Robot Motion Planning Capstone Project

Plot and Navigate a Virtual Maze

Robert Latimer

Udacity Machine Learning Nanodegree

June 15th, 2017

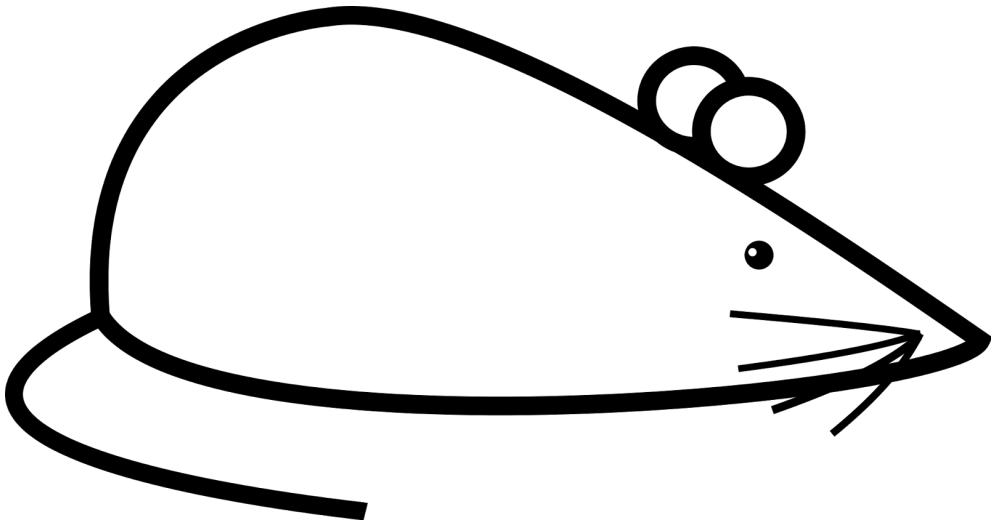

TABLE OF CONTENTS

I. DEFINITION

Project Overview

This project is inspired by Micromouse competitions that originated in the 1970s. A Micromouse competition is an event where a small robot attempts to navigate an unfamiliar maze. The robot starts in a corner and must find its way to a designated goal area commonly located near the center. The robot is allotted two attempts with the maze - the first attempt is an exploratory attempt where the robot will survey and map out its environment. The second attempt is where the robot will attempt to traverse an optimal route to the goal area as fast as possible. Using Python 2.7, our robot will attempt to a navigate a series of virtual mazes and determine the shortest paths to the goal area located in the center.

Problem Statement

Simply stated, the goal is to get the robot to the center of the maze as fast as possible. It will start in the bottom-left corner of the maze and must navigate from the origin to the goal in the shortest possible time, and in as few moves as possible. Sensors on the front and sides of the robot deliver positioning data at the start of each new time step. The maze is a square with equal rows and columns where the number of squares along each side must be 12, 14, or 16. The entire perimeter of the maze is enclosed by walls preventing the robot from escaping. The starting position will have walls on the left, right, and back sides, so the first move will always be forward. The goal area will be made up of a 2-by-2 square in the center of the maze.

The robot will run a first trial that is exploratory in nature where it will aim to build knowledge of the structure and shape of the maze, including all possible routes to the goal area. The robot must travel to the goal area at least once in order to complete a successful *Exploration Trial*, but is allowed to continue its exploration after reaching the goal. The second trial is where the action happens. In this trial, known as the *Optimization Trial*, the robot will recall the information from the first trial and will attempt to reach the goal area in an optimal route. The robot's performance will be scored by adding the following:

- **Number of total steps in the *Exploration Trial* divided by 30**
- **Total number of steps to reach the goal in the *Optimization Trial***

Additionally, each trial is capped at 1000 steps. The robot can only make 90° turns (clockwise or counterclockwise) and is allowed to move up to three consecutive spaces forward or backwards in a single movement.

Metrics

As discussed in the previous section, the main evaluation metric to quantify the performance of the benchmark and solution models is:

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

This evaluation metric is impacted by both the *Exploration Trial* and *Optimization Trial*, however the *Optimization Trial* will impact the score significantly more than the *Exploration Trial*. The goal is to minimize this score, which would be the result of an optimal trial.

For example: let's assume the robot took 300 steps during its *Exploration Trial*, and 15 steps during its *Optimization Trial*. The score would be equal to: **15** (*steps from Trial 2*) + **300/30** (*steps from Trial 1 divided by 30*) = **25**. If the optimization algorithm found the optimal route for this particular maze, then 25 would be the optimal score.

II. ANALYSIS

Data Exploration

Starter code for this project was provided by Udacity and included:

- **robot.py** - This establishes the Robot class, but is the main script where modifications were made to the project.
- **tester.py** - This script is run to test the robot's ability to navigate the mazes.
- **maze.py** - This script is used to construct each maze and interacts with the robot whenever it is moving or checking its sensors.
- **showmaze.py** - This script creates a visual layout of each maze.
- **test_maze_##.txt** - These consist of three sample mazes to test the robot.

Each maze is drawn on a square grid consisting of either 12, 14, or 16 rows and columns. The maze is enclosed by walls surrounding its perimeter that act as a barrier, blocking all

movement. Our robot starts in the bottom left-hand corner of the maze (coordinate (0,0)) facing in an upward direction. It has walls on its left, right, and bottom sides with an opening on its top side, forcing its first move to be forward (or 'Up'). The robot is tasked with navigating to a 2-by-2 goal area located in the center of the maze. A successful run of the maze is not complete without the robot reaching the goal at least once.

The pertinent information on the structure of each maze is provided by the test_maze_##.txt file, which, to the average eye, will just look like several rows of numbers.

```
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
```

Figure 1: Contents of *test_maze_01.txt*

The first line of the text file is a number (12, 14, or 16) which describes the length of one side of the square maze. The following lines consist of comma-delimited numbers ranging from 1 to 15. These numbers describe the position of walls and openings for each cell in the maze grid. 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.[1] For example, the number 5 translates to a square that has walls on its left and right sides and openings on its bottom and top sides (1*1 + 0*2 + 1*4 + 0*8 = 5). Because of array indexing, the text file describes the maze column-by-column from left to right, so the first

[1] https://docs.google.com/document/d/1ZFCH6jS3A5At7_v5lUM5OpAXJYiutFuSljTzV_E-vdE/pub

number actually represents the starting cell in the bottom-left corner. Maze cell labels follow a Cartesian coordinate system with the rows and columns ranging from 0 to 11.

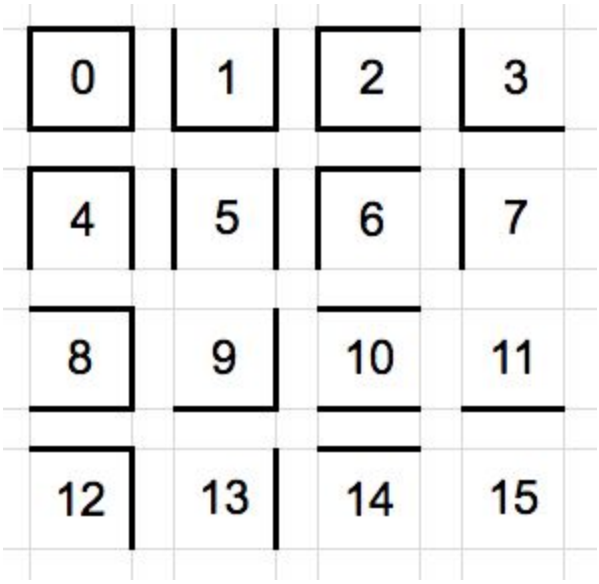

Figure 2: Wall Description Numbers

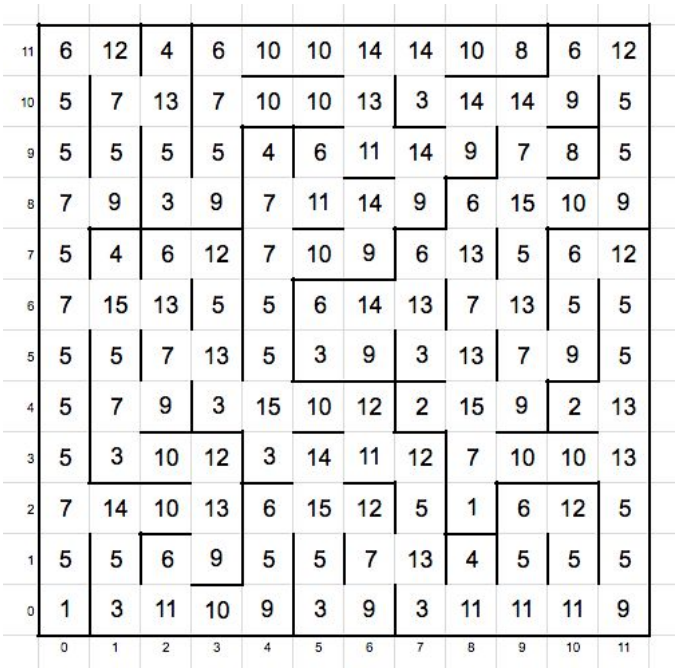

Figure 3: Test Maze 1 with Wall Description Numbers

It can be assumed that the robot is positioned directly in the center of each cell and is facing either left, up, right, or down. The robot is equipped with three sensors. *Sensor 0* is on its left side, *Sensor 1* is on top, and *Sensor 2* in on the right side. Each sensor can detect an opening to an adjacent cell or the presence of a wall blocking its path. Our robot has the ability to precisely rotate clockwise or counterclockwise in 90° intervals and can choose to move forward or backward up to three consecutive spaces. One time-step passes after each movement to an open cell, whereupon the sensors will update their readings for the new position.

Exploratory Visualization

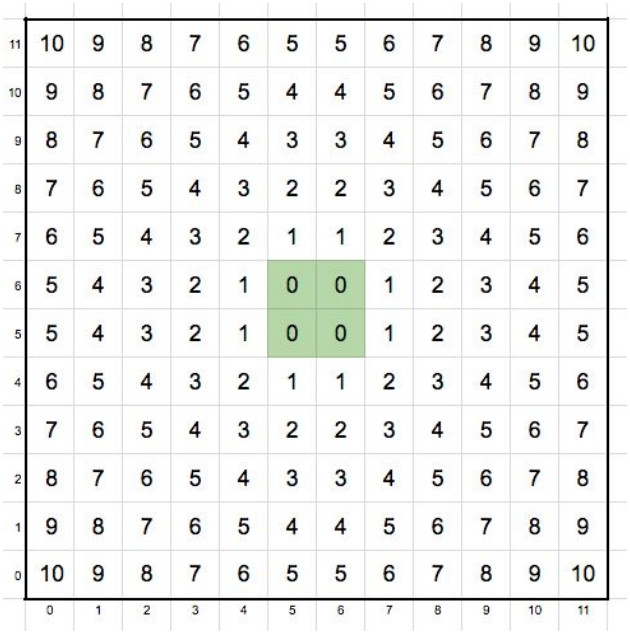

Figure 4: Heuristic Grid for Test Maze 1

The heuristic function is used to help guide A* by providing it an estimate of the minimum cost from any cell to the goal.[2] In our case, the Heuristic Grid represents the maze layout where every cell is labeled 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 labeled '9', its next logical step would be to a bordering cell labeled '8'. It should repeat this process until it reaches '0'.

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

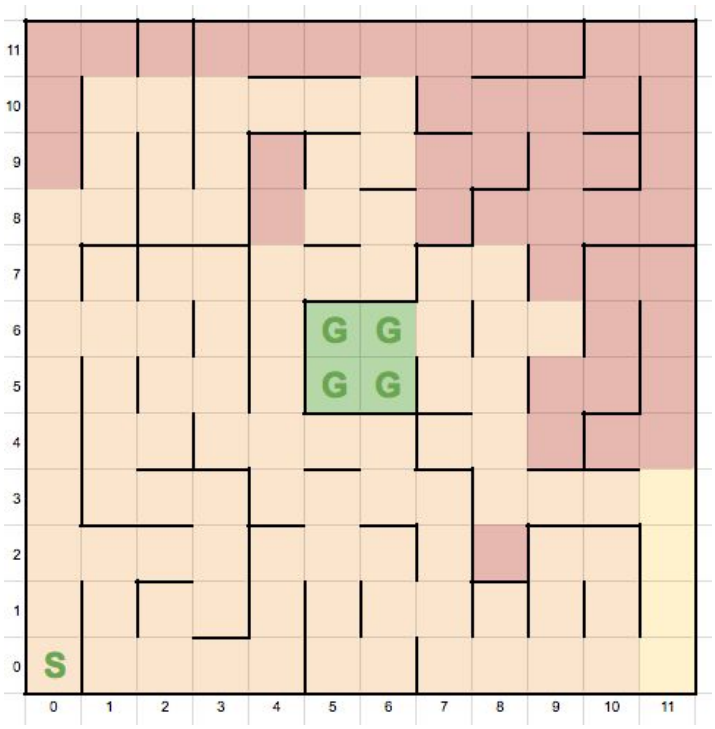

Figure 5: Uncovered cells from *Exploration Trial* in Test Maze 1

Figure 5 displays Test Maze #1, which is 12-by-12 (labeled 0 to 11). The cells highlighted in yellow have all been uncovered by the robot during the *Exploration Trial*, while the squares highlighted in red were not discovered. 'S' is the starting cell and the four cells labeled 'G' represent the goal area. In order to uncover a significant amount of the maze, the robot was directed to continue its *Exploration Trial* (even if it had already found the goal) until it visited at least 70% of the total maze. Ideally, the robot would uncover 100% of the maze, but due to our scoring model, the final score will be negatively impacted by additional *Exploration Trial* steps. Once the robot has discovered 70% of the maze and has found the goal at least once, it will be directed to reset and move back to the starting position to begin the *Optimization Trial*.

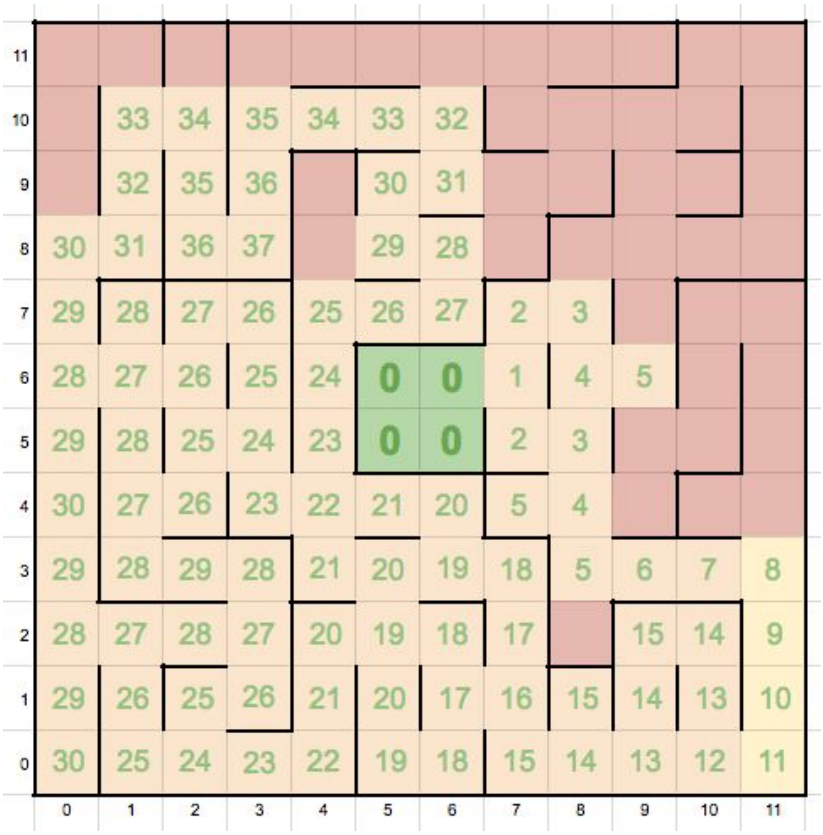

Figure 6: Path Value Grid for Test Maze 1

As the robot maneuvers the maze, it will utilize a value function to create an updated version of the Heuristic Grid, called the Path Value, that takes into account the structure of the maze. In Figure 6 above, the robot starts on the cell labeled ‘30’ in the bottom left corner and is directed to choose to move to bordering cells (that are not separated by a wall) with a smaller number.

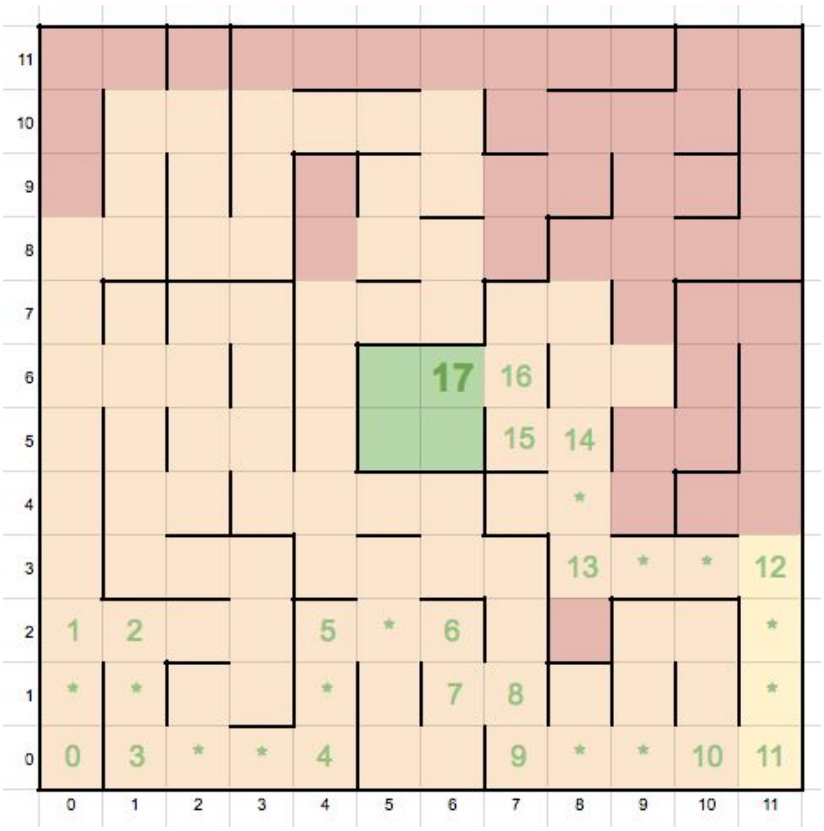

Figure 7: Model's Optimal Path for Test Maze 1

Being steered by the value function, the robot is able to determine an optimal path (to the best of its knowledge). Note that the robot is allowed to move up to three consecutive spaces in any one particular direction per move. An example of this can be seen in its movement between steps 3 and 4 where the robot moves from (1,0) to (4,0), and again between steps 9 and 10, where the robot moves from (7,0) to (10,0).

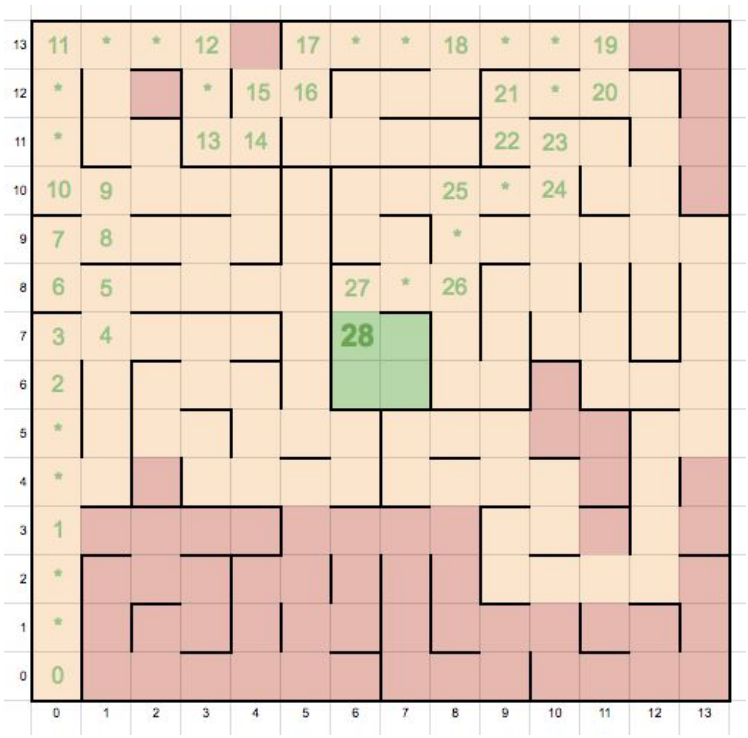

Figure 8: Model's Optimal Path for Test Maze 2

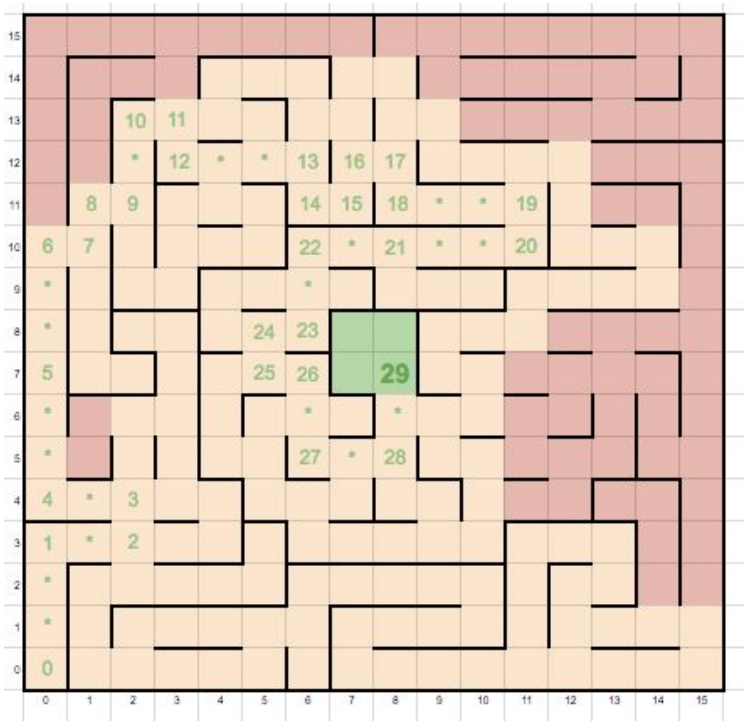

Figure 9: Model's Optimal Path for Test Maze 3

Algorithms and Techniques

When it comes to solving maze problems, the common idiom “there is more than one way to skin a cat” comes to mind. Specifically, there exist several distinct strategies and algorithms one may apply when attempting solve a maze. Before attempting to program an exact algorithm to use, I uncovered a bit of information on several applicable solutions - some more suited for the *Exploration Trial* and others best fitting of the *Optimization Trial*.

Random Turn: Algorithm that makes a random movement decision at each step, slightly favoring the unexplored path. This is entirely simple and slow, but should eventually discover the goal.

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

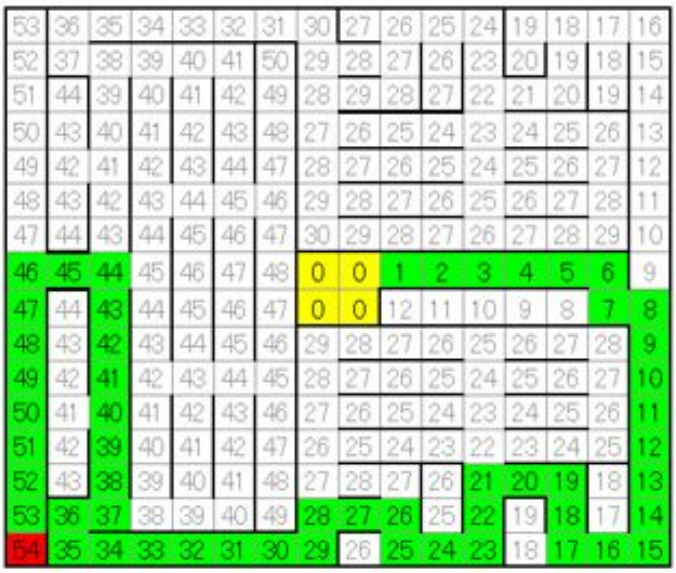

Figure 10: Example of Flood Fill Algorithm[3]

Wall Follower: A Wall Follower algorithm works by simply directing the robot to hug the left side of the wall and take every left turn possible (or equally, right wall paired with right turns), which should eventually lead to the goal area (assuming the robot does not get caught in a circular loop). An example of a Left Hand Rule (LHR) is shown below in Figure 11.

[3] http://isobe.typepad.com/robotics/micromouse/page/12/

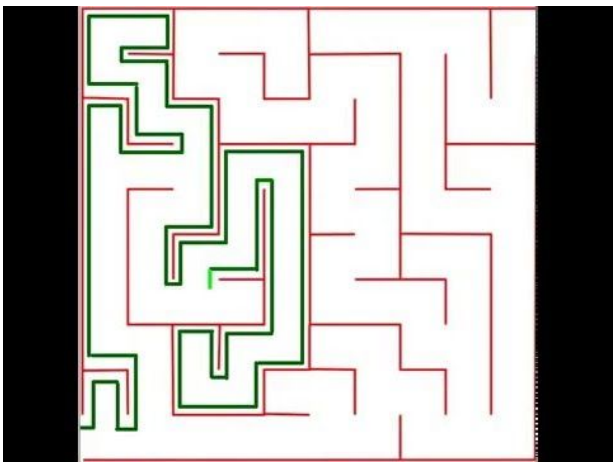

Figure 11: Example of Left Hand Rule[4]

A*: 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 based upon the total cost to take a particular path. Cost can come in the form of shortest distance, least amount of time, fewest turns and more. Finding a solution is accomplished by creating a weighted graph that starts in a specific node and creates a tree path of each possible direction until a path reaches the goal area.

At each step, A* will determine which of its possible paths to expand upon by projecting the total distance to the goal from each path and choosing the minimum cost which is calculated using the value of the path of its initial state with the help of a heuristic estimate of its path cost to the goal. The heuristic typically underestimates the cost of the optimal path to the goal and is used to focus the search.[5]

Dijkstra's 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.[6] Unlike A*, which focuses on finding the path between two single nodes, Dijkstra's algorithm links a path between all nodes on a network.

Depth-First Search & Breadth-First Search: These two algorithms are known as graph traversal algorithms. Depth-First Search starts at a root and follows a single path as far as possible before considering a different path. Similar to Depth-First Search, Breadth-First

[4] https://www.youtube.com/watch?v=NA137qGmz4s
[5] https://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/maxim/files/hsplanguide_icaps05ws.pdf
[6] http://matwbn.icm.edu.pl/ksiazki/cc/cc35/cc3536.pdf

Search will start at a tree root, but will consider all neighboring paths before looking on to its next level neighbors.[7] Both algorithms are suitable for exploring the maze and finding an optimal path, but Breadth-First Search would require the robot to travel back to previously explored parts of the maze to discover new paths, and would thus likely cause a much larger exploration score.

Dynamic Programming: Dynamic Programming is similar to A* in the fact that, provided a map of the maze and a goal location, it will provide an optimal path from any possible starting location.[8] For any cell in the maze, Dynamic Programming will yield an optimum action, known as the Policy (see Figure 12 below).

For this project, I decided to use a modified version of A* for the *Exploration Trial* and used Dynamic Programming to create an optimal policy to use for the *Optimization Trial*.

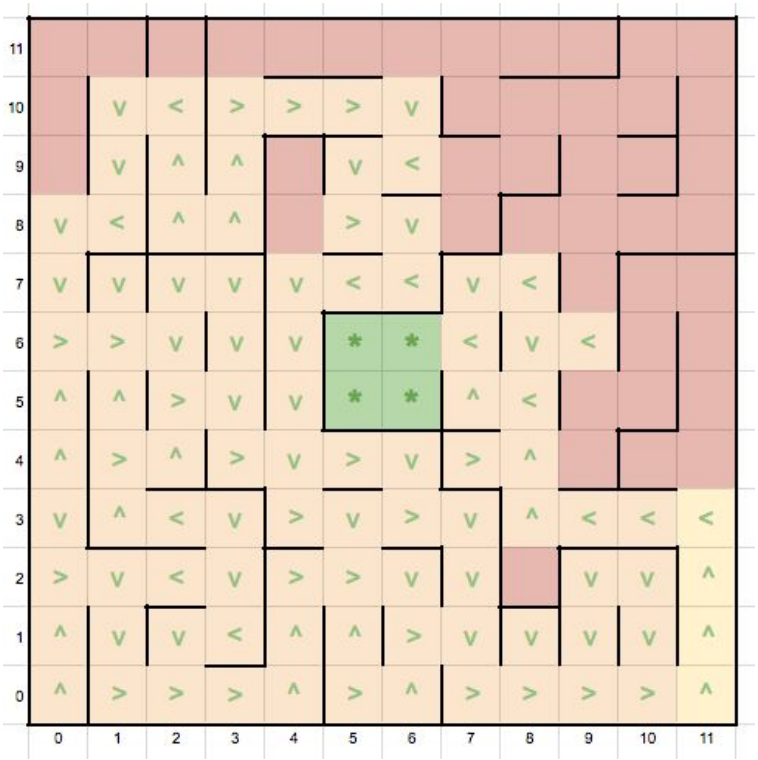

Figure 12: Policy Grid for Test Maze 1

[7] http://people.mpi-inf.mpg.de/~mehlhorn/ftp/Toolbox/GraphTraversal.pdf
[8] https://www.udacity.com/course/artificial-intelligence-for-robotics--cs373

Benchmark

The benchmark model will be evaluated by the performance score previously discussed in the *Problem Statement* section: **score = [Number of Steps in Trial 2] + [Number of Steps in Trial 1 / 30]**. Because we have capped the number of possible steps at 1000, this will be the most basic form of a benchmark performance. The benchmark model will consist of a robot that makes a random movement decision at each step. If the mouse is able to 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 robot has gained knowledge of the maze layout in its *Exploration Trial*, it should then optimize its route to the goal in the *Optimization Trial*. Regardless of the number of steps taken during the *Exploration Trial*, by nature of design, an optimal route exists for each maze - which is the minimum amount of steps required to get from the starting location to the goal area.

Maze 1, 2, and 3 have optimal routes of 17, 23, and 25 steps, respectively. A functional benchmark would consist of the sum of an *average* path to the goal and one thirtieth of an *average* amount of steps needed to explore the maze. The challenge is defining what "average" actually means. Given the size of each maze and the allotted maximum amount 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 a number of 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 will be multiplied by 2.5 to generate the number of steps the benchmark model will take during the exploration phase.

Benchmark Maze 1 Score: 17 + (144 * 2.5) / 30 **= 29.00**

Benchmark Maze 2 Score: 23 + (196 * 2.5) / 30 **= 39.33**

Benchmark Maze 3 Score: 25 + (256 * 2.5) / 30 **= 46.33**

III. METHODOLOGY

Data Preprocessing

No data preprocessing was needed for this project. The robot gathers the necessary information from its sensors and the maze environment. The robot will gather and store information until it has enough to implement an algorithm to optimally solve the maze.

Implementation

After fully defining and understanding the problem, and then considering the uses of varying algorithms, it became apparent that the implementation of this project must be broken into smaller sub-sections.

1. **Interpreting Sensor Readings and Recording Gathered Information**
2. **Robot Movement During *Exploration Trial***
3. **Finding Goal Area During *Exploration Trial***
4. **Identifying Possible and Optimal Paths to Goal**
5. **Ending *Exploration Trial* and Starting *Optimization Trial***

1. Interpreting Sensor Readings and Recording Gathered Information. The robot's knowledge of the maze is stored in a two-dimensional numpy array, called Maze Grid. This list stores the information gained on the actual structure of the maze walls identified from the robot's sensors during its *Exploration Trial*. This working-memory list, populated by the *wall_locations* method, will enable the robot to know the locations of walls and openings in the maze, and what area of the maze is undiscovered.

Starting each time-step, the robot receives a new set of sensor readings from its left, front, and right sensors. Combining the sensor readings with information describing the direction the robot is facing and its current location, the robot is able to generate and store a four-bit number describing the surrounding maze walls and openings for its specific location (previously discussed in the *Data Exploration* section). In addition to the Maze Grid, I created an additional two-dimensional numpy array, called Path Grid, that represented a grid of the undiscovered maze with 0's on 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 particular space in the maze. See an image of the Path Grid in Figure 13 below.

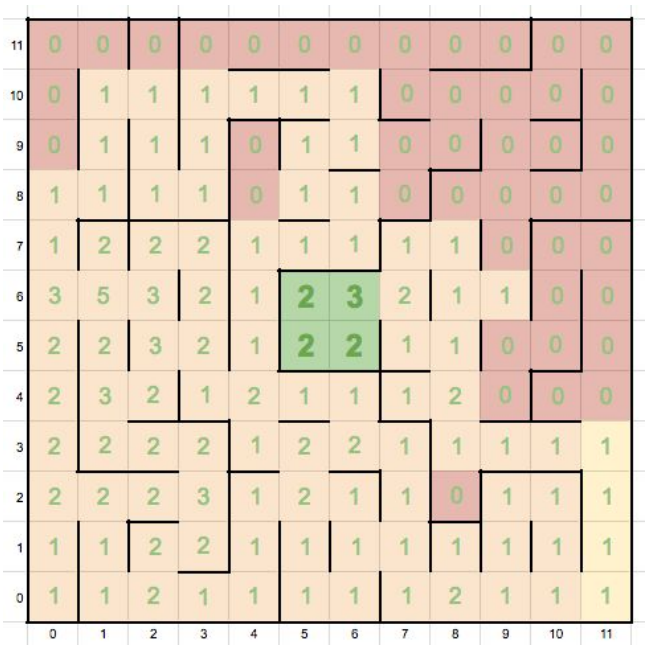

Figure 13: Path Grid for Test Maze 1

2. Robot Movement During *Exploration Trial*. After storing information on the maze walls, the next step is to determine how the robot should move throughout the maze during its *Exploration Trial*. Not only are we concerned with the robot moving through the openings in the walls, but we also want the robot to favor previously-undiscovered cells while maneuvering in a general direction toward the goal area.

Implementing a strategy learned from Udacity's Artificial Intelligence for Robotics class[9], I created a Heuristic Grid to help steer the robot toward the goal area during its exploration phase. Previously discussed in the *Exploratory Visualization* section, the Heuristic Grid represents the maze layout where every cell is labeled with the number of steps away from the goal it is located.

To determine the robot's exploration path, I created a method called *determine_next_move* that guides the robot toward the goal based on its sensor readings, current position and

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

orientation. This method works by first identifying if the robot has reached a dead-end (signaled by sensors identifying walls on all sides), and directing the robot to move backwards if it has. If the robot's sensors identify a single open path in the maze, then it is directed to rotate toward the open path a move into the next cell. If the robot identifies multiple open paths, it is directed to consult the Heuristic Grid and move toward whichever neighboring space is closest to the goal area.

3. Finding Goal Area During *Exploration Trial*. The next process in implementation was determining how to ensure the robot will discover the goal area at least once during its *Exploration Trial*, and how much of the maze the robot should explore after it has found the goal (if continue exploring at all). Upon initializing the Robot class, the robot is already provided the coordinates corresponding to the goal area in an instance variable called *self.found_goal*. 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. If it has found the goal during exploration, a boolean instance variable called *self.discovered_goal* will be set to "True" and the robot will be directed to continue exploring until it has uncovered at least 70% of the map. The robot was required to discover at least 70% of the map in order to possibly discover additional paths to the goal area.

4. Identifying Possible and Optimal Paths to Goal. Now that the robot has discovered the goal area, it is time to determine all possible paths to the goal, and deduce one optimal path for the robot to take during its second trial. Again, borrowing learnings from Udacity's Artificial Intelligence for Robotics course[10], I implemented a Dynamic Programming method, called *compute_value*, that, given the map of the maze and the location of the goal area, outputs an optimal path to the goal from any starting cell.

The Dynamic Programming method works by considering the locations of the walls in the maze and creates a Policy Grid that displays the optimal action (move left, right, up, or down) at every single grid cell leading to the goal area. One issue with Dynamic Programming is that it is very computationally involved.

In order for the Dynamic Programming method to work, I first created a grid called Path Value (see Figure 6) that has the same dimensions of the maze and each cell is initially labeled '99'. After a few iterations, the Path Value grid will have a similar structure to the Heuristic Grid, but adjusted to compensate for the maze walls. After the initial Path Value grid was created, the first thing I did was assign '0' to each of the cells in the goal area, and assign '*' to each of the cells in the goal area of the Policy Grid (see Figure 12). Starting in the goal area and spanning

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

out incrementally wherever there is an opening, each cell in the Path Value grid is updated with a number that corresponds to the amount of steps away from the goal it is located.

5. Ending *Exploration Trial* and Starting *Optimization Trial*. The last portion of implementation is determining when and how to end the *Exploration Trial*, and start the *Optimization Trial*. Once the robot has visited the goal at least once and has uncovered at least 70% of the map, the *compute_value* method is called. Once an optimal path is identified, the robot stores it with the proper movements and rotations to follow the path. 'Rotation' and 'Movement' are both updated to 'Reset' which signals the *Optimization Trial,* aptly named *speed_racer*, to begin.

Refinement

Techniques used for implementation were A* for exploration and Dynamic Programming for optimization. Primarily to help tidy up my code, the first decent sized tweak I made was creating a separate script called *globalvariables.py* that stored all of the global dictionaries used throughout the project such as *dir_sensors, dir_move, dir_reverse*, and more.

I originally allowed the *Exploration Trial* to end once the goal area was found during exploration, but realized that the robot had not even seen half of the maze before stopping and is potentially missing superior paths to the goal. I then required the robot to uncover at least 80% of the maze before moving on to *Optimization Trial*, but ultimately settled on 70%.

Thinking it would limit the amount of moves in the *Exploration Trial*, I initially directed the robot to move -1 space every time it found a dead-end, but it had a tendency of wanting to travel right back into the dead-end when its only possible move was still 'forward' after it had retreated. This ultimately led to the robot reaching the step limit of 1000 by going back and forth between the two same cells over and over again. I then tried implementing logic that would require the robot to move -2 spaces if it had reached a dead-end, and if the previous cell it had visited had walls on the left and right sides. This solution appeared to work wonderfully on the 12-by-12 maze and the 14-by-14 maze, but the robot was unable to find an optimal path in the 16-by-16 maze. Doing a bit of debugging, I realized that the Policy Grid (Figure 12) was not being updated with the optimal directions on cells where the robot had reached a dead-end. 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.

To further refine my code in the future, I would likely spend time preventing the robot from visiting all four cells in the goal area during the *Exploration Trial*. Because the *Exploration Trial* is so lightly weighted on the final score, I did not make this adjustment a priority in my current model, and the robot will visit each cell in the goal area multiple times, even though I recognize this to be unnecessary.

Another weakness may arise if the robot was on a different maze that included numerous dead-ends that span more than three cells, which I have dubbed a “deep dead-end”. The robot’s logic to this point only requires it to move backwards a maximum of two spaces, but it could be prone to an endless exploration loop if moving backwards two spaces did not reveal additional options to travel other than forward (see Figure 14 below). This issue does not arise with the current mazes used for this project, but could be an issue with alternative maze designs.

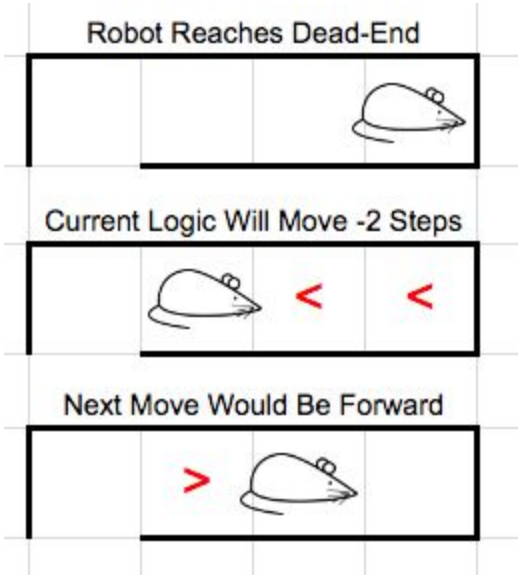

Figure 14: Robot Dead-End Loop

IV. RESULTS

Model Evaluation and Validation

The model is robust because it is programmed to dynamically find the goal in any type of maze. Because it requires a coverage of a certain percent of total cells before moving onto the *Optimization Trial*, the model allows more opportunity to discover several paths to the goal area. The major benefit of Dynamic Programming is that it will reveal a path to the goal from any cell in the maze - not just the starting point.

The final model is reasonable and yields respectable results that outperform the benchmark model. The final parameters are appropriate, but the performance score could be a bit lower if the robot was not required to discover at least 70% of the maze, but having that parameter in place makes the model much more robust for use in other mazes.

The final model was tested with several different required coverage percentage inputs. See results below. An interesting thing to note is the performance of the 14-by-14 maze when the coverage is set to 85%*. This will be discussed in more detail in the *Free-Form Visualization* section.

		Test Maze 1	Test Maze 2	Test Maze 3
	Dimensions	12-by-12	14-by-14	16-by-16
Min. Coverage	Benchmark Score	**29.00**	**39.33**	**46.33**
None	Exploration Steps	141	163	197
	Optimization Steps	17	31	29
	Score	**21.70**	**36.43**	**35.56**
50%	Exploration Steps	141	170	197
	Optimization Steps	17	31	29
	Score	**21.70**	**36.67**	**35.56**
55%	Exploration Steps	141	180	213
	Optimization Steps	17	31	29
	Score	**21.70**	**37.00**	**36.10**
60%	Exploration Steps	141	229	267

	Optimization Steps	17	28	29
	Score	**21.70**	**35.63**	**37.90**
65%	Exploration Steps	141	256	291
	Optimization Steps	17	28	29
	Score	**21.70**	**36.53**	**38.70**
70%	Exploration Steps	146	266	304
	Optimization Steps	17	28	29
	Score	**21.87**	**36.87**	**39.13**
75%	Exploration Steps	170	279	325
	Optimization Steps	17	28	29
	Score	**22.67**	**37.30**	**39.83**
80%	Exploration Steps	184	291	369
	Optimization Steps	17	28	29
	Score	**23.13**	**37.70**	**41.30**
85%	Exploration Steps	227	309	503
	Optimization Steps	17	23	29
	Score	**24.57**	**33.30***	**45.77**
90%	Exploration Steps	276	450	602
	Optimization Steps	17	27	27
	Score	**26.20**	**42.00**	**47.07**
95%	Exploration Steps	306	607	674
	Optimization Steps	17	26	27
	Score	**27.20**	**46.23**	**49.47**
100%	Exploration Steps	315	659	867
	Optimization Steps	17	26	27
	Score	**27.50**	**47.97**	**55.90**

Overall, I would argue that the results can be trusted. Aside from a few small weaknesses (which have already been discussed), the results and performance tend to be consistent, robust, and also routinely outperform the benchmark model.

Justification

Min. Coverage		Test Maze 1	Test Maze 2	Test Maze 3
70%	Exploration Steps	146	266	304
	Optimization Steps	17	28	29
	Model Score	**21.87**	**36.87**	**39.13**
	Benchmark Score	**29.00**	**39.33**	**46.33**

The final results of the model outperformed the benchmark model, however the model failed to find the optimal solution for Test Maze 2 (23 steps) and Test Maze 3 (25 steps), but the routes the model did find were not too far off. By adjusting the preference of taking a route which allows many consecutive steps of two to three spaces, uncovering these routes should be possible.

Although every model can be fine tuned and improved, I would argue that the performance of my model has resulted in a final solution that is significant and robust enough to have solved the problem.

V. CONCLUSION

Free-Form Visualization

In this section I will dig into the performance of Maze 2 with a required coverage of 85%, and attempt to explain why its path was more optimal than the same maze with a required coverage of 90%. As you can see in Maze 2 with 85% required coverage (Figure 15), the robot only needed 23 steps to reach the goal area, which is far fewer than any other trial. Peering deeper, I can conclude a little bit of luck came into play here. Because the robot did not discover cell (4,13) during the *Exploration Trial*, the Path Value became skewed, ultimately leading the robot to make a different path decision at cell (1, 7).

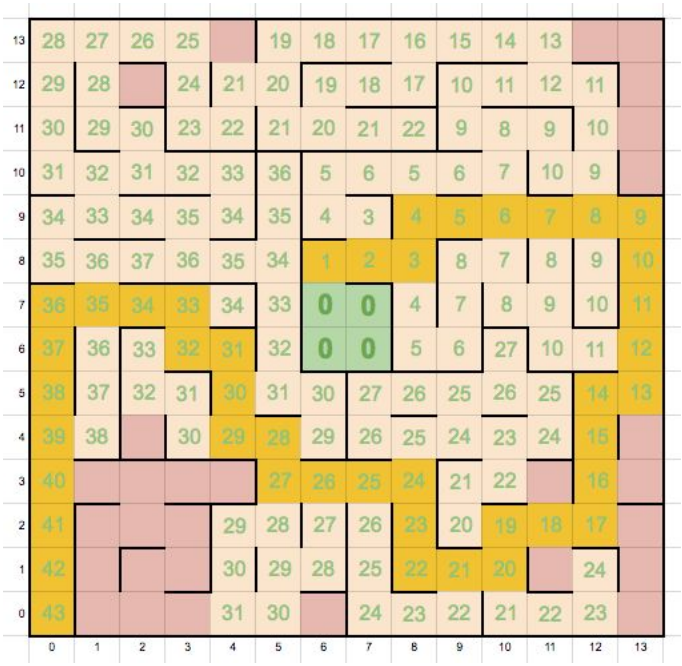

Figure 15: Maze 2, 85% Coverage Required, 23 Steps

Figure 16 shows the Path Value for Maze 2 where 90% coverage was required and the robot took 27 steps. As you can see, starting in cell (1,7), the robot equally could have travelled to cell (2,7) as opposed to cell (1,8), because either cell had a cost of 34.

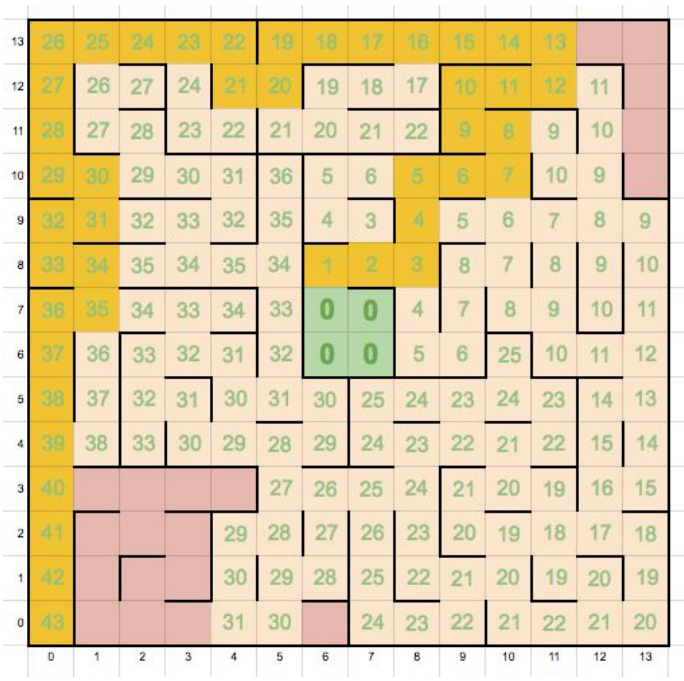

Figure 16: Maze 2, 90% Coverage Required, 27 Steps

In the *determine_next_move* method, an A* technique is used to generate a list of all possible moves the robot is able to take in a particular cell. In cases where multiple paths appear to be equal, the robot will choose the path aligned with the first sensor reading. Because the sensors are list in the order of *Left, Up, Right*, all else being equal, the robot will choose to travel *Left* before *Up*, and *Up* before *Right*. Although both paths span a total of 43 cells, the robot is, unfortunately, not programmed to favor a path that allows making many consecutive moves of up to 3 steps. In the 85% path, the robot has the benefit of covering more spaces by taking larger steps more frequently than in the 90% path. This functionality should be addressed in future revisions, as it can make a large impact on the final outcome.

Reflection

I was inspired to take on this project because I have always had an interest in robotics. My earliest memory is of my older brother chasing after me down the hallway as I was captaining his newest toy - a remote-control robot. Growing up, my dad bought me an erector set that included battery powered motors and I would invent creations for hours on end. Having a background working in the manufacturing space, I got to experience a fully automated factory setting complete with driverless forklifts. Now, with more and more of the products in the world becoming “smart”, my interest in the world of robotics has never been greater.

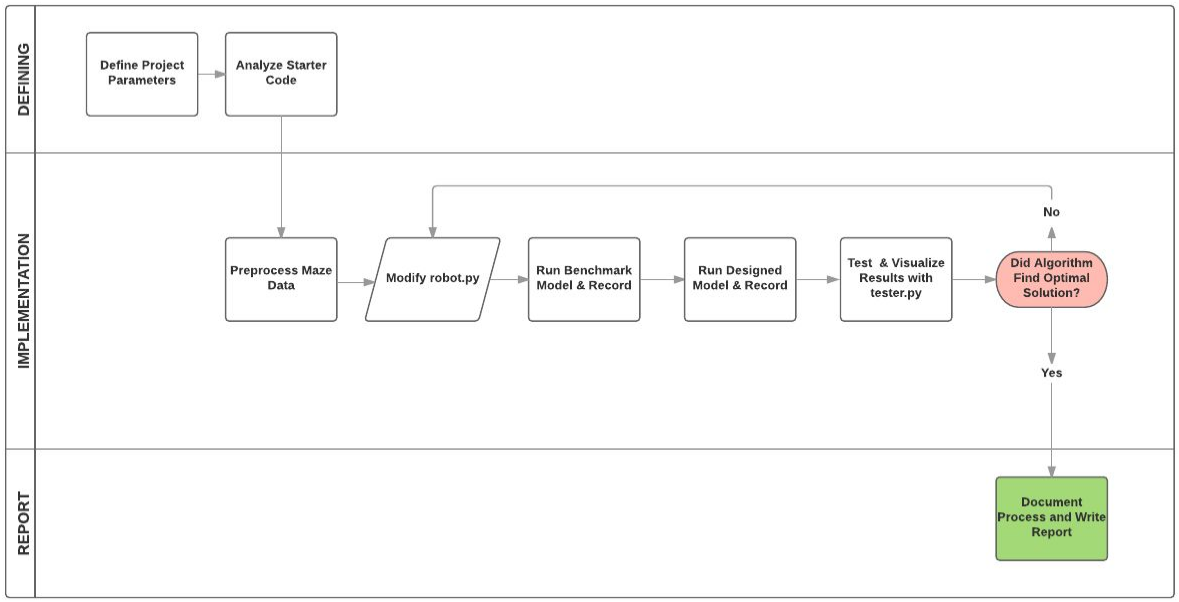

Figure 17: Project Development Process

The entire process used for this project can be defined in the following steps:

1. Record and translate sensor readings into maze information and incorporate it with the robot's orientation (heading) and location.
2. As the robot navigates the maze, update the map of the wall locations in the maze.
3. Implement a heuristic map to help lead the robot toward 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 70% 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.

It felt like a great accomplishment once the robot was actually moving throughout the maze during the *Exploration Trial,* but frustration set in when the robot would get caught in recursive loops, inaccurately map wall locations, and exceed the maximum number of allowed steps. For a tinkerer like myself, I found joy in "getting under the hood" of the code to pinpoint the causes of error, and then conceive and implement possible solutions and patches. On numerous nights, I would go to sleep in frustration after spending hours trying to fix seemingly small errors to no avail, but apparently my subconscious would wrestle with the problem and I would wake up the next day with a fresh solution. Through trial and error, lots of debugging, and a lot of persistence, the robot was finally able to efficiently explore the maze and generate an optimal path to the goal area. And with a bit more keyboard crunching and head-scratching, I was able to develop strategies to further improve the robot's performance.

My final model fits as a solution to the problem, and I would recommend its use in a general setting to solve this problem. I would hesitate to say it is the best possible model, because mazes come in different shapes and forms. Some algorithms are more suited for solving a particular type of maze than others, but this model should serve as a good place to start.

Improvement

For specific improvements, I feel that my method is a bit more computationally involved than other potential solutions, so I would like to find a way to help speed up the script processing. As previously discussed, by implementing a logic that allows the robot to favor and identify optimal routes that allow taking multiple consecutive steps could lead to a great improvement. In addition, identifying and preventing the robot from entering repetitive loops could add to the model's robustness.

I would also be interested in implementing an array of alternative algorithms, mentioned in the *Algorithms and Techniques* section, and comparing their performance against my current model. Because certain algorithms work better with certain maze styles, I would love to implement a model that has an arsenal of multiple path finding algorithms at its disposal. Based on its findings in the *Exploration Trial*, the model will choose to implement an algorithm that is most suited for a particular type of maze layout. For example, if the goal area is located in a corner of the maze, the model may use one type of algorithm to determine an optimal path, but if the goal is located in the center, it would use a different algorithm.

Being a strong believer in continuous improvement, I am certain a better solution exists. A perfect solution is like the core of an onion - it can only be found by peeling away layers of skin and excess. Making small improvements over time will compound to a vastly improved model. That being said, I consider this model to be a "work-in-progress" that can be improved in numerous ways. Our robot has the benefit of perfect sensor readings and a foolproof rotating ability. Unfortunately, real-life situations come with much more complexity and ambiguity, but to an engineer like myself, additional complexity is what makes a project fun.

SOURCES

Udacity Project Files

https://docs.google.com/document/d/1ZFCH6jS3A5At7_v5lUM5OpAXJYiutFuSljTzV_E-vdE/pub

Robots Dreams

http://isobe.typepad.com/robotics/micromouse/page/12/

Maze Solving Algorithms on YouTube

https://www.youtube.com/watch?v=NA137qGmz4s

A Note on Two Problems in Connexion with Graphs

http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf

Micromouse USA

http://micromouseusa.com/

USC Micromouse Project

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

Red Blob Games

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

A Guide to Heuristic Based Planning

https://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/maxim/files/hsplanguide_icaps05ws.pdf

Dijkstra's algorithm revisited: the dynamic programming connexion

http://matwbn.icm.edu.pl/ksiazki/cc/cc35/cc3536.pdf

Graph Traversal

http://people.mpi-inf.mpg.de/~mehlhorn/ftp/Toolbox/GraphTraversal.pdf

Udacity's Artificial Intelligence for Robotics Course

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