Jeffrey Zhou
Udacity
Machine Learning Nanodegree
11/26/17
Capstone Project

# Robot Motion Planning

## Definition

***Project Overview***

Since I don't know yet what my passion is, I have picked one of the few suggested problem areas to explore, Robot Motion Planning. After researching and looking through all the suggested problems, I am most interested in this one because this project is actually inspired from the Micromouse competitions that took place in the late 1970s. I thought it would be fun challenge to analyzing how the Micromouse behaves and finding the solution to the Micromouse problem. The goal of this project is to have a Micromouse traverse an *n x n* grid maze to find the center. The Micromouse must not only find the center but also find the shortest path within the allowed 2 attempts. The Micromouse will start at a different corner or different starting point after every attempt. The first attempt will be for the Micromouse to explore the environment and the second attempt will be the actual test for the Micromouse to find the most optimal path to the center. This will be demonstrated using a virtual robot mouse and virtual test mazes.

***Problem Statement***

As said above, the goal is for the Micromouse to reach the center of the grid in the shortest amount of time. The maze is a square grid of *n* rows x *n* columns. Since this is a square grid, the number of rows must equal the number of rows. For example, 12x12, 14x14, 16x16, etc. Just like a maze, there will be walls to keep the Micromouse from going outside and collections of paths, as obstacles, for the Micromouse to overcome in order to find the center. The goal will be a sub-grid in the center of the main grid. With the 2 attempts that the Micromouse will have to reach its goal, the Micromouse will first use the first attempt to explore every path possible. The second attempt, the Micromouse will use the information that it gained from the first attempt to find the fastest path. So the first attempt can be seen as a testing trial and the second attempt can be seen as a training trial. Each trial, the mouse will have a maximum of 1000 steps. The Micromouse can turn left or right in a 90 degrees fashion and can move forward and backward. The performance of the Micromouse will be based on this metric: *1/30 * time_steps_of_first_run + time_steps_of_second_run.*

***Metrics***

Both the first and the second attempts will play a role in the evaluation metric. The final score will depend on the steps taken in both the attempts. As said before the performance metric will be based on this formula: *1/30 * time_steps_of_first_run + time_steps_of_second_run.* The score consist of 1/30 times the number of steps that the Micromouse took in the first attempt plus the number of steps that it took in the second attempt. For example, if the Micromouse took 500 steps in the first attempt and then 350 on the second attempt, the overall steps will be 850. Divide that by 30 and we have our final score: ~28.33. This would be most simple and easy way to evaluate the performance of the Micromouse. The goal is to get the lowest score possible. If we run more tests(2 attempts) and the score cannot get lower than 28.33, then 28.33would be the optimal score. Evaluation would be done on the final score and if the Micromouse has improved on the steps for reaching the goal on the second attempt.

## Analysis

***Data Exploration***

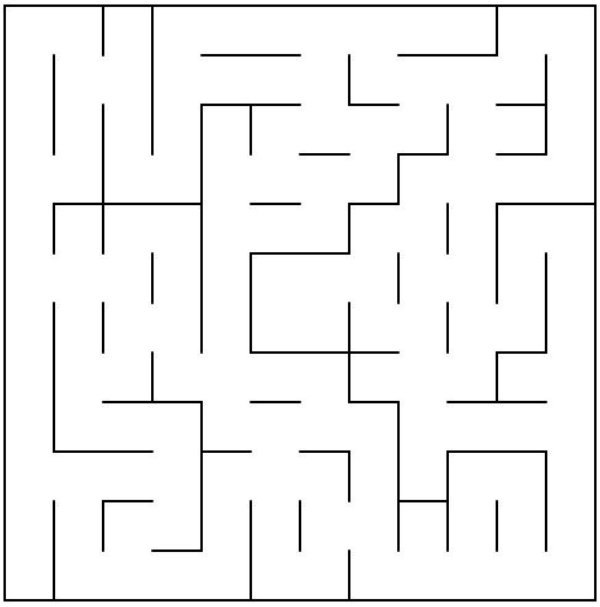
Given the text files, the architecture of the maze will be built based on them. Once a text file, for example *text_maze_01.txt*, is opened, you will see numerous lines of number values. The first line, a single number, will represent the number for *n.* So for example the first line is 12, then the grid will be 12x12. The following lines will represent the maze. The comma-separated number values represent the walls and openings of a square. We can decode the numbers to find number of walls/openings using this rule: “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); 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, if we see a square that indicates a 15. We will see that 1 + 2 + 4 + 8 = 15. In binary that would be 1111, which means that all the edges are open and there will be no walls. For the first attempt, the Micromouse will gain knowledge of the environment. This consists of the exact starting location, the number of squares in the grid, the number of walls, and all the possible routes that can lead to the center of the grid. The location for the Micromouse or any other squares can be defined as x,y coordinates, {2,10} for example. The movements will a number in the range from -3 to 3(negative for backwards and positive for forward) and the rotations will be -90, 0, and 90.

***Exploratory Visualization***

Here are the visualizations of the test mazes. On the left side is the encoding of the maze when I open up the text file. On the right side is the actual visualization created showmaze.py

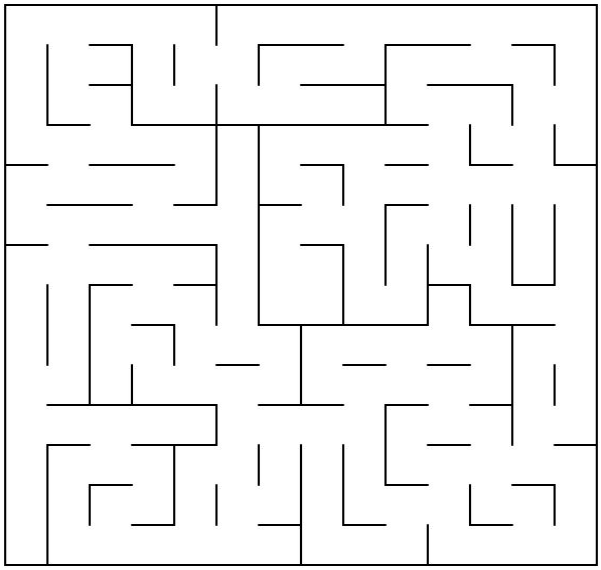
*test_maze_01.txt*

```
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
```



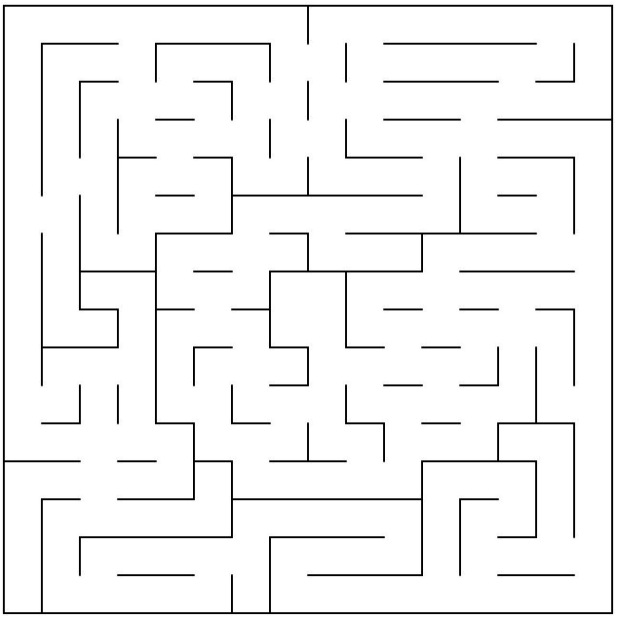
*test_maze_02.txt*

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



*test_maze_03.txt*

```
16
1,5,5,6,3,7,5,5,5,5,7,5,5,5,5,6
3,5,6,10,10,9,6,3,5,5,13,7,5,5,6,10
11,6,11,15,15,5,14,8,2,3,5,13,5,6,10,10
10,10,10,10,11,5,13,5,12,9,7,6,3,15,13,14
10,10,10,9,12,3,5,6,3,6,10,11,14,11,6,10
9,14,9,4,3,13,6,11,14,10,9,12,11,12,10,10
1,13,6,3,14,3,15,12,9,15,6,3,13,7,12,10
3,6,10,10,9,14,8,3,6,8,10,9,7,13,7,12
10,10,10,10,3,13,7,13,12,3,14,3,13,7,13,6
10,10,10,11,12,3,14,3,6,10,10,10,3,15,7,14
10,9,12,9,7,14,11,14,10,8,10,10,10,10,10,10
11,5,5,6,10,11,14,11,15,6,9,13,14,10,10,10
11,7,6,10,9,14,9,14,10,10,3,7,15,14,10,10
10,10,9,12,2,9,5,15,14,10,10,10,10,11,14,10
10,11,5,5,12,3,5,12,10,11,13,12,10,10,9,14
9,13,5,5,5,13,5,5,13,13,5,5,12,9,5,12
```



***Algorithms and Techniques***

The algorithms that I used for this project are Depth-First Search, Flood-fill, and A* Search.

Depth-First Search(DFS) - DFS is an algorithm that given a starting point, the algorithm will explore as far as possible along each branch. DFS is not an optimal algorithm and will not be guaranteed to find the best path, so I used this algorithm for the exploration phase. For this project, the algorithm will explore the left branch first, then the middle branch, and then the right branch. So for every step, if the Micromouse can go left, it will choose that direction first and then forward, and then right. This algorithm is used in the exploration trial since DFS is an optimal path finding algorithm and will not guaranteed to find the best path. During exploration phase, this algorithm will a build a map(initialized as a 2D array of zeros) out of the maze which will optimize pathfinding in the optimization phase. If the optimization phase is implementing Flood-fill, the DFS algorithm will set each neighbor next to the current location of the map to a certain value. If the starting location is [0, 0], then every neighbors, if traversable, will be given a value 1. The subsequent neighbors will be given a value 1 higher than the parent. So 1, 2, 3, etc. If the optimization phase is implementing A*, then DFS will set each location of the map to a value of 1

Flood-fill - Flood-fill is an algorithm that finds a path to the goal by selecting the location with the greater value(or lesser depending how you implement it). For example, if the starting location is [0, 0], then we check the values of the neighbors, which will be 1. The Micromouse will select and move to one of the locations that have the value of 1 and then choose a new node to move to with a value of 2 and so forth until it reaches the goal. The values will be preassigned in the map

from the exploration phase as described above. This algorithm will be used as the benchmark model.

A* Search - A* is an algorithm that is an extension of Edsger Dijkstra's 1959 algorithm. A* uses a greedy search and finds a least-cost path from the given starting point to the goal point. As the algorithm traverses the maze, it follows a path of the lowest expected total cost or distance. For this algorithm, we pick the first node that has the lowest. For this project, I have implemented this algorithm to pick the path with the lowest f cost. If there are nodes with the same f costs, the algorithm will then chose the lowest h cost of those nodes. If the h costs are tied, then the algorithm will chose one of the nodes randomly. The f cost is the sum of the g cost and the h cost. The g cost is 10 + parent node's g cost(initialized to 10). The h cost or heuristic cost is calculated using the Manhatten Distance formula:

```
h = abs (current_cell.x – goal.x) +  abs (current_cell.y – goal.y)
```

In addition to finding the goal with selecting the lowest f cost(or h cost if tied), this algorithm will traversing the maze using the map created in the exploration phase as described above. The Micromouse will only move in to a location where the value is a 1 in the map. This algorithm will be used in the optimization phase.

For any algorithm, Micromouse will explore each step and add the locations to a visited list and a stack list on the way. The visited list will keep track of the locations that the Micromouse have visited already so that the Micromouse won't go to the same location twice. The stack list will keep track of the order of all the states(heading, location) that the Micromouse have made. For example:

| ('left', [0, 3]) |
| --- |
| ('up', [1, 3]) |
| ('right', [1, 2]) |
| ('up', [0, 2]) |
| ('up', [0, 1]) |
| ('up', [0, 0]) |

This is needed for when the Micromouse hits a wall. If the Micromouse does hit a wall, it will be reverted to the previous state on top of the stack. On the example above, it would be reverted to ('left', [0, 3]).

***Benchmark***

The benchmark model will be the Flood-fill algorithm. This algorithm will visit most points, given the starting point and the destination, in a maze. Because this algorithm visits most points in the maze, this will not be the most optimal algorithm for this problem. However, this algorithm does eventually find the destination. So this will make a good benchmark for me to compare results against once I have a successful solution. Once I have a perfect solution, I will test my algorithm on a certain test data and compare the results with the results of the Flood-fill algorithm, executed with the same test data.

## Methodology

***Data Preprocessing***

I have already described the layout and the data encoding of the grid/text files in *Data Exploration* and *Exploratory Visualization*. To see a visualization of one of the mazes, one would need to run *python showmaze.py text_maze_0x.txt* in the command line.

***Implementation***

To actually test my algorithms in project, one would need to run *python tester.py test_maze_0x.txt.* In tester.py, it look loops 1000 times(steps) as a limit for 2 trials for the Micromouse to find a path to the goal. During each step, it calls next_move() in robot.py for a pair-value of (rotation, movement). In next, I have implemented that logic to check which algorithm to run for which trial:

```
if self.run == 0:
        return self.depth_first_search(sensors)
elif self.run == 1:
        return self.depth_first_search(sensors)#Flood fill using DFS
        #return self.a_star_search(sensors)
```

These lines of codes checks if the current run is 0(exploration) or 1(optimization). If it is 0, then run DFS. If it is 1 run either Flood-Fill or A*. One of the complications during the coding

process was testing these algorithms. To test just one algorithm, I would need to comment out the entire *elif self.run == 1* block. If ran into an error or a bug, I would need to walk through each step to see what went wrong. Unfortunately, there isn't a simulator for this project. So I would have to print out a lot of debug statements and walk through a picture of the maze manually in my head to see where the Micromouse have went. I would also need to lower the limit by a lot as 1000 is way too many steps to walk through in the command line.

***Refinement***

When starting this project and researching the algorithms. I found explanations on how algorithms should be done. For example, for A*, I need would need to create 8 nodes for each surrounding neighbors and give each of them f cost, g cost, and h cost. But during the coding process, I figured I wouldn't need to create all 8 nodes. Since tester.py calls next_move() in robot.py each step for the rotation and movement, there is no way that I will actually be able to reach the diagonal nodes(northwest, northeast, southwest, southeast) in one step and also I would not know if that diagonal node is traversable ahead of time because the Micromouse is given a set of sensors for the current state. To go diagonal, I would need the sensors for the next step too. So I decide to just make 4 nodes(north, east, south, west). This goes for Flood-fill as well. Further into the coding process, I decide to just create each of the 4 nodes only if they are traversable and have not been visited.

## Results

***Model Evaluation and Validation***

Out of all the algorithms used, A* is the best option. It is best because it involves calculations which provides a better result. A* selects lowest f costs(shortest to the goal) and h costs to reach it's destination. Whereas, DFS just traverse in a certain order(left, then middle, then right in this project) until it reaches the goal and Flood-fill picks subsequent neighbors with a greater value.

***Justification***

The scores for Flood-fill are:
*test_maze_01.txt* - 247.533
*test_maze_02.txt* - 257.733
*test_maze_03.txt* - 199.933

The scores for A* are :
*test_maze_01.txt* - 48.533

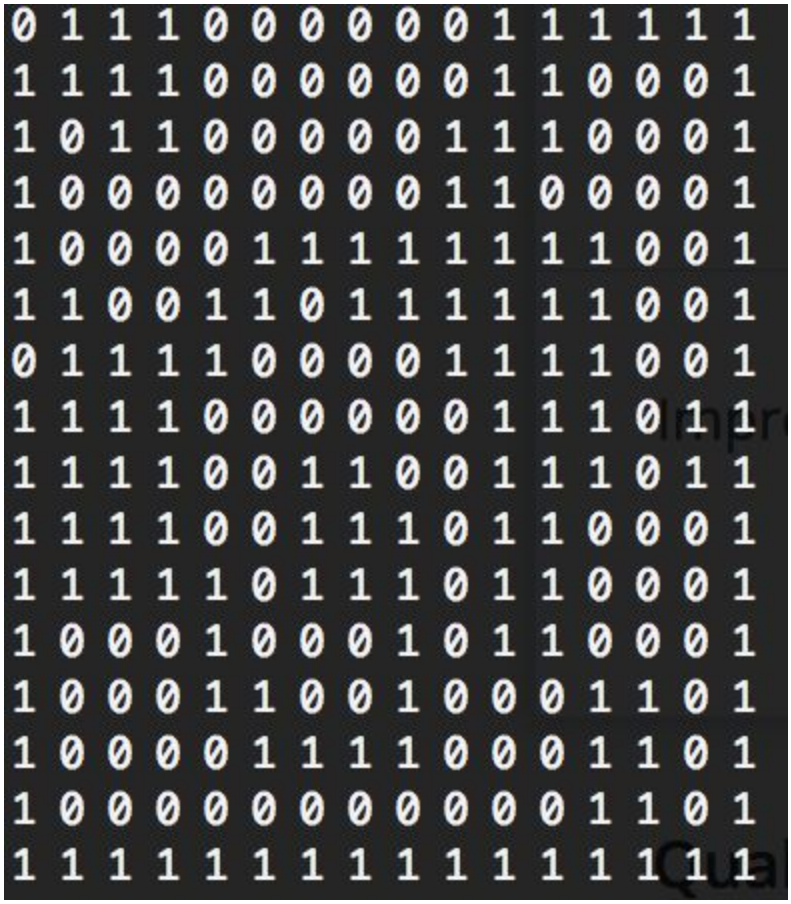
*test_maze_02.txt* - 71.733
*test_maze_03.txt* - 146.933

As you can see, the scores for each maze are a lot smaller for A* than Flood-fill so therefore a Micromouse using A* will reach the goal faster than a Micromouse using Flood-fill. This proves that A* is a better model than the benchmark model.

## Conclusion

***Free-Form Visualization***

Here is an example of the map in exploration phase for A* in the command line using *test_maze_03.txt*. The 1s are visited locations and 0s are unvisited.




Here is an example for the same test above for running A*

***Reflection***

Overall, this was a very challenging and interesting project. Prior to this project, I have little to no experience with Python, now I feel more confident of starting a Python project. I have learned 3 algorithms(DFS, Flood-fill, and A*) that can help me with programming interviews in the future. It feels amazing to actually implement the algorithms and see the final results. Another aspect of this project that I really liked is that I needed to make robot.py work with tester.py, so being able to look through tester.py, understand the logic, and make robot.py functional with it was a great feeling.

***Improvement***

For this project, I feel that I could have understood the existing codes(tester.py) and the algorithms better before actually coding. During the whole coding process, I was learning the algorithms and tester.py at the same time. So this made me change and revert a lot of code, which resulted is a lot of time spent. If I would have had a better understanding, I could have spent less time and maybe even write better codes.

## Sources


Robot Motion Planning Capstone Project
*https://docs.google.com/document/d/1ZFCH6jS3A5At7_v5IUM5OpAXJYiutFuSIjTzV_E-vdE/pub*

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

Labyrinth Algorithms
*http://bryukh.com/labyrinth-algorithms/*