NAME : MUHAMMAD TAYYAB

ROLL NO :22F-3419

SECTION: BCS 6C

ASSIGNMENT NO 1

CODE

from typing import List, Tuple

import heapq

from collections import deque

import matplotlib.pyplot as plt

import numpy as np

import tkinter as tk

import seaborn as sns

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

class SearchAlgorithm:

@staticmethod

def get\_neighbors(x: int, y: int, grid: List[List[str]]) -> List[Tuple[int, int]]:

directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

neighbors = []

for dx, dy in directions:

nx, ny = x + dx, y + dy

if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and grid[nx][ny] != '-1':

neighbors.append((nx, ny))

return neighbors

@staticmethod

def get\_start\_target(grid: List[List[str]]) -> Tuple[Tuple[int, int], Tuple[int, int]]:

start, target = None, None

for i, row in enumerate(grid):

for j, cell in enumerate(row):

if cell == 's':

start = (i, j)

elif cell == 't':

target = (i, j)

return start, target

@staticmethod

def ucs(grid: List[List[str]]) -> Tuple[int, List[Tuple[int, int]], set]:

start, target = SearchAlgorithm.get\_start\_target(grid)

pq = [(0, start)]

visited = set()

parent = {start: None}

while pq:

cost, current = heapq.heappop(pq)

if current in visited:

continue

visited.add(current)

if current == target:

path = SearchAlgorithm.construct\_path(parent, target)

return 1, path, visited

for neighbor in SearchAlgorithm.get\_neighbors(\*current, grid):

if neighbor not in visited:

heapq.heappush(pq, (cost + 1, neighbor))

parent[neighbor] = current

return -1, [], visited

@staticmethod

def dfs(grid: List[List[str]]) -> Tuple[int, List[Tuple[int, int]], set]:

start, target = SearchAlgorithm.get\_start\_target(grid)

stack = [start]

visited = set()

parent = {start: None}

while stack:

current = stack.pop()

if current in visited:

continue

visited.add(current)

if current == target:

path = SearchAlgorithm.construct\_path(parent, target)

return 1, path, visited

for neighbor in SearchAlgorithm.get\_neighbors(\*current, grid):

if neighbor not in visited:

stack.append(neighbor)

parent[neighbor] = current

return -1, [], visited

@staticmethod

def bfs(grid: List[List[str]]) -> Tuple[int, List[Tuple[int, int]], set]:

start, target = SearchAlgorithm.get\_start\_target(grid)

if not start or not target:

return -1, [], set()

queue = deque([start])

visited = set([start])

parent = {start: None}

while queue:

current = queue.popleft()

if current == target:

path = SearchAlgorithm.construct\_path(parent, target)

return 1, path, visited

for neighbor in SearchAlgorithm.get\_neighbors(\*current, grid):

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

parent[neighbor] = current

return -1, [], visited

@staticmethod

def best\_first\_search(grid: List[List[str]]) -> Tuple[int, List[Tuple[int, int]], set]:

start, target = SearchAlgorithm.get\_start\_target(grid)

pq = [(0, start)]

visited = set()

parent = {start: None}

while pq:

\_, current = heapq.heappop(pq)

if current in visited:

continue

visited.add(current)

if current == target:

path = SearchAlgorithm.construct\_path(parent, target)

return 1, path, visited

for neighbor in SearchAlgorithm.get\_neighbors(\*current, grid):

if neighbor not in visited:

heapq.heappush(pq, (SearchAlgorithm.manhattan\_distance(neighbor, target), neighbor))

parent[neighbor] = current

return -1, [], visited

@staticmethod

def a\_star(grid: List[List[str]]) -> Tuple[int, List[Tuple[int, int]], set]:

start, target = SearchAlgorithm.get\_start\_target(grid)

pq = [(0, start)]

g\_cost = {start: 0}

parent = {start: None}

visited = set()

while pq:

\_, current = heapq.heappop(pq)

if current in visited:

continue

visited.add(current)

if current == target:

path = SearchAlgorithm.construct\_path(parent, target)

return 1, path, visited

for neighbor in SearchAlgorithm.get\_neighbors(\*current, grid):

new\_g\_cost = g\_cost[current] + 1

if neighbor not in g\_cost or new\_g\_cost < g\_cost[neighbor]:

g\_cost[neighbor] = new\_g\_cost

f\_cost = new\_g\_cost + SearchAlgorithm.manhattan\_distance(neighbor, target)

heapq.heappush(pq, (f\_cost, neighbor))

parent[neighbor] = current

return -1, [], visited

@staticmethod

def construct\_path(parent, target):

path = []

current = target

while current:

path.append(current)

current = parent[current]

return path[::-1]

@staticmethod

def manhattan\_distance(a, b):

return abs(a[0] - b[0]) + abs(a[1] - b[1])

class SearchVisualizerApp:

def \_init(self, root):

self.root = root

self.root.title("Search Algorithm Visualizer")

self.grid = [

['5', '7', '8', '9'],

['3', '-1', '-1', 't'],

['s', '1', '-1', '0'],

['2', '4', '6', '-1']

]

self.algo\_var = tk.StringVar(value="BFS")

algorithms = ["BFS", "DFS", "UCS", "Best First Search", "A\*"]

self.algo\_menu = tk.OptionMenu(root, self.algo\_var, \*algorithms)

self.algo\_menu.pack(pady=10)

self.run\_button = tk.Button(root, text="Run Algorithm", command=self.run\_algorithm)

self.run\_button.pack(pady=10)

self.figure, self.ax = plt.subplots(figsize=(6, 6))

self.canvas = FigureCanvasTkAgg(self.figure, master=root)

self.canvas.get\_tk\_widget().pack()

def run\_algorithm(self):

algo = self.algo\_var.get()

if algo == "BFS":

found, path, visited = SearchAlgorithm.bfs(self.grid)

elif algo == "DFS":

found, path, visited = SearchAlgorithm.dfs(self.grid)

elif algo == "UCS":

found, path, visited = SearchAlgorithm.ucs(self.grid)

elif algo == "Best First Search":

found, path, visited = SearchAlgorithm.best\_first\_search(self.grid)

elif algo == "A\*":

found, path, visited = SearchAlgorithm.a\_star(self.grid)

else:

return

grid\_output = [[0 for \_ in row] for row in self.grid]

for x, y in visited:

grid\_output[x][y] = 1

for x, y in path:

grid\_output[x][y] = 2

heatmap\_data = np.array(grid\_output)

cmap = plt.cm.colors.ListedColormap(['white', 'lightpink', 'blue'])

self.ax.clear()

sns.heatmap(

heatmap\_data,

annot=np.array(self.grid),

cmap=cmap,

cbar=False,

linewidths=2,

linecolor='black',

fmt='',

ax=self.ax

)

self.ax.set\_title(f"{algo} Path Visualization")

self.canvas.draw()

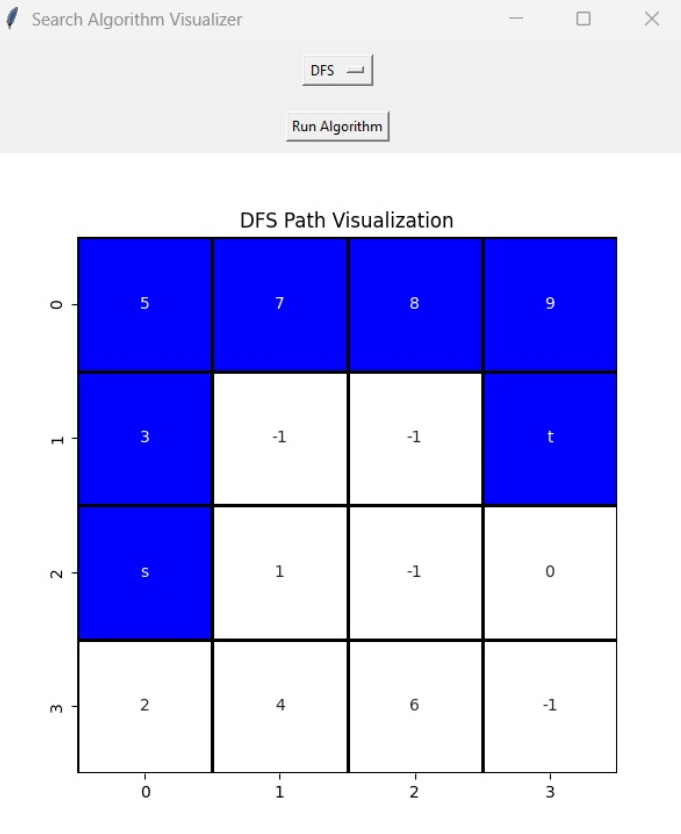
if \_name\_ == "\_main\_":

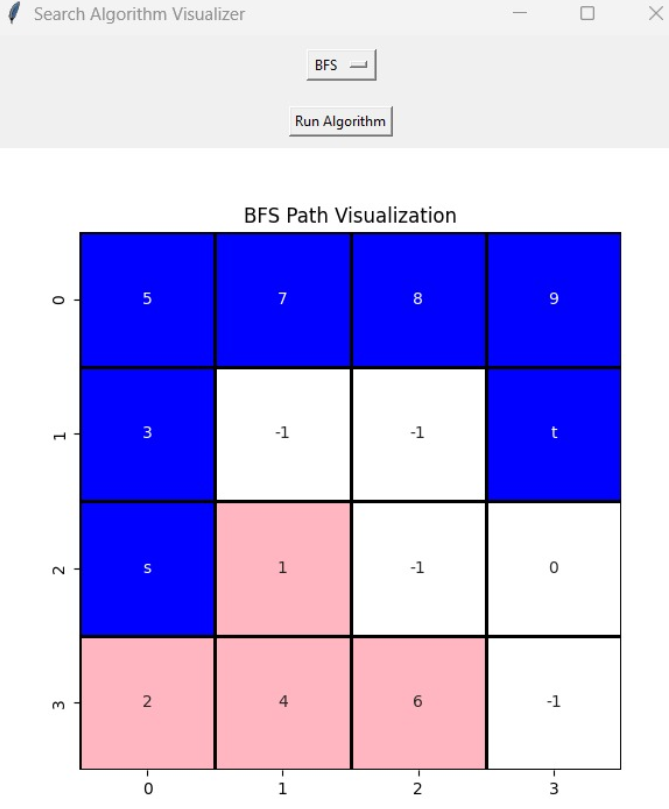
root = tk.Tk()

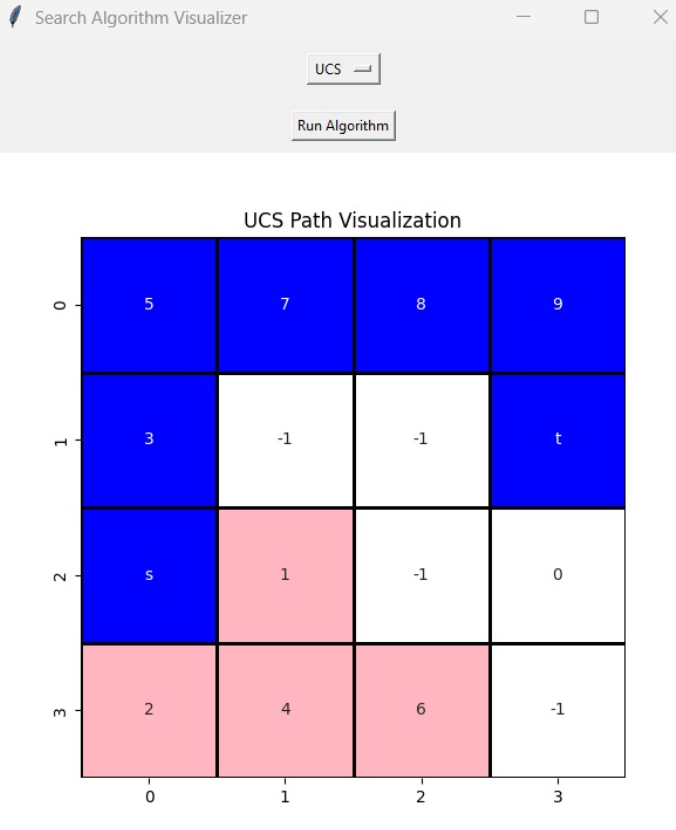
app = SearchVisualizerApp(root)

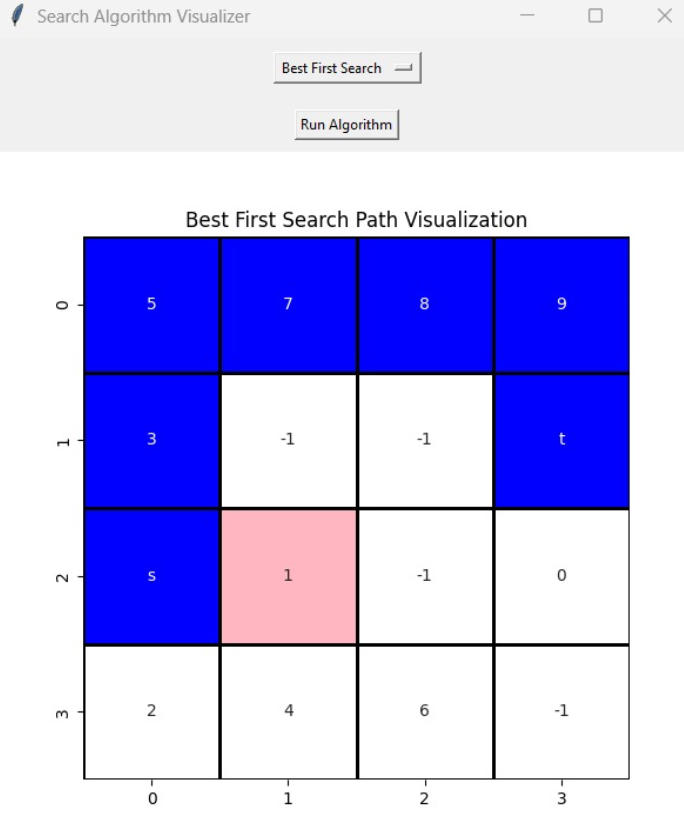
root.mainloop()

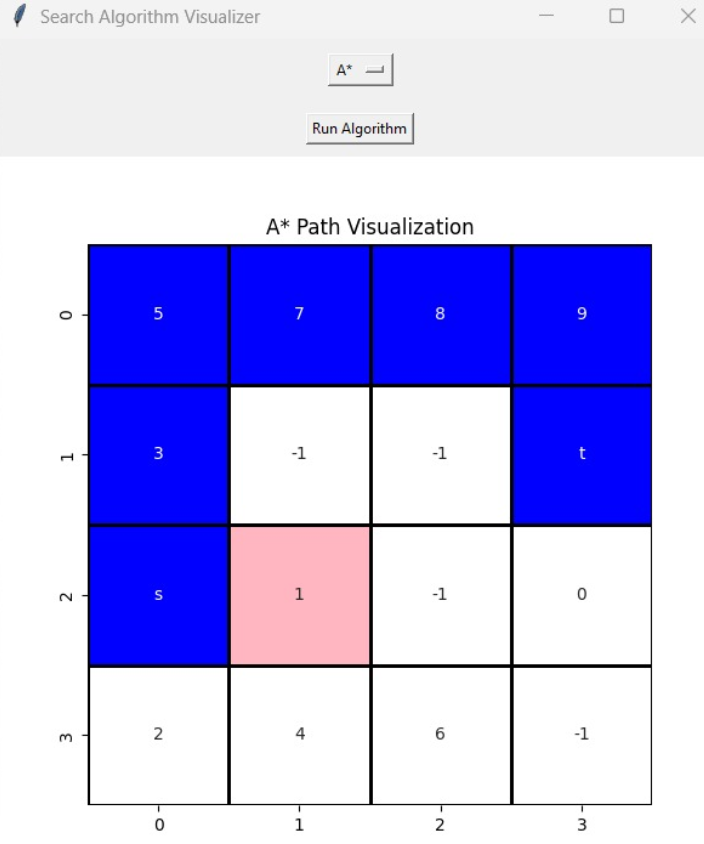
**OUTPUT**

****

****

****

****

****