Data Science Nanodegree Project

Capstone Project – Robot Motion Planning

Pradeep Kumar
16 May 2020

Plot and Navigate a Virtual Maze

I. Definition

Project Overview

This project is inspired from an event 'Micromouse Competitions' where a small robot navigates a maze and learns the shortest path to the center of that maze. The competition originated in the 1970s and is a popular event held around the globe. The project deals with the basic concepts of motion planning in artificial intelligence and is an attempt to apply machine learning in the field of autonomous robotics.

The maze in the micromouse competition is typically made up of 16x16 grid cells with walls 50mm high. The robot is given multiple runs around the maze. The first run is for exploration where the robot tries to map out the maze, find the center, and learn the best path to the center. Then in the subsequent runs the robot is expected to reach the center in the fastest time possible, using what it has learned from the first run. The first micro mouse competition was held in 1979 in New York where the robot mouse was intended to find its way out of a 10x10 maze and the winner was a wall follower robot. In recent years the performance of the robots has increased significantly and they are now among the highest performing autonomous robots.

The mazes to be used in this project is provided on the Udacity Project Page in the form of text files along with some starter code files. The source of the files is provided through this link. These are the samples mazes provided by Udacity to test the robot.

Problem Statement

In simple words, the task is to plot an optimal path from the corner of a virtual maze to the center of the maze. To be precise, the goal of the project is to make an autonomous virtual robot which can gain information about the structure of a maze and then learn to make decisions which lead to a goal in the fastest time possible. The robot should be able to keep the track of its position in the maze, discover walls, dead ends, block the dead ends & detect the goal. The robot must map out the maze and discover the possible paths to a goal, then it should utilize its learning to find the optimal path for reaching the goal in shortest time.

The robot will make two runs in each maze to complete the task. The first run is for exploration where the robot will gain information about the structure of the maze, learn all the possible routes to the goal, and find an optimal path to the goal. It will try to explore as much unexplored area as it can. The robot must reach the goal in order to register a successful *run* around the maze. It should also continue to explore the maze after reaching the goal if it hasn't explored enough maze area. In the second run, the robot will resume from the starting position and attempt to follow the optimal path & reach the goal in fastest time possible using what it has learned in the first run.

Metrics

The robot's performance score will be calculated by adding 'no of time steps taken in the second run' & 'one-thirtieth the no of time steps taken in the first run'. Our goal is to minimize this score.

Score = (no. of steps in trial 2) + (no. of steps in trial 1) / 30

The score takes both exploration and optimization into consideration with optimization having more weight in the score. There is a trade-off between the exploration and optimization as the second run costs a complete step count for each step while the trial run costs 1/30th of a step count for each step. For example, if in the first run the robot takes 600 steps to explore the maze and then 20 steps to reach the goal in the second run, then the score will be 20 +600/30 = 40. Also, the robot is given a maximum of 1000 time steps to complete a run.

So our goal is to explore the maze as much as we can in the first run in order to minimize the cost for the second run. Another thing to consider here is that while navigating the maze in the first run the robot should avoid multiple visits to the already visited cells along with avoiding blocked cells, so as to minimize the cost in the trial run too.

II. Analysis

Data Exploration

For this project, Udacity provided the starter code files and the maze files. The archive includes the following files:

- **robot.py** - This script establishes the robot class. This is the script that we will be modifying, and is the main script for this project to be submitted.
- **maze.py** - This script contains functions for constructing the maze and for checking for walls upon robot movement or sensing. This is the file which will provide the sensory inputs required at each step.
- **tester.py** - This script will be run to test the robot's ability to navigate mazes.

- **showmaze.py** - This script can be used to create a visual demonstration of what a maze looks like.
- **test_maze_##.txt**- These files provide three sample mazes upon which we will test our robot. The ## is replaced by 01, 02 & 03 for three sample mazes.

The robot can rotate 90 degrees left and right, and can move a maximum of three steps at a time. The movement and rotation are assumed to be perfectly accurate. If the robot try to move into a wall, it stays where it is.

Each maze is an *n* x *n* square grid where n is even. The minimum value for n is 12 and maximum is 16. The walls in the maze restrict all movements. The walls are along the entire parameter of the maze and along some of the connected edges of the squares in the maze. The robot will start from bottom left corner facing upwards; the square in the starting position will have walls on the left, right and back allowing a forward movement only. The goal will be at the center of the grid consisting of a 2 x 2 square. The robot will be tested on the provided 3 mazes one by one.

The contents of a maze file looks like this:

```
12
1,5,7,5,5,5,7,5,7,5,5,6
3,5,14,3,7,5,15,4,9,5,7,12
11,6,10,10,9,7,13,6,3,5,13,4
10,9,13,12,3,13,5,12,9,5,7,6
9,5,6,3,15,5,5,7,7,4,10,10
3,5,15,14,10,3,6,10,11,6,10,10
9,7,12,11,12,9,14,9,14,11,13,14
3,13,5,12,2,3,13,6,9,14,3,14
11,4,1,7,15,13,7,13,6,9,14,10
11,5,6,10,9,7,13,5,15,7,14,8
11,5,12,10,2,9,5,6,10,8,9,6
9,5,5,13,13,5,5,12,9,5,5,12
```

These are the contents from the first test maze. Each maze file’s first line contains a number which is the dimension of the maze. Our maze files have 12, 14 and 16 in its first line, thus the dimensions of the mazes are 12 x 12, 14 x 14 & 16 x 16. Then follows the comma-delimited numbers in the file, which tells us the position of the walls in each square thus describing the open square edges for movement. The numbers are in range of 0 to 15. These numbers are four bit representations where the 1s represent the front/upward side, 2s represent the right side, 4s represent the back/downward side and 8s represent the left side. The bit value of 0 represents a wall and 1 represents an open edge. For example, number 9’s four bit representation is 1001. Thus 9 number tells us that the front and the left side is open for movement while there is a wall at the back and right.

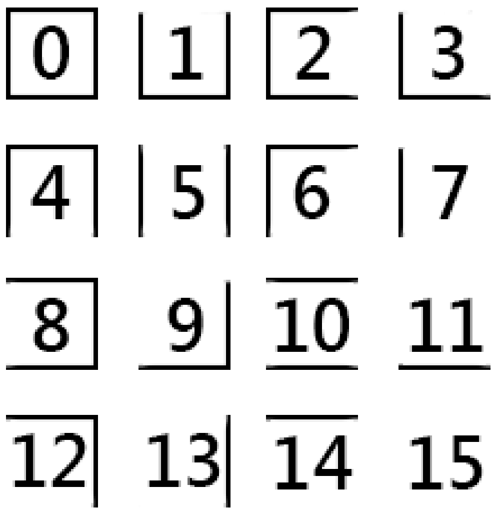
Our robot has no information about these numbers at the beginning and is completely unaware of its environment. The **maze.py** file provided to us helps the robot in studying and creating its environment i.e. the maze it is in. The robot has sensors on its three sides; left, front and right. These sensors detect open spaces and return a distance to the wall from the current location in the corresponding directions. The distance is more than 0 if space is open and 0 if closed i.e.

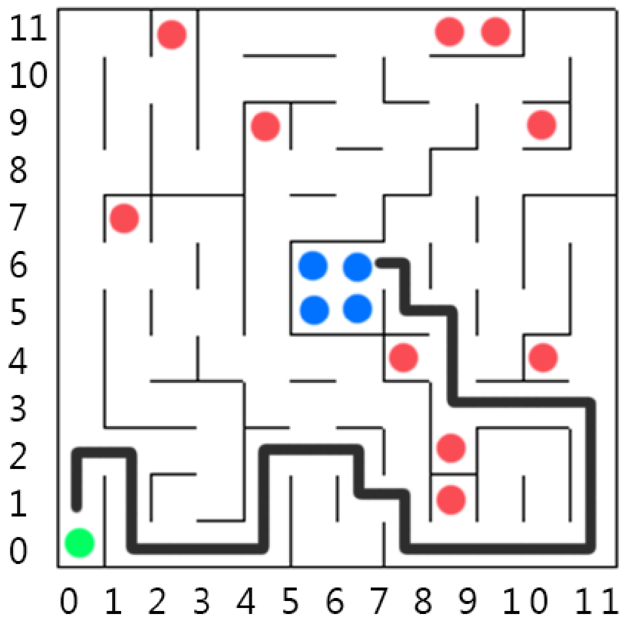
there is a wall in the current cell in that direction. The main file **robot.py** will receive these sensor values in the form of a list as an input. Then our robot will build the map of the maze using these sensor inputs in the first run. This learned map of the maze will help the robot to find the optimal path to the goal in the second run.

Exploratory Visualization

A visualization of the walls corresponding to the numbers between 0 – 15 is shown on left in the figures below.

On right is the visualization of the first maze mentioned in the previous section. The location of the robot is defined by the value of two point coordinates (X, Y) & it is considered to be at the center of a square. The green dot (0, 0) is the starting location and the blue dots represent the goal. The path shown in the figure is the optimal path to reach the goal. There are some dead ends in the maze represented by the red dots.





As mentioned in the previous section, the robot will read sensor values and calculate the numeric representation of the walls on its current location. Then to proceed to next location the robot will perform a rotation and a movement. The robot will block the dead ends whenever it encounters one and it will also block the path leading to dead ends. It will also prefer to proceed to unexplored cells. The robot must also prefer a straight path over a path having many turns if both the paths require same steps; as it allows the robot to take maximum no of steps allowed in a straight line i.e. 3 in single time step & hence minimizes the total step count. In this particular example, the path shown takes 17 steps to reach the goal, however total no of steps are 30.

Algorithms & Techniques

1. Exploration (first run)

To conduct a search for the shortest path, we need to know the maze first. Our maze here is completely unknown to the robot at the beginning and robot cannot make decisions in an unknown environment. So, the first task will be to simultaneously explore and learn the environment. The robot will accomplish this by using the sensor inputs, as the very first thing which the robot will do is receive inputs from its three sensors. Now let’s look at some important points to consider in order to successfully conduct our exploration.

- The robot must keep track of its location and heading
- The robot must be able to draw the map of the maze with the wall locations
- The robot must prefer going towards unexplored cells
- The robot should be able to detect the goal
- The robot should be able to detect, block and avoid dead ends
- The robot must explore enough maze to conduct the shortest path search

The algorithm I will use to perform the exploration task will be a modified **random movement algorithm**. The robot will pick random directions to move at each step and will reverse from the dead ends after blocking them. The pseudo code is as follows:

```
random_move(sensor value):
    -   calculate wall value for current location using sensor values
    -   update the maze map for the location
    -   if goal has been reached:
            set goal_found = True
    -   if current location is a dead end:
            block dead ends and reverse
    -   if current location is not a dead end:
            make a list of valid actions avoiding walls and dead ends
            if list is empty:
                robot heading to a blocked cell, reverse
            else:
                select a random action from the list
                if new location based on selected action is not explored:
                    perform rotation based on the action, and update heading and location
                elif new location is explored:
                    select an action leading to unexplored cell
                    if all actions in list lead to an explored cell, keep the previous action
                    perform rotation based on the action, and update heading and location
    -   check if goal_found is True and enough maze has been explored
            Reset
```

Now, after the end of first run our robot knows the maze. The maze map has been stored in a 2D grid named **maze_map**, which has the wall values for the explored cells along with blocked dead ends.

2. Optimization (second run)

For the second run we will use a bunch of path finding algorithms on the known maze now. The algorithms we will be using require a prior knowledge of the maze. Since we have mapped the maze in the first run, it is expected that these algorithms will work well.

The approach here is that, having a map of the maze with blocked cells the robot can apply a search algorithm to find the goal and define an optimal path to the goal. It will then create a policy and follow that policy to reach the goal. So at the start of the second run, first a policy will be created by applying a path search algorithm on the maze map and then the robot will begin to move following the policy. The 2D grids **path_grid, policy_grid** & **action_grid** will be used in the second run to keep track of the actions taken in each cell, to create a policy and to create an action grid for path visualization. Now let’s discuss the algorithms to be used.

1) Breadth First Search

This is the most common graph traversing algorithm. This algorithm traverses a tree or a graph starting from the root node and exploring all the neighbor nodes first before moving to the next level neighbors. The pseudo code is as follows:

```
breadth_first_search()

 -   queue = [start]
 -   parent = {}
 -   while goal is not found :
         node = queue.pop(0)
         check for valid actions for node extension
         for all valid actions:
             extend current location
             if extended location not visited and not blocked:
                 if goal is reached:
                     goal found = true
                     create a policy using parent dictionary and path grid
                 else:
                     update queue, path_grid & parent dictionary
```

2) Depth First Search

This algorithm is a recursive algorithm which uses backtracking, i.e. moving forward until there are no more nodes along the current path and then go backwards and find a new node to traverse. The pseudo code is as follows:

```
depth_first_search()

 -   stack = [start]
 -   parent = {}
 -   while goal is not found :
         node = stack.pop()
         check for valid actions for node extension
         for all valid actions:
             extend current location
             if extended location not visited and not blocked:
                 if goal is reached:
                     goal found = true
                     create a policy using parent dictionary and path grid
                 else:
                     update stack, path_grid & parent dictionary
```

3) Dijkstra Algorithm

Dijkstra algorithm determines shortest paths between nodes in a graph. This algorithm continuously calculates the shortest distances and excludes longer distances. The Dijkstra algorithm has many variants. The most common variant fixes a single node as the source node and finds shortest paths from that source node to all other nodes in the graph, producing a shortest-path tree. It starts from the source node and marks all direct neighbors of the source node with the cost to get there. Then it proceeds from the node with the lowest cost to all of its adjacent nodes and again marks the node with the lowest cost. Another variant of dijkstra is called 'uniform cost search' where the algorithm stops when the goal is reached.

The algorithm starts with assigning a prior distance to all the cells and then proceed to improve. A 2D grid named **dist** will be used for this purpose, and the distance to all cells will be set to infinity. This algorithm will work well for our problem as it considers the cost to reach a cell and our goal is certainly to minimize this cost. The pseudo code is as follows:

```
djikstra_algorithm()

    set all cell distances to infinity

    set the initial cell distance to zero

    cost for starting location, g = 0
cost = 1
queue = [(g, heading, start)]
while goal is not found:
    sort the queue as per the cost
    pop the node with the lowest cost to extend first
    check for valid actions for the current location in the node
    for all valid actions:
        extend the current location
        if extended location is not blocked and unvisited:
            if goal is found:
                create the policy
            else:
                new cost = cost of current location + cost
                update the dist grid with new cost for extended location
                extend the queue and update path grid
```

4) A Star Algorithm

A* is a very popular, flexible and fast path finding algorithm. It solves problems by searching among all possible paths to the goal for the one which has the minimum cost. A* combines pieces of djikstra algorithm with the heuristics as a guide. At each iteration of its main loop, A* determines which of its partial paths to expand into one or more longer paths. It is done on an estimate of the cost denoted by f(n) and selects the path that minimizes, $f(n) = g(n) + h(n)$

where n = the last node on the path,
g(n) = cost of the path from the start node to nth node
h(n) = a heuristic that estimates the cost of the cheapest path from nth node to the goal

There are three types of heuristics used in A* i.e. Manhattan distance, Diagonal distance and Euclidean distance. Since our problem deals with discrete locations and there are four possible directions to move, we will use Manhattan distance heuristics. Manhattan heuristics is the sum of absolute value of differences between the x & y coordinates of a location and the goal i.e.

$$H = abs(current.location.x – goal.x) + abs(current.location.y – goal.y)$$

The pseudo code for A* is as follows:

```
A_star_algorithm()
    generate the heuristics h
    cost = 1
    g for starting location = 0
    f = g + h(starting position)
    open = [(f, g, h, heading, start)]
    while goal is not found:
        sort the open list as per the f value
        pop the node with the lowest f value to extend first
        check for valid actions for current location in the node
        for all valid actions:
            extend the current location
            if extended location is not blocked and unvisited:
                if goal is found:
                    create the policy
                else:
                    g2 = current g + cost
                    h2 = h(extended_location)
                    f2 = g2 + h2
                    extend the open list and update path grid
```

5) Dynamic Programming & Flood Fill

In general, flood fill algorithm determines the area connected to a given node in a multidimensional array. The same algorithm is used in bucket fill tool of paint programs which fills the connected and similarly-colored pixels or area with a new same color. There are many ways in which the flood-fill algorithm can be structured. For this problem, the algorithm will start from the goal location and track back to the starting location giving each surrounding cell a number along the path. The number will represent the minimum steps required to reach the goal from that location. We will use dynamic programming approach for this algorithm.

While studying '*Udacity's Artificial Intelligence for Robotics Course*' to prepare for this project, I came across dynamic programming and found it really interesting and powerful. Dynamic programming is a technique which returns an optimal path to reach the goal from any location. Given a map and the goal location, dynamic programming creates the best policy to follow from every position in the map. The algorithm will use a value function to calculate a value (the shortest distance to the goal) for each location in a recursive fashion by considering the best neighbor cell, and adding its value and the cost to move to next cell.

$$V(x, y) = min(V(x', y')) + cost$$

To begin with, the value grid will be initiated with a large value for each cell and then the value function is applied recursively until each cell has a minimum value for reaching the goal.

Let's take a look at the pseudo code for better understanding:

```
value_function()
    Initialize the value grid with a large value for all cells, say 99
    value_fun()
      change = true
      cost = 1
      while change is true:
        change = false
        for each location starting from the initial position:
          if goal is reached:
            if v(goal) > 0
              v(goal) = 0
              Change = True
          else:
            check for valid actions
            for all valid actions:
              extend current location
              v2 = v(extended_location) + cost
              if v2 < v(current_location):
                change = true
                v(current_location) = v
```

The algorithm begins with the starting location and under the ‘for loop’, the first change will happen when the iteration reaches the goal location. Since we have initialized all the cell values as 99, no update will be lesser than 99 until the loop reaches the goal; and then a value of zero is given to the goal location. Then the algorithm enters the second iteration of while loop and then under the for loop, the next change will happen when the iteration reaches the location adjacent to the goal and a value of 1 is given to those positions. The algorithm will always enter the while loop until there is a change and the while loop ends when the starting position has been reached and assigned a value. This way dynamic programming creates a policy for each cell by going back from the goal assigning best values to the adjacent positions.

Benchmark

For a benchmark model we are going to use Breadth First Search Algorithm. First, the random movement algorithm will map the maze. Then in the second run, Breadth-First Search Algorithm will find the optimal path to the goal. Each run should be completed in a maximum of 1000 time steps. The cost of a step in the second run will be 30 times more than in the first run. The performance score is defined by the following metric:

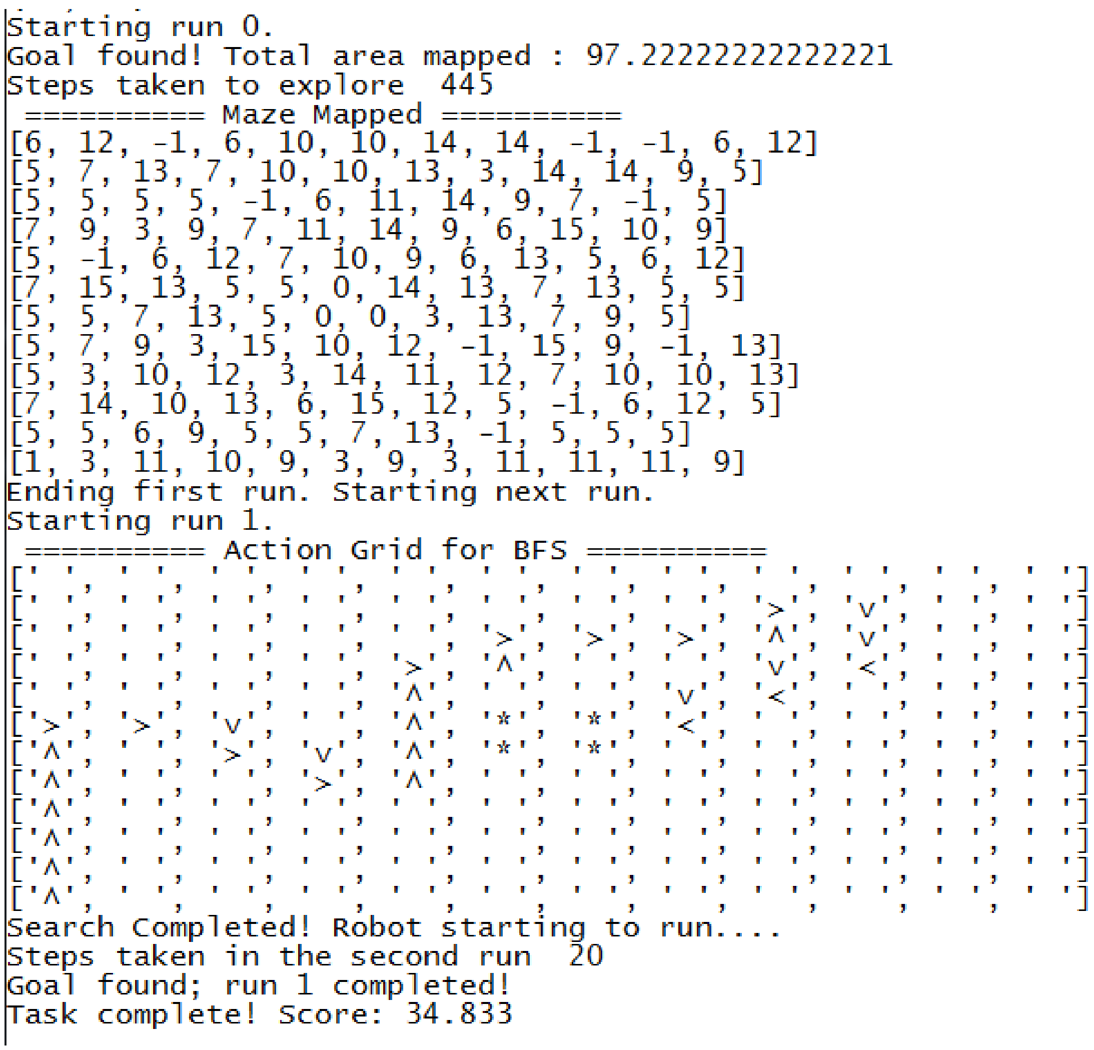
Benchmark Score = (no. of steps in trial 2) + (no. of steps in trial 1) / 30

We will also test the model on two area thresholds. A larger area coverage will take more steps and lower area coverage will take less. We need to see how this affects the robot’s performance.

Maze	Area	Steps in first run	Steps in second run	Score
Maze 1	97.22	445	20	34.883
	70.1	176	17	22.867
Maze 2	90.3	729	27	51.3
	70.4	228	27	34.6
Maze 3	90.23	559	25	43.633
	70.3	318	28	38.6

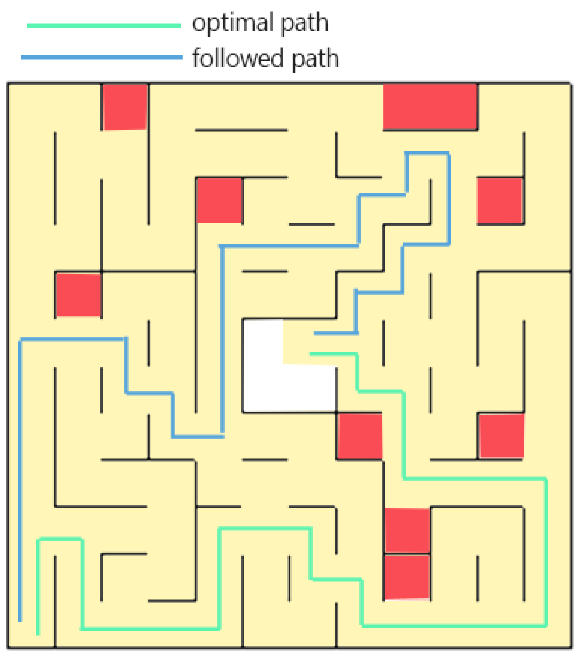
It’s evident that the threshold will definitely affect the score. So we will be looking for an optimum value for our threshold. A better model should be able to explore enough in the first run, find the best optimal path always in the second run and get a better score.

The next important point to discuss for our benchmark model is that, even though the robot has managed to explore all the cells of the best possible path, yet it is not recognizing that as the optimal path. Let’s take a look a particular example of the first case in above table to discuss this further:



```
Starting run 0.
Goal found! Total area mapped : 97.2222222222221
Steps taken to explore  445
 ========== Maze Mapped ==========
[6, 12, -1, 6, 10, 10, 14, 14, -1, -1, 6, 12]
[5, 7, 13, 7, 10, 10, 13, 3, 14, 14, 9, 5]
[5, 5, 5, 5, -1, 6, 11, 14, 9, 7, -1, 5]
[7, 9, 3, 9, 7, 11, 14, 9, 6, 15, 10, 9]
[5, -1, 6, 12, 7, 10, 9, 6, 13, 5, 6, 12]
[7, 15, 13, 5, 5, 0, 14, 13, 7, 13, 5, 5]
[5, 5, 7, 13, 5, 0, 0, 3, 13, 7, 9, 5]
[5, 7, 9, 3, 15, 10, 12, -1, 15, 9, -1, 13]
[5, 3, 10, 12, 3, 14, 11, 12, 7, 10, 10, 13]
[7, 14, 10, 13, 6, 15, 12, 5, -1, 6, 12, 5]
[5, 5, 6, 9, 5, 5, 7, 13, -1, 5, 5, 5]
[1, 3, 11, 10, 9, 3, 9, 3, 11, 11, 11, 9]
Ending first run. Starting next run.
Starting run 1.
 ========== Action Grid for BFS ==========
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '>', 'v', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', '>', '>', '>', '^', 'v', ' ', ' ']
[' ', ' ', ' ', ' ', '>', '^', ' ', ' ', 'v', '<', ' ', ' ']
[' ', ' ', ' ', ' ', '^', ' ', ' ', 'v', '<', ' ', ' ', ' ']
['>', '>', 'v', ' ', '^', '*', '*', '<', ' ', ' ', ' ', ' ']
['^', ' ', '>', 'v', '^', '*', '*', ' ', ' ', ' ', ' ', ' ']
['^', ' ', ' ', '>', '^', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
['^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
['^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
['^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
['^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
Search Completed! Robot starting to run....
Steps taken in the second run  20
Goal found; run 1 completed!
Task complete! Score: 34.833
```

The results above are for the first maze using breadth first search algorithm for optimization. It shows the map of maze_1 our robot has after its exploratory run. -1 indicates a blocked cell, 0 indicates an unexplored cell and rest are the numbers representing the walls. A proper visualization of this mapped maze is shown on right. Light yellow cells are explored, red are blocked and whites are unexplored. We can see that the robot has explored almost all the cells (97.2 %) in the maze, except for three cells inside the goal area as it stopped after just entering the first goal cell. The action grid below the mapped maze shows the policy taken by the robot to reach the goal. The figure on right also shows the optimal and followed path for this maze. We can see that even after mapping every cell, breadth first search did not follow the best path available. So we will also be looking to solve this issue in our final model. Finally all things considered, I am expecting an average score of 25, 36 and 42 for maze_1, maze_2 & maze_3.



III. Methodology

Data Preprocessing

Data preprocessing is not needed for this project. The mazes have been provided to us by Udacity and the robot reads the mazes using sensors which are accurate. The movement and rotation are also assumed to be accurate.

Implementation

The process involves three files mainly: ***tester.py, maze.py and robot.py***. The main code for the robot is written in ***robot.py*** and this is the file meant for submission and evaluation for this project. The other two files have been provided to us. The ***tester.py*** is for running and testing the robot, while the ***maze.py*** file is used for sensing the environment.

The basic flow is as follows:

1. The ***tester.py*** file first sense the environment for current location using the ***maze.py*** file
2. The ***maze.py*** file returns a list of sensor inputs containing the distance of a wall from the corresponding sensor's direction.

3. The ***tester.py*** file then sends this list as an input to the ***next_move function*** of the ***robot.py*** file to receive the next movement and rotation

The next move function

In the ***next_move function*** first the sensor list will be converted to 1 & 0. 1 for open and 0 for closed. Then for the first run, the robot will enter the exploration phase; and for the second run, the robot will enter the optimization phase.

For the first run, the ***next_move function*** will call the ***explorer_robot function*** with the sensor list as the input parameter and receive the next rotation and movement. For the second run it will call the ***runner_robot function*** and receive the next rotation and movement. The ***next_move function*** will then return the next rotation and movement back to the tester.py file.

Before we talk about the exploration and optimization phase, let's first define some basic functions and variables required to carry out some supportive tasks.

Basic Functionalities

In this section we'll first discuss implementing the logic for giving a basic intelligence to the robot.

1) Global Dictionaries
   The global dictionaries provide the robot a logical understanding for making proper actions.
   - *dir_sensor*s : for correctly interpreting sensor information according to the robot's heading
   - *dir_move* : for applying left, up, right and downward movement
   - *dir_reverse* : for movement in reverse direction based on robot's heading
   - *dir_rotation* : for proper rotation based on robot's heading
   - *dir_delta* : dictionary for using action symbols for right, left, up and down. Used in the visualization of an action grid

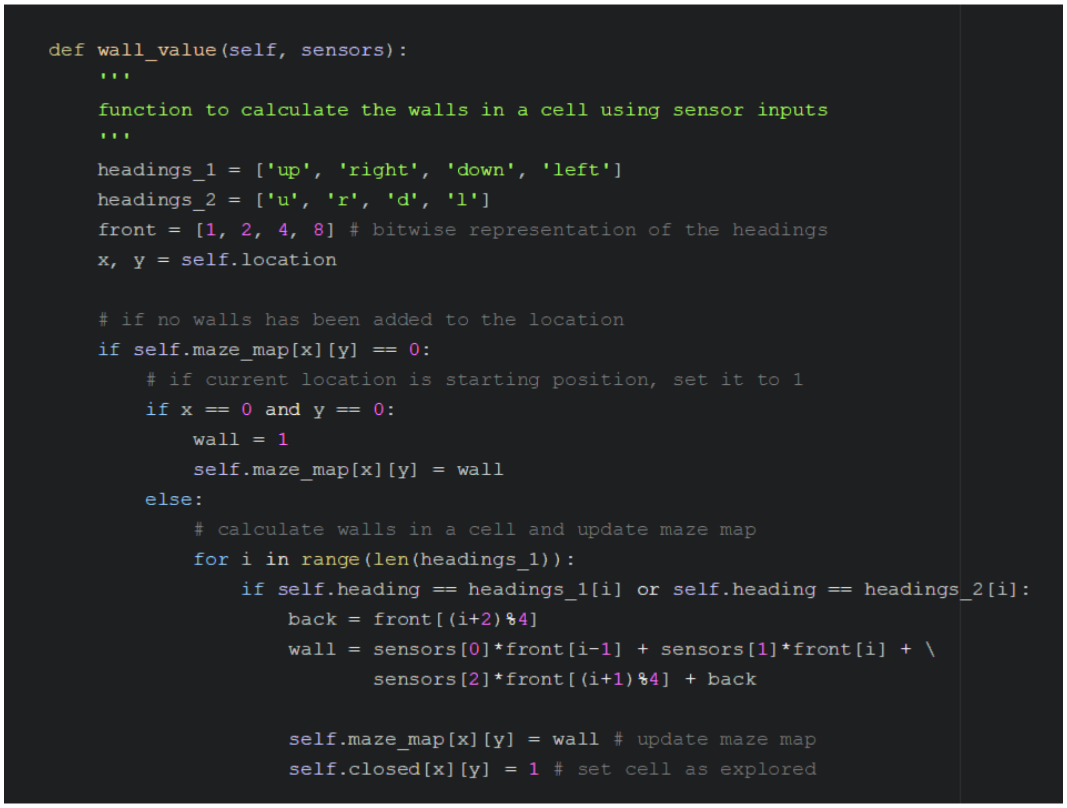
2) Supportive Attributes
   - *closed grid* : keeping track of mapped cells in exploration
   - *explored grid* : similar to closed grid, except used in second run to keep track of visited locations
   - *maze grid* : for creating and saving the map of the maze
   - *path grid* : to keep track of actions performed in each cell. Used to track the path followed to any location
   - *policy grid* : for saving the optimal policy. Used in the optimization phase
   - *action grid* : policy grid with action symbols for path visualization
   - *dist grid* : distance grid used to carry out dijkstra algorithm
   - *heuristics grid* : heuristics grid for carrying out A star search
   - *value grid* : grid for value function in dynamic programming
   - *new grid* : placeholder grid for print a proper view of any other grid in the shell

- *goal bounds* : goal boundary
- *steps* : step counts
- *search* : set to true and false as per the algorithm's need
- *goal found* : set to true and false for goal found or not

3) Determining the walls inside a cell

To determine the walls surrounding a cell, the 'sensor list' for a location will be sent to the *wall_value* function which will calculate the number representing the walls in that location. This function will also assist in updating the maze_map grid creating a map of the maze. The logic applied to calculate the wall value is that the 1s represent the upward side, 2s represent the right side, 4s represent the back side and 8s represent the left side.

A 2D maze_map grid is used to create the map of the maze as the robot explores, another 2D grid named closed is used to keep track of visited cells. The closed grid will also be used to calculate the area covered and also for preference of movements to unexplored cells. The code applied is shown below:

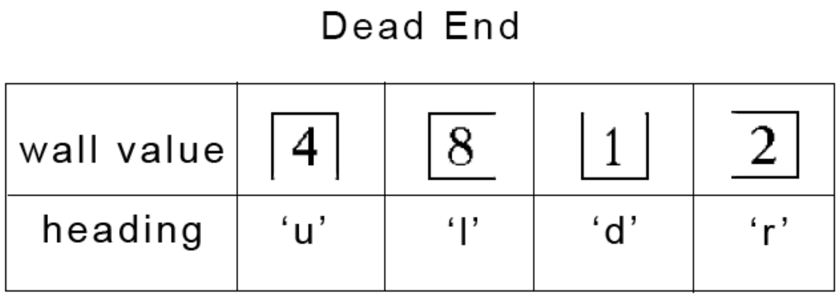


```
def wall_value(self, sensors):
    '''
    function to calculate the walls in a cell using sensor inputs
    '''
    headings_1 = ['up', 'right', 'down', 'left']
    headings_2 = ['u', 'r', 'd', 'l']
    front = [1, 2, 4, 8] # bitwise representation of the headings
    x, y = self.location

    # if no walls has been added to the location
    if self.maze_map[x][y] == 0:
        # if current location is starting position, set it to 1
        if x == 0 and y == 0:
            wall = 1
            self.maze_map[x][y] = wall
        else:
            # calculate walls in a cell and update maze map
            for i in range(len(headings_1)):
                if self.heading == headings_1[i] or self.heading == headings_2[i]:
                    back = front[(i+2)%4]
                    wall = sensors[0]*front[i-1] + sensors[1]*front[i] + \
                           sensors[2]*front[(i+1)%4] + back

                    self.maze_map[x][y] = wall # update maze map
                    self.closed[x][y] = 1 # set cell as explored
```

4) Blocking dead ends

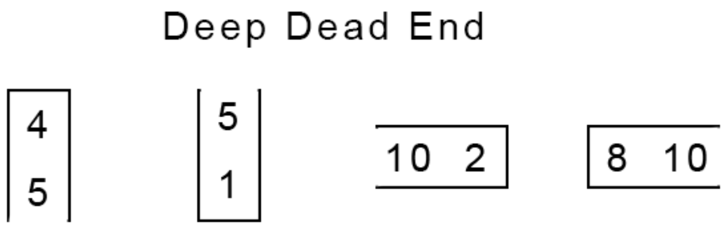
While exploring the maze the robot can run into dead ends time and again which could cost us an increase in the score. So to prevent this we can block the dead ends whenever it is detected.

The logic to be applied here is that when the robot enters a new cell, all three sensor values will be 0 for a dead end or; for each heading a particular wall value will be a dead end. In addition, we must also take care of the previous cell for which there exists only one valid action which can lead to the dead end. For example, a robot heading ‘up’ in a cell of wall value ‘5’ may reach a cell with a wall value of ‘4’. The figure below describes the conditions for a dead end:



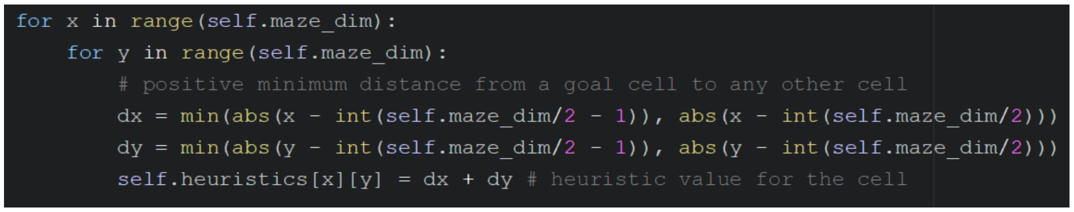
Dead End

wall value	4	8	1	2
heading	‘u’	‘l’	‘d’	‘r’



The cells meeting these conditions will be blocked by putting a -1 in those locations in our maze_map grid. And whenever we run into a dead end the robot will reverse and move to another cell

5) <u>Other functions:</u>
Additionally three other functions named ***actions***, ***area_cov*** and ***heuristic_func*** are also provided.
- The function ***actions*** receives the current location and then picks out the wall value for that location from the maze map , then it returns a list of valid actions based on the walls surrounding the robot in that location.
- The ***area_cov*** function returns the maze area covered using the closed grid which keeps a track of cells explored.
- The ***heuristic_func function*** is called during A* search algorithm to create a heuristic grid for A*. The code for creating the heuristic grid is shown below:



```
for x in range(self.maze_dim):
    for y in range(self.maze_dim):
        # positive minimum distance from a goal cell to any other cell
        dx = min(abs(x - int(self.maze_dim/2 - 1)), abs(x - int(self.maze_dim/2)))
        dy = min(abs(y - int(self.maze_dim/2 - 1)), abs(y - int(self.maze_dim/2)))
        self.heuristics[x][y] = dx + dy # heuristic value for the cell
```

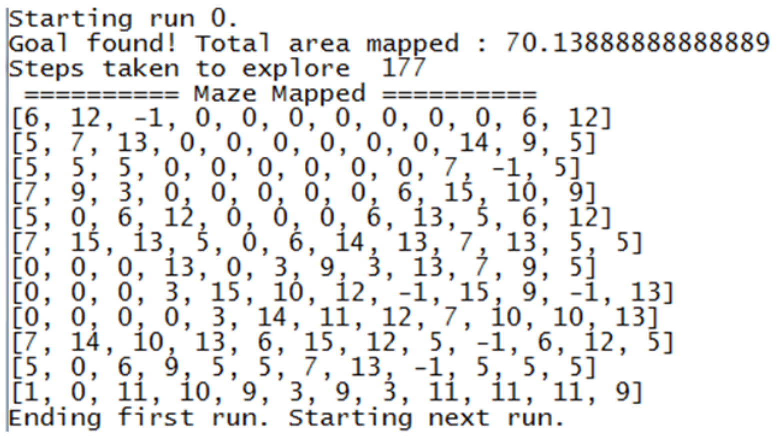
<u>Exploration</u>

The robot will enter the exploration phase in the first run where it will explore and map the maze using a modified random movement algorithm. The exploration will be carried out by the

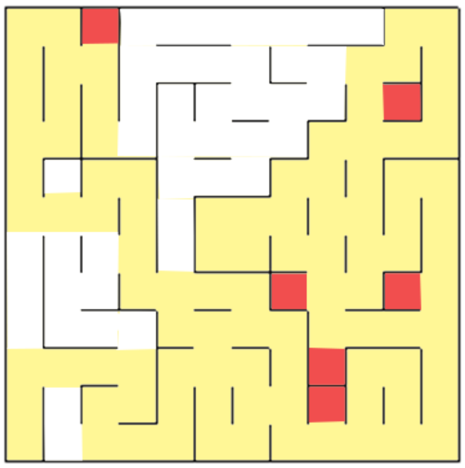
explorer_robot function. The important part of this phase is that the robot should explore enough area of the maze so as to discover the best path available in the maze and that too without taking too many steps. To accomplish this we will keep blocking the dead ends as we encounter them and give preference to unexplored cells.

Now, to begin the robot will first check for the walls in its current cell by calling the ***wall_value function***. The wall value function as explained in the previous section will return the walls of the current cell and update the maze map. Then the robot will check if it has encountered a dead end. If the robot has reached a dead end, it will reverse without any rotation. Then it will chose a different action leading to a new cell as we have just blocked the dead we encountered. The robot will keep making random movements until it has found the goal and reached the area threshold for exploration. With a preference given to unexplored cells the robot is expected to move faster in the maze without making multiple visits to a single cell. However, if there is no choice the robot can still move to the explored cell until it has found a new unexplored cell. This will also prevent getting stuck in a loop up to an extent. Every time the robot takes an action, its rotation is changed using the ***dir_rotation*** dictionary and the new heading is changed to the action taken. For example, for a heading of 'up' a rotation of -90 degree is applied if the action taken is left, 0 degree if the action is 'up' and +90 degree is it is right. Once the goal is found and the area threshold is reached, the robot will reset and enter the optimization phase.

This way the robot will learn about its environment and in the next run it will try to use this learning to accomplish the task it has been assigned. The maze mapped by the robot will be printed in the shell. In the figure below, the maze mapped by the robot is shown on left. The visualization of the mapped maze is shown on right. In this mapped maze grid -1 represent the blocked cells, 0 represent unexplored cells and the numbers represent the walls in the corresponding cells. On the visualization, red cells are blocked, yellow are explored and whites are unexplored.



```
Starting run 0.
Goal found! Total area mapped : 70.13888888888889
Steps taken to explore  177
 ========== Maze Mapped ==========
[6, 12, -1, 0, 0, 0, 0, 0, 0, 0, 6, 12]
[5, 7, 13, 0, 0, 0, 0, 0, 0, 14, 9, 5]
[5, 5, 5, 0, 0, 0, 0, 0, 0, 7, -1, 5]
[7, 9, 3, 0, 0, 0, 0, 0, 6, 15, 10, 9]
[5, 0, 6, 12, 0, 0, 0, 6, 13, 5, 6, 12]
[7, 15, 13, 5, 0, 6, 14, 13, 7, 13, 5, 5]
[0, 0, 0, 13, 0, 3, 9, 3, 13, 7, 9, 5]
[0, 0, 0, 3, 15, 10, 12, -1, 15, 9, -1, 13]
[0, 0, 0, 0, 3, 14, 11, 12, 7, 10, 10, 13]
[7, 14, 10, 13, 6, 15, 12, 5, -1, 6, 12, 5]
[5, 0, 6, 9, 5, 5, 7, 13, -1, 5, 5, 5]
[1, 0, 11, 10, 9, 3, 9, 3, 11, 11, 11, 9]
Ending first run. Starting next run.
```



Optimization

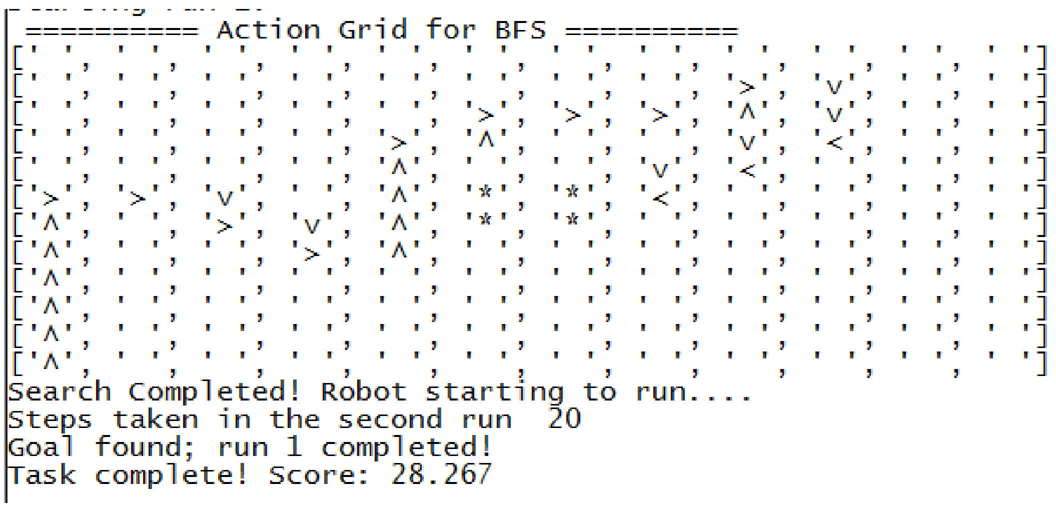
After exploration the robot now know the maze and has a map of it. In the optimization phase the robot will first apply a path search algorithm on the map it has created to find the shortest route to the goal and then create a policy to follow before it actually begins to move. The robot will then follow the policy to reach the goal.

The optimization is carried out by the ***runner_robot function***. At the beginning of this function we will choose an algorithm to perform a search which will create a policy. Then the robot will start moving following that policy. The algorithms available for us to apply are listed below with their function name:

1) Breadth first search : bfs()
2) Depth first search function : dfs()
3) Dijkstra : dijkstra()
4) A star : a_star
5) Dynamic Programming : value_function()

While following the policy the robot will move up to a maximum of three steps if the next three consecutive actions are the same, and two steps if two consecutive actions are same. Thus a preference will be given to a straight path. The policy created will be printed in the shell along with the no of steps taken as shown below:

Policy created by breadth first Search (bfs) algorithm



```
 ========== Action Grid for BFS ==========
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '>', 'v', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', '>', '>', '>', '^', 'v', ' ', ' ']
[' ', ' ', ' ', ' ', '>', '^', ' ', ' ', 'v', '<', ' ', ' ']
[' ', ' ', ' ', ' ', '^', ' ', ' ', 'v', '<', ' ', ' ', ' ']
['>', '>', 'v', ' ', '^', '*', '*', '<', ' ', ' ', ' ', ' ']
['^', ' ', '>', 'v', '^', '*', '*', ' ', ' ', ' ', ' ', ' ']
['^', ' ', ' ', '>', '^', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
['^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
['^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
['^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
['^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
Search Completed! Robot starting to run....
Steps taken in the second run  20
Goal found; run 1 completed!
Task complete! Score: 28.267
```

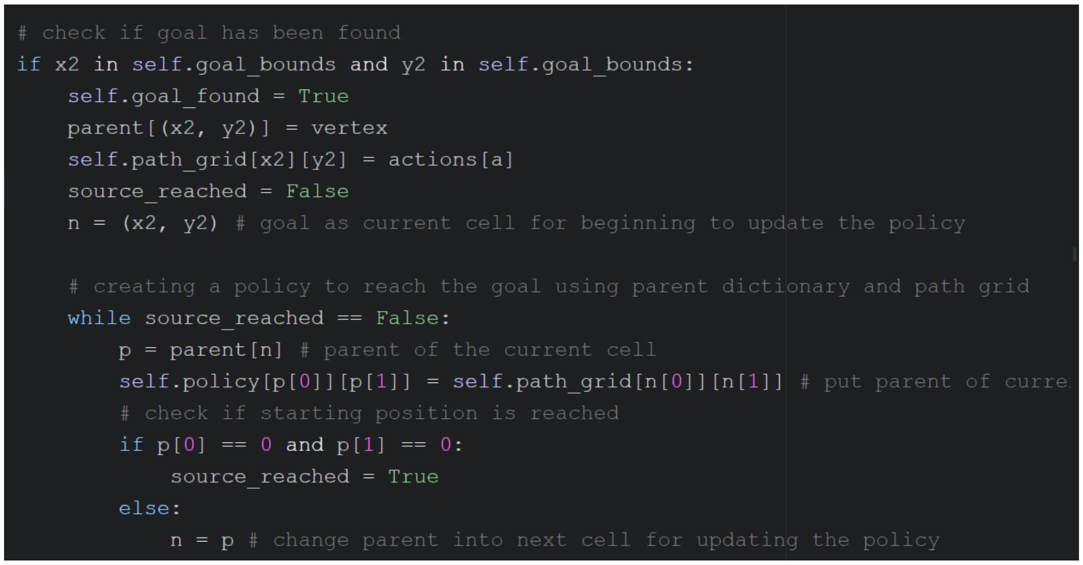
The '*' represent the goal area and the arrows represent the actions taken

As we discussed earlier too the challenge here will be to attain a balance between exploration and optimization, as the optimization phase will need a map which contains the best possible path. This will depend on the exploration phase and the total score will be affected significantly by total no of steps taken in the exploration in order to create a map containing the best path for each maze.

Let's now discuss how each algorithm creates the policy:

1) <u>Breadth first search & Depth First Search Algorithm</u>

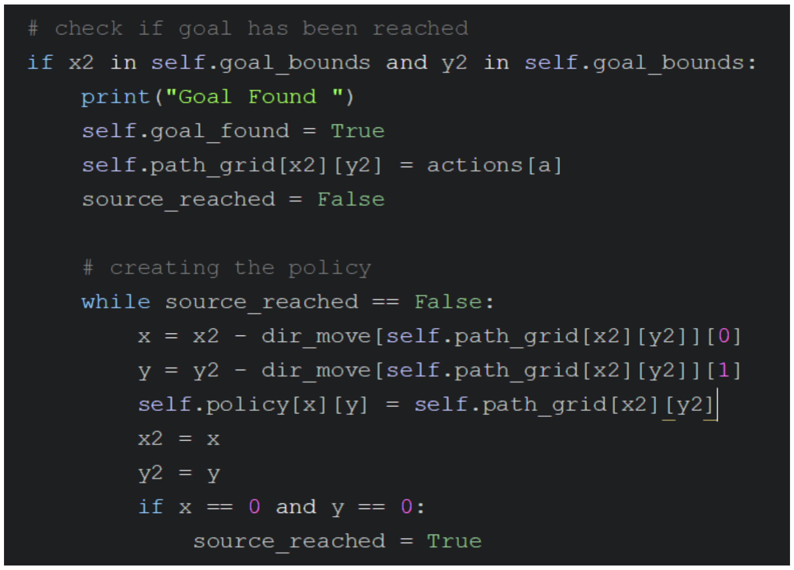
The breadth first search and depth first search algorithms maintain a dictionary of parent nodes for every extended node till the goal. After reaching the goal, the algorithm stops and backtracks to the starting position using the parent dictionary. It starts with the goal location and starts putting the action taken in the parent node of its current location into the policy grid using the path grid. It then set the parent node as the current node and repeat the process until it reaches back to the starting position. The code to create the policy is shown below:



```
# check if goal has been found
if x2 in self.goal_bounds and y2 in self.goal_bounds:
    self.goal_found = True
    parent[(x2, y2)] = vertex
    self.path_grid[x2][y2] = actions[a]
    source_reached = False
    n = (x2, y2) # goal as current cell for beginning to update the policy

    # creating a policy to reach the goal using parent dictionary and path grid
    while source_reached == False:
        p = parent[n] # parent of the current cell
        self.policy[p[0]][p[1]] = self.path_grid[n[0]][n[1]] # put parent of curre
        # check if starting position is reached
        if p[0] == 0 and p[1] == 0:
            source_reached = True
        else:
            n = p # change parent into next cell for updating the policy
```

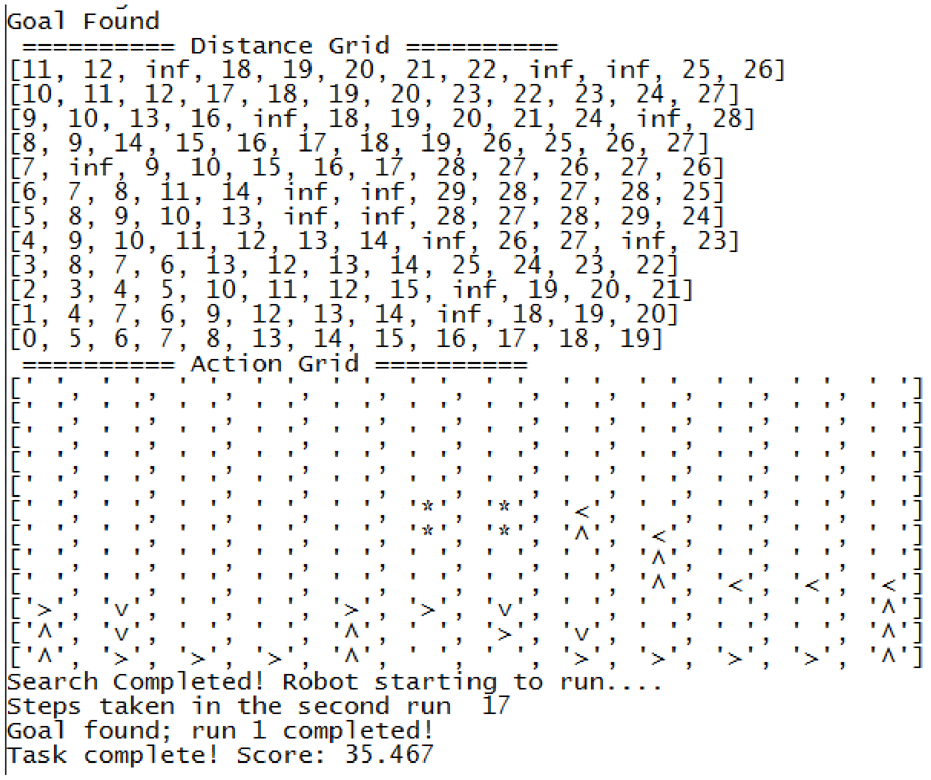
2) <u>Dijkstra Algorithm</u>

The variant of Dijkstra algorithm applied here will search the maze until it reaches the goal. After the goal is reached the path grid is used to backtrack to the starting location creating a policy. It starts from the goal location & moves back to the cell it came from by using the action taken in the current location. Then that action is put into the policy grid for the cell we just moved back to. Then that cell is set as current location and we move back again using the path grid until we reach the starting location creating out policy. The code to create the policy is shown below:



```
# check if goal has been reached
if x2 in self.goal_bounds and y2 in self.goal_bounds:
    print("Goal Found ")
    self.goal_found = True
    self.path_grid[x2][y2] = actions[a]
    source_reached = False

    # creating the policy
    while source_reached == False:
        x = x2 - dir_move[self.path_grid[x2][y2]][0]
        y = y2 - dir_move[self.path_grid[x2][y2]][1]
        self.policy[x][y] = self.path_grid[x2][y2]
        x2 = x
        y2 = y
        if x == 0 and y == 0:
            source_reached = True
```

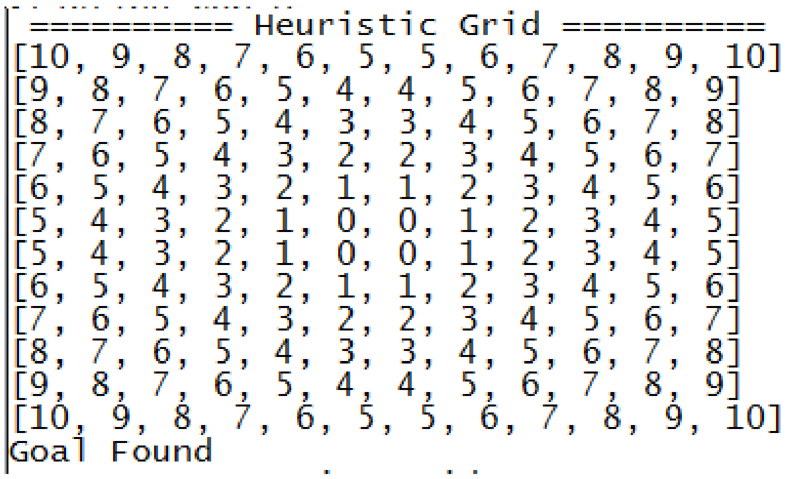
The distance grid used in dijkstra algorithm and the policy created will be printed in the shell as below:



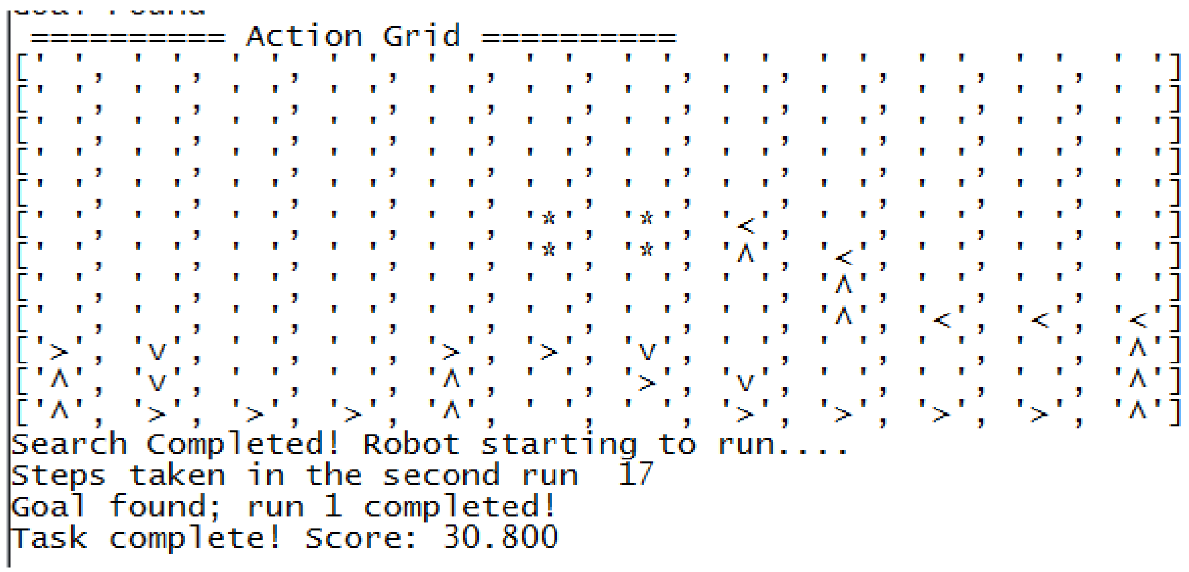
```
Goal Found
 ========== Distance Grid ==========
[11, 12, inf, 18, 19, 20, 21, 22, inf, inf, 25, 26]
[10, 11, 12, 17, 18, 19, 20, 23, 22, 23, 24, 27]
[9, 10, 13, 16, inf, 18, 19, 20, 21, 24, inf, 28]
[8, 9, 14, 15, 16, 17, 18, 19, 26, 25, 26, 27]
[7, inf, 9, 10, 15, 16, 17, 28, 27, 26, 27, 26]
[6, 7, 8, 11, 14, inf, inf, 29, 28, 27, 28, 25]
[5, 8, 9, 10, 13, inf, inf, 28, 27, 28, 29, 24]
[4, 9, 10, 11, 12, 13, 14, inf, 26, 27, inf, 23]
[3, 8, 7, 6, 13, 12, 13, 14, 25, 24, 23, 22]
[2, 3, 4, 5, 10, 11, 12, 15, inf, 19, 20, 21]
[1, 4, 7, 6, 9, 12, 13, 14, inf, 18, 19, 20]
[0, 5, 6, 7, 8, 13, 14, 15, 16, 17, 18, 19]
 ========== Action Grid ==========
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', '*', '*', '<', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', '*', '*', '^', '<', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', '<', '<', '<']
['>', 'v', ' ', ' ', '>', '>', 'v', ' ', ' ', ' ', ' ', '^']
['^', 'v', ' ', ' ', '^', ' ', '>', 'v', ' ', ' ', ' ', '^']
['^', '>', '>', '>', '^', ' ', ' ', '>', '>', '>', '>', '^']
Search Completed! Robot starting to run....
Steps taken in the second run  17
Goal found; run 1 completed!
Task complete! Score: 35.467
```

3) <u>A* Algorithm</u>

A star extends the dijkstra algorithm by adding a heuristic guide to strengthen its search. It also extends the nodes with the lowest cost. The details of the algorithm has been discussed already in the previous sections. And to create the policy, it will follow the same process as dijkstra, which is discussed above. The heuristics and the policy will be shown in the shell as below:



```
 ========== Heuristic Grid ==========
[10, 9, 8, 7, 6, 5, 5, 6, 7, 8, 9, 10]
[9, 8, 7, 6, 5, 4, 4, 5, 6, 7, 8, 9]
[8, 7, 6, 5, 4, 3, 3, 4, 5, 6, 7, 8]
[7, 6, 5, 4, 3, 2, 2, 3, 4, 5, 6, 7]
[6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6]
[5, 4, 3, 2, 1, 0, 0, 1, 2, 3, 4, 5]
[5, 4, 3, 2, 1, 0, 0, 1, 2, 3, 4, 5]
[6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6]
[7, 6, 5, 4, 3, 2, 2, 3, 4, 5, 6, 7]
[8, 7, 6, 5, 4, 3, 3, 4, 5, 6, 7, 8]
[9, 8, 7, 6, 5, 4, 4, 5, 6, 7, 8, 9]
[10, 9, 8, 7, 6, 5, 5, 6, 7, 8, 9, 10]
Goal Found
```



```
 ========== Action Grid ==========
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', '*', '*', '<', ' ', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', '*', '*', '^', '<', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', ' ', ' ', ' ']
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '^', '<', '<', '<']
['>', 'v', ' ', ' ', '>', '>', 'v', ' ', ' ', ' ', ' ', '^']
['^', 'v', ' ', ' ', '^', ' ', '>', 'v', ' ', ' ', ' ', '^']
['^', '>', '>', '>', '^', ' ', ' ', '>', '>', '>', '>', '^']
Search Completed! Robot starting to run....
Steps taken in the second run  17
Goal found; run 1 completed!
Task complete! Score: 30.800
```

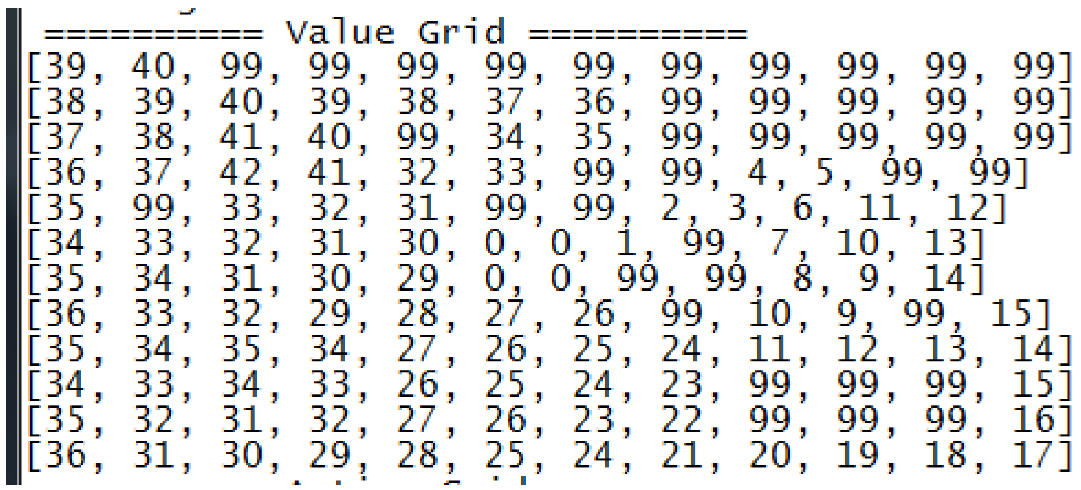
4) Dynamic Programming/Value Function

The details of the algorithm has been discussed already in the previous sections. Basically what it does is create a heuristic but by taking the walls into consideration, and then assign each cell a value which represents the no of steps that cell is away from the goal location. Then this value grid is used to create the policy.

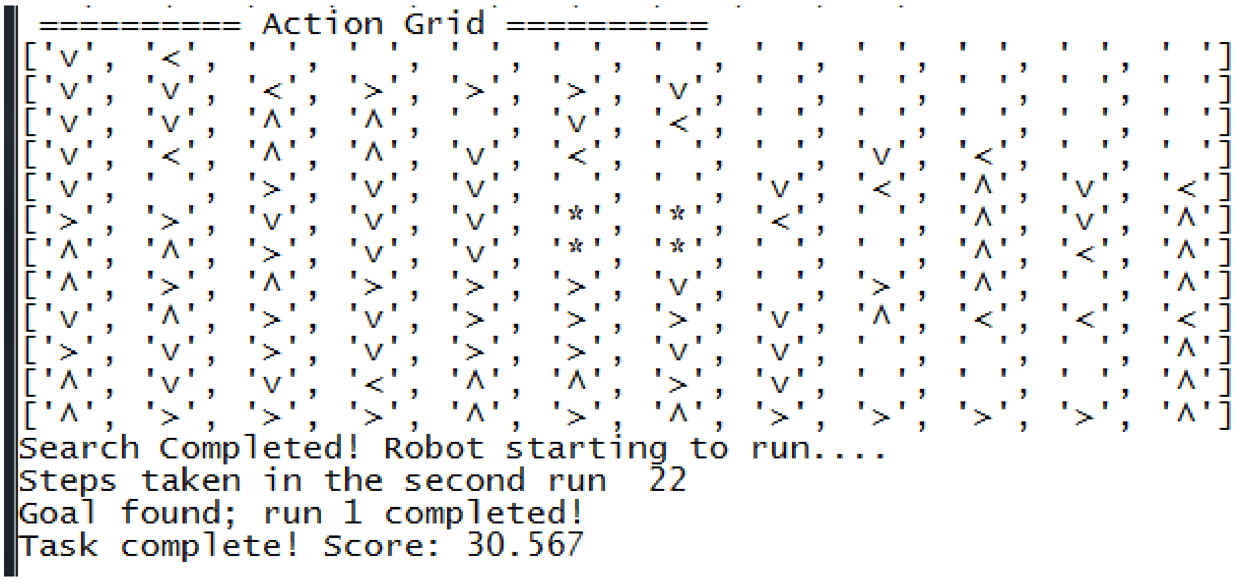
The algorithm will use a value function to calculate a value (the shortest distance to the goal) for each location in a recursive fashion by considering the best neighbor cell, and adding its value and the cost to move to next cell.

$$V(x, y) = min(V(x', y')) + cost$$

To begin with, the value grid will be initiated with a large value for each cell and then the value function is applied recursively until each cell has a minimum value for reaching the goal. The code for value function can be viewed in the robot.py file. The value grid and policy will be shown in the shell as below:



```
 ========== Value Grid ==========
[39, 40, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99]
[38, 39, 40, 39, 38, 37, 36, 99, 99, 99, 99, 99]
[37, 38, 41, 40, 99, 34, 35, 99, 99, 99, 99, 99]
[36, 37, 42, 41, 32, 33, 99, 99, 4, 5, 99, 99]
[35, 99, 33, 32, 31, 99, 99, 2, 3, 6, 11, 12]
[34, 33, 32, 31, 30, 0, 0, 1, 99, 7, 10, 13]
[35, 34, 31, 30, 29, 0, 0, 99, 99, 8, 9, 14]
[36, 33, 32, 29, 28, 27, 26, 99, 10, 9, 99, 15]
[35, 34, 35, 34, 27, 26, 25, 24, 11, 12, 13, 14]
[34, 33, 34, 33, 26, 25, 24, 23, 99, 99, 99, 15]
[35, 32, 31, 32, 27, 26, 23, 22, 99, 99, 99, 16]
[36, 31, 30, 29, 28, 25, 24, 21, 20, 19, 18, 17]
```



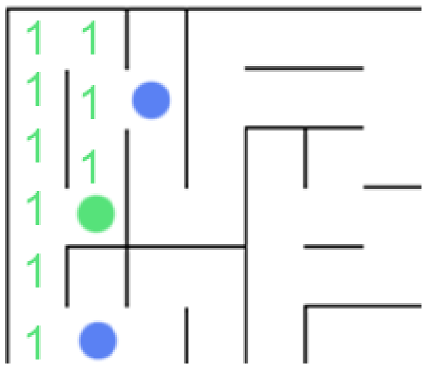
```
 ========== Action Grid ==========
['v', '<', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
['v', 'v', '<', '>', '>', '>', 'v', ' ', ' ', ' ', ' ', ' ']
['v', 'v', '^', '^', ' ', 'v', '<', ' ', ' ', ' ', ' ', ' ']
['v', '<', '^', '^', 'v', '<', ' ', ' ', 'v', '<', ' ', ' ']
['v', ' ', '>', 'v', 'v', ' ', ' ', 'v', '<', '^', 'v', '<']
['>', '>', 'v', 'v', 'v', '*', '*', '<', ' ', '^', 'v', '^']
['^', '^', '>', 'v', 'v', '*', '*', ' ', ' ', '^', '<', '^']
['^', '>', '^', '>', '>', '>', 'v', ' ', '>', '^', ' ', '^']
['v', '^', '>', 'v', '>', '>', '>', 'v', '^', '<', '<', '<']
['>', 'v', '>', 'v', '>', '>', 'v', 'v', ' ', ' ', ' ', '^']
['^', 'v', 'v', '<', '^', '^', '>', 'v', ' ', ' ', ' ', '^']
['^', '>', '>', '>', '^', '>', '^', '>', '>', '>', '>', '^']
Search Completed! Robot starting to run....
Steps taken in the second run  22
Goal found; run 1 completed!
Task complete! Score: 30.567
```

In action grid the cells which are blank are those which remain unexplored, thus no policy was created for them. In the value grid these cells are represented by the no 99.

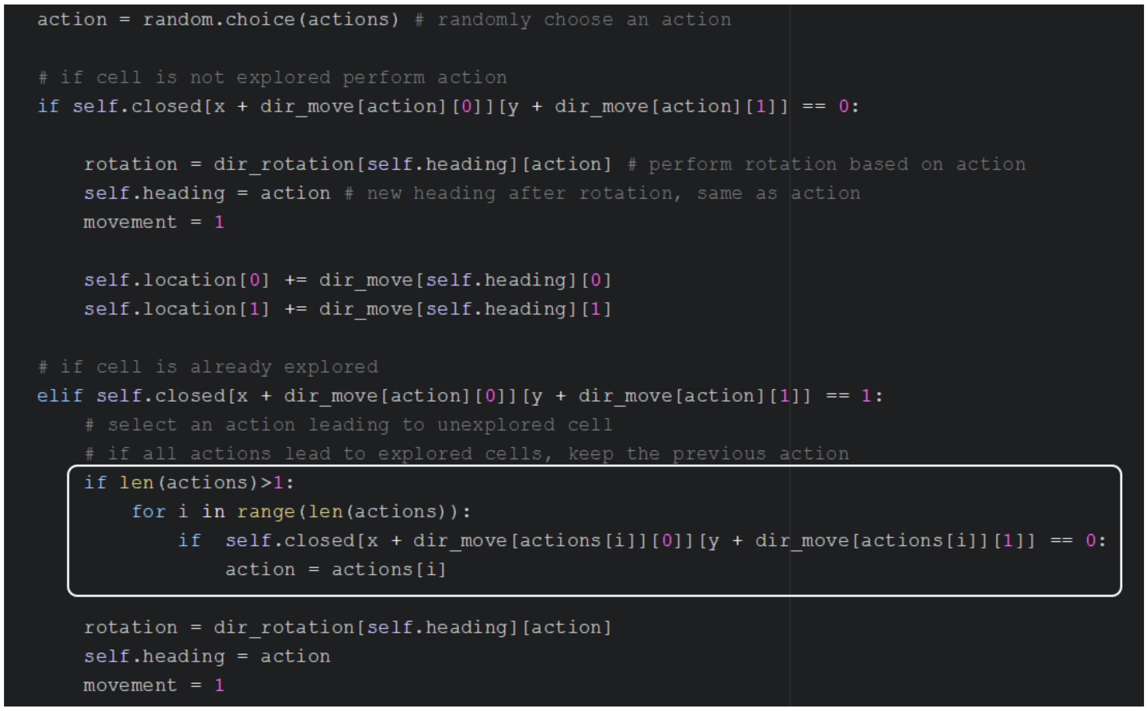
<u>Complications & Challenges in Implementation</u>

1) <u>Calculating wall values</u>
   There were few issues I faced, like the first one was to write a function to calculate the wall value. The complicated part for me was to include the wall at the back of our robot. The robot has no sensors at its back, so how to check if there is a wall behind us? First I ignored the starting cell and imagined the robot moving in the maze, then it was clear that if the robot enters a cell it is because that cell was open for it to enter. Then having entered that cell, the opposite of robot's heading will serve for the bit value of the back. The code for calculating the wall value of a cell have been discussed earlier in ***determining walls inside** a cell* under ***basic functionalities*** section.

2) <u>Preferring Unexplored cells</u>
   This is another important part under exploration which helps in keeping the steps count to minimum. From the beginning we keep a track of visited cells during exploration using the closed grid. It helps us in preferring unexplored cells. But there are possibilities in which the robot will have no choice over moving to an already explored cell. An instance is shown in the figure



Here the robot's position is at green dot, 1s represent the visited cells and blues as the next opening to unexplored area. Now the robot have to take an action in order to move and until it reaches near one of the blue dots it will visit the explored cells and then have a choice in preferring the unexplored area. The code to accomplish this is shown below:

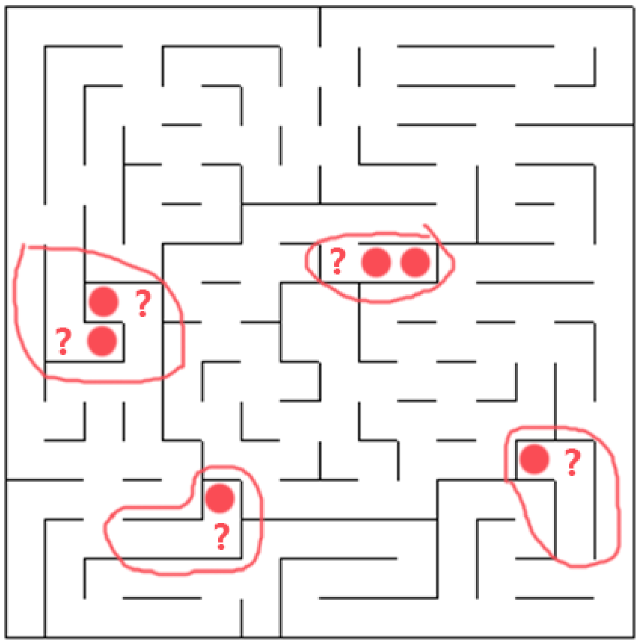


```
action = random.choice(actions) # randomly choose an action

# if cell is not explored perform action
if self.closed[x + dir_move[action][0]][y + dir_move[action][1]] == 0:

    rotation = dir_rotation[self.heading][action] # perform rotation based on action
    self.heading = action # new heading after rotation, same as action
    movement = 1

    self.location[0] += dir_move[self.heading][0]
    self.location[1] += dir_move[self.heading][1]

# if cell is already explored
elif self.closed[x + dir_move[action][0]][y + dir_move[action][1]] == 1:
    # select an action leading to unexplored cell
    # if all actions lead to explored cells, keep the previous action
    if len(actions)>1:
        for i in range(len(actions)):
            if  self.closed[x + dir_move[actions[i]][0]][y + dir_move[actions[i]][1]] == 0:
                action = actions[i]

    rotation = dir_rotation[self.heading][action]
    self.heading = action
    movement = 1
```

We start with a random action taken from the list of possible actions, and check for unexplored cells. If the action lead to an explored cell, the lines of code in the white rectangle first check for an action leading to unexplored area from the list of possible actions. If none, the previous action is kept. The code make sure that the next time robot have an option towards unexplored cell, it takes one, until then it can go towards the explored cells.

Refinement

A very interesting development comes to light while exploring maze_3; the dead ends in maze_3 are a bit different and deeper than the dead ends in other two mazes. The instances are shown in the figure below:



Till now during exploration of maze_3, the robot has been blocking the 'red dotted' cells only. And the next time it comes near any of these blocked dead ends (shown by ? mark) the robot becomes clueless, as it is left with no choice of actions. Thus an index error is raised saying, "cannot chose from empty sequence". This is because the actions leading to a blocked cell are restricted, and no other choice has been provided for this particular instance in maze_3. Other two mazes have no such dead ends in them. In order to solve this problem, an improvement is made in the code. The robot is made to go reverse, and block the path leading to these dead ends. It also helps in keeping the step counts low upto an extent, as the robot will not enter these paths again.

IV. Results

Model Evaluation and Validation

As there is randomness in exploration, it is less probable that the robot will map the maze perfectly with the best available path everytime. It will certainly map 'the next best' if not 'the best' path while exploration. Whichever the case, the robot must choose the best path available in the mapped maze. And if there exists two or more paths which require equal steps to the goal, the robot should prefer the path with less turns. Path with more turns will require more steps. The robot can take three straight steps at max, so it should utilize it.

Keeping the randomness in mind, for final evaluation I will run each algorithm 10 times on each maze and calculate an average score.

1. Depth First Search

Maze	Area Explored	Steps in 1st Run	Steps in 2nd Run	Score	Average	Standard Deviation
1	70.138	156	25	30.2	30.3	4.8
	70.138	186	21	27.2		
	97.35	296	23	32.867		
	70.138	212	23	30.067		
	70.138	154	28	33.133		
	75	156	18	23.2		
	70.138	175	25	30.8		
	87.5	342	30	41.4		
	70.138	166	24	29.533		
	83.33	168	19	24.6		
2	94.89	652	51	72.733	53.36	11.68
	70.4	229	31	38.633		
	70.4	249	46	54.3		
	70.4	235	33	40.833		
	95.4	520	51	68.33		
	91.32	368	39	51.267		
	70.4	415	30	43.833		
	89.79	415	51	64.833		
	72.95	249	33	41.3		
	81.12	288	48	57.6		
3	86.7	521	39	56.367	51.77	5.6
	89.84	454	39	54.133		
	70.31	252	35	43.4		
	75.39	277	32	41.233		
	70.31	580	36	55.333		
	70.31	460	39	54.333		
	70.31	340	39	50.333		
	88.28	760	31	56.333		
	70.31	287	38	47.567		
	70.31	261	50	58.7		

The depth first search algorithm seems to be inefficient. The average scores are higher than the benchmark we set. And individual scores varies a lot. Even after mapping a large area of the maze, the robot fails to recognize the best path available. Thus, it is not a good model for our problem.

2. Dijkstra

Maze	Area Explored	Steps in 1st Run	Steps in 2nd Run	Score	Average	Standard Deviation
1	81.25	188	19	25.267	25.475	1.96
	81.94	212	19	26.06		
	70.138	169	23	28.633		
	75	238	17	24.93		
	70.138	141	17	21.7		
	94.44	312	17	27.4		
	70.138	139	23	27.63		
	79.16	186	19	25.2		
	70.138	190	17	23.33		
	70.138	138	20	24.6		
2	78.57	292	27	36.733	36.493	2.64
	70.4	201	25	31.7		
	70.4	242	28	36.067		
	70.4	286	25	34.533		
	70.4	315	25	35.5		
	77.04	303	29	39.1		
	78.06	447	25	39.9		
	77.04	468	25	40.6		
	77.55	208	27	33.93		
	70.4	356	25	36.867		
3	70.31	366	28	40.2	42.15	3.88
	89.8	321	29	39.7		
	70.31	363	35	47.1		
	70.31	292	29	38.733		
	74.21	454	28	43.133		
	82.03	290	28	37.667		
	72.26	304	30	40.13		
	96.4	630	30	51		
	70.31	247	33	41.233		
	70.31	380	30	42.667		

Dijkstra’s performance turns out to be pretty decent. The results show that the paths found are indeed good and optimal paths. This algorithm has worked really well in maze_1 and in maze_2. It has managed to find the best path for both, which takes 17 and 25 steps. Not only this, it has found these paths multiple times and with low exploration score too. However for maze_3, there is still scope for improvement in scores. What’s impressive here is that we have begun to find a good exploration and optimization balance. More exploration only hurts the score. With around 70 percent of exploration, the robot has begun to find the optimal paths. The deviation in the scores are also less.

3. A Star

Maze	Area Explored	Steps in 1st Run	Steps in 2nd Run	Score	Average	Standard Deviation
1	70.13	151	17	22.03	26.09	2.02
	70.13	256	21	29.53		
	91.66	300	17	27		
	81.25	261	17	25.7		
	89.58	311	17	27.36		
	70.13	165	20	25.5		
	70.13	200	20	26.66		
	82.63	146	23	27.867		
	90.27	258	17	25.6		
	75	140	19	23.667		
2	93.877	470	26	41.667	36.11	2.41
	70.4	203	28	34.767		
	75	237	28	35.9		
	70.4	357	25	36.9		
	70.4	250	27	35.33		
	79.08	202	25	31.733		
	86.7	275	28	37.167		
	70.4	230	28	35.667		
	83.67	343	26	37.433		
	70.4	199	28	34.633		
3	70.31	264	30	38.80	39.96	4.21
	70.31	571	30	49.03		
	81.25	317	32	42.567		
	70.31	374	28	40.46		
	79.68	323	28	38.767		
	92.18	456	30	45.2		
	70.31	264	28	36.8		
	70.31	245	27	35.167		
	77.734	263	28	36.767		
	70.31	274	27	36.133		

The performance of 'A star' is very similar to dijkstra except for in maze_3. While dijkstra seems to perform better in maze_1 and maze_2, 'A star' has achieved a decent performance score for all mazes. It has also found the best paths for all mazes. Comparing to our benchmark, we have achieved a pretty significant score. We are also very near to our expected scores defined in the benchmark. Overall, this is a good model.

4. Dynamic Programming

Maze	Area Explored	Steps in 1st Run	Steps in 2nd Run	Score	Average	Standard Deviation
1	70.13	131	22	26.367	23.78	2.02
	70.13	157	19	24.23		
	70.13	117	19	22.9		
	92.36	264	20	28.8		
	70.13	198	17	23.36		
	88.19	323	20	30.76		
	70.13	231	20	27.7		
	70.13	159	17	22.3		
	93.75	320	17	27.667		
	70.13	202	17	23.733		
2	70.4	267	26	34.9	34.12	1.54
	70.4	315	24	34.5		
	79.59	262	27	35.733		
	70.4	301	27	37.03		
	71.93	236	25	32.867		
	72.44	272	26	35.067		
	70.4	229	24	31.633		
	71.42	236	25	32.867		
	80.1	205	26	32.83		
	70.4	296	24	33.867		
3	70.31	257	25	33.567	37.29	2.54
	73.04	274	27	36.133		
	83.20	419	25	38.967		
	91.79	450	25	40		
	70.3	351	30	41.7		
	70.3	277	25	34.233		
	92.96	435	25	39.5		
	70.3	241	27	35.033		
	70.3	258	28	36.6		
	70.3	306	27	37.2		

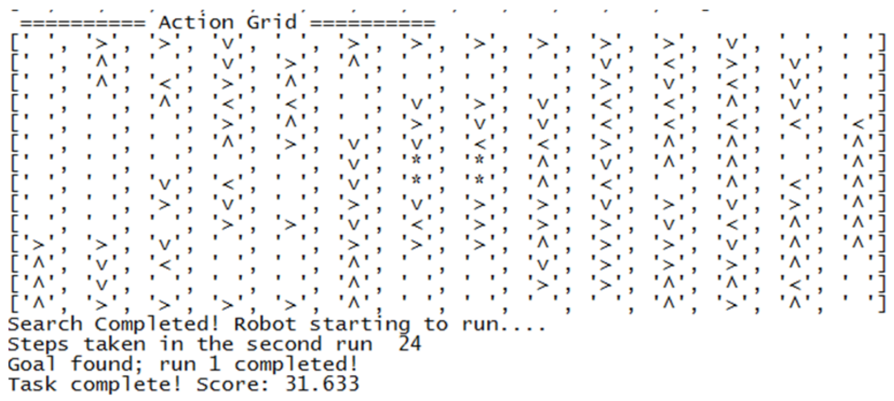
Dynamic programming has proven to be the best for all the mazes. The results clearly shows that it outperforms all other algorithms above. We have not only crossed the benchmark, but attained a really good score. The most exciting development is seen for maze_2, where it has found a very impressive path with just 24 steps. Similar thing happened for maze_3, for which it has found the best path with just 25 steps. No other algorithms have managed to find these paths for maze_2 and maze_3.

Justification

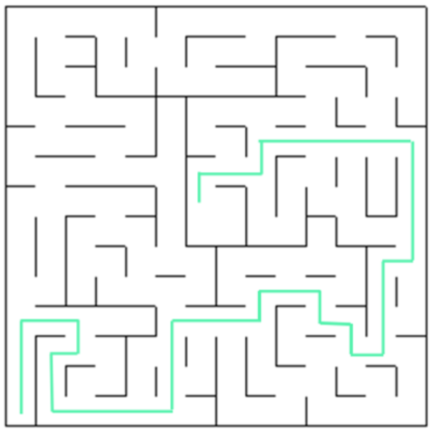
Dynamic programming outperforms other algorithms because the policy it creates has an optimal action from every cell mapped. So it can consider every path available in the map and chose the best among them. Additionally it has also preferred straight paths. That's why it has managed to find either an optimal or suboptimal path for each trial. I expected a score of 25, 36 and 42 for maze 1, 2 & 3. Comparing to this benchmark, it has attained better scores than expected.

I am also sharing the path it has found for maze_2 and maze_3 along with the policy created below:

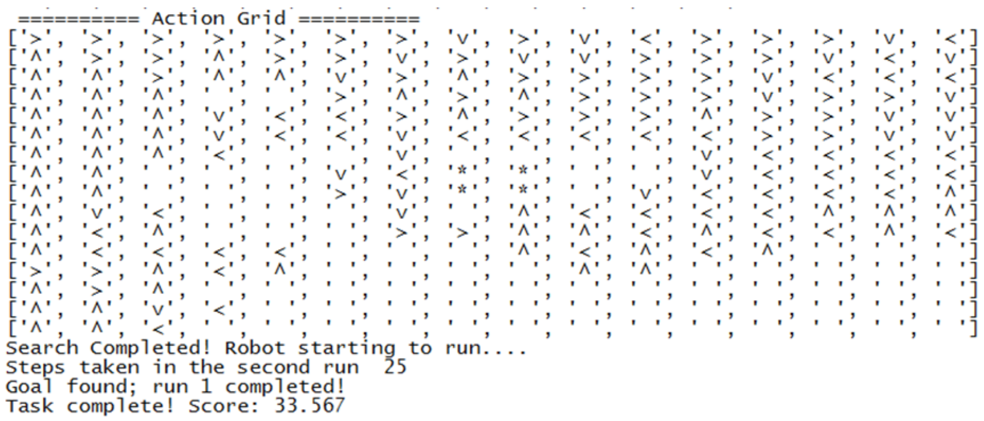
Maze 2



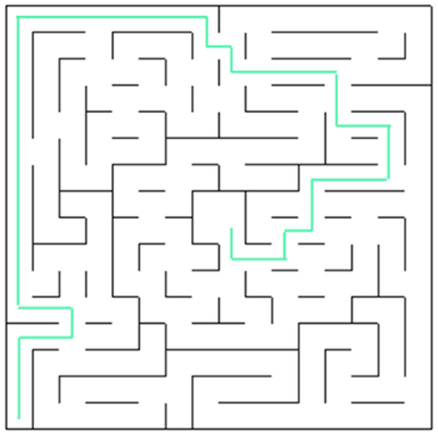
```
 ========== Action Grid ==========
[' ', '>', '>', 'v', ' ', '>', '>', '>', '>', '>', '>', 'v', ' ', ' ']
[' ', '^', ' ', 'v', '>', '^', ' ', ' ', ' ', 'v', '<', '>', 'v', ' ']
[' ', '^', '<', '>', '^', ' ', ' ', ' ', ' ', '>', 'v', '<', 'v', ' ']
[' ', ' ', '^', '<', '<', ' ', 'v', '>', 'v', '<', '<', '^', 'v', ' ']
[' ', ' ', ' ', '>', '^', ' ', '>', 'v', 'v', '<', '<', '<', '<', '<']
[' ', ' ', ' ', '^', '>', 'v', 'v', '<', '<', '>', '^', '^', ' ', '^']
[' ', ' ', ' ', ' ', ' ', 'v', '*', '*', '^', 'v', '^', '^', ' ', '^']
[' ', ' ', 'v', '<', ' ', 'v', '*', '*', '^', '<', ' ', '^', '<', '^']
[' ', ' ', '>', 'v', ' ', '>', 'v', '>', '>', 'v', '>', 'v', '>', '^']
[' ', ' ', ' ', '>', '>', 'v', '<', '>', '>', '>', 'v', '<', '^', '^']
['>', '>', 'v', ' ', ' ', '>', '>', '>', '^', '>', '>', 'v', '^', '^']
['^', 'v', '<', ' ', ' ', '^', ' ', ' ', 'v', '>', '>', '>', '^', ' ']
['^', 'v', ' ', ' ', ' ', '^', ' ', ' ', '>', '>', '^', '^', '<', ' ']
['^', '>', '>', '>', '>', '^', ' ', ' ', ' ', ' ', '^', '>', '^', ' ']
Search Completed! Robot starting to run....
Steps taken in the second run  24
Goal found; run 1 completed!
Task complete! Score: 31.633
```



Maze 3



```
 ========== Action Grid ==========
['>', '>', '>', '>', '>', '>', '>', 'v', '>', 'v', '<', '>', '>', '>', 'v', '<']
['^', '>', '>', '^', '>', '>', 'v', '>', 'v', 'v', '>', '>', '>', 'v', '<', 'v']
['^', '^', '>', '^', '^', 'v', '>', '^', '>', '>', '>', '>', 'v', '<', '<', '<']
['^', '^', '^', ' ', ' ', '>', '^', '>', '^', '>', '>', '>', 'v', '>', '>', 'v']
['^', '^', '^', 'v', '<', '<', '>', '^', '>', '>', '>', '^', '>', '>', 'v', 'v']
['^', '^', '^', 'v', '<', '<', 'v', '<', '<', '<', '<', '<', '>', '>', 'v', 'v']
['^', '^', '^', '<', ' ', ' ', 'v', ' ', ' ', ' ', ' ', 'v', '<', '<', '<', '<']
['^', '^', ' ', ' ', ' ', 'v', '<', '*', '*', ' ', ' ', 'v', '<', '<', '<', '<']
['^', '^', ' ', ' ', ' ', '>', 'v', '*', '*', ' ', 'v', '<', '<', '<', '<', '^']
['^', 'v', '<', ' ', ' ', ' ', 'v', ' ', '^', '<', '<', '<', '<', '^', '^', '^']
['^', '<', '^', ' ', ' ', ' ', '>', '>', '^', '^', '<', '^', '<', '<', '^', '<']
['^', '<', '<', '<', '<', ' ', ' ', ' ', '^', '<', '^', '<', '^', ' ', ' ', ' ']
['>', '>', '^', '<', '^', ' ', ' ', ' ', ' ', '^', '^', ' ', ' ', ' ', ' ', ' ']
['^', '>', '^', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
['^', '^', 'v', '<', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
['^', '^', '<', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
Search Completed! Robot starting to run....
Steps taken in the second run  25
Goal found; run 1 completed!
Task complete! Score: 33.567
```

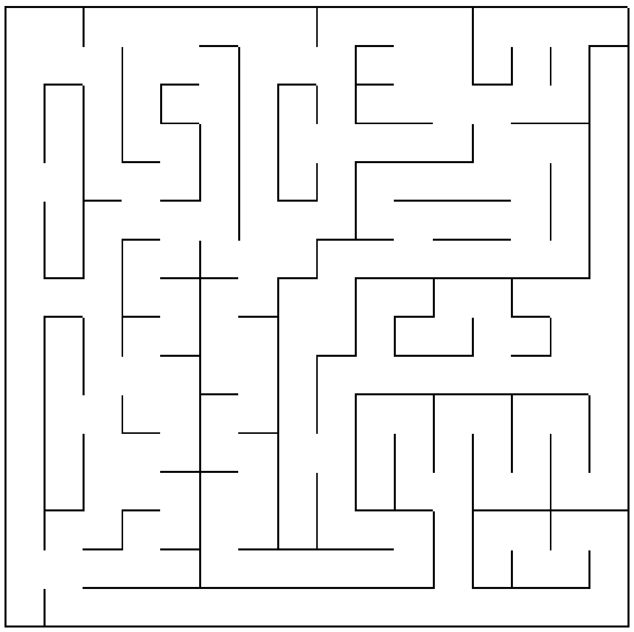


Finally, considering our criteria for straight and shorter paths it is clearly the best model for our project. It has also proven to be successful with a threshold exploration of 70 percent thus attaining a good exploration and optimization balance too.

V. Conclusion

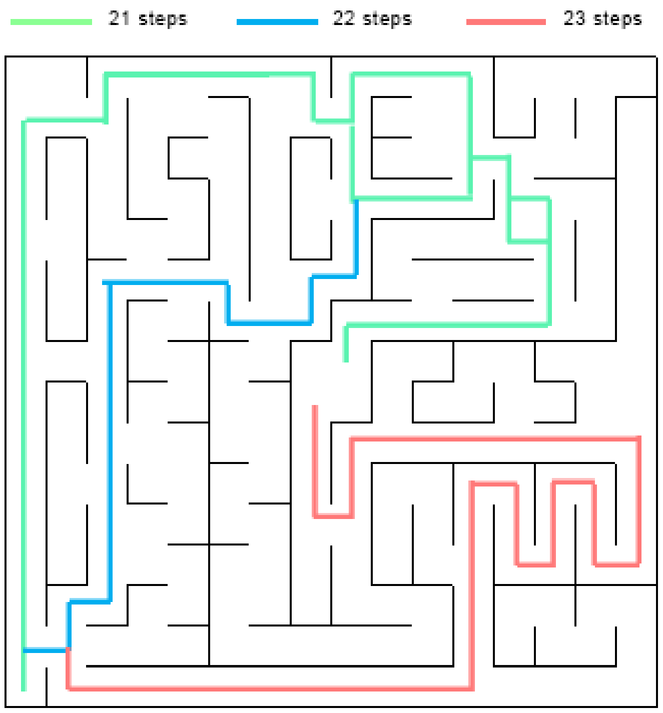
Free Form Visualization

I have created another maze which contains some new examples of dead ends. The text file of the maze has been included in the project files among other three mazes. I have tried to create some deeper dead ends, and some dead ends which constitute a block of cells. One example is the lower right block in the visualization on the right. Another example is the block on the right of the goal area.



My intention here is to check if the robot is able to identify some different sort of dead ends. The maze contains a few suboptimal paths and one best path, which the robot is intended to find.

On testing my model with ten trials, I get an average score of 30.1. The robot is frequently able to find the best path. The best path found takes just 21 steps. Additionally, the model is indeed able to block the dead ends except for the lower right block. Apparently, the robot needs to revisit the cells adjacent to a blocked cell in order to block the surrounding cells. It is really not an efficient thing to do. Thus there is still a lot of scope for improvement here. The paths found by the robot is shown in the visualization below.



Reflection

The project was to give an intelligence to a robot which can learn by itself. The specification of the maze was provided to us, and the robot was expected to read the maze by itself. This was really a challenging part; to enter a completely unknown environment and study it. I began with studying *'Udacity's Artificial Intelligence for Robotics'* course in order to learn about the problem domain, algorithms and how to think like a robot. It was really very helpful.

Finally we have made an autonomous robot which can learn about an unknown maze, create its structure and then run to the goal location from the start in the fastest time. The problem was challenging. It began with giving the robot a basic intelligence to localize itself, and enable it to perform actions, read its surroundings and create the maze world. The most interesting and difficult part was the exploration part. For a good score we needed the perfect map, which was difficult to achieve. And with adding randomness to the exploration it became more challenging. The randomness could take it anywhere, so we made the robot give preference to unexplored area and avoid dead ends. It turned out to be really beneficial. Then came the challenge to decide when to stop the exploration and start the final run to the goal. But it was observed that running around unnecessary in order to map the complete maze would only hurt the score. So with observations we came up with a decent threshold. Then comes the part of giving the robot the final piece of intelligence which it could apply in order to find the best path. We tried several algorithms and got our winner. Overall I feel there is still a better way to explore the maze. As with randomness the chance of getting a good score every time is less. Maybe apart from random movement we can try another technique or algorithm to explore and then use dynamic programming to find the best path. Basically I feel, if we somehow kill the random factor, this model can really become more impressive.

Improvement

There is still a lot that can be improved. One improvement is discussed in the above section which is to kill the randomness. If the robot is able to explore the maze with perfection and without taking much steps it can score a lot better leading to a consistent performance. Then another scope of improvement lies in identifying and blocking any sort of dead ends.

Till now the problem have been discussed according to the discrete domain. But the real competition plays in the real world, thus it lies in the continuous domain. For now our model is not equipped for the continuous domain and need a lot of adjustments and improvements. The first thing to consider is the errors. Errors in sensing, movement, rotation, breaking, accelerating etc. Then the movement and rotation should also be dealt according to the continuous domain, like the speed, amount of movement and rotation. If the diagonal movement is also allowed then it can be added to improve our robot's performance. Apart from these needed adjustments, we would also require to apply the algorithms considering the continuous domain. Then there is also the physical body part as it acts in the real world. So there is a lot of scope to improve and it will only become more interesting.

Resources

Udacity Project Files Link

https://docs.google.com/document/d/1ZFCH6jS3A5At7_v5IUM5OpAXJYiutFuSIjTzV_E-vdE/pub

Micromouse Online

http://www.micromouseonline.com/

Udacity's Intro to Artificial Intelligence Course

https://in.udacity.com/course/artificial-intelligence-for-robotics--cs373

Wikipedia

https://en.wikipedia.org/wiki/Micromouse

APEC Micromouse Contest Rules

http://micromouseusa.com/wp-content/uploads/2013/10/APEC-Rules.pdf

Algorithms - Hackerearth

https://www.hackerearth.com/practice/algorithms/graphs

Path Planning – Correl Lab

http://correll.cs.colorado.edu/?p=965

Breadth-First Search Algorithm

https://en.wikipedia.org/wiki/Breadth-first_search

Depth-First Search Algorithm

https://en.wikipedia.org/wiki/Depth-first_search

Dijkstra Algorithm

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Uniform-Cost Search Algorithm

https://en.wikipedia.org/wiki/Talk%3AUniform-cost_search

https://stackoverflow.com/questions/12806452/whats-the-difference-between-uniform-cost-search-and-dijkstras-algorithm