Machine Learning Engineer Nanodegree

Capstone Project

Shashank P
December 2nd, 2017

Plot and Navigate a Virtual Maze

I. Definition

Project Overview

This project takes inspiration from Micromouse competitions, wherein a robot mouse is tasked with plotting a path from a corner of the maze to its center. The mice are completely autonomous robots that must find their way from a predetermined starting position to the central area of the maze unaided. The mouse will need to keep track of where it is, discover walls as it explores, map out the maze and detect when it has reached the goal. Having reached the goal, the mouse will typically perform additional searches of the maze until it has found an optimal route from the start to the center. Once the optimal route has been found, the mouse will run that route in the shortest possible time.

In this project, I will create functions to control a virtual robot to navigate a virtual maze. A simplified model of the world is provided along with specifications for the maze and robot; my goal is to obtain the fastest times possible in a series of test mazes.

Problem Statement

On each maze, the robot must complete two runs. In the first run, the robot is allowed to freely roam the maze to build a map of the maze. It must enter the goal room at some point during its exploration, but is free to continue exploring the maze after finding the goal. After entering the goal room, the robot may choose to end its exploration at any time. The robot is then moved back to the starting position and orientation for its second run. Its objective now is to go from the start position to the goal room in the fastest time possible. The robot's score for the maze is equal to the number of time steps required to execute the second run, plus one thirtieth the number of time steps required to execute the first run. A maximum of one thousand time steps are allotted to complete both runs for a single maze.

The aim of this project is to balance the tradeoff between time taken for exploration or learning and time taken to reach the goal in the shortest time with the paths found during the exploration phase. The robot's score is a quantitative measure of this tradeoff. To make sure that the robot reaches the goal atleast once during the exploration phase, a good approach is to make the robot reach the goal as soon as possible.

Metrics

The evaluation metric for this project is the robot's score which is

Robot's score = number of time steps required to execute the second run + (number of time steps required to execute the first run)/30

Robot's score is a tradeoff between the time taken for exploring and time taken to traverse the optimum path found. During the first run, the robot explores the maze and the number of steps taken during this run is the time steps taken for exploration. During the second run, the robot reaches the goal using the optimum policy found. The time steps taken by this optimum policy plus one thirtieth the number of time steps required to execute the first run makes the robot's score. Lesser the robot's score(on an average), better is its performance.

II. Analysis

Data Exploration and Visualization

The robot has three obstacle sensors, mounted in the front of the robot, its left side and its right side. Each of these sensors detect the number of open squares in the direction of the sensor. For example, in its starting position, the robot's left and right sensors detect that there are no open squares in these directions and the front sensor will detect atleast one open square. On each time step of the simulation, the robot may choose to rotate clockwise or counterclockwise by ninety degrees or to not rotate at all. It can also choose to move either forwards or backwards a distance of upto three units. It is assumed that robot's rotation and movement is perfect. If the robot tries to move into a wall, the robot stays where it is. After movement, one time step has passed, and the sensors return readings for the open squares in the robot's new location and/or orientation to start the next time unit.

Rotation is expected to be an integer taking one of three values: -90, 90, or 0, indicating a counterclockwise, clockwise, or no rotation, respectively. Movement follows rotation, and is expected to be an integer in the range [-3, 3] inclusive.

Analysis of Test Maze 3

1) Loops

A path with one or more loops or cycles cannot be the shortest path. This is because a shorter path can be constructed by remove all the loops in that path.

A major challenge is to identify loops in the maze and prevent the robot from traversing these loops as much as possible. This will not only help in finding the shortest path, but will also make better use of time in exploration phase by preventing wastage of time in exploring loops again and again.
Few loops in the maze are shown in fig. 1.

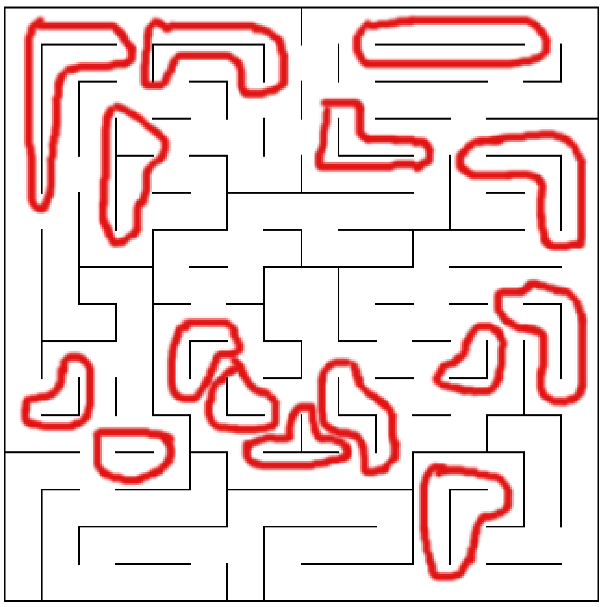


fig. 1

2) Dead-ends and dead-paths

Another major challenge is to prevent the robot from going into dead-ends in the maze. A dead-end is a cell in the maze where the robot is surrounded by walls in three directions. The start location is an exception, lets not call it a dead-end. A robot in a dead-end has no choice but to return back to the cell from where it arrived.

Dead-paths are paths with cells where the robot has two choices only – i) traverse a path that lead to a dead-end, ii) return back from where it arrived. Preventing the robot from traversing these dead-paths more than once reduces wastage of time in exploration phase. And removing dead-paths from a path certainly makes the path shorter.
Dead-paths in test maze 3 are shown in fig. 2

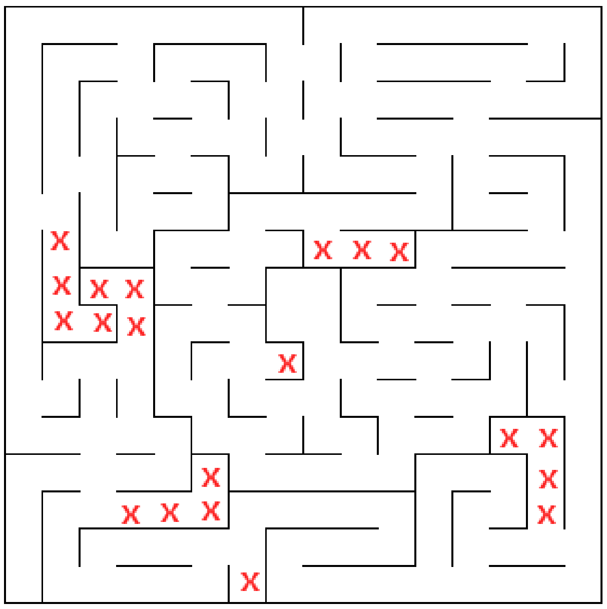


fig. 2

3) Shortest path

The path shown in fig.3 is one of the shortest paths in the maze. It has a path length of 49.
The objective is to not only find the path with shortest length but to also find the path with minimum number of time steps or moves. Number of moves required for this path is 25. This is one of the paths with shortest number of moves.

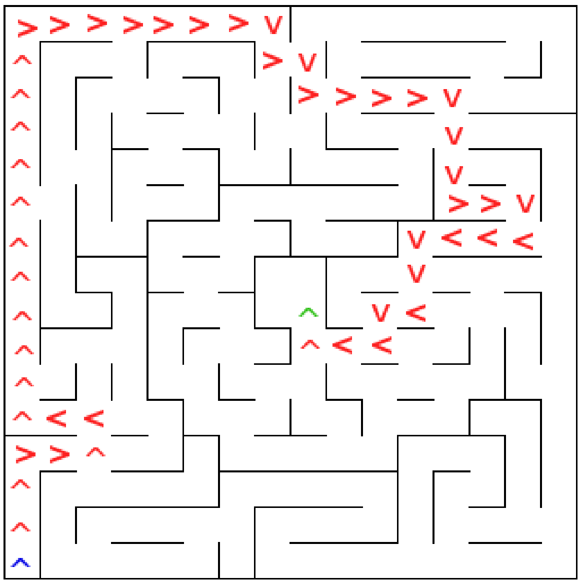
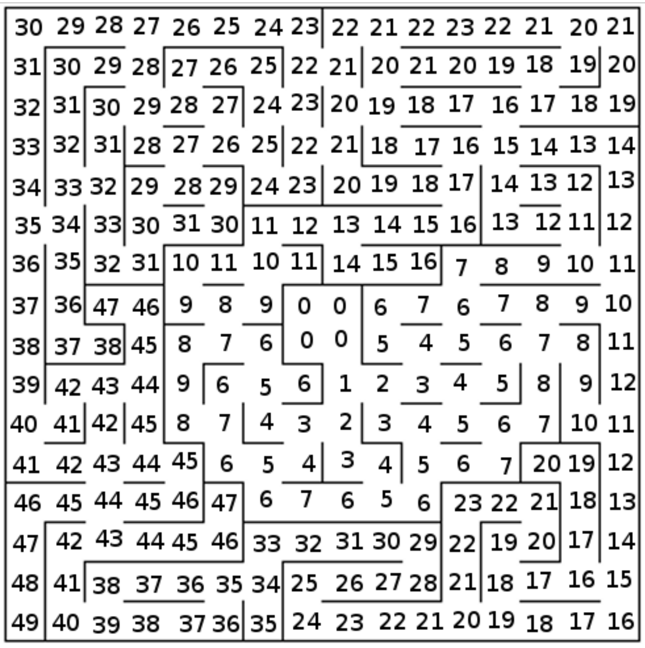


fig. 3

Algorithms and Techniques

Dynamic Programming

The algorithm that I have used is dynamic programming. Dynamic Programming is used to find the shortest path to the destination from any point in the maze. In dynamic programming, the distance from any point on the maze to the destination, called the ‘distance value’, is calculated and stored as the robot explores the maze. The ‘distance value’ of test maze 3 when the robot has visited all cells is shown in fig. 4. The ‘distance value’ of all cells in the maze is updated on each move of the robot. From the figure, we can see that the start location has a distance value of 49, which means that length of the shortest path from the start to the goal is 49.



30	29	28	27	26	25	24	23	22	21	22	23	22	21	20	21
31	30	29	28	27	26	25	22	21	20	21	20	19	18	19	20
32	31	30	29	28	27	24	23	20	19	18	17	16	17	18	19
33	32	31	28	27	26	25	22	21	18	17	16	15	14	13	14
34	33	32	29	28	29	24	23	20	19	18	17	14	13	12	13
35	34	33	30	31	30	11	12	13	14	15	16	13	12	11	12
36	35	32	31	10	11	10	11	14	15	16	7	8	9	10	11
37	36	47	46	9	8	9	0	0	6	7	6	7	8	9	10
38	37	38	45	8	7	6	0	0	5	4	5	6	7	8	11
39	42	43	44	9	6	5	6	1	2	3	4	5	8	9	12
40	41	42	45	8	7	4	3	2	3	4	5	6	7	10	11
41	42	43	44	45	6	5	4	3	4	5	6	7	20	19	12
46	45	44	45	46	47	6	7	6	5	6	23	22	21	18	13
47	42	43	44	45	46	33	32	31	30	29	22	19	20	17	14
48	41	38	37	36	35	34	25	26	27	28	21	18	17	16	15
49	40	39	38	37	36	35	24	23	22	21	20	19	18	17	16

fig. 4

Techniques used:

During exploration phase:
1) Traverse the path of decreasing 'distance value'
By traversing the path of decreasing 'distance value', the robot is made to reach the goal(destination) as soon as possible. If there is more than one choice for next move, unvisited cells are given higher priority for the next move.
2) After reaching the goal, go back to start.
When robot reaches the goal, make the start location as the destination and goal as the starting point. Change the 'distance values' start location has a 'distance value' of 0 and 'distance values' of other locations are updated accordingly. Using technique 1, the robot now moves to the start location as soon as possible. The shortest path length from start to goal is the same as the shortest path length from goal to start. So this technique makes the best use of time during exploration phase to find the shortest path.

Advantages of Dynamic Programming

1) Prevents going into loops

For example,

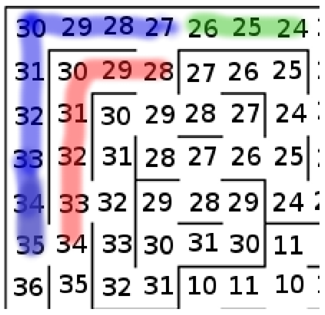


fig. 5

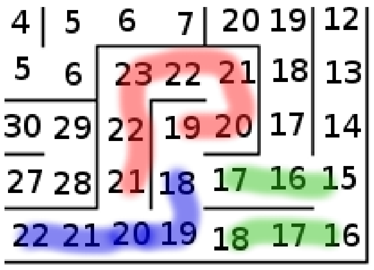


fig. 6

In fig. 5 and fig. 6, after traversing the blue path, the robot goes to the green path and not the red path. If the robot had traversed the red path, it would have been a loop. The 'distance value' prevents it from traversing the red path as it traverses the path of decreasing 'distance value'.

2) Prevents going into dead-ends and dead-paths

For example,

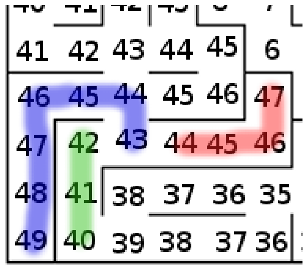


fig. 7

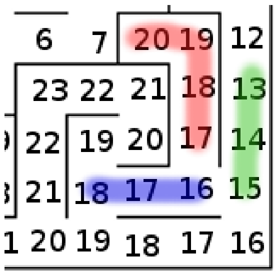


fig. 8

In fig. 7 and fig. 8, after traversing the blue path, the robot goes to the green path and not the red path. The red paths are the dead-paths. The 'distance value' prevents it from traversing the red path as it traverses the path of decreasing 'distance value'.

3) Finds the shortest path length

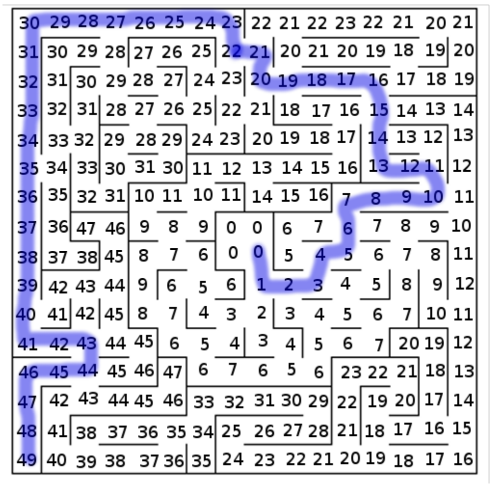


fig. 9

As the robot traverses the path of decreasing ‘distance values’, robot reaches the goal with the shortest path length. One of the shortest paths of test maze 3 is shown in fig.9.

Algorithm in brief:

1) Using technique 1, robot moves to the next cell with ‘distance value’ of equal or lesser than the present cell’s ‘distance value’. Higher priority is given to unvisited cells. If there is no such cell, robot makes a rotation of -90 or 90 randomly with 0 movement.
2) Using technique 2, robot tries to go to the goal from the start position and vice versa as soon as possible. In each step, rotation and movement is stored in a list. When the robot reaches the goal from start or goes from goal to start, a path is obtained.
3) The path is edited for removing redundant paths and loops to get a policy where every step has movement = 1. This policy is edited so that the path can be traversed in minimum number of steps.
4) The policy obtained is compared with previous optimum policy and optimum policy is updated if required.
5) If (goal has been found atleast once) and ((the robot has explored atleast 70% of the maze) or (number of traversals between start and goal is greater than or equal to 5)), then stop the first run and start the second run. In the second run, use the optimum policy found to reach the goal.

Note: I have used ‘number of traversals between start and goal is greater than or equal to 5’ as a corner case when robot fails to explore 70% of maze even after a long time.
**(Refinement) Combine depth first search and dynamic programming: I have improved the algorithm by using depth first search to find the shortest path after exploration phase.

Benchmark

The benchmark model that I have used is breadth first search algorithm with random exploration. The scores obtained by the benchmark model are as follows:

Sl.no	Test maze 1			
	% of maze explored	Path length	No. of moves	Score
1	94.44	30	17	30.5
2	74.31	30	21	32.47
3	Goal not found	Goal not found	Goal not found	Goal not found
4	77.78	34	23	39.6
5	56.94	40	26	34.17

6	41.67	46	29	31.67
7	79.17	30	21	30.23
8	Goal not found	Goal not found	Goal not found	Goal not found
9	82.64	32	22	43.167
10	67.36	34	20	24.47
Average score				33.28

Sl.no	Test maze 2			
	% of maze explored	Path length	No. of moves	Score
1	74.49	51	36	47.6
2	53.57	43	27	39.53
3	50	45	28	35.53
4	80.61	43	26	44.03
5	49.49	51	30	35.7
6	Goal not found	Goal not found	Goal not found	Goal not found
7	Goal not found	Goal not found	Goal not found	Goal not found
8	52.55	43	26	33.7
9	49.48	55	34	40.93
10	41.83	45	29	34.2
Average Score				38.9

Sl.no	Test maze 3			
	% of maze explored	Path length	No. of moves	Score
1	75	49	29	55.47
2	Goal not found	Goal not found	Goal not found	Goal not found
3	60.16	55	35	62.5
4	85.16	49	30	54.53
5	27.73	69	41	43.43
6	64.45	51	29	44.9
7	Goal not found	Goal not found	Goal not found	Goal not found
8	76.95	49	27	49.63
9	30.86	61	32	35.73
10	Goal not found	Goal not found	Goal not found	Goal not found
Average score				49.46

Note: Goal not found means that time step limit of 1000 steps was exceeded

Benchmark scores are
1) Test maze 1 : 33.28
2) Test maze 2 : 38.9
3) Test maze 3 : 49.46

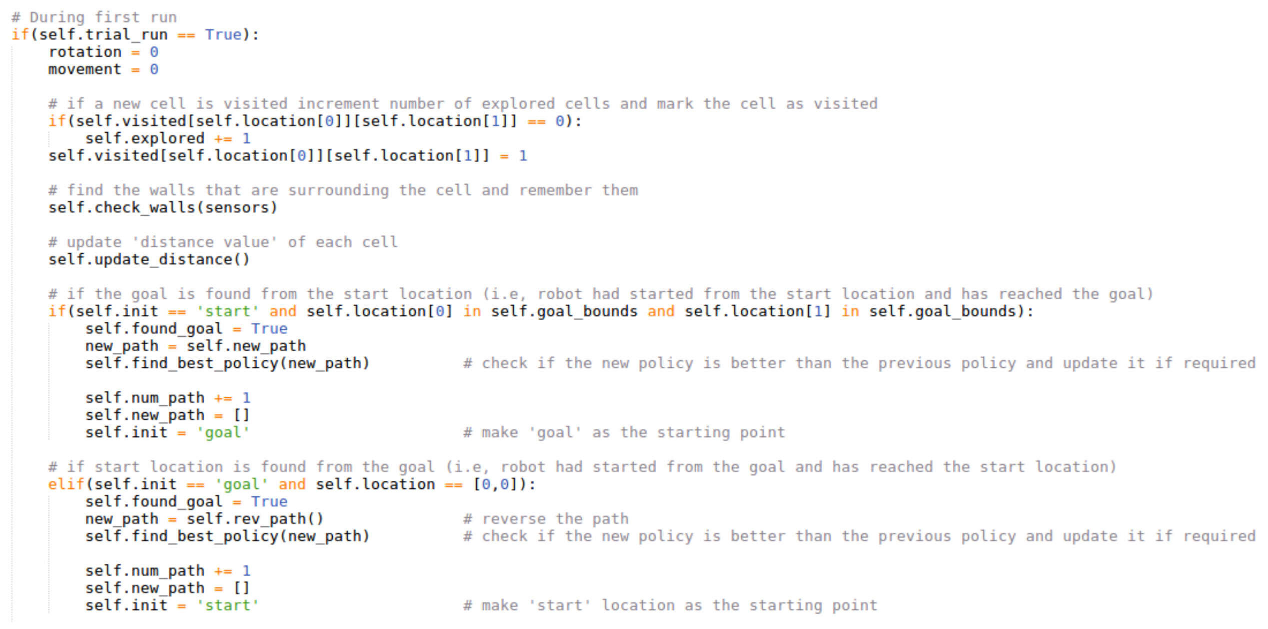
III. Methodology

Data Preprocessing

The sensor specifications and environment designs are already provided and they are 100% accurate. Hence data preprocessing is not required.

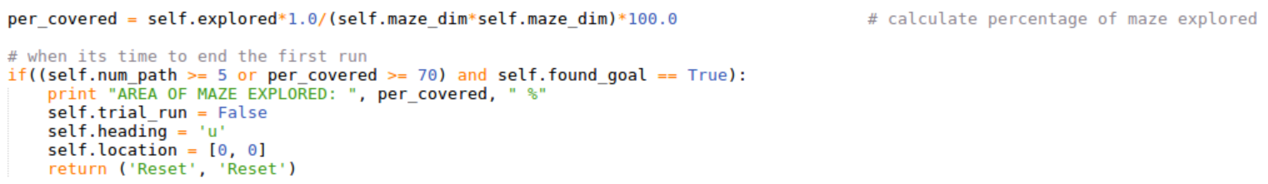
Implementation

During first run:



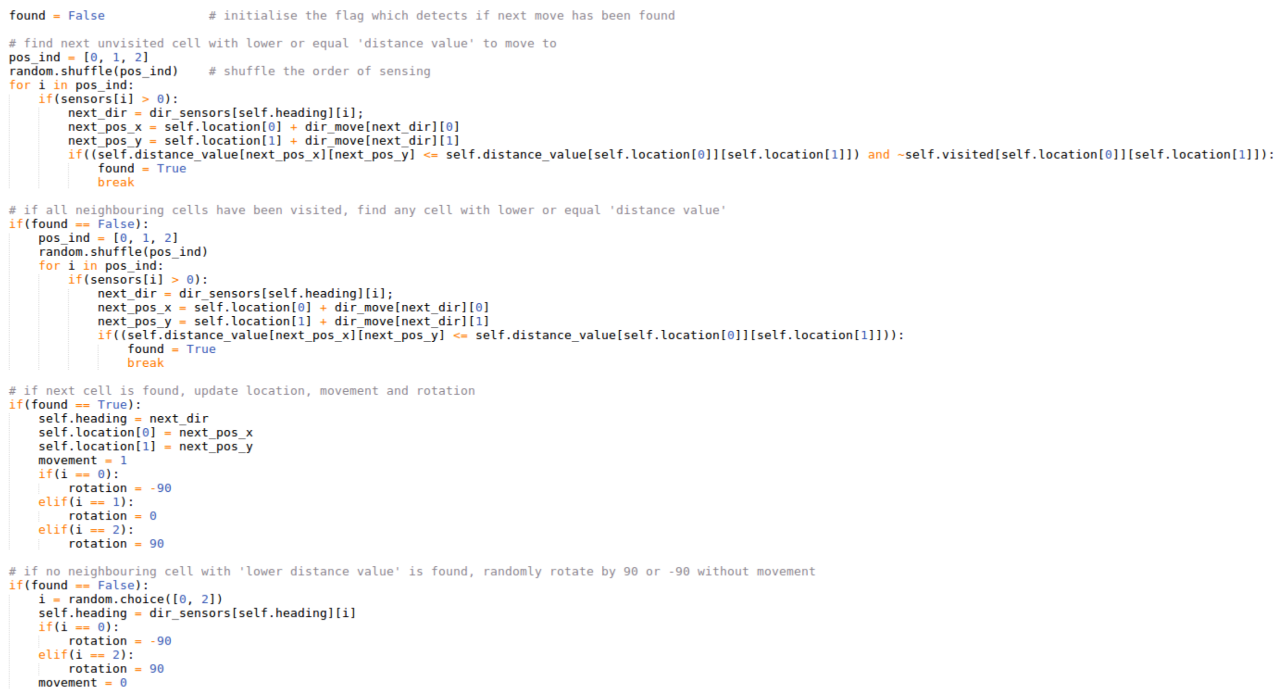
```
# During first run
if(self.trial_run == True):
    rotation = 0
    movement = 0

    # if a new cell is visited increment number of explored cells and mark the cell as visited
    if(self.visited[self.location[0]][self.location[1]] == 0):
        self.explored += 1
    self.visited[self.location[0]][self.location[1]] = 1

    # find the walls that are surrounding the cell and remember them
    self.check_walls(sensors)

    # update 'distance value' of each cell
    self.update_distance()

    # if the goal is found from the start location (i.e, robot had started from the start location and has reached the goal)
    if(self.init == 'start' and self.location[0] in self.goal_bounds and self.location[1] in self.goal_bounds):
        self.found_goal = True
        new_path = self.new_path
        self.find_best_policy(new_path)          # check if the new policy is better than the previous policy and update it if required

        self.num_path += 1
        self.new_path = []
        self.init = 'goal'                       # make 'goal' as the starting point

    # if start location is found from the goal (i.e, robot had started from the goal and has reached the start location)
    elif(self.init == 'goal' and self.location == [0,0]):
        self.found_goal = True
        new_path = self.rev_path()               # reverse the path
        self.find_best_policy(new_path)          # check if the new policy is better than the previous policy and update it if required

        self.num_path += 1
        self.new_path = []
        self.init = 'start'                      # make 'start' location as the starting point
```

On each visit to a cell, it is marked as visited. A function ‘check_walls’ is called to remember the directions of walls surrounding the cell. A function ‘update_distance’ is called to update ‘distance value’ of each cell. If the robot has reached the goal from the start location, the path that was followed is optimised by the ‘find_best_policy’ function and the obtained policy is compared with the previous optimum policy to update it if required. ‘self.init’ is set to ‘goal’ to indicate that the robot is now going from goal to the start location. Similarly, if robot has reached start location from the goal, the path that was followed is first reversed by the ‘rev_path’ function and then optimised by the ‘find_best_policy’ function and the obtained policy is compared with the previous optimum policy to update it if required. ‘self.init’ is set to ‘start’ to indicate that the robot is now going from start location to the goal.



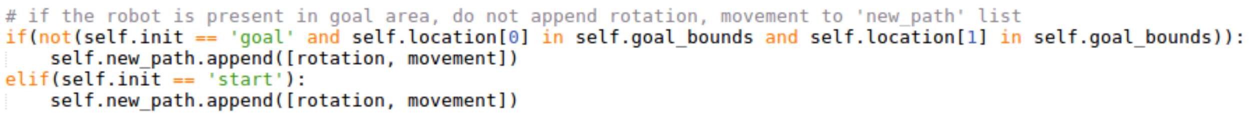
```
per_covered = self.explored*1.0/(self.maze_dim*self.maze_dim)*100.0          # calculate percentage of maze explored

# when its time to end the first run
if((self.num_path >= 5 or per_covered >= 70) and self.found_goal == True):
    print "AREA OF MAZE EXPLORED: ", per_covered, " %"
    self.trial_run = False
    self.heading = 'u'
    self.location = [0, 0]
    return ('Reset', 'Reset')
```

If goal is reached atleast once and (number of paths obtained between start and goal exceeds 5 or percentage of maze explored exceeds 70 %) , then end the first run



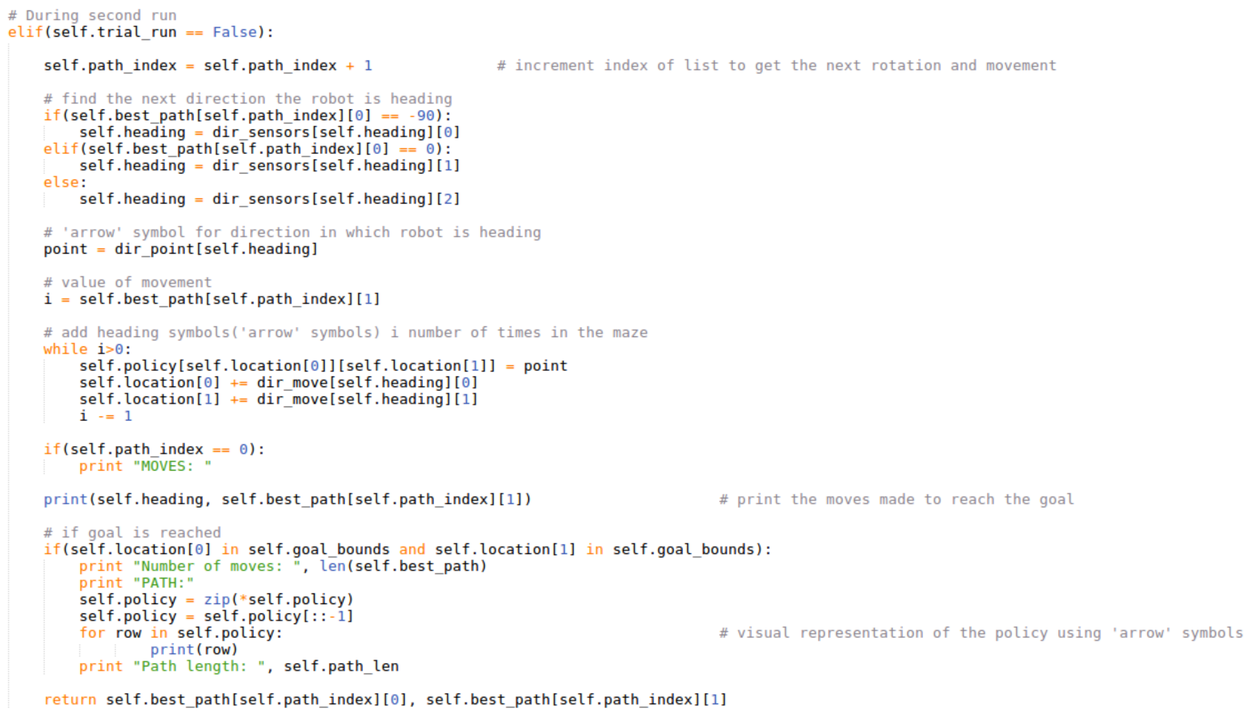
```
found = False               # initialise the flag which detects if next move has been found

# find next unvisited cell with lower or equal 'distance value' to move to
pos_ind = [0, 1, 2]
random.shuffle(pos_ind)     # shuffle the order of sensing
for i in pos_ind:
    if(sensors[i] > 0):
        next_dir = dir_sensors[self.heading][i];
        next_pos_x = self.location[0] + dir_move[next_dir][0]
        next_pos_y = self.location[1] + dir_move[next_dir][1]
        if((self.distance_value[next_pos_x][next_pos_y] <= self.distance_value[self.location[0]][self.location[1]]) and ~self.visited[self.location[0]][self.location[1]]):
            found = True
            break

# if all neighbouring cells have been visited, find any cell with lower or equal 'distance value'
if(found == False):
    pos_ind = [0, 1, 2]
    random.shuffle(pos_ind)
    for i in pos_ind:
        if(sensors[i] > 0):
            next_dir = dir_sensors[self.heading][i];
            next_pos_x = self.location[0] + dir_move[next_dir][0]
            next_pos_y = self.location[1] + dir_move[next_dir][1]
            if((self.distance_value[next_pos_x][next_pos_y] <= self.distance_value[self.location[0]][self.location[1]])):
                found = True
                break

# if next cell is found, update location, movement and rotation
if(found == True):
    self.heading = next_dir
    self.location[0] = next_pos_x
    self.location[1] = next_pos_y
    movement = 1
    if(i == 0):
        rotation = -90
    elif(i == 1):
        rotation = 0
    elif(i == 2):
        rotation = 90

# if no neighbouring cell with 'lower distance value' is found, randomly rotate by 90 or -90 without movement
if(found == False):
    i = random.choice([0, 2])
    self.heading = dir_sensors[self.heading][i]
    if(i == 0):
        rotation = -90
    elif(i == 2):
        rotation = 90
    movement = 0
```

To decide the next move to be made, randomly iterate through the neighbouring cells with no wall in between to check if it is unvisited and has lower 'distance value'. If these two conditions are satisfied, then next move is made to this cell. If all the neighbouring cells with no wall in between have been visited, then randomly choose a neighbouring cell with no wall in between and with lower 'distance value'. If there are no such cells with lower 'distance value', the robot is made to rotate either 90 or -90 degrees randomly with 0 movement.



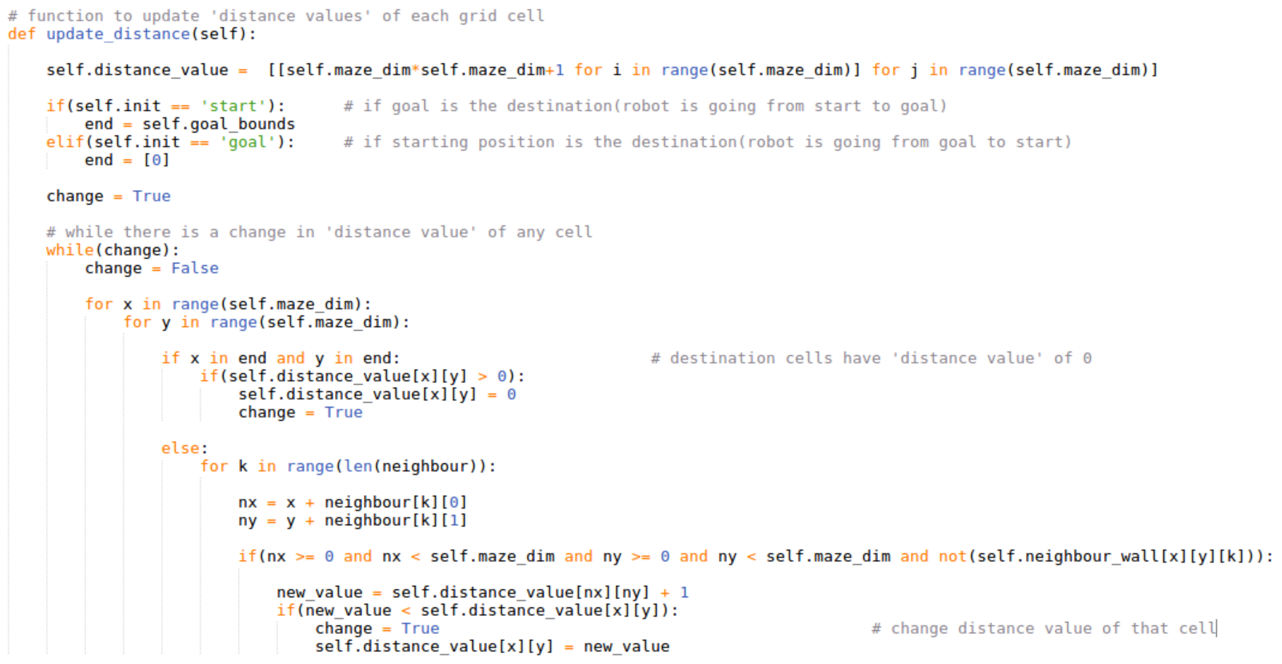
```
# if the robot is present in goal area, do not append rotation, movement to 'new_path' list
if(not(self.init == 'goal' and self.location[0] in self.goal_bounds and self.location[1] in self.goal_bounds)):
    self.new_path.append([rotation, movement])
elif(self.init == 'start'):
    self.new_path.append([rotation, movement])
```

If the robot is starting from the goal and is still present in the goal area, do not append rotation, movement to 'new_path' list as these are redundant steps.



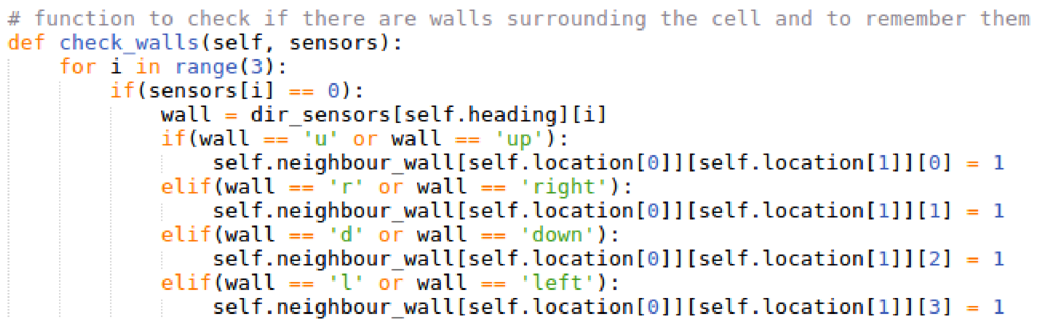
```
# During second run
elif(self.trial_run == False):

    self.path_index = self.path_index + 1              # increment index of list to get the next rotation and movement

    # find the next direction the robot is heading
    if(self.best_path[self.path_index][0] == -90):
        self.heading = dir_sensors[self.heading][0]
    elif(self.best_path[self.path_index][0] == 0):
        self.heading = dir_sensors[self.heading][1]
    else:
        self.heading = dir_sensors[self.heading][2]

    # 'arrow' symbol for direction in which robot is heading
    point = dir_point[self.heading]

    # value of movement
    i = self.best_path[self.path_index][1]

    # add heading symbols('arrow' symbols) i number of times in the maze
    while i>0:
        self.policy[self.location[0]][self.location[1]] = point
        self.location[0] += dir_move[self.heading][0]
        self.location[1] += dir_move[self.heading][1]
        i -= 1

    if(self.path_index == 0):
        print "MOVES: "

    print(self.heading, self.best_path[self.path_index][1])                  # print the moves made to reach the goal

    # if goal is reached
    if(self.location[0] in self.goal_bounds and self.location[1] in self.goal_bounds):
        print "Number of moves: ", len(self.best_path)
        print "PATH:"
        self.policy = zip(*self.policy)
        self.policy = self.policy[::-1]
        for row in self.policy:                                              # visual representation of the policy using 'arrow' symbols
                print(row)
        print "Path length: ", self.path_len

    return self.best_path[self.path_index][0], self.best_path[self.path_index][1]
```

In the second run, use the best policy(self.best_path) obtained during exploration to reach the goal. Print the moves and visual representation (using ‘arrow’ symbols) of the best policy.



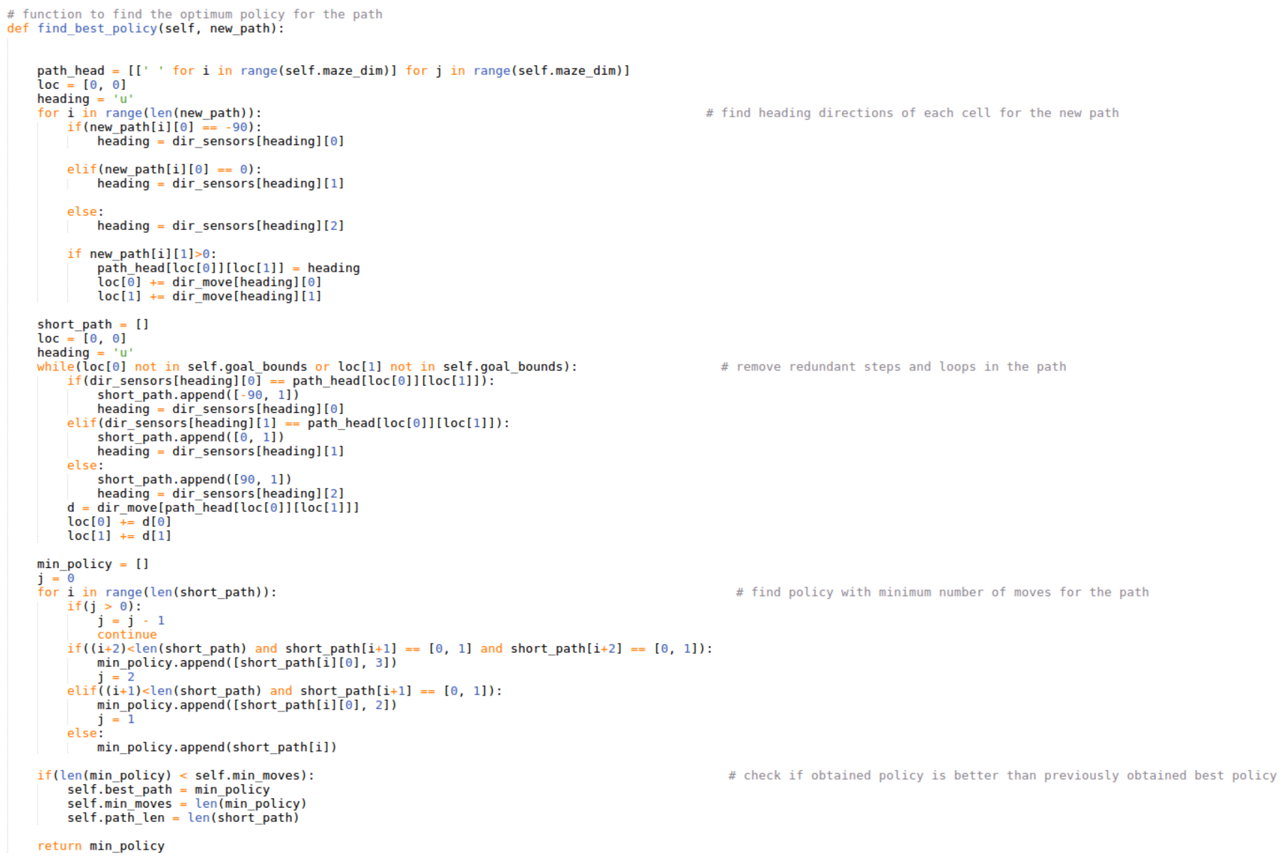
```
# function to update 'distance values' of each grid cell
def update_distance(self):

    self.distance_value =  [[self.maze_dim*self.maze_dim+1 for i in range(self.maze_dim)] for j in range(self.maze_dim)]

    if(self.init == 'start'):      # if goal is the destination(robot is going from start to goal)
        end = self.goal_bounds
    elif(self.init == 'goal'):     # if starting position is the destination(robot is going from goal to start)
        end = [0]

    change = True

    # while there is a change in 'distance value' of any cell
    while(change):
        change = False

        for x in range(self.maze_dim):
            for y in range(self.maze_dim):

                if x in end and y in end:                      # destination cells have 'distance value' of 0
                    if(self.distance_value[x][y] > 0):
                        self.distance_value[x][y] = 0
                        change = True

                else:
                    for k in range(len(neighbour)):

                        nx = x + neighbour[k][0]
                        ny = y + neighbour[k][1]

                        if(nx >= 0 and nx < self.maze_dim and ny >= 0 and ny < self.maze_dim and not(self.neighbour_wall[x][y][k])):

                            new_value = self.distance_value[nx][ny] + 1
                            if(new_value < self.distance_value[x][y]):
                                change = True                              # change distance value of that cell
                                self.distance_value[x][y] = new_value
```

The ‘update_function’ is used to update ‘distance values’ of each cell in the maze. In this function, if start location is the destination, then ‘distance value’ of start location is set to 0 and ‘distance value’ of other cells are updated. If goal is the destination, then ‘distance value’ of goal is set to 0 and ‘distance value’ of other cells are updated. The ‘distance value’ of each cell(other than destination) is updated such that it is the minimum of the ‘distance values’ of neighbouring cells with no wall in between them plus one. The distance values are updated in a loop until there is no change in any of the ‘distance value’, i.e values have reached a stable state.



```python
# function to check if there are walls surrounding the cell and to remember them
def check_walls(self, sensors):
    for i in range(3):
        if(sensors[i] == 0):
            wall = dir_sensors[self.heading][i]
            if(wall == 'u' or wall == 'up'):
                self.neighbour_wall[self.location[0]][self.location[1]][0] = 1
            elif(wall == 'r' or wall == 'right'):
                self.neighbour_wall[self.location[0]][self.location[1]][1] = 1
            elif(wall == 'd' or wall == 'down'):
                self.neighbour_wall[self.location[0]][self.location[1]][2] = 1
            elif(wall == 'l' or wall == 'left'):
                self.neighbour_wall[self.location[0]][self.location[1]][3] = 1
```

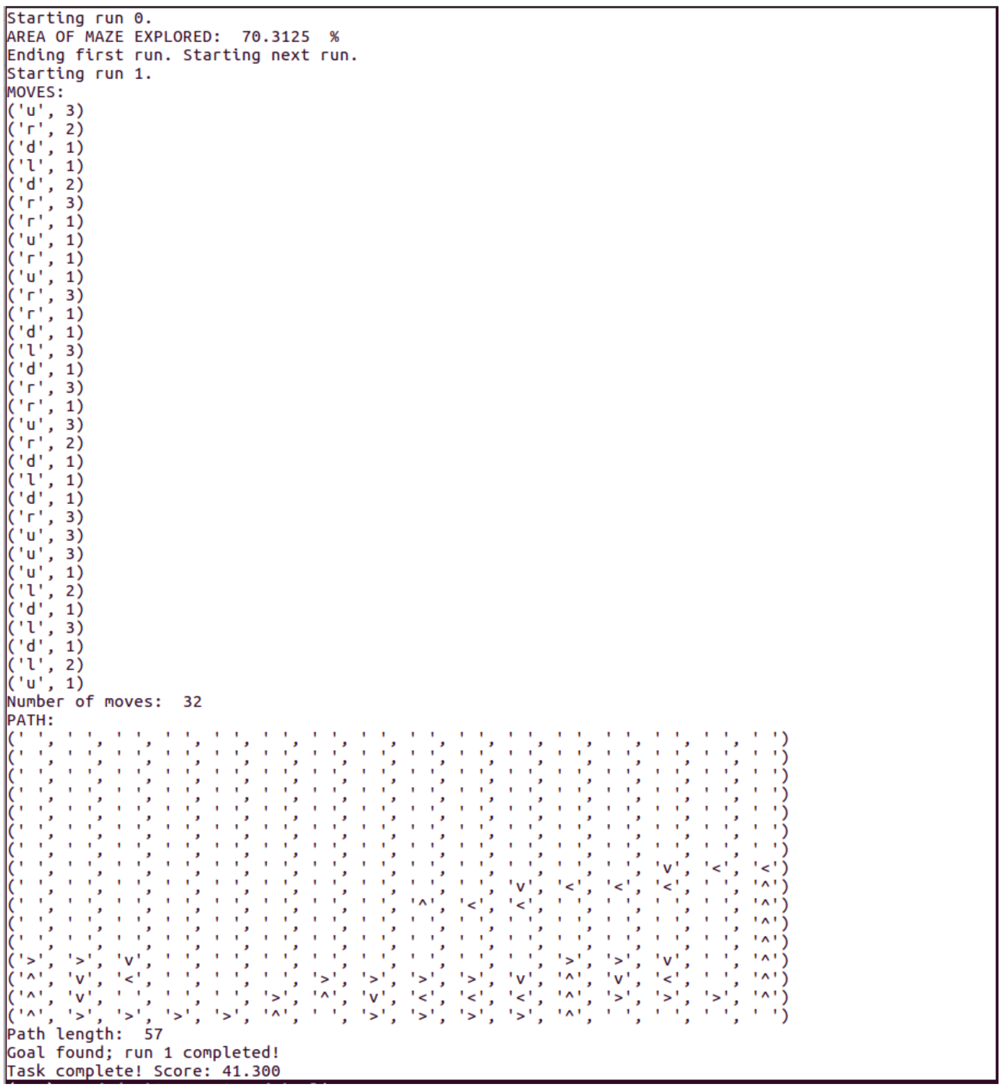
The ‘check_walls’ function is used to find walls surrounding the cell and to remember them. It is stored as a list with 4 elements of binary numbers – 0 indicating there is no wall and 1 indicating there is a wall. The zeroth element of list is for the upper wall, first element for the right wall, second element for lower wall and third element for the left wall. For example, [0, 1, 1, 0] indicates that the cell has no wall in the up and left direction and it has walls in the right and down direction.



```python
# function to find the optimum policy for the path
def find_best_policy(self, new_path):


    path_head = [[' ' for i in range(self.maze_dim)] for j in range(self.maze_dim)]
    loc = [0, 0]
    heading = 'u'
    for i in range(len(new_path)):                                              # find heading directions of each cell for the new path
        if(new_path[i][0] == -90):
            heading = dir_sensors[heading][0]

        elif(new_path[i][0] == 0):
            heading = dir_sensors[heading][1]

        else:
            heading = dir_sensors[heading][2]

        if new_path[i][1]>0:
            path_head[loc[0]][loc[1]] = heading
            loc[0] += dir_move[heading][0]
            loc[1] += dir_move[heading][1]

    short_path = []
    loc = [0, 0]
    heading = 'u'
    while(loc[0] not in self.goal_bounds or loc[1] not in self.goal_bounds):                     # remove redundant steps and loops in the path
        if(dir_sensors[heading][0] == path_head[loc[0]][loc[1]]):
            short_path.append([-90, 1])
            heading = dir_sensors[heading][0]
        elif(dir_sensors[heading][1] == path_head[loc[0]][loc[1]]):
            short_path.append([0, 1])
            heading = dir_sensors[heading][1]
        else:
            short_path.append([90, 1])
            heading = dir_sensors[heading][2]
        d = dir_move[path_head[loc[0]][loc[1]]]
        loc[0] += d[0]
        loc[1] += d[1]

    min_policy = []
    j = 0
    for i in range(len(short_path)):                                                    # find policy with minimum number of moves for the path
        if(j > 0):
            j = j - 1
            continue
        if((i+2)<len(short_path) and short_path[i+1] == [0, 1] and short_path[i+2] == [0, 1]):
            min_policy.append([short_path[i][0], 3])
            j = 2
        elif((i+1)<len(short_path) and short_path[i+1] == [0, 1]):
            min_policy.append([short_path[i][0], 2])
            j = 1
        else:
            min_policy.append(short_path[i])

    if(len(min_policy) < self.min_moves):                                              # check if obtained policy is better than previously obtained best policy
        self.best_path = min_policy
        self.min_moves = len(min_policy)
        self.path_len = len(short_path)

    return min_policy
```

The ‘find_best_policy’ function is used to find the optimum policy. The heading directions of each cell in the new path is found and stored in ‘path_head’. Redundant paths and loops are removed from the new path and stored in ‘short_path’. The policy with minimum number of steps to traverse ‘short_path’ is found by using maximum movement for each step and stored in ‘min_policy’. ‘min_policy’ is then compared with previous optimum policy(self.best_path) and ‘self.best_path’ is updated if required.

Output:



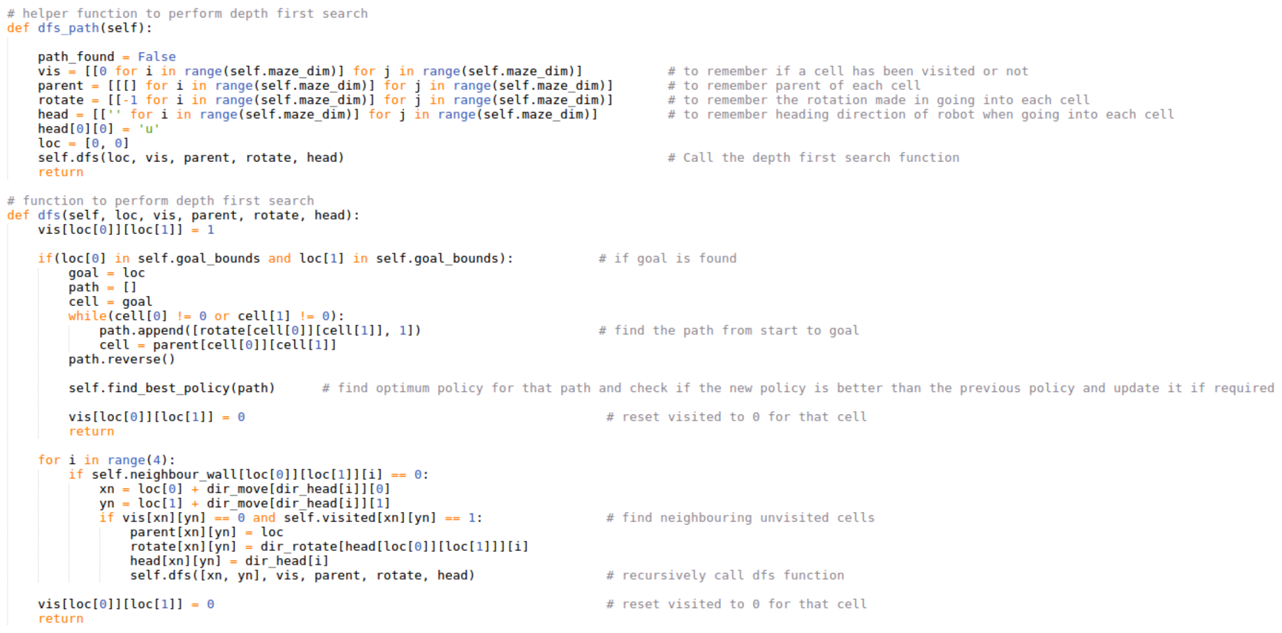
```
Starting run 0.
AREA OF MAZE EXPLORED:  70.3125  %
Ending first run. Starting next run.
Starting run 1.
MOVES:
('u', 3)
('r', 2)
('d', 1)
('l', 1)
('d', 2)
('r', 3)
('r', 1)
('u', 1)
('r', 1)
('u', 1)
('r', 3)
('r', 1)
('d', 1)
('l', 3)
('d', 1)
('r', 3)
('r', 1)
('u', 3)
('r', 2)
('d', 1)
('l', 1)
('d', 1)
('r', 3)
('u', 3)
('u', 3)
('u', 1)
('l', 2)
('d', 1)
('l', 3)
('d', 1)
('l', 2)
('u', 1)
Number of moves:  32
PATH:
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'v', '<', '<')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'v', '<', '<', '<', ' ', '^')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', '<', '<', ' ', ' ', ' ', ' ', '^')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^')
('>', '>', 'v', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '>', '>', 'v', ' ', '^')
('^', 'v', '<', ' ', ' ', ' ', '>', '>', '>', '>', 'v', '^', 'v', '<', ' ', '^')
('^', 'v', ' ', ' ', ' ', '>', '^', 'v', '<', '<', '<', '^', '>', '>', '>', '^')
('^', '>', '>', '>', '>', '^', ' ', '>', '>', '>', '>', '^', ' ', ' ', ' ', ' ')
Path length:  57
Goal found; run 1 completed!
Task complete! Score: 41.300
```

Refinement

Using Depth First Search

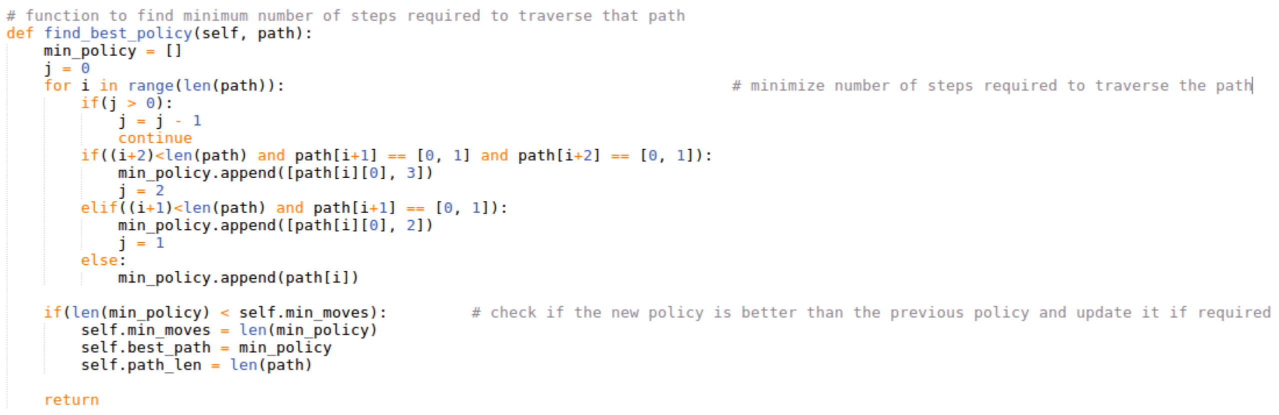
Dynamic programming finds the optimal policy by calculating the policy based on the paths traversed by the robot in exploration phase. But this may not give the best policy always. So I have improved the algorithm by using depth first search after exploration phase to find the optimum policy.

Depth first search algorithm finds all the paths from start location to the goal based on the cells visited in the exploration phase. So the path with minimum number of steps among all these paths can then be obtained by comparing each of the paths. Depth first search makes the best use of the data obtained during the exploration phase to find the optimum policy. Thus this improvement helps find a better policy or atleast as good a policy as the ones found out by just using dynamic programming.



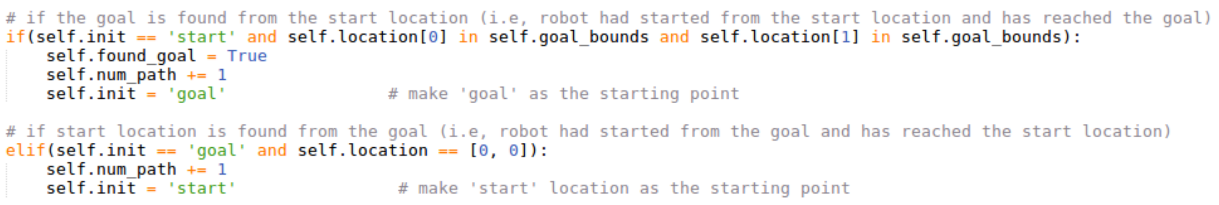
```python
# helper function to perform depth first search
def dfs_path(self):

    path_found = False
    vis = [[0 for i in range(self.maze_dim)] for j in range(self.maze_dim)]         # to remember if a cell has been visited or not
    parent = [[[] for i in range(self.maze_dim)] for j in range(self.maze_dim)]     # to remember parent of each cell
    rotate = [[-1 for i in range(self.maze_dim)] for j in range(self.maze_dim)]     # to remember the rotation made in going into each cell
    head = [['' for i in range(self.maze_dim)] for j in range(self.maze_dim)]       # to remember heading direction of robot when going into each cell
    head[0][0] = 'u'
    loc = [0, 0]
    self.dfs(loc, vis, parent, rotate, head)                                        # Call the depth first search function
    return

# function to perform depth first search
def dfs(self, loc, vis, parent, rotate, head):
    vis[loc[0]][loc[1]] = 1

    if(loc[0] in self.goal_bounds and loc[1] in self.goal_bounds):          # if goal is found
        goal = loc
        path = []
        cell = goal
        while(cell[0] != 0 or cell[1] != 0):
            path.append([rotate[cell[0]][cell[1]], 1])                      # find the path from start to goal
            cell = parent[cell[0]][cell[1]]
        path.reverse()

        self.find_best_policy(path)      # find optimum policy for that path and check if the new policy is better than the previous policy and update it if required

        vis[loc[0]][loc[1]] = 0                                              # reset visited to 0 for that cell
        return

    for i in range(4):
        if self.neighbour_wall[loc[0]][loc[1]][i] == 0:
            xn = loc[0] + dir_move[dir_head[i]][0]
            yn = loc[1] + dir_move[dir_head[i]][1]
            if vis[xn][yn] == 0 and self.visited[xn][yn] == 1:               # find neighbouring unvisited cells
                parent[xn][yn] = loc
                rotate[xn][yn] = dir_rotate[head[loc[0]][loc[1]]][i]
                head[xn][yn] = dir_head[i]
                self.dfs([xn, yn], vis, parent, rotate, head)                # recursively call dfs function

    vis[loc[0]][loc[1]] = 0                                                  # reset visited to 0 for that cell
    return
```

This is the code for Depth First Search. It is a recursive function whose base case is when the path reaches the goal. The function finds all paths leading to the goal. The paths obtained are then optimised by minimizing the number of steps. The minimized policy obtained for each path is compared with the previous best policy and it is updated if required.



```python
# function to find minimum number of steps required to traverse that path
def find_best_policy(self, path):
    min_policy = []
    j = 0
    for i in range(len(path)):                                  # minimize number of steps required to traverse the path
        if(j > 0):
            j = j - 1
            continue
        if((i+2)<len(path) and path[i+1] == [0, 1] and path[i+2] == [0, 1]):
            min_policy.append([path[i][0], 3])
            j = 2
        elif((i+1)<len(path) and path[i+1] == [0, 1]):
            min_policy.append([path[i][0], 2])
            j = 1
        else:
            min_policy.append(path[i])

    if(len(min_policy) < self.min_moves):        # check if the new policy is better than the previous policy and update it if required
        self.min_moves = len(min_policy)
        self.best_path = min_policy
        self.path_len = len(path)

    return
```

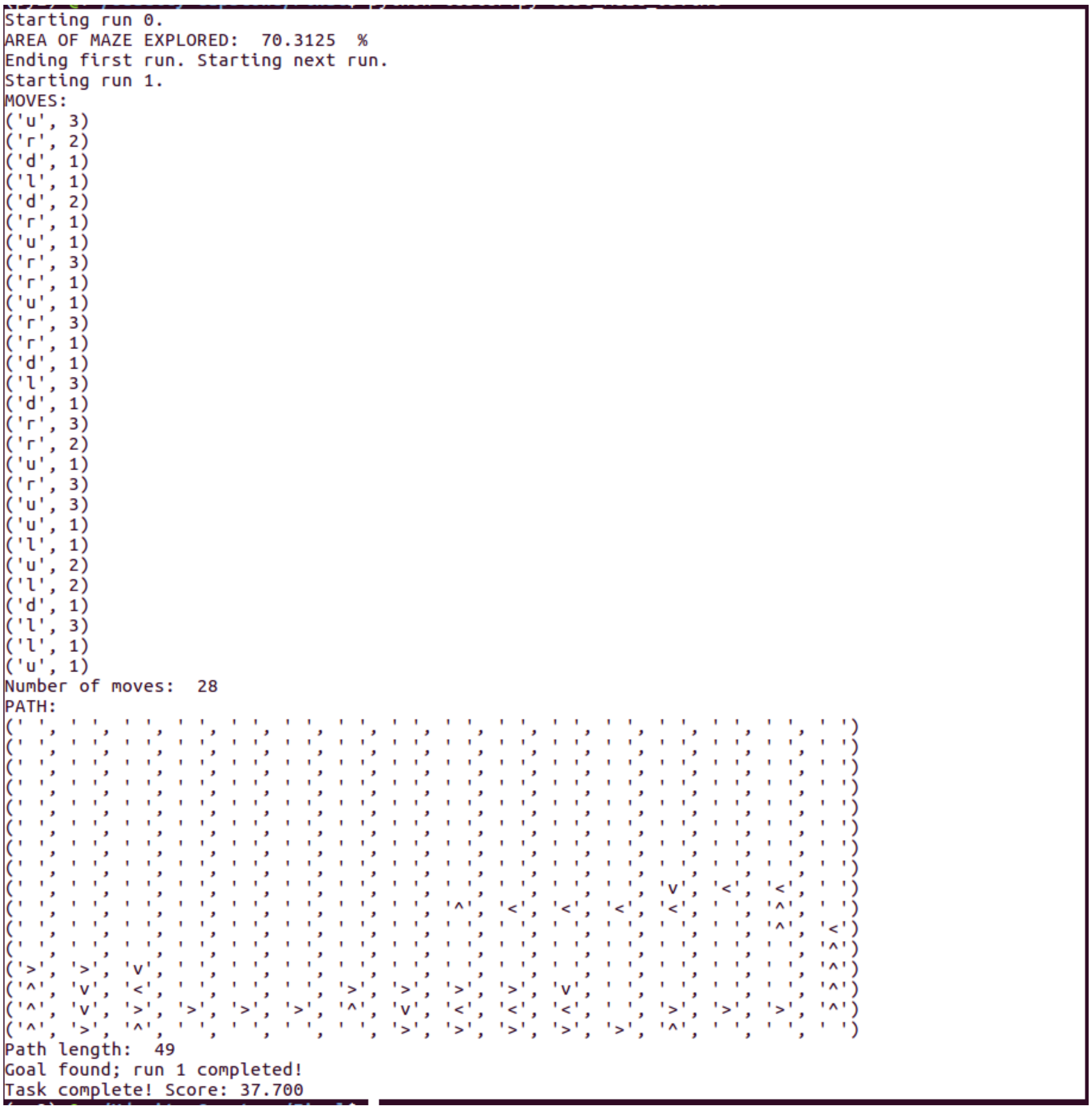
The ‘find_best_policy’ function is the shorter version of the previous one because depth first search takes care of finding the path with the shortest length.



```python
# if the goal is found from the start location (i.e, robot had started from the start location and has reached the goal)
if(self.init == 'start' and self.location[0] in self.goal_bounds and self.location[1] in self.goal_bounds):
    self.found_goal = True
    self.num_path += 1
    self.init = 'goal'                  # make 'goal' as the starting point

# if start location is found from the goal (i.e, robot had started from the goal and has reached the start location)
elif(self.init == 'goal' and self.location == [0, 0]):
    self.num_path += 1
    self.init = 'start'                  # make 'start' location as the starting point
```

This part of the code is a little different from the older version. The robot no longer needs to remember the path taken by it to go from start to goal and vice-versa because depth first search finds the path. Hence ‘self.path’ is no longer required.

Output:



```
Starting run 0.
AREA OF MAZE EXPLORED:  70.3125  %
Ending first run. Starting next run.
Starting run 1.
MOVES:
('u', 3)
('r', 2)
('d', 1)
('l', 1)
('d', 2)
('r', 1)
('u', 1)
('r', 3)
('r', 1)
('u', 1)
('r', 3)
('r', 1)
('d', 1)
('l', 3)
('d', 1)
('r', 3)
('r', 2)
('u', 1)
('r', 3)
('u', 3)
('u', 1)
('l', 1)
('u', 2)
('l', 2)
('d', 1)
('l', 3)
('l', 1)
('u', 1)
Number of moves:  28
PATH:
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'v', '<', '<', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', '<', '<', '<', '<', ' ', '^', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', '<')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^')
('>', '>', 'v', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^')
('^', 'v', '<', ' ', ' ', ' ', '>', '>', '>', '>', 'v', ' ', ' ', ' ', ' ', '^')
('^', 'v', '>', '>', '>', '>', '^', 'v', '<', '<', '<', ' ', '>', '>', '>', '^')
('^', '>', '^', ' ', ' ', ' ', ' ', '>', '>', '>', '>', '>', '^', ' ', ' ', ' ')
Path length:  49
Goal found; run 1 completed!
Task complete! Score: 37.700
```

Compared to the previous version, combining depth first search with dynamic programming, gives a better score or atleast an equal score as that of the previous version on average.

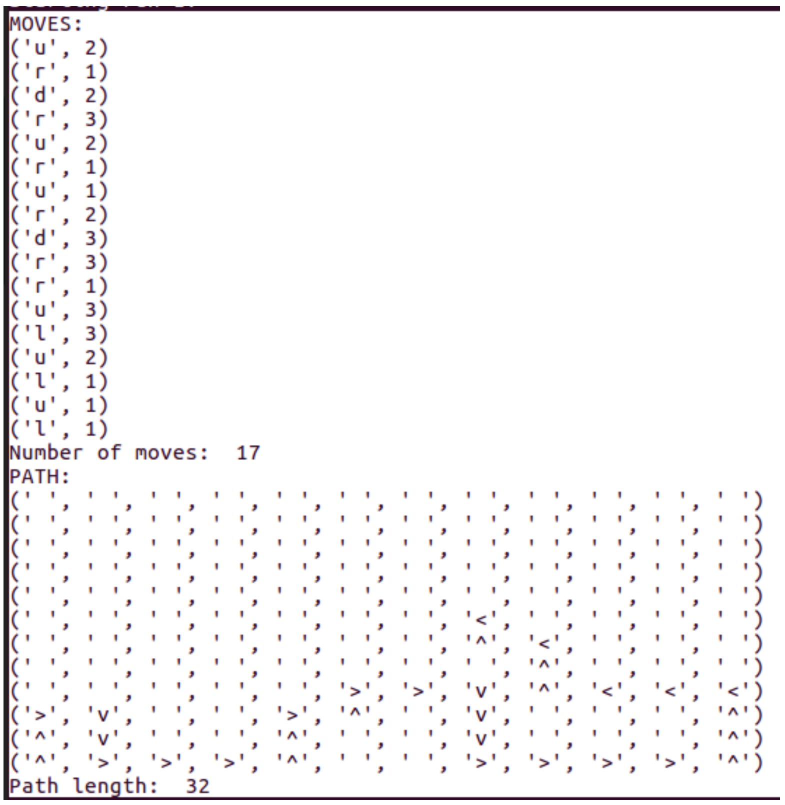
IV. Results

Model Evaluation and Validation

Comparison of my output with the most optimal path

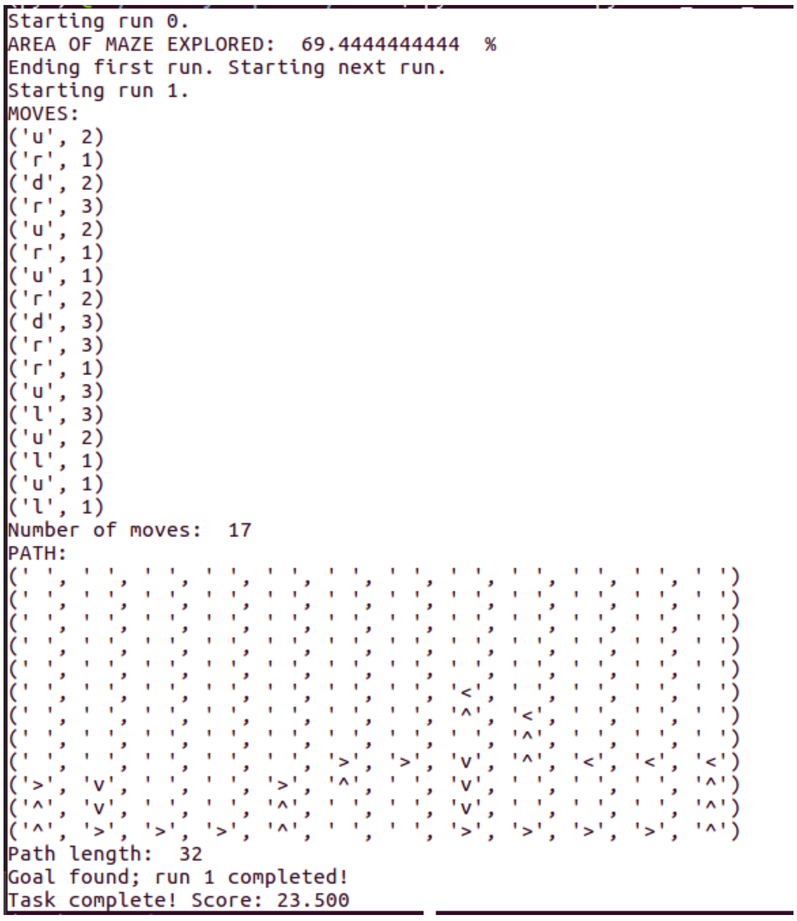
Test maze 1

Optimal path :



```
MOVES:
('u', 2)
('r', 1)
('d', 2)
('r', 3)
('u', 2)
('r', 1)
('u', 1)
('r', 2)
('d', 3)
('r', 3)
('r', 1)
('u', 3)
('l', 3)
('u', 2)
('l', 1)
('u', 1)
('l', 1)
Number of moves:  17
PATH:
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', '<', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', '<', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', '>', '>', 'v', '^', '<', '<', '<')
('>', 'v', ' ', ' ', '>', '^', ' ', 'v', ' ', ' ', ' ', '^')
('^', 'v', ' ', ' ', '^', ' ', ' ', 'v', ' ', ' ', ' ', '^')
('^', '>', '>', '>', '^', ' ', ' ', '>', '>', '>', '>', '^')
Path length:  32
```

My Output:

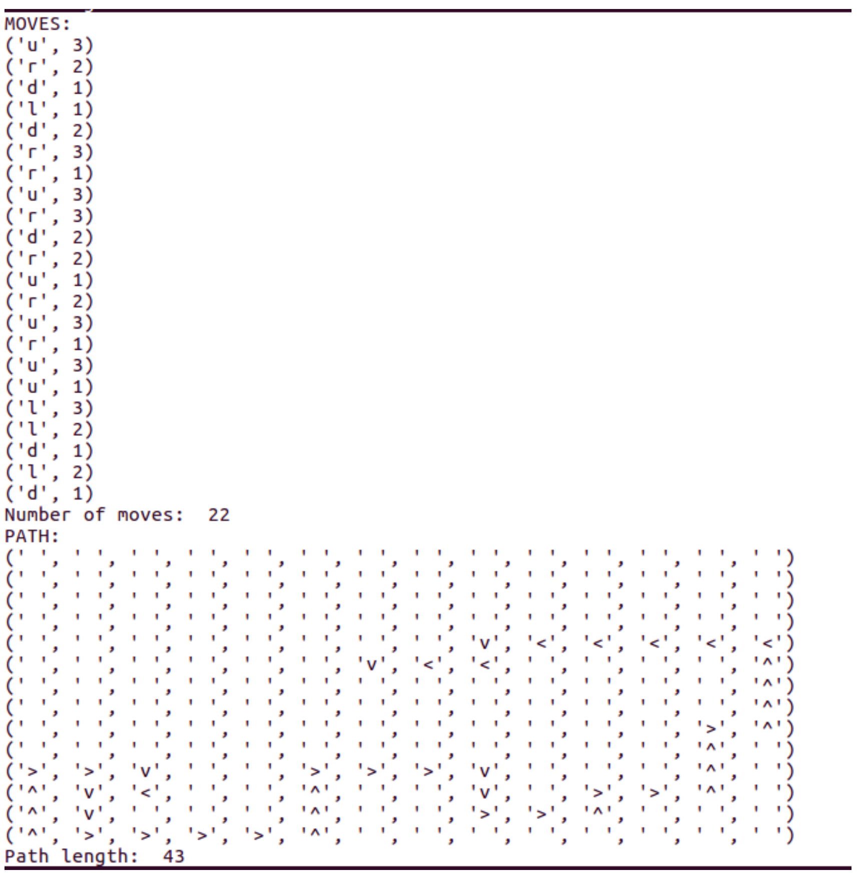


```
Starting run 0.
AREA OF MAZE EXPLORED:  69.4444444444  %
Ending first run. Starting next run.
Starting run 1.
MOVES:
('u', 2)
('r', 1)
('d', 2)
('r', 3)
('u', 2)
('r', 1)
('u', 1)
('r', 2)
('d', 3)
('r', 3)
('r', 1)
('u', 3)
('l', 3)
('u', 2)
('l', 1)
('u', 1)
('l', 1)
Number of moves:  17
PATH:
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', '<', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', '<', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', '>', '>', 'v', '^', '<', '<', '<')
('>', 'v', ' ', ' ', '>', '^', ' ', 'v', ' ', ' ', ' ', '^')
('^', 'v', ' ', ' ', '^', ' ', ' ', 'v', ' ', ' ', ' ', '^')
('^', '>', '>', '>', '^', ' ', ' ', '>', '>', '>', '>', '^')
Path length:  32
Goal found; run 1 completed!
Task complete! Score: 23.500
```

The algorithm comes up with the most optimal path for test maze 1.

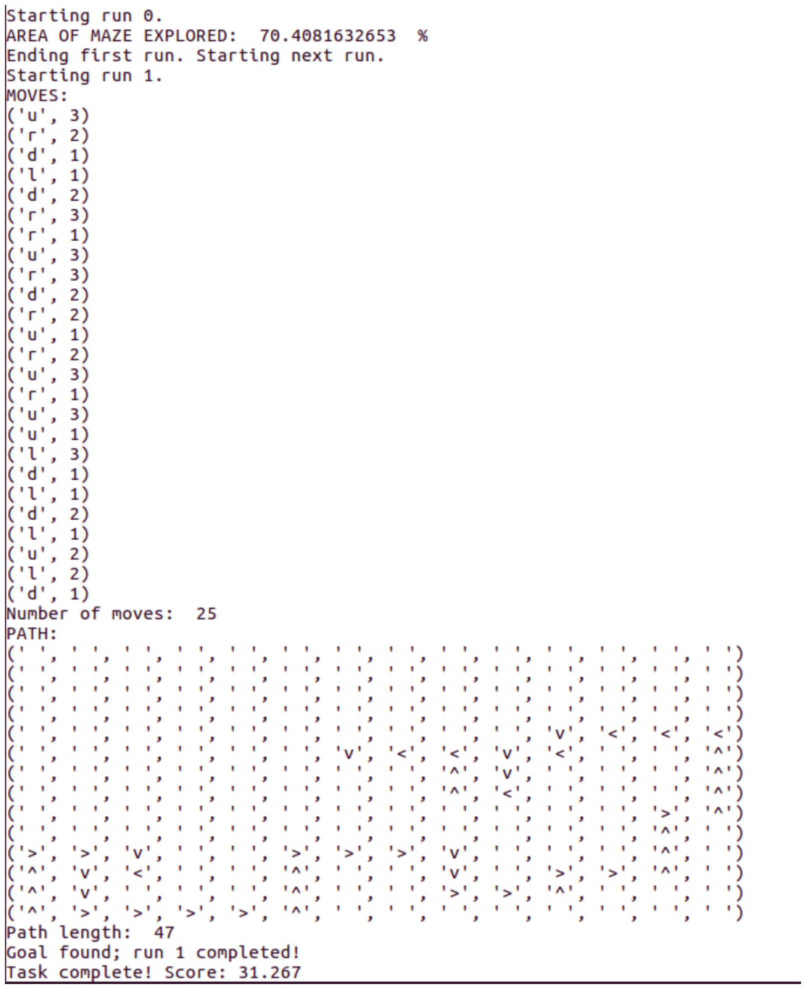
Test maze 2:

Optimal path:



```
MOVES:
('u', 3)
('r', 2)
('d', 1)
('l', 1)
('d', 2)
('r', 3)
('r', 1)
('u', 3)
('r', 3)
('d', 2)
('r', 2)
('u', 1)
('r', 2)
('u', 3)
('r', 1)
('u', 3)
('u', 1)
('l', 3)
('l', 2)
('d', 1)
('l', 2)
('d', 1)
Number of moves:  22
PATH:
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'v', '<', '<', '<', '<', '<')
(' ', ' ', ' ', ' ', ' ', ' ', 'v', '<', '<', ' ', ' ', ' ', ' ', '^')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '>', '^')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', ' ')
('>', '>', 'v', ' ', ' ', '>', '>', '>', 'v', ' ', ' ', ' ', '^', ' ')
('^', 'v', '<', ' ', ' ', '^', ' ', ' ', 'v', ' ', '>', '>', '^', ' ')
('^', 'v', ' ', ' ', ' ', '^', ' ', ' ', '>', '>', '^', ' ', ' ', ' ')
('^', '>', '>', '>', '>', '^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
Path length:  43
```

My Output:

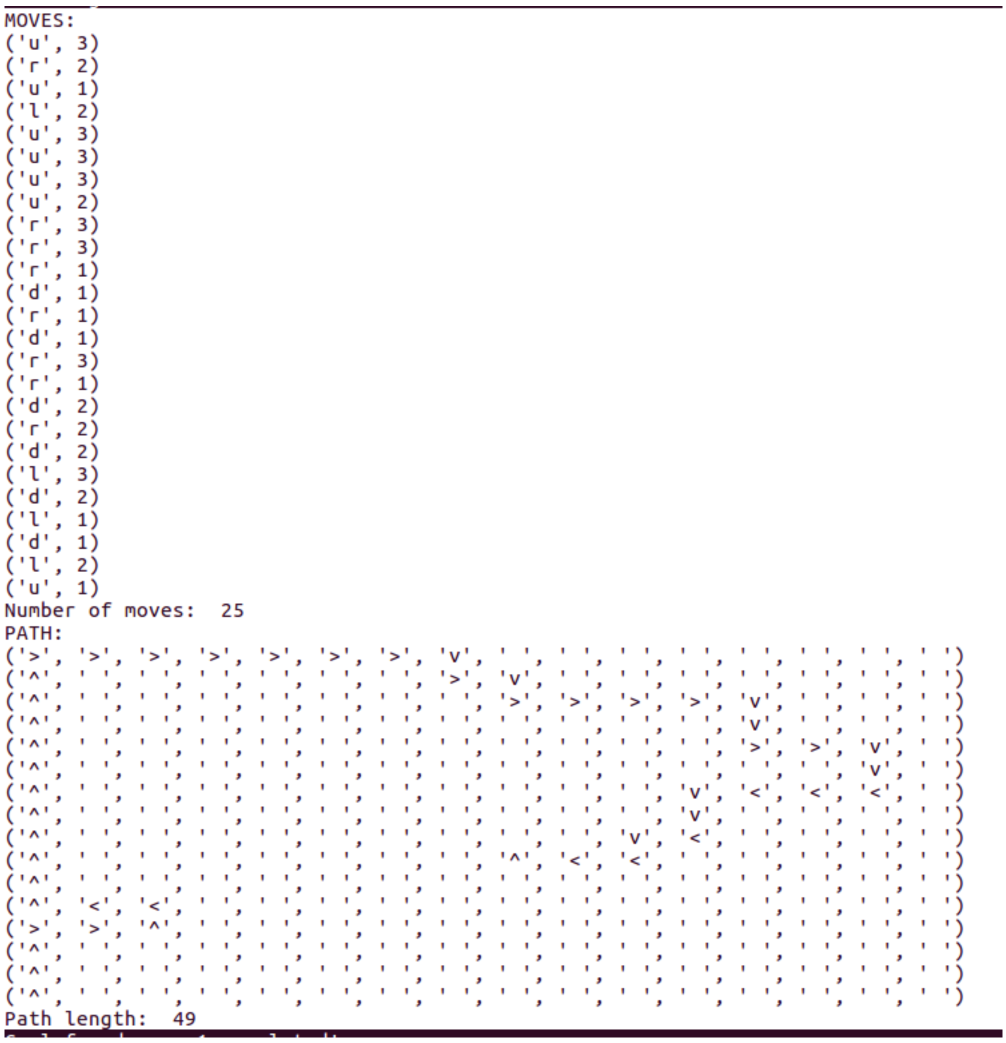


```
Starting run 0.
AREA OF MAZE EXPLORED:  70.4081632653  %
Ending first run. Starting next run.
Starting run 1.
MOVES:
('u', 3)
('r', 2)
('d', 1)
('l', 1)
('d', 2)
('r', 3)
('r', 1)
('u', 3)
('r', 3)
('d', 2)
('r', 2)
('u', 1)
('r', 2)
('u', 3)
('r', 1)
('u', 3)
('u', 1)
('l', 3)
('d', 1)
('l', 1)
('d', 2)
('l', 1)
('u', 2)
('l', 2)
('d', 1)
Number of moves:  25
PATH:
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'v', '<', '<', '<')
(' ', ' ', ' ', ' ', ' ', ' ', 'v', '<', '<', 'v', '<', ' ', ' ', '^')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', 'v', ' ', ' ', ' ', '^')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', '<', ' ', ' ', ' ', '^')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '>', '^')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', ' ')
('>', '>', 'v', ' ', ' ', '>', '>', '>', 'v', ' ', ' ', ' ', '^', ' ')
('^', 'v', '<', ' ', ' ', '^', ' ', ' ', 'v', ' ', '>', '>', '^', ' ')
('^', 'v', ' ', ' ', ' ', '^', ' ', ' ', '>', '>', '^', ' ', ' ', ' ')
('^', '>', '>', '>', '>', '^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
Path length:  47
Goal found; run 1 completed!
Task complete! Score: 31.267
```

For test maze 2, the algorithm comes up with a path with 25 moves which is 3 more than that of optimal path and whose path length is 47 which is 4 more than that of optimal path.

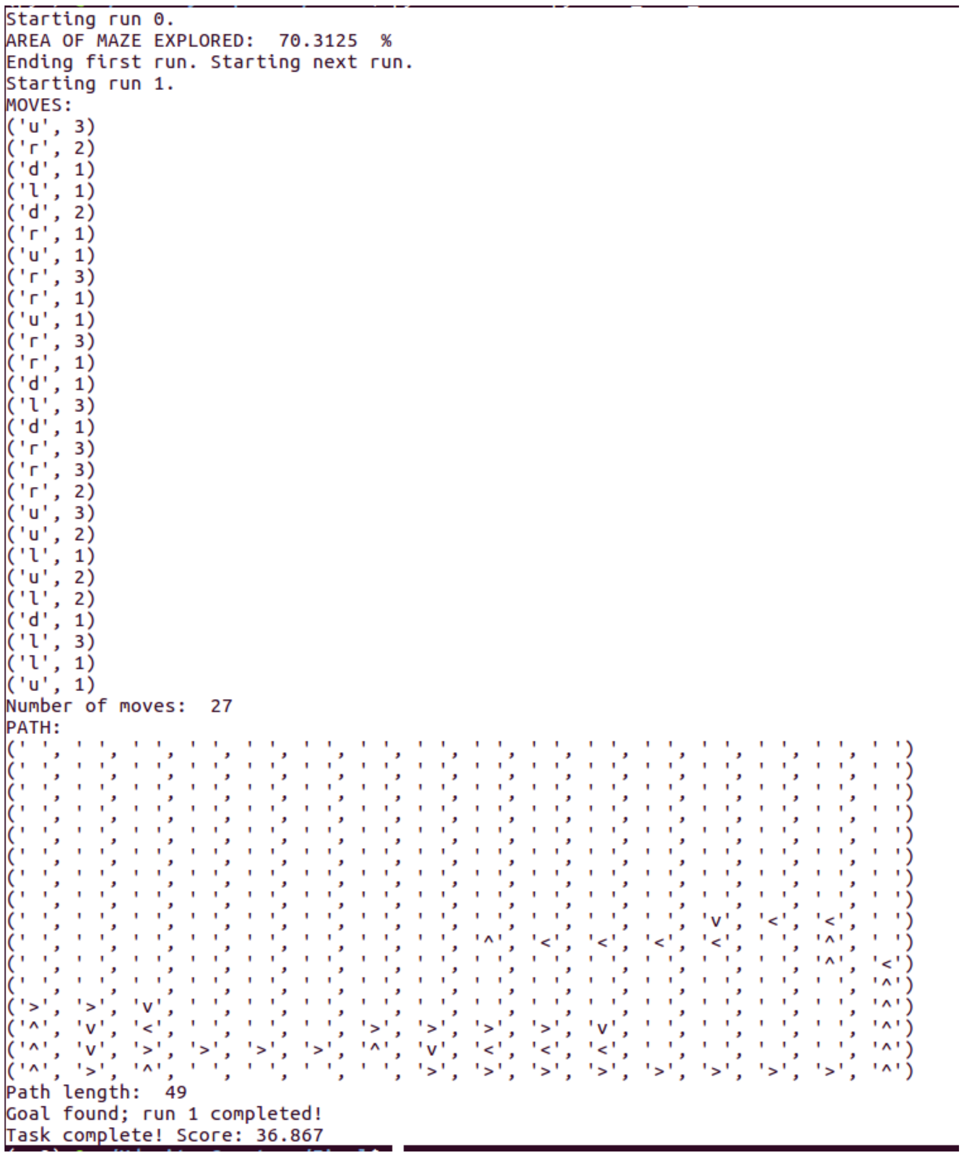
<u>Test maze 3</u>

<u>Optimal path:</u>



```
MOVES:
('u', 3)
('r', 2)
('u', 1)
('l', 2)
('u', 3)
('u', 3)
('u', 3)
('u', 2)
('r', 3)
('r', 3)
('r', 1)
('d', 1)
('r', 1)
('d', 1)
('r', 3)
('r', 1)
('d', 2)
('r', 2)
('d', 2)
('l', 3)
('d', 2)
('l', 1)
('d', 1)
('l', 2)
('u', 1)
Number of moves:  25
PATH:
('>', '>', '>', '>', '>', '>', '>', 'v', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('^', ' ', ' ', ' ', ' ', ' ', ' ', '>', 'v', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '>', '>', '>', '>', 'v', ' ', ' ', ' ')
('^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'v', ' ', ' ', ' ')
('^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '>', '>', 'v', ' ')
('^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'v', ' ')
('^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'v', '<', '<', '<', ' ')
('^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'v', ' ', ' ', ' ', ' ')
('^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'v', '<', ' ', ' ', ' ', ' ')
('^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', '<', '<', ' ', ' ', ' ', ' ', ' ')
('^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('^', '<', '<', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('>', '>', '^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
('^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
Path length:  49
```

My output:



```
Starting run 0.
AREA OF MAZE EXPLORED:  70.3125  %
Ending first run. Starting next run.
Starting run 1.
MOVES:
('u', 3)
('r', 2)
('d', 1)
('l', 1)
('d', 2)
('r', 1)
('u', 1)
('r', 3)
('r', 1)
('u', 1)
('r', 3)
('r', 1)
('d', 1)
('l', 3)
('d', 1)
('r', 3)
('r', 3)
('r', 2)
('u', 3)
('u', 2)
('l', 1)
('u', 2)
('l', 2)
('d', 1)
('l', 3)
('l', 1)
('u', 1)
Number of moves:  27
PATH:
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'v', '<', '<', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', '<', '<', '<', '<', ' ', '^', ' ')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', '<')
(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^')
('>', '>', 'v', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^')
('^', 'v', '<', ' ', ' ', ' ', '>', '>', '>', '>', 'v', ' ', ' ', ' ', ' ', '^')
('^', 'v', '>', '>', '>', '>', '^', 'v', '<', '<', '<', ' ', ' ', ' ', ' ', '^')
('^', '>', '^', ' ', ' ', ' ', ' ', '>', '>', '>', '>', '>', '>', '>', '>', '^')
Path length:  49
Goal found; run 1 completed!
Task complete! Score: 36.867
```

For test maze 3, the algorithm comes up with a path with same length of that of optimal path and number of moves equal to 27 which is 2 more than that of the optimal path.

From these observations, we can say that the algorithm is performing well as the solutions obtained are very close to the most optimal solutions.

Justification

Sl.no	Test maze 1			
	% of maze explored	Path length	No. of moves	Score
1	70.14	32	17	23.2
2	70.14	32	17	22.47
3	70.14	36	18	23.13
4	70.14	32	17	22.5
5	65.97	32	17	23.5

6	70.14	32	17	22.87
7	70.14	36	18	22.7
8	70.14	32	17	23.33
9	70.14	32	17	22.3
10	70.14	36	18	22.97
Average score				22.9

Sl.no	Test maze 2			
	% of maze explored	Path length	No. of moves	Score
1	70.4	43	24	30.7
2	70.4	47	30	36.4
3	70.4	47	30	36.13
4	70.4	47	27	34.57
5	70.4	51	34	40.2
6	70.4	45	28	34.47
7	70.4	49	30	36.33
8	70.4	51	32	38.17
9	70.4	45	28	34.27
10	70.4	45	28	34.27
Average Score				35.55

Sl.no	Test maze 3			
	% of maze explored	Path length	No. of moves	Score
1	70.3	49	27	36.2
2	70.3	51	31	40.6
3	70.3	51	27	36.1
4	70.3	51	25	34.53
5	70.3	49	27	36.77
6	70.3	51	27	36.57
7	70.3	49	28	39.13
8	70.3	49	28	37.43
9	70.3	49	27	36.07
10	70.3	51	28	36.47
Average score				36.99

Average scores are
1) Test maze 1 : 22.9
2) Test maze 2 : 35.55
3) Test maze 3 : 36.99

Compared to the benchmark scores, the average scores obtained using this algorithm is much better. Also unlike the benchmark model, this model will surely reach the goal. The scores of multiple attempts are much closer to each other than it was in the benchmark model. Hence we can conclude that this model performs much better than the benchmark model.

V. Conclusion

Free-Form Visualization

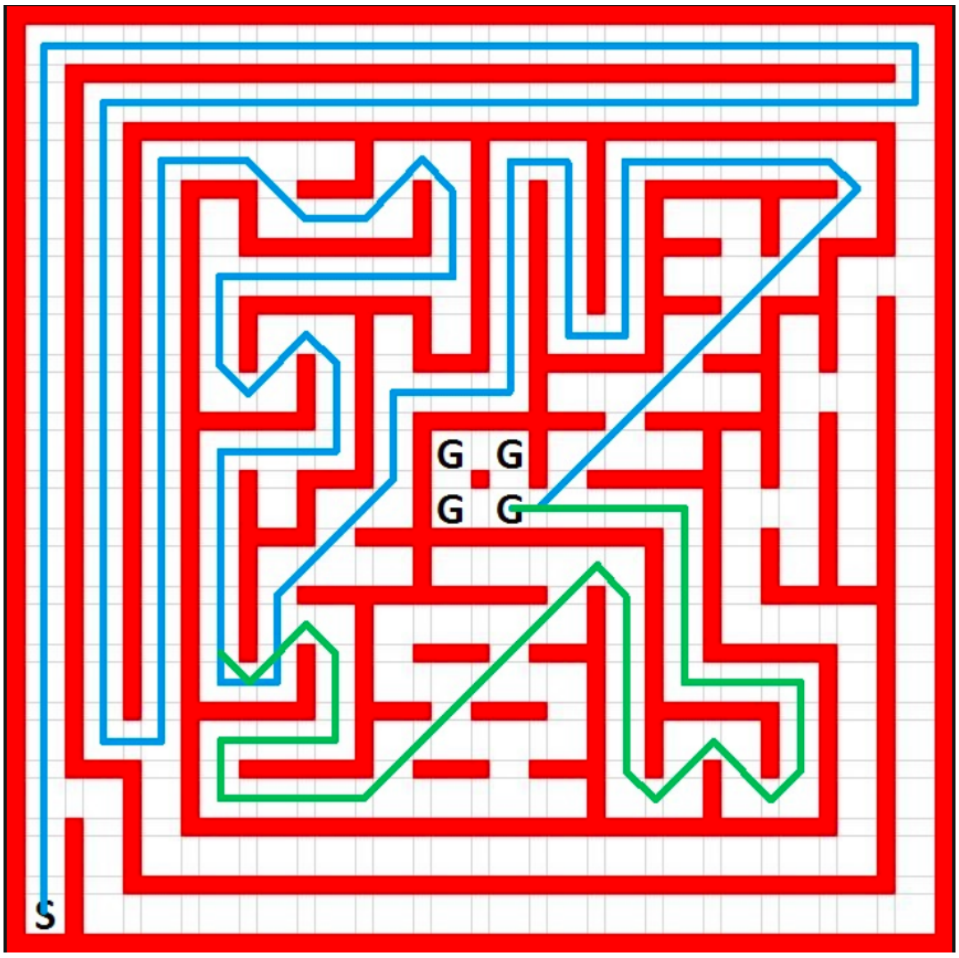


fig. 10

The image above is the maze for the 2016 APEC Micromouse Competition held in USA. There were typically 2 paths for speed runs, Marked in Blue and Green. Both paths have exactly same number of cells which is 131, but the green path requires minimum of 67 moves while the blue path requires minimum of 68 moves.

Characteristics of the maze:

1) Has lot of long straight paths
The algorithm takes advantage of this because the optimum policy obtained from the algorithm minimizes the number of steps needed to traverse a path.

2) Has very few entries into inner parts of the maze
Here 'distance values' play an important role in finding paths to the centre of the maze. With random rotations, there is very less chance for the robot to reach the goal atleast once. In fact, the benchmark model could not find the goal for more than 50% of the attempts. While this model always finds the goal.

The results of the algorithm on this maze are as follows:

Sl.no	2016 APEC Micromouse Competition			
	% of maze explored	Path length	No. of moves	Score
1	70.3	135	72	79.4
2	70.3	133	72	78.5
3	70.3	135	72	79.17
4	70.3	133	70	76.8
5	70.3	133	67	73.63
6	70.3	137	71	77.77
7	70.3	135	72	79
8	70.3	135	72	78.93
9	70.3	133	71	77.93
10	70.3	133	68	74.9
Average score				77.6

In the 5th attempt, the number of moves is equal to the green path (as shown in fig. 10) but the path length is 2 more than that of the green path. In the 10th attempt, the number of moves is equal to the blue path (as shown in fig. 10) but the path length is 2 more than that of the blue path. The scores, path length and number of moves in these attempts are pretty close to the optimal solution and are close to each other too. The robot will surely reach the goal atleast once unlike the benchmark model.

Reflection

Main challenges that I came across in the project were

1) Reaching the goal as soon as possible in the exploration phase;
Using dynamic programming made things much simpler. The robot would just have to traverse the path of decreasing 'distance value'. The 'update_distance' function had the greatest role to play here.

2) Reaching the start location from goal:
This required a clever trick which was to make the start location as the destination and goal as the starting point in the 'update_distance' function.

3) Reversing the path obtained when robot moves from goal to start:
Reversing movement is easy. But reversing rotation is not so easy. The main idea was to identify the inverse of each rotation.

4) Finding optimum policy for each path:
This was one the biggest challenges. Once the path from start to goal was obtained, a basic policy would be obtained without any changes. Then loops and redundant paths from the basic policy would be removed by traversing using the policy to get a path with shortest path length. This path with shortest path length would then be edited so that it could be traversed with minimum number of moves.

5) Idea of depth first search to find the optimum policy among all policies:
Using depth first search with dynamic programming ensured that the optimal policy obtained would be better or atleast as good as the optimal policy obtained using dynamic programming only.

Improvement

In this project, movement, rotations, time, locations and other factors were in discrete domain. While in the actual Micromouse Competition, everything is continuous domain. In continuous domain, we face lots of challenges and the path finding algorithm may be the least of the worries.

In continuous domain, the robot will need to monitor its exact location in the maze. The robot will also have to maintain certain distance with the walls to avoid collisions.. During turns, the robot will need to slow down and perform rotations which are continuous unlike the the discrete values of 90 and -90. On straight paths, the robot will need to increase its speed to save time. The robot can also move diagonally in continuous domain, the free form visualization above has diagonal paths. While robot cannot move diagonally in discrete domain.

Possible extensions for the project:

1) Random start and goal locations unlike the fixed ones in this project.
2) Each cell has a weight and the objective is to go from start to goal in the path with minimum total weight.