Task 1 Part 1

**Maze generation**

def maze():

mat = np.full((40, 40), 255)

for i in range(40):

for j in range(40):

pixel\_val\_probability = random.randrange(1, 100)

critical\_val = random.randrange(20, 30)

if pixel\_val\_probability <= critical\_val:

mat[i][j] = 0

mat = upsize(mat)

count = 0

grey = []

while count < 2:

i, j = random.randrange(0, 159), random.randrange(0, 159)

if (mat[i + 1, j] != 0 or mat[i - 1, j] != 0 or mat[i, j + 1] != 0 or mat[i, j - 1] != 0) and mat[

i + 1, j] != 127 and mat[i - 1, j] != 127 and mat[i, j + 1] != 127 and mat[i, j - 1] != 127:

mat[i, j] = 127

count += 1

grey.append((i, j))

return mat.astype(np.uint8), grey

This function creates a maze of shape (40,40) with each pixel having a probability 0.2-0.3 of being black in colour. This numpy array is then passed into the upsize function which upsizes this array by a factor of 4. Now 2 randomly generated values (i, j) and generated which represent the position of the pixel where the start and stop points are located. This pixel is then coloured grey.

def upsize(mat):

new = np.full((160, 160), 255)

for i in range(160):

for j in range(160):

x = (i - i % 4) // 4

y = (j - j % 4) // 4

new[i][j] = mat[x][y]

return new

This function takes the maze of shape (40,40) created using the numpy array as input and upsizes it by a factor of 4 and returns a numpy array of shape (160,160).

**Calling the Maze generation function and opening the text file for documentation**

# Opening/Creating a text file

f = open("Task 1 Documentation.txt", "w")

# Creating the maze

mat, critical = maze()

start = critical[0]

stop = critical[1]

cv2.namedWindow("Maze", cv2.WINDOW\_NORMAL)

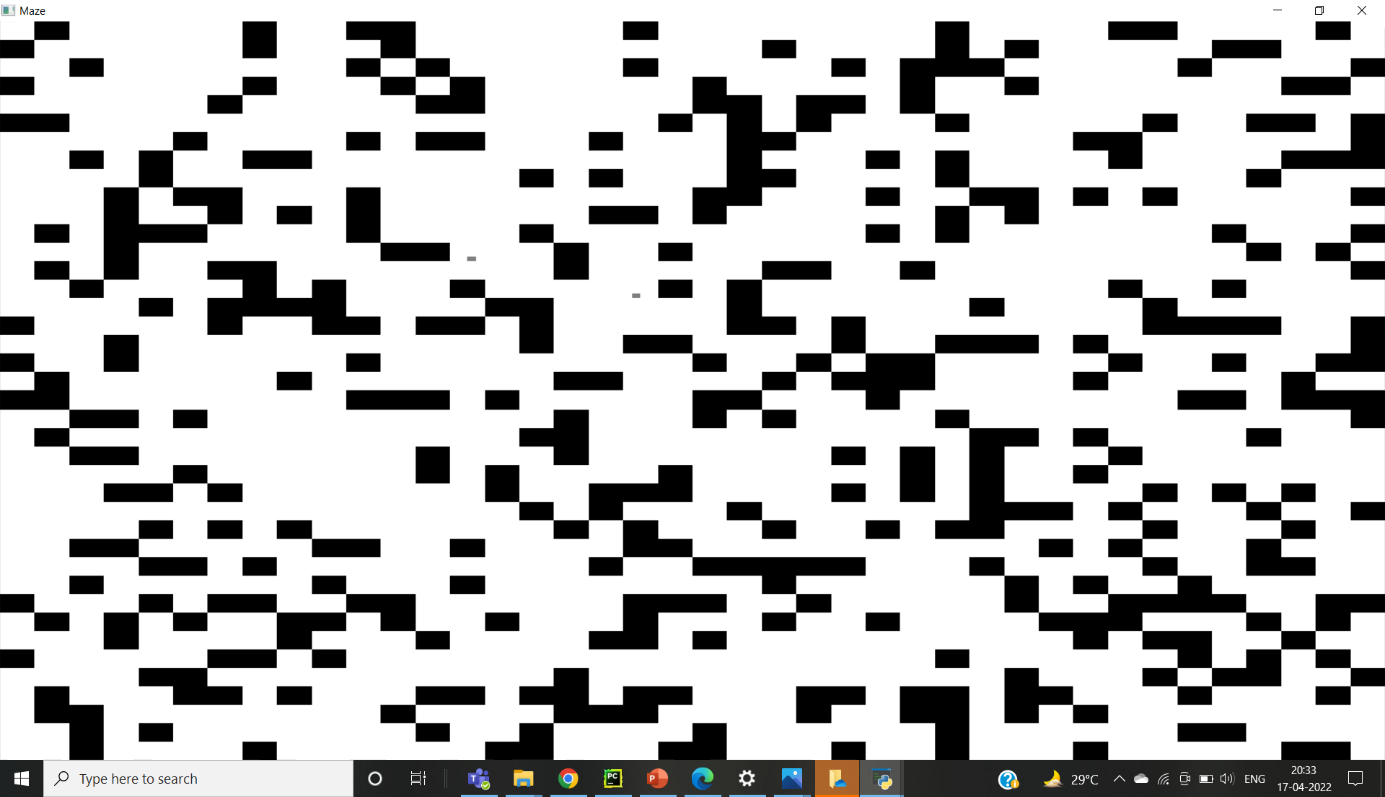
cv2.imshow("Maze", mat)

The first line creates a text file with the name “Task 1 Documentation” in which the documentation for this part of the project is stored.

The variable mat stores the maze generated using numpy array and critical contains a list with the elements as the start and stop pixels which are then stored in the start and stop variables.

The last 2 lines display the generated maze.

**Output Maze**



**Node Creation**

class Node:

def \_\_init\_\_(self, index, parent):

self.x = index[0]

self.y = index[1]

self.parent = parent

Here, a class Node is defined with its members x, y and parent. This data type will be used in path finding using the DFS and BFS algorithms.

**Path finding using BFS**

def bfs(mat, img, start):

q = deque()

q.append(start)

cv2.namedWindow("BFS", cv2.WINDOW\_NORMAL)

cv2.imshow("BFS", mat)

cv2.waitKey(1)

while len(q):

current = q.popleft()

i, j = current.x, current.y

if j + 1 < mat.shape[1]:

if mat[i][j + 1] != 0 and mat[i][j + 1] != 200:

if mat[i][j + 1] == 127 and (i != start.x) and (i != start.x):

break

mat[i][j + 1] = 200

n = Node((i, j + 1), current)

q.append(n)

cv2.imshow("BFS", mat)

cv2.waitKey(1)

if i + 1 < mat.shape[0]:

if mat[i + 1][j] != 0 and mat[i + 1][j] != 200:

if mat[i + 1][j] == 127 and (i != start.x) and (i != start.x):

break

mat[i + 1][j] = 200

n = Node((i + 1, j), current)

q.append(n)

cv2.imshow("BFS", mat)

cv2.waitKey(1)

if i >= 1:

if mat[i - 1][j] != 0 and mat[i - 1][j] != 200:

if mat[i - 1][j] == 127 and (i != start.x) and (i != start.x):

break

mat[i - 1][j] = 200

n = Node((i - 1, j), current)

q.append(n)

cv2.imshow("BFS", mat)

cv2.waitKey(1)

if j >= 1:

if mat[i][j - 1] != 0 and mat[i][j - 1] != 200:

if mat[i][j - 1] == 127 and (i != start.x) and (i != start.x):

break

mat[i][j - 1] = 200

n = Node((i, j - 1), current)

q.append(n)

cv2.imshow("BFS", mat)

cv2.waitKey(1)

dist = show\_path\_bfs(start, current, img)

if dist == 0:

print("Path Not Found")

exit()

return dist

This function finds the path using BFS algorithm. It appends the start node into a queue and then searches its neighbours for the end node. It then converts the neighbour into a node with its parent as the current node. This node is then marked as visited by changing the pixel value to 200 and appending this node to the queue. This process is then repeated for the neighbour till the stop node if finally reached.

Now, the start and end nodes are passed into the show\_path\_bfs functionto display the path and the path length returned from the show\_path\_bfs function is returned out of the BFS function.

**Displaying the path found using BFS**

def show\_path\_bfs(start, end, mat):

dist = 0

current = end

while current != start:

dist += 1

mat[current.x][current.y] = 50

current = current.parent

cv2.namedWindow("Path using BFS", cv2.WINDOW\_NORMAL)

cv2.imshow("Path using BFS", bfs\_path)

return dist

This function displays the path found using BFS algorithm. It takes the start and end nodes along with a copy of the original matrix on which the path is displayed and displays the path by changing the pixel value to 50. It generates the path by first changing the value of end node to 50 and then going to its parent node and changing its pixel value to 50. This process keeps on repeating till the start node is reached. The path length is calculated as the number of nodes traversed in this process. It then returns the path length out of the function.

**Calling the BFS Function**

# BFS Algorithm

img\_bfs = mat.copy()

bfs\_path = mat.copy()

start\_time = time.time()

start\_node = Node((start[0], start[1]), None)

dist\_bfs = bfs(img\_bfs, bfs\_path, start\_node)

time\_bfs = time.time() - start\_time

line = """BFS :

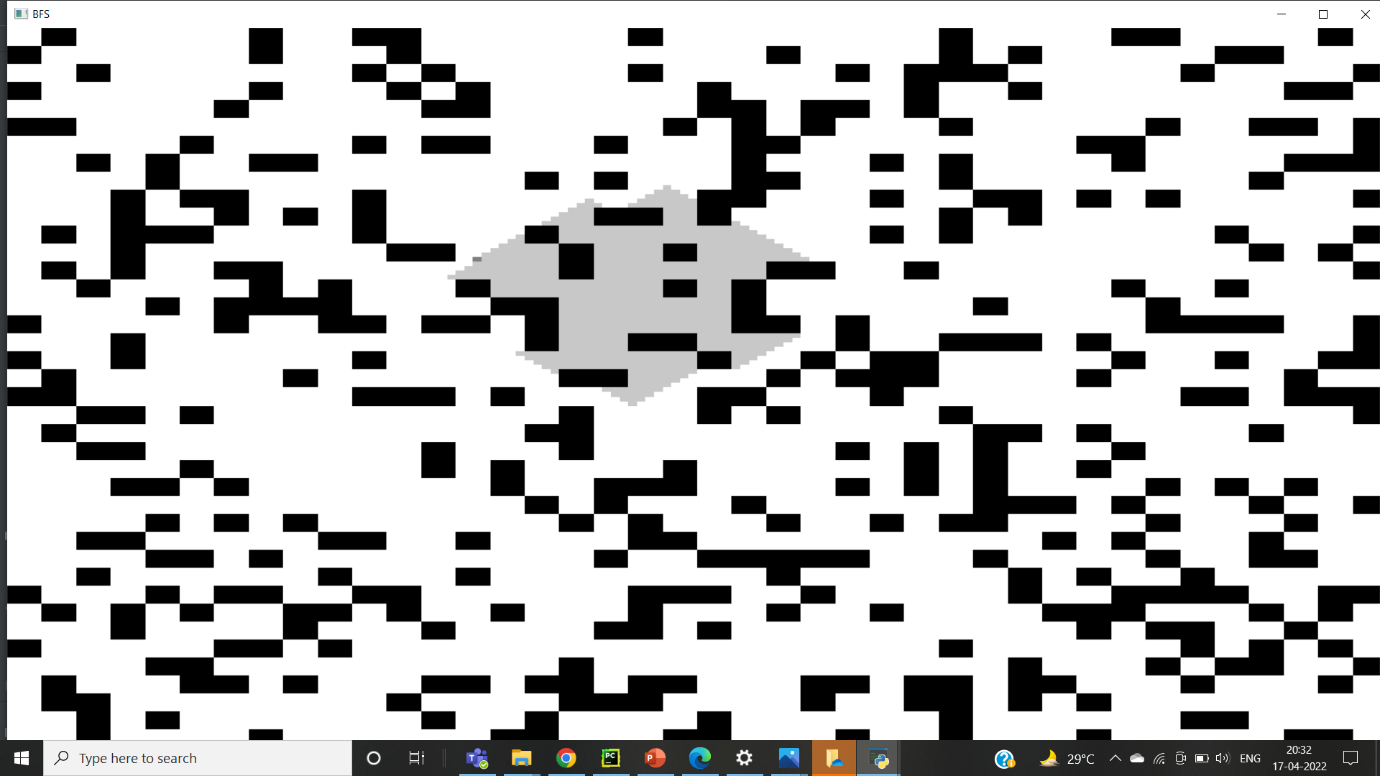
Time = %f sec Distance covered = %d pixels""" % (time\_bfs,dist\_bfs)

f.write(line)

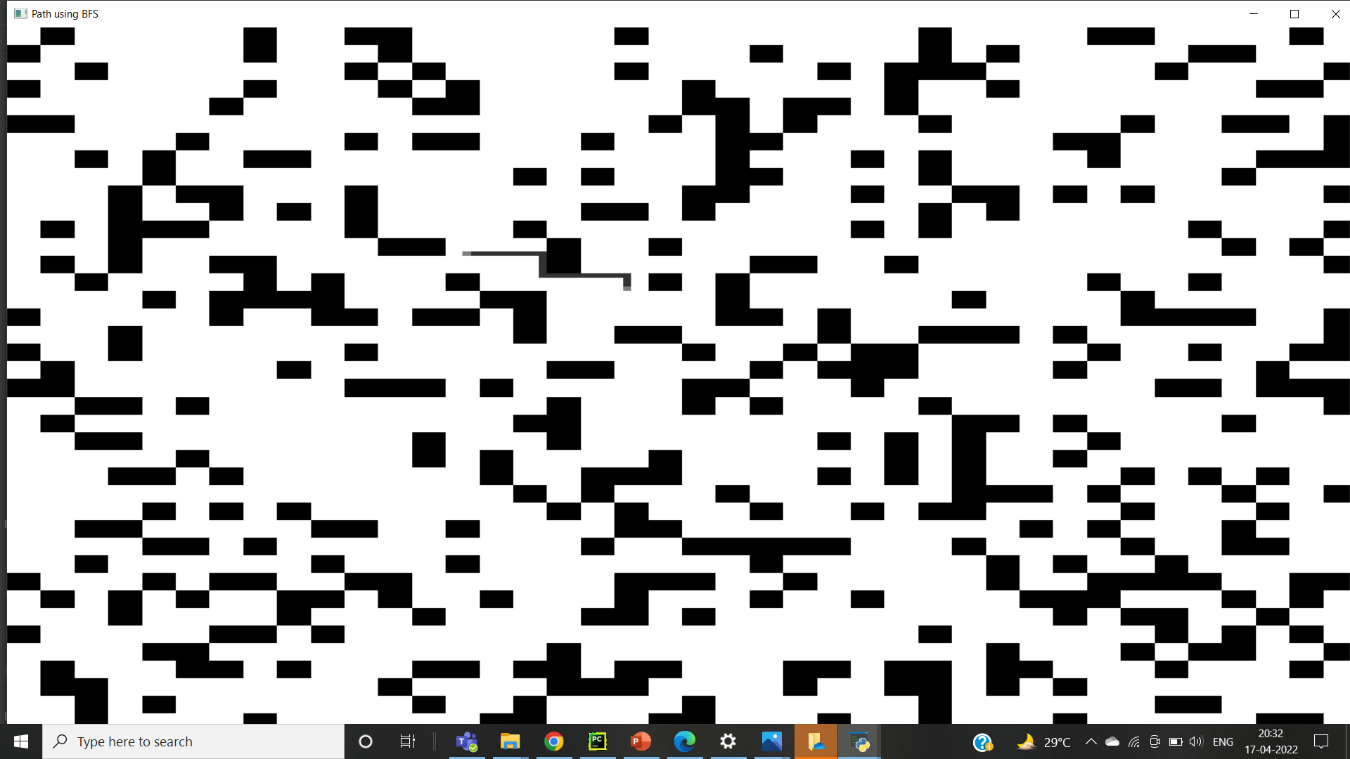
The first 2 lines create 2 copies img\_bfs and bfs\_path of the generated maze. img\_bfs will display the pixels visited while searching for the stop node and bfs\_path will display the final path. The time taken for this process is found by using the time.time() function in the time module before and after calling the bfs function and subtracting the 2 values. This time and the path length found are then stored in the documentation text file.

**Output Path using BFS**

Finding the path using BFS

****

Path using BFS

****

**Path finding using DFS**

def dfs(current, img):

global check

if check == 1:

return

i, j = current.x, current.y

path\_dfs.append(current)

cv2.namedWindow('DFS', cv2.WINDOW\_NORMAL)

cv2.imshow('DFS', img)

cv2.waitKey(1)

if i == stop[0] and j == stop[1]:

global dist\_dfs

check = 1

dist = show\_path(path\_dfs, path\_using\_dfs)

dist\_dfs = dist

return

if i >= 1:

if img[i - 1][j] != 0 and img[i - 1][j] != 200:

node = Node((i - 1, j), current)

img[i][j] = 200

dfs(node, img)

if j >= 1:

if img[i][j - 1] != 0 and img[i][j - 1] != 200:

node = Node((i, j - 1), current)

img[i][j] = 200

dfs(node, img)

if j + 1 < img.shape[1]:

if img[i][j + 1] != 0 and img[i][j + 1] != 200:

img[i][j] = 200

node = Node((i, j + 1), current)

dfs(node, img)

if i + 1 < img.shape[0]:

if img[i + 1][j] != 0 and img[i + 1][j] != 200:

img[i][j] = 200

node = Node((i + 1, j), current)

dfs(node, img)

This function finds the path using DFS algorithm. First the start node is passed into the function and it checks for the neighbours to find if the end node is there. If it is not present, it sets the pixel value of the neighbour to 200 to demarcate that it is a visited node and calls the function recursively by passing the neighbour into the function till the end node is reached. When the end node is reached, the value of the global variable check is set to 1 and the function is exited.

**Displaying the path found using DFS**

def show\_path(path,mat):

dist=0

cv2.namedWindow("Path using DFS",cv2.WINDOW\_NORMAL)

while len(path)>=1:

current=path[-1]

dist+=1

mat[current.x][current.y] = 50

path.pop()

cv2.imshow("Path using DFS",mat)

return dist

This function displays the path found using DFS algorithm. It takes the start node along with a copy of the original matrix on which the path is displayed and displays the path by changing the pixel value to 50. It generates the path by first changing the value of the last node in the stack to 50 and popping it out. It then goes to the now last member of the stack and changing its pixel value to 50. This process keeps on repeating till the start node is reached. The path length is calculated as the number of nodes traversed in this process. It then returns the path length out of the function.

**Calling the BFS Function**

#Dfs

start\_time = time.time()

start\_node = Node((start[0],start[1]),None)

img\_dfs = mat.copy()

path\_using\_dfs = mat.copy()

path\_dfs=deque()

dist\_dfs=-1

check=0

dfs(start\_node,img\_dfs)

time\_dfs = time.time() - start\_time

line = """DFS :

Time = %d sec\nDistance covered = %d"""%(time\_dfs,dist\_dfs)

f.write(line)

if dist\_dfs!=-1:

print("")

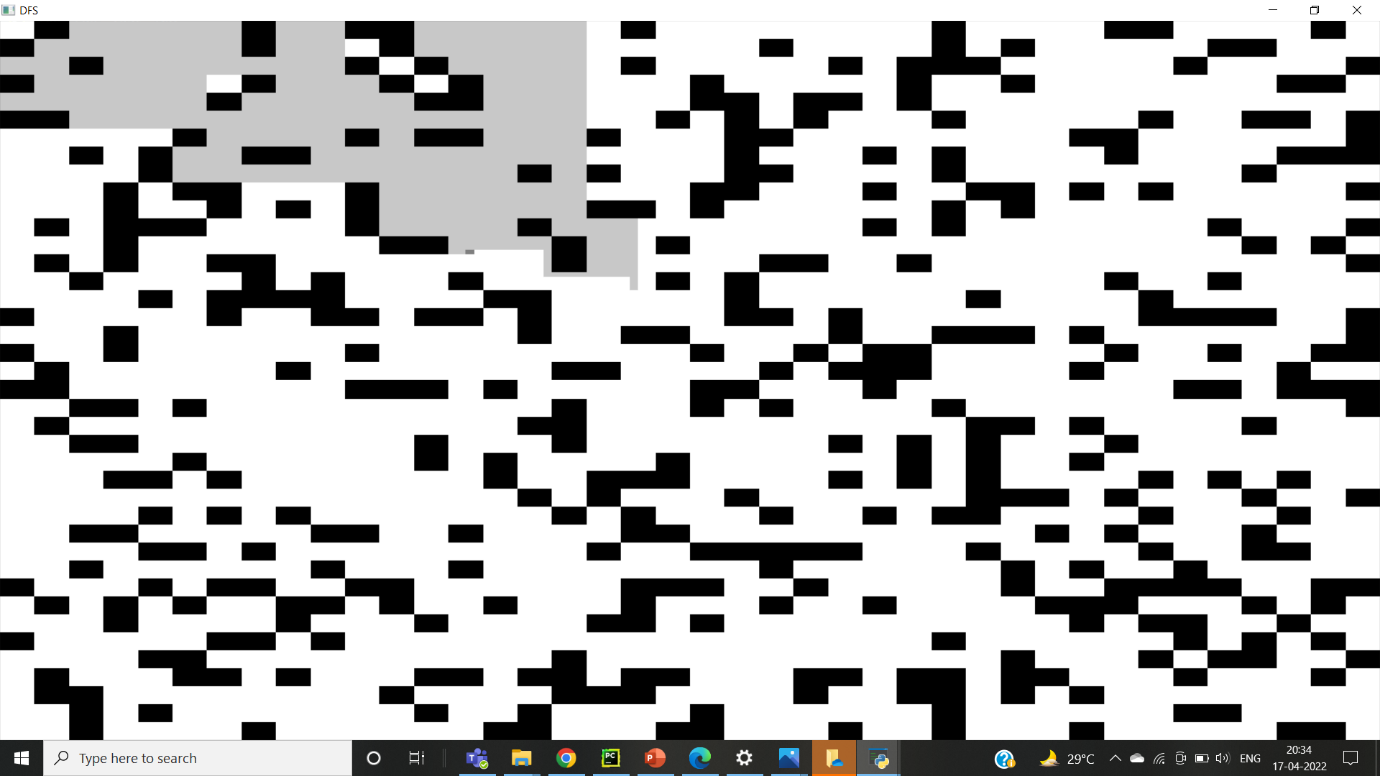
else:

print("No path found")

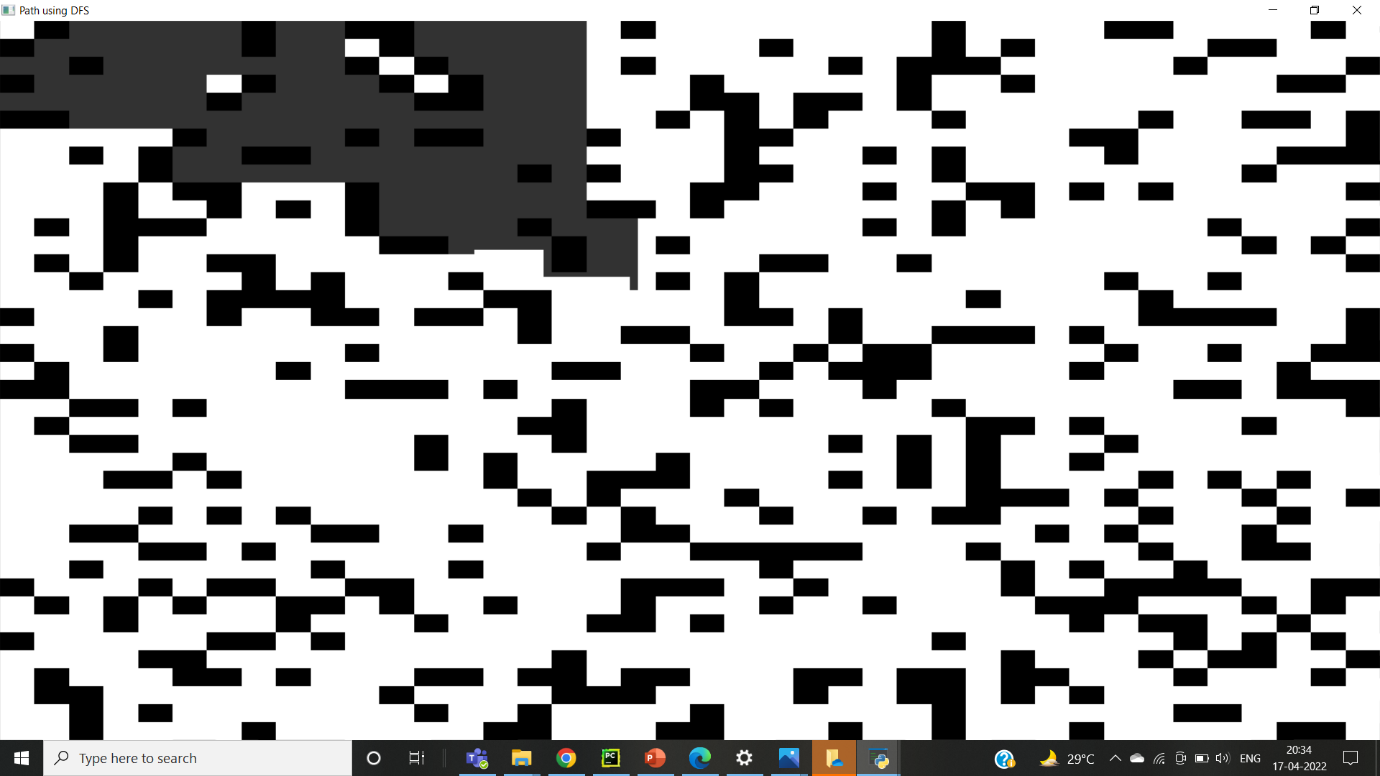
img\_node and path\_using\_dfs re 2 copies of the matrix in which the visited pixels and the path found are shown respectively. path\_dfs is a stack in which the pixels along the path are stored. The time taken for this process is found by using the time.time() function in the time module before and after calling the bfs function and subtracting the 2 values. This time and the path length found are then stored in the documentation text file.

**Output Path using BFS**

Finding the path using DFS



Path using DFS



**Task 1 part 2 and 3**

**BFS**

The code is the same as that of part 1. However, here we are dealing with an RGB image so the visited pixels are marked as img[i][j][0] = 127, img[i][j][1] = 127 and img[i][j][2] = 127 and the path is marked as mat[current.x][current.y] 0] = 50 mat[current.x][current.y] [1] = 50 and mat[current.x][current.y] [2] = 50.

**DFS**

The code is the same as that of part 1. However, here we are dealing with an RGB image so the visited pixels are marked as img[i][j][0] = 127, img[i][j][1] = 127 and img[i][j][2] = 127 and the path is marked as mat[current.x][current.y] 0] = 50 mat[current.x][current.y] [1] = 50 and mat[current.x][current.y] [2] = 50.

**Task 2**

**Arduino Code**

int lrpm=0;

int rrpm=0;

String LRPM;

String RRPM;

//Motor 1

int enA = 9;

int in1 = 8;

int in2 = 7;

//Motor 2

int enB = 3;

int in3 = 5;

int in4 = 4;

void setup() {

pinMode(enA,OUTPUT);

pinMode(enB,OUTPUT);

pinMode(in1,OUTPUT);

pinMode(in2,OUTPUT);

pinMode(in3,OUTPUT);

pinMode(in4,OUTPUT);

Serial.begin(9600);

}

void loop() {

if (Serial.available()>0)

{

LRPM=Serial.readStringUntil('\n');

RRPM=Serial.readStringUntil('\n');

lrpm=LRPM.toInt();

rrpm=RRPM.toInt();

digitalWrite(in1,HIGH);

digitalWrite(in2,LOW);

digitalWrite(in3,HIGH);

digitalWrite(in4,LOW);

lrpm = map(lrpm,0,200,0,255);

rrpm = map(rrpm,0,200,0,255);

analogWrite(enA,rrpm);

analogWrite(enB,lrpm);

delay(2000);

}

else

{

digitalWrite(in1,LOW);

digitalWrite(in2,LOW);

digitalWrite(in3,LOW);

digitalWrite(in4,LOW);

}

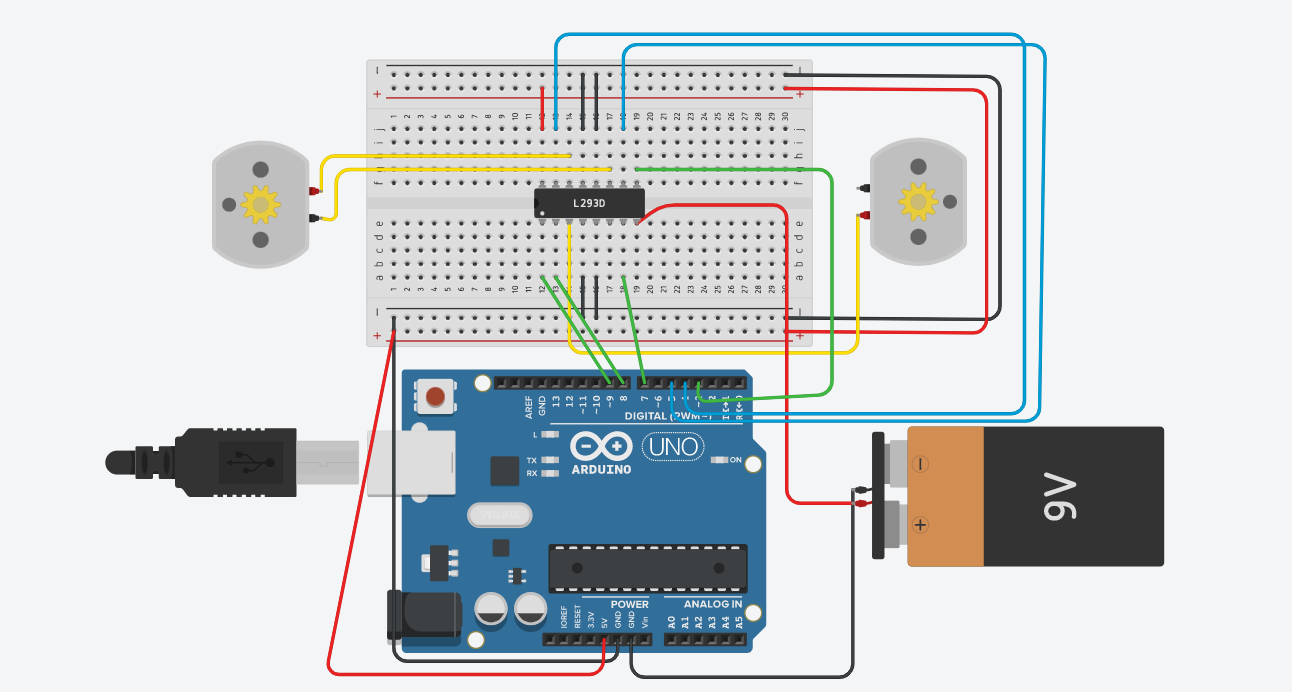
}

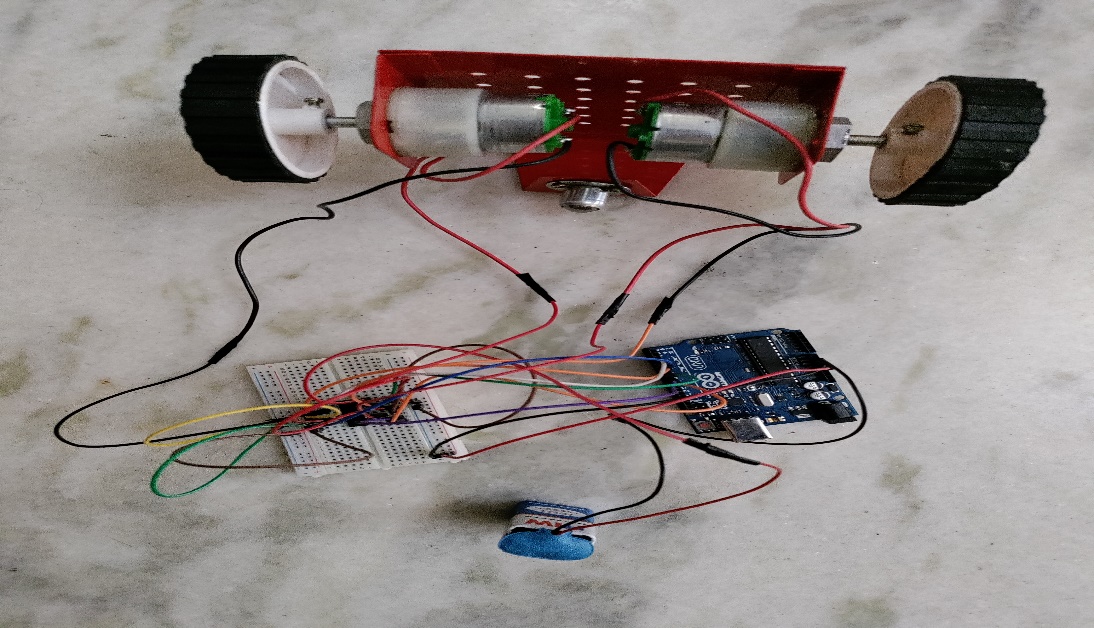
This code reads the values of the rpm of the left and right wheels of the car from the python code as a string and then converts them into int type. It then scales it into a range of 0-255 using the map function and passes these values to the enA and enB pins.

However, due to some reason, we were unable to send the output into the Arduino vehicle but have stored the output in a text file.

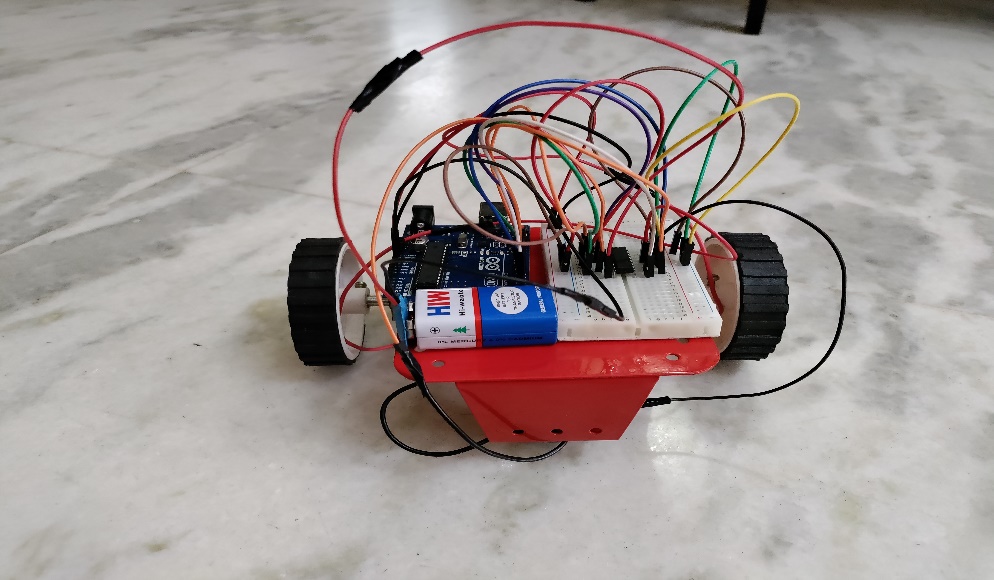
**Car model**

**Circuit**

****

****

**Car Model**

****