The Implementation of a Maze-Solving Robot

Using the Modified Flood-Fill Algorithm

John Henry Sottile

A project presented for the degree of
Master of Science

CalPolyPomona

College of
Engineering

Department of Mechanical Engineering
California State Polytechnic University, Pomona
Pomona, CA
10 December 2019

Signature Page

Project: The Implementation of a Maze-Solving Robot

Author: John Henry Sottile

Date Submitted: 10 December 2019

Approved: _____

John Caffrey, Ph.D.

Project Committee Chair

Department of Mechanical Engineering

Approved: _____

Behnam Bahr, Ph.D.

Department of Mechanical Engineering

Approved: _____

Jawaharlal Mariappan, Ph.D.

Department of Mechanical Engineering

Acknowledgements

I want to thank my advisor, Professor John Caffrey, for his support and guidance during this past fall semester.

I also want to thank Professor Behnam Bahr for encouraging me to apply for the MENTORES (Mentoring, Educating, Networking, and Thematic Opportunities for Research in Engineering and Science) project funded by a Title V, PPOHA (Promoting Post-baccalaureate Opportunities for Hispanic Americans) grant from the U.S. Department of Education.

Last but not least, I want to thank Professor Jawaharlal Mariappan for generously offering to review my project report.

Abstract

In this project, a maze-solving robot was successfully implemented using a Raspberry Pi 3 Model B V1.2. The modified flood-fill algorithm was used to solve the maze by updating the distance value assigned to each maze cell. The robot used two stepper motors and four light sensors to move forward and to confirm when the robot was approaching a wall. The modified flood-fill algorithm was first implemented in Matlab then translated to Python to run on the hardware. The maze was built roughly to the specifications of the IEEE region 6 micromouse competition. The objective was to implement an autonomous, maze-solving robot similar to those used in the micromouse competition. The Python and Matlab code used to control the robot is available to students interested in learning how to physically implement search algorithms. Additional sensors are required to make the robot fully autonomous, i.e. capable of solving the maze without prior knowledge of the wall positions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Maze solving is a millennia-old problem that dates back to antiquity. According to Greek mythology, Theseus solved the Minotaur's labyrinth by marking his path with a string. Today, robots use sensors and algorithms to autonomously navigate through mazes. Autonomous robots are the result of modern advances in digital electronics and computer science. One of the first demonstrations of artificial intelligence was a maze-solving robot built and designed by Claude Shannon of Bell Labs in 1950 [4]. His robotic mouse used metal whiskers and relay switches to detect and record wall positions. The "on" state indicated a wall and the "off" state indicated open space. Shannon's use of binary digits to encode information was revolutionary at the time since digital electronics was still a relatively young technology. Furthermore, the walls were re-configurable so that the robotic mouse could learn to solve the maze by applying an algorithm to the recorded bits. Most importantly,

Shannon proved that a machine could be programmed with the ability to adapt to a changing environment through trial and error.

Chapter 2

Literature Review

This literature review covers the search algorithms that are commonly used to solve mazes. The modified flood-fill algorithm was chosen based on its relative computational efficiency and guarantee to find the shortest path. Others include breadth-first search, Dijikstra's algorithm, greedy best-first search, and A* search.

Breadth-first search is a widely used graph search algorithm. It can be applied to maze solving if a maze is interpreted as a graph with nodes and edges. That is, the nodes are the cells and the edges are the connections between open neighbor cells. A graph is defined here as a set of nodes connected to each other by edges. In this case, a node is a Cartesian point with a value assigned to it. An edge connects any two nodes in the graph. If a graph is weighted, the edge can have a value assigned to it. If a graph is directed, the edge only exists in one direction. That is, the two nodes

in question are only connected in one direction [1]. However, graphs do not have to be weighted or directed. Implementing the breadth-first search algorithm to solve a maze requires constructing a search tree. A tree is a connected, acyclic graph [3]. Root nodes have no parents and leaf nodes have no children. All the other nodes in-between have one parent and at least one child.

10.61	10.83	11.24	12	13.24	14.83
9.16	11.24	11.14	13	13.14	15.24
8	12	16	GOAL	14	16
7.16	5.24	17.41	16	13.41	13.24
4.61	4.83	7.24	8	11.24	12.83
START	6.61	7.16	10	11.16	14.61

Table 2.1: This table shows the sum of the cost and heuristic values for each maze cell. The cost values for each cell account for the maze wall positions used in this project. The heuristic value is the Euclidean distance from each cell to the goal.

The A* search algorithm is a combination of Dijikstra's algorithm and greedy best-first search. It is important to note that Dijkstra's algorithm is a breadth-first search that takes cost into consideration. That is, the edges between nodes have different weights. The greedy best first search uses a heuristic such as the Euclidean or Manhattan distance for optimization. In the case of a maze-solving robot, each cell is assigned a value equal to the sum of the cost and the heuristic. The cost is the distance from the start and the Euclidean distance heuristic, for example, is the distance along the line between the current cell and the goal [2]. The sum of the cost and heuristic

values for each cell in the maze used for this project can be seen in table 2.1. The heuristic is the Euclidean distance. The cost is the number of lateral movements between the start cell and the current cell, taking the maze walls into consideration. The most efficient way to implement this algorithm is to take a picture of the maze layout before the robot moves. This is because finding the shortest path would require moving many times across the maze to all the children nodes on the frontier of the search tree.

The modified flood-fill algorithm is an improvement to the conventional flood-fill algorithm in that it is more computationally efficient. The modified flood-fill algorithm starts with a distance matrix initialized with values corresponding to cells with no wall separation. However, unlike the conventional algorithm, the modified flood-fill algorithm uses the stack data structure to look through distance values of select neighbor cells instead of looping through every cell in the matrix. The first step is to check all the open neighbor cells for the smallest distance value. If the distance value of the current cell minus one is not equal to the smallest value of the open neighbor cells, then the current cell is pushed onto the stack. Next, while the stack is not empty, the cell on the top of the stack is popped. If the distance value of the popped cell minus one is not equal to the smallest value of the open neighbor cells, then the neighbor cells are pushed onto the stack. It is important to note that neighbor cells that are also destination cells cannot be pushed onto the stack. When the stack is empty, the algorithm applies the steps above to the open neighbor cell with the smallest value until a

destination cell is reached [5].

Chapter 3

Background

The IEEE micromouse competition is a popular event where university teams design robots to navigate through a maze in the least time possible. To do this, teams must work together to integrate hardware, software, controls, and mechanical design into a single system. The maze-solving process is broken into distinct exploration and time-trial phases. In the former, the robot uses its sensors to detect the position of the walls as it finds the shortest path to the center of the maze. In the latter, the robot moves along the shortest path to the center as fast as it can. To find a solution, the robot can use a variety of algorithms ranging from the flood-fill algorithm to A* search. All the algorithms find the shortest path with varying levels of computational efficiency. In this project, I chose to use the modified flood-fill algorithm which requires the maze to be decomposed into a grid of square cells. The number of lateral steps between the center of the maze and each cell is stored

as the distance value of that cell. One advantage of the modified flood-fill algorithm over the original flood-fill algorithm is computational efficiency. The modified algorithm is more efficient than the original because it does not need to update distance values each time it reaches a new cell. Instead, it only updates the distance values of relevant neighbor cells [6].

Chapter 4

Implementation

The objective of this project was to program a robot using the Raspberry Pi to autonomously navigate a maze. An image of the completed maze is in figure 4.1. Additionally, I wanted my work to be available to students as a starting point for building a micromouse robot as a club project. Therefore, I built the maze roughly according to the specifications for the IEEE region 6 competition [7]. The base of the maze was made from a $\frac{3}{4}$ inch thick sheet of MDF. Three 29 inch x 29 inch squares were cut out from the sheet. Another 47 $\frac{1}{4}$ inch x 20 inch rectangular piece of the sheet was used to complete the base. This provided enough surface area for a 6 x 6 cell maze since each cell measured 18 centimeters x 18 centimeters to comply with the rules [8]. The maze could be expanded to 8 x 8 cells by adding another square of $\frac{3}{4}$ inch thick MDF. Furthermore, if a full, competition-sized maze were desired, thirteen more squares of $\frac{3}{4}$ inch thick MDF could be added. The maze walls

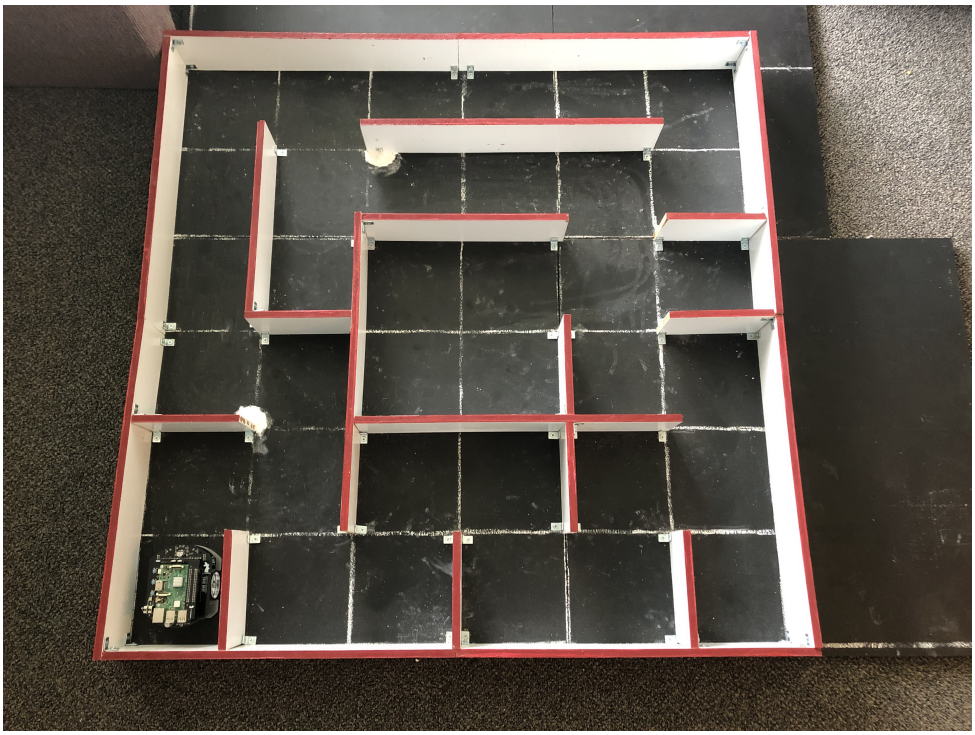

Figure 4.1: Here the maze can be seen with the robot at the start location ready to go. Many test runs were required to determine the optimal sensor threshold value of 500.

were cut from three $\frac{1}{2}$ inch thick sheets of MDF. I chose not to follow the competition maze wall height of 5 centimeters to ensure the sensors could be read on my robot. The LEDs were mounted on the chassis 4 centimeters above the ground. The height was set to 4 inches so that if the competition standard of 5 centimeters or roughly 2 inches were desired, the walls could be simply cut in half. The walls were painted with two coats of white paint and the base was painted with two coats of black paint. To secure the walls to the base, $\frac{3}{4}$ inch braces were fastened down with $\frac{3}{4}$ inch wood screws.

The microcontroller was a Raspberry Pi 3 Model B V1.2. The model name of the sensor on the light-emitting side was optoSupply's OS5RKA5111A. There were four LEDs that emitted red light with a wavelength between 620 nanometers and 630 nanometers. The sensors can be seen in figure 4.2. The model name of the sensor on the receiving side was KDENSHI's ST-1KL3A. The output voltage was an analog output of 0V - 3.3 V. The analog output from the sensors was converted to a digital value by the microchip's MCP3208 analog-to-digital converter. The digital signal was then sent to the Raspberry Pi via SPI communication protocol. In other words, the analog signal ranging from 0V to 3.3V was converted to a discrete value between 0 and 4095 because the analog-to-digital converter had 12-bit resolution. Furthermore, the sensor value displayed was the difference between the analog-to-digital converter value taken before and after the LED emitted light. The sensor value was occasionally negative in cases where the output of analog-to-digital converter was greater before the LED emitted light than it was after. This can be easily

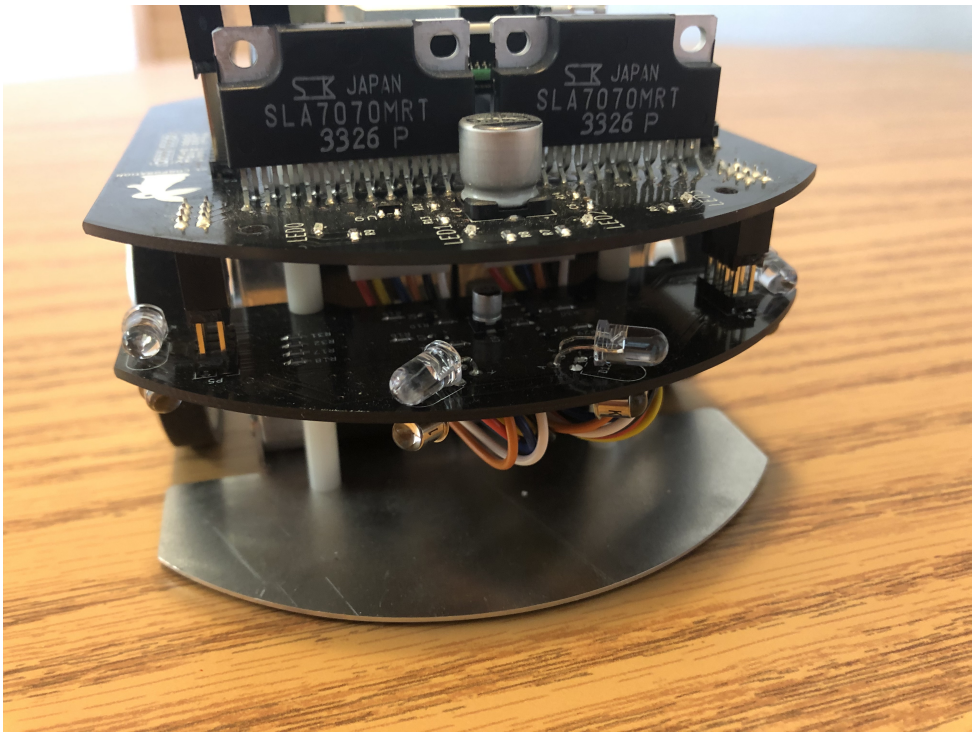

Figure 4.2: Here, the front view of the robot reveals the four LEDs and corresponding phototransistors for wall detection.

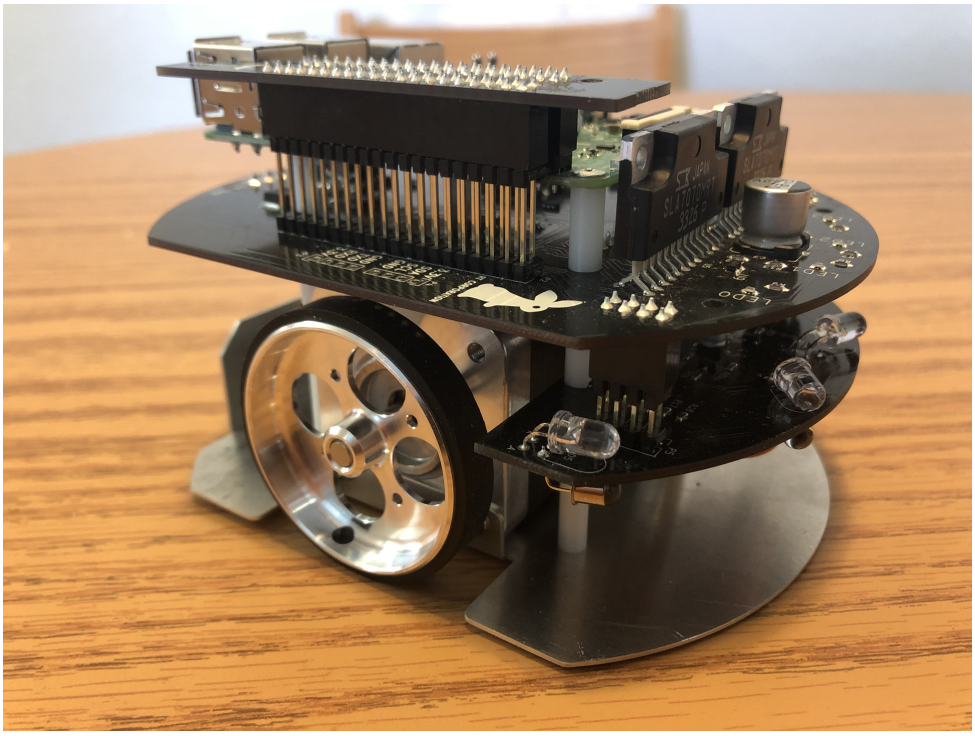

Figure 4.3: The stepper motors allowed the left and right wheels to move independently. This made for smooth turns as one wheel would move forward while the other moved backward.

understood with the following equation.

$$sensor\ output = ADC\ value\ after - ADC\ value\ before \tag{4.1}$$

The stepper motors were Minebea's 4-phase model 17PM-K064UN02CN no. T18912-01. One can be seen in figure 4.3. The specifications for the stepper motors were 0.9 degrees per pulse, i.e. one rotation required 400 pulses. The motors also communicated with the Raspberry Pi via SPI protocol. Lastly a Tattu, 3s 11.1V lithium polymer battery was used so that the power cable

would not impede the robot's motion through the maze.

Base	West	South	East	North
2	1	1	1	0
10	1×2^3	1×2^2	1×2^1	0×2^0

Table 4.1: This table shows how the wall information for each maze cell was encoded as a 4-bit or decimal number. To convert from binary to decimal form, one would simply sum the values in the last row of this table.

The first two Matlab files in appendix A were created to simulate the flood-fill algorithm before running it on the robot's hardware. The function "find min" was included to find the open neighbor cell with the smallest distance value. Similarly, the first two Python files in appendix B were translated directly from the Matlab files. The "flood fill" Python file solved the maze using the modified flood-fill algorithm as described in chapter 2 of this report. To do this, a matrix containing the wall information was translated from decimal to binary. The binary digits "1" and "0" represented the presence and the absence of a wall, respectively. The location of a given wall depended on the position of the "1" values in the 4-bit number. Reading from left to right, the first position represented west, the second position represented south, the third position represented east, and the fourth position represented north. This information encoding scheme can be seen in table 4.1. This matrix was then input into the "flood fill" file. The output was stored in a second distance value matrix shown in table 4.2. The robot was then able to follow the shortest path to the center of the maze by moving towards the cell with the next smallest distance value. To calculate the

distance values for each cell, the "flood fill" file looped though maze cells until the destination cell was reached. The distance values were updated when the following equation was false.

$$current\ dist.\ value - 1 = min.\ dist.\ value\ of\ open\ neighbors \tag{4.2}$$

The location of the current cell was first pushed onto the stack as in figure 4.5. While the stack was not empty, the stack's top element was popped. If equation 4.2 was false, the current cell's distance value was increased by one and its neighbors were pushed onto the stack. It's important to note that if any of the neighbors were destination cells, they were not pushed onto the stack. The next three Python files were "init", "read sensor", and "direct".

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

Table 4.2: This table shows the distance values corresponding to each cell in the maze. A given cell's distance value is the number of lateral movements required to get from the center of the maze to the cell in question.

The "init" and "direct" files were run from the bash startup file when the Raspberry Pi was turned on. The "init" file installed the device drivers and then gave permission to run them. The "read sensor" file returned a list of the sensor values. Lastly, the "direct" file determined whether or not the

robot was at a border cell and the robot's previous absolute orientation, i.e. north, south, west, or east. With that information, the robot moved a combination of forward, left and, right to arrive at the open neighbor cell with the smallest distance value. To keep the robot from crashing into a wall, the diagonal left and right sensors were used.

Lastly, it is also important to note that the positions of the maze walls were known to the robot before it moved. Further improvements to this project would include writing an explore file that assumes the robot has no previous knowledge of the maze wall positions. The process of solving the maze without prior knowledge of the wall positions is outlined in figure 4.4. The robot is represented by the arrow on the red matrix. Before each movement, the robot would use ultra-sonic sensors to detect the walls surrounding the current cell.

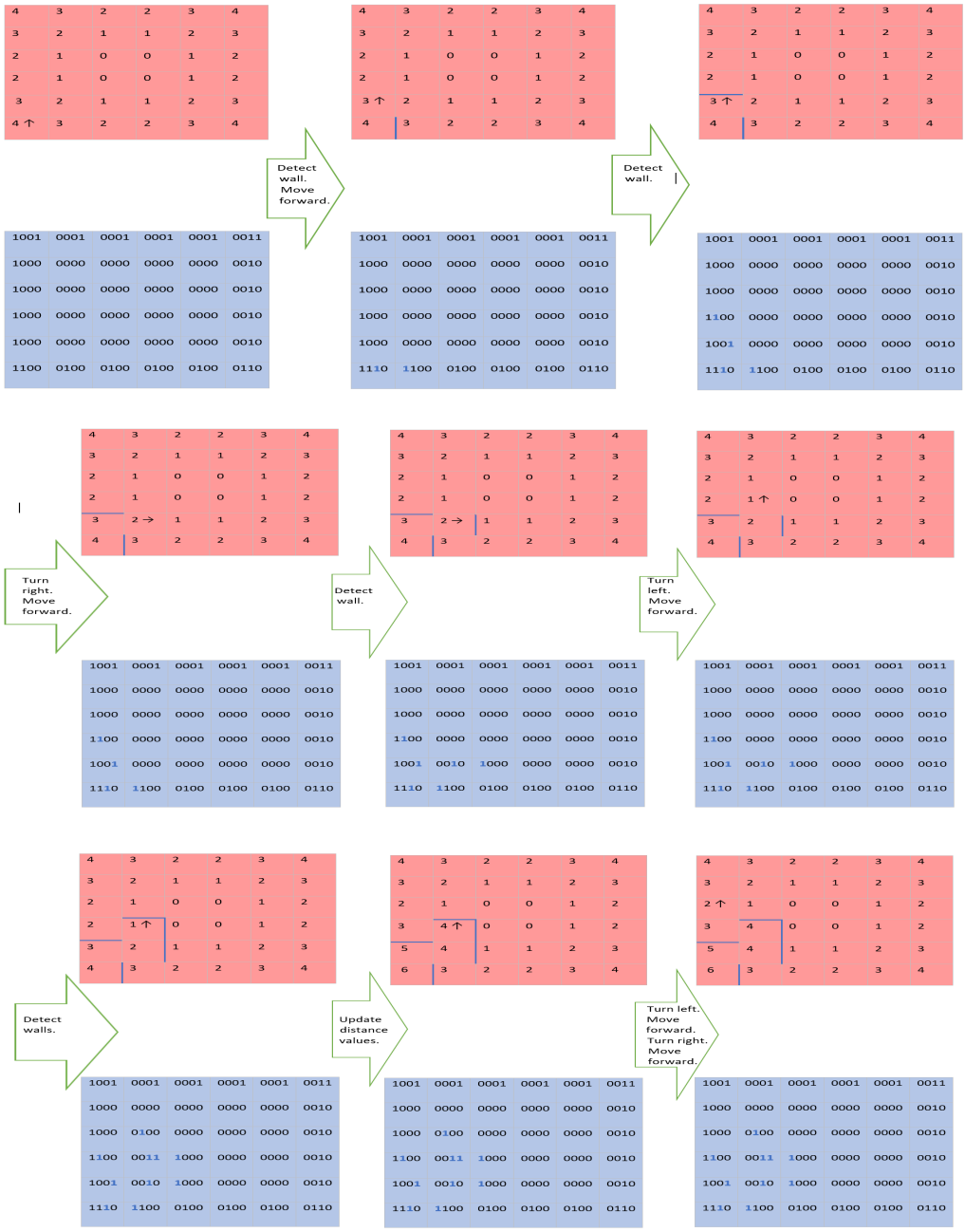

Figure 4.4: The red matrices hold distance values and the blue matrices hold wall information. Each red-blue matrix pair represents the robot's memory at one point in time. The blue lines in the red matrices and the blue-colored digits in the blue matrices represent newly detected walls.

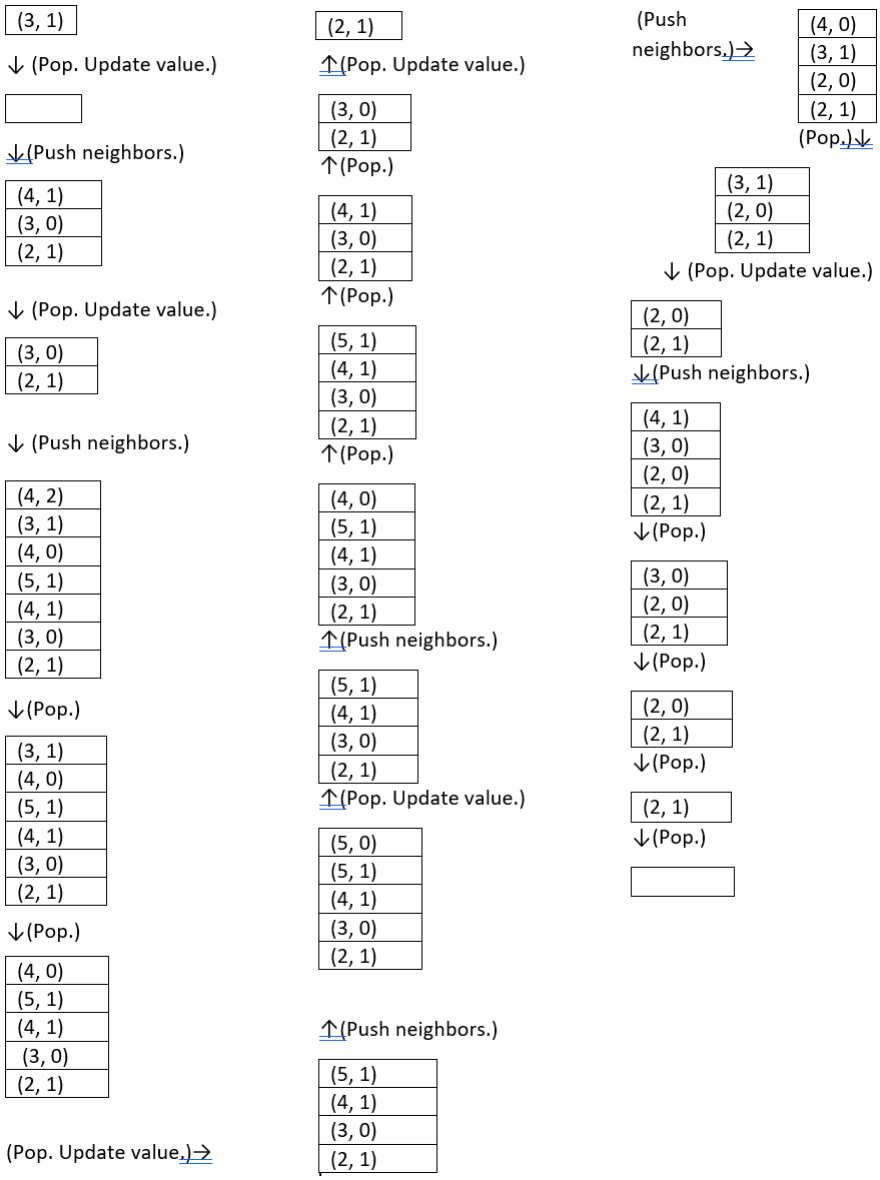

Figure 4.5: When the distance values needed to be updated, the stack was used. Only select neighbor cells were pushed onto the stack, making the modified flood-fill algorithm more computationally efficient than the original flood-fill algorithm.

Chapter 5

Results

The robot was able to navigate the maze by using its on-board sensors to detect when it was approaching a wall. Only one intervention was needed to readjust the robot's course when it was unable to detect a wall in the neighboring cell. If the value of the two front sensors angled toward the left and right surpassed a threshold value of 500, the corresponding stepper motor frequency would be increased by 50 pulses. That is, if the right sensor value surpassed 500, the right stepper motor would increase from its default frequency of 135 to 185 pulses for 0.1 seconds. It would remain at that frequency until the sensor value dropped below the threshold value. It is important to note that it was impossible to measure the distance from the robot to the wall with the on-board sensors due to extraneous factors such as the changing angle of incidence of the robot and the lighting conditions of the room.

When threshold values below 500 were used in the testing phase, the robot would not increase the frequency of the motor in time to avoid collision. It was only after observing the values printed to the screen when running the program for maze navigation that the correct threshold value was determined.

If more time had allowed, the maze wall configurations could have been changed and the maze dimensions expanded to compare and contrast runs in different environments. Expanding the maze dimensions would have also more closely replicated the conditions for the IEEE micromouse competition. In further work, the robot will be completely autonomous when it is able to successfully navigate different mazes without previous knowledge of the wall configuration.

Chapter 6

Conclusion

This project was successful in that a functional maze-solving robot was implemented. However, the robot had previous knowledge of the wall locations. That is, the wall information was input into the flood-fill algorithm to find the shortest path before the motors were run. Further work could be done to make the robot fully autonomous. This would most likely require using a more reliable sensor such as an ultrasonic sensor to detect the wall positions as the robot moves through the maze. The ultrasonic sensor would be more reliable because it would not be susceptible to the changing light conditions of different rooms. Another useful sensor would be the gyroscope to give the orientation of the robot with respect to the absolute directions. This information is crucial to solving the maze as the implementation of the flood-fill algorithm requires the wall information to be stored in bits that represent absolute north, south, east, and west.

Appendix A

Matlab Code

"flood fill"

```
clear; clc;

%read in values from maze text file
%only mazes with even numbered dimensions work
fileID = fopen('maze1.txt','r');
formatSpec = '%d';
A = fscanf(fileID,formatSpec);
fclose(fileID);

%find maze dimensions
dim = sqrt(numel(A));
```

```
%convert A from vector to matrix form
A = vec2mat(A,dim);

%create a new maze with initial flood fill values
B = ones(dim);

%initialize the destination squares to zero
B(dim/2:dim/2+1, dim/2:dim/2+1) = 0;

%fill remaining squares with number of cells away from destination
%fill squares to the left and right of destination
for i=2:dim/2-1
    B(dim/2:dim/2+1, dim/2-i) = i;
    B(dim/2:dim/2+1, dim/2+1+i) = i;
end

%fill squares to top and bottom of destination
for i=2:dim/2-1
    B(dim/2-i, dim/2:dim/2+1) = i;
    B(dim/2+1+i, dim/2:dim/2+1) = i;
end
```

```
%fill upper left corner
for i=dim/2:-1:2
    for j=dim/2:-1:1
        B(i-1,j) = B(i,j)+1;
    end
end

%fill upper right corner
for i=dim/2:-1:2
    for j=dim/2+1:dim
        B(i-1,j) = B(i,j)+1;
    end
end

%fill lower left corner
for i=dim/2+1:dim-1
    for j=dim/2:-1:1
        B(i+1,j) = B(i,j)+1;
    end
end

%fill lower right corner
for i=dim/2+1:dim-1
```

```
    for j=dim/2+1:dim
        B(i+1,j) = B(i,j)+1;
    end
end

B

i = 6;
j = 1;
%initialize stack
stack = java.util.Stack();

while B(i,j) ~= 0
    [min_open, W, S, E, N] = find_min(dim,A,B,i,j);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    if B(i,j) - 1 ~= min_open
        %push current cell onto stack
        stack.push([i j]);
        while ~stack.isEmpty()
            coord = stack.pop();
            m = coord(1);
            n = coord(2);
            [min_open] = find_min(dim,A,B,m,n);
```

```
if B(m,n) - 1 ~= min_open
    B(m,n) = min_open + 1;
    %push North neighbor
    if m ~= 1
        if B(m-1,n) ~= 0
            stack.push([m-1 n]);
        end
    end
    %push South neighbor
    if m ~= dim
        if B(m+1,n) ~= 0
            stack.push([m+1 n]);
        end
    end
    %push West nieghbor
    if n ~= 1
        if B(m,n-1) ~= 0
            stack.push([m n-1]);
        end
    end
    %push East neighbor
    if n ~= dim
        if B(m,n+1) ~= 0
```

```
                    stack.push([m n+1]);
                end
            end
        end
    end
    %lead robot to nearest neighbor not separated by a wall and with th
    %lowest flood fill value
    if min_open == W
        j = j - 1;
    end
    if min_open == S
        i = i + 1;
    end
    if min_open == E
        j = j + 1;
    end
    if min_open == N
        i = i - 1;
    end

end
```

B

“find min”

```
function [min_open, W, S, E, N] = find_min(dim,A,B,i,j)
    binary = de2bi(A(i,j),4,'left-msb');

    count = 1;

    W = dim^2;
    S = dim^2;
    E = dim^2;
    N = dim^2;

    for m=1:4
        if binary(m) == 0 && count == 1
            W = B(i,j-1);
        end
        if binary(m) == 0 && count == 2
            S = B(i+1,j);
        end
        if binary(m) == 0 && count == 3
            E = B(i,j+1);
```

```
        end
        if binary(m) == 0 && count == 4
            N = B(i-1,j);
        end
        count = count + 1;
    end
    min_open = min([W S E N]);
end
```

Appendix B

Python Code

"flood fill"

```
import numpy as np
from find_min import find_min

def floodfill(A):
    #input maze dimensions
    #only even numbers
    dim = 6

    #distance value matrix
    B = np.zeros((dim, dim), dtype=int)
```

```
#set int type
half_dim = int(dim/2)

#fill squares to the left and right of destination
for i in range(1, half_dim):
    B[half_dim - 1:half_dim + 1, half_dim - 1 - i] = i
    B[half_dim - 1:half_dim + 1, half_dim + i] = i

#fill squares to top and bottom of destination
for i in range(1, half_dim):
    B[half_dim - 1 - i, half_dim - 1:half_dim + 1] = i
    B[half_dim + i, half_dim - 1:half_dim + 1] = i

#fill upper left corner
for i in range(half_dim - 1, -1, -1):
    for j in range(half_dim - 2, -1, -1):
        B[i - 1, j] = B[i, j] + 1

#fill upper right corner
for i in range(half_dim - 1, -1, -1):
    for j in range(half_dim + 1, dim):
        B[i - 1, j] = B[i, j] + 1
```

```
#fill lower left corner
for i in range(half_dim, dim - 1):
    for j in range(half_dim - 2, -1, -1):
        B[i + 1, j] = B[i, j] + 1

#fill lower right corner
for i in range(half_dim, dim - 1):
    for j in range(half_dim + 1, dim):
        B[i + 1, j] = B[i, j] + 1

#start in lower left corner
i = dim - 1
j = 0

#initialize stack
stack = []

while B[i, j] != 0:
    #binary = read_sensor()
    min_open, W, S, E, N = find_min(dim, A, B, i, j)
    if B[i, j] - 1 != min_open:
        #push current cell onto stack
        stack.append( (i, j) )
```

```
while len(stack) != 0:
    m, n = stack.pop()
    #binary = read_sensor()
    min_open = find_min(dim, A, B, m, n)[0]
    if B[m, n] - 1 != min_open:
        B[m, n] = min_open + 1
        #push North neighbor
        if m != 0:
            if B[m - 1, n] != 0:
                stack.append( (m-1, n) )
        #push South neighbor
        if m != dim - 1:
            if B[m + 1, n] != 0:
                stack.append( (m+1, n) )
        #push West neighbor
        if n != 0:
            if B[m, n - 1] != 0:
                stack.append( (m, n-1) )
        #push East neighbor
        if n != dim - 1:
            if B[m, n + 1] != 0:
                stack.append( (m, n+1) )
#lead robot to nearest neighbor not separated by a wall and with
```

```
        if min_open == W:
            j = j - 1
        if min_open == S:
            i = i + 1
        if min_open == E:
            j = j + 1
        if min_open == N:
            i = i - 1
    return B
```

"find min"

```
def find_min(dim, A, B, i, j):
    binary = []

    for char in A[i, j]:
        binary.append(int(char))

    count = 1

    W = dim**2
    S = dim**2
    E = dim**2
```

```
    N = dim**2

    for index in range(4):
        if binary[index] == 0 and count == 1:
            W = B[i, j-1]
        if binary[index] == 0 and count == 2:
            S = B[i+1, j]
        if binary[index] == 0 and count == 3:
            E = B[i, j+1]
        if binary[index] == 0 and count == 4:
            N = B[i-1, j]
        count += 1

    min_open = min([W, S, E, N])

    return [min_open, W, S, E, N]
```

“init”

```
import os

os.chdir('./RaspberryPiMouse/src/drivers')
os.system('sudo insmod rtmouse.ko')
```

```
os.system('sudo chmod 666 /dev/rt*')
```

"read sensor"

```
import subprocess

def read_sensor():
    #returns output as byte string
    sensor_reading = subprocess.check_output('cat /dev/rtlightsensor0',

    #using decode() function to convert byte string to string
    sensor_reading_string = sensor_reading.decode('utf-8')

    #convert string into a list of integers
    sensor_reading_list = list(map(int, sensor_reading_string.split()))

    print(sensor_reading_list)

    return sensor_reading_list
```

"direct"

```
import time
import numpy as np
```

```
from floodfill import floodfill
from read_sensor import read_sensor

def go_forward():
    r_thresh = 50
    l_thresh = 50
    fr_thresh = 500
    fl_thresh = 500

    freq = 135
    df = 50

    for i in range(30):
        sensor_reading_list = read_sensor()
        right = sensor_reading_list[0]
        front_right = sensor_reading_list[1]
        front_left = sensor_reading_list[2]
        left = sensor_reading_list[3]

        if front_left > fl_thresh:
            with open('/dev/rtmotor_raw_l0', 'w') as f:
                f.write(str(freq+df))
            with open('/dev/rtmotor_raw_r0', 'w') as f:
```

```
            f.write(str(freq))

    elif front_right > fr_thresh:
        with open('/dev/rtmotor_raw_l0', 'w') as f:
            f.write(str(freq))
        with open('/dev/rtmotor_raw_r0', 'w') as f:
            f.write(str(freq+df))

    else:
        for filename in files:
            with open(filename, 'w') as f:
                f.write(str(freq))
    time.sleep(0.1)

    #for filename in files:
    #    with open(filename, 'w') as f:
    #        f.write('0')
    #time.sleep(0.5)

for filename in files:
    with open(filename, 'w') as f:
        f.write('0')
time.sleep(1.0)
```

```
    #for filename in files:
    #    with open(filename, 'w') as f:
    #        f.write('500')
    #time.sleep(1.0)

def turn_right():
    with open('/dev/rtmotor_raw_l0', 'w') as f:
        f.write('210')
    with open('/dev/rtmotor_raw_r0', 'w') as f:
        f.write('-210')
    time.sleep(1.0)

    for filename in files:
        with open(filename, 'w') as f:
            f.write('0')
    time.sleep(1.0)

def turn_left():
    with open('/dev/rtmotor_raw_l0', 'w') as f:
        f.write('-210')
    with open('/dev/rtmotor_raw_r0', 'w') as f:
        f.write('210')
    time.sleep(1.0)
```

```
    for filename in files:
        with open(filename, 'w') as f:
            f.write('0')
    time.sleep(1.0)

def go_backward():
    for filename in files:
        with open(filename, 'w') as f:
            f.write('-500')
    time.sleep(1.0)

    for filename in files:
        with open(filename, 'w') as f:
            f.write('0')
    time.sleep(1.0)

dim = 6

#wall information matrix
#A = np.zeros((dim, dim), dtype=int)
A = []
file = open('maze1.txt', 'r')
```

```
for line in file.readlines():
    line_list = line.split()
    line_list = [format(int(i), '04b') for i in line_list]
    A.append(line_list)
A = np.array(A)

print(A)

B = floodfill(A)

print(B)

i = dim-1
j = 0

files = ['/dev/rtmotor_raw_l0', '/dev/rtmotor_raw_r0']

abs_dir = 'N'

while B[i, j] != 0:
    print(B[i, j])
    if (j in range(1, dim - 1)) and (i in range(1, dim - 1)):
        #West
```

```
if B[i, j - 1] == B[i, j] - 1:
    if abs_dir == 'W':
        #go forward
        go_forward()

    if abs_dir == 'S':
        #turn right
        turn_right()
        #go forward
        go_forward()

    if abs_dir == 'E':
        #turn left
        turn_left()
        #turn left
        turn_left()
        #go forward
        go_forward()

    if abs_dir == 'N':
        #turn left
        turn_left()
        #go forward
```

```
            go_forward()


        abs_dir = 'W'
        j = j - 1
    #South
    if B[i + 1, j] == B[i, j] - 1:
        if abs_dir == 'W':
            #turn left
            turn_left()
            #go forward
            go_forward()

        if abs_dir == 'S':
            #go forward
            go_forward()

        if abs_dir == 'E':
            #turn right
            turn_right()
            #go forward
            go_forward()

        if abs_dir == 'N':
```

```
            #turn left
            turn_left()
            #turn left
            turn_left()
            #go forward
            go_forward()

        abs_dir = 'S'
        i = i + 1
    #East
    if B[i, j + 1] == B[i, j] - 1:
        if abs_dir == 'W':
            #turn left
            turn_left()
            #turn left
            turn_left()
            #go forward
            go_forward()

        if abs_dir == 'S':
            #turn left
            turn_left()
            #go forward
```

```
            go_forward()

        if abs_dir == 'E':
            #go forward
            go_forward()

        if abs_dir == 'N':
            #turn right
            turn_right()
            #go forward
            go_forward()

        abs_dir = 'E'
        j = j + 1
    #North
    if B[i - 1, j] == B[i, j] - 1:
        if abs_dir == 'W':
            #turn right
            turn_right()
            #go forard
            go_forward()

        if abs_dir == 'S':
```

```
                #turn left
                turn_left()
                #turn left
                turn_left()
                #go forward
                go_forward()

            if abs_dir == 'E':
                #turn left
                turn_left()
                #go forward
                go_forward()

            if abs_dir == 'N':
                #go forward
                go_forward()

            abs_dir = 'N'
            i = i - 1
    elif (j - 1 < 0) and (i in range(1, dim - 1)):
        #South
        if B[i + 1, j] == B[i, j] - 1:
            if abs_dir == 'W':
```

```
        #turn left
        turn_left()
        #go forward
        go_forward()

    if abs_dir == 'S':
        #go forward
        go_forward()

    if abs_dir == 'E':
        #turn right
        turn_right()
        #go forward
        go_forward()

    if abs_dir == 'N':
        #turn left
        turn_left()
        #turn left
        turn_left()
        #go forward
        go_forward()
```

```
        abs_dir = 'S'
        i = i + 1
    #East
    if B[i, j + 1] == B[i, j] - 1:
        if abs_dir == 'W':
            #turn left
            turn_left()
            #turn left
            turn_left()
            #go forward
            go_forward()

        if abs_dir == 'S':
            #turn left
            turn_left()
            #go forward
            go_forward()

        if abs_dir == 'E':
            #go forward
            go_forward()

        if abs_dir == 'N':
```

```
            #turn right
            turn_right()
            #go forward
            go_forward()

        abs_dir = 'E'
        j = j + 1
    #North
    if B[i - 1, j] == B[i, j] - 1:
        if abs_dir == 'W':
            #turn right
            turn_right()
            #go forard
            go_forward()

        if abs_dir == 'S':
            #turn left
            turn_left()
            #turn left
            turn_left()
            #go forward
            go_forward()
```

```
            if abs_dir == 'E':
                #turn left
                turn_left()
                #go forward
                go_forward()

            if abs_dir == 'N':
                #go forward
                go_forward()

            abs_dir = 'N'
            i = i - 1
    elif (i + 1 > dim - 1) and (j in range(1, dim - 1)):
        #West
        if B[i, j - 1] == B[i, j] - 1:
            if abs_dir == 'W':
                #go forward
                go_forward()

            if abs_dir == 'S':
                #turn right
                turn_right()
                #go forward
```

```
            go_forward()

        if abs_dir == 'E':
            #turn left
            turn_left()
            #turn left
            turn_left()
            #go forward
            go_forward()

        if abs_dir == 'N':
            #turn left
            turn_left()
            #go forward
            go_forward()

        abs_dir = 'W'
        j = j - 1
    #East
    if B[i, j + 1] == B[i, j] - 1:
        if abs_dir == 'W':
            #turn left
            turn_left()
```

```
        #turn left
        turn_left()
        #go forward
        go_forward()

    if abs_dir == 'S':
        #turn left
        turn_left()
        #go forward
        go_forward()

    if abs_dir == 'E':
        #go forward
        go_forward()

    if abs_dir == 'N':
        #turn right
        turn_right()
        #go forward
        go_forward()

    abs_dir = 'E'
    j = j + 1
```

```
#North
if B[i - 1, j] == B[i, j] - 1:
    if abs_dir == 'W':
        #turn right
        turn_right()
        #go forard
        go_forward()

    if abs_dir == 'S':
        #turn left
        turn_left()
        #turn left
        turn_left()
        #go forward
        go_forward()

    if abs_dir == 'E':
        #turn left
        turn_left()
        #go forward
        go_forward()

    if abs_dir == 'N':
```

```
            #go forward
            go_forward()

        abs_dir = 'N'
        i = i - 1
elif (j + 1 > dim - 1) and (i in range(1, dim - 1)):
    #West
    if B[i, j - 1] == B[i, j] - 1:
        if abs_dir == 'W':
            #go forward
            go_forward()

        if abs_dir == 'S':
            #turn right
            turn_right()
            #go forward
            go_forward()

        if abs_dir == 'E':
            #turn left
            turn_left()
            #turn left
            turn_left()
```

```
            #go forward
            go_forward()

        if abs_dir == 'N':
            #turn left
            turn_left()
            #go forward
            go_forward()

        abs_dir = 'W'
        j = j - 1
    #South
    if B[i + 1, j] == B[i, j] - 1:
        if abs_dir == 'W':
            #turn left
            turn_left()
            #go forward
            go_forward()

        if abs_dir == 'S':
            #go forward
            go_forward()
```

```
        if abs_dir == 'E':
            #turn right
            turn_right()
            #go forward
            go_forward()

        if abs_dir == 'N':
            #turn left
            turn_left()
            #turn left
            turn_left()
            #go forward
            go_forward()

        abs_dir = 'S'
        i = i + 1
    #North
    if B[i - 1, j] == B[i, j] - 1:
        if abs_dir == 'W':
            #turn right
            turn_right()
            #go forard
            go_forward()
```

```
            if abs_dir == 'S':
                #turn left
                turn_left()
                #turn left
                turn_left()
                #go forward
                go_forward()

            if abs_dir == 'E':
                #turn left
                turn_left()
                #go forward
                go_forward()

            if abs_dir == 'N':
                #go forward
                go_forward()

            abs_dir = 'N'
            i = i - 1
    elif (i - 1 < 0) and (j in range(1, dim - 1)):
        #West
```

```
if B[i, j - 1] == B[i, j] - 1:
    if abs_dir == 'W':
        #go forward
        go_forward()

    if abs_dir == 'S':
        #turn right
        turn_right()
        #go forward
        go_forward()

    if abs_dir == 'E':
        #turn left
        turn_left()
        #turn left
        turn_left()
        #go forward
        go_forward()

    if abs_dir == 'N':
        #turn left
        turn_left()
        #go forward
```

```
            go_forward()


        abs_dir = 'W'
        j = j - 1
    #South
    if B[i + 1, j] == B[i, j] - 1:
        if abs_dir == 'W':
            #turn left
            turn_left()
            #go forward
            go_forward()

        if abs_dir == 'S':
            #go forward
            go_forward()

        if abs_dir == 'E':
            #turn right
            turn_right()
            #go forward
            go_forward()

        if abs_dir == 'N':
```

```
            #turn left
            turn_left()
            #turn left
            turn_left()
            #go forward
            go_forward()

        abs_dir = 'S'
        i = i + 1
    #East
    if B[i, j + 1] == B[i, j] - 1:
        if abs_dir == 'W':
            #turn left
            turn_left()
            #turn left
            turn_left()
            #go forward
            go_forward()

        if abs_dir == 'S':
            #turn left
            turn_left()
            #go forward
```

```
            go_forward()

        if abs_dir == 'E':
            #go forward
            go_forward()

        if abs_dir == 'N':
            #turn right
            turn_right()
            #go forward
            go_forward()

        abs_dir = 'E'
        j = j + 1
elif (i == dim - 1) and (j == 0):
    #East
    if B[i, j + 1] == B[i, j] - 1:
        if abs_dir == 'W':
            #turn left
            turn_left()
            #turn left
            turn_left()
            #go forward
```

```
            go_forward()

        if abs_dir == 'S':
            #turn left
            turn_left()
            #go forward
            go_forward()

        if abs_dir == 'E':
            #go forward
            go_forward()

        if abs_dir == 'N':
            #turn right
            turn_right()
            #go forward
            go_forward()

        abs_dir = 'E'
        j = j + 1
    #North
    if B[i - 1, j] == B[i, j] - 1:
        if abs_dir == 'W':
```

```
        #turn right
        turn_right()
        #go forard
        go_forward()

    if abs_dir == 'S':
        #turn left
        turn_left()
        #turn left
        turn_left()
        #go forward
        go_forward()

    if abs_dir == 'E':
        #turn left
        turn_left()
        #go forward
        go_forward()

    if abs_dir == 'N':
        #go forward
        go_forward()
```

```
            abs_dir = 'N'
            i = i - 1
    elif (i == 0) and (j == 0):
        #South
        if B[i + 1, j] == B[i, j] - 1:
            if abs_dir == 'W':
                #turn left
                turn_left()
                #go forward
                go_forward()

            if abs_dir == 'S':
                #go forward
                go_forward()

            if abs_dir == 'E':
                #turn right
                turn_right()
                #go forward
                go_forward()

            if abs_dir == 'N':
                #turn left
```

```
            turn_left()
            #turn left
            turn_left()
            #go forward
            go_forward()

        abs_dir = 'S'
        i = i + 1
    #East
    if B[i, j + 1] == B[i, j] - 1:
        if abs_dir == 'W':
            #turn left
            turn_left()
            #turn left
            turn_left()
            #go forward
            go_forward()

        if abs_dir == 'S':
            #turn left
            turn_left()
            #go forward
            go_forward()
```

```
            if abs_dir == 'E':
                #go forward
                go_forward()

            if abs_dir == 'N':
                #turn right
                turn_right()
                #go forward
                go_forward()

            abs_dir = 'E'
            j = j + 1
    elif (i == 0) and (j == dim - 1):
        #West
        if B[i, j - 1] == B[i, j] - 1:
            if abs_dir == 'W':
                #go forward
                go_forward()

            if abs_dir == 'S':
                #turn right
                turn_right()
```

```
            #go forward
            go_forward()

        if abs_dir == 'E':
            #turn left
            turn_left()
            #turn left
            turn_left()
            #go forward
            go_forward()

        if abs_dir == 'N':
            #turn left
            turn_left()
            #go forward
            go_forward()

        abs_dir = 'W'
        j = j - 1
    #South
    if B[i + 1, j] == B[i, j] - 1:
        if abs_dir == 'W':
            #turn left
```

```
        turn_left()
        #go forward
        go_forward()

    if abs_dir == 'S':
        #go forward
        go_forward()

    if abs_dir == 'E':
        #turn right
        turn_right()
        #go forward
        go_forward()

    if abs_dir == 'N':
        #turn left
        turn_left()
        #turn left
        turn_left()
        #go forward
        go_forward()

    abs_dir = 'S'
```

```
            i = i + 1
    else:
        #West
        if B[i, j - 1] == B[i, j] - 1:
            if abs_dir == 'W':
                #go forward
                go_forward()

            if abs_dir == 'S':
                #turn right
                turn_right()
                #go forward
                go_forward()

            if abs_dir == 'E':
                #turn left
                turn_left()
                #turn left
                turn_left()
                #go forward
                go_forward()

            if abs_dir == 'N':
```

```
            #turn left
            turn_left()
            #go forward
            go_forward()

        abs_dir = 'W'
        j = j - 1
    #North
    if B[i - 1, j] == B[i, j] - 1:
        if abs_dir == 'W':
            #turn right
            turn_right()
            #go forard
            go_forward()

        if abs_dir == 'S':
            #turn left
            turn_left()
            #turn left
            turn_left()
            #go forward
            go_forward()
```

```
if abs_dir == 'E':
    #turn left
    turn_left()
    #go forward
    go_forward()

if abs_dir == 'N':
    #go forward
    go_forward()

abs_dir = 'N'
i = i - 1
```

Bibliography

[1] Yizhe Chang. *Robot Motion Planning Lecture No. 13*. 2019.

[2] Yizhe Chang. *Robot Motion Planning Lecture No. 15*. 2019.

[3] *Graph Theory - Trees*. 2019. URL: https://www.tutorialspoint.com/graph_theory/graph_theory_trees.htm.

[4] Daniel Klein. *Mighty Mouse*. 2018. URL: https://www.technologyreview.com/s/612529/mighty-mouse/.

[5] George Law. *Modified Floofill Algorithm*. 2019.

[6] George Law. "Quantitative Comparison of Flood Fill and Modified Flood Fill Algorithms". In: *International Journal of Computer Theory and Engineering* 5.3 (2013), pp. 503–508. DOI: 10.7763/IJCTE.2013.V5.738.

[7] *Micromouse Maze*. 2017. URL: https://site.ieee.org/r6-central/student-contests/.

[8] *MicroMouse Rules - Region 6 Central Area*. 2017. URL: https://site.ieee.org/r6-central/student-contests/.