23CCE201 - Data Structures

Name: Anirudh Anand

Roll No.: CB.EN.U4CCE23002

Directed Graphs – Traversal Schemes

I.Aim:

To implement the various traversal schemes on Weighted Directed Graphs.

II. Logic:

The two traversal schemes implemented in this report are the Breadth First Search (BFS) and Depth First Search (DFS).

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

  It explores the graph level by level. Starting from the source node, it visits all the neighbouring nodes first (all the nodes in that depth) before moving onto the nodes that are in the next depth(neighbour of neighbour). The BFS is also thus called a layered approach. A queue is used to manage the vertices required to be explored. Vertices are added to the queue as they are discovered and are processed thus in the order in which they are added (FIFO).

- **Depth First Search (DFS):**
  It explores the graph deeply along each branch. Starting from the source node, DFS visits as far as possible down one path before backtracking to explore other parts. In this approach, we traverse in one direction until we reach a dead end and then backtrack to the previous branching point. A stack is used to manage the vertices to be explored where they are processed in the LIFO order ensuring that the most recently discovered vertex is explored next, allowing us to traverse deep.

III. Algorithm:

- **BFS:**
  1. Initialize a queue to keep track of the vertices to visit and initialize a set to keep track of visited vertices
  2. Enqueue the starting vertex and mark as visited
  3. Process the queue
     a. While the queue is not empty
        i. Dequeue
        ii. Visit the unvisited neighbours of the dequeued vertex

            1. Mark the neighbour as visited
            2. Add the neighbour to the queue to be explored later
        4. Repeat step 3 until the queue is empty
        5. End

- **DFS:**
    1. Initialise a stack to keep track of vertices to visit and a set to keep track of visited vertices
    2. Push the starting node onto the stack and mark it as visited
    3. Process the stack
        a. While the stack is not empty
            i. Pop the top vertex from the stack
            ii. For each unvisited neighbour of the popped vertex
                1. Mark the neighbour as visited
                2. Push the neighbour onto the stack to explore it deeply before backtracking
    4. Repeat step 3 until the stack is empty
    5. End

IV. Code:

```
from collections import deque

class Graph:

    def __init__(self):

        # Dictionary to store the adjacency list of the directed graph

        self.graph = {}

    def add_edge(self, u, v):

        """Add a directed edge from node u to node v."""

        if u not in self.graph:

            self.graph[u] = []

        self.graph[u].append(v)  # Only one direction for directed graph

    def bfs(self, start):

        """Perform Breadth-First Search (BFS) starting from the given node."""

        visited = set()  # Set to keep track of visited nodes

        queue = deque([start])  # Queue for BFS

        bfs_result = []  # To store the order of traversal

        while queue:

            node = queue.popleft()
```

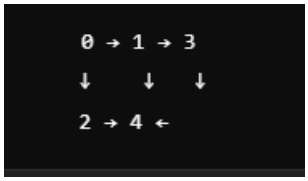
```
            if node not in visited:
                visited.add(node)
                bfs_result.append(node)
                # Add all unvisited neighbors to the queue
                for neighbor in self.graph.get(node, []):
                    if neighbor not in visited:
                        queue.append(neighbor)
        return bfs_result
    def dfs(self, start):
        """Perform Depth-First Search (DFS) starting from the given node."""
        visited = set()  # Set to keep track of visited nodes
        dfs_result = []  # To store the order of traversal
        def dfs_recursive(node):
            """Helper function to perform DFS using recursion."""
            visited.add(node)
            dfs_result.append(node)
            # Visit all unvisited neighbors
            for neighbor in self.graph.get(node, []):
                if neighbor not in visited:
                    dfs_recursive(neighbor)
        dfs_recursive(start)  # Start the DFS from the starting node
        return dfs_result

g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(1, 3)
g.add_edge(2, 4)
g.add_edge(3, 4)
```

```
# Perform BFS and DFS from a starting node (e.g., node 0)
start_node = 0
print("BFS Traversal:", g.bfs(start_node))
print("DFS Traversal:", g.dfs(start_node))
```

V. Results:

Graph:



Traversal:

```
BFS Traversal: [0, 1, 2, 3, 4]
DFS Traversal: [0, 1, 2, 4, 3]
```

VI. Inference:

Thus, we have performed the BFS and DFS traversal schemes on a given directed graph. Starting from vertex 0, BFS visits vertices in a level-wise manner, exploring all reachable vertices from 0 before moving on to vertices further out. On the contrary, DFS dives as deep as possible along one path before backtracking to explore other paths. DFS can be adapted for that purpose by checking if any vertex visited during DFS is revisited within the same traversal path. These traversal schemes can also be fused with shortest path algorithms for specific applications.