CHAPTER 1

INTRODUCTION

1.1 Introduction

This chapter is going to provide an introduction about this project. These following topics have been described in this chapter, General Information about Autonomous Maze Solving Robot, Problem Statement, Objective of the project, Project Organization and Outline.

A maze is a network of paths, typically from an entrance to exit. The concept of Maze started approximately thousand years old which was invented in Egypt. Then, many mathematician made various algorithms to solve the maze.

Nowadays, maze solving problem is an important field of robotics. It is based on one of the most important areas of robot, which is “Decision Making Algorithm”. As this robot will be placed in unknown place, it requires to have a good decision making capability. There are many types of maze solving robot using various type of algorithms.

In this project, the system design of Maze solving robot consists of reflective infrared photoelectric sensors and then sensors will detect the black line. To solve the maze, this robot will apply wall following algorithms such as left or right hand rule. It will also follow the *Bellman Ford* algorithm for finding the shortest path.

1.2. Autonomous Maze Solving Robot

An autonomous robot is a category of robot that can perform tasks intelligently depending on themselves, without any human assistance. Maze Solving Robot is one of the most popular autonomous robots. It is a small self-reliant robot that can solve a maze from a known starting position to the center area of the maze in the shortest possible time. A maze solving robot make multiple runs in a maze, first it create a map of the maze layout and store it in its memory, then run through a shortest path.

1.3. Problem Statement

1. To design a hardware for maze solving robot.
2. To construct a software with the combination of wall following and Bellman-Ford algorithms
3. To implement the software in the hardware of maze solving robot.
4. At last, to make a maze 4×4 maze to verify the robot.

1.4 Objectives

- Understand and implement the wall follower and Bellman Ford algorithm.
- Design and build reflective infrared photoelectric sensors array.
- Build Arduino based hardware.
- Program the robot to solve the maze.
- Make a small 4×4 maze.

1.5 Project Outline

This project is divided into **7** chapters. Each chapter discusses in the different issues which related to this project. The outline of each chapter is stated in the paragraphs below.

Basic introduction about autonomous robot, maze solving robot have been described in **Chapter 1**. Project statement and objectives of the project have also been described in this chapter. **Chapter 2** covers the important background information and history about the Maze Solving Robot. Many important theories, method and algorithms on maze solving problem is also given there. This section will start with providing some research about Wall Following Algorithm and Bellman Ford Algorithm. **Chapter 3** is going to give some details for each component including features, specifications and how it operates. Important Algorithms which is applied in this project is described in **Chapter 4**. Methodology, Software development, Hardware implementation is described in **Chapter 5.** In **Chapter 6,** simulated result and important discussion is described. Conclusion and Further development is described in **Chapter 7**.

CHAPTER 2

LITERATURE OVERVIEW

This chapter is going to provide important background information and history about the Maze solving robot. Many important theories, method and algorithms on maze solving problem is also given there. This section will start with providing some research about Wall Following Algorithm and Bellman Ford Algorithm.

2.1 Background History

In the middle of the 20th century, Maze solving problems become an important field of robotics. In the year of 1972, editors of IEEE Spectrum magazine came up with the concept of micromouse which is a small microprocessor controlled vehicle with self-intelligence and capability to navigate a critical maze. Then in May 1977, the fast US Micro mouse contest, called “Amazing Micromouse Maze Contest” was announced by IEEE Spectrum. From then, this type of contest became more popular, and many type of maze solving robots are developed every year.

Late 1970s the designs of the maze solving robots designs were used to have huge physical shapes that contain many blocks logic gates. Figure 1.1 and 1.2 show the example of early the maze solving robots (micro mouse). Due to technological development the physical size of the robot becomes smaller and the features of the robot becomes modern.

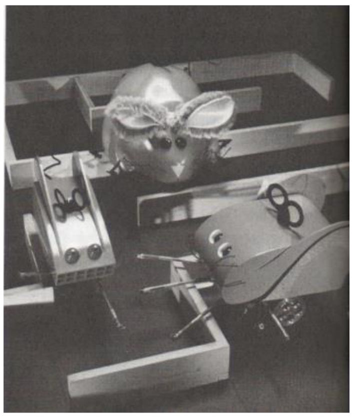


Figure-2.1

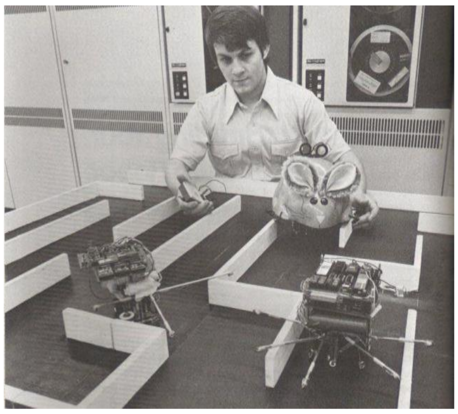


Figure-2.2

Figure-2.1 & 2.2: Old generation of micro-mouse robots.

2.2 Pulse Width Modulation

For controlling the motors speed, pulse width modulation (PWM) is used. Pulse width modulation is a simple method of controlling analogue devices via a digital signal through changing or modulating the pulse width. An analogue device is become digital by powering it with a pulse signal switches between on and off (5V and 0V). This digital control is used to create a square wave. The duty cycle is defined as the percentage of digital 'High' to digital 'Low' plus digital 'High' pulse-width during a PWM period. The average DC voltage value for 0% duty cycle is zero; with 25% duty cycle the average value is 1.25V (25% of 5V). With 50% duty cycle the average value is 2.5V, and if the duty cycle is 75%, the average voltage is 3.75V and so on. So, by varying the average voltage, the motor speed can be controlled.

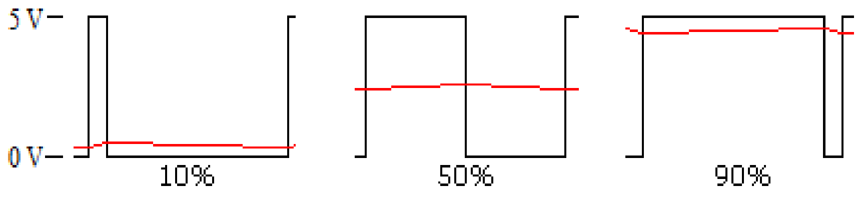


Figure-2.5: Example of PWM wave.

The power loss in PWM switching devices is very low. In many cases it is near to 0. So, this is the main advantage of PWM. While being used, resistors will tend to loss more power because of its heat dissipation. So, PWM is efficient in controlling motors.

CHAPTER 3

HARDWARE DESCRIPTION

This chapter covers the important information about all hardware which is used in this project. It is going to give some details for each component including features, specifications and how it operates.

3.0 Constructed Maze Solver

This project consists of several hardware components such as Arduino, reflective infrared photoelectric sensors, motor driver, voltage regulator, batteries, etc. In figure 3.1 show the full project.

Figure-3.1: Autonomous Maze solving Robot

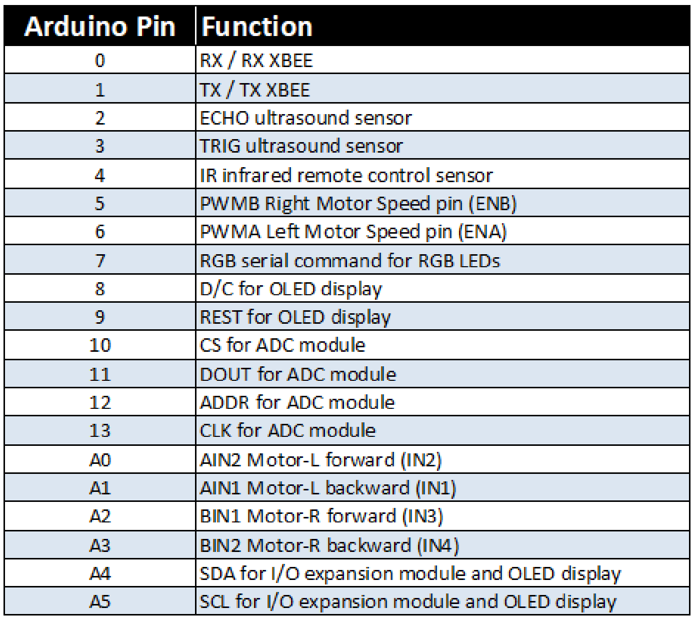
3.1 AlphaBot2

AlphaBot2 is composed of two circular frames, doubling as printed circuit and chassis for the robot. The bottom one has two motor reducers with high precision, low noise metal cogs, commanded by a double H-bridge driver with high-efficiency based on the TB6612FNG, the wheels are 42 x 19 mm, made out of high grip rubber. We still have five ITR20001/T infrared reflection linear sensors, two ST188 front infrared obstacle sensors, and an HC-SR04 ultrasound sensor. The hardware is finished off by four Neopixel LEDs, independently and serially controlled.

Anyways, these are SMD latest generation components (except for the ultrasound sensor) of the highest quality. The power is provided by two, 3.7 V 14500 lithium polymer batteries, with 800 mAh capacity, providing long working life with reduced weight and encumbrance. There is no internal recharge in the circuit but charging circuit, therefore, once the batteries are dead, the bot must be plugged in.

The upper frame has a housing for the Arduino board (not included in the kit), 0.66 inches, 128×64 yellow/blue bicolour OLED display, a TLC1543 AD converter, a joystick, a buzzer and an expansion module for I/O is based on the integrated PCF8574. The latter component is used to optimize the use of the I/O lines of the Arduino board, which are not enough to handle the many onboard peripherals.

On the upper part, there are also all the classic connectors of an Arduino board and an XBee-compatible connector. A connector called Control JMP is also available, allowing to redefine the features of the Arduino pin in relation to the hardware on the board. For all the details on the functionalities of the Arduino pin, please look at **Table 1**.



Arduino Pin	Function
0	RX / RX XBEE
1	TX / TX XBEE
2	ECHO ultrasound sensor
3	TRIG ultrasound sensor
4	IR infrared remote control sensor
5	PWMB Right Motor Speed pin (ENB)
6	PWMA Left Motor Speed pin (ENA)
7	RGB serial command for RGB LEDs
8	D/C for OLED display
9	REST for OLED display
10	CS for ADC module
11	DOUT for ADC module
12	ADDR for ADC module
13	CLK for ADC module
A0	AIN2 Motor-L forward (IN2)
A1	AIN1 Motor-L backward (IN1)
A2	BIN1 Motor-R forward (IN3)
A3	BIN2 Motor-R backward (IN4)
A4	SDA for I/O expansion module and OLED display
A5	SCL for I/O expansion module and OLED display

The ADC module is used to measure the batteries voltage and the light values of the five line sensors. The expansion module connected to the I²C port is used to read the status of the joystick keys and the two front distance sensors; these have a digital output, therefore they allow to determine the presence of an object in front of the robot but they do not provide information on the distance.

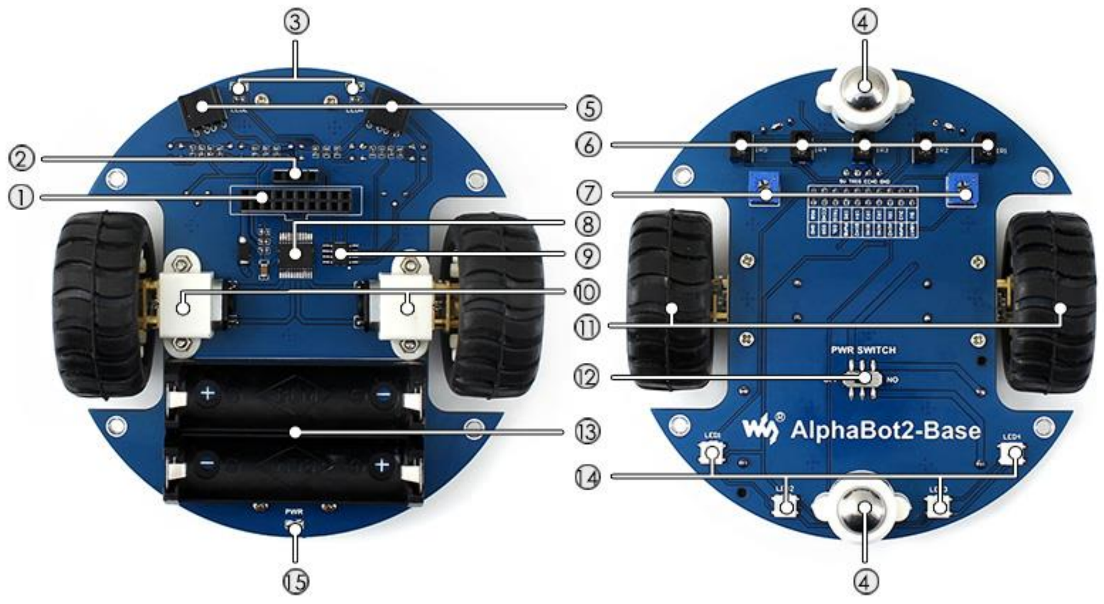


Figure-3.2: AlphaBot2-Base

1. **AlphaBot2 control interface:** for connecting sorts of controller adapter board
2. **Ultrasonic module interface**
3. **Obstacle avoiding indicators**
4. **Omni-direction wheel**
5. **ST188:** reflective infrared photoelectric sensor, for obstacle avoiding
6. **ITR20001/T:** reflective infrared photoelectric sensor, for line tracking
7. **Potentiometer** for adjusting obstacle avoiding range
8. **TB6612FNG** dual H-bridge motor driver
9. **LM393** voltage comparator
10. **N20 micro gear motor** reduction rate 1:30, 6V/600RPM
11. **Rubber wheels'** diameter 42mm, width 19mm
12. **Power switch**
13. **Battery holder:** supports 14500 batteries
14. **WS2812B:** true color RGB LEDs
15. **Power indicator**

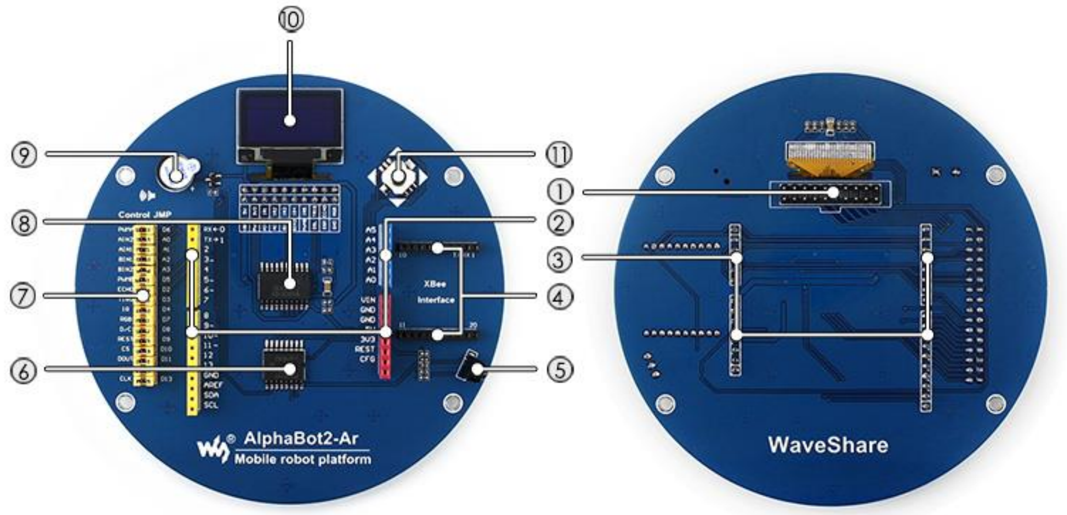


Figure-3.2: AlphaBot2-Upper Part

1. **AlphaBot2 control interface:** for connecting AlphaBot2-Base
2. **Arduino expansion header:** for connecting Arduino shields
3. **Arduino interface:** for connecting Arduino compatible controller
4. **Xbee connector:** for connecting dual-mode Bluetooth module, remotely control the robot via Bluetooth
5. **IR receiver**
6. **PC8574:** I/O expander, SPI interface
7. **Arduino peripheral jumpers**
8. **TLC1543:** 10-bit AD acquisition chip
9. **Buzzer**
10. **0.96inch OLED** SSD1306 driver, 128x64 resolution
11. **Joystick**

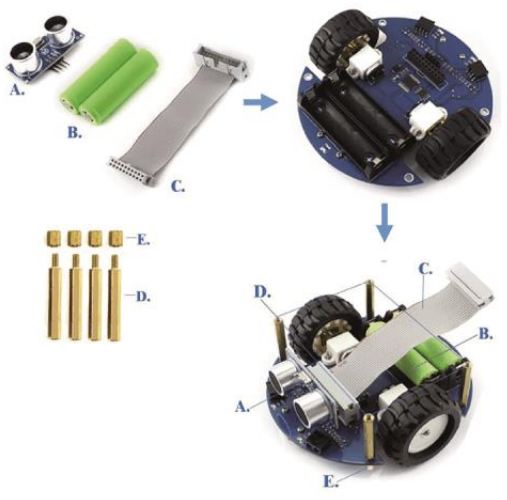
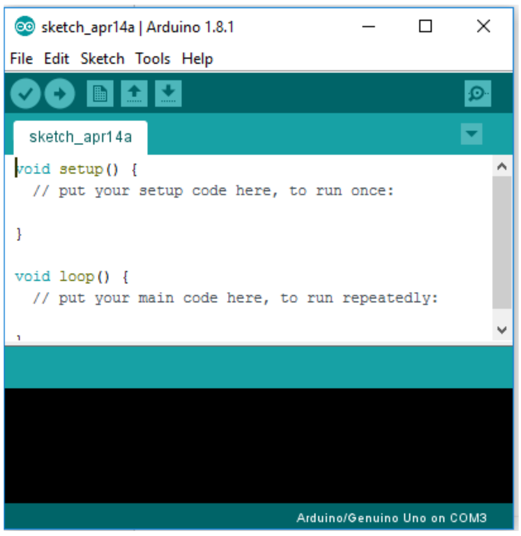


Figure-3.3: AlphaBot2- Assembling

3.2 Arduino

The heart of this project is Arduino. All program of this project is stored in its microprocessor. Arduino is an open source hardware development board. Arduino hardware consists of an open hardware design with an Atmel AVR processor. Arduino programming language is used to program the processor.
There are many types of Arduino board available in the market. But, in this project Arduino UNO has been used. Figure 3.4 shows the Arduino UNO board and Software IDE.


sketch_apr14a | Arduino 1.8.1
File Edit Sketch Tools Help
sketch_apr14a
Arduino/Genuino Uno on COM3

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

```

Figure-3.4: Arduino UNO Board and Software (IDE)

Arduino UNO is based on the ATmega328P microprocessor. It has 14 input/output pins. 6 digital pin can be used as PWM outputs. It has 6 analog input pins. It has a 16 MHz quartz crystal. It contains USB connection port, dc power port. The microcontroller is programmed via Arduino Software (IDE). Figure 3.5 shows the pin mapping of Arduino UNO.

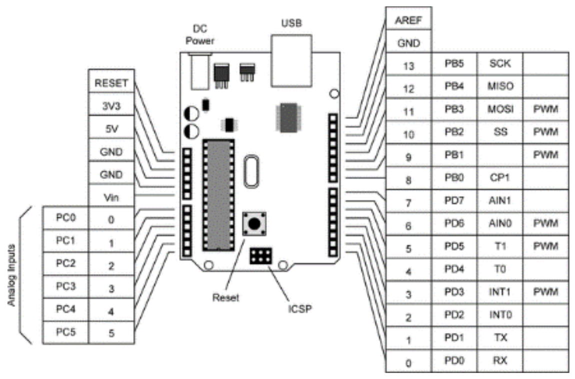


Figure-3.5: Pin Mapping of Arduino UNO

3.2 Tracker Sensor

Tracker sensor has five analog outputs, and the outputted data are affected by the distance and the color of the detected object. The detected object with higher infrared reflectance (in white) will make larger output value, and the one with lower infrared reflectance (in black) will make smaller output value. When the sensor is getting close to a black line, the output value will come to smaller and smaller. So it is easy to get the distance from the black line by checking the analog output (The closer distance between the sensor and the black line, the smaller output value you will get).

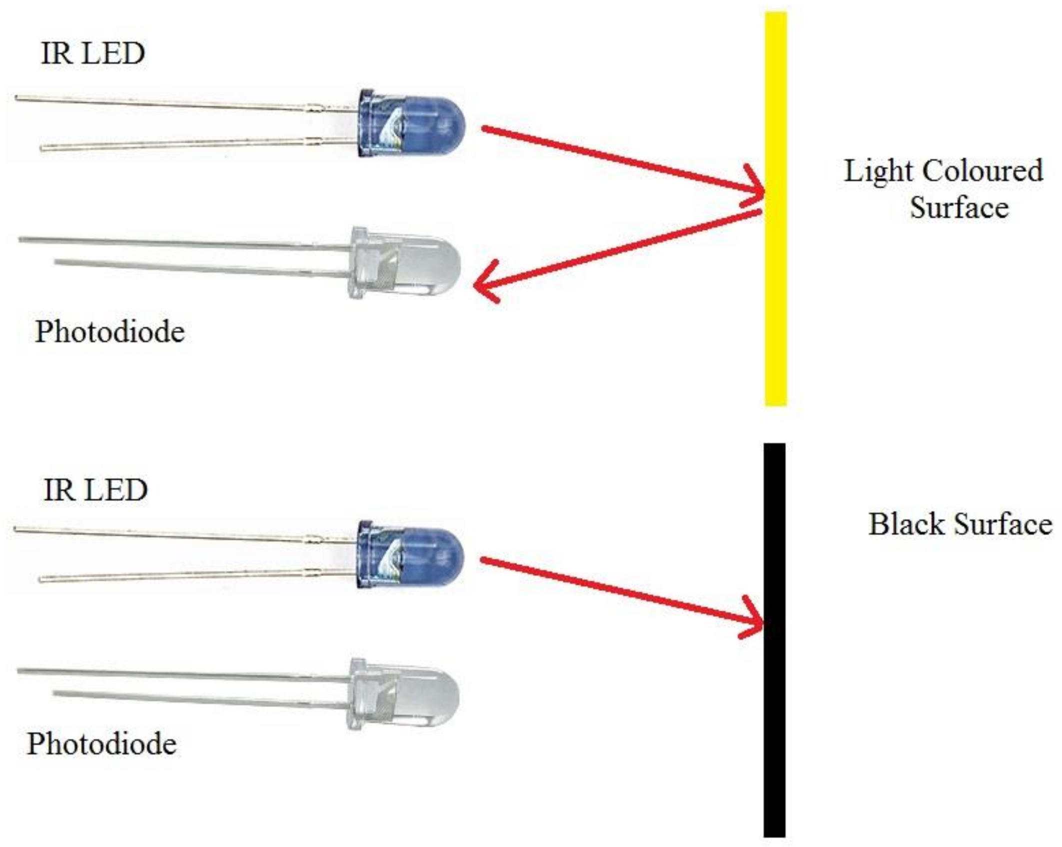


Figure-3.6:Detect on Black and White Surface

3.3 TB6612FNG Dual H-bridge Motor Driver

The TB6612FNG is an easy and affordable way to control motors. The TB6612FNG is capable of driving two motors at up to 1.2A of constant current. Inside the IC, you'll find two standard H-bridges on a chip allowing you to not only control the direction and speed of your motors but also stop and brake. This guide will cover in detail how to use the TB6612FNG breakout board. The library for this guide will also work on the RedBot Mainboard as well since it uses the same motor driver chip.Its specifications are as follows:

- Logic supply voltage (VCC) of 2.7-5.5 VDC
- Motor supply voltage (VM) up tp 15VDC
- Current of 1.2A constant per channel (3.2A peak)
- Drives up to 2 motors

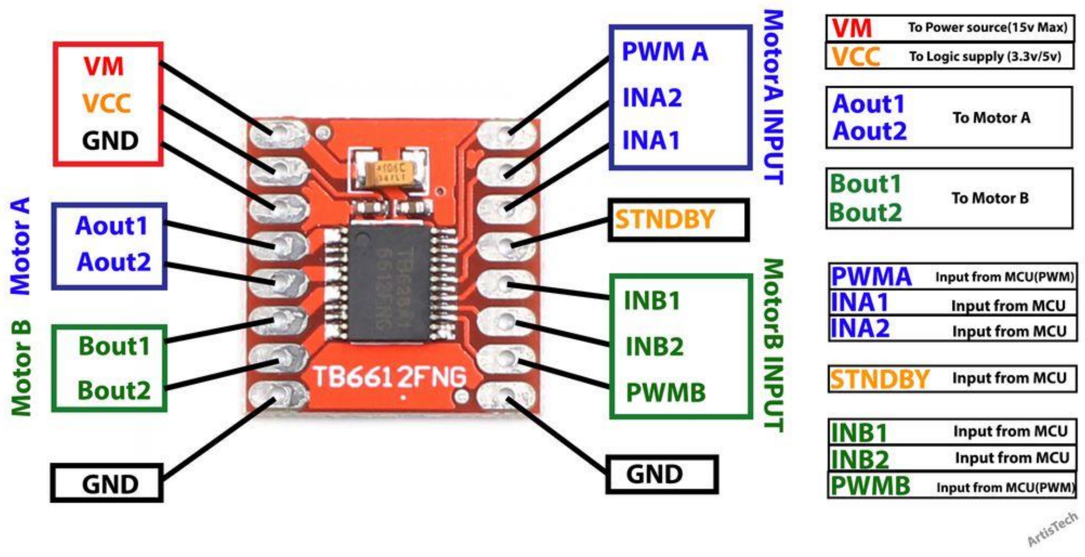


Figure-3.7: **Dual H-bridge Motor Driver**

3.4 N20 micro gear motor

D-Shaped Output Shaft
High Quality Stainless Steel
High Hardness & Wear Resistance
Three-Layer Copper Plate
Precision Mold Stamping
Vibration Resistance & High Tempera-ture Resistance
Durable Structure
Small Size & High Torque & Low Noise & Long Life & Smooth Operation

Figure-3.8: N20 Micro Gear Motor

This gear motor is a miniature high-power, 6 V brushed DC motor with a 150.58:1 metal gearbox. It has a cross section of 10 × 12 mm, and the D-shaped gearbox output shaft is 9 mm long and 3 mm in diameter.

Specifications:

Rated Voltage: DC 12V

Reduction Ratio: 1:30

No-Load Speed: 1000RPM

Rated Torque: 0.3Kg.cm

Rated Current: 200mA

D Shaped Output Shaft Size: 2.5*9.2mm (0.098" x 0.36") (D*L)

Gearbox Size: 9 x 12 x 10mm (0.35 x 0.47 x 0.39inch) (L*W*H)

Motor Size: 15.2 x 12 x 10mm (0.598 x 0.47 x 0.39) (L*W*H)

Net weight: 9g

3.4 OLED

This OLED display module is small, only 0.96" diagonal, it is made of 128x64 individual yellow and blue OLED pixels, each one is turn on or off by the controller chip. It works without backlight, that is, in a dark environment OLED display is higher compared to that of LCD display you will like the miniature for its crispness. The Driver chip of this OLED is SSD1306, which is compatible with I2C communication. So this module can be controlled by I2C. That is, except the VCC and GND, 2 wires would be needed when using 4-wires I2C mode. There is also a simple switch-cap charge pump that turns 5v into a low voltage drive for the OLEDs, making this module the easiest ways to get an OLED into our project.

Figure-3.9: 0.96inch OLED, 128x64 resolution

Features:

OLED self-luminous, no backlight

Size: 0.96”

Driver IC: SSD1306

Voltage: 3.3V-5V DC

Viewing angle: > 160°

High resolution: 128 x 64

Working Temperature: -30°C~70°C

Display: 2 rows of yellow, 6 rows of blue

Module Size: 27mmx 27mm x 4mm

Screen material: glass, need good protection

I2C Interface:

GND: Ground

VCC: 3.3V~5V

SCL: I2C Serial Clock (UNO: A5; MEGA: 21)

SDA: I2C Serial Data (UNO: A4; MEGA: 20)

3.5 Batteries

We are using two 14500 batteries to power AlphaBot2. Each battery relates to other in series connection. As a battery can supply 3.7V, two batteries in series become 7.4V.

Figure-3.10: 14500 batteries

CHAPTER 4

ALGORITMHS

This chapter will give the details about the algorithms. Mainly, Wall following algorithm and Flood fill algorithm are in this project.

4.1 Wall following algorithm

The most common algorithm for maze solving robot is Wall following algorithm. The robot will take its direction by following either left or right wall. This algorithm also called, left hand-right hand rules. Whenever the robot reaches a junction, it will sense for the opening walls and select it direction giving the priority to the selected wall. By taking the walls as guide, this strategy is capable to make the robot reaches the finish point of the maze without actually solving it. But, this algorithm is not efficient method to solve a maze. Cause, the wall follower algorithm will fail to solve some maze construction, such as a maze with a closed loop region.

Right wall following routine	**Left wall following routine**
`if there is no wall at right,`	`if there is no wall at left`
`turn right`	`turn left`
`else`	`else`
`if there is no wall at straight`	`if there is no wall at straight`
`keep straight`	`keep straight`
`else`	`else`
`if there is no wall at left`	`if there is no wall at right`
`turn left`	`turn right`
`else`	`else`
`turn around`	`turn around`

The instructions used in the algorithm for both left and right wall is given in a table:

4.2 Bellman-Ford Algorithm

Like Dijkstra's algorithm, Bellman–Ford proceeds by relaxation, in which approximations to the correct distance are replaced by better ones until they eventually reach the solution. In both algorithms, the approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value and the length of a newly found path. However, Dijkstra's algorithm uses a priority queue to greedily select the closest vertex that has not yet been processed, and performs this relaxation process on all of its outgoing edges.

In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances. This method allows the Bellman–Ford algorithm to be applied to a wider class of inputs than Dijkstra.

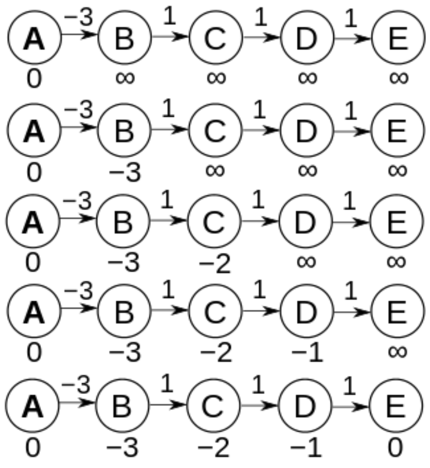


Figure-4.1: Bellman-Ford Algorithm

CHAPTER 5

Methodology

This chapter covers the flow chart of this project, block diagram of this robot, physical implementation, software development, software implementation etc.

5.1 Operation Flowchart

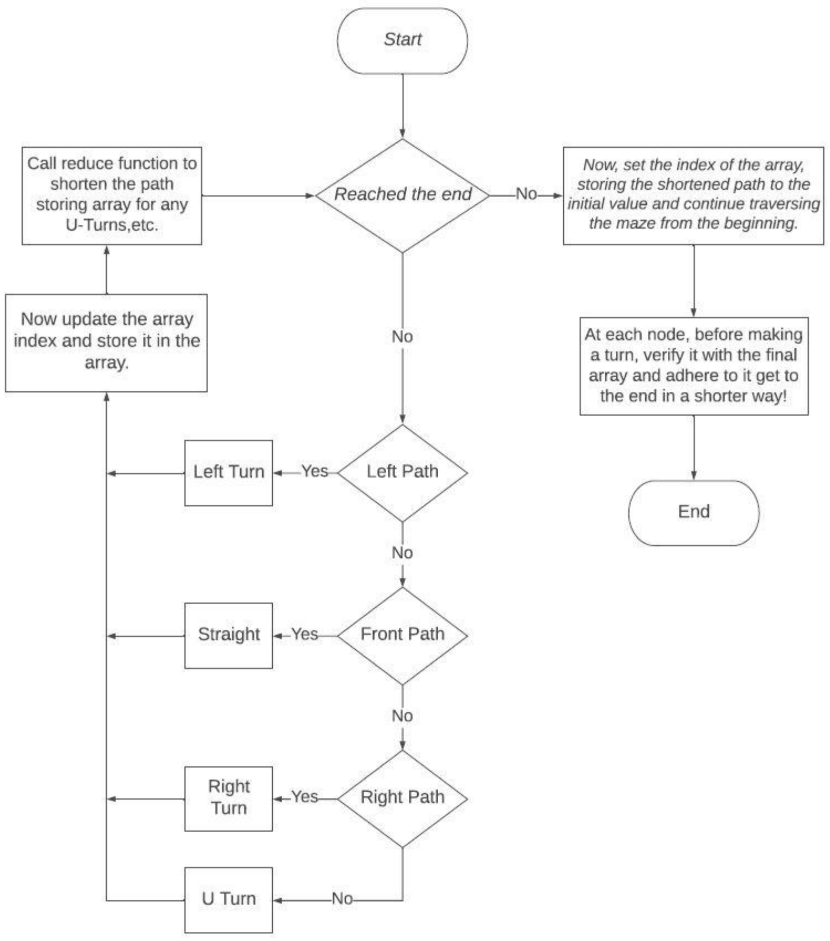

Start
Reached the end
No
Now, set the index of the array, storing the shortened path to the initial value and continue traversing the maze from the beginning.
At each node, before making a turn, verify it with the final array and adhere to it get to the end in a shorter way!
End
Call reduce function to shorten the path storing array for any U-Turns,etc.
Now update the array index and store it in the array.
No
Left Path
Yes
Left Turn
No
Front Path
Yes
Straight
No
Right Path
Yes
Right Turn
No
U Turn

Figure-5.1: Operation Flowchart

5.2 Block Diagram

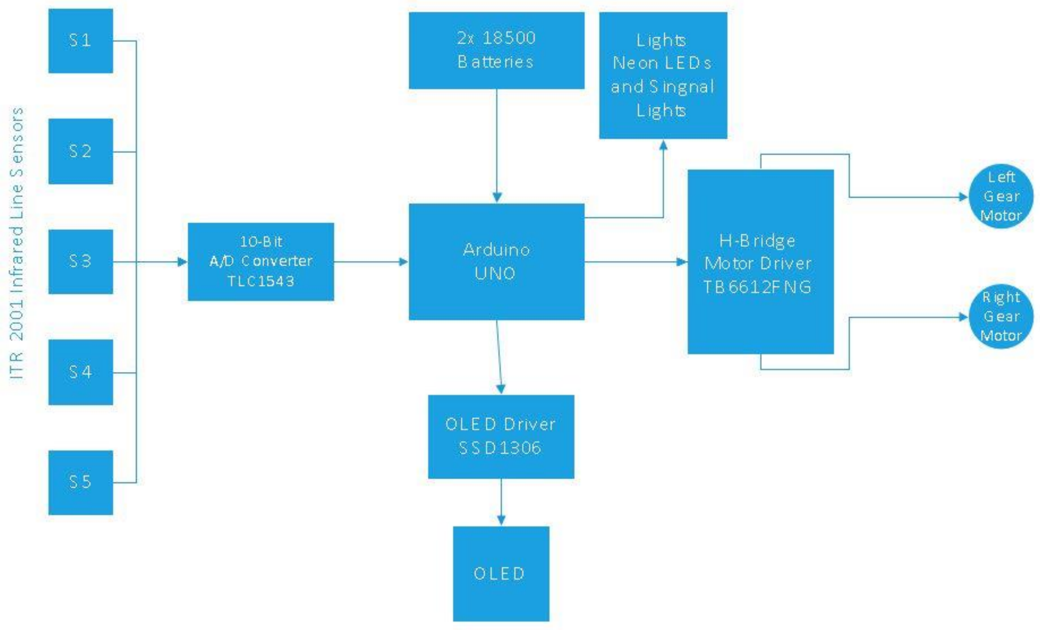

ITR 2001 Infrared Line Sensors
S1
S2
S3
S4
S5
10-Bit A/D Converter TLC1543
2x 18500 Batteries
Arduino UNO
Lights Neon LEDs and Singnal Lights
H-Bridge Motor Driver TB6612FNG
Left Gear Motor
Right Gear Motor
OLED Driver SSD1306
OLED

Figure-5.2: Block Diagram

5.3 Software Development

The maze running software have been developed using two known maze solving algorithms.

Before software preparing for running and finding the shortest way in maze using these two algorithms, first explain the "Lighting and Display process for show for the car of analyzing in the maze. In this we use Header file of -

#include <Adafruit_NeoPixel.h>

#include <Adafruit_GFX.h>

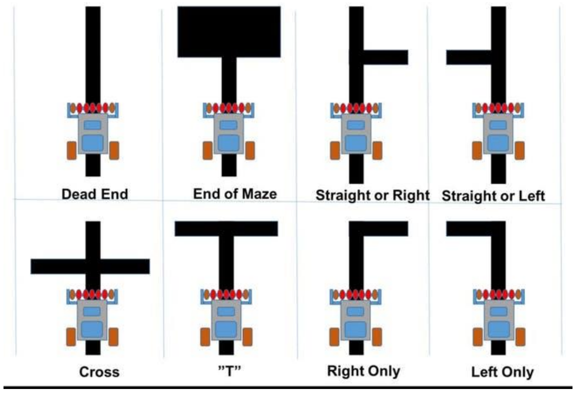
#include <Adafruit_SSD1306.h>

To show lighting with 4 Neo Pixel under the base bot developed this 4 neons to change of customize colours change of millions of color is based on only RGB channel.

For Displaying on OLED board, we use the above under 2 libraries file. On board

First display "TU(Hmawbi)", "Maze Solver", "Press to calibrate" with white color before robot running, If we push the switch, this calibrate done and Display "Calibration Done !!!", "AlhpaBot2" "Go!" and staring to running.

Bot runs first along the routine in the maze as it wish without unknown where the shortest way is and whenever it reaches the dead end it turn to left and more 90 degree(U turn) and delete the routine that way to dead end from the path way. After this bot is at end, we again run the bot from where we started first and in this time the bot is running the shortest way to the goal. For this we use #include "TRSensors.h" library. To solve the maze, we have Left hand rule.



Looking the picture, we can realize that the possible actions at intersections are:

1. At a "Cross":

- Go to Left, or
- Go to Right, or
- Go Straight

2. At a "T":

- Go to Left, or
- Go to Right

3. At a "Right Only":

- Go to Right

4. At a "Left Only":

- Go to Left

5. At "Straight or Left":

- Go to Left, or
- Go Straight

6. At "Straight or Right":
    - Go to Right, or
    - Go Straight
7. At a "Dead End":
    - Go back ("U turn")
8. At "End of Maze":
    - Stop

But, applying the "Left-Hand Rule", the actions will be reduced to one option each:

- At a "Cross": Go to Left
- At a "T": Go to Left
- At a "Right Only": Go to Right
- At a "Left Only": Go to Left
- At a "Straight or Left": Go to Left
- At a "Straight or Right": Go Straight
- At a "Dead End": Go back ("U turn")
- At the "End of Maze": Stop

While running the bot along the path it need to normalize of go. For this we use Normalization process and weighted average and PID control.

5.3.1 Normalization process

Different sensors may output different results for the same color and distance. Furthermore, environment can affect the range of analog output. For example, if we apply 10AD for sampling, we may get the output range from 0 to 1023 theoretically.

However, what we get actually will be the Min output value higher than 0 and the Max output value lower than 1023. Normalization process is important and necessary for reducing the affecting factors from different sensors and different environments. Normalization process is a kind of linear transformation by transforming the range of Min~Max to the range of 0~1 with the following formulas.

$$y = (x - Min) / (Max - Min)$$

In which, x is the original output value from sensor, y is the transformed value, and Max and Min are the maximum output value and the minimum output value, respectively.

```
y = (x - Min) * 1000 / (Max - Min)
```

After transformed, the output value will be in the range of 0~1000, in which 1000 means the sensor is far away from the black line, and 0 means the sensor is above the black line.

The program will sample the values from the sensors for many times to get the proper value of Min and Max. In order to get the precise Min and Max, the car should be always running in course of sampling.

5.3.2 Weighted average

Using normalization process to deal with five sets of output data, we will get five sets of data about the distances between the sensors and the black line. Then, we should use weighted average to transform these data into a value to determine center line of the route with the following formula:

```
y = (0 * value0 + 1000 * value1 + 2000 * value2 + 3000 * value3 +4000 * value4) /
(value0 + value1 + value2 + value3 + value4)
```

In which, 0, 1000, 2000, 3000, 4000 are the weights for the five detectors, respectively, from left to right. And value0~value4 are the data with normalization process.

Now, we can get the data in the range of 0~4000, which can tell you the position of the black line. For example, 2000 means the black line is in the middle of the module, 0 means the black line is on the leftmost side of the module, and 4000 means the black line is on the rightmost side of the module.

For more precise detection, we have some requirements on the height of the module and the width of the black line. The width of the black line should be equal to or less than the distance of two sensors (16mm). The proper height of the module is that when the black line is in the middle of two sensors, both sensors can detect the black line.

5.3.3 PID control

From Part 2, we can get the position of the black line. You should make sure the black line is always under the car, so that the car can run along the black line. So, the output value after weight average process should be kept at 2000. Here, we employ positional PID control to make the car run smoothly. About the PID algorithm, you can easy get a lot of information via Internet. In here, we only have a brief description on it.

PID control can feedback and regulate the error with three factors, proportional (P), Integral (I), derivative (D). The followings are PID algorithm.

```
proportional = position - 2000;
derivative = proportional - last_proportional;
integral += proportional;

last_proportional = proportional;

power_error = proportional * Kp + integral * Ki + derivative * Kd;
```

in which:

Ideally, the weighted average output is 2000, that is, the black line is kept in the middle. The proportional is the result of current position(Position) minus objective position (2000). It is the position error, of which the positive number means the car is on the right of the black line, and the negative number means the car is on the left of the black line.

Integral is the sum of all the errors. When the absolution value is large, the error accumulation is large too, which means the car go far away from the route.

Derivative is the difference of the current proportional and the last proportional, reflecting the response speed of the car. The large derivative value means the fast response speed.

You can adjust the parameters Kp, Ki and Kd to have the better performance. Firstly, we adjust Kp; set the Ki and Kd to 0, and adjust the value of Kp to make the car run along the black line. Then, adjust Ki and Kd; set the parameters to a small value or 0.

5.3 Maze Construction

A 4×4 physical maze is constructed to verify the robot performance. And simulate the algorithms.

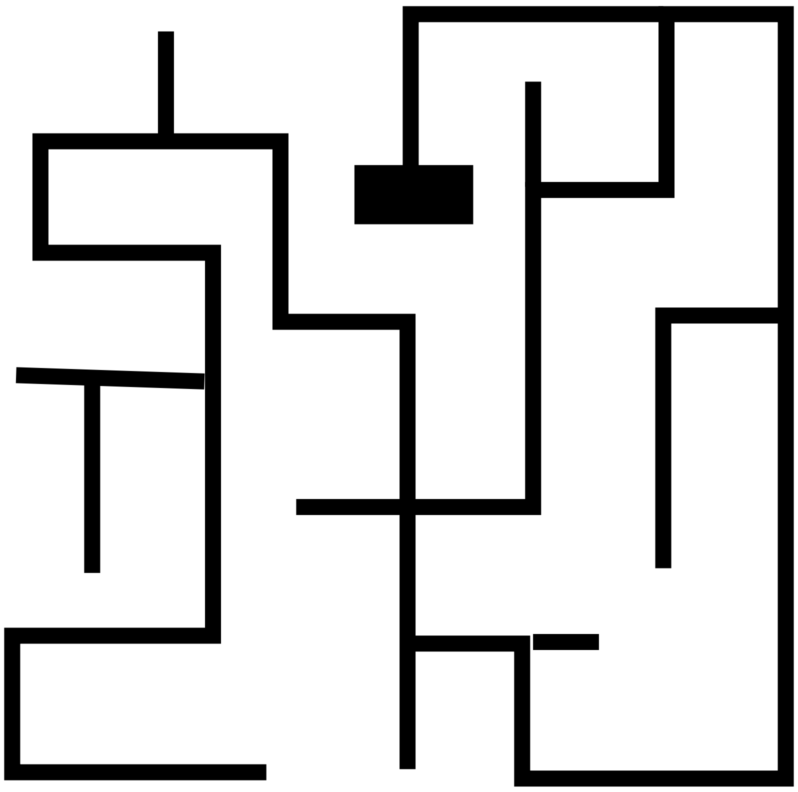


Figure-5.2: Maze Construction

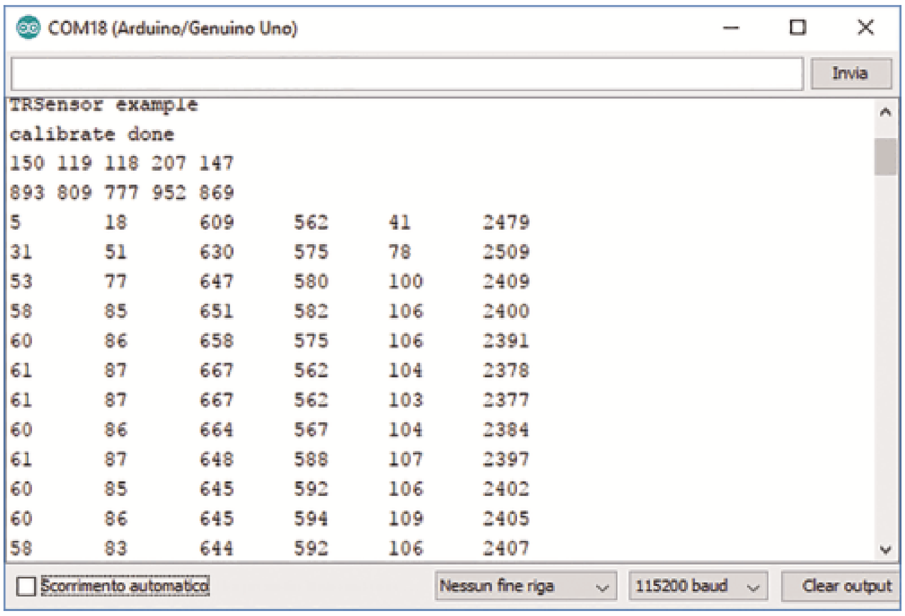
CHAPTER 6

Result and Discussion

This chapter covers the simulated results, compare results with others system, and discussion about this project.

6.1 Tracker Sensors' Response

Figure 6.1 shows the values of reflective infrared photoelectric sensors. They are used to detect the black path. The values are measured by using serial monitor of Arduino IDE.



```
TRSensor example
calibrate done
150 119 118 207 147
893 809 777 952 869
5       18      609     562     41      2479
31      51      630     575     78      2509
53      77      647     580     100     2409
58      85      651     582     106     2400
60      86      658     575     106     2391
61      87      667     562     104     2378
61      87      667     562     103     2377
60      86      664     567     104     2384
61      87      648     588     107     2397
60      85      645     592     106     2402
60      86      645     594     109     2405
58      83      644     592     106     2407
```

Figure-6.1: Values of Reflective Infrared Photoelectric Sensors

6.2 Run on Maze

6.2.1 Wall follower

The robot solves the maze by following the path. In real maze, this robot was tested. In figure 6.2, there is a straight line so, it moves forward. In figure 6.3, both left and front path is located. So, as defined as code, it turns left first and there is an end line,. Then, it makes 180° turn.

After that, both left and right path exit but no front path. So, it turns left again and go straight. In the maze, it works in this manner and solves the maze.

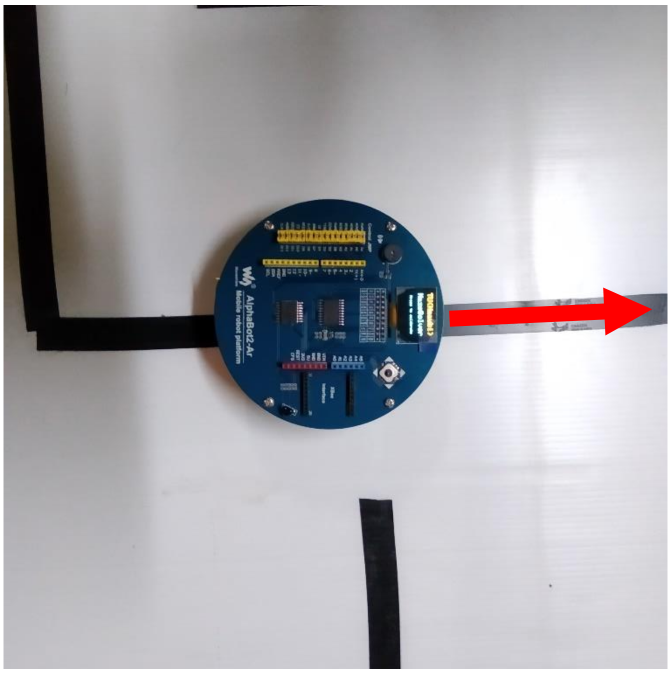


Figure-6.2: Run in maze. (Straight line)

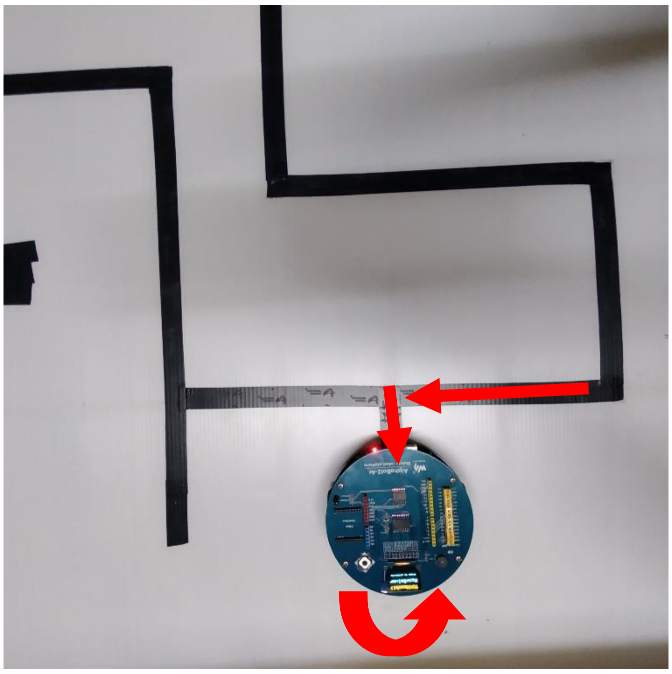


Figure-6.3: Run in maze. (180 degree turn)

6.3 Application of this project

Nowadays robots are widely used in everyday life. This project is based on decision making algorithms. So, it can be used in various intelligent fields such as:

- **Industrial Applications**: These robots can be used as automated equipment carriers in industries replacing traditional conveyer belts.
- **Automobile applications**: These robots can also be used as automatic cars running on roads with embedded magnets.
- **Domestic applications**: These can also be used at homes for domestic purposes like floor cleaning etc.
- **Guidance applications**: These can be used in public places like shopping malls, airports, museums to provide path guidance.

6.4 Cost Analysis

The total cost of this project is 120,000 kyats. It is very cheap price for a robot. Mobile robot requires a chassis for assembling all equipment. The cost of the AlphaBot2 chassis which is used in this project is 115,000 kyats. But, it is worth to buy and test such a perfect mobile robot.

The price of Arduino Uno is 9000 kyats. However, the cost of Arduino can be reduced by using PIC microcontroller. But, the circuit might be more complex and need external cost for run that microprocessor.

The maze playground costs only 8,000 kyats for 4’ x 4’ and the three wire tapes for 200 kyats per each.

Item #	Description	Qty	Unit Price	Price
1	Alphabot2 Kit	1	$ 93,000.00	$ 93,000.00
2	Arduino Uno Board	1	$ 9,000.00	$ 9,000.00
3	Male to Male	1	$ 1,800.00	$ 1,800.00
4	14500 Battery	4	$ 1,800.00	$ 7,200.00
5	Black wire tape	5	$ 200.00	$ 1,000.00
6	Playground (PP Board)	1	$ 8,000.00	$ 8,000.00
				$ 120,000.00

Table-2: The used parts list and the total cost of the project.

CHAPTER 7

Conclusion

This chapter covers the conclusion and further development of this project.

7.1 Conclusion

As a conclusion, the two mazes solving algorithm have successfully been implemented in the robot and the objectives of the project have been achieved. The first algorithm was wall following algorithm. The basic method shows a good result for solving the maze. But, due to lack of self-intelligence, it could not solve to close loop maze. So, an efficient method has been used to find the shortest path that is Bellman Ford algorithm method. After applying all methods, the robot was trained in a real maze. Several tests has been run to ensure the best performance of the robot.

This project helps to improve various important information about robotics, knowledge about many decision making algorithms. It’s also helped to learn about many electronics components such as motor driver, sensors, etc. This gained knowledge will have a significant impact on future work.

7.2 Further Development

This robot is little bit slow, so new method should be developed. The size of the robot can be made smaller. In this project, the line tracking sensors are used for mapping the maze. Other efficient navigating sensors such as Lidar, Camera can be used. At last, it needs development in its wheels and body to make it comfortable in real world.

Reference

[1] **Musfiqur Rahman**, Autonomous Maze Solving Robot, Availabe: https://www.researchgate.net/publication/316664613

[2] **Boris Landon**, Alphabot2: the OpenSource Robot, Availabe: https://www.open-electronics.org/alphabot2-the-opensource-robot/

[3] **Waveshare Co.Ldt**, Alphabot2-Ar, Availabe: https://www.waveshare.com/wiki/AlphaBot2-Ar

[4] **Waveshare Co.Ldt**, Tracker Sensor, Availabe: https://www.waveshare.com/wiki/TrackerSensor

[5] **Arduino CC,** Pulse Width Modulation, Availabe: https://www.arduino.cc/en/Tutorial/PWM

[6] **Pololu Robotics and Electronics**, Building Line Maze, Availabe: https://www.pololu.com

[7] **Wikipedia** , Bellman Ford Algorithm, Wall Following Algorithm, Availabe: https://en.wikipedia.org/wiki/Bellman–Ford_algorithm

[8] **Wikipedia,** Maze Solving Algorithm, Availabe: https://en.wikipedia.org/wiki/Maze_solving_algorithm

Arduino Code

```
#include <Adafruit_NeoPixel.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include "TRSensors.h"
#include <Wire.h>

#define PWMA   6          //Left
Motor Speed pin (ENA)
#define AIN2   A0         //Motor-L
forward (IN2).
#define AIN1   A1         //Motor-L
backward (IN1)
#define PWMB   5          //Right
Motor Speed pin (ENB)
#define BIN1   A2         //Motor-R
forward (IN3)
#define BIN2   A3         //Motor-R
backward (IN4)
#define PIN 7
#define NUM_SENSORS 5
#define OLED_RESET 9
#define OLED_SA0   8
#define Addr  0x20

Adafruit_SSD1306
display(OLED_RESET,
OLED_SA0);

TRSensors trs =   TRSensors();
unsigned int
sensorValues[NUM_SENSORS];
unsigned int position;
uint16_t i, j;
byte value;
unsigned long lasttime = 0;
Adafruit_NeoPixel RGB =
Adafruit_NeoPixel(4, PIN,
NEO_GRB + NEO_KHZ800);

void PCF8574Write(byte data);
byte PCF8574Read();
uint32_t Wheel(byte WheelPos);

void setup() {
  delay(1000);
  Serial.begin(115200);
  Serial.println("TRSensor
example");
  Wire.begin();
  pinMode(PWMA,OUTPUT);
  pinMode(AIN2,OUTPUT);
  pinMode(AIN1,OUTPUT);
  pinMode(PWMB,OUTPUT);
  pinMode(AIN1,OUTPUT);
  pinMode(AIN2,OUTPUT);
  SetSpeeds(0,0);
  // by default, we'll generate the high
voltage from the 3.3v line internally!
(neat!)

display.begin(SSD1306_SWITCHC
APVCC, 0x3C);  // initialize with the
I2C addr 0x3D (for the 128x64)
  // init done

  // Show image buffer on the display
hardware.
  // Since the buffer is intialized with
an Adafruit splashscreen
  // internally, this will display the
spxdd d lashscreen.
  display.clearDisplay();
  display.setTextSize(2);
  display.setTextColor(WHITE);
  display.setCursor(5,0);
  display.println("TU(Hmawbi)");
  display.setCursor(5,25);
  display.println("MazeSolver");
  display.setTextSize(1);
  display.setCursor(10,55);
  display.println("Press to
calibrate");
  display.display();

  while(value != 0xEF)  //wait button
pressed
  {
    PCF8574Write(0x1F |
PCF8574Read());
    value = PCF8574Read() | 0xE0;
  }

  RGB.begin();
  RGB.setPixelColor(0,0x00FF00 );
  RGB.setPixelColor(1,0x00FF00 );
  RGB.setPixelColor(2,0x00FF00 );
  RGB.setPixelColor(3,0x00FF00);
  RGB.show();
  delay(500);
  for (int i = 0; i < 100; i++)  // make
the calibration take about 10
seconds
  {
    if(i<25 || i >= 75)
    {
      SetSpeeds(80,-80);
    }
    else
    {
       SetSpeeds(-80,80);
    }
    trs.calibrate();       // reads all
sensors 100 times
  }
  SetSpeeds(0,0);
  RGB.setPixelColor(0,0x0000FF );
  RGB.setPixelColor(1,0x0000FF );
  RGB.setPixelColor(2,0x0000FF );
  RGB.setPixelColor(3,0x0000FF);
  RGB.show(); // Initialize all pixels
to 'off'

  value = 0;
  while(value != 0xEF)  //wait button
pressed
  {
    PCF8574Write(0x1F |
PCF8574Read());
    value = PCF8574Read() | 0xE0;
    position =
trs.readLine(sensorValues)/200;
    display.clearDisplay();
    display.setCursor(0,25);
    display.println("Calibration Done
!!!");
    display.setCursor(0,55);
    for (int i = 0; i < 21; i++)
    {
      display.print('_');
    }
    display.setCursor(position*6,55);
    display.print("**");
    display.display();
  }

  display.clearDisplay();
  display.setTextSize(2);
  display.setTextColor(WHITE);
  display.setCursor(10,0);
  display.println("AlhpaBot2");
  display.setTextSize(3);
  display.setCursor(40,30);
  display.println("Go!");
  display.display();
  delay(500);
}
```

```
// This function, causes the 3pi to
follow a segment of the maze until
// it detects an intersection, a dead
end, or the finish.
void follow_segment()
{
  int last_proportional = 0;
  long integral=0;

  while(1)
  {
    // Normally, we will be following a
line.  The code below is
    // similar to the 3pi-linefollower-
pid example, but the maximum
    // speed is turned down to 60 for
reliability.

    // Get the position of the line.
    unsigned int position =
trs.readLine(sensorValues);

    // The "proportional" term should
be 0 when we are on the line.
    int proportional = ((int)position) -
2000;

    // Compute the derivative (change)
and integral (sum) of the
    // position.
    int derivative = proportional -
last_proportional;
    integral += proportional;
  // Remember the last position.
  last_proportional = proportional;

  // Compute the difference between
the two motor power settings,
  // m1 - m2.  If this is a positive
number the robot will turn
  // to the left.  If it is a negative
number, the robot will
  // turn to the right, and the
magnitude of the number
determines
  // the sharpness of the turn.
  int power_difference =
proportional/20 + integral/10000 +
derivative*10;

  // Compute the actual motor
settings.  We never set either motor
  // to a negative value.
  const int maximum = 97; // the
maximum speed
  if (power_difference > maximum)
    power_difference = maximum;
  if (power_difference < -maximum)
    power_difference = - maximum;

  if (power_difference < 0)
  {
    analogWrite(PWMA,maximum
+ power_difference);
    analogWrite(PWMB,maximum);
  }
  else
  {
    analogWrite(PWMA,maximum);
    analogWrite(PWMB,maximum -
power_difference);
  }

  // We use the inner three sensors
(1, 2, and 3) for
  // determining whether there is a
line straight ahead, and the
  // sensors 0 and 4 for detecting
lines going to the left and
  // right.
 if(millis() - lasttime >100)
 {
  if (sensorValues[1] < 150 &&
sensorValues[2] < 150 &&
sensorValues[3] < 150)
  {
    // There is no line visible ahead,
and we didn't see any
    // intersection.  Must be a dead
end.
    SetSpeeds(80,80);
    return;
  }
  else if (sensorValues[0] > 600 ||
sensorValues[4] > 600)
  {
    // Found an intersection.
  SetSpeeds(100, 100);
    return;
  }
 }
```

```
  }
}

// Code to perform various types of
turns according to the parameter
dir,
// which should be 'L' (left), 'R'
(right), 'S' (straight), or 'B' (back).
// The delays here had to be
calibrated for the 3pi's motors.
void turn(unsigned char dir)
{
 // if(millis() - lasttime >500)
  {
    switch(dir)
    {
    case 'L':
      // Turn left.
      SetSpeeds(-100, 100);
      delay(180);
      break;
    case 'R':
      // Turn right.
      SetSpeeds(100, -100);
      delay(180);
      break;
    case 'B':
      // Turn around.
      SetSpeeds(80, -80);
      delay(350);
      break;
    case 'S':
      // Don't do anything!
      break;
    }
  }
  SetSpeeds(0, 0);
 // value = 0;
//  while(value != 0xEF)  //wait
button pressed
//  {
//    PCF8574Write(0x1F |
PCF8574Read());
//    value = PCF8574Read() | 0xE0;
//  }
  Serial.write(dir);
  Serial.println();
  lasttime = millis();
}

// This function decides which way to
turn during the learning phase of
// maze solving.  It uses the variables
found_left, found_straight, and
// found_right, which indicate
whether there is an exit in each of
the
// three directions, applying the "left
hand on the wall" strategy.
unsigned char select_turn(unsigned
char found_left, unsigned char
found_straight, unsigned char
found_right)
{
  // Make a decision about how to
turn.  The following code
  // implements a left-hand-on-the-
wall strategy, where we always
```

```
  // turn as far to the left as possible.
  if (found_left)
    return 'L';
  else if (found_straight)
    return 'S';
  else if (found_right)
    return 'R';
  else
    return 'B';
}

// The path variable will store the
path that the robot has taken.  It
// is stored as an array of characters,
each of which represents the
// turn that should be made at one
intersection in the sequence:
//  'L' for left
//  'R' for right
//  'S' for straight (going straight
through an intersection)
//  'B' for back (U-turn)
//
// Whenever the robot makes a U-
turn, the path can be simplified by
// removing the dead end.  The
follow_next_turn() function checks
for
// this case every time it makes a
turn, and it simplifies the path
// appropriately.
char path[100] = "";
unsigned char path_length = 0; // the
length of the path

// Path simplification.  The strategy
is that whenever we encounter a
// sequence xBx, we can simplify it
by cutting out the dead end.  For
// example, LBL -> S, because a
single S bypasses the dead end
// represented by LBL.
void simplify_path()
{
  // only simplify the path if the
second-to-last turn was a 'B'
  if (path_length < 3 ||
path[path_length-2] != 'B')
    return;

  int total_angle = 0;
  int i;
  for (i = 1; i <= 3; i++)
  {
    switch (path[path_length - i])
    {
    case 'R':
      total_angle += 90;
      break;
    case 'L':
      total_angle += 270;
      break;
    case 'B':
      total_angle += 180;
      break;
    }
  }
  // Get the angle as a number
between 0 and 360 degrees.
  total_angle = total_angle % 360;

  // Replace all of those turns with a
single one.
  switch (total_angle)
  {
  case 0:
    path[path_length - 3] = 'S';
    break;
  case 90:
    path[path_length - 3] = 'R';
    break;
  case 180:
    path[path_length - 3] = 'B';
    break;
  case 270:
    path[path_length - 3] = 'L';
    break;
  }

  // The path is now two steps
shorter.
  path_length -= 2;
}

void loop() {
  while (1)
  {
    follow_segment();

    // Drive straight a bit.  This helps
us in case we entered the
    // intersection at an angle.
    // Note that we are slowing down -
this prevents the robot
    // from tipping forward too much.
    SetSpeeds(30, 30);
    delay(40);

    // These variables record whether
the robot has seen a line to the
    // left, straight ahead, and right,
whil examining the current
    // intersection.
    unsigned char found_left = 0;
    unsigned char found_straight = 0;
    unsigned char found_right = 0;

    // Now read the sensors and check
the intersection type.
   trs.readLine(sensorValues);

    // Check for left and right exits.
    if (sensorValues[0] > 600)
      found_left = 1;
    if (sensorValues[4] > 600)
      found_right = 1;

    // Drive straight a bit more - this is
enough to line up our
    // wheels with the intersection.
    SetSpeeds(30, 30);
    delay(100);

    // Check for a straight exit.
    trs.readLine(sensorValues);
   if (sensorValues[1] > 600 ||
sensorValues[2] > 600 ||
sensorValues[3] > 600)
     found_straight = 1;

   // Check for the ending spot.
   // If all three middle sensors are
on dark black, we have
   // solved the maze.
   if (sensorValues[1] > 600 &&
sensorValues[2] > 600 &&
sensorValues[3] > 600)
   {
     SetSpeeds(0, 0);
     break;
   }

   // Intersection identification is
complete.
   // If the maze has been solved, we
can follow the existing
   // path.  Otherwise, we need to
learn the solution.
   unsigned char dir =
select_turn(found_left,
found_straight, found_right);

   // Make the turn indicated by the
path.
   turn(dir);

   // Store the intersection in the path
variable.
   path[path_length] = dir;
   path_length++;

   // You should check to make sure
that the path_length does not
   // exceed the bounds of the array.
We'll ignore that in this
   // example.

   // Simplify the learned path.
   simplify_path();

   // Display the path on the LCD.
  // display_path();
 }

 // Solved the maze!

 // Now enter an infinite loop - we
can re-run the maze as many
 // times as we want to.
 while (1)
 {
   Serial.println("End !!!");
//    display.display();
   delay(500);

   value = 0;
   while(value != 0xEF)  //wait
button pressed
   {
     PCF8574Write(0x1F |
PCF8574Read());
     value = PCF8574Read() | 0xE0;
   }
```

```
    delay(1000);

    // Re-run the maze.  It's not
necessary to identify the
    // intersections, so this loop is
really simple.
    int i;
    for (i = 0; i < path_length; i++)
    {
      follow_segment();

      // Drive straight while slowing
down, as before.
      SetSpeeds(30, 30);
      delay(40);
      SetSpeeds(30, 30);
      delay(150);

      // Make a turn according to the
instruction stored in
      // path[i].
      turn(path[i]);
    }

    // Follow the last segment up to
the finish.
    follow_segment();

    // Now we should be at the finish!
Restart the loop.
  }
}
```

```
void SetSpeeds(int Aspeed,int
Bspeed)
{
  if(Aspeed < 0)
  {
    digitalWrite(AIN1,HIGH);
    digitalWrite(AIN2,LOW);
    analogWrite(PWMA,-Aspeed);
  }
  else
  {
    digitalWrite(AIN1,LOW);
    digitalWrite(AIN2,HIGH);
    analogWrite(PWMA,Aspeed);
  }

  if(Bspeed < 0)
  {
    digitalWrite(BIN1,HIGH);
    digitalWrite(BIN2,LOW);
    analogWrite(PWMB,-Bspeed);
  }
  else
  {
    digitalWrite(BIN1,LOW);
    digitalWrite(BIN2,HIGH);
    analogWrite(PWMB,Bspeed);
  }
}

void PCF8574Write(byte data)
{
  Wire.beginTransmission(Addr);
  Wire.write(data);
  Wire.endTransmission();
}

byte PCF8574Read()
{
  int data = -1;
  Wire.requestFrom(Addr, 1);
  if(Wire.available()) {
    data = Wire.read();
  }
  return data;
}
```