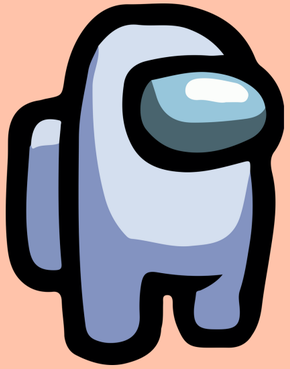
IIT Kharagpur

Sussy Pixels

WinterSchool 2022
Computer Vision

OpenCV

Team Members



Yash Sirvi
(21CS10083)

Ashwin Prasanth
(21CS30009)

Yelisetty Karthikeya S M
(21CS30060)

Task 1

Maze Generation

Maze Generation

- Probability of becoming an obstacle: 0.2 to 0.3.
- Our maze is colored in BGR format.
    - Red: Starting pixel
    - Blue: Target pixel
- The maze generation file can be found in

  Task1/maze_generation.py

50x50px maze scaled upto 500x500px

Code for Maze Generation

Check Functions

```python
def inRange(canvas,point):
    #checks if point is within the range of the image's dimensions
    return (point[0]>=0 and point[0]<canvas.shape[0] and point[1]>=0 and point[1]<canvas.shape[1])


def checkNbhd(canvas, x, y):
    #check if atleast more than one white neighbour is present
    count = 0
    for (i, j) in [(-1,0), (1,0), (0,-1), (0,1)]:
        if inRange(canvas, (x+i, y+j)) and (canvas[x+i][y+j] == (255,255,255)).all():
            count+=1
    if(count>1):
        return True
    else:
        return False
```

Generating maze

```python
def createMaze():
    canvas = np.full((50, 50, 3), 255, dtype=np.uint8)
    # canvas = np.full((50, 50,3), 255, dtype=np.uint8)
    for i in range(canvas.shape[0]):
        for j in range(canvas.shape[1]):
            randomnumber = random.randint(1, 100)
            if randomnumber<prob and checkNbhd(canvas,i,j):
                canvas[i][j] = (0,0,0)
            else:
                if checkNbhd(canvas,i,j):
                    maze.append((i,j))

    startXY = random.choice(maze)
    while(not checkNbhd(canvas,startXY[0],startXY[1])):
        startXY = random.choice(maze)
    endXY  = random.choice(maze)
    while(endXY==startXY or not checkNbhd(canvas,endXY[0],endXY[1])):
        endXY  = random.choice(maze)

    canvas[startXY] = RED
    canvas[endXY] = BLUE
    smaller_canvas = canvas
    canvas = cv2.resize(canvs,(500,500),
interpolation=cv2.INTER_AREA)
    return canvas, startXY, endXY, smaller_canvas
```

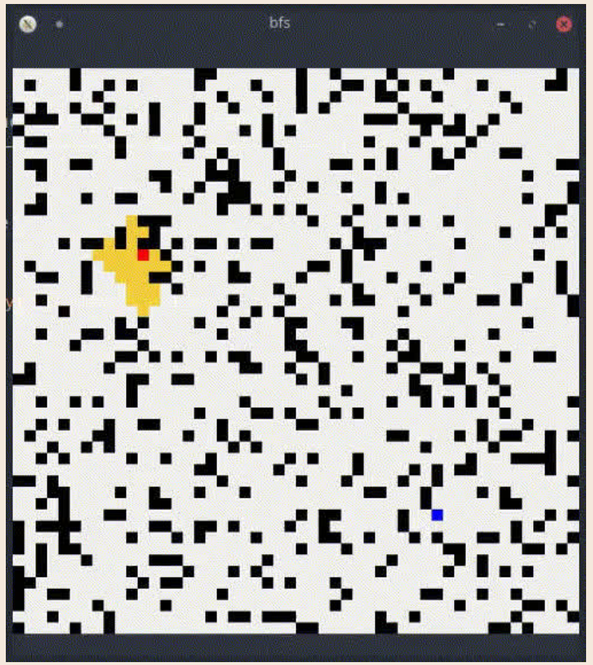
Better Check function

```
    while True:
        global startXY, endXY
        startXY = random.choice(maze)   # Choosing a start location in the list of white
pixels
        endXY  = random.choice(maze)    # Choosing a end location in the list of white
pixels
        temp = canvas.copy()
        algos = MazeTraversal(temp, startXY, endXY)
        a, b = algos.aStar(temp)
        if a is not None:
            break
```

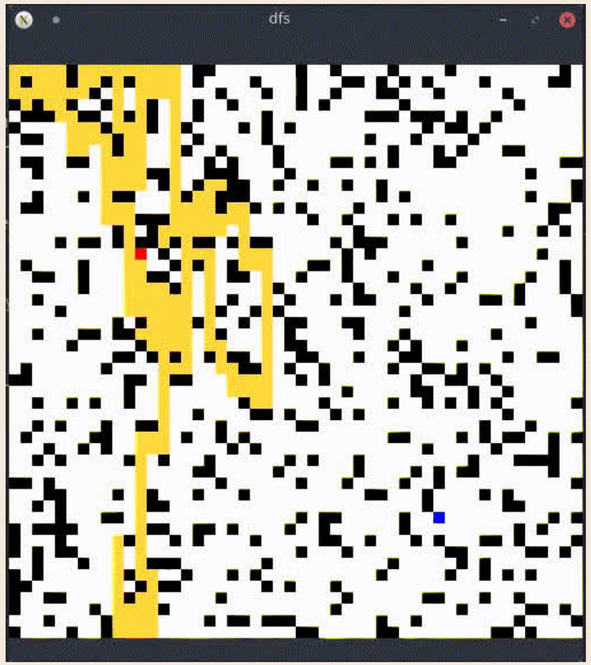
We will explain how the A* algorithm and the MazeTraversal module is implemented later.

Task 1: Part 1

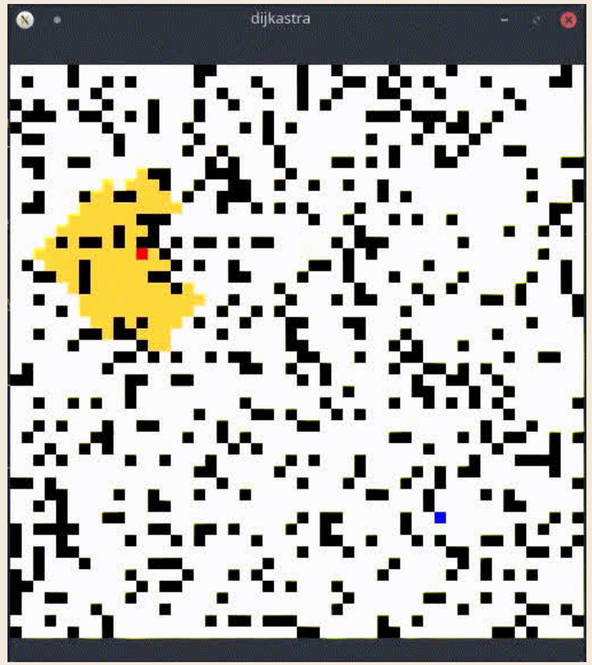
Path finding with different Algorithms



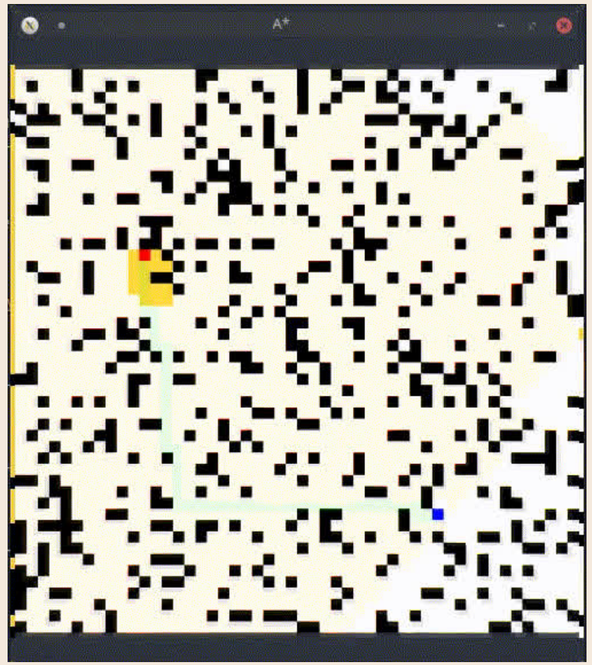
BFS:
Distance: 49 pixels
Time: 8.8563 seconds



DFS:
Distance: 523 pixels
Time: 5.103 seconds



Dijkstra:
Distance: 49 pixels
Time: 17.604 seconds



A*:
Distance: 49 pixels
Time: 0.7088 seconds

*Time includes the time taken for visualisation as well. Without visualization the time taken to solve a maze decreases 10-15 times

Comparison of these algorithms

```
Average time taken by BFS: 0.02754771661758423 Seconds
Average distance travelled in BFS: 35.625 pixels
----------------------------------------------------
Average time taken by DFS: 0.025121071577072142 Seconds
Average distance travelled in DFS: 226321.209 pixels
----------------------------------------------------
Average time taken by Dijkstra: 0.021741742610931397 Seconds
Average distance travelled in Dijkstra: 35.625 pixels
----------------------------------------------------
Average time taken by Astar: 0.0014459338188171386 Seconds
Average distance travelled in Astar: 37.275 pixels
----------------------------------------------------
```

Running all of these algorithms on a set of 1000 randomly generated mazes and taking an average
Go to Task1/part1Results and run part1_scores.py

Explanation for Different Path Planning Algorithms

Maze Traversal Class

We have created a common class called "MazeTraversal" in "Task1/Traversal.py" under which we have included the functions for each path traversal algorithm, which will make use of the following parameters of the class:

- **inimg -** The image of the maze
- **start -** The coordinates of the start of the maze
- **end -** The coordinates of the end of the maze

```
class MazeTraversal:
    def __init__(self, inimg, start, end):
        self.inimg = inimg
        self.start = start
        self.end = end
```

We have also defined some common functions outside the class that will be utilized in the implementation of some of the traversal algorithms, that are explained in the next slide

Common functions

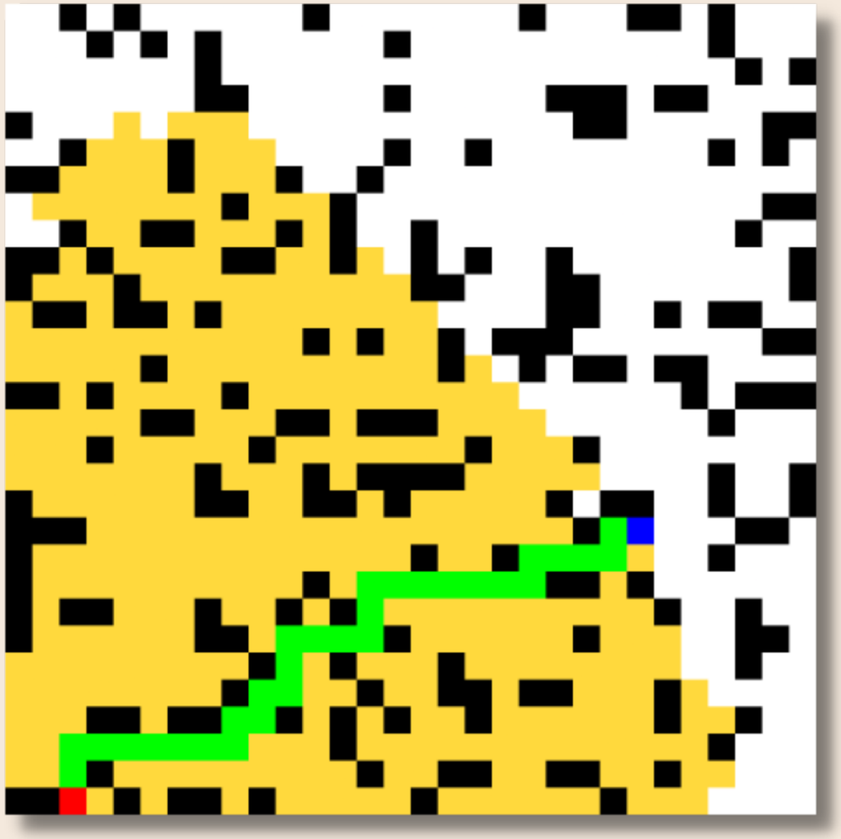
```python
def find_dist(point, current):
    return (point[0] - current[0])**2 + (point[1] - current[1])**2
    #Since we dont need the exact distance, we will keep it as square itself

def inRange(img,point):
    #checks if point is within the range of the image's dimensions
    return (point[0]>=0 and point[0]<img.shape[0] and point[1]>=0 and point[1]
<img.shape[1])
```

1. **find_dist(point, current):** To calculate the square of the distance between two pixels 'point' and 'current', which will be used as the cost for traveling in Dijkstra's and A* algorithms
2. **inRange(img, point):** To ensure that the given point 'point' lies within the bounds of the dimensions of the image 'img'

Bread First Search (BFS) Algorithm

- Go to any vertex, explore it completely, add the explored vertices in a queue, move to the next vertex in the queue and explore that completely, and so on...
- Guarantees shortest path



```
def bfs(self, img):
    q = [[self.start]]       # Queue for keeping track of path

    while q:
      path = q.pop(0)         # Choosing a path from the queue
      r, s = path[-1]         # Coordinates of the last pixel in path
      for (u,v) in [(-1,0), (1,0), (0,-1), (0,1)]:        # Looping over the four corners of the pixel
        if inRange(img, (r+u,s+v)) and (img[r+u][s+v]!=BLACK).any() and (img[r+u][s+v]!=COLOR2).any():
            if (img[r+u][s+v] == WHITE).all():
                img[r+u][s+v] = COLOR2          # Coloring the visited pixel
                new_path = list(path)
                new_path.append((r+u, s+v))     # Adding the visited pixel to it's parent path
                q.append(new_path)              # updating the path in queue
            elif ((r+u, s+v) == self.end):
                return path           # Returning the path from self.start to the self.end when the
self.end is visited
```

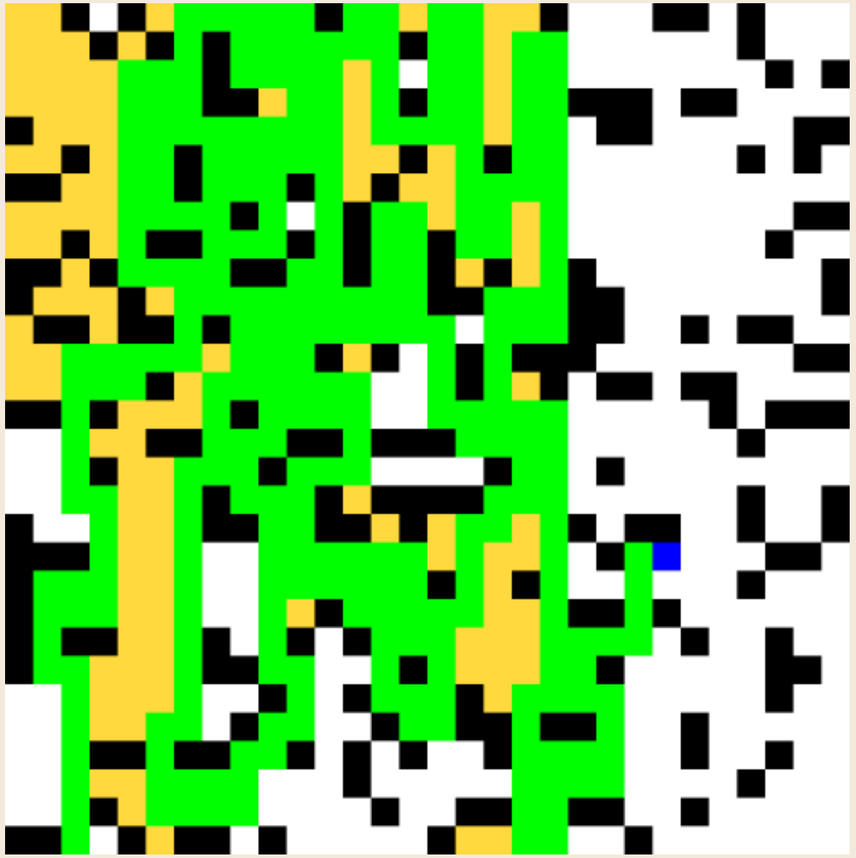
q is a list of paths

Tracking the final Path

```
def trackBfs(self):
    img = np.copy(self.inimg)
    path= self.bfs(img)
    path.pop(0)
    for pixel in path:
        img[pixel] = GREEN
        if img.shape[0] < 100:
            showImg = cv2.resize(img, (500,500), interpolation=cv2.INTER_AREA)
            cv2.imshow("bfs", showImg)
            cv2.waitKey(1)
        else:
            cv2.imshow("bfs", img)
    print("Distance: "+str(len(path)+1) + " pixels")
```

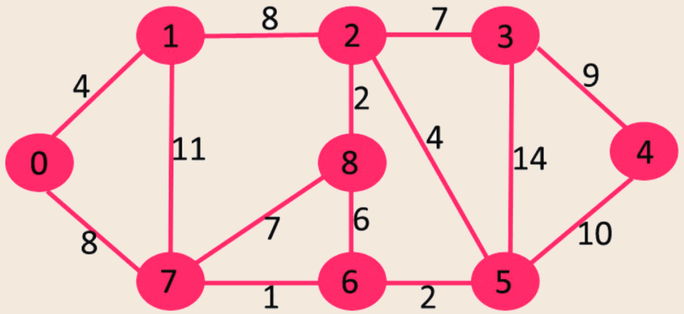
Depth First Search (DFS) Algorithm

- Go to any vertex, explore any neighbor, start exploring that neighbor until you have no more children to explore, then go back and explore the other neighbors.
- Does not guarantee the shortest path
- We are using the recursive approach in our implementation



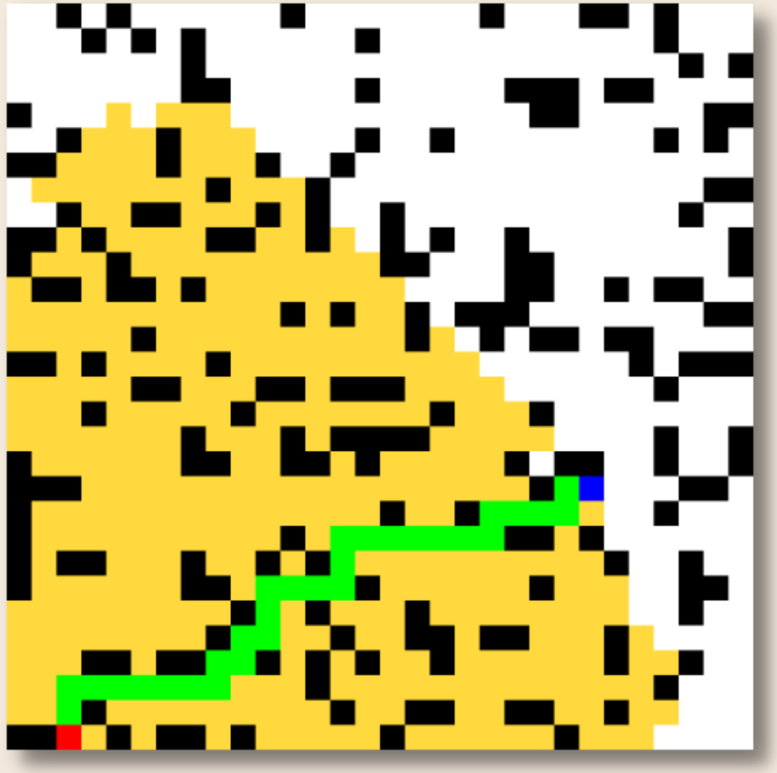
```
    def dfs2(self, img, start):
        global path_found
        global dist
        i, j = start
        for (x,y) in [(-1,0), (1,0), (0,-1), (0,1)]:        # Loops over the surrounding pixels
            if inRange(img, (x+i, y+j)) and not (img[x+i][y+j]==BLACK).all() and not (img[x+i][y+j]==COLOR2).all():
# Check if pixel lies in the valid path
                if ((x+i, y+j) == self.end):
                    path_found = True
                    img[i][j] = GREEN
                    dist +=1
                    break
                if not (img[x+i][y+j] == BLUE).all() and not (img[x+i][y+j] == RED).all():
                            img[x+i][y+j] = COLOR2
                            self.dfs2(img, (x+i,y+j))
                if path_found:      # Stops further recursion and starts traceback
                    img[i][j] = GREEN
                    dist +=1
                    break
```

Dijkstra's Algorithm



Graph with cost for traveling from one node to another denoted

- Finds the shortest distance between nodes in a graph
- The cost of traveling between two pixels are defined as the square of the distance between them, found using distance formula



Initialization

```
def dijkstra(self, img):
        n, m = img.shape[:2]    # Height and Width of the maze
        parent = np.zeros((n,m,2))  # Keeping track of the parent pixels
        visited = []  # Keeping a list of visited pixels so that we dont visit the same pixel again
        current = self.start
        distance = np.full((n,m), np.inf)   # initialising the distance of every pixel to infinity
        distance[self.start] = 0 # Distance of 'start' from 'start' will be 0
```

Looping through pixels until target is found

```
# Loop runs until we reach the target
        while True:
            if current != self.end:
                for (i, j) in [(-1,0), (1,0), (0,-1), (0,1)]:   # Visiting all 4 neighboring pixels

                    # Code for visualisation
                    if img.shape[0] < 100:       # Conditions for scaling
                        showImg = cv2.resize(img, (500,500), interpolation=cv2.INTER_AREA)
                        cv2.imshow("dijkstra", showImg)
                        cv2.waitKey(1)
                    else:
                        cv2.imshow("dijkstra", img)
```

The image is reshaped into 500 x 500 dimensions if it's too small, for better visualisation

Looping through pixels until target is found

```
newpoint = (current[0]+i, current[1]+j)
# Checking if the pixel lies in the valid path or not
if inRange(img, newpoint) and not (img[newpoint] == BLACK).all():
    if distance[newpoint] > distance[current] + find_dist(newpoint, current):
        distance[newpoint] = distance[current] + find_dist(newpoint, current)
        img[newpoint] = COLOR2  # Coloring the visited pixels
        parent[newpoint] = current  # Assigning a parent to the visited pixel
        visited.append(newpoint)
```

For each neighbour of the current point, we update it's distance if it is lesser than the current distance and also update its parent while ensuring that the neighbor is in range and not an obstacle. We then color the pixel for visual purposes and add it to the list of "visited" pixels.

Finding the next 'current' point and code for last pixel

```
    # Choosing the pixel with the least current distance for analysis in the next cycle
        min = np.inf
    for point in visited:
        if distance[point] < min:
            min = distance[point]
            current = point
    visited.remove(current)
else:
    img[self.end] = BLUE
    distance[self.end] = distance[current] +find_dist(self.end, current)
    return distance[self.end], parent
```

The next 'current' point is selected by checking the pixel with least distance out of all the visited pixels. On finding this point, it is removed from the 'visited' list.

Also, we color the ending pixel blue, update it's distance and return it along with 'parent' array

Visualising the path found by Dijkstra's Algorithm

```
def trackDijkstra(self):
        img = np.copy(self.inimg)       # Making a copy of the maze for the algorithm to work on
        dist, parent = self.dijkstra(img)   # Running the algorithm
        # Tracking the final path
        current = tuple(list(map(int, parent[self.end])))
        while current != self.start:
            img[current] = GREEN
            current = tuple(list(map(int, parent[current])))
            # Scaling the right maze
            if img.shape[0] < 100:
                showImg = cv2.resize(img, (500,500), interpolation=cv2.INTER_AREA)
                cv2.imshow("dijkstra", showImg)
                cv2.waitKey(1)
            else:
                cv2.imshow("dijkstra", img)
        # Printing the path distancce
        print("Distance: " + str(int(dist)) + " pixels")
```

A* Algorithm

- Works just like Dijkstra with a small change in

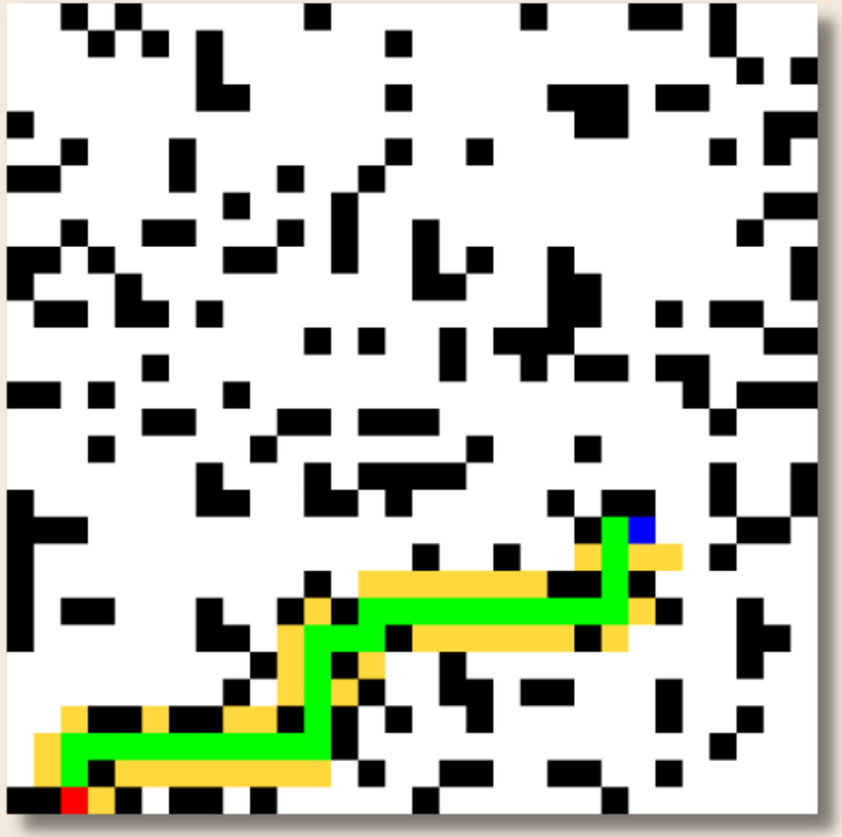
  the way

  the distance is calculated.

- distance =

  distance_from_start_along_path +

  distance_from_end



Change from Dijkstra

```
...
if inRange(img, newpoint) and not (img[newpoint] == BLACK).all():
    pixel_distance[newpoint] = pixel_distance[current] + find_dist(newpoint, current)
    if distance[newpoint] > pixel_distance[newpoint]+ find_dist(newpoint, current) +
find_dist(newpoint,self.end):
...
```

Maze for Task 2

Path finding with different Algorithms Part 2

BFS:
Distance: 957 pixels
Time: 4.1725 seconds

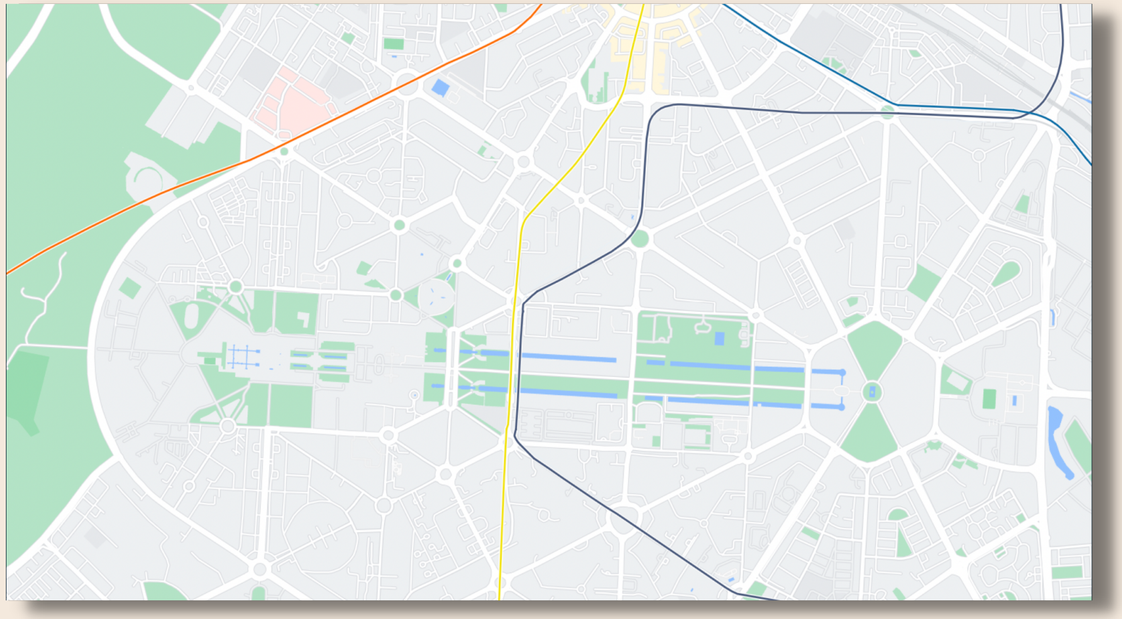
DFS:
Distance: 4933 pixels
Time: 1.6195 seconds

Dijkstra:
Distance: 957 pixels
Time: 5.2352 seconds

A*:
Distance: 979 pixels
Time: 3.4153 seconds

Image Processing for Part 3

- We have used the Google Maps image of the parliament region in Connaught Place, New Delhi
- We then had to process the image in order to run the algorithms on it effectively, using "Task1/processing_image.py"
- The code we ran for the same is explained in the next few slides:



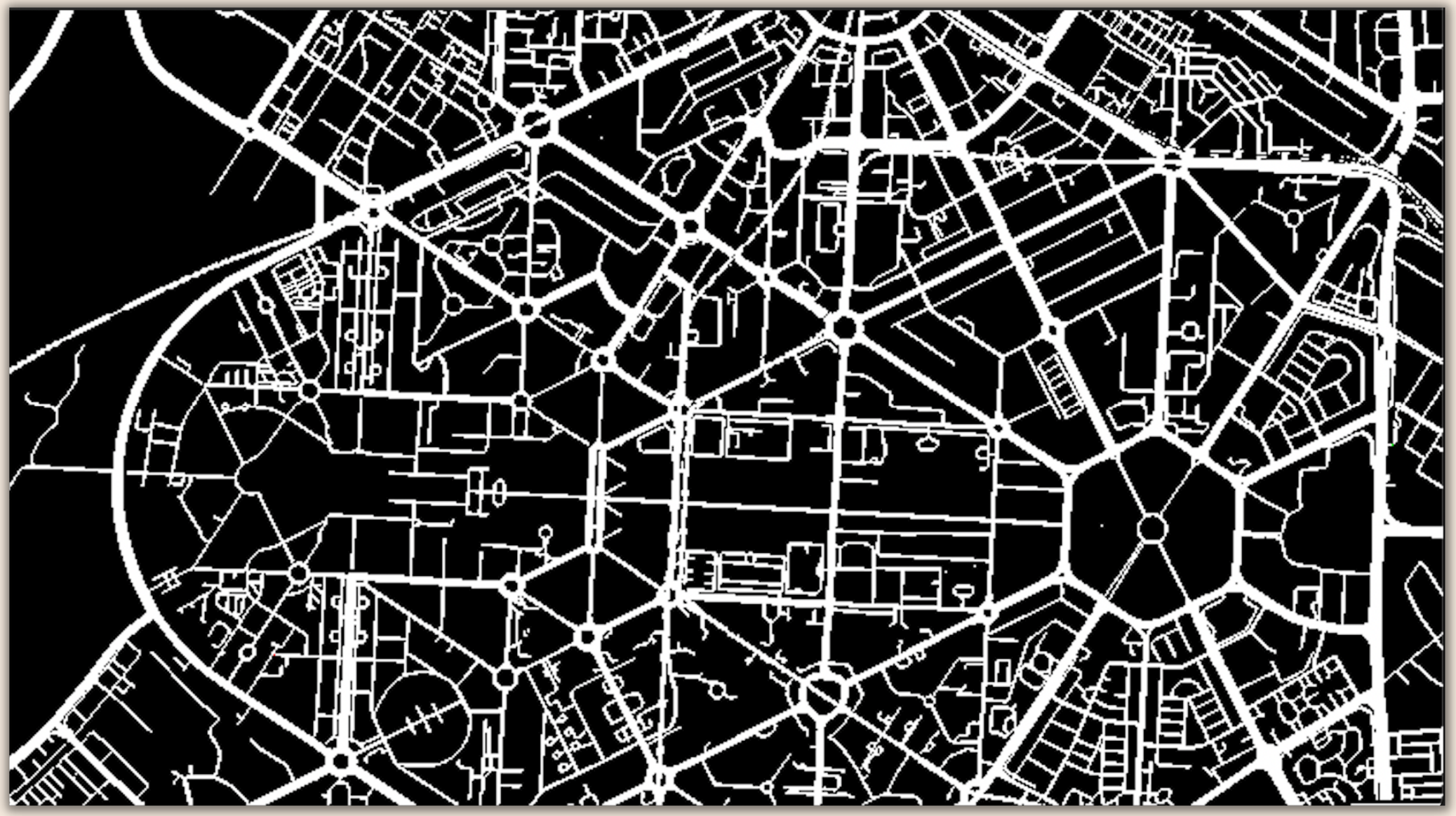
```python
colors = [[118,86,73], [68,227,239], [37,117,236], [244,243,241], [255,255,255],[128,98,86]]
maze = cv2.imread("Task1/googlemaps.png") # Reading google maps image
n, l = maze.shape[:2] # Getting dimensions of the image
tempmaze = maze.copy() # Making a copy of maze which we will process
for i in range(n):
    for j in range(l):
      # Looping through every pixel of the maze
        for color in colors:
            if (maze[i][j] == color).all():
                tempmaze[i][j] = [255, 255, 255] # The path is made white
                for (r,s) in [(-1,0), (1,0), (0,-1), (0,1)]:
                    if inRange(maze, (r+i, s+j)):
                        tempmaze[r+i][s+j] =  [255, 255, 255] # It's neighbors also made white
                break
        tempmaze[i][j] = [0,0,0] # If it's not in path, it is made black
```

Using a color picker, all the colors appearing along the required path were found, and the pixels with these colors, along with their neighbors were made white, and the rest of the pixels were made black.

```
tempmaze = cv2.resize(tempmaze,(int(l*0.5), int(n*0.5)), interpolation=INTER_AREA)
# The maze is resized to an apt size
n, l = tempmaze.shape[:2] # New dimensions are stored
# Applying thresholding to ensure that the image is binary
for i in range(n):
    for j in range(l):
        if tempmaze[i][j][0] > 127:
            tempmaze[i][j] = (255,255,255)
        else:
            tempmaze[i][j] = (0,0,0)
# Defining start and end points and coloring them red and green respectively
startXY = (335,139)
endXY = (226,720)
tempmaze[startXY]=RED
tempmaze[endXY]=GREEN
```

The final maze is shown in the next slide!

Maze for Task 3

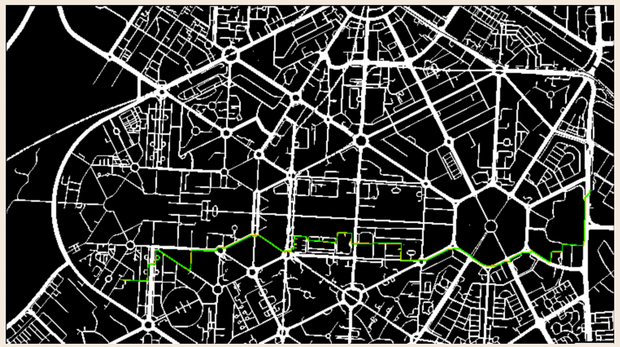


Pathfinding with different Algorithms Part 3

BFS:
Distance: 786 pixels
Time: 5.021 seconds

DFS:
Distance: 16014 pixels
Time: 2.5113 seconds

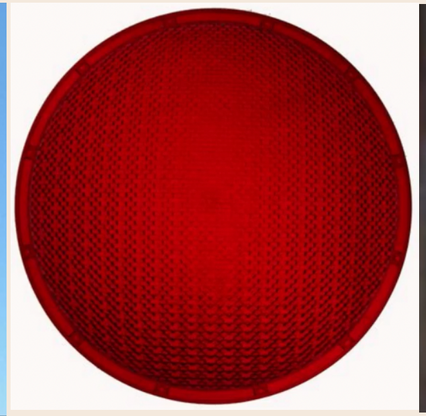
Dijkstra:
Distance: 786 pixels
Time: 6.7691 seconds



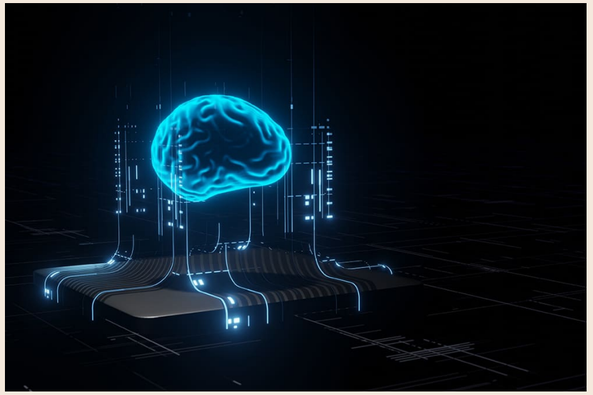
A*:
Distance: 942 pixels
Time: 0.244 seconds

Task 2

Signs Used in the Video



The Machine Learning Model



- For this task, we used a pretrained Machine Learning model to predict the class to which the traffic sign belongs to.

- The model used for this project is a classification model, which was trained by using supervised learning algorithms.

- We used a python library - Tensorflow, for the purpose of loading and using the pre-trained saved model.

TensorFlow

```
from tensorflow.keras.models import load_model

model = load_model('Task2/traffic_classifier.h5')
cap = cv.VideoCapture('Task2/video.mp4')
sign = 0
diameter = 0.1524         # diameter = 6 inches or 15.24 cm

while True:
    with open("Task2/output.txt", "a") as myfile:
        success, frame = cap.read()
        if not success:
            break
        image = Image.fromarray(frame)
        image = image.resize((30, 30))
        image = np.expand_dims(image, axis=0)
        image = np.array(image)
        pred = model.predict([image]).argmax()
        tempsign = classes[pred]
```

The **load_model** function imported from tensorflow returns a **Machine Learning model** which was saved using **model.save()** while training the model.

There is some preprocessing to be done before using **model.predict()**, so as to fit the image in the dimensions of the images on which the model was trained.

model.predict() returns a matrix of normalised probability, of which we take out the max probability using **.argmax()** and store the corresponding class to the processed image.

```
if tempsign != sign:
            sign = tempsign
            if sign in ["Stop", "Red Traffic Light"]:
                myfile.write("Stop Motor\n")
                print(sign)
            elif sign == "Green Traffic Light":
                myfile.write("Start Motor\n")
                print(sign)
            elif sign in ['20','30','50','60','70','80','100','120'] :
                rpm = (50*int(sign))/(3*np.pi*diameter)             # calculating rpm from speed(km/hr)
                myfile.write("%.2f"%rpm)
                myfile.write("\n")
                print(sign + " km/hr")
            else:
                myfile.write(sign + "\n")
                print(sign)


        cv.imshow('Output', frame)
```

Here is the continuation of that previous code snippet. Here we append the instructions file for the self driving vehicle, as per the class predicted by the model.

The output of task2.py

```
Start Motor
696.22
1740.54
Turn left ahead
1044.32
Turn right ahead
3481.08
4177.30
2088.65
696.22
Stop Motor
Start Motor
2088.65
3481.08
1740.54
Stop Motor
```

Thank you!