

AI\_A1\_22F-3850

Asad Raza



March 3, 2025

ASad RAZA

FAST NUCES

# Code:

import heapq  
from typing import List, Tuple  
import seaborn as sb  
import numpy as np  
import matplotlib.pyplot as plt  
  
  
class SearchAlgorithm:  
 directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  
  
 @staticmethod  
 def get\_neighbors(x: int, y: int, grid: List[List[str]], wall: set) -> List[Tuple[int, int]]:  
 row = len(grid)  
 col = len(grid[0])  
 neighbor = []  
 for dx, dy in SearchAlgorithm.directions:  
 nx = x + dx  
 ny = y + dy  
 if 0 <= nx < row and 0 <= ny < col and (nx, ny) not in wall:  
 neighbor.append((nx, ny))  
 return neighbor  
  
 @staticmethod  
 def get\_start\_target(grid: List[List[str]]) -> Tuple[Tuple[int, int], Tuple[int, int], set]:  
 start = target = (-1, -1)  
 wall = set()  
 for i, row in enumerate(grid):  
 for j, cell in enumerate(row):  
 if cell == 's':  
 start = (i, j)  
 elif cell == 't':  
 target = (i, j)  
 elif cell == '-1':  
 wall.add((i, j))  
 return start, target, wall  
  
 @staticmethod  
 def reconstruct\_path(parent, curr):  
 path = []  
 while curr:  
 path.append(curr)  
 curr = parent[curr]  
 return path[::-1]  
  
 @staticmethod  
 def dfs(grid: List[List[str]]) -> Tuple[int, List[Tuple[int, int]]]:  
 result = SearchAlgorithm.get\_start\_target(grid)  
 start = result[0]  
 target = result[1]  
 wall = result[2]  
  
 stack, visited = [start], set()  
 parent = {start: None}  
  
 while stack:  
 curr = stack.pop()  
 if curr == target:  
 return 1, SearchAlgorithm.reconstruct\_path(parent, curr)  
 if curr in visited:  
 continue  
 visited.add(curr)  
 for neighbor in reversed(SearchAlgorithm.get\_neighbors(\*curr, grid, wall)):  
 if neighbor not in visited:  
 stack.append(neighbor)  
 parent[neighbor] = curr  
 return -1, []  
  
 @staticmethod  
 def ucs(grid: List[List[str]]) -> Tuple[int, List[Tuple[int, int]]]:  
 start, target, wall = SearchAlgorithm.get\_start\_target(grid)  
 priority\_queue = [(0, start)]  
 visited = set()  
 parent = {start: None}  
  
 while priority\_queue:  
 cost, curr = heapq.heappop(priority\_queue)  
 if curr == target:  
 return 1, SearchAlgorithm.reconstruct\_path(parent, curr)  
 if curr in visited:  
 continue  
 visited.add(curr)  
 for neighbor in SearchAlgorithm.get\_neighbors(\*curr, grid, wall):  
 if neighbor not in visited:  
 neighbor\_cost = int(grid[neighbor[0]][neighbor[1]]) if grid[neighbor[0]][  
 neighbor[1]].isdigit() else 1  
 heapq.heappush(priority\_queue, (cost + neighbor\_cost, neighbor))  
 parent[neighbor] = curr  
 return -1, []  
  
 @staticmethod  
 def best\_first\_search(grid: List[List[str]]) -> Tuple[int, List[Tuple[int, int]]]:  
 start, target, wall = SearchAlgorithm.get\_start\_target(grid)  
 priority\_queue = [(SearchAlgorithm.manhattanDistance(start, target), start)]  
 visited = set()  
 parent = {start: None}  
  
 while priority\_queue:  
 item = heapq.heappop(priority\_queue)  
 curr = item[1]  
  
 if curr == target:  
 return 1, SearchAlgorithm.reconstruct\_path(parent, curr)  
 if curr in visited:  
 continue  
 visited.add(curr)  
 for neighbor in SearchAlgorithm.get\_neighbors(\*curr, grid, wall):  
 if neighbor not in visited:  
 priority = SearchAlgorithm.manhattanDistance(neighbor, target)  
 heapq.heappush(priority\_queue, (priority, neighbor))  
 parent[neighbor] = curr  
 return -1, []  
  
 @staticmethod  
 def manhattanDistance(x, y):  
 return abs(x[0] - y[0]) + abs(x[1] - y[1])  
  
 @staticmethod  
 def a\_star(grid: List[List[str]]) -> Tuple[int, List[Tuple[int, int]]]:  
 start, target, wall = SearchAlgorithm.get\_start\_target(grid)  
 open\_set = [(SearchAlgorithm.manhattanDistance(start, target), 0, start)]  
 parent, totalCost = {start: None}, {start: 0}  
 while open\_set:  
 \_, curr\_cost, curr = heapq.heappop(open\_set)  
 if curr == target:  
 return 1, SearchAlgorithm.reconstruct\_path(parent, curr)  
 for neighbor in SearchAlgorithm.get\_neighbors(\*curr, grid, wall):  
 neighbor\_cost = int(grid[neighbor[0]][neighbor[1]]) if grid[neighbor[0]][neighbor[1]].isdigit() else 1  
 new\_cost = curr\_cost + neighbor\_cost  
 if neighbor not in totalCost or new\_cost < totalCost[neighbor]:  
 totalCost[neighbor] = new\_cost  
 priority = new\_cost + SearchAlgorithm.manhattanDistance(neighbor, target)  
 heapq.heappush(open\_set, (priority, new\_cost, neighbor))  
 parent[neighbor] = curr  
 return -1, []  
  
 @staticmethod  
 def bfs(grid: List[List[str]]) -> Tuple[int, List[Tuple[int, int]]]:  
 start, target, wall = SearchAlgorithm.get\_start\_target(grid)  
 queue, visited = [(start, [start])], {start}  
 while queue:  
 curr, path = queue.pop(0)  
 if curr == target:  
 return 1, path  
 for neighbor in SearchAlgorithm.get\_neighbors(\*curr, grid, wall):  
 if neighbor not in visited:  
 visited.add(neighbor)  
 queue.append((neighbor, path + [neighbor]))  
 return -1, []  
  
 @staticmethod  
 def visualize\_grid(grid: List[List[str]], path: List[Tuple[int, int]] = [], color='red', title="Search"):  
 grid\_map = np.array([[int(cell) if cell not in ['s', 't'] else 0 for cell in row] for row in grid])  
 plt.figure(figsize=(7, 5))  
 plt.title(title)  
 sb.heatmap(grid\_map, annot=True, cmap='Greys', cbar=False, linewidths=.5, linecolor='black')  
 for (x, y) in path:  
 plt.plot(y + 0.5, x + 0.5, 'o', color=color)  
 plt.show()  
  
  
filename=input("Enter File Name that you want run :")  
with open(filename, 'r') as file:  
 grid = [line.strip().split() for line in file]  
  
algos = {  
 "BFS": (SearchAlgorithm.bfs, 'blue'),  
 "DFS": (SearchAlgorithm.dfs, 'green'),  
 "UCS": (SearchAlgorithm.ucs, 'purple'),  
 "A\* Search": (SearchAlgorithm.a\_star, 'orange'),  
 "Best First Search": (SearchAlgorithm.best\_first\_search, 'red')  
}  
  
for name, (func, color) in algos.items():  
 found, path = func(grid)  
 if found == 1:  
 print(f"\n--------------- {name} --------------------")  
 print(f"Path Found With {name}...")  
 print("-->", path)  
 SearchAlgorithm.visualize\_grid(grid, path, color,name)  
 else:  
 print(f"There is No Path Found With {name}...")

# Screenshot (BFS):

A grid of squares with black and white squares

AI-generated content may be incorrect.

# Screenshot (DFS):

A screenshot of a computer screen

AI-generated content may be incorrect.

# Screenshot (UCS):

A grid of squares with different colors

AI-generated content may be incorrect.

# Screenshot (A\* Search):

A grid of squares with black and white squares

AI-generated content may be incorrect.

# Screenshot (Best First Search):

A grey squares with red dots

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.