23CCE201 Data Structures

Name : E Eeshwar

Roll No : CB.EN.U4CCE23012

AIM :

- To implement various traversal schemes on weighted directed graphs.

LOGIC :

- To implement traversal schemes on weighted directed graphs, there are two default methods: Breadth First Search (BFS) and Depth First Search (DFS).
- In Breadth First Search, each layer of vertices is assessed and then we proceed to the next layer. First, we assign a source vertex from which we traverse through the rest of the vertices. From the current vertex, we search all other connected vertices and mark them as viewed vertices by either colouring them or adding them to a list. If a vertex is already coloured or is present in the list, we will not add colour it or add it to the list again. After viewing all the edges, we move on to the next vertex until all the vertices have been covered.
- In Depth First Search, one vertex from each layer is assessed and we move on to the next layer. First, we assign a source vertex from which we traverse through the rest of the vertices. From the current vertex, we find a connected vertex and go to the connected edge. This process occurs until there are no more unvisited vertices. If any vertex is left unvisited and not part of the taken path of traversal, we go to the source vertex and proceed with another traversal path.

ALGORITHM :

- **Breadth First Search (BFS) :**
    1. Initialise user_input function and get input from the user for the number of vertices, information about the vertex and its neighbours, and store it in a dictionary.
    2. Initialize the BFS function with a source vertex, marking it as visited and adding it to a queue.
    3. Initialize the bfs_algo function, in which a vertex is dequeued from the queue and added in the traversal list. Then the function iterates through the neighbouring vertices as well.
    4. Each vertex iterated is marked as visited.
    5. After initial run of the bfs traversal, the code checks whether all the vertices are visited or not. If any vertex is missing, another bfs traversal is started from that vertex.
    6. After completion of traversal, return the BFS traversal list and display to the user.
- **Depth First Search (DFS) :**
    1. Initialise user_input function and get input from the user for the number of vertices, information about the vertex and its neighbours, and store it in a dictionary.
    2. Initialize DFS function from the source vertex using a stack and track visited vertex.
    3. Initialize the dfs_algo function, in which vertex is popped out from the top of the stack and added to the traversal list. Then the neighbouring vertices are iterated in a reverse oreder.
    4. Each vertex iterated is marked as visited.
    5. After initial run of the dfs traversal, the code checks whether all the vertices are visited or not. If any vertex is missing, another dfs traversal is started from that vertex.
    6. After completion of traversal, return the DFS traversal list and display to the user.

CODE :

- **Breadth First Search (BFS) :**

```
from collections import deque
def user_input():
  graph = {}
```

```
    n = int(input("Enter the number of vertices: "))
    for i in range(n):
        vertex = input("\nEnter the vertex: ")
        neighbours = []
        m = int(input(f"Enter the number of edges for {vertex}: "))
        for j in range(m):
            neighbour = input(f"Enter neighbour of {vertex}: ")
            neighbours.append(neighbour)
        graph[vertex] = neighbours
return graph

def bfs_algo(q, bfs_result, visited, graph):
    while q:
        vertex = q.popleft()
        bfs_result.append(vertex)
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                q.append(neighbour)

def BFS(graph, source):
    q = deque([source])
    visited = set([source])
    bfs_result = []
    bfs_algo(q, bfs_result, visited, graph)
    for vertex in graph:
        if vertex not in visited:
            q.append(vertex)
            visited.add(vertex)
            bfs_algo(q, bfs_result, visited, graph)
    return bfs_result

graph = user_input()
source = input("Enter the source node: ")
bfs_result = BFS(graph, source)
print("BFS Traversal:", bfs_result)
```

- **Depth First Search (DFS) :**

```
def user_input():
    graph = {}
    n = int(input("Enter the number of vertices: "))
    for i in range(n):
        vertex = input("\nEnter the vertex: ")
        neighbours = []
        m = int(input(f"Enter the number of edges for {vertex}: "))
        for j in range(m):
            neighbour = input(f"Enter the neighbour of {vertex}: ")
            neighbours.append(neighbour)
        graph[vertex] = neighbours
    return graph

def dfs_algo(stack,visited,dfs):
    while stack:
        vertex = stack.pop()
        dfs.append(vertex)
        for neighbour in reversed(graph[vertex]):
            if neighbour not in visited:
                visited.add(neighbour)
                stack.append(neighbour)

def dfs(graph,source):
    stack = [source]
    visited = set()
    visited.add(source)
    dfs = []
    while stack:
        dfs_algo(stack,visited,dfs)
    while len(visited) < len(graph):
        for vertex in graph:
            if vertex not in visited:
                stack.append(vertex)
                visited.add(vertex)
                dfs_algo(stack,visited,dfs)
    return dfs
```

```
graph = user_input()
source = input("Enter the source vertex: ")
dfs = dfs(graph,source)
print(dfs)
```

RESULTS :

- **Breadth First Search (BFS) :**

```
Enter the number of vertices: 8
Enter the vertex: A
Enter the number of edges for A: 2
Enter neighbour of A: B
Enter neighbour of A: C
Enter the vertex: B
Enter the number of edges for B: 3
Enter neighbour of B: C
Enter neighbour of B: D
Enter neighbour of B: E
Enter the vertex: C
Enter the number of edges for C: 1
Enter neighbour of C: A
Enter the vertex: D
Enter the number of edges for D: 1
Enter neighbour of D: E
Enter the vertex: E
Enter the number of edges for E: 1
Enter neighbour of E: C
Enter the vertex: F
Enter the number of edges for F: 2
Enter neighbour of F: G
Enter neighbour of F: H
Enter the vertex: G
Enter the number of edges for G: 1
Enter neighbour of G: B
Enter the vertex: H
Enter the number of edges for H: 1
Enter neighbour of H: D
Enter the source node: A
BFS Traversal: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
```

- **Depth First Search (DFS) :**

```
Enter the number of vertices: 8
Enter the vertex: A
Enter the number of edges for A: 2
Enter the neighbour of A: B
Enter the neighbour of A: C
Enter the vertex: B
Enter the number of edges for B: 3
Enter the neighbour of B: C
Enter the neighbour of B: D
Enter the neighbour of B: E
Enter the vertex: C
Enter the number of edges for C: 1
Enter the neighbour of C: A
Enter the vertex: D
Enter the number of edges for D: 1
Enter the neighbour of D: E
Enter the vertex: E
Enter the number of edges for E: 1
Enter the neighbour of E: C
Enter the vertex: F
Enter the number of edges for F: 2
Enter the neighbour of F: H
Enter the neighbour of F: G
Enter the vertex: G
Enter the number of edges for G: 1
Enter the neighbour of G: B
Enter the vertex: H
Enter the number of edges for H: 1
Enter the neighbour of H: D
Enter the source vertex: A
['A', 'B', 'D', 'E', 'C', 'F', 'H', 'G']
```

INFERENCES :

- The Breadth First Search and Depth First Search traversal methods work effectively when used in weighted directed graphs. Breadth First Search offers layer-by-layer approach and Depth First Search allows discovery of depth of the directed graphs. Both traversal algorithms ensures that all sorts of directed graphs are covered, connected and disconnected alike.