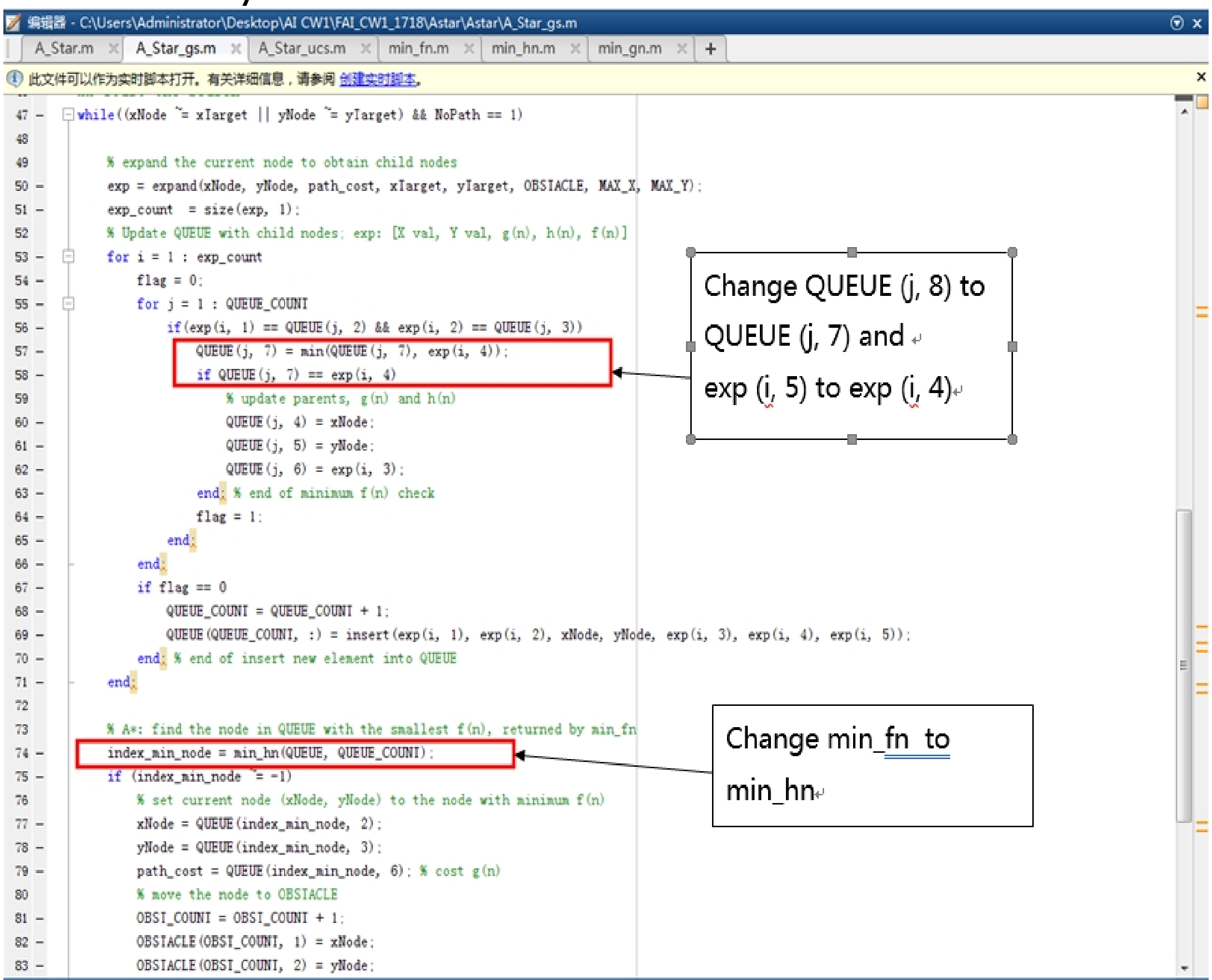
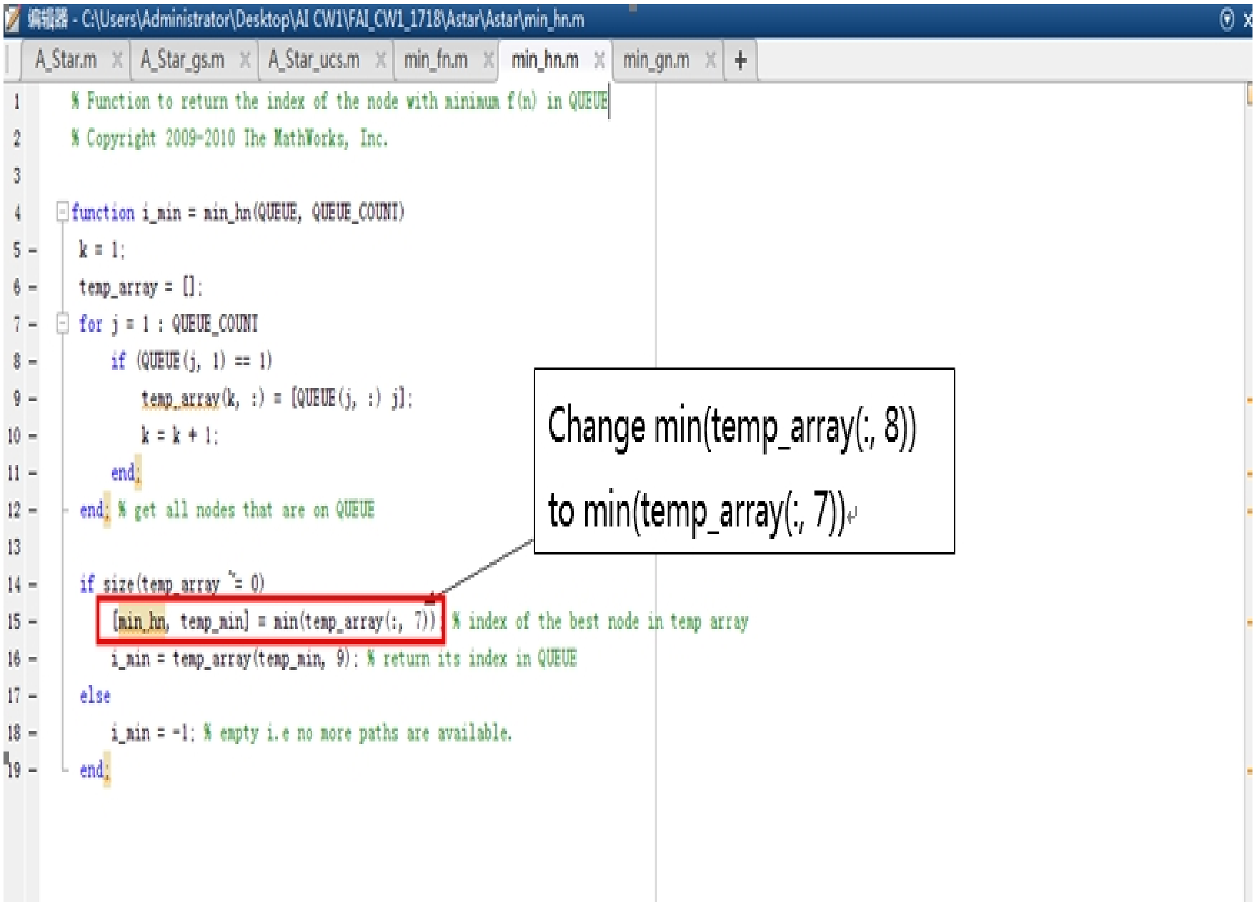
1. *This part is based on the A* matlab demo (A_Star). You need to understand how this demo works and answer the following questions.* ***(6 marks)***

   (a) You are required to implement Greedy Search and Uninform Cost Search algorithms based on the A* code (In the report, show what changes you have made and explain why you make these changes. You can use screenshot to demonstrate your code verification). (2 marks)

（1）Greedy Search:





Explanation: Greedy search's evaluation function is $f(n) = h(n)$. Therefore, the code is changed to compare the h(n) of nodes, then the program will select node with smallest h(n) value.

（2）Uniform Cost Search:

编辑器 - C:\Users\Administrator\Desktop\AI CW1\FAI_CW1_1718\Astar\Astar\A_Star_ucs.m

A_Star.m | A_Star_gs.m | A_Star_ucs.m | min_fn.m | min_hn.m | min_gn.m | +

此文件可以作为实时脚本打开。有关详细信息，请参阅 创建实时脚本。

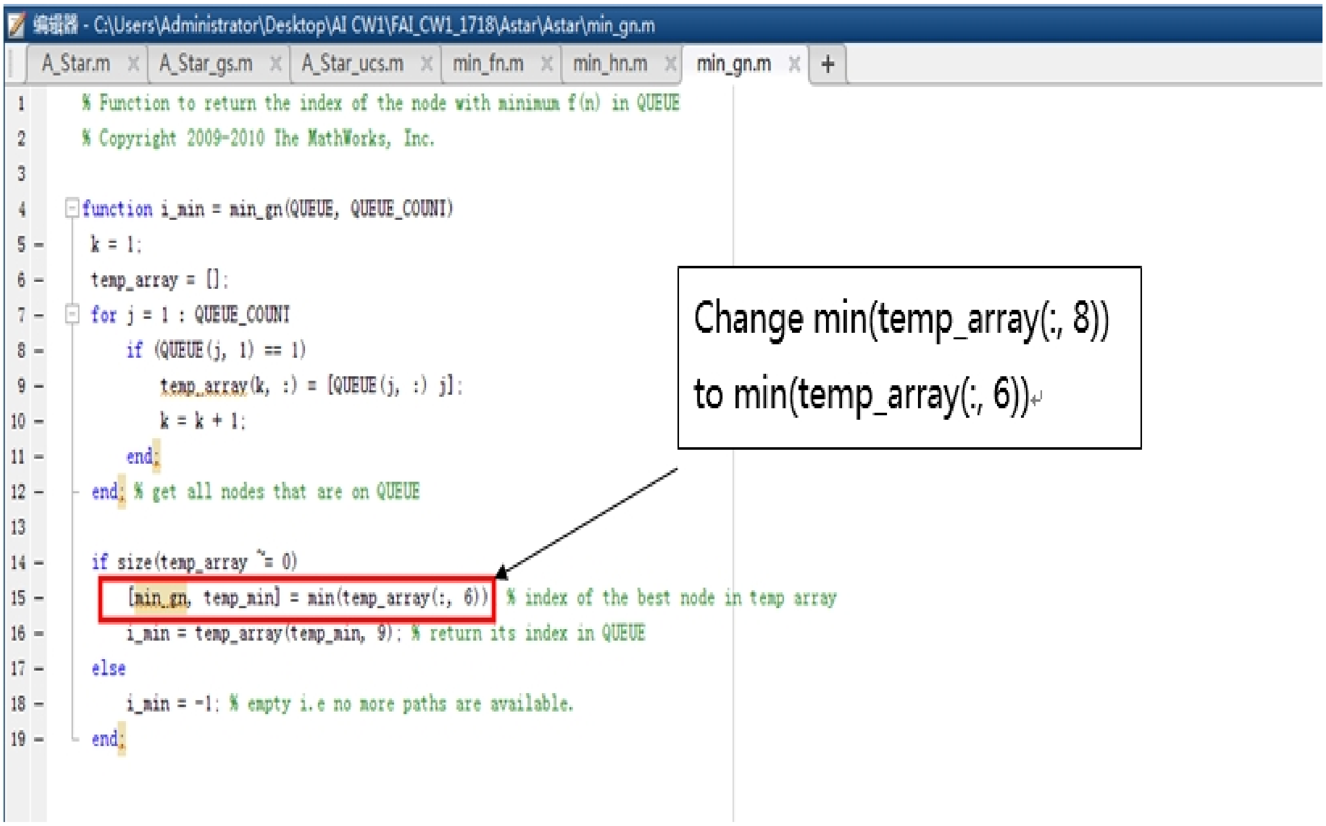
```
%% Start the search
while((xNode ~= xTarget || yNode ~= yTarget) && NoPath == 1)

    % expand the current node to obtain child nodes
    exp = expand(xNode, yNode, path_cost, xTarget, yTarget, OBSTACLE, MAX_X, MAX_Y);
    exp_count  = size(exp, 1);
    % Update QUEUE with child nodes: exp: [X val, Y val, g(n), h(n), f(n)]
    for i = 1 : exp_count
        flag = 0;
        for j = 1 : QUEUE_COUNT
            if(exp(i, 1) == QUEUE(j, 2) && exp(i, 2) == QUEUE(j, 3))
                QUEUE(j, 6) = min(QUEUE(j, 6), exp(i, 3));
                if QUEUE(j, 6) == exp(i, 3)
                    % update parents, g(n) and h(n)
                    QUEUE(j, 4) = xNode;
                    QUEUE(j, 5) = yNode;
                    QUEUE(j, 6) = exp(i, 3);
                    QUEUE(j, 7) = exp(i, 4);
                end; % end of minimum f(n) check
                flag = 1;
            end;
        end;
        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));
        end; % end of insert new element into QUEUE
    end;

    % A*: find the node in QUEUE with the smallest f(n), returned by min_fn
    index_min_node = min_gn(QUEUE, QUEUE_COUNT);
    if (index_min_node ~= -1)
        % set current node (xNode, yNode) to the node with minimum f(n)
        xNode = QUEUE(index_min_node, 2);
        yNode = QUEUE(index_min_node, 3);
        path_cost = QUEUE(index_min_node, 6); % cost g(n)
        % move the node to OBSTACLE
        OBST_COUNT = OBST_COUNT + 1;
```

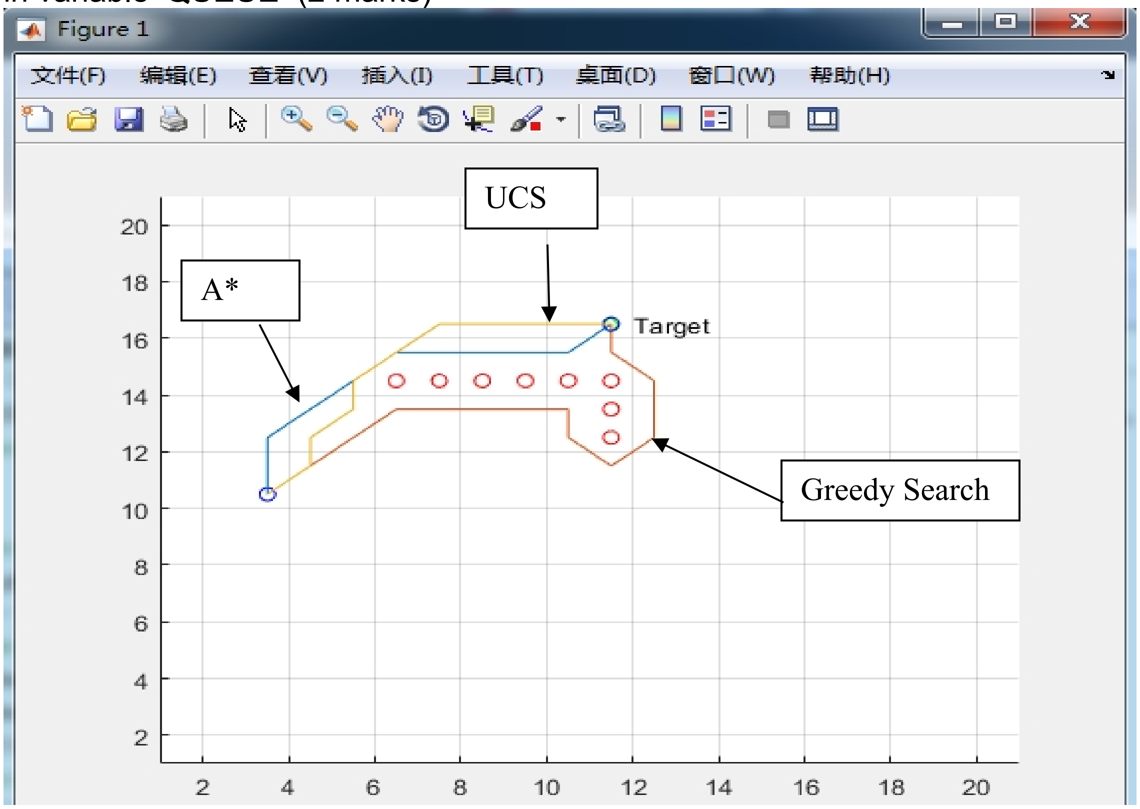
Change QUEUE (j, 8) to QUEUE (j, 6) and exp (i, 5) to exp (i, 3)

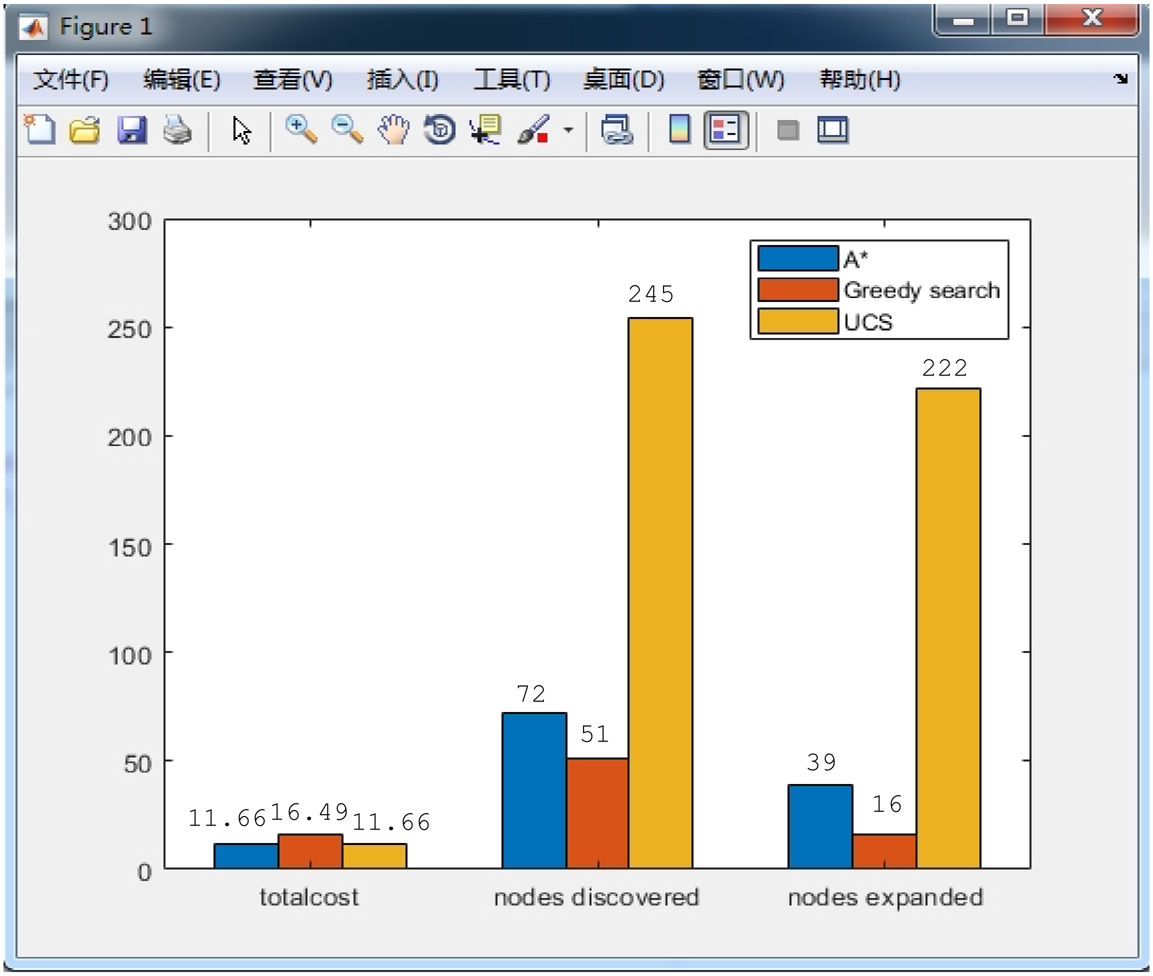
Change min_fn to min_gn



Explanation: UCS works by expanding the lowest cost node on the fringe. Therefore, the code is changed to compare g(n) of each node. In this way the program will compare g(n) of all the nodes expended.

(b) You are required to use the matlab basics from the first lab session to show the evaluation results of the three searching methods (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 actual distance between initial node and target. It is calculated in function expand(), exp_array(exp_count, 3) = gn + distance(node_x, node_y, s_x, s_y). This formula is used to calculate g(n) of every node in a recursive way.

Nodes discovered are nodes stored in the matrix QUEUE. Every time a new node is discovered by function expand(), a new coordinate is added into matrix QUEUE and then QUEUE_COUNT plus one. Therefore, QUEUE_COUNT is the number of nodes discovered.

Nodes expanded are nodes that have smallest f(n) in a path. Every time the function expand() is called, the node with smallest f(n) will be regarded as the parent node of next expand() function. To gain the number of nodes expanded, a node counter is set after function min_fn(). The counter will plus one when a new index_min_node is found.

(c) Design and implement another heuristic h2 which is different from the one (h1) is used in the A* matlab code, explain how h2 works and show what changes you have made to change the heuristic function from h1 to h2. Is h2 optimal? Why? Which heuristic is better? Why? (2 marks)

```
if (flag == 1)
    exp_array(exp_count, 1) = s_x;
    exp_array(exp_count, 2) = s_y;
    exp_array(exp_count, 3) = gn + distance(node_x, node_y, s_x, s_y); % cost g(n)
    if(abs(xTarget - s_x) > abs(yTarget - s_y)) % cost h(n)
         exp_array(exp_count, 4) = abs(xTarget - s_x) + sqrt(2) * abs(yTarget - s_y);
    else
        exp_array(exp_count, 4) = abs(yTarget - s_y) + sqrt(2) * abs(xTarget - s_x);
    end
    exp_array(exp_count, 5) = exp_array(exp_count, 3) + exp_array(exp_count, 4); % f(n)
    exp_count = exp_count + 1;
```

The code in the red box is heuristic h2. It uses a new way to estimate h(n).

This A* algorithm is in a grid, which means there are only eight choices for a node to go to its next step. Therefore, straight-line distance is not the actual cost in most cases. The heuristic h2 will calculate the distance between initial node to a node which is able to link the target in the direction of forty-five degrees, then h2 add up this distance and the straight-line distance between the node and the target as the h(n) of heuristic h2.

Heuristic h2 is optimal and better than heuristic h1. Because Heuristic h2 will never over-estimate the cost to reach the goal and its h(n) is closer to the actual cost than heuristic h1.

2. 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 depth-first approach to randomly generate a maze with user defined size and difficulty. ***(9 marks)***

   (a) In the demo code, show which line(s) of code is used to implement depth-first approach, explain the logic the student adopts to generate the maze. (2 marks)

```
% Check if that route can continue or not
if any(directions) == 0
    if same(position, nodes) == 1
        % Remove last node because all positions are exhausted
        nodes = nodes(:, 1 : end - 1);
    else
        position = point(nodes(1, end), nodes(2, end));
    end
```

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

```
%% CALCULATIONS ---
% 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
```

These lines of node are used to implement depth-first approach. This section of code judge that if the previous position is not at the same row and column with future position, the program will create a node. The program explore one branch until there is no way to expand before exploring another branch. Then the program will return to the nearest node to continue to generate path until there are no more nodes left.

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

First, the parameter cmap is corresponding to number 0 to 7 in dispMaze(), however the boundary nodes are represented by number 8. It causes a little confusion.
Second, a maze is randomly generated using depth-first approach only, which means the maze generator will explore one branch until there is no way to expand before exploring another branch. In this case, the maze will just have some shallow branch and it’s easy to find the maze path.

(c) Write a maze solver using A* algorithm. (6 marks)

i) The solver need be called by command ‘**AStarMazeSolver(maze)**’
ii) The maze solver should be able to solve any maze generated by the maze generator
iii) The maze solver should be able to find the optimal solution.
iv) Your code need display the all the routes that A* has processed with RED color.
v) Your maze solver should be about to display the final result BLACK color.

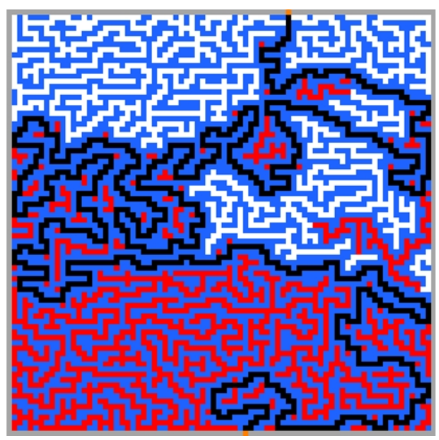


Figure 1. Sample output