Udacity Machine Learning Nanodegree Capstone Project

Robot Motion Planning - Plot and Navigate a Virtual Maze

Report By:

Aditya Karlekar

June, 2018

---

1. Project Definition

Overview:

The project is inspired by the Mircomouse competitions. The goal is to make the mouse find an optimal path from one corner of the maze to its center. The mouse is allotted two attempts to navigate the n x n maze. In the first run, the mouse surveys the maze to find the center and find the best path to reach the center. In the second run, the mouse tries to reach the center in the shortest time possible by following the most optimal path calculated with information from the first run. In this project, the mouse will navigate through a virtual maze. The project uses python code to train the robot to explore the maze in the first run and find optimal path in second run.

Problem Statement:

The goal is to reach the center of the maze in the shortest time possible. The robot will start from the bottom left corner of the n x n maze. Two tires will be provided to accomplish this task. The first run to explore the maze and second run to reach the center as soon as possible. The maze will be of size n x n where n could be of the value of 12, 14 or 16. The center will be of 2 x 2. The maze is enclosed by walls so that the mouse cannot move out of the maze. Also, the first step will be forward as the starting square will have a wall on right, left and bottom side.

The first run of the robot is for exploration purpose. In the first run, the robot tries to find the structure and shape of the maze, including all possible routes to the goal area. For the run to be successful, the robot must travel to the goal area at least once. However, the robot is allowed to continue its exploration after the goal has been reached. In the second trial, the actual optimization takes place. In this trial, the robot uses the information learned from the first trial and will attempt to reach the goal area in an optimal route. The robot's performance will be evaluated by adding the following:

● **Number of total steps in the *First Run* divided by 30**

● Total number of steps to reach the goal in the *Second Run*
Additionally, each trial has a maximum of 1000 steps. The robot can only make 90° turns (clockwise or counterclockwise) and is allowed to move up to three consecutive spaces forward or backwards in a single movement.

Metrics:

The performance will be evaluated as the sum of one-thirtieth of steps in first run and number of steps in the second run. This is our benchmark model.

Score = [Number of Steps in Second Trial] + [Number of Steps in First Trial / 30]

For example: If the robot takes 150 steps in First Trial and 50 steps in the Second Trial, then the score will be sum of 50 and 150/30 i.e. 50 + (150/30) which results in total score of 55.

By examining the evaluation formula for score, we can see that the score value is affected more by the performance of robot in second run than in first run. Hence, we can say that the better the optimization algorithm is, the better will be the score value.

2. Analysis

Data Exploration:

The starter code for this project provided by Udacity contained following files

- robot.py: It contained the robot class. The main logic of the code is implemented in this file.
- test_maze_##.txt: These files contains the maze structure. ## is replaced by a number in the file name.
- tester.py: This file is used to test the performance of robot. The test_maze_##.txt is given as an input.
- maze.py: This file is used to construct maze from the test_maze_##f.txt files and works in conjunction with the robot.py file. It provides wall and sensor values to the robot.
- showmaze.py: This file is used to visualize the maze. The test_maze_##.txt is given as an input.

The robot can only make 90° turns left or right and is allowed to move up to three consecutive spaces forward or backwards in a single movement.

Each maze is drawn on a square grid. The grid is made up of either 12, 14, or 16 rows and columns. The maze is enclosed by walls which act as barrier and does not allow the robot to move out of it. The robot starts in the bottom left-hand corner of the maze at coordinates (0, 0) and is facing in an upward direction. The first move is always forward or ‘up’ as the maze has walls on its left, right, and bottom sides with an opening only on its topside. The robot's task is to navigate through the maze and reach the goal state which is situated at the center of the maze. It is of dimension 2-by-2. For a successful run, the robot has to reach the goal state at least once. The robot is tested on three different maze configurations.

The structure of the maze provided in the test_maze_##.txt file is as follows:

```
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
```

Figure 1. Contents of test_maze_##.txt file

The first line of the text file shows the length of side of square maze. The line could contain any value of 12, 14 and 16. Thus, we have mazes of dimensions 12x12, 14x14 and 16x16. The following lines consist of comma separated numbers which tells the position of wall and open spaces for movements. The numbers are of range 1 to 15. Each number represents a four-bit number that shows the position of wall and open edge in the cell. The bit value of 0 represents a wall while 1 represents open edge. In the four bit number, the 1s register represent front/upward, 2s register represents right, 4s register represents the backward/down side and the 8s register value represents the left side. For example, number 7 with four bit representation as 0111 will have closed edge (wall) in front while the left side, right side and back side are open for movement.

The maze cell follows a Cartesian coordinate system. The text-file describes the maze column by column. The first number (excluding the number on first line which represents the length) represents the starting cell in the bottom-left corner. Figure 2 shows the number and closed (walled) and open edges in a cell. Figure 3 shows a maze along with dead-ends, starting position and goal state. The green dot represents the goal state, the black dot represents the start state and the red dot represents the dead-ends.

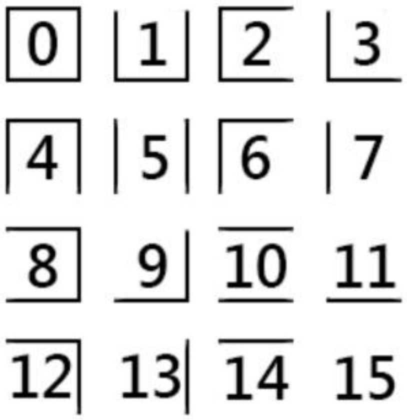


Figure 2. Cell Structure as described by numbers

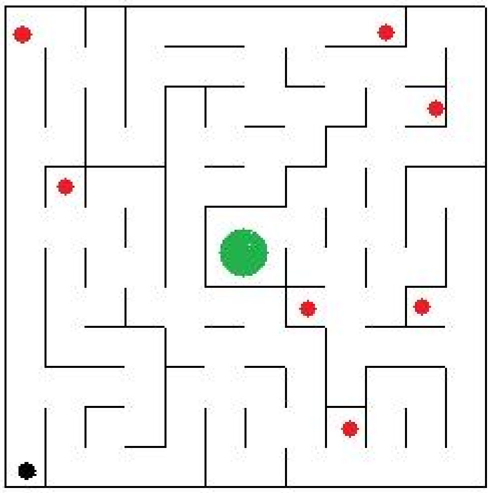
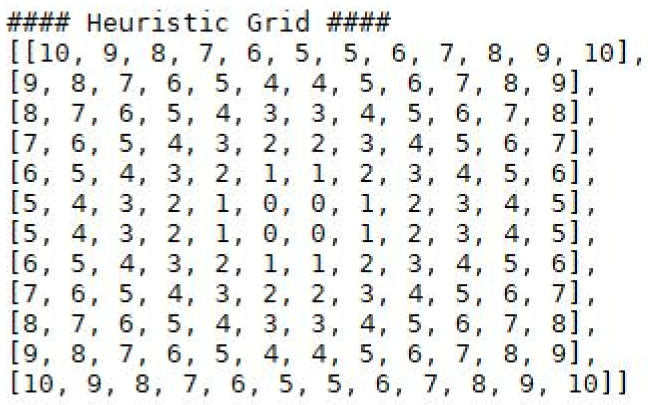


Figure 3. Maze visualization with start, goal and dead-ends

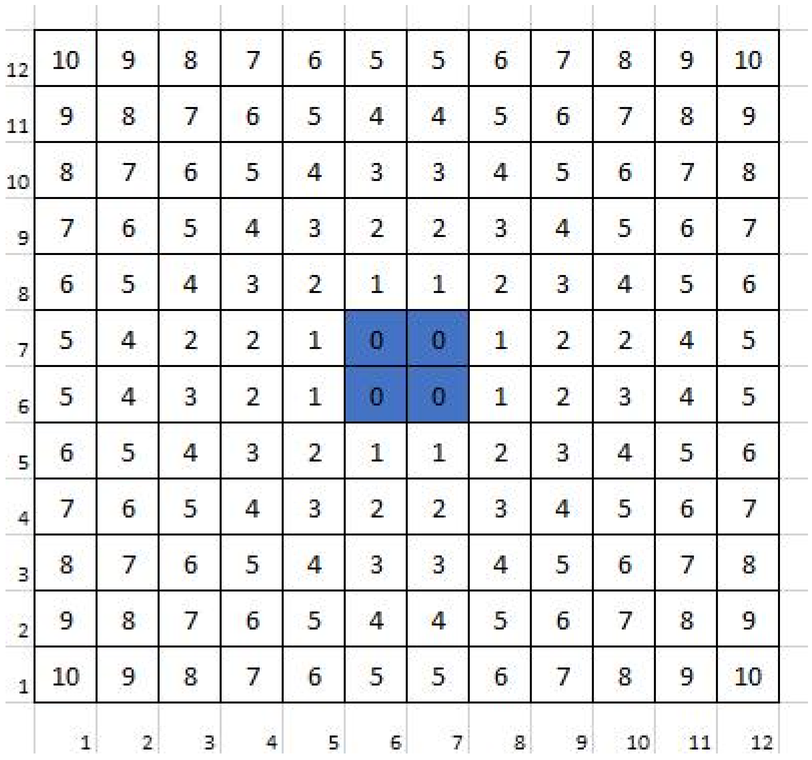
The cell structure is provided to robot by maze.py file through the variable name 'sensors'. The robot can detect three sensors. Sensor 0 represents left, Sensor 1 represent top and Sensor 2 represent top. The robot can rotate precisely 90 degrees clockwise or counter clockwise and can move up to three steps backward or forward at once. Once the robot reaches a new position, the sensors values are updated.

Exploratory Visualization:



```
#### 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]]
```

Figure 4. Maze 1 Heuristic Grid during Second (Optimization) Run



12	10	9	8	7	6	5	5	6	7	8	9	10
11	9	8	7	6	5	4	4	5	6	7	8	9
10	8	7	6	5	4	3	3	4	5	6	7	8
9	7	6	5	4	3	2	2	3	4	5	6	7
8	6	5	4	3	2	1	1	2	3	4	5	6
7	5	4	2	2	1	0	0	1	2	2	4	5
6	5	4	3	2	1	0	0	1	2	3	4	5
5	6	5	4	3	2	1	1	2	3	4	5	6
4	7	6	5	4	3	2	2	3	4	5	6	7
3	8	7	6	5	4	3	3	4	5	6	7	8
2	9	8	7	6	5	4	4	5	6	7	8	9
1	10	9	8	7	6	5	5	6	7	8	9	10
	1	2	3	4	5	6	7	8	9	10	11	12

Figure 5. Maze 1 Heuristic Grid Visualization

The Heuristic Grid is representing a number indicating the number of steps required from that cell to reach the goal state. The A* algorithm used in the second run can use Manhattan, Diagonal and Euclidean Distance as a measure. In this implementation, Manhattan distance is used. The four zeros at the center (marked blue in Figure 5) represents the goal state. As each cell represent the distance from center (1 unit increase per cell) then if the robot is at cell of value 7, then logically it should move to cell with value 6. This process should continue till 0 value is reached.

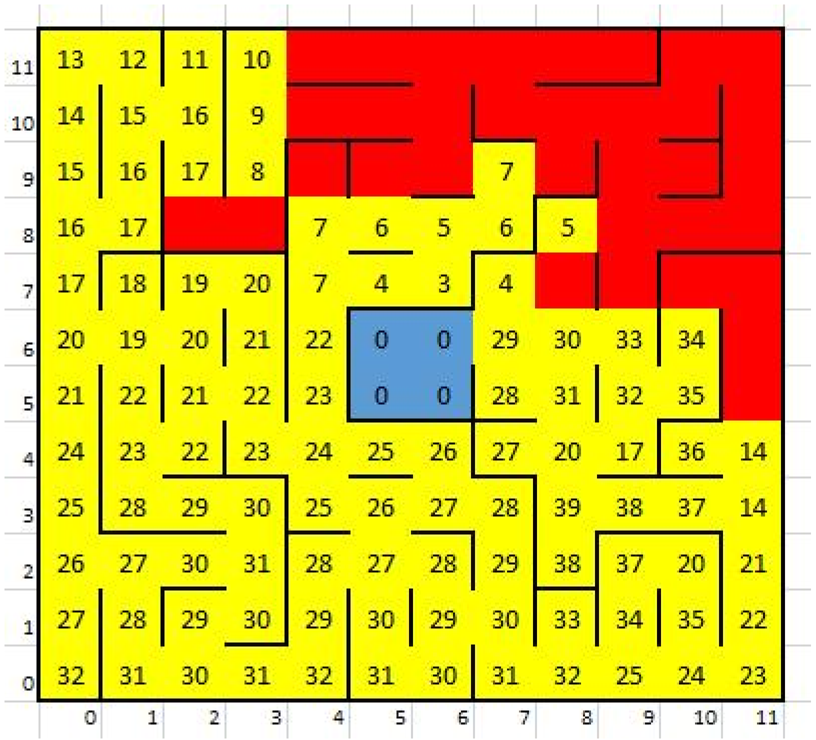


Figure 6. Path Value Grid for Test Maze 1

As the robot explores the maze, it updates the heuristic grid. The updated grid is showed in Figure 6. The robot starts at position with cell value 32. The cells colored yellow are the ones visited by robot in first run. The cells in red color are the ones not explored by the robot in first run. The exploration limit (i.e. first run) is set at 70% maximum. The search continues till 70% of maze has been explored even though the goal state has been found. For evaluation purpose, I have tested the robot with different exploration limit and noted the results.

The Figure 7 shows the optimal path taken by the robot to navigate to the goal state. The robot uses dynamic programming to find the optimal path. The numbers in the maze shows the position at that particular step while the arrow shows the direction. As expected, the robot takes much fewer steps in the second run as compared with the first run. This is what we wanted from the optimized run.

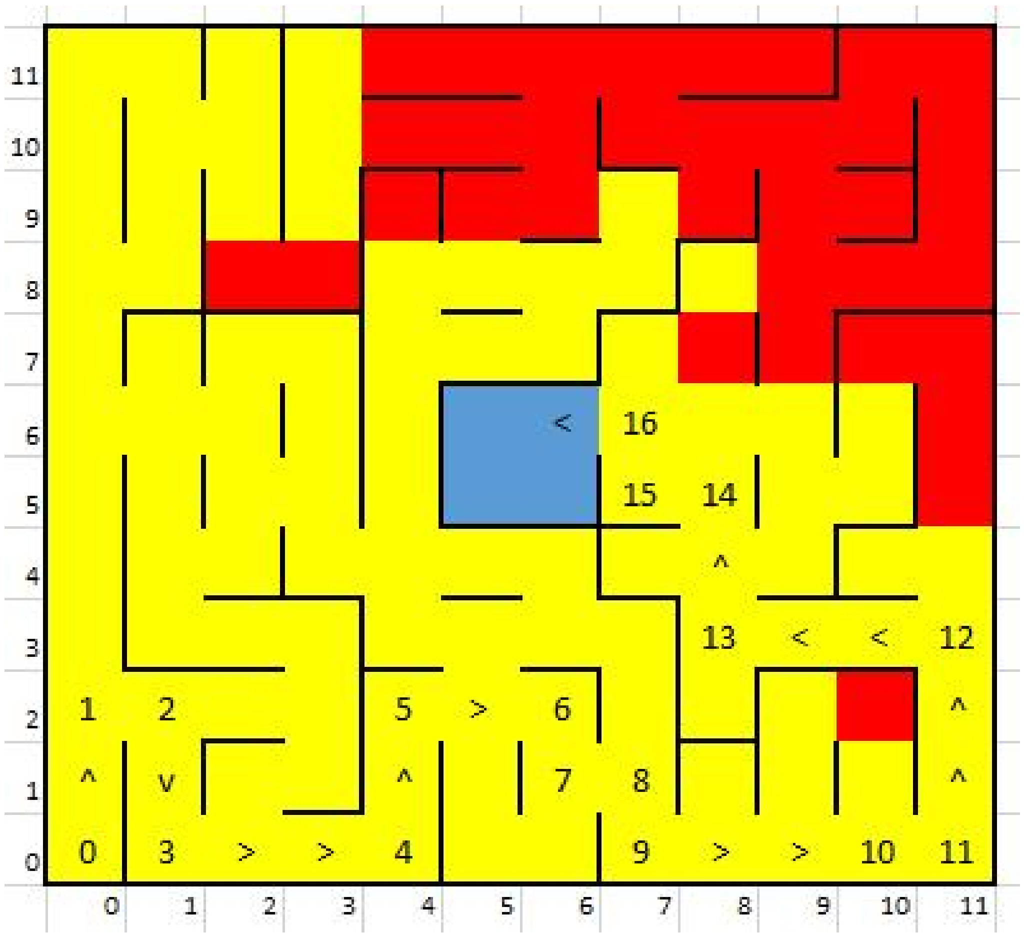


Figure 7. Optimal path for Test Maze 1

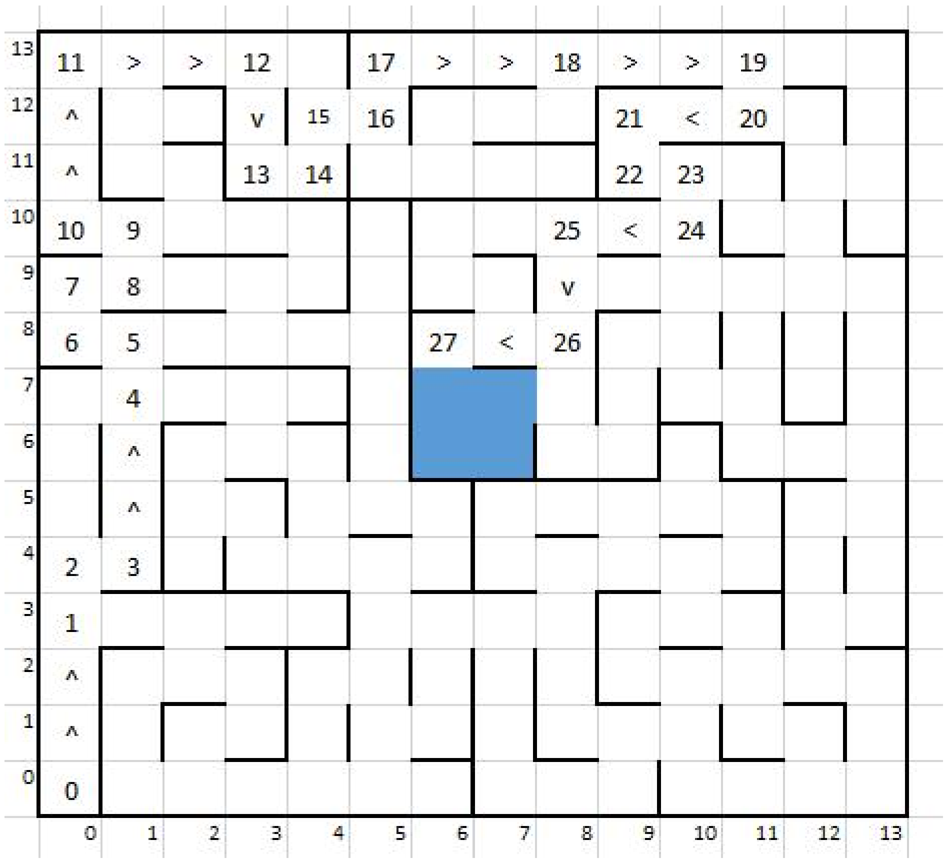


Figure 8. Optimal Path for Test Maze 2

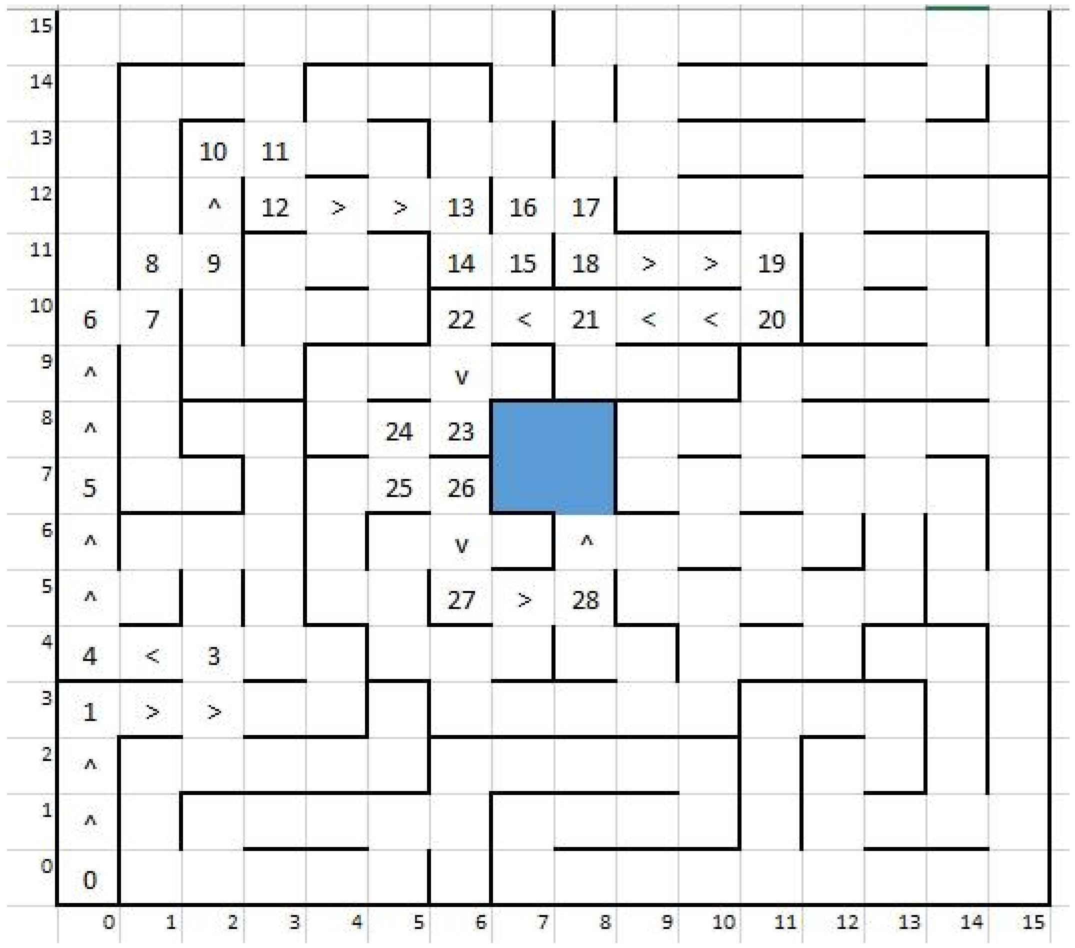


Figure 9. Optimal path for Maze 3

Algorithms and Techniques:

There are several algorithms that could be used to traverse the maze. First, I'll mention the algorithms I considered that could effectively solve the problem and then discuss the algorithms I used for exploration and optimization runs. Let see the algorithms:

Wall Follower:

The wall follower makes the agent to take either only the right or left turn, eventually reaching the goal state. The agent can get caught in a circular loop.

Depth First Search (DFS):

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

The pseudocode for DFS is as follows:

```
preorder(node v)
    {
    visit(v);
    for each child w of v
        preorder(w);
    }
dfs(vertex v)
    {
    visit(v);
    for each neighbor w of v
        if w is unvisited
        {
        dfs(w);
        add edge vw to tree T
        }
    }
```

Breadth First Search:

The BFS is also used for searching tree-like data structure. The algorithm starts with the root node and then moves to the node at the same level of the tree (closest neighbor) before moving to next level.

The pseudocode for BFS is as follows:

```
unmark all vertices
    choose some starting vertex x
    mark x
    list L = x
    tree T = x
    while L nonempty
    choose some vertex v from front of list
    visit v
    for each unmarked neighbor w
        mark w
        add it to end of list
        add edge vw to T
```

Dijkstra’s Algorithm:

Dijkstra’s algorithm is used to find shortest path from one node to all nodes in the graph. In maze problem, we can use this algorithm as the algorithm will find shortest path from the starting stage to the goal stage.

Djikstra’s Pseudocode:

```
// Let v1 be the origin vertex,
//   and initialize W and ShortDist[u] as
   W := {v1}
   ShortDist[v1] :=0
   FOR each u in V - {v1}
      ShortDist[u] := T[v1,u]

// Now repeatedly enlarge W
//   until W includes all verticies in V
   WHILE W <> V

      // Find the vertex w in V - W at the minimum distance
      //   from v1
      MinDist := INFINITE
      FOR each v in V - W
         IF ShortDist[v] < MinDist
            MinDist = ShortDist[v]
            w := v
         END {if}
      END {for}

      // Add w to W
      W := W U {w}

      // Update the shortest distance to vertices in V - W
      FOR each u in V - W
         ShortDist[u] := Min(ShorDist[u],ShortDist[w] + T[w,u])
   END {while}
```

A* Algorithm:

A* algorithm works by creating a tree of all paths from starting point to end point and then chooses the most optimal path from them. It’s possibly the best algorithm to find the shortest path.

The pseudocode 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
```

Dynamic Programming:

The Dynamic Programming is similar to A*. the Dynamic Programming approach finds the shortest or optimal path for all possible starting points. Thus it has an advantage over A* in coming up with the best route to destination. The pseudocode is as follows:

```
value_function()
Initialize the value grid with a large value for all cells.
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
```

Benchmark:

The benchmark model will be evaluated by the performance score previously discussed in the *Problem Statement* section:

Score = [Number of Steps in Trial 2] + [Number of Steps in Trial 1 / 30] .

The robot can use maximum of 1000 steps to navigate the maze in the first run. The exploration will fail if the robot fails to find the goal state within 1000 steps, then the whole learning attempt fails. Once the first run is complete, then, the knowledge gained in first run is used to find the optimal (or at least better than first run) path from start point to goal. As the robot has learned during first run, we can expect the robot to use the learned information to perform better in the second run.

The Optimal values for mazes 1, 2 and 3 are 17, 23 and 25 respectively. The score takes into consideration total value is second run and one-thirtieth value of first run. Since, whole value of second run is used for score calculation, the more optimized the learning is, the better will be the score value. For the exploration run to be successful, the robot needs to visit the cells more than once, thus the value for first trail will be higher. For benchmark, I've used optimal values for second run. Since, a much smaller value from second trial is used, I've multiplied the first value with a factor of 1.7.

So the new benchmark is defined as follows:

Benchmark = [Optimal Steps for Trial 2] + [No. of trials in Trial 1 * 1.7] / 30

Benchmark Maze 1 Score: 15 + (145 * 1.7) / 30 = 23.216

Benchmark Maze 2 Score: 23 + (265 * 1.7) / 30 = 38.01

Benchmark Maze 3 Score: 25 + (303* 1 .7) / 30 = 42.17

Another factor that can affect the score is the threshold value of maximum area that the robot will traverse in the first run. We'll investigate further what percentage area traversal can give us the best results.

3. Methodology

Data Processing:

The project did not required any data processing. The mazes used were provided by Udacity. The robot traverses the maze and learn the information on the go. The learned information is then used to find the optimum solution.

Implementation:

After understanding the problem and reviewing the starter code files provided, I designed the solution. Before discussing the solution methodology, let's discuss the starter files provided.

- robot.py: The file where the actual solution has to be implemented.
- maze.py: Creates the maze from provided maze files.
- tester.py: The tester.py files checks the implementation of code from robot.py
- test_maze_##.txt: These files contains the maze structure. ## is replaced by a number in the file name.
- showmaze.py: This file is used to visualize the maze. The test_maze_##.txt is given as an input.

The implemented solution is as follows:

- Interpreting the sensor values: The maze.py file provides sensor values that contains the information of closed (wall) and open space of the cell.
- The First Exploration Trial: The first run is made to traverse the maze and find the goal state.

- The Second Optimization Trial: The Second run consist of following:
  - Finding Optimized Route: From the information learned in the first run, the shortest path from start position to goal position is calculated.
  - Traversing the optimized route.
- Calculating the score value using the defined score metric.

Functionalities:

The solution is implemented in robot.py file. The following data variables are used to store the relevant data used throughout the code.

- Global variables and directories:

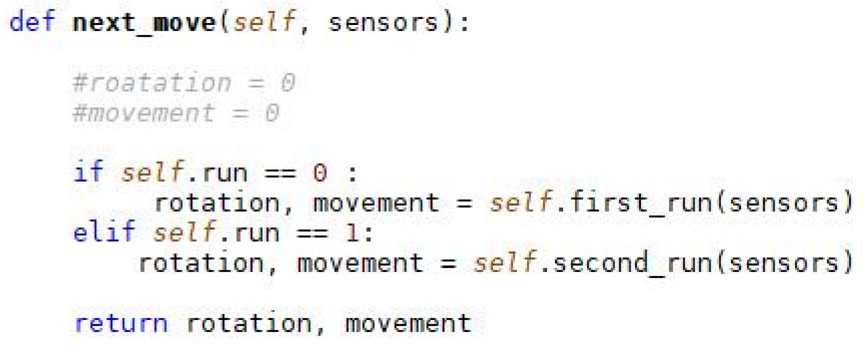
  These variables and directories help the robot to take the logical actions. The variables and directory values remains constant throughout the code. The global variables defined are as follows:

  - *dir_sensors*: The directory is used to interpret the sensor information used in robot's heading.
  - *dir_move:* This directory provides movement information to the robot based on heading values.
  - *dir_reverse*: reverse movement information based on heading.
  - *degrees*: Directory holding rotation values

- Other Attributes:
  Following other variables are also defined to store pertinent information.
  - *self.maze_area*: For storing maze area value
  - *self.maze_grid*: grid for wall locations for each maze
  - *self.path_grid*: grid for path
  - *self.path_value*: Grid for value function in Dynamic Programming
  - *self.policy*: Grid for optimal route
  - *self.goal_discovered*: To define goal area in the center
  - *self.heuristics_grid*: To store the heuristic values during A*

next_move function:

The next_move function determines the trail number that the robot will execute.



```
def next_move(self, sensors):

    #roatation = 0
    #movement = 0

    if self.run == 0 :
         rotation, movement = self.first_run(sensors)
    elif self.run == 1:
        rotation, movement = self.second_run(sensors)

    return rotation, movement
```

Figure 10**. next_move** implementation

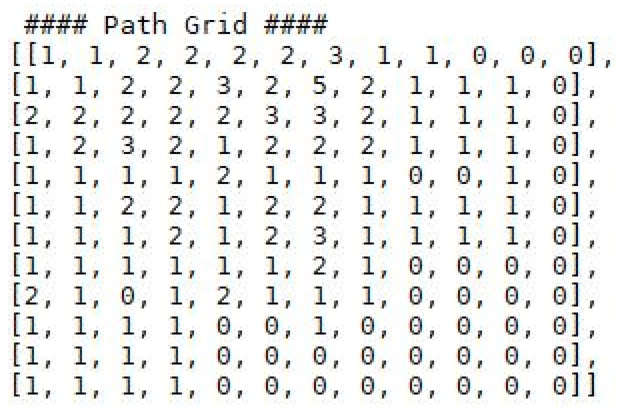
First, the first run or exploration trail is done. The first_run function is executes the exploration trial.

first_run function:

For the purpose of exploration, I've Implemented A* algorithm. In this trail, the robot traverses the maze and maps the open spaces, closed spaces (walls) and goal state. The robot reads the sensor values supplied by the maze.py file to detect closed and open space. The walls are mapped using **wall_positions** function. The robot has three sensors: front, right and left. Using the sensor reading and rotation value, the robot generates four-bit number which describes the wall position and open spaces. The values of explored maze us stores in maze_grid variable. Also, the number of times a particular cell is visited is stored in path_grid variable. The path_grid is two-dimensional zero array.

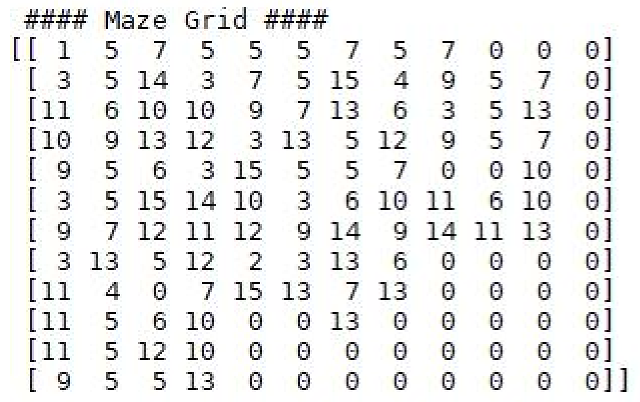
A* uses the heuristic grid to find the optimal path. The heuristic value of a cell is defined as the number of steps it would take to reach the goal state. The function **calculate_next_move** is used to determine the robot's next move. The next move is determined by the sensor values, available open spaces (wall positions) and orientation (rotation value). If the robot is facing a dead end, then the robot is instructed to reverse. If only one path opening is present, then the robot follows that opening. If more than one path is available, then, the robot uses heuristic grid and moves to cell with lower heuristic value (closer to cell).

To ensure that the goal is discovered in the first run, I've created a variable named goal_found. The goal_found variable is Boolean in nature and initially set as False. The location of center is provided to the robot using goal_discovered variable. The robot is expected to search a minimum of 70% (other percentage values are also used for scoring purpose) area even when the goal is found.



```
 #### Path Grid ####
[[1, 1, 2, 2, 2, 2, 3, 1, 1, 0, 0, 0],
[1, 1, 2, 2, 3, 2, 5, 2, 1, 1, 1, 0],
[2, 2, 2, 2, 2, 3, 3, 2, 1, 1, 1, 0],
[1, 2, 3, 2, 1, 2, 2, 2, 1, 1, 1, 0],
[1, 1, 1, 1, 2, 1, 1, 1, 0, 0, 1, 0],
[1, 1, 2, 2, 1, 2, 2, 1, 1, 1, 1, 0],
[1, 1, 1, 2, 1, 2, 3, 1, 1, 1, 1, 0],
[1, 1, 1, 1, 1, 1, 2, 1, 0, 0, 0, 0],
[2, 1, 0, 1, 2, 1, 1, 1, 0, 0, 0, 0],
[1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0],
[1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
[1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]]
```

Figure 11. Path Grid for test maze 1



```
 #### Maze Grid ####
[[ 1  5  7  5  5  5  7  5  7  0  0  0]
 [ 3  5 14  3  7  5 15  4  9  5  7  0]
 [11  6 10 10  9  7 13  6  3  5 13  0]
 [10  9 13 12  3 13  5 12  9  5  7  0]
 [ 9  5  6  3 15  5  5  7  0  0 10  0]
 [ 3  5 15 14 10  3  6 10 11  6 10  0]
 [ 9  7 12 11 12  9 14  9 14 11 13  0]
 [ 3 13  5 12  2  3 13  6  0  0  0  0]
 [11  4  0  7 15 13  7 13  0  0  0  0]
 [11  5  6 10  0  0 13  0  0  0  0  0]
 [11  5 12 10  0  0  0  0  0  0  0  0]
 [ 9  5  5 13  0  0  0  0  0  0  0  0]]
```

Figure 12. Maze Grid for test maze 1

Finding optimal value:

Once the maze is traversed, we compute the optimal route. For this I've implemented the Dynamic Programming technique, which allows the robot to find the optimal path. The Dynamic Programming approach used in this project is borrowed *from Udacity Artificial Intelligence for Robotics Course*. The Dynamic Programming creates a policy grid which includes a direction (left, right or up) for each cell leading to the goal state.

To implement Dynamic Programming, a Path value grid is created. The path_value variable stores the Path Value Grid. The Path Value grid has same dimensions as the maze and is initialized by some high value. Starting from goal area, the grid updates (based on values from value_calculation), slowly moving outwards to wherever there is an open space. The values in Path Value Grid changes slowly and after few iterations the Path Value Grid starts to look like heuristic grid with each cell value corresponding to the distance of the cell from the center.

Once, all steps are finished, the robot is placed again at the start position and variables like movement, rotation, heading are reverted to their original value for second run.

second_run function:

Once the optimal path is calculated, the robot simply traverses that path. The second_run function simply traverses the path computed using Dynamic Programming.

Challenges, Improvements and Refinements:

My initial approach to this problem was very different. Initially, I implemented a random movement algorithm to trace the maze in the first trail and decided to use algorithms like Depth First Search, Breadth First Search, A* etc. for the optimization trail. However, this approach did not provided the desired result. There was also a challenge to keep track of visited cells and determining the wall positions. The overcome this, I implemented the wall_positions function and store the visited cells in path_grid variable.

The biggest problem with using random movement was that for most part it used to fail to discover the goal state under the maximum limit of 1000 steps. This problem was more acute is mazes which had deep dead-ends. The deep dead-ends were the dead-ends where when the robot reaches the dead-end, it cannot move anywhere as there are no valid positions for it to move. Figure 13 shows the maze with deep dead ends. To overcome this obstacle, I decided to write the code to reverse the robot. However, the modified code still did not perform as per the expectations. This meant that I have to radically change my approach.

So, in my second attempt, I decided to go with DFS or BFS for first run and A* for second run. This approach proved to be more effective than my first approach, but still it has issues. For example, the DFS and BFS were slow and took a lot of steps to find the goal state. This resulted in a very poor score metric. So, now I knew, I needed A* for exploration and something better than A* for second trail. On this capstone project page, a link is provided for Udacity Artificial Intelligence for Robotics Course. I studied that course especially the search part and leaned about the Dynamic Programming Approach to find the optimal path. One reason why the combination of A* and Dynamic Programming works is due to the efficiency of A* in finding the open paths. The A* algorithm is very cost effective search method as the algorithm emphasis on finding the shortest path first with the help of heuristic values. This plays a vital role as it reduces the steps in reaching the goal state and also the more optimal paths are discovered first. This makes it easy for us to find the best path from the learned information.

One other parameter that decides the performance is the minimum area the robot has to explore before stopping the run. With several tries, I realized that having the minimum limit to around 65% to 70% usually ensured that the robot finds the goal state. However, it remains a issue, that even though the goal state has been found, the robot have missed some better alternatives in the process. Thus, I kept the minimum area to be explored at 70% and have tested the robot for many other minimum area coverage values.

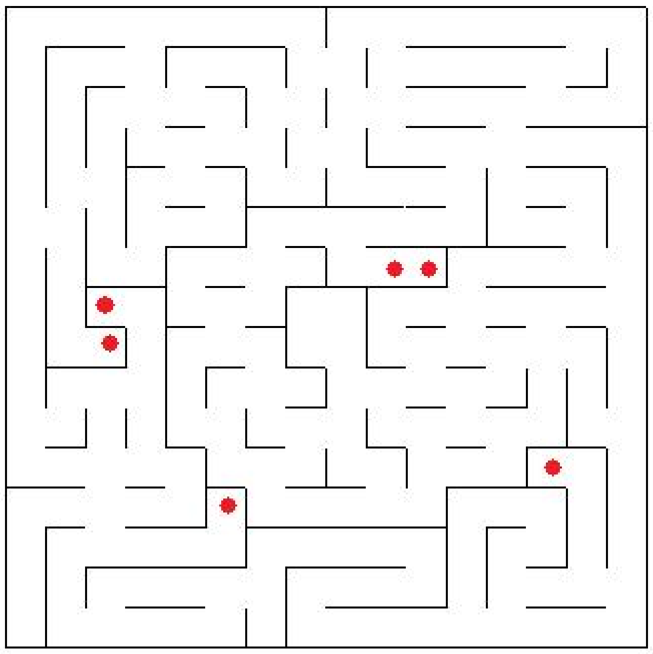


Figure 13. Maze 3 with red dot showing deep dead-ends

4. Results

Model Evaluation and Validation:

The final model designed uses A* for exploration and Dynamic Programming for optimization. This allows the model to discover a more optimal path if not the most optimal path to the goal state. The benefit of Dynamic Programming is that it finds path from every cell to the goal state and not just the starting point.

The model is tested by setting the minimum required area during the first (exploratory) run. The results obtained shows that the model's performance is reasonable and gives satisfactory results while beating the benchmark model. The following table showcases the performance of the robot on various mazes and under various minimum percentage coverage.

		Test Maze 1	Test Maze 2	Test Maze 3
	Dimensions	12 x 12	14 x 14	16 x 16
Minimun Coverage	Benchmark Score	**23.216**	**38.01**	**42.17**
50%	First Run	141	169	196
	Second Run	16	32	28
	Score	**21.700**	**38.667**	**35.567**
55%	First Run	140	179	212
	Second Run	16	32	28
	Score	**21.700**	**39.00**	**36.10**
60%	First Run	140	228	266
	Second Run	16	29	28
	Score	**21.700**	**37.633**	**37.90**
65%	First Run	140	255	290
	Second Run	16	28	28
	Score	**21.70**	**37.533**	**38.70**
70%	First Run	145	265	303
	Second Run	16	28	28
	Score	**21.867**	**37.867**	**39.133**
75%	First Run	169	278	324
	Second Run	16	27	29
	Score	**22.667**	**37.30**	**40.833**
	First Run	183	290	368

80%	Second Run	16	27	29
	Score	**23.133**	**37.70**	**42.30**
85%	First Run	226	308	502
	Second Run	16	25	28
	Score	**24.567**	**36.30**	**45.767**
90%	First Run	275	449	604
	Second Run	16	27	28
	Score	**26.20**	**43.00**	**49.067**
95%	First Run	305	606	673
	Second Run	16	27	28
	Score	**27.200**	**48.233**	**51.467**
100%	First Run	314	658	866
	Second Run	16	28	28
	Score	**27.50**	**50.967**	**57.900**

Justification:

Overall, the model trained performed well. The results obtained are robust and consistent. Even though the model has failed to find the optimal solution, it has outperformed the benchmark model.

Although, it could be argued that the model has failed to find the optimum solution, the results obtained are not too far from the optimum solution. Keeping in mind that the model need to perform well not just on these three mazes but other mazes as well, we can say that the model is robust enough to find a good solution to any maze. The results obtained at 70% coverage (highlighted in yellow) are by far the best. Even though lesser percentage values have given better results, one could argue that for the model to be effective in various situation, it should discover as much area as possible so that it does not lose out on some vital information.

5. Conclusion

Free Form Visualization:

To further test my model, I've created two more mazes named as test_maze_04.txt and text_maze_05.txt with dimension 12 x 12 and 14 x 14 respectively. Figure 14 shows test maze 04 and Figure 15 shows test maze 05. I've tried to create mazes with deep dead ends and convoluted path to make it difficult for the robot to find the optimal solution.

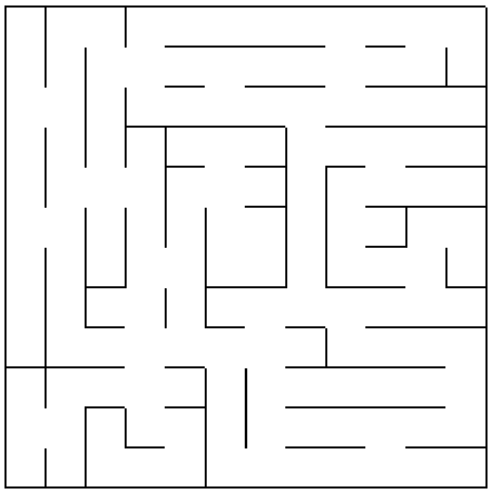


Figure 14. Test maze 04

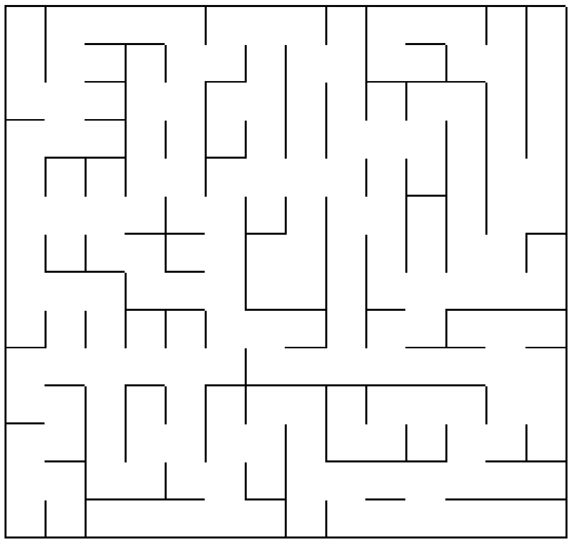


Figure 15. Test maze 05

On executing the robot on maze 04, the exploration run took 248 steps while the optimization run took only 9 steps. The model gave a score of 18.30. Figure 16 shows the optimal path traced by the robot.

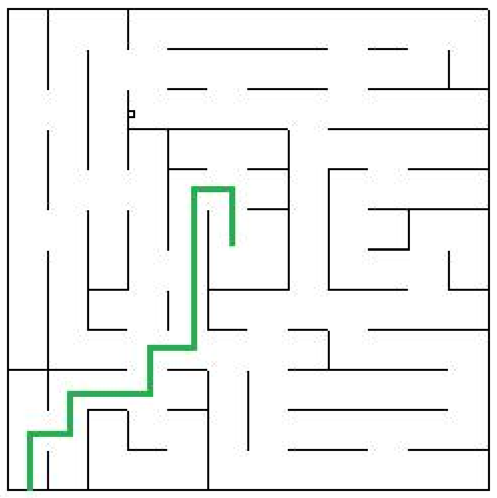


Figure 16. Robot Path for maze 04

On executing the robot on maze 05, the exploration run took 398 steps while the optimization run took only 10 steps. The model gave a score of 24.30. Figure 17 shows the optimal path traced by the robot.

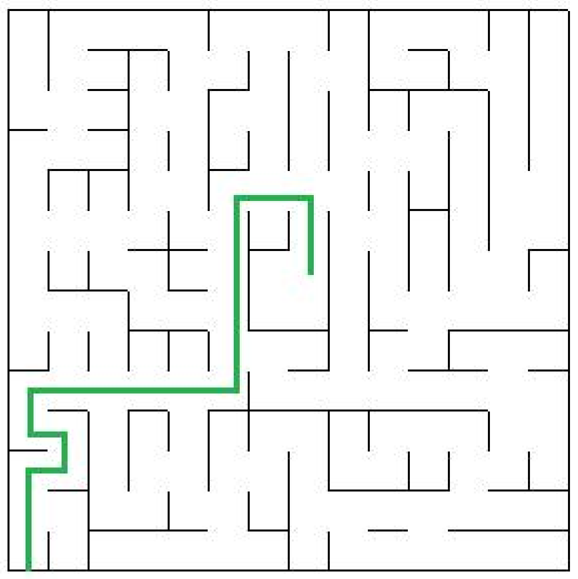


Figure 17. Robot Path for maze 05

Reflection:

The objective of this project was to create an intelligent system that could traverse its way through the maze by following the most optimal path. For me, the project was a challenging one, as I had very little experience with traversal algorithms. This was also one of the reason for taking this project, as I knew that I would learn most by doing something that's beyond my comfort zone.

At the beginning, the sheer complexity of the task was very daunting. However, as I kept making progress, I started to understand the problem and was able to draft a solution. I read books on Machine Learning (Machine Learning by Tom Mitchell was really helpful) and on Artificial Intelligence. Artificial Intelligence: A Modern Approach by Russell and Norvig was a great starter. I'm also thankful to *Udacity's Artificial Intelligence for Robotics* course for teaching the advanced algorithms that played vital role in completing this project.

The most difficult part of this project for me was to come up with an approach for the exploration part. Since, the optimization process depended upon the exploratory process, It was necessary to find a good way to traverse the maze such that we end up gathering most amount of information in least step possible. Once, this problem is solved, the optimization problem became easy to figure out.

Having work on a project of this scope and complexity helped me gain insights into the field artificial intelligence and autonomous systems. Implementing various search algorithms helped me in honing my programming skills and helped me improve my overall software development skills. As the project was very large, I created a workflow for myself. The following flowchart shows my workflow:

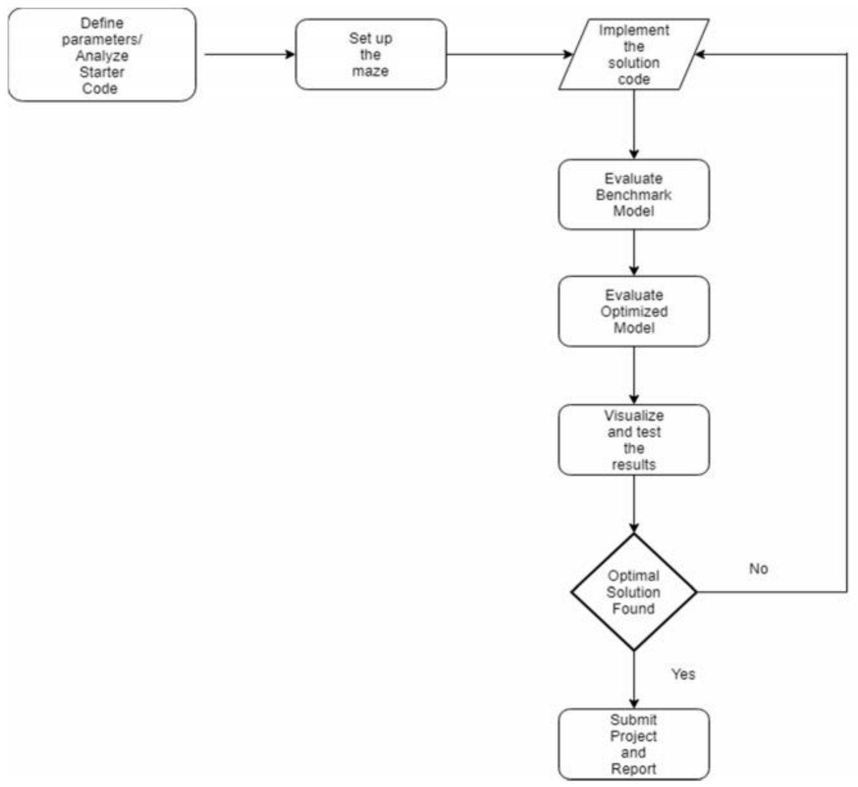

Define parameters/ Analyze Starter Code
Set up the maze
Implement the solution code
Evaluate Benchmark Model
Evaluate Optimized Model
Visualize and test the results
Optimal Solution Found
No
Yes
Submit Project and Report

Figure 18. Flow Chart for Workflow

The whole solution can be summarized in following steps:

- Get the position of the robot and plan the next move based on sensor values.
- Keep track of the cells visited by the robot.
- Based on the information gathered during first run, extrapolate the best solution for second run.
- Traverse the extrapolated path.

While working on this project, I encountered various obstacles. Sometimes the solution was elementary other times not so elementary. There were problems with robot exceeding step limit, going into infinite loop or failing to beat the benchmark values to name a few. However, with enough tries and tinkering, I was able to put together a solution. To be honest, I don't believe that I solution that I provided is the best there is. This would certainly be a hyperbole. However, my solution is robust enough that one can consider it to be a good solution if not the best solution to this problem. Having the solution performed on additional two mazes proves my point.

I believe the skills I gained during this project will proved to be beneficial in my future learning and endeavors.

Improvements:

Even though this project is complex, the complexity has no match to what real world situations has to offer. This leaves a room of enormous improvements in this project.

For one, we start by adding a sensor to the back of the robot. Currently, the robot could only detect spaces in front, right and left. Having information about the back cell will certainly help it will reduce the number of rotations and movements the robot has to make. The robot could also be allowed to move diagonally. This will also improve its performance.

This particular problem is defined as such that the robot takes discrete number of steps. However, in real world, things are more complex. In that respect, the model has to be improved so that it can work with continuous values.

To be more specific towards my solution, I believe that there are number of improvements that I can make. First, both the methods that I've used A* and Dynamic Programming are computationally involved. There is a scope of using less intensive method that can give us the same results. Secondly, a single algorithm does not provide good solution in all situation. To overcome this, we can have the system to deploy multiple search algorithms simultaneously to get to the best possible answer. The performance of the model is greatly dependent upon the information garnered during the first run. Thus, we can improve the system such that it effectively evades dead ends and each cell is visited a minimum number of times.

Thus, there are a number of improvements that could be applied so that the system becomes more robust and performs well in both constrained and real world situations.

Resources:

Udacity Project Files:

https://www.google.com/url?q=https://drive.google.com/open?id%3D0B9Yf01UaIbUgQ2tjRHhKZGlHSzQ&sa=D&ust=1523961264287000

Micromouse Wikipedia Page:

https://www.google.com/url?q=https://en.wikipedia.org/wiki/Micromouse&sa=D&ust=1523961264283000

Apec 2014 Micromouse Texas:

https://www.youtube.com/watch?v=0JCsRpcrk3s

Dijkstra's Algorithm:

https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/

A* algorithm:

https://www.geeksforgeeks.org/a-search-algorithm/

Depth-first Search:

https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

Breadth-first Search:

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

Dynamic Programming:

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