Plot and Navigate a Virtual Maze

Chao Wang

Udacity Machine Learning Nano-degree Capstone Project

I. Project Description

Atonomous robots have wide reaching applications. From Bomb sniffing to finding humans in wreckage to home automation. Major problems facing designers are power and reliable sensing mechanism and unfamiliar terrain robotic competitions have inspired engineers for many years [1]. Competitions are held all around the world based on autonomous robots. One of the competions with the richest history is micromouse. Micromouse is an event where small robot mice solve a 16x16 maze. It began in late 1970s, although there is some indication of events in 1950. Events are held worldwide, and are most popular in the UK, U.S., Japan, Singapore, India and South Korea [2], [3].

This project takes inspiration from Micromouse competitions, wherein a robot mouse is tasked with plotting a path from a corner of the maze to its center. The robot mouse may make multiple runs in a given maze. In the first run, the robot mouse tries to map out the maze to not only find the center, but also figure out the best paths to the center. In subsequent runs, the robot mouse attempts to reach the center in the fastest time possible, using what it has previously learned. In this project, we will create functions to control a virtual robot to navigate a virtual maze.

A. *Problem Formulation*

Given a maze of $n \times n$ dimension, where $n = 12, 14$, or 16, starting coordinates of $(0,0)$, a robot has two objectives:

- **First Run**: In the first run, the robot is allowed to freely roam the maze to build a map of the maze. It must enter the goal room at some point during its exploration, but is free to continue exploring the maze after finding the goal. After entering the goal room, the robot may choose to end its exploration at any time. The robot is then moved back to the starting position and orientation for its second run.
- **Second Run**: The robot attempts to reach the center in the fastest time possible. In this project, it means the robot needs to reach the center in the minimum number of steps.

In this project, a maximum of 1000 time steps are allotted to complete both runs for a single maze.

B. *Main Results*

In order to build the map of the maze, the robot needs to constantly keep track of its location i the maze, and process the input form the sensor, and record the necessary information to be able to navigate in the later round. In this report we will use the *Flood-Fill* algorithm to explore the maze and find the fastest path to the center of the maze. Details of the Flood-Fill algorithm will be introduced in the following sections.

C. *Performance Metric*

The robots score for the maze is equal to the number of time steps required to execute the second run, plus one thirtieth the number of time steps required to execute the first run. A maximum of one thousand time steps are allotted to complete both runs for a single maze.

We can find that the metric defined in this way penalizes the robot that fails to find the optimal path to the center in the second run as well as the inefficient exploration in the first run. In the first run, the robot is supposed to explore a lot of cells, especially for the complicate mazes. Therefore, it is reasonable to times $\frac{1}{30}$ to the total steps as a cost of exploration. The main goal is to have a short time running in the second run, that's why the second run has weight equals to 1 in the final score.

II. Data Exploration

In this section, we will introduce the specifications of the maze and the behavior of the robot.

A. *Maze Specifications*

The maze exists on an $n \times n$ grid of squares. The minimum value of n is 12, the maximum is 16. Along the outside perimeter of the grid, and on the edges connecting some of the internal squares, are walls that block all movement. The robot will start in the square in the bottom-left corner of the grid, facing upwards. The starting square will always have a wall on its right side (in addition to the outside walls on the left and bottom) and an opening on its top side. In the center of the grid is the goal room consisting of a 2×2 square; the robot must make it here from its starting square in order to register a successful run of the maze.

Mazes are provided to the system via text file. On the first line of the text file is a number describing the number of squares on each dimension of the maze n. On the following n lines, there will be n comma-delimited numbers describing which edges of the square are open to movement. Each number represents a four-bit number that has a bit value of 0 if an edge is closed (walled) and 1 if an edge is open (no wall); the 1s register corresponds with the upwards-facing side, the 2s register the right side, the 4s register the bottom side, and the 8s register the left side. For example, the number 10 means that a square is open on the left and right, with walls on top and bottom ($0 \times 1 + 1 \times 2 + 0 \times 4 + 1 \times 8 = 10$). Note that, due to array indexing, the first data row in the text file corresponds with the leftmost column in the maze, its first element being the starting square (bottom-left) corner of the maze.

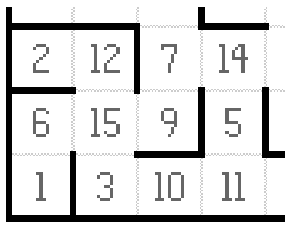


Fig. 1. Maze Example

B. Robot Specifications

The robot can be considered to rest in the center of the square it is currently located in, and points in one of the cardinal directions of the maze. The robot has three obstacle sensors, mounted on the front of the robot, its right side, and its left side. Obstacle sensors detect the number of open squares in the direction of the sensor; for example, in its starting position, the robot's left and right sensors will state that there are no open squares in those directions and at least one square towards its front. On each time step of the simulation, the robot may choose to rotate clockwise or counterclockwise ninety degrees, then move forwards or backwards a distance of up to three units. It is assumed that the robot's turning and movement is perfect. If the robot tries to move into a wall, the robot stays where it is. After movement, one time step has passed, and the sensors return readings for the open squares in the robots new location and/or orientation to start the next time unit.

More technically, at the start of a time step the robot will receive sensor readings as a list of three numbers indicating the number of open squares in front of the left, center, and right sensors (in that order) to its "next_move" function. The "next_move" function must then return two values indicating the robot's rotation and movement on that timestep. Rotation is expected to be an integer taking one of three values: -90, 90, or 0, indicating a counterclockwise, clockwise, or no rotation, respectively. Movement follows rotation, and is expected to be an integer in the range $[-3,3]$ inclusive. The robot will attempt to move that many squares forward (positive) or backwards (negative), stopping movement if it encounters a wall.

III. Exploratory Visualization

In this section, we will provide visualizations of one of the three example mazes ('Test_Maze_01') using the showmaze.py file. The original maze is shown in Fig. 2, which is a 12×12 maze.

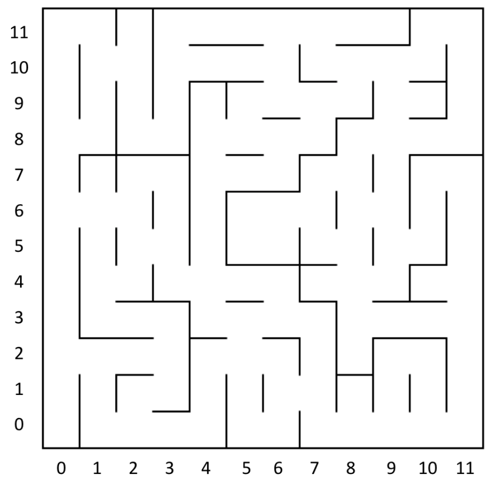


Fig. 2. Test_Maze_01

A. Traps in Maze

By checking the structure of the maze, we can find that in order to have a efficient way to roam the maze, the robot should know how to avoid some traps in the maze. As shown in Fig. III-A and Fig. III-A, there are at least two different traps can be found in the Test_Maze_01.

- **Cycles**: As shown in Fig. III-A, a maze may have some loops that can trap the robot. The algorithm of the robot should have a mechanism to avoid leading the robot in circle.
- **Dead ends**: As shown in Fig. III-A, a maze may have a lot of dead ends, which are the cells that have three walls. Once the robot visits these cells, the only available moves for next step is to move back. In order to make the exploration efficient, the algorithm of the robot should flag the dead end once the cell is first visited to avoid visiting it in the future. One should notice that, sometimes, the dead end may not necessarily be a signal cell. It may be an "alley" such as the path formed by cell $(8,11)$ and $(9,11)$.

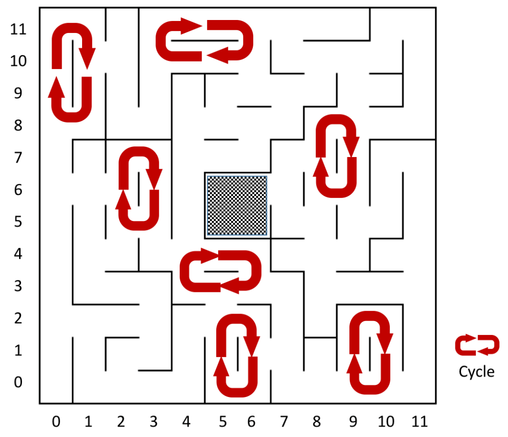


Fig. 3. Test_Maze_01: Cycles

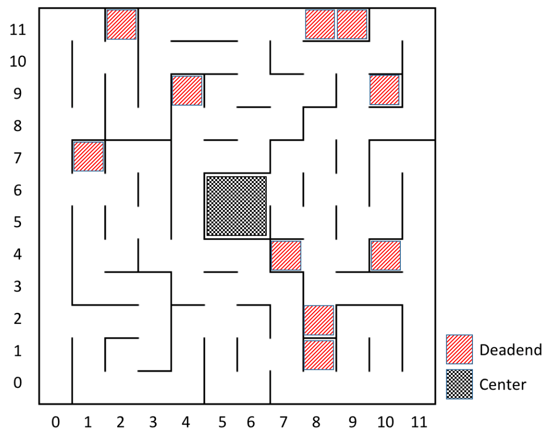


Fig. 4. Test_Maze_01: Deadend

B. Optimal Path of Test_Maze_01

In this section, we will trying to find the optimal path of the Test_Maze_01 by hand. As shown in Fig. 6, the path indicated by the green-solid line is the shortest path. The distance from the origin to the destination through this path is 30. There is another path which is indicated by the red-dash line in Fig. 7 also has 30 distance to the target. **However, since the robot is allowed to move forward up to 3 cells in one move step, the optimal path should be the one with more forward directions and less turns.** The green-solid path will need 17 steps to reach the destination and the red-dash path will need 21 steps to reach the destination. Therefore, the green-solid path shown in Fig. 6 is the optimal path of the Test_Maze_01.

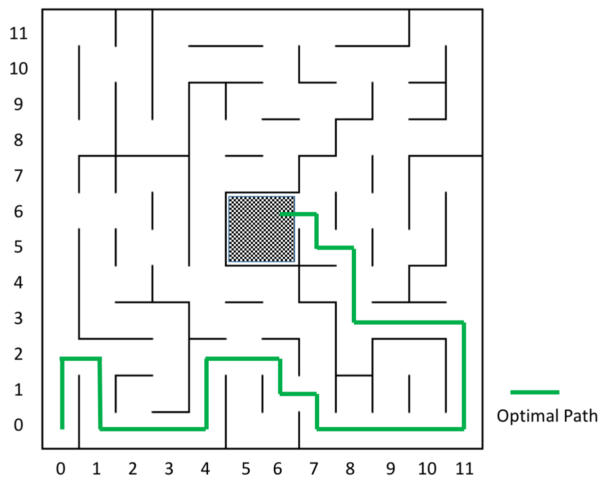


Fig. 5. Test_Maze_01: Optimal Path

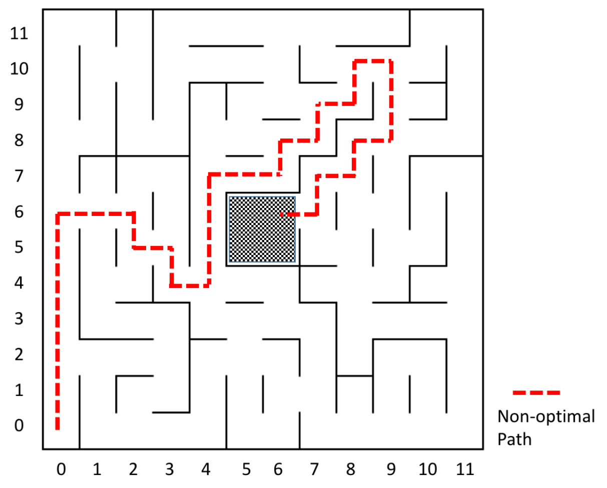


Fig. 6. Test_Maze_01: Non-optimal Path

IV. Benchmark

For the Test_Maze_01, the shortest distance from the origin to the destination is 30 which need 17 steps to finish. I would expect that the score for the most efficient solution is around 30 or less than 30 points. For the other two mazes (Test_Maze_02 and Test_Maze_01), since they are in the similar size, I would imagine the benchmark scores of those two mazes are also less than 30 points.

V. Algorithms

A. Right First

Here we consider a very simple algorithm, which is called as “Right First” algorithm. As the name implies, in the first run (exploration), the robot always turns right when right direction is a feasible direction. If the right direction is not available, the robot prefers to go straight without turning. If the “up” is not available either, then turn left. If the current cell is a dead end, the robot goes backward. In the other word, at any cell, the turning priority is ordered as “right” > “up” > “left” > “down”. Once the maze reach the center, the robot check the history path, and find out the path with minimum number of moves for the second round.

This simple algorithm will work if there are no cycles in the maze. If there is a cycle, the robot will be trapped. This algorithm will be used only for testing and free-form visualization later, since all the three test mazes have cycle.

B. New First

In this algorithm, at a intersection the robot will prefer to turn to the un-visited cell first. It is modified from the "Right-First" algorithm in order to avoid the cycles in the maze. To make it simple, the turning priority of the “New First” algorithm is ordered as

“right,un-visited” > “up,un-visited” > “left,un-visited”

> “right,visited” > “up,visited” > “left,visited” > “down,visited”

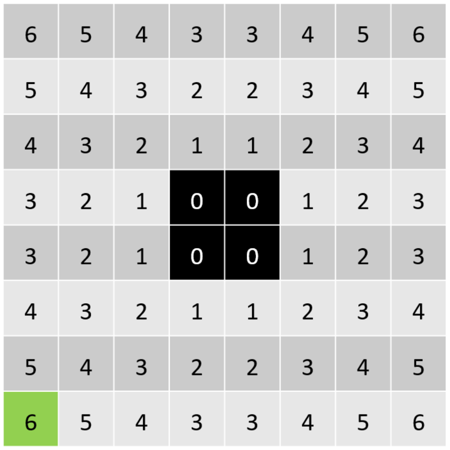
This algorithm enable the robot to reach the center of the maze even if there are cycles inside the maze. It allows the robot to explore bigger area of the maze than the “Right First” algorithm. Therefore, it has higher probability to find the optimal path than the “Right First” algorithm. However, this algorithm make the robot explore the maze without reasonable logic.

C. Flood Fill

The “Flood Fill” Algorithm is the optimal solution for this problem. Imagine we are stood in a maze which has non-permeable walls and flat, level floors. We also have a hosepipe. If we turn the hosepipe on, the water will start filling the maze. The shortest route to the target destination will be taken by the first drop of water that arrives there. This is why it is called the ’flooding’ algorithm. This algorithm ensures you get the shortest route (not necessarily the fastest though).

More specifically, the algorithm works as follows.

- Starting from the center ans assume there are no walls in the maze, calculate the distance of the center to the perimeter of the walls by adding 1 to each adjacent cell. For example, for the a maze with 6×6 dimension, we have the initial distance shown in Fig. 8. We call this figure as "distance table" of the maze.



6	5	4	3	3	4	5	6
5	4	3	2	2	3	4	5
4	3	2	1	1	2	3	4
3	2	1	0	0	1	2	3
3	2	1	0	0	1	2	3
4	3	2	1	1	2	3	4
5	4	3	2	2	3	4	5
6	5	4	3	3	4	5	6

Fig. 7. Example: Flood Fill Distance Initialization

- The distance value of each cell equals to the minimum distance value of the open adjacent cells (not separated by wall) plus 1.
- When the robot roam inside the maze, it will discover new walls. Once there is a new wall founded, the robot should update current "distance table". Not only the distance of current cell is updated, but also all the related adjacent cells and adjacent cells of the adjacent cells need to be updated, and so on. It is possible to update the cells all over the maze. There are two methods can be used to update the "distance table". One is the Brute-force method, which iteratively update all cells' distance value. The other one is the Stack method, where if a cell need to be updated, we push it onto the stack. This method keep updating the distance in the stack until the stack is empty. Either method will work, and both will do exactly the same thing. The better one to choose depends on processor speed and memory availability. For a slow processor with plenty of memory, the Stack method is preferred, but for a faster processor and limited memory, the Brute-force method is preferred. In this project, we will use the **Stack method** to update the "distance table".
- When moving, the robot will try to move to the adjacent cell which has the minimum distance to the destinations. If there are multiple adjacent cells has the same distance value, the robot will prefer to the cell that is facing forward. This will be helpful to get better performance in the second round.

VI. Data Processing

There is no data preprocessing needed in this project. All we need to do is to read the .txt files of three mazes. During the simulation, the sensor specification and environment designs are provided to the robot from the tester.py.

The turning of the robot is perfect and the sensor information is also always reliable.

VII. Methodology

In this section, I will briefly introduce the classes that I created for implementation of the three algorithms and the plotting of the final results. Besides the default scripts provided by Udacity, I create four more Python scripts which are "algorithms.py", "cell.py", "training.py", "global_values.py". I also finish the "robot.py".

A. Global Values

All the global variables are store in the "global_values.py", which makes it easy to import in any other classes.

B. Algorithm Class

The Algorithm class is defined in "algorithms.py".

```python
from global_values import WALL_VALUE

class Algorithm(object):
    def __init__(self):
        self.name = None

    def get_feasible_dir(self, adj_distances, adj_visited):
        raise NotImplementedError('Please specify the algorithm')
```

Three sub-classes of the Algorithm classes (RightFirst, NewFirst, and FloodFill) are inherited. Specifically, the FloodFill class is defined as:

```python
class FloodFill(Algorithm):

    def __init__(self):
        super(FloodFill, self).__init__()
        self.name = 'flood-fill'

    def get_feasible_dir(self, adj_distances, adj_visited):
        valid_dir= adj_distances.index(min(adj_distances))
        possible_distance = WALL_VALUE
        for i, dist in enumerate(adj_distances):
                # Prefer unvisited cells
            if dist != WALL_VALUE and adj_visited[i] is '':
                if dist <= possible_distance:
                    valid_dir = i
                    smallest_distance = dist
                    possible_distance = smallest_distance
                    # Index 1: forwarding
                    # go forward as much as possible
                    if valid_dir == 1:
                        break
        return valid_dir
```

The method “get_feasible_dir” take the information of the adjacent cells, including the distances to the center and the “visited” flags, as the inputs. The output is the index of the direction for next movement.

C. Cell Class

The Cell class is defined in “cell.py” stands for the cell in the maze. We generate the real-wall and virtual-wall for the the cell. The cell also has a distance to the destination, and a flag indicates whether this cell has been visited before. One cell in the maze will be printed out as:

```
'''
Structure of the cell
+ --- +   ---> roof: --- (real wall), ... (virtual wall)
| dis |   ---> frame: | (real wall), : (virtual wall)
+ --- +   ---> floor (same as roof)
'''
```

D. Training Class

The Training class is defined in “training.py”. There Training class the defined for representing the maze from the robot’s point of view. First it initializes the distances and visited flag of all the cells. The “New First” and the “Flood Fill” algorithms will use this information and update the information during roaming in the first run. All the three algorithms will use the distance and visited information on the way back to the origin and the second run.

The “update” method in the class is called by the robot class when it changes the location. The input are the location (x, y), heading direction, walls of current location, and whether it will exploring after reaching the center. The “update” method will update all the real walls information, distance of all the cells that need to be updated.

```
# update the cell
def update(self, x, y, heading, walls, explore):
    '''
    Here we update the cell's information
    '''
    cell = self.grid[x][y]

    # record the real walls if the cell is frist time visited
    if cell.visited == '':
        cell.real_walls = walls
        # here we updated the walls of the adjacent's wall
        # since two adjacent cells share the wall
        self.update_adj_walls(x,y,walls,'real')
    cell.visited = directions[heading]
    self.change_visual_prev_cell(cell, explore)
    self.last_visited_cell = cell
    self.update_dist()
```

The “draw” method will used to plot the maze in cell structure in the terminal, which shows the path and learning process of the robot.

```
    def print_row(self, cells, include_delimiters = True):
        if include_delimiters:
            roof = ''
            frame = '\n'
            floor = '\n'

            for cell in cells:
                roof += cell.roof()
                frame += cell.frame()
                floor += cell.floor()
            res = roof + frame + floor
        else:
            frame = ''
            for cell in cells:
                frame += cell.frame()
            res = frame

        print(res)
```

E. Robot Class

The Robot class is the main class that control the robot in the maze. The “next_move” method in the class is used to determine the next move the robot should make, based on the input from the sensors after its previous move. Sensor inputs are a list of three distances from the robot’s left, front, and right-facing sensors, in that order.

Outputs is a tuple of two values. The first value indicates robot rotation (if any), as a number: 0 for no rotation, +90 for a 90-degree rotation clockwise, and −90 for a 90-degree rotation counterclockwise. Other values will result in no rotation. The second value indicates robot movement, and the robot will attempt to move the number of indicated squares: a positive number indicates forwards movement, while a negative number indicates backwards movement. The robot may move a maximum of three units per turn. Any excess movement is ignored.

If the robot wants to end a run (e.g. during the first training run in the maze) then returning the tuple (‘Reset’, ‘Reset’) will indicate to the tester to end the run and return the robot to the start.

```
    def next_move(self, sensors):
        x, y, heading = self.get_current_location()
        walls = self.current_walls(x, y, heading, sensors)

        if self.is_at_centers(x, y):
            rotation = 0
            movement = -1
            self.training.update(x, y, heading, walls, self.explore)
            self.reach_dest = True
            self.explore = True

        else:
            self.training.cells_to_check.append([x,y])
            if [x,y] not in self.training.visited_before_reaching_destination:
                self.training.visited_before_reaching_destination.append([x,y])

            self.training.update(x,y,heading, walls, self.explore)
            rotation, movement = self.get_next_move(x,y,heading,sensors)

        self.update_location(rotation, movement)
        if rotation == 'Reset' and movement == 'Reset':
            self.reset_to_home()

        if self.location in self.destination and self.moves_round_2 != 0:
            self.report_results()

        return rotation, movement
```

VIII. Refinements

The above three algorithm show sequential improvement. However, when I first implement them I found that sometimes the robot may visit the same dead end so many times, or take unnecessary exploration in the second round. Therefore, I take the following robustness considerations to refine the simulations.

A. *Virtual Wall for Dead End*

As we discussed in Section III-A, there will be dead ends, cells with three walls, in the maze. In order to prevent the robot to visit the same dead end over and over again, we will place a "virtual" wall, at the entrance of the dead end, to signal the robot that this location is out of bounds. We will apply this "virtual" wall strategy to the "Flood Fill" algorithm.

B. *New Cells First*

In order to avoid being trapped in a cycle, both of the "New First" and "Flood Fill" algorithm will be prefer to move to the unvisited cells. In the "Flood Fill" algorithm, this strategy also make the robot explore more new cells in the maze.

C. *Moving Distance in Step*

As we mention before in Section II-B, the robot is allowed to move forward up to 3 cells in one move step. In order to get a better performance, the robot must take advantage of this rules, especially in the second run. Therefore, in the "Flood Fill" algorithm, I set the robot prefer to explore unvisited cells in the forward direction and try to move as much as it can (≤ 3). I believe that this will improve the performance of the robot.

D. *Exploration After Center*

After the robot reaches the centers of the maze, we have the options to make it continue exploring the maze on its way back to the origin. The objective is to see whether we can find a better solution than the current one. Later, we will find that this "exploration after reaching center" may not be necessary.

E. *Virtual Walls for Unvisited Cells*

After the robot finishes the exploration run (first run), it will palce "virtual" walls on all the unvisited cells in the first run. This will make sure that the robot will not make any unnecessary exploration in the second run.

IX. Results Discussion

In this section, we will discuss the simulation results in details. In order to run the "test.py" script, type the following command in the terminal:

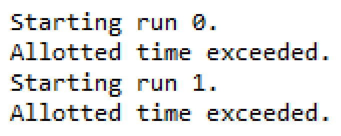
```
python tester.py test_maze_01.txt ff no
```

There are five arguments.

- The first one is "python" command.
- The second one is the "tester.py" script.
- The third one is the file name of the maze we want to test. There are three maze provided by Udacity, which are "test_maze_01.txt", "test_maze_02.txt", and "test_maze_03.txt". Later, I will add another test maze "test_maze_04.txt" which is generated by myself.
- The fourth argument is the algorithm we want to use. There are three algorithms can be chosen: "rf" stands for the Right-First algorithm; "nf" stands for the New-First Algorithm; "ff" stands for the "Flood-Fill" algorithm.
- The fifth argument can be "yes" or "no", which indicate whether the robot is set to keep exploring new cells after reaching the center and on the way back to the origin.

Once the simulation is finished, there will be two kinds of result printed out in the terminal window.

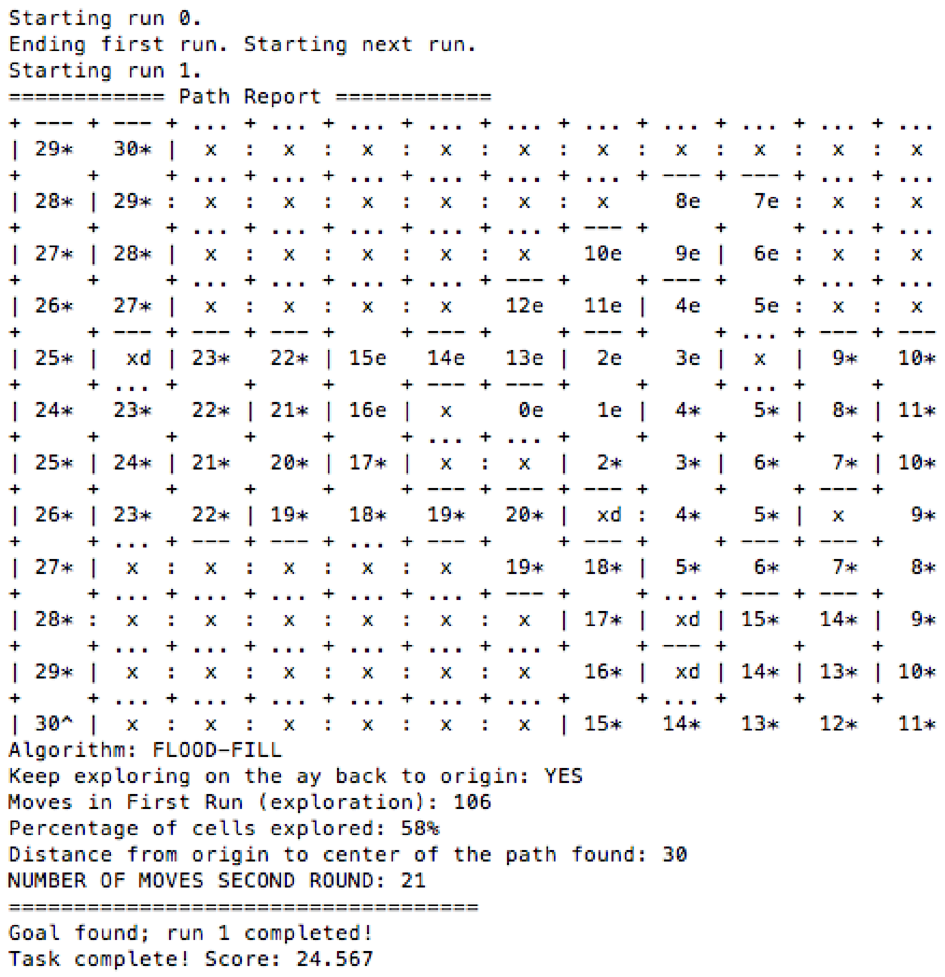
- The first possible output will be



```
Starting run 0.
Allotted time exceeded.
Starting run 1.
Allotted time exceeded.
```

It means that he algorithm used up all the 1000 steps budget but still hasn't found the center.

- The other possible output will be like:



```
Starting run 0.
Ending first run. Starting next run.
Starting run 1.
============ Path Report ============
+ --- + --- + ... + ... + ... + ... + ... + ... + ... + ... + ... + ...
| 29*   30* |  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x
+     +     + ... + ... + ... + ... + ... + ... + --- + --- + ... + ...
| 28* | 29* :  x  :  x  :  x  :  x  :  x  :  x     8e    7e :  x  :  x
+     +     + ... + ... + ... + ... + ... + --- +     +     + ... + ...
| 27* | 28* |  x  :  x  :  x  :  x  :  x    10e    9e |  6e :  x  :  x
+     +     + ... + ... + ... + ... + --- +     + --- +     + ... + ...
| 26*   27* |  x  :  x  :  x  :  x    12e   11e |  4e    5e :  x  :  x
+     + --- + --- + --- +     + --- +     + --- +     + ... + --- + ---
| 25* |  xd | 23*   22* | 15e   14e   13e |  2e    3e |  x  |  9*   10*
+     + ... +     +     +     + --- + --- +     +     + ... +     +
| 24*   23*   22* | 21* | 16e |  x     0e    1e |  4*    5* |  8* | 11*
+     +     +     +     +     + ... + ... +     +     +     +     +
| 25* | 24* | 21*   20* | 17* |  x  :  x  |  2*    3* |  6*    7* | 10*
+     +     +     +     +     + --- + --- + --- +     +     + --- +
| 26* | 23*   22* | 19*   18*   19*   20* |  xd :  4*    5* |  x     9*
+     + ... + --- + --- + ... + --- +     + --- +     + --- + --- +
| 27* |  x  :  x  :  x  :  x  :  x    19*   18* |  5*    6*    7*    8*
+     + ... + ... + ... + ... + ... + --- +     + ... + --- + --- +
| 28* :  x  :  x  :  x  :  x  :  x  :  x  | 17* |  xd | 15*   14* |  9*
+     + ... + ... + ... + ... + ... + ... +     + --- +     +     +
| 29* |  x  :  x  :  x  :  x  :  x  :  x    16* |  xd | 14* | 13* | 10*
+     + ... + ... + ... + ... + ... + ... +     + ... +     +     +
| 30^ |  x  :  x  :  x  :  x  :  x  :  x  | 15*   14*   13*   12*   11*
Algorithm: FLOOD-FILL
Keep exploring on the ay back to origin: YES
Moves in First Run (exploration): 106
Percentage of cells explored: 58%
Distance from origin to center of the path found: 30
NUMBER OF MOVES SECOND ROUND: 21
====================================
Goal found; run 1 completed!
Task complete! Score: 24.567
```

Not only the maze is printed out, but also all related information, such as moves in the first and second runs, percentage of the cells been visited, path length, and the final score are printed out. For the plotting of the maze, the legends are listed as following:

- "♯∗": The cell is visited on the ways from the origin to the center in the first run. "♯" is the distance from current cell to the center.
- "♯*e*": The cell is visited after the robot reaching the center and on the way back to the origin. "♯" is the distance from current cell to the center. The notation will shown up only in the case where the robot is allowed to take extra exploration after reaching the center. "e" stands for "exploration".
- "—", "|": Real walls in the maze.
- "...", ":": Virtual walls added in the maze by the robot to avoiding the dead end.
- "x", "xd": Unvisited cells or dead end.
- "<, ∧, >, v": Current direction (left, up, right, down).

The performance of all the three algorithms are listed in TABLE I.

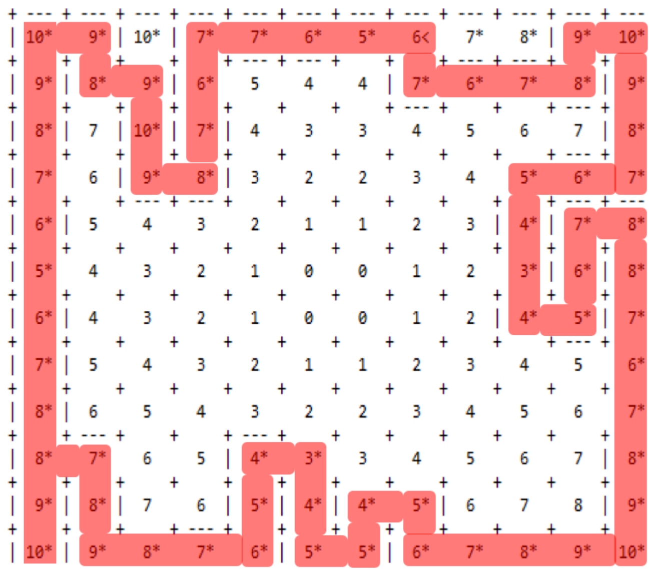
TABLE I

PERFORMANCE OF THREE ALGORITHMS

	Explore after center	Algorithm	Moves(Run1)	Cells Explored	Path Length	Moves(Run2)	Score
Maze 01	No	Right-First	N/A	N/A	N/A	N/A	Time exceeded
		New-First	145	74%	32	24	28.867
		Flood-Fill	90	47%	34	18	**21.033**
	Yes	Right-First	N/A	N/A	N/A	N/A	Time exceeded
		New-First	151	76%	32	24	29.067
		Flood-Fill	106	58%	30	21	24.567
Maze 02	No	Right-First	N/A	N/A	N/A	N/A	Time exceeded
		New-First	196	72%	45	30	36.567
		Flood-Fill	239	87%	43	25	**33.000**
	Yes	Right-First	N/A	N/A	N/A	N/A	Time exceeded
		New-First	209	77%	43	27	34.000
		Flood-Fill	239	87%	43	25	33.000
Maze 03	No	Right-First	N/A	N/A	N/A	N/A	Time exceeded
		New-First	N/A	N/A	N/A	N/A	Time exceeded
		Flood-Fill	109	35%	51	25	**28.667**
	Yes	Right-First	N/A	N/A	N/A	N/A	Time exceeded
		New-First	N/A	N/A	N/A	N/A	Time exceeded
		Flood-Fill	125	41%	51	25	29.200

A. *Performance of the Right-First Algorithm*

From TABLE I, we can find that, the Right First Algorithm never found the center within 1000 steps. The reason is that the robot got stuck in a cycle. The example of the cycle in Test_Maze_01 is shown in Fig 9.

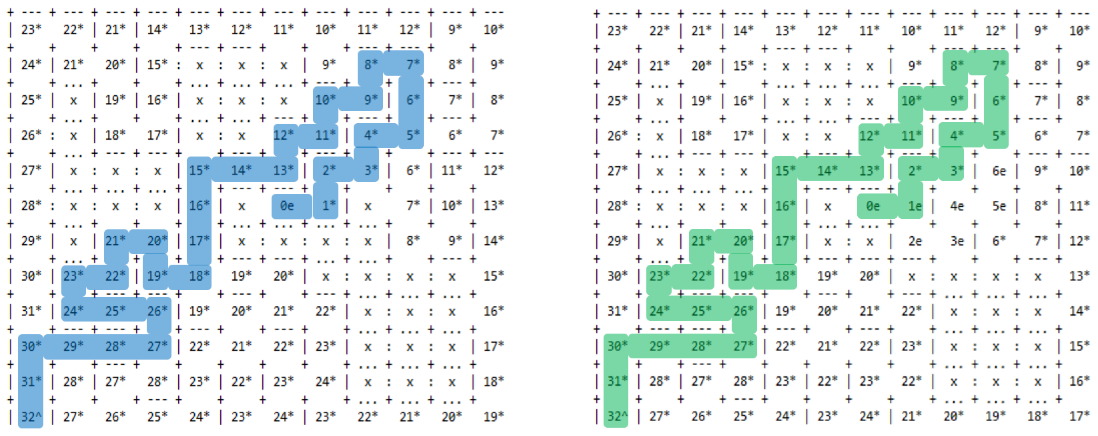


```
+ --- + --- + --- + --- + --- + --- + --- + --- + --- + --- + --- + ---
| 10*   9* | 10* |  7*   7*   6*   5*   6<   7*   8* |  9*  10*
+     +     +     +     + --- + --- +     +     + --- + --- +     +
|  9* |  8*   9* |  6*   5    4    4  |  7*   6*   7*   8* |  9*
+     +     +     +     +     +     +     + --- +     +     + --- +
|  8* |  7  | 10* |  7* |  4    3    3    4    5    6    7  |  8*
+     +     +     +     +     +     +     +     +     +     + --- +
|  7*   6  |  9*   8* |  3    2    2    3    4    5*   6*   7*
+     +     + --- + --- +     +     +     +     +     +     + --- + ---
|  6* |  5    4    3    2    1    1    2    3  |  4* |  7*   8*
+     +     +     +     +     +     +     +     +     +     +     +
|  5*   4    3    2    1    0    0    1    2    3* |  6* |  8*
+     +     +     +     +     +     +     +     +     +     +     +
|  6* |  4    3    2    1    0    0    1    2  |  4*   5* |  7*
+     +     +     +     +     +     +     +     +     +     + --- +
|  7* |  5    4    3    2    1    1    2    3    4    5    6*
+     +     +     +     +     +     +     +     +     +     +     +
|  8* |  6    5    4    3    2    2    3    4    5    6    7*
+     + --- +     +     + --- +     +     +     +     +     +     +
|  8*   7*   6    5  |  4*   3*   3    4    5    6    7  |  8*
+     +     +     +     +     +     +     +     +     +     +     +
|  9* |  8* |  7    6  |  5* |  4* |  4*   5* |  6    7    8  |  9*
+     +     +     + --- +     +     +     +     +     +     +     +
| 10* |  9*   8*   7*   6* |  5*   5* |  6*   7*   8*   9*  10*
```

Fig. 8. Example: Cycle in Test_maze_01

B. Performance of the New-First Algorithm

By using the New-First Algorithm, the robot is able to reach the center and have a relative good performance. Use the Test_maze_01 as an example, If the robot is not allowed to explore after reaching the center, the final score is 28.867. The path that the robot found has length equals to 32 and the robot need 24 steps to reach the center. Even if the robot is allowed to explore after reaching the center of the maze, it cannot find find a better path. Due to extra time spend on exploring, the final score is 29.067. The path in the maze is shown in Fig. 10.



(a) Exploration After Center: NO

(b) Exploration After Center: YES

Fig. 9. Test_Maze_01: New First Algorithm

C. Performance of the Flood-Fill Algorithm

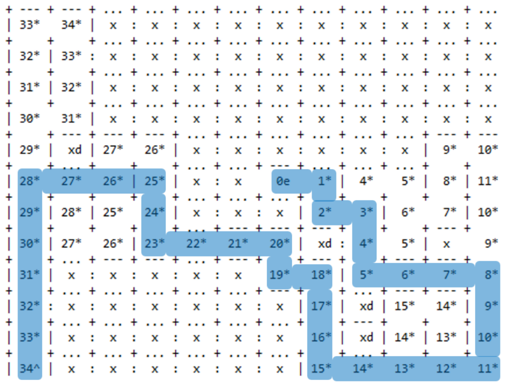
From TABLE I, we can find that the Flood-Fill Algorithm has the best performance among all the three algorithms. For Test_Maze_01, the best score is 21.033. For Test_Maze_02, the best score is 33.000. For Test_Maze_03, the best score is 28.667. All these best scores are achieved when the robot is not allowed to explore more cells after reaching the center. From this point of view, the extra exploration is not helpful in these three test mazes. The extra exploration will only increase the number of moves in the first round and drag down the final score. However, in Section X, we will show that in some particular cases, this extra exploration may be helpful.

The plotting of the paths for all the three test mazes are shown in Fig. 11, 12, 13, respectively.

Now we are comparing the performance with the optimal solution:

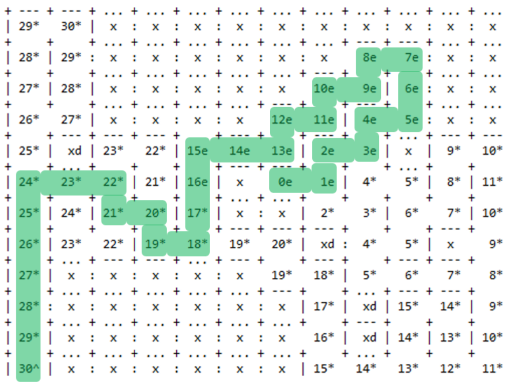
- For the Test_Maze_01, the optimal path has length 30 and the robot need 17 moves to reach the center. The solution that Flood-Fill algorithm found is a path with length 34 and 18 moves to the center.
- For the Test_Maze_02, the optimal path has length 43 and the robot need 23 moves to reach the center. The solution that Flood-Fill algorithm found is a path with length 43 and 25 moves to the center.
- For the Test_Maze_03, the optimal path has length 49 and the robot need 25 moves to reach the center. The solution that Flood-Fill algorithm found is a path with length 51 and 25 moves to the center.

We can see that solutions found by the Flood-Fill Algorithm are very close to the optimal solutions. They are also very closed to the benchmark we estimated in Section IV. We can conclude that the Flood Fill Algorithm is a very efficient algorithm.



```
+ --- + --- + ... + ... + ... + ... + ... + ... + ... + ... + ... + ...
| 33*   34* |  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x
+     +     + ... + ... + ... + ... + ... + ... + ... + ... + ... + ...
| 32* | 33* :  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x
+     +     + ... + ... + ... + ... + ... + ... + ... + ... + ... + ...
| 31* | 32* |  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x
+     +     + ... + ... + ... + ... + ... + ... + ... + ... + ... + ...
| 30*   31* |  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x
+     + --- + --- + --- + ... + ... + ... + ... + ... + ... + --- + ---
| 29* |  xd | 27*   26* |  x  :  x  :  x  :  x  :  x  :  x  |  9*   10*
+     + ... +     +     + ... + ... + --- + ... + ... + ... +     +
| 28*   27*   26* | 25* |  x  :  x     0e    1* |  4*    5* |  8* | 11*
+     +     +     +     + ... + ... + ... +     +     +     +     +
| 29* | 28* | 25*   24* |  x  :  x  :  x  |  2*    3* |  6*    7* | 10*
+     +     +     +     + ... + --- + --- + --- +     +     + --- +
| 30* | 27*   26* | 23*   22*   21*   20* |  xd :  4*    5* |  x     9*
+     + ... + --- + --- + ... + --- +     + --- +     + --- + --- +
| 31* |  x  :  x  :  x  :  x  :  x    19*   18* |  5*    6*    7*    8*
+     + ... + ... + ... + ... + ... + --- +     + ... + --- + --- +
| 32* :  x  :  x  :  x  :  x  :  x  :  x  | 17* |  xd | 15*   14* |  9*
+     + ... + ... + ... + ... + ... + ... +     + --- +     +     +
| 33* |  x  :  x  :  x  :  x  :  x  :  x    16* |  xd | 14* | 13* | 10*
+     + ... + ... + ... + ... + ... + ... +     + ... +     +     +
| 34^ |  x  :  x  :  x  :  x  :  x  :  x  | 15*   14*   13*   12*   11*
```

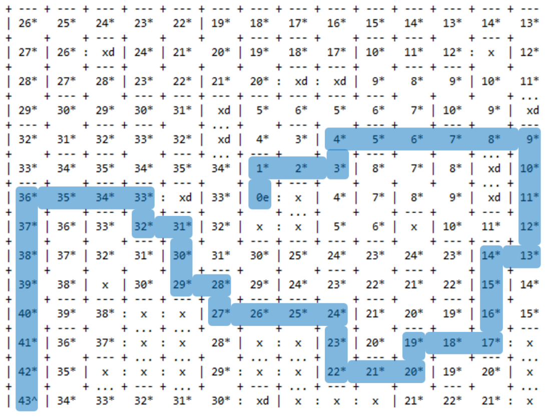
(a) Exploration After Center: NO



```
+ --- + --- + ... + ... + ... + ... + ... + ... + ... + ... + ... + ...
| 29*   30* |  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x  :  x
+     +     + ... + ... + ... + ... + ... + ... + --- + --- + ... + ...
| 28* | 29* :  x  :  x  :  x  :  x  :  x  :  x     8e    7e :  x  :  x
+     +     + ... + ... + ... + ... + ... + --- +     +     + ... + ...
| 27* | 28* |  x  :  x  :  x  :  x  :  x    10e    9e |  6e :  x  :  x
+     +     + ... + ... + ... + ... + --- +     + --- +     + ... + ...
| 26*   27* |  x  :  x  :  x  :  x    12e   11e |  4e    5e :  x  :  x
+     + --- + --- + --- +     + --- +     + --- +     + ... + --- + ---
| 25* |  xd | 23*   22* | 15e   14e   13e |  2e    3e |  x  |  9*   10*
+     + ... +     +     +     + --- + --- +     +     + ... +     +
| 24*   23*   22* | 21* | 16e |  x     0e    1e |  4*    5* |  8* | 11*
+     +     +     +     +     + ... + ... +     +     +     +     +
| 25* | 24* | 21*   20* | 17* |  x  :  x  |  2*    3* |  6*    7* | 10*
+     +     +     +     +     + --- + --- + --- +     +     + --- +
| 26* | 23*   22* | 19*   18*   19*   20* |  xd :  4*    5* |  x     9*
+     + ... + --- + --- + ... + --- +     + --- +     + --- + --- +
| 27* |  x  :  x  :  x  :  x  :  x    19*   18* |  5*    6*    7*    8*
+     + ... + ... + ... + ... + ... + --- +     + ... + --- + --- +
| 28* :  x  :  x  :  x  :  x  :  x  :  x  | 17* |  xd | 15*   14* |  9*
+     + ... + ... + ... + ... + ... + ... +     + --- +     +     +
| 29* |  x  :  x  :  x  :  x  :  x  :  x    16* |  xd | 14* | 13* | 10*
+     + ... + ... + ... + ... + ... + ... +     + ... +     +     +
| 30^ |  x  :  x  :  x  :  x  :  x  :  x  | 15*   14*   13*   12*   11*
```

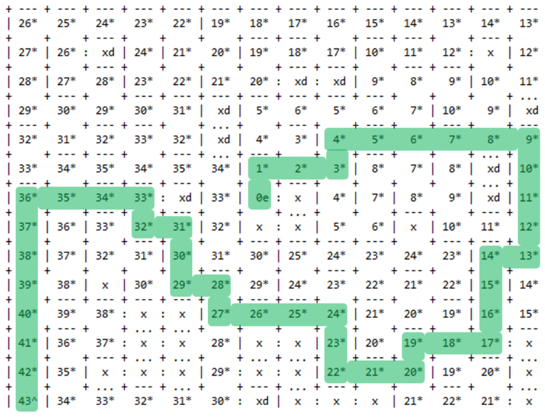
(b) Exploration After Center: YES

Fig. 10. Test_Maze_01: Flood Fill Algorithm



```
+ --- + --- + --- + --- + --- + --- + --- + --- + --- + --- + --- + --- + --- + ---
| 26*   25*   24*   23*   22* | 19*   18*   17*   16*   15*   14*   13*   14*   13*
+     +     + --- +     +     +     + --- + --- +     + --- + --- +     + --- +
| 27* | 26* :  xd | 24* | 21*   20* | 19*   18*   17* | 10*   11*   12* :  x  | 12*
+     +     + --- +     +     +     +     + --- + --- +     + --- + --- +     +
| 28* | 27*   28* | 23*   22* | 21*   20* :  xd :  xd |  9*    8*    9* | 10*   11*
+     + --- +     + --- + --- + --- + --- + --- + --- + --- +     +     +     + ...
| 29*   30*   29*   30*   31* |  xd |  5*    6*    5*    6*    7* | 10*    9* |  xd
+ --- +     + --- + --- +     + ... +     + --- +     + --- +     + --- +     + ---
| 32*   31*   32*   33*   32* |  xd |  4*    3* |  4*    5*    6*    7*    8*    9*
+     + --- + --- +     + --- + ... + --- +     +     + --- +     +     + ... +
| 33*   34*   35*   34*   35*   34* |  1*    2*    3* |  8*    7* |  8* |  xd | 10*
+ --- +     + --- + --- + --- +     +     + --- +     +     +     +     + ... +
| 36*   35*   34*   33* :  xd | 33* |  0e :  x  |  4* |  7* |  8*    9* |  xd | 11*
+     +     + --- +     + --- +     +     + ... +     +     + --- +     + --- +
| 37* | 36* | 33*   32*   31* | 32* |  x  :  x  |  5*    6* |  x  | 10*   11*   12*
+     +     +     + --- +     +     + --- + --- + --- + --- +     + --- + --- +
| 38* | 37* | 32*   31* | 30*   31*   30* | 25*   24*   23*   24*   23* | 14*   13*
+     +     +     +     +     + --- +     +     + --- +     + --- +     +     +
| 39*   38* |  x  | 30*   29*   28*   29* | 24*   23*   22*   21*   22* | 15* | 14*
+     + --- + --- + --- + --- +     + --- + --- +     + --- +     + --- +     +
| 40*   39*   38* :  x  :  x  | 27*   26*   25*   24* | 21*   20*   19* | 16*   15*
+     + --- +     + ... + ... +     +     +     +     +     + --- +     +     + ---
| 41* | 36*   37* :  x  :  x    28* |  x  :  x  | 23* | 20*   19*   18*   17* :  x
+     +     + --- + ... + ... +     + ... + ... +     + --- +     +     + --- + ...
| 42* | 35* |  x  :  x  :  x  | 29* :  x  :  x  | 22*   21*   20* | 19*   20* |  x
+     +     + ... + --- + ... +     + --- + ... + --- +     +     + --- +     + ...
| 43^ | 34*   33*   32*   31*   30* :  xd |  x  :  x  :  x  | 21*   22*   21* :  x
```

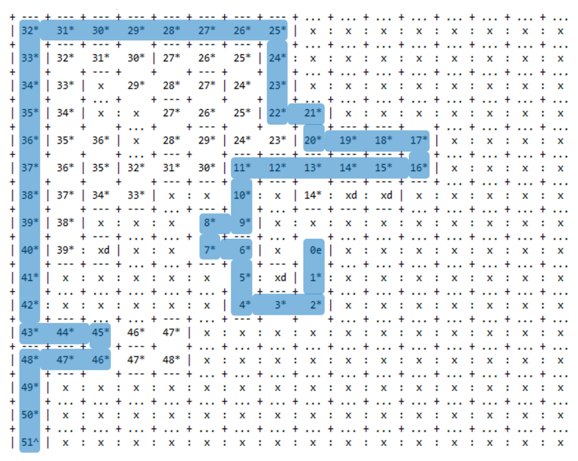
(a) Exploration After Center: NO



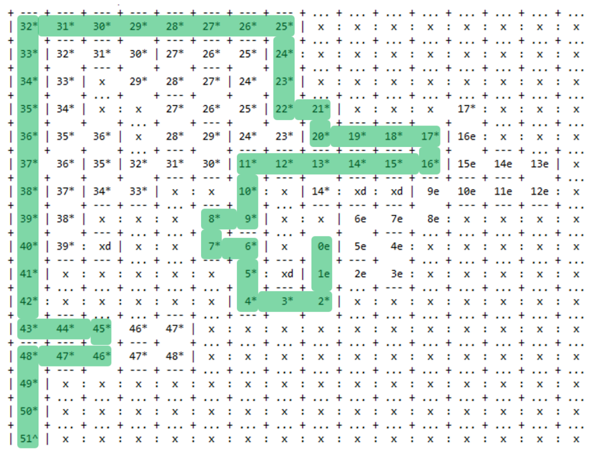
```
+ --- + --- + --- + --- + --- + --- + --- + --- + --- + --- + --- + --- + --- + ---
| 26*   25*   24*   23*   22* | 19*   18*   17*   16*   15*   14*   13*   14*   13*
+     +     + --- +     +     +     + --- + --- +     + --- + --- +     + --- +
| 27* | 26* :  xd | 24* | 21*   20* | 19*   18*   17* | 10*   11*   12* :  x  | 12*
+     +     + --- +     +     +     +     + --- + --- +     + --- + --- +     +
| 28* | 27*   28* | 23*   22* | 21*   20* :  xd :  xd |  9*    8*    9* | 10*   11*
+     + --- +     + --- + --- + --- + --- + --- + --- + --- +     +     +     + ...
| 29*   30*   29*   30*   31* |  xd |  5*    6*    5*    6*    7* | 10*    9* |  xd
+ --- +     + --- + --- +     + ... +     + --- +     + --- +     + --- +     + ---
| 32*   31*   32*   33*   32* |  xd |  4*    3* |  4*    5*    6*    7*    8*    9*
+     + --- + --- +     + --- + ... + --- +     +     + --- +     +     + ... +
| 33*   34*   35*   34*   35*   34* |  1*    2*    3* |  8*    7* |  8* |  xd | 10*
+ --- +     + --- + --- + --- +     +     + --- +     +     +     +     + ... +
| 36*   35*   34*   33* :  xd | 33* |  0e :  x  |  4* |  7* |  8*    9* |  xd | 11*
+     +     + --- +     + --- +     +     + ... +     +     + --- +     + --- +
| 37* | 36* | 33*   32*   31* | 32* |  x  :  x  |  5*    6* |  x  | 10*   11*   12*
+     +     +     + --- +     +     + --- + --- + --- + --- +     + --- + --- +
| 38* | 37* | 32*   31* | 30*   31*   30* | 25*   24*   23*   24*   23* | 14*   13*
+     +     +     +     +     + --- +     +     + --- +     + --- +     +     +
| 39*   38* |  x  | 30*   29*   28*   29* | 24*   23*   22*   21*   22* | 15* | 14*
+     + --- + --- + --- + --- +     + --- + --- +     + --- +     + --- +     +
| 40*   39*   38* :  x  :  x  | 27*   26*   25*   24* | 21*   20*   19* | 16*   15*
+     + --- +     + ... + ... +     +     +     +     +     + --- +     +     + ---
| 41* | 36*   37* :  x  :  x    28* |  x  :  x  | 23* | 20*   19*   18*   17* :  x
+     +     + --- + ... + ... +     + ... + ... +     + --- +     +     + --- + ...
| 42* | 35* |  x  :  x  :  x  | 29* :  x  :  x  | 22*   21*   20* | 19*   20* |  x
+     +     + ... + --- + ... +     + --- + ... + --- +     +     + --- +     + ...
| 43^ | 34*   33*   32*   31*   30* :  xd |  x  :  x  :  x  | 21*   22*   21* :  x
```

(b) Exploration After Center: YES

Fig. 11. Test_Maze_02: Flood Fill Algorithm



(a) Exploration After Center: NO



(b) Exploration After Center: YES

Fig. 12. Test_Maze_03: Flood Fill Algorithm

X. Free-Form Visualization

In this section, I will apply the Flood-Fill algorithm to a maze generated by myself, which is named as "test_maze_04.txt". As we discussed in Section V-C, the Flood-Fill algorithm will prefer to going straight to make a turn. This strategy is chosen to take the advantage that the robot can move up 3 cells in one step. However, if the fastest path can only be found by taking an early right turn but not going straight, the algorithm will fail to find it. I generate the Test_Maze_04 to illustrate this situation. As shown in Fig 14, the green-solid line indicates the optimal fastest path. However, because of the mechanism of Flood-Fill algorithm, if we do not allow the robot to explore extra cells after reaching the center, it may only find the non-optimal path indicated by the red-dash line. Let's check the simulation result.

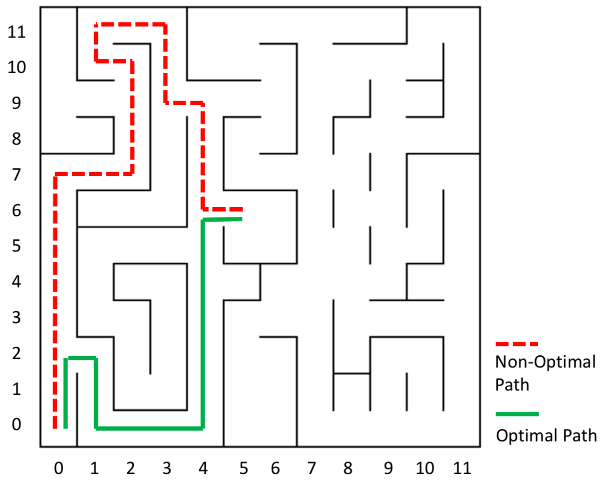
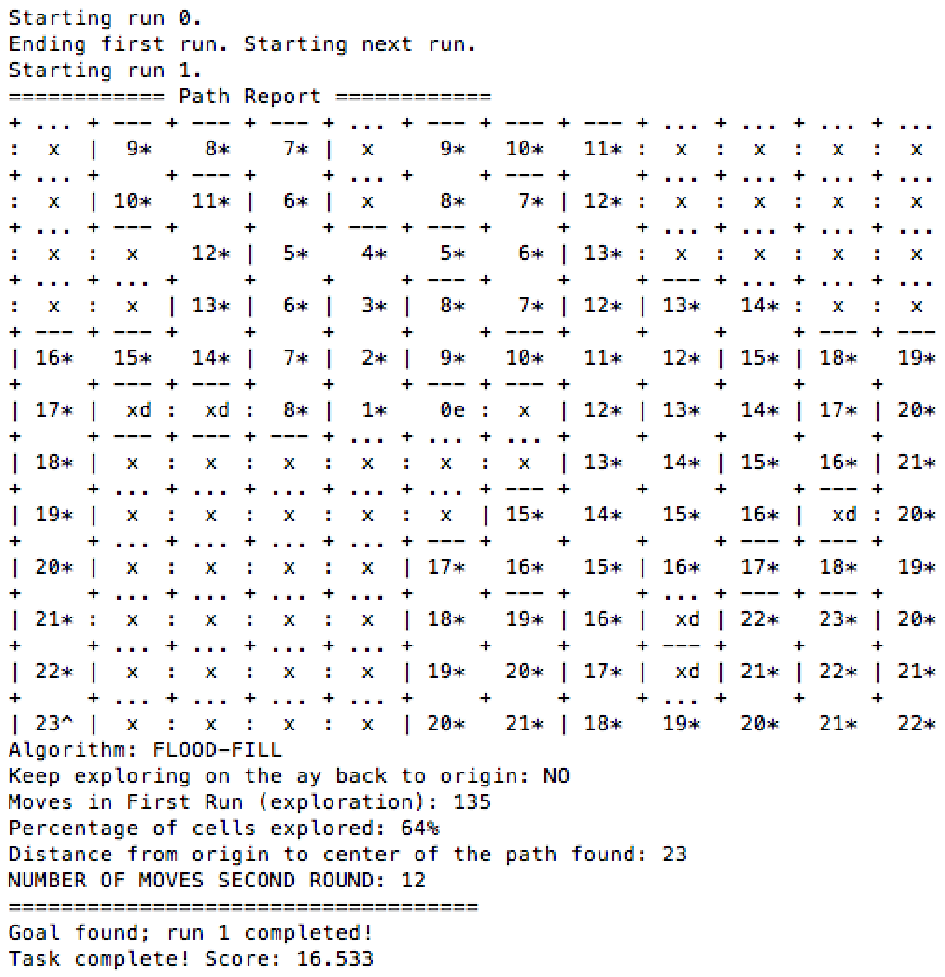


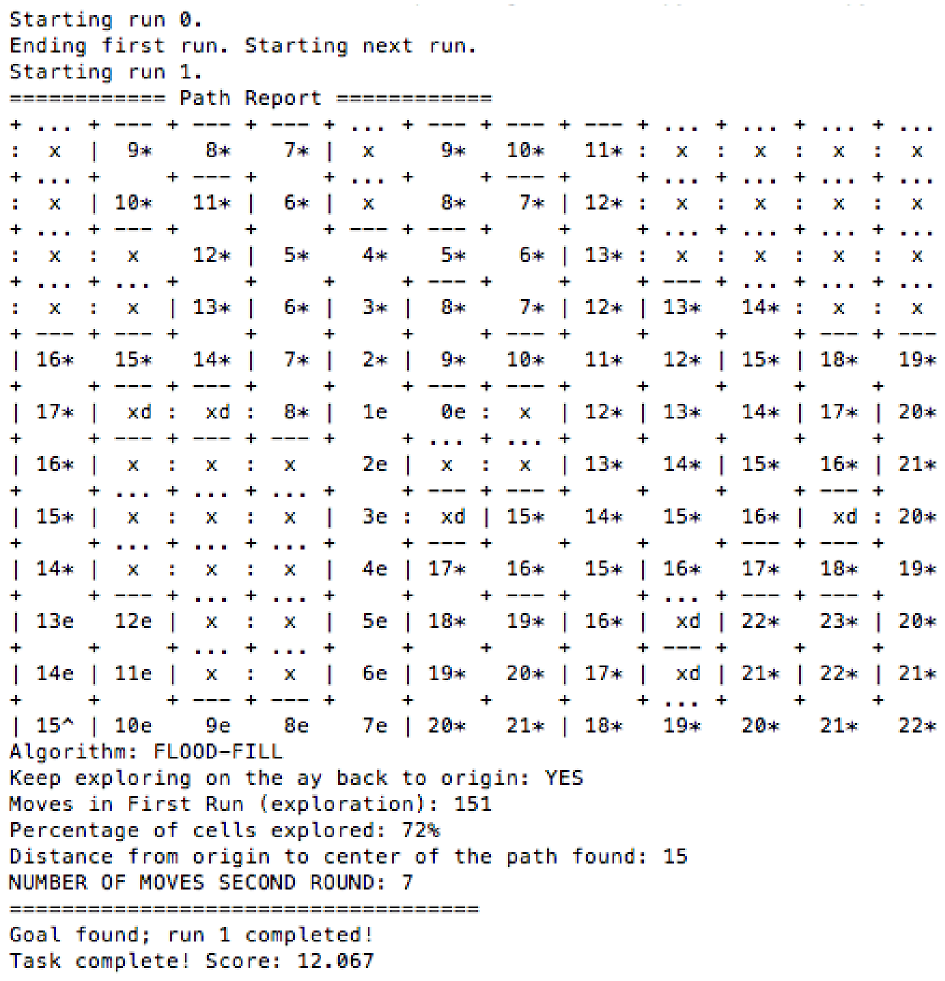
Fig. 13. Test_Maze_04: Optimal and Non-optimal Paths



```
Starting run 0.
Ending first run. Starting next run.
Starting run 1.
============ Path Report ============
+ ... + --- + --- + --- + ... + --- + --- + --- + ... + ... + ... + ...
:  x  |  9*    8*    7* |  x     9*   10*   11* :  x  :  x  :  x  :  x
+ ... +     + --- +     + ... +     + --- +     + ... + ... + ... + ...
:  x  | 10*   11* |  6* |  x     8*    7* | 12* :  x  :  x  :  x  :  x
+ ... + --- +     +     + --- + --- +     +     + ... + ... + ... + ...
:  x  :  x    12* |  5*    4*    5*    6* | 13* :  x  :  x  :  x  :  x
+ ... + ... +     +     +     + --- +     +     + --- + ... + ... + ...
:  x  :  x  | 13* |  6* |  3* |  8*    7* | 12* | 13*   14* :  x  :  x
+ --- + --- +     +     +     +     + --- +     +     +     + --- + ---
| 16*   15*   14* |  7* |  2* |  9*   10*   11*   12* | 15* | 18*   19*
+     + --- + --- +     +     + --- + --- +     +     +     +     +
| 17* |  xd :  xd :  8* |  1*    0e :  x  | 12* | 13*   14* | 17* | 20*
+     + --- + --- + --- + ... + ... + ... +     +     +     +     +
| 18* |  x  :  x  :  x  :  x  :  x  :  x  | 13*   14* | 15*   16* | 21*
+     + ... + ... + ... + ... + ... + --- +     +     +     + --- +
| 19* |  x  :  x  :  x  :  x  :  x  | 15*   14*   15*   16* |  xd : 20*
+     + ... + ... + ... + ... + --- +     +     +     + --- + --- +
| 20* |  x  :  x  :  x  :  x  | 17*   16*   15* | 16*   17*   18*   19*
+     + ... + ... + ... + ... +     + --- +     + ... + --- + --- +
| 21* :  x  :  x  :  x  :  x  | 18*   19* | 16* |  xd | 22*   23* | 20*
+     + ... + ... + ... + ... +     +     +     + --- +     +     +
| 22* |  x  :  x  :  x  :  x  | 19*   20* | 17* |  xd | 21* | 22* | 21*
+     + ... + ... + ... + ... +     +     +     + ... +     +     +
| 23^ |  x  :  x  :  x  :  x  | 20*   21* | 18*   19*   20*   21*   22*
Algorithm: FLOOD-FILL
Keep exploring on the ay back to origin: NO
Moves in First Run (exploration): 135
Percentage of cells explored: 64%
Distance from origin to center of the path found: 23
NUMBER OF MOVES SECOND ROUND: 12
====================================
Goal found; run 1 completed!
Task complete! Score: 16.533
```

If the robot is not allowed to explore after reaching the center. The path found has length 23, which requires 12 moves. The final score is 16.533.

However, if the robot is allowed to explore more cells after reaching the center, things will change. Let's check the new simulation result:



```
Starting run 0.
Ending first run. Starting next run.
Starting run 1.
============ Path Report ============
+ ... + --- + --- + --- + ... + --- + --- + --- + ... + ... + ... + ...
:  x  |  9*    8*    7* |  x     9*   10*   11* :  x  :  x  :  x  :  x
+ ... +     + --- +     + ... +     + --- +     + ... + ... + ... + ...
:  x  | 10*   11* |  6* |  x     8*    7* | 12* :  x  :  x  :  x  :  x
+ ... + --- +     +     + --- + --- +     +     + ... + ... + ... + ...
:  x  :  x    12* |  5*    4*    5*    6* | 13* :  x  :  x  :  x  :  x
+ ... + ... +     +     +     + --- +     +     + --- + ... + ... + ...
:  x  :  x  | 13* |  6* |  3* |  8*    7* | 12* | 13*   14* :  x  :  x
+ --- + --- +     +     +     +     + --- +     +     +     + --- + ---
| 16*   15*   14* |  7* |  2* |  9*   10*   11*   12* | 15* | 18*   19*
+     + --- + --- +     +     + --- + --- +     +     +     +
| 17* |  xd :  xd :  8* |  1e    0e :  x  | 12* | 13*   14* | 17* | 20*
+     + --- + --- + --- +     + ... + ... +     +     +     +
| 16* |  x  :  x  :  x     2e |  x  :  x  | 13*   14* | 15*   16* | 21*
+     + ... + ... + ... +     + --- + --- +     +     +     + --- +
| 15* |  x  :  x  :  x  |  3e :  xd | 15*   14*   15*   16* |  xd : 20*
+     + ... + ... + ... +     + --- +     +     +     + --- + --- +
| 14* |  x  :  x  :  x  |  4e | 17*   16*   15* | 16*   17*   18*   19*
+     + --- + ... + ... +     +     + --- +     + ... + --- + --- +
| 13e   12e |  x  :  x  |  5e | 18*   19* | 16* |  xd | 22*   23* | 20*
+     +     + ... + ... +     +     +     +     + --- +     +     +
| 14e | 11e |  x  :  x  |  6e | 19*   20* | 17* |  xd | 21* | 22* | 21*
+     +     + --- + --- +     +     +     +     + ... +     +     +
| 15^ | 10e    9e    8e    7e | 20*   21* | 18*   19*   20*   21*   22*
Algorithm: FLOOD-FILL
Keep exploring on the ay back to origin: YES
Moves in First Run (exploration): 151
Percentage of cells explored: 72%
Distance from origin to center of the path found: 15
NUMBER OF MOVES SECOND ROUND: 7
====================================
Goal found; run 1 completed!
Task complete! Score: 12.067
```

From the report, we can find that the optimal path is found by the robot on it way back to the origin. The new path has length 15 which only requires 7 moves. The final score is 12.067, which is a big improvement.

XI. Improvement

In this project, all the moves and turns are considered in the discrete case. If the scenario took place in a continuous domain, things will be very different. For example, each square has a unit length, walls are 0.1 units thick, and the robot is a circle of diameter 0.4 units. In this case, the sensor reading would be some real numbers, which are not perfect. We may need take floor or ceiling function to quantize the input. Also we may need to consider continuous speed and ration of the robot. I also checked some videos on the Internet, if there is a zigzag path in the maze, the robot may be able to go through that path diagonally(in the middle of zigzag path). This will definite improve the performance, since making turns will always taking more time.

XII. Conclusions and Reflections

In this project, we implement the Flood-Fill Algorithm to navigate the robot in a virtual maze. The Flood-Fill algorithm is then shown to be a very efficient way to solving this problem, which gives a solution that is very closed to the optimal one.

In the Flood-Fill algorithm, the robot calculate the distances of the open adjacent cells and move to the right direction. Putting a “virtual” wall in front of the dead ends and preferring to the unvisited cells improve the performance of the robot.

To implement the first two algorithms, “Right-First” and “New-First” algorithms is relative simple. The most challenge part is to implement the “Flood-Fill” algorithm. Each time, when the robot move, we have to update the distance of the related cells. I spend quite some time to write the updating codes with Stack method. Even though, the algorithm is straight forward, to realize it is not a easy work.

Overall, it is a very interesting problem. I really enjoy it and feel proud that I make it to the end. The most interesting part is to refine the robot, once a basic implementation of the robot that successfully reaches the center was finished. It is very excited to see a little refinement improve a lot in the final results.

XIII. Acknowledgment

During the project, I reference a lot of materials online, which are not limited to [1]–[6]. Thanks a lot for those Blog writers and the open-source developers on GitHub. I also want to thank the Udacity mentors for their grading and advice.

References

[1] M. K. Thangavelu, “Micromouse: Maze solving algorithm.” `https://madan.wordpress.com/2006/07/24/micromouse-maze-solving-algorithm/`.

[2] Wikipedia, “Micromouse.” `https://en.wikipedia.org/wiki/Micromouse`.

[3] M. Online, “History of micromouse.” `http://www.micromouseonline.com/micromouse-book/history/`.

[4] D. M. Willardson, “Analysis of micromouse maze solving algorithm,” *Learning from Data*, 2001.

[5] R. Villatoro, “Plot and navigate a virtual maze.” `https://github.com/RodrigoVillatoro/machine_learning_projects/tree/master/capstone`.

[6] E. Minnett, “Plot and navigate a virtual maze.” `https://github.com/eminnett/ml-nanodegree-capstone/`.