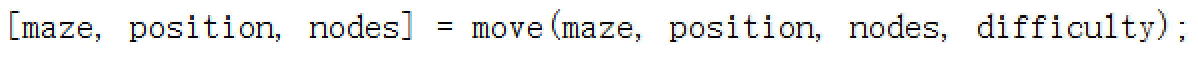
Coursework Report

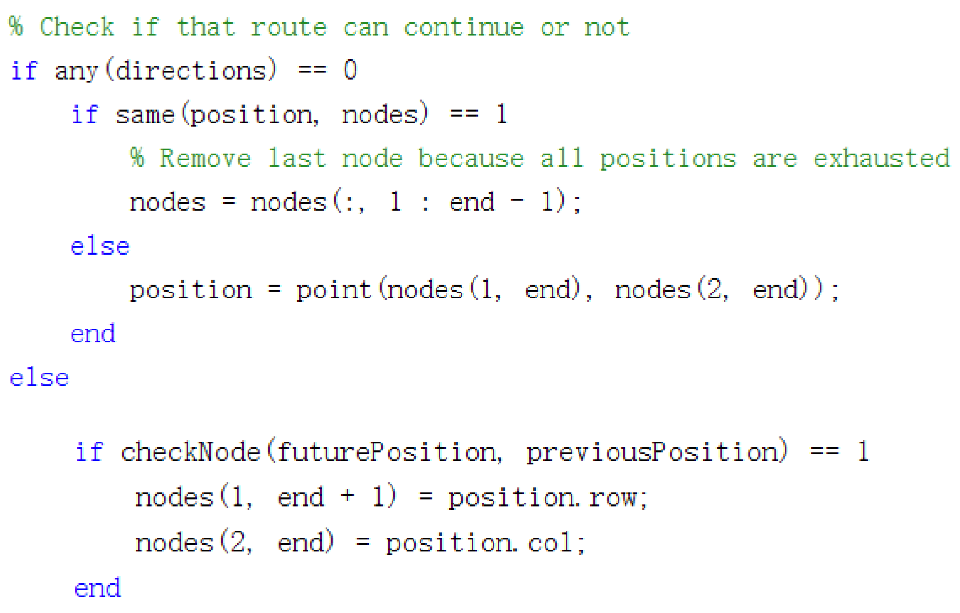
Chen Liao **20030694**

• About tree search approach implementation

The following lines of code in ‘main.m’ and ‘move.m’ implements the tree approach in generating the maze.



```
[maze, position, nodes] = move(maze, position, nodes, difficulty);
```



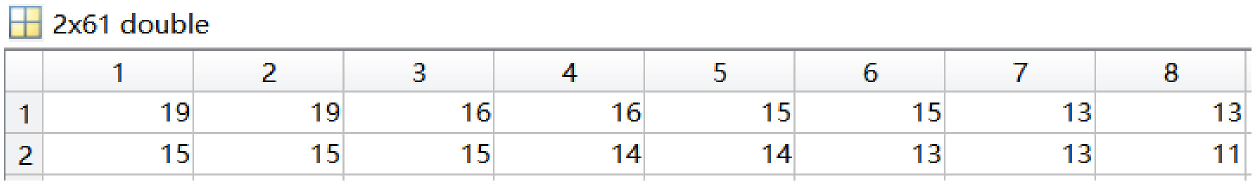
```
% Check if that route can continue or not
if any(directions) == 0
    if same(position, nodes) == 1
        % Remove last node because all positions are exhausted
        nodes = nodes(:, 1 : end - 1);
    else
        position = point(nodes(1, end), nodes(2, end));
    end
else

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

The “move” function implements one step of tree construction at a time which is called recursively to generate the whole maze. The algorithm firstly check if there is any direction to go, if there is not, the last node would be removed from the queue. Then, if the way is valid, the current point would be appended to the queue for further expansion.

Inner the ‘move’ function, the ‘validateMove’ function is firstly called to expand the corresponding four positions around the current position and randomly choose one according to the difficulty then set it to ‘futurePosition’. If there is possible direction to go, the ‘currentPoint’ is set equal to the ‘futurePoint’ and that point in the maze is set to 1, if there is no possible direction to go, the algorithm will trace back to the nearest node that it recently visited by deleting the node with no path in the tail of the node list.

The basic data structure of implementing the tree is the matrix called ‘nodes’. This matrix, with a size of 2 * n, is a queue used for storing the position (rol, col) of the point in the maze as a node. However, only the point of inflexion will be recorded in this matrix, which is identified by the following codes.


2x61 double

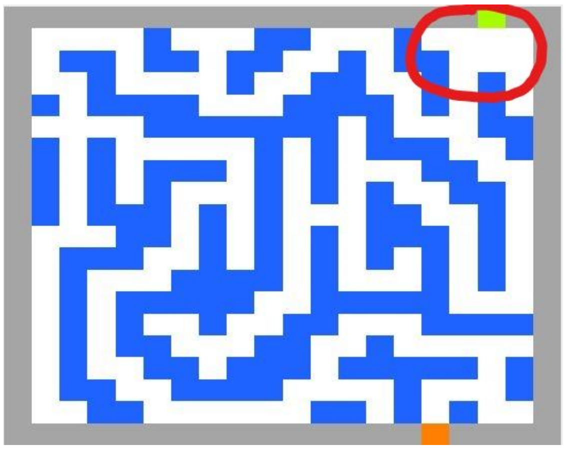
	1	2	3	4	5	6	7	8
1	19	19	16	16	15	15	13	13
2	15	15	15	14	14	13	13	11

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

The basic logic of this tree implementation is very identical to "Depth First Search", meaning that the algorithm will expand the node which is lately added at the tail if there is still a way to go. As long as the position explored is valid, this point in the maze will be set to 1, indicating this is a point of path. However, if there is no valid direction to go, the algorithm will head back to the previous inflexion point (node stored) to see if there is other possible way from here.

- **Logic problem**

From my perspective, the logic problem that I found in this maze generator is in the strategy of connecting the destination to the pathway. In the logic of the generator, the exit point of the maze is randomly generated but always in the first row. Therefore, when linking it with the rest of the maze, the generator just simply keep exploring downwards until the path began with the end point is connected to the rest of the maze. This strategy is feasible but there would be possibly a maze generated as the following.

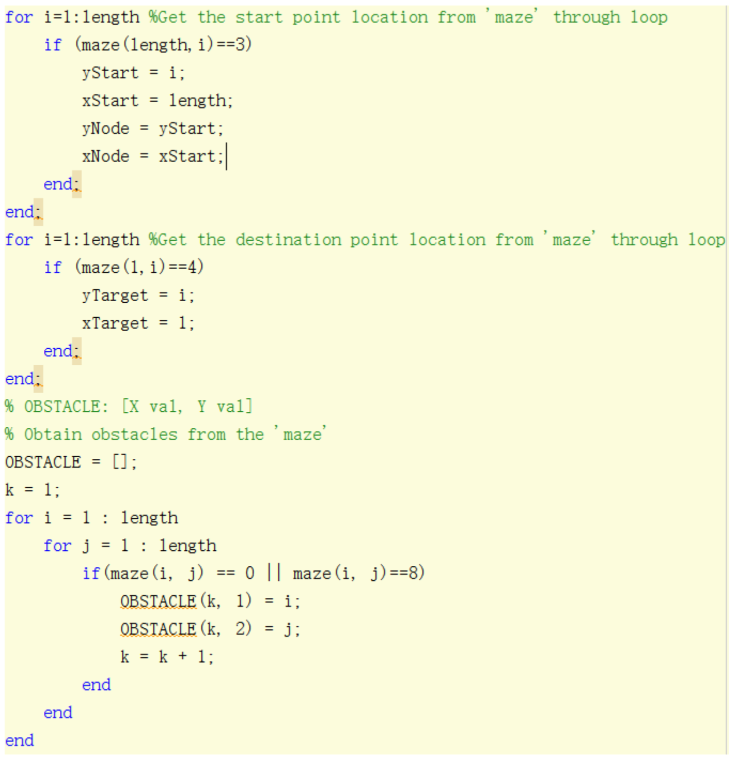


In the strict sense, every individual branch of the pathway should have a width of 1.

However, because of the strategy above, the maze generator generates a 2*3 space.

- **A* algorithm**

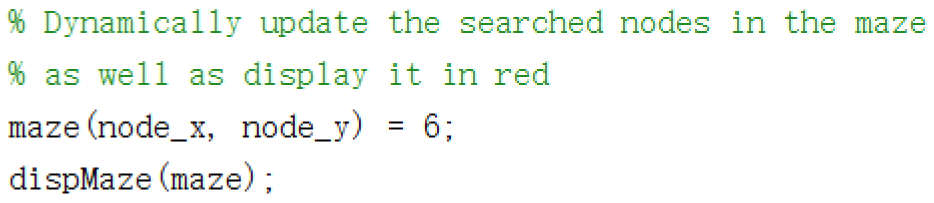
The first change I made to implement the ‘AstarMazeSolver’ is the strategy of problem fetching.



```
for i=1:length %Get the start point location from 'maze' through loop
    if (maze(length,i)==3)
        yStart = i;
        xStart = length;
        yNode = yStart;
        xNode = xStart;
    end;
end;
for i=1:length %Get the destination point location from 'maze' through loop
    if (maze(1,i)==4)
        yTarget = i;
        xTarget = 1;
    end;
end;
% OBSTACLE: [X val, Y val]
% Obtain obstacles from the 'maze'
OBSTACLE = [];
k = 1;
for i = 1 : length
    for j = 1 : length
        if(maze(i, j) == 0 || maze(i, j)==8)
            OBSTACLE(k, 1) = i;
            OBSTACLE(k, 2) = j;
            k = k + 1;
        end
    end
end
```

The input of the problem is a maze, which is actually a matrix. Therefore, I traversed the whole maze from the saved ‘maze.mat’ to obtain essential data of start point, destination, obstacles and boundaries.

This change is made in the function ‘expand’ because this function is recursively called. By changing the number in maze one at a time, the algorithm will dynamically draw all the nodes that it has visited one by one in to red.



```
% Dynamically update the searched nodes in the maze
% as well as display it in red
maze(node_x, node_y) = 6;
dispMaze(maze);
```

This change is made in the ‘result’ function which was invoked after the whole search. In this function, the optimal path will be generated by tracing from the destination to the start point (the matrix ‘optimal_path()’ is a data structure to store the nodes in the selected path). These two lines of code were added in an iteration, that is to say, when the optimal queue is updated, the display is updated too. In this way we can dynamically draw the optimal path into black after the search.

```
% Dynamically update the optimal path as well as display the maze
maze(parent_x, parent_y)=5;
dispMaze(maze);
```

- **Greedy Search Algorithm**

The only difference between A* algorithm and greedy search algorithm is the estimation criterion. The sum of h(n) and g(n) is considered in the A* but only h(n) in Greedy. Therefore, I only change the strategy in choosing the next node in 'min_fn()'

```
%changed from 8 to 7 to get the minist of h(n)
[min_fn, temp_min] = min(temp_array(:, 7)); % index of the best node in temp array
i_min = temp_array(temp_min, 9); % return its index in QUEUE
```

- **DFS Search Algorithm**

For the Depth First Searching, the search process should head back if there is no possible way to go.

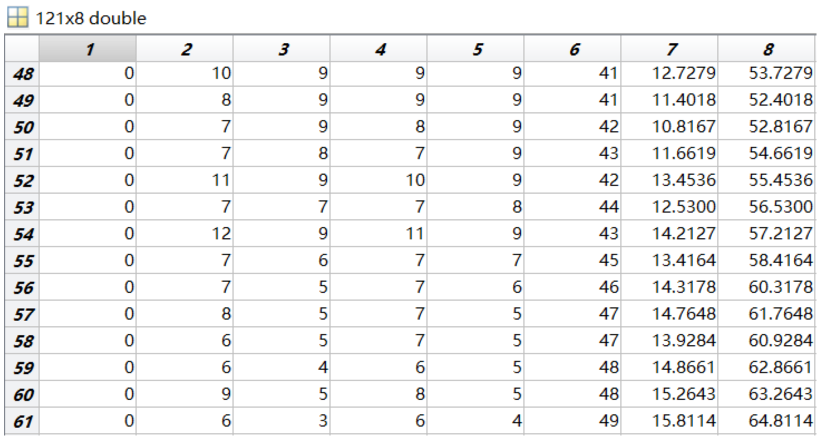
```
% If no possible direction to go at the current point
% then trace back
if (exp_count == 0)
    P = point(xNode, yNode);
    maze = setMazePosition(maze, P, 6);
    for i=1:QUEUE_COUNT
        if (QUEUE(i, 2)==xNode && QUEUE(i, 3)==yNode)
            xNode = QUEUE(i, 4);
            yNode = QUEUE(i, 5);
            % adjust the current path cost
            path_cost = QUEUE(i, 6) - 1;
            % move the node to OBSTACLE
            OBST_COUNT = OBST_COUNT + 1;
            OBSTACLE(OBST_COUNT, 1) = xNode;
            OBSTACLE(OBST_COUNT, 2) = yNode;
            % QUEUE(index_min_node, 1) = 0;
            break;
        end
    end
```

Then, if there is a possible way to go, a direction depends on current queue order would be selected for the solver to go further until it has no direction to go or it finds the destination.

- **Extracting information from "QUEUE"**

The 'QUEUE' is a matrix of n*8 to store information of the points (nodes) in maze, it is the central data structure for implementing the A* Search Algorithm.

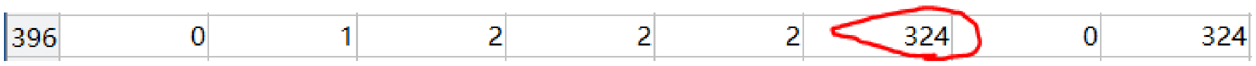
```
QUEUE(QUEUE_COUNT, :) = insert(xNode, yNode, xNode, yNode, path_cost, goal_distance, goal_distance);
```



121x8 double

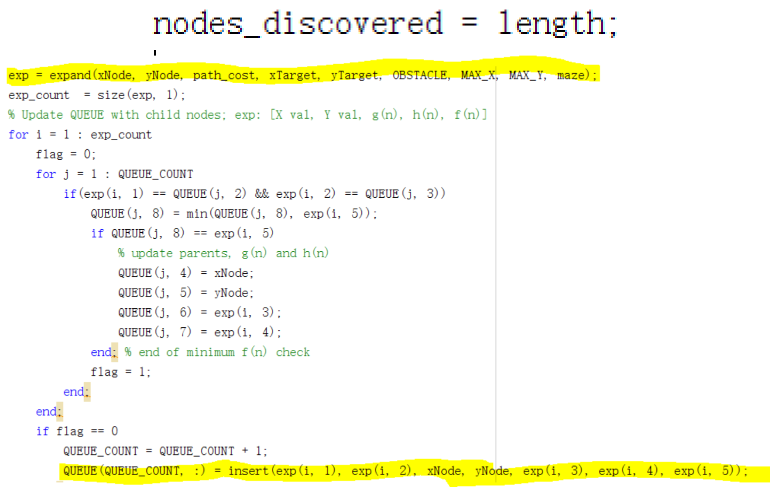
	1	2	3	4	5	6	7	8
48	0	10	9	9	9	41	12.7279	53.7279
49	0	8	9	9	9	41	11.4018	52.4018
50	0	7	9	8	9	42	10.8167	52.8167
51	0	7	8	7	9	43	11.6619	54.6619
52	0	11	9	10	9	42	13.4536	55.4536
53	0	7	7	7	8	44	12.5300	56.5300
54	0	12	9	11	9	43	14.2127	57.2127
55	0	7	6	7	7	45	13.4164	58.4164
56	0	7	5	7	6	46	14.3178	60.3178
57	0	8	5	7	5	47	14.7648	61.7648
58	0	6	5	7	5	47	13.9284	60.9284
59	0	6	4	6	5	48	14.8661	62.8661
60	0	9	5	8	5	48	15.2643	63.2643
61	0	6	3	6	4	49	15.8114	64.8114

As we could see, the current path cost of a certain node is stored in the 6th column of this matrix. Therefore, to derive the total path cost from the 'QUEUE', what we need to acquire is the value in the 6th column of the last row (reaching the destination).



396	0	1	2	2	2	324	0	324

To derive the total number of nodes discovered, we need to notice that all nodes expanded have to be discovered, but not all nodes discovered are expanded. Additionally, all discovered nodes are added into 'QUEUE' because all children nodes of the current expanding node would be added into 'QUEUE' whether they were chosen to be expanded next or not. That is to say, the total number of rows in this data structure indicates the number of nodes discovered. Therefore, we could use function 'size()' to get the number of nodes discovered.

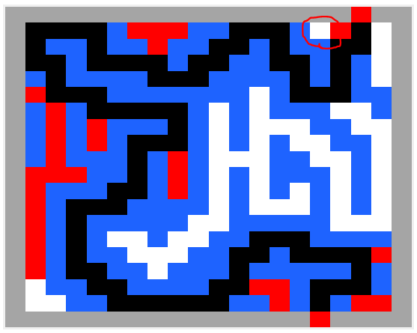


```
nodes_discovered = length;
```

```
exp = expand(xNode, yNode, path_cost, xTarget, yTarget, OBSTACLE, MAX_X, MAX_Y, maze);
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, 8) = min(QUEUE(j, 8), exp(i, 5));
            if QUEUE(j, 8) == exp(i, 5)
                % 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));
```

(The logic of nodes expanded and nodes inserted, highlighted)

To derive the number of nodes discovered, we need to know that all nodes discovered are expanded, but not all nodes discovered are expanded (see screen shot below).



The point circled out is a typical representation of the nodes discovered but not expanded (because the expanded nodes would be plotted into red). Therefore, to acquire the number of nodes discovered we need to find out how many points like this are there in the maze and the subtract it from the magnitude of 'QUEUE_COUNT'.

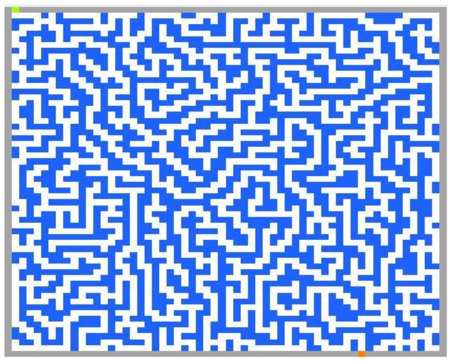
Since within every loop the point with the minimum f(n) would be set to be the current point and be ready for expansion, correspondingly, the index in the first row of the current point would be set to 0 from 1. Therefore, if there are nodes with the index of 1 in 'QUEUE', they are discovered but not expanded.

```
xNode = QUEUE(index_min_node, 2);
yNode = QUEUE(index_min_node, 3);
```

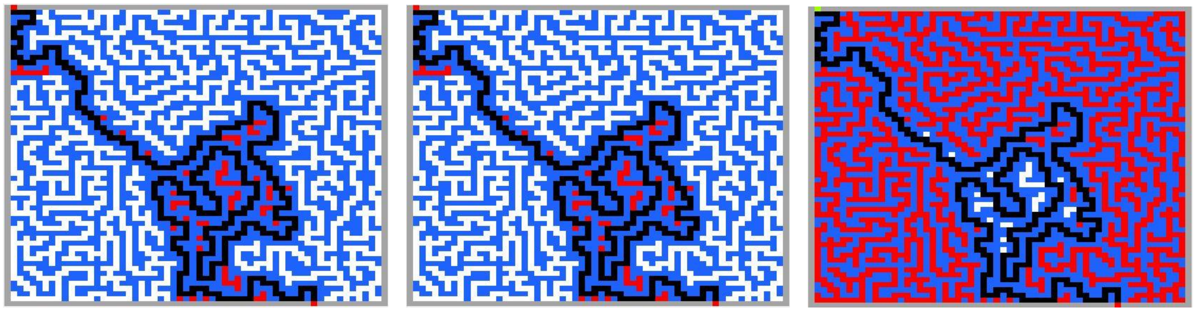
```
index_min_node = min_fn(QUEUE, QUEUE_COUNT);
QUEUE(index_min_node, 1) = 0;
```

nodesDiscovered = nodesExpanded + nodesWithIndex_1

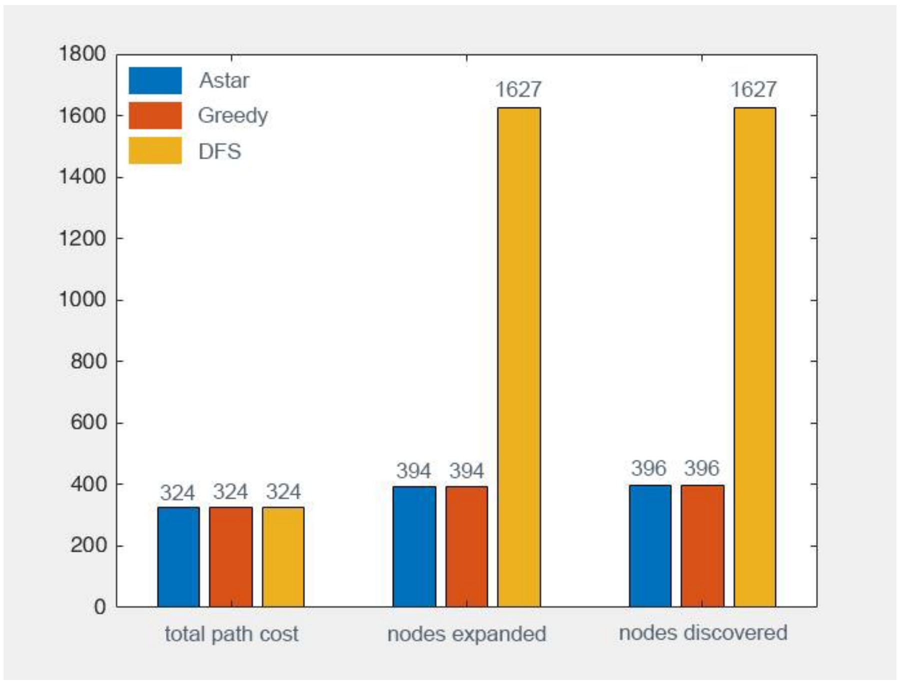
To test the algorithms as well as to get statistics, a maze of 60*60 with the difficulty of 4 is generated.



Original maze

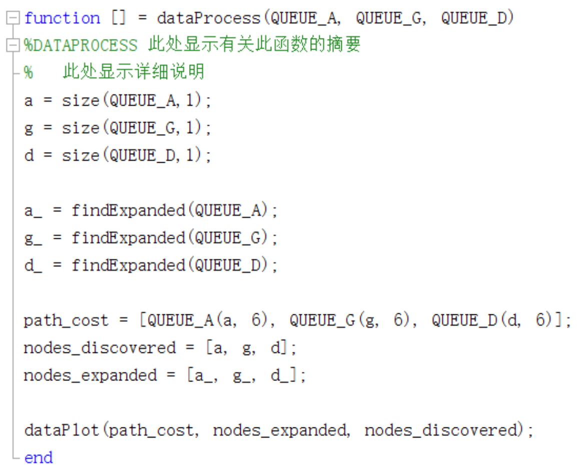


Then the data mentioned above is derived from the data structure 'QUEUE'



Generated by matlab (bar)

The code for deriving data from data structure 'QUEUE':

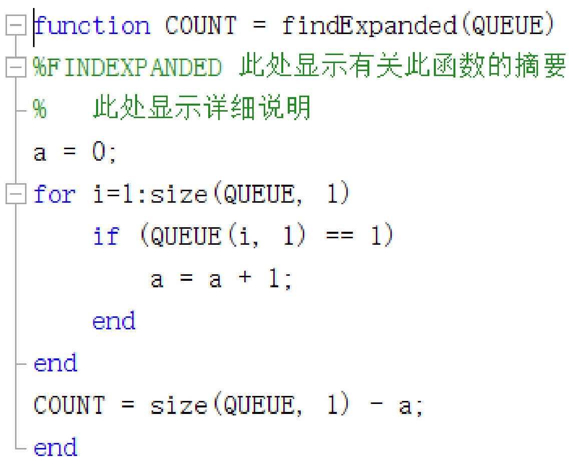


```
function [] = dataProcess(QUEUE_A, QUEUE_G, QUEUE_D)
%DATAPROCESS 此处显示有关此函数的摘要
%   此处显示详细说明
a = size(QUEUE_A,1);
g = size(QUEUE_G,1);
d = size(QUEUE_D,1);

a_ = findExpanded(QUEUE_A);
g_ = findExpanded(QUEUE_G);
d_ = findExpanded(QUEUE_D);

path_cost = [QUEUE_A(a, 6), QUEUE_G(g, 6), QUEUE_D(d, 6)];
nodes_discovered = [a, g, d];
nodes_expanded = [a_, g_, d_];

dataPlot(path_cost, nodes_expanded, nodes_discovered);
end
```



```
function COUNT = findExpanded(QUEUE)
%FINDEXPANDED 此处显示有关此函数的摘要
%   此处显示详细说明
a = 0;
for i=1:size(QUEUE, 1)
    if (QUEUE(i, 1) == 1)
        a = a + 1;
    end
end
COUNT = size(QUEUE, 1) - a;
end
```