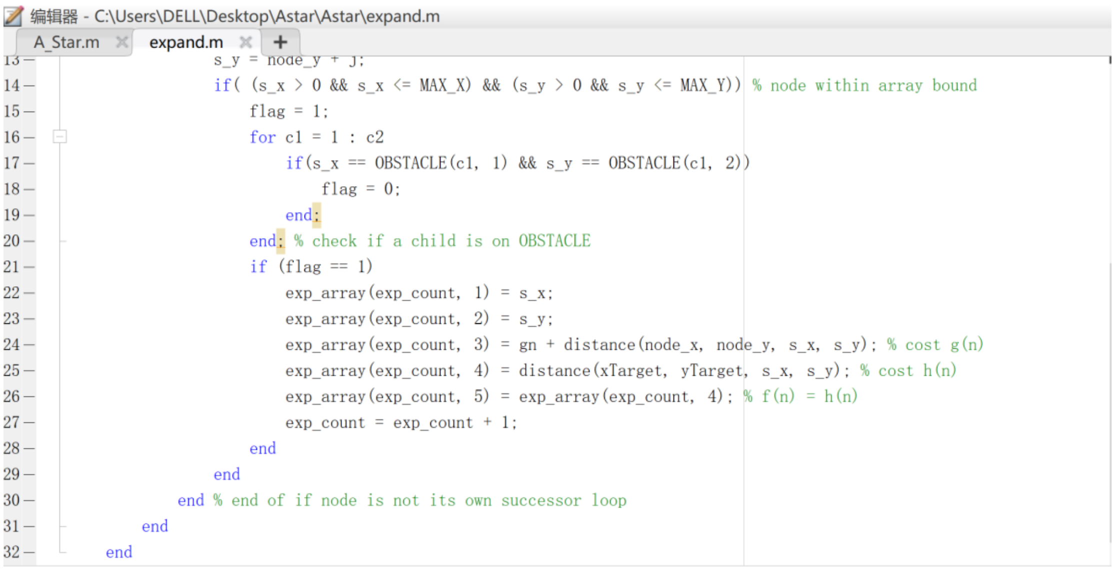
Q1

a

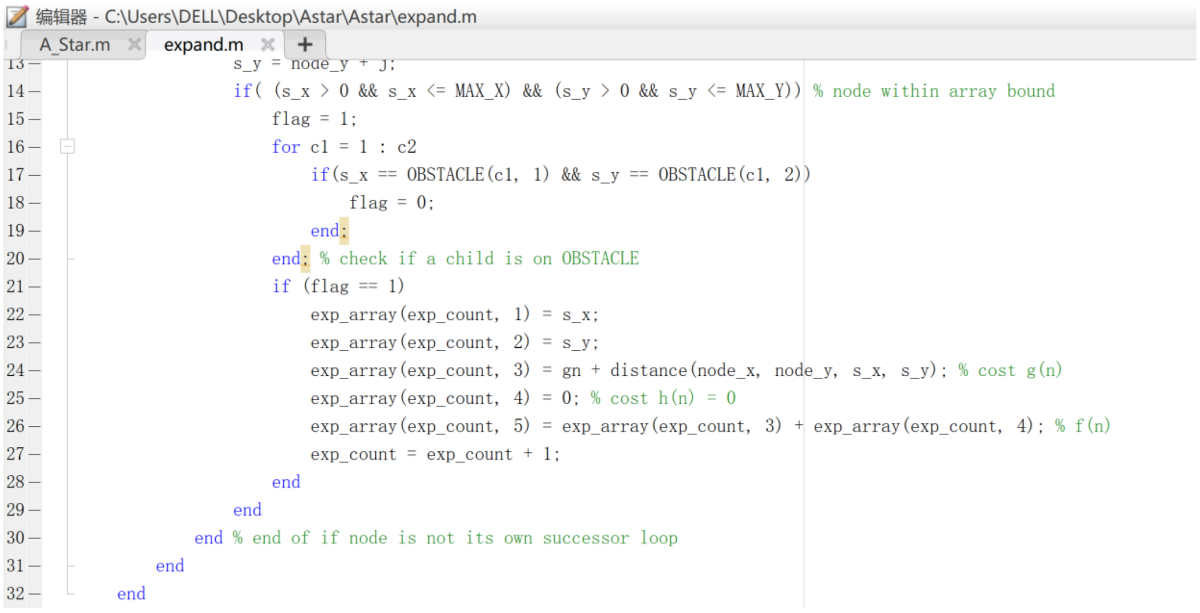
1. For Greedy Search, the value of f(n) equal to h(n), so we can let

   exp_array(exp_count, 5) = exp_array(exp_count, 4); to make sure f(n)=h(n).



```
                s_y = node_y + j;
                if( (s_x > 0 && s_x <= MAX_X) && (s_y > 0 && s_y <= MAX_Y)) % node within array bound
                    flag = 1;
                    for c1 = 1 : c2
                        if(s_x == OBSTACLE(c1, 1) && s_y == OBSTACLE(c1, 2))
                            flag = 0;
                        end;
                    end; % check if a child is on OBSTACLE
                    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)
                        exp_array(exp_count, 4) = distance(xTarget, yTarget, s_x, s_y); % cost h(n)
                        exp_array(exp_count, 5) = exp_array(exp_count, 4); % f(n) = h(n)
                        exp_count = exp_count + 1;
                    end
                end
            end % end of if node is not its own successor loop
        end
    end
```

2. For UCS, the value of f(n) equal to g(n) and we can let h(n) equal to zero to implement UCS(Let exp_array(exp_count, 4)=0;).



```
                s_y = node_y + j;
                if( (s_x > 0 && s_x <= MAX_X) && (s_y > 0 && s_y <= MAX_Y)) % node within array bound
                    flag = 1;
                    for c1 = 1 : c2
                        if(s_x == OBSTACLE(c1, 1) && s_y == OBSTACLE(c1, 2))
                            flag = 0;
                        end;
                    end; % check if a child is on OBSTACLE
                    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)
                        exp_array(exp_count, 4) = 0; % cost h(n) = 0
                        exp_array(exp_count, 5) = exp_array(exp_count, 3) + exp_array(exp_count, 4); % f(n)
                        exp_count = exp_count + 1;
                    end
                end
            end % end of if node is not its own successor loop
        end
    end
```

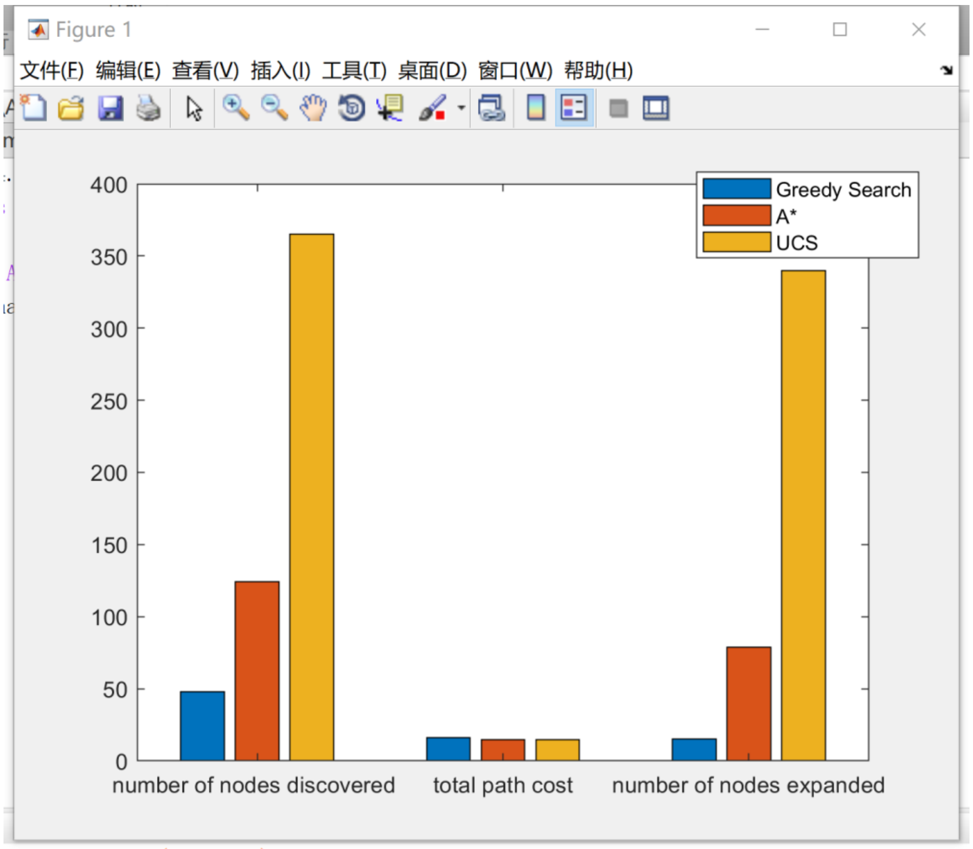
Also

The value of h(n) of first node also need to be set to zero (goal_distance = 0).

```
%% add the starting node as the first node (root node) in QUEUE
% QUEUE: [0/1, X val, Y val, Parent X val, Parent Y val, g(n),h(n), f(n)]
xNode = xStart;
yNode = yStart;
QUEUE = [];
QUEUE_COUNT = 1;
NoPath = 1; % assume there exists a path
path_cost = 0; % cost g(n): start node to the current node n
goal_distance = 0; % cost h(n): heuristic cost of n
QUEUE(QUEUE_COUNT, :) = insert(xNode, yNode, xNode, yNode, path_cost, goal_distance, goal_distance);
QUEUE(QUEUE_COUNT, 1) = 0; % What does this do?
```

b

	Greedy Search	A *	UCS
Total path cost	16.3137	14.8995	14.8995
Number of nodes discovered	48	124	365
Number of nodes expanded	15	79	340



In QUEUE, QUEUE(QUEUE_COUNT, 6) is the total path cost and the number of nodes discovered equal to the number of set in QUEUE. For the expand node, once it is expended, its flag equal to 0,and we can find the number by count the number of set which flag is 0.

c

h2= sqrt((x1-x2)^2)+sqrt((y1-y2)^2); (Lateral distance plus longitudinal distance)

h2 is optimal it can find the minimum path with only four directions.

In my opinion, h1 is better. The reason is that although h2 can also find the optimal path, it can only look for from four directions and discover more nodes. Instead, h2 can look for from eight directions and in this way, the path may smaller than h2 with less nodes discovered.

Q2

a.

The lines of code that are used to implement depth-first approach mainly in file move.m

```
if any(directions) == 0
    if same(position, nodes) == 1
        nodes = nodes(:, 1 : end - 1);
    else
        position = point(nodes(1, end), nodes(2, end));
    end
else
```

and

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

b.

c.

The logic the student adopts to generate the maze.is as follows:

Firstly, the program will get the size and difficulty of the maze， after checking the value is valid, this program will generate a zero matrix of size*size and set the value of the boundary to eight. Then, the starting and ending points will be generated randomly at the boundary. The value of starting point is 3 and the ending point is 4. After these steps, the point above the starting point will add into the node and points around this node will be detected if they can join and will be randomly added, then the maze will be printed. In the maze, the nodes are blue, the boundary is gray, the starting and ending points is orange and others are white, this step will be repeated many times until there is no node to get. Finally, the points below the ending point will be detected whether connect the blue part, if not, the program will connect this point with the blue part and then print it.