1 a)Python Code for Breadth First Search algorithm

from collections import deque

def bfs(graph, start):

visited = set()

queue = deque([start])

while queue:

vertex = queue.popleft()

if vertex not in visited:

visited.add(vertex)

print(vertex)

neighbors = graph[vertex]

for neighbor in neighbors:

if neighbor not in visited:

queue.append(neighbor)

graph = {

'A': ['B', 'C'],

'B': ['A', 'D', 'E'],

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

'F': ['C', 'E']

}

start\_vertex = 'A'

bfs(graph, start\_vertex)

**1b) Python code for Iterative Depth First Search algorithm**

**from collections import defaultdict**

class Graph:

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

self.graph = defaultdict(list)

def add\_edge(self, u, v):

self.graph[u].append(v)

self.graph[v].append(u)

def iterative\_dfs(self, start, end):

if start == end:

return [start]

visited = set()

stack = [(start, [start])]

while stack:

current\_vertex, path = stack.pop()

visited.add(current\_vertex)

for neighbor in self.graph[current\_vertex]:

if neighbor not in visited:

if neighbor == end:

return path + [neighbor]

stack.append((neighbor, path + [neighbor]))

return None

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

g = Graph()

g.add\_edge(1, 2)

g.add\_edge(1, 3)

g.add\_edge(2, 4)

g.add\_edge(2, 5)

g.add\_edge(3, 6)

g.add\_edge(3, 7)

g.add\_edge(4, 8)

g.add\_edge(4, 9)

g.add\_edge(5, 10)

g.add\_edge(5, 11)

g.add\_edge(6, 12)

g.add\_edge(6, 13)

g.add\_edge(7, 14)

g.add\_edge(7, 15)

start\_node = 1

end\_node = 9

shortest\_path = g.iterative\_dfs(start\_node, end\_node)

if shortest\_path:

print(f"Shortest path from {start\_node} to {end\_node}: {shortest\_path}")

else:

print(f"No path found from {start\_node} to {end\_node}")

**2) A\* Search** and **Recursive Best-First Search**

import heapq

romania\_map = {

'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},

'Zerind': {'Arad': 75, 'Oradea': 71},

'Timisoara': {'Arad': 118, 'Lugoj': 111},

'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},

'Oradea': {'Zerind': 71, 'Sibiu': 151},

'Lugoj': {'Timisoara': 111, 'Mehadia': 70},

'Fagaras': {'Sibiu': 99, 'Bucharest': 211},

'Rimnicu Vilcea': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},

'Mehadia': {'Lugoj': 70, 'Drobeta': 75},

'Drobeta': {'Mehadia': 75, 'Craiova': 120},

'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},

'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},

'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},

'Giurgiu': {'Bucharest': 90},

'Urziceni': {'Bucharest': 85, 'Hirsova': 98, 'Vaslui': 142},

'Hirsova': {'Urziceni': 98, 'Eforie': 86},

'Eforie': {'Hirsova': 86},

'Vaslui': {'Urziceni': 142, 'Iasi': 92},

'Iasi': {'Vaslui': 92, 'Neamt': 87},

'Neamt': {'Iasi': 87}

}

class Node:

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

self.city = city

self.cost = cost

self.parent = parent

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

return self.cost < other.cost

def heuristic(city, goal):

# Using a simple heuristic (0 for now, but could be improved)

return 0

def astar\_search(graph, start, goal):

open\_list = []

closed\_set = set()

heapq.heappush(open\_list, start)

while open\_list:

current\_node = heapq.heappop(open\_list)

if current\_node.city == goal.city:

path = []

while current\_node:

path.append(current\_node.city)

current\_node = current\_node.parent

return path[::-1] # Reverse to get the correct order

closed\_set.add(current\_node.city)

for neighbor, distance in graph[current\_node.city].items():

if neighbor not in closed\_set:

new\_cost = current\_node.cost + distance + heuristic(neighbor, goal.city)

new\_node = Node(neighbor, new\_cost, current\_node)

heapq.heappush(open\_list, new\_node)

return None

start\_city = 'Arad'

goal\_city = 'Bucharest'

start\_node = Node(start\_city, 0)

goal\_node = Node(goal\_city, 0)

path = astar\_search(romania\_map, start\_node, goal\_node)

if path:

print("Path found:", path)

else:

print("No path found")