#Bài 1

import heapq

# Define the graph structure

adjac\_lis = {

'A': [('B', 5), ('C', 1), ('D', 10)],

'B': [('D', 5)],

'C': [('D', 5)],

'D': []

}

# Define the heuristic values

H = {

'A': 1,

'B': 1,

'C': 1,

'D': 1

}

def a\_star\_algorithm(start\_node, stop\_node):

open\_list = set([start\_node])

closed\_list = set([])

g = {} # Store distance from starting node

parents = {} # Parents map to reconstruct path

# Distance of starting node from itself is zero

g[start\_node] = 0

# Start node has no parent

parents[start\_node] = start\_node

while len(open\_list) > 0:

n = None

# Node with lowest f() value

for v in open\_list:

if n == None or g[v] + H[v] < g[n] + H[n]:

n = v

if n == None:

print('Path does not exist!')

return None

# If the current node is the target node

if n == stop\_node:

reconstruct\_path = []

while parents[n] != n:

reconstruct\_path.append(n)

n = parents[n]

reconstruct\_path.append(start\_node)

reconstruct\_path.reverse()

print('Path found: {}'.format(reconstruct\_path))

return reconstruct\_path

# For all the neighbors of the current node

for (m, weight) in adjac\_lis[n]:

# If the current node is not present in both open\_list and closed\_list

# add it to open\_list and record the parent

if m not in open\_list and m not in closed\_list:

open\_list.add(m)

parents[m] = n

g[m] = g[n] + weight

# Otherwise, check if it's quicker to first visit n, then m

else:

if g[m] > g[n] + weight:

# Update g

g[m] = g[n] + weight

# Change parent

parents[m] = n

# If the node is in the closed list, move it to open list

if m in closed\_list:

closed\_list.remove(m)

open\_list.add(m)

# Remove n from the open list and add it to the closed list

open\_list.remove(n)

closed\_list.add(n)

print('Path does not exist!')

return None

# Execute A\* algorithm from node A to D

a\_star\_algorithm('A', 'D')

#Bài 2

import heapq

class Graph:

def \_\_init\_\_(self):

self.adjacency\_list = {}

self.heuristic = {}

def add\_edge(self, from\_node, to\_node, weight):

if from\_node not in self.adjacency\_list:

self.adjacency\_list[from\_node] = []

self.adjacency\_list[from\_node].append((to\_node, weight))

def set\_heuristic(self, node, value):

self.heuristic[node] = value

def searchA\_star(self, start\_node, goal\_node):

open\_list = set([start\_node])

closed\_list = set([])

g = {start\_node: 0}

parents = {start\_node: start\_node}

pq = [(self.heuristic[start\_node], start\_node)]

while pq:

\_, current = heapq.heappop(pq)

if current == goal\_node:

path = []

while parents[current] != current:

path.append(current)

current = parents[current]

path.append(start\_node)

path.reverse()

return path, g[goal\_node]

open\_list.remove(current)

closed\_list.add(current)

for (neighbor, weight) in self.adjacency\_list.get(current, []):

if neighbor in closed\_list:

continue

tentative\_g = g[current] + weight

if neighbor not in open\_list or tentative\_g < g[neighbor]:

open\_list.add(neighbor)

g[neighbor] = tentative\_g

parents[neighbor] = current

f = tentative\_g + self.heuristic[neighbor]

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

return None, float('inf')

# Create the graph

graph = Graph()

# Add edges

edges = [

('Arad', 'Zerind', 75), ('Arad', 'Sibiu', 140), ('Arad', 'Timisoara', 118),

('Zerind', 'Oradea', 71), ('Oradea', 'Sibiu', 151), ('Timisoara', 'Lugoj', 111),

('Lugoj', 'Mehadia', 70), ('Mehadia', 'Dobreta', 75), ('Dobreta', 'Craiova', 120),

('Craiova', 'Rimnicu Vilcea', 146), ('Craiova', 'Pitesti', 138),

('Rimnicu Vilcea', 'Sibiu', 80), ('Rimnicu Vilcea', 'Pitesti', 97),

('Pitesti', 'Bucharest', 101), ('Bucharest', 'Giurgiu', 90),

('Bucharest', 'Urziceni', 85), ('Urziceni', 'Hirsova', 98),

('Hirsova', 'Eforie', 86), ('Urziceni', 'Vaslui', 142), ('Vaslui', 'Iasi', 92),

('Iasi', 'Neamt', 87), ('Sibiu', 'Fagaras', 99), ('Fagaras', 'Bucharest', 211)

]

for edge in edges:

graph.add\_edge(edge[0], edge[1], edge[2])

graph.add\_edge(edge[1], edge[0], edge[2])

# Set heuristic values

heuristics = {

'Arad': 366, 'Bucharest': 0, 'Craiova': 160, 'Dobreta': 242,

'Eforie': 161, 'Fagaras': 178, 'Giurgiu': 77, 'Hirsova': 151,

'Iasi': 226, 'Lugoj': 244, 'Mehadia': 241, 'Neamt': 234,

'Oradea': 380, 'Pitesti': 98, 'Rimnicu Vilcea': 193, 'Sibiu': 253,

'Timisoara': 329, 'Urziceni': 80, 'Vaslui': 199, 'Zerind': 374

}

for node, h\_value in heuristics.items():

graph.set\_heuristic(node, h\_value)

# Perform A\* search

path, cost = graph.searchA\_star('Arad', 'Bucharest')

print(f"Path: {path}")

print(f"Total cost: {cost}")

#Bài 3

import heapq

class Water:

def \_\_init\_\_(self, filename):

self.graph = self.read\_file(filename)

def read\_file(self, filename):

graph = {}

with open(filename, 'r') as file:

lines = file.readlines()

for line in lines:

parts = line.split()

if len(parts) == 3:

node1, node2, weight = parts[0], parts[1], float(parts[2])

if node1 not in graph:

graph[node1] = []

if node2 not in graph:

graph[node2] = []

graph[node1].append((node2, weight))

graph[node2].append((node1, weight))

return graph

def heuristic(self, node1, node2):

# Example heuristic (placeholder, can be replaced with actual distance calculation)

return 0

def a\_star\_search(self, start, goal):

open\_set = []

heapq.heappush(open\_set, (0, start))

came\_from = {}

g\_score = {node: float('inf') for node in self.graph}

g\_score[start] = 0

f\_score = {node: float('inf') for node in self.graph}

f\_score[start] = self.heuristic(start, goal)

while open\_set:

\_, current = heapq.heappop(open\_set)

if current == goal:

return self.reconstruct\_path(came\_from, current)

for neighbor, weight in self.graph[current]:

tentative\_g\_score = g\_score[current] + weight

if tentative\_g\_score < g\_score[neighbor]:

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g\_score

f\_score[neighbor] = g\_score[neighbor] + self.heuristic(neighbor, goal)

heapq.heappush(open\_set, (f\_score[neighbor], neighbor))

return None

def reconstruct\_path(self, came\_from, current):

total\_path = [current]

while current in came\_from:

current = came\_from[current]

total\_path.append(current)

total\_path.reverse()

return total\_path

def calculate\_total\_weight(self, path):

total\_weight = 0

for i in range(len(path) - 1):

for neighbor, weight in self.graph[path[i]]:

if neighbor == path[i + 1]:

total\_weight += weight

break

return total\_weight

# Usage example:

water\_system = Water('/mnt/data/image.png') # Path to your file

start\_node = 'A' # Replace with your actual start node

goal\_node = 'B' # Replace with your actual goal node

path = water\_system.a\_star\_search(start\_node, goal\_node)

if path:

print("Path found:", path)

total\_weight = water\_system.calculate\_total\_weight(path)

print("Total weight of the path:", total\_weight)

else:

print("No path found.")