE(j, 3))
67 —                    QUEUE(j,1)=0;
68 —                    flag = 1;
69 —                end
70 —            end
71 —            if flag == 0
72 —                QUEUE_COUNT = QUEUE_COUNT + 1;
73 —                QUEUE(QUEUE_COUNT, :) = insert(exp(i, 1), exp(i, 2), xNode, yNode);
74 —            end % end of insert new element into QUEUE
75 —        end
```

Figure 8.

Then, the solver will set the new node to be the current position. The solver also adds the current node into 'OBSTACLE', which can prevent DFS from infinite-loop. In Figure 9.

```
81 —            xNode=exp(i,1);
82 —            yNode=exp(i,2);
83 —            QUEUE(QUEUE_COUNT,1)=0;
84 —            expandnode=expandnode+1;
85 —            OBST_COUNT = OBST_COUNT + 1;
86 —            OBSTACLE(OBST_COUNT, 1) = xNode;
87 —            OBSTACLE(OBST_COUNT, 2) = yNode;
88 —            if(maze(xNode,yNode)~=4)
89 —                maze(xNode,yNode)=7;
90 —                dispMaze(maze);
91 —            end
```

Figure 9.

Otherwise, if the current node has no direction to expand, solver will set the parent node to be the current node, and the parent node should be checked. In Figure 10.

```
92 —    else
93 —        for i = 1 : QUEUE_COUNT
94 —            if((QUEUE(i,2)==xNode)&&(QUEUE(i,3)==yNode))
95 —                xNode=QUEUE(i,4);
96 —                yNode=QUEUE(i,5);
97 —            end
98 —        end
99 —    end
```

Figure 10.

These two steps should be executed continually until the algorithm find the final route.

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

The difference between Astar algorithm and Greedy algorithm is that Greedy doesn't consider about the 'path cost'. Therefore, the only change required is to

set the 'path cost' to zero in function 'expand'. In Figure 11.

```
21 —        if (flag == 1)
22 —            exp_array(exp_count, 1) = s_x;
23 —            exp_array(exp_count, 2) = s_y;
24 —            exp_array(exp_count, 3) = 0; % cost g(n)
25 —            exp_array(exp_count, 4) = distance(xTarget, yTarget, s_x, s_y); % cost h(n)
26 —            exp_array(exp_count, 5) = exp_array(exp_count, 3) + exp_array(exp_count, 4); % f(n)
27 —            exp_count = exp_count + 1;
28 —        end
```

Figure 11. expand in Greedy

(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).

Total path cost is the number of elements in 'optimal_path'. It starts from the Target, and then continually searches for the parent node in QUEUE(i,4) and QUEUE(i,5).I get the data of 'optimal_path' from the loop time parameter 'i'.

Number of nodes discovered is the elements in QUEUE, which QUEUE(i,1) =0. I set up a for-loop to count the number of the elements.

Number of nodes expanded is the total number of elements in QUEUE. I get this data from the parameter 'QUEUE-COUNT'.

```
29 —    while(parent_x ~= xStart || parent_y ~= yStart)
30 —        i = i + 1;
31 —        Optimal_path(i, 1) = parent_x; % store nodes on the optimal path
32 —        Optimal_path(i, 2) = parent_y;
33 —        inode = index(QUEUE, parent_x, parent_y); % find the grandparents
34 —        parent_x = QUEUE(inode, 4);
35 —        parent_y = QUEUE(inode, 5);
36 —    end
37 —    Optimal_path(i+1,1) = xStart;    % add start node to the optimal path
38 —    Optimal_path(i+1,2) = yStart;
39 —    cost=i;
40 —        while(i>1)
41 —        maze( Optimal_path(i, 1),Optimal_path(i, 2))=5;
42 —        dispMaze(maze);
43 —        i=i-1;
44 —        end
45 — end
46 — expandnode=0;
47 — for i=1:QUEUE_COUNT
48 —    if(QUEUE(i,1)==0)
49 —        expandnode=expandnode+1;
50 —    end
51 — end
52 — fprintf("path=%d",cost);
53 — fprintf("discover=%d",QUEUE_COUNT);
54 — fprintf("expand=%d",expandnode);
```

Figure 12. Code

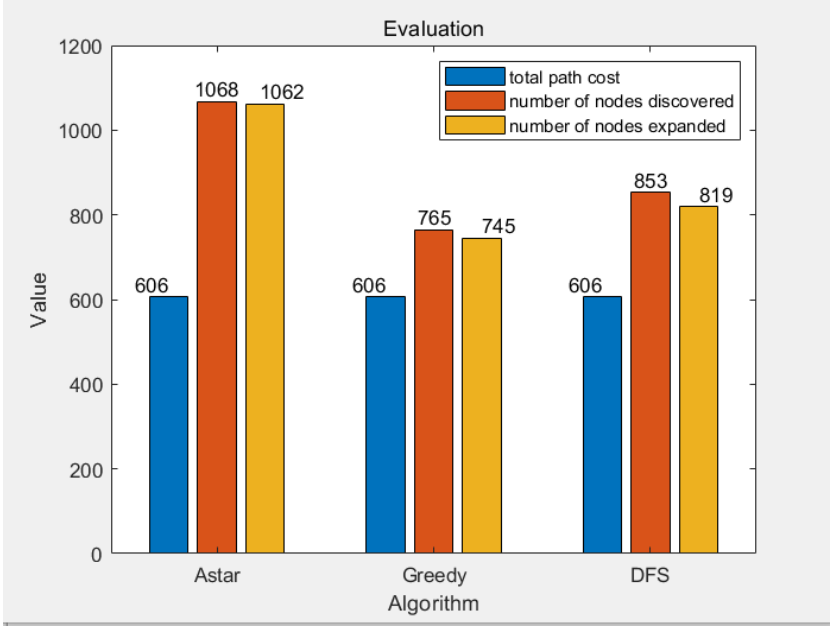

Figure 13. bar

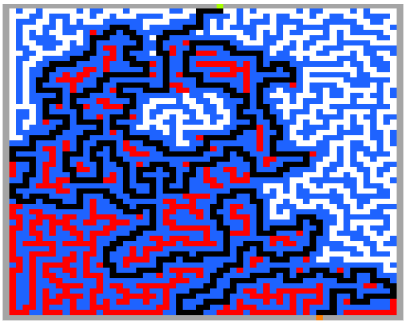

Figure 14. Astar

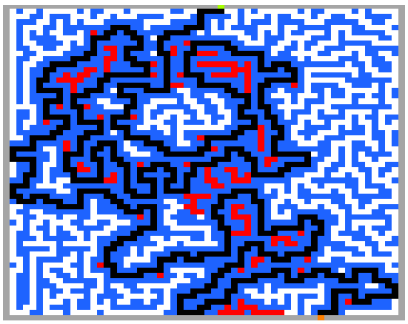

Figure 15. Greedy

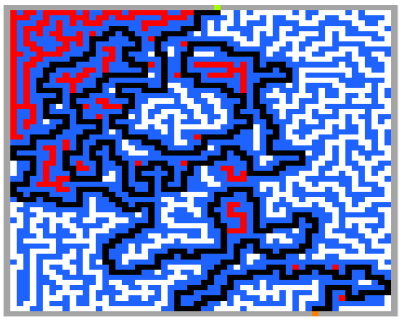

Figure 16. DFS

```
>> AStarMazeSolver(maze)
path=606discover=1068expand=1062>>
>> GreedyMazeSolver(maze)
path=606discover=765expand=745>>
>> DFSMazeSolver(maze)
path=606discover=853expand=819>>
```

Figure 17. Data