Robot Motion Planning Capstone Project

Plot and Navigate a Virtual Maze

Nidhi Nayak
Udacity Machine Learning Nanodegree
December 1th, 2017

TABLE OF CONTENTS

I. DEFINITION

Project Overview

This project is motivated by Micromouse competitions that began in the 1970s. A micromouse competition is an event where a small robotic mouse attempts to navigate an unfamiliar grid maze. The micromouse starts in a corner and must find its way to a designated goal area commonly located near the center. The robot is allocated two attempts with the maze - the first attempt is an exploratory attempt where the robot will review and plan out its environment. The second attempt is where the robot will attempt to traverse an optimal route to the goal area as fast as possible. Using Python 3, our micromouse will attempt to navigate a virtual maze and determine the fastest times possible to traverse an optimal route to the goal area located in the center in a series of test mazes.

Problem Statement

The objective is simple - get to the middle of the maze as fast as possible. The robot will start in the bottom-left corner of the maze and must navigate from the origin to the goal in the shortest possible time, and in as few moves as possible. The maze exists on an n x n grid of squares, n even. The minimum value of n is twelve, the maximum sixteen. The whole boundary of the maze is surrounded by walls preventing the micromouse from traveling outside the maze. The initial position will have walls on the left, right, and back sides, so the first move will always be forward. In the center of the grid is the goal room consisting of a 2 x 2 square; the robot must make it here from its starting square in order to register a successful run of the maze.

The micromouse will route a first trial that is exploratory in class where its main aim is to build information about the structure and shape of the maze, that includes all possible routes to the goal area. The micromouse need to travel the goal area in order to

complete a successful exploration trial, but is permissible to continue its exploration after reaching the goal. In the second trial, the micromouse will recollect the information from the first trial and will try to reach the goal area in an optimal route. The micromouse's performance will be recorded by adding the following:

- Total number of steps in the exploration trial divided by 30
- Total number of steps to reach the goal in the optimization (second) trial

Each of the trial is capped at 1000 steps. The micromouse can only make 90 degree turns (clockwise or counterclockwise) and can move up to 3 spaces forward or backwards in a single movement.

Project Structure and How to Run

Starter code for this project was provided by Udacity but I have added functionality to see realtime robot movement over maze and have added algorithms in a plugin pattern to robot movement logic . Information about Project files are as follows:-

- **robot.py**- This establishes the Robot class, but is the main script where modifications were made to the project.
- **tester.py**- This script is run to test the robot's ability to navigate the mazes.
- **maze.py**- This script is used to construct each maze and interacts with then robot whenever it is moving or checking its sensors.
- **showmaze.py**- udacity provided this script to view a visual layout of each maze, I modified this file & added functionality for better data exploration & visualization. To show maze & robot position in real time during exploration & testing phase. I have used python turtle animation library coordinate system in rendered maze window has origin in left most bottom positive x axis lies toward left to right, positive y axis lies from bottom to top.
- **test_maze_##.txt**- These files provide three sample mazes upon which to test the robot.
- **Pathoptimization.py:** this file contains the logic to optimize path. If path contains 3 step/2 step in a straight line they will be converted into 3 step/2step robot jump.

As this is to include the fact that robot can take -3 to +3 steps, path optimization logic is only applied over single step path.

Drawback of this approach is during test phase we find the shortest path among all single step path & then applied path optimization over it due to which shortest may not be the fastest path if robot can take 3 step/2 step jump & not just 1 step jump.

- **Robot.py:** I have modified this udacity provided python class which controls the robot movement & rotation, I have added few object variables such as Exploreaftergoal: it controls whether robot still be exploring unvisited shells after it reach goal during training phase.

  Algo object: it calls algo.py file which contains different algorithms for maze exploration.
- **Algo.py:** it creates an object which provides different algorithm for maze exploration to robot object.

Whole project is ported to python3, you need to install missing libraries such python graphics library turtle. Also install any transitive dependence library to run this project in your computer.

[shell]$ python3 tester.py test_maze_01.txt

edit robot.py

line 19 :- self.exploreAfterGoalReached= False # for maze exploration full/partial

line 23 to 25 :- to change maze algorithm

self.algoObj= AlgoPackage('floodfill' ,self.location, self.heading,self.goal_bounds, self.maze_dim ,self.exploreAfterGoalReached)

Metrics

As per the previous sections, the main evaluation metric to quantify the performance of the benchmark and solution models is:

score = [Number of Steps in Run 2] + [Number of Steps in Run 1 / 30]

This evaluation metric is impacted by both the exploration run and optimization run, however the optimization run will impact the score significantly more than the exploration run. Our goal is to minimize this score, which would be the result of an optimal run.

For example: let's assume the micromouse took 400 steps during its exploration run, and 16 steps during its optimization run. The score would be equal to: 16(steps from Run 2) + 400/30(steps from Run 1 divided by 30) = 29. If the optimization algorithm found the optimal route for this particular maze, then 29 would be the optimal score.

II. ANALYSIS

Data Exploration

Each maze is sketched on a square grid containing of either 12, 14, or 16 rows and columns. The maze is surrounded by walls neighboring its perimeter that act as a barrier, blocking all movement.
Our robot leads in the bottom left-hand corner of the maze (coordinate (0,0)) facing in an upward direction. It has walls on its left, right, and bottom sides with an opening on its top side, forcing its first move to be forward (or 'Up'). The robot aim is to navigate to a 2-by-2 goal area located in the center of the maze. A successful run of the maze is not complete without the robot getting to the goal at least once.

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

```
12
1,5,7,5,5,5,7,5,7,5,5,6
3,5,14,3,7,5,15,4,9,5,7,12
11,6,10,10,9,7,13,6,3,5,13,4
10,9,13,12,3,13,5,12,9,5,7,6
9,5,6,3,15,5,5,7,7,4,10,10
3,5,15,14,10,3,6,10,11,6,10,10
9,7,12,11,12,9,14,9,14,11,13,14
3,13,5,12,2,3,13,6,9,14,3,14
11,4,1,7,15,13,7,13,6,9,14,10
11,5,6,10,9,7,13,5,15,7,14,8
11,5,12,10,2,9,5,6,10,8,9,6
9,5,5,13,13,5,5,12,9,5,5,12
```

Figure 1: Contents of *test_maze_01.txt*

The first line of the text file is a number (12, 14, or 16) which defines the length of one side of the square maze. The subsequent lines contain comma-delimited numbers ranging from 1 to 15. These numbers define the position of walls and openings for each cell in the maze grid. Each number represents a four-bit number that has a bit value of 0 if an edge is closed (walled) and 1 if an edge is open (no wall); the 1s register relates with the upwards-facing side, the 2s register the right side, the 4s register the bottom side, and the 8s register the left side. For example, the number 7 translates to a square that has walls on its left and right sides and openings on its bottom and top sides (1*1 + 1*2 + 1*4 + 0*8 = 7). Because of array indexing, the text file defines the maze column-by-column from left to right, so the first number actually represents the starting cell in the bottom-left corner. Maze cell labels follow a Cartesian coordinate system with the rows and columns ranging from 0 to 11.

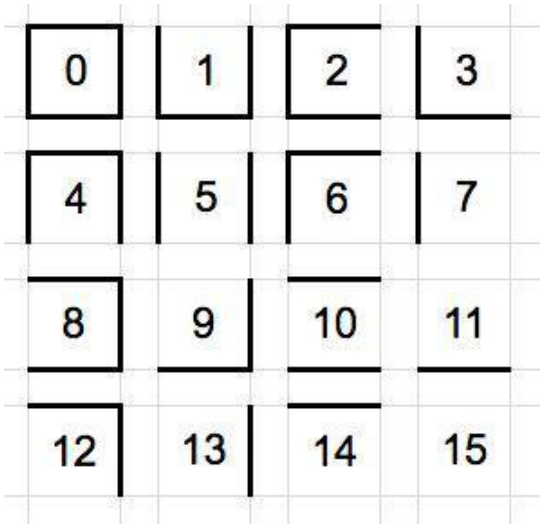


Figure 2: Wall Description Numbers

11	6	12	4	6	10	10	14	14	10	8	6	12
10	5	7	13	7	10	10	13	3	14	14	9	5
9	5	5	5	5	4	6	11	14	9	7	8	5
8	7	9	3	9	7	11	14	9	6	15	10	9
7	5	4	6	12	7	10	9	6	13	5	6	12
6	7	15	13	5	5	6	14	13	7	13	5	5
5	5	5	7	13	5	3	9	3	13	7	9	5
4	5	7	9	3	15	10	12	2	15	9	2	13
3	5	3	10	12	3	14	11	12	7	10	10	13
2	7	14	10	13	6	15	12	5	1	6	12	5
1	5	5	6	9	5	5	7	13	4	5	5	5
0	1	3	11	10	9	3	9	3	11	11	11	9
	0	1	2	3	4	5	6	7	8	9	10	11

Figure 3: Test Maze 1 with Wall Description Numbers

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

Note: Relevant python function in maze.py is is_permissible() and *init*(), is_permissible() checks whether or not a cell is passable in the given direction. Cell is input as a list. Directions may be input as single letter 'u', 'r', 'd', 'l'. init() reads the maze as text file and build a maze object based on above explained logic.

Exploratory Visualization

Data visualization is an important part in machine learning, I have used showmaze.py to view the mazes. Just viewing the mazes is not sufficient, while developing & debugging algorithm so I added functionality using python turtle animation class which rendered maze & robot in real time at every step to make this rendering real time took significant development effort in visualizing the mazes as well as robot.

This visualization window will open along with command line execution of tester.py class. Using this visualization, we can clearly understand various algorithm such as flood fill, depth first search & breadth first search.

This visualization helps me in debugging issues such as robot stuck in dead end, crossing over walls, cases such as going outside & handling edges cases such as robot going outside the maze boundary, debugging all above issues will be very difficult without this visualization window. It also explained distance matrix of each cell in the maze from the starting position.

Distance matrix is printed in command line at every robot step. training & testing path is shown in this visualization window in different colors to clearly understood when robot has completed training phase & what steps it took for testing phase.

These screenshots for testing & training is shown below for each of the mazes.

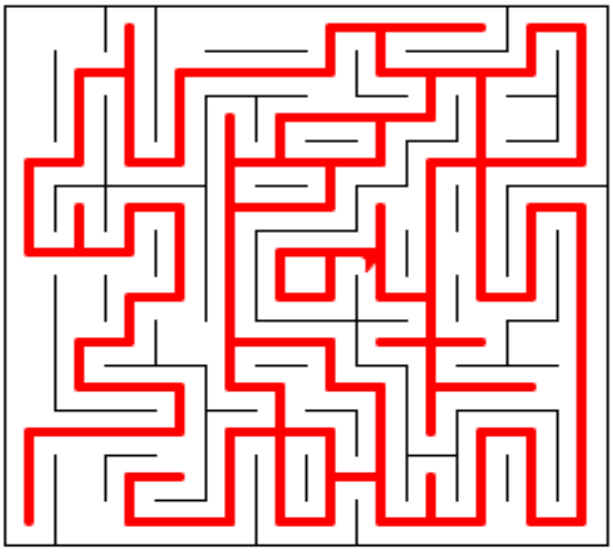


Figure 4: Uncovered cells from *Exploration Trial* in Test Maze 1

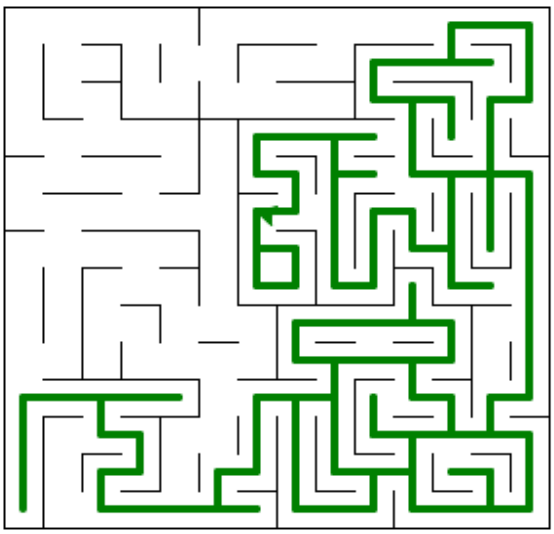


Figure 5: Uncovered cells from *Exploration Trial* in Test Maze 2

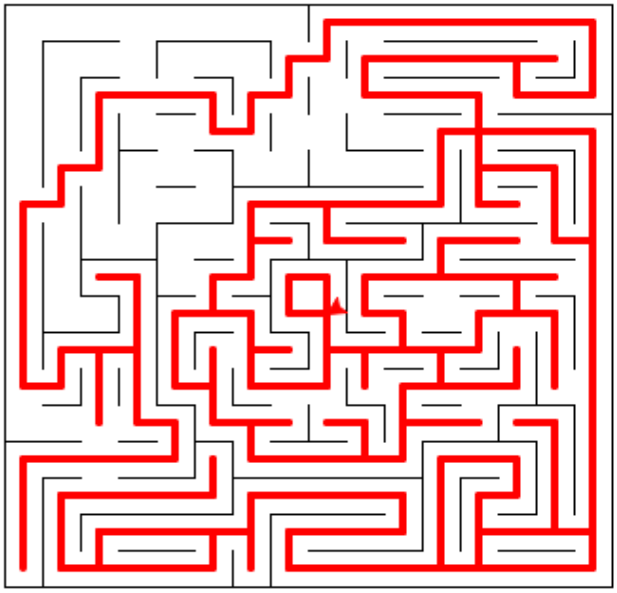


Figure 6: Uncovered cells from *Exploration Trial* in Test Maze 3

Figure 4,5,6 displays Test Maze, which is 12-by-12 (labeled 0 to 11). The cells in white have all been uncovered by the robot during the ***Exploration Trial***, while the squares highlighted in red were discovered. bottom left-hand corner of the maze is the starting cell and the four cells in the middle in square shape represent the goal area.

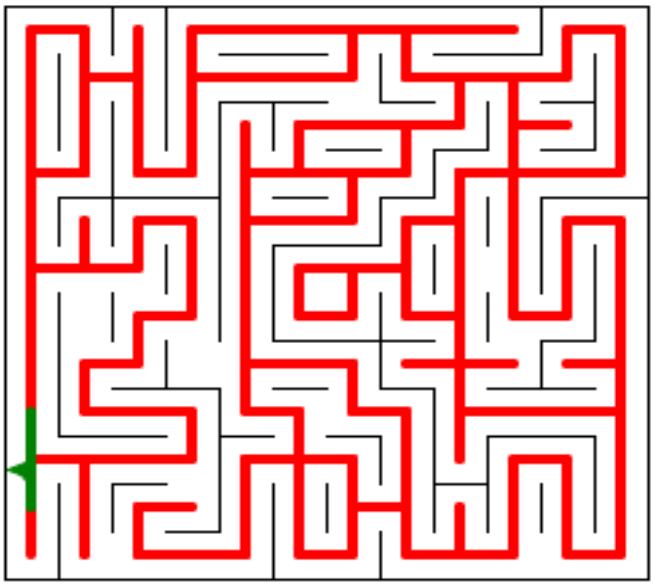


Figure 7: Covering all Uncovered cells from *Exploration Trial* in Test Maze 1

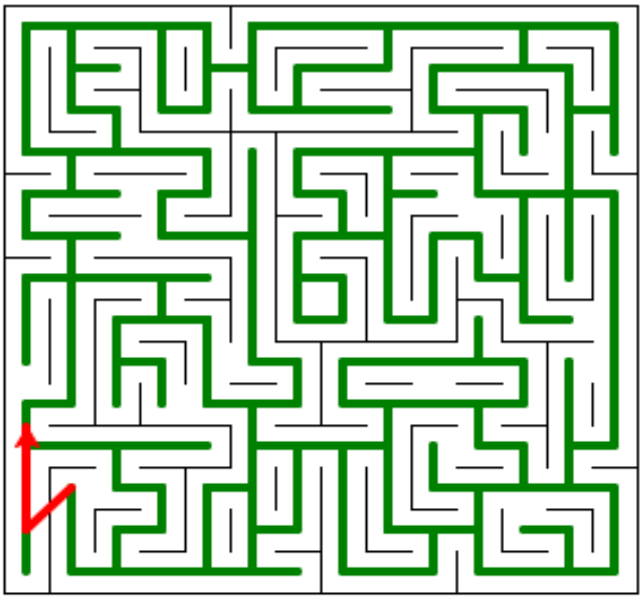


Figure 8: Covering all Uncovered cells from *Exploration Trial* in Test Maze 2

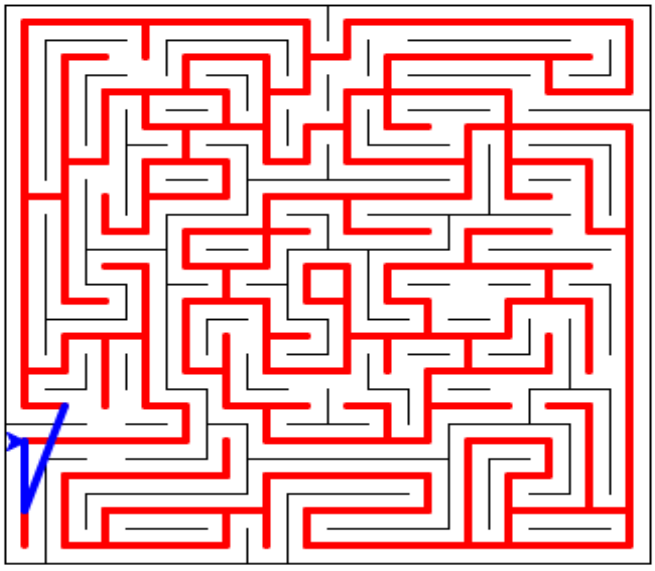


Figure 9: Covering all Uncovered cells from *Exploration Trial* in Test Maze 3

In order to uncover a significant amount of the maze, the robot was directed to continue its ***Exploration Trial*** (even if it had already found the goal) until it visited 100% of the total maze.

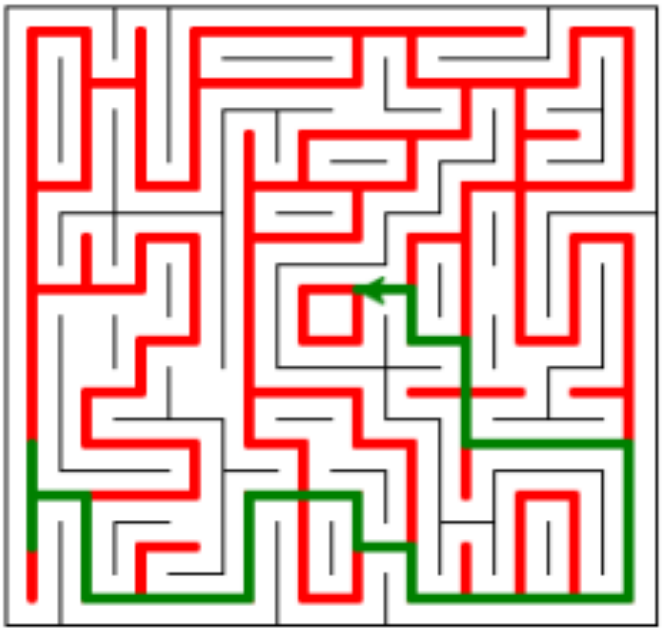


Figure 10: Covering shortest path from Optimization Trial Test Maze 1

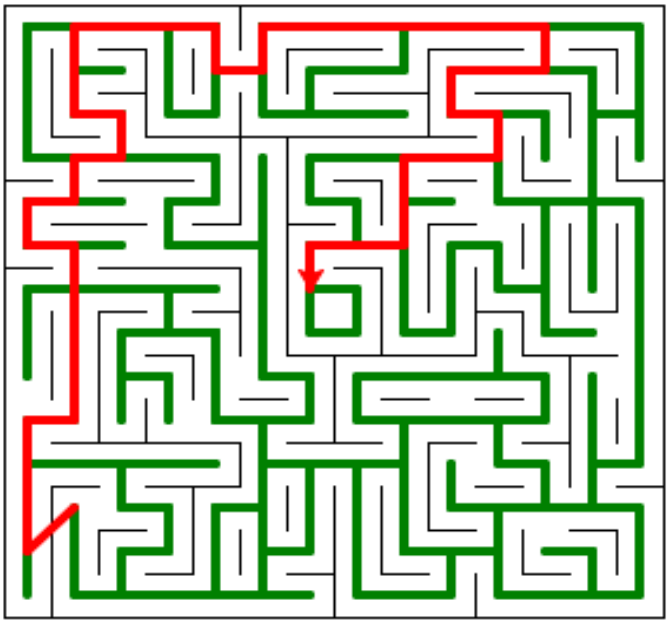


Figure 11: Covering shortest path from Optimization Trial Test Maze 2

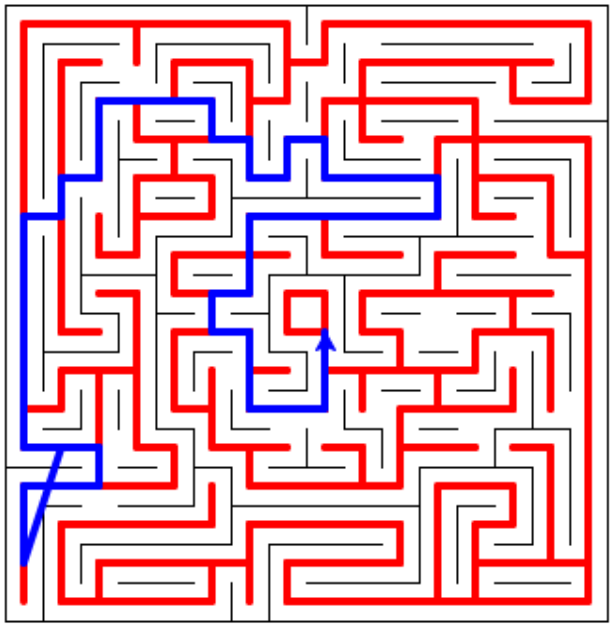


Figure 12: Covering shortest path from Optimization Trial Test Maze 3

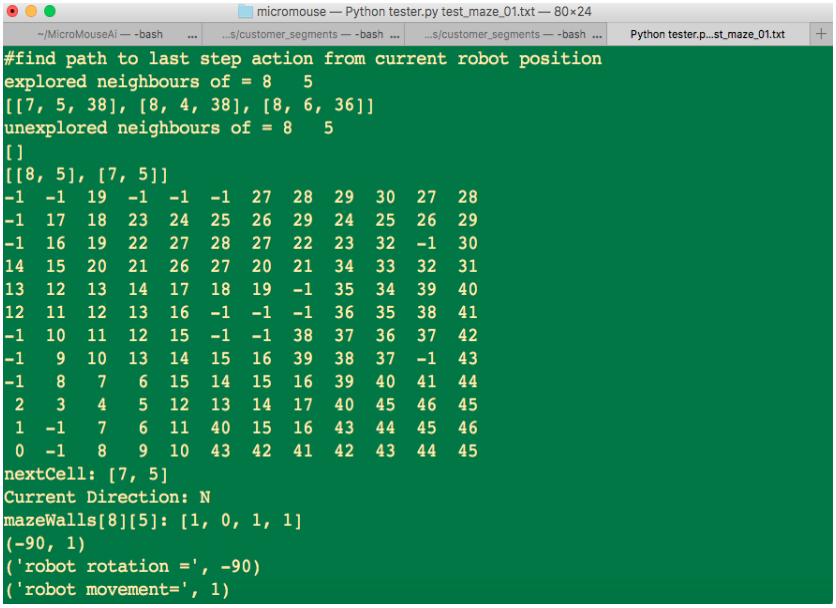


Figure 13: Terminal output showing intermediate path matrix for robot during exploration Trial in Test Maze 1

Similar kind of terminal output as shown in figure 13 is generated for each robot step along with real time robot tracking window.

Approach

Once the robot has discovered 100% of the maze and has found the goal at least once, it will be directed to reset and move back to the starting position to begin the Optimization Trial.

Before starting test trial we have explored the distance matrix for each cell in the maze using algorithm class object. Distance matrix is calculated based on single robot step so the path optimization algorithm will read this distance matrix and gives us shortest path, Then it applies multiple robot steps in previous explored shortest path from starting to goal. we have mentioned the optimal path for three mazes as following.

Optimal Path for Test Maze 1:

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

Optimal Path for Test Maze 2

[[0, 1], [0, 3], [1, 3], [2, 3], [2, 2], [3, 2], [3, 1], [2, 1], [2, 0], [3, 0], [5, 0], [5, 1], [6, 1], [6, 2], [6, 3], [7, 3], [8, 3], [8, 4], [9, 4], [10, 4], [10, 3], [11, 3], [11, 2], [12, 2], [12, 3], [13, 3], [13, 4], [13, 6], [12, 6], [11, 6], [11, 7], [10, 7], [10, 8], [10, 9], [9, 9], [8, 9], [8, 8], [7, 8], [6, 8], [6, 7], [7, 7]]

Optimal Path for Test Maze 3

[[0, 1], [0, 3], [1, 3], [2, 3], [2, 4], [1, 4], [0, 4], [0, 5], [0, 8], [0, 10], [1, 10], [1, 11], [2, 11], [2, 12], [2, 13], [3, 13], [5, 13], [5, 12], [6, 12], [6, 13], [7, 13], [7, 14], [8, 14], [8, 13], [9, 13], [12, 13], [12, 12], [12, 11], [13, 11], [14, 11], [14, 10], [14, 9], [13, 9], [11, 9], [11, 8], [11, 7], [10, 7], [10, 6], [9, 6], [8, 6], [8, 7]]

Note: shortest single step path may not be the fastest/optimal path as robot can take jumps[-3 to +3].we have included multiple steps logic to shortest single step pat

Algorithms and Techniques

When it comes to solving maze problems, there be several different strategies and algorithms one might use while attempting solve a maze. Before trying to program an exact algorithm to use, I found a bit of information on several applicable solutions - some more suitable for the ***Exploration Trial*** and others best appropriate for the ***Optimization Trial***.

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

Flood Fill: Algorithmic rule that starts in the middle and will fill each surrounding cell with a number of its relative distance from the end area.

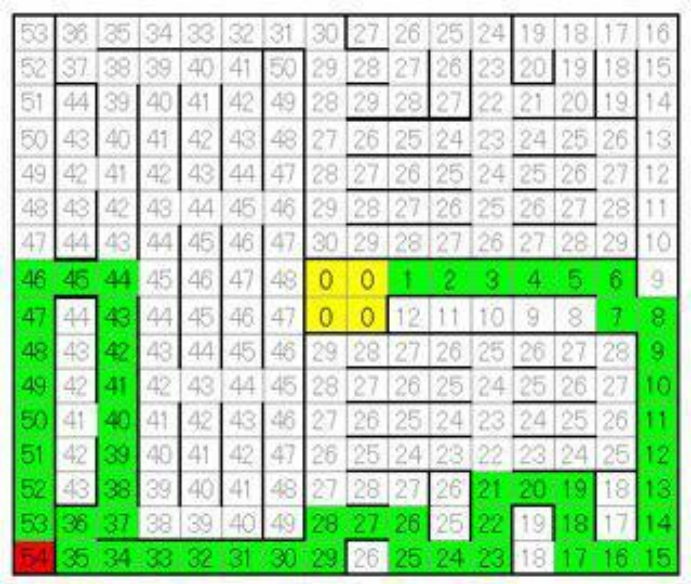


Figure 14: Example of Flood Fill Algorithm

Dead ends or loops: -

We have implemented flood fill algorithm it can handle loop and dead-end easily . Recursive Flood fill algorithm maintains a queue of all nodes whose distance matrix needs to be updated. There is a global 2d array mazeDepth which stores distance matrix information as well as weather that node is explored or not. This mazeDepth 2d array is initialized with -1 means this cell is not yet explored. When robot is reached a dead-end and an unvisited note is at head of queue then we call

Line 581: flooodfill.py
def findPathWhenStuck(self, sourceCell , targetCell):

this function gives us path to travel from dead-end node to new unexplored cell. Then robot takes step until this new path is fully traversed.

Loops can be easily handled as distance matrix for all the cells in the loop are already updated so robot when try to check for any new unexplored cell for next steps these loop related cells doesn't not exist in the queue once their distance is already updated in earlier steps.

Wall Follower: A Wall Follower algorithm works by simply leading the micro mouse to take every left or right turn possible, which should finally lead to the goal area (assuming the micro mouse does not get caught in a circular loop).

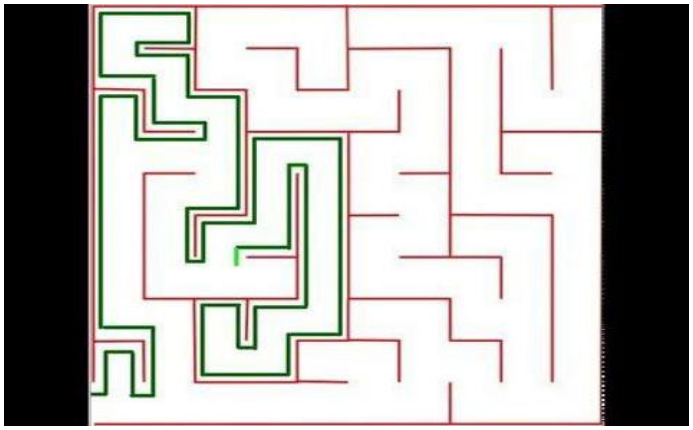


Figure 15: Example of Left Hand Rule

Dead ends or Loops: -

We haven't implemented wall follower as it is prone to fail with mazes having loops.

Depth-First Search & Breadth-First Search: These two algorithmic rule are known as graph traversal algorithmic rule. Depth-First Search starts at a root and follows a single path as far as feasible before seeing a different path. Similar to Depth-First Search, Breadth-First Search will start at a tree root, but will consider all neighboring paths before looking on to its next level neighbors. Both algorithms are proper for exploring the maze and finding an optimal path, but Breadth-First Search would require the robot to travel back to previously explored parts of the maze to discover new paths, and would thus likely cause a much larger exploration score.

Dead ends or Loops: -

As explained earlier during exploration trail we are using DFS queue for unexplored cells and explored cells for updating their distances . Distance matrix is updated for these cells using flood fill approach based on neighboring cells. Only during Testing/Optimisation step we are changing algorithm based such as Floodfill, DFS, BFS to find shortest path so loops and deadend during exploration is taken care by same approach as mentioned in Floodfill section.

For this project, I decided to use a modified version of Depth-First Search for the ***Exploration Trial*** and used one algorithm among Flood Fill, DFS and BFS based on project initialization parameters to create an optimal policy to use for the ***Optimization Trial.***

Benchmark

The benchmark model will be calculated by the performance score earlier discussed in the ***Problem Statement*** section: **score = [Number of Steps in Trial 2] + [Number of Steps in Trial 1 / 30]** . As we have capped the number of achievable steps at 1000, this

will be the simplest form of a benchmark performance. The benchmark model will contain a robot that makes a random movement decision at each step. If the mouse is able to navigate to the goal in less than 1000 steps in its ***Exploration Trial***, a new benchmark should be fixed to the total number of steps in Trial 1. Because the robot has gained knowledge of the maze layout in its ***Exploration Trial***, it should then optimize its route to the goal in the ***Optimization Trial***. Regardless of the number of steps taken during the ***Exploration Trial***, by nature of design, an optimal path exists for each maze - which is the minimum amount of steps needed to get from the starting location to the goal area.

Maze 1, 2, and 3 have optimal paths of 26, 41, and 41 steps, respectively. A functional benchmark would contain the sum of an ***average*** path to the goal and one thirtieth of an ***average*** amount of steps required to explore the maze. The challenge is defining what "average" actually means. Given the size of each maze and the allotted maximum amount of steps for the ***Exploration Trial*** , an ***average*** robot should absolutely be able to discover the goal area during exploration in far fewer than 1000 steps. Because the goal is to find an optimal path to the goal, the optimal routes of 26, 41, and 41 will be used for the ***Optimization Trial*** benchmark. For the ***Exploration Trial***, a benchmark model must discover every cell, which is impossible to achieve by only visiting every cell once. To discover every cell, a robot must visit a number of cells multiple times. In order to balance the use of the optimal route for the benchmark's performance in the ***Optimization Trial***, the total number of cells for each maze will be multiplied by 2.5 to generate the number of steps the benchmark model will take during the exploration phase.

Benchmark Maze 1 Score: 26 + (144 * 2.5) / 30**= 38.00**
Benchmark Maze 2 Score: 41 + (196 * 2.5) / 30**= 57.33**
Benchmark Maze 3 Score: 41 + (256 * 2.5) / 30**= 62.33**

III. METHODOLOGY

Data Preprocessing

No data preprocessing was needed for this project. The robot collects the essential information from its sensors and the maze environment. The robot will collect and store information until it has enough to apply an algorithm to optimally solve the maze.

Implementation

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

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

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

Beginning each time-step, the robot collects a new set of sensor readings from its left, front, and right sensors. Joining the sensor readings with information defining the direction the robot is facing and its current location, the robot is able to produce and store a four-bit number describing the surrounding maze walls and openings for its specific location (previously discussed in the *Data Exploration* section).

2. Robot Movement During *Exploration Trial*. After storing information on the maze walls, the next step is to determine how the robot should move throughout the maze during its *Exploration Trial*. Not only we are concerned with the robot moving through the openings

in the walls, but we also want the robot to favor previously-undiscovered cells while maneuvering in a general direction toward the goal area.

3. Finding Goal Area During *Exploration Trial*. The next process in implementation was determining how to ensure the robot will discover the goal area at least once during its *Exploration Trial*, and how much of the maze the robot should explore after it has found the goal (if continue exploring at all).

4. Identifying Possible and Optimal Paths to Goal. Now that the robot has discovered the goal area, it is time to determine all possible paths to the goal, and deduce one optimal path for the robot to take during its second trial. Again, borrowing learnings from Udacity's Artificial Intelligence for Robotics course, I implemented a Dynamic Programming method that given the map of the maze and the location of the goal area, outputs an optimal path to the goal from any starting cell. The Dynamic Programming method works by considering the locations of the walls in the maze and creates a Policy Grid that displays the optimal action (move left, right, up, or down) at every single grid cell leading to the goal area. One issue with Dynamic Programming is that it is very computationally involved

5. Ending *Exploration Trial*and Starting *Optimization Trial*. The last portion of implementation is determining when and how to end the *Exploration Trial*, and start the *Optimization Trial*. Once the robot has visited the goal at least once and has uncovered 100% of the map. Once an optimal path is identified, the robot stores it with the proper movements and rotations to follow the path. 'Rotation' and 'Movement' are both updated to 'Reset' which signals the *Optimization Trial*.

Refinement

Techniques used for exploration are flood fill and during testing phase we are using flood fill, dfs and bfs. During exploration distance matrix are also updated using flood fill based on neighboring cells.

Maze	Change	Value	Result
Test Maze 1	(a)full maze exploration. (b)path Optimization	Test path remains at 40 Steps even at full exploration. Path optimization reduces 40 Steps to 25 steps after considering multiple consecutive moves.	Final result of test phase became 25 steps and total maze score reduced to 38 from 47.
Test Maze 2	(a)full maze exploration. (b)path Optimization	Test path length reduced to 35 steps from 40 steps at full exploration. That means we were able to find unexplored optimal path.	Final result of test phase became 35 steps and there is increase in training due to which total score increased from 48 to 54. See improvement section to reduce training steps.
Test Maze 3	(a)full maze exploration. (b)path Optimization	Test path length reduced to 21 steps from 41 steps at full exploration. That means we were able to find unexplored optimal path.	Final result of test phase became 21 steps and there is increase in training steps to explore maze which results in increase in training steps from 492 to 662.

As Distance matrix is updated using flood fill approach so distance to unexplored cell can also be updated if it is neighbor of any currently explored cell.

This will help robot to reach unexplored cell without visiting them. During testing phase I have added path optimization logic to include multiple consecutive moves which can make robot take up to three steps instead of single steps.

In future same path optimization logic can also be applied in exploration phase for moving across explored cells.

IV. RESULTS

Model Evaluation and Validation

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

With 80% Exploration:

(test phase starts immediately after robot reached goal during exploration phase)

		Test Maze 1	Test Maze 2	Test Maze 3
	Dimensions	12-by-12	14-by-14	16-by-16
Algorithm Used	Benchmark Score	**38.00**	**57.33**	**62.33**
	Training steps	307	252	492
Flood Fill	Test steps	25	40	41
	Score	**35.233**	**48.400**	**57.400**
Depth-First Search	Training steps	307	252	492
	Test steps	24	38	36
	Score	**34.233**	**46.400**	**52.400**
Breadth-First Search	**Score**	**Time limit exceeded**	**Time limit exceeded**	**Time limit exceeded**

Overall, I would argue that the results can be trusted. the results and performance tend to be consistent, robust, and also routinely outperform the benchmark model.

With Full Maze Exploration:

(test phase starts after robot has fully explored all the cells)

		Test Maze 1	Test Maze 2	Test Maze 3
	Dimensions	12-by-12	14-by-14	16-by-16
Algorithm Used	Benchmark Score	**38.00**	**57.33**	**62.33**
Flood Fill	Training steps	405	596	662
	Test steps	25	35	22
	Score	**38.500**	**54.867**	**63.067**
Depth-First Search	Training steps	405	596	662
	Test steps	24	34	36
	Score	**37.500**	**53.867**	**58.067**
Breadth-First Search	**Score**	**Time limit exceeded**	**Time limit exceeded**	**Time limit exceeded**

Justification

Min. Coverage		Test Maze 1	Test Maze 2	Test Maze 3
	Train Score(training steps/30)	10.233	8.400	16.400
	Test Score(total test steps)	25.000	40.000	41.000
	Total Score	**35.233**	**48.40**	**57.40**
	Benchmark Score	**38.00**	**57.33**	**62.33**

The final results of the model outperformed the benchmark model, however if we build distance matrix by considering multiple steps jump then our algorithm would be able to find globally optimum path which can make robot reach in much less steps then current in the current approach. As per rubric robot is able to reach in much less than 1000 steps

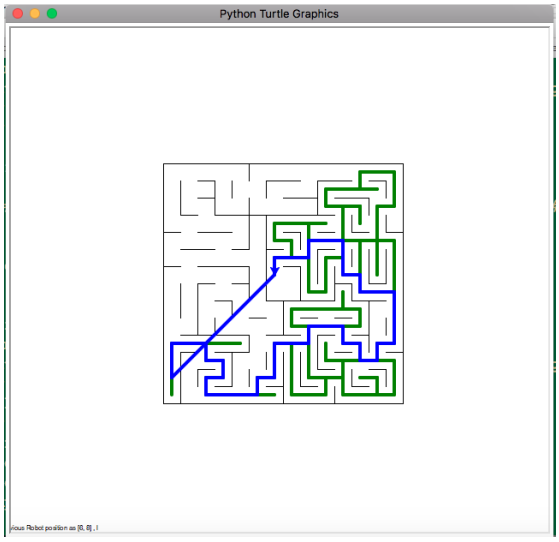
& also able to perform better then benchmark model. our approach can handle any kind of maze as we are using flood fill algorithm.

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

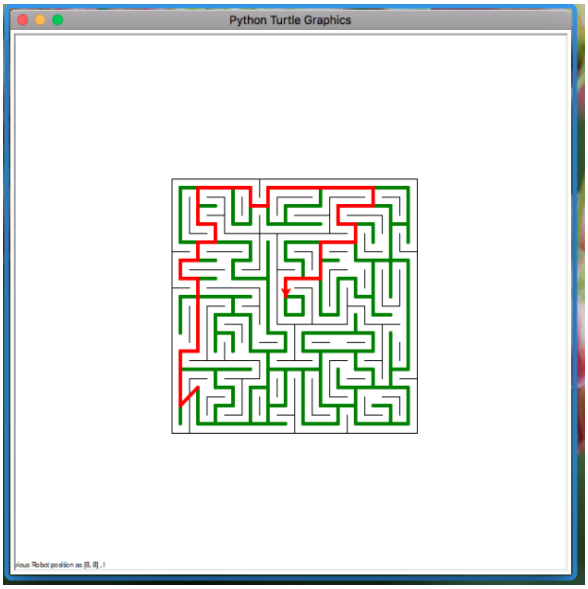
V. CONCLUSION

Free-Form Visualization

In this section I will dig into the performance of Maze 2 without full coverage versus full maze exploration . Below is the fig showing robot exploring maze 2 using flood fill algorithm in green color and completed test run in blue color. Same thing is written in chart above Flood fill algorithm without full maze exploration as robot has taken 40 steps during test run.



Below is the fig showing robot exploring maze 2 sing floodfill algorithm in green color and completed test run in red color with setting as full maze exploration. Same thing is written in chart above Flood fill algorithm with full maze exploration as robot has taken 35 steps during test run.



Above figures shows benefit of full maze exploration for finding optimal path in test run.If we are lucky to find globally shortest path without doing full maze exploration, then we will can generate better total score in such cases if robot still continues to do full maze exploration then it will add little cost to total final score. As we can see from score table for maze 1 using flood fill algorithm that robot is able to find optimal path in 10 steps without doing full maze exploration so total final score for robot is 35 compared to 38.5 with full maze exploration in case of maze 1 .

Robot is, unfortunately, not programmed to favor a path that allows making many consecutive moves of up to 3 steps during training and testing phase (partially implemented). In testing phase shortest path is found and then implemented consecutive move logic above that shortest path. Also during training if consecutive move logic is implemented for moving over already explored cell then we could improve final score little bit. All of mentioned functionality should be addressed in future revisions, as it can make a large impact on the final outcome.

Reflection

I was inspired to take on this project because I have always had an interest in AI. My masters these is also on similar topics. Now, with more and more of the products in the world becoming “smart”, my interest in the world of Game theory, Q-learning, Reinforcement Learning, A* learning clubbed with other deep learning based techniques is immense.

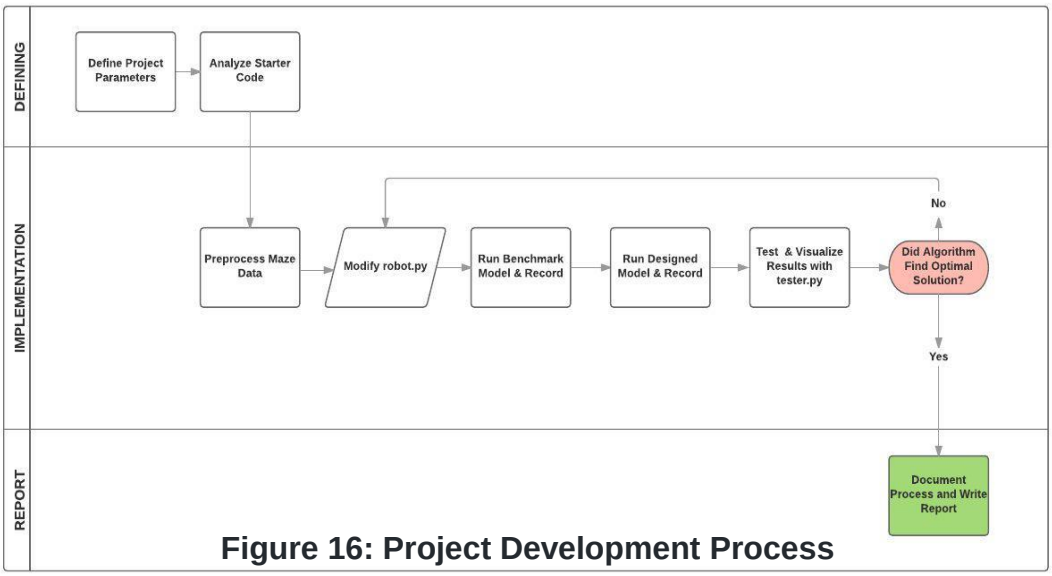


Figure 16: Project Development Process

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

1. Record and translate sensor readings into maze information and incorporate it with the robot’s orientation (heading) and location.
2. As the robot navigates the maze, update the map of the wall locations in the maze.
3. Implement a distance matrix from start for each cell to help lead the robot toward the goal area during exploration.
4. Update the robot’s location after every move and output the rotation angle and movement directions to assist the robot to the next space.
5. At each robot position during exploration phase we are updating distance matrix for neighboring cells using flood fill. If any explored neighbor of

new explored cell has value greater than this new explored cell then those nodes are again put into queue to update distance using flood fill.

6. Continue the previous steps until the robot has discovered the goal area or immediately start the test phase once robot has reached the goal once during exploration phase.
7. Find the shortest distance using distance matrix from start to goal and apply path optimization to include logic to include consecutive move upto 3 steps in the shortest path if possible.
8. Initiate the second run and ensure the robot follows the optimal path to the goal area.

I have used flood fill at every step during exploration not just during final test phase. Flood fill has performed well. It is possible to implement the non-recursive version of flood fill algorithm but for current project I have used Queue based flood fill approach.

Improvement

If maze dimension are in continuous domain and robot steps are also in continuous domain. Then we have to take average of lot of sensor values to determine the exact position of robot as well as the distance of wall. Such approach to build micromouse bot for real maze is available online. For this Udacity capstone I have worked on the part after Robot get response from environment and those response are recorded by robot sensors into discrete units. We have worked on algorithmic approach to solve maze. Implemented Flood fill algorithm can solve any kind of maze.

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

I would also be interested in implementing an array of alternative algorithms, mentioned in the ***Algorithms and Techniques*** section, and comparing their performance against

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

SOURCES

Udacity Project Files

https://docs.google.com/document/d/1ZFCH6jS3A5At7_v5IUM5OpAXJYiutFuSIjTzV_E-vdE/pub

Robots Dreams

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

Maze Solving Algorithms on YouTube

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

A Note on Two Problems in Connexion with Graphs

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

Micromouse USA

http://micromouseusa.com/

USC Micromouse Project

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

Red Blob Games

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

A Guide to Heuristic Based Planning

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

Dijkstra's algorithm revisited: the dynamic programming connexion

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

Graph Traversal

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

Udacity's Artificial Intelligence for Robotics Course

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