**COMSATS UNVERISTY ISLAMABAD**



**Artificial Intelligence**

**Lab 4**

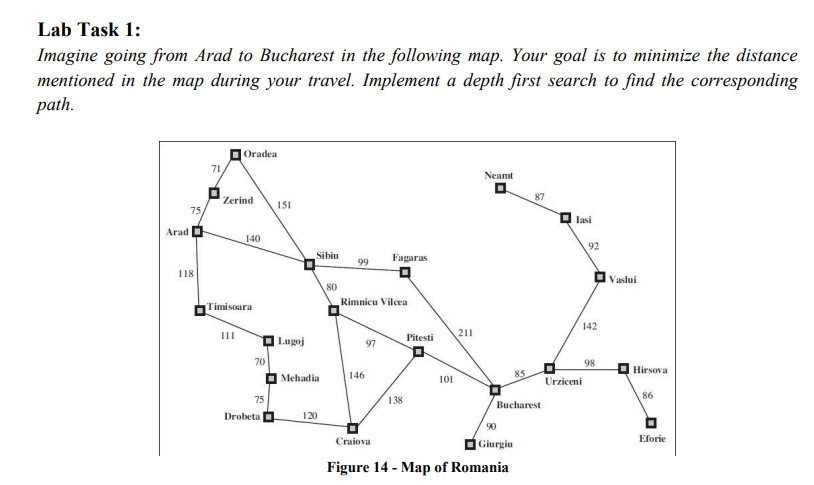
**Submitted by:**

Hasaan Ahmad SP22-BSE-017

**Submitted to:**

**Sir Waqas Ali**

**Lab 1:**



# SP22-BSE-017 HASAAN AHMAD

class Node:

    def \_\_init\_\_(self, name, neighbors=None):

        self.name = name

        self.neighbors = neighbors if neighbors else []

        self.visited = False

graph = {

    'Arad': Node('Arad', [('Zerind', 75), ('Sibiu', 140), ('Timisoara', 118)]),

    'Bucharest': Node('Bucharest', [('Giurgiu', 85), ('Pitesti', 211), ('Urziceni', 98)]),

    'Craiova': Node('Craiova', [('Drobeta', 120), ('Rimnicu Vilcea', 146), ('Pitesti', 138)]),

    'Drobeta': Node('Drobeta', [('Mehadia', 80)]),

    'Eforie': Node('Eforie'),

    'Fagaras': Node('Fagaras', [('Sibiu', 99), ('Bucharest', 211)]),

    'Giurgiu': Node('Giurgiu', [('Bucharest', 90)]),

    'Hirsova': Node('Hirsova', [('Urziceni', 98)]),

    'Iasi': Node('Iasi', [('Neamt', 87)]),

    'Lugoj': Node('Lugoj', [('Mehadia', 70)]),

    'Mehadia': Node('Mehadia', [('Lugoj', 75), ('Drobeta', 151)]),

    'Neamt': Node('Neamt', [('Iasi', 92)]),

    'Oradea': Node('Oradea', [('Zerind', 140)]),

    'Pitesti': Node('Pitesti', [('Rimnicu Vilcea', 97), ('Craiova', 138), ('Bucharest', 101)]),

    'Rimnicu Vilcea': Node('Rimnicu Vilcea', [('Sibiu', 80), ('Pitesti', 97), ('Craiova', 146)]),

    'Sibiu': Node('Sibiu', [('Fagaras', 99), ('Rimnicu Vilcea', 80), ('Arad', 140), ('Oradea', 151)]),

    'Timisoara': Node('Timisoara', [('Arad', 118)]),

    'Urziceni': Node('Urziceni', [('Hirsova', 86), ('Bucharest', 98), ('Vaslui', 142)]),

    'Vaslui': Node('Vaslui', [('Urziceni', 98), ('Iasi', 92)]),

    'Zerind': Node('Zerind', [('Oradea', 71), ('Arad', 75)])

}

def DFS(graph, initialstate, goalstate):

    frontier = [initialstate]

    explored = []

    while frontier:

        currentNode = frontier.pop()

        explored.append(currentNode)

        if currentNode == goalstate:

            return actionSequence(graph, initialstate, goalstate)

        for child in graph[currentNode].neighbors:

            if child[0] not in frontier and child[0] not in explored:

                graph[child[0]].parent = currentNode

                frontier.append(child[0])

def actionSequence(graph, initialstate, goalstate):

    solution = [goalstate]

    currentParent = graph[goalstate].parent

    while currentParent != initialstate:

        solution.append(currentParent)

        currentParent = graph[currentParent].parent

    solution.append(initialstate)

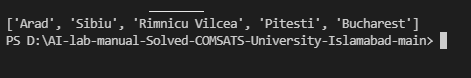
    solution.reverse()

    return solution

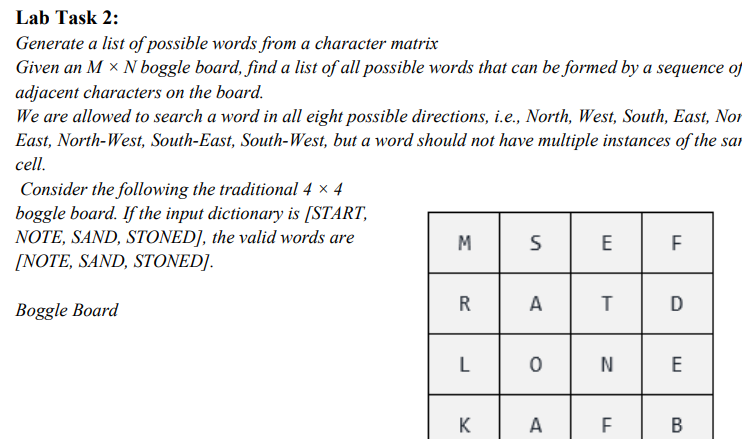
initialstate = 'Arad'

print(DFS(graph, initialstate, 'Bucharest'))

**Output:**



**Lab 2:**



**Approach 1:**

Using no data structure like Trie but multiple nested loops. It Increases the time complexity of program but gives right output. Code is as follows:

board = [

    ['M','S','E','F'],

    ['R','A','T','D'],

    ['L','O','N','E'],

    ['K','A','F','B']

]

dictionary = ['START','NOTE','SAND','STONED']

def find\_word(board, word,visited,x,y,result):

    if word in dictionary:

        result.add(word)

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

    for direction in directions:

        dx,dy = direction

        if 0 <= x+dx < 4 and 0 <= y+dy < 4 and not visited[x+dx][y+dy]:

            visited[x+dx][y+dy] = True

            find\_word(board,word+board[x+dx][y+dy],visited,x+dx,y+dy,result)

            visited[x+dx][y+dy] = False

    return result

def find\_all\_words(board):

    result = set()

    for i in range(4):

        for j in range(4):

            visited = [[False]\*4 for \_ in range(4)]

            visited[i][j] = True

            result = find\_word(board,board[i][j],visited,i,j,result)

    return result

print(find\_all\_words(board))

The time complexity for given program is O(4n)

**Output:**



**Approach 2:**

This uses a data structure like Trie which reduces the time complexity to almost 0ms as it makes the Trie using the board according to the words to find given in dictionaries.

#SP22-BSE-017 HASAAN AHMAD

class TrieNode:

    def \_\_init\_\_(self):

        self.children = {}

        self.isEndOfWord = False

class Trie:

    def \_\_init\_\_(self):

        self.root = TrieNode()

    def insert(self, word):

        node = self.root

        for char in word:

            if char not in node.children:

                node.children[char] = TrieNode()

            node = node.children[char]

        node.isEndOfWord = True

    def search(self, word):

        node = self.root

        for char in word:

            if char not in node.children:

                return False

            node = node.children[char]

        return node.isEndOfWord

    def startsWith(self, prefix):

        node = self.root

        for char in prefix:

            if char not in node.children:

                return False

            node = node.children[char]

        return True

    def remove(self, word):

        def helper(node, word, index):

            if index == len(word):

                if not node.isEndOfWord:

                    return False

                node.isEndOfWord = False

                return len(node.children) == 0

            char = word[index]

            if char not in node.children:

                return False

            shouldDeleteCurrentNode = helper(node.children[char], word, index + 1)

            if shouldDeleteCurrentNode:

                del node.children[char]

                return len(node.children) == 0

            return False

        helper(self.root, word, 0)

def findWords(board, words):

    trie = Trie()

    for word in words:

        trie.insert(word)

    result = set()

    for i in range(len(board)):

        for j in range(len(board[0])):

            dfs(board, trie.root, i, j, "", result)

    return list(result)

def dfs(board, node, i, j, path, result):

    if node.isEndOfWord:

        result.add(path)

    if i < 0 or i >= len(board) or j < 0 or j >= len(board[0]):

        return

    temp = board[i][j]

    if temp not in node.children:

        return

    board[i][j] = "#"

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

    for direction in directions:

        x, y = i + direction[0], j + direction[1]

        dfs(board, node.children[temp], x, y, path + temp, result)

    board[i][j] = temp

board = [

    ["M", "S", "E", "F"],

    ["R", "A", "T", "D"],

    ["L", "O", "N", "E"],

    ["S", "T", "O", "N"]

]

words = ["START", "NOTE", "SAND", "STONED"]

print(findWords(board, words))

**Output:**

