Plot and Navigate a Virtual Maze

I. Definition

Project Overview

This project, the robot motion planning, in layman's terms, is about making a mobile robot follow the fastest or shortest way from one location to another after exploring the possible ways between the two locations inside a maze from the first run.

This project originated from Micro mouse, wherein the robot mouse may make multiple runs inside a given maze. In the first run, the mobile robot tries to map out the maze to not only find the center but also figure out the best paths to the center.

In this project, the maze size, the robot initial location, and the goal bounds are given as input in robot.py module. The specifications of the maze and virtual robot are provided. The robot can measure a distance to the walls with its sensors. I should implement the robot.py module to control a virtual robot to navigate a virtual maze such that it can rotate to avoid walls, and move forward or backward to explore maze cell by cell all the way to the goal.

The robot's movements and the environment are discrete and deterministic. I implemented the robot module to make the robot steadily gain and save information about its environment during the first run. The robot use that information to choose and follow the optimal route.

Problem Statement:

The robot can make two runs. The first run is for the robot to learn possible paths to the goal from the initial position or even from any position inside the maze. Therefore, the robot can take its time to explore the maze. The second run has the most significant impact on the outcome. The robot may choose to end the first run after finding the goal or to continue the exploration to visit the other cells inside the maze. The time spent on both runs are used to evaluate the robot's policy performance. On the first run the robot use information from the sensors before each move in the maze to construct an internal representation of the maze and save its location and heading after each move. I used A* method to explore the maze cells and dynamic programming to find the optimal path. I create tables of the same size as the virtual maze to keep track of each maze cell number of walls, the possible move directions, the distance from each cell to the goal, and the shortest path to the goal.

Metrics

The score is evaluated as the number of steps taken by the mobile robot in the first run divided by thirty plus the number of steps taken in the second run. The steps are the number of move the robot makes in each of the two runs. The time it takes for each step is the same. The time of a step is the time it takes for the robot to move from one cell to the next cell. The less steps it takes the robot to reach the goal from initial position the better the metrics performance is. The total number of steps for both runs should be limited to 1000 steps. The performance score is calculated as follow:

Final score = (Time spent on the First run / 30) + (Time spent on the second run)

II. Analysis

Data Exploration:

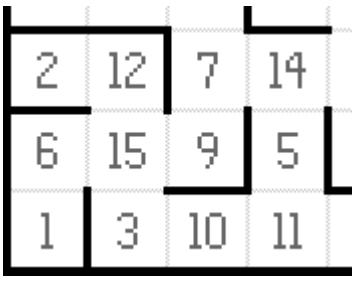
To get better idea of this project, I carefully read maze.py, showmaze.py, tester.py and robot.py modules. I realized that for the robot to be able to move around inside the maze, it must know its position, its surrounding (obstacles and no obstacles) and its direction. Therefore, the sensors, the maze dimension, the location and the heading from robot.py module and the global dictionaries for robot movement and sensing variables, called dir_sensors and dir_move from tester.py module served as hint on how to approach this project.

I ran python tester.py test_maze_01.txt in the command line after writing the following codes in robot.py next_move method:

print len(self.maze_dim)

print sensors

The command output the maze length of 12 and the sensors' wall detection values with zero and non-wall detection values greater than zero. It worth noting that the lengths of 14 and 16 were outputted for test_maze_02.txt, test_maze_03.txt respectively. Thus, the maze dimension changes depending on maze text file. This output was like the grid length with 0 for navigable grid-cell, 1 for non-navigable grid-cell in Artificial Intelligence for Robotics course(AIR), precisely the A* and dynamic programming sections. Though there are similarities between this project and those two sections, I realized that they are different in some prospects, such as the goal of this project is located not at the end of the grid as it is in those sections but in the center of the grid. This means that the heuristic method should be implemented a little bit differently such that it counts from the center of the maze out in all four regions of the axis (The goal in the center is 2 by 2). In this project, unlike A* and dynamic programming implementation in AIR, the robot needs to rotate to turn its heading toward a given direction to move to that direction of course if no obstacle is in that direction. The last challenge in this project is that, unlike in AIR, the wall is not a grid cell but an obstacle (a wall) in between some maze-cells as the following example illustrates: The bold segments represent the walls and other segments are open.



Exploratory Visualization:

heuristic

[[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]

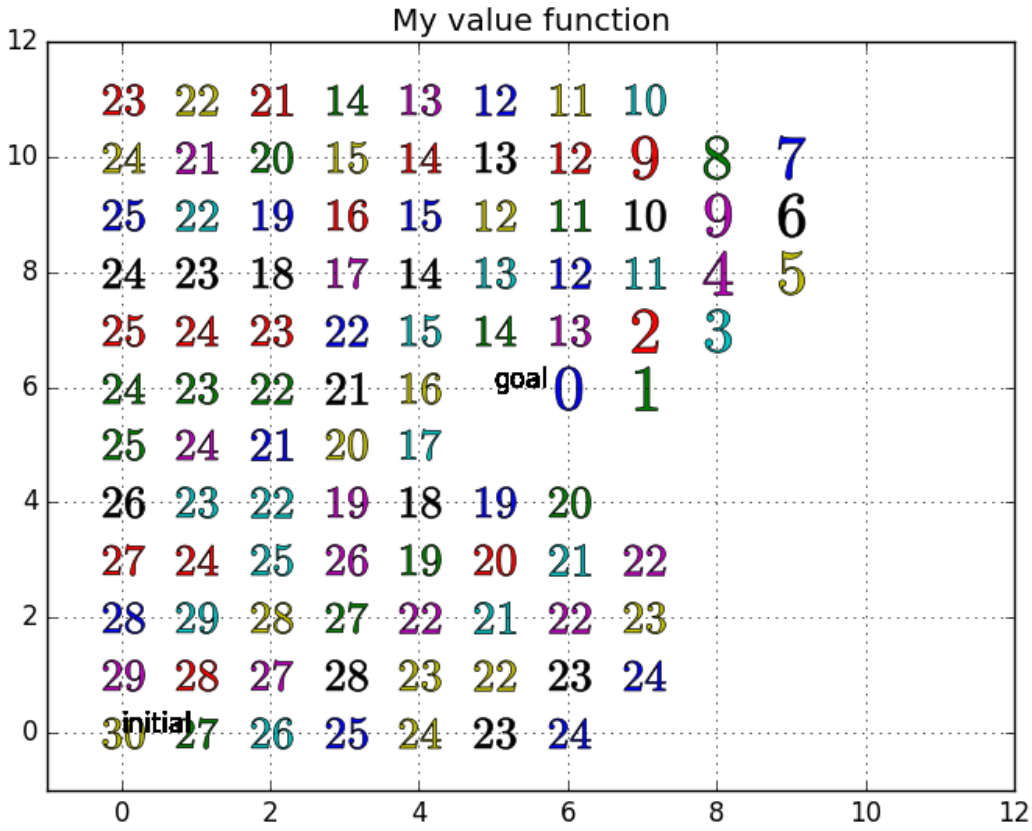
[7 6 5 4 3 2 2 3 4 5 6 7]

[8 7 6 5 4 3 3 4 5 6 7 8]

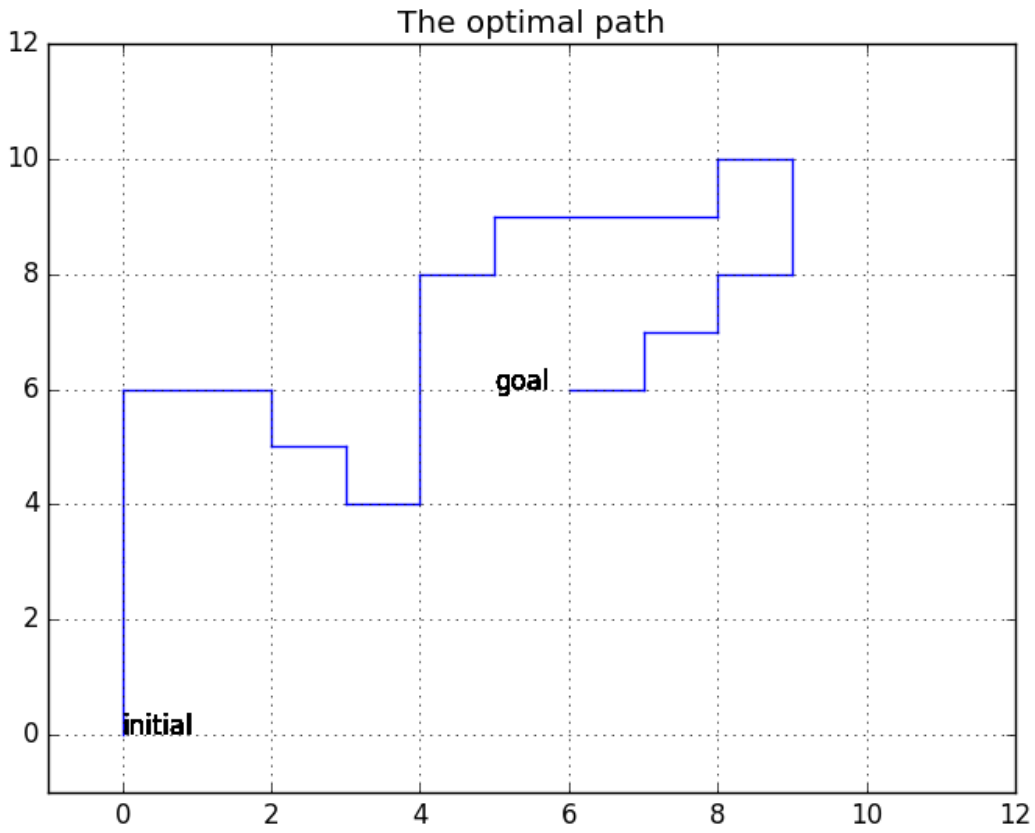
[9 8 7 6 5 4 4 5 6 7 8 9]

[10 9 8 7 6 5 5 6 7 8 9 10]]

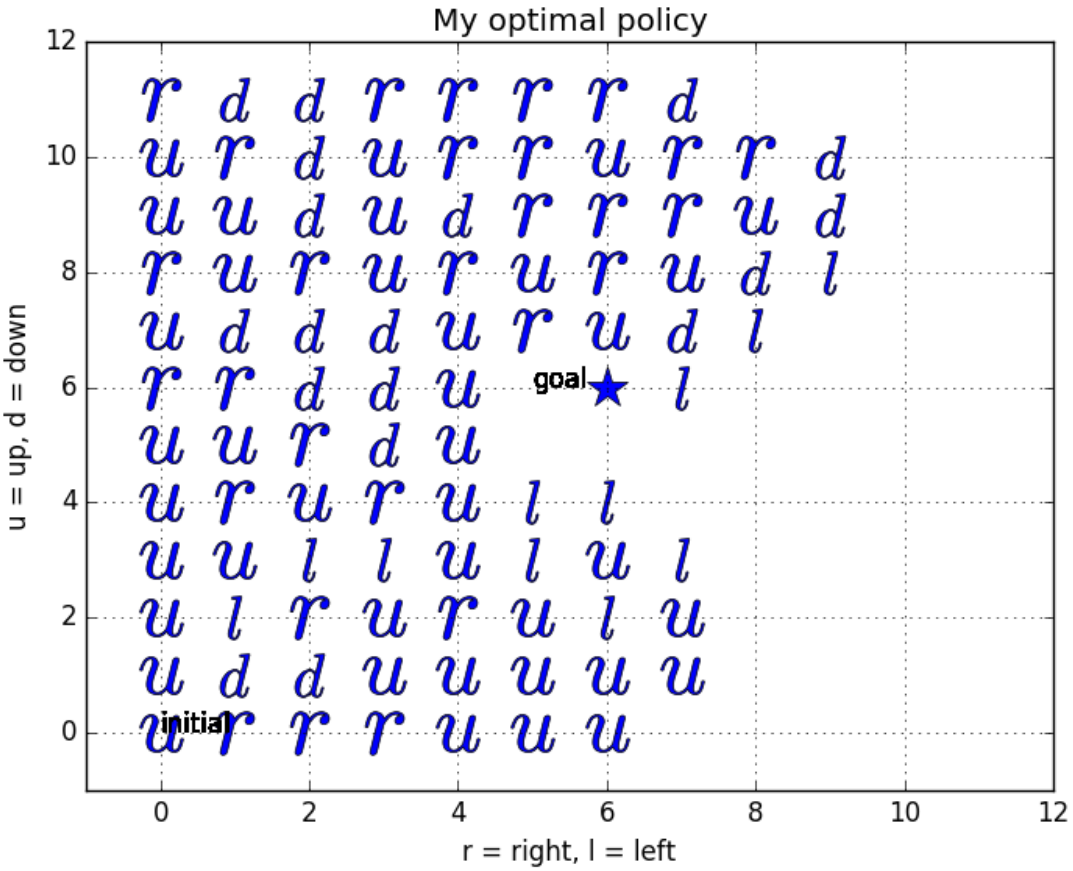
The heuristic is a representation of the virtual maze that determines how far each cell is from the goal. If you take look at the four zeros in the center (the goal bounds), you can realize that each zero in the center is followed by 1, 2, 3 and so on, horizontally and vertically without regards to whether there is a wall or not in between any two given cells. If the robot was at 7 in any given region of this heuristic, the logical next step for the robot would be to go to the cell with 6 toward the goal but not 8 which is away from the goal. But given that there might be a wall between 7 and 6 makes me create additional implementations that will consider whether there is wall or not between any two cells.



The above value function takes into account the internal structure of the maze. For example the initial position in this value function is lebelled 30, since there is a wall between the initial position's cell and the immidiate cell at its right side, and there is no wall toward the cell at the top thus the cell at the top is labelled 29 meaning that the robot, to avoid the wall, should go to 29 cell first, then 28, then 27.



The above optimal path is the path with least number of steps among all the possible paths between the initial position and the targeted goal postion. The optimal path can change depending on the internal structure of the virtual maze. The straight segment from initial (0, 0) to maze(0, 6) means that the robot should make consecutive moves upward or to the north. It should head east or rotate to the right to move from maze(0, 6) to maze(2, 6) and so on.



The above optimal policy is implemented to assist the robot to rotate its heading toward an open way to move to a cell that will lead to the goal without hitting the wall and in less steps. “U” means the robot should turn its heading toward up before moving. Similarly, “d” means down, “r” means right and “l” means left and the star sign means the goal.

Algorithms and Techniques:

The number of steps is limited to one thousand for the two runs. To accomplish this, the following implementations were made:

- Build a maze grid based on maze dimensions
- create a heuristic method to guide the robot direction toward the goal.
- Keep track of robot’s position based on the next move for the exploration and exploitation phases.
- Make policies to uncover the wall and find an optimal solution

How I came up with these implementations to make the robot search and find the goal on the first run, then use the information gained on the first run to follow the shortest path to the goal on the second run as follows:

I first realized that I should implement a method that will save the internal structure of a given virtual maze by saving information on no wall or wall at every direction of every cell the robot visits in the maze. To make the navigation easier the robot must know where no wall and walls are. I called this method maze_number. The sensors' inputs were critical for this implementation. I set the sensor input to one for no wall and zero for wall. And depending on the robot heading, I set the sensors' coefficients as indicated in maze specifications as follows:

The 1s register corresponds with the upwards-facing side, the 2s register the right side, the 4s register the bottom side, and the 8s register the left side. For example, the number 10 means that a square is open on the left and right, with walls on top and bottom (1*sensors[0] + 2*sensors[1] + 4* back + 8* sensors[2] = 10). Note that the robot is facing up in this example, the bottom is the back of the robot and because there is a wall at the bottom in this case, back=0, therefore 4*back=0 and similarly there is a wall at the top, 1*sensors[0] = 0. No wall on the left and right, sensors[1] = 1 and sensors[2] = 1, thus 2*sensors[1] + 8*sensors[2] = 2*1 + 8*1 = 10. This example from the maze specifications allowed me to verify the accuracy of this method.

Because the robot has no sensors mounted on its back and it can move forward and backward but not left and right wards, unless it rotates to face either side, like cars, I consider the forward movement to be the main move direction and I implemented a method that keep track of robot location and headings as it moves through maze navigable cells looking for the goal location. The global dictionaries for robot movement already implemented in tester.py module, such as dir_sensors, dir_move were good hints in implementing this method. The sensors directions' dictionary or dir_sensors, for example, defines the loop directions of the three sensors for all possible heading directions as a list for each heading. Let's say the robot is heading up, one sensor will sense left, the second sensor will sense up and the third one will sense right, thus this key-pair value 'up': ['left', 'up', 'right']. Therefore, if the robot is facing upward no available sensor will be sensing downward. For the robot to sense down, it must rotate to face either right, left or down. If the robot rotate or faces right, one sensor will sense up, another sensor will sense right, and the third sensor will sense down, thus the key-pair value 'right' : ['up',' right', 'down'] . The dir_move dictionary is the coordinates or move vectors for up, down, left, and right. For example to move up one unit the ordinates x = 0 and y=1, thus, the key-pair value 'up': [0, 1]. If the robot senses its environment again and finds no wall at a given angle, it should rotate with that angle to have its heading toward that direction and then move, thus, the input of this method is rotation and movement variables. I define the three rotation indices And I set the new heading variable to the dir_sensors from the previous heading to the new heading caused by new rotation. And I set the new location to be the sum of the previous coordinates and the new coordinates time the movement unit, because the movement can vary from [-3, 3] and movement = 1 can move the robot forward from one maze cell to another. Movement = -1 can possibly move the robot backward from one cell to another.

To make the search for the goal position more promising and to better contribute in guiding the robot accomplish that task, like I mentioned in exploratory visualization above, I define a method called make_heuristic method to determine how far the robot is from the goal at each step of the exploration.

The interactive ipython was my best friend in accomplishing this goal. I created a 12 by 12 array to figure out how to get zeros at the center of the maze for each quadran up to 11 using for loop, and absolute value intead of square root distance calculation method, then I replaced 12 by maze_dim in the for loop and the output was terrific.

Thanks to scipy and matplotlib documentations, I could implement the plotting methods throughout the project such as plot_robot_path, plot_value methods and so on for visualization.

The implementation was going to be easy if the internal structure of the maze was defined and uniform, meaning that for example every cell had only two opening sides, one for entrance and one for exit. But since that was not the case, I had to implement a method called compute_next_step method to guide the robot toward the goal based on the sensors inputs, the robot current position and heading find the next step without exiting the maze and to rotate to the appropriate heading. Again, dir_move dictionary and sensors indexing came handy in this implementation. All I had to do was to tell the robot that if you are in a cell and your sensors tells you that there is no opening at either front, right and left side, that's a dead end, turn around or rotate first but if your sensors tells you that there is an opening at a given direction if the cell in that direction is inside the maze and your current cell is not the goal cell, then rotate your heading to that direction and your next step is to move to the cell in front of you and if you have many openings sides for a given cell, just use the heuristic to see which one is closer to the goal first and rotate to head toward the cell closer to the goal and move to that cell. Also save the coordinates of the locations you go to. Thus, this method takes as inputs its current position and sensors' readings and output the appropriate rotation and movement.

My curiosity went as far as to ask myself how can I implement the robot.py such that if the robot finds the goal and for some reasons the internal structure of the maze changes or the initial position changes?

I got this incredible idea from Udacity Artificial Intelligence for Robotics course in which they combine heuristic, the steps, and obstacle, no-obstacle cells but in this case the sensors inputs to come up with a method that can assist the robot to find the best path to the goal not only from the initial position but also from any position in the maze. I called this method compute_value_function method. The heuristic is like a map of the maze, and the basic idea is that for any location the robot is inside the maze and any of the four directions it may be heading rotate to the appropriate angle and move to the cell that lead to goal. It is pretty much the implementation as in next_step method but this method is more broad to any position and any direction the robot may be heading to. In this method, I saved the x and y positions that yield the shortest path to the goal that I called optimal path. As I explained earlier, the dictionaries and list indexing are key to this implementation.

I defined a method where all the exploration takes place, called exploration_phase, and another method called exploitation_phase where optimal path to the goal found after exploration is used to find the goal in possible minimum steps.

I called the exploration_phase method for the first run and the exploitation_phase method for the second run in next_move method to reach this increadible result.

The advantage of these algorithms and techniques are that no matter the initial position, the robot mouse will follow the shortest path to the goal. The cons of these algorithms and techniques are that it may take more code implementation and possibly larger memory space than algorithms and techniques

that are solely concerned with finding a path between the two static positions (initial and goal) in this problem. But when the robot finds itself in a location other than this initial position or to find a different goal, my algorithms and techniques will still prevail.

Benchmark:

Robot should avoid illegal move or moving while heading toward a wall to follow a path to the goal.

The robot can obtain the best score by following the optimal path in the first and second run. I mean the minimum steps possible is when the first run follow the shortest path from initial position to the goal position without any zero move in another without hitting the wall or any dead end. The time it takes to achieve such result is:

Score (lower bound) = optimal time + optimal time/30

In the other hand if the robot explores all cells in the maze on the first run and the optimal path on the second run, such score is:

Score (upper bound) = optimal time-step + time-step to visit all cells/30

II. Methodology:

Data Preprocessing:

No additional data preprocessing is necessary because the robot specifications, environment and sensors that detect the environment are given. The virtual robot rotates, moves, senses and determines the distance to the walls without any problem. The virtual maze was also provided.

Implementation:

I define a few methods to assist the robot to:

- discover and avoid the walls
- update its position and heading
- explore paths between the initial and goal locations in the exploration phase
- follow the path with the minimum steps between initial and goal positions in exploitation phase
- plot the optimal path and policy to visualize the moves.

The methods implemented are:

- maze_number(self, sensors, back): To compute the wall number that describes which edges are closed or open at each position from the robot's sensors inputs.
- update_robot_heading_location(self, rotation, movement): To Update robot heading and location.
- make_heuristic_grid(self,maze_dim): to define heuristic function for any maze layout. i.e. it is only based on the distance goal. Assign 0 at the center and incrementally higher number as span out away from the center.

- plot_robot_path(self,x1,y1,x2,y2,number,i): To plot each step path and display the assigned step number.
- compute_next_step(self,x1,y1,sensors): To guide robot rotation and movement such that it doesn't hit the wall which could increase the number of steps and time to reach the goal.
- compute_value_function(self, goal): Thanks to dynamic programming method used to find the optimal path in Artificial Intelligence For Robotics course mix with some additional code blocks to read the maze map; decode the wall number into 4 digits binary; and check which direction is open.
- plot_value(x,y,number,i): To customize the graph of my plots.
- exploration_phase(self,sensors): To keep track of robot's positions, directions, and paths during the training session or first run. This method uses most of the previous methods to explore and save the necessary information about every maze cell the robot mouse visited during this run.
- exploitation_phase(self,sensors): This method used the information learnt from the first run to choose the path with the possible minimum steps between the initial and goal location.
- next_move(self, sensors): To use the exploration_phase() method for the first run and the exploitation method for the second run.

The attributes I used are:

- self.maze_grid = np.zeros((maze_dim, maze_dim), dtype=np.int) for Number for wall description at each maze.
- self.path_grid = np.zeros((maze_dim, maze_dim), dtype=np.int) for the number of the repeated path.
- self.exploration_path_symbol_grid = [[' ' for _ in range(maze_dim)] for _ in range(maze_dim)] for optimal path direction with symbol.
- self.exploration_path_grid = [[' ' for _ in range(maze_dim)] for _ in range(maze_dim)] for optimal path direction with 'up','left', right, down.
- self.exploration_path_value = [[' ' for _ in range(maze_dim)] for _ in range(maze_dim)]
- self.path_value = [[99 for _ in range(maze_dim)] for _ in range(maze_dim)] # value assigned from goal to start.
- self.optimal_policy = [[' ' for _ in range(maze_dim)] for _ in range(maze_dim)]
- self.goal_bound = [maze_dim/2 - 1, maze_dim/2] # destination area - center of the maze

Refinement:

The robot did not find the goal when I first implemented it but it kept going from cell to cell without knowing which direction to take to find the goal until the robot runs out of time. I did implement next_move method of the robot.py module based on the sensors inputs only. I typed in next_move method the following codes:

print the "robot length is : ", len(self.maze_dim)

print "sensors inputs: ", sensors

if sensors[0] > 0:

rotation = -90; if sensors[1] > 0:

rotation = 0; if sensors[2] > 0:

rotation = 90; if sensors == [0, 0, 0]:

rotation = 90

movement = 1

The output was a series of repeated robot length is: 12 and different sensors' output

For example, sensors inputs: [0, 7, 0] and so on. Then, fails to reach the goal. I went back to my online classroom and I took the Artificial Intelligence for robotics course where I found the algorithms and techniques to instruct the robot to not only recognize the goal but also to map the maze and guide the robot to explore the maze and find the shortest path to the goal called A* method and even further the dynamic programming method. I then realized that I should define other methods for the exploration of maze during the first run and for finding the shortest path to the goal on second run.

IV. Results

Model Evaluation and Validation:

The final model qualities are robust because this model has been implemented to dynamically find the goal. This means that the virtual robot mouse can not only find the goal from any location but also figure out the shortest path in between the two locations after the first run.

Maze number	Steps in the first run	Steps in the second run	Final score
1	215	20	27.167
2	258	31	39.600
3	200	29	35.667

Justification:

Maze number	Steps in the first run	Steps in the second run	Total Steps
1 →	215	20	235
2 →	258	31	289
3 →	200	29	229
Maze # 1 + 2 + 3	673	80	753 < 1000

The total steps taken by the mobile robot inside the first, second and third mazes in both first and second run are: 753 steps. Which is incredibly less than 1000 steps.

V. Conclusion

Free-form Visualization:

exploration phase with symbols indicating the robot's move directions while exploring the cells of a 12x12 maze

[['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', '>', ' '], [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '<', '^', '^', ' '], [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']]

A 12x12 maze grid indicating the walls and open directions before reaching the goal

[[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 0 0 10 11 6 10 10]

 [9 7 12 11 12 0 14 9 14 11 13 14]

 [0 13 5 12 0 0 13 6 9 14 3 14]

 [0 0 0 0 0 0 0 13 6 9 14 0]

 [0 0 0 0 0 0 0 0 15 7 14 0]

 [0 0 0 0 0 0 0 0 0 0 0 0]

 [0 0 0 0 0 0 0 0 0 0 0 0]]

The path taken by the robot before reaching the goal in a 12x12 maze

[['u', 'u', 'r', 'd', 'd', 'd', 'd', 'd', 'd', 'd', 'd', 'd'], ['r', 'd', 'd', 'u', 'u', 'u', 'r', 'd', 'u', 'u', 'u', 'l'], ['r', 'r', 'l', 'l', 'l', 'd', 'u', 'r', 'u', 'u', 'l', 'd'], ['r', 'u', 'l', 'l', 'r', 'd', 'd', 'd', 'l', 'd', 'd', 'd'], ['u', 'u', 'r', 'u', 'u', 'u', 'u', 'r', 'd', 'd', 'r', 'l'], ['r', 'd', 'u', 'l', 'r', ' ', ' ', 'r', 'l', 'd', 'r', 'l'], ['u', 'u', 'l', 'r', 'd', ' ', 'l', 'u', 'r', 'l', 'd', 'l'], [' ', 'l', 'd',

'd', ' ', ' ', 'l', 'd', 'u', 'r', 'u', 'l'], [' ', ' ', ' ', ' ', ' ', ' ', ' ', 'l', 'd', 'u', 'r', ' '], [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'l', 'd', 'd', ' '], [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']]

exploration phase with symbols indicating the robot's move directions while exploring the cells of a 14x14 maze

[['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', '<', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '], [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']]

A 14x14 maze grid indicating the walls and open directions before reaching the goal

[[1 5 5 7 7 5 5 6 3 6 3 5 5 6]

 [0 0 0 0 9 5 5 15 14 11 14 3 7 14]

 [0 0 0 0 1 7 6 10 10 10 11 12 0 10]

 [0 0 0 0 3 12 11 14 11 14 10 3 5 14]

 [11 5 6 0 11 7 12 8 10 9 12 9 7 0]

 [11 7 13 7 14 11 5 5 13 5 4 3 13 6]

 [8 9 5 14 9 12 0 7 6 3 6 11 6 10]

 [3 5 5 14 3 6 0 0 11 12 10 10 10 10]

 [10 3 5 13 14 10 0 0 0 0 14 8 9 14]

 [9 14 3 6 11 14 0 0 0 0 10 3 6 10]

 [3 13 14 11 14 0 0 0 0 0 13 14 10 10]

 [10 3 15 12 0 0 0 0 0 0 0 0 11 14]

 [11 12 0 0 0 0 0 0 0 0 0 0 0 0]

 [0 0 0 0 0 0 0 0 0 0 0 0 0 0]]

The path taken by the robot before reaching the goal in a 14x14 maze

[['u', 'u', 'u', 'u', 'u', 'u', 'u', 'r', 'u', 'r', 'u', 'u', 'u', 'r'], [' ', ' ', ' ', ' ', 'l', 'd', 'd', 'r', 'l', 'r', 'l', 'r', 'd', 'r'], [' ', ' ', ' ', ' ', 'u', 'r', 'd', 'r', 'l', 'r', 'l', 'd', ' ', 'r'], [' ', ' ', ' ', ' ', 'r', 'd', 'l', 'd', 'l', 'r', 'l', 'r', 'd', 'd'], ['r', 'd', 'd', ' ', 'r', 'u', 'l', 'l', 'l', 'u', 'l', 'u', 'r', ' '], ['u', 'u', 'u', 'u', 'r', 'u', 'u', 'u', 'l', 'd', 'd', 'r', 'd', 'd'], ['l', 'l', 'd', 'd', 'u', 'l', ' ', 'd', 'd', 'r', 'd', 'u', 'r', 'l'], ['r', 'd', 'd', 'l', 'u', 'r', ' ', ' ', 'l', 'd', 'l', 'l', 'r', 'l'], ['r', 'u', 'u', 'l', 'l', 'r', ' ', ' ', ' ', ' ', 'l', 'l', 'u', 'r'], ['u', 'l', 'r', 'd', 'r', 'd', ' ', ' ', ' ', ' ', 'l', 'r', 'd', 'r'], ['r', 'l', 'd', 'l', 'd', ' ', ' ', ' ', ' ', ' ', 'l', 'd', 'l', 'r'], ['r', 'u', 'u', 'l', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'l', 'd'], ['u', 'l', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '], [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']]

A 16x16 maze grid indicating the walls and open directions before reaching the goal

[[1 5 5 6 3 7 5 5 5 5 7 0 0 0 0 0]

 [3 5 6 10 10 0 0 3 5 5 13 7 0 0 0 0]

 [11 6 11 15 15 5 14 8 2 3 5 13 5 6 0 0]

 [10 10 10 10 11 5 13 5 12 9 7 6 3 15 0 0]

 [10 10 10 9 12 3 5 6 0 0 10 11 14 0 0 0]

 [9 14 9 4 0 13 6 11 14 0 9 12 11 0 0 0]

 [1 13 6 0 0 3 15 12 9 15 6 3 13 0 0 0]

 [3 6 10 0 0 14 8 0 0 0 10 9 7 0 0 0]

 [10 10 10 0 0 13 7 13 0 3 14 3 13 0 0 0]

 [10 10 10 0 0 0 0 0 0 10 10 10 0 0 0 0]

 [10 9 12 0 0 0 0 0 0 8 10 10 0 0 0 0]

 [11 5 5 6 0 0 0 0 0 0 9 13 0 0 0 0]

 [11 7 6 10 0 0 0 0 0 0 0 0 0 0 0 0]

 [10 10 9 12 0 0 0 0 0 0 0 0 0 0 0 0]

 [10 11 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

 [9 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]

Reflection:

I love and I am passionate about learning or improving any idea that addresses the issue of making machine or any object smart enough to do anything for our high-level well-being. I implemented this project to make a moving object, the robot mouse, smart enough to uncover a simplified world, a virtual maze, to choose the best or the shortest way between two given locations.

The most challenging part of this project was to implement the robot module to make the robot avoid walls among navigable maze cells. If the walls were cells, it would have been easier. But the for the fact that the cell behind the wall on the right for example can be visited by the robot by implementing the robot to rotate its heading to up to move to the next cell on top, rotate its heading to the right to move to the top right cell then rotate its heading to down and finally move to that cell to explore that cell. It could also be easier if the maze was fully observable. But such method of implementation would of hard coding methods. I think obscuring the maze is better because it pushed me to implement the robot to navigate in any similar maze no matter its internal structure. One robot implementation to navigate many mazes is better than to have different robot implementation for navigating different mazes.

Improvement:

If we were to consider the scenario took place in a continuous domain. For example, each square has a unit length, walls are 0.1 units thick, and the robot is a circle of diameter 0.4 units we would need algorithms like PID, smoothing, and SLAM.

I gained knowledge from Udacity's Artificial Intelligence for Robotics courses. I also implemented some codes based on this project requirements such as the goal being at the center and the distance from the center to others cells in the maze as an important factor for the heuristic function to guide the robot toward the goal.