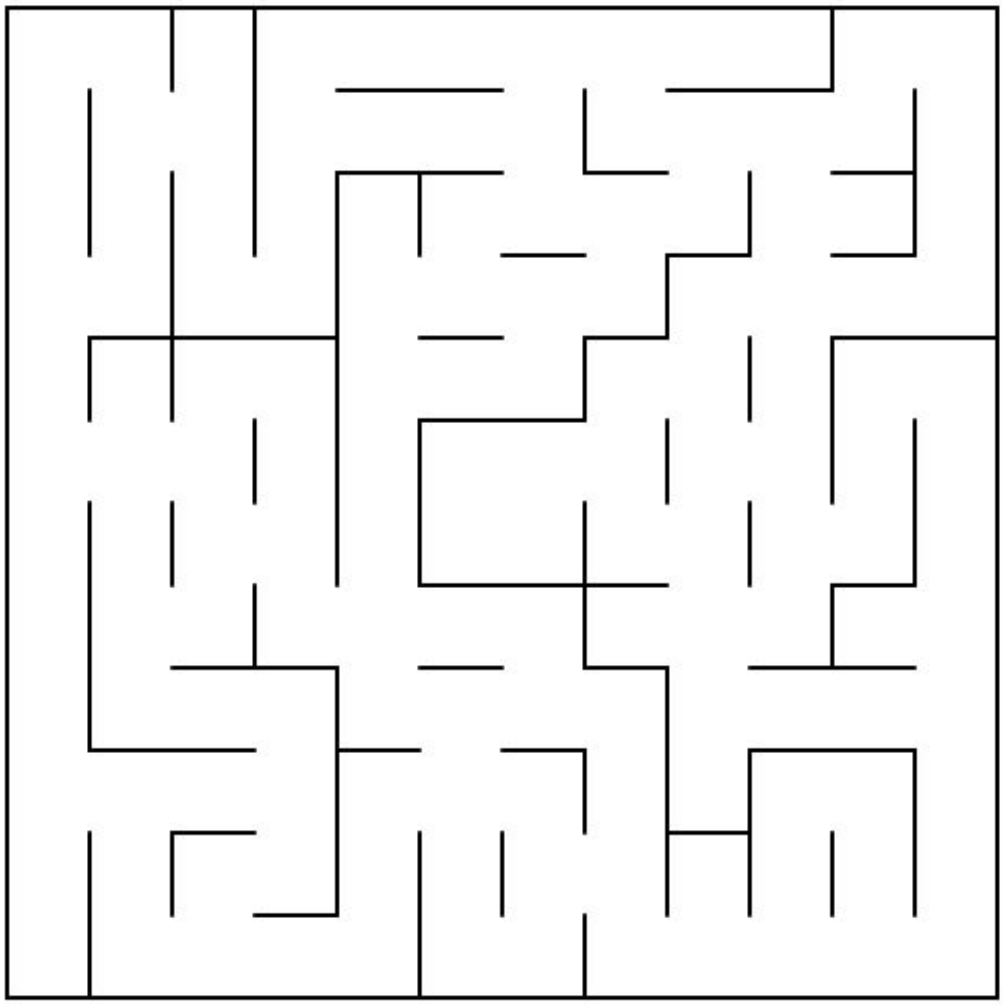
Machine Learning Nanodegree:

Justin Heaton

Capstone Project

Plot and Navigate a Virtual Maze

August, 30, 2016



Definition

Project Overview:

This project is based off of the micromouse competitions, https://en.wikipedia.org/wiki/Micromouse, in which a robotic mouse is placed in an unknown maze and must first plot out the entire maze and then use what it has learned to determine the optimal path from the starting position to the goal. In this version of the project, a virtual robot will be plotting and navigating a virtual maze environment, and attempting to determine the shortest path to the goal.

Problem Statement:

The maze in this problem is an n x n grid where n is either 12, 14 or 16. The mouse will begin its exploration from the bottom left corner of the maze and must navigate a path to the goal which is a 2x2 square in the center of the grid.

In each cell the mouse can can sense how far away the nearest wall is in three directions (left, forward, right), it can move as far as 3 units in in any direction (depending on where the walls are) and it has the option to rotate 90° to the left or right before moving. A rotation and a movement defines one time step.

Metrics:

The mouse has two runs through the maze, it is given a limit of 1,000 time steps combined, and it can reset to the starting position any time after it has reached the goal in the first run. In the first run, each time step costs 1/30 of a point, whereas in the second run each time step costs an entire point. Therefore, it is beneficial to take the extra steps to map out the entire grid in the first run in order to take the minimum number of steps in the second run.

A final score is the sum of the first round cost and the second round cost. A lower score is considered to be better because the score is measuring the total cost of all of the movements. As an example, if the robot uses 900 first round moves that costs 30 points, then it uses 30 second round moves costing and additional 30 points, the final score would then be 60.

Analysis

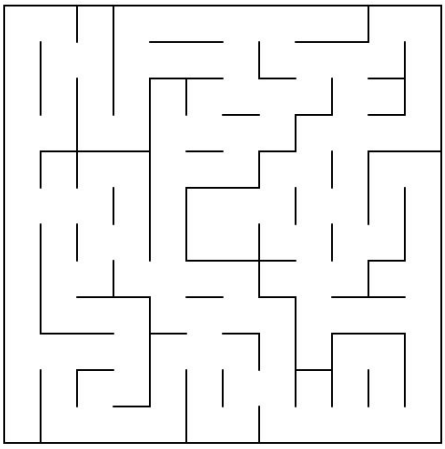
Data Exploration and visualization:

Each maze is constructed from a text file which is an n x n grid of numbers 1-15, where each number represents the presence or absence of walls in the cell. If the cell is open on the top, the cell value increases by 1, if it is open on the right it increases by 2, if it is open on the bottom it increases by 4 and if it is open on the left it increases by 8. So for example, if the cell is open on the top and bottom but there are walls on the left and right, the value will be 5 (1 for the top and 4 for the bottom), if the cell is open on all four sides the value will be 15 (1 + 2 + 4 + 8). These values are unknown to the mouse in the beginning, but as it moves through the maze it will attempt to build a grid with these same values based on the information passed by the sensors.

Here is the text file which defines the first maze (12x12):

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

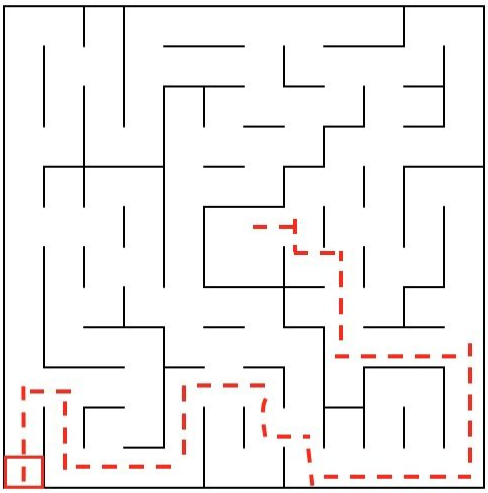
And here is a visualization of that maze:



(Notice that the rows of the text file correspond to the columns of the maze, so the 1 in the first row on the text file represents the starting position in the bottom left corner of the maze).

The mouse has sensors in the left, front and right, which pass information on how far away the nearest wall is in any direction. For example, in the starting position, the left and right sensors would pass the value 0 because there are walls directly on either side, and the forward sensor would pass the value 11 because it can move all the way from the bottom to the top of the maze unobstructed.

The shortest path to this particular maze is as follows:



The goal of this project will be to get the robot to determine the above path by its own exploration and mapping, and to do so in the fewest possible moves.

Algorithms and Techniques:

The real challenge in this problem is due to the fact that the maze is entirely unknown to the robot when it first starts out. There are several algorithms for finding the shortest path through a maze, but the majority of them require prior knowledge of the maze. Therefore, the first step is to have the mouse explore and map the maze, so that the unknown maze can become a known maze and a pathfinding algorithm can be applied.

For the exploration and mapping phase I have chosen to apply a modified random movement algorithm where priority is given to moves toward cells which have not been visited before. The robot will be passed a list of possible movements and it will make a random choice from that list.

After each movement, the wall positions of the current cell will be passed to a separate grid called self.val_grid which is essentially a map of the maze. Once the maze is entirely mapped out, and the mouse has visited the goal, it will be passed instructions to reset to the starting position and begin its second round.

At this point I will employ a heuristic technique which will allow me to make use of a shortest path algorithm. The heuristic used in this solution is another grid with the same parameters as the maze, which starts from the goal with the value 1, and counts up as it moves outward from the goal back to the starting position. The completed heuristic will have a value in each cell representing the number of moves that cell is away from the goal.

Here is a visualization of the heuristic for maze 1:

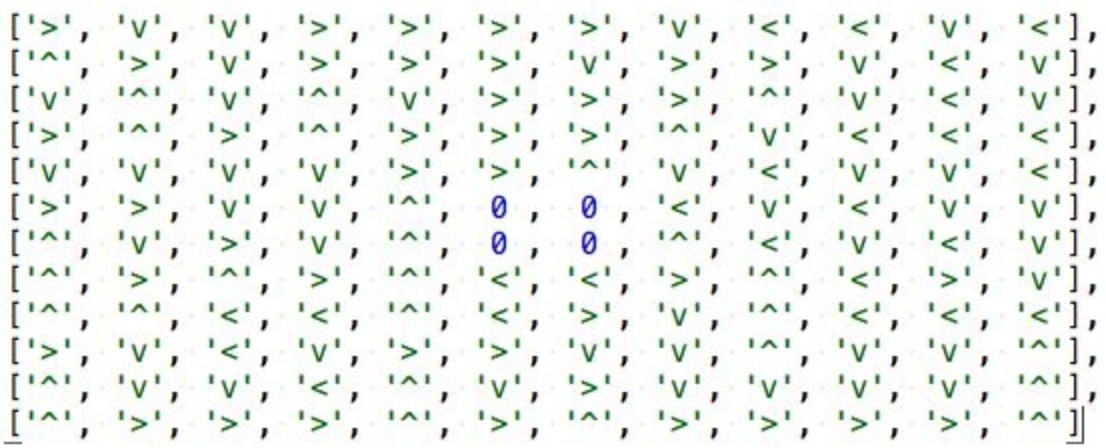
[24, 23, 22, 15, 14, 13, 12, 11, 12, 13, 10, 11],
[25, 22, 21, 16, 15, 14, 13, 10, 9 , 8 , 9 , 10],
[26, 23, 20, 17, 16, 13, 12, 11, 10, 7 , 8 , 9],
[25, 24, 19, 18, 15, 14, 13, 12, 5 , 6 , 7 , 8],
[26, 25, 24, 23, 16, 15, 14, 3, 4, 7 , 10, 11],
[25, 24, 23, 22, 17, 2 , 1 , 2 , 5, 6, 9, 12],
[26, 25, 22, 21, 18, 3, 2, 3, 4, 7, 8, 11],
[27, 24, 23, 20, 19, 20, 21, 6, 5, 6, 11, 10],
[28, 25, 26, 27, 20, 21, 20, 19, 6, 7, 8, 9],
[29, 28, 29, 28, 21, 20, 19, 18, 7, 16, 15, 10],
[30, 27, 26, 27, 22, 21, 18, 17, 16, 15, 14, 11],
[31, 26, 25, 24, 23, 20, 19, 16, 15, 14, 13, 12]

Once the heuristic is complete, I will then apply a dynamic programming algorithm which creates another grid called self.action_grid, with the optimal action for each cell as the value in that cell. The purpose of this algorithm is that, no matter where the robot is in the maze, it knows what is the best possible action to take from there to get to the goal as quickly as possible.

Here is a visualization of the action grid for the first maze:

[['right', 'down', 'down', 'right', 'right', 'right', 'right', 'down', 'left', 'left', 'down', 'left'],
['up', 'right', 'down', 'right', 'right', 'right', 'down', 'right', 'right', 'down', 'left', 'down'],
['down', 'up', 'down', 'up', 'down', 'right', 'right', 'right', 'up', 'down', 'left', 'down'],
['right', 'up', 'right', 'up', 'right', 'right', 'right', 'up', 'down', 'left', 'left', 'left'],
['down', 'down', 'down', 'down', 'right', 'right', 'up', 'down', 'left', 'down', 'down', 'left'],
['right', 'right', 'down', 'down', 'up', G, G, 'left', 'down', 'left', 'down', 'down'],
['up', 'down', 'right', 'down', 'up', G, G, 'up', 'left', 'down', 'left', 'down'],
['up', 'right', 'up', 'right', 'up', 'left', 'left', 'right', 'up', 'left', 'right', 'down'],
['up', 'up', 'left', 'left', 'up', 'left', 'right', 'down', 'up', 'left', 'left', 'left'],
['right', 'down', 'left', 'down', 'right', 'right', 'down', 'down', 'up', 'down', 'down', 'up'],
['up', 'down', 'down', 'left', 'up', 'down', 'right', 'down', 'down', 'down', 'down', 'up'],
['up', 'right', 'right', 'right', 'up', 'right', 'up', 'right', 'right', 'right', 'right', 'up']]

And here is another way of visualizing the action grid using arrows:



If the robot just follows the arrows from the starting position, it will find it's way to the goal, passing through the minimum number of squares. And it is also worth noting that by following the arrows from the starting position to the goal, the robot will follow the same path which I outlined earlier as the optimal solution to the maze.

But I also wanted to take into account the fact that the robot can take as many as three steps in one direction, so I passed the robot the additional instructions that if there are two or three consecutive arrows pointing in the same direction, it is to take that many steps. With these instructions, even though the mouse must pass through 30 cells in maze 1, it only requires 17 time steps to get to the goal.

Benchmarks:

Before I begin programming the solution to the problem, it will be good to set some preliminary benchmarks so that I can determine what I might expect out of a successful solution. Since a second round move costs 30 times more than a move in the first round, it will be necessary to get as much as possible out of the first round. Depending on the size and layout of the maze, I am assuming that the ideal path to the goal will require anywhere from 20 to 50 time steps. There are a maximum of 1000 time steps allowed and if the mouse uses 900 moves for the first round and then anywhere from 20 to 50 steps in the second round I should be expecting scores in the range of 50 - 80, and that is what I will be aiming at initially as a goal.

Methodology

Data Preprocessing:

Because the specifications of both the maze environment and the robot are provided for me in this project, no further preprocessing of data is necessary to complete the problem.

Implementation:

The first step was just get get the mouse to move through the maze without crashing into walls or getting stuck in a dead end, and to update the mouse's location and heading (which direction it is facing) after each movement.

I started off with the following piece of code which reads the sensors and builds a list of possible moves:

```
moves = ['left','forward','right']
self.possible_moves = []
self.sensors = sensors
for i in range(len(self.sensors)):
    if self.sensors[i] > 0: # A value of zero means there is a wall there.
        self.possible_moves.extend([moves[i]])
if self.possible_moves == []: #No possible moves means the robot is in a dead end
    rotation = +90
    movement = 0
```

Then a random choice is taken from the possible actions and the robot will choose a rotation and movement. If there are no possible moves then the mouse is in a dead end and it is then instructed to rotate 90° without movement.

The next step was to identify when the robot has reached the goal. This was done with the following code:

```
goal = [self.maze_dim/2 - 1, self.maze_dim/2]
if self.x in goal and self.y in goal:
    self.found_goal += 1 #Need to know that the goal has been reached in order to reset.
    if self.found_goal == 1:
        print '##### You have reached the goal!!! Goal Location: {},{}'.format(self.x,self.y)
```

I also created the following additional functions to help with the exploration and mapping of the maze:

```
# Return to start position at the end of round 1.
def reset(self):
    self.x = self.maze_dim - 1
    self.y = 0
    self.heading = 'up'
    print self.count_grid
    print self.val_grid
    print self.action_grid
    print "Reset!"
    return ('Reset','Reset')


# Marks whether each cell has been visited or not.
def count_cell(self):
    x,y = self.location
    if self.count_grid[x][y]  == 0:
        self.count_grid[x][y] = 1
        self.unique += 1 # Count how many unique cells visited
    print 'Unique Cells Visited: {}.'.format(self.unique)
    return self.unique


# Maps the maze with a value for each cell
def cellValue(self, sensors):
    x,y = self.location
    self.sensors = sensors
    headings = ['left','up','right','down']
    dir_vals = [8, 1, 2, 4]
    if self.val_grid[x][y] == 0:
        for i in range(len(headings)):
            if self.heading == headings[i]:
                self.val_grid[x][y] += dir_vals[(i + 2) % 4]
                if sensors[0] > 0:
                    self.val_grid[x][y] += dir_vals[i-1]
                if sensors[1] > 0:
                    self.val_grid[x][y] += dir_vals[i]
                if sensors[2] > 0:
                    self.val_grid[x][y] += dir_vals[(i+1)%4]
    self.val_grid[self.maze_dim-1][0]=1
```

```
# Identify possible actions in each cell.
def allowedActions(self, location):
    x,y = location
    allowed_actions = []
    vals = [[1,3,5,7,9,11,13,15],[2,3,6,7,10,11,14,15],
    [4,5,6,7,12,13,14,15],[8,9,10,11,12,13,14,15]]
    possible = ['up','right','down','left']
    for i in range(len(vals)):
        if self.val_grid[x][y] in vals[0]:
            allowed_actions.extend([possible[0]])
        if self.val_grid[x][y] in vals[1]:
            allowed_actions.extend([possible[1]])
        if self.val_grid[x][y] in vals[2]:
            allowed_actions.extend([possible[2]])
        if self.val_grid[x][y] in vals[3]:
            allowed_actions.extend([possible[3]])
        return allowed_actions
```

```
# Shows the optimal move from each position.
def actionGrid(self):
    vals = [[1,3,5,7,9,11,13,15],[2,3,6,7,10,11,14,15],
    [4,5,6,7,12,13,14,15],[8,9,10,11,12,13,14,15]]
    possible = ['up','right','down','left']
    delta = [[-1,0],[0,1],[1,0],[0,-1]]
    for i in range(len(self.heuristic)):
        for j in range(len(self.heuristic[0])):
            for k in range(len(vals)):
                if self.val_grid[i][j] in vals[k]:
                    if self.heuristic[i + delta[k][0]][j + delta[k][1]] == self.heuristic[i][j] - 1:
                        self.action_grid[i][j] = possible[k]
    self.arrowGrid()
```

```
# Replaces the strings in action_grid with arrows.
def arrowGrid(self):
    possible = ['up','right','down','left']
    arrows = ['^','>','v','<']
    for i in range(len(self.action_grid)):
        for j in range(len(self.action_grid[0])):
            for k in range(len(possible)):
                if self.action_grid[i][j] == possible[k]:
                    self.arrow_grid[i][j] = arrows[k]
    print self.arrow_grid
```

Once the heuristic and action grid have been established, and the robot has been reset. The following code provides the instructions for movement in round two:

```
if self.run == 1: #Instructions for movement in the second run
    directions = ['up','right', 'down', 'left']
    delta = [[-1,0],[0,1],[1,0],[0,-1]]
    action = self.action_grid[self.x][self.y]
    for i in range(len(directions)):
        if self.action_grid[self.x][self.y] == directions[i]:
            if self.action_grid[self.x + delta[i][0]][self.y + delta[i][1]] == directions[i]:
                if self.action_grid[self.x + (2 * delta[i][0])][self.y + (2 * delta[i][1])] == directions[i]:
                    movement = 3
                else:
                    movement = 2
            else:
                movement = 1
        if self.heading == directions[i]:
            if action == directions[i]:
                rotation = 0
            elif action == directions[i-1]:
                rotation = -90
            elif action == directions[(i+1)%4]:
                rotation = +90
```

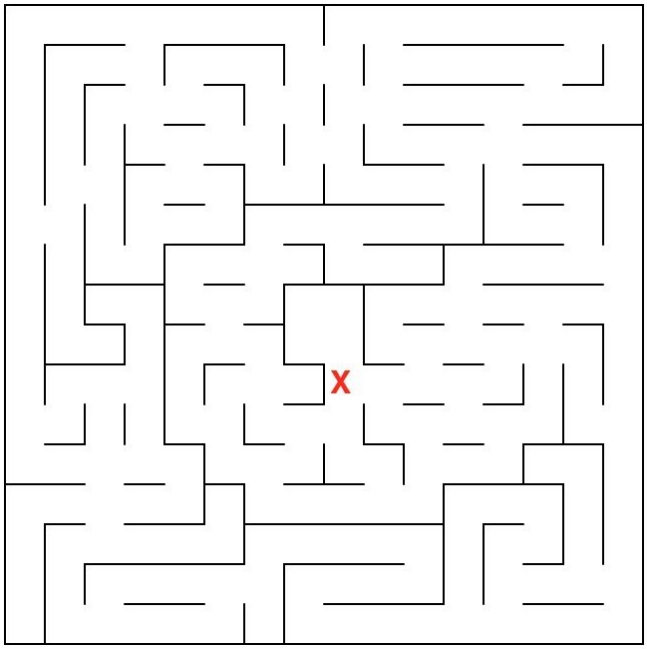
The code for building the heuristic is very long and will not be shared in this report but is viewable on the accompanying robot.py file which contains the entire script written for this project.

Refinement:

The code that I have written functions to find the optimal path, given the condition that every cell in the maze has been visited and the entire grid has been mapped out. If the robot does not visit every cell, it is still able to find a path, either optimal or suboptimal, the majority of the time.

The one challenge that my model faces is when the maze is larger, for example in the 16x16 maze, the robot occasionally struggles to explore and map out the entire maze. Usually the robot will still be able to apply the algorithms to find a path, but if there are multiple cells which have never been visited the model can fail.

Here is a visualization of a condition under which the model will fail. If the robot never occupies the cell marked with the red “X”, it will be unable to initiate the heuristic function and will therefore fail.



This particular obstacle has been addressed by adding the instruction that while the robot is inside the goal, every movement should be just one unit in length, so that the cell marked with the X can never be skipped.

Additional refinements could be made to improve the model including avoiding multiple visits to dead end cells and not getting caught travelling in loops in the maze. Furthermore, no model is perfect, and the results of this model in its current state, which will be discussed in the next section, match up favorably against the benchmarks that were set earlier, which suggests that my model is performing adequately for the scope of this project.

Results

Model Evaluation and Validation:

I decided to compile the results as an average score over ten trials for each maze. I think it is useful to report an average rather than a single result because there is some element of randomness to the robot's initial exploration and there will therefore be some fluctuation in the final score for each trial. It is also important to show success over a series of trials in order to validate the robustness of the model. Standard deviations are included in parentheses to show the variations between trials for each maze.

	First Round Moves	Second Round Moves	Score
Maze 1 (12x12)	704 (158.5)	17.4 (1.265)	40.89 (6.07)
Maze 2 (14x14)	904.3 (106)	26.2 (0.632)	56.3766 (3.68)
Maze 3 (16x16)	949.2 (2.53)	27.9 (1.20)	59.5733 (1.22)

It is to be expected that there would be a positive correlation between the size of the maze and the number of moves that would be required in each round, and this phenomenon can be observed clearly in the results table above.

Additionally, it is worth noting that the standard deviations are negatively correlated with an increase in maze size, which means that there is less variance between trials for the larger mazes. This can be explained by the fact that the maze is being forced to reset after 950 moves if the entire maze has not been mapped out, and for the larger mazes, this is happening more often, making the scores more consistent across trials.

Justification:

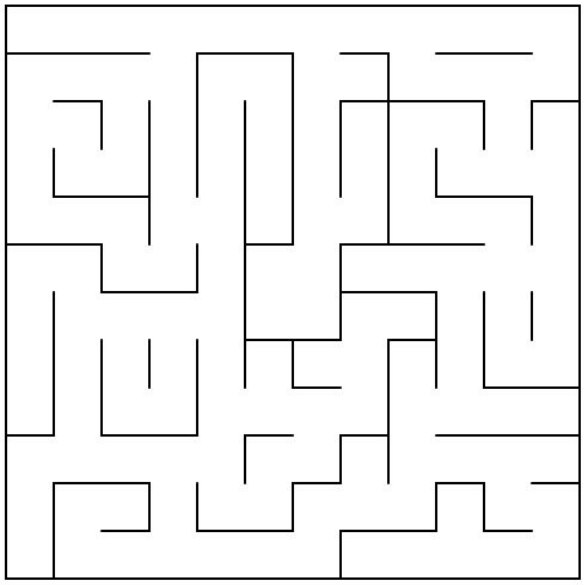
As a reminder, the benchmark scores that I estimated earlier in this report were a range from 50-80 depending on the maze. These final scores are very close to what I was expecting, and even slightly better, which shows that the algorithms and techniques that were implemented were successful in achieving the desired results.

Conclusion

Free-Form Visualization:

Finally I will test my model on an additional maze which I have constructed following the specifications of the provided mazes. My maze is a 12x12 matrix, and in constructing the maze I made attempted to add a few more dead ends and potential loops that the robot could get stuck in. If my model truly is robust it should not have any problem navigating this new maze environment which it has not been tested in in the programming phase.

Here is a visualization of my additional maze:



In ten trials through the new maze environment, the robot performed just as it had on the previous three mazes, reaching the target in ten attempts out of ten. The average number of first round moves was 783.9 with a standard deviation of 183.2, average number of second round moves of 11 with a standard deviation of 0 and an average score of 36.2, standard deviation 6.1. In this maze, the variation of the final score was due entirely to the variation in first round moves because the number of second round moves was consistent throughout each trial.

Reflection:

In this project I was given specifications for several virtual maze environments and asked to program a robotic mouse to both navigate and plot the maze in an attempt to identify the shortest path to the goal. Since the maze is unknown to the robot, it is impossible to just start it off with perfect instructions and watch it travel directly to the goal, and this problem made for an extremely interesting and challenging project.

Every step along the way was a difficult challenge, starting from “Okay how do I get this robot to move?” and then, “How do I update the location and heading after each move?” To “How do I know when each cell has been visited?” and “How can I make the heuristic to identify the optimal path to the goal?”

There were so many obstacles to overcome along the way and although it seemed almost overwhelming at the beginning, I knew that what I had to do was break it up into manageable pieces which could be tackled one at a time. I started out by writing an outline of what the final model would look like and what functions I would require to program along the way.

I took it one step at a time, facing and overcoming each obstacle when I came to it, and before I knew it, I found myself at the end of the road with a finished model and a working implementation of the algorithms and techniques which I had planned to implement in my solution.

Improvement:

In thinking about how my model would apply in a real life situation, with an actual robot in a physical maze, there are several improvements which would have to be made in order to for the model to be applied.

First of all, the virtual maze in this project is a discrete environment, meaning that the robot is considered to exist in the center of one cell, then move directly to the center of the next cell, but there is no state in between cells where the robot would ever exist. In reality, this kind of environment is impossible, so the robot would have to be programmed to navigate a continuous environment.

In a real world, continuous environment, it would be necessary to take into account the size of the robot, the width of the passages between walls, and the thickness of the walls. Additionally, the robot's motion would not be perfect, as it was in the virtual world, so it would be necessary to account for some noise in the movement and steering of the robot. It might also be necessary to employ some more advanced techniques for solving the maze, for example, the SLAM (simultaneous localization and mapping) method, which constantly updates the robot's location in relation to visible landmarks, and it allows the robot to map the maze as it moves.

As it stands, my model is a workable solution to the discrete maze environment. There are however, conditions in the continuous environment which my model would fail to navigate including curved passageways and walls and openings at inconsistent intervals. To navigate these types of environments, I would have to apply some of the changes mentioned in the previous paragraph.