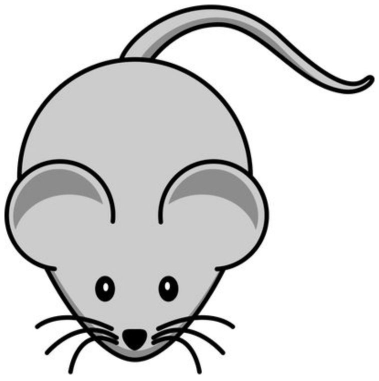
Machine Learning Engineer Nanodegree

Capstone Project for

Robot Motion Planning

Plot and Navigate a Virtual Maze



By

Halimat Mercy Oseni

March 27th, 2018

TABLE OF CONTENTS

DEFINITION

This section consists of Project Overview, Problem Statement and Metrics subsections.

1.1 PROJECT OVERVIEW

This project is inspired by the world wide Micromouse competition that holds almost every year and are most popular in the UK, US, Japan, Singapore, India and South Korea. Micromouse is an event where a small autonomous robot mouse solves a maze. It began in the late 1970s, although there is some indication of events in 1950.

The micromouse is expected to move around the unfamiliar maze and find its way to the designated goal which is commonly the middle of the maze starting from a designated corner of the maze.

The robot is allotted two(2) attempts with the maze; the initial attempt is for exploration, where the mouse keeps track of its position, discover walls, map out the maze and detect when it has reached the goal and the final attempt is for optimization, which is mainly for the mouse to find an optimal route and shortest possible time to get to the goal.

The dataset for the maze layout is provided in a text file. The first line of the text file is a number that describes the number of squares on each dimension of the maze. The following lines describe the actual structure of the maze with the edges that are open for movement and the numbers are separated by commas. The code that constructs the maze can be found in the *maze.py* file. The starter codes are provided by Udacity and are explained in more details in section 2.1 (Data Exploration and Visualization).

In this project, I programmed a micromouse (robot mouse) that finds an optimal way to reach a goal area in an unfamiliar virtual maze.

1.2 PROBLEM STATEMENT

The micromouse is subjected to two(2) different trials. The first trial is the exploratory trial where the robot mouse is expected to gain a detailed knowledge of the maze structural layout and also reach the goal which is the center of the maze. The mouse must also figure the best possible path to reach the goal. After the goal has been reached, the mouse can continue to explore the maze. The micromouse is then moved back to the starting position and set for the second trial. The second trial is the optimization trial. The mouse is expected to use the previously learned knowledge to navigate through the maze to the goal using an optimal route in the shortest possible time.

As part of the prerequisites for the project is a simplified model of the world along with the maze and the robot. My main objective is to implement an algorithm that takes the fastest/shortest possible paths to reach the center of the maze in a series of test mazes. Some of the important considerations to solve this problem are:

- Get the sensor readings and combine it with the robot's location and heading and update the robot's knowledge of the maze.
- With the knowledge at hand, determine where the robot should travel next and continue to learn about the maze.
- Update headings and location, continue exploring the maze until the robot discovers the goal area at least once and preferably the optimal path.
- Stop the exploratory trial and commence the optimal trial.

1.3 METRICS

The performance score is the sum of the one-thirtieth number of steps taken in the exploratory trial and the number of steps taken in the optimization trial. Each trial has a maximum of 1000 steps for a single maze.

Mathematically,

$$\textit{Performance score} = (1/30 \times A) + B$$

Where A is the total number of steps taken in the exploratory trial, $A <= 1000$ and

B is the total number of steps taken in the optimization trial, $B <= 1000$

ANALYSIS

This section consists of Data Exploration and Visualization, Algorithms and Techniques and Benchmark subsections.

2.1 DATA EXPLORATION AND VISUALIZATION

The starter code for this project was provided by Udacity and includes the following files:

- **robot.py:** This script establishes the robot class and it contains the implemented code for this project.
- **maze.py:** This script contains functions for constructing the maze and for checking for walls upon robot movement or sensing.
- **tester.py:** This script will be run to test the robot's ability to navigate mazes.
- **showmaze.py:** This script can be used to create a visual demonstration of what a maze looks like.
- **test_maze_##.txt:** This consists of three(3) sample mazes to test the robot's ability.

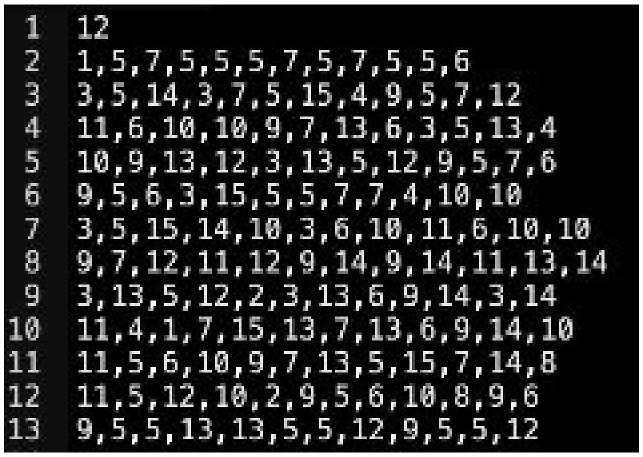
The maze is square in nature and a $n \times n$ grid, where n is either 12, 14 or 16. The start location is at the bottom-left corner of the maze (0,0) facing upwards. The goal area is situated at the center of the maze and it's a 2×2 grid.

The robot has three obstacle sensors, mounted on the front of the robot, its right side and left side. The obstacle sensors detect the number of open squares in the direction of the sensor; for example, at the starting position (bottom-left corner), the robot's left and right sensors will state that the walls are blocked and it's only expected to move forward. The sensors will return the distance between the micromouse and walls in a tuple as left, forward and right. After movement, the sensors return readings for the open squares in the robot's new location and heading to start the next time unit.

On each step of the trial, the micromouse can only rotate clockwise or counterclockwise ninety (90) degrees and can move forwards or backwards a distance of up to three units. The movement follows a rotation which can be -90, 90 or 0 and each movement is an integer which ranges from

-3 to 3. The micromouse must always have knowledge of its position in the maze, the number of walls surrounding it, its intended action and information from previous trials.

The test files (test_maze_##.txt) contains the details about the maze's structure and it looks like a mix of rows of numbers. Below is a screenshot of *test_maze_01.txt* file. The first line represents the size of the maze; 12 indicates that the maze is a 12 × 12 maze. The other lines consist of numbers ranging from 1 to 15 and are separated by a comma.



```
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
```

Image 1: contents of test_maze_01.txt

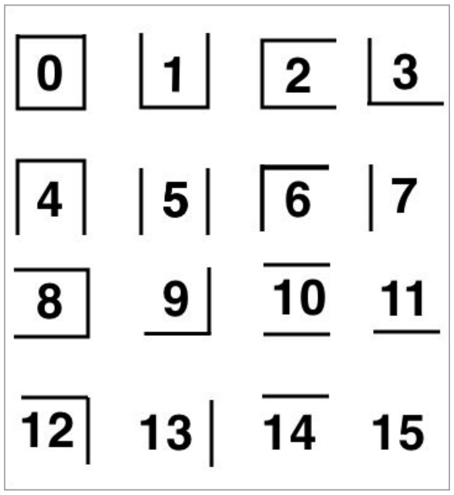


Image 2: Wall representations of numbers

The numbers represent the position of the walls and openings of each cell in the maze. Each number represents a 4-bit number that has a value of 0 if an edge is closed (i.e. a wall) and 1 if an edge is open. The 1s register corresponds to the upwards-facing side (top), the 2s register the right side, the 4s register the bottom side and the 8s register the left side.

For example: If all edges are open, we'll have

(1× 1 + 1 × 2 + 1 × 4 + 1 × 8) which is equal to 15

2.1.1 Maze 01 (12 × 12)

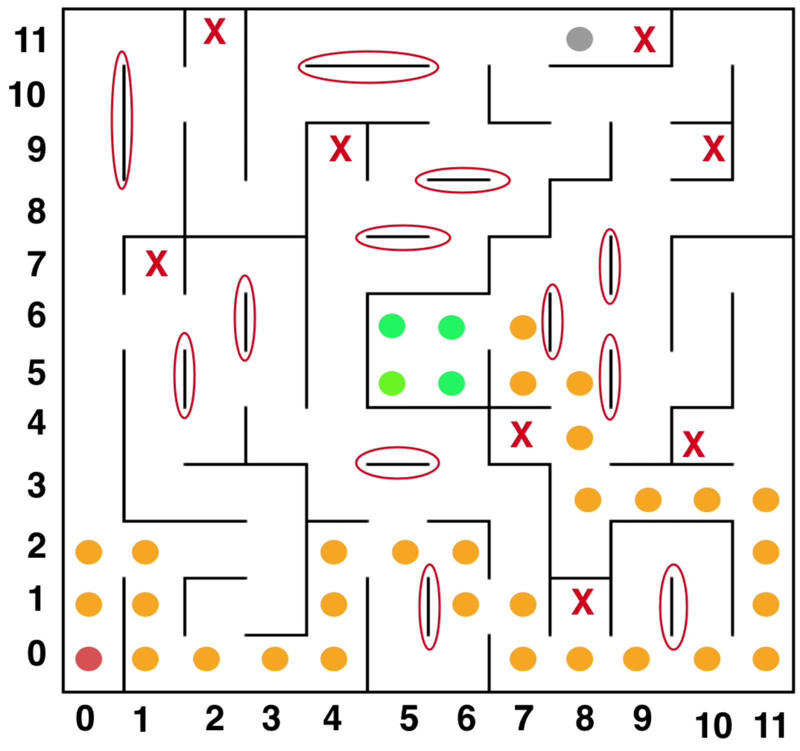


Image 3: maze_01(12 × 12)

The starting point is at (0,0) and it is shown with the red ball. The yellow balls represent the shortest path to the goal area which is at the center of the maze and displayed with green balls. The robot receives sensor readings as a list of three numbers indicating the number of open squares in front of the left, center, and right sensors. The robot is facing upwards (north) at the starting location and the sensors will return a list of (0, 11, 0). The shortest path to the goal area takes thirty(30) single steps from the starting point. However, the robot is allowed to take maximum of three(3) steps in one time step. Hence, the minimum time steps to reach the goal is

seventeen (17) steps. The robot should be able to identify and avoid dead ends and one-way paths to dead ends. I represented the dead ends using X marks and one-way dead ends with the grey ball. It should also be able to avoid going through a loop many times so as reduce the time steps to reach the goal area. I represented some of the loops using the red oval-like shapes.
The robot should be able to keep track of visited cells during the exploratory trial so as to explore other cells.

2.2 ALGORITHMS AND TECHNIQUES

The major challenge in this project is that the maze is entirely unknown to the robot during its first trial. The initial step to solving the problem is for the robot to have a prior knowledge of the structure of the maze by exploring, hence the algorithm to find the optimal path will be applied. The robot should also try to discover the goal on time during the exploratory trial as the performance score is dependent on the time step in both trials. While researching on the algorithms to use for this project, I was able to discover different search algorithms that could be implemented, some of which are best fit for the trials.

Random Turn: This algorithm makes a random decision at each step, it is inconsistent and slow but it's likely to discover the goal area.
Flood-Fill: This algorithm involves assigning values to each of the cells in the maze where these values represent the distance from any cell on the maze to the goal area. The destination cell, therefore, is assigned a value of 0. If the mouse is standing in a cell with a value of 2, it is 2 cells away from the goal.
Wall follower: This algorithm works on the rule of following either left wall or right wall continuously until it leads to the center (goal area). The micro mouse senses the wall on the left or right, and follows it to wherever it leads until the center is reached.

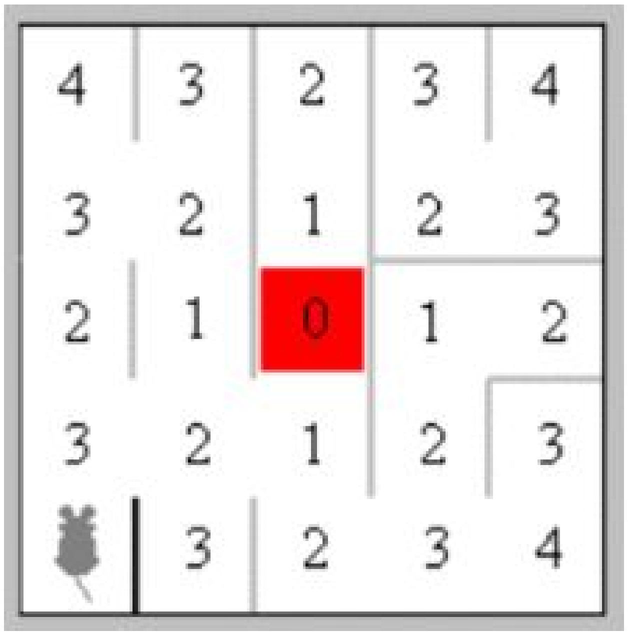


Image 4: Representation of flood-fill algorithm (http://automouse.sdsu.edu/design.html)

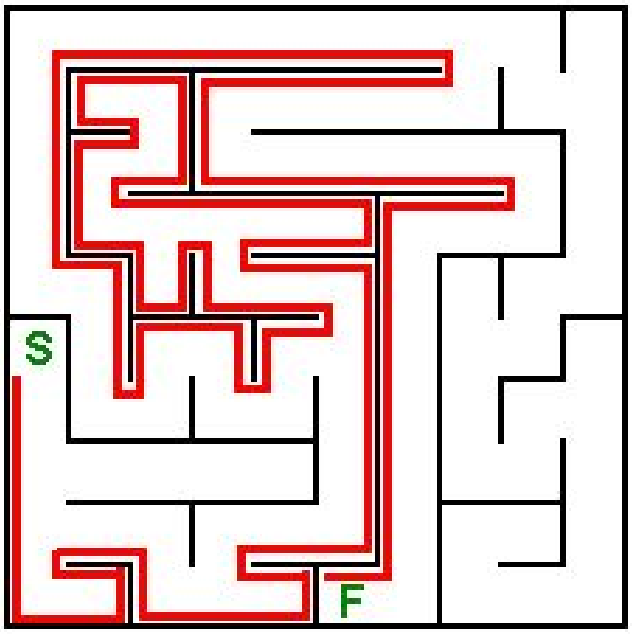


Image 5: Representation of wall follower

(http://www.mathgoespop.com/2008/09/math-on-tvmath-gets-around-brooke-knows-best.html)

Depth- First search: This is an intuitive algorithm for searching a maze in which the mouse first starts moving forward and randomly chooses a path when it comes to an intersection. If that path leads to a dead end, the mouse returns to his intersection and chooses another path. This algorithm forces the mouse to explore each possible path within the maze, and by exploring every cell, the mouse eventually finds the center. It is called "depth first" because if the maze is considered a spanning tree, the full depth of one branch is searched before moving onto the next branch. The relative advantage of this method is that the micro mouse always finds the route. Unfortunately, the major drawback is that the mouse does not necessarily find the shortest or quickest route and it wastes too much time exploring the entire maze.

Breadth-First search: The analogy of BFS with Micro mouse maze is drawn by considering vertices as maze cells. In Micro mouse, the BFS algorithm labels cells searching from the start cell to all the nearing neighbors. The program keeps records of which cell are immediate neighbors of start cell. It is capable of finding the shortest path, the search continues until it finds the goal.

A*: is a smart algorithm that searches for all the possible routes to the goal area and it chooses the shortest path from the start and end nodes based on the cost to take a particular path. At each step, it picks the node according to a value '**f**' which is a parameter equal to the sum of two other parameters – '**g**' and '**h**'. At each step it picks the node/cell having the lowest '**f**', and processes that node/cell.

We define '**g**' and '**h**' as simply as:

- **g** is the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
- **h** is the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is a smart guess.

For this project, I used A* algorithm for the exploratory trial and used dynamic programming to create an optimal policy for the optimization trial. I ensured that the robot expands the mapping area while finding the goal and counts the number of visits for each cell in order to visit the less visited area. During the exploratory trial, the robot is passed a list of possible movements to the goal with heuristic prediction of the distance to the goal area. The mapped area from the exploratory trial is used to find the optimal path to the goal during the optimization trial.

2.3 BENCHMARK

As discussed in section 1.3 (Metrics), the performance score is the sum of the one-thirtieth number of steps taken in the exploratory trial and the number of time steps taken in the optimization trial. The performance score is dependent on both trials and it is necessary for the robot not to take much time during the first trial. The trials have a maximum of 1000 steps each for each maze.

For Maze_01, the number of steps required to reach the goal area is 30 steps and since the robot can move up to three(3) space in one-time step (for the same direction), the optimal number of steps from the starting location to the goal area is 17. I would expect the benchmark score to be about 25 to 30 and shouldn't exceed 30.

For Maze_02, the number of steps required to reach the goal area is 43 steps and the optimal route to the goal area takes 23 steps. Hence, I'd set the benchmark to about 34 to 39 steps.

For Maze_03, the number of steps required to reach the goal area is 49 steps and the optimal route to the goal takes 25 steps. Hence, I'd set the benchmark to about 41 to 45 steps too.

Maze	**Path length**	**Optimal moves**	**Benchmark**
Maze_01	30	17	25 to 30
Maze_02	43	23	34 to 39
Maze_03	49	25	41 to 45

Table 1: Benchmark scores

METHODOLOGY

This section consists of Data Preprocessing, Implementation and Refinement subsections.

3.1 DATA PREPROCESSING

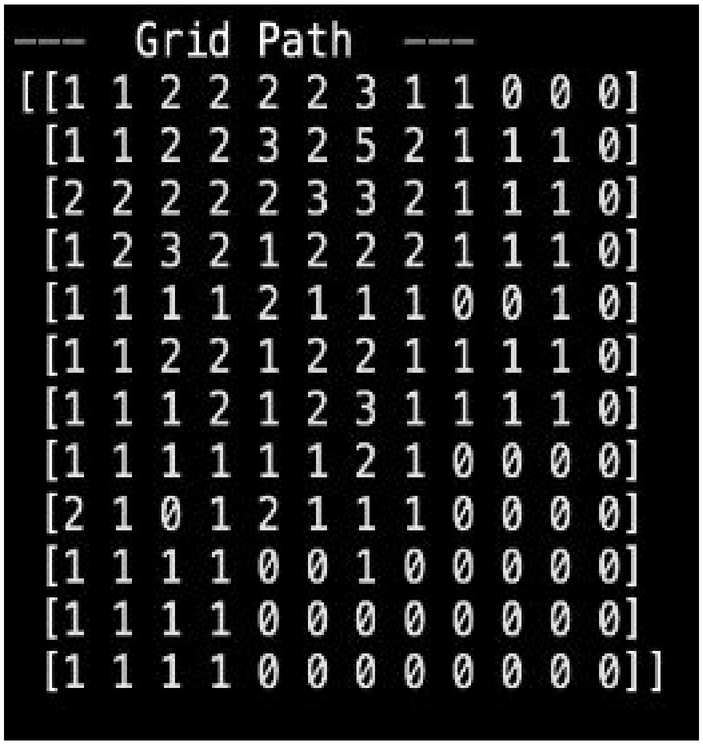
The sensor specification and environment designs were provided by Udacity for this project and it's accurate and stable. Hence, there isn't need for data preprocessing.

3.2 IMPLEMENTATION

Before I started implementing the algorithms, I ensured that I understood the architecture of the robot's environment. The *tester.py* tests the maze environment. It calls the Maze class *maze.py* to obtain the given maze. The tester initializes the robot *robot.py* and sends the dimension of the maze to the robot. The robot's location and heading is always updated and it has sensors that detect the distances from itself and the walls on its left side, front (forward) and right side.

The implementation of this project is further broken into the following steps as previously outlined in section 1.2 (Problem Statement subsection):

- Get the sensor readings and combine them with the robot's location and heading and update the robot's knowledge of the maze: The robot's knowledge of the maze is stored in a two-dimensional numpy array called maze grid. It stores the information gained on the structure of the maze walls which is identified by the sensors during the exploratory trial. The list is populated by the *wall_locations* method which returns a number that represents the description of walls surrounding the robot. On every time-step, the robot receives the sensor readings from its left, forward and right sensors. The robot is able to generate and store a 4-bit number that describes the surrounding maze walls and openings for its specific location by combining the sensor readings with the direction the robot is facing and its current location. I also created another two-dimensional numpy array called *grid path*. *Grid path* represents a grid of the undiscovered maze with 0's on every cell. During the exploratory trial, as the robot navigates through the maze, the maze grid will be updated with 4-bit numbers describing the wall locations. A value of 1 is added to each cell in the path grid that's visited by the robot. This keeps count of the number of times the robot has visited a particular cell in the maze.

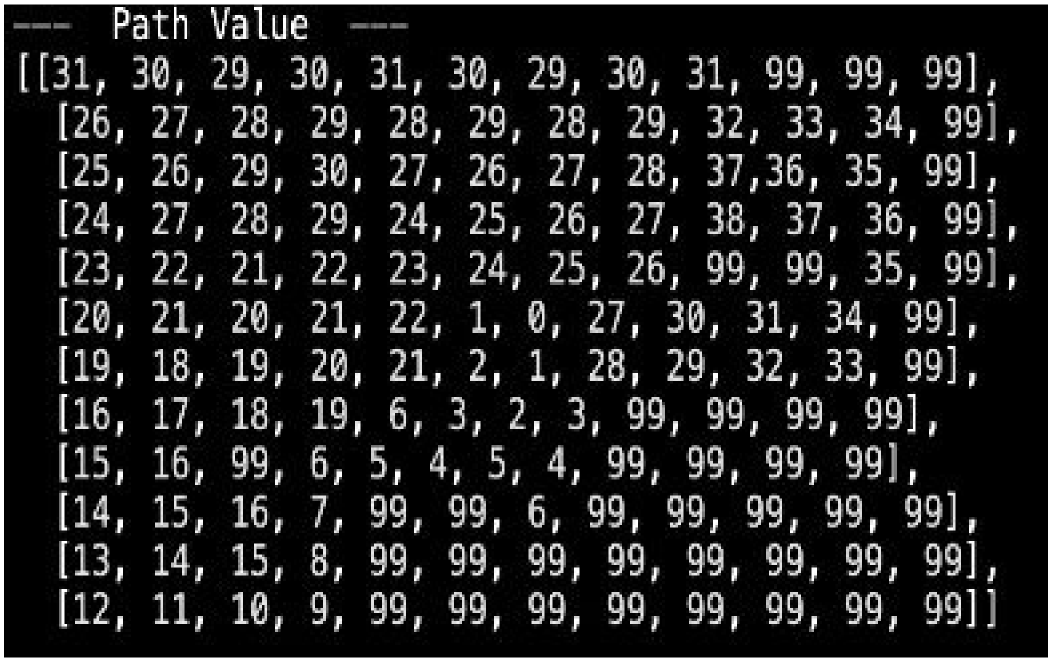


```
---  Grid Path  ---
[[1 1 2 2 2 2 3 1 1 0 0 0]
 [1 1 2 2 3 2 5 2 1 1 1 0]
 [2 2 2 2 2 3 3 2 1 1 1 0]
 [1 2 3 2 1 2 2 2 1 1 1 0]
 [1 1 1 1 2 1 1 1 0 0 1 0]
 [1 1 2 2 1 2 2 1 1 1 1 0]
 [1 1 1 2 1 2 3 1 1 1 1 0]
 [1 1 1 1 1 1 2 1 0 0 0 0]
 [2 1 0 1 2 1 1 1 0 0 0 0]
 [1 1 1 1 0 0 1 0 0 0 0 0]
 [1 1 1 1 0 0 0 0 0 0 0 0]
 [1 1 1 1 0 0 0 0 0 0 0 0]]
```

Image 6: Grid Path for Maze_01

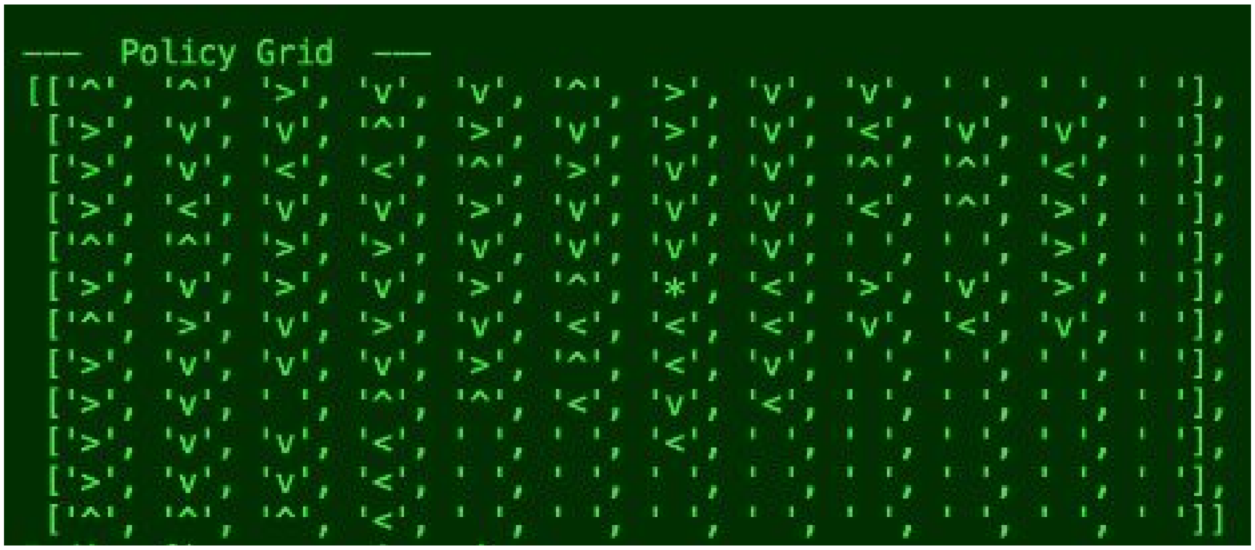
- With the knowledge at hand, determine where the robot should travel next and continue to learn about the maze: The only concern isn't about the robot moving through the openings but the robot should also be able to visit the cells that are yet to be discovered while maneuvering towards the goal area. I created a heuristic grid to guide the robot towards the goal area during the exploratory phase, the heuristic grid represents the maze layout where every cell is labelled with the number of steps away from the goal area. To determine the robot's next move during the exploratory trial, I created a method called *control_next_move* that guides the robot towards the goal based on the sensor readings, current position and heuristics. The method works by first checking if the robot has hit a dead end (i.e. if movement is stopped by a wall) and it redirects the robot to move backwards if it is in a dead end. The robot is directed to rotate towards an open path a move into the next cell if the sensors identify a single open path in the maze but it is directed to use the heuristic grid if there are multiple open paths and move towards the cell closest to the goal area.

- Update headings and location, continue exploring the maze until the robot discovers the goal area at least once and preferably the optimal path: I created an instance variable called *self.goal_area* that is checked whenever the robot moves into a new cell to see if the robot's current position is in the goal area. I also created a boolean instance variable called *self.discovered_goal* whose default value is *False* and it's set to *True* if the robot finds the goal during the exploratory trial. Aside the goal being discovered during the exploratory trial, I ensured that the robot continues to explore the maze until it has visited at least 70% of the maze so as to possibly discover additional paths to the goal area. After that, I deduce the optimal route for the robot to follow during the optimization trial. This was implemented using the dynamic programming approach (*compute_value)* from Udacity's AI for Robotics course. This approach outputs an optimal path to the goal area from any starting cell given the map of the maze and the location of the goal area. It works by considering the location of the maze walls and creates a policy grid that displays the optimal action ('<', '>', '^', 'V') at every single cell, leading to the goal area. I created a grid called *path value* that has the same dimensions as the maze and every cell is initially labelled '99'. After the initial path value grid was created, the cells in the goal area were assigned value '0'. Starting in the goal area, each cell in the path value grid is updated with a number that corresponds to the amount of steps away from the goal. In the policy grid, the cells in the goal area are assigned '*' .
- Stop the exploratory trial and commence the optimal trial: The exploratory trial is aborted when the robot has discovered the goal area at least once and explored at least 70% of the maze. The *compute_value* method is called and once an optimal path is identified, the robot stores the info. The *rotation* and *movement* are updated to 'Reset' and I called the *reset_values* method to reset the necessary values (*run_trial, location, step_count, heading and discovered_goal)* and the optimization trials commences.



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

Image 7: Path Value for Maze_01



```
---  Policy Grid  ---
[['^', '^', '>', 'v', 'v', '^', '>', 'v', 'v', ' ', ' ', ' '],
 ['>', 'v', 'v', '^', '>', 'v', '>', 'v', '<', 'v', 'v', ' '],
 ['>', 'v', '<', '<', '^', '>', 'v', 'v', '^', '^', '<', ' '],
 ['>', '<', 'v', 'v', '>', 'v', 'v', 'v', '<', '^', '>', ' '],
 ['^', '^', '>', '>', 'v', 'v', 'v', 'v', ' ', ' ', '>', ' '],
 ['>', 'v', '>', 'v', '>', '^', '*', '<', '>', 'v', '>', ' '],
 ['^', '>', 'v', '>', 'v', '<', '<', '<', 'v', '<', 'v', ' '],
 ['>', 'v', 'v', 'v', '>', '^', '<', 'v', ' ', ' ', ' ', ' '],
 ['>', 'v', ' ', '^', '^', '<', 'v', '<', ' ', ' ', ' ', ' '],
 ['>', 'v', 'v', '<', ' ', ' ', '<', ' ', ' ', ' ', ' ', ' '],
 ['>', 'v', 'v', '<', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 ['^', '^', '^', '<', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']]
```

Image 8: Policy Grid for Maze_01

3.3 REFINEMENT

I used A* (for the exploratory trial) and dynamic programming (for the optimization trial) techniques for this project. I created a method called *reset_values* to help reset all the necessary values after the exploratory trial was completed. During the exploratory trial, I set this trial to be aborted after the goal area has been discovered, though the total time-steps were reasonable but most times, I realised the robot was yet to explore the maze well enough and it missed some of the optimal paths to the goal. In order to ensure the robot explored the maze well enough and is aware of the optimal routes, I required the robot to have visited at least 70% of the maze. I also tried increasing the requirements from about 71% to 90% and the total time-steps increased as expected. I also created a new file called *global_variables.py*, where I stored the global dictionaries (*dir_sensors, dir_move, dir_reverse)* and other variables I used for the implementation. My implementation won't do well on a maze that has dead ends that span more than three cells because the robot's logic only requires it to move a max of two spaces backwards. The robot might be prone to an endless loop if moving two spaces backwards does not reveal other travel directions aside moving forward.

RESULTS

This section consists of Model Evaluation and Validation, and Justification subsections.

4.1 MODEL EVALUATION AND VALIDATION

The table below shows the optimal moves for the mazes and this is measured using the A* search algorithm.

Maze	**Path length**	**Optimal route**
Maze_01	30	17
Maze_02	43	23
Maze_03	49	25

Table 2: Optimal moves

The model is programmed to find the goal area in any type of maze and it also requires the robot to have visited at least 70% of the maze during the exploratory trial before commencing the optimization trial. This enables the robot to discover multiple paths to the goal area. The final performance scores are reasonable and are somewhat different when tested using different coverage. The performance scores for the maze when the coverage was set to at least 10% to 47% was the same as when coverage wasn't required. I noticed the change in performance score for some of the test mazes when the coverage was set to at least 48%. The test mazes used are the ones provided by Udacity for this project.

The table below (Table 3) shows the results from testing the mazes for different coverages:

Maze	Exploratory Trial			Optimization Trial		Score	
	Discovered Goal?	**No. of steps**	**Coverage**	**Found Goal?**	**No. of steps**	**Benchmark score**	**Performance score**
Test Maze_01	Yes	141	**None**	Yes	17	25 to 30	**21.700**
Test Maze_02	Yes	163		Yes	31	34 to 39	**36.433**
Test Maze_03	Yes	197		Yes	29	41 to 45	**35.567**
Test Maze_01	Yes	141	**48%**	Yes	17	25 to 30	**21.700**
Test Maze_02	Yes	167		Yes	31	34 to 39	**36.567**
Test Maze_03	Yes	197		Yes	29	41 to 45	**35.567**
Test Maze_01	Yes	141	**50%**	Yes	17	25 to 30	**21.700**
Test Maze_02	Yes	170		Yes	31	34 to 39	**36.667**
Test Maze_03	Yes	197		Yes	29	41 to 45	**35.567**
Test Maze_01	Yes	141	**55%**	Yes	17	25 to 30	**21.700**
Test Maze_02	Yes	180		Yes	31	34 to 39	**37.000**
Test Maze_03	Yes	213		Yes	29	41 to 45	**36.100**

Test Maze_01	Yes	141		Yes	17	25 to 30	**21.700**
Test Maze_02	Yes	229	**60%**	Yes	28	34 to 39	**35.633**
Test Maze_03	Yes	267		Yes	29	41 to 45	**37.900**
Test Maze_01	Yes	141		Yes	17	25 to 30	**21.700**
Test Maze_02	Yes	255	**65%**	Yes	28	34 to 39	**36.533**
Test Maze_03	Yes	291		Yes	29	41 to 45	**38.700**
Test Maze_01	Yes	146		Yes	17	25 to 30	**21.867**
Test Maze_02	Yes	266	**70%**	Yes	28	34 to 39	**36.867**
Test Maze_03	Yes	304		Yes	29	41 to 45	**39.133**
Test Maze_01	Yes	170		Yes	17	25 to 30	**22.667**
Test Maze_02	Yes	279	**75%**	Yes	28	34 to 39	**37.300**
Test Maze_03	Yes	325		Yes	29	41 to 45	**39.833**
Test Maze_01	Yes	184		Yes	17	25 to 30	**23.133**

Test Maze_02	Yes	291	**80%**	Yes	28	34 to 39	**37.700**
Test Maze_03	Yes	369		Yes	29	41 to 45	**41.300**
Test Maze_01	Yes	227		Yes	17	25 to 30	**24.567**
Test Maze_02	Yes	309	**85%**	Yes	26	34 to 39	**36.300**
Test Maze_03	Yes	503		Yes	29	41 to 45	**45.767**
Test Maze_01	Yes	276		Yes	17	25 to 30	**26.200**
Test Maze_02	Yes	450	**90%**	Yes	26	34 to 39	**41.000**
Test Maze_03	Yes	602		Yes	27	41 to 45	**47.067**
Test Maze_01	Yes	306		Yes	17	25 to 30	**27.200**
Test Maze_02	Yes	607	**95%**	Yes	26	34 to 39	**46.233**
Test Maze_03	Yes	674		Yes	27	41 to 45	**49.467**
Test Maze_01	Yes	315		Yes	17	25 to 30	**27.500**
Test Maze_02	Yes	659	**100%**	Yes	26	34 to 39	**47.967**
Test Maze_03	Yes	867		Yes	27	41 to 45	**55.900**

Generally, the model did a good job and the results are fair enough. The performance scores were consistent and the changes were expected as there was increase in coverage. It also outperformed the benchmark model in some cases (except coverages from 90% to 100% for Maze_02 and Maze_03). I also realised the decrease in performance score for Maze_02 when the coverage was set to 85%. The score decreased to 36.300 and number of optimization steps was 26. This was as a result of multiple paths that have the same steps and the robot was programmed to choose a path that aligned with the first sensor reading. Hence, it favours *left* before *up* and *up* before *right* directions.

4.2 JUSTIFICATION

As I mentioned earlier, the model did a good job and the final results are fair enough. Although, the model failed to find the optimal path for Maze_02 and Maze_03 with 23 and 25 steps respectively. The model can be fine-tuned and improved to perform better.

Below is the best performance for this model:

Maze	**Exploratory Trial**			**Optimization Trial**		**Score**	
	Discovered Goal?	**No. of steps**	**Coverage**	**Found Goal?**	**No. of steps**	**Benchmark score**	**Performance score**
Test Maze_01	Yes	146	**70%**	Yes	17	25 to 30	**21.867**
Test Maze_02	Yes	266		Yes	28	34 to 39	**36.867**
Test Maze_03	Yes	304		Yes	29	41 to 45	**39.133**

Table 4: Best performance results

CONCLUSION

This section consists of Free-Form Visualization, Reflection and Improvement subsections.

5.1 FREE-FORM VISUALIZATION

I'll test the model on an additional maze. The new maze is of 12 × *12* dimension. Below is a visualization of the maze:

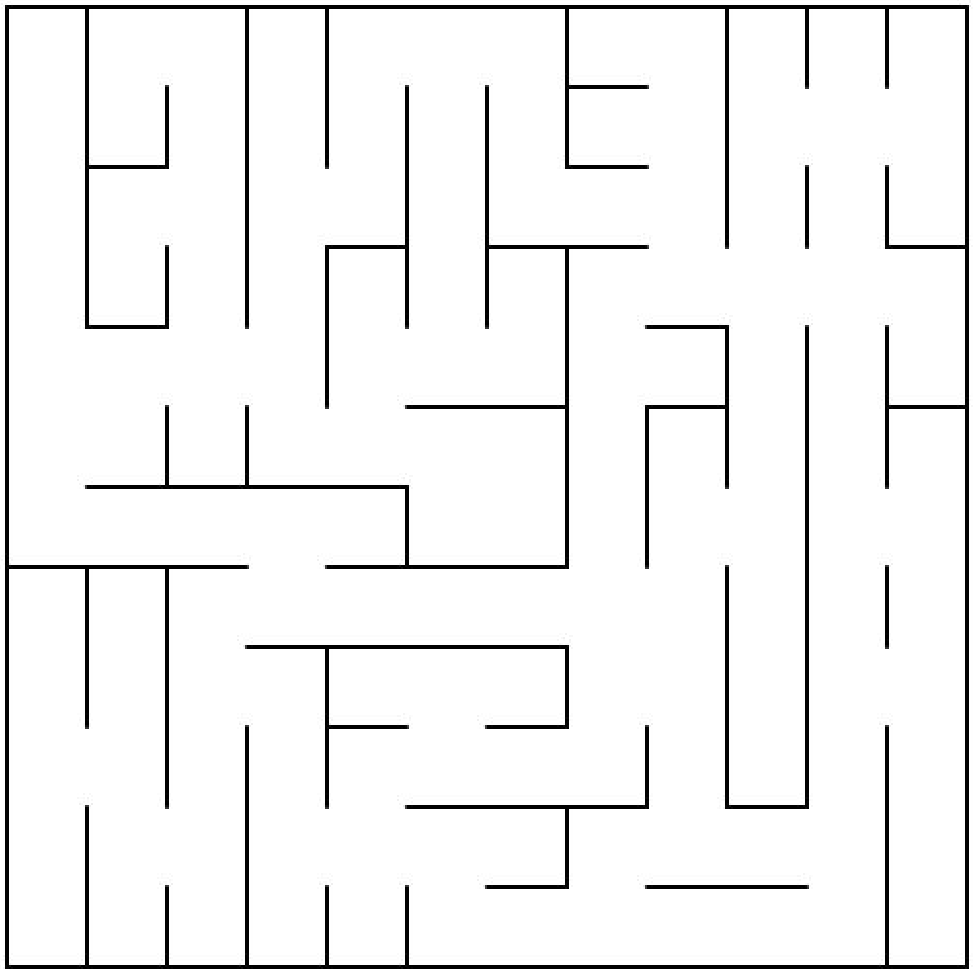


Image 9: test_maze_04

In this new maze, there are multiple routes to the goal. The maze has more dead ends with few loops. The image below shows the dead ends, possible loops and optimal path to the goal area in the maze.

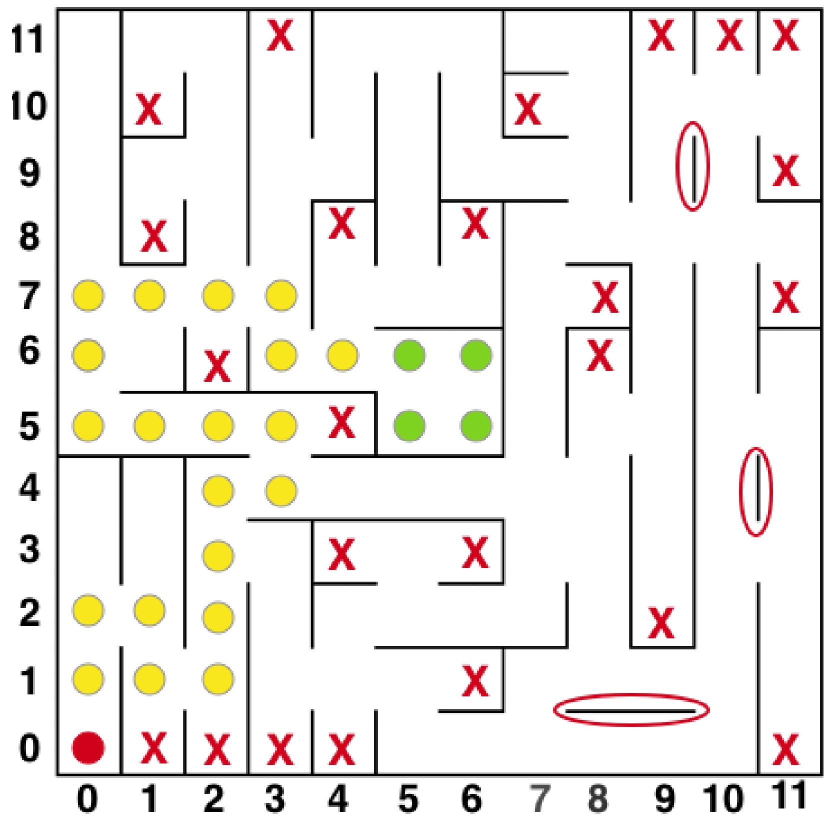


Image 10: Maze_04(12 × 12)

The starting point is at (0,0) and its shown with the red ball, the yellow balls represents the shortest path to the goal area which is at the center of the maze and displayed with green balls. I represented the dead ends using X marks and potential loops using the red oval-like shapes. The maze requires 21 steps to get to the goal area and 14 time-steps if the robot follow the optimal path.

The model should be able to perform well on this maze as the implementation covers for dead ends than spans not more than three cells.

Maze	**Exploratory Trial**			**Optimization Trial**		**Performance score**
	Discovered Goal?	**No. of steps**	**Coverage**	**Found Goal?**	**No. of steps**	
Test Maze_04	Yes	180	**70%**	Yes	14	**20.000**

Table 5: Test Maze_04 results

During the exploratory trial, the robot made 180 steps and covered exactly 70.14% of the maze, hence the performance score is 20.000.

5.2 REFLECTION

I admit that this project was draining and challenging because I haven't worked on similar projects before. The whole process was demanding, as I had to learn about different search algorithms and how to apply them in a similar domain. My main inspiration for taking this project is that I'm fascinated by how robots perform mind-blowing activities and I always look forward to building mine and also work with a team of brilliant minds whose goal is also to use technology to solve real-life problems.

The entire process used for this project can be summarised as follows (similar to that of section 1.2):

- Get the sensor readings and combine them with the robot's location.
- As the robot explores the maze, update the map of wall locations in the maze.
- Implement a heuristic method to lead the robot towards the goal area.
- The robot's location is updated after every move and the angle of rotation and direction of movement is outputted to help the robot navigate to the next space in the maze.
- The exploratory trial is aborted after the robot has discovered the goal area and has visited at least 70% of the cells in the maze.
- Calculate the optimal path to the goal area so it can be used in the optimization trial
- Start the optimization trial.

The steps outlined above may sound simple but the actual implementation of the model is difficult (at least for a learner like me). I remembered when I first took the Udacity's AI for Robotics course, I was enjoying every bit of it, then it suddenly went from easy to hard. I had to understand the Search lessons, how to expand a maze and also following the optimal path.

To complete this project, I had to make a lot of research and also look into past projects so as to get a clearer and better understanding of what's expected and what's to be done. Getting the robot to move round the maze, discovering the goal area and following the optimal path was a

relief. The most interesting part for me was learning and implementing the heuristics method. The model is consistent and can be used to solve this type of problem in a general setting.
Overall, this project is an eye opener and I look forward to exploring deeper in the AI/ML domain.

5.3 IMPROVEMENT

The scenarios in this project are in a discrete domain while the real micromouse competition is a continuous domain. The real micromouse competition is more complex because everything is physical - the robot, sensors, environment, size of the robot, location of sensors etc.
For the implemented model to be applied in reality with an actual robot, a lot of improvements have to be made. The robot is programmed to navigate in a discrete environment, hence, it has to be programmed to navigate in a continuous environment. Some advanced techniques like SLAM (Simultaneous Localization and Mapping) method will also be applied to explore the maze. It also needs to use PID (Proportional Integral Derivative) control to continuously adjust the directions and turns so it can travel through the maze without colliding with the walls. The speed also needs to be controlled instead of the number of steps, and rotation has to be continuous.
The current model is only workable in a discrete environment and would fail in a continuous environment.

REFERENCES

Udacity project files and example report

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

https://github.com/udacity/machine-learning/blob/master/projects/capstone/report-example-3.pdf

History of Micromouse

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

APEC Micromouse competition

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

Udacity's AI for Robotics course

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

http://swarm.cs.pub.ro/~anpetre/dynamic_prog.pdf (lecture notes)

Game programming

http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

DFS and BFS in Python

http://eddmann.com/posts/depth-first-search-and-breadth-first-search-in-python/

Plot and navigate a virtual maze

https://srikanthpagadala.github.io/notes/2016/11/06/plot-and-navigate-a-virtual-maze

https://github.com/rtlatimer/MLND-Capstone-Robot-Motion-Planning/blob/master/capstone_report.pdf

A Comprehensive and Comparative Study Of Maze-Solving Techniques by Implementing Graph Theory
https://pdfs.semanticscholar.org/e2dd/6802c910a7aaf7c9d77d2d1bf02b63541e2a.pdf

Search Algorithms in AI
https://hackernoon.com/search-algorithms-in-artificial-intelligence-8d32c12f6bea

A* Search Algorithm
https://www.geeksforgeeks.org/a-search-algorithm/

Chapter-3 and Chapter-4 from AIMA(Artificial Intelligence a Modern Approach)
http://aima.eecs.berkeley.edu/slides-pdf/

Labyrinth Algorithms: How to find a path between two points in the maze
http://bryukh.com/labyrinth-algorithms/#maze2graph

A Mobile Robot Solving a Virtual Maze Environment
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.885.9214&rep=rep1&type=pdf