Plot and Navigate a Virtual Maze

November 3, 2016

1 Machine Learning Capstone Project

Machine Learning Nanodegree (Udacity) Project submission by Edward Minnett (ed@methodic.io).

1.1 Definition

1.1.1 Project Overview

The primary purpose of this project is to program a virtual robot to explore a simple maze, find a route from a corner of that maze to its centre, and traverse the path from start to finish. Of course, there will be an optimal path and less optimal paths as well as efficient and inefficient ways of finding those paths. The virtual robot's score for a given maze is a calculation based upon how many steps it uses first to explore and then race through the maze. The parameters and context of this project are inspired by the Micromouse robot competition [1] and is, in effect, a virtual version of the Micromouse problem. The robot mouse traverses the maze twice. The first traversal offers the opportunity to explore and map the maze while the second run requires the robot mouse attempt to reach the centre of the maze as quickly as it can using the knowledge acquired while exploring the maze. The goal of this project is to define and implement a strategy to consistently discover an optimal path through a series of test mazes that exist within a virtual world inspired by the Micromouse problem.

1.1.2 Problem Statement

Each maze used in the project follows a strict specification. The maze is a fully enclosed square with an even numbered dimension along each side. The test mazes one, two and three are sized 12x12, 14x14, and 16x16 units respectively. At the centre of each maze is a 2x2 area enclosed by seven walls and one entrance. This area is the goal of the maze, and the robot must enter this space to complete a successful run. The robot will always start in the bottom left corner of the maze where the cell will always have three walls with a single opening at the top.

Each 1x1 cell within the maze occupiable by the robot can have one of 16 possible shapes defined by the presence or absence of a wall on each side. There is a 17th possible shape where all four walls are present, but this cell would not be occupiable by the robot and does not appear in any of the test mazes. It is worth noting that, as mentioned above, the start cell is always the same shape and edge cells as well as the centre goal cells have fewer than 16 potential shapes given the constraints that the maze is fully enclosed and the goal area has a single entrance.

In the case of the virtual environment, a text file defines each maze. The value on the first line of the file states the dimension of the maze followed by a series of lines of comma separated values. Due to array indexing, the first line is the left side of the maze and the first value in the first line represents the bottom-left corner. Each value describes a cell within the maze. These values are four-bit integers from 1 to 15 (0 represents the inaccessible, fully enclosed space). Each bit in the four-bit integer represents a wall or opening, 0 or 1 respectively, on the side of the cell. The bit sequence starts with the 1s at the top of the cell with each additional bit (2s, 4s, and 8s) defining the edge clockwise around the cell. As an example, the calculation of the value for the starting cell, that has a wall on every side except the top, is as follows:

```
1*1 + 0*2 + 0*4 + 0*8 = 1
```

The robot is considered to be in the centre of the cell it occupies and can only face one of the four cardinal directions. The robot is capable of taking perfect sensor readings and making perfect movements. This means the robot is an entirely deterministic agent, but its sensors can only detect the distance to the next wall in each of the three directions, forward, left, and right. The robot is only capable of moving forward and backwards but is capable of moving up to three spaces in either direction. At the beginning of each time-step, the robot receives its sensor readings based on the direction it is facing after the last rotation. It can then chose to keep its current direction or rotate either left or right by 90 degrees before moving forward or backwards. If the robot hits a wall before completing its movement, the robot will remain where it is facing the wall that blocked its path. The act of moving ends the time-step allowing the robot to receive its new sensor readings starting the next sense-move loop.

More specifically, the robot's `next_move` function receives the sensor readings as a list of 3 integers representing the distances to the left, forward, and right closest walls in that order. If the wall is an edge of the currently occupied space, the distance is 0. The `next_move` function must then return two values representing the robot's rotation and movement in that order. The rotation value can be one of -90, 0, or 90 representing counterclockwise, no rotation or, clockwise rotation respectively. The movement value must be an integer between negative three and three including those values. A negative integer is a backwards movement, and positive is forward.

In its first run of the maze, the robot may move freely within the maze to explore and map it. If the robot chooses to end the exploration run, it can return 'Reset' for both its rotation and movement values. This action resets the robot's position to the bottom left corner or the maze and begins the second run. The second run ends when the robot reaches the goal at the centre of the maze.

The robot will need to explore the maze sufficiently and maintain a map of the explored cells to solve the problem outlined above. While exploring, the robot will need to discover at least one path to the goal and visit it before attempting the second run. If the robot explores the whole maze, then it should be able to determine the optimal path to the goal and use that path in the second run. The ideal robot strategy for this problem successfully and consistently finds the optimal path to the goal while minimising the steps required to explore the maze. The Algorithms and Techniques section of this report includes a discussion of several ways to solve these navigation and mapping problems.

A solution to this problem will require, at the very least, the following core processes:

1. Take the sensor readings, combine this information with the robot's location and heading and update the robot's knowledge of the maze.
2. With the updated knowledge of the maze, determine where the robot should travel to learn more about the maze.
3. Move to the desired location and update the robot's heading and location.
4. Continue exploring the maze until at least one path to the goal is found, but preferably find the optimal path to the goal.
5. Start the next run of the maze.
6. Follow the chosen path from the start to the goal.

This is a gross simplification of the logic required to solve this problem but works as a starting point. Later sections of this report will elaborate on this process explore each part of the implementation in much greater detail.

1.1.3 Metrics

For each of the test mazes, the virtual robot moves through the maze twice. In the first run, the virtual robot can move freely through the maze in an attempt to explore and map it. It is free to continue exploring the maze even after entering the goal area. Once the virtual robot has found the goal, it may choose to end the exploration run at any time. For the second run, the virtual robot is expected to traverse the maze and reach the goal as quickly as it can. The score awarded to the robot for each maze is the sum of the two following terms:

- The number of steps taken while exploring the maze during the first run divided by 30.
- The number of steps taken to reach the goal during the second run of the maze.

A virtual robot with a smaller score performs better than one with a larger score.

Each run of the maze is limited to 1000 steps.

It is important to note that this scoring metric penalises both robots that fail to find the optimal path to the goal as well as those that explore the maze in an inefficient manner. That said, a robot that limits exploration and fails to find the optimal path to the goal is likely to, but may not necessarily, perform worse than a robot that takes the time to explore the maze sufficiently and finds the optimal path to the goal.

1.2 Analysis

1.2.1 Data Exploration

We will take a closer look at Test Maze 1 in an attempt to understand the structure of the maze more completely. Below is the maze showing the optimal path to the goal. It takes the robot 17 steps to follow this path.

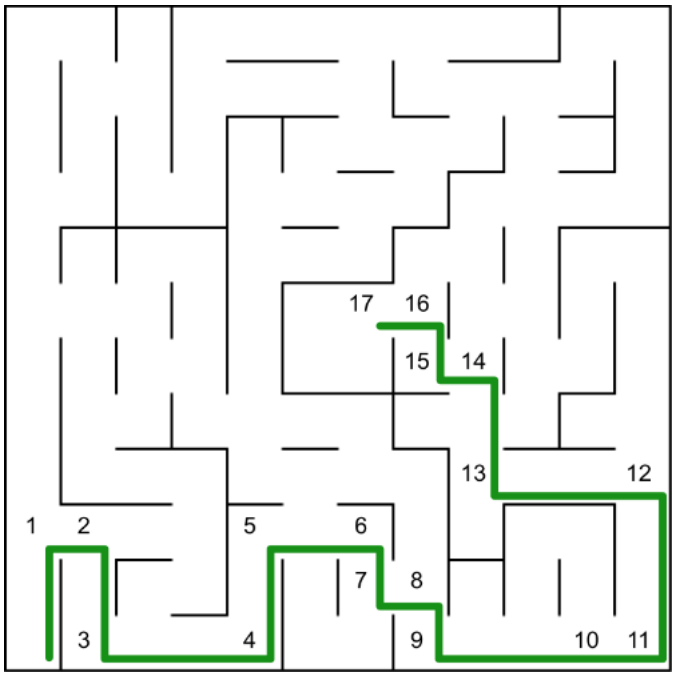


Figure 1: Test Maze 1 with a green line showing the optimal path from the start to the goal and the sequence of steps along that path.

There are a few structural elements present in this maze that are worth noting. The zig-zag walls that meet the two right corners of the goal area mean that all paths to the goal must first travel to one of the right corners of the maze before reaching the goal. Given that the top right corner of the maze is the furthest area from the start position, it is not surprising the optimal path passes around the bottom right corner. It is also interesting to note that there are very few horizontal walls in the left third of the maze allowing the robot to explore that part of the maze more efficiently than it would otherwise. If more corners were present, they would prevent the robot from seeing down long 'corridors'. This shows just how much of the maze the Robot can explore with just a few steps. The first sensor reading alone will provide enough information to show that every single edge within the leftmost cells of the maze are openings.

1.2.2 Exploratory Visualization

The structure of Test Maze 1 discussed in the previous section is even more apparent when laid over a heatmap that describes the path cost to reach each cell. These values are the minimum number of steps required for the robot to reach each cell via any path.

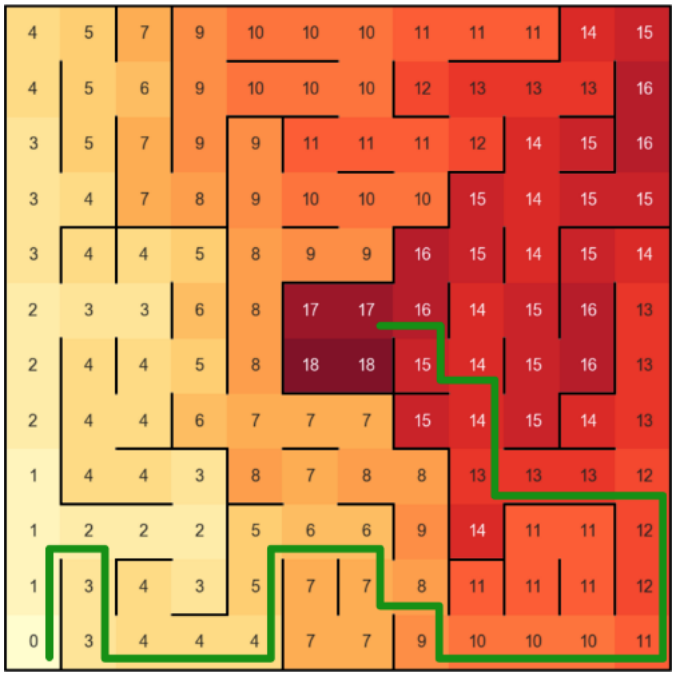


Figure 2: Test Maze 1 with a heatmap showing the path cost for each cell along with the optimal path from the start to the goal.

The heatmap makes it very clear how the robot needs to visit one of the right-hand corners before traversing the rightmost triangular wedge of the maze that leads to the goal. The heatmap also shows how few steps are required for the robot to traverse the left third of the maze. Even without the green line showing the optimal path to the goal, it would be very clear that the robot's best strategy is to head to the bottom right corner rather than the top right corner. That said, this is just an illustration showing how the optimal path compares to the alternatives. In reality, the robot will calculate the optimal path and use an efficient strategy to explore the maze taking advantage of any long corridors within it.

1.2.3 Algorithms and Techniques

The act of first exploring the maze and then finding the optimal path are both search problems. There are a few algorithms that can be used to solve search problems. These include Dijkstra's algorithm [2], A* [3], as well as a host of graph [4] and tree [5] traversal algorithms. The following will explain for which, if either, search problem the algorithm(s) would be suitable.

- **Dijkstra's algorithm** is used to find the shortest path between nodes in a graph [2]. The algorithm takes a graph and then iteratively steps through each possible path through graph storing the smallest path cost for each node. The algorithm selects the node with the smallest path cost and selects all of the neighbours for that node. If the path cost through the current node to reach a given neighbour is less than the current path cost for that neighbour, then the path cost is updated for the neighbour with the lower value. This process is repeated until the fastest path to the target node is found. The optimal path can then be constructed by walking through the graph and following the nodes with the lowest path costs.Once the robot fully explores the maze and there is a graph that describes each fork, then Dijkstra's algorithm could be used to find the optimal path through the graph.
- **A*** is an algorithm used for pathfinding and graph traversal [3]. When given a starting position and target position, A* will progressively search through all possible paths of length n increasing n through the search space and around any found obstacles until the target is found. As soon as the target is found, that path of length n must be the shortest path to the target. There are less naive variants of A* that will also calculate the direct route to the target until an obstacle is found and only then naively

search for a path around the obstacle by evaluating all possible paths around the obstacle. Once the obstacle is circumvented, the algorithm continues to take a direct path to the target either until it is found or until the next obstacle is located. Like Dijkstra's algorithm, A* could be used to find the optimal path through the maze once it has been explored, but could also be used to help the robot explore the maze even when the robot's knowledge of the maze is incomplete. The robot could, given whatever its knowledge of the maze, attempt to move toward the centre of the maze. As walls blocking the robot's path are discovered and the robot's knowledge of the maze increases, the algorithm can be used to circumvent the obstacles consistently. This strategy will find a path to the goal, but may not find the optimal path. Once the robot reaches the goal, A* could then be used to reach the unexplored areas within the maze. At face value, this sounds perfect, but may not be very efficient as the robot would likely pass through already explored areas incurring a scoring penalty.

- **Graph traversal algorithms**. There are two types of graph traversal algorithms, depth-first search and breadth-first search [3]. As the names suggest, depth-first and breadth-first differ in how they select the next node to expand as a part of the search process. Depth-first search will expand a single path until it comes to an end and then start with the next neighbouring node. Breadth-first search will expand all neighbouring nodes progressively working through the graph until the target is discovered. The two strategies can either be a good or poor choice depending what information is known about the graph and the relative locations of the start and target nodes. If the graph is not very well connected and there is a way to filter the spokes of the graph leading from the start node as to choose a spoke with the highest likelihood of including the target, then depth-first search can prove to find the target more efficiently than depth-first search. If, however, the target node could be anywhere in the graph but is thought to be closer to the start node than the average depth of all of the paths leading from the start node, then breadth-first search is likely to be more efficient. Both families of algorithms could be used to both, though independently, explore the maze as well as find the optimal path once the exploration is complete. Both approaches would complete these tasks, but they would not receive the same score. The breadth-first search, which is a special case of Dijkstra's algorithm, requires that the robot continually revisit explored parts of the maze to access the different parts of the unexplored boundary. This strategy would result in a very large exploration scoring penalty.
- **Tree traversal algorithms** are a subset of graph traversal algorithms. T2hese traversal algorithms visit each leaf node exactly once [4]. A graph can be a tree, but a tree is not necessarily a graph. A tree is defined as having exactly one root node and each branch from the root leads to another node. Every node in the tree, apart from the root node, must have exactly one parent node. Every node in the tree can have zero or more children except the root that must have at least one child. Assuming a tree data structure is a suitable representation of a maze, then the tree traversal algorithms would work in a very similar way to the depth-first and breadth-first search algorithms discussed already. They do differ in that trees do not have any loops to contend with and the fractal structure of trees make them very suitable for recursive algorithms. This allows tree traversal algorithms to be very efficient. However, the strict definition of the tree data structure means it would only be a suitable representation of a maze if all paths from the start position lead to one or more dead-ends except for the one path which would lead to the goal. The goal could only be reached via that single path. This maze structure would not be suitable for the large majority of mazes including all three test mazes that each include at least one loop. As a result, using tree traversal algorithms would not present an improvement over graph traversal algorithms.

As we have seen, the majority of these algorithms would be suitable to find the optimal path once the exploration is complete, but are less well suited for the actual exploration of the maze. The other observation is that all of these algorithms, as would be expected, are naive to the specific details of this maze problem. For example, the robot's sensors provide information about more than one space. As we observed in the Data Exploration section of this project, the first sensor reading for Test Maze 1 gives the robot information about 12 cells within the maze. This information allows the robot to explore the maze more efficiently than the naive version of the above algorithms would allow. It would be more efficient to tailor one of the search algorithms to use this additional information.

In reality, each sensor reading can collapse the knowledge of multiple cells simultaneously. This knowledge may allow the robot to know exactly the shape of an unvisited cell or at the very least limit the number

of possible shapes the cell may take. This knowledge in combination with additional sensor readings may then allow the robot to determine with all certainty, as the sensor readings are perfect, the shape of the unvisited cell. The implication of this is that the robot could know the shape of the whole maze without needing to visit all of the cells. The exploration process could be further optimised by determining that the optimal path has been found even with incomplete knowledge of the maze. If the unknown elements of the maze could not result in a more optimal route, then the current optimal path is the path that robot should take. At this point, the robot would maximise its score by terminating the exploration run immediately and proceed to the second run of the maze.

The ideal strategy is to program the robot to keep track of the what is known about the maze, the map, as well as keep track of possible shapes for each space given what has been learnt about the maze so far. A* search can then be used to explore the maze directing the robot to the areas the robot knows least about. This should maximise knowledge of the maze while taking advantage of the fact that the robot can learn about the unexplored areas without visiting them. It should also minimise the number of steps required to learn enough about the maze to find the optimal route. Dijkstra's algorithm can then be used to find the optimal path through a graph representation of the maze given what is known.

1.2.4 Benchmark

The benchmark score for a given maze is the sum of a reasonable path to the goal and one thirtieth of the reasonable number of steps needed to explore the maze given its size. The trick is to define what is 'reasonable'. There is no reason why the optimal path through the maze can not be found as long as the virtual robot completes its exploration of the maze. Any other distance for the final path portion of the benchmark would be entirely arbitrary so the optimal path will be used for the benchmark. For Test Maze 1, discussed earlier, the optimal path is 17 steps.

The exploration portion of the benchmark is much harder to define. This ambiguity suggests that two benchmark scores would be appropriate. An Upper Benchmark score will define a threshold that any robot using even a naive exploration technique should beat while a Lower Benchmark will define a much harder to achieve threshold that should require a robot to use a much more sophisticated and tailored exploration logic. Another way to compare the two scores is that the Upper Benchmark is the largest acceptable score a working robot needs to beat while the Lower Benchmark is the score that a robot should try and beat to prove the quality of the implementation.

To visit every cell of a maze, it would be impossible for a robot to traverse the entire maze and visit every cell without stopping at a cell more than once. Naive exploration logic could be implemented in such a way as to instruct the robot to navigate the maze and stop at every unvisited cell. Once all cells have been visited, then the robot will have a complete map of the maze and will be able to find the optimal path to the goal. This suggests that the Upper Benchmark needs to allow for more exploration steps than there are cells in the maze, but even naive exploration logic should allow the robot to traverse the entire maze without using as many steps as twice the number of cells and if it does, then that is a symptom that something is wrong with the exploration logic. For this reason, the exploration portion of the Upper Benchmark score will be the dimension of the maze squared multiplied by two and divided by thirty.

As discussed above, it would require reasonably sophisticated exploration logic to find the optimal path without visiting even as many spaces as there are in the maze. The exploration portion of the Lower Benchmark score will be the dimension of the maze squared multiplied divided by thirty or, in other words, half the exploration portion of the Upper Benchmark.

The race portion of the two benchmark scores will be the same and match the number of steps required to follow the optimal path to the goal.

The total benchmark scores for Test Maze 1 are as follows:

```
Upper Benchmark: 17 + (12 x 12 x 2) / 30 = 17 + 9.6 = 26.6

Lower Benchmark: 17 + (12 x 12 ) / 30    = 17 + 4.8 = 21.8
```

1.3 Methodology

1.3.1 Data Preprocessing

The nature of this project leaves an absence of data preprocessing. The virtual robot sensor specification and design of the virtual environment have been given as a part of the project definition.

1.3.2 Implementation

After studying the problem for this project and thinking through which algorithms and techniques could be used to solve different aspects of the problem, it became clear that the larger maze mapping and navigation problem should be broken down into the following sub-problems:

- Decide how to store and update the robot's knowledge of the maze.
- Determine how far the robot should move and in which direction to maximise knowledge of the maze while exploring yet minimise the number of steps taken during exploration.
- Find the goal location and make sure the Robot has visited it (this is a requirement to allow the robot to end the exploration run early).
- Once a path to the goal has been found, determine if it is the optimal path.
- Once the optimal path has been found, save it, end the exploration run, and execute the second (race) run.

The process of solving these sub-problems and implementing the code that allows the robot to solve the maze and successfully complete both runs resulted in a further indirectly related sub-problem.

- Determine how to quickly pinpoint and debug logical errors in the implementation.

Decide how to store and update the robot's knowledge of the maze. It was felt that the robot's knowledge of the maze should be stored in two two-dimensional lists. One would store knowledge about the edges bordering each cell of the maze, the Wall Map. This would allow the robot to know where the walls and openings exist within the maze and which edges are of unknown state (wall or opening). The robot's prior knowledge of how the mazes are constructed allowed some of this information to be pre-populated upon initialisation. It could know that all exterior edges are walls and the four edges interior to the goal area must be openings.

The second two-dimensional list would maintain knowledge of the possible shapes each cell could have, the Uncertainty Map. As discussed in the Problem Statement of this report, a given interior cell could have one of 15 different shapes given the permutations for walls and openings with the one exception being four walls. As a result of the prior knowledge of the maze walls, some of the cell shape possibilities could be narrowed down upon initialisation. Corner cells could only have two possible shapes, edge cells could have 7, and goal cells could have 3 possibilities. When the robot is certain of the shape of a cell, the uncertainty for that cell decreases to 1.

At the beginning of each time-step, when the robot receives a new set of sensor readings, the data structure storing information about the walls would be updated with the knowledge of the walls and openings on the left, front, and right sides of the robot given its current location and heading. This updated Wall Map could then be used to update the Uncertainty Map. While the Robot explores new areas of the maze, the uncertainty surrounding the robot will decrease.

The data structure storing the cell shape possibilities is very straightforward. As every maze is square, the outer list, as well as each sublist, has the same length as the dimension of the maze. The data structure that stores knowledge about the cell edges is a bit more complicated. To conform to the array indexing used elsewhere in this project (ie. the first index of the first sublist is the bottom left corner of the maze), the edge information would be stored from left to right and from bottom to top. The first sublist represents the left edge of exterior vertical walls and the next sublist represents the horizontal walls separating the leftmost column of cells within the maze. This pattern then repeats with each sublist alternating between vertical and horizontal edges. What complicates this data structure is that there is one extra horizontal wall for every vertical wall and the total number of sublists is twice the dimension of the maze plus one. This results in atypical and somewhat confusing array indexing, but it does result in a single data structure that stores

information about every edge in a maze without repeating edges (as would be the case if all four edges were stored for every cell even though two neighbouring cells share an edge).

***N.B.**: Even though one of these two-dimensional lists could have been stored as a NumPy matrix, the other could not as the sublists do not have consistent length. As a result, it was decided not to use Numpy for either two-dimensional list for the sake of consistency.*

Determine how far the robot should move and in which direction to maximise knowledge of the maze while exploring yet minimise the number of steps taken during exploration. Now that the robot can keep track of the edges within the maze, via the Wall Map, and the number of possible shapes for each cell, via the Uncertainty Map, it is time to decide how to efficiently explore the maze. The best approach would be to determine the cell with the greatest degree of uncertainty within the maze and navigate the robot toward that cell. If the Uncertainty Map has more than one peak, the robot will attempt to reach the peak closest to it. To find the area of greatest uncertainty, it made sense to scale the values to exaggerate areas of greatest uncertainty. This felt necessary as the values within the map would, for the most part, have the same value especially at the beginning of the exploration run. To differentiate between these values, a scaling factor was devised and then applied to differentiate between the cells with greatest uncertainty. The method of scaling in the initial implementation was to multiply the cells of greatest uncertainty by the radius between that cell and the closest neighbouring cell with a smaller uncertainty value. A naive approach was chosen to simplify the implementation.

For each cell with maximum uncertainty, look at the uncertainty value for the eight cells surrounding it. If all of those cells also have the maximum uncertainty value, add one to the radius and look at the eight cells that define the corners and middle of the edges of the ring around the original cell in question. Once a cell was found with an uncertainty value less than the cell in question, multiply the original cell's uncertainty by the radius of the ring where the more certain cell was found. This new value was then stored in a different two-dimensional list to avoid a multiplicative effect. This approach is far from perfect but felt like a reasonable first approach to finding the areas of greatest uncertainty within the maze.

This, however, was only the first part of the solution. The robot still needed to be able to find a path from its current location to the location of greatest uncertainty. Ultimately, it was decided that Dijkstra's algorithm would be easier to implement and was used to find paths through the maze. To use Dijkstra's algorithm, the Wall Map of the maze would need to be converted into an undirected graph. The graph could then be fed through Dijkstra's algorithm. The path costs generated by Dijkstra's algorithm would then be used to find the fastest path from the current cell to the target cell. This was achieved by working backwards from the target to the current cell following the gradient of smallest path costs.

There were a few complications while implementing Dijkstra's algorithm. It was clear that the optimal paths were not being found as the path costs were not correct for every cell. At first, it was thought this was due to the highly connected nature of the graph and that a given cell could be accessed via a large number of paths. It was later discovered that there were a few errors with the implementation. An initial fix for the problem was to repeat the algorithm multiple times until the path costs no longer changed and had converged upon the correct values. This obviously was not the correct solution but worked well enough to move onto other issues with the code. Ultimately, it was discovered that sorting the unvisited nodes list by path cost was returning incorrect values when the path cost was set to the initial value of `float("inf")`. Rather than using this expression for infinity as the initial path cost, the number of cells in the maze was used instead. Given that no path could be longer than the total number of cells, the code was left logically consistent. This small change fixed the comparison while sorting nodes by path cost and ultimately fixed the original problem with the algorithm. The code then accurately and consistently mapped the path costs and returned the optimal path through the maze.

The path was constructed in such a way that the robot could move further in a single time-step as long as it did not pass through a cell with an uncertainty value greater than 1. This ensures that the Robot would progressively and accurately explore the maze without skipping uncertain cells.

For each time-step in the exploration run, this process would be repeated and the robot would move the number of cells and in the direction as defined by the first step in the path generated by Dijkstra's algorithm.

Find the goal location and make sure the robot has visited it. Knowing the shape of the goal area, a 2x2 square of cells surrounded by seven walls and a single opening, the robot could discover how to get

to the goal of the maze by either finding all seven walls or the single opening. Upon either happening, the uncertainty of all four goal cells would collapse to a value of 1 even if all 8 edges had not yet been sensed by the robot.

Once the entrance to the goal is discovered by the robot, its target location within the maze would become the goal itself. This ensures that the robot has visited the goal before attempting to end the exploration run. Once the goal has been visited, the robot could continue exploring as it had before.

Once a path to the goal has been found, determine if it is the optimal path. Determining whether a found path was the optimal path proved easier than expected. The algorithms used to navigate to the areas of greatest uncertainty within the maze could be re-used. The process would simply require finding the best path through the maze from the start to the goal once using a version of the Wall Map assuming all unknown edges are walls (preventing the path passing through unexplored areas within the maze) and again using the Wall Map assuming all unknown edges are openings. This second path would find the fastest possible path to the goal assuming all walls had been discovered. If the two paths created by this process are of the same length, then the optimal path has been found.

Once the optimal path has been found, save it, end the exploration run, and execute the second (race) run. Once the optimal path has been found, the robot simply stores it along with the movements and rotations required to follow the path. As long as the goal location has been visited, the robot could then complete the exploration run by returning 'Reset' as both the movement and rotation values. The act of performing the second race run of the maze is as simple as keeping track of the time-step and following the movement and rotation instructions for that time-step to follow the optimal path.

Determine how to quickly pinpoint and debug logical errors in the software implementation. The rather complicated indexing of the Wall Map made debugging via conventional means (using PUDB and inspecting variable values) very time-consuming and not very productive. To aid this process, it felt like the best approach was to write a method that printed an Ascii drawing of the maze to the terminal while the robot explored it. This was later improved by adding the location and heading of the robot to the drawing. Even later, the function was amended to display the path costs for each cell within the drawing. This made a great difference when attempting to debug the issues with the implementation of Dijkstra's algorithm discussed previously. Having this information printed to the terminal along with sensor readings, location, heading, movement, and rotation values meant detecting problems in the robot's logic and when the behaviour deviated from the expectation became very fast and easy.

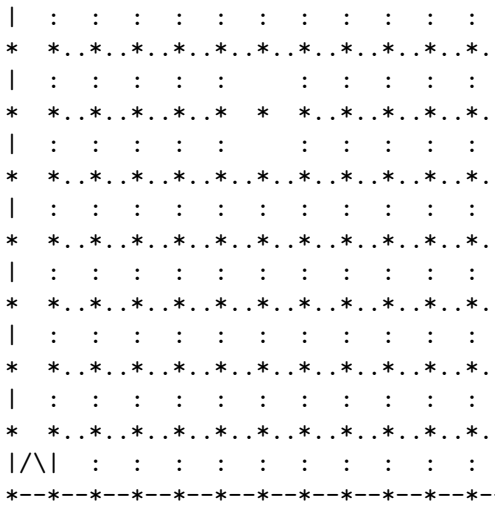
As an illustration, these are the first six terminal outputs for the robot while exploring Test Maze 1 along with the final map with path costs.

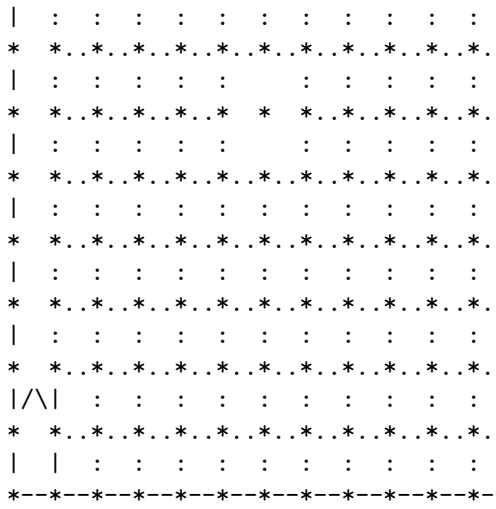
```
The information above each drawing of the maze:
Sensor Readings [left, forward, right] ||| Location (x,y) |
Heading ||| Instructions for the Next Step (rotation, movement)

Knowledge of each edge is represented by lines for walls,
spaces for openings, and dots for unknown states.
```

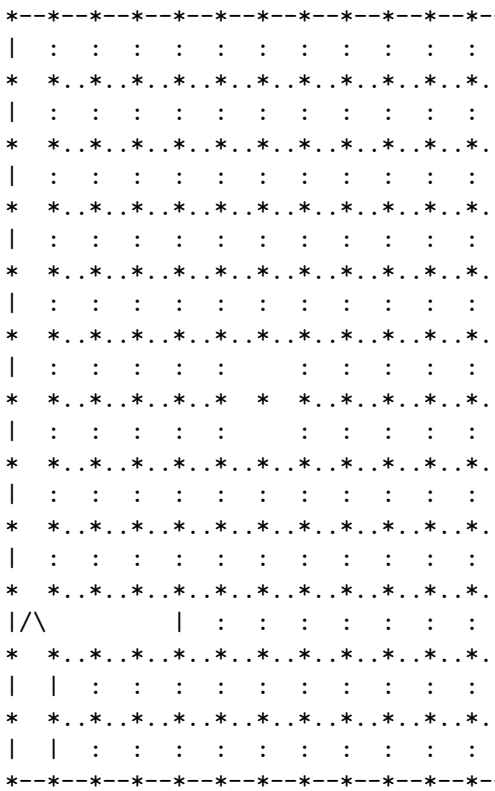
```
Step 1
[0, 11, 0] ||| (0, 0) | up ||| (0, 1)
*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*
|  :  :  :  :  :  :  :  :  :  :  :  :  :  :  |
*  *..*..*..*..*..*..*..*..*..*..*..*..*..*..*
|  :  :  :  :  :  :  :  :  :  :  :  :  :  :  |
*  *..*..*..*..*..*..*..*..*..*..*..*..*..*..*
|  :  :  :  :  :  :  :  :  :  :  :  :  :  :  |
*  *..*..*..*..*..*..*..*..*..*..*..*..*..*..*
|  :  :  :  :  :  :  :  :  :  :  :  :  :  :  |
*  *..*..*..*..*..*..*..*..*..*..*..*..*..*..*
```

```
Step 2
[0, 10, 0] ||| (0, 1) | up ||| (0, 1)
*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*
|  :  :  :  :  :  :  :  :  :  :  :  :  :  :  |
*  *..*..*..*..*..*..*..*..*..*..*..*..*..*..*
|  :  :  :  :  :  :  :  :  :  :  :  :  :  :  |
*  *..*..*..*..*..*..*..*..*..*..*..*..*..*..*
|  :  :  :  :  :  :  :  :  :  :  :  :  :  :  |
*  *..*..*..*..*..*..*..*..*..*..*..*..*..*..*
|  :  :  :  :  :  :  :  :  :  :  :  :  :  :  |
*  *..*..*..*..*..*..*..*..*..*..*..*..*..*..*
```

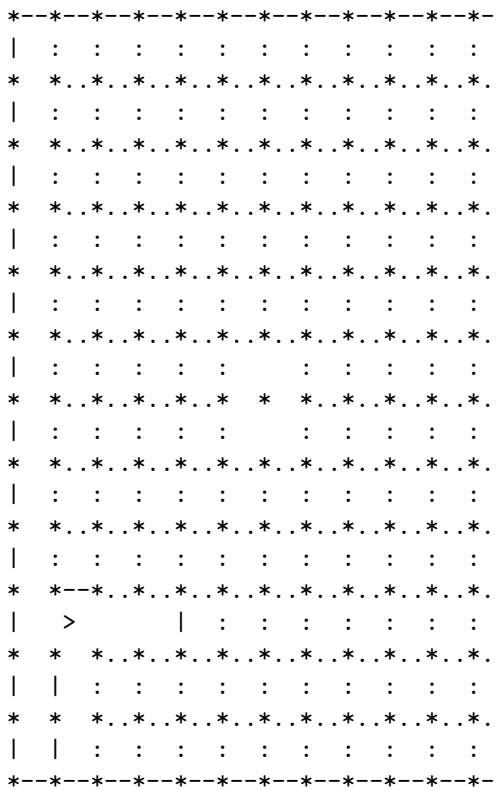




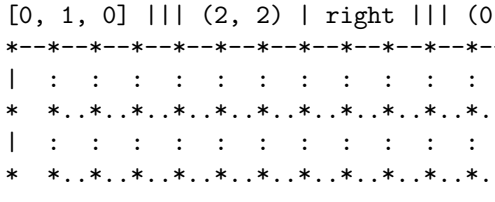
Step 3
[0, 9, 3] ||| (0, 2) | up ||| (90, 1)



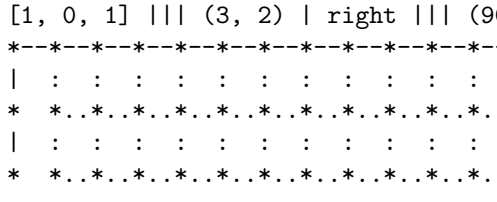
Step 4
[0, 2, 2] ||| (1, 2) | right ||| (0, 1)

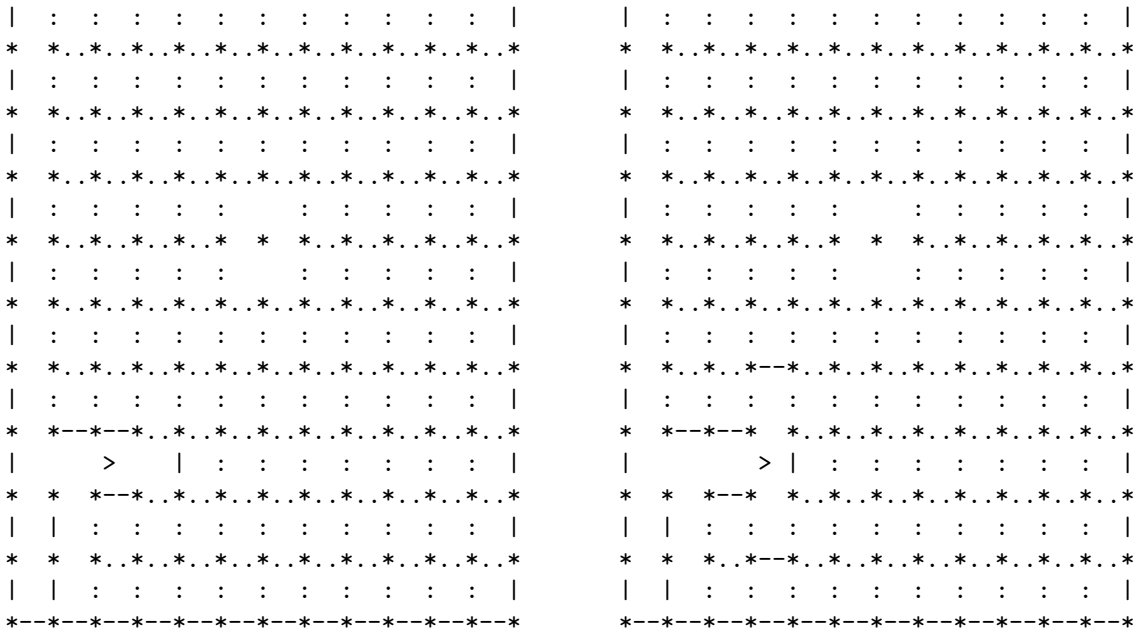


Step 5
[0, 1, 0] ||| (2, 2) | right ||| (0, 1)



Step 6
[1, 0, 1] ||| (3, 2) | right ||| (90, 1)





Goal Location is: (6, 6)
Path costs for each explored space within the maze:

```
*--*--*--*--*--*--*--*--*--*--*--*--*
| 4  5| 8|10 11 11 11 12 12 12|??:??|
*  *  *  *  *--*--*  *  *--*--*..*..*
| 4| 6  7|10 11 11 11|13 13 14 14|??|
*  *  *  *  *--*--*  *--*  *  *..*..*
| 3: 6| 8|10| 9|11 12 11 12|15 16|??|
*  *  *  *  *  *  *--*  *--*  *..*..*
| 3: 6| 8  9| 9 10 10 10|15 15 16 16|
*  *--*--*--*  *--*  *--*  *  *--*--*
| 3: 4: 4: 6| 8  9  9|16 15|15|15 14|
*  *  *  *  *  *--*--*  *  *  *  *  *
| 2  3  3| 6| 8|17 17 16|14:16|16|13|
*  *  *  *  *  *  *  *  *  *  *  *  *
| 2| 4| 4  5| 8|18 18|15 14|16 16|13|
*  *  *  *  *  *--*--*--*  *  *--*  *
| 2| 4  4| 6  7  7  7|15 14 15|14 13|
*  *  *--*--*  *--*  *--*  *--*..*  *
| 1| 4  4  3| 8  7  8  8|13 13 13 12|
*  *--*--*  *--*  *--*  *  *--*--*  *
| 1  2  2  2| 5  6  6| 9|14|11 11|12|
*  *  *--*  *  *  *  *  *--*  *  *  *
| 1| 3| 4  3| 5| 7| 7  8|11|11|11|12|
*  *  *..*--*  *  *  *  *  *  *  *  *
| 0| 3  4  4  4| 7  7| 9 10 10 10 11|
*--*--*--*--*--*--*--*--*--*--*--*--*
```

Where the path cost can not be calculated due
to the cell not having been explored
sufficiently, the value is '??'.

It is clear that the robot's knowledge of the maze increases with each additional step and sensor reading. It is also worth noting that the path costs calculated by the robot, for the fully explored areas of the maze, match exactly the heatmap illustration shown earlier in this report.

1.3.3 Refinement

The process of refining the robot implementation resulted in three primary endeavours. The code was refactored so that instead of keeping all of the logic in the robot class as per the initial implementation, almost all of the logic was moved to navigator and mapper classes both to tidy up the code and to help seperate and encapsulate the logical ownership of information between the classes. This helps both the organisation and the readability of the robot, navigator, and mapper code.

Beyond the improvements and refinements already discuss in the Implementation section above, there were two key additional refinements.

The first was to change the scaling of uncertainty values when the navigator attempts to find the best target cell to navigate to while exploring the maze. Various scaling techniques were explored including taking the sum of uncertainties and adding or multiplying them to peak uncertainty values, but this ultimately made little difference to the robot's performance. The initial idea of finding the location of greatest uncertainty ultimately left the robot exploring in a very inefficient manner. The robot was found to explore one part of the maze for a short time, then move across the maze, explore there, and then return to the first area of uncertainty. Upon reflection of how the algorithm worked, this behaviour was not that surprising. If the maze had two or more areas of uncertainty, the robot would explore one until it was a little bit more certain than another area at which point it would move to the other area. Once it was a little more certain than the first area, it would then repeat this process until it had sufficiently explored both. This left the robot exploring very inefficiently as it was taking many steps moving between the areas of uncertainty rather than exploring one before moving onto the next. The strategy that proved to result in the best score was to leave the uncertainty values unscaled. This allows the robot to progressively seek out each cell of greatest uncertainty without travelling too far to find them. The robot still bounces around the maze to some extent very close to the end of the exploration run when there are far fewer cells with an uncertainty value greater than one. There is, without a doubt, room for additional improvement, but this small change had the greatest impact on improving the robot's score.

The second improvement was primarily implemented to solve an occasional problem with the robot's behaviour but ended up improving the efficiency of the robot's exploration of the maze as well. On occasion, the robot was found to get stuck between two cells bouncing back and forth between them indefinitely. This was clearly because the optimal path the robot's target cell when at cell A was to move to cell B, but when at cell B, the robot's optimal path to the target meant moving back to cell A. This was easily fixed by refactoring the exploration logic. Rather than only using the first step in the path, the robot would use the first two steps in the path generated by Dijkstra's algorithm unless one of the following exceptions occurred:

- The generated path only had a single step.
- The sensor readings after taking the first step in the path exposed walls that would block the robot when taking the second step.

After taking the second step in the path or encountering either of the above exceptions, the exploration process simply reverted back to the original logic seeking out the best path to the closest cell of greatest uncertainty.

Testing the robot against a large number of randomly generated mazes (discussed in greater detail in the next section), illustrated that, under certain circumstances, the robot could still get stuck looping between more than two nodes. This meant that a more robust solution was required to combat these loops. Additional logic was added to search for repeated patterns in the most recently visited nodes. For each loop length between 2 and 6, the navigator determines whether the robot has traversed a set of nodes, retraced its steps back to the beginning of the sequence and then repeated the first set of nodes again. When this occurs, the navigator ensures that robot takes an additional step in the path to its target in an attempt to break the loop. A maximum loop length of 6 was selected by trial and error to minimise the chance of the robot getting stuck without causing a significant impact on performance. The performance was a concern as this process is performed for every step taken by the robot during the exploration run. This additional logic did

resolve the infinite loop problem for quite a few randomly generated mazes, but not all of them. A more detailed discussion of this problem is included in the Free-Form Visualization section of this report.

The progressive change in performance is displayed in the table below. The changes follow the order in which they were made to the code.

Implementation	Test Maze 1		Test Maze 2		Test Maze 3	
	Score	% change	Score	% change	Score	% change
First working version	26.700		31.400		34.567	
Take second step	23.633	11.49%	34.033	-8.39%	34.467	0.29%
Use optimal path	21.233	10.16%	29.433	13.52%	33.267	3.48%
Simplify uncertainties	20.667	9.73%	28.567	2.94%	31.767	4.51%
Break recursive loops	20.667	0	28.567	0	31.767	0
Total Improvement	**6.033**	**22.60%**	**2.833**	**9.02%**	**2.8**	**8.10%**

These refinements along with fixing the implementation of Dijkstra's algorithm resulted in an average improvement of 13% for the scores achieved on the three test mazes provided for this project with the greatest improvement coming from the performance against Test Maze 1.

It helps to know the Upper and Lower Benchmark scores for each maze to put the improvements into context and better understand how important these improvements have been.

	Robot's First Score	Upper Benchmark	Lower Benchmark	Robot's Final Score
Test Maze 1	26.700	**26.600**	**21.800**	20.667
Test Maze 2	31.400	**35.067**	**28.533**	28.567
Test Maze 3	34.567	**42.067**	**33.533**	31.767

The first working version of the robot did not even pass the Upper Benchmark for Test Maze 1 though it did for the other two mazes. It was not until the improvements were complete that the robot was able to beat the Lower Benchmark score for Mazes 1 and 3 and come very close to that threshold for Test Maze 2.

1.4 Results

1.4.1 Model Evaluation and Validation

To adequately evaluate the performance of the robot navigation and mapping logic, it felt as though the three test mazes would not be enough to confirm how well the robot would perform when confronted with a greater variety of maze shapes and sizes. Instead, it was decided that a larger sample of randomly generated mazes would be needed. They were created using a modified version of the Recursive Division maze generation algorithm [6]. This process is quite simple to implement and ensures that all cells are accessible and, by definition, ensures that every maze is solvable. In essence, the maze generation process takes a blank maze starting with the initial conditions required to conform to this project's problem statement and then progressively alternates between randomly selected sections of horizontal and vertical edges. The basic Recursive Division algorithm takes stretches of walls from edge to edge between areas where walls / openings have already been defined, but depending on the random selection, this method can result in mazes that have long corridors with few openings ultimately limiting the variety in mazes generated. Instead, the edge selection criteria limit each selection to a stretch of only four edges. From these four edges, a single opening and three walls are defined. This proved to result in a much greater variety of maze shapes than the basic Recursive Division algorithm. Inevitably, there will be similarities in the maze structure given the method of generation, but this felt to be a reasonable compromise.

The maze generator was then set to generate 100 mazes for each even dimension from 12 to 20 creating a total of 500 test mazes.

The robot successfully completed all but 17 of these mazes. While attempting the 17 outliers, the robot entered a recursive loop it could not escape. The 12x12 example from these 17 mazes will be discussed in more detail in the Free-Form Visualization section of this project.

The robot performed so well when compared to the maze benchmark outlined in the Benchmark section of this report, that a stricter benchmark was defined to better analyse the results.

The race benchmark is the same for both the Upper (less strict) Benchmark and Lower (more strict) Benchmark, but the exploration scores differ by a single factor. The Upper Benchmark includes an explor1tion score that is based on the robot stopping in each cell twice while the Lower Benchmark allows the robot to, on average, only stop at each cell once.

Figure 3 shows the difference between the robot's score and each of the two benchmarks for each successfully completed maze for each of the five maze dimensions. This difference is defined in such a way that a larger positive value indicates that the robot outperformed the Benchmark score by that amount. This means that a larger positive number is better than a smaller number. Negative numbers indicate that the robot did not beat the benchmark.

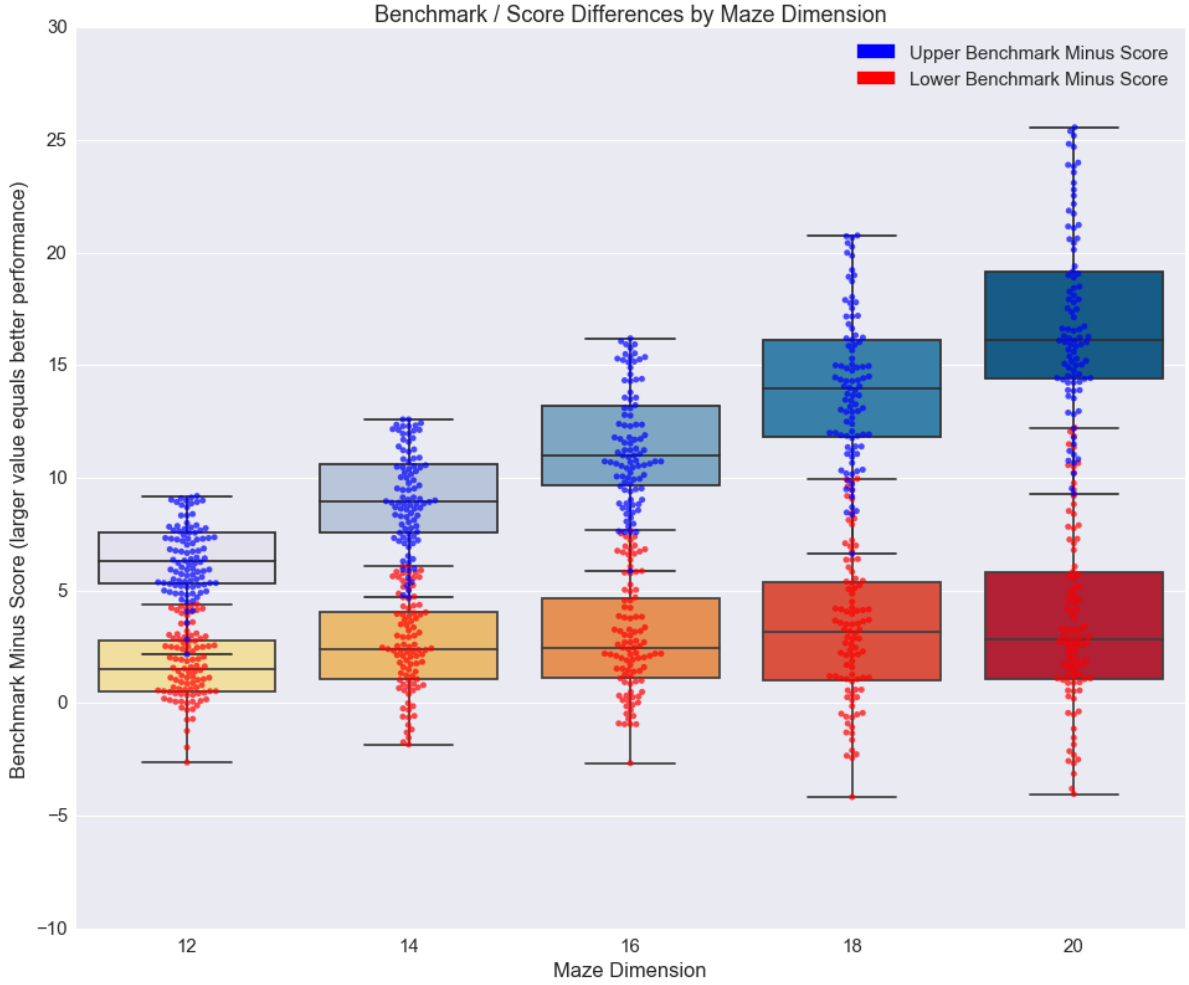


Figure 3: The robot's scores compared to the benchmark scores across all 483 completed randomly generated mazes.

It is clear that for the very large majority of mazes, the robot beat both of the Upper and Lower Benchmark scores. As the maze dimension increases, the robot outperformed the benchmarks by a larger amount.

1.4.2 Justification

As displayed previously, the Benchmark scores for the original 3 test mazes are as follows along with the maze dimensions, optimal path length, and the robot's final scores:

	Dimension	Optimal Path	Upper Benchmark	Lower Benchmark	Robot's Score
Test Maze 1	12	17	26.6	21.8	20.667
Test Maze 2	14	22	35.067	28.533	28.567
Test Maze 3	16	25	42.067	33.533	31.767

The robot's scores for these three mazes are 20.667, 28.567, and 31.767 respectively. This means that the robot managed to find the optimal path in only 110 steps for the first maze (34 steps fewer than the number of cells), 197 steps for the second maze (1 step more than the number of cells), and 203 steps for the third maze (53 steps fewer than the number of cells). The robot outperformed the Lower Benchmark for two out of three mazes and missed this benchmark by only a single step for the second maze suggests that the solution implemented for this project adequately meets the performance requirements.

A closer examination of the robot's performance against the 483 completed randomly generated mazes shows the following:

- The robot outperformed the Lower Benchmark for 420 of the 483 mazes (86.96%).
- On average, the robot outperformed the Lower Benchmark by 2.77.
- On average, the robot outperformed the Upper Benchmark by 11.53.

Given these results, it feels as though this implementation of the robot successfully solves the maze mapping and navigation problem as defined for this project.

1.5 Conclusion

1.5.1 Free-Form Visualization

The random maze generator used to evaluate the robustness of the robot navigation algorithms managed to generate several mazes that caused the robot to get stuck. Figure 4 depicts the single 12x12 example where the robot was unable to complete the maze.

After exploring the maze nearly to completion in 131 steps, the robot cycles indefinitely causing the run to time out after 1000 steps. This is particularly unexpected considering the lengths taken to improve the navigation robustness specifically to avoid this behaviour. When the robot gets stuck, it repeats the following 33 node loop again and again.

```
(2, 7) (2, 8) (3, 8) (2, 8) (2, 7) (2, 8) (3, 8) (2, 8) (2, 5) (2, 8)
(3, 8) (2, 8) (2, 7) (2, 8) (3, 8) (2, 8) (2, 7) (2, 8) (3, 8) (2, 8)
(2, 5) (2, 8) (3, 8) (2, 8) (2, 7) (2, 8) (3, 8) (2, 8) (2, 7) (2, 6)
(2, 8) (3, 8) (2, 8)
The robot then moves to (2, 7) to reach the beginning of the loop
causing it to repeat the path again.
```

This behaviour is clearly the result of the combination of the exploration traits where the navigator follows the first two steps in a path and the fact that when looking for loops within the visited nodes, the navigator attempts to match a path three times (in the original direction, retracing those steps, and repeating the original set of steps. The loop condition for a loop of 3 nodes is met several times within the larger loop. The following is the three node loop repeated three times that occurs at the beginning of the 33 node loop as well as within it.

```
(2, 7) (2, 8) (3, 8) (2, 8) (2, 7) (2, 8) (3, 8)
```

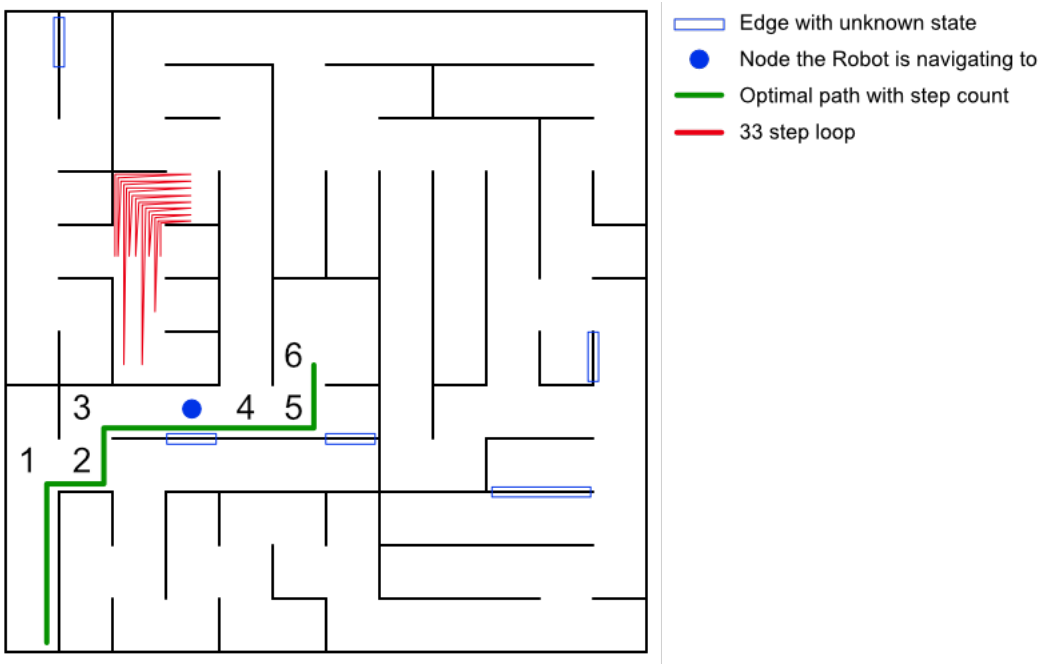


Figure 4: The problematic 12x12 maze annotated with the robot's knowledge of the maze, the loop where robot gets stuck, and the optimal path through the maze.

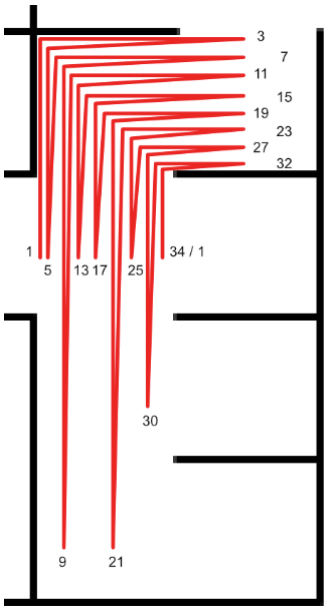


Figure 5: Closer view of the loop within the context of the maze shown above with the step numbers from 1 through 33.

The logic used to combat this behaviour is what causes the nodes (2,5) and (2, 6) to show up within the larger loop though less frequently than the other nodes.

What is surprising is that the optimal path to goal is very short and straight forward. Even with the robot's incomplete knowledge of the maze, the path to the goal only requires six steps. This suggests that the robot would have ideally stopped exploring long before it got stuck. However, the reason it did not is that there is a possible path that would reach the goal in one fewer steps if the wall between (3, 3) and (3, 4) was an opening instead of a wall.

1.5.2 Reflection

At its simplest, the problem and solution for this project can be described by the following process:

1. Receive a small amount of information about the maze (sensor readings).
2. Combine this information with the heading and location of the robot.
3. Update the maze map with any new information inferred directly or indirectly from the sensor readings.
4. Given the current knowledge of the maze, decide where to travel to maximise the acquisition of new information.
5. Decide how to get from the current location to the desired location.
6. Move toward the desired location.
7. Repeat steps 1-6 until the optimal path from the start to goal locations has been found.
8. Start the next run of the maze.
9. Follow the optimal path from the start to the goal.

As straightforward as this sounds, the actual implementation of the robot that can consistently and successfully complete this process in an efficient manner even when faced with a large number of maze permutations is considerably more difficult. Even after iterating through several techniques to prevent the robot from getting stuck, it still managed to enter recursive loops that it could not escape. An entirely robust solution would likely increase the complexity of the exploration logic and might require sacrificing the efficiency by which the robot explores the majority of mazes it faces and thus causing the robot to perform worse on the majority of mazes simply to perform better for a small number of others. Whether this would be an acceptable compromise is largely academic and is outside of the scope of this project (given the current project parameters and scoring rules).

Despite this difficulty, the algorithms used to map and navigate the maze are quite straight forward. It is interesting to consider how a small number of reasonably simple algorithms, when combined in a coherent way, can result in quite complex behaviour and manage to solve the problem at hand so well. The two most complicated algorithms used in the project are the algorithm that converts the maze map into an undirected graph and Dijkstra's Algorithm used to find the optimal path through that graph. Even for these two algorithms, the implementation is reasonably simple even if debugging and refining them was more challenging than expected.

The most interesting part of the project was, once a basic implementation of the robot that managed to complete the maze was developed, to find ways to improve the performance of the robot as well as its robustness. Devising ways to test the robot and see how it would cope dealing with different shaped mazes further increased the enjoyment of solving these problems.

1.5.3 Improvement

If for a moment we consider the impact of extending the problem for this project, we will see just how much more difficult solving maze mapping and navigation problems could be. If instead of having a well-defined maze where sensor readings, movements, and rotations are perfect, walls have no thickness and the robot is always perfectly positioned within a cell, and instead, movements and sensor readings were noisy then the implementation used for this project would no longer work quite so well. Having a thickness of the walls and a diameter fo the robot would result in sensor readings that would be more challenging to work with, but not by much.

For example, if the wall thickness was 0.1 (0.05 in each cell between which the wall resides) and the robot's diameter was 0.4 located at the centre of the occupied cell, the robot's first sensor readings for Test

Maze 1 would have been `[0.25, 11.25, 0.25]` instead of `[0, 11, 0]`. The new sensor readings could be easily converted to the original readings by simply taking the floor of each value converting them to the easy to work with integers.

If, however, the new sensor readings, movements and rotations were no longer perfect and were, in fact, probabilistic, the robot would need to use a very different approach for mapping and navigation. For this type of robot control scenario, there are a few algorithms that could be employed to solve different aspects of the problem.

- **Kalmen Filters** takes noisy measurements observed over time of unknown variables and produces estimates for those variables that are more precise than a single measurement. The algorithm applies Bayesian inference and develops a joint probability distribution for the variables within the context of the timeframe during which the observations are made [7]. Kalmen Filters can be used for robot navigation and localisation.
- **Particle Filters** is a genetic type mutation-selection particle algorithm where a collection of virtual representations of the robot are initialised randomly [8]. With each additional observation, the virtual representations are filtered based on the likelihood that the given representation accurately describes the state of the robot. Over time, the Particle Filters converge on the best estimate of the state of the robot. Like Kalmen Filters, Particle Filters can be used for navigation and localisation.
- **PID Controller** is a control loop feedback mechanism that continuously calculates an error term that is the difference between the desired value and an observation of that value. The controller then corrects the robot's movement using proportional, integral, and derivative terms (hence PID) to allow for the steady convergence of the robot's position and the desired position [9]. This mechanism can be used to control a robot's movements when those movements are noisy or delayed (such as steering a ship).
- **Simultaneous Localisation and Mapping (SLAM)** is a process that allows a robot to simultaneously build up a map of an unknown environment while navigating it [10]. SLAM describes a group of algorithms and techniques that, when implemented in concert, work together to solve the SLAM problem. SLAM applies Bayes rule allowing for the sequential updating of localisation posteriors given a map and transition functions. Through a similar method, the map can be sequentially updated. SLAM has obvious applications for robot mapping and navigation.

A simpler way to make this project more challenging without having to significantly alter the project parameters is to require the robot to navigate and map the maze without informing it of the dimension of the maze from the outset. This would force the robot to not only update its knowledge of the edges within the maze but also expand the map itself as new parts of the maze are discovered and explored. Even more challenging would be to allow the map to have an odd number of edges on each side with a 3x3 goal area in the centre but keeping all other rules consistent. The combination of these two changes would significantly increase the challenge presented by this project.

As it was defined, this project presented an enjoyable problem to solve without being too difficult.

1.6 References

1. Wikipedia contributors, "Micromouse," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Micromouse&oldid=709118923 (accessed March 9, 2016).
2. Wikipedia contributors, "Dijkstra's algorithm," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=745950368 (accessed October 24, 2016).
3. Wikipedia contributors, "A* search algorithm," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=744637356 (accessed October 16, 2016).
4. Wikipedia contributors, "Graph traversal," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Graph_traversal&oldid=703421335 (accessed February 5, 2016).

5. Wikipedia contributors, "Tree traversal," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Tree_traversal&oldid=745817776 (accessed October 23, 2016).

6. Wikipedia contributors, "Maze generation algorithm," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Maze_generation_algorithm&oldid=745040926 (accessed October 18, 2016).

7. Wikipedia contributors, "Kalman filter," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Kalman_filter&oldid=744979113 (accessed October 18, 2016).

8. Wikipedia contributors, "Particle filter," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Particle_filter&oldid=747290664 (accessed November 1, 2016).

9. Wikipedia contributors, "PID controller," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=PID_controller&oldid=747236397 (accessed November 1, 2016).

10. Wikipedia contributors, "Simultaneous localization and mapping," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Simultaneous_localization_and_mapping&oldid=747305950 (accessed November 1, 2016).