Task 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

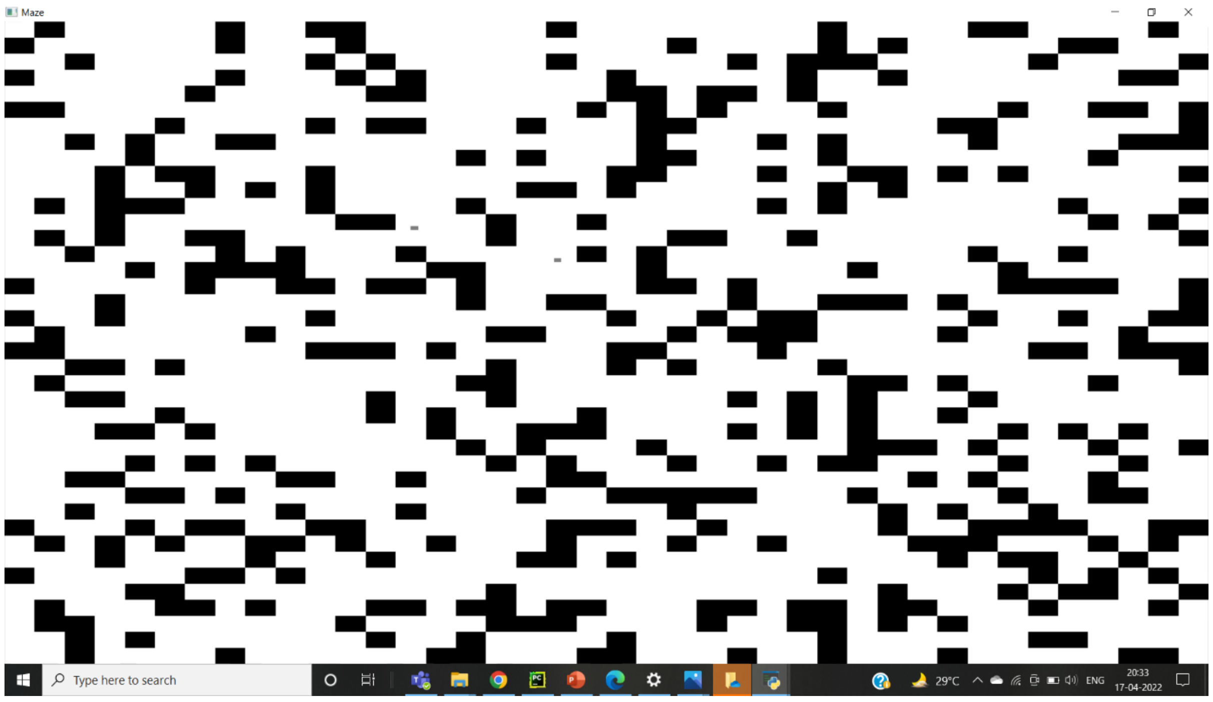
```python
# Opening/Creating a text file
f = open("Task 1 Documentation.txt", "w")

# 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.

Map Processing

```
n,m,l = img.shape

#processing

for i in range(n):
    for j in range(m):
        if(sum(img[i,j]) <= 254*3):
            img[i][j] = (0,0,0)
        else:
            img[i][j] = (255,255,255)

####
```

Here n and m denote the number of rows and columns of pixels in the image.
Looping over the pixels: if the sum of values of all the three fields (B, G, R) is less than or equal to (254*3) {or if the average of the colors in a pixel is less than or equal to 254, then it has been set to [0, 0, 0] (black). Otherwise, the pixel value is made [255,255,255] (white).

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

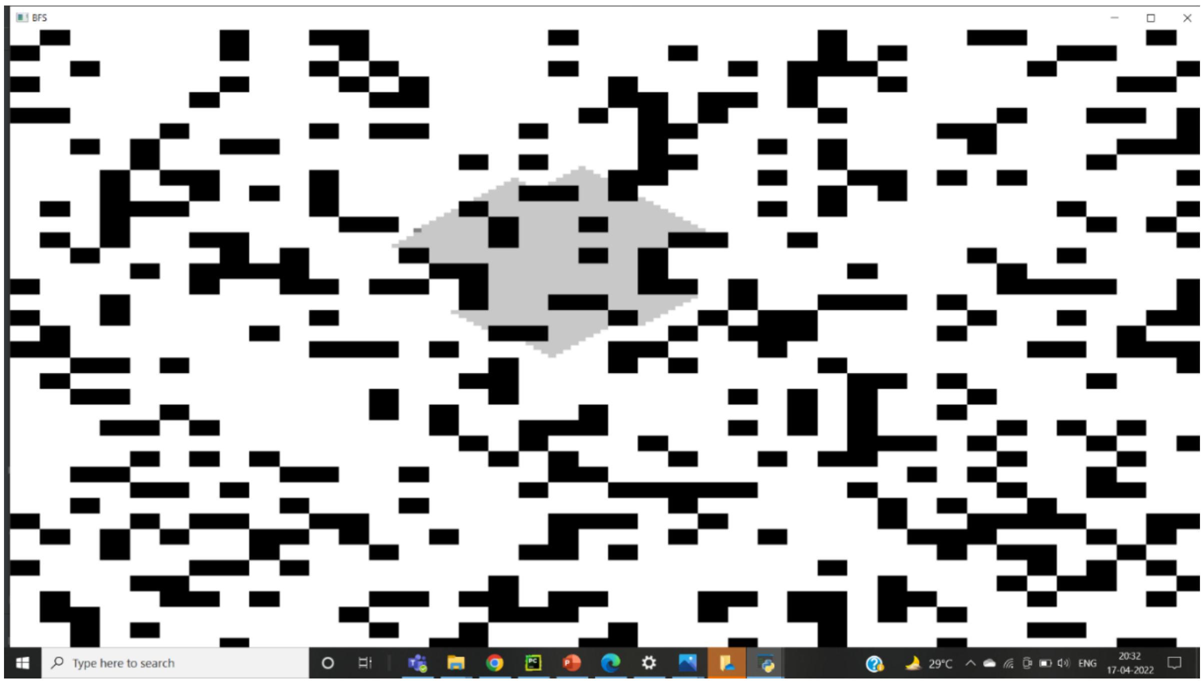
```
# BFS Algorithm
```

```
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)
```

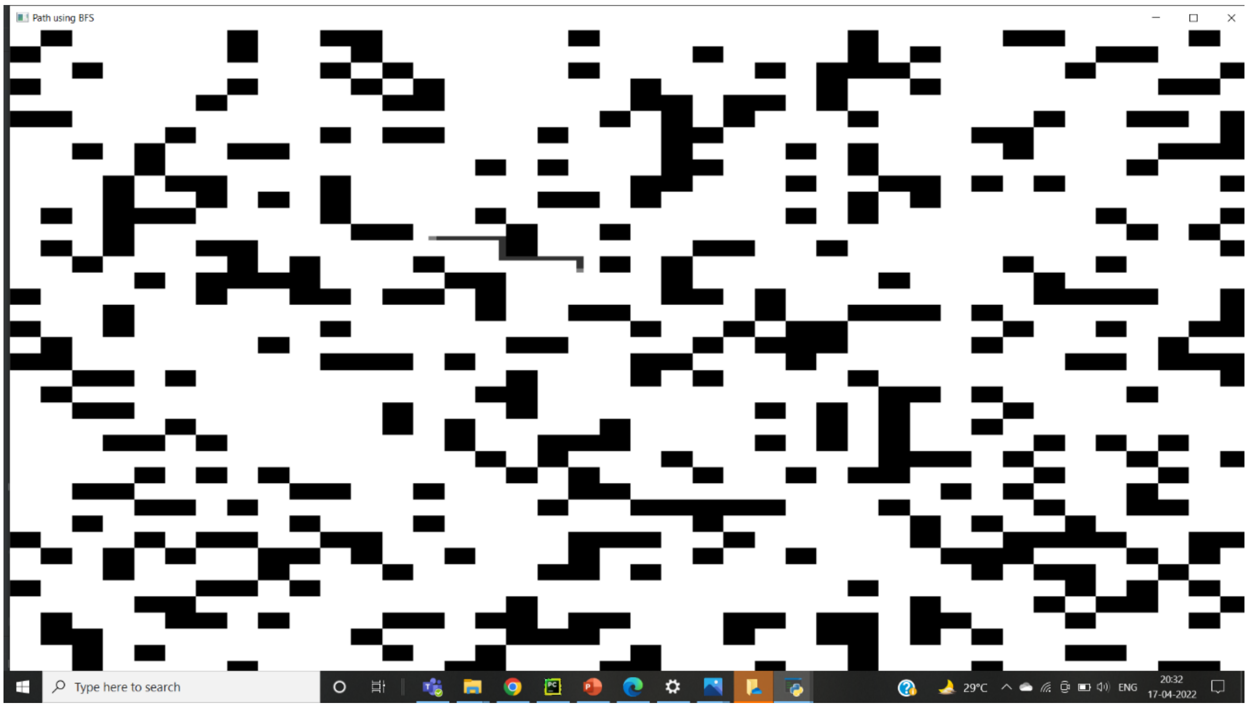
The first 2 lines create 2 copies img_bfs and bfs_path of the generated maze. img_bfs will display the pixels visited while searching for the stop node and bfs_path will display the final path. The time taken for this process is found by using the time.time() function in the time module before and after 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



BFS for map

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.

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.

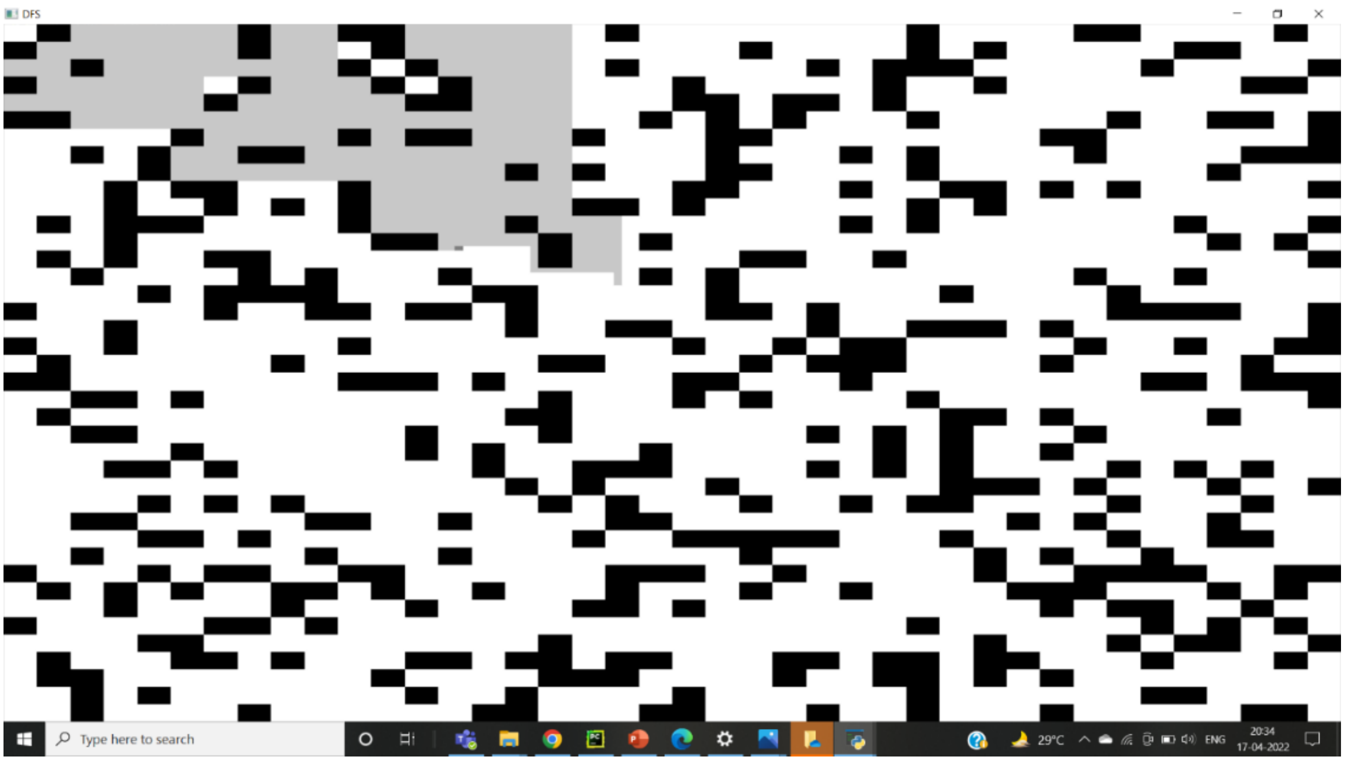
Calling the DFS Function

```
#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")
```

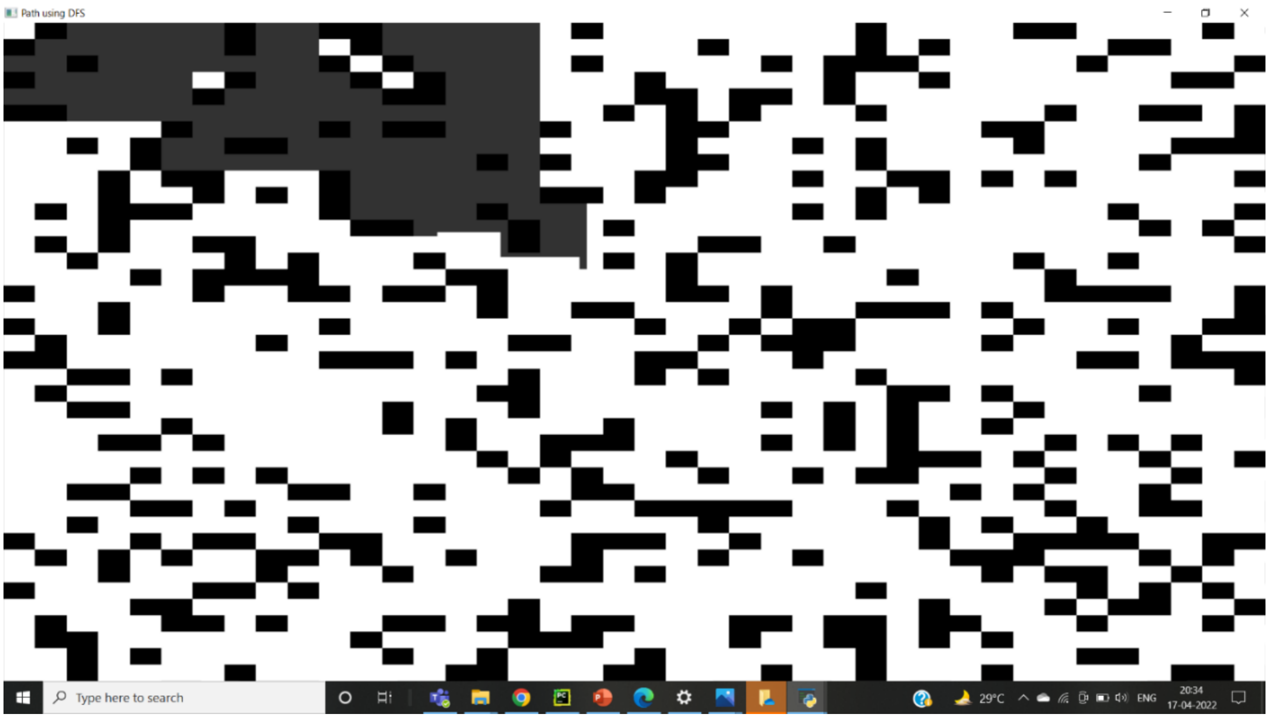
img_node and path_using_dfs re 2 copies of the matrix in which the visited pixels and the path found are shown respectively. path_dfs is a stack in which the pixels along the path are stored. The time taken for this process is found by using the time.time() function in the time module before and after 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



DFS for map

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.

Dijsktra

```
closed_list = []
queue = []
min_distance = np.full((n,m),np.inf)
visited = np.full((n,m), False)
parent = np.full((n,m), None)
```

These are a few declarations used.
closed_list stores the traversed nodes.
min_distance is a 2D array to store the minimum distance between a point and the starting point.
visited is a 2D array to denote whether a node has been visited or not.
parent is a 2D array storing the parent of each node

```
dijsktra(img_djiksktra,start,stop)
```

Calling the dijsktra function

```
def dijsktra(img, start, stop):

    global parent
    queue.append(start)
    min_distance[start[0],start[1]] = 0
    visited[start[0],start[1]] = True
    parent[start[0],start[1]] = None
    node = None
```

The start node is being appended to the queue, min_distance of start is updated to 0 and that node is marked visited.

```
while(len(queue) != 0):

        cv2.imshow("Path Planning",img)
        cv2.waitKey(1)

        node = queue.pop(0)
        if(node == stop):
            break
        (i,j) = node
        for i1 in range(i-1,i+2):
            for j1 in range(j-1,j+2):
                if(np.array_equal(img[i1,j1], [255,255,255]) == True or
(i1,j1) == stop) and (i1,j1) not in closed_list:

                    dist = math.sqrt((i1-i)**2 + (j1-j)**2)

                    if(visited[i1,j1] == True):
                        if(dist + min_distance[i,j] <
min_distance[i1,j1]):
```

```
                            min_distance[i1,j1] = dist +
min_distance[i,j]
                            parent[i1,j1] = node

                    else:
                        queue.append((i1,j1))
                        min_distance[i1,j1] = min_distance[i,j] +
dist
                        parent[i1,j1] = node
                        visited[i1,j1] = True
                        if(np.array_equal(img[i1,j1], [255,255,255])):
                            img[i1,j1] = yellow
```

Repetitively doing steps while the length of queue doesn't become 0.
The first element of the queue is popped and stored in **node**.
Immediately break out of the loop if the popped node is the end node.
Inspecting all the neighbors of the current node,

- If the node is white in color(i.e not visited) or if it is the stop node and it is not traversed, then the distance between the node and neighbor is stored in **dist** .
    - If that node is not visited, the min_distance and its parent is updated. It is marked visited and colored yellow.
    - Otherwise, a comparison is made between the previously stored min_distance. And if and only if the dist < min_distance , the min_distance is updated to dist and the parent is updated to **node**.
- Otherwise, continue to the next iteration.

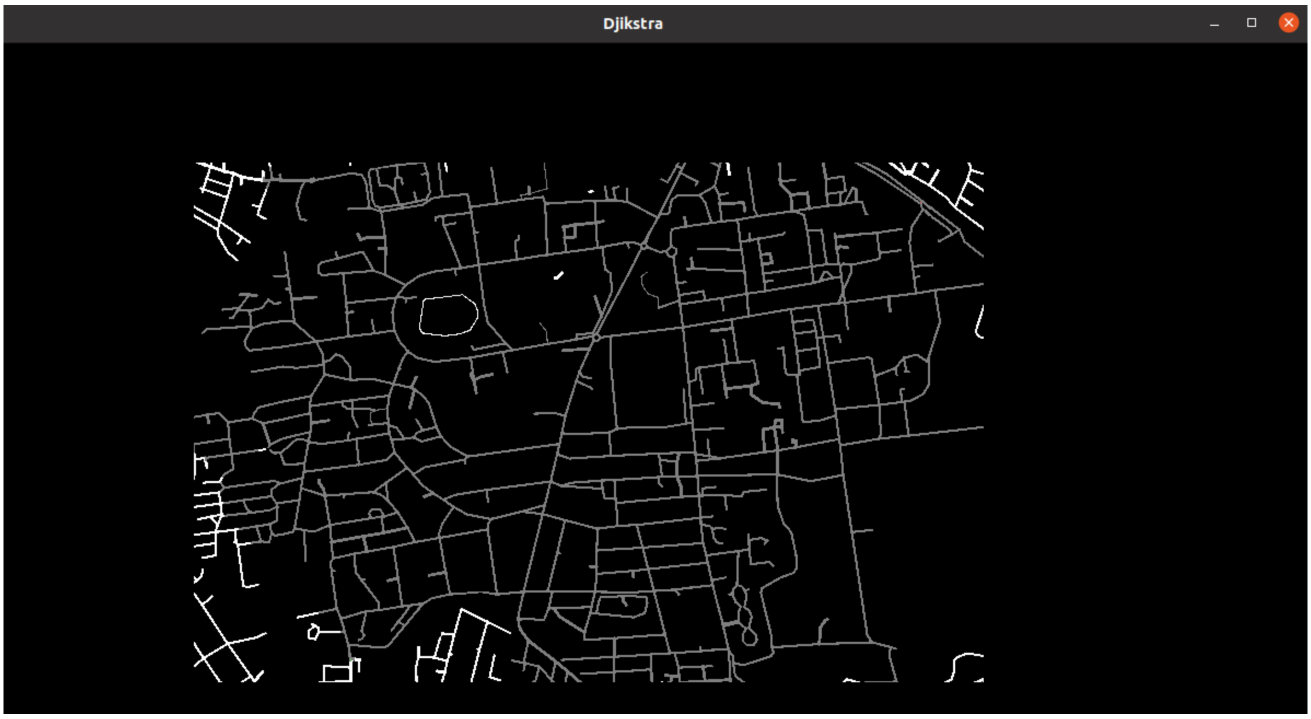
Path Tracing

```
stack = []

node = stop

while(node != None):

    stack.append(node)
    node = parent[node[0]][node[1]]

while(len(stack) != 0):

    x = stack.pop()
    for i in range(-1,2):
        for j in range(-1,2):
            img_djiksktra_path[i + x[0], j + x[1]] = (255,0,127)

    cv2.imshow("Path",img_djiksktra_path)
    cv2.waitKey(2)
```

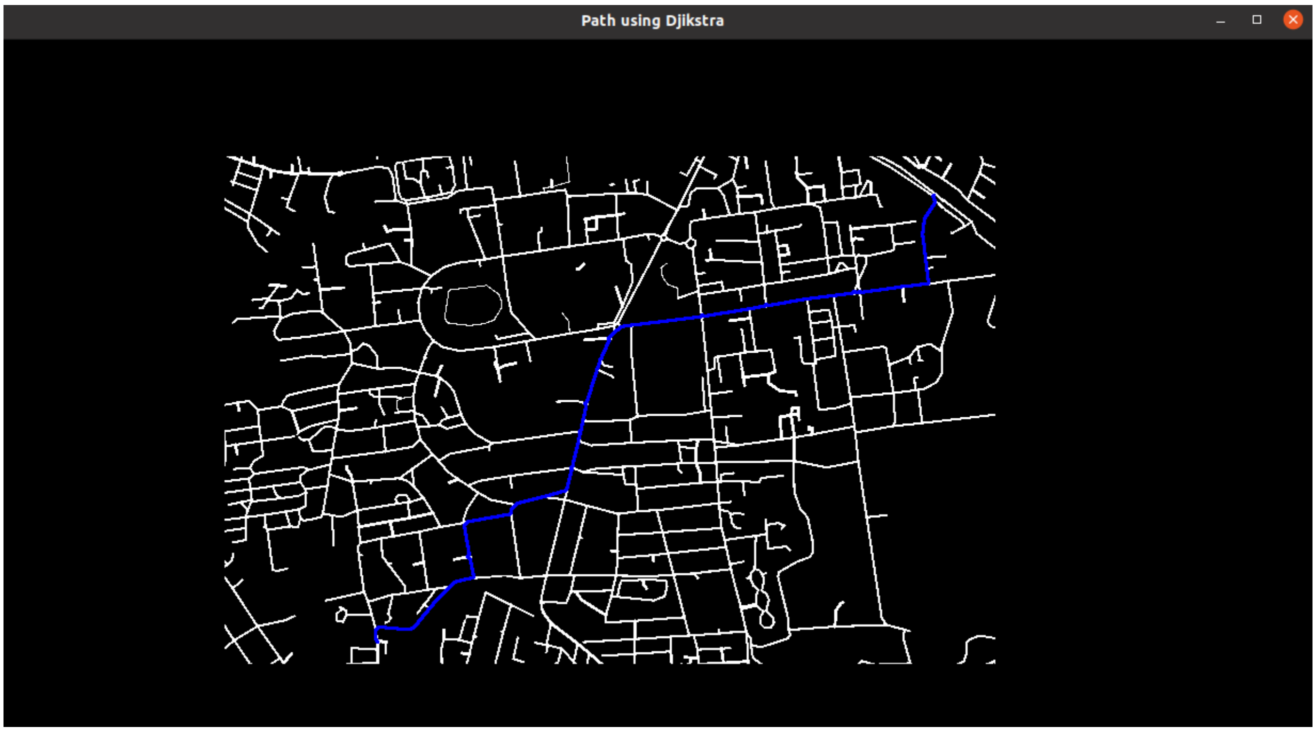
while the node is not equal to **None** [the parent of the start node is None]
    The node is appended into the stack.
Then, until the stack is empty, elements are popped out from the rear repeatedly, and these nodes are marked [colored violet].

```
for i in range(-1,2):
        for j in range(-1,2):
            img_djiksktra_path[i + x[0], j + x[1]] = (255,0,127)
```

This is done to broaden the line tracing shortest path.



Path traversal using Dijkstra



Path found using Dijkstra

Astar

```
def locked(img, p, i):
    if (i[0] == -1 & i[1] == -1):
        if (img[p[0] + i[0]][p[1]] !=0 or img[p[0]][p[1] + i[1]] !=0):
            return False
    elif (i[0] == 1 & i[1] == 1):
        if (img[p[0] + i[0]][p[1]] !=0 or img[p[0]][p[1] + i[1]] !=0):
            return False
    elif (i[0] == -1 & i[1] == 1):
        if (img[p[0] + i[0]][p[1]] !=0 or img[p[0]][p[1] + i[1]] !=0):
            return False
    elif (i[0] == 1 & i[1] == -1) :
        if (img[p[0] + i[0]][p[1]] !=0 or img[p[0]][p[1] + i[1]] !=0):
            return False

    if i == (-1,-1) or i == (1,1) or i == (1,-1) or i == (-1,1):
        return True

    return False
```

The above function which takes the image , the dequeued pixel and the ensures that the path traversal algorithm for Astar, while covering diagonals, do not leak through spaces where the diagonal is flanked by two obstacles. Basically it returns True if the open pixel (node) is a diagonal flanked by obstacles.

```
def inv(img, n, m):
    if n>(w-1) or n<0 or m>(t-1) or m<0:
        return True
    elif img[n][m] == 0 :
        return True
    else:
        return False
```

The above function takes the image array and the y and x coordinates of the system and returns if the object is within bounds of the image.

```
def dist(a, b):
    return ((a[0] - b[0])**2 + (a[1] - b[1])**2)**0.5
```

The above function gives the Euclidean distance between the two pixels a and b

```
def Astar(img, start, end):
    open = {}
    par = np.full((t, w, 2), -1)
    h = np.full((t, w), np.inf)
    g = np.full((t, w), np.inf)
    n = [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1,
1)]
    p = []
    g[start[0]][start[1]] = 0
    h[start[0]][start[1]] = 0
    open[start] = start
```

The above is the initial part of function Astar, which takes the image array, start and stop points as input. Here we have declared a dictionary open to store the open pixels under consideration, par array to store the parent(previous pixel) of the given pixel, h to store distance from the last pixel and g to store the distance from the origin. N stores the directions in which traversal can take place. The g and h values for start position are set to 0 and start is added to open.

```
while (len(open) > 0):
    midi = np.inf
    p = None
    for v in open:
        if (g[v[0]][v[1]] + h[v[0]][v[1]]) < midi:
            midi = g[v[0]][v[1]] + h[v[0]][v[1]]
            p = v
    c = midi
    open.pop(p)
```

Here we start the evaluation of the image numpy array, continuing as long as the open list is not empty. The minimum distance is set to infinity and the chosen pixel to none. Now we check for the open node that has minimum value of sum of g and h values and hence the one expected to cover the least path. This node is taken to evaluate its neighbours for further traversal and is popped from open.

```
for i in n:
    con = (p[0] + i[0], p[1] + i[1])
    if con == end:
        d = 0
        par[con[0], con[1], 0] = p[0]
        par[con[0], con[1], 1] = p[1]
        stop = time.time()
        m = (par[end[0]][end[1]][0], par[end[0]][end[1]][1])
while (True):
    img_cp[m[0]][m[1]] = 127
    d = d + dist(m, (par[m[0]][m[1]][0], par[m[0]][m[1]][1]))
    m = (par[m[0]][m[1]][0], par[m[0]][m[1]][1])
    if m == start or m == (-1, -1):
        break
cv2.namedWindow('Path using Astar', cv2.WINDOW_NORMAL)
cv2.imshow("Path using Astar", img_cp)
cv2.waitKey(0)

return d, stop
```

Now we begin evaluating the neighbours. We take a direction from n and take the pixel in that direction and store it in con. It terminates the loop if the end pixel is encountered and finds the distance covered in traversing the parents and also changes the colour of the pixels covered in a copy of the original array to demarcate the path. This is then displayed. The distance covered and the stop time is returned.

```
if not (inv(img, con[0], con[1]) or img[con[0]][con[1]] == 127):
    if not locked(img, p, i):
        img[con[0], con[1]] = 197
        if con in open:
            if (g[p[0]][p[1]] + dist(con, (p))) < g[con[0]][con[1]]:
                g[con[0]][con[1]] = (g[p[0]][p[1]] + dist(con, (p)))
                par[con[0], con[1], 0] = p[0]
                par[con[0], con[1], 1] = p[1]
                h[con[0]][con[1]] = dist(con, end)
        else:
            g[con[0]][con[1]] = (g[p[0]][p[1]] + dist(con, (p)))
            par[con[0], con[1], 0] = p[0]
            par[con[0], con[1], 1] = p[1]
            h[con[0]][con[1]] = dist(con, end)
            open[con] = con
```

Here we check if a neighbouring pixel is eligible to be considered as an open pixel and part of the path. By calling the inv and locked functions we eliminate the invalid pixels. After that we first check if the present path is a better path to the same pixel if a path leading to it already exists, otherwise we discard it. If a path to the pixel doesn't exist, we add the pixel to the open list as a prospective candidate for a new path.

```
if p == end:
    print('Goal reached')
    break
img[p[0]][p[1]] = 127
img_cpy = img.copy()
    img_cpy = cv2.resize(img_cpy, (800, 800),
interpolation=cv2.INTER_AREA)
    cv2.imshow("AStar", img_cpy)
    cv2.waitKey(1)
```

We set the path colour to gray which demarcates it as a visited pixel. A copy of the original image is produced, resized and displayed as a live depiction of the path traversal with a frame time of one milliseconds.

Calling Astar

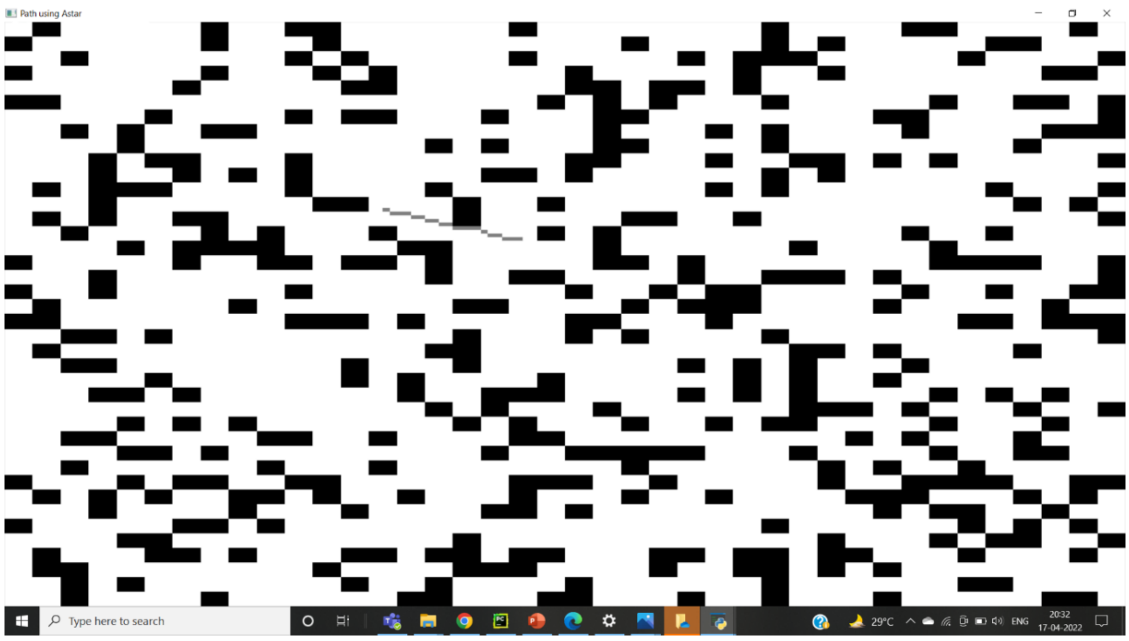
```
#Astar

img = mat.copy()
img_cp = mat.copy()
t, w = img.shape
start = critical[0]
end = critical[1]

beg = time.time()
```

```
d, stop = Astar(img, start, end)
tyme = stop - beg
line = """Astar :
Time = %f sec      Distance covered = %d pixels
""" % (tyme, d)
f.write(line)
p = end
```

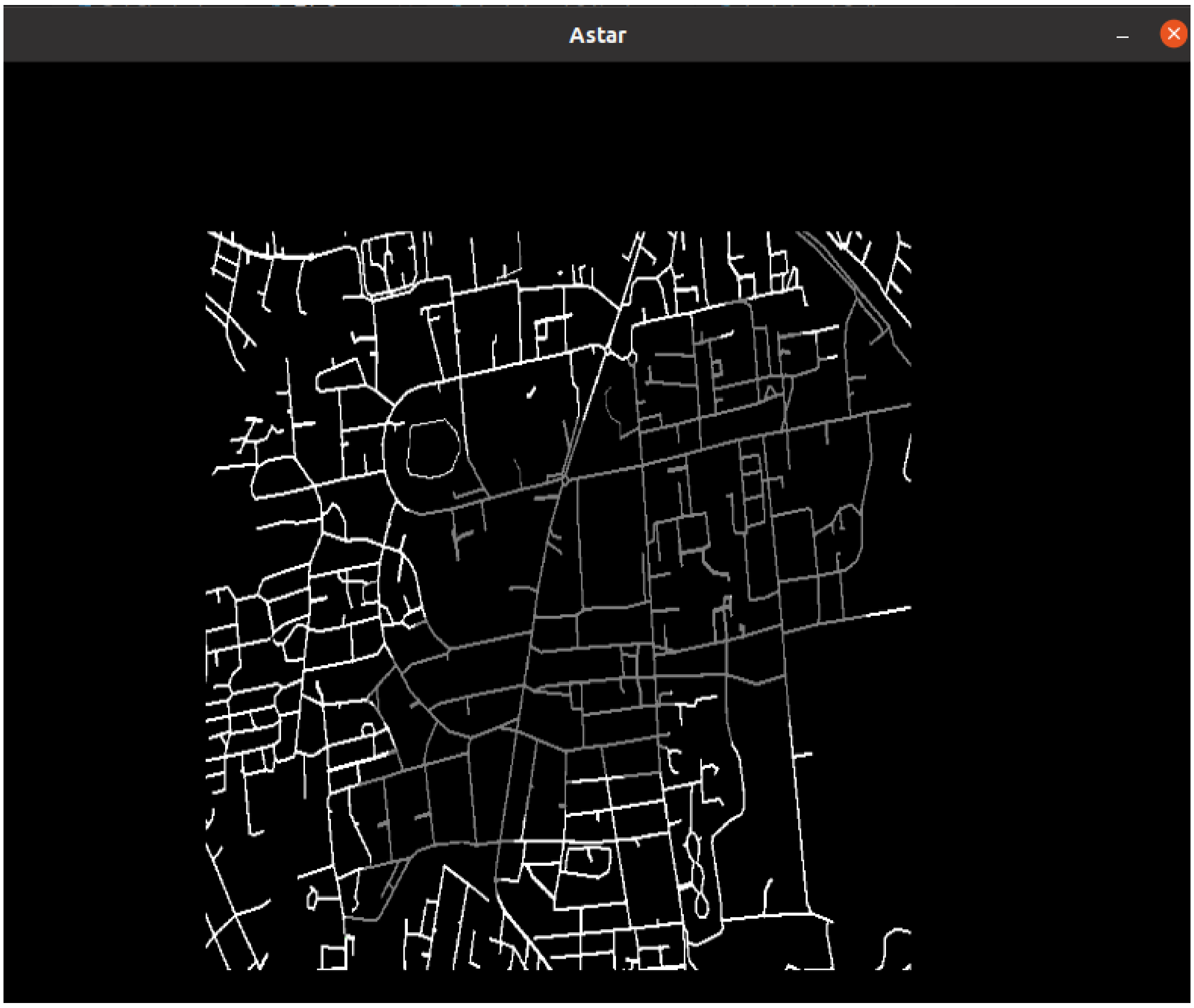
Here we call the Astar function to evaluate the maze generated. The returned distance and time is then written in the text file.

```
class car:
    def __init__(self, rrpm, lrpm):
        self.rrpm = rrpm
        self.lrpm = lrpm
```

In case of maps, the only difference is in the fact that for map ,3 values, red, green and blue values had to be set and checked.



Path generated using Astar algorithm



Path traversal in map using Astar

Task 2

Video generation for task 2

A prototype for a car object is created with two variables, lrpm signifying left wheel rpm and rrpm signifying right wheel rpm.

```
file = open("Car Reading.txt", "w+")
file.write("\n")
file.write("Motor Readings\n")
file.write("\n")
```

A .txt file called “Car Reading” is opened and prepared for storing the readings from video.

```
p = []
templates = [cv2.imread("301o.jpeg"), cv2.imread("601o.jpeg"),
cv2.imread("hump2o.jpeg"), cv2.imread("green2o.jpg"),
             cv2.imread("left2o.jpg"), cv2.imread("left3o.jpeg"),
cv2.imread("red2o.jpg"), cv2.imread("red3o.jpg"),
             cv2.imread("right3o.jpeg"), cv2.imread("right2o.jpeg"),
cv2.imread("road.png"), cv2.imread("slow2o.jpeg"),
             cv2.imread("stop1o.jpeg")]
```

A list is initialised for storing the order of pictures in video and the templates are stored in the templates list

```
##video creation

l = cv2.imread("301.jpeg")
p.append(l)

for i in range(23):

    r = rn.random()

    if r < 0.05:
        l = cv2.imread('red2.jpg', cv2.IMREAD_COLOR)
        p.append(l)
        l = cv2.imread('green2.jpg', cv2.IMREAD_COLOR)
        p.append(l)

    elif r < 0.1:
        l = cv2.imread('red3.jpg', cv2.IMREAD_COLOR)
        p.append(l)
        l = cv2.imread('green2.jpg', cv2.IMREAD_COLOR)
        p.append(l)

    elif r < 0.15:
        l = cv2.imread('left2.jpg', cv2.IMREAD_COLOR)
        p.append(l)

    elif r < 0.2:
        l = cv2.imread('left3.jpeg', cv2.IMREAD_COLOR)
        p.append(l)
    elif r < 0.25:
        l = cv2.imread('right2.jpeg', cv2.IMREAD_COLOR)
        p.append(l)

    elif r < 0.3:
        l = cv2.imread('right3.jpeg', cv2.IMREAD_COLOR)
        p.append(l)

    elif r < 0.35:
        l = cv2.imread('slow2.jpeg', cv2.IMREAD_COLOR)
        p.append(l)

    elif r < 0.4:
        l = cv2.imread('hump2.jpeg', cv2.IMREAD_COLOR)
        p.append(l)

    elif r < 0.45:
        l = cv2.imread('301.jpeg', cv2.IMREAD_COLOR)
        p.append(l)

    elif r < 0.5:
        l = cv2.imread('601.jpeg', cv2.IMREAD_COLOR)
        p.append(l)

    elif r < 0.55:
        l = cv2.imread('stop1.jpeg', cv2.IMREAD_COLOR)
        p.append(l)
        l = cv2.imread('green2.jpg', cv2.IMREAD_COLOR)
        p.append(l)

    else:
        l = cv2.imread('road.png', cv2.IMREAD_COLOR)
        p.append(l)

l = cv2.imread("red2.jpg")
p.append(l)
###
```

The video is created here by randomly choosing the image from a pre-existing set of images depending on a randomly generated number.

A sample image used in video generation

Template matching:

```
num_of_templates = len(templates)

arduino = serial.Serial('/dev/ttyACM0')
time.sleep(2)

while (len(p) != 0):

    pic = p.pop(0)
    cv2.imshow("Video", pic)
    cv2.waitKey(500)

    for j in range(num_of_templates):

        ##if template found
        cv2.imshow("Video", templates[j])

        w, h, l = templates[j].shape

        if (pic.shape[0] < w or pic.shape[1] < h):
            continue

        threshold = 0.99

        res = cv2.matchTemplate(pic, templates[j], cv2.TM_CCOEFF_NORMED)

        loc = np.where(res >= threshold)
if (len(loc[0]) != 0 and len(loc[1]) != 0):

    for pt in zip(*loc[::-1]):
        cv2.rectangle(pic, pt, (pt[0] + w, pt[1] + h), (0, 255, 255), 3)

    cv2.imshow("Video", pic)
    cv2.waitKey(500)
```

This is the connector between the Arduino and the python code. We first take each picture of the video and do template matching for each template with a matching threshold of 99%. The location is extracted in case a match is found (loc = np.where(res >= threshold)) and a rectangle is drawn to demarcate the region. The resulting image is shown temporarily.

Sample template used

Setting vehicle's parameters

```
if (j == 0):
    my_car.lrpm = my_car.rrpm = 60

elif (j == 1):
    my_car.lrpm = my_car.rrpm = 120

elif (j == 2):

    my_car.lrpm = my_car.rrpm = 40

elif (j == 3):

    my_car.lrpm = my_car.rrpm = 100

elif (j == 4 or j == 5):

    my_car.lrpm = 20
    my_car.rrpm = 60

elif (j == 6 or j == 7):

    my_car.lrpm = my_car.rrpm = 0

elif (j == 8 or j == 9):

    my_car.rrpm = 20
    my_car.lrpm = 60

elif (j == 10):

    my_car.lrpm = my_car.rrpm = 100

elif (j == 11):

    my_car.lrpm = my_car.rrpm = 40

elif (j == 12):

    my_car.lrpm = my_car.rrpm = 0

file.writelines(f"Left Motor rpm = {my_car.lrpm}    Right Motor rpm = 
{my_car.rrpm}")
file.write("\n")
```

Here the my_car object's lrpm and rrpm is set according to the template detected in the video. The resulting values are written in the file "Car Reading".

```
            arduino.write(b'%d\n' % my_car.lrpm)
            arduino.write(b'%d\n' % my_car.rrpm)
            cv2.waitKey(2000)

        ####

file.close()
cv2.destroyAllWindows()
```

The values are sent to the Arduino code. At the end of the code all files and windows are closed.

Task 2

Arduino Code

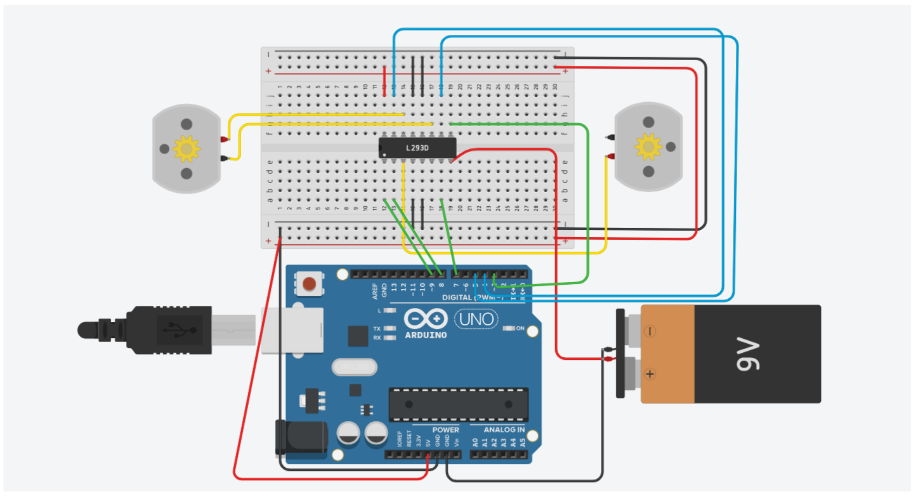
```
int lrpm=0;
int rrpm=0;
String LRPM;
String RRPM;

//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);
  }
}
```

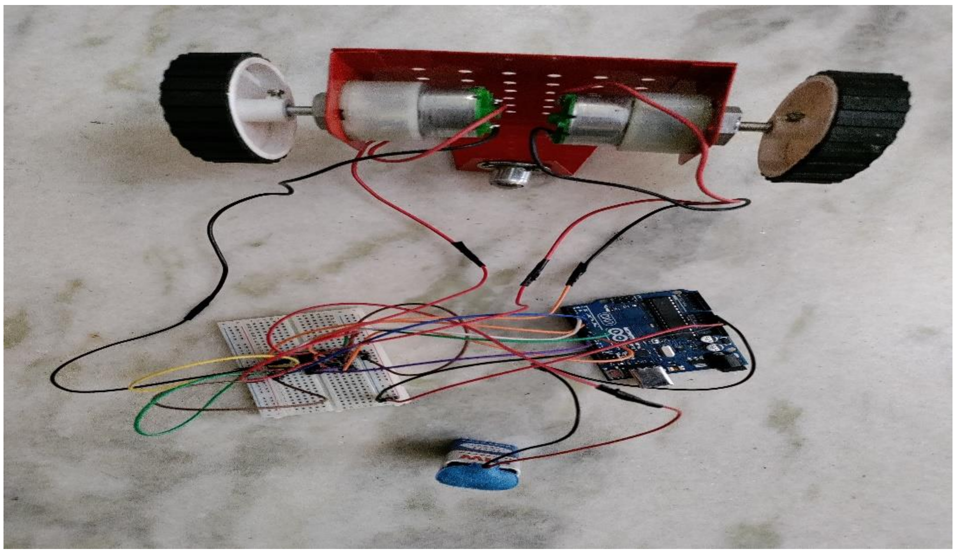
This code reads the values of the rpm of the left and right wheels of the car from the python code as a string and then converts them into int type. It then scales it into a range of 0-255 using the map function and passes these values to the enA and enB pins.

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.

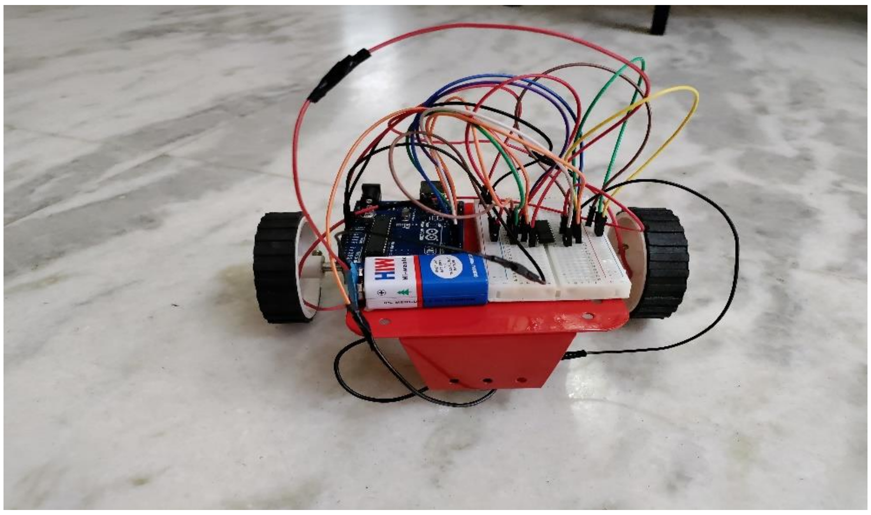
Car model

Circuit





Car Model



Packages used:

```
cv2
random as rn
numpy as np
serial
time
```