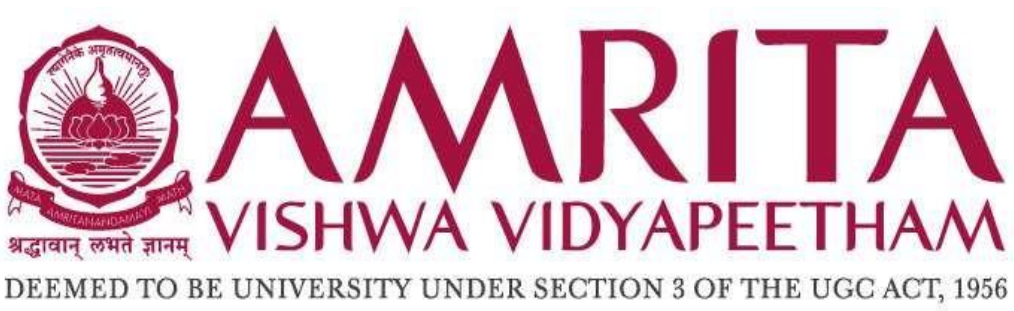
A Project

Submitted in partial fulfilment of the completion of the course 19CCE301 Internet of Things under the Faculty of Computer and Communication Engineering (CCE)

S. No.	Name	Roll No.
1	Akkineni Nithin Chowdary	CB.EN.U4CCE20002
2	Chilaka Sohith Reddy	CB.EN.U4CCE20012
3	Revanth Damisetty	CB.EN.U4CCE20013
4	Siva Dhanush Kosuri	CB.EN.U4CCE20057



AMRITA VISHWA VIDYAPEETHAM

COIMBATORE CAMPUS (INDIA)

JANUARY 2023

DECLARATION

We hereby declare that the project work entitled, "MazeRunner" submitted to the Department of Computer and Communication Engineering is a record of the original work done by us under the guidance of Mr Peeyush K. P., Faculty, Assistant Professor (Sr. Gr) at Amrita School of Engineering, Amrita Vishwa Vidyapeetham.

Signature of the Faculty

Introduction

Modern robotics technology aims to create self-navigating autonomous robots that can automate our daily tasks. This implies that the majority of research is focused on enhancing sensors and algorithms in order to create flexible and accurate robots.

The maze-solving robot, sometimes known as a micro mouse, is intended to discover a way without aid or assistance. To successfully overcome the maze, it must decipher the way on its own in a form of an autonomous robot. As a result, its logic differs greatly from that of a line-following robot, which follows a predefined path. These autonomous mobile robots have a wide range of applications, including material handling, warehouse management, pipe inspection, and bomb disposal.

In this report, will learn to build a simple Raspberry Pi-based maze-solving robot.

Abstract

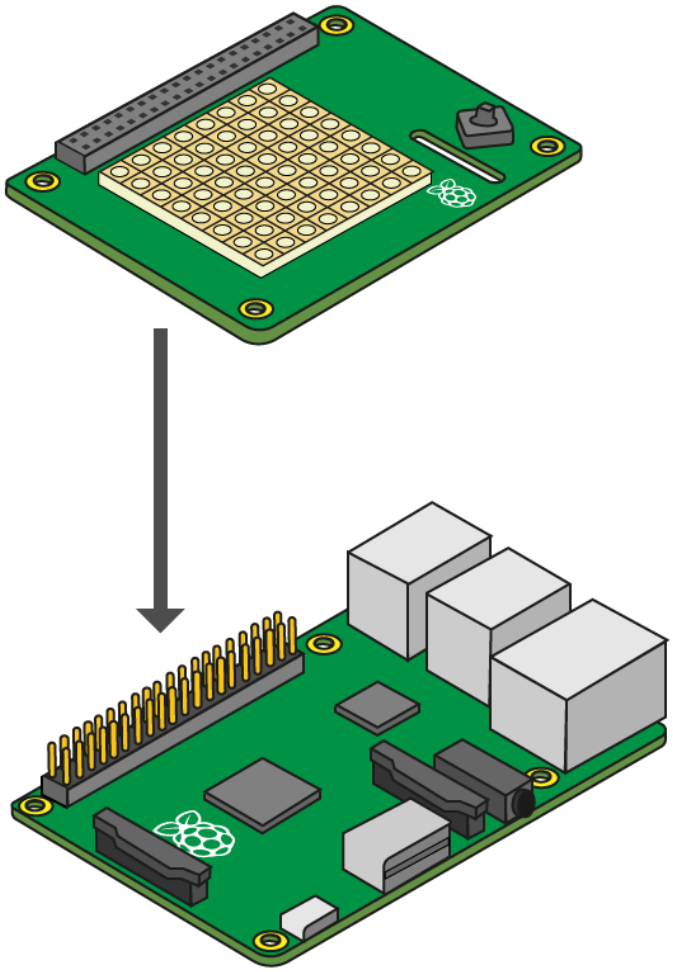
In this model developed, a maze is predefined with a single start point and single exit point. This maze consists of many possible ways of connecting to the exit. The entire model is basically divided into two subsequent parts - First determining the optimum path using Backtracking Algorithm, Secondly After getting the optimum path the robot moves in a determined way. This robot is built on an L239d motor driver integrated with a Raspberry pi like a client. Parallel one more Raspberry pi which consists of a sense-hat that simulates processes and acts as a server sends commands to the integrated robot using the default Bluetooth present in raspberry pi.

Components required

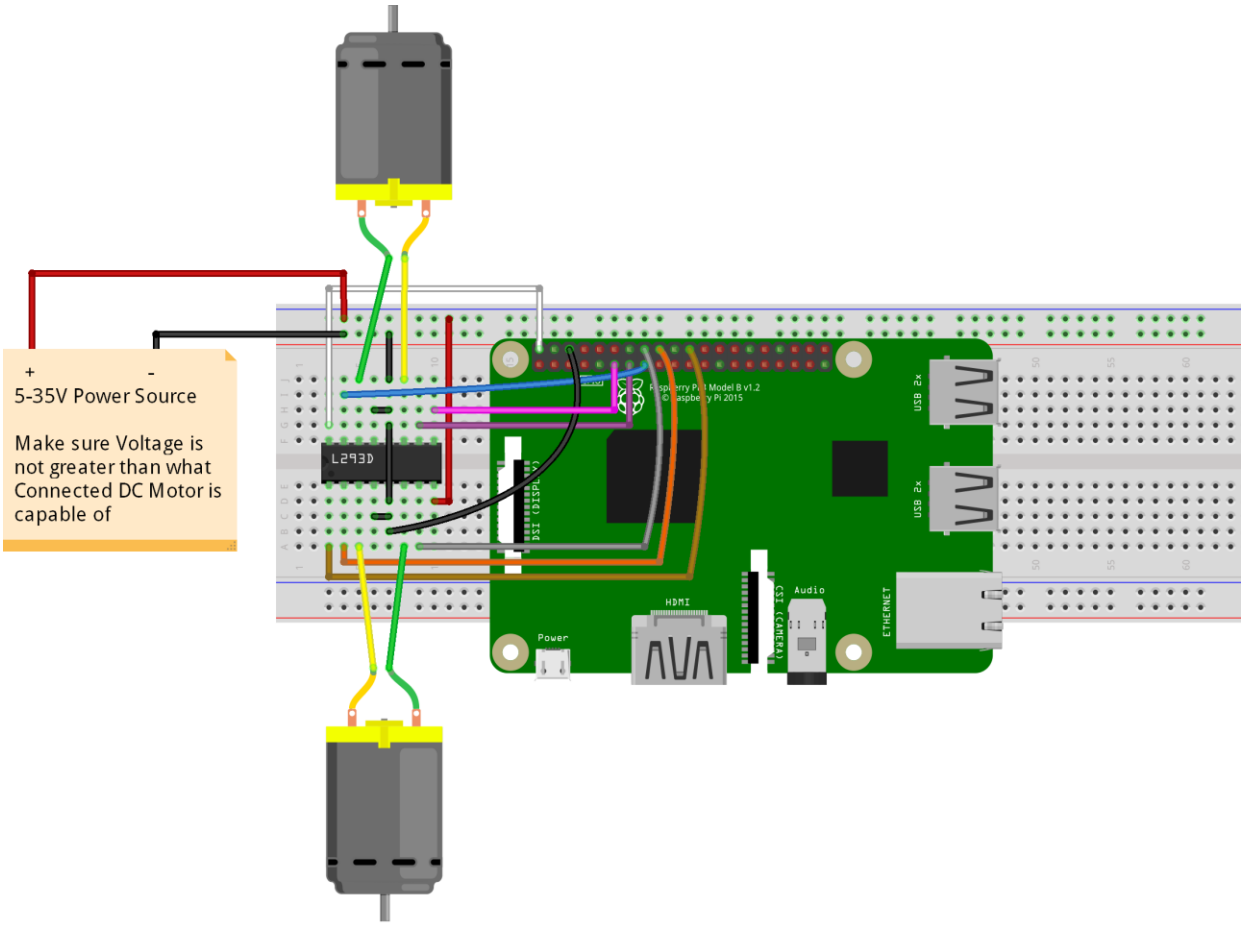
- Raspberry pi – 2 units.
- One Sense Hat module.
- A l239d motor Driver.
- Required female ports to connect l239d motor drivers.

Circuit Diagram

- Sense Hat installed Raspberry pi for a simulation – Act as a server,



- L239d motor Driver with Raspberry for movement of bot in the determined way – Act as a client.



Algorithm:

- Start
- Firstly, a maze input of an 8X8 matrix of binary information is taken from the user
- This Maze is printed to represent the unsolved maze
- The Maze is sent into a function to obtain the path traversal to cross the maze
- The robot considers the path obtained from the function as the path to travel and begins moving
- The Next algorithm checks the present direction of the robot and the next position of maze and determines the relative direction and acts in three ways:
    1. Forward (Moves forward)
    2. Right (rotates right and then moves forward)
    3. Left (rotates left and then moves forward)
- The robots travel in the above motions to reach the destination
- End

Working

- Backtracking Algorithm:
    - This is a set of algorithms that create the solution step by step.
    - In our implementation, the right box and the bottom boxes are checked for a possible path to move to the next path
    - If the path exists, it goes in that way and if it ends before reaching the destination, we retrace back a single step and search for alternative paths.
    - This Algorithm is much similar to the Dept-First-Search approach.

Future scope

- Creating a automated delivery drone which navigates itself to destination – this developed system can be a base for the delivery as the exit can be destinated house and all ways are predefined.
- Additional Proximity sensors would increase the possibility of reduction of error – by installing proximity sensors just detects the obstacles and the bot reroutes itself and hence reduces the error while finding an optimum path to the destination.
- The addition of picamera can analyze the road pattern which could help in formulation of automated maneuvering system – machine learning techniques like classifiers can be used to analyze the road patterns.

Code

- **Server code** – to simulate the path and sending commands to client.

```
from sense_hat import SenseHat

import time

import RPi.GPIO as GPIO

from time import sleep

from bluedot.btcomm import BluetoothServer
```

```
sense=SenseHat()
green=(0,255,0)
blue=(0,0,100)
orange=(255, 213, 128)
# Maze size
N = 8
# A utility function to print solution matrix sol
def printMaze( sol ):
    w=(150,150,150)
    b=(64,40,14)
    maze=[]
    for i in sol:
        for j in i:
            print(str(j) + " ", end ="")
            if j==1:
                maze.append(w)
            else:
                maze.append(b)
        print("\n")
    sense.set_pixels(maze)
# A utility function to check if x, y is valid
# index for N * N Maze

def isSafe( maze, x, y):
    if x >= 0 and x < N and y >= 0 and y < N and maze[x][y] == 1:
        return True
    return False

""" This function solves the Maze problem using Backtracking.
        It mainly uses solveMazeUtil() to solve the problem. It
```

```
        returns false if no path is possible, otherwise return
        true and prints the path in the form of 1s. Please note
        that there may be more than one solutions, this function
        prints one of the feasible solutions. """
def solveMaze( maze ):
    # Creating a 4 * 4 2-D list
    sol = [ [ 0 for j in range(N) ] for i in range(N) ]

    if solveMazeUtil(maze, 0, 0, sol) == False:
        print("Solution doesn't exist");
        return False
    return sol
# A recursive utility function to solve Maze problem
def solveMazeUtil(maze, x, y, sol):
    # if (x, y is goal) return True
    if x == N - 1 and y == N - 1:
        sol[x][y] = 1
        return True
    # Check if maze[x][y] is valid
    if isSafe(maze, x, y) == True:
        # mark x, y as part of solution path
        sol[x][y] = 1
        # Move forward in x direction
        if solveMazeUtil(maze, x + 1, y, sol) == True:
            sense.set_pixel(y,x,blue)
            return True
        # If moving in x direction doesn't give solution
        # then Move down in y direction
        if solveMazeUtil(maze, x, y + 1, sol) == True:
            sense.set_pixel(y,x,blue)
```

```
                return True
            # If none of the above movements work then
            # BACKTRACK: unmark x, y as part of solution path
            sol[x][y] = 0
            return False
def next(sol,x,y,dire):
    a=False
    if x>0 and a==False:
        if sol[x-1][y]==1:
            if dire!="down":
                x-=1
                dire="up"
                a=True
    if x<7 and a==False:
        if sol[x+1][y]==1 :
            if dire!="up":
                x+=1
                dire="down"
                a=True
    if y<7 and a==False:
        if sol[x][y+1]==1 :
            if dire!="left":
                a=True
                y+=1
                dire="right"
    if y>0 and a==False:
        if sol[x][y-1]:
            if dire!="right" :
                y-=1
                a=True
```

```
                    dire="left"
        print(x,y,dire)
        for i in range(8):
            for j in range(8):
                if maze[i][j] ==1:
                    sense.set_pixel(j,i,orange)
                if sol[i][j] ==1:
                    sense.set_pixel(j,i,blue)
        sense.set_pixel(y,x,green)
        time.sleep(1)
        return x,y,dire
# Driver program to test above function
if True:
        # Initialising the maze
        maze =      [[1, 0, 0, 0,0,0,0,0],
                     [1, 1, 0, 1,0,0,0,0],
                     [0, 1, 0, 0,0,0,0,0],
                     [1, 1, 1, 1,1,1,1,1],
                     [0, 1, 0, 1,0,0,0,1],
                     [0, 1, 0, 1,1,1,1,1],
                     [0, 1, 1, 1,0,1,0,1],
                     [1, 1, 1, 0,1,0,1,1]]
        printMaze(maze)
        time.sleep(5)
        print("\n")
        sol=solveMaze(maze)
        x,y=0,0
        dire="right"
        for i in range(8):
            for j in range(8):
```

```
            if maze[i][j] ==1:
                sense.set_pixel(j,i,orange)
sense.set_pixel(y,x,green)
time.sleep(2)
c = BluetoothServer(None)
while True:
    t=dire
    x,y,dire=next(sol,x,y,dire)
    if t==dire:
        c.send("forward")
    elif t=="up":
        if dire=="left":
            c.send("left")
        else:
            c.send("right")
    elif t=="right":
        if dire=="up":
            c.send("left")
        else:
            c.send("right")
    elif t=="left":
        if dire=="down":
            c.send("left")
        else:
            c.send("right")
    else:
        if dire=="right":
            c.send("left")
        else:
            c.send("right")
```

```
        sleep(2)
        if x==7 and y==7:
            break
time.sleep(5)
sense.clear()
```

- **Client code** – gets commands and directs bot.

```
#reciver side
import RPi.GPIO as GPIO
from time import sleep
from bluedot.btcomm import BluetoothClient
from signal import pause
GPIO.setmode(GPIO.BOARD)
GPIO.setwarnings(False)
set1=[19,21,23]#INPUT1,INPUT2,ENABLE OF LEFT MOTOR
set2=[11,13,15]#INPUT1,INPUT2,ENABLE OF RIGHT MOTOR

def forward():
    GPIO.output(set1[2],GPIO.HIGH)
    GPIO.output(set2[2],GPIO.HIGH)
    GPIO.output(set1[0],GPIO.HIGH)
    GPIO.output(set1[1],GPIO.LOW)
    GPIO.output(set2[0],GPIO.HIGH)
    GPIO.output(set2[1],GPIO.LOW)
    sleep(1)
    GPIO.output(set1[2],GPIO.LOW)
    GPIO.output(set2[2],GPIO.LOW)

def left():
    GPIO.output(set1[2],GPIO.HIGH)
```

```
        GPIO.output(set2[2],GPIO.HIGH)
        GPIO.output(set1[0],GPIO.HIGH)
        GPIO.output(set1[1],GPIO.LOW)
        GPIO.output(set2[0],GPIO.LOW)
        GPIO.output(set2[1],GPIO.HIGH)
        sleep(1)
        GPIO.output(set1[2],GPIO.LOW)
        GPIO.output(set2[2],GPIO.LOW)
        #move front
        forward()

def right():
        GPIO.output(set1[2],GPIO.HIGH)
        GPIO.output(set2[2],GPIO.HIGH)
        GPIO.output(set2[0],GPIO.HIGH)
        GPIO.output(set2[1],GPIO.LOW)
        GPIO.output(set1[0],GPIO.LOW)
        GPIO.output(set1[1],GPIO.HIGH)
        sleep(1)
        GPIO.output(set1[2],GPIO.LOW)
        GPIO.output(set2[2],GPIO.LOW)
        #move front
        forward()
for i in range(len(set1)):
        GPIO.setup(set1[i],GPIO.OUT)
        GPIO.setup(set2[i],GPIO.OUT)

def data_received(data):
        if data=="forward":
                forward()
```

```
            print("f")
        elif data == "left":
            left()
            print("l")
        elif data == "right":
            right()
            print("r")
    sleep(7.5)
    c = BluetoothClient("rncs", data_received)
    pause()
    GPIO.cleanup()
```

Output

- Red brown led – walls in the maze.
- Orange led – possible paths to the exit.
- Blue led – optimum path for the exit.
- Green led – indicates robot movement.