**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| **Figures** | **Details** | **Page.No.** |
| Fig 1.1 | Forklift AGV | 7 |
| Fig 1.2 | Towling AGV | 8 |
| Fig 1.3 | Unit load handler AGV | 9 |
| Fig 1.4 | Autonomous mobile robot | 10 |
| Fig 1.5 | Types of AGV’s | 11 |
| Fig 3.1 | Basic Layout Of AGV | 17 |
| Fig 3.2 | Micro Metal Geared Motor | 18 |
| Fig 3.3 | IR sensor array | 19 |
| Fig 3.4 | Magnetic Encoders | 19 |
| Fig 3.5 | ATMEGA 4809 | 20 |
| Fig 3.6 | Front View of Motor Driver | 21 |
| Fig 3.7 | Back View of The Motor Driver | 21 |
| Fig 3.8 | Xbee Series 2 | 22 |
| Fig 3.10 | LIPO Battery | 23 |
| Fig 3.11 | model of the AGV | 23 |
| Fig 3.12 | prototype of the AGV | 24 |
| Fig 3.13 | Top view of the AGV | 24 |
| Fig 4.1 | Block diagram of control system implemented | 26 |
| Fig 4.2 | Flow chart of the mapping algorithm | 27 |
| Fig 4.3 | Representation of adjacent matrix list | 29 |
| Fig 4.4 | Flow chart of dijkstra’s algorithm | 30 |
| Fig 6.1 | Floor plan 1 | 68 |
| Fig 6.2 | Floor plan 2 | 69 |
| Fig 6.3 | Floor plan 3 | 71 |
| Fig 6.4 | Floor plan 4 | 72 |
| Fig 6.5 | Floor plan 5 | 74 |

**LIST OF TABLES**

|  |  |  |
| --- | --- | --- |
| **Tables** | **Details** | **Page.No.** |
| Table 4.1 | Detection of nodes | 28 |
| Table 4.2 | Updating coordinates of each node | 28 |
| Table 4.3 | parameters for configuration of Xbee modules | 31 |
| Table 6.1 | Adjacent matrix list of case1 | 68 |
| Table 6.2 | comparision of algorithms for case1 | 69 |
| Table 6.3 | Path planning for case1 | 69 |
| Table 6.4 | Adjacent matrix list of case2 | 70 |
| Table 6.5 | comparision of algorithms for case2 | 70 |
| Table 6.6 | Path planning for case2 | 70 |
| Table 6.7 | Adjacent matrix list of case3 | 71 |
| Table 6.8 | comparision of algorithms for case3 | 72 |
| Table 6.9 | Path planning for case3 | 72 |
| Table 6.10 | Adjacent matrix list of case4 | 73 |
| Table 6.11 | comparision of algorithms for case4 | 73 |
| Table 6.12 | Path planning for case4 | 73 |
| Table 6.13 | Adjacent matrix list of case5 | 74 |
| Table 6.14 | comparision of algorithms for case5 | 75 |
| Table 6.15 | Path planning for case5 | 75 |
| Table 7.1 | Results of mapping | 76 |

**ABSTRACT**

Automated Guided Vehicle (AGV) is a mobile robot which moves along a prescribed path. It is one of key equipments in modern industrial automation logistics systems and flexible production systems like Computer integrated manufacturing system. It is widely used in food ,Manufacturing and Pharmaceutical industries . In this project we are going to design a two wheeled differential drive based mobile robot (AGV) which tracks its trajectory following a guide tape which also maps the entire shop floor autonomously and plans the shortest path between two prescribed nodes along with traffic control like collision avoidance. To track the tape PID control system is implemented and UART based wireless communication system of range 300ft is also used for interfacing the robot with computer. An optimised Flood Fill algorithm is used to map the shop floor and stored in the form of adjacent matrix type format of graph data structure and on given commands the robot plans the shortest path using dijkstra algorithm.

**Key words**: Automated Guided vehicle, Mapping, Path Planning, Differential drive system, PID control system, Flood Fill algorithm, dijkstra algorithm

**CHAPTER 1: INTRODUCTION**

**1.INTRODUCTION:**

An automated guided vehicle (AGV) is a portable robot that navigates by following indicated long lines or cables on the floor, or by using radio waves, vision cameras, magnets, or lasers. They're most commonly employed in industrial settings to move heavy materials across a vast industrial structure, such as a factory or warehouse**.**

The AGV may two objects behind them in trailers that they can connect to themselves. Raw materials or completed goods can be transported in the trailers. The AGV can also be used to store items on a bed. The items can be placed on a conveyor with motorised rollers and then pushed off by reversing the rollers. AGVs are employed in nearly every industry, including pulp, paper, metals, newspaper, and general.

A laser guided vehicle is another name for an AGV (LGV). The technology is also known as Fahrerloses Transportsystem (FTS) in Germany and förarlösa truckar in Sweden. Automated Guided Carts (AGCs) are lower-cost counterparts of AGVs that are usually guided by magnetic tape. AGCs come in a variety of models and can be used to move products through an assembly line, transfer goods through a warehouse, and more.

Barrett Electronics of Northbrook, Illinois, introduced the first AGV in the 1950s, and it was basically a tow truck that followed a wire in the floor instead of a rail. This technique gave birth to a new form of AGV that instead of being towed by a chain, follows invisible UV marks on the floor. The Willis Tower was the first to use such a technology (formerly Sears Tower).

The technology has advanced throughout time, and today's automated vehicles, such as LGVs, are mostly guided by lasers (Laser Guided Vehicle). LGVs are configured to communicate with other robots in an automated procedure to guarantee that merchandise moves smoothly through the warehouse, whether it is being kept for future use or transferred immediately to shipping regions. The AGV now has a significant role to play.

A versatile automated guided vehicle (AGV) solution is laser guided vehicles (LGVs).... The reflected signals are received by the laser sensors, which are then used to triangulate the vehicle's position. The navigation algorithms at JBT recalculate the vehicle position 40 times per second, ensuring that path corrections are kept to a minimum.

* 1. **TYPES OF AUTOMATED GUIDED VEHICLES:**

Automated guided vehicles come in a variety of shapes and sizes. Many AGVs are meant to run without direct human interaction or guidance, comparable to other human-operated vehicles.

**1.1.1 Automated Guided Carts:**

The most basic sort of AGV, an automatic guided cart (AGC), has the fewest features. Navigation systems can range from rudimentary magnetic tape systems to complicated sensor-based navigation systems that use artificial intelligence to traverse their surroundings. They can convey a wide range of commodities, from small pieces to fully loaded pallets, and are frequently used for sorting, storage, and cross-docking.

An automated hospital cart transporter, for example, is a device that transports small loads such as meals and empty food trays, clean or soiled linens, biohazard waste, and sterile supplies effectively across a hospital. Automated hospital cart transporters can assist cut labour expenses by eliminating the need for a staff member to manually push the cart from one location to another.

### Forklift AGVs

### Towing AGVs

### Unit Load Handlers

### Heavy Burden Carriers

### Autonomous Mobile Robots

### 1.1.2 Forklift AGV:

### An ALT (Automatic Guided Forklift) is a computer-controlled self-driving forklift. So a forklift that moves around and transports things without the need for human interaction is simply referred to as a driverless forklift. The AGV Forklift is part of a larger AGV System that includes robots, software, and accessories. This type of AGV is modelled after traditional human-operated forklifts. The AGV forklift is designed to transfer the cargo both horizontally and vertically. They can be set up to stack loads in racks as well. Automated forklift trucks are becoming increasingly important in industrial and warehouse environments where processes are highly standardised, repetitive, and simple to complete without the need for human participation. In warehouses, forklift robots are commonly utilised for high rack management. Learn about the various material handling procedures that an automated guided forklift can handle in this article.

### 

### Fig1.1Forklift AGV

### 1.1.3 Towing AGV:

### Towing vehicles, also known as tugger automatic guided vehicles, are vehicles that tow one or more non-powered, load-carrying vehicles in a train-like pattern behind them. Powered towing vehicles, sometimes known as driverless trains, run on wheels. Tugger guided automatic vehicles are frequently utilised to deliver huge cargo over long distances. They may have a number of drop-off and pick-up points along a predetermined route. Towing AGV, also known as Tow Vehicle or Lurking Towing Agv, is a type of AGV that can pull many vehicles at once to achieve effective handling. Furthermore, the Tow Vehicles AGV body can be used to transport pallets, material racks, material boxes, and other items. Towing AGVs increase transport efficiency, as do traction AGVs with capacities of 500 kg/1 ton/2 ton.

### 

### Fig1.2 Towling AGV

### 1.1.4 Unit Load Handlers:

### These robots, in simple terms, are portable and autonomous cargo delivery devices that can navigate around a warehouse or facility using various agv navigation technologies. These intriguing robots are primarily used for transporting items and large things around warehouses and storage facilities in the industrial sector. AGVs with Unit Loads, also known as Unit Load Decks, have been around since the beginning of AGVs. Some AGVs pull or carry goods, while others tug them. Unit Loads are designed to transport one or more unit loads simultaneously to and from conveyors, stands, end-of-line equipment (palletizers, wrappers, robots), and automated storage and retrieval systems (AS/RSs). Consider a mobile platform.

### 

### Fig1.3 Unit load handler AGV

### 1.1.5 Heavy Burden Carrier AGV:

### Heavy Burden Carriers AGV, also known as Heavy Burden Carrier Vehicles and Heavy Large-Platform Bidirectional Driving Knapsack AGV, are self-loading automatic guided carts that can be equipped with standard, pivot, and omni-directional steering.

### The Advantages of a Heavy Burden Carrier Vehicle are as follows:

### 1. Loads are delivered and shifted as needed.

### 2. Response and transport times have improved.

### 3. Product damage is reduced.

### 4. Scheduling that works.

### 5. Reduced traffic in the aisle

### 

### Fig1.4 Autonomous mobile robot

### 1.1.6 Autonomous Mobile Robot:

### Autonomous mobile robots (AMRs) are typically more technologically advanced than other types of AGVs. Many AGVs use fixed navigation systems like wires or magnetic tape, but many AMRs have intelligent navigation capabilities like sensors and camera systems that allow them to detect and navigate around obstacles. Thanks to more sophisticated technology, AMRs can dynamically.

### 

### Fig1.5 Types of AGV’s

### 1.2 BENEFITS OF AGV:

### Increase efficiency and productivity:

### AGVs boost efficiency and production by operating independently, and they are predictable and reliable for repetitive operations. AGVs remove the need for unneeded walking as well as the physical effort involved in transporting products. They also set the tone for employees, ensuring that they stay on track. AGVs, similar to collaborative mobile robots, coach associates through the process.

### Flexibility:

### Some AGVs allow for easy route changes (in contrast to others that involve rerouting guide wires or other infrastructure to change a vehicle's route). Automated guided vehicles are also a scalable option, with the capacity to scale up or down depending on demand.

### Less space Required:

### AGVs take up less area than other automation solutions, such as conveyor systems. Because AGVs are smaller than typical warehouse equipment like forklifts, floor layouts with narrower lanes and higher space utilisation are possible.

### 1.3 APPLICATIONS FOR AGVS:

### For activities that would normally be handled by forklifts, conveyor systems, or manual carts, automated guided vehicle systems are utilised to move huge amounts of cargo in a repeated manner.

### AGVs are employed in a wide range of situations. They're frequently utilised to transport raw commodities including metal, plastic, rubber, and paper. AGVs can, for example, transfer raw materials from receiving to the warehouse or deliver goods to production lines directly. AGVs transport needed raw materials consistently and reliably without the need for human involvement, ensuring that manufacturing lines are always stocked.

### AGVs are also employed for replenishing and picking in inbound and outbound handling. AGVs could be used to transfer inventory from receiving to storage or from long-term storage to forward picking locations in order to restock stock, for example. Moving goods from long-term storage to forward picking locations ensures that pickers have enough inventory to work with, making the process more efficient.

### AGVs are utilised in work-in-process applications and with finished goods to support production or manufacturing lines, in addition to moving raw materials. The word "work-in-process" refers to "partially completed commodities, which are often turned from raw material to finished product in a short amount of time," such as manufactured goods, according to Investopedia. In a work-in-progress situation.

### CHAPTER 2: LITERARURE SURVEY

### 2.0 LITERATURE SURVEY:

**1.Xinmin Zhou ,Tianwei Chen and Yimei Zhang-2013:**

This work proposes an AGV control system based on fuzzy PID control, which ensures the AGV trolley's robustness and the system's long-term stability. The main control chip in this design is the STM32F207, and it includes gray-scale sensors, ultrasonic, infrared sensors, cameras, wireless modules, and other peripherals, as well as the C/OS-II real-time operating system.

**2.Shadman Sakib, Anik Chowdhury, Shekh Tanvir and Imam Hasan:**

In this study, we compare and contrast a unique universal method for exploring and solving any type of line maze with a simpler approach for simple mazes without loops or with loops that contain no inward branches. We need a way to map the entire maze for the general algorithm, which is required if the maze is complex. The maze mapping system suggested is based on coordinate systems. The shortest path and shortest time path were found after mapping the entire maze as a graph using the conventional ‘Adjacency-list representation' method.Dijkstra's algorithm was used to extract the data.

**3.Vladan vuckovic,Boban Arizanovic:**

This study proposes a method for finding optimum connections based on region filling algorithms. The proposed method is aimed for maze-solving and is optimised using the heuristic. The suggested technique outperforms non-heuristic alternatives in some instances, and it is dependent on the labyrinth structure and position of the starting and ending pixels.

**4. Tao Zheng , Yanqiang Xu ,Da Zheng:**

The vast applicability of Automated Guided Vehicles (hereafter referred to as AGV) in modern manufacturing, logistics, and transportation industries is presented in this paper, and boosting AGV work efficiency has become a significant focus of industry research. The goal of this paper is to lessen the impact of two aspects on AGV efficiency: AGV optimal running route search and path search speed. he A-star algorithm leverages the characteristics of jump point search to increase the node search mode and speed, and adds the angle evaluation cost function to the A-star algorithm's cost function to discover the path with the least inflection point, allowing it to find the ideal path rapidly. Furthermore, the simulation studies presented in this work show that the enhanced A-star algorithm can find which has a faster path search speed than the A-star algorithm.

**5. Roman Khudeev:**

A new flood fill approach is presented in this article. It is a quick method that may be used on both basic and self-crossing contours. It can also be used to fill in the spaces between the contours.

**6. John Faber Archila, Marcelo Becker:**

The advanced improvements of automatic robots are one of the research domains in flexible manufacturing systems (FMS) aimed at carrying out duties in the logistic regions, as shown in this article. This eliminates the need for human intervention and automates repetitive and/or ineffective tasks. The design and mathematical models of an AGV, or automated guided vehicle, are presented in this article as part of the FMS. It uses CAD and CAE tools to present a mechatronics design process for mechanics, electronics, and control.

**7. BoYang Xu, Dongqing Wang:**

This study discusses precise AGV (Automatic Guided Vehicle) navigation. Based on Kalman filtering (KF) and PID (Proportion-integral-derivative) control, we propose an AGV navigation with magnetic nail positioning approach. The magnetic nail is placed in the area where the AGV travels, and the magnetic sensor is used to receive the AGV's current pose information. The AGV's position adjustment amount is used as the PID controller's input and output. Simultaneously, the Kalman filtering (KF) technique is employed to estimate the state with noise interference, improving the AGV's positioning accuracy even more. The simulation results suggest that the proposed Kalman filtering and PID control for magnetic nail positioning AGV navigation are effective.

**8. Shyamprasad Konduri, Edison Orlando Cobos Torres, Prabhakar R. Pagilla:**

Wheel slip has a significant impact on the ability to track a desired motion trajectory, and the problem is exacerbated when differential drive robots are used in applications requiring coordination of multiple robots, as discussed in this paper. This subject is investigated, and a simple slide avoidance control approach is given based on the wheel–ground traction forces. Robots having two driving wheels and one or more ball-type caster wheels are evaluated for differential drive. The traction forces between the wheels and the ground surface are calculated using the rigid wheel-rigid-ground interaction assumption. These traction forces are used to calculate the maximum input wheel torque that can be supplied before the wheel slips. This limiting torque value is used to define a saturation limit for the input torque computed by a trajectory tracking controller in order to avoid wheel slide.

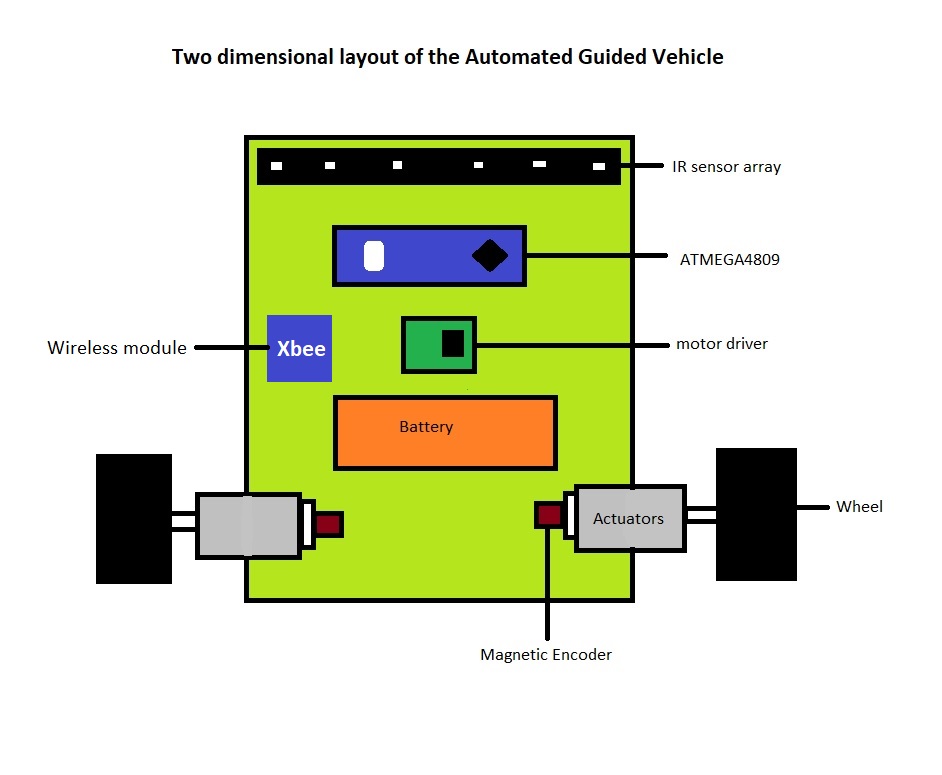
**CHAPTER 3:** **MODELLING AND FABRICATION**

**3.MODEL OF THE AGV**

**3.1** **BASIC LAYOUT OF THE AGV**

The robot consists of the following key components

* Actuators-Micro metal geared motors(300RPM)
* Sensors-Magnetic encoders, Infrared sensor sensor array
* Microcontroller-ATMEGA4809
* Motor driver-Toshiba based TB6612FNG
* Wireless module-Xbee series 2
* Battery-LIPO 7.4V (850mah)

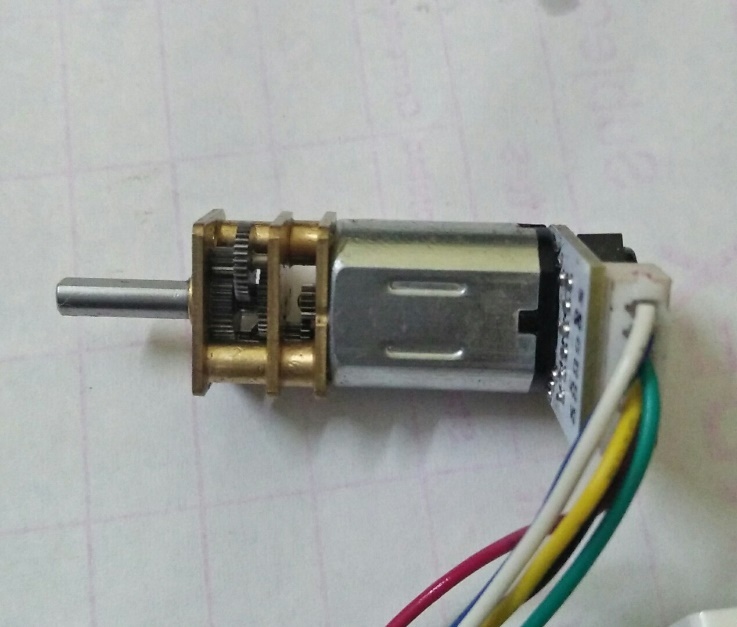


**Fig3.1 Basic Layout Of AGV**

**3.2** **LAYOUT EXPLANATION**

**3.2.1 Actuators**

The actuators used for the robot are micro metal geared DC motor of 300 RPM. These motors comes with a standard wheels and magnetic encoders.



**Fig3.2 Micro Metal Geared Motor**

Specifications:

* Manufacturer-Pololu
* Dimensions-10 × 12 × 26 mm
* Weight-9.5 g
* No load speed-270RPM
* Gear ratio-1:50
* Operating Voltage- 0-6V
* Stall torque-0.44 kg⋅cm

Reasons to select:

* Light in weight and comes with a durable metal gear box
* Has a secondary shaft for encoders
* D shaped main shaft to avoid slippage of wheels

**3.2.2 Sensors**

**Infrared sensor array:**

This sensor is used for sensing the guide path on the ground. It consists of six infrared sensors with a pitch of 18mm.



**Fig3.3 IR sensor array**

Specifications:

* Manufacturer-Grey Robotics
* Dimensions-100 x 20 x 12 mm
* Sensing distance-8mm
* Output-Analog
* Operating voltage-5V
* Pitch (b/w) sensors-18mm
* Minimum thickness of sensing line-20mm

Reasons to select:

* No need of manual calibration
* Typically used for PID controller based line following
* Uses Reflectance Sensors for high precision

**Quadrature Magnetic Encoders:**

This sensors are used to know to position of the motor shaft. It consists a magnetic disc and two hall effect sensors.



**Fig3.4 Magnetic Encoders**

Specifications:

* Manufacturer-Pololu
* Type-Incremental
* Cycles per Revolution (motor shaft)-14
* Operating voltage-3.3-5V
* Hall effect sensors-2(at a phase 90)
* Magnetic disc-7 pole

Reasons to select:

* Compatible with micro metal geared motors
* High resolution 1400 pulses per revolution of main shaft
* Not effected by external electromagnetic noise

**3.2.3 Microcontroller:**

ATMEGA4809 also known as Arduino nano every is used as the main brain for the AGV



**Fig3.5 ATMEGA 4809**

Specifications:

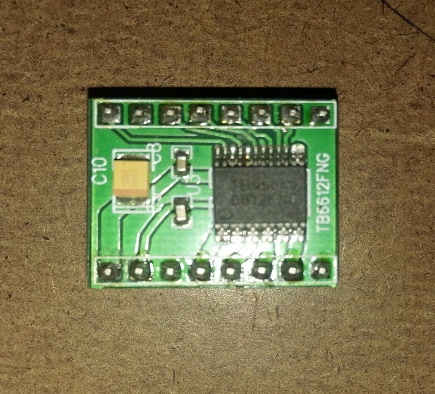
* Manufacturer-Arduino(Italy)
* Clock Speed-20MHz
* SRAM-6KB
* Flash memory-48KB
* EEPROM-256B
* Analog pins-8
* PWM pins-5
* Interrupt pins-All digital pins
* Serial ports-1

Reasons to select:

* Has 50% more memory compared to ATMEGA 328P
* Every digital pin can be used for external interrupt
* Cheaper than other boards

**3.2.4 Motor Driver:**

TB6612FNG dual channel motor driver is used to drive both the DC motors. It is manufactured by Toshiba.

**Fig3.6 Front View of Motor Driver Fig3.7 Back View of The Motor Driver**

Specifications:

* Manufacturer-Aimoni Robotics
* IC manufacturer-Toshibha
* Channels-2
* Max supply voltage-15V
* IC operating voltage-6V
* Output Current per channel-1.2A
* Operating temperature-(-20 to 85°C)

Reasons to select:

* IC is made of mosfets so less heat is dissipated (no need of heat sink)
* Stand By mode is provided which draws low current when there is no use of motors
* Has an internal thermal shutdown to protect the IC from high temperatures

**3.2.5 Wireless Module:**

Xbee series two is used for wireless communication for the AGV it gives a range of 400ft



**Fig3.8 Xbee Series 2**

Specifications:

* Manufacturer-DIGI international
* Operating voltage-3.3V
* Range-400ft(120m)
* Data rate-250kbps(max)
* Command modes available- AT & API

Reason to select:

* Can be easily interfaced with the Arduino
* Used for real time industrial applications

**3.2.6 Battery:**

The power source for the AGV is the rechargeble 7.4V Lithium Polymer battery



**Fig3.9 LIPO Battery**

Specifications:

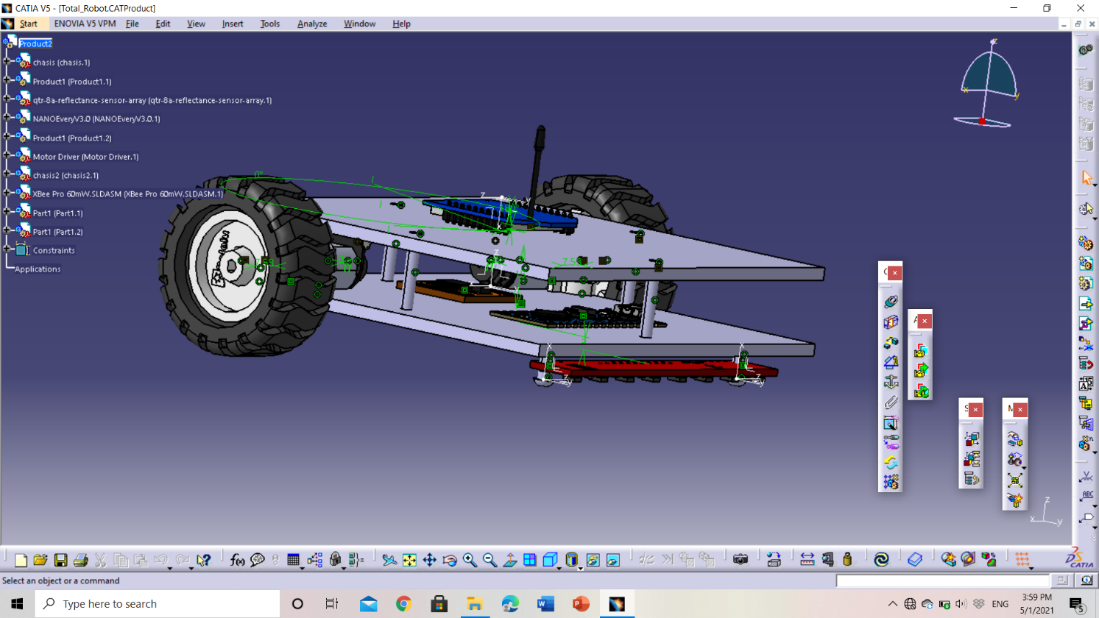
* Voltage-7.4V
* 850mah
* Maxdraw-2 amps
* Cells-2

Reasons to select:

* Lipo battery are light in weight compared to Lion batteries
* Lipo battery produces high current which is a need for actuators

**3.3 3D MODEL OF THE AGV**

The prototype of the AGV is modelled in CATIA V5

****

**Fig3.10 model of the AGV**

**3.4 PROTOTYPE OF THE FABRICATED AGV**

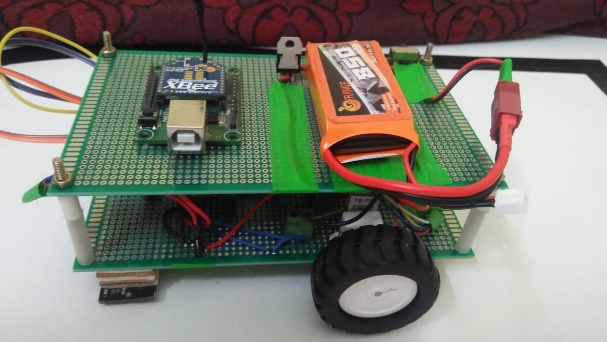
The chassis of the robot is made of perf board (prototype PCB) where all the hardware components are soldered and actuators are bolted. We also added a voltage regulation circuit which regulates the battery voltage to 5V.The microcontroller and motor driver are connected through header pins so that if anything damages it can be easily replaced **.**

**Dimensions:**

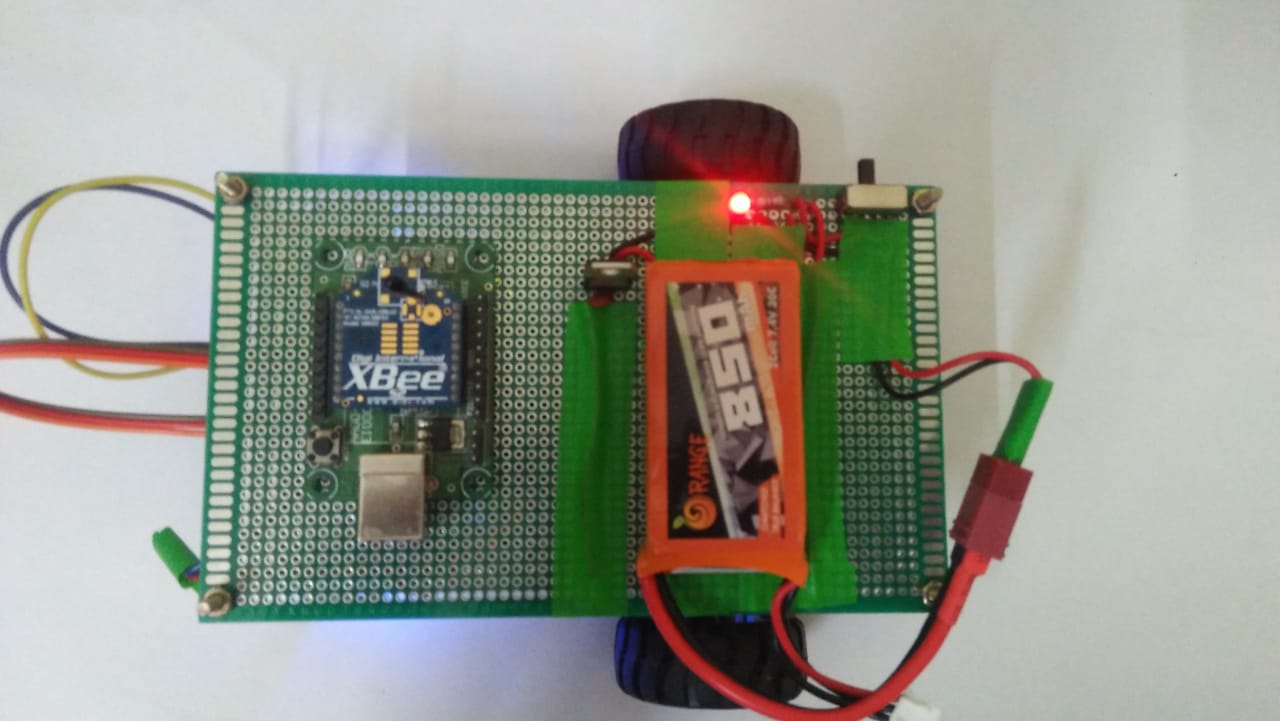
Length -150mm

Width(max) -150mm

Height -50mm

****

**Fig3.11 prototype of the AGV**

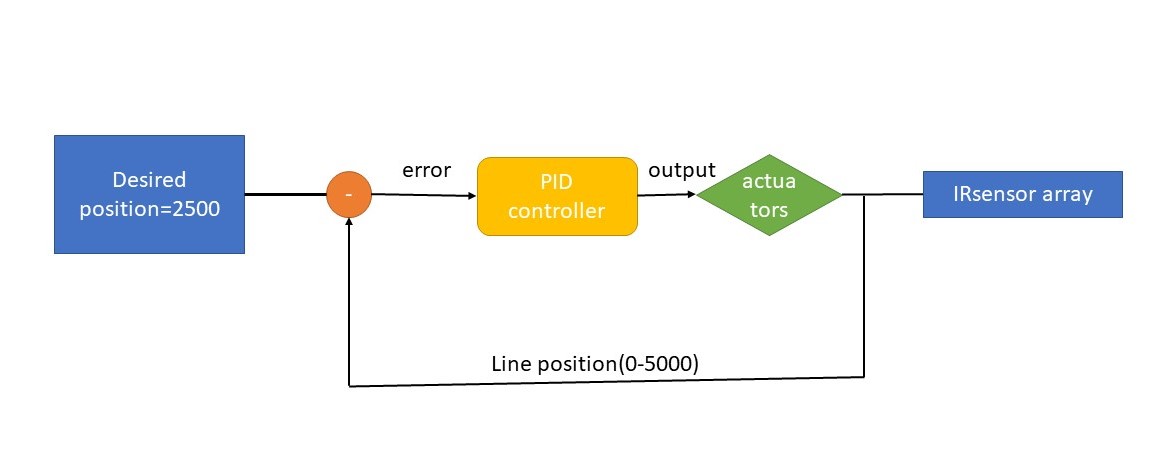
****

**Fig3.12 Top view of the AGV**

**CHAPTER 4: IMPLEMENTATION OF MAPPING AND PATH PLANNING**

**4.1TRAJECTORY PLANNING:**

To follow the guide path on the floor plan precisely a closed loop PID control system is implemented.The guide path is sensed by the IR sensor array which gives the position of the path below the robot from 0 to 5000 the desired position is 2500.So the line position is subtracted from the desired position this error is given as input to controller which gives an output to correct the error.

****

**Fig4.1 Block diagram of control system implemented**

**4.1.1 PID Tuning:**

The tuning of the PID control system is done manually where is proportional constant is calculated and rest are estimated by trial and error method.

Proportional constant:

Kp = base speed PWM/desired position = 150/2500 = 0.06

Derivative constant:

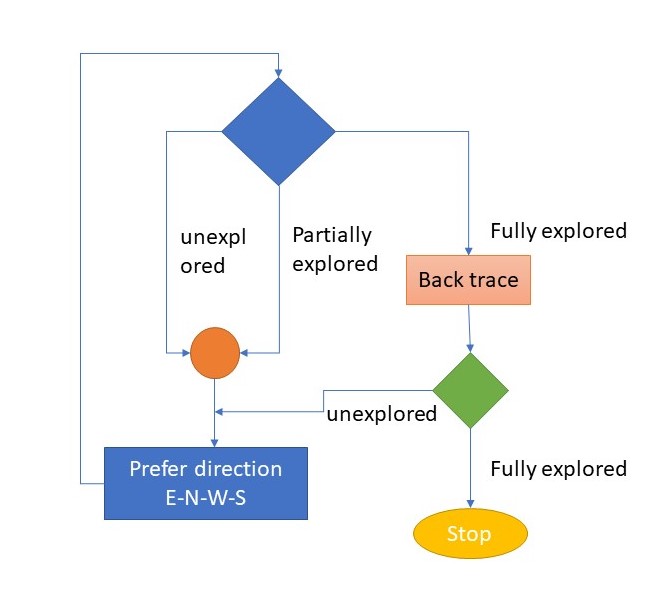
Kd = 0.5

Integral constant:

Ki = 0

**4.2 MAPPING USING OPTIMISED FLOOD FILL ALGORITHM:**

To map the entire floor plan made by a guide tape an optimised flood fill algorithm is implemented. The basic working is that AGV follows the guide tape until a node is detected. A node is nothing but a point which paves path to other nodes or nothing. Whenever a node is detected it may be fully explored, partially explored or unexplored. If the node is unexplored or partially explored the robot takes a direction according to preference of East-North-West-South sequence. If the detected node is fully explored then the robot back traces to a node which is partially explored if there is no partially explored node in its path then it reaches to the starting node and completes the mapping. This explored path is tracked along the way in an array which is converted graph data structure format called adjacent matrix list.

****

**Fig4.2 Flow chart of the mapping algorithm**

**4.2.1 Detection of Node:**

The outermost two IR sensors are used to detect the nodes whenever one or both values of the outermost sensors are greater than the threshold ‘L’,’T’ or ’ +’ type nodes are detected. When two sensors are less than the threshold then a ‘dead end’ node is detected.

**Table4.1 Detection of nodes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Type of node** | **Sensor 0** | **Logic** | **Sensor 5** |
| ‘L’ type node | >threshold | OR | >threshold |
| ‘T’ type node | >threshold | AND | >threshold |
| ‘+’ type node | >threshold | AND | >threshold |
| ‘Dead end’ type node | <threshold | AND | <threshold |

**4.2.2** **Distinguishing Each Node**

To know whether the node is explored or not explored it has to be identified by the robot first. So each node should be assigned with a unique identity.The identity in our project for each node are coordinates which are given by using encoders.The first node coordinates are taken as (0,0) and direction as ‘Y’ and hence the rest of the coordinates for other nodes are given with respective the first node.The coordinates of present node are assigned to temporary variables and then updated whenever direction of the bot is changed.

**Table 4.2 Updating coordinates of each node**

|  |  |  |
| --- | --- | --- |
| **Direction took** | **temp\_X** | **temp\_Y** |
| East | +distance traveled | +0 |
| North | +0 | +distance traveled |
| West | -distance traveled | +0 |
| South | +0 | -distance traveled |

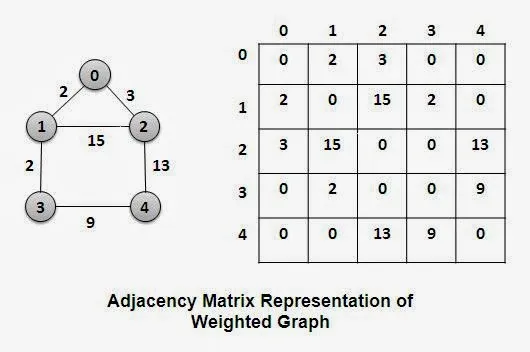
**4.2.3 Back Tracing**

During the exploration if an encountered node is fully explored the the AGV has to move to a an unexplored node in its traced path and begin its exploration.This is known as back tracing.If a regular flood fill algorithm is used then the AGV has to back trace in the same path which it came.This path may contain many loops which are eliminated in our proposed algorithm thus the distance traveled during back tracing is reduced compared to regular flood fill agorithm.Thus the time and power consumed during mapping is reduced.

Eg : back tracing path – 1>2>3>9>6>2>7 reduced to 1>2>7

**4.2.4 Storing The Map**

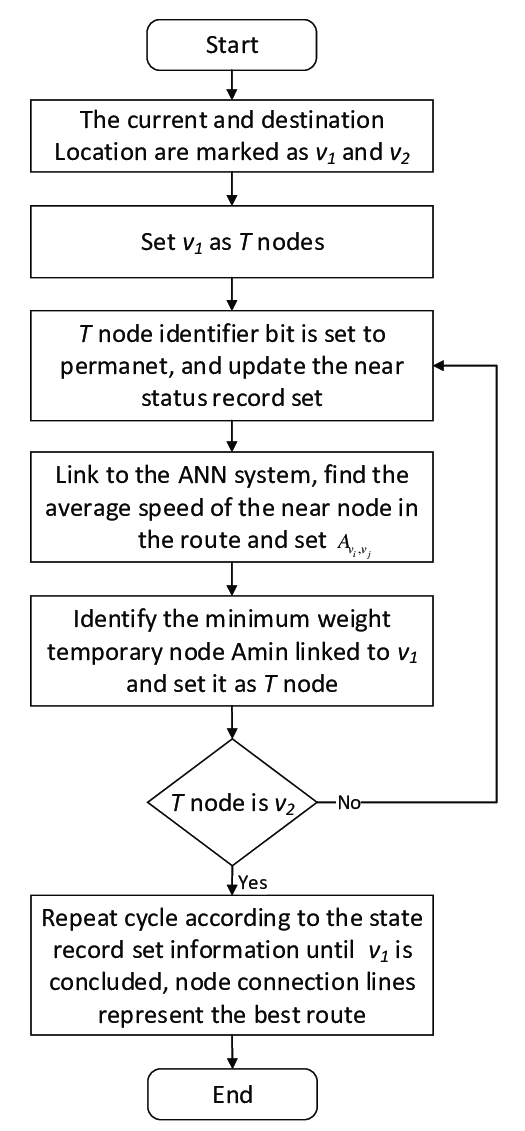
During the exploration the traced path is stored in an array .This traced path is stored in the form of graph data structure.The format of graph data structure used in this project is adjacent matrix list which is a two dimensional array where each index indicates a node .If there is a path between two nodes then the value is the distance between nodes if there is no path then the value is zero.



**Fig4.3 Representation of adjacent matrix list**

**4.3 PATH PLANNING USING DIJKSTRA’S ALGORITHM**

As the mapping completed autonomously next step is to deploy the AGV for performing operations from one node to another node.To move from one destination to another in the shortest path possible dijkstra’s algorithm is used .This is a greedy search based algorithm where a path which has least cost is selected on the given start and destination.The adjacent matrix from the mapping is converted into a cost matrix which is used to compute the cost of each possible path from a given start and destination.A least cost path is selected which is converted into a direction array which incates the direction to be taken at a node to reach the destination.The directions are computed by using coordinates of each node



**Fig4.4 Flow chart of dijkstra’s algorithm**

**4.3.1** **Sending Commands To The Robot**

To communicate with the AGV and send commands like start and destination for path planning we are using xbee series 2 modules.One module is directly connected to the system through USB and other module is connected to the microcontroller of the AGV .The xbee’s are configured in Transparent(AT) mode for communication through XCTU software

**Basic configuartion of Xbee modules:**

* BAUD rate – 9600
* Bits per packet- 8bits
* Stop bit -1
* Parity bit – None
* Mode -AT
* Protocol - zigbee

During the configuration one module is set to coordinator and other as router .The coordinator is connected to laptop and router to the AGV.This has a range 400ft in open area and 200ft in urban area

**Table4.3 parameters for configuration of Xbee modules**

|  |  |  |
| --- | --- | --- |
| **parameter** | **Coordinator** | **Router** |
| PAN ID | 84 | 84 |
| Destination High | 0013A200 | 0013A200 |
| Destination Low | 40D8581F | 40D8582E |

**CHAPTER 5: EXECUTABLE CODE FOR THE AGV**

**5.1** **SOFTWARE USED FOR PROGRAMMING MICROCONTROLLER**

The software used for programming microcontroller is Arduino IDE also the serial terminal in the software is used to communicate wirelessly with the AGV and also used to debug the errors if encountered

**5.1.1About Arduino**

Arduino is an open-source electronics platform that uses simple hardware and software to make it easy to use. Arduino boards can take inputs - such as light from a sensor, a finger on a button, or a Twitter message - and convert them to outputs - such as turning on an LED, triggering a motor, or publishing anything online. By providing a set of instructions to the board's microcontroller, you may tell it what to do. The Arduino programming language (based on Wiring) and the Arduino Software (IDE) (based on Processing) are used to accomplish this.Thousands of projects have used Arduino throughout the years, ranging from simple household items to complicated scientific apparatus. A global community of makers - students, enthusiasts, and professionals.

**5.2 CODE FOR PROGRAMMING THE AGV**

**/\*PROJECT : Autonomous Mapping and Path Planning Using a Differential Drive Based Automated Guided Vehicle**

**Date : 23 april 2021**

**Developed By : Sai Kumar**

**Geswanth**

**Teja Sundhar**

**Rajesh\*/**

**//Including some libraries**

**#include <QTRSensors.h> //IR sensor array library**

**#include <Encoder.h> //Quadrarture Encoders library**

**//defining all pins for pin manpulation using Registers**

**#define PWMA\_PIN 2 //port B (D5) /\***

**#define PWMB\_PIN 4 //Port F (D6)**

**#define R1\_PIN 6 //Port C (D4)**

**#define R2\_PIN 0 //Port A (D2) For MOTOR DRIVER**

**#define L1\_PIN 1 //Port A (D7)**

**#define L2\_PIN 3 //Port E (D8)**

**#define STNDBY\_PIN 2 //Port E (D13) \*/**

**#define L\_OUTA\_PIN 4 //Port D (A6) /\***

**#define L\_OUTB\_PIN 5 //Port D (A7) For ENCODERS**

**#define R\_OUTA\_PIN 0 //Port E (D11)**

**#define R\_OUTB\_PIN 1 //Port E (D12) \*/**

**//defining pins for regular Arduino functions**

**#define PWML 6 //PWM for Left motor**

**#define PWMR 5 //PWM for Right Motor**

**#define L\_OUTA A6 //left motor encoder output A**

**#define L\_OUTB A7 //left motor encoder output B**

**#define R\_OUTA 11 //right motor encoder output A**

**#define R\_OUTB 12 //right motor encoder output B**

**//Creating objects for the mentioned library clases**

**QTRSensors LineSensor;**

**Encoder LeftEncoder(L\_OUTB,L\_OUTA);**

**Encoder RightEncoder(R\_OUTA,R\_OUTB);**

**// Some important Macros**

**#define MTR\_SLEEP PORTE.OUT &=~(1<<STNDBY\_PIN); //Keeps the motor driver in stand by mode(draws low current)**

**#define MTR\_WAKEUP PORTE.OUT |=(1<<STNDBY\_PIN); //wakes up the motor driver**

**#define LEFTMOTOR\_FWRD PORTE.OUT &=~(1<<L2\_PIN);PORTA.OUT |=(1<<L1\_PIN); //set the left motor direction to move forward**

**#define LEFTMOTOR\_BWRD PORTE.OUT |=(1<<L2\_PIN);PORTA.OUT &=~(1<<L1\_PIN); //set the left motor direction to move backward**

**#define RIGHTMOTOR\_FWRD PORTA.OUT &=~(1<<R2\_PIN); PORTC.OUT |=(1<<R1\_PIN); //set the right motor direction to move forward**

**#define RIGHTMOTOR\_BWRD PORTA.OUT |=(1<<R2\_PIN);PORTC.OUT &=~(1<<R1\_PIN); //set the right motor direction to move backward**

**#define LEFTMOTOR\_STOP PORTE.OUT |=(1<<L2\_PIN);PORTA.OUT |=(1<<L1\_PIN); //Set left motor to stop**

**#define RIGHTMOTOR\_STOP PORTA.OUT |=(1<<R2\_PIN);PORTC.OUT |=(1<<R1\_PIN); //Set right motor to stop**

**#define BOTHMOTORS\_STOP PORTE.OUT |=(1<<L2\_PIN);PORTA.OUT |=(1<<L1\_PIN);PORTA.OUT |=(1<<R2\_PIN);PORTC.OUT |=(1<<R1\_PIN); //set both motors to stop**

**// defining constants**

**#define KP 0.06 // proportional constant**

**#define KI 0 // Integral constant**

**#define KD 0.04 // Derivative constant**

**#define BaseSpeed 150**

**#define MaxSpeed 150**

**#define threshold 550**

**#define speedturn 70**

**#define Hspeedturn 100**

**#define delayspeed 90**

**#define leftrotationspeed 65**

**#define rightrotationspeed 60**

**#define E 0**

**#define N 90**

**#define W 180**

**#define S 270**

**#define STRAIGHT 1**

**#define LEFT 2**

**#define RIGHT 3**

**#define BACK 4**

**#define STOP 5**

**#define CYCLE\_PER\_CM 101.28041**

**#define CM\_PER\_CYCLE 0.0098735**

**#define Radius 6.7 //half of the width between wheels(in Cm)**

**#define Half\_of\_line\_thickness 1.6 //half of the thickness of guide path**

**#define INF 9999**

**#define MAX 30**

**//definining global variables**

**int present\_direction = N ;**

**uint8\_t present\_node = 1 ;**

**uint8\_t no\_of\_nodes = 1 ;**

**int temp\_X = 0 ;**

**int temp\_Y = 0 ;**

**int \*Co\_X; //pointer for Xcoordinate array**

**int \*Co\_Y; //pointer for Ycoordinate array**

**uint8\_t \*nodes\_track; //pointer for Node tracking array //'''''''''''''''' check the data types of the global variables'''''''''''''''''''**

**unsigned char \*East, \*North, \*West, \*South; //pointers for the explored direction array**

**double dist;**

**uint8\_t size\_of\_node\_array=1;**

**uint8\_t actual\_size\_of\_nodes\_array=1;**

**const uint8\_t SensorCount = 6;**

**uint16\_t sensorValues[SensorCount];**

**uint16\_t line\_position;**

**float Calibration\_Constant=1; //calibration constant for distance measuring**

**unsigned char distance\_flag=1;**

**uint16\_t ADJ\_matrix[MAX][MAX];**

**void setup() {**

**// put your setup code here, to run once:**

**//intialising some variables**

**Serial1.begin(9600);**

**//while(!Serial1)**

**Intialise();**

**MTR\_WAKEUP**

**int i,j;**

**nodes\_track=(uint8\_t\*)malloc(1\*sizeof(uint8\_t));**

**East=(unsigned char\*)malloc(1\*sizeof(unsigned char));**

**North=(unsigned char\*)malloc(1\*sizeof(unsigned char));**

**West=(unsigned char\*)malloc(1\*sizeof(unsigned char));**

**South=(unsigned char\*)malloc(1\*sizeof(unsigned char));**

**Co\_X=(int\*)malloc(1\*sizeof(int));**

**Co\_Y=(int\*)malloc(1\*sizeof(int));**

**Co\_X[0] = 0; //co\_ordinates of first Node**

**Co\_Y[0] = 0;**

**nodes\_track[0]=1;**

**East[0]=1;**

**North[0]=1;**

**West[0]=1;**

**South[0]=1;**

**for(i=0;i<MAX;i++) //intialising the adjacent matrix list values to zero**

**{**

**for(j=0;j<MAX;j++)**

**{**

**ADJ\_matrix[i][j]=0;**

**}**

**}**

**delay(5000);**

**Serial1.print("start calibrating");**

**Serial1.print("\n");**

**RIGHTMOTOR\_FWRD**

**LEFTMOTOR\_BWRD**

**analogWrite(PWML,150);**

**analogWrite(PWMR,150);**

**for (uint16\_t i = 0; i < 200; i++)**

**{**

**LineSensor.calibrate();**

**}**

**BOTHMOTORS\_STOP**

**Serial1.print("Completed calibrating");**

**Serial1.print("\n");**

**delay(5000);**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_FWRD**

**while(1){**

**line\_position=LineSensor.readLineBlack(sensorValues);**

**if (sensorValues[0] > threshold || sensorValues[1] > threshold || sensorValues[2] > threshold || sensorValues[3] > threshold || sensorValues[4] > threshold || sensorValues[5] > threshold)**

**{**

**BOTHMOTORS\_STOP**

**break;**

**}**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_FWRD**

**analogWrite(PWML,leftrotationspeed);**

**analogWrite(PWMR,rightrotationspeed);**

**}**

**Serial1.println("Mapping the arena");**

**map\_the\_arena();**

**Serial1.println("Completed mapping");**

**Serial1.println();**

**Serial1.println();**

**Serial1.println("Adjacent matrix list");**

**for(i=0;i<no\_of\_nodes;i++)**

**{**

**for(j=0;j<no\_of\_nodes;j++)**

**{**

**Serial1.print(ADJ\_matrix[i][j]);**

**Serial1.print("\t");**

**}**

**Serial1.print("\n");**

**}**

**Serial1.println("Enter the start and destination ");**

**delay(5000);**

**plan\_path(0,4);**

**Serial1.println("Enter the start and destination ");**

**delay(2000);**

**plan\_path(4,1);**

**Serial1.println("Enter the start and destination ");**

**delay(2000);**

**plan\_path(1,3);**

**}**

**void loop() {**

**// put your main code here, to run repeatedly:**

**/\*uint8\_t start,destination;**

**if(Serial1.available()>0)**

**{**

**start=Serial1.parseInt();**

**destination=Serial1.parseInt();**

**plan\_path(start-1,destination-1);**

**Serial1.println("Enter the start and destination ");**

**}\*/**

**}**

**//Fuction to intialise all the pins to the required state**

**void Intialise()**

**{**

**Serial1.print("Intialising Port Variables");**

**Serial1.print("\n");**

**PORTD.DIR &=~((1<<L\_OUTA\_PIN)|(1<<L\_OUTB\_PIN ));**

**PORTE.DIR &=~((1<<R\_OUTA\_PIN)|(1<<R\_OUTB\_PIN )); //intialising the port pins to act as input with pullup resisters for encoders**

**PORTD.PIN4CTRL |=(1<<3);**

**PORTD.PIN5CTRL |=(1<<3);**

**PORTE.PIN4CTRL |=(1<<3);**

**PORTE.PIN4CTRL |=(1<<3);**

**PORTE.DIR |=((1<<L2\_PIN)|(1<<STNDBY\_PIN));**

**PORTA.DIR |=((1<<L1\_PIN)|(1<<R2\_PIN)|(1<<3)); //intialising the port pins to act as output for motor driver and aetting the output to STOP mode**

**PORTC.DIR |=(1<<R1\_PIN);**

**PORTB.DIR |=(1<<PWMA\_PIN);**

**PORTF.DIR |=(1<<PWMB\_PIN);**

**/\*analogWrite(PWML,0);**

**analogWrite(PWMR,0);\*/**

**PORTD.DIR &=~((1<<0)|(1<<1)|(1<<2)|(1<<3)|(1<<4));**

**PORTA.DIR &=~((1<<2)|(1<<3)); //intialising the analolog pins and sensor library for IR sensor array**

**LineSensor.setTypeAnalog();**

**LineSensor.setSensorPins((const uint8\_t[]){A0, A1, A2, A3, A4, A5},SensorCount);**

**LineSensor.setSamplesPerSensor(4); // '''''''''''''''''''set the samples '''''''''''''''''''''''''''''**

**return;**

**}**

**//function to turn the robot acoording to mentioned dir**

**void turn(uint8\_t dir,unsigned char move\_straight)**

**{**

**switch (dir)**

**{**

**///////////////////////////Go straight\\\\\\\\\\\\\\\\\\\\\\\\\\\\\**

**case STRAIGHT:**

**//Serial1.println("Moving straight");**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_FWRD**

**motor\_speed(150,150);**

**LineSensor.readLineBlack(sensorValues);**

**while(sensorValues[0]>threshold && sensorValues[5]>threshold)**

**{**

**LineSensor.readLineBlack(sensorValues);**

**}**

**break;**

**/////////////////////////Turn Left 90deg\\\\\\\\\\\\\\\\\\\\\\\\\\\\\**

**case LEFT:**

**//Serial1.println("Turning Left");**

**if(move\_straight)**

**{**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_FWRD**

**motor\_speed(150,150);**

**delay(350);**

**}**

**LEFTMOTOR\_BWRD**

**RIGHTMOTOR\_FWRD**

**motor\_speed(speedturn,speedturn);**

**line\_position = LineSensor.readLineBlack(sensorValues);**

**while (sensorValues[5] < threshold)**

**{**

**line\_position = LineSensor.readLineBlack(sensorValues);**

**}**

**motor\_speed(Hspeedturn,Hspeedturn);**

**line\_position = LineSensor.readLineBlack(sensorValues);**

**while (sensorValues[3] < threshold || sensorValues[4] < threshold) // wait for outer most sensor to find the line**

**{**

**line\_position = LineSensor.readLineBlack(sensorValues);**

**}**

**RIGHTMOTOR\_FWRD**

**LEFTMOTOR\_STOP**

**analogWrite(PWMR, 255);**

**break;**

**///////////////////////Turn Right 90deg\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\**

**case RIGHT:**

**//Serial1.println("Turning Right");**

**if(move\_straight)**

**{**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_FWRD**

**motor\_speed(150,150);**

**delay(350);**

**}**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_BWRD**

**motor\_speed(speedturn,speedturn);**

**line\_position = LineSensor.readLineBlack(sensorValues);**

**while (sensorValues[0] < threshold) // wait for outer most sensor to find the line**

**{**

**line\_position = LineSensor.readLineBlack(sensorValues);**

**}**

**motor\_speed(Hspeedturn,Hspeedturn);**

**line\_position = LineSensor.readLineBlack(sensorValues);**

**while (sensorValues[2] < threshold || sensorValues[1] < threshold) // wait for outer most sensor to find the line**

**{**

**line\_position = LineSensor.readLineBlack(sensorValues);**

**}**

**RIGHTMOTOR\_STOP**

**LEFTMOTOR\_FWRD**

**analogWrite(PWML, 255);**

**break;**

**////////////////////////Turn right 180deg to go back\\\\\\\\\\\\\\\\\\\\\\\**

**case BACK:**

**//Serial1.println("Turning Back");**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_BWRD**

**motor\_speed(speedturn,speedturn);**

**delay(2000);**

**line\_position = LineSensor.readLineBlack(sensorValues);**

**while (sensorValues[0] < threshold) // wait for outer most sensor to find the line**

**{**

**line\_position = LineSensor.readLineBlack(sensorValues);**

**}**

**motor\_speed(Hspeedturn,Hspeedturn);**

**line\_position = LineSensor.readLineBlack(sensorValues);**

**while (sensorValues[2] < threshold || sensorValues[1] < threshold) // wait for outer most sensor to find the line**

**{**

**line\_position = LineSensor.readLineBlack(sensorValues);**

**}**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_BWRD**

**analogWrite(PWML, 255);**

**break;**

**////////////////////////Stop the robot\\\\\\\\\\\\\\\\\\\\\\\**

**case STOP:**

**//Serial1.println("Stop");**

**BOTHMOTORS\_STOP**

**analogWrite(PWMR,0);**

**analogWrite(PWML,0);**

**break;**

**}**

**}**

**// Move straight to a specified distance**

**void Straight(float straight\_distance,unsigned char \*left\_found,unsigned char\*right\_found)**

**{**

**uint8\_t drum=1;**

**Serial1.print("moving straight by ");**

**Serial1.println(straight\_distance);**

**unsigned int cycles=straight\_distance\*CYCLE\_PER\_CM;**

**RightEncoder.write(0);**

**LeftEncoder.write(0);**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_FWRD**

**while (1)**

**{**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_FWRD**

**analogWrite(PWML,leftrotationspeed);**

**analogWrite(PWMR,rightrotationspeed);**

**if (((LeftEncoder.read()+RightEncoder.read())\*0.5)>=cycles)**

**{**

**BOTHMOTORS\_STOP//stop the motors**

**Serial1.println("moved straight");**

**return;**

**}**

**if(drum)**

**{**

**LineSensor.readLineBlack(sensorValues);**

**if (sensorValues[0] > threshold)**

**{**

**for (int i = 0; i < 60; i++)**

**{**

**LineSensor.readLineBlack(sensorValues);**

**if (sensorValues[5] > threshold)**

**{**

**\*left\_found = 1;**

**}**

**}**

**\*right\_found = 1;**

**}**

**else if (sensorValues[5] > threshold)**

**{**

**for (int i = 0; i < 60; i++)**

**{**

**LineSensor.readLineBlack(sensorValues);**

**if (sensorValues[0] > threshold)**

**{**

**\*right\_found = 1;**

**}**

**}**

**\*left\_found = 1;**

**}**

**Serial1.print("Found right ");**

**Serial1.print(\*right\_found);**

**Serial1.print(" Found left ");**

**Serial1.print(\*left\_found);**

**drum=0;**

**}**

**}**

**}**

**// Rotate to a specified angle about mid axis**

**void Rotate(int angle)**

**{**

**//reset the encoders**

**BOTHMOTORS\_STOP**

**RightEncoder.write(0);**

**LeftEncoder.write(0);**

**if (angle < 0) //Rotate anti clock wise about mid axis**

**{**

**Serial1.print("Roataing anti clockwise by ");**

**Serial1.println(angle);**

**uint16\_t cycles = ((0.0174532\*angle\*Radius\*CYCLE\_PER\_CM)\*-1)-50; //Expression to determine required cycles to rotate the bot with predetermined angle**

**LEFTMOTOR\_BWRD**

**RIGHTMOTOR\_FWRD**

**while (1)**

**{**

**LEFTMOTOR\_BWRD**

**RIGHTMOTOR\_FWRD**

**analogWrite(PWML,leftrotationspeed);**

**analogWrite(PWMR,rightrotationspeed);**

**if (((RightEncoder.read()-LeftEncoder.read())\*0.5)>= cycles)**

**{**

**BOTHMOTORS\_STOP//stop the motors**

**analogWrite(PWML,0);**

**analogWrite(PWMR,0);**

**break;**

**}**

**}**

**}**

**else if(angle>0) //Rotate clock wise about mid axis**

**{**

**Serial1.print("Roataing clockwise by ");**

**Serial1.println(angle);**

**uint16\_t cycles = (0.0174532\*angle\*Radius\*CYCLE\_PER\_CM)-50; //Expression to determine required cycles to rotate the bot with predetermined angle**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_BWRD**

**while (1)**

**{**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_BWRD**

**analogWrite(PWML,leftrotationspeed);**

**analogWrite(PWMR,rightrotationspeed);**

**if (((LeftEncoder.read()-RightEncoder.read())\*0.5)>=cycles)**

**{**

**BOTHMOTORS\_STOP//stop the motors**

**analogWrite(PWML,0);**

**analogWrite(PWMR,0);**

**break;**

**}**

**}**

**}**

**else**

**{**

**return;**

**}**

**/\*line\_position=LineSensor.readLineBlack(sensorValues);**

**if(line\_position>2500)**

**{**

**LEFTMOTOR\_BWRD**

**RIGHTMOTOR\_FWRD**

**while(line\_position>2500)**

**{**

**line\_position=LineSensor.readLineBlack(sensorValues);**

**LEFTMOTOR\_BWRD**

**RIGHTMOTOR\_FWRD**

**analogWrite(PWML,leftrotationspeed);**

**analogWrite(PWMR,rightrotationspeed);**

**}**

**BOTHMOTORS\_STOP**

**return;**

**}**

**else if(line\_position<2500)**

**{**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_BWRD**

**while(line\_position<2500)**

**{**

**line\_position=LineSensor.readLineBlack(sensorValues);**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_BWRD**

**analogWrite(PWML,leftrotationspeed);**

**analogWrite(PWMR,rightrotationspeed);**

**}**

**BOTHMOTORS\_STOP**

**return;**

**}\*/**

**return;**

**}**

**//rotate to specified direction**

**void Go\_To\_Direction(int go\_to\_direction)**

**{**

**Serial1.println("Computing angle required");**

**int theta1 = present\_direction;**

**int theta2 = go\_to\_direction;**

**switch (go\_to\_direction)**

**{**

**case E:**

**switch (present\_direction)**

**{**

**case S:**

**theta2 += 360;**

**break;**

**}**

**break;**

**case S:**

**switch (present\_direction)**

**{**

**case E:**

**theta1 += 360;**

**break;**

**}**

**break;**

**}**

**int angle = theta1 - theta2;**

**Rotate(angle);**

**present\_direction = go\_to\_direction;**

**return;**

**}**

**//fuction to rotate the robot according to the preference 'E-N-W-S' if not possible return true**

**bool prefer\_the\_direction(int node)**

**{**

**Serial1.print("Node ");**

**Serial1.print(node);**

**Serial1.print(" Selecting the direction according to the preference E-N-W-S ");**

**Serial1.print("E ");**

**Serial1.print(East[present\_node-1]);**

**Serial1.print(" N ");**

**Serial1.print(North[present\_node-1]);**

**Serial1.print(" W ");**

**Serial1.print(West[present\_node-1]);**

**Serial1.print(" S ");**

**Serial1.print(South[present\_node-1]);**

**Serial1.print("\n");**

**if(!East[node-1])**

**{**

**Serial1.println("Moving East");**

**Go\_To\_Direction(E);**

**East[node-1]=1;**

**return false;**

**}**

**else if(!North[node-1])**

**{**

**Serial1.println("Moving North");**

**Go\_To\_Direction(N);**

**North[node-1]=1;**

**return false;**

**}**

**else if(!West[node-1])**

**{**

**Serial1.println("Moving West");**

**Go\_To\_Direction(W);**

**West[node-1]=1;**

**return false;**

**}**

**else if(!South[node-1])**

**{**

**Serial1.println("Moving south");**

**Go\_To\_Direction(S);**

**South[node-1]=1;**

**return false;**

**}**

**else**

**{**

**Serial1.println("Found a Dead end");**

**return true;**

**}**

**}**

**// updates the co\_ordinates after reaching a new node**

**void update\_temp\_coordinates() // check the data type of dist in all functions**

**{**

**Serial1.println("Updating the coordinates");**

**switch (present\_direction)**

**{**

**case E:**

**temp\_X += int(dist);**

**break;**

**case N:**

**temp\_Y += int(dist);**

**break;**

**case W:**

**temp\_X -= int(dist);**

**break;**

**case S:**

**temp\_Y -= int(dist);**

**break;**

**}**

**}**

**//To reduce the error between motor speeds**

**void motor\_speed(int left,int right)**

**{**

**if(right<41)**

**right-=4;**

**else if(right>=41 && right<64)**

**right-=5;**

**else if(right>=64 && right<96)**

**right-=6;**

**else if(right>=96 && right<132)**

**right-=7;**

**else if(right>=132 && right<156)**

**right-=8;**

**analogWrite(PWML,left);**

**analogWrite(PWMR,right);**

**return;**

**}**

**//follow the BLACK LINE Trajectory untill you detect a NODE**

**float follow\_line() //follow the line**

**{**

**Serial1.println("Following the trajectory");**

**int divi;**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_FWRD**

**int lastError = 0;**

**int i;**

**dist=0;**

**RightEncoder.write(0);**

**LeftEncoder.write(0);**

**while (1)**

**{**

**line\_position = LineSensor.readLineBlack(sensorValues);**

**if (sensorValues[0] > threshold || sensorValues[5] > threshold)// Found a node**

**{**

**Serial1.print("Found a Node and distance ");**

**dist=((LeftEncoder.read()+RightEncoder.read())\*0.5\*CM\_PER\_CYCLE\*Calibration\_Constant)+Half\_of\_line\_thickness+9.3;**

**/\*Serial1.print((LeftEncoder.read()+RightEncoder.read())\*0.5);**

**Serial1.print(" calibrated distance ");**

**Serial1.print(dist);\*/**

**BOTHMOTORS\_STOP**

**if(distance\_flag)**

**{**

**dist-=9.3;**

**distance\_flag=0;**

**}**

**divi=dist/5;**

**if(dist>(5\*divi+2.5))**

**dist=5\*divi+5;**

**else**

**dist=5\*divi;**

**//Serial1.println(dist);**

**return Half\_of\_line\_thickness;**

**}**

**if (sensorValues[0] < threshold && sensorValues[1] < threshold && sensorValues[2] < threshold && sensorValues[3] < threshold && sensorValues[4] < threshold && sensorValues[5] < threshold )// Found a Dead end**

**{**

**Serial1.println("Found a Dead end");**

**dist=(LeftEncoder.read()+RightEncoder.read())\*0.5\*CM\_PER\_CYCLE\*Calibration\_Constant+9.3;**

**BOTHMOTORS\_STOP**

**if(distance\_flag)**

**{**

**dist-=9.3;**

**distance\_flag=0;**

**}**

**divi=dist/5;**

**if(dist>(5\*divi+2.5))**

**dist=5\*divi+5;**

**else**

**dist=5\*divi;**

**//Serial1.println(dist);**

**return 0;**

**}**

**int error = line\_position - 2500;**

**int error1 = error - lastError;**

**int error2 = (2.0 / 3.0) \* error2 + error ;**

**int motorSpeed = KP \* error + KD \* error1 + KI \* error2;**

**int rightMotorSpeed = BaseSpeed + motorSpeed;**

**int leftMotorSpeed = BaseSpeed - motorSpeed;**

**if (rightMotorSpeed > MaxSpeed ) rightMotorSpeed = MaxSpeed; // prevent the motor from going beyond max speed**

**if (leftMotorSpeed > MaxSpeed ) leftMotorSpeed = MaxSpeed; // prevent the motor from going beyond max speed**

**if (rightMotorSpeed < 0)rightMotorSpeed = 0;**

**if (leftMotorSpeed < 0)leftMotorSpeed = 0;**

**motor\_speed(leftMotorSpeed,rightMotorSpeed);**

**lastError = error;**

**}**

**}**

**//function to set directions for new node**

**void set\_directions\_for\_new\_node(unsigned char found\_left,unsigned char found\_straight,unsigned char found\_right )**

**{**

**Serial1.println("Setting directions for new node");**

**switch(present\_direction)**

**{**

**case E:**

**West[present\_node-1]=1; //''''''''''''''''reallocate memory for East-South,Co\_X,Co\_Y pointer when u update the present variable'''''''''''''''''**

**East[present\_node-1]=found\_straight;**

**North[present\_node-1]=found\_left;**

**South[present\_node-1]=found\_right;**

**break;**

**case N:**

**South[present\_node-1]=1;**

**North[present\_node-1]=found\_straight;**

**West[present\_node-1]=found\_left;**

**East[present\_node-1]=found\_right;**

**break;**

**case W:**

**East[present\_node-1]=1;**

**West[present\_node-1]=found\_straight;**

**South[present\_node-1]=found\_left;**

**North[present\_node-1]=found\_right;**

**break;**

**case S:**

**North[present\_node-1]=1;**

**South[present\_node-1]=found\_straight;**

**East[present\_node-1]=found\_left;**

**West[present\_node-1]=found\_right;**

**break;**

**}**

**Serial1.print("E ");**

**Serial1.print(East[present\_node-1]);**

**Serial1.print(" N ");**

**Serial1.print(North[present\_node-1]);**

**Serial1.print(" W ");**

**Serial1.print(West[present\_node-1]);**

**Serial1.print(" S ");**

**Serial1.print(South[present\_node-1]);**

**Serial1.print("\n");**

**return;**

**}**

**// Function to check the node if its new or old**

**bool check\_if\_new()**

**{**

**Serial1.println("Check the coordinate if new or old");**

**int i;**

**unsigned char is\_new=1;**

**for(i=0;i<no\_of\_nodes;i++)**

**{**

**if(temp\_X==Co\_X[i] && temp\_Y==Co\_Y[i]) //checking whether the current coordinates present in the nodes track coordinates**

**{**

**is\_new=0;**

**break;**

**}**

**}**

**if(size\_of\_node\_array==actual\_size\_of\_nodes\_array) // if the nodes track is out of memory reallocate the memory**

**{**

**Serial1.println("Reallocating memory for nodes track");**

**actual\_size\_of\_nodes\_array+=1;**

**nodes\_track=(uint8\_t\*)realloc(nodes\_track,actual\_size\_of\_nodes\_array\*sizeof(uint8\_t));**

**//free(temp\_addr);**

**}**

**size\_of\_node\_array+=1;**

**if(is\_new)**

**{**

**Serial1.print("It is a new node ");**

**no\_of\_nodes+=1;**

**present\_node=no\_of\_nodes;**

**nodes\_track[size\_of\_node\_array-1]=present\_node;**

**Serial1.print(" Nodes track ");**

**for(i=0;i<size\_of\_node\_array;i++)**

**{**

**Serial1.print(nodes\_track[i]);**

**Serial1.print("\t");**

**}**

**Serial1.print("\n");**

**return true;**

**}**

**else**

**{**

**Serial1.print("It is a old node ");**

**present\_node=i+1;**

**nodes\_track[size\_of\_node\_array-1]=present\_node;**

**Serial1.print(" Nodes track ");**

**for(i=0;i<size\_of\_node\_array;i++)**

**{**

**Serial1.print(nodes\_track[i]);**

**Serial1.print("\t");**

**}**

**Serial1.print("\n");**

**return false;**

**}**

**}**

**//detect the node**

**void check\_for\_node()**

**{**

**float extra\_dist=follow\_line();**

**LeftEncoder.write(0);**

**RightEncoder.write(0);**

**unsigned char found\_left = 0;**

**unsigned char found\_straight = 0;**

**unsigned char found\_right = 0;**

**Straight(extra\_dist+9,&found\_left,&found\_right);**

**line\_position = LineSensor.readLineBlack(sensorValues);**

**if (sensorValues[2] > threshold || sensorValues[3] > threshold)**

**found\_straight = 1;**

**Serial1.print("found straight ");**

**Serial1.println(found\_straight);**

**Serial1.print("Distance ttravelled ");**

**Serial1.println(dist);**

**update\_temp\_coordinates(); //updates temporary co ordinates**

**Serial1.print("Coordiantes ");**

**Serial1.print(temp\_X);**

**Serial1.print("\t");**

**Serial1.println(temp\_Y);**

**if(check\_if\_new()) //'''''''''create this fuction to check the node is new(true) or old(false) also update the node variable and the nodes track(by reallocing the array) also sizes of nodes array ''''''''''''''''''''**

**{**

**// A function to find the direction for the node and update the direction arrays (by reallocing the arrays) and co\_ordinates**

**Serial1.println("Reallocating memory for all variables for the new node");**

**East=(unsigned char\*)realloc(East,present\_node\*sizeof(unsigned char));**

**North=(unsigned char\*)realloc(North,present\_node\*sizeof(unsigned char));**

**West=(unsigned char\*)realloc(West,present\_node\*sizeof(unsigned char));**

**South=(unsigned char\*)realloc(South,present\_node\*sizeof(unsigned char));**

**Co\_X=(int\*)realloc(Co\_X,present\_node\*sizeof(int));**

**Co\_Y=(int\*)realloc(Co\_Y,present\_node\*sizeof(int));**

**Co\_X[present\_node-1]=temp\_X;**

**Co\_Y[present\_node-1]=temp\_Y;**

**set\_directions\_for\_new\_node(!(found\_left),!(found\_straight),!(found\_right));**

**}**

**else**

**{**

**Serial1.println("Updating the E-N-W-S varibles");**

**switch(present\_direction)**

**{**

**case E:**

**West[present\_node-1]=1;**

**break;**

**case N:**

**South[present\_node-1]=1;**

**break;**

**case W:**

**East[present\_node-1]=1;**

**break;**

**case S:**

**North[present\_node-1]=1;**

**break;**

**}**

**}**

**return ;**

**}**

**//function to find direction for back tracking and path planning**

**void find\_directions(int indx,uint8\_t \*tracing\_path,uint8\_t \*tracing\_directions)**

**{**

**int temp\_direction=present\_direction;**

**int go\_to,i;**

**for(i=0;i<indx-1;i++)**

**{**

**if(Co\_Y[tracing\_path[i]-1]==Co\_Y[tracing\_path[i+1]-1])**

**{**

**if(Co\_X[tracing\_path[i]-1]>Co\_X[tracing\_path[i+1]-1])**

**go\_to=W;**

**else**

**go\_to=E;**

**}**

**else**

**{**

**if(Co\_Y[tracing\_path[i]-1]>Co\_Y[tracing\_path[i+1]-1])**

**go\_to=S;**

**else**

**go\_to=N;**

**}**

**switch(temp\_direction)**

**{**

**case E:**

**switch(go\_to)**

**{**

**case E:**

**tracing\_directions[i]=STRAIGHT;**

**break;**

**case N:**

**tracing\_directions[i]=LEFT;**

**break;**

**case W:**

**tracing\_directions[i]=BACK;**

**break;**

**case S:**

**tracing\_directions[i]=RIGHT;**

**break;**

**}**

**break;**

**case N:**

**switch(go\_to)**

**{**

**case E:**

**tracing\_directions[i]=RIGHT;**

**break;**

**case N:**

**tracing\_directions[i]=STRAIGHT;**

**break;**

**case W:**

**tracing\_directions[i]=LEFT;**

**break;**

**case S:**

**tracing\_directions[i]=BACK;**

**break;**

**}**

**break;**

**case W:**

**switch(go\_to)**

**{**

**case E:**

**tracing\_directions[i]=BACK;**

**break;**

**case N:**

**tracing\_directions[i]=RIGHT;**

**break;**

**case W:**

**tracing\_directions[i]=STRAIGHT;**

**break;**

**case S:**

**tracing\_directions[i]=LEFT;**

**break;**

**}**

**break;**

**case S:**

**switch(go\_to)**

**{**

**case E:**

**tracing\_directions[i]=LEFT;**

**break;**

**case N:**

**tracing\_directions[i]=BACK;**

**break;**

**case W:**

**tracing\_directions[i]=RIGHT;**

**break;**

**case S:**

**tracing\_directions[i]=STRAIGHT;**

**break;**

**}**

**break;**

**}**

**temp\_direction=go\_to;**

**}**

**present\_direction=temp\_direction;**

**/\*Serial1.println("computed directions to trace back");**

**for(i=0;i<=indx;i++){**

**Serial1.print(tracing\_directions[i]);**

**Serial1.print("\t");**

**}\*/**

**return;**

**}**

**//fuction to back track the trajectory**

**bool back\_trace()**

**{**

**Serial1.println("Back tracing");**

**Store\_the\_map();**

**uint8\_t back\_tracking\_path[size\_of\_node\_array],back\_tracking\_directions[size\_of\_node\_array];**

**int i;**

**int j = 0;**

**int k = size\_of\_node\_array - 1; //define the node array size variable**

**int loop\_found = 0;**

**LeftEncoder.write(0);**

**RightEncoder.write(0);**

**while (k >= 0)**

**{**

**int explored = East[nodes\_track[k]-1] + North[nodes\_track[k]-1] + West[nodes\_track[k]-1] + South[nodes\_track[k]-1]; //how many ways are explored by the robot at a node**

**if (explored < 4) // if the explored ways are less than 4 break out of the while loop**

**{**

**back\_tracking\_path[j] = nodes\_track[k];**

**j++;**

**break;**

**}**

**for (i = 0; i < j; i++) //a loop to find for loops in the back tracking path and remove them**

**{**

**if (back\_tracking\_path[i] == nodes\_track[k])**

**{**

**loop\_found = 1;**

**j = i + 1;**

**break;**

**}**

**}**

**if (loop\_found==0 || j == 0)**

**{**

**back\_tracking\_path[j] = nodes\_track[k]; //assign the node if it is not in the back tracking path array(no loop found)**

**j++;**

**}**

**loop\_found = 0;**

**k--;**

**}**

**size\_of\_node\_array=k+1;**

**Serial1.print("back tracing path ");**

**for(i=0;i<j;i++){**

**Serial1.print(back\_tracking\_path[i]);**

**Serial1.print("\t");**

**}**

**Serial.println();**

**find\_directions(j,back\_tracking\_path,back\_tracking\_directions);**

**unsigned char move\_straight=0;**

**float extra\_distance=0;**

**for(i=0;i<j-1;i++)**

**{**

**turn(back\_tracking\_directions[i],move\_straight);**

**extra\_distance=follow\_line();**

**if(!move\_straight)**

**{**

**move\_straight=1;**

**}**

**}**

**unsigned char found\_left=0;**

**unsigned char found\_right=0;**

**Straight(extra\_distance+9.3,&found\_left,&found\_right);**

**Serial1.print("Indx ");**

**Serial1.println(k);**

**present\_node=nodes\_track[k+1];**

**if(prefer\_the\_direction(nodes\_track[k+1]))**

**{**

**BOTHMOTORS\_STOP//stop the motors**

**return true;**

**}**

**return false;**

**}**

**//function to calculate the distance between to nodes**

**uint16\_t distance\_between\_nodes(uint8\_t node1,uint8\_t node2)**

**{**

**if(Co\_X[node1-1]==Co\_X[node2-1])**

**{**

**if(Co\_Y[node1-1]>Co\_Y[node2-1])**

**{**

**return (Co\_Y[node1-1]-Co\_Y[node2-1]);**

**}**

**else**

**{**

**return (Co\_Y[node2-1]-Co\_Y[node1-1]);**

**}**

**}**

**else if (Co\_Y[node1-1]==Co\_Y[node2-1])**

**{**

**if(Co\_X[node1-1]>Co\_X[node2-1])**

**{**

**return (Co\_X[node1-1]-Co\_X[node2-1]);**

**}**

**else**

**{**

**return (Co\_X[node2-1]-Co\_X[node1-1]);**

**}**

**}**

**}**

**//function to store the map in the form of adjacent matrix list**

**void Store\_the\_map()**

**{**

**Serial1.println("Storing the map ");**

**int i;**

**for(i=0;i<size\_of\_node\_array-1;i++)**

**{**

**if(ADJ\_matrix[nodes\_track[i]-1][nodes\_track[i+1]-1]==0)**

**{**

**ADJ\_matrix[nodes\_track[i]-1][nodes\_track[i+1]-1]=distance\_between\_nodes(nodes\_track[i],nodes\_track[i+1]);**

**ADJ\_matrix[nodes\_track[i+1]-1][nodes\_track[i]-1]=ADJ\_matrix[nodes\_track[i]-1][nodes\_track[i+1]-1];**

**}**

**}**

**return;**

**}**

**//function to map the arena**

**void map\_the\_arena()**

**{**

**int i,j;**

**while (1)**

**{**

**check\_for\_node();**

**if(prefer\_the\_direction(present\_node))**

**{**

**if(back\_trace())**

**{**

**//Serial1.println("Completed mapping");**

**BOTHMOTORS\_STOP**

**return;**

**}**

**}**

**}**

**}**

**//function to**

**int dijkstra(uint8\_t \*computed\_path,int startnode,int endnode)**

**{**

**int cost[MAX][MAX],distance[MAX],pred[MAX],indx;**

**int visited[MAX],count,mindistance,nextnode,i,j;**

**uint8\_t temp\_path[no\_of\_nodes];**

**//pred[] stores the predecessor of each node**

**//count gives the number of nodes seen so far**

**//create the cost matrix**

**for(i=0;i<no\_of\_nodes;i++)**

**{**

**for(j=0;j<no\_of\_nodes;j++)**

**{**

**if(ADJ\_matrix[i][j]==0)**

**cost[i][j]=INF;**

**else**

**cost[i][j]=ADJ\_matrix[i][j];**

**}**

**}**

**//initialize pred[],distance[] and visited[]**

**for(i=0;i<no\_of\_nodes;i++)**

**{**

**distance[i]=cost[startnode][i];**

**pred[i]=startnode;**

**visited[i]=0;**

**}**

**distance[startnode]=0;**

**visited[startnode]=1;**

**count=1;**

**while(count<no\_of\_nodes-1)**

**{**

**mindistance=INF;**

**//nextnode gives the node at minimum distance**

**for(i=0;i<no\_of\_nodes;i++)**

**{**

**if(distance[i]<mindistance&&!visited[i])**

**{**

**mindistance=distance[i];**

**nextnode=i;**

**}**

**}**

**//check if a better path exists through nextnode**

**visited[nextnode]=1;**

**for(i=0;i<no\_of\_nodes;i++)**

**{**

**if(!visited[i])**

**{**

**if(mindistance+cost[nextnode][i]<distance[i])**

**{**

**distance[i]=mindistance+cost[nextnode][i];**

**pred[i]=nextnode;**

**}**

**}**

**}**

**count++;**

**}**

**//print the path and distance of each node**

**if(endnode!=startnode)**

**{**

**indx=0;**

**temp\_path[indx]=endnode;**

**j=endnode;**

**do**

**{**

**indx+=1;**

**j=pred[j];**

**temp\_path[indx]=j;**

**}while(j!=startnode);**

**}**

**for(i=0;i<=indx;i++)**

**{**

**computed\_path[indx-i]=temp\_path[i]+1;**

**}**

**return indx;**

**}**

**//function to plan the shortest path**

**void plan\_path(int start,int destination)**

**{**

**Serial1.print("Planning path from node ");**

**Serial1.print(start+1);**

**Serial1.print(" to ");**

**Serial1.println(destination+1);**

**int indx,i;**

**uint8\_t planned\_path[no\_of\_nodes];**

**uint8\_t planned\_directions[no\_of\_nodes];**

**indx=dijkstra(planned\_path,start,destination);**

**Serial1.println("Computed shortest path");**

**for(i=0;i<indx+1;i++)**

**{**

**Serial1.print(planned\_path[i]);**

**Serial1.print("\t");**

**}**

**Serial1.print("\n");**

**find\_directions(indx+1,planned\_path,planned\_directions);**

**/\*Serial1.print(" planned directions ");**

**for(i=0;i<indx;i++)**

**{**

**Serial1.print(planned\_directions[i]);**

**Serial1.print("\t");**

**}\*/**

**unsigned char move\_straight=0;**

**float extra\_distance=0;**

**for(i=0;i<indx;i++)**

**{**

**turn(planned\_directions[i],move\_straight);**

**extra\_distance=follow\_line();**

**if(!move\_straight)**

**{**

**move\_straight=1;**

**}**

**}**

**LEFTMOTOR\_FWRD**

**RIGHTMOTOR\_FWRD**

**motor\_speed(150,150);**

**delay(350);**

**BOTHMOTORS\_STOP**

**Serial1.println("moved to destination");**

**Serial1.println();**

**return;**

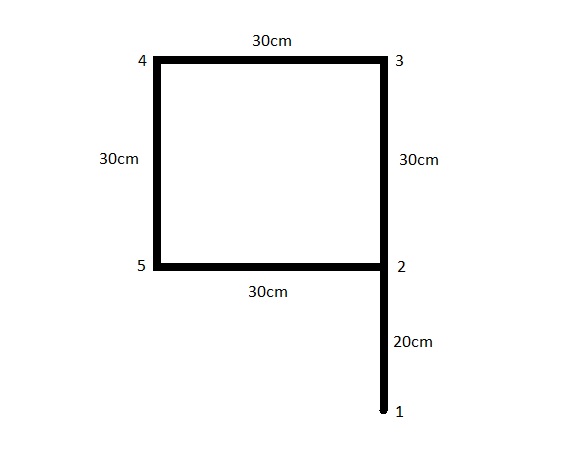
**}**

**CHAPTER 6: METHODOLGY OF TESTING**

**6.1 STRATEGY**

Different types of floor plans are made on an white chart with black tape as a guide path .The strategy is to test the optimised autonomous mapping on different floor plans and its ability to store the map. Distance covered during our proposed mapping technique is noted and compared with regular mapping technique. After the mapping is completed commands are given wirelessly to bot to move from one destination to another and test path planning algorithm.

**6.2 CASE 1**

****

**Fig6.1 Floor plan 1**

**6.2.1 Mapping**

In this case the plan consists of a simple loop having 5 nodes which is explored by the bot by covering a distance of 160cm .After that path planning is also executed

Adjacent matrix list:

After mapping the generated adjacent matrix list

**Table6.1 Adjacent matrix list of case1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** |
| **1** | 0 | 20 | 0 | 0 | 0 |
| **2** | 20 | 0 | 30 | 0 | 30 |
| **3** | 0 | 30 | 0 | 30 | 0 |
| **4** | 0 | 0 | 30 | 0 | 30 |
| **5** | 0 | 30 | 0 | 30 | 0 |

Comparision:

**Table6.2 comparision of algorithms for case1**

|  |  |  |
| --- | --- | --- |
|  | **Optimised flood fill mapping** | **Regular flood fill mapping** |
| **Distance travelled** | 160cm | 280cm |

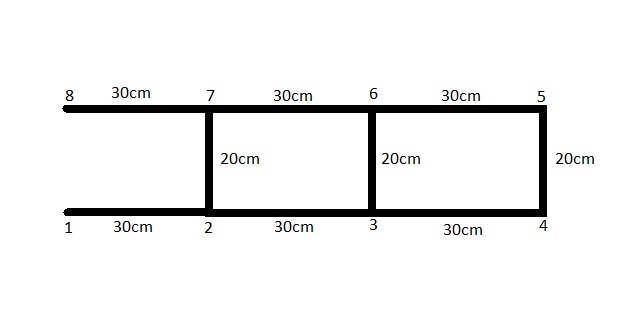
**6.2.2 Path Planning**

The map stored is now used for planning the shortest path between given nodes.The following commands are given to the AGV for path planning.

**Table6.3 Path planning for case1**

|  |  |  |  |
| --- | --- | --- | --- |
| **start** | **destination** | **Shortest Path computed** | **Distance covered** |
| 1 | 5 | 1-2-5 | 50cm |
| 5 | 2 | 5-2 | 30cm |
| 2 | 4 | 2-3-4 | 60cm |

**6.3 CASE 2**

****

**Fig6.2 Floor plan 2**

**6.3.1 Mapping**

In this case the plan consists of two loops having 8 nodes which is explored by the bot by covering a distance of 440cm .After that path planning is also executed

Adjacent matrix list:

After mapping the generated adjacent matrix list

**Table6.4 Adjacent matrix list of case2**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **1** | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 30 | 0 | 30 | 0 | 0 | 0 | 20 | 0 |
| **3** | 0 | 30 | 0 | 30 | 0 | 20 | 0 | 0 |
| **4** | 0 | 0 | 30 | 0 | 20 | 0 | 0 | 0 |
| **5** | 0 | 0 | 0 | 20 | 0 | 30 | 0 | 0 |
| **6** | 0 | 0 | 20 | 0 | 30 | 0 | 30 | 0 |
| **7** | 0 | 20 | 0 | 0 | 0 | 30 | 0 | 30 |
| **8** | 0 | 0 | 0 | 0 | 0 | 0 | 30 | 0 |

Comparision:

**Table6.5 comparision of algorithms for case2**

|  |  |  |
| --- | --- | --- |
|  | **Optimised flood fill mapping** | **Regular flood fill mapping** |
| **Distance travelled** | 480cm | 560cm |

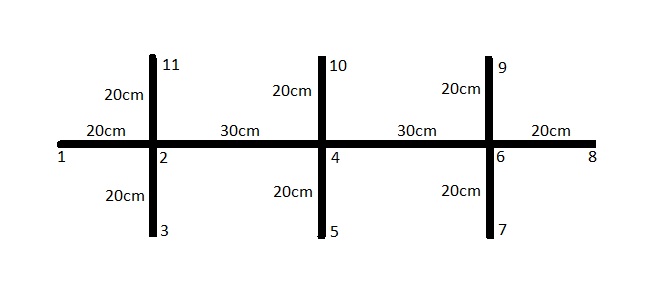
**6.3.2 Path Planning**

The map stored is now used for planning the shortest path between given nodes.The following commands are given to the AGV for path planning.

**Table6.6 Path planning for case2**

|  |  |  |  |
| --- | --- | --- | --- |
| **start** | **destination** | **Shortest Path computed** | **Distance covered** |
| 1 | 5 | 1-2-7-6-5 | 200cm |
| 5 | 2 | 5-4-3-2 | 80cm |
| 2 | 4 | 2-3-4 | 60cm |

**6.4 CASE 3**

****

**Fig6.3 Floor plan 3**

**6.4.1 Mapping**

In this case the plan consists of a simple loop having 11 nodes which is explored by the bot by covering a distance of 440cm .After that path planning is also executed

Adjacent matrix list:

After mapping the generated adjacent matrix list

**Table6.7 Adjacent matrix list of case3**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** |
| **1** | 0 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 20 | 0 | 25 | 30 | 0 | 0 | 0 | 0 | 0 | 0 | 25 |
| **3** | 0 | 25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **4** | 0 | 30 | 0 | 0 | 25 | 30 | 0 | 0 | 0 | 25 | 0 |
| **5** | 0 | 0 | 0 | 25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **6** | 0 | 0 | 0 | 30 | 0 | 0 | 20 | 25 | 25 | 0 | 0 |
| **7** | 0 | 0 | 0 | 0 | 0 | 20 | 0 | 0 | 0 | 0 | 0 |
| **8** | 0 | 0 | 0 | 0 | 0 | 25 | 0 | 0 | 0 | 0 | 0 |
| **9** | 0 | 0 | 0 | 0 | 0 | 25 | 0 | 0 | 0 | 0 | 0 |
| **10** | 0 | 0 | 0 | 25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **11** | 0 | 25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Comparision:

**Table6.8 comparision of algorithms for case3**

|  |  |  |
| --- | --- | --- |
|  | **Optimised flood fill mapping** | **Regular flood fill mapping** |
| **Distance travelled** | 440cm | 840cm |

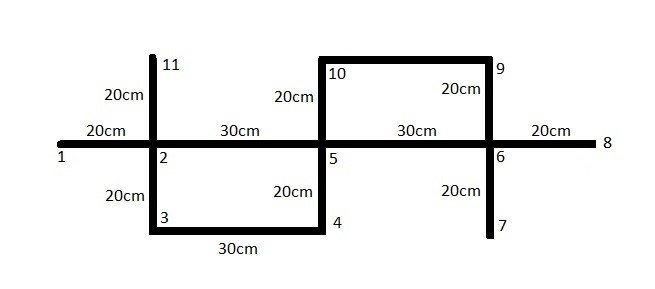
**6.4.2 Path Planning**

The map stored is now used for planning the shortest path between given nodes.The following commands are given to the AGV for path planning.

**Table6.9 Path planning for case2**

|  |  |  |  |
| --- | --- | --- | --- |
| **start** | **destination** | **Shortest Path computed** | **Distance covered** |
| 1 | 5 | 1-2-4-5 | 70cm |
| 5 | 2 | 5-4-2 | 50cm |
| 2 | 4 | 2-4 | 30cm |

**6.5 CASE 4**

****

**Fig6.4 Floor plan 4**

**6.5.1 Mapping**

In this case the plan consists of a simple loop having 11 nodes which is explored by the bot by covering a distance of 160cm .After that path planning is also executed

Adjacent matrix list:

After mapping the generated adjacent matrix list

**Table6.10 Adjacent matrix list of case4**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** |
| **1** | 0 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 20 | 0 | 20 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 20 |
| **3** | 0 | 20 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **4** | 0 | 0 | 30 | 0 | 20 | 0 | 0 | 0 | 0 | 0 | 0 |
| **5** | 0 | 30 | 0 | 20 | 0 | 30 | 0 | 0 | 0 | 20 | 0 |
| **6** | 0 | 0 | 0 | 0 | 30 | 0 | 25 | 25 | 20 | 0 | 0 |
| **7** | 0 | 0 | 0 | 0 | 0 | 25 | 0 | 0 | 0 | 0 | 0 |
| **8** | 0 | 0 | 0 | 0 | 0 | 25 | 0 | 0 | 0 | 0 | 0 |
| **9** | 0 | 0 | 0 | 0 | 0 | 20 | 0 | 0 | 0 | 30 | 0 |
| **10** | 0 | 0 | 0 | 0 | 20 | 0 | 0 | 0 | 30 | 0 | 0 |
| **11** | 0 | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Comparision:

**Table6.11 comparision of algorithms for case4**

|  |  |  |
| --- | --- | --- |
|  | **Optimised flood fill mapping** | **Regular flood fill mapping** |
| **Distance travelled** | 340cm | 600cm |

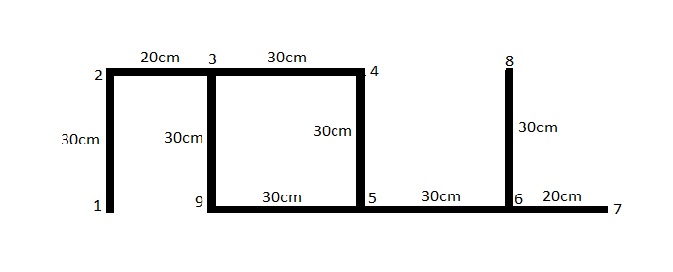
**6.5.2 Path Planning**

The map stored is now used for planning the shortest path between given nodes.The following commands are given to the AGV for path planning.

**Table6.12 Path planning for case4**

|  |  |  |  |
| --- | --- | --- | --- |
| **start** | **destination** | **Shortest Path computed** | **Distance covered** |
| 1 | 5 | 1-2-5 | 50cm |
| 5 | 2 | 5-2 | 30cm |
| 2 | 4 | 2-3-4 | 50cm |

**6.6 CASE 5**

****

**Fig6.5 Floor plan 5**

**6.6.1 Mapping**

In this case the plan consists of a simple loop having 9 nodes which is explored by the bot by covering a distance of 160cm .After that path planning is also executed

Adjacent matrix list:

After mapping the generated adjacent matrix list

**Table6.13 Adjacent matrix list of case5**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| **1** | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 30 | 0 | 20 | 0 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 20 | 0 | 30 | 0 | 0 | 0 | 0 | 30 |
| **4** | 0 | 0 | 30 | 0 | 30 | 0 | 0 | 0 | 0 |
| **5** | 0 | 0 | 0 | 30 | 0 | 30 | 0 | 0 | 30 |
| **6** | 0 | 0 | 0 | 0 | 30 | 0 | 20 | 30 | 0 |
| **7** | 0 | 0 | 0 | 0 | 0 | 20 | 0 | 0 | 0 |
| **8** | 0 | 0 | 0 | 0 | 0 | 30 | 0 | 0 | 0 |
| **9** | 0 | 0 | 30 | 0 | 30 | 0 | 0 | 0 | 0 |

Comparision:

**Table6.14 comparision of algorithms for case5**

|  |  |  |
| --- | --- | --- |
|  | **Optimised flood fill mapping** | **Regular flood fill mapping** |
| **Distance travelled** | 350cm | 650cm |

**6.6.2 Path Planning**

The map stored is now used for planning the shortest path between given nodes.The following commands are given to the AGV for path planning.

**Table6.15 Path planning for case5**

|  |  |  |  |
| --- | --- | --- | --- |
| **start** | **destination** | **Shortest Path computed** | **Distance covered** |
| 1 | 5 | 1-2-3-4-5 | 110cm |
| 5 | 2 | 5-4-3-2 | 80cm |
| 2 | 4 | 2-3-4 | 50cm |

**CHAPTER 7: RESULTS AND DISCUSSION**

**7.1 RESULT**

Our proposed mapping algorithm can map any type of floor plan autonomously by covering less distance compared to regular flood fill algorithm and motion planning is also implemented by using dijstrak’s algorithm by commanding the AGV wirelessly.

**Table7.1 Results of Mapping**

|  |  |  |
| --- | --- | --- |
|  | **Distance covered during regular flood fill (cm)** | **Distance covered during optimised flood fill (cm)** |
| **Case1** | 280 | 160 |
| **Case2** | 560 | 480 |
| **Case3** | 840 | 440 |
| **Case4** | 600 | 340 |
| **Case5** | 650 | 350 |

**7.2 DISCUSSIONS**

Challenges faced during our project are

* PID tuning
* Measuring the coordinates

**7.2.1 PID Tuning**

As the tuning is done manually multiple trails are attempted to find the derivative constant and integral constant

**7.2.2 Measuring The Coordinates**

To measure the coordinates distance is measured by using encoders but the measurement is not accurate because of the following reasons

* Difference between the motor speeds for particular PWM value
* Following trajectory is not exactly straight it is sinusoidal
* Angle taken by the AGV is not exact

this causing a centimeter error in measuring the distance .So a tolerence value of +/- 2.5cm is considered while measuring the distance

**CHAPTER 8: CONCLUSION AND FUTURE SCOPE**

**8.1 CONCLUSION**

As systems get larger and more sophisticated, the industry 4.0 approach will force AGV makers to consider decentralization as a critical component in meeting these demands. Through this project an attempt is made to design and development of AGV with autonomous mapping and motion planning. An optimized Flood Fill algorithm was applied to map the shop floor and stored in the form of adjacent matrix type format of graph data structure and while commanding wirelessly the robot plans the shortest path using dijkstra algorithm. The distance covered during mapping is optimised for the proposed work.

**8.2 FUTURE SCOPE**

* Identifying coordinates for a given node can be made more precise by fusing Encoders with other sensors like Inertial measurement unit like gyroscope, accelerometer.
* Interface for communicating with AGV can be made more user friendly.
* Swarm robotics system can be implemented so that the AGV’s can be fully automated without any need of central computer system.
* Microcontroller with more memory like STM32 might help for mapping large area floor plans.
* More sophisticated algorithm like A\* can be used for path planning where many other variables apart distance like traffic, time can be involved.

**REFERENCES**

1. Huijuan Wang, Yuan Yu and Yuan, Q. ―Application of Dijkstra Algorithm in robot path planning‖, Proceedings of the Second International Conference on Mechanical Automation and Control Engineering (MACE), pp. 1067-1069 (2011).

2. Xinmin Zhou, Tianwei Chen, Yimei Zhang. "Research on Intelligent AGV Control System", Chinese Automation Congress (CAC), 2018.

4. S. Sakib, A. Chowdhury, S. T. Ahamed and S. I. Hasan, "Maze solving algorithm for line following robot and derivation of linear path distance from nonlinear path," *16th Int'l Conf. Computer and Information Technology*, 2014, pp. 478-483.

5. V. Vuckovic and B. Arizanovic, "Heuristic-modified region filling algorithms for A-B connection searching problem," 2017 13th International Conference on Advanced Technologies, Systems and Services in Telecommunications (TELSIKS), 2017, pp. 399-402,

6. X. Zhou, T. Chen and Y. Zhang, "Research on Intelligent AGV Control System," 2018 Chinese Automation Congress (CAC), 2018, pp. 58-61,

7. T. Zheng, Y. Xu and D. Zheng, "AGV Path Planning based on Improved A-star Algorithm," *2019 IEEE 3rd Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, 2019, pp. 1534-1538,

8. S. Ranade and P. V. Manivannan, "Quadcopter Obstacle Avoidance and Path Planning Using Flood Fill Method," 2019 2nd International Conference on Intelligent Autonomous Systems (ICoIAS), 2019, pp. 166-170, doi: 10.1109/ICoIAS.2019.00036.

9. S. Ranade and P. V. Manivannan, "Quadcopter Obstacle Avoidance and Path Planning Using Flood Fill Method," 2019 2nd International Conference on Intelligent Autonomous Systems (ICoIAS), 2019, pp. 166-170, doi: 10.1109/ICoIAS.2019.00036.

10. B. Xu and D. Wang, "Magnetic Locating AGV Navigation Based on Kalman Filter and PID Control," 2018 Chinese Automation Congress (CAC), 2018, pp. 2509-2512, doi: 10.1109/CAC.2018.8623691.