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)

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

(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.

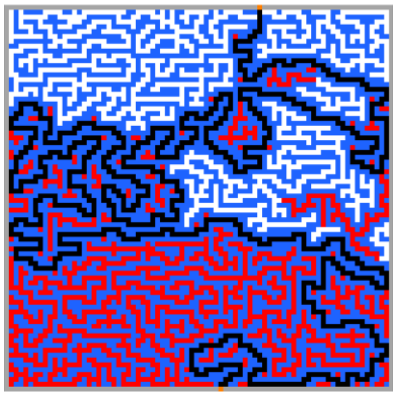


Figure 1. Sample output

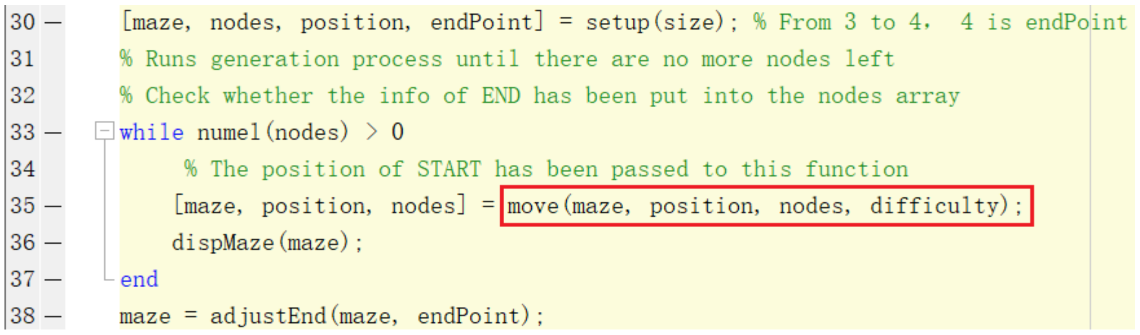
(d) Based on the previous AStarMazeSolver, implement a maze solver using the DFS algorithm. Similar to question (c), the solver need by called and ran by command '**DFSMazeSolver(maze)**'. (3 marks)

(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)

(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).

ANSWER:

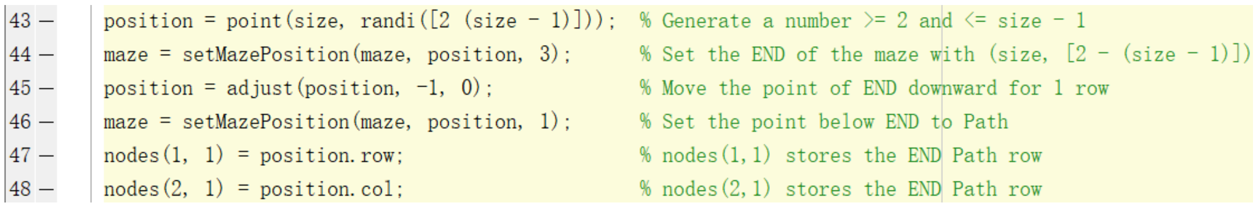
(a)



```
[maze, nodes, position, endPoint] = setup(size); % From 3 to 4,  4 is endPoint
% Runs generation process until there are no more nodes left
% Check whether the info of END has been put into the nodes array
while numel(nodes) > 0
     % The position of START has been passed to this function
    [maze, position, nodes] = move(maze, position, nodes, difficulty);
    dispMaze(maze);
end
maze = adjustEnd(maze, endPoint);
```

main.m

Above is the searching part of *MazeGeneration-master*, which could generate maze matrixes following user's instruction. A **DFS method** is used to create maze in a function named *"move()"*.

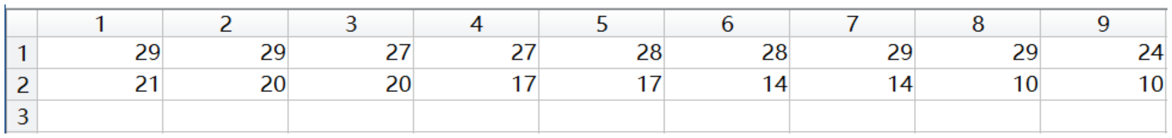


```
position = point(size, randi([2 (size - 1)])); % Generate a number >= 2 and <= size - 1
maze = setMazePosition(maze, position, 3);     % Set the END of the maze with (size, [2 - (size - 1)])
position = adjust(position, -1, 0);            % Move the point of END downward for 1 row
maze = setMazePosition(maze, position, 1);     % Set the point below END to Path
nodes(1, 1) = position.row;                    % nodes(1,1) stores the END Path row
nodes(2, 1) = position.col;                    % nodes(2,1) stores the END Path row
```

setup.m

Before entering into the *"move()"* part, a stack named "nodes" has been initialized in a function named *"setup()"*. In *"setup()"* a stack structure named "nodes" has been constructed to store position of the point one row above the entrance.

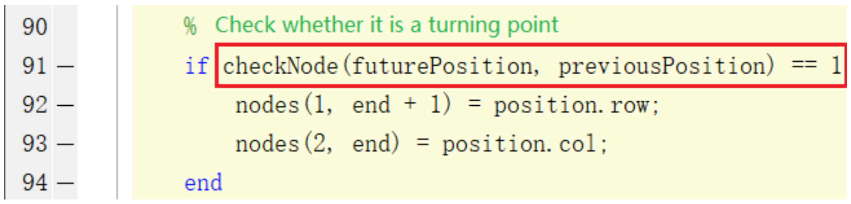
Within *"move()"* function, a **DFS logic** could be observed to implement maze construction.



	1	2	3	4	5	6	7	8	9
1	29	29	27	27	28	28	29	29	24
2	21	20	20	17	17	14	14	10	10
3									

"nodes" structure

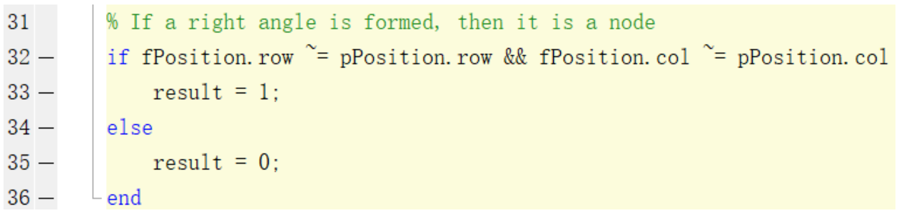
First, "nodes" has been passed to *"move()"* containing position of start point and previous turning points (corners). The structure of “nodes” is shown above. Within the picture, each column of “nodes” stores a turning point of maze matrix which is used as branching nodes in tree diagram later to generate new route for the maze.



```
    % Check whether it is a turning point
    if checkNode(futurePosition, previousPosition) == 1
        nodes(1, end + 1) = position.row;
        nodes(2, end) = position.col;
    end
```

move.m

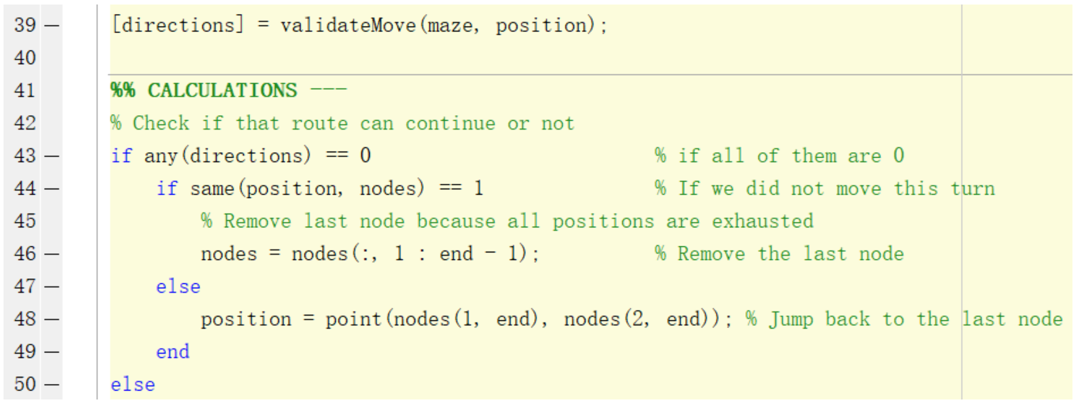
This part of code works like a push-in function in **DFS method**. It selects out new branches and pushes them into the end of a stack called “nodes”. However, “nodes” is checked from its bottom to its top, so the code implement a representative **DFS tree search method**.



```
% If a right angle is formed, then it is a node
if fPosition.row ~= pPosition.row && fPosition.col ~= pPosition.col
    result = 1;
else
    result = 0;
end
```

checkNode.m

Here is the code from *“checkNode()”*. It is clear to see that only corners have been selected out.



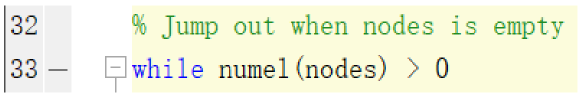
```
39  [directions] = validateMove(maze, position);
40
41  %% CALCULATIONS ---
42  % Check if that route can continue or not
43  if any(directions) == 0                        % if all of them are 0
44      if same(position, nodes) == 1              % If we did not move this turn
45          % Remove last node because all positions are exhausted
46          nodes = nodes(:, 1 : end - 1);         % Remove the last node
47      else
48          position = point(nodes(1, end), nodes(2, end)); % Jump back to the last node
49      end
50  else
```

move.m

Above code operates like a pop-out function in **DFS method**. The process is divided into two cases.

First case: As function reaches a point which could not generate a new point to go any more, the 43th line of code will be implemented. If present point is the last point in “nodes” which meets the requirement of code line 44, present point will be popped out from “nodes” (line 46).

Second case: When function meets the same situation as the first one. If preent node is not the last point in “nodes”, the 48th line of code will force function to jump back to the last point stores in “nodes”.

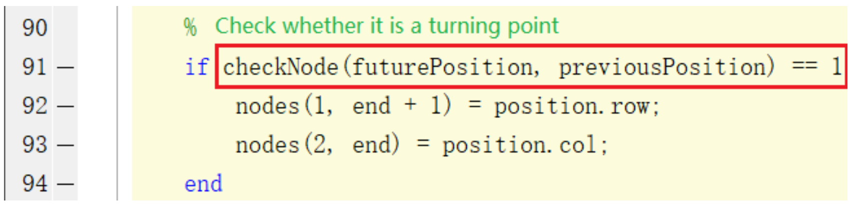


```
% Jump out when nodes is empty
while numel(nodes) > 0
```

main.m

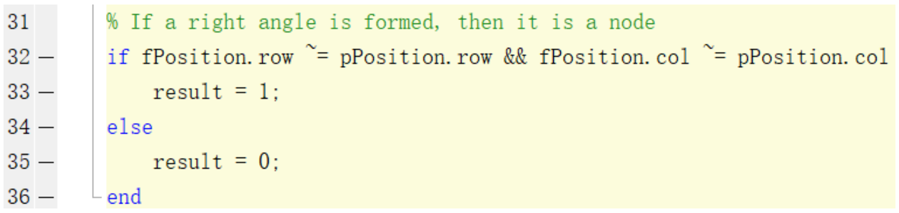
Finally, when “nodes” has been reduced to an empty stack (matrix), which means the entire maze has been completely explored, function will jump out of the **tree searching** while loop. And as a result, **only one solution** has been generated.

(b)



```
    % Check whether it is a turning point
    if checkNode(futurePosition, previousPosition) == 1
        nodes(1, end + 1) = position.row;
        nodes(2, end) = position.col;
    end
```

move.m



```
% If a right angle is formed, then it is a node
if fPosition.row ~= pPosition.row && fPosition.col ~= pPosition.col
    result = 1;
else
    result = 0;
end
```

checkNode.m

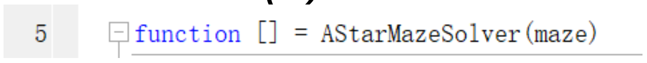
Be inferred from first question, the maze is generated by using **DFS method**. However, the process of expanding new points from this program is different from standard DFS method.

In **standard DFS**, when a branching node is reached, its child nodes will be discovered and added into head of the queue. And after program has reached an end leaf, it will jump back to one of the sibling nodes of present point’s parent node.

However, by executing above code, function pushes the branching node itself instead of its child nodes into the stack during expanding process. This modification completely changes the entire traceback process. Specifically speaking, during each time of traceback, function firstly jumps back to the branch point which generates duplicated meaningless searching work. Consequently, this program’s search method is no longer a **standard DFS** method any more.

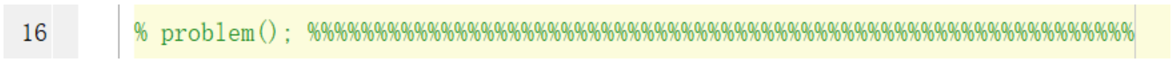
(c)

AStarMazeSolver(…)



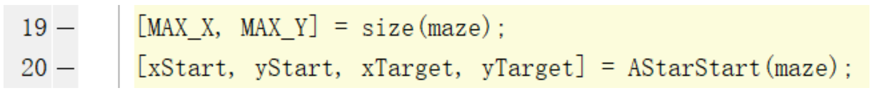
```
function [] = AStarMazeSolver(maze)
```

First, to meet the requirement demonstrated in report, the program has been changed into a function named *“AStarMazeSolver()”*. Which takes in a maze matrix and returns nothing at all.



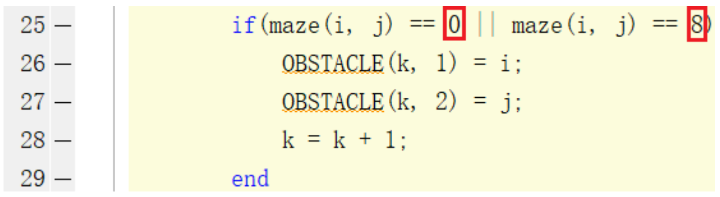
```
% problem(); %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Then *“problem()”* has been commented out in that the program does not need to generate a new maze itself.



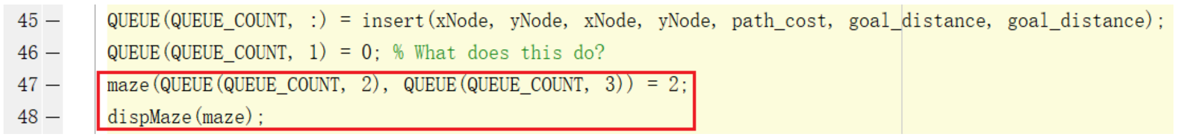
```
[MAX_X, MAX_Y] = size(maze);
[xStart, yStart, xTarget, yTarget] = AStarStart(maze);
```

Next, function sets MAX_X and MAX_Y to be border of the maze. Then, it assigns position of the point one row above entrance to be start point’s position and position of the point one row below exit to be target point’s position. This series of actions is accomplished by a function named *“AStarStart()”*.



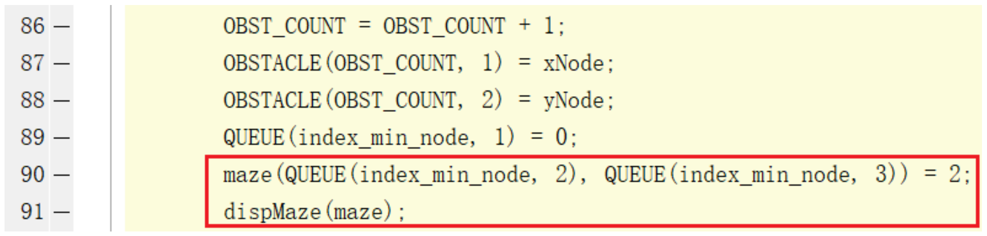
```
        if(maze(i, j) == 0 || maze(i, j) == 8)
            OBSTACLE(k, 1) = i;
            OBSTACLE(k, 2) = j;
            k = k + 1;
        end
```

Search through the entire maze to add **Walls** (0) and **Borders** (8) into a matrix named OBSTACLE.



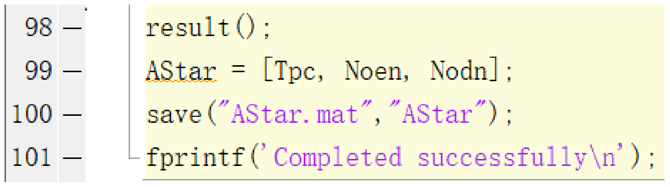
```
QUEUE(QUEUE_COUNT, :) = insert(xNode, yNode, xNode, yNode, path_cost, goal_distance, goal_distance);
QUEUE(QUEUE_COUNT, 1) = 0; % What does this do?
maze(QUEUE(QUEUE_COUNT, 2), QUEUE(QUEUE_COUNT, 3)) = 2;
dispMaze(maze);
```

After function expands the first point, maze will be refreshed and displayed to user. The expanded point will be changed from 1 to 2, which is colored into red in *“dispMaze()”* function.



```
        OBST_COUNT = OBST_COUNT + 1;
        OBSTACLE(OBST_COUNT, 1) = xNode;
        OBSTACLE(OBST_COUNT, 2) = yNode;
        QUEUE(index_min_node, 1) = 0;
        maze(QUEUE(index_min_node, 2), QUEUE(index_min_node, 3)) = 2;
        dispMaze(maze);
```

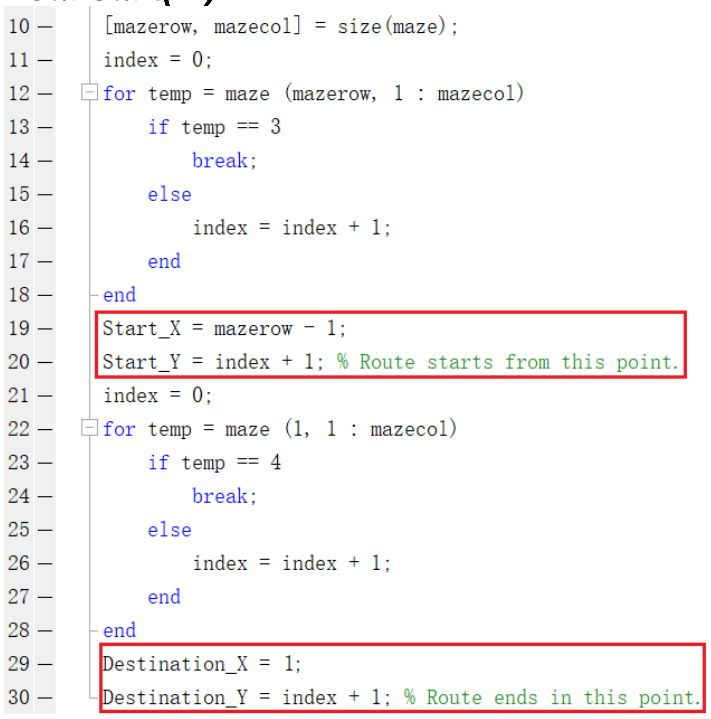
When a new point is expanded, function will refresh maze matrix once as well, then display it to the user.



```
result();
AStar = [Tpc, Noen, Nodn];
save("AStar.mat","AStar");
fprintf('Completed successfully\n');
```

Finally, the optimal route will be generated and demonstrated in *“result()”* function. Three integers: “Total path cost”, “Number of expanded nodes” and “Number of discovered nodes” will be constructed within *“result()”* as well. Finally, saves the data into document and terminates itself by printing “Completed successfully” into console.

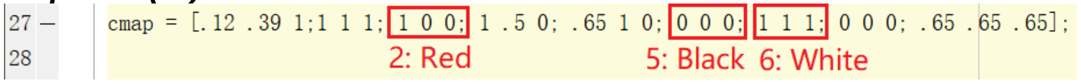
AStarStart(…)



```
[mazerow, mazecol] = size(maze);
index = 0;
for temp = maze (mazerow, 1 : mazecol)
    if temp == 3
        break;
    else
        index = index + 1;
    end
end
Start_X = mazerow - 1;
Start_Y = index + 1; % Route starts from this point.
index = 0;
for temp = maze (1, 1 : mazecol)
    if temp == 4
        break;
    else
        index = index + 1;
    end
end
Destination_X = 1;
Destination_Y = index + 1; % Route ends in this point.
```

This function search through the first row and last row of maze matrix and returns 4 numbers to locate 2 points. The former two is position of the point one row above maze entrance and the latter two is position of maze exit.

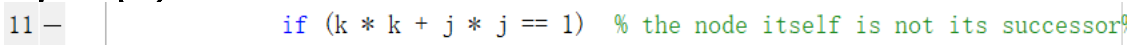
dispMaze(…)



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

Here a little bit of modification has been done to let this function display points with value of 2 in red, value of 5 in black and value of 6 in white.

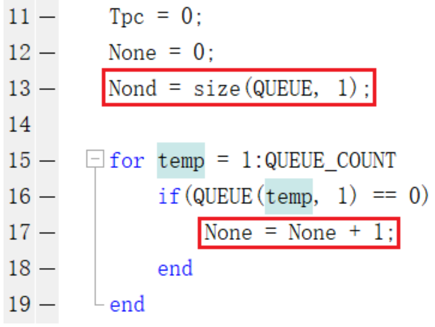
expand(…)



```
            if (k * k + j * j == 1)  % the node itself is not its successor
```

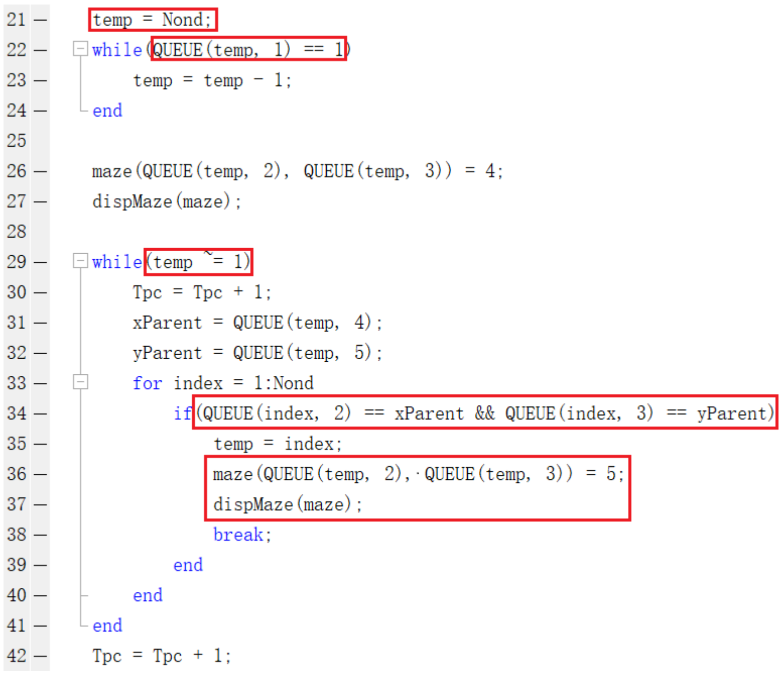
Check merely the point’s successors: (x+1, y), (x-1, y), (x, y+1) and (x, y-1).

result()



```
Tpc = 0;
None = 0;
Nond = size(QUEUE, 1);

for temp = 1:QUEUE_COUNT
    if(QUEUE(temp, 1) == 0)
        None = None + 1;
    end
end
```

Initialize “total path cost counter” and “number of nodes expanded counter”. And “number of nodes discovered” equals size of QUEUE structure which contains the position of points from the point one row above maze entrance to maze exit.



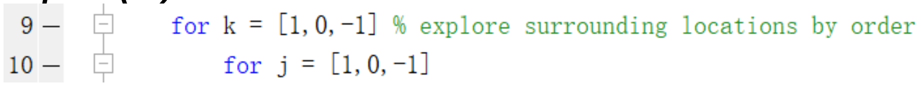
```
temp = Nond;
while(QUEUE(temp, 1) == 1)
    temp = temp - 1;
end

maze(QUEUE(temp, 2), QUEUE(temp, 3)) = 4;
dispMaze(maze);

while(temp ~= 1)
    Tpc = Tpc + 1;
    xParent = QUEUE(temp, 4);
    yParent = QUEUE(temp, 5);
    for index = 1:Nond
        if(QUEUE(index, 2) == xParent && QUEUE(index, 3) == yParent)
            temp = index;
            maze(QUEUE(temp, 2), QUEUE(temp, 3)) = 5;
            dispMaze(maze);
            break;
        end
    end
end
Tpc = Tpc + 1;
```

This part uses QUEUE to traceback optimal route by applying parent point's position of present point to search previous point to jump to.

d)

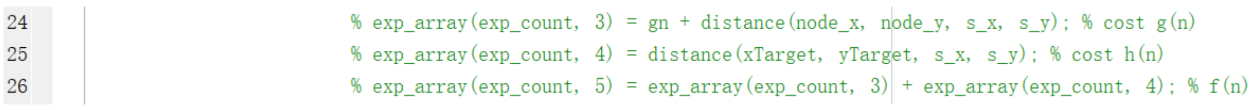
Comparing to the original AStar function, several parts have been modified:

Expand(…)



```
for k = [1,0,-1] % explore surrounding locations by order
    for j = [1,0,-1]
```

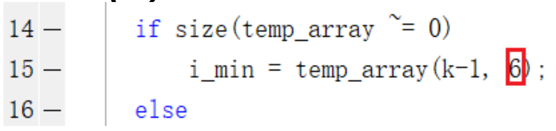
Expand order has been changed to "Up first, left second, right then, down last".



```
% exp_array(exp_count, 3) = gn + distance(node_x, node_y, s_x, s_y); % cost g(n)
% exp_array(exp_count, 4) = distance(xTarget, yTarget, s_x, s_y); % cost h(n)
% exp_array(exp_count, 5) = exp_array(exp_count, 3) + exp_array(exp_count, 4); % f(n)
```

The *"expand()"* expands 5 rows instead 8 in that **DFS method** merely has to store present position and parent's position of discovered points.

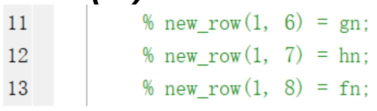
min_fn(…)



```
if size(temp_array ~= 0)
    i_min = temp_array(k-1, 6);
else
```

The *"min_fn()"* dose not select points anymore. Instead, it just takes one child point form branching point.

insert(…)



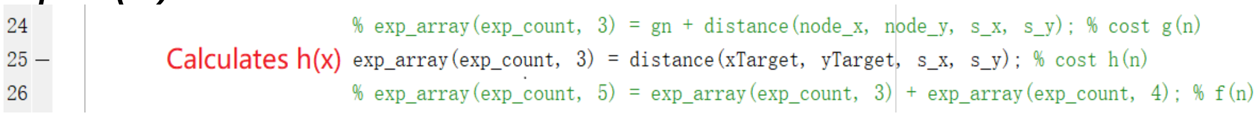
```
% new_row(1, 6) = gn;
% new_row(1, 7) = hn;
% new_row(1, 8) = fn;
```

In *"insert()"* function, some parts has been commented out to coorperate with the other part of function.

e)

Comparing to the original AStar function, several parts of code have been modified as well:

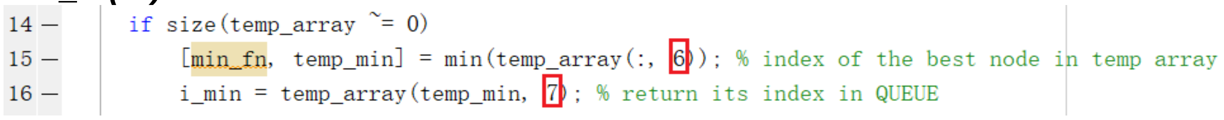
expand(…)



```
% exp_array(exp_count, 3) = gn + distance(node_x, node_y, s_x, s_y); % cost g(n)
exp_array(exp_count, 3) = distance(xTarget, yTarget, s_x, s_y); % cost h(n)
% exp_array(exp_count, 5) = exp_array(exp_count, 3) + exp_array(exp_count, 4); % f(n)
```

Within *"expand()"* function, rows for g(x) and f(x) have been deleted in that the function merely needs h(x) to decide which point to go.

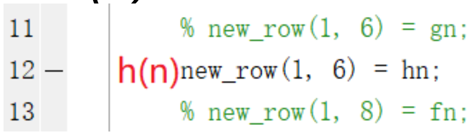
min_fn(…)



```
if size(temp_array ~= 0)
    [min_fn, temp_min] = min(temp_array(:, 6)); % index of the best node in temp array
    i_min = temp_array(temp_min, 7); % return its index in QUEUE
```

In *"min_fn()"* instead of selecting out the point with smallest f(x), point with smallest h(x) has been selected out.

insert(…)



```
% new_row(1, 6) = gn;
new_row(1, 6) = hn;
% new_row(1, 8) = fn;
```

In *"insert()"* function, the original 6th and 8th rows have been commented out. Only h(n) has been kept to evaluate points' quality.

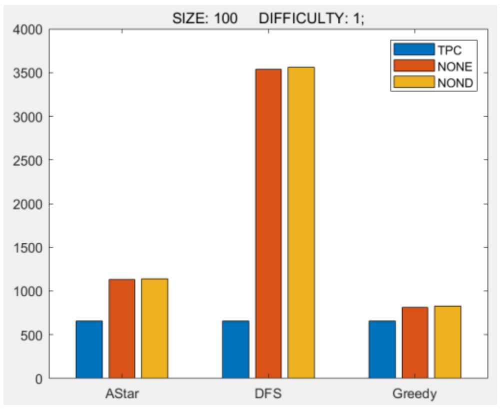
f)

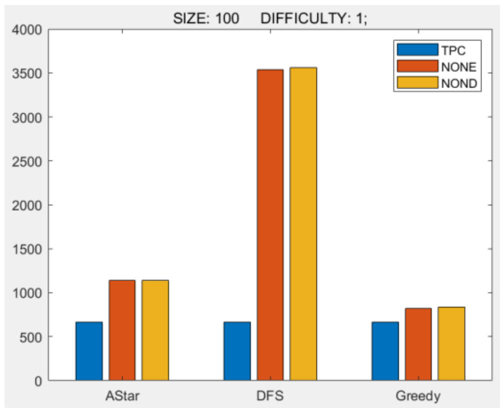
Result Evaluation:

Total path cost:
Total path cost equals the number of nodes visited within optimal path traceback process from target node to start node.

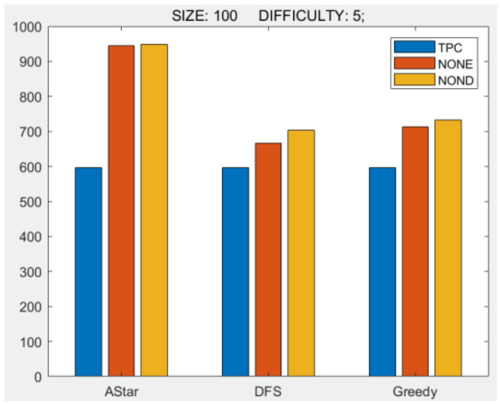
Number of nodes discovered:
Number of nodes discovered equals size of the QUEUE, in that every discovered node have been stored into it.

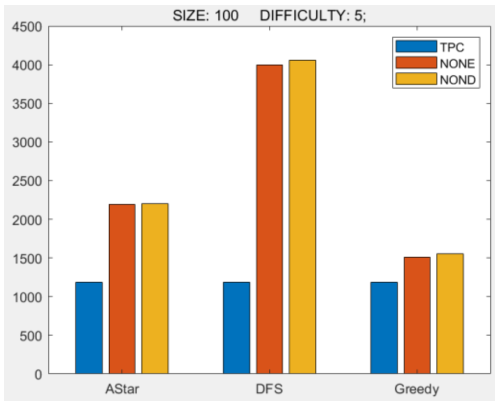
Number of nodes expanded:
Number of nodes expanded is the number of visited nodes whose first column in QUEUE stores value 0. In other words, if QUEUE (_, 1) == 0, this node has already been visited.



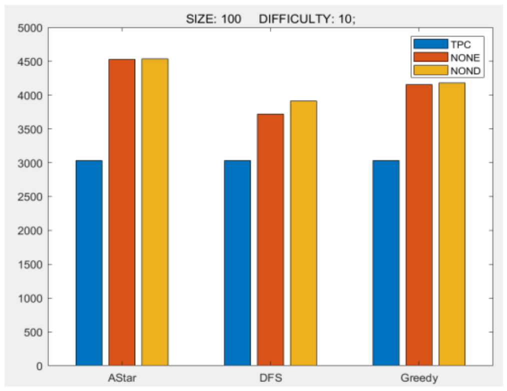


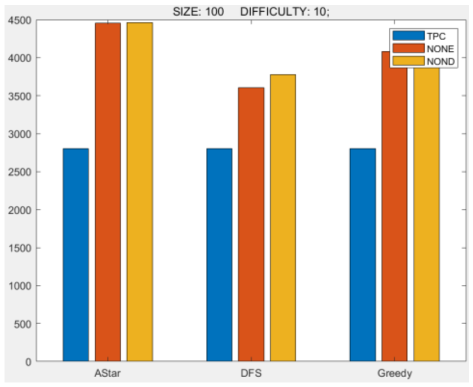
Above is comparing figures of three different searching methods towards a maze of size 100 and difficulty 1. With the same total path cost, AStar and Greedy search have almost same **None** and **Nond** (AStar's is slightly higher). However, DFS's **None** and **Nond** is dramatically high in that it searches driftlessly.





When difficult is adjusted to 5, AStar and Greedy search methods still have almost same **None** and **Nond** in which AStar is slightly higher as well. Interestingly, DFS's data size fluctuate dramatically under this circumstance which is caused by its driftlessness.





Eventually, difficulty has been adjusted to 10. And within this situation, AStar and

Greedy search method are confused by the winding optimal route. As a result, **DFS's driftless search** sometime becomes an advantage which saves some searching space.

From previous analyze, several evaluations could be generated:

1) Each of these three searching methods could find the **optimal path**.

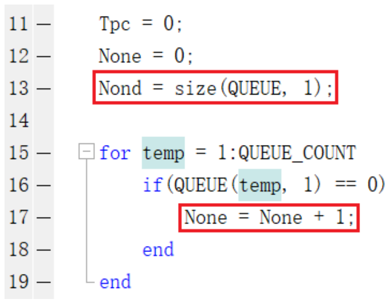
2) **Nond** is always slightly higher than **None** from maze applying same searching method.

3) Towards the same maze, **Astar's Nond** and **None** are always slightly higher than **Greedy's**.

4) With low difficulty, **AStar** and **Greedy** often perform better than **DFS**. **DFS's** performance is unpredictable because if its **driftless searching method**. However, whsen difficulty increases, **AStar** and **Greedy** always expend great amount of space and time to check path's veracity. Conversely, **DFS** sometimes performs better than the others.

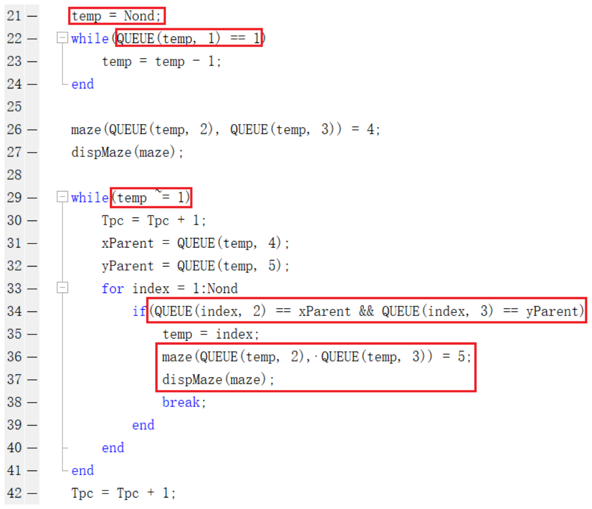
Explanation of gaining this data from QUEUE:

Three search methods use same method to gain information from QUEUE structure.



```
Tpc = 0;
None = 0;
Nond = size(QUEUE, 1);

for temp = 1:QUEUE_COUNT
    if(QUEUE(temp, 1) == 0)
        None = None + 1;
    end
end
```

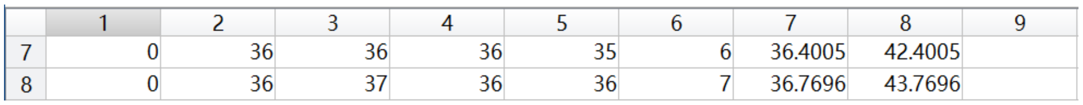
First, **Tpc** and **None** are initialized to be 0. **Nond** could be directly calculated by measuring the size of QUEUE, which stores all discovered nodes within. Then, function searches through entire QUEUE structure to compute the number of rows with "**0**" in their first column (means this point has been visited already).



```
temp = Nond;
while(QUEUE(temp, 1) == 1)
    temp = temp - 1;
end

maze(QUEUE(temp, 2), QUEUE(temp, 3)) = 4;
dispMaze(maze);

while(temp ~= 1)
    Tpc = Tpc + 1;
    xParent = QUEUE(temp, 4);
    yParent = QUEUE(temp, 5);
    for index = 1:Nond
        if(QUEUE(index, 2) == xParent && QUEUE(index, 3) == yParent)
            temp = index;
            maze(QUEUE(temp, 2), QUEUE(temp, 3)) = 5;
            dispMaze(maze);
            break;
        end
    end
end
Tpc = Tpc + 1;
```

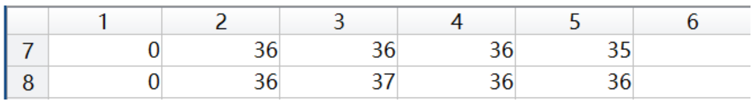
Finally, to traceback the optimal path, function finds **target point** and **starting point** by finding the **last** and **first visited** point. During tracebacking, **Tpc** increases by **1** each time after function finds a parent point to jump to (By using the 4th and 5th column of each row.). And in this part of code, maze is only implemented to *"dispMaze()"* as a parameter.

QUEUE structures' demonstration:



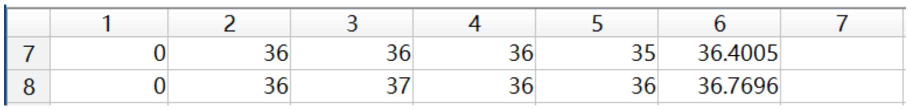
	1	2	3	4	5	6	7	8	9
7	0	36	36	36	35	6	36.4005	42.4005	
8	0	36	37	36	36	7	36.7696	43.7696	

AStar QUEUE**:** [0/1, X val, Y val, Parent X val, Parent Y val, g(n), h(n), f(n)]



	1	2	3	4	5	6
7	0	36	36	36	35	
8	0	36	37	36	36	

DFS QUEUE**:** [0/1, X val, Y val, Parent X val, Parent Y val]



	1	2	3	4	5	6	7
7	0	36	36	36	35	36.4005	
8	0	36	37	36	36	36.7696	

Greedy QUEUE**:** [0/1, X val, Y val, Parent X val, Parent Y val, h(n)]