**ARTIFICIAL INTELLIGENCE**

PROJECT REPORT

**TOPIC:** To Find the Best Route from Source to Destination using A Star Technique.

**X-X-X-X-X-X-X-X-X-X-X-X-X**

**CODE**

import numpy as np

from tkinter import \*

class Node:

"""

A node class for A\* Pathfinding

parent is parent of the current Node

position is current position of the Node in the maze

g is cost from start to current Node

h is heuristic based estimated cost for current Node to end Node

f is total cost of present node i.e. : f = g + h

"""

def \_\_init\_\_(self, parent=None, position=None):

self.parent = parent

self.position = position

self.g = 0

self.h = 0

self.f = 0

def \_\_eq\_\_(self, other):

return self.position == other.position

#This function return the path of the search

def return\_path(current\_node,maze):

path = []

no\_rows, no\_columns = np.shape(maze)

# here we create the initialized result maze with -1 in every position

# result = [[-1 for i in range(no\_columns)] for j in range(no\_rows) ]

result = [[-1 for i in range(no\_columns)] for j in range(no\_rows) ]

for i in range(no\_rows):

for j in range(no\_columns):

if(maze[i][j]==1):

result[i][j]=-2

else:

result[i][j]=-1

current = current\_node

while current is not None:

path.append(current.position)

current = current.parent

# Return reversed path as we need to show from start to end path

path = path[::-1]

start\_value = 0

# we update the path of start to end found by A-star serch with every step incremented

#by 1

for i in range(len(path)):

result[path[i][0]][path[i][1]] = start\_value

start\_value += 1

return result

def search(maze, cost, start, end):

"""

Returns a list of tuples as a path from the given start to the given end in

the given maze

:param maze:

:param cost

:param start:

:param end:

:return:

"""

# Create start and end node with initized values for g, h and f

start\_node = Node(None, tuple(start))

start\_node.g = start\_node.h = start\_node.f = 0

end\_node = Node(None, tuple(end))

end\_node.g = end\_node.h = end\_node.f = 0

# Initialize both yet\_to\_visit and visited list

# in this list we will put all node that are yet\_to\_visit for exploration.

# From here we will find the lowest cost node to expand next

yet\_to\_visit\_list = []

# in this list we will put all node those already explored so that we don't

#explore it again

visited\_list = []

# Add the start node

yet\_to\_visit\_list.append(start\_node)

# Adding a stop condition. This is to avoid any infinite loop and stop

# execution after some reasonable number of steps

outer\_iterations = 0

max\_iterations = (len(maze) // 2) \*\* 10

# what squares do we search . serarch movement is left-right-top-bottom

#(4 movements) from every positon

move = [[-1, 0 ], # go up

[ 0, -1], # go left

[ 1, 0 ], # go down

[ 0, 1 ]] # go right

"""

1) We first get the current node by comparing all f cost and selecting the lowest cost node for further expansion

2) Check max iteration reached or not . Set a message and stop execution

3) Remove the selected node from yet\_to\_visit list and add this node to visited list

4) Perofmr Goal test and return the path else perform below steps

5) For selected node find out all children (use move to find children)

a) get the current postion for the selected node (this becomes parent node for the children)

b) check if a valid position exist (boundary will make few nodes invalid)

c) if any node is a wall then ignore that

d) add to valid children node list for the selected parent

For all the children node

a) if child in visited list then ignore it and try next node

b) calculate child node g, h and f values

c) if child in yet\_to\_visit list then ignore it

d) else move the child to yet\_to\_visit list

"""

#find maze has got how many rows and columns

no\_rows, no\_columns = np.shape(maze)

# Loop until you find the end

while len(yet\_to\_visit\_list) > 0:

# Every time any node is referred from yet\_to\_visit list,

#counter of limit operation incremented

outer\_iterations += 1

# Get the current node

current\_node = yet\_to\_visit\_list[0]

current\_index = 0

for index, item in enumerate(yet\_to\_visit\_list):

if item.f < current\_node.f:

current\_node = item

current\_index = index

# if we hit this point return the path such as it may be no solution or

# computation cost is too high

if outer\_iterations > max\_iterations:

print ("giving up on pathfinding too many iterations")

return return\_path(current\_node,maze)

# Pop current node out off yet\_to\_visit list, add to visited list

yet\_to\_visit\_list.pop(current\_index)

visited\_list.append(current\_node)

# test if goal is reached or not, if yes then return the path

if current\_node == end\_node:

return return\_path(current\_node,maze)

# Generate children from all adjacent squares

children = []

for new\_position in move:

# Get node position

node\_position = (current\_node.position[0] + new\_position[0], current\_node.position[1]

+ new\_position[1])

# Make sure within range (check if within maze boundary)

if (node\_position[0] > (no\_rows - 1) or

node\_position[0] < 0 or

node\_position[1] > (no\_columns -1) or

node\_position[1] < 0):

continue

# Make sure walkable terrain

if maze[node\_position[0]][node\_position[1]] != 0:

continue

# Create new node

new\_node = Node(current\_node, node\_position)

# Append

children.append(new\_node)

# Loop through children

for child in children:

# Child is on the visited list (search entire visited list)

if len([visited\_child for visited\_child in visited\_list if visited\_child == child]) > 0:

continue

# Create the f, g, and h values

child.g = current\_node.g + cost

## Heuristic costs calculated here, this is using eucledian distance

child.h = (((child.position[0] - end\_node.position[0]) \*\* 2) +

((child.position[1] - end\_node.position[1]) \*\* 2))

child.f = child.g + child.h

# Child is already in the yet\_to\_visit list and g cost is already lower

if len([i for i in yet\_to\_visit\_list if child == i and child.g > i.g]) > 0:

continue

# Add the child to the yet\_to\_visit list

yet\_to\_visit\_list.append(child)

def btn(self,fr):

path = search(maze,cost, start, end)

sol(path,fr)

def sol(path,fr):

for i in range(1,6):

for j in range(6):

if path[i-1][j]==-2:

fr1 = Frame(fr,height=60,width=60,bg='crimson')

fr1.grid(row=i, column=j)

elif path[i-1][j]==-1:

fr1 = Frame(fr,height=60,width=60,bg='#FFB266')

fr1.grid(row=i, column=j)

else:

fr1 = Frame(fr,height=60,width=60,bg='#FF3333')

fr1.grid(row=i, column=j)

if \_\_name\_\_ == '\_\_main\_\_':

maze = [[0, 1, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 1],

[0, 1, 0, 0, 1, 0],

[1, 1, 1, 0, 0, 0],

[0, 0, 0, 0, 1, 0]]#if someone wants to change the obstacle please change

#the position of in the maze matrix and do not changt the size

#of the matrix as gui is connected with it.

start = [0, 0] # starting position

end = [4,0] # ending position

cost = 1 # cost per movement

root = Tk()

root.geometry('1200x900')

root.title("BEST ROUTE FINDER")

f=Frame(root,height=120,width=1200,bg='crimson')

f.pack()

label10 = Label( f, text="Best Route Finding Using A Star", font='Times 32 bold',bg='crimson',

fg='black', height=2, width=1200)

label10.pack()

h=Frame(root,height=120,width=1200)

h.pack()

q=Frame(h,height=60,width=60,bg='crimson')

q.grid(row=0,column=1)

label = Label( h, text="Obstacle", font='Times 16 bold',

fg='black', height=1, width=8)

label.grid(row=0,column=2)

w=Frame(h,height=60,width=60,bg='#009900')

w.grid(row=0,column=3)

label1 = Label( h, text="END", font='Times 16 bold',

fg='black', height=1, width=8)

label1.grid(row=0,column=4)

er=Frame(h,height=60,width=60,bg='#FFB266')

er.grid(row=0,column=5)

label2= Label( h, text="Allowed Path ", font='Times 16 bold',

fg='black', height=1, width=12)

label2.grid(row=0,column=6)

e=Frame(h,height=60,width=60,bg='#CC6600')

e.grid(row=0,column=7)

label3 = Label( h, text="START", font='Times 16 bold',

fg='black', height=1, width=8)

label3.grid(row=0,column=8)

q1=Frame(h,height=60,width=60,bg='#FF3333')

q1.grid(row=0,column=9)

label4 = Label( h, text="Best Route", font='Times 16 bold',

fg='black', height=1, width=12)

label4.grid(row=0,column=10)

q2=Frame(root,height=100,width=1200)

q2.pack()

label5 = Label( q2, text="If you want to change the obstacles please "+

"change the position of 1's in the 'MAZE' matrix without "+

"changing the size of matrix ", font='Times 16 bold',

fg='black', height=2, width=1200)

label5.pack()

fr=Frame(root,height=300,width=360)

fr.pack()

for i in range(1,6):

for j in range(6):

if maze[i-1][j]==1:

fr1 = Frame(fr,height=60,width=60,bg='crimson')

fr1.grid(row=i, column=j)

elif i-1==start[0] and j==start[1]:

fr1 = Frame(fr,height=60,width=60,bg='#CC6600')

fr1.grid(row=i, column=j)

elif i-1==end[0] and j==end[1]:

fr1 = Frame(fr,height=60,width=60,bg='#009900')

fr1.grid(row=i, column=j)

else:

fr1 = Frame(fr,height=60,width=60,bg='#FFB266')

fr1.grid(row=i, column=j)

q3=Frame(root,height=20,width=1200)

q3.pack()

button1 = Button(root, text="FIND PATH", font='Times 20 bold', bg='crimson',

fg='white', height=1, width=10,command=lambda:btn(btn,fr))

button1.pack()

root.mainloop()

##-1 in the matrix int the printed below shows unused path

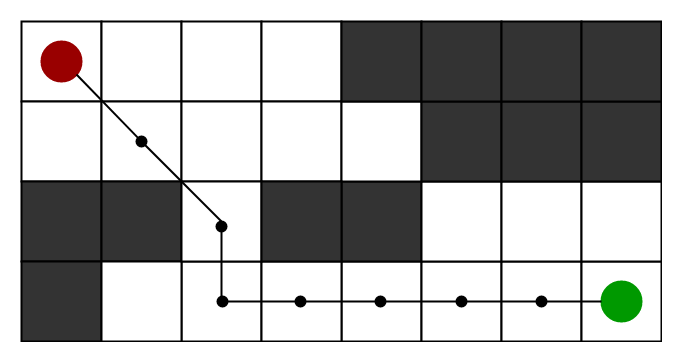
#-2 denotes obstacles

#anything else shows the cost of the paths

X-X-X-X-X-X-X-X-X-X-X-X-X

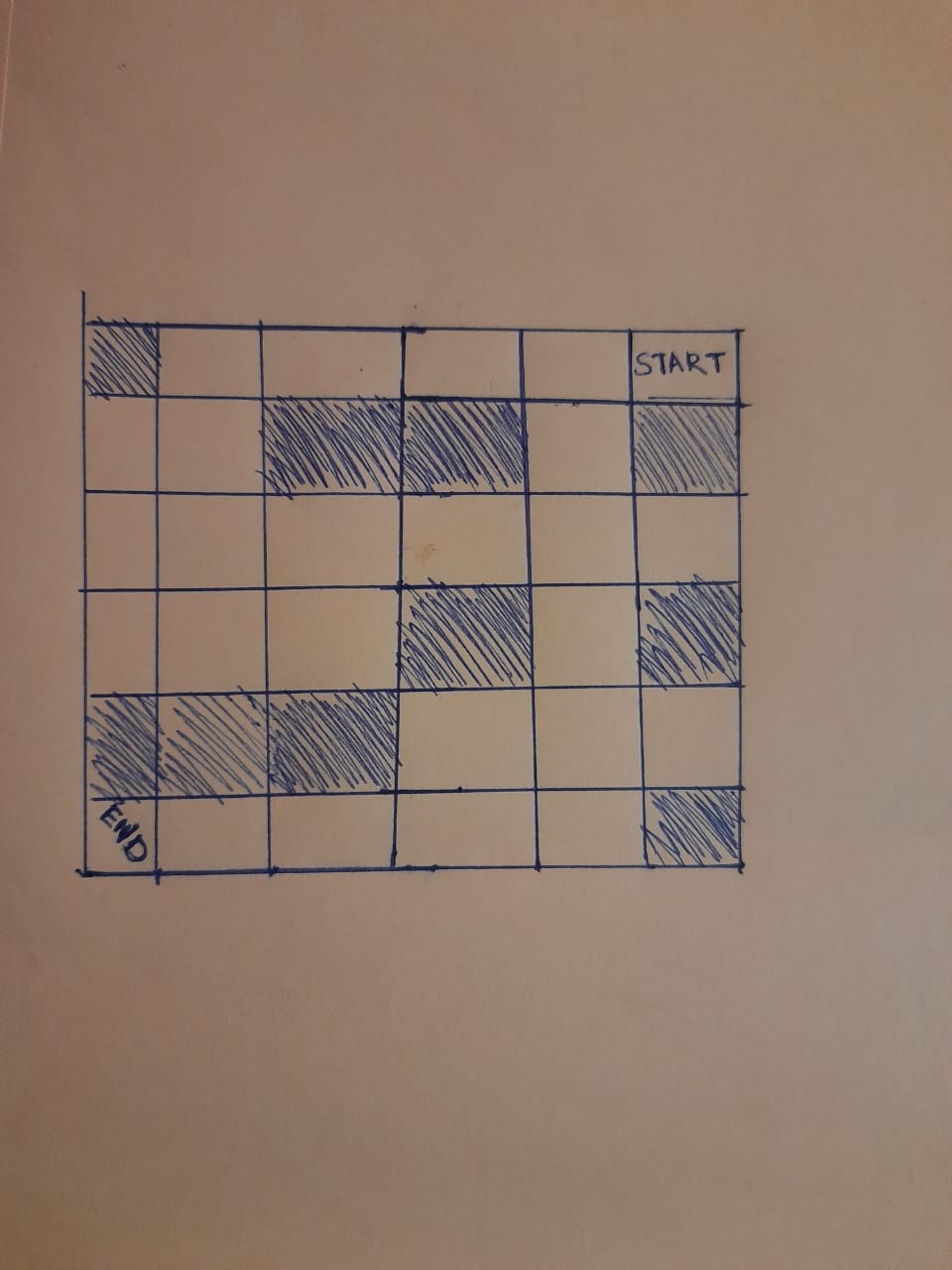
**ABSTRACT**

Inside of a Maze, there are multiple obstacles, walls boundary around the maze in which we have to reach a defined **Destination Point** from the **Start Point** dodging all the **Obstacles** coming along the path. The Main Objective is to reach the Destination using the most efficient and shortest path according to the **A-Star Algorithm**.



**INTRODUCTION**

Reaching a destination via the shortest route is a daily activity we all do. A-star (also referred to as A\*) is one of the most successful search algorithms to find the shortest path between nodes or graphs. It is an informed search algorithm, as it uses information about path **cost** and also use **heuristics** to find the solution.



For this problem, there are **Four moves (left, right, up, and down)** from a maze position provided a valid step is available. The shaded square blocks, no movement through them is allowed, (they represent obstacles) like for Start position only left motion is available since up and right move are blocked by the wall while for the down is a shaded square block (represent obstacle), thus no movement allowed is allowed downwards.

**PROPOSED METHODOLGY**

This problem is solved using the Concept of A-Star Algorithm. It is really a smart algorithm which separates it from the other conventional algorithms.   
And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

**How Do We Do This?**

We define a parameter called “Cost” for every node of the Path. Node which takes the least Cost for reaching the Destination, we jump on that that and keep continuing until we reach the destination. This way we get a path with least cost to reach the Destination from the start point.

**How Do We Implement This?**

We implemented this concept using 3 variables => *G, H and F*

*G is cost from start to current Node.*

*H is heuristic based estimated cost for current Node to end Node*

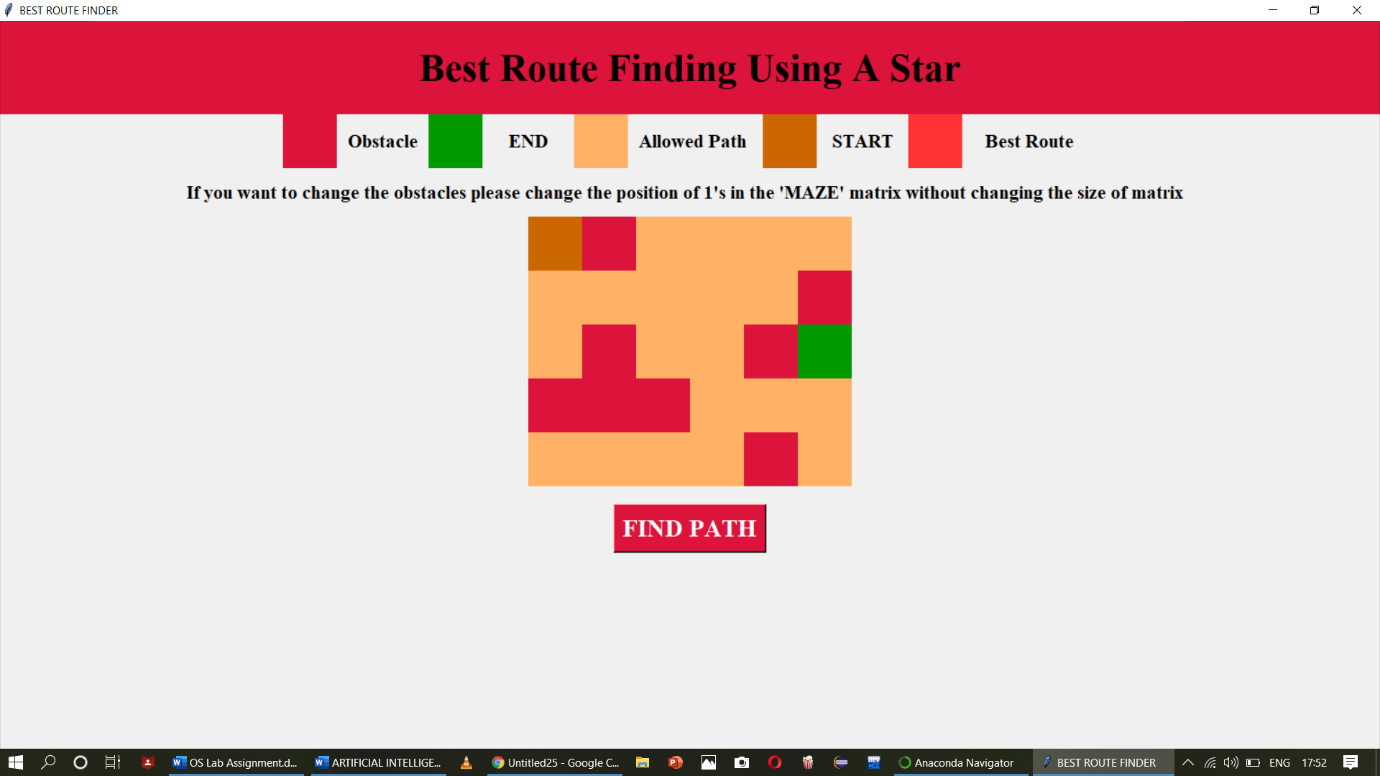
*F is total cost of present node i.e. F = G +H*

Each block inside of the maze is implemented as a Node with particular attributes like Parent, Position etc. In the problem, every node has a unique identity i.e. Each node has a different parent, different ancestor, different children, different values of F, G, H.

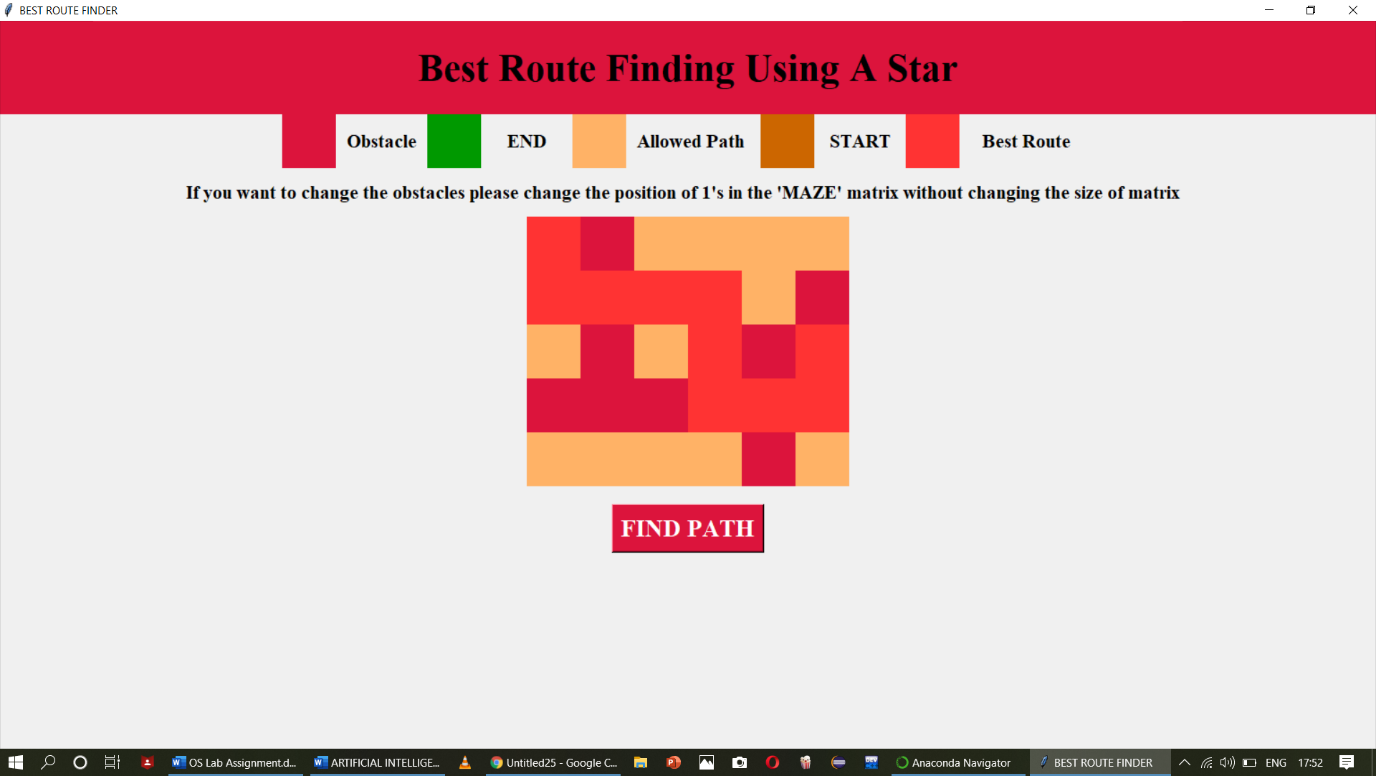
Each node can have at most 4 children and children node which has the least value of F is chosen as the next node in the path to the destination. This is continued until the whole path to reach the destination is explored.

**RESULTS**

CASE 1-: where the maze is :

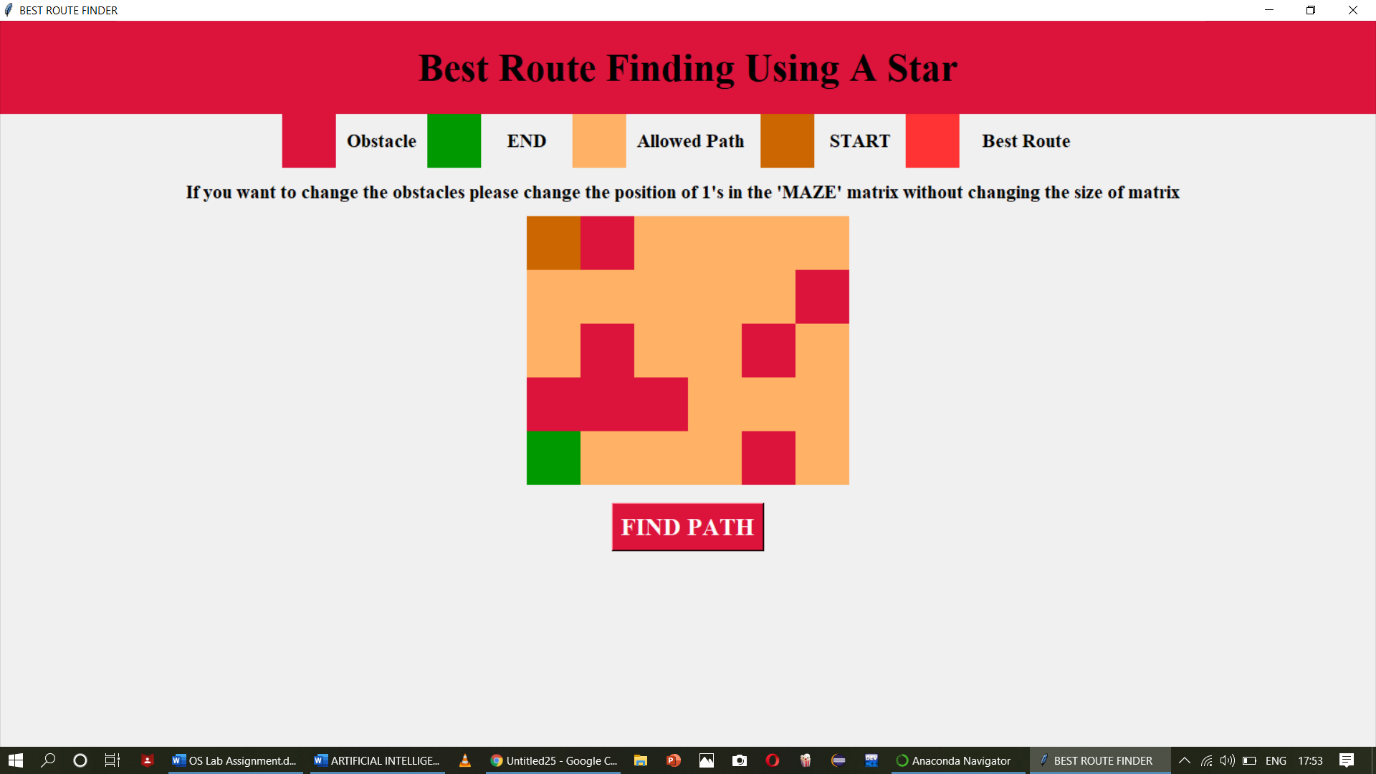
****

A-Star proposed Solution[Red color represents path]

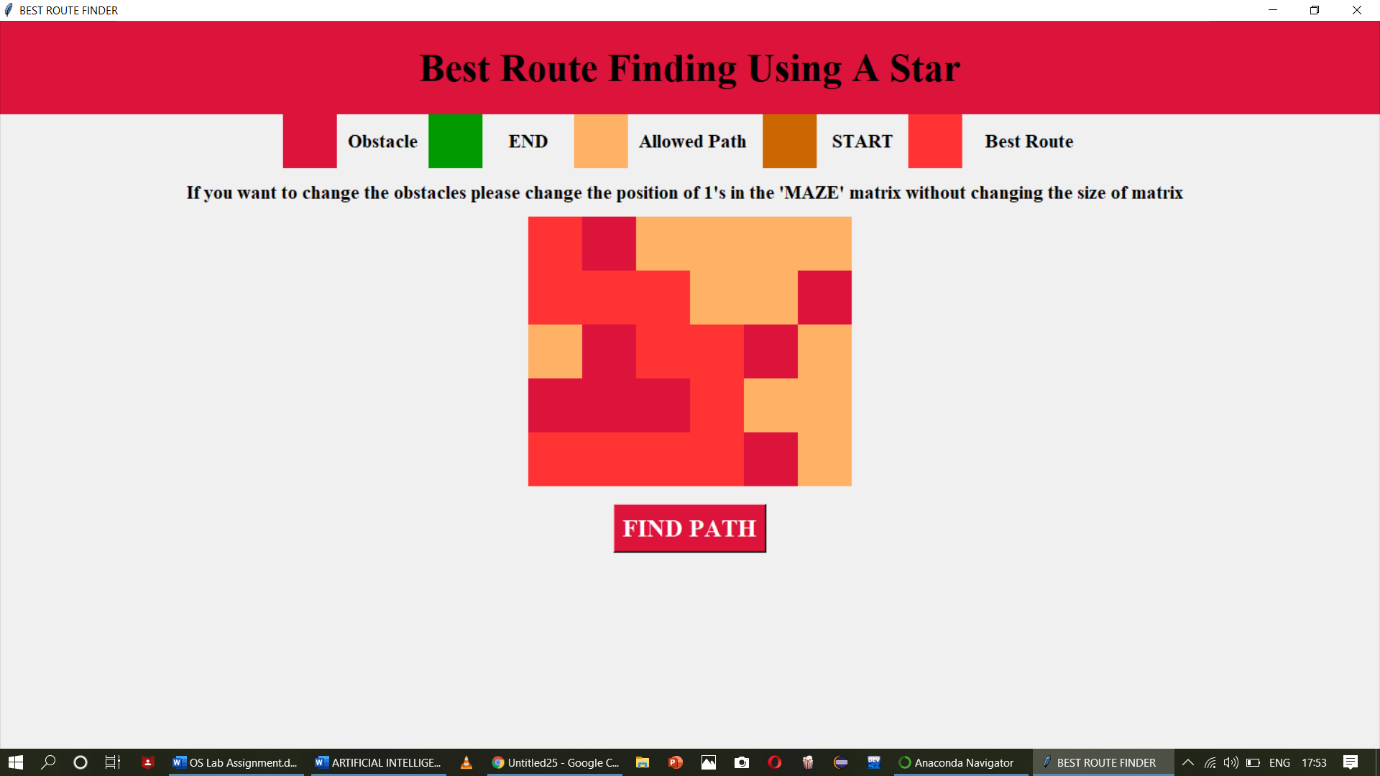
****

A-Star is successful to find out the shortest path possible to reach the Destination from Start

CASE 2 -: Maze is



A-Star proposed Solution[Red color represents path]



A-Star is successful again to find out the shortest path possible to reach the Destination from Start.

**CONCLUSION**

1. A-Star is admissible: This means that provided the solution exists, first solution provided by A-Star is the Optimal solution.
2. It is an informed search technique where we have a way to find out the approximation of each node closeness to goal, and it is much better than uninformed search techniques like BFS, DFS and the output by the A-Star technique is much Faster.
3. It is optimally efficient i.e. there is no other optimal algorithm guaranteed to expand fewer nodes than A\*.

**REFERENCES**

**Geeks for geeks**

<https://www.geeksforgeeks.org/a-search-algorithm/>

**Hacker Earth**

<https://www.hackerearth.com/practice/notes/a-search-algorithm/>

**SlideShare**

<https://www.slideshare.net/hemak15/lecture-14-heuristic-searcha-star-algorithm>

**TEXTBOOK:**

Artificial Intelligence by Kevin Knight, Elaine Rich, B. Shivashankar Nair, MC Graw Hill

Submitted By

(48)Paras Jain REG NO: 11806350

Submitted TO: Pooja Rana (47)Kiran Chavhan REG NO:11806960

(46)Adnan Raghib REG NO: 11807062