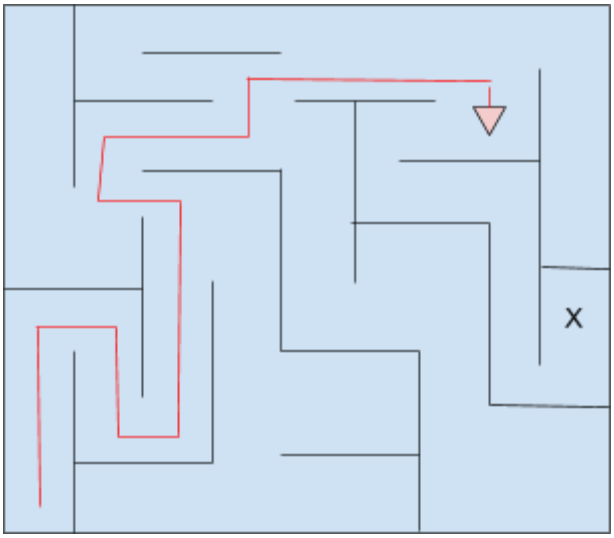
Plot and Navigate a Virtual Maze



Udacity

Jonathan Lee

Revision 1.1

2016.08.25

- Definition
  - Overview
  - Problem Statement
  - Metrics
- Analysis
  - Data Exploration
    - Robot sensor specification
    - Robot Movement Specification
    - Maze Specification
  - Exploratory Visualization
  - Algorithms and Techniques
  - Benchmark
- Methodology
  - Data Preprocessing
  - Implementation
    - Flood Fill Algorithm
    - Sense Wall Algorithm
    - Moving Algorithm for the first Run
    - Moving Algorithm for the Second Run
  - Refinement
    - Block Out Algorithm
- Results
  - Model Evaluation and Validation
  - Justification
- Conclusion
  - Free-Form Visualization
  - Reflection
  - Improvement

I. Definition

Overview

This project is based on the micromouse competition where each challenger submits a physical robot mouse which is designed to travel and learn the shortest path to the goal in a maze. The winner of the competition would be the mouse that reaches the goal in the shortest time. However, in this project there are less restrictions, such as, no physical mouse is required. Also, the sensors and movement are also 100% perfect. The ultimate goal for both the micromouse competition and this project is that a robot mouse needs to learn the maze and get to the goal in the fastest time possible.

Problem Statement

The goal of this project is to implement a robot in python which can, ideally, find the shortest path to a goal in a mazes of three different dimensions (12x12, 14x14, and 16x16). In each of these mazes, the goal is located in the center 2x2 grid. The robot is given a maximum of 1000 time steps to make two runs to the goal. The first run is the learning phase where the robot can explore the maze to find the goal and more, if needed. Then, in the second run, the robot is placed back at the initial starting point and should go to the goal in the shortest time possible.

Metrics

The goal is to seek to minimize the score. During the training phase, the robot is "awarded" a point for each 30 steps it takes to explore the maze. During the second phase, the robot is "awarded" a point for each 1 step it takes to get to the goal. See the equation below.

$$score(t_0, t_1) = \frac{t_0}{30} + t_1$$

t_0 = Total number of time steps from the first run (Training run)

t_1 = Total number of time steps from the second run

Therefore, the less time steps (t_0) the robot requires to train and the less steps (t_1) it takes for the robot to get to the goal will equate to a lower score.

II. Analysis

Data Exploration

The data which was provided includes the following, the robot specifications (sensors and movements) and maze specifications (wall descriptions and goal location).

Robot sensor specification

The robot will have three sensors mounted in three positions (left, front and right). These sensors will be able to detect the distance a wall is from the robot (see Figure 1).

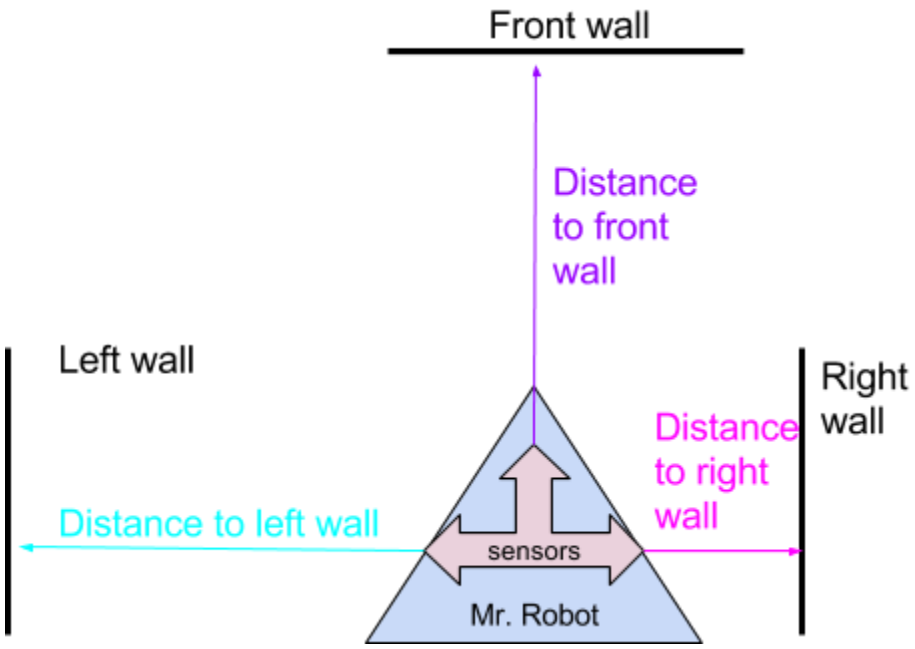


Figure 1: Illustration of sensor detection

For example, this case above, the sensors array may provide the following information:

sensors = [9,7,1]

sensors[0] = 9, means there is a wall 9 spaces to the left of the robot
sensors[1] = 7, means there is a wall 7 spaces in front of the robot
sensors[2] = 1, means there is a wall 1 spaces to the right of the robot

The interpretation for the sensors array can be summarized as follows:
["integer distance to left wall", "integer distance to front wall", "integer distance to right wall"]

The reader should also keep in mind that positions are relative to current robot's heading. A maze is defined with x and y location (0,0) at the bottom left hand corner. Coordinate axis are chosen such that, 'up' as the north of the maze, 'left' as the west of the maze, 'right' as the east of the maze and 'down' as the south of the maze (see Figure below).

Therefore, if the robot's heading is 'up', the sensors array will correspond to:
['distance to left wall', 'distance to up wall', 'distance to right wall']

What if the robot's heading is 'down'? The sensors array will now correspond to:
['distance to right wall', 'distance to down wall', 'distance to left wall']

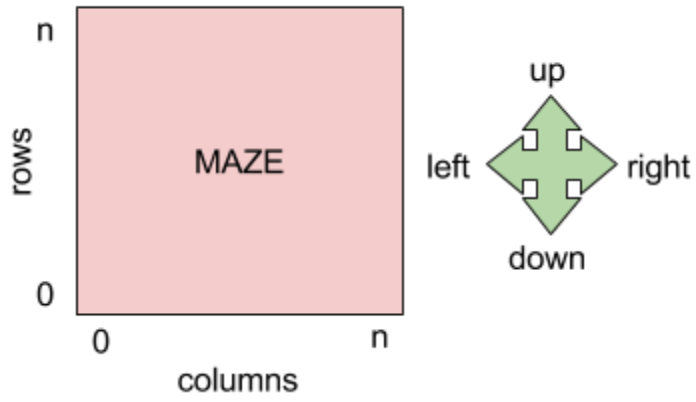


Figure 2 - Maze orientation and general heading definitions

Robot Movement Specification

Now that it is understood how the robot interprets the environment with its sensors. The focus now will be on how the robot can move about. The robot is equipped with exactly two types of motions, rotation and movement. At each time step, the robot is allowed to, first, make one rotation and a maximum of three steps should the environment allow the robot to do so. Also note that the robot is only allowed to rotate either exactly 90 degrees or -90 degrees.

From these two motions, four general movements can be defined (See Table 1).

Motion relative to heading	Rotation	Movement
left	-90	1,2, or 3
right	90	1,2, or 3
forward	0	1,2, or 3
backward	0	-1,-2, or -3

Table 1 - Robot motion Summary

Note that it is possible that the robot can have movement = 3 or 2 only if the situation allows for it such as there is no wall impeding the movement. Sorry, but it cannot do a robocop power charge through walls here.

Maze Specification

Focusing now on exploring one of the provided maze files, test_maze01.txt. It contains information such as 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
```

The first integer is the dimension of the maze. Here, 12 indicates that the maze_dim is 12x12. Also note here that to clearly picture the maze the reader must keep in mind that it is a rotated version of Figure 2. See Figure 3 to visualize what has just been described.

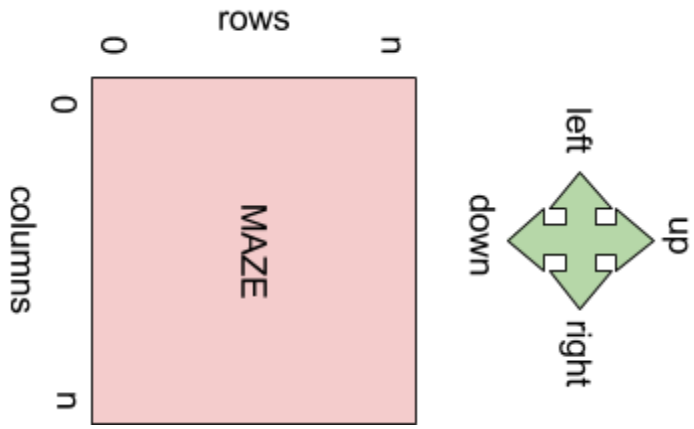


Figure 3: Rotated version of the maze orientation to fit the maze text file array

After the first line, the reader will observe all these seemingly random numbers with values between 1-15. Each of these decimal numbers in the maze can to be converted to its binary counterpart which describes where an opening is located. See Figure 4 for an illustration.

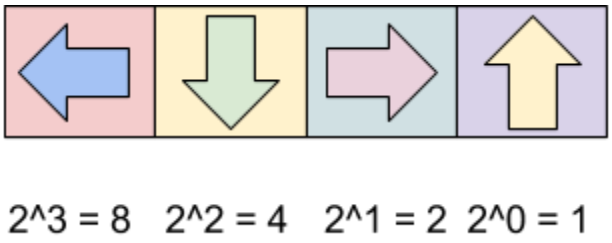


Figure 4: Directional Representation of Wall Openings

Therefore, the number 1 on top left corner or the maze means (0*8 + 0*4 + 0*2 + 1*1 = 1) there is an opening facing ‘up’. Then the number 5 next to the number 1 mean (0*8 + 1*4+0*2+1*1 = 5) that there is an opening at ‘up’ and ‘down’. To get a better idea of the logic, see the following figures 5a and 5b:

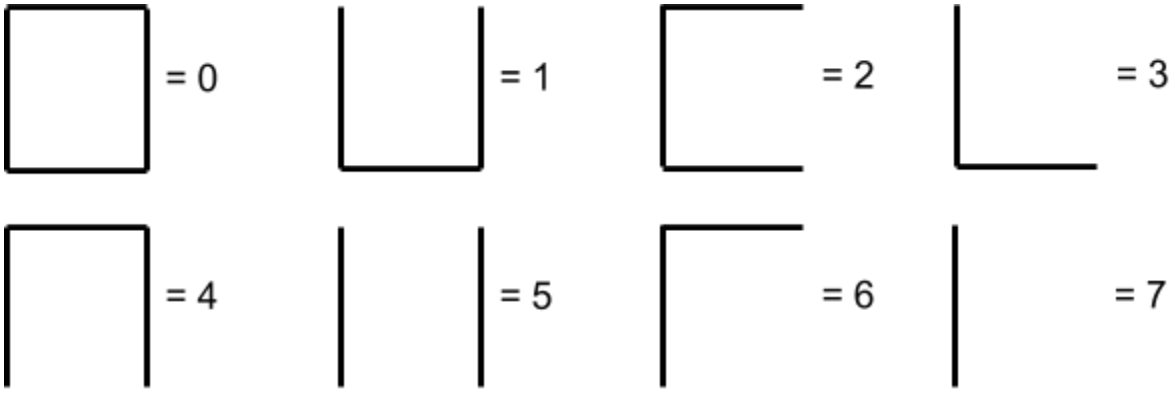


Figure 5a: Wall formations for decimal value 0-7

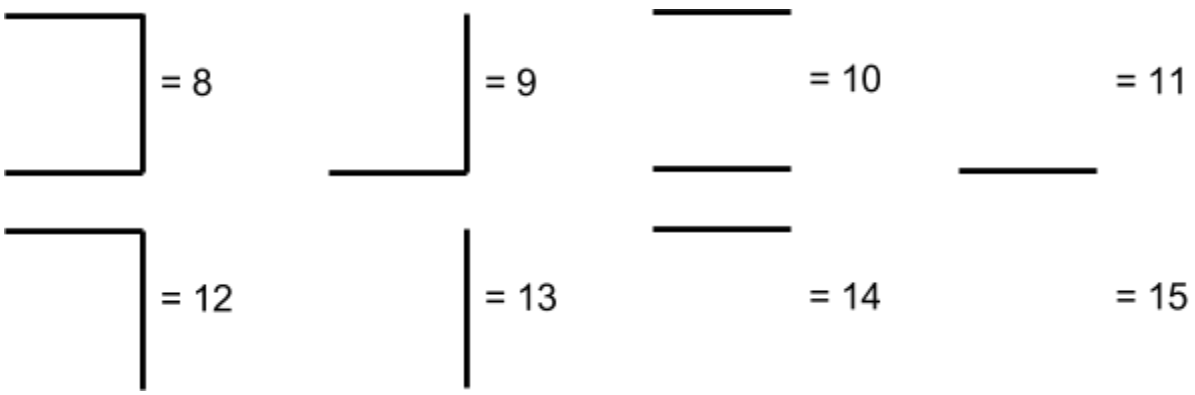


Figure 5b: Wall formations for decimal values 8-15

From Figure 4, one can easily deduce the patterns of each grid. For example, one quick deduction is that dead-end formations would be the number 1,2,4 or 8, since that would mean there is only one opening. Also, note that there are some grids that have 15 which means there are no walls. This one fact later caused some issues with implementation which will be touched upon later on.

One possible solution to the goal would be the following:

0. [0,0] #initial state
1. [0, 3]
2. [0, 6]
3. [2, 6]
4. [2, 5]
5. [3, 5]
6. [3, 4]
7. [4, 4]
8. [4, 7]
9. [4, 8]
10. [7, 8]
11. [7, 9]
12. [8, 9]
13. [8, 10]
14. [9, 10]
15. [9, 8]
16. [8, 8]
17. [8, 7]
18. [7, 7]
19. [7, 6]
20. [5, 6] #goal state

This is a total of 20 time steps. However, this is not the actual fastest time step because this route has too many turns and less straight edges. The actual optimum path would actually take 17 timesteps instead of 20 timesteps. Note that, it is also estimated that the shortest path for the test_maze2 is 23, and test_maze3 is 27. These numbers will later be used in the benchmark and summarized in the Table 2 below.

Maze	Approx min timestep
test_maze1	17
test_maze2	23
test_maze3	27

Table 2: Calculated minimum timestep per maze

Exploratory Visualization

The focus of this visualization will be based on testmaze1.txt using the showmaze.py script provided. All the mazes can be visualized in the same methodology but for simplicity only test_maze1 will be shown.

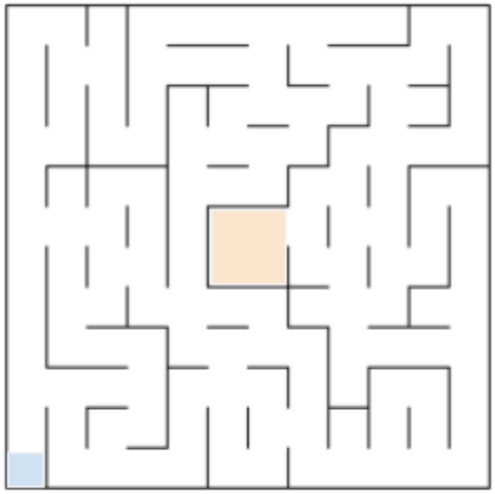


Figure 6: This is test_maze01 after running the showmaze.py
Note that the starting point and goal locations have been added after running the script.

The blue square at the bottom left-hand corner of the maze is the starting location (See Figure 2 to refresh memory regarding maze orientation). The orange square in the center of the maze is the goal zone.

Referring back to the previous section on the test_maze01.txt, it was observed that the first square displayed 1 (indicates an opening on ‘up’) and then the square after it displayed a 5 (indicated an opening on ‘up’ and ‘down’). In Figure 6, this consistency is observed. The reader will also be able to find that the dead-end locations are there as well and that the squares with number 15’s represent openings which have no walls.

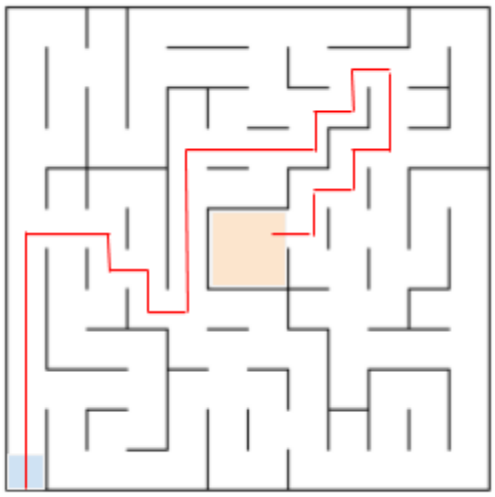


Figure 7: Maze path to the goal which took 20 timesteps

In Figure 7, the path that took 20 timesteps is drawn for illustrative purposes. One can see that this path has a lot of turns, thus, possibly slowing down the robot since it is rarely able to do a maximum of three steps within one timestep. The following Figure displays an alternate, faster route because it has less turns.

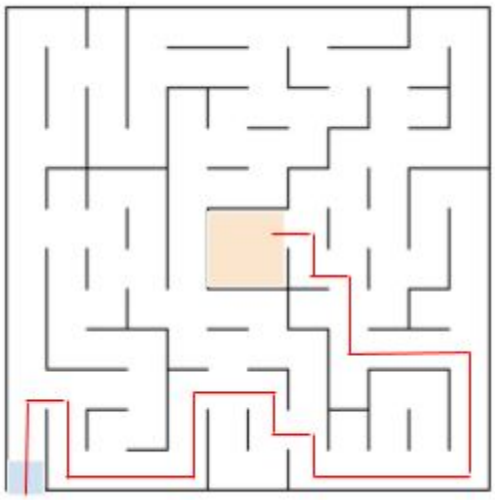
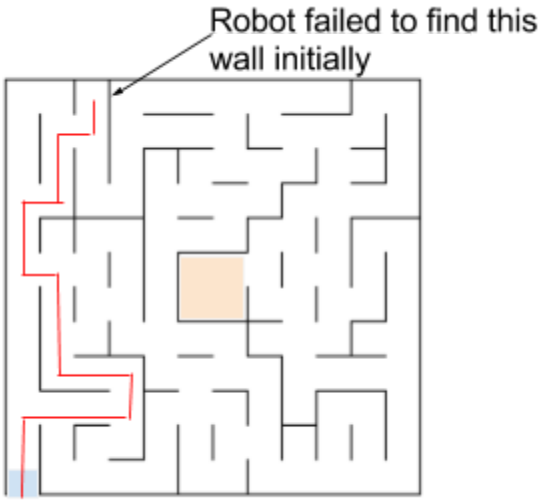
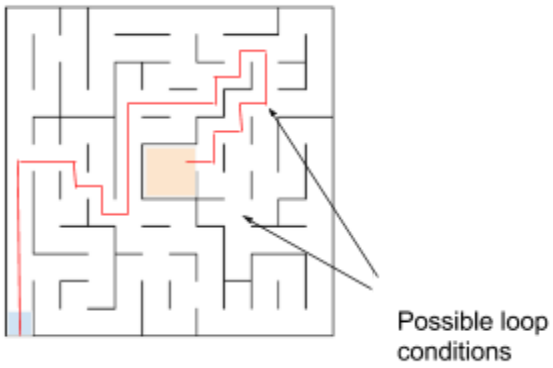


Figure 8: Maze path to goal with 17 timesteps

When or if the map is inadequately mapped out by the robot, then it can get stuck in some dead-end locations. This actually occurred to the robot in my initial implementation when exploration and blocking untraveled spaces were not used.



Above is an example of one case where my robot got stuck in the dead-end. This only occurred when the robot failed to find that wall specified by the arrow. Therefore, it thought it can go to that location and turn right. But it couldn't do so since there was a wall there. However, this problem is solved by traveling the robot around to learn the whole map and/or blocking out the untraveled paths. Therefore, it is believed that this is not an issue anymore.



Also, one loop condition that occurred at one of those 15 blocks. This is because there is an array called walls_updated, which is initialized to all zeroes. And this array would be updated each time a wall was found. So, these squares are never updated because there are no walls to update so it always remains zero. This issue came to play when having the robot explore the whole map after finding the goal. An array was created which had all the points where the walls_updated == 0. And the robot would go and check each of those location and update the walls. The loop would end when the whole walls_updated file be all ones. And of course this would never happen because the robot would be stuck in one of those 15 points. So, a modification was made where if the robot was in the same space at at least two different rotations, then mark the square was wall updated. This fixed this one particular loop. Also note that the final implementation with the block out technique only looks at the spaces the robot

actually as moved_to and does not care about walls_updated. Therefore, this also should not be an issue anymore.

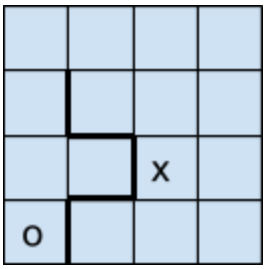
Algorithms and Techniques

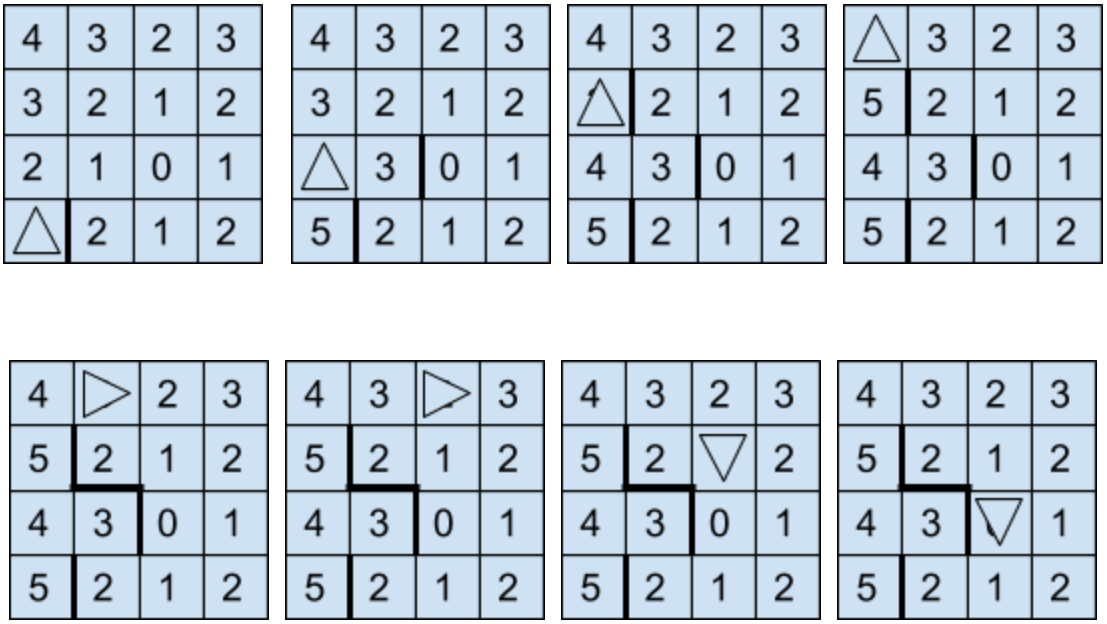
Now, the reader should have the basic knowledge on how the robot can sense the environment and move around. The reader should also have the basic knowledge on how the maze is constructed and what should be the minimum timestep to the goal. The focus now should be on what strategies can be implemented so that the robot can travel and learn the maze and travel straight to the goal.

Given the metrics as previously stated, where the goal is basically seek to minimize the score. Therefore, it makes sense to find a way to obtain not only the shortest path to the goal but also a method that will train the robot quickly.

There are two main techniques used in this project, mapping and pathfinding. Mapping is the process of creating a map of the environment as the robot travels about. Pathfinding is the process of finding the shortest path given a particular map. The algorithm chosen for pathfinding is the flood-fill algorithm. The flood-fill algorithm will be touched upon in more detail in the implementation section.

The basic question given the current map status, what is the shortest path to the goal? The robot will automatically choose the move which gets it closer to the goal. But as the robot makes this move, the sensors will detect a change in the map and it will update accordingly. But once the map has been updated accordingly the path may change as soon as it's provided with new information. The robot will still move in the the direction closer to the goal. Let's give a simple illustration. Assume the actual maze looks like the following where o is the starting location and x is the goal location. At each time step, the robot first senses environment and then run flood fill algorithm to obtain updated distances and move where distances decreases.





Figures 9: Example of Robot finding the goal Using flood fill and mapping

The reader should observe that initially the robot did not know where all the walls are. But as it traveled toward the goal, it was able to update the wall information and the distances still directed it toward the goal. If there were no walls, the robot would have simply walked to the goal in 3 moves. But because of the walls, the robot actually traveled 7 steps.

Note also that in this case the robot will have the option to explore the environment more or block off the untraveled paths because one can see that if the robot was placed back at the starting point it will erroneously believe that there is only 5 steps to the goal because it happened to now discover that final wall. So one solution to find the wall is to have the robot travel to it's unmarked places.

1	1	1	0
1	0	1	0
1	0	1	0
1	0	0	0

Figure 10: Example of moved_to array

The above grid marks where the robot has traveled so, a list can be created after the robot reaches the goal to append all the positions marked with a zero and have the robot travel to those locations. Then after all spots are traveled then the robot should find the following distance grid.

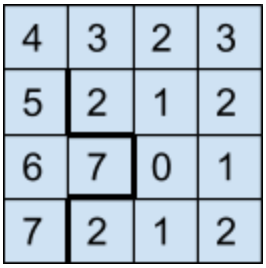


Figure 11: Distance array after map has been recovered

Here clearly states that it will take the robot 7 steps as before to get to the goal. The alternate methodology is to block off all the spaces not traveled by the robot. This was an improvement made to minimize training time. But note that in some mazes this may not generate the lowest score. For example, in test_maze_02.txt, the "block off" method did not generate the lowest score but was at least better than the average of max and min!

Figure 12: Distance array using block out technique

Note that in this case, all the spaces that the robot didn't walk to was simply just closed off completely and the only path the robot sees is the one that it has walked. So, in this case the robot would have found the same best path but with less training time!

As the reader may have already observed to make use of this strategy, the robot needs to keep a tally of two informations: walls for each grid and distances to the goal. For example, using test_maze01, the robot initializes two 2D arrays. One array, which contains all the current wall information in the whole grid, is initialized to all 15 (Recall that 15 is binary 1111 which stands for no walls). Another array, which provides the current known distance from each square to the goal, is initialized to 99.

After the first time step (Sense1 + FloodFill1), the distance_to_goal array:

[11, 10, 9, 8, 7, 6, 5, 6, 7, 8, 9, 10]
[10, 9, 8, 7, 6, 5, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 4, 5, 6, 7, 8]
[8, 7, 6, 5, 4, 3, 2, 3, 4, 5, 6, 7]
[7, 6, 5, 4, 3, 2, 1, 2, 3, 4, 5, 6]

[6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5]
[7, 6, 5, 4, 3, 2, 1, 2, 3, 4, 5, 6]
[8, 7, 6, 5, 4, 3, 2, 3, 4, 5, 6, 7]
[9, 8, 7, 6, 5, 4, 3, 4, 5, 6, 7, 8]
[10, 9, 8, 7, 6, 5, 4, 5, 6, 7, 8, 9]
[11, 10, 9, 8, 7, 6, 5, 6, 7, 8, 9, 10]
[12, 11, 10, 9, 8, 7, 6, 7, 8, 9, 10, 11]

The walls array after the first time step (Sense1 + FloodFill1):

[5, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 14]
[7, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
[15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15]

The reader should notice that the robot has found a three walls from its initial position. One wall straight ahead to the maze dimension limitation and one wall to the right of it. And one wall to the left of it.

Here is a summary of all that has been mentioned thus far.

1) Initialize all walls to 15
2) Initialize all distances to 99
3) While the goal is not found
    a) Sense the walls (This will update the wall file)
    b) Find the distance to go for each square on the grid (This is running flood fill algo)
    c) As long as there is no wall impeding, move the robot in the direction toward the goal (the distance from the goal should be decreasing)
4) After goal is found there are two choices
    a) Can explore the spots that robot didn't explore
    b) Can block out all the spots the robot didn't explore

5) Use the final distance map to move the robot directly to the goal

In the context of having the most robust long-term solution it is better to explore all the spots the robot didn't explore. But in the context of this project or a more short-term solution, to obtain the lower score it may be more beneficial to simply block out all the spots the robot didn't explore.

Here is a good analogy, a pizza delivery doesn't have a map but needs to get a pizza to the customer as soon as possible. A scout is sent out first to find the location of the pizza delivery. He only knows the general location but not the actual route. As soon as he founds the actual route, would he want to waste time driving around to parts of the city to see if there is a shorter path? Or shall he instantly call back home and tell them to come quickly? If they are in a time crunch, then it would probably make sense to tell the other person to delivery the pizza quickly! Because who knows how long it will take to travel the full route? But on the other hand, if this will be a frequent customer and the pizza delivery shop will have to deliver to this same location multiple times, then it would make sense to sacrifice some initial time to find the shortest path. But of course, in this project context, the goal is more geared toward short term rewards.

Benchmark

To determine the benchmark it makes sense to start by defining the best case scenario for the lowest score ever. For the first map, the lowest score would be if the robot walked straight to the goal in the first run and learned the path and then traveled the same exact path.

The minimum time step required for each maze was estimated by inspection and is about 17 for maze1, 23 for maze2 and 27 for maze3.

Therefore the minimum score shall be as follows:

Maze1_min = 17/30 + 17 = 17.56
Maze1_min = 23/30 + 23 = 23.76
Maze3_min = 27/30 + 27 = 27.9

For maximum let's assume the robot takes up all the extra training time to train and move at the slowest possible to the goal.

Maze1 takes 31 single steps to goal
Maze1_max = (1000-31)/30 + 31 = 63.3
Maze2 takes 43 single steps to goal
Maze2_max = (1000-43)/30 + 43 = 74.9
Maze3 takes 51 single steps to goal
Maze3_max = (1000-51)/30 + 51 = 82.6

Maze	Min Score estimate	Max Score estimate
1	17.56	63.3
2	23.76	74.9
3	27.9	82.6

Table 3: Benchmark minimum and maximum scores

The goal of the project will be to obtain a result as close to the minimum estimate as possible. And the final score should fall between these two numbers. Obviously, the robot has failed if the score is above the max. And it is impossible to get below the minimum score unless the robot gained superman abilities and broke through the walls.

III. Methodology

Data Preprocessing

There is no data preprocessing required since the sensors specifications and environments were already provided. In an actual case where data preprocessing is needed would be a situation where data must be gathered from a noisy analog sensor data and convert it to a nice readable digital format to work with. In this process, repetitive data may be removed or missing information may be added. In this case, all data is used as is. Or for the case of the maze, it could be that a person takes a picture of a maze. And converts that to the format which was already provided. There may be some irrelevant data removed and missing data corrected into the best guess. Again, the maze information is used as is.

Implementation

The meat of this project lies within this section. As a quick reminder, restating the goal of this project, the robot must find the goal twice within 1000 total steps. A lower score is given to the robot which trains the fastest and finds the shortest path. Reiterating the basic strategy is:

1. Assume there are no walls
2. Run flood-fill algorithm to find the distance to the goal from any square in the maze
3. Sense where are the walls and update the walls array
4. Run flood-fill algorithm at the point where the goal is to update the distance_to_goal array
5. Move robot toward the square of decreasing distance (without any wall impedance)
6. Repeat steps 3 - 5 until goal is reached
7. When goal is reached there are two options:

a. Explore the rest of the map to see how does the map really look and then find the shortest path after that
b. Block off all the untravel spaces and take the current path to goal as gold

Implementing first step is simple, the code utilizes list comprehension to initialize a 2D array: self.walls = [[15 for i in range(maze_dim)] for j in range(maze_dim)]

Note that there are multiple arrays which also need to be initialized using the same list comprehension method shown above. The reader will find all information in the code.

Flood Fill Algorithm

The flood fill algorithm implementation is as follows:

1) Initialize another list called traveled (this will keep track which square has be flooded so they won't be flooded more than once):
    a) traveled = [[0 for i in range(self.maze_dim)] for j in range(self.maze_dim)]
2) Set the goal location as traveled and distance_to_goal to one:
    a) self.distance_to_goal[xlocation][ylocation] = 0 (see Figure below as an example to follow along)
    b) traveled[xlocation][ylocation] = 1

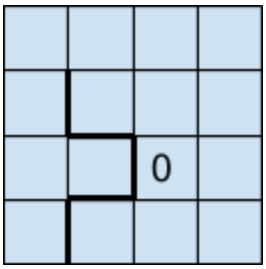


Figure 13: simple example showing starting point with o and endpoint with x

3) Initialize list of locations for the squares that need to have their distance score updated and to be the next location (note that it is initialized with the current position):
    a) next_list = [[xlocation,ylocation]]
4) Remove the current location from the list and append the next possible locations and add 1 to the distance.
    a) Check all four directions of the current square
        i) If the location, that is intended to be moved upon has no wall and has not already been traveled to.
            (1) Append this location to the "next_list" (see Figure below)
            (2) give it a distance that is one point higher than the current location (In figure below, distance of current location is 0 therefore each of the distance of neighboring square is now 0+1 = 1)
            (3) Also mark these location as traveled. (Done in another array not shown)

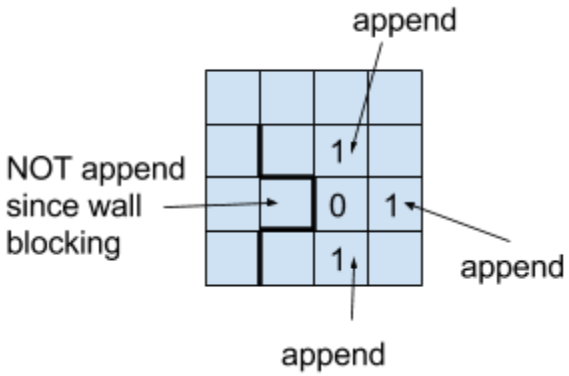


Figure 14: Example of appending neighboring locations to list

5) Loop step four until the traveled array has all ones

There were generally no issues with this flood fill implementation and it did not need to be modified through multiple runs.

Sense Wall Algorithm

The basic sense wall implementation is implemented as a function which takes two inputs, the direction (robot's heading) and sensor information.

The sense wall algorithm uses some core equations (with minor changes) which can be used regardless of robot orientation. In order to do the minor tweak, a set of generic parameters have been predetermined depending on which heading is used. This minimizes the lines of code and creates less clutter. This methodology was chosen because the sense_wall was becoming way to complicated with repetitive code where it was mostly just sign changes, etc.

Here is the list of these parameters:

Parameter	Description	'left'	'down'	'right'	'up'
xparam1	Modifies the left_sq_loc x-parameter	0	1	0	-1
xparam2	Modifies the Front_sq_loc x-parameter	-1	0	1	0
xparam3	Modifies the right_sq_loc x-parameter	0	-1	0	1
yparam1	Modifies the left_sq_loc y-parameter	-1	0	1	0
yparam2	Modifies the front_sq_loc y-parameter	0	-1	0	1

yparam3	Modifies the right_sq_loc y-parameter	1	0	-1	0
opposite	Opposite direction of current orientation	‘right’	‘up’	‘left’	‘down’
add_param1	Modifies the left_add_x, Front_add_y, Right_add_x parameter	0	-1	0	1
add_param2	Modifies the Left_add_y, Front_add_x, Right_add_y parameter	1	0	-1	0

Table 4: sense_wall function parameters

The reasoning behind these parameters is that the difference between heading = ‘up’ and heading = ‘down’ are sign changes in both x and y direction. For example, when facing ‘up’ and moving ‘up’ the direction is in positive. But when facing ‘down’, and moving ‘up’, the direction is negative. Also when facing ‘up’ and moving ‘left’, the direction is negative. But when facing ‘down’ and moving ‘left’, the direction is positive.

The same reasoning occurs for when the robot is facing left or right, it is only the sign change. But the difference between up and left, require zeroing and adding different parameters. For example, it is because moving forward or backward when facing ‘left’ does not affect the y direction and moving forward or backward in the y direction while facing ‘up’ does not affect the x direction. Therefore, certain parameters need to be zeroed out when moving left versus moving up.

Moving on to how the robot can detect walls, as an illustration, it is assumed the robot is facing forward. The robot will first check the value within the wall array at the number of sensed spaces away from left, front and right. Then it will convert those numbers into binary format. Recall that before the robot learns the maze all squares in the maze have value 15 or 1111. If the reader is having trouble picturing these binary to decimal conversion, a table has been created below.

		‘left’	‘down’	‘right’	‘up’
Wall Situation	Decimal	3	2	1	0
All Closed	0	0	0	0	0
Up is open	1	0	0	0	1
Right is open	2	0	0	1	0
Right and up is open	3	0	0	1	1
Down is open	4	0	1	0	0

Down and up are open	5	0	1	0	1
Down and right are open	6	0	1	1	0
Left wall closed	7	0	1	1	1
Left wall open	8	1	0	0	0
Left and up are open	9	1	0	0	1
Left and right are open	10	1	0	1	0
Down is closed	11	1	0	1	1
Left and down are open	12	1	1	0	0
Right is closed	13	1	1	0	1
Up is closed	14	1	1	1	0
All open	15	1	1	1	1

The strategy now is that each time the robot detects a wall, it will subtract the the corresponding value from the initial 15. For example, if the robot senses a wall to the left, then check to see if bit3 is 1 or 0. If it is a 1 (which it is in this case since the robot hasn't learned the environment yet), then we will subtract 2^3 from the wall number which makes it 7 or 0111. If the wall has already been updated, then the robot would find a 0 when it checks the bit and there is no need to subtract anything.

Here is the code implementation of the above. Prior to updating the wall information for the square, the robot will first need to determine which square to look at based on the sensor distance from the current position:

left_sq_loc = self.walls[self.location[0]+xparam1*sense[0]][self.location[1]+yparam1*sense[0]]

Here xparam1 and yparam1, as specified, is defined differently depending on the robot orientation (See the table above).

So one can see that we seek to take the current robot x location and subtract from it what sensor on the left sees. So, if the robot was at [3,4] facing up and the sensors see [1,0,3]. Then we know there is a wall in the square [2,4].

Next we also need to check the wall adjacent to the left_sq_loc for walls by using the following:
left_add_x = self.location[0]+xparam1*sense[0]-add_param1
left_add_y = self.location[1]+yparam1*sense[0]-add_param2

These left_add_x and left_add_y define the x and y location to the left of the left_sq_loc because the wall needs to be modified here also.

Note that also a check_limit(self,param1) function is defined such that:

```
if int(param1) >=0 and int(param1) < self.maze_dim:
        return True
else:
        return False
```

In multiple situations when looking at the next possible location, the location coordinates needs to be checked to see if it's within the dimensions of the maze. So, this simple function was created to avoid writing the blahblah >=0 and blahblah<self.maze_dim multiple times.

In summary, if the robot sees a wall, then it check the map file to see if there is a wall there. If the map file doesn't show a wall it means it hasn't been updated. If not updated, then the robot will update that bit to a 0 (meaning there is a wall). Then if the square on the other side of the wall is not out of bounds, it is also updated (see Figure 15).

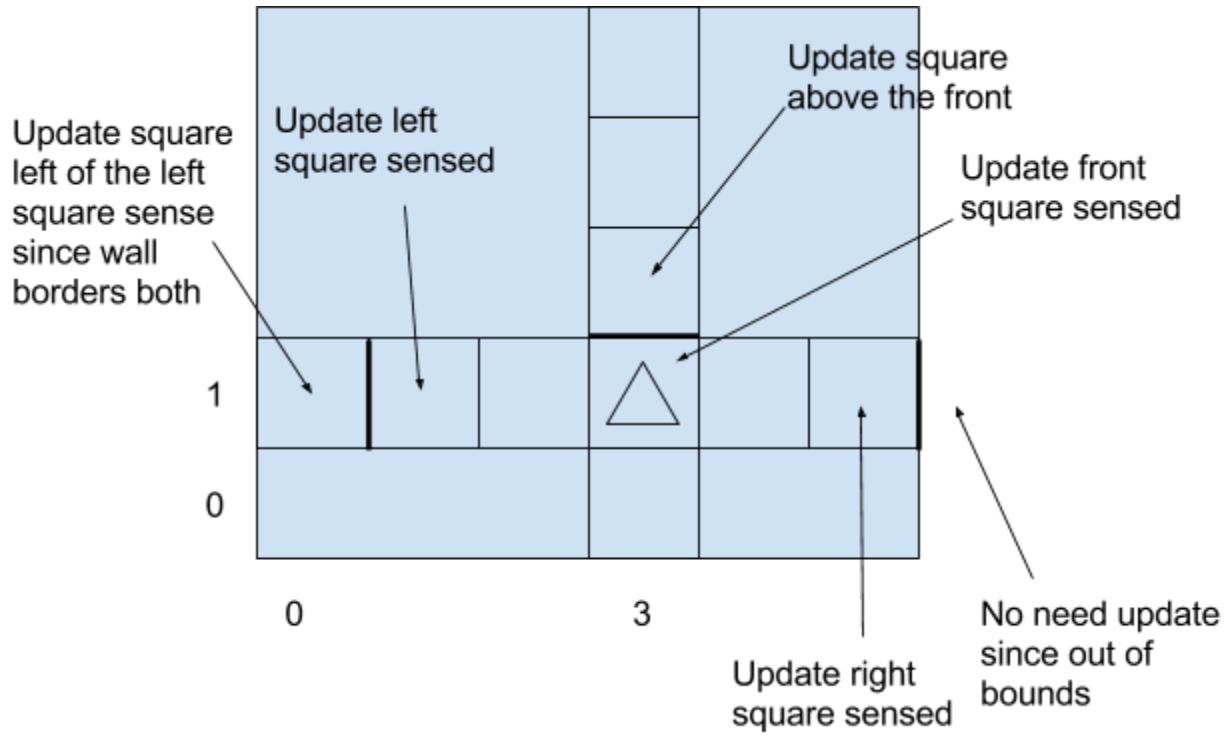


Figure 15: Wall sensing illustration

Sense wall function was implemented in a few iterations:

1) Initially the wall detection could only detect only walls current square. This however proved unable to work when doing the flood fill algorithm since it didn't make sense that the current square contained a wall but the neighboring square didn't see that wall
2) In the next update, the wall detection STILL only detected walls in one space. But it was updated to detect the walls in the current square and also the wall at the adjacent squares. For example, if the robot found a wall to the east of the current square the robot is standing then a wall was updated to the west of the east adjacent square.
3) Final update included detecting walls at the provided distance from the sensors and adjacent squares to those squares. This is the algorithm which was described above. This made more efficient use of the sensors.

Moving Algorithm for the first Run

Now that we have updated the wall information and flood-filled the robot needs to know how to move. How the robot will move will firstly depend on what direction the robot is facing. For example, if the robot was facing 'up', to go 'up' it simple needs to move without rotation. So, the strategy is:

1) Check the robot orientation
2) Then check the wall map file to see which sides do not have falls
3) For the particular sides that do not have a wall and are within the maze dimensions, append them to a move_list
4) At the same time append the current distance of that move to a dist_list
5) Then choose argmin of the items appended to the dist_list and use that index to choose the move with the least distance
6) This move will be chosen_move. Then we determine what is the deltax and deltay from the chosen_move to the original location
7) Based on this deltax and deltay robot can determine whether it simply moves or rotates and moves

This move function had to be modified to have the robot not move backwards. Since there was an issue where the robot might try to move backwards because there was decrease in distance to the goal but there was actually a wall that it didn't see because it does not have any sensors behind it. So, instead of having the robot move backwards when the ideal move is backwards, the robot simply rotates in that situation so that it can sense if there is a wall or not.

So, as the robot travels around and learns the map it will eventually find the way to the goal. As soon as the robot finds the goal. Using the exploration method, the robot will check which square in the map has not been explored and personally travel to each of those squares to complete the wall map.

As mentioned before, there are two strategies that could be employed after robot finds the goal. The first strategy will be discussed here as the 2nd strategy is more of a refinement.

Explore the whole map strategy:

1. When main goal is found
2. Take an array of the points which the robot has not traveled to
3. Sort the array such as we check the farthest most points first
4. Set each point which has not been traveled to as the “new goal”
5. Repeat steps 1-4 until each square of the maze has been traveled

When this is complete the robot was able to find all the walls which is virtually identical to the provided wall file. Then the robot employs the flood-fill algorithm one final time to obtain the final distances. Then it will reset itself and go straight to goal using the updated distances file. But how does it use the distance file to make the final run? This will be covered in the next section.

Moving Algorithm for the Second Run

The final move sequence was based on the Moving Algorithm for the First run where the robot would move one step at a time. However, it is a dumbed-down version of the move sequence. The robot not look needs to detect or sense anything in the environment since everything is all found. It simply only needs to move in the direction of the decreasing distance.

To ensure that it does it’s job correctly, it only needs to check of the next move location’s distance to goal plus the number of steps taken is equivalent to the current location (that is the location prior to making the move).

The move sequence was designed in two iterations:

- Initially the move function was designed to take only one step in the direction of decreasing distance
- Then the move function was modified to take more than one step and up to maximum of three steps, if possible

There was a little bit of challenge into designing the logic upon choosing to move 2 or 3 steps. It is not always the case that moving 3 steps is the best, it could be that if the robot moves 3 steps it might move into a square of increasing distance. Therefore, as mentioned above, a condition needs to be checked such that the new_move + number of steps equals the initial move (see simple illustration).

```
if self.distance_to_goal[the_chosen_one[0]][the_chosen_one[1]] + 1*the_chosen_one[2] ==
self.distance_to_goal[robot_loc[0]][robot_loc[1]]
```

The_chosen_one[0] is the x location of the next move
The_chosen_one[1] is the y location of the next move
The_chosen_one[2] is the number of steps to get to the designated location
Robotloc[0] is the initial x location
Robotloc[1] is the initial y location

See the illustration below to see how this works.

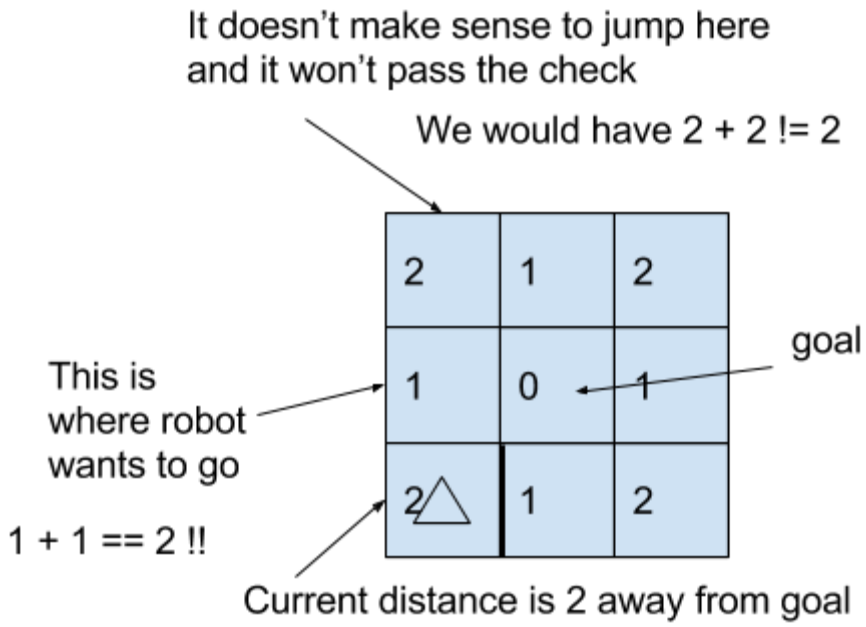


Figure 16: Final motion algorithm

In summary, the key for moving the final robot toward the goal is to move in decreasing distances such that the next location distance + number of steps = current location.

Refinement

In the refinement section, we present a huge upgrades to this robot motion. In the previous implementation after the robot has found the goal. We allow the robot to travel around to learn the whole maze so that it can be sure that it has found the shortest path. The reader shall soon see that this search to learn the full map will make the training period much, much longer and therefore increasing the score.

This is why a new strategy was implemented. When the robot has initial found the path, then take that path as the final path and block out all the other paths.

Block Out Algorithm

In the “block out” algorithm, it utilizes a an array which tracks which places the robot has moved upon.

1) Initialize an array called moved_to to all zeros
2) During the first run the robot will travel to find the goal. For every spot the robot moves to it is marked as a 1 in the moved_to array.
3) When the goal is found, any space that the robot has not traveled to is marked with a 0
4) Loop through all the locations marked with a 0 and set the self.walls array of those particular locations to zero
5) And set the neighboring walls of the neighboring locations of to zero

Let Method1 = “Exploration of maze after goal”
Let Method2 = “End after goal and block out unexplored territory solution”

Maze	Method1	Method2
test_maze_01	33.733	25.4
test_maze_02	43.367	48
test_maze_03	52.4	40.767

In general the Method2 with the blocking solution performs better!

IV. Results

Model Evaluation and Validation

Both robot model passes all three test mazes with a good margin. The first model is more robust in the long term as it recovers the full map and finds the shortest path in terms of distance. One possible improvement could be that it records all the possible paths and counts how many time steps it would take prior to taking the path. If the goal of this project was to find the shortest path distance regardless of training time, then the first model would be best. However, where this model fails is in terms of the training time.

In this project’s context, the second model performs better in general. The second model requires much, much less training time and therefore has a much lower score than the first. This makes the second model more desirable in this context. This will be justified in the following section when comparing the results against the benchmark.

Justification

Maze	Min estimate	Max estimate	Method 1 (actual)	Method2 (actual)
1	17.56	63.3	33.733	25.4
2	23.76	74.9	43.367	48
3	27.9	82.6	52.4	40.767

In each case, the score for both method1 and method2 are within the maximum spec. But also the difference between method1/method2 measurements are closer to the minimum than to the maximum. For example, here are the midpoints of each maze:

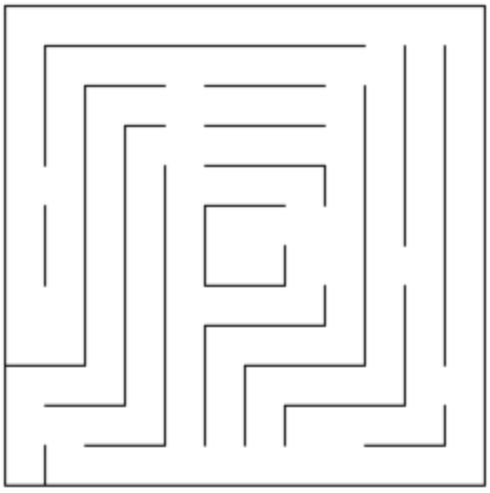
Maze1_mid = 40.43
Maze2_mid = 49.33
Maze3_mid = 55.25

Assume that the mid point is neutral point. It could so be taken as the average of the two extremes. Therefore since the actual score for both methods all lie closer to the min, it can be claimed that the results shown perform better than average! The Method2 has better results in 66% of the mazes, so if I had to choose, i would pick the one with the higher chance of having a higher score!

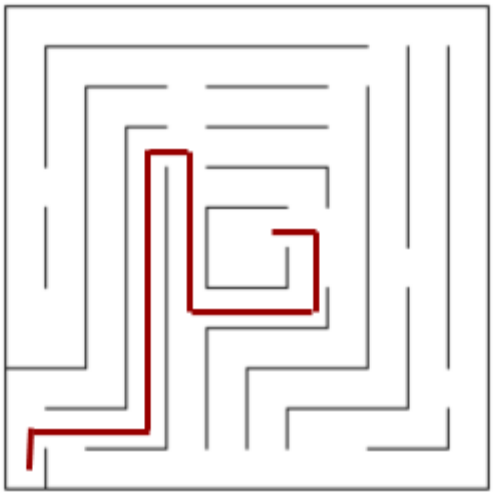
V. Conclusion

Free-Form Visualization

Created a maze which multiple paths in parallel, to see if the robot would get confused as to which parallel path to choose. But in fact, it was too easy for the robot to solve using both methods.



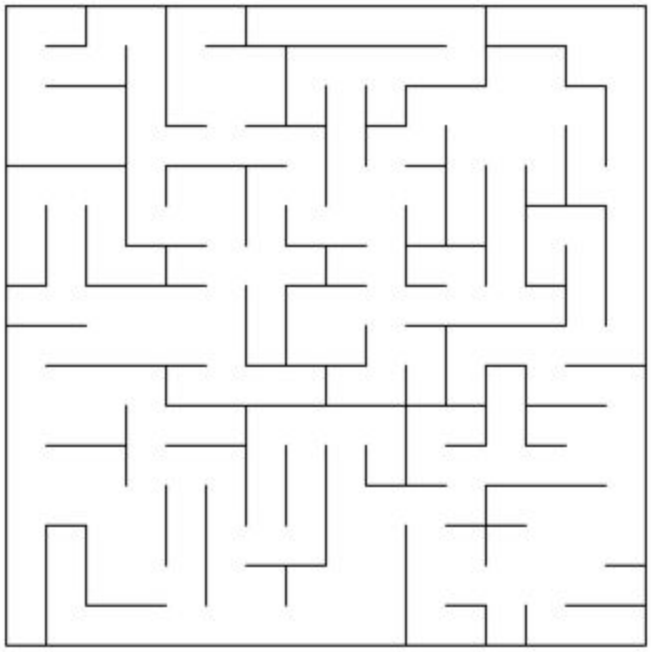
Test_maze_04



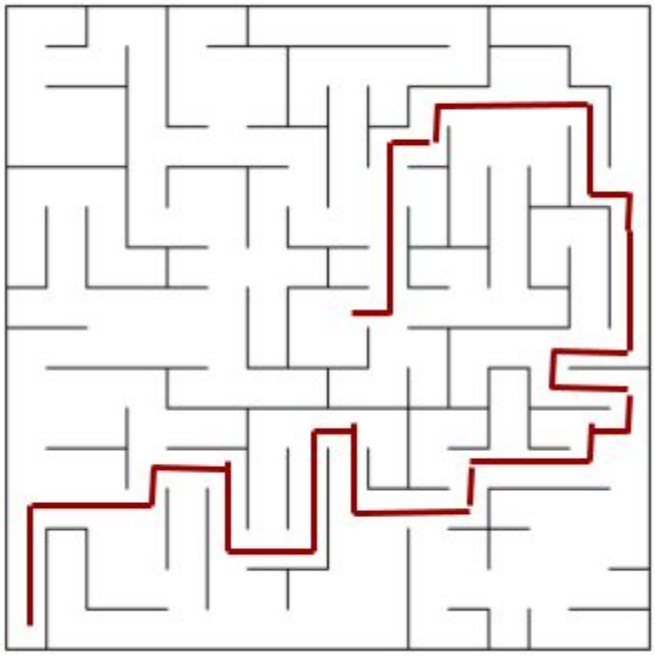
Here is the path taken by the preferred method, method2. In the algorithm, when the robot is presented with two locations of the same distances, then it chooses the left-most square most. This is the reason why it doesn't choose to turn right after the first right. This could be another item that can be improved upon in future work.

Since the above maze seemed too easy. A 16x16 maze was constructed with the goal to purposely make the path very long. The algorithm where the robot explores extra space actually cannot pass the maze. But the algorithm where any untraveled space is blocked off is able to pass the 16x16 maze with a long path. This shows that with a large maze, it is very

difficult to pass in a short amount of time if the robot is to explore the whole maze. This makes sense, the larger the area, the more time steps it will take to travel to each square.



test_maze_05



The long path traveled by the robot in Method2 is shown in the figure above. Method1 fails because after it finds the goal it takes too long traveling around to the different locations in the maze to complete the map. And it takes too long because there is more surface area to cover

and perhaps this particular maze arrangement makes it more difficult for the robot to explore efficiently. If I increase the maximum timestep of the method2 to 2000. It actually is able to find the goal after 1260 timesteps with a score of 55.533. So, it is not failing because of any error, it is simply just walking around too much.

Maze	Method1	Method2
test_maze_04	18.8	11.8
test_maze_05	55.533 (in 1260 timesteps)	32.333

Again, Method2 performs much better! And it seems all the fingers are pointing at Method2 as the best solution!

Reflection

The end-to-end process for completion of this project began with just simply reading the requirements on capstone page, over and over. To help me understand it more, I outlined the different sections either on trello or workflowy just to get a big picture flow as to what is required to complete the project. Then, just went step by step, through each of the sections to make sure I understood the requirements or specifications of the maze and robot. Then came some countless days of just brainstorming and doing some research on maze solving algorithms (see the links added in References section). It was from this research that led me to choose the flood-fill algorithm to solve this problem.

The first function that was implemented was the flood-fill algorithm. First confirmed that it would work in general in the maze without any walls. Then the next step was to figure out how to get the wall information. So, first developed a more simplified version of the sense_wall function which would only sense a wall when the sensor displayed a 0 to either left, front, or right of the robot. And got this to update the map with flood-fill and actually direct the bot to the goal.

Then came the issue, on how the robot would use that updated wall and distance information to travel to the goal. And the robot would seem to get lost or stuck initially because the walls weren't updated fully. Then, the implementation had to be revised to make the robot randomly travel about after it reaches the goal. This implementation would work only for the small maze but continue to fail for the medium and large maze.

So, then instead of having the robot randomly travel about the idea came to have the robot more systematically travel the untraveled spots since that might minimize the step count. This idea would still fail based on the big of the 15 (no wall locations). Because initially, the map would be marked updated when there was a change in the wall formation but because these 15 spots don't have a wall, then it would be as if it was never updated! Therefore the code had to be modified in the sense where if the robot was stuck in one space more than once, I would just

mark the space as wall_updated. After this bug was fixed then the maze was solved for all mazes.

Then came to the challenge on how to benchmark this performance. I concluded that if the robot were to somehow randomly travel straight to the goal and use that path, that would be the most minimum of all time steps. Then I thought to myself that should be what I am doing. I should just take the initial path the robot found as gold and forget all this explore the whole map idea. So, then I believe I remember from one of the articles something about blocking out untraveled spaces.

So, then the blocking technique was implemented and it gave me much more confidence that this was a good result. So, then the focus went into writing this report and explaining all the techniques and algorithms. In the end, it finally brings me down to this reflection.

After testing out my two implementations with some test mazes that I created and seeing that Method2 (block out method) is performing better it is definitely leading me to choose that methodology over doing extra exploration. Especially seeing that Method1 (exploring after finding the goal to get the full map) fails on my large 16x16 maze, it has made me lose confidence in that technique at least for the context of this project. I definitely want to conclude that I choose Method2 as my goto robot maze solver!

Overall, this was a fun and challenging project. It required a lot of print statements in almost every block of code just to check to see of each little section is doing what it's supposed to do. The most challenging and informative portion would actually be creating this report document. Creating this report is in a sense a way to pretend teaching robot motion planning to an individual who has no clue about the topic. There was also some effort for me to create the example diagrams and visual tables. The idea is that people are usually more visual and it helps to be able to visualize everything. Since it has taken much more thought on how to verbalize the thoughts in my head in a clear and concise manner.

Improvement

First and foremost, there could be some slight improvements to the discrete implementation. It is noted that the robot actually does not truly pick the shortest path. It could include another algorithm that calculates all the paths before choosing them. This could be similar to what is seen in google maps. The car hasn't driven there but one could see which route is shortest and choose that route.

In this design, the focus was on a discrete time domain with thin walls and the robot being pretty much a point source. In continuous time domain, it would definitely require path smoothing in terms of only the movement control. But all other techniques should remain the same. The only part that needs modification is how to control the robot properly using smoothing control and possible PID to avoid the walls.

VI. References

http://www.redblobgames.com/pathfinding/a-star/introduction.html
http://www.laurentluce.com/posts/solving-mazes-using-python-simple-recursivity-and-a-search/
http://www.societyofrobots.com/member_tutorials/book/export/html/94
http://web.cecs.pdx.edu/~edam/Reports/2001/DWillardson.pdf
https://mmalsubaie.files.wordpress.com/2013/08/mohamed-alsubaie-final-report-final.pdf
http://www.alexhadik.com/micromau5/