Rodrigo Villatoro

Plot and Navigate a Virtual Maze

I. Definition

Project Overview

This project takes inspiration from Micromouse competitions, where a robot starts at the lower left corner of a maze and is tasked with a) finding the center of the maze, and b) reaching the center in the fastest possible way. Instead of a physical maze and a physical robot, this project implements a virtual maze and a virtual robot.

Problem Statement

Given a maze of n x n dimensions —where n is an even number between 12 and 16— and starting coordinates of (0, 0) —the lower left corner—, a robot is tasked with two objectives:

> ROUND 1: The robot can navigate the maze to determine the most efficient path to the center. The robot has to reach the center of the maze in order to be able to start the second round, but can alternatively continue to explore the maze after finding the goal.
>
> ROUND 2: The robot attempts to reach the center of the maze in the fastest possible way. In the case of this project, that means reaching the center in the minimum number of steps.

In total, the robot hast a maximum of 1000 time steps to complete both rounds.

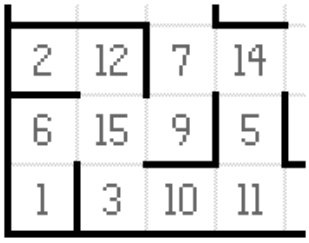
Metrics

The score of an implementation is calculated by adding the number of steps required to complete the second round, plus 1/30 times the number of steps taken during the first round. In other words, the lowest the score, the better. The worst possible outcome: exceeding the 1000 time steps given to complete both rounds.

II. Analysis

Data Exploration

a) The **maze** consists of grid of n x n, where n is an even number between 12 and 16, inclusive. The maze's perimeter is surrounded by walls, and the goal lies in a 2 x 2 square the center of the maze. Inside the maze, between any two cells, there might also be walls blocking the path of the robot, but it is guaranteed that there exists at least one path that leads to the center of the maze. The maze is provided via a text file which contains the dimensions of the maze, as well as information for each wall by using 4-bit values: 0 means there is a wall, 1 if there is no wall; the 1's corresponds to the upwards-facing die, 2's to the right side, 4's to the bottom side, and 8's to the left side. For example, 15 represents a cell with no walls (1*1 + 1*2 + 1*4 + 1*8).



b) The **robot** starts at the lower left corner of the maze, facing up. This corner is guaranteed to have a wall on the left, bottom, and right side, and no wall forward facing up. To navigate the maze, the robot has three obstacle sensors mounted on the left, front, and right side. The sensors detect the number of open squares in the direction of the sensor. The robot has the option of rotating -90, 0, or 90 degrees and then moving up to a maximum of 3 steps forward or 3 steps backward [-3, 3]. If the robot hits a wall the movement is stopped

Exploratory Visualization

There are three different test mazes provided to run the initial analysis. After studying them, we notice that in order to come up with an efficient solution, there are some pitfalls that must be avoided:

1) **Dead ends:** locations in the maze that have three walls, meaning that if the robot reaches one of these locations, the only possible move is back. In order to implement an efficient algorithm, dead ends must be identified as such, to prevent the robot from visiting them in the future. It is important to note that not only should the cell with three walls must be marked as dead end, but also the path leading to that cell, for example, location (8, 11) in the pictures below.

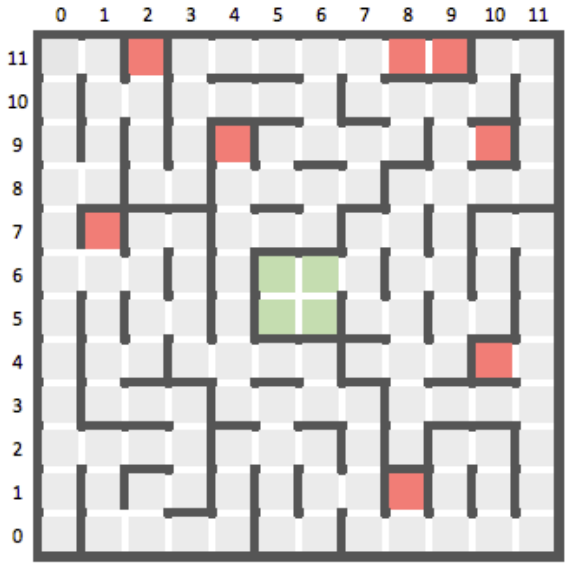
2) **Cycles:** cycles are loops inside the maze, in other words, paths that could potentially lead the robot in circles, preventing it from reaching the goal or causing it to take a long path when a shorter one is available (i.e. going forward instead of entering in the cycle and ending in the previous location).

There are also some considerations that must be given attention:

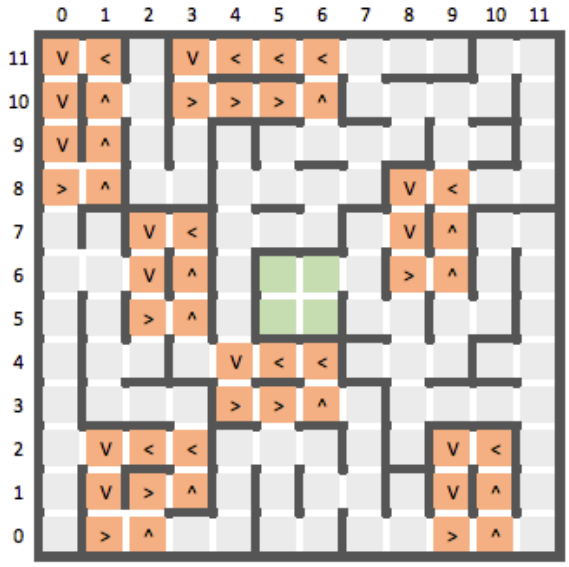
1) **There may be more than one solution to the maze.** The first solution found is not necessary the best, as there may be others with fewer steps.

2) **Even if the robot finds two solutions that have the same distance to the center, one of those solutions may not be optimal.** For example, in the mazes below, the distance to the center of the paths is both 30, but since the robot can move forward up to 3 cells in one move, the optimal path will be the one with more forward directions and less turns. Specifically, in *test_maze_01.txt*, the optimal solution presented on the image below (lower left) is the one that results in 17 steps in the last round, versus an alternative solution that shares the same distance to the center from the initial location, but that requires a total of 21 steps to reach the center (lower right).

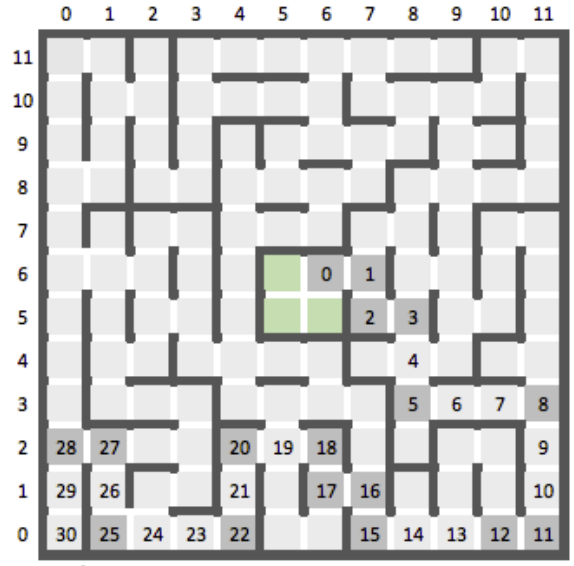
MAZE 01: DEAD ENDS



MAZE 01: CYCLES

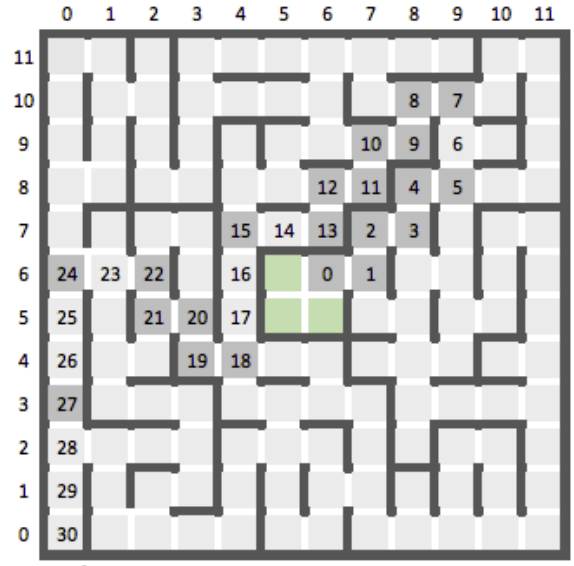


MAZE 01: OPTIMAL SOLUTION



# of steps: 17

MAZE 01: NOT OPTIMAL



# of steps: 21

Algorithms and Techniques

ALWAYS RIGHT

The simplest algorithm implemented in this project was the “Always-Right” algorithm. In this case, as the name of the algorithm implies, the robot will always take a right turn if the option is available. This works if there are no cycles, which is not the case in any of the three mazes.

The use of this algorithm is not recommended at all as it was only implemented for testing and benchmarking purposes.

MODIFIED RIGHT

The modification that this algorithm makes is that it introduces a new condition: the robot prefers unvisited cells to visited cells. In other words, if there is a right turn available, but taking that turn will take the robot to a cell that was previously visited, the robot will look for unvisited cells in the other directions. If there are any unvisited cells available, the robot will move in this direction, otherwise it will still move toward visited cells.

This change enables the robot not only to reach the center of the maze, but to perform relatively well in the second round. The reason for performing well in the second round is that, by following this algorithm, the robot is able to explore a big area of the maze. The downside of this is that the robot explores the maze without any real logic, except trying to keep next to a right wall and trying to avoid previously visited cells.

When the robot reaches the center, the distances of the visited cells are updated in a similar fashion to the flood-fill algorithm (by checking/updating distances of adjacent cells –refer to the section below), to find the most direct path from the corner to the center of the maze.

FLOOD-FILL

This is the most robust algorithms of the three implemented in this project. It works like this:

1) Starting from the center and, assuming 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. Like so:

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

2) The value of each cell must be 1 + the minimum value of the open adjacent cells. “Open adjacent cell” means adjacent cells that are not separated by a wall.
3) Each time the robot moves through the maze it may discover new walls. This will trigger the distances to be updated. It is very important to check not only the distance that the robot is standing on, but also its adjacent distances (and the distances adjacent to them, and so on), potentially updating the distances of the whole maze. This can be achieved through recursion, or by creating a stack of adjacent distances (the second one is considered a better solution for Micromouse competitions as its calculation is more efficient).
4) The robot will attempt to move to the cell that has the minimum distance to the center. In the implementation of this algorithm for this project, the robot will:
    a. Try to move to the cell with the minimum distance (giving preference to cells that have not been visited).
    b. If two adjacent cells have the same distance to the center, the robot will prefer to visit the cell that is facing forward, as this will yield better results in the second round.
5) In this particular implementation, the robot also takes note of dead ends and paths that lead to dead ends. It marks them as visited and places imaginary walls on them to prevent from visiting them again (more on this later).
6) The robot follows these steps until it reaches the center of the maze, at which point all the cells that have not been visited are marked as inaccessible (visiting them in the second round can be risky, since the robot does not know exactly where they will lead), and the distances are updated one last time to obtain the final, direct path to the center of the maze.

Benchmark

The three different algorithms described above provide scores to compare the results. *Always-Right* is the most basic of all algorithms implemented, which provides a very inefficient result. *Modified-Right* and *Flood-Fill* implementations provide more efficient solutions.

In any case, since the robot is able to move up to 3 spaces at the time, we should expect the score to be less than the shortest distance from the start to the center. For example, in MAZE 01 the shortest path to the center has 30 cells. I would expect that the score for most efficient solution (including the first and second round) does not exceed these 30 points.

III. Methodology

Data Preprocessing

No data preprocessing was needed for this project: the robot's tuning is perfect and the information form the sensors is always reliable.

Implementation

The implementation consists of four different classes: cell, terrain, robot, algorithms.

(A) CELL

Represents a square (location) in the maze.

```
class Cell:
    """
    Represents any of the cells in a maze (it has walls, a distance to the
    center, and information on whether it has been visited before by the
    robot or not).
    """

    def __init__(self, real_walls=None, distance=0, visited=''):
        """
        :param real_walls: array of 0's (no walls) and 1's (walls) [l, u, r, d]
        :param distance: distance from that cell to the center of the maze
        :param visited: str; directions = < ^ > V; x = dead end, * = visited
        """
        if real_walls is None:
            real_walls = [0, 0, 0, 0]

        self.real_walls = real_walls  # Actual walls on the maze
        self.imaginary_walls = [0, 0, 0, 0]  # Walls added to avoid dead ends
        self.distance = distance  # Distance to the center of the maze
        self.visited = visited  # Used to know if robot has visited the cell
```

It is important to mention that a cell can have real walls (those that are provided by the maze and captured by the sensors) and imaginary walls (those that are placed by the robot in order to avoid entering dead ends repeatedly). There is a method inside the cell class that will return all walls, real or imaginary:

```
def get_total_walls(self):
    """
    Get real and imaginary walls for a particular cell
    :return: array of 0s and 1s with real and imaginary walls
    """
    total_walls = [0, 0, 0, 0]
    for i in range(len(self.real_walls)):
        if self.real_walls[i] == 1 or self.imaginary_walls[i] == 1:
            total_walls[i] = 1
    return total_walls
```

A cell has also the ability to represent itself visually, by printing information about its horizontal and vertical walls, it's distance to the center, and visited flags. This is useful when printing the entire maze (as we will see in the next sections).

(B) TERRAIN

The terrain is the representation of the maze from the robot's point of view.

```
class Terrain:
    """
    Used by the robot to create a visual representation of the maze.
    It contains all the logic for the movement (fill maze, update distances,
    etc) as well as methods for debugging the program (i.e. printing the maze).
    """
    def __init__(self, maze_dim):
        self.maze_dim = maze_dim
        self.grid = [[Cell() for i in range(maze_dim)] for j in range(maze_dim)]
        self.fill_distances()
        self.last_visited_cell = None
        self.cells_to_check = []
        self.visited_before_reaching_destination = []
```

In the initialization process, the terrain makes an empty grid with size *maze_dim* x *maze_dim*, and then fills it by following the procedure described in the Algorithm's section (*Flood-Fill*). This information is not necessarily used by all the algorithms during the first round (for example *Always-Right* algorithm completely ignores it), but it is used in all cases by the robot to obtain logical way back to the center during round 2 by updating the distances accordingly. The two most important methods in this class are update and draw.

update is to be called by the robot when it changes location. In this case, the robot passes information regarding its location in the maze, the way it is heading, the walls at that particular location, among others. With this information, the terrain is updated, including the distances of the open adjacent cells, again, by following the procedure described in the algorithms section. As with the case of other algorithms, the updated distances are only relevant during round 1 when using the *Flood-Fill* algorithm, but are left here for practical reasons.

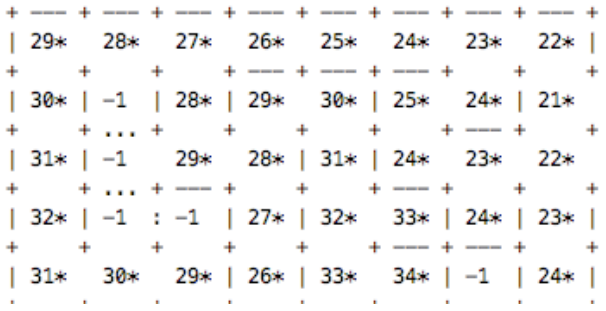
```
def update(self, x, y, heading, walls, exploring):
    """
    Updates the maze's:
    - Real walls (including adjacent wall's)
    - Visual representation of the current cell
    - Visual representation of the previous cell
    - Distances of adjacent cells
    """

    # Get reference to current position
    cell = self.grid[x][y]

    # Store real_walls only if cell has not been visited.
    # Imaginary walls can't change; walls are updated before location.
    if cell.visited == '':
        cell.real_walls = walls
```

```
        # Set adjacent walls (i.e. right wall of cell A is left wall of B)
        self.update_adjacent_walls(x, y, walls, 'real')

    # Change the visual representation of the current cell
    cell.visited = robot_directions[heading]

    # Change visual representation of the previous cell
    self.change_visual_representation_of_prev_cell(cell, exploring)

    # Set last visited cell to this cell
    self.last_visited_cell = cell

    # Update all the distances of the visited cells of the maze
    self.update_distances()
```

draw is used mostly for debugging purposes, to print a visual representation of the maze.

```
def draw(self):
    """
    Prints maze with correct x and y axes.
    """
    mod_terrain = []
    for i in range(self.maze_dim):
        mod_terrain.append([x[i] for x in self.grid])

    print_delimiters = True
    for i, row in enumerate(reversed(mod_terrain)):
        self.print_row_of_cells(row, print_delimiters)
        print_delimiters = not print_delimiters  # Flip value
```

The following image shows a section of a printed maze, and the table below indicates how to understand it.



HOW TO READ A PRINTED MAZE

Variable	Description	Visual representation
Visited Cell	Cells that were visited by the robot during round 1.	x*, where x is the distance to the center of the maze
Unvisited Cell	Cells that were NOT visited by de robot during round 1.	-1
Dead End	Cell in which the only possible move is backwards.	-1
Real Wall	Actual walls on the maze.	--- for horizontal walls | for vertical walls

Imaginary Wall	Walls added by the program, to prevent dead ends or to make unvisited cells inaccessible to the robot in the final round.	: for horizontal imaginary walls … for vertical imaginary walls
Current Direction	Shows where the robot is heading at a particular moment.	< left, ^ up, > right, V down

(C) ALGORITHMS

The Algorithms class very simple, and is to abstract the logic for the robot to decide where to move. In fact, it only has one method, which must be implemented by subclasses). When a robot is determining the next move, it asks for a valid index to move to. The index is based on the robot's coordinates: [left, forward, right, backward].

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

    def get_valid_index(self, adjacent_distances, adjacent_visited):
        raise NotImplementedError('Subclasses need to implement this method')
```

For example, the *Always-Right* algorithm checks first whether there is a wall on the right [2]. If there is no wall in this location, it will return this index so the robot can decide on the number movement and rotation. If there is a wall on the right side, it will try in this order: forward, left, backward.

```
class AlwaysRight(Algorithm):
    def __init__(self):
        super(AlwaysRight, self).__init__()
        self.name = 'always-right'

    def get_valid_index(self, adjacent_distances, adjacent_visited):
        if adjacent_distances[2] != WALL_VALUE:
            valid_index = 2
        elif adjacent_distances[1] != WALL_VALUE:
            valid_index = 1
        elif adjacent_distances[0] != WALL_VALUE:
            valid_index = 0
        else:
            valid_index = 3

        return valid_index
```

The *Modified-Right* algorithm will essentially follow the same logic, except it will prefer to move to unvisited cells rather than previously visited cells. This will lead the robot out of many cycles, but not all of them, as we will later see.

Finally, the *Flood-Fill* algorithm asks the maze for the adjacent cell with the minimum distance. But this implementation tries to make use of the fact that it is better to go straight rather than to zigzag (since the robot can move up to 3 spaces at a time if it's in a straight direction). Therefore, it makes another check to see if there is unvisited cell that lies straight head, so that the robot can visit that location.

This implementation also shows that the robot, during the first round, will prefer unvisited cells rather than visited cells, the reason is so that a larger portion of the maze is explored before reaching the center of the maze.

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

    def get_valid_index(self, adjacent_distances, adjacent_visited):

        # Get min index (guaranteed to not be a wall)
        valid_index = adjacent_distances.index(min(adjacent_distances))

        possible_distance = WALL_VALUE
        for i, dist in enumerate(adjacent_distances):
            # Prefer unvisited cells
            if dist != WALL_VALUE and adjacent_visited[i] is '':
                if dist <= possible_distance:
                    valid_index = i
                    smallest_distance = dist
                    possible_distance = smallest_distance
                    # Index 1 is for the forward direction (the fastest)
                    if valid_index == 1:
                        break

        return valid_index
```

(D) ROBOT

Finally, the robot class is responsible for the agent moving around the maze. The most important method of this class is *get_next_move*, which among other things, gives a rotation and movement to the robot, depending on whether is looking to:

a) Explore (still first round but continues to move after reaching destination)
b) Move optimally (if it's the final round, since up to 3 steps can be taken)
c) Move around the maze in the first round, depending on the algorithm (and using the *get_valid_*index method described above).

```
def get_next_move(self, x, y, heading, sensors):

    if self.reached_destination and self.exploring:
        rotation, movement = self.explore(x, y, heading, sensors)
        self.steps_exploring += 1

    elif not self.reached_destination and not self.exploring:

        if self.algorithm.name == 'flood-fill' and self.is_at_a_dead_end(sensors):
            rotation, movement = self.deal_with_dead_end(x, y, heading)

        else:

            adj_distances, adj_visited = self.terrain.get_adj_info(
                x, y, heading, sensors)
            valid_index = self.algorithm.get_valid_index(adj_distances,
```

```
adj_visited)
            rotation, movement = self.convert_from_index(valid_index)

        self.steps_first_round += 1

    else:
        # Final round (optimized movements)
        if self.steps_final_round == 0:
            print('******* FINAL REPORT *******')
            self.terrain.draw()
        rotation, movement = self.final_round(x, y, heading, sensors)
        self.steps_final_round += 1

    return rotation, movement
```

Refinement

The three algorithms described above (*Always-Right, Modified-Right, Flood-Fill*) show the sequential improvement as project advanced. Besides, some other important variables and improvements were introduced, reflected in some way in one or all the algorithm:

1) **Visited vs unvisited cells during round one**. With the *Always-Right* algorithm I confirmed the fact that the robot could go in an endless loop if it only tried to advance as long as it had a wall on its right, otherwise it would turn right if that option was available. As we will see later, this strategy would fail except on the simplest of mazes. One easy way to avoid this, implemented in the *Modified-Right* algorithm was to follow that same logic (going forward if there was a wall on the right, otherwise turning or going back) as long as the cell had not previously been visited. This prevents the robot from getting stuck in most situations, but there is still a risk of getting stuck in a cycle where all the adjacent cells have already been visited and there is always a right turn available.

   Both the *Modified-Right* implementation and the *Flood-Fill* implementation prefer to go to unvisited cells rather than visited cells, in the case of the *Modified-Right* to avoid simple cycles, and in the case of the *Flood-Fill* is used as a way to get more cells of the maze explored.

2) **Dead ends.** One of the problems tackled early on where the dead ends: locations that have 3 walls (left, forward, right). During the first round, the robot would sometimes visit these same locations over and over again, although no new information could come from such an action. The solution to this problem was to place “imaginary walls” to signal the robot that these locations were out of bounds. This solution is only implemented in the *Flood-Fill* algorithm.

3) **Exploration (during round one).** After the robot has reached the center of the maze, it can continue to explore to determine if there is a better solution. An “explore” algorithm was implemented (basically it tries to move to adjacent cells that have not been visited), but in the three mazes provided, this resulted in worst scores. Nevertheless, as we will see further on, there may be some cases where exploring might actually improve the score considerably.

4) **Unwanted exploration (during round two).** After finding a path to the center, updating the distances of all the cells (*Flood-Fill* algorithm), and finishing round one, the robot was ready to start round two. In round two, I noticed that the robot sometimes moved around parts of the maze that

had not been explored during round one. The robot took these actions since it believed the distances in these unexplored paths there were shorter, when in fact were full of (previously unknown) walls. In order to fix this, after reaching the center, the robot places imaginary walls on all unvisited cells. This effectively prevents the robot from visiting those locations during round 2.

5) **Number of moves during round two.** One of the important rules is that the robot is allowed to advance up to 3 spaces per move, if the movement is done in the same direction. In order to have a good score, the robot must make the most out of this rule, especially during round 2, where each move has a cost of 1. I implemented a solution in the *Flood-Fill* algorithm that prefers to explore unvisited cells in front of the robot (forward direction), to try to maximize the number of long steps the robot can take during round 2, improving the score of the algorithm.

IV. Results

Model and Evaluation

A script called tester.py has been provided to test the mazes and the algorithms. We can run the script by writing the following command in the console:

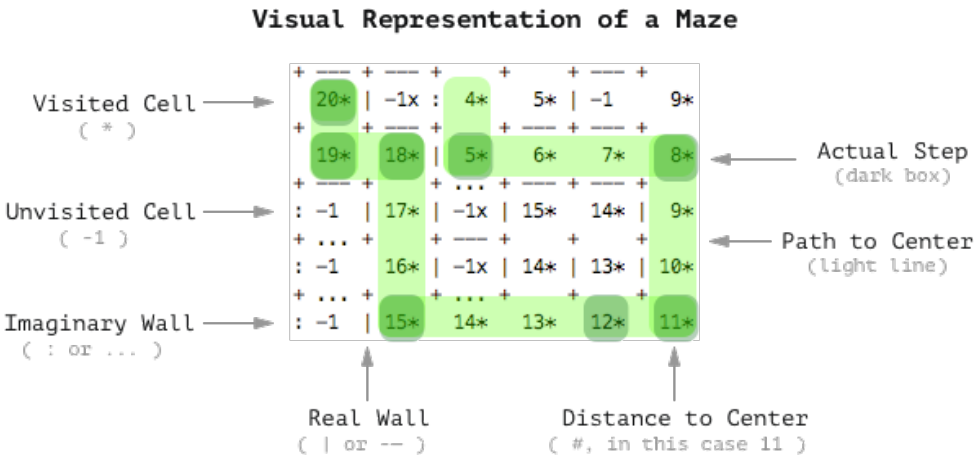
```
python tester.py test_maze.01 ff false
```

The third argument is the maze we want to test, the fourth argument is the algorithm we want to use, and the fifth argument states if the robot should explore more cells after reaching the center. These are the options for each argument

- ***(arg # 3):*** *test_maze_01.txt, test_maze_02.txt, test_maze_03.txt, test_maze_04.txt*
- ***(arg # 4):*** *ff = Flood-Fill, ar = Always-Right, mr = Modified-Right*
- ***(arg # 5):*** *true, false*

An analysis and robustness of each algorithm is provided in the section below, first as a detailed analysis for Maze 01, and then as a summary of statistics for all the three mazes.

For the visual representation, besides the description in the table HOW TO READ A MAZE (above), colors were added to show the path that the robot follows during second round (our round 1, if the robot never reached the center), and the actual steps the robot took during round 2.

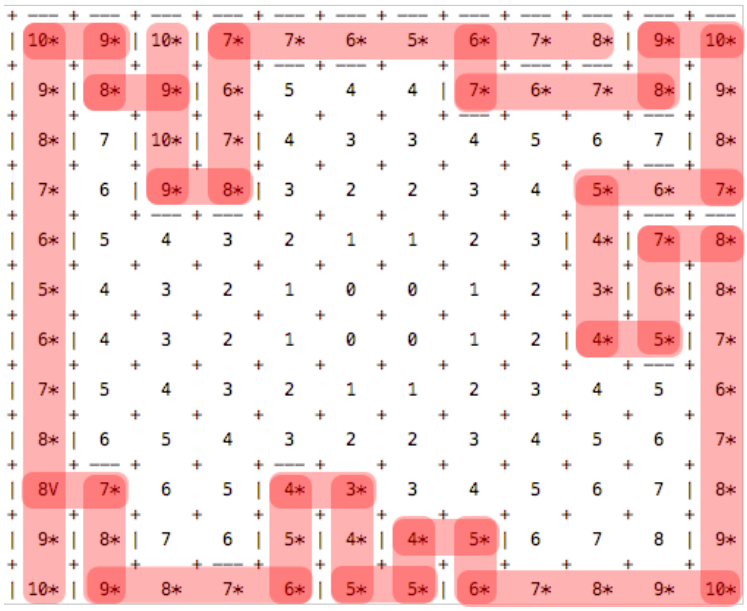


ALWAYS-RIGHT

For the maze provided in *test_maze_01*, there robot gets stuck in the infinite loop that is shown by the red path in the image below. The result: the robot never reaches the center, so the allotted time gets exceeded. This is the worst possible outcome an algorithm can have.

Always-Right, never reaches the center

test_maze_01.txt => score: n/a

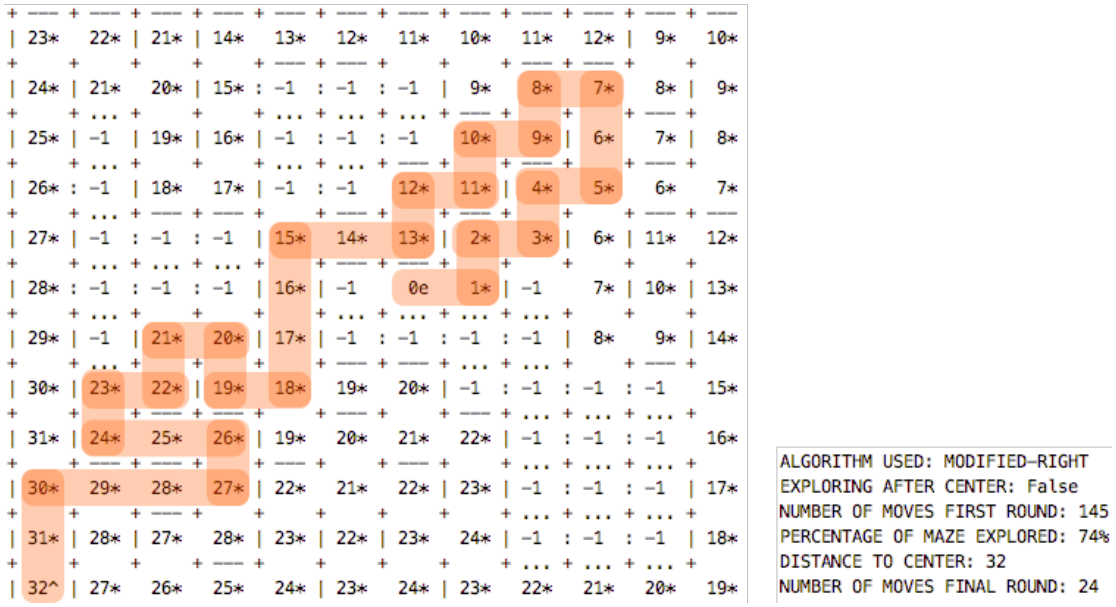


MODIFIED-RIGHT

When using this algorithm, the robot seeks to always have a wall on the right, but it has a strong preference for moving towards unvisited cells (even if that means going forward or left). The shortest distance the robot could find for this maze was 32. The path that the robot followed in the second round is colored in light orange, but the steps the robot actually took are represented by darker orange (since a robot can move to a maximum or 3 spaces at a time). The total number of steps needed on the second round where 24 and the final score 28.867.

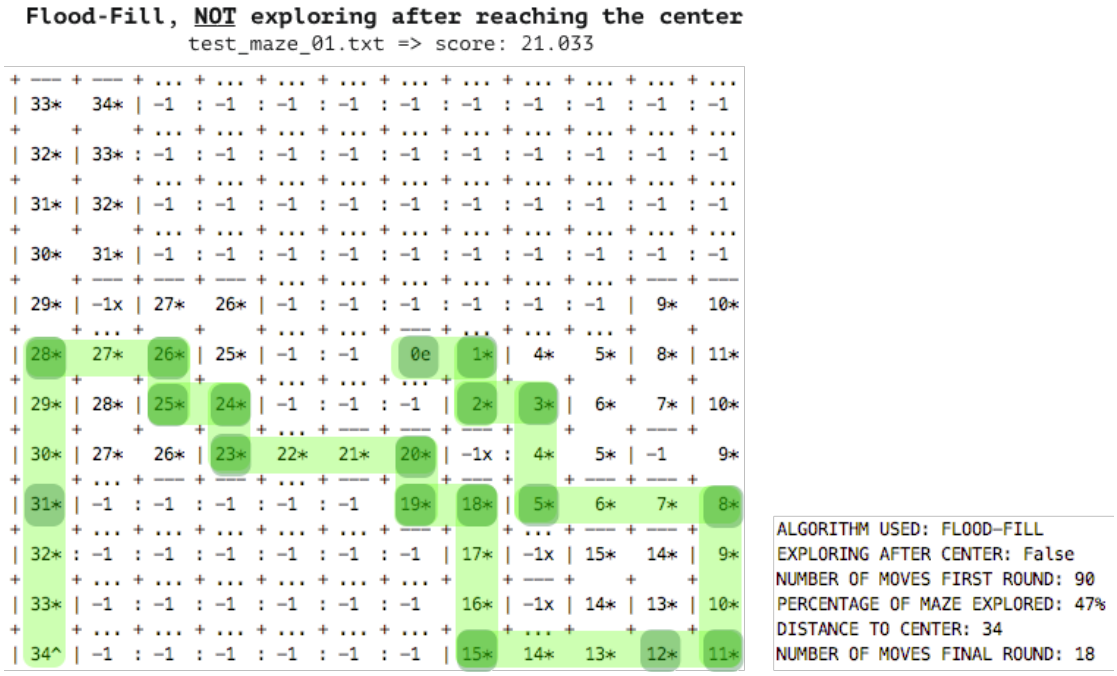
Mod-Right, NOT exploring after reaching the center

test_maze_01.txt => score: 28.867



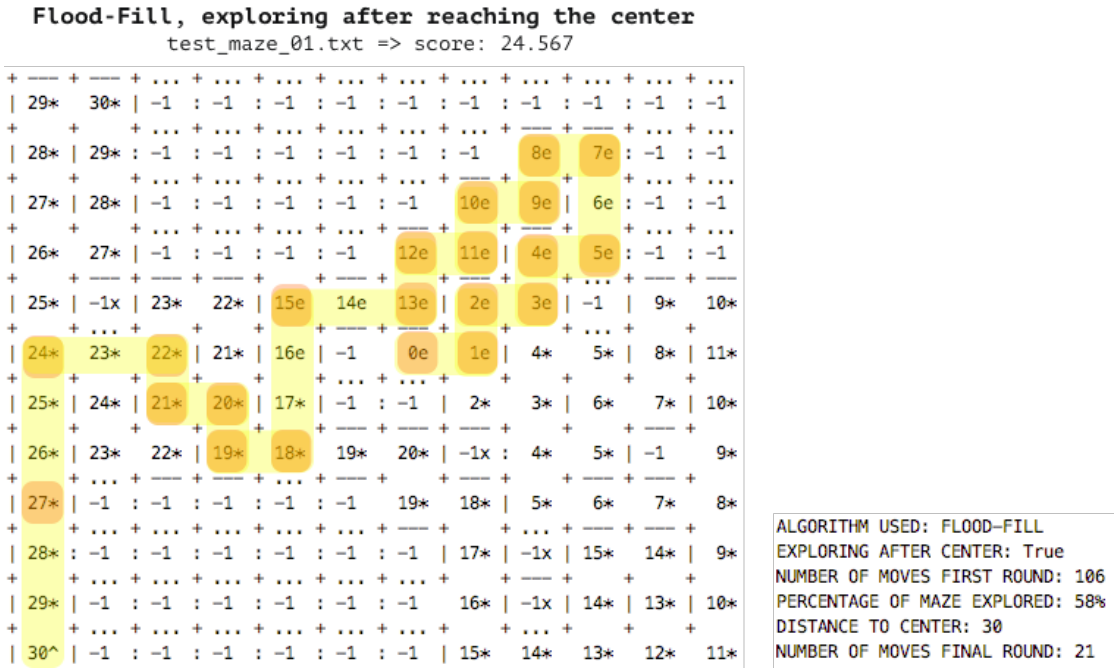
FLOOD-FILL, NOT EXPLORING

This is the algorithm that performed the best in the three mazes provided for testing. The interesting point about this algorithm is that even though the distance to the center is 2 cells more than the previous algorithm (34 vs 32), the robot is able to reach the center in only 18 steps vs 24 steps, representing a great improvement. The reason for finding this path is that the robot prefers to cells that are forward, rather than those on the left side or right side. The score of this algorithm was 21.033, almost 8 points less than the *Modified-Right* algorithm.



FLOOD-FILL, EXPLORING

From our analysis in Section II, we know that neither 34 nor 32 are the shortest distances to the center of the maze. By giving the robot orders to explore the maze after reaching the center of the maze we are able to find one path that has a total distance of 30. The interesting thing about this is that that path is full of turns, so even though it may be the shortest path of the different algorithms, it is not the fastest, and thus should be avoided. In fact, in the three mazes, exploring was never a good option, as it only increased the number of moves taken, hardly finding any improvement. As we will see later, this is not always the case.



The tables below summarize the results of all the algorithms. In short, *Always-Right* never makes it to the center. *Modified-Right* at first seemed like a good candidate for an efficient algorithm, as it the explores the maze in such a way that it covers a lot of ground in a relatively effective way, but fails to reach the center of Maze 03 by entering an endless loop; it also fails to consider the fact going forward is faster than turning right. The best results come from the *Flood-Fill* algorithm, which consistently reaches the center of the maze and yields the bests results.

MAZE 01

	Algorithm	Moves	% Explored	Distance to Center	Moves	Score
Exploring	Always Right	n/a	n/a	n/a	n/a	Time exceeded
	Mod. Right	151	76%	32	24	29.067
	Flood-Fill	106	58%	30	21	24.567
Not Exploring	Always Right	n/a	n/a	n/a	n/a	Time exceeded
	Mod. Right	145	74%	32	24	28.867
	Flood-Fill	90	47%	34	18	21.030

MAZE 02

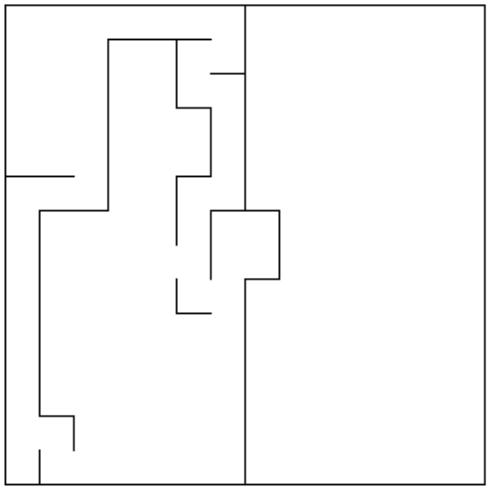
	Algorithm	Moves	% Explored	Distance to Center	Moves	Score
Exploring	Always Right	n/a	n/a	n/a	n/a	Time exceeded
	Mod. Right	209	77%	43	27	34.000
	Flood-Fill	239	87%	43	25	33.000
Not Exploring	Always Right	n/a	n/a	n/a	n/a	Time exceeded
	Mod. Right	196	72%	45	30	36.567
	Flood-Fill	239	87%	43	25	33.000

MAZE 03

	Algorithm	Moves	% Explored	Distance to Center	Moves	Score
Exploring	Always Right	n/a	n/a	n/a	n/a	Time exceeded
	Mod. Right	n/a	n/a	n/a	n/a	Time exceeded
	Flood-Fill	125	41%	51	25	29.200
Not Exploring	Always Right	n/a	n/a	n/a	n/a	Time exceeded
	Mod. Right	n/a	n/a	n/a	n/a	Time exceeded
	Flood-Fill	109	35%	51	25	28.667

V. Conclusion

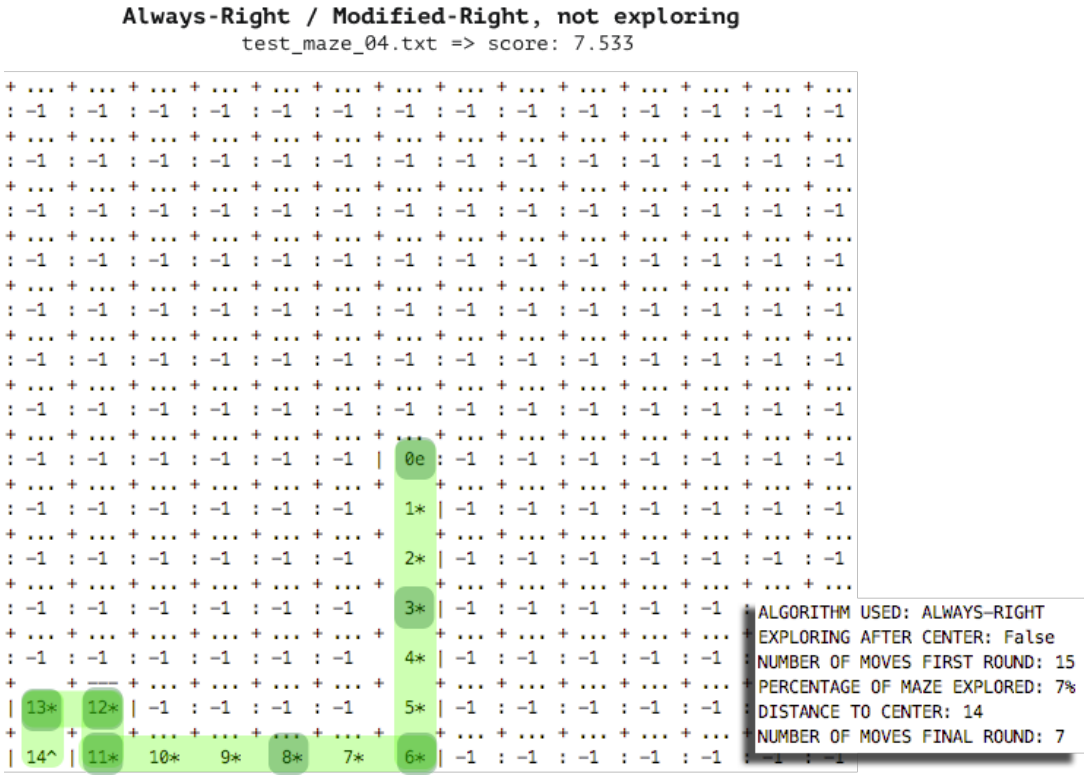
Free-Form Visualization



The *Flood-Fill* algorithm, as it is implemented, has a weak point: if the fastest path is found by taking an early right (instead of going forward), this algorithm may never discover it, unless the robot explores the maze after reaching the center (something that could be expensive as it would need to backtrack on some visited cells, not gaining any new information). To demonstrate this concept, I created a very simplistic maze:

ALWAYS RIGHT / MODIFIED RIGHT

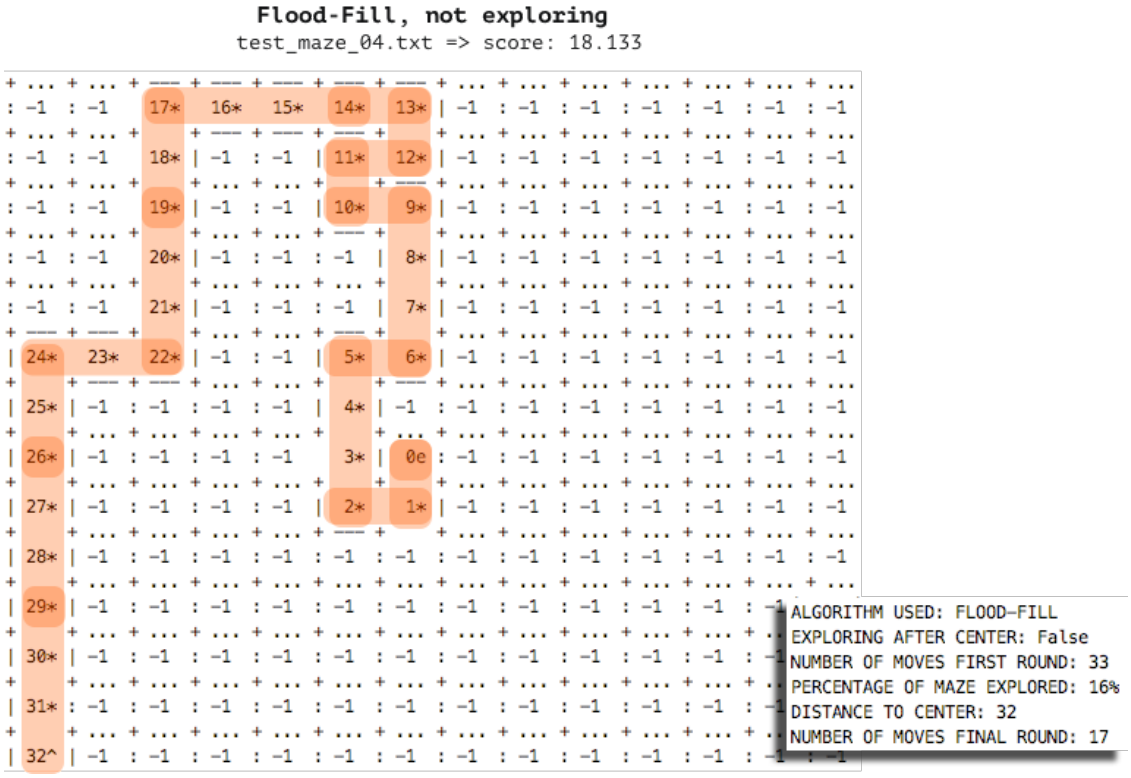
In this maze, the best score is achieved by both *Always-Right* and *Modified-Right* algorithms, as shown in the following picture (score = 7.533).



FLOOD-FILL, NOT EXPLORING

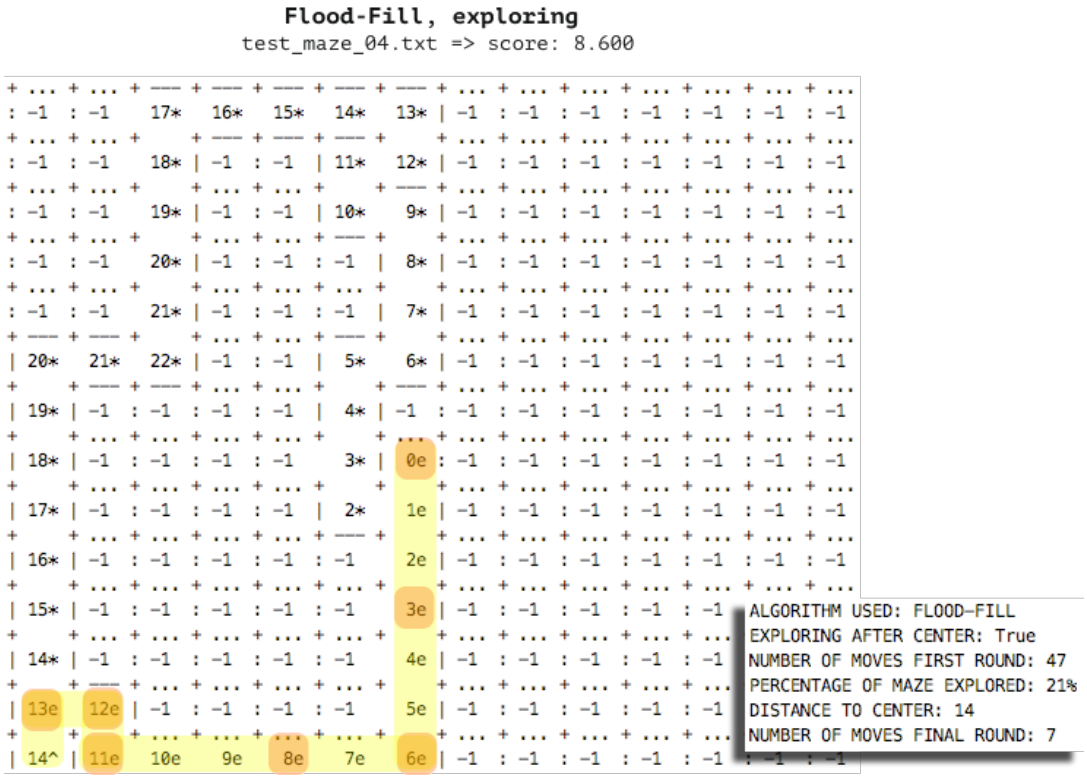
The *Flood-Fill* algorithm, with its current implementation, prefers to go forward instead of taking that early right. It eventually finds a way to the center of the maze, but it's not the optimal path. Since its explore flag

is set to False, it ends the first round immediately after reaching the center of the maze. The final score: 18.133.



FLOOD-FILL, EXPLORING

This is a case where exploring the maze after reaching the center will yield its reward. The robot is able to find the shortest path (cells marked with an ‘e’), and thus follow it in the second round. Since it only found this path after exploring the north-side of the maze, it’s score (8.600) is not as good as the *Always-Right* algorithm, but it’s a big improvement versus *Flood-Fill* without exploring.



Reflection

The best way to solve the problem (finding the fastest path to the center of the maze in an efficient manner), was to use the *Flood-Fill* algorithm. By calculating the distances of the open adjacent cells, the robot is able to move in the right direction at all times. Taking notes of dead ends and preferring unvisited cells that lie straight ahead (instead of left or right), are all strategies that can improve the result of the robot.

The implementation of the *Flood-Fill* algorithm was a challenge in itself, as there is little documentation available on the details of how to approach it. Although there are many visualizations that show how a robot that is following this algorithm moves through the maze, there are very few references regarding the calculations.

Besides this, one of the most difficult tasks was to define a correct algorithm to explore the maze after reaching the center. As can be seen the first three test mazes, the exploration only worsened the score, but as shown in Maze 04, there might be cases where exploring results in an improved score.

Improvement

This project and simulation takes place in a discrete domain (robot advances 1 cell, it then gets feedback from sensors, etc.). If the scenario took place in a continuous domain, where the robot advances not by cells but in inches, the robot has certain size (for example, diameter of 0.4 units), walls have a thickness (for example, 0,1 units thick), etc., the model would have to be improved to take into account all this information.

For example, the sensors would have to be pointing in such a way that the measurements are precise and also reliable. Instead of a series of discrete steps, continuous speed and rotation must be taken into account. In addition, if there was a zigzag along the maze, the robot might be able to cover that path diagonally, and not cell by cell, improving considerably the speed of that fragment of the maze.

Developing an efficient algorithm for a virtual maze is by no means an easy feat, but implementing that on a real robot that has to move on a real maze might prove a bigger challenge… one while worth exploring after this project.