**Bahria University,**

Karachi Campus



## LAB EXPERIMENT NO

## 3

## LIST OF TASKS

|  |  |
| --- | --- |
| **TASK NO** | **OBJECTIVE** |
| **1** | Implement BFS & DFS Algorithm in python on the given graph |
| **2** | Implement the BFS and DFS Algorithm using recursion on the given graph: |
| **3** | Apply the UCS algorithm on a map given below. Find optimal cost from ARAD to BUCHAREST |
| **4** | Implement the Travelling Salesmen problem using uninformed searches on given Directed graph |

**Submitted On:**

18 feb 2024

(Date: DD/MM/YY)

**TASK # 1:** Implement BFS & DFS Algorithm in python on the given graph

BFS

from collections import deque

class Graph:

    def \_\_init\_\_(self, adj\_list):

        self.adj\_list = adj\_list

def bfs(graph, start\_node):

    visited = set()

    queue = deque([start\_node])

    while queue:

        current\_node = queue.popleft()

        print(current\_node, end=" ")

        visited.add(current\_node)

        for neighbor in graph.adj\_list[current\_node]:

            if neighbor not in visited:

                queue.append(neighbor)

adj\_list = {

    'A': ['B', 'C', 'D'],

    'B': ['E', 'F'],



    'C': [],

    'D': [],

    'E': [],

    'F': []

}

graph = Graph(adj\_list)

print("BFS traversal starting from node 'A':")

bfs(graph, 'A')

**DFS:**

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

        self.right = None

def dfs(node):

    if node is not None:

        print(node.data, end=" ")

        dfs(node.left)

        dfs(node.right)

root = Node('A')

root.left = Node('B')



root.right = Node('C')

root.left.left = Node('E')

root.left.right = Node('F')

root.right.left = Node('D')

print("DFS traversal:")

dfs(root)

**TASK # 2:** Implement BFS & DFS Algorithm in python on the given graph

graph = {

    1: [2, 3, 4],

    2: [5, 6],

    3: [],

    4: [7, 8],

    5: [9, 10],

    6: [],

    7: [11],

    8: [12],

    9: [],

    10: [],

    11: [],

    12: []

}

from collections import deque

def bfs\_recursive(graph, queue, visited):

    if not queue:

        return visited

    vertex = queue.popleft()

    visited.append(vertex)

    for neighbor in graph[vertex]:

        if neighbor not in visited and neighbor not in queue:

            queue.append(neighbor)

    return bfs\_recursive(graph, queue, visited)

def dfs\_recursive(graph, vertex, visited):

    visited.append(vertex)

    for neighbor in graph[vertex]:

        if neighbor not in visited:

            dfs\_recursive(graph, neighbor, visited)

    return visited

bfs\_visited = bfs\_recursive(graph, deque([1]), [])

dfs\_visited = dfs\_recursive(graph, 1, [])

print("BFS",bfs\_visited)

print("DFS",dfs\_visited)



**TASK # 3:** Apply the UCS algorithm on a map given below. Find optimal cost from ARAD to BUCHAREST

import heapq

def ucs(graph, start, goal):

    visited = set()

    priority\_queue = [(0, start, [])]

    while priority\_queue:

        (cost, node, path) = heapq.heappop(priority\_queue)

        if node not in visited:

            visited.add(node)

            path = path + [node]

            if node == goal:

                return (cost, path)

            for (next\_node, weight) in graph.get(node, []):

                if next\_node not in visited:

                   heapq.heappush(priority\_queue, (cost + weight, next\_node, path))

    return (float("inf"), [])

graph = {

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

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

    'Oradea': [('Zerind', 71), ('Sibiu', 151)],

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

    'Timisoara': [('Arad', 118), ('Lugoj', 111)],

    'Lugoj': [('Timisoara', 111), ('Mehadia', 70)],

    'Mehadia': [('Lugoj', 70), ('Dobreta', 75)],

    'Dobreta': [('Mehadia', 75), ('Craiova', 120)],

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

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

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

    '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)]

}

result = ucs(graph, 'Arad', 'Bucharest')

result

import heapq

def ucs(graph, start, goal):

    visited = set()

    priority\_queue = [(0, start, [])]

    while priority\_queue:

        (cost, node, path) = heapq.heappop(priority\_queue)

        if node not in visited:

            visited.add(node)

            path = path + [node]

            if node == goal:

                return (cost, path)

            for (next\_node, weight) in graph.get(node, []):

                if next\_node not in visited:

                   heapq.heappush(priority\_queue, (cost + weight, next\_node, path))

    return (float("inf"), [])

graph = {

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

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

    'Oradea': [('Zerind', 71), ('Sibiu', 151)],

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

    'Timisoara': [('Arad', 118), ('Lugoj', 111)],

    'Lugoj': [('Timisoara', 111), ('Mehadia', 70)],

    'Mehadia': [('Lugoj', 70), ('Dobreta', 75)],

    'Dobreta': [('Mehadia', 75), ('Craiova', 120)],

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

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

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

    '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)]

}

result = ucs(graph, 'Arad', 'Bucharest')

result



**TASK # 4:** Implement the Travelling Salesmen problem using uninformed searches on given Directed graph

from itertools import permutations

graph = {

    'a': {'b': 10, 'f': 6},

    'b': {'c': 19},

    'c': {'d': 27},

    'd': {'e': 5},

    'e': {'b': 10, 'd': 5, 'f': 12},

    'f': {'a': 9, 'c': 15}

}

def calculate\_distance(path):

    distance = 0

    for i in range(len(path)):

        if path[i] in graph and path[(i + 1) % len(path)] in graph[path[i]]:

            distance += graph[path[i]][path[(i + 1) % len(path)]]

        else:

            return float('inf')

    return distance

def traveling\_salesman(graph):

    nodes = list(graph.keys())

    shortest\_path = None

    min\_distance = float('inf')

    for perm in permutations(nodes):

        current\_distance = calculate\_distance(perm)

        if current\_distance < min\_distance:

            min\_distance = current\_distance

            shortest\_path = perm

    return shortest\_path, min\_distance

shortest\_path, min\_distance = traveling\_salesman(graph)

shortest\_path, min\_distance

