Capstone Project

Programming an Autonomous Robot

Machine Learning Engineer Nanodegree

Timothy Watson
February 22, 2017

Definition

Project Overview

The aim of this project is to design an algorithm by which a virtual robot mouse can learn the specifics of a virtual maze and then complete the maze in the fastest time possible. The project is based on the idea of the micromouse competition where competitors build a robot mouse with the goal of finding the shortest path to the center of a real maze. In the current project, besides being entirely virtual, the mouse differs in that the sensors and movements are without error. The performance of the virtual mouse will be graded on the basis of the combined time it takes to explore and then complete three different virtual mazes.

Problem Statement

The objective of this project is to use python to implement a virtual robot that can determine the ideal path to the center of three different virtual mazes of differing sizes and geometries. Each maze contains a goal location at its center and is a 2x2 grid. The three mazes have the following dimensions: 12x12, 14x14 and 16x16 with walls along the outsides. The virtual robot is given two distinct runs at the maze. In the first run, the robot is allowed to explore the maze in any way that it wishes and may end the run at any time. The purpose of this run is to gain as much information about the maze as possible in the shortest time frame. In the second run, the robot must attempt to reach the goal in the shortest time possible. In total, for both runs, the robot is allowed a total of 1000 time steps.

In this project, we will attempt to minimize the time required for both runs. An important factor to consider is that the more we learn in the first run, the faster we can potentially solve the maze in the second run. For this reason, we should seek to experiment with different methods of exploration in the first run to optimize the second run and the over all score. The best solution will be one that balances exploration with speed in the second run to achieve the best score possible.

Metrics

In this project, we are looking to minimize the total score accumulated over the two runs. In the first run, which will be referred to as the exploration phase, the robot is given one point for each 30 steps that it takes. During the second run, or the completion run, the robot is given a point for every step it takes. The total score can then be measured on the basis of the following equation:

$$Total\ Score\ (t_1, t_2) = 1^{st}\ run\ score + \ 2^{nd}\ run\ score = \frac{t_1}{30} + t_2$$

In this Equation, t_1 is the number of steps in the first run and t_2 is the number of steps taken in the second run. Therefor, by lowering the amount of steps we take in both runs, we can lower our total score, however it is important to note that the number of steps taken in the second run contributes far more significantly to the total score.

Analysis

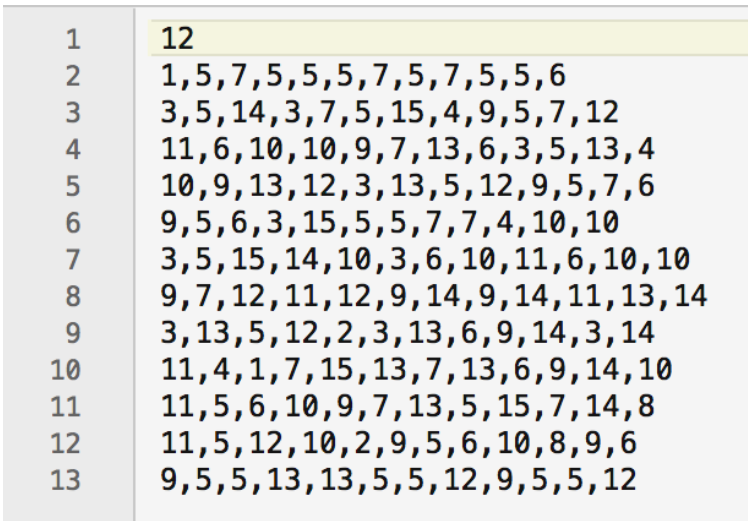
Data Exploration

At the outset of this project, we are provided with two things: the specifications of the mazes that the virtual robot will be asked to navigate, and the specifications of the robot itself, including details related to its sensors and movement abilities.

Maze Specification

The maze is laid out as a *nxn* grid and *n* is always even and between 12 and 16. The outside perimeter of the maze is all walls and all of the walls block movement. The inside of the maze contains walls at various locations. The robot always starts in the bottom left square of the grid facing upwards and with a wall on its right. The goal is always located in the center of the maze and is a 2x2 square.

We are initially provided with information about the virtual mazes in the form of text files (for example test_maze01.txt). These text files contain a series of numbers which serve to specify the wall locations in each cell of the maze. For example, test_maze01.txt contains the following:



```
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: Example data contained in test_maze_01.txt

The first number (12 in this case) indicates the dimension of the maze which is always square. The numbers on the following lines represent each of the squares in the maze. Importantly, this representation is actually a counter-clockwise rotated representation and that the columns in the text file correspond to the rows of the maze and vice-versa (the first column of the text file corresponds to the first row of the maze – a more accurate visual mapping requires simply rotating this image clockwise by a quarter turn).

The numbers contained in the file correspond to the location of walls in each cell. By converting the given numbers to their binary form and by looking at whether each character is a “1” or a “0”, one can determine the location of walls in that cell. A “0” corresponds to a wall in that location and a “1” corresponds to an opening in that location. The first digit in the binary number represents the wall on the left, the second represents the bottom wall, the third digit represents the right wall and the fourth digit represents the top wall. The number “5” for example, can be represented as 0101 in binary, meaning that it has a wall in its left side, an opening on bottom, a wall on its right side, and an opening on top. In this way, the numbers from the text file can be converted into a virtual maze as below.

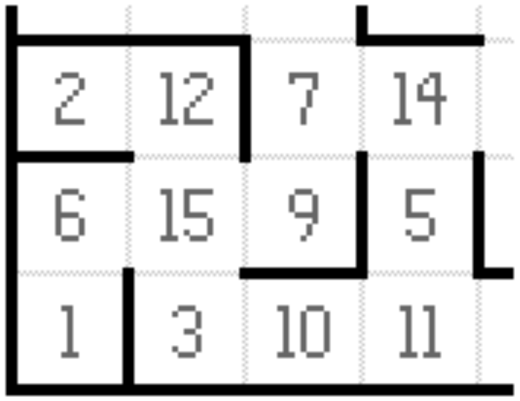


Figure 2: Mapping of .txt numbers to maze walls

Robot Specification

We are given a virtual robot equipped with sensors mounted on its right, left and front sides. The robot is always located at the center of a square facing one of the cardinal directions. Its sensors relay the number of open squares in each of its sensor directions. In the first position for example, the robot will sense a distance of 0 for both of its left and right sensors since there are wall on either side of it. It will then pick up a sensor reading of 1 or more for its front senor depending on the number of open spaces in front of it. At each time step, the robot can rotate either clockwise or counter clockwise by 90 degrees, or it can maintain its current heading. In the same time step, it has the option of then moving forwards or backwards by a distance of 1, 2 or 3 units but no more. If the robot hits a wall during the movement, it stops in its tracks. Once movement is complete, the time step is finished and the sensors return the new sensor information for the new location.

In more detail, at the beginning of each time step, the robot receives a list of three integers corresponding to the sensor distance for left, forwards and right in that order. The robot has a “next_move” function that receives this list and returns the robot’s rotation (-90,90 or 0 corresponding to counter clockwise, clockwise and no rotation respectively), followed by the robot’s motion (an integer between -3 and 3 where negative values indicate a backwards movement). The robot will then attempt to execute to rotation and movement provided by the "next_move" function.

Exploratory Visualization

In this section, we will visually explore the specifics of our first test maze. By running the included showmaze.py script with test_maze_01.txt as an argument, we are provided with the following visualization of the 12 x 12 maze. The goal (red) and start (green) locations have been highlighted for clarity.

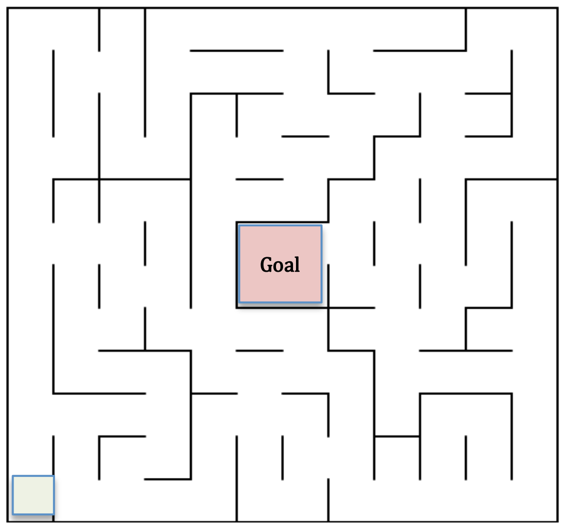
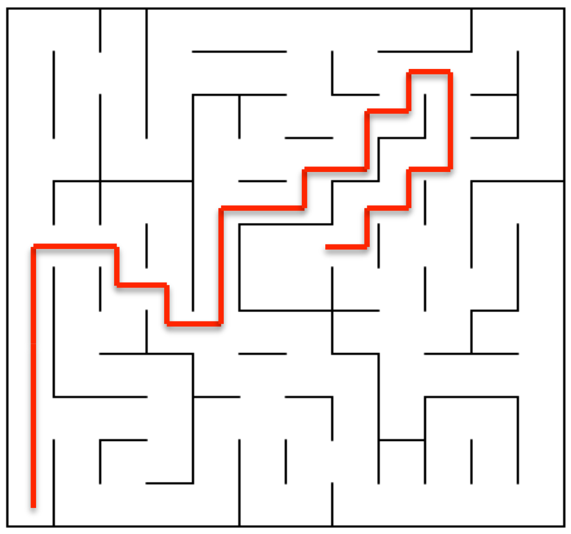


Figure 3: Maze 1 with start and goal locations highlighted

With the maze visualized, we can now look at possible routes to the goal in order to determine the optimal performance for our robot. For example, the following path gets the robot to the goal fairly quickly in a total of 21 time steps. One thing to notice is that this path has many turns. We can improve upon this path by selecting a path with fewer turns – meaning that the robot can travel over further distances during each time step.



In figure 5, a more efficient path is laid out which requires fewer turns and therefore fewer steps. This path can be completed in only 17 timesteps. Based on this analysis, we should program the robot to prefer longer moves. An interesting problem that comes up however is that, if we program in a preference for long moves, we don't necessarily get the shortest path each time. In the above examples for example, a preference for long moves with cause the robot to go straight at first, instead of turning the first corner despite the latter path being shorter. This can be fixed by applying a second variation of the flood fill algorithm after the first run and before the second which takes into account the ability to move up to 3 spaces. This has not been done in this project.

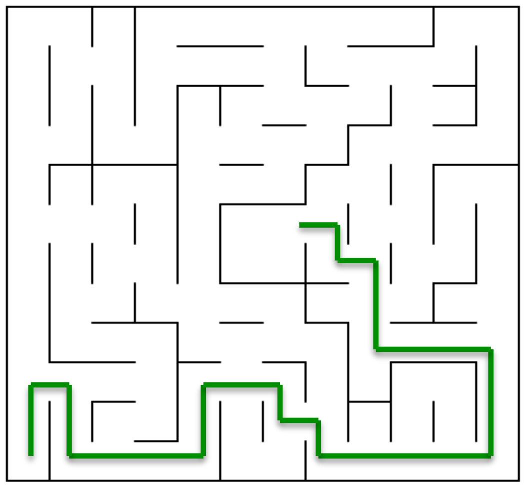


Figure 5: More efficient solution for Maze 1

The next thing to consider is whether or not the robot should be allowed to travel into unexplored territory during the second run. In figure 6 below, the robot has explored a large portion of the maze (represented in light green) before deciding to end the first run and move on the second run. If we allow the robot to move into unexplored territory in the second run, we can run into a problem. In the example below, the robot thinks that by moving out of the previously explored territory, in can find a shorter route to the goal, but because the robot has no knowledge of walls in the unexplored area. Because of this, it failed to find the dead-end and ran into it, effectively adding unnecessary time to the score. For this reason, a good strategy is to only allow the robot to move into explored space during its second run to prevent getting stuck in dead-ends and running into unexpected walls.

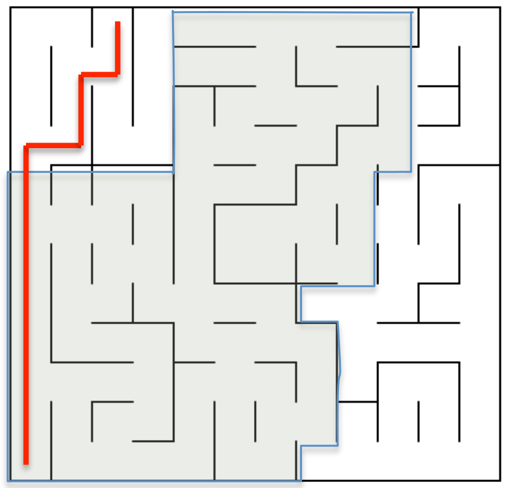


Figure 6: Running into a dead-end in unexplored space

A final important thing to consider is the interplay between exploration and completion. In figure 7 for example, the robot has explored a portion of the maze (represented in light green) before ending the exploration run. From the figure, we can see that although the robot has explored a significant portion of the maze, it has left the bottom right corner unexplored. If we are not allowing the robot to move into unexplored space in its second run, then it will not be able to move along the optimal path that was described above (17 time steps) and instead is forced to take a sub-optimal path of 20 time steps. This is the risk that one takes when deciding not to explore the entire maze - although the robot will save time during the exploration phase, it may lose time during the second run. We need to find a way to optimize both of the runs.

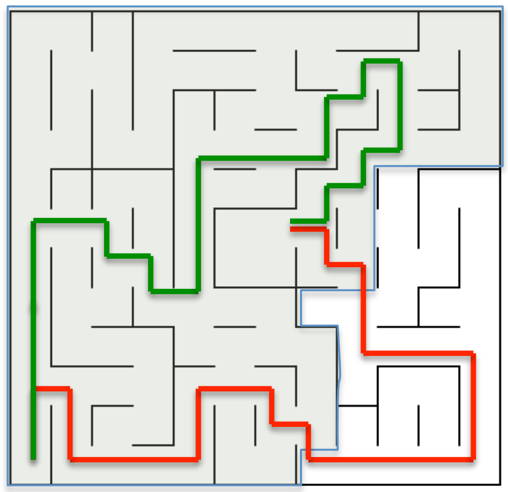


Figure 7: Dangers of leaving portions of the maze unexplored

In summary, based on this visual exploration of the first maze, we should look for algorithms that favour longer moves and fewer turns, that prevent the robot from entering uncharted territory during the second run, and that find the right balance of exploration time.

Algorithms and Techniques

The basic techniques that we will be employing to complete this project are as follows. First, our robot must keep track of three different things: knowledge of the walls in the maze, a record of which cells the robot has been to, and a record of which cells have had wall knowledge updated. This information is kept in 3 separate arrays called "wall_map", "visited" and "updated_the_walls" respectively. With these arrays initialized, the robot can embark on its first exploratory run.

In the exploratory run, the robot first must read its sensors to determine the location of walls in its surroundings. It does so by receiving a list of values from its sensors and then using this data to populate the "wall_map" and "updated_the_walls" array appropriately. Next, the robot must figure out the distance to goal of each square based on its current knowledge of the maze. To do this, we use a **flood fill algorithm** to fill in the distance to goal for each cell. This algorithm first creates an array with an entry of "99" corresponding to each maze cell. It then sets the goals cell to "0" and works it way outwards, replacing each adjacent cell with the same number plus 1 to indicate it requires one more step to move to the goal from that cell. It continues to do this until each cell has a number associated with it that indicates the required number of moves from that cell to the goal cell. The flood fill algorithm will take into account any walls that the robot knows about at that point.

After updating its knowledge of the maze and then implementing the flood fill algorithm, the robot then must decide on whether it should rotate or not and how many squares to move. It does so by looking at all of the options and deciding which of these options will get it closer to the goal based on its current knowledge of the distance to goal. In the first run, the robot will only ever move one space in order to ensure that it can sense as many of the appropriate walls as possible. Once the robot has reached its goal, it can either stop and move onto the next run or continue exploring, this will be explored further later. At the end of each move, the robot updates the "visited" array to indicate that we have visited that cell.

In the second run, we use an almost identical technique but for a few differences. We no longer need to perform flood fill at each step since we already have a mostly complete picture of the maze saved from the first run. Additionally, we allow the robot to move up to 3 spaces per turn as long as the move gets the robot closer to the goal by 3 spaces. In fact, we set the robot to prefer longer moves. Finally, we block off all of the cells we have not yet visited on the basis of the "visited" array.

In summary, the techniques for each run are as follows.

First Run:

1. Initialize arrays to keep track of walls and robot movement and distances to goal for each cell
2. Receive sensor data and update knowledge of walls
3. Perform flood fill and update knowledge of distance to goal for each cell

4. Analyze all possible rotations/moves to determine which one(s) will get robot closer to goal. Limit to moves of 1 cell.
5. Select a rotation/move pair and execute move. Record that we have visited the new cell.
6. Repeat from beginning until goal is reached.
7. Stop when goal is reached or continue exploring.
8. Block out all cells we have no information about.
9. Run Flood Fill one last time.

Second Run:

1. Receive sensor data and analyze all possible rotations/moves to determine which one(s) will get robot closer to goal.
2. Select a rotation/move pair and execute move with a preference for longer moves.
3. Execute the move.
4. Repeat 1-3 until goal is reached.

Benchmark

Determining the best possible score for each of the mazes is easily done by visual inspection. Maze 1 can be completed in 17 steps, Maze 2 in 23 steps and Maze 3 in 27 steps. By using the previously mentioned formula for calculating score, we can determine the minimum score for each maze which turns out to be 17.56, 23.76 and 27.9 for Mazes 1, 2 and 3 respectively.

Determining the worst possible score is also relatively straightforward. We can determine the longest route (utilizing single cell moves and assuming full knowledge of the maze) by visual inspection and then assume that the exploration phase takes up all of the remaining time. For Maze 1, the worst possible path takes 31 steps, Maze 2 takes 43 steps and Maze 3 takes 51 steps. Assuming for example that in the case of maze 1, we then using 1000-31 = 969 steps to explore, we would get a score of 63.3. Equivalently, we would get a score of 74.9 and 82.6 for Maze 2 and 3 respectively. It is important to note that this maximum score assumes full knowledge of the maze. In reality, we can envision even worse scenarios in which we do not achieve full knowledge of the maze during the exploration step – in which case our score could be even worse due to even more sub optimal paths being chosen in the second run. For now, we will keep these values as worst-case benchmark for algorithms that provide full maze knowledge prior to embarking on the second run.

In summary, we are aiming to get as close to the minimum score in each maze which is summarized in the table below.

Maze Number	Minimum Score	Maximum Score Estimate
1	17.56	63.3
2	23.76	74.9
3	27.9	82.6

Table 1: Benchmark scores for each maze

Methodology

Data Preprocessing

For this project, there is no requirement for data preprocessing as all data is used as is. Sensor data is noise free and maze data is given to us in a clean and clear format.

Implementation

This section will provide details on the implementation of the basic strategy seen in previous sections. As a reminder, the strategy for the first run is as follows.

First Run:

1. Initialize arrays to keep track of walls and robot movement and distances to goal for each cell
2. Receive sensor data and update knowledge of walls
3. Perform flood fill and update knowledge of distance to goal for each cell
4. Analyze all possible rotations/moves to determine which one(s) will get robot closer to goal. Limit to moves of 1 cell.
5. Select a rotation/move pair and execute move. Record that we have visited the new cell.
6. Repeat from beginning until goal is reached.
7. Stop when goal is reached or continue exploring.
8. Block out all cells we have no information about.
9. Run Flood Fill one last time.

1) To implement the first step, we simply use list comprehension to initialize 2D arrays with dimensions matching that of the maze as follows:
   a) Initialize the wall map. Start with no walls in each square or "15" in each array cell: *self.wall_map = [[15 for a in range(maze_dim)] for b in range(maze_dim)]*
   b) Initialize an array to keep track of which cells contain walls that have been updated: *self.updated_the_walls = [[0 for a in range(maze_dim)] for b in range(maze_dim)]*
   c) Initialize an array to keep track of which cells that the robot has visited: *self.visited = [[0 for a in range(maze_dim)] for b in range(maze_dim)]*
   d) Record that we have visited the initial cell by placing a "1" in the corresponding array entry: *self.visited[self.location[0]][self.location[1]] = 1*

2) Next, we take the sensor data and the known heading and use this knowledge to update the robots knowledge of the maze walls
   a) Save location of the grid cell in which a wall is being sensed in each direction
   b) Save the location of the grid cell next to the one we are sensing in the direction of sensing
   c) For each direction, we update the wall that we sense in the appropriate location .
      a. First convert the wall number to binary and check if there is a wall in the appropriate position already.
      b. If no wall exists already, adjust the wall number to add the appropriate wall.
      c. Keep track of the fact we have updated the cell by updating the "updated_the_walls" array.

   d. Check to see if the next cell in the direction of sensing actually exists and if it does, add the appropriate wall (opposite compared to above)
   e. Record that we have updated that cell

3) In the third step, we implement the **flood fill algorithm** as follows.
   a) Initialize the distance map. Start with all distances = 99 except for the goal location which we set to 0: *self.dist_to_g = [[99 for i in range(self.dimensions)] for j in range(self.dimensions)]*
   b) Initialize an array to keep track of which cells we have updated in this round of the flood fill algorithm and create a list containing all cells we have already updated: *updated = [[0 for a in range(self.dimensions)] for b in range(self.dimensions)]*
   c) Choose a cell from the list of updated cells, and look in each direction for a wall.
   d) If a wall does not exist in that direction and we have not already updated the adjacent cell in that direction, add the adjacent cell in that direction to the updated_cells list.
   e) Record the distance_to_goal for that cell as 1 + the distance to goal of the current cell.
   f) Record that we have updated the distance_to_goal of the new cell.

4) In the fourth step, we look through each direction to determine if moving that direction gets us closer to the goal.
   a) Create a list to keep track of potential moves: *move_list = []*
   b) Create a list to keep track of the distance to goal for the cells reached by the potential moves: *dist_list = []*
   c) Determine binary wall number of current location
   d) Go through each direction and if no wall exists in that direction, save the cell one away in that direction as a possible next move.
   e) Check that the possible move exists and if it does, save the location to the "move_list". Save the distance to goal of that cell by adding it to the "dist_list".
   f) Pick the move that yields the smallest distance to goal
   g) Determine rotation (-90,90 or 0) and movement (1 or -1) based on the chosen next cell and return these two values.

5) Execute the move (done in tester.py)

6) Repeat parts 1-5 until goal is reached or continue exploring.

7) Block out all cells that have not been visited

   a) For each cell in the maze, check to see if we have updated a wall in that cell or visited it
   b) If we have done neither, change the "wall_map" value for that cell to 15, effectively blocking it off
   c) Add corresponding walls to all of the adjacent cells

So after completing the first run, we are left with a "wall_map" array containing information about all of the walls that the robot knows about. Any cell that the robot has no information on is blocked out and we know the distance to goal from every cell in the maze. Next, the robot can embark on its second run.

Second Run:

5. Receive sensor data and analyze all possible rotations/moves to determine which one(s) will get robot closer to goal.
6. Select a rotation/move pair and execute move with a preference for longer moves.
7. Execute the move.
8. Repeat 1-3 until goal is reached.

1) Receive sensor data and analyze all possible rotations/moves to determine which one(s) will get robot closer to goal.
   a) Create a list to keep track of the potential cells to move to: *potential_move_list = []*
   b) Go through each sensor direction
   c) If the sensor in that direction indicates a distance of 3 or higher, save the cell 3 steps away in that direction.
   d) Check to see if that cell is 3 steps closer to the goal than the current cell. If it is, save it to the "*potential_move_list = []*" and move on to the next direction.
   e) If it is not*,* save the cell 2 steps away in that direction. Check to see if that cell is 2 steps closer to the goal than the current cell. If it is, save it to the "*potential_move_list = []*" and move on to the next direction*.*
   f) If it still is not*,* save the cell 1 steps away in that direction. Check to see if that cell is 1 steps closer to the goal than the current cell. If it is, save it to the "*potential_move_list = []*" and move on to the next direction*.*
   g) If the sensor in that direction indicates a distance of 2, repeat this process: first check the cell 2 away to see if that cell works, if not, check the cell 1 away. If the sensor indicates a distance of 1, one only needs to check the cell that is one move away. In this way, we first look at long moves before checking shorter and shorter moves.

2) Select a rotation/move pair and execute move with a preference for longer moves.
   a) From the "potential_move_list", select the move with the longest distance
   b) Determine the required rotation and return both the rotation (-90,90 or 0) and the movement values (-3 to 3).

3) Execute the move (done in tester.py)

4) Repeat 1-3 until goal is reached

- *Is it made clear how the algorithms and techniques were implemented with the given datasets or input data?*
- *Were there any complications with the original metrics or techniques that required changing prior to acquiring a solution?*
- *Was there any part of the coding process (e.g., writing complicated functions) that should be documented?*

Refinement

When the strategy defined above was first implemented, the exploratory run was as simple as it could be. Essentially, in the first run, the maze goal was defined as the exploration goal, and the robot was allowed to explore the maze until it got to the goal. This exploration strategy resulted in acceptable results which are documented in the table below.

Maze Number	**Score**
1	25.4
2	47.8
3	40.3

Table 2: Scores for implementation V1

This strategy (V1) was relatively successful, but as discussed above, means that the robot has limited knowledge of the maze when attempting the second run. This can lead to sub-optimal performance on the second run.

To improve upon the scores, we can implement "intermediate goals". For example, we can ask the robot to first reach the top left corner (A) and once it has reached that, reset the goal to the center of the maze (*) as before. This means that it must explore the maze to reach cell A before then changing course to reach the maze goal. Therefore, the second run will usually finish with improved knowledge of the maze compare to the first strategy (V1). Using this second strategy (V2), we can specify and number of intermediate goals for the robot to explore to. By experimenting then with various combinations of intermediate goals (top-left corner cell (A), top-right corner cell (B) and bottom-right corner cell (C)) we can control the amount of exploration that the robot does. Experiments were conducted with various different combinations of intermediate goals and the full results will be presented in the following section. The table below presents the best results for each of the 3 mazes.

Maze Number	**Intermediate Goals**	**Score**
1	C-B-*	22.0
2	B-C-*	38.2
3	C-*	29.9

Table 3: Best scores for implementation V2

The reader will notice that the intermediate goals for corresponding to the best score for each maze are not the same for each maze. The best over all intermediate goal combination (which lead to the lowest total score across the 3 mazes was the combination on C-B-* which resulted in the following scores.

Maze Number	**Intermediate Goals**	**Score**
1	C-B-*	22.0
2	C-B-*	38.9
3	C-B-*	35.3

Table 4: Scores for implementation V2 using the
best over all intermediate goals

Strategy V2 is implemented by changing step 6 in the first run as follows:

6) Reset Goal to reflect the next intermediate goal.

A final strategy was attempted (V3) that forced the robot to explore every single space in the maze before moving on to the next run. In this implementation, first a list of the cells in the maze is created. A series of intermediate goals is then set, starting with the cells closest to the start position. Each time a goal is reached, a new intermediate goal is set corresponding to the next cell in the list which was set up so that subsequent entries in the list are always one cell away from the previous one. Any cell that is visited in instantly removed from the goal list. This continues until every cell in the maze has been explored.

V3 produced the following results:

Maze Number	**Score**
1	34.0
2	42.0
3	53.2

Table 5: Scores for implementation V3

Over all, comparing the 3 strategies, it appears that strategy V2 is the most promising. Manipulating the intermediate goals leads to not only the lowest scores in each maze when tested with unique intermediate goals, but also to lowest total scores over the three maze when tested with intermediate goals in common.

In summary, the three implementations used are as follows:

V1 – In the exploratory run, search for the goal, then stop and begin second run
V2 – In the exploratory run, search for a series of intermediate goals, followed by the center goal, then stop and begin second run
V3 – In the exploratory run, search for every cell, then stop and begin second run

Results

Model Evaluation and Validation

Both models V1 and V2 have strengths. Model V1 has the lowest training time since, in the first run, it seeks to move directly to the goal and immediately moves on to the second run. While this strategy does save on training time, its downfall is that it does not always find the most optimal route to the goal because it does not learn about the entire maze. On the other hand, model V3 does learn about the entire maze and therefore, usually selects the most optimal path in the second run. For this reason, it can be considered to be the best model if training time is ignored. Its downfall however is that it spends too much time in the training phase.

The best performing model was found to be model V2, which consisted of controlling intermediate goals that the robot has to reach before moving on to the center goal. Model V3 is a good balance of the two other strategies in that it allows for sufficient training time to find a good solution to the maze but does not spend too much time exploring the entirety of the maze. Additionally, this model is the most versatile in that by altering the intermediate goals (number or order), one can exert even more control over the robot. My tuning the number and order and location of intermediate goals, we can optimize the final score well.

The best results using this model are good when compared to the benchmark and this model can be expected to perform well and generalize well to other mazes dues to its robustness and versatility. Much testing was performed with various combinations of intermediate goals in order to quantify the model's tuneability and robustness. The results of these tests are presented in the table below. An intermediate goals specification of C-B-* for example means the robot first goes to the bottom-right corner, then the top-right corner, then the center.

Intermediate Goals	A	A-*	B-*	C-*	A-B-*	B-A-*	A-C-*	C-A-*	B-C-*	C-B-*	A-B-C-*	A-C-B-*	B-A-C-*	B-C-A-*	C-A-B-*	C-B-A-*
Maze 1 Score	25.4	31.2	26.8	29.7	35.6	27.2	30.3	25.7	23.8	22.0	23.9	22.6	28.1	28.4	24.4	27.6
Maze 2 Score	47.8	42.5	45.1	51.9	52.1	44.9	43.3	56.4	38.2	38.9	43.4	38.5	43.9	39.2	43.4	46.6
Maze 3 Score	40.3	35.5	36.8	29.9	31.6	37.9	30.8	31.1	34.4	35.3	34.6	36.2	32.4	34.5	44.5	39.1
Total Score ->	113.5	109.2	108.6	111.4	119.2	110.0	104.4	113.2	96.4	96.1	101.9	97.2	104.4	102.0	112.3	113.4

Table 6: Scores for implementation V2 using different combinations of intermediate goals

Looking at these results, the reader can see that by tuning the order and number of intermediate goals, one can optimize the scores for each maze. Further optimization is possible since this study limited itself to using corner points as intermediate goals.

Justification

In this project, we implemented three different strategies for solving the maze. The following table compares the best results obtained from all three strategies to our previous benchmarks.

Maze Number	**Minimum Score**			**Maximum Score Estimate**	**V1 Score**	**V2 Score (best for each maze)**	**V2 Score (best overall)**	**V3 Score**
1	17.56			63.3	25.4	22.0 (C-B-*)	22.0 (C-B-*)	34.0
2	23.76			74.9	47.8	38.2 (B-C-*)	38.9 (C-B-*)	42.0
3	27.9	82.6	40.3	29.9 (C-*)	35.3 (C-B-*)		53.2	

Table 7: Scores for all implementations compared with Benchmark scores

The results confirm that all though all three methods give promising results, way below the maximum score estimate. Despite usually finding the best path, V3 appears to perform the worst overall, due to the excessive time it spends in the exploration phase. For most mazes, V1 performs slightly better than V3 but we can expected that for more complicated and bigger mazes, V1 would perform less well due to the fact that it explores very little of the maze before moving on to the second run. Model V2 consistently performs better than the other models and even approaches the minimum score in some cases. This model is the most versatile since we can quickly alter the intermediate goal points however, it requires some tuning in order to optimize the performance, which could be considered a downside if only one run is allowed.

Conclusion

Free-Form Visualization

In this section, we will look at how our V2 strategy can improve upon the V1 strategy using Maze 3. The figure below presents simulated results when applying strategy V1 (explore directly to goal and then stop) with the green square representing the goal, the blue square representing the start location, the line representing the chosen path in the second run, and the green shading representing the explored areas. The reader can see that the robot explores towards the goal and selects what is quite a long and sub-optimal path, due to having limited knowledge of the rest of the maze.

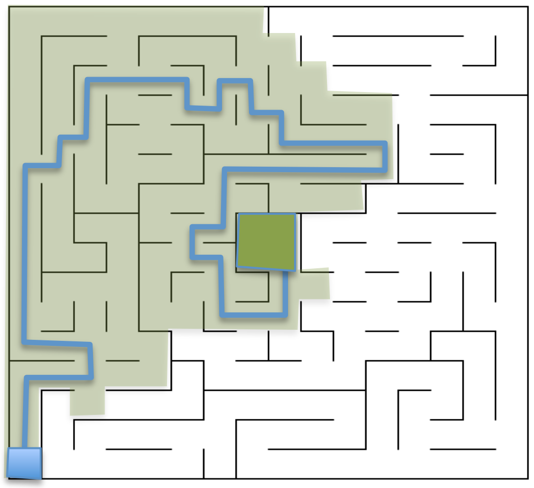
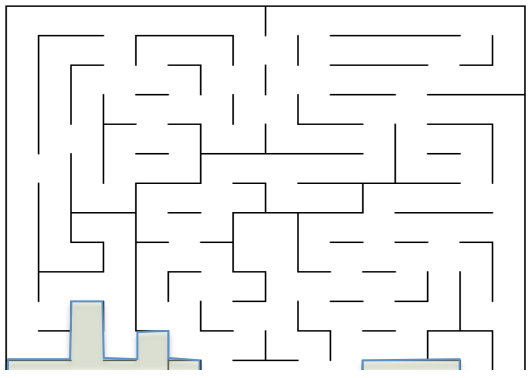


Figure 8: Visualization of path finding in Maze 3 using implementation V3

In the next figure, we can see the simulated results for the first exploratory phase for the implementation of V2. In this implementation, we specify the bottom left corner as an



intermediate goal, and allow the robot to explore to that goal.
Then, in the next phase and once the intermediate goal has been reached, the goal is reset to reflect the center of the maze and the robot is allowed to continue exploring. In this way, the robot can achieve more knowledge of the maze and actually learn about the most optimal path.

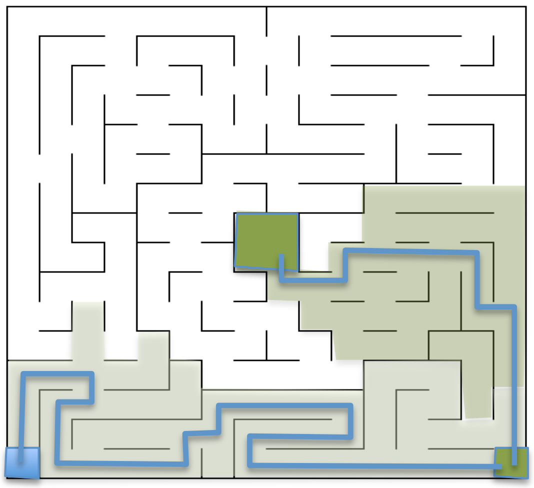


Figure 10: Visualisation of pathfinding from an intermediate goal using implementation V2

Reflection

In this section, you will summarize the entire end-to-end problem solution and discuss one or two particular aspects of the project you found interesting or difficult. You are expected to reflect on the project as a whole to show that you have a firm understanding of the entire process employed in your work. Questions to ask yourself when writing this section:

In this project, our goal was to develop an algorithm to allow a robot to navigate a maze as efficiently as possible. I started the project by doing some research into the micromouse concept and then proceeded to lay out and summarize the steps required for the robot to complete its tasks. These steps included high level tasks like: sense environment, decide on movement, make movement, record surroundings...etc. Once the high level tasks were laid out, I began working on the implementation including writing scripts for the flood fill algorithm, sensing algorithm, planning algorithms...etc.

After putting together the bones of the code, it was time to start experimenting on improving performance. I began by implementing V1 – the simplest code model which told the robot to navigate directly to the goal before switching to the second run. After noticing that this implementation does not always find the optimal path in the second run, I began experimenting with the idea of using intermediate goals to control exploration. These experiments lead me to the V2 model which had good results. This portion of the project was the most complex as it required lots of optimization and testing to really flesh out the performance of this system. Finally, I sought to understand what would happen if we asked the robot to explore the entire maze – this lead to model V3 which ended up having the worst results.

Over all, I believe this project was a success as all three of the models that I implemented had satisfactory results. The final solution actually achieved a very good score for all mazes. The major difficulties in the project revolved around weighing up the balance between the two run times but I think that the solutions presented dealt with this problem well.

Improvement

One aspect of the implementation that I would like to improve on is the flood fill algorithm for the second run. I think that if we implement a flood fill algorithm that fills each square with the minimum number of “moves” instead of cells to get to the goal, performance can be improved even further. In the second run of the current implementation, the robot decides which way to move based on the number of cells remaining in each direction to the goal. It also prefers long moves. This is problematic since, if two directions have the same distance to goal, the robot will always chose the direction that has the longer move. This direction is not necessarily the one that will get the robot to the goal quicker as it may have more shorter moves in later stages compared to the direction that it did not choose. I think that have the robot be controlled by the moves to goal instead of the distance to goal would benefit the scores.