Contents

PART I: TOPIC INTRODUCTION

1.Product description:

1.1 Product name: Solving maze car.
1.2 Purpose: The product describes the process of finding the path to the destination in a maze. The product includes 5 basic blocks for finding directions. Besides, the product also has the ability to develop in finding the shortest path.
1.3 Cost: about 650.000VND
1.4 Capacity: 12V/2A ensures sufficient capacity.
1.5 Size/weight: Dimensions about 25x20x10cm, weighs less than 5kg.

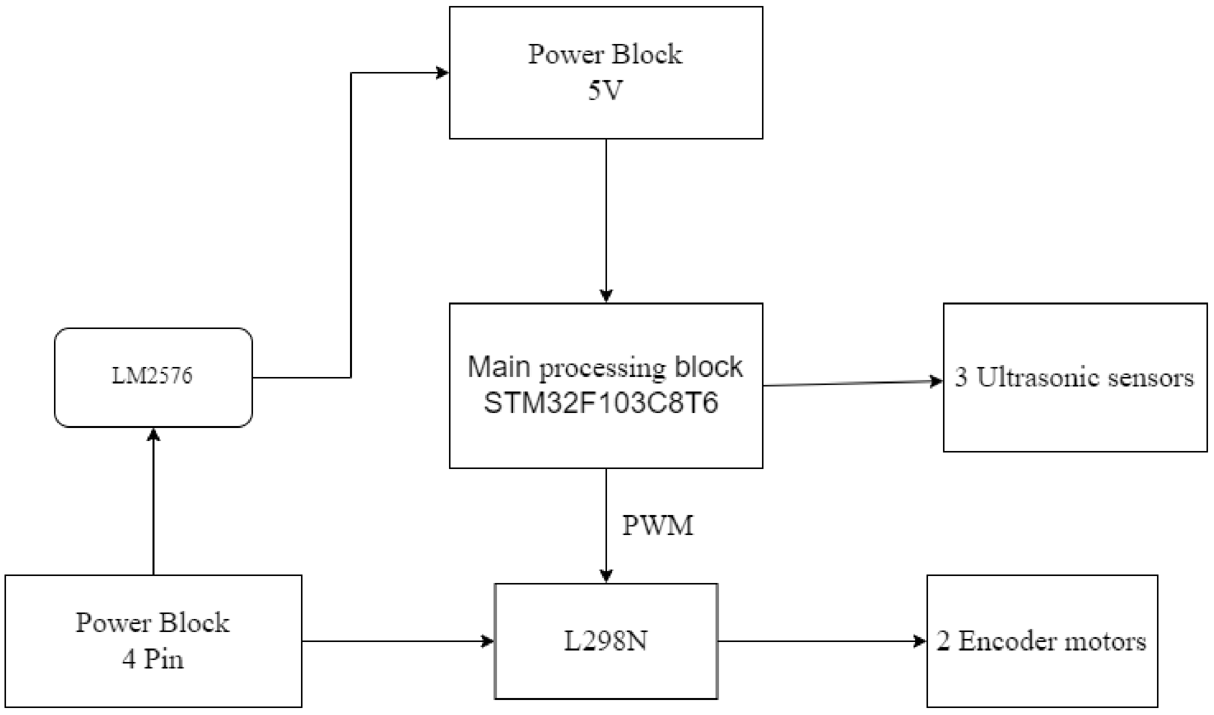
2.Operating principle:

When the car moves to the cells in the maze, it will retrieve information about the walls by ultrasonic sensors and save it in a pre-allocated memory. After receiving wall information, the vehicle will search for the next direction based on the Flood_flow algorithm, this process continues until the vehicle reaches the destination.

Operating environment:

The product can be easily moved and used in dry environments.

2.1 System flowchart



2.2 Description of main blocks:

Power Block: 4 pin 3.7/4.2V 6800mAh

Central processing block: Use STM32F103C8T6 chip as the central processing block. Receive signals from the sensor and output signals to control the H-bridge circuit for the vehicle to run

Ultrasonic Sensor: connected to the microcontroller, the sensor is used to measure distance, determine data as well as send data so the vehicle can navigate.

H-bridge circuit: PWM pulse signal is output from the VCR and signals are input to the pins to control the speed and direction of the vehicle.

3. Kế hoạch dự án

Project planning	
Team name	2 HMV
Product name	Solving maze car
Main goals	Find the way out of the maze
Estimated Time	2 months

Estimated Cost	650.000			
Team members	Name member: Nguyễn Quang Vinh: design car, build algorithm Nguyễn Hùng Minh: design motor, read and calculate value of sensors Phạm Nguyễn Minh Hiếu: building algorithm, general review, test algorithm.			
Schedule	Week 1	Week 2	Week 3	Week 4- 8
1. Design, building ideas, and buy components	→			
2. Build car		→		
3. Build algorithm software		→	→	→
4. Test product and review overall				→

PART II: INTRODUCE PID CONTROLLER.

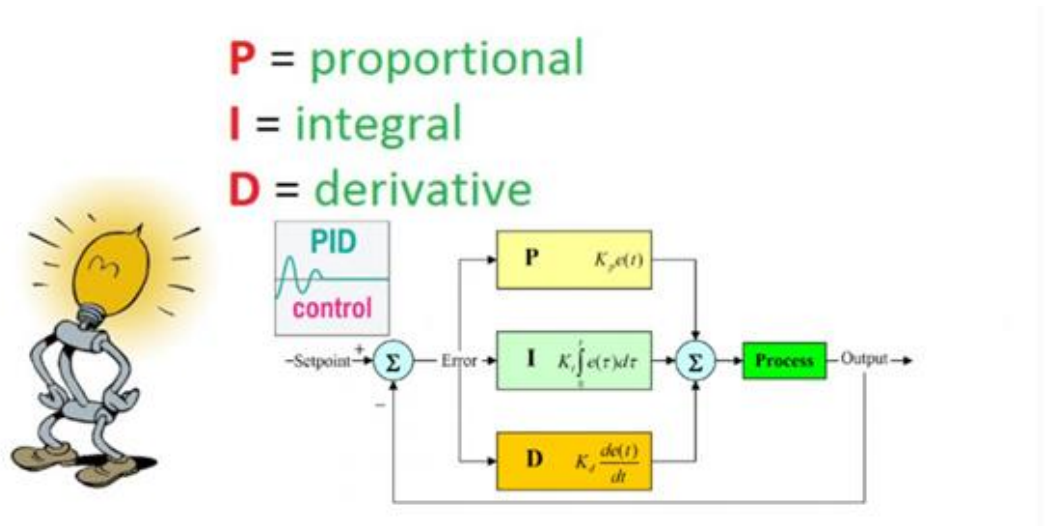
1.Introduction to PID

In order to achieve a statically stable gait, in spite of external forces and disturbances from the environment affecting the body’s center of gravity, a controller is used. The basic idea behind a controller is to read a sensor, compare the sensor reading to the desired reading and then compute the actuator input required in order to reach the desired value. In essence, a controller moves a system towards a desired output.

For instance, a controller can be used to keep a robot’s COM inside it’s support polygon as long as the position of its COM can be measured. For the sake of this project, a Proportional–integral–derivative (PID) controller will be implemented. This is due to a PIDs relative simplicity and ease of use. Using this kind of controller, the actuator input signal can be calculated as:

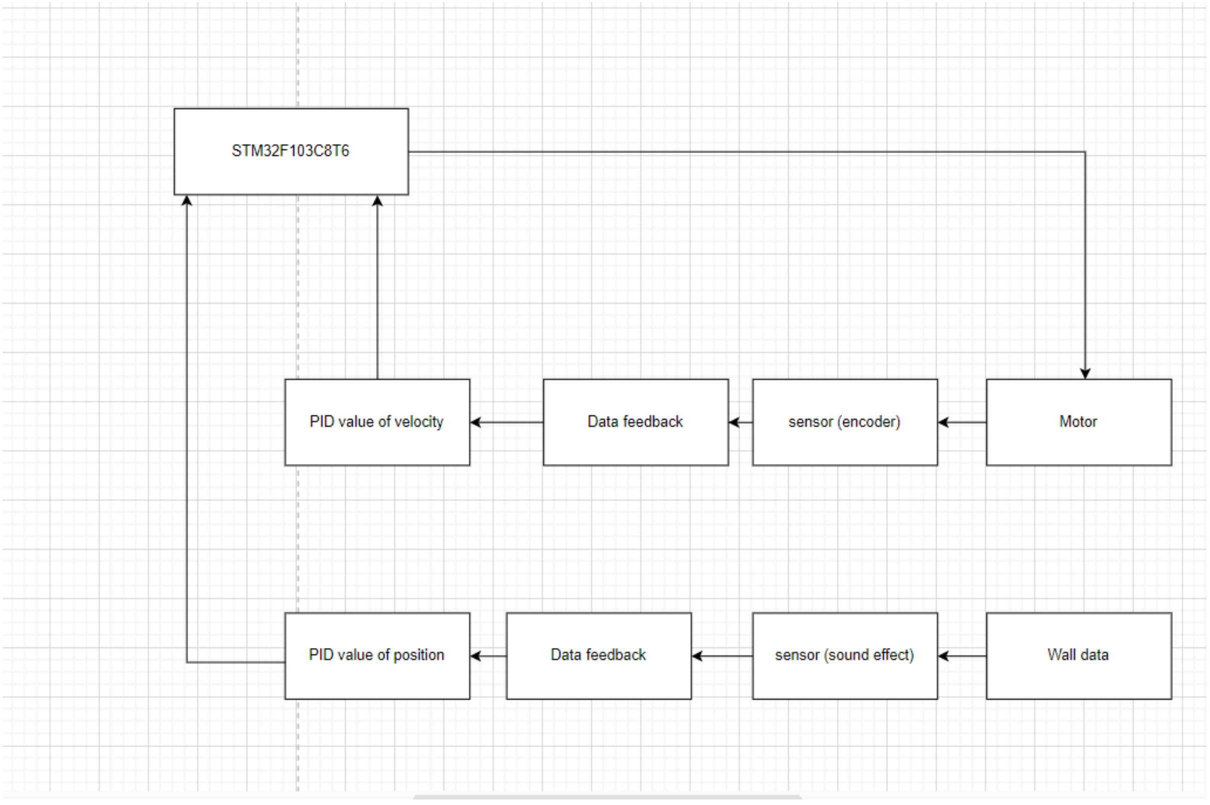
ut=KPet+ KI0teTdT+ KDde(t)dt

Where e(t) is the error between the desired output r and the real output according to a sensor y. KP , KI and KD are the proportional, integral and derivative components of the controller respectively. These values are adjusted in order to obtain a system that moves towards the desired value optimally.

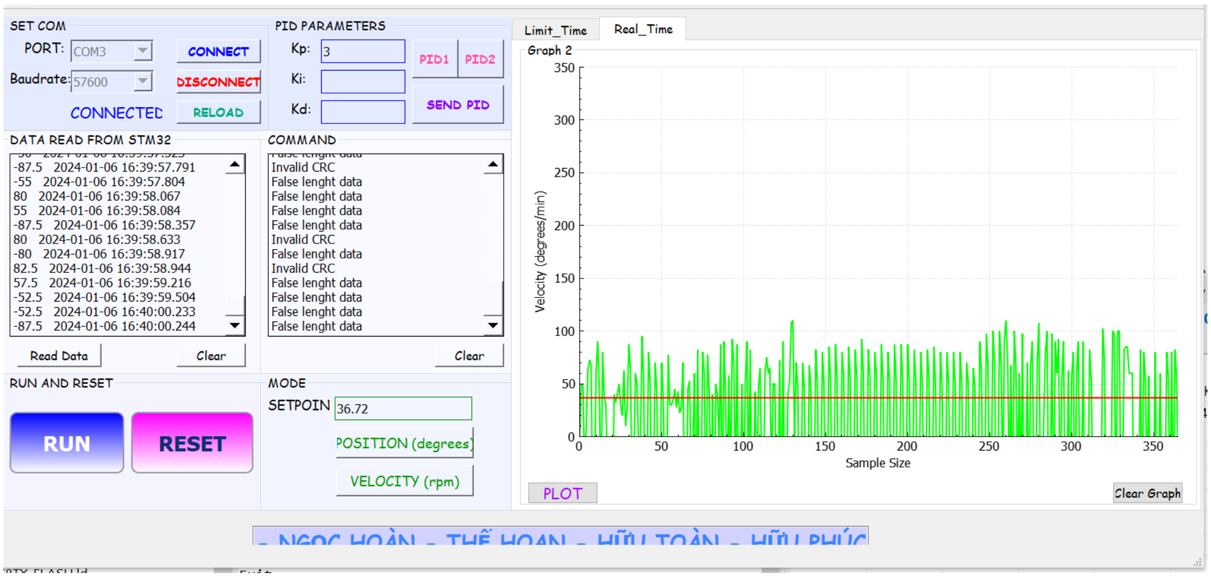


2. Application PID in project

flow chart of getting and controlling PID value



From the data we get from the sensor value if we have only Kp and Kp high we will have the picture below

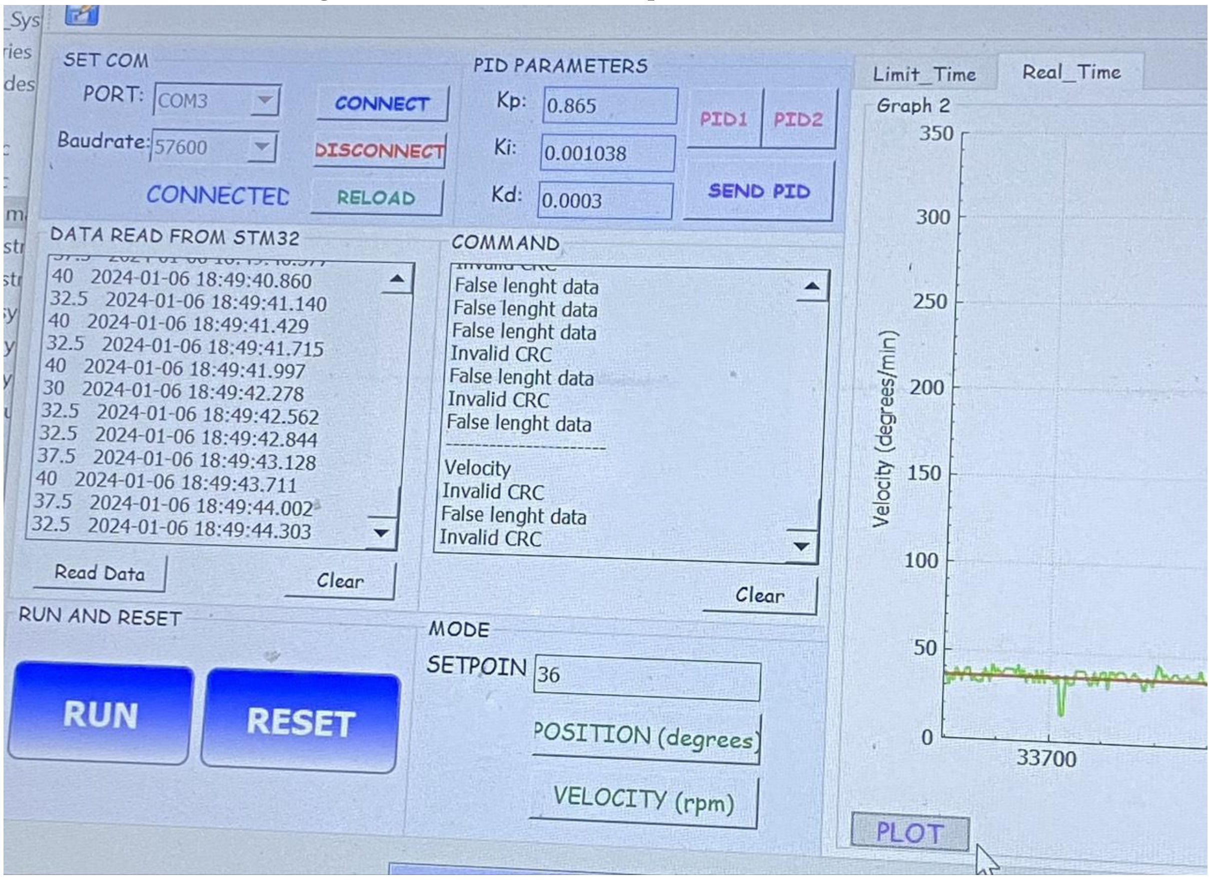


PORT: COM3
Baudrate: 57600
Kp: 3
Ki:
Kd:
SETPOIN 36.72

```
-87.5  2024-01-06 16:39:57.791
-55  2024-01-06 16:39:57.804
80  2024-01-06 16:39:58.067
55  2024-01-06 16:39:58.084
-87.5  2024-01-06 16:39:58.357
80  2024-01-06 16:39:58.633
-80  2024-01-06 16:39:58.917
82.5  2024-01-06 16:39:58.944
57.5  2024-01-06 16:39:59.216
-52.5  2024-01-06 16:39:59.504
-52.5  2024-01-06 16:40:00.233
-87.5  2024-01-06 16:40:00.244
```

```
Invalid CRC
False lenght data
False lenght data
False lenght data
False lenght data
Invalid CRC
False lenght data
Invalid CRC
False lenght data
False lenght data
False lenght data
False lenght data
```

And if we get the good value of PID, the differences between the value desire and the value in reality will be small. And we can get the result from the example results.



PORT: COM3
Baudrate: 57600
Kp: 0.865
Ki: 0.001038
Kd: 0.0003
SETPOIN 36

```
40  2024-01-06 18:49:40.860
32.5  2024-01-06 18:49:41.140
40  2024-01-06 18:49:41.429
32.5  2024-01-06 18:49:41.715
40  2024-01-06 18:49:41.997
30  2024-01-06 18:49:42.278
32.5  2024-01-06 18:49:42.562
32.5  2024-01-06 18:49:42.844
37.5  2024-01-06 18:49:43.128
40  2024-01-06 18:49:43.711
37.5  2024-01-06 18:49:44.002
32.5  2024-01-06 18:49:44.303
```

```
False lenght data
False lenght data
False lenght data
Invalid CRC
False lenght data
Invalid CRC
False lenght data
----------------------
Velocity
Invalid CRC
False lenght data
Invalid CRC
```

And From doing tuning PID we can implement that

Tune Kp until there is overshoot and undershoot to provide enough energy for the system

Tune Kd to reduce the overshoot caused by Kp, down to about 5 - 10%
Tune Ki to eliminate remaining setup errors and reduce rise and zero time
Increase Ki too large to avoid the overshoot from continuing to increase

PART III: FlOOD FLOW ALGORITHM

1. Introduction

Flood fill, also called seed fill, is a flooding algorithm that determines and alters the area connected to a given node in a multidimensional array with some matching attribute.
It is used in the "bucket" fill tool of paint programs to fill connected, similarly-colored areas with a different color, and in games such as Go and Minesweeper for determining which pieces are cleared. A variant called boundary fill uses the same algorithms but is defined as the area connected to a given node that does not have a particular attribute.
Many maze solving algorithms are readily available. With less demand for computation speed and with the assurance that the best run gives the smallest number of cells traveled, the modified flood fill algorithm is, by far, the most commonly used one in micromouse competitions.

2. Advantage and Disadvantage

Advantage: Using the FF algorithm can help the vehicle find the shortest path to the destination by scanning the maze and assigning each move with a certain value. At the same time, FF also helps the car avoid going around and getting stuck in loops in the maze.

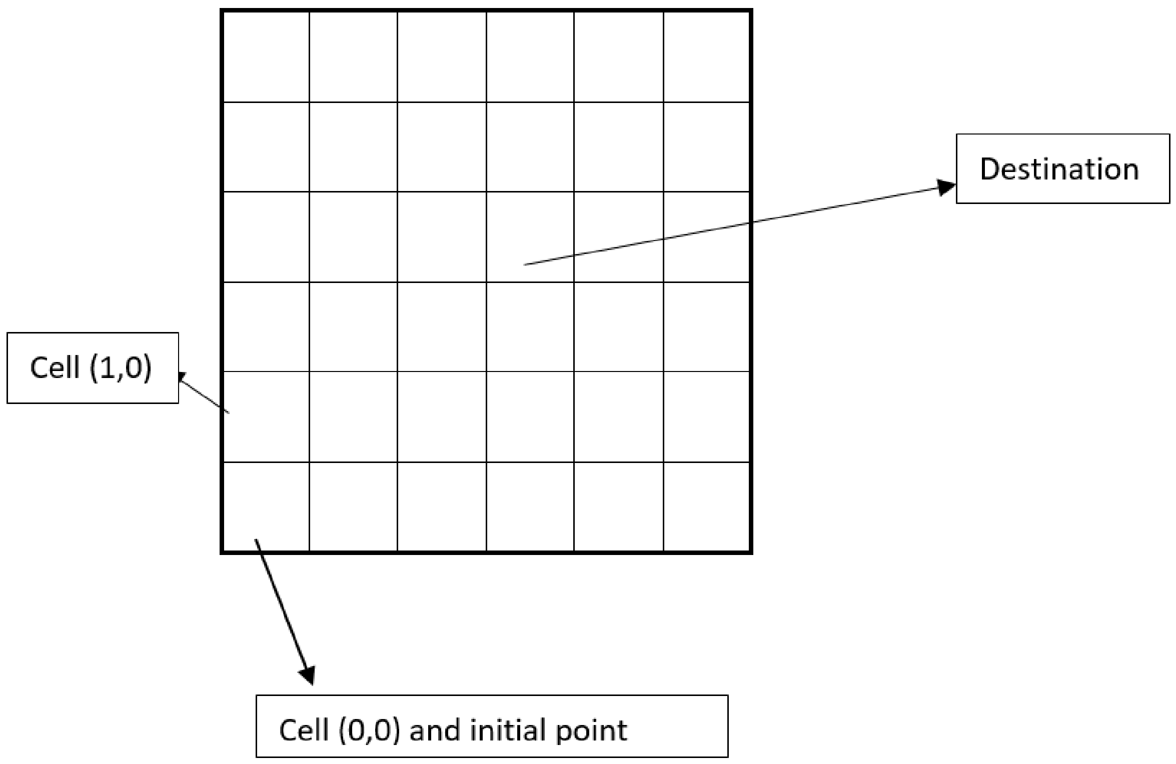
Disadvantage: For the algorithm to work, the beginning and end of the maze must be defined.

3. Overall explanation about FF

FF operates based on the principle of water movement in an area, it will find the shortest path to the destination through value processing. With assigning values, the principle is to go from high value to low value.
Initially, after determining the size of the maze, the starting and ending positions of the journey. Assume that we don't know the walls in the maze, only the walls outside them. We start assigning initial values to the cells based on the FF value,

with the destination point with value 0, and from this point we will fill in the values +1 for the next cell.
There are 4 cells next to the original 1 cell. If it is blocked, there is no need to fill in the value. Continue until all initial values have been filled in.

Then, as the vehicle drives, they will update the walls if encountered. We will use 3 ultrasonic sensors to read whether or not there are walls and save the data. At the same time, the car will check to see whether in the neighboring cells, the cell with the smallest value has a value equal to the value of the current cell -1. If yes, skip the update and move to the smaller cell, otherwise we have to update the current cell value with the smallest neighbor cell value + 1, and also need to update the cells affected by Their value will be different.
Here is more detail about the process.
Take the example of a 6x6 maze.



We shall use the matrix notation (x, y) for the cell location where x is the row value, x = 0 is the bottom row, and x = 5 is the top row; y is the column value, y = 0 is left-most column, and y = 5 is the right-most column. The micromouse starts at cell (0, 0) which is the cell at the lowest left-hand corner. Initially, the micromouse is placed at (0, 0) facing upward, as shown by the arrow. Before the micromouse starts its exploration, the maze is assumed to have no walls and each cell is assigned a value based on the number of cells distance from the center. The figure below shows the initial cells' distance values for a 6x6 cell maze.

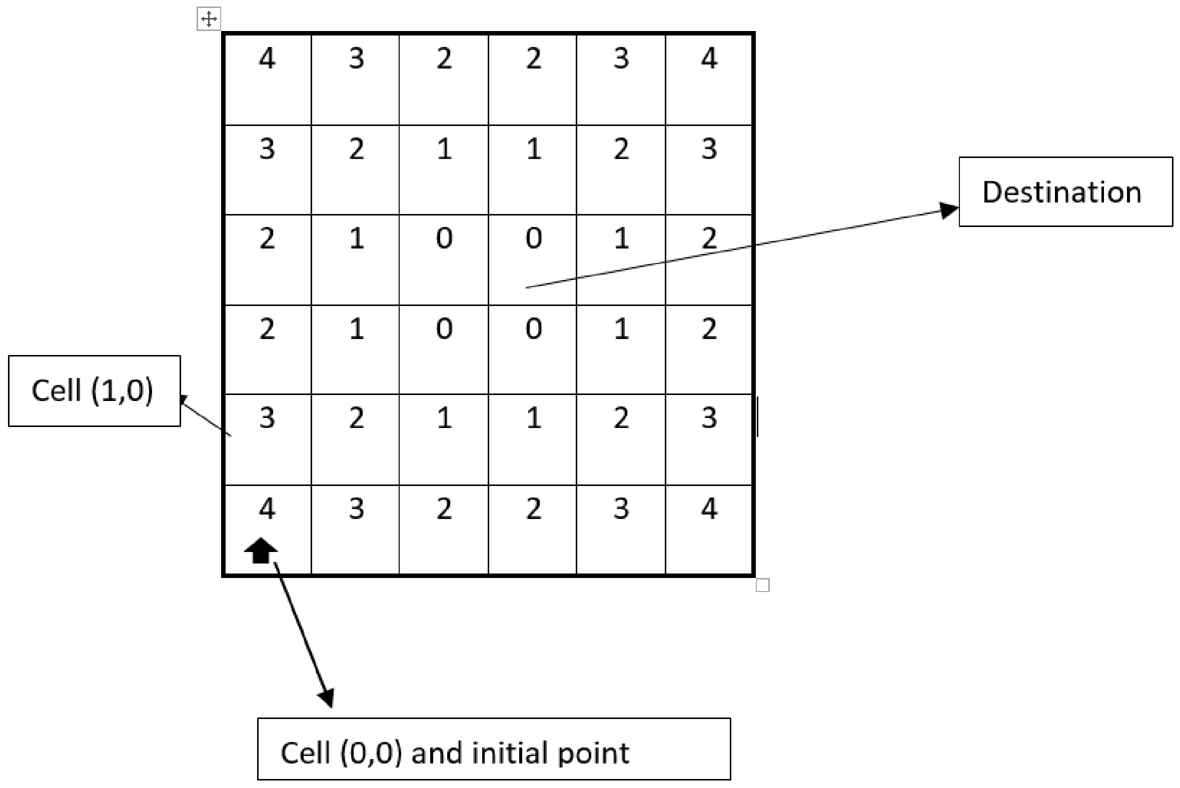


Fig. 1b. Initial distance values from center cells.

At cell (0, 0), the cell has only one open neighbor (1, 0) which has a smaller distance from the center. Hence the micromouse will move upward (north), from a higher elevation to a lower one.

With the new wall detected north of cell (1, 0) and the old wall east of cell (0, 0), the Flood Fill algorithm floods the maze with the distances from the center cells.

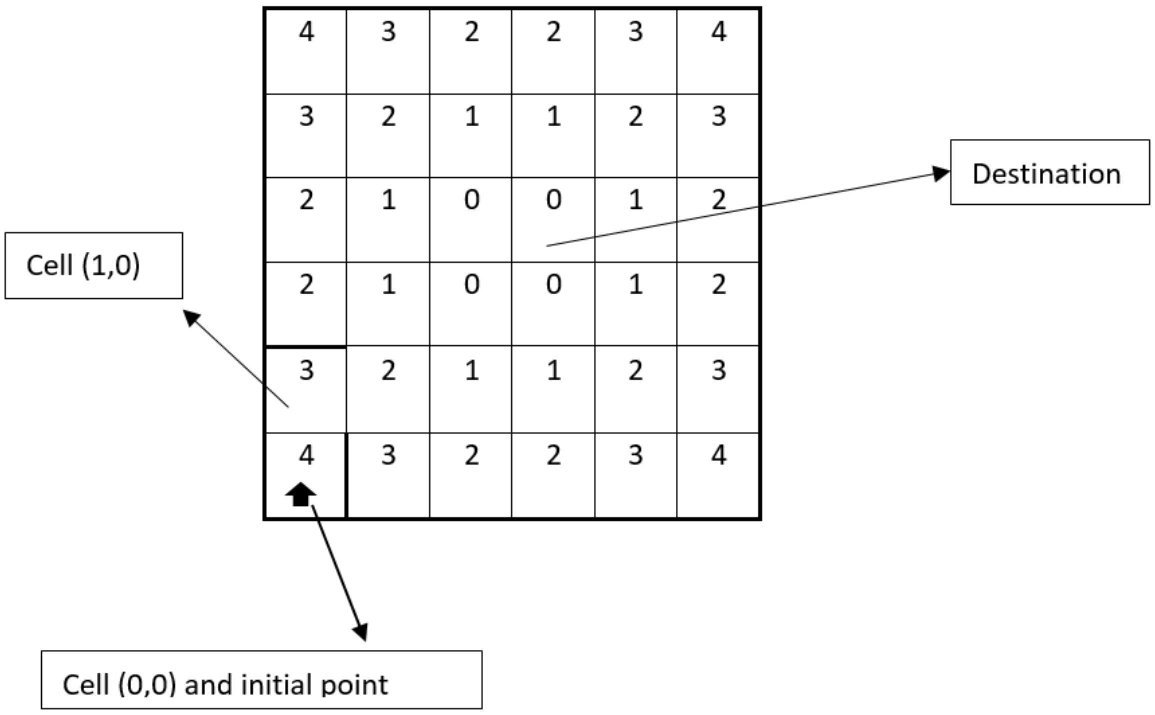


Fig. 1c. Distance values when micromouse reaches cell (1, 0).

At cell (1, 0), the micromouse encounters the north wall. This cell has two open neighbors (0, 0) and (1, 1). Since the open east neighbor (1, 1) has a distance smaller than cell (1, 0), the micromouse will turn and move to (1, 1) eastwards, again from a higher elevation to a lower one, as shown in Fig. 1d. With the new wall detected east of cell (1, 1) and the old walls north of cell (1, 0) and east of cell (0, 0), the Flood Fill algorithm floods the maze with the distances from the center cells as shown in Fig. 1d. Refer to Fig. 1a for the other walls which are yet to be detected.

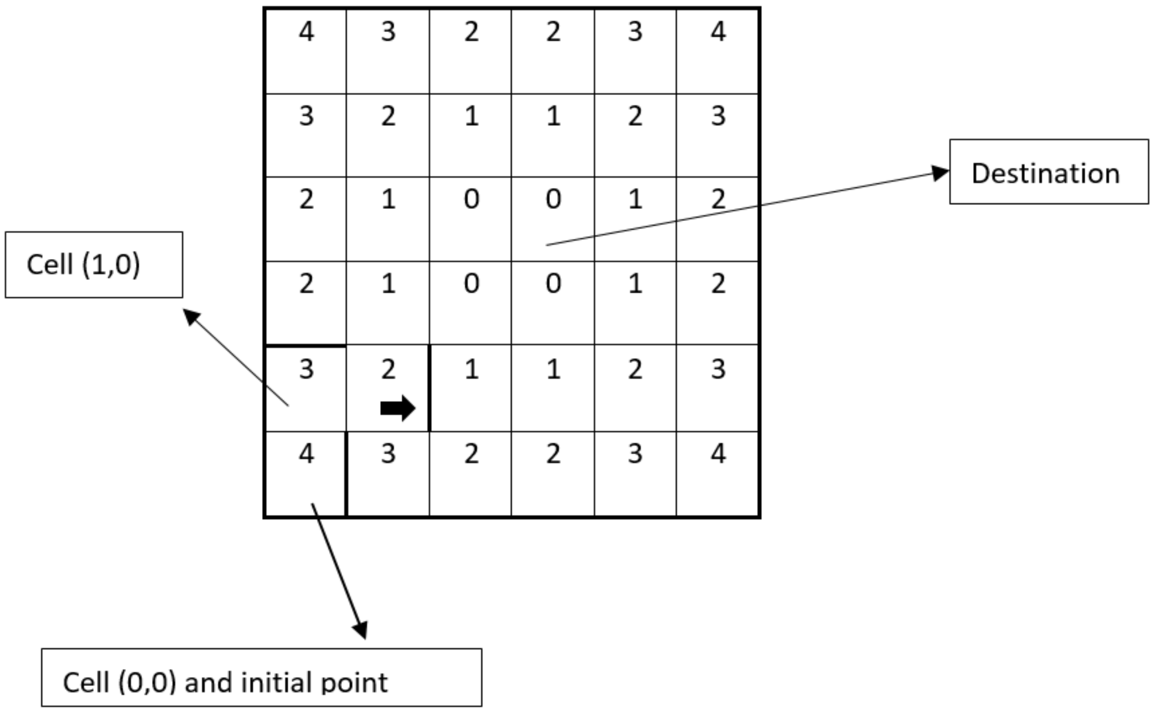


Fig. 1d. Distance values when micromouse reaches cell (1, 1)

At cell (1, 1), the micromouse encounters the east wall. This cell has three open neighbors (0, 1), (2, 1) and (1, 0). Since the open north neighbor (2, 1) has a distance smaller than cell (1, 1), the micromouse will turn and move to (2, 1) northwards, again from a higher elevation to a lower one, as shown in Fig. 1e. With the new walls detected north of cell (2, 1) and east of cell (2, 1), and the old walls east of cell (1, 1), north of cell (1, 0), and east of cell (0, 0), the Flood Fill algorithm floods the maze with the distances from the center cells as shown in Fig. 1e.

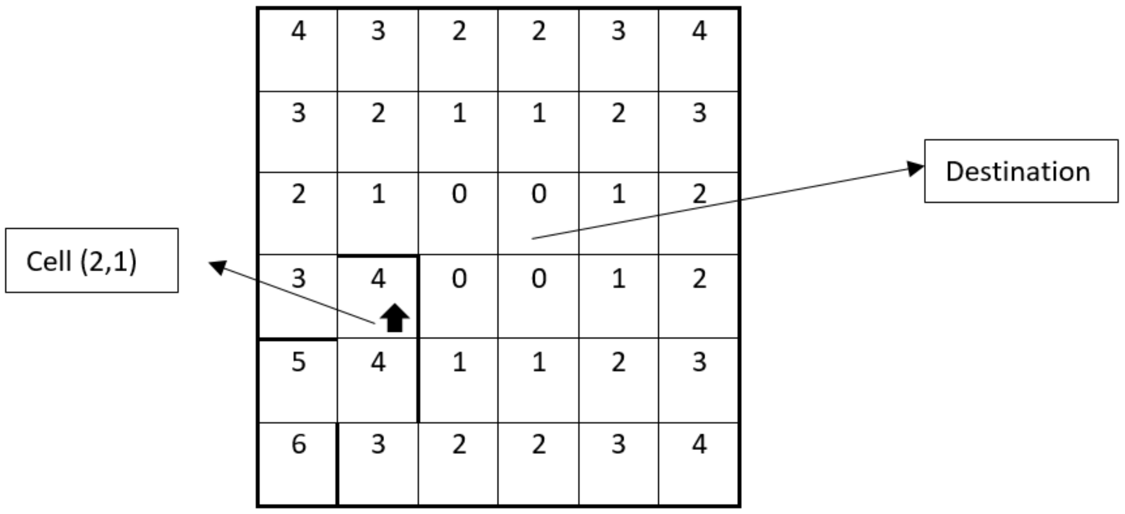


Fig. 1e. Distance values when micromouse reaches cell (2, 1).

The same process continues until the micromouse reaches the center cell.

4. Modified Flood Fill Algorithm.

The modified flood fill algorithm does not flood the maze each time a new cell is reached. Instead it updates only the relevant neighboring cells using the following revised recursive steps:

1) Push the current cell location (x, y) onto the stack.
2) Repeat this step while the stack is not empty.
Pull the cell location (x, y) from the stack.
If the minimum distance of the neighboring open cells, md, is not equal to the present cell's distance - 1, replace the present cell's distance with md + 1, and push all neighbor locations onto the stack.

Like the flood fill algorithm, the maze is first initialized (flooded) with the distances from the center with the assumption that there is no wall in the maze. Since there is no distance update until the micromouse reaches cell (2,1), the distance values remain unchanged as in Fig. 2a (same as Fig. 1d). Recall that neighboring cells' distances are updated only if the condition specified in the recursive step 2b is satisfied.

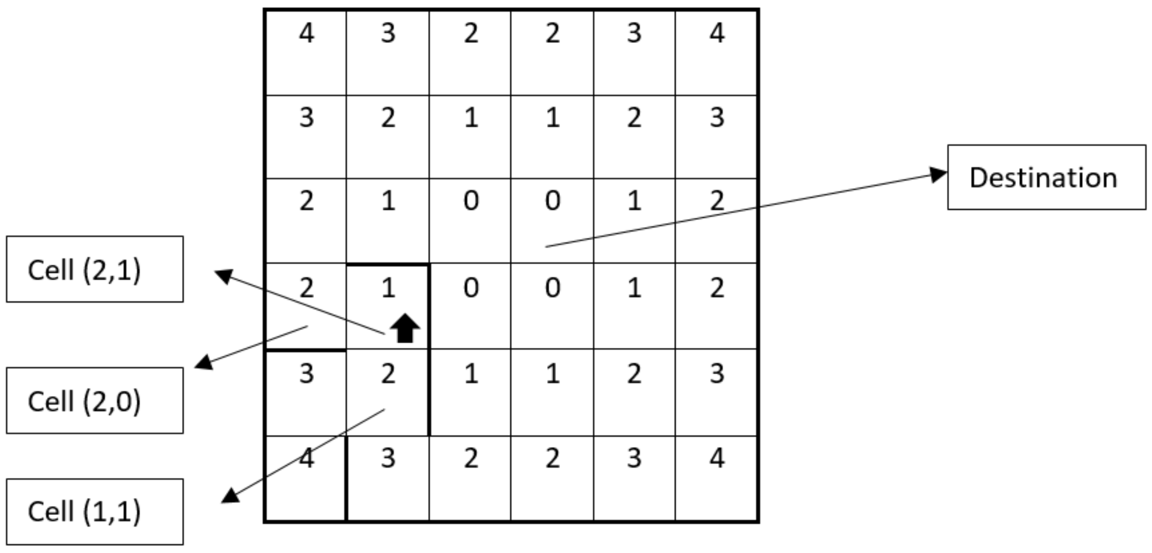


Fig. 2a. Distance values when micromouse reaches cell (2, 1) before distance update.

Since the distance of the current cell (2, 1) – 1 = 0 is not equal to the minimum of the open neighbors (2, 0) and (1, 1), which is 2, the distance update is necessary. We shall follow the revised recursive distance update steps to see how the distance values are updated.

- Pull the cell location (2, 1) from the stack.ko

- Since the distance at (2, 1) – 1 = 0 is not equal to md = 2, the minimum of its open neighbors (2, 0) and (1, 1), update the distance at (2, 1) to md + 1 = 2 + 1 = 3. Push all neighbor locations (3, 1), (2, 0) and (1, 1), except the center location (2, 2), onto the stack. Fig. 2b shows the updated distances and Fig. 2c shows the current stack contents.

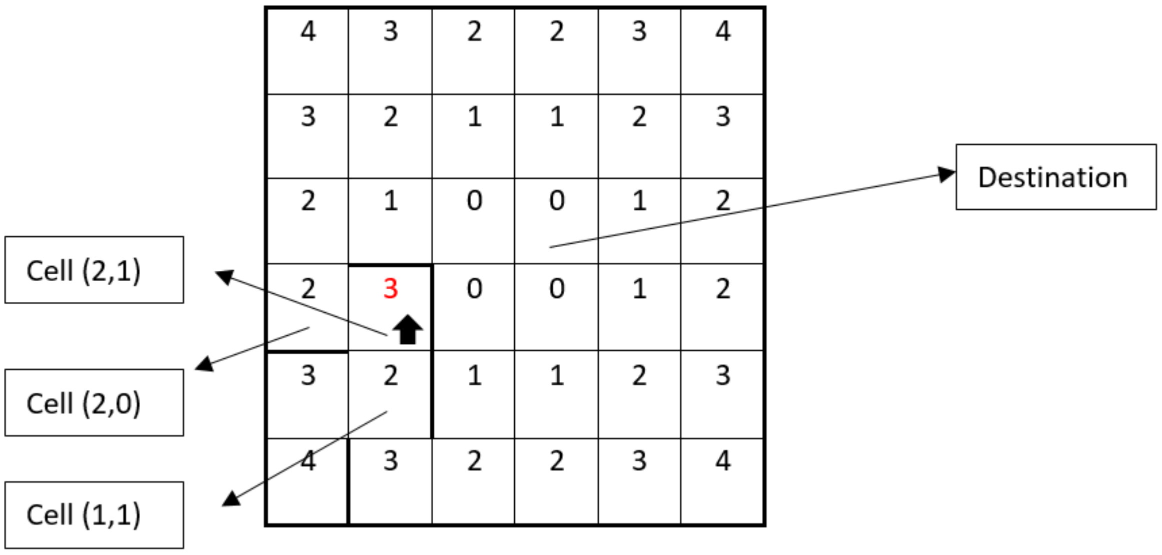


Fig. 2b. Distance values when micromouse reaches cell (2, 1) after distance update at cell (2, 1).

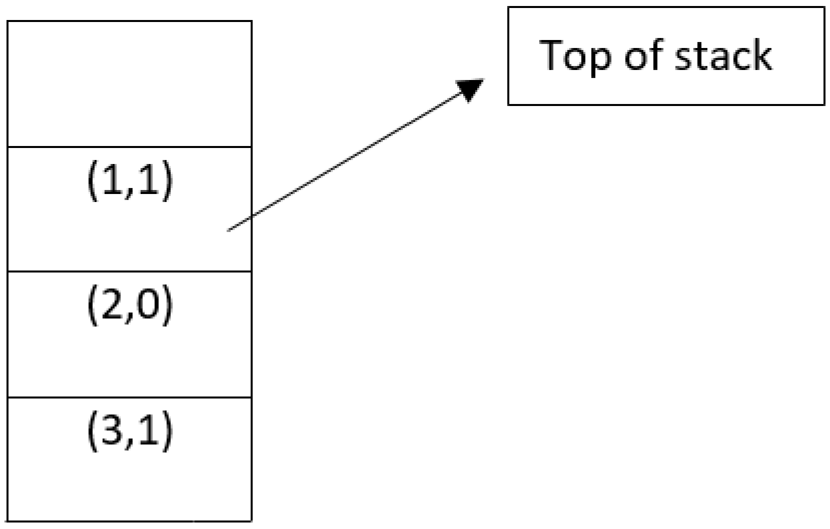


Fig. 2c. Contents of Stack when micromouse reaches cell (2, 1) after distance update at cell (2, 1).

- Recursively, since the stack is not empty, pull the cell location (1, 1) from the stack.

- Since the distance at (1, 1) – 1 = 1 is not equal to md = 3, the minimum of its open neighbors (2, 1), (0, 1) and (1, 0), update the distance at (1,1) to md + 1 = 3 + 1 = 4. Push all neighbor locations (2, 1), (0, 1), (1, 0), and (1, 2) onto the stack. Fig. 2d shows the updated distances and Fig. 2e shows the current stack contents.

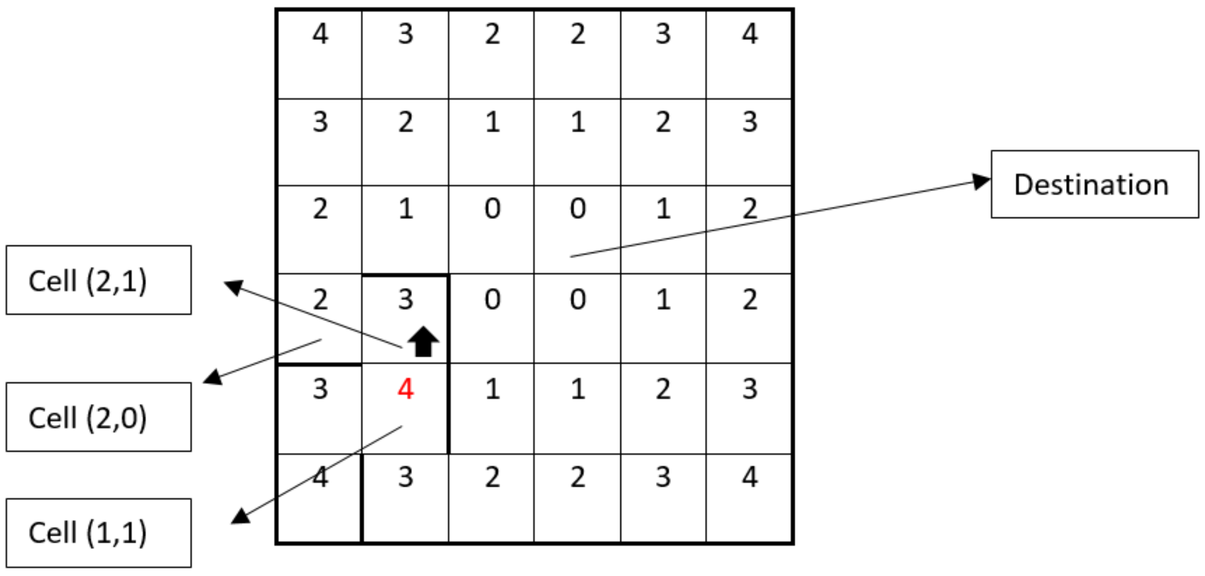


Fig. 2d. Distance values when micromouse reaches cell (2, 1) after distance update at cell (1, 1).

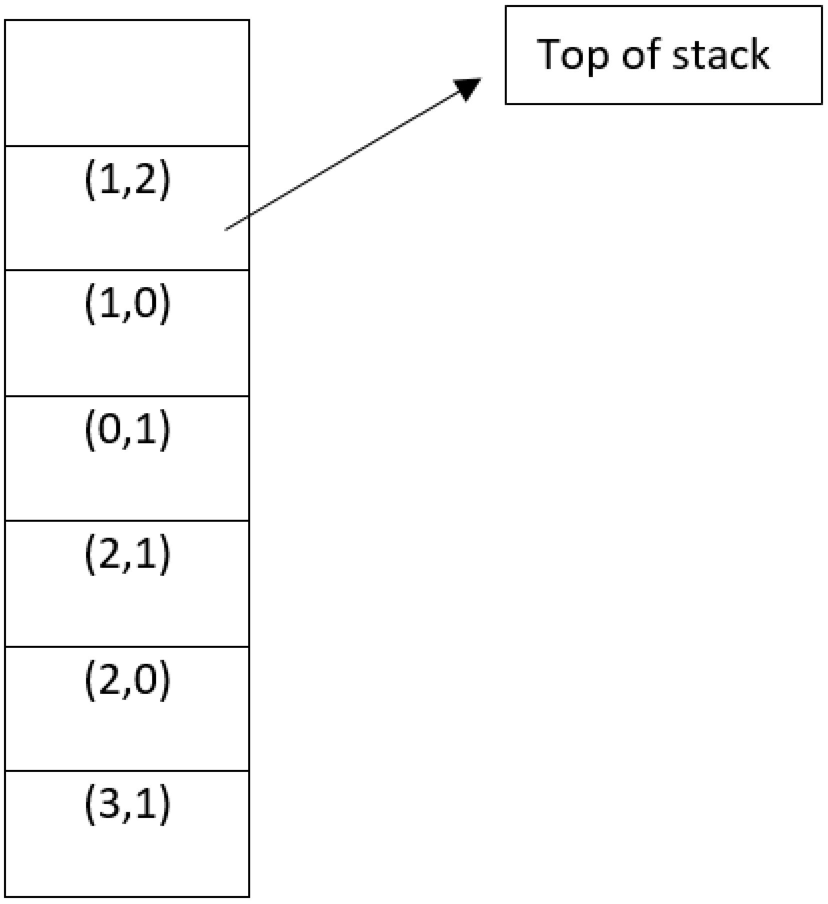


Fig. 2e. Contents of Stack when micromouse reaches cell (2, 1) after distance update at cell (1, 1).

- Recursively, since the stack is not empty, pull the cell location (1, 2) from the stack.
- Since the distance at (1, 2) – 1 = 0 is equal to md = 0, the minimum of its open neighbors (2, 2), (0, 2) and (1, 3), no distance update is necessary.
- Recursively, since the stack is not empty, pull the cell location (1, 0) from the stack.
- Since the distance at (1, 0) – 1 = 2 is not equal to md = 4, the minimum of its open neighbors (0, 0) and (1, 1), update the distance at (1, 0) to md + 1 = 4 + 1 = 5. Push all neighbor locations (2, 0), (0, 0), and (1, 1) onto the stack.

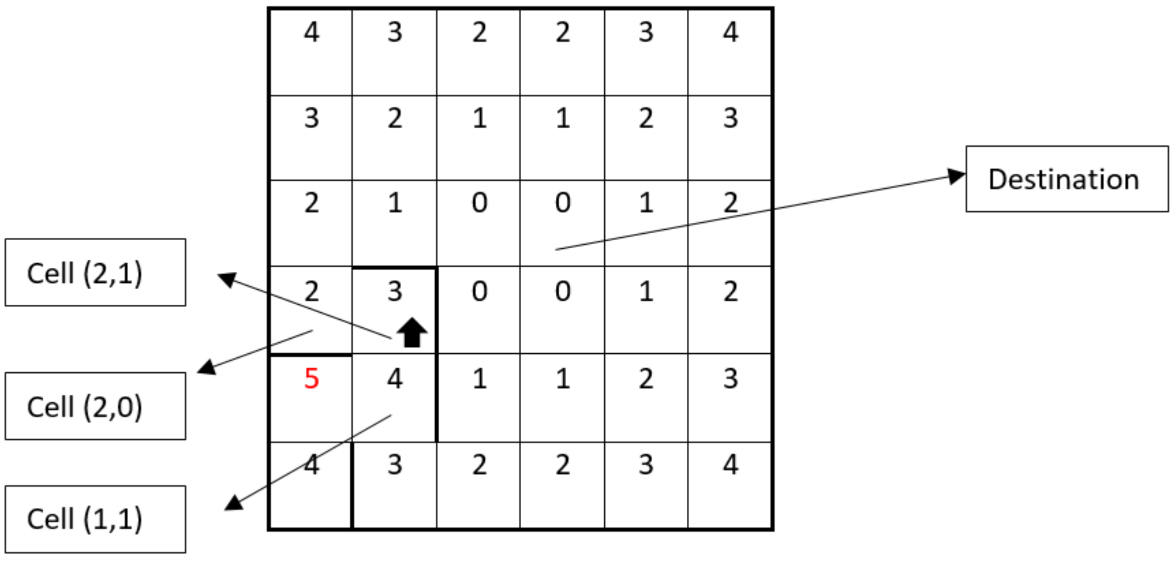


Fig. 2f. Distance values when micromouse reaches cell (2, 1) after distance update at cell (1, 0).

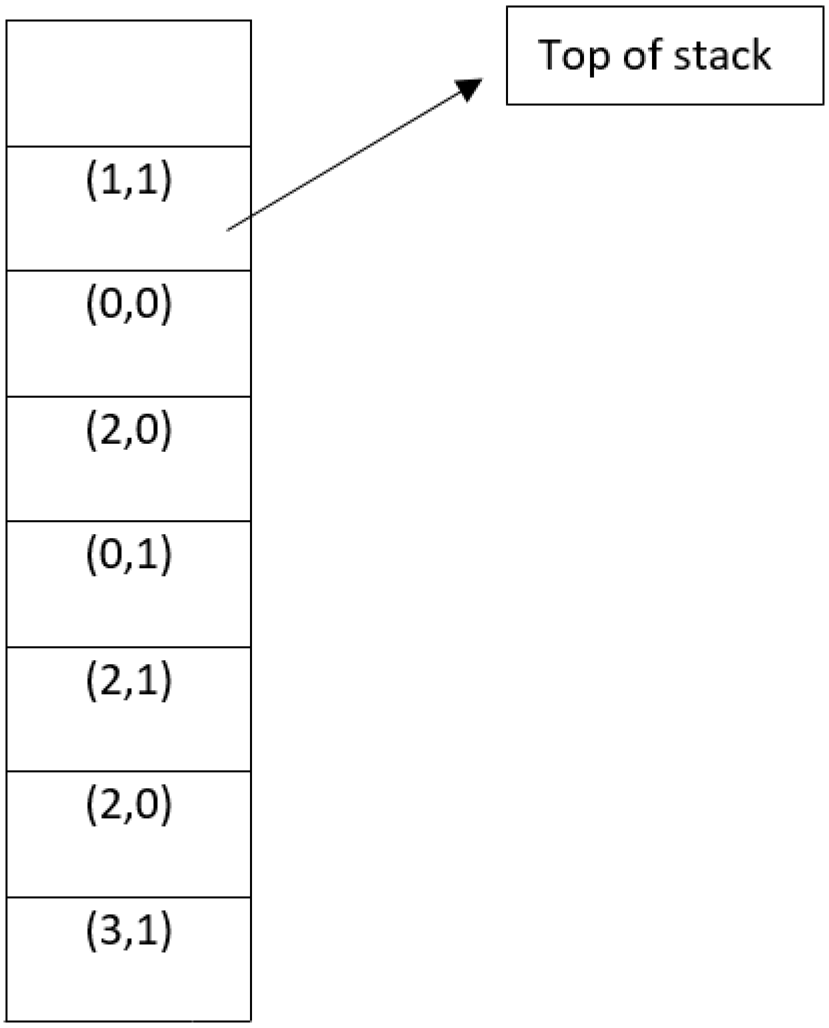


Fig. 2g. Contents of Stack when micromouse reaches cell (2, 1) after distance update at cell (1, 0).

Recursively, since the stack is not empty, pull the cell location (1, 1) from the stack.

Since the distance at (1, 1) – 1 = 3 is equal to md = 3, the minimum of its open neighbors (1, 0), (2, 1) and (0,1), no distance update is necessary.

Recursively, since the stack is not empty, pull the cell location (0, 0) from the stack.

Since the distance at (0, 0) – 1 = 3 is not equal to md = 5, the minimum of its only open neighbor (1, 0), update the distance at (0, 0) to md + 1 = 6. Push all neighbor locations (1, 0) and (0, 1) onto the stack. Fig. 2h shows the updated distances and Fig. 2i shows the current stack contents.

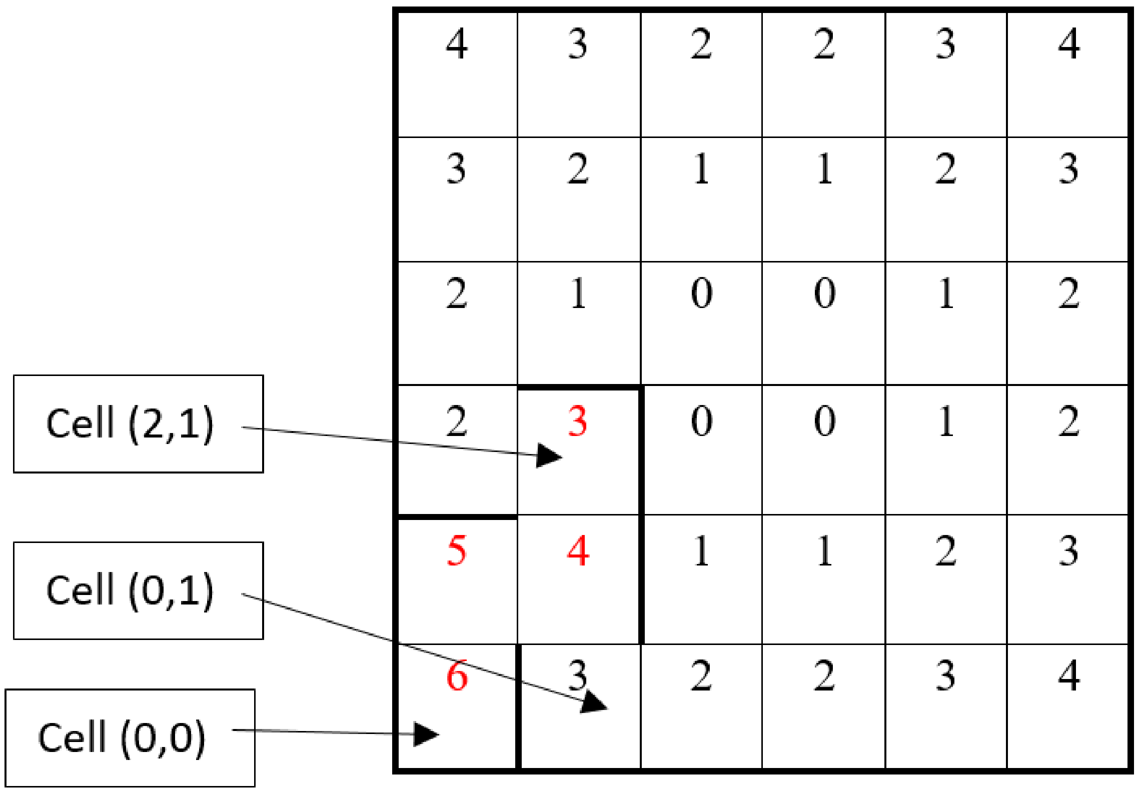


Fig. 2h. Distance values when micromouse reaches cell (2, 1) after distance update at cell (0, 0).

We shall not show the similar steps for the remaining cell locations in the stack. One can follow the previous steps, to obtain the distance values. When the stack is empty, the distance map of the maze will look like Fig. 2j

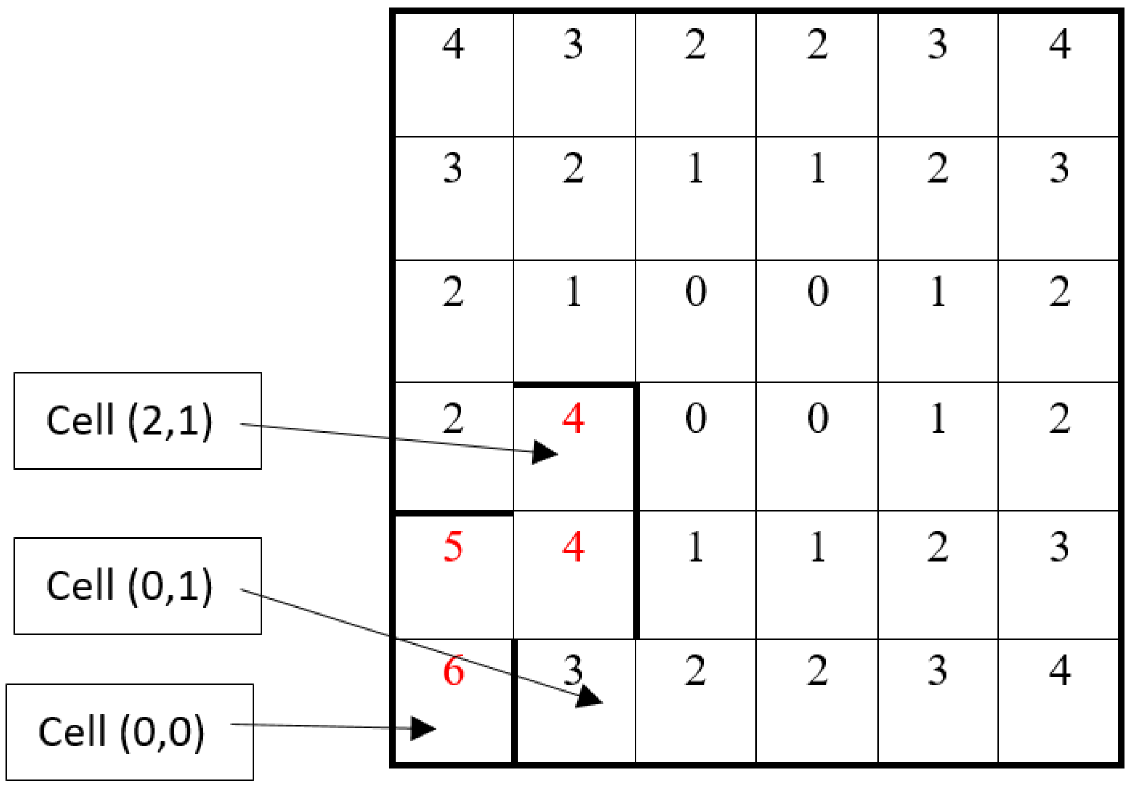


Fig. 2j. Distance values when micromouse reaches cell (2,1) after all neighbor cells' distances have been updated.

The same process continues until the micromouse reaches the center cell. Fig. 2k shows the distance map when the micromouse reaches the center cell. Overall, to reach the center on the first run for this sample maze, the total number of distance updates is 36, which is obtained from the maze-solver's tabulated value.

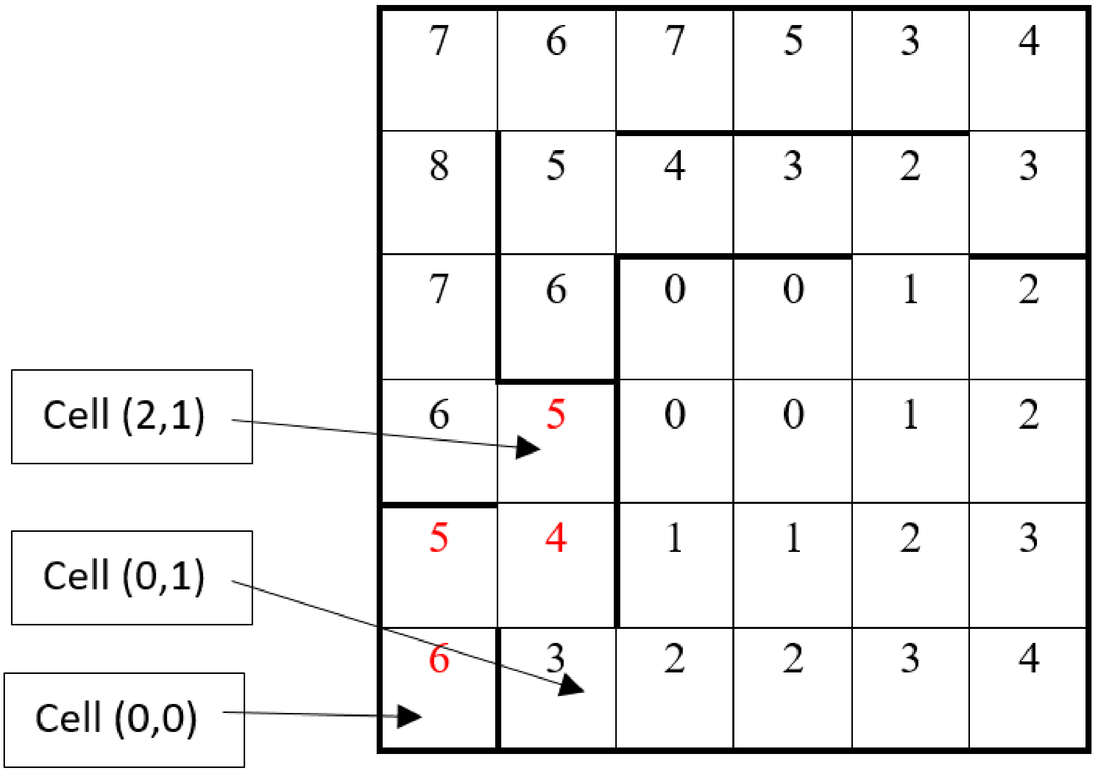


Fig. 2k. Distance values when micromouse reaches the center cell on the 1st run.

5. **Comparison between Flood Fill Algorithm and Modified Flood Fill Algorithm.**

The Modified Flood Fill algorithm provides appreciably less cell distance updates than the Flood Fill technique. The micromouse that utilizes the improved flood fill may move between cells faster and with less processing effort because there are less distances to update. Moreover, the total number of cells explored will be lowered if the dead end cells are noted at first arrival as they won't be investigated in later trips. little processing power is required, and the least number of cells visited during the optimal run is guaranteed.

PART III: DESIGN

Overall, in this project, our team will use a modified Flood Filling Algorithm to control the car. We will receive and read actual values through 3 ultrasonic sensors, thereby providing appropriate parameters for the motors, as well as the central algorithm processor. The table below shows our hardware and its description.

Hardware			
Order	**Name**	**Quantities**	**Description**
1	**Kit STM32**	**1**	
2	**Ultrasonic sensor**	**3**	**Working voltage: 5 VDC** **Static current: < 2mA.** **Output signal: electrical frequency signal, high 5V, low 0V.** **Variable angle induction: No more than 15 degrees.** **Detection distance: 2cm ~ 450cm.** **High precision: Up to 3mm** **Connection mode: VCC / trig (T) / echo (R) / GND**

3	Encoder motor	2	Transmission ratio 30:1 (the motor rotates 30 revolutions, the main shaft of the gear box rotates 1 revolution). No-load current: 200mA Maximum current under load: 5A No-load speed: 208RPM (208 rounds per minute) Maximum speed under load: 176 RPM (176 rounds per minute) Rated pulling force Moment: 2.4KG.CM Maximum Moment Force: 15KG.CM Gearbox length L: 22mm
4	Chassis	2	Mica chassis
5	Pin	4	3.7/4.2V 6800mAh
6	LM2576	1	Decrease to 5V
7	L298N	1	
8	Others (wires, breadboard,…)		

I.Hardware:

1. **Design requirements:**

The circuit runs stably.

The product is aesthetic.
Efficient handling of real values.
Use components effectively.

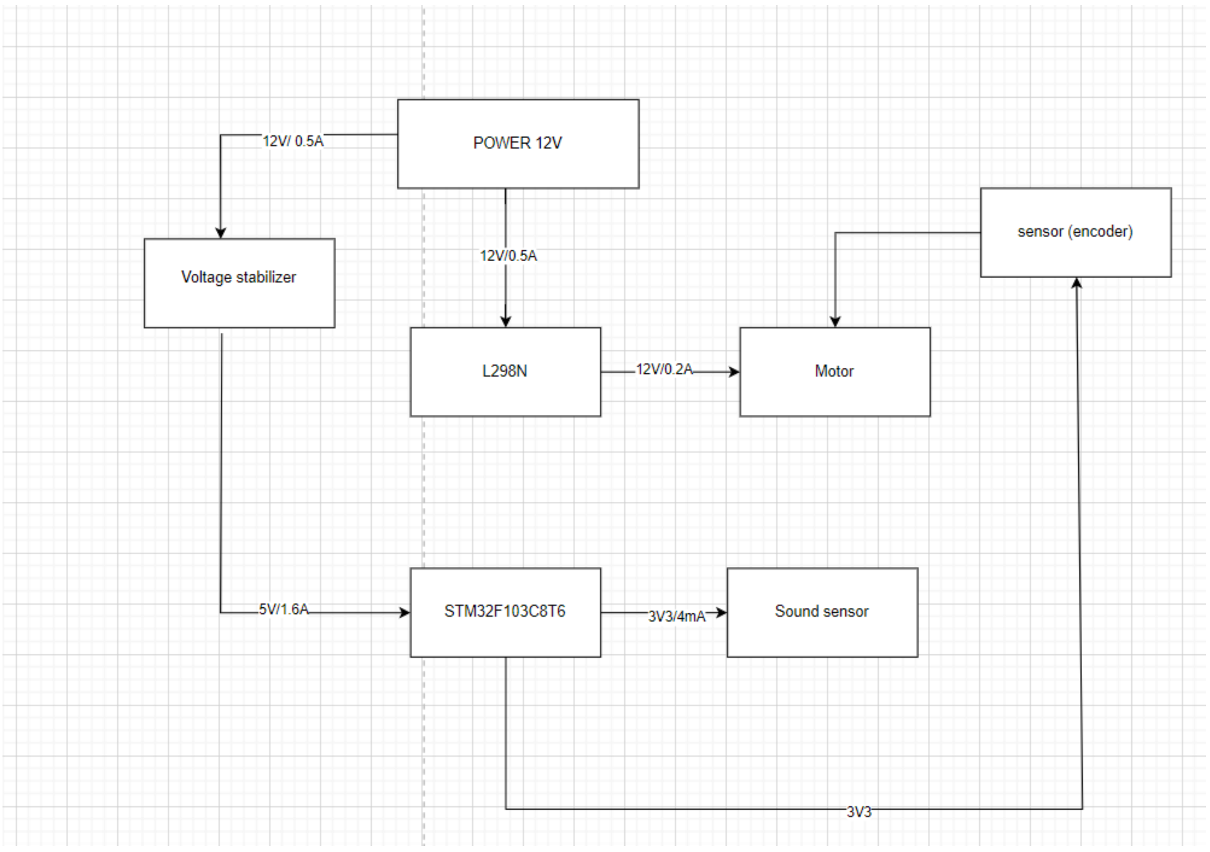
2. **Design analysis**

- Design of central control block: using STM32F205VCT6

Communication with ultrasonic sensors: communicate with microcontrollers via GPIO.
Communication with L298N: communication with control via GPIO with Timer function to pulse PWM output.

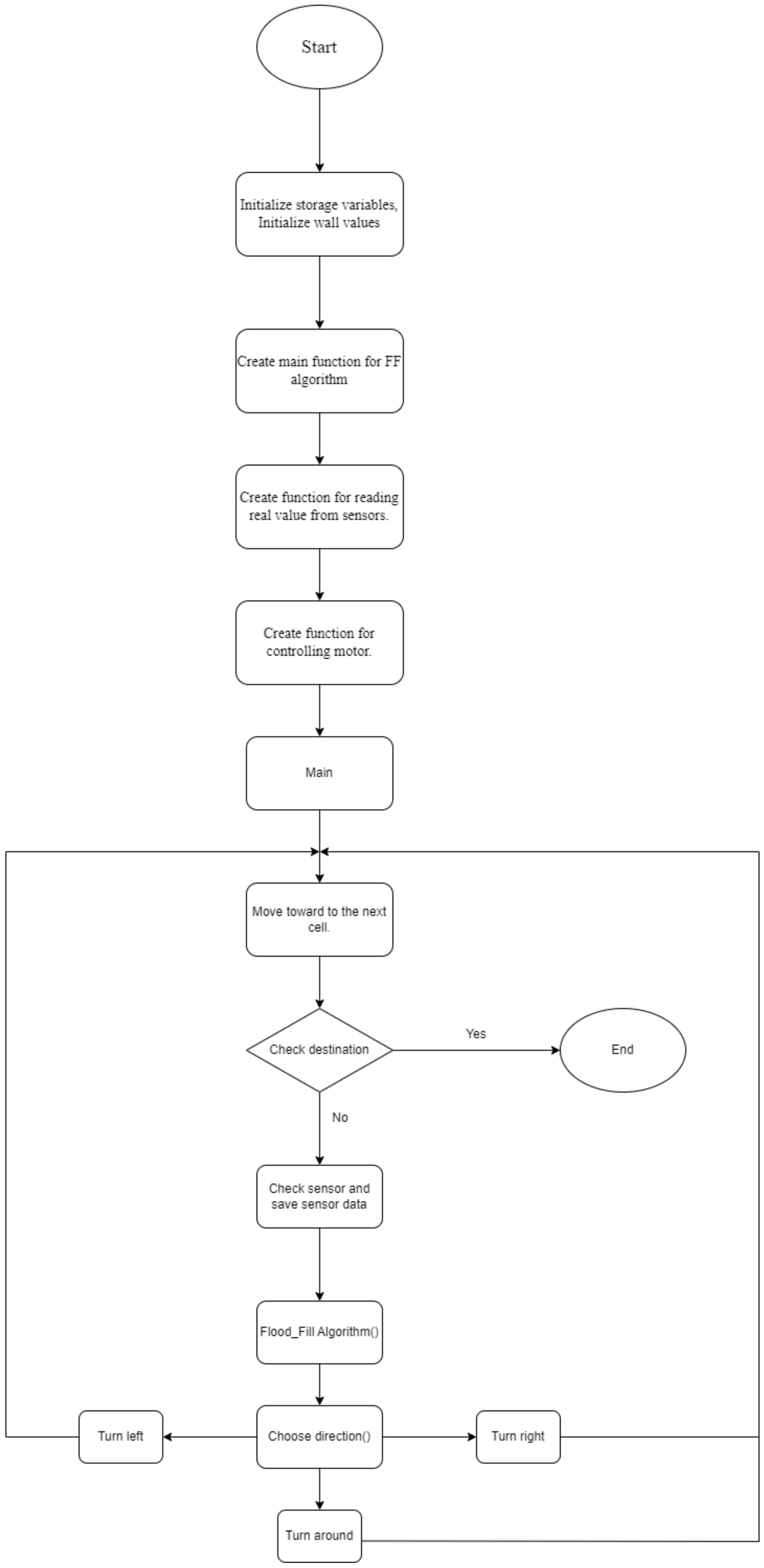
- Power block design:

With the 14.8-16.8V source taken from 4 batteries, the voltage is lowered to 5V through STM32 and 3V3 through the ultrasonic sensor.



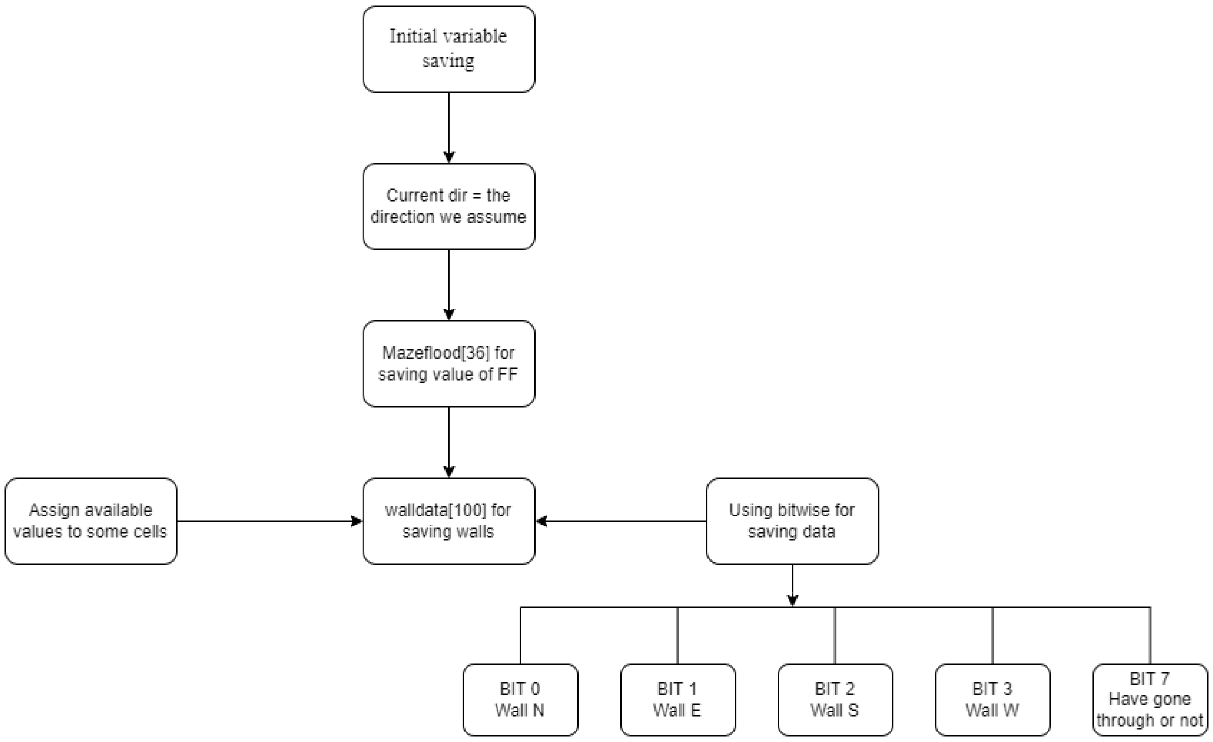
II.Software:

Overall algorithm flowchart

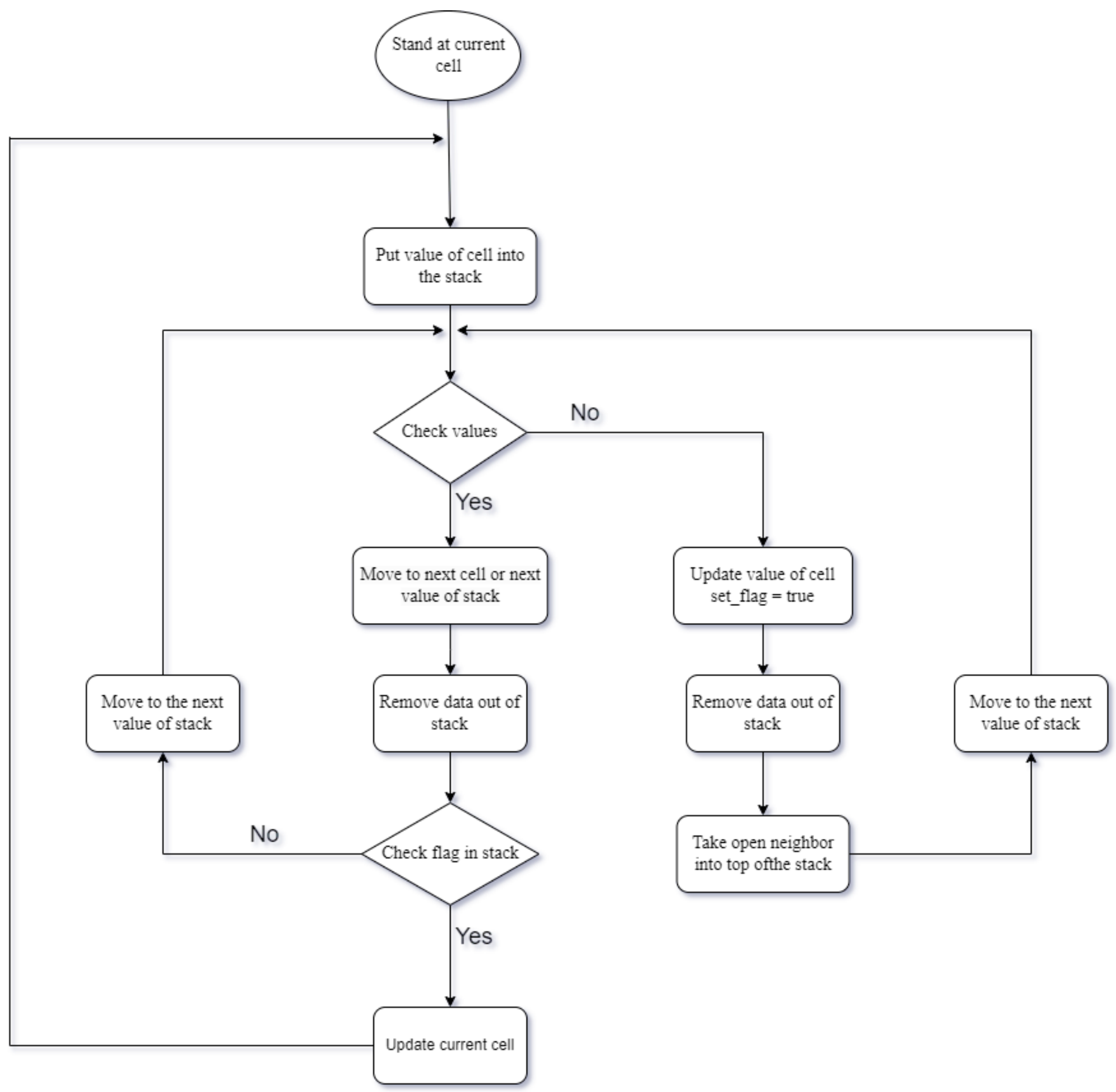

Start
Initialize storage variables, Initialize wall values
Create main function for FF algorithm
Create function for reading real value from sensors.
Create function for controlling motor.
Main
Move toward to the next cell.
Check destination
Yes
End
No
Check sensor and save sensor data
Flood_Fill Algorithm()
Turn left
Choose direction()
Turn right
Turn around

Specific algorithm flowchart.

Flowchart of initial variables:

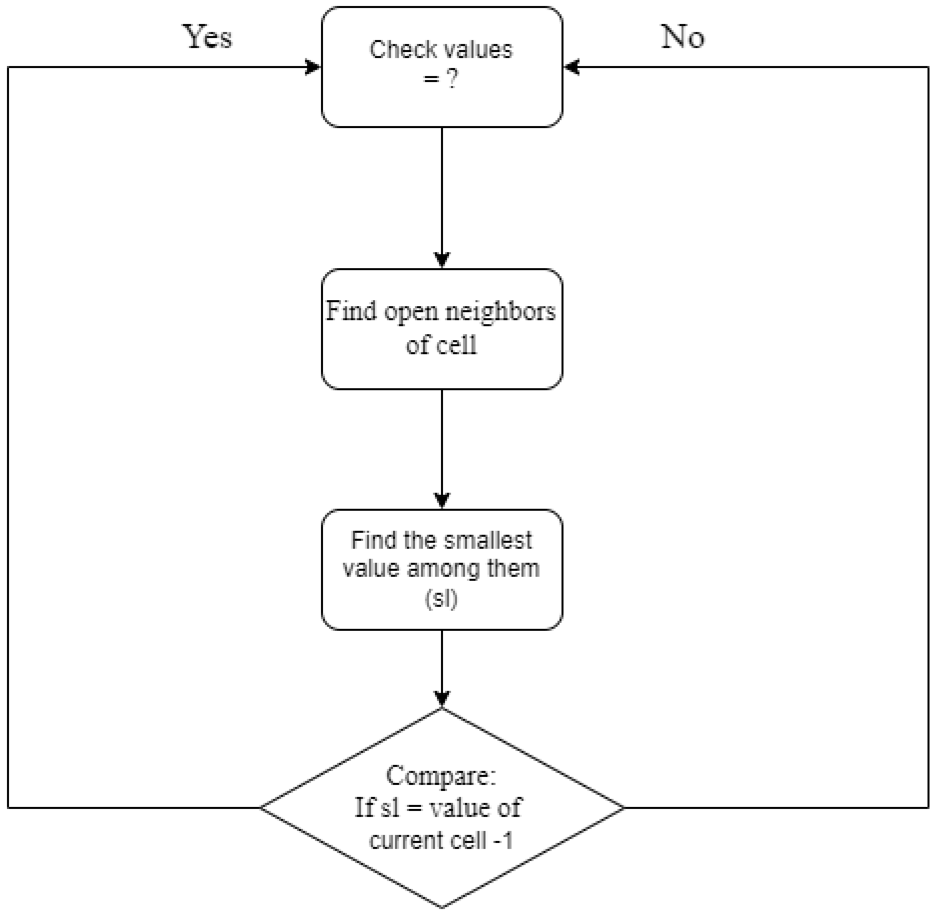

Initial variable saving
Current dir = the direction we assume
Mazeflood[36] for saving value of FF
Assign available values to some cells
walldata[100] for saving walls
Using bitwise for saving data
BIT 0 Wall N
BIT 1 Wall E
BIT 2 Wall S
BIT 3 Wall W
BIT 7 Have gone through or not

Flowchart of Flood_fill algorithm:

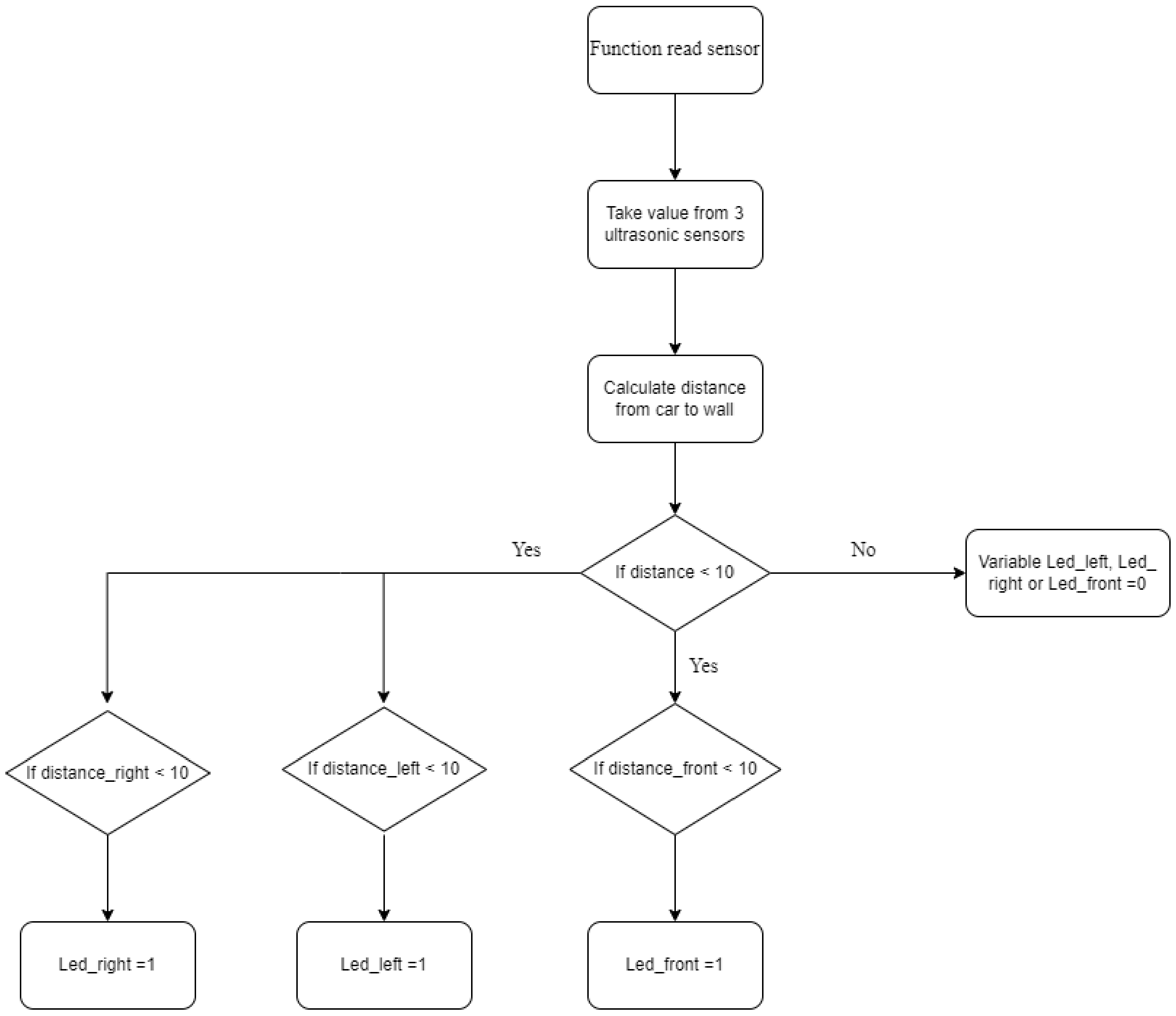


Stand at current cell
Put value of cell into the stack
Check values
No
Yes
Move to next cell or next value of stack
Update value of cell set_flag = true
Move to the next value of stack
Remove data out of stack
Remove data out of stack
Move to the next value of stack
No
Check flag in stack
Take open neighbor into top ofthe stack
Yes
Update current cell

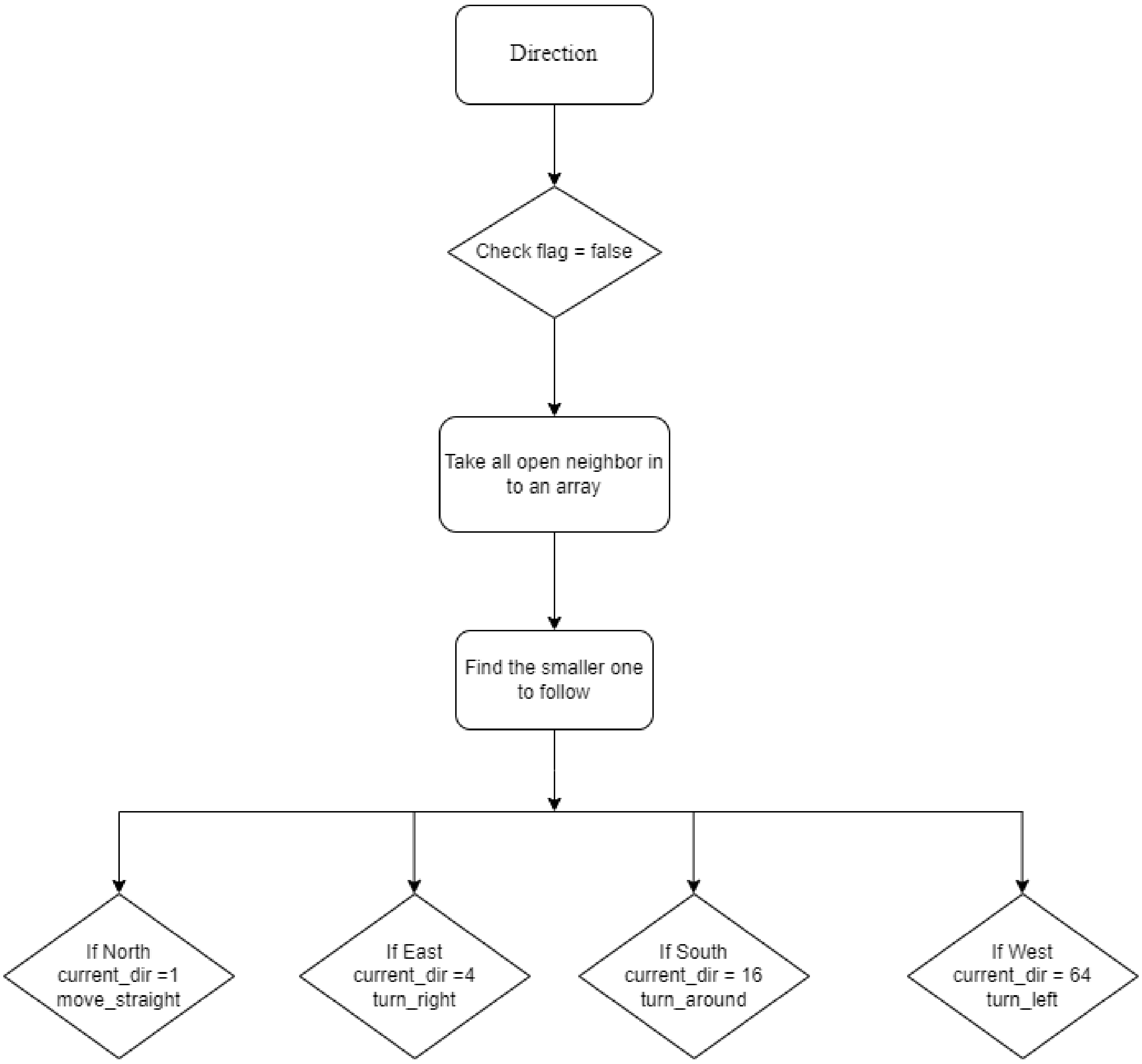
Note: The order of neighbor value putting in is not important.
If the stack is empty, set flag is true, else it is false.


Yes
Check values = ?
No
Find open neighbors of cell
Find the smallest value among them (sl)
Compare: If sl = value of current cell -1

Flowchart of reading ultrasonic sensor:



Flowchart of taking direction for car:


Direction
Check flag = false
Take all open neighbor in to an array
Find the smaller one to follow
If North current_dir =1 move_straight
If East current_dir =4 turn_right
If South current_dir = 16 turn_around
If West current_dir = 64 turn_left