Robot Motion Planning

Report for the Capstone Project

Neeleshkumar Srinivasan Mannur

Udacity Machine Learning Engineer Nanodegree

Contents

DEFINITION

Project Overview

The project takes inspiration from Micromouse competitions, wherein a robot mouse is tasked with plotting a path from a corner of the maze to its centre in an unknown environment. The robot mouse may make multiple runs in each maze. In the first run, the robot mouse tries to map out the maze to not only find the centre, but also figure out the best paths to the centre. Simply put, the mouse is in an exploratory mode trying to explore various ways to reach the centre. In subsequent runs, the robot mouse attempts to reach the centre in the fastest time possible, using what it has previously learnt. This project will create functions to control a virtual robot to navigate a virtual maze. The goal is to obtain the fastest times possible in a series of test mazes from a simplified model which is provided along with the specifications for the maze.

I could see that Q Learning Algorithm is used for solving the maze problem. This encourages me to take up this problem and solve it with some other algorithm

Source: http://ieeexplore.ieee.org/document/5967320/?reload=true

I have used the dataset from the Udacity Project files for solving the problem.

Problem Statement

The goal is to get to the centre of the maze as fast as possible. The robot will start in the bottom-left corner of the maze and must navigate from the start point to the goal in a reasonably short possible time, and in an optimal number of steps or moves. The maze is a square with equal rows and columns where the number of squares along each side (n) should be either 12, 14, or 16. The circumference / perimeter of the maze is surrounded by walls that act as a preventing barrier, causing the micromouse not to move outside the maze.

The origin will have barriers on 3 sides viz. left, right, and back. This will cause the micromouse to move only in one direction i.e. forward only. The goal zone will be a 2x2 square in the centre of the maze. The micromouse will run a first trial, exploratory in nature to understand the structure and shape of the maze, including all possible routes to the goal area. The micromouse must travel to the goal area to complete a successful exploration journey, but can continue its exploration after reaching the goal. The second trial is where the action happens. In this trial, the micromouse will recall the information from the first trial and will attempt to reach the goal area in an optimal route.

The micromouse's performance will be scored by adding the following:

1. Number of total steps in the exploration trial divided by 30 [Specification in Project Files]
2. Total number of steps to reach the goal in the optimization (second) trial [Specification in Project Files]

A maximum of one thousand time steps are allotted to complete both runs for a single maze. The micromouse can only make 90 degree turns in either clockwise or counter clockwise directions and can move up to 3 spaces forward or backward in a single move.

Metrics

We need to find the optimal score to reach the destination from the origin. The below evaluation metric considers the second trial as the prime factor for evaluating the performance. The score is impacted by both exploration trial and the optimisation trial, however, the optimisation trial impacts the score to a significant level. The exploration trial is capped to 1000 steps. Under any case we are not going to cross that step barrier. Also, the main motive of the exploration trial is to discover the maze grid so that an optimal route can be decided. The main crux is enforced over the optimisation trial since this is where we are looking for the shortest / optimal path. This value is an important benchmark. In order to normalise the score with an average value, we consider 1/30th of the number of steps during the exploration trial as an average value. This is why I have selected the given metric for benchmark evaluation.

The main evaluation metric to determine the performance of the benchmark and solution model is

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

To demonstrate with an example, let us consider the micromouse takes 600 steps during the first trial and 30 steps in the second trial, then the score will be

$$30 + (600 / 300) = 32$$

ANALYSIS

Data Exploration

The code to start off the project was provided by Udacity. They included the following files

1. **maze.py**: From the documentation within the script, it consists of functions that find the distance to the wall, whether it is permissible to pass a cell or not, etc.
2. **showmaze.py**: Creates a visual layout of each maze.
3. **robot**.**py**: The main class where all the code related to the robot is written. The crux of the project is this class.
4. **tester**.**py**: The tester code to test the functionality of the robot
5. **test_maze_xx.txt**: CSV files consisting of Maze information such as layout, dimension, etc.

Examining the project files, we find out that the maze layout is available (test_maze_xx.txt). I will go ahead and use the maze files provided in the Udacity project files. The following are the observations from the data files.

1. First line determines the number of squares for each side in the maze.
2. Subsequent lines determine structure of maze via comma-separated binary values that define which sides have walls and which have openings.

To make a move, the micromouse must have following information –

1. The micromouse's exact location in the maze
2. The number of walls surrounding it
3. What is the intended action
4. Any previous knowledge and routes favouring the maze from prior trials.

The location of the micromouse will be defined by a pair of 2 digits, such as [1,10]. Each action contains movement which will be a number from -3 to 3, and rotation which will be -90, 0, or 90.

Brief Description of the Input [Source: Udacity Project Files]

The maze exists on an *n* x *n* grid of squares, *n* even. The minimum value of *n* is twelve, the maximum sixteen. Along the outside perimeter of the grid, and on the edges connecting some of the internal squares, are walls that block all movement. The robot will start in the square in the bottom- left corner of the grid, facing upwards. The starting square will always have a wall on its right side (in addition to the outside walls on the left and bottom) and an opening on its top side. In the centre of the grid is the goal room consisting of a 2 x 2 square; the robot must make it here from its starting square to register a successful run of the maze.

Mazes are provided to the system via text file. On the first line of the text file is a number describing the number of squares on each dimension of the maze *n* which is either 12, 14 or 16. On the following *n* lines, there will be *n* comma-delimited numbers describing which edges of the square are open to movement.

```
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
```

Each number represents a four-bit number that has a bit value of 0 if an edge is closed (walled) and 1 if an edge is open (no wall); 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 (0*1 + 1*2 + 0*4 + 1*8 = 10). Note that, due to array indexing, the first data row in the text file corresponds with the leftmost column in the maze, its first element being the starting square (bottom-left) corner of the maze.

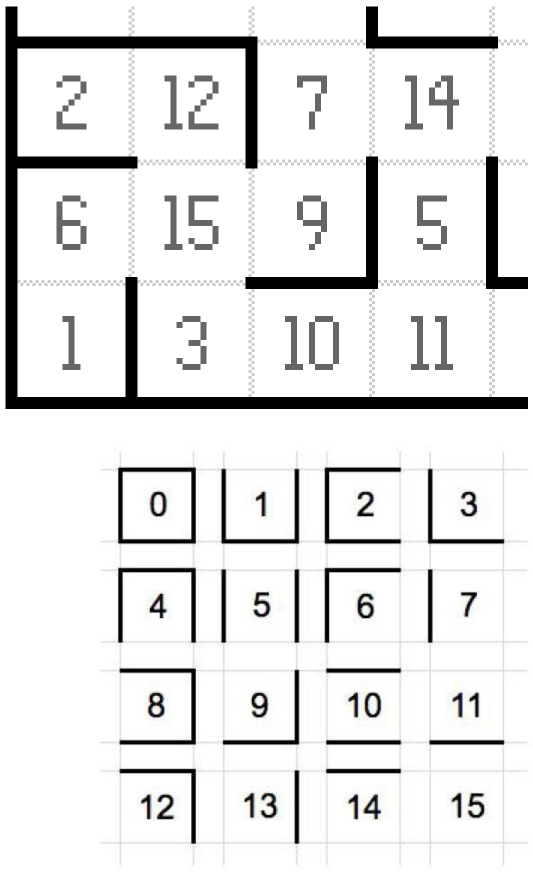


Figure 1: The position of walls based on the integer values

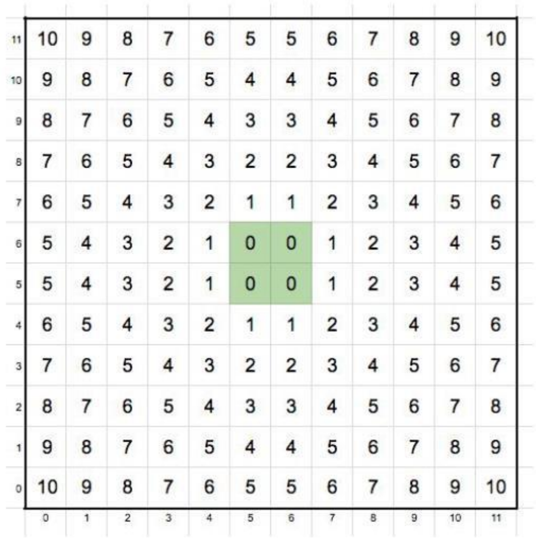
The figure above gives a clear picture how the values in the csv file are interpreted by the code and how the maze is created.

Robot Specification [Source: Udacity Project Files]
The robot can be considered to rest in the centre of the square it is currently located in, and points in one of the cardinal directions of the maze. The robot has three obstacle sensors, mounted on the front of the robot, its right side, and its left side. Obstacle sensors detect the number of open squares in the direction of the sensor; for example, in its starting position, the robot's left and right sensors will state that there are no open squares in those directions and at least one square towards its front. On each time step of the simulation, the robot may choose to rotate clockwise or counter clockwise ninety degrees, then move forwards or backwards up to three units. It is assumed that the robot's turning and movement is perfect. If the robot tries to move into a wall, the robot stays where it is. After movement, one time step has passed, and the sensors return readings for the open squares in the robot's new location and/or orientation to start the next time unit.

More technically, at the start of a time step the robot will receive sensor readings as a list of three numbers indicating the number of open squares in front of the left, centre, and right sensors (in that order) to its "next_move" function. The "next_move" function must then return two values indicating the robot's rotation and movement on that time step

Exploratory Visualisation
Reference:
1. http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

The heuristic function is used to help guide A* by providing it an estimate of the minimum cost from any cell to the goal. In our case, the Heuristic Grid represents the maze layout where every cell is labelled with the number of steps it is located from the goal. The four 0's in the middle represent the goal area while each step away from the goal results in an increase of 1. If the robot was positioned on a cell labelled '9', its next logical step would be to a bordering cell labelled '8'. It should repeat this process until it reaches '0'.



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

Figure 2: Heuristic Grid representation of Test Maze 01.

The value function directs the robot to determine an optimal path (to the best of its knowledge). The robot can move up to three consecutive spaces in any one direction per move. Consider an example for moves between steps 3 and 4 where the robot moves from (1, 0) to (4, 0), and again between steps 11 and 12, where the robot moves from (11, 0) to (3, 0) below

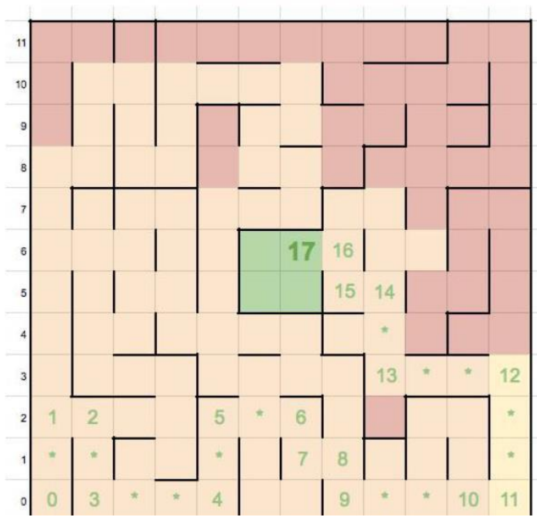


Figure 3: Optimal path for Maze 1

Similarly, the optimal path can be found for mazes 2 and 3 as 23 and 25 respectively

Algorithms & Techniques

In the first trial, we try to find the solution to the question - what is the layout of the maze? So, the first trial can be considered as a learning trial wherein the micromouse tries to understand or rather collect information about the layout of the maze. Also, it will find all the possible paths to the centre of the maze. In the second trial, we try to reach the centre of the maze using an optimal path, so that the micromouse reaches the centre in the least possible number of moves.

We constantly assess the number of total move taken, along with the total time the micromouse is navigating the maze. Thus, it can be conveyed that the environment is quantifiable. After the first trial, the micromouse must have recorded multiple possible routes to the centre of the maze, and must be able to choose the optimum route. The solution can be replicated by repeating the optimization trial for additional rounds. Assuming the micromouse initially identified and voyaged on an optimal route, its path to the centre of the maze should remain the same if it were subject to few more trials.

The following algorithms were considered to solve the maze problem:

1. Dijkstra's Shortest Path Algorithm
2. A* Best First Search Algorithm
3. Breadth First Search Algorithm
4. Flood Fill Algorithm
5. Dynamic Programming

1. Dijkstra's Shortest Path Algorithm: Dijkstra's algorithm is an algorithm that finds the shortest path from one node to all other nodes in its network. By finding the distance to all nodes in its path, it will inevitably create a connection with the nodes represented by the goal area.

2. A* Best First Search Algorithm: A* is a best-first search algorithm that searches for all possible solutions to the goal area and will choose the shortest path from the starting node to the ending node. Starting from a specific node, it creates a tree path of each possible direction until it reaches the goal area.

3. Breadth First Search Algorithm: This algorithm will start at a tree root and consider its closest neighbours before looking on to its next level neighbours.

4. Flood Fill Algorithm: This algorithm starts in the centre area and will fill each surrounding cell with a number of its relative distance from the goal area.

5. Dynamic Programming: Dynamic Programming performs a function like the operation of A*. It provides an optimal path from any origin to the goal.

However, I settled with A* and Dynamic Programming for this problem. I have used a modified version of A* for the first / exploration trial and Dynamic Programming for creating an optimal policy in the second trial.

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

Benchmark

The benchmark model is evaluated as:

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

Because we limit the number of steps to 1000, this will be the most basic form of a benchmark performance. The benchmark model will consist of a micromouse that makes a random movement decision at each step. If the mouse can navigate to the goal in fewer than 1000 steps during its exploration trial, a new benchmark should be set to the total number of steps in Trial 1. Because the micromouse has gained knowledge of the maze layout in its exploration trial, it should then optimise its route to the goal in Trial 2. Regardless of the number of steps taken during Trial 1, by nature of design, each maze has an optimal route - which is the minimum number of steps required to get from the origin to the centre area.

Maze 1, 2, and 3 have optimal routes of 17, 23, and 25 step counts, respectively. A benchmark consists of the sum of an average path to the goal and one thirtieth of an average number of steps needed to explore the maze. The challenge is defining what 'average' means. Given the size of each maze and the allotted maximum number of steps for the Exploration Trial, an average robot should absolutely be able to discover the goal area during exploration in far fewer than 1000 steps. Because the goal is to find an optimal path to the goal, the optimal routes of 17, 23, and 25 will be used for the Optimization Trial benchmark. For the Exploration Trial, a benchmark model must discover every cell, which is impossible to achieve by only visiting every cell once. To uncover every cell, a robot must visit several cells multiple times. To compensate for the use of the optimal route for the benchmark's performance in the Optimization Trial, the total number of cells for each maze (Squaring the dimensions) will be multiplied by 2.5 to generate the number of steps the benchmark model will take during the exploration phase.

Benchmark Scores

1. **Maze 1 Score:** 17 + (144 * 2.5) / 30 **= 29.00**
2. **Maze 2 Score:** 23 + (196 * 2.5) / 30 **= 39.33**
3. **Maze 3 Score:** 25 + (256 * 2.5) / 30 **= 46.33**

METHODOLOGY

Data Pre-Processing
Since the data regarding the location of walls was concisely provided in the input files and the robot can understand the readings from those input files, no pre-processing was needed for the datasets.

Implementation
The following procedure was followed to tackle the problem

1. Understanding sensor readings and recording gathered information
2. Robot movement during exploration trial
3. Locating the goal during exploration trial
4. Determining possible and optimal paths to goal
5. Finishing exploration trial and beginning optimization trial

Description

1. Understanding sensor readings and recording gathered information: Firstly, interpreting the sensor readings to determine whether there is a wall in the adjacent side or not. It is stored as a 2 dimensional numpy array called Maze Grid. The information about the walls is filled in this dictionary during the robot's exploration trial. The wall_locations method in the robot.py file helps the robot to find out the information about the walls.

   In every step taken, the robot receives information from the sensor readings (right / left / front) regarding the location of the walls. Clubbing together those readings, the robot constructs a 4-bit number describing the positions of the walls and the openings in the maze.
   Apart from that a 2-dimensional array is created called Path Grid, which represents the grid of the undiscovered maze with 0s in every cell. As the robot navigates throughout the maze during its *Exploration Trial*, the Maze Grid will be updated with the four-bit number describing the wall locations. Additionally, each cell in the Path Grid the robot visits will result in adding 1 to the corresponding cell in the Path Grid, essentially keeping count of the amount of times the robot has visited a space in the maze.

2. Robot movement during exploration trial: Once the robot is set in motion, it is left to explore the maze. Apart from passing through the openings in the maze, our concern is that the robot covers as much mazes as possible and visit undiscovered cells while moving towards its goal.

   The heuristic grid represents the layout of the maze, where every cell is labelled with the number of steps away from the goal. To determine the next move of the robot, the 'get_next_move' is defined. It helps the robot on what move to make next, what is its current position in the maze and orientation. The method works by first identifying if the robot has reached a dead end (by sensors identifying walls on all sides), and asking the robot go step back. If a single path is open in the maze, then we guide the robot to proceed in that direction. If there are multiple open paths, the heuristic grid is used to determine the closest path to the goal area.

3. Locating the goal during exploration trial: Each time the robot moves to a new cell, an if-statement will check to see if its current position is in the goal area. An instance variable called 'centre_found' is used, whose value is set to 'True' once the goal is found. After this, the robot is left to explore the maze until at least 75% of the maze grid is covered.

4. Determining possible and optimal paths to goal: Once the goal area is discovered, the main duty now is to determine all the possible paths to the goal area and then determine which is the optimal path to reach the goal area. A reference to Udacity's AI for Robotics Course is made to implement the dynamic programming method. The method 'compute_value' is the implementation of the dynamic programming model. For implementing dynamic programming, a grid labelled Path Value is created. The dimensions for this grid is same as that of the maze / heuristic grid. It is initialized to 1000 which is a very big number stating that the position in the grid is not covered yet. After sequential iterations, the path grid has a structure like the heuristic grid, but adjusted to compensate for the maze walls. Starting in the goal area and spanning out incrementally wherever there is an opening, each cell in the Path Value grid is updated with a number that corresponds to the number of steps away from the goal it is located.

5. Finishing exploration trial and beginning optimization trial: Once the robot has visited the goal at least once and approx. 75% of the maze grid is uncovered, the 'compute_value' method is called. Once the optimal path is found, the robot stores it with accurate movements and rotations to reach the goal. The rotation and the movement variables are reset to their default values in order that optimisation_trial method gets to know that it must be triggered.

When working with dead-ends in the exploration trial, the model fails when a random maze is constructed and then the model is tested against it. For the given input datasets, I managed to get the model to work with values -1 and then -2 to reverse itself when a dead end is encountered and rotate the robot so that the robot moves out of the dead end. However, I still found it little difficult to tweak to deal with any random maze.

Refinements

1. Creation of a separate script called 'config.py' that stored all the global dictionaries used throughout the project such as *dir_sensors*, *dir_move*, *dir_reverse*, etc.
2. The Exploration Trial was allowed to end once the goal area was found during exploration, but it was found that the robot had not seen half of the maze before stopping and was potentially missing superior paths to the goal.
3. The robot was then forced to uncover at least 80% of the maze before moving on to Optimization Trial, but then it was set to 75% as a threshold for optimal path discovery.
4. I discovered that whenever the robot hit a dead-end and retreated backwards, it would always believe that there was no wall behind it in its new position, which created some false directions to be stored on the Optimal Policy grid. This was alleviated by anticipating special scenarios where the robot would record a false reading and correcting them with a somewhat lengthy if-statement in the 'exploration_trial' method.

RESULTS

Model Evaluation and Validation

The following chart demonstrates the performance of the model for all the 3 mazes with the comparison of the benchmark score alongside.

Minimum Coverage	**Maze**	**Test Maze 1**	**Test Maze 2**	**Test Maze 3**
	Dimensions	**12 X 12**	**14 X 14**	**16 X 16**
	Benchmark Score	**29**	**39.33**	**46.33**
0%	Exploration Steps	141	163	197
	Optimisation Steps	17	31	29
	Score	**21.7**	**36.43**	**35.56**
25%	Exploration Steps	141	163	197
	Optimisation Steps	17	31	29
	Score	**21.7**	**36.43**	**35.56**
50%	Exploration Steps	141	170	197
	Optimisation Steps	17	31	29
	Score	**21.7**	**36.67**	**35.56**
75%	Exploration Steps	170	279	325
	Optimisation Steps	17	28	29
	Score	**22.67**	**37.3**	**39.83**
100%	Exploration Steps	315	659	867
	Optimisation Steps	17	26	27
	Score	**27.5**	**47.97**	**55.9**

If we look carefully at the chart, it is visible that the model outperforms the benchmark score for most of the cases. It however fails for the mazes 14 and 16 when we consider the coverage to be 100%. It is anyway not a mandatory criterion to cover the entire maze to find out the optimal path. Thus, there are scores which are beyond the benchmark score.

Talking about the robustness of the model, I think the model is not very robust. It definitely has scope for improvement. It was not able to handle a sample random maze. In simple words, the model cannot be repeated over other random mazes.

When I tested it with a randomly generated maze with lots of dead ends, the model did not perform well. It was not even able to complete the exploration trial within 1000 steps. Since I have considered the backtracking factor to 2, any maze with the dead end beyond 2 steps will not be handled properly by this model. This is the reason the current implementation will not scale to random mazes.

Justification

I would take up the minimum coverage criterion to be 75% to justify the model's performance and validation. If we look carefully at the row which has the coverage scores for 75%, we are well within the range of the benchmark score.

75%	Exploration Steps	170	279	325
	Optimisation Steps	17	28	29
	Score	**22.67**	**37.3**	**39.83**

It is a proper trade-off stating knowledge of the maze and the decision to adopt the optimal path for reaching the destination. If we consider the coverage to be 0%, we cannot assume that the optimal path is found for the problem since the maze is not seen thoroughly. Also, the same can be conveyed when the coverage is less than 50%. But beyond 50%, there can be a trade-off that the optimal path can be found even when 50% of the maze is seen. To have more chances of obtaining an optimal path, I have considered 75% coverage as the optimal coverage criterion. And I believe it is a valid stake.

CONCLUSION

Free Form Visualisation

The mazes which were provided in Udacity's project files achieved results which were better than the benchmark metric. But, I decided to test it with some different combination of mazes. I used the following new maze to test my code. The input file is provided in the project submission files.

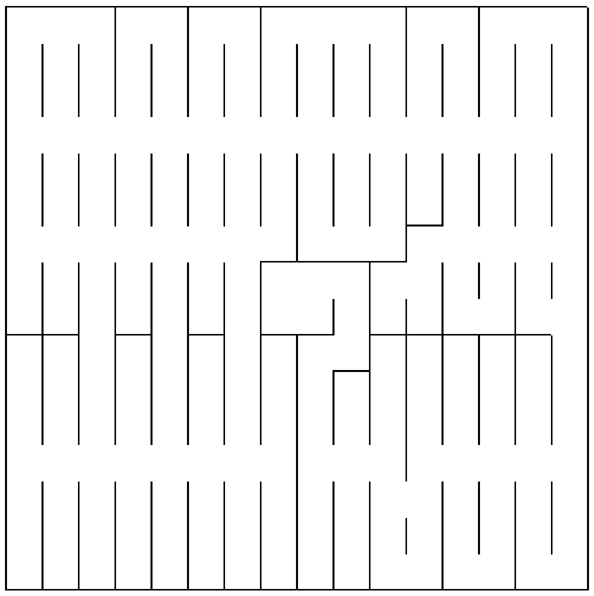


Figure 4: Test maze generated randomly to test the code

The maze above has many dead-ends and loops. The -2 and -1 logic of backtracking does not work here. The robot ultimately gets stuck in the dead end and is unable to retrace the route. This is where Dynamic Programming fails. The results were bad. The number of steps reached 1000 for the exploration trial itself, but not even 75% of the maze grid was covered. This demonstrates that there is scope for improvement.

Reflection

When I was a kid, I used to see the maze puzzles in the newspaper and in comic books. It was fun to solve it by finding the correct way to the centre or to get out of the maze! When I found the project in the suggestions for Capstone, I wondered how do computers solve the same kind of problems and then I decided to take up this project as my capstone project.

Finally, I followed the following steps to reach to one of the solutions to the maze problem.

1. Record and translate sensor readings into maze information and incorporate it with the robot's heading and location.
2. As the robot navigated the maze, update the map of the wall locations in the maze.
3. Implement a heuristic map to help lead the robot towards the goal area during exploration.
4. Update the robot's location after every move and output the rotation angle and movement directions to assist the robot to the next space.

5. Continue the previous steps until the robot has discovered the goal area at least once and has uncovered a minimum of 75% of the possible maze cells.
6. Calculate an optimal path to the goal area.
7. Initiate the second run and ensure the robot follows the optimal path to the goal area.

Initially I found it difficult to understand the mazes and the theory behind the integers that represent the wall in the mazes. But however, with little more understanding of the problem and going through some documents and papers on Maze Problem helped me understand it thoroughly.

Improvement

There are improvements possible to the current model. The A* and Dynamic Programming are used to get the optimal paths. As seen in Free Form Visualisation, the current implementation suffers badly when a very complex maze is chosen to test the code. Thus, this gives out more scope for improvement so that such mazes can also be handled appropriately by the code.

I would like to try out Q-Learning Algorithm for the same. The Q-Learning Algorithm is a model free reinforcement learning technique. Q learning learns transition probability by trail-error process. Since Q-Learning is model free, we get the following advantages:

- less computation, as low as O(1) suffices, which will eventually make the process faster.
- less space, O(States x Actions) versus Model-based O(States x Actions x States)

Apart from Q-Learning, I would also like to try out Breadth First Search Algorithm. I would like to compare the performances using the different programming techniques and compare with my current implementation and see that the code is able to handle complex and random mazes with ease.

I will consider my work as an incomplete work since there is always scope for improvement and that is the reason why we have advancements in technology every now and then.

REFERENCES

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

Maze Solving Algorithms on YouTube
https://www.youtube.com/watch?v=NA137qGmz4s

Micromouse USA
http://micromouseusa.com/

USC Micromouse Project
http://robotics.usc.edu/~harsh/docs/micromouse.pdf

Maze Generation Algorithm
https://en.wikipedia.org/wiki/Maze_generation_algorithm

Maze Solving Techniques using Dynamic Programming
http://www.geeksforgeeks.org/backttracking-set-2-rat-in-a-maze/

Code references for Dynamic Programming
https://algorithmsandme.in/2014/04/17/dynamic-programming-count-all-possible-paths-in-maze/

Exploring the maze problem
http://interactivepython.org/courselib/static/pythonds/Recursion/ExploringaMaze.html