CU CHANDIGARH UNIVERSITY

UNIVERSITY INSTITUTE of COMPUTING
Asia's Fastest Growing University

NAAC GRADE A+ ACCREDITED UNIVERSITY

Worksheet 3

Student Name: Rahul Saxena **UID:** 24MCI10204
Branch: MCA(AI&ML) **Section/Group:** 3-B
Semester: 1st semester **Date of Performance:** 10/09/2024
Subject Name: Desing and Analysis of Algorithm Lab **Subject Code:** 24CAP-612

Aim/Overview of the practical:

Print all the nodes reachable from a given starting node in a digraph using BFS method and Check whether a given graph is connected or not using DFS method.

Task To be done:

- **Graph Representation Using Vector:** Represent a directed graph using an adjacency list, where each node's neighbors are stored in a Vector<Vector<Integer>>. This provides a dynamic structure for handling graph nodes and edges.
- **Breadth-First Search (BFS) to Find Reachable Nodes:**
    - Implement BFS using a Queue and a Vector<Boolean> to track visited nodes.
    - Starting from a given node, print all reachable nodes in the graph by exploring **layer by layer.**
- **Depth-First Search (DFS) to Check Graph Connectivity:**
    - Implement DFS using recursion and a Vector<Boolean> to track visited nodes.
    - Starting from node 0, determine whether the entire graph is connected (i.e., if all nodes can be reached from the starting node).

Source Code:

Task 1: Printing all the reachable note from the given starting node using BSF

```
import java.util.*;
class BFSReachableNodes {
    public static void bfs(int start, Vector<Vector<Integer>> graph) {
        Vector<Boolean> visited = new Vector<>(Collections.nCopies(graph.size(), false));
        Queue<Integer> queue = new LinkedList<>();
        visited.set(start, true);
        queue.offer(start);
        System.out.println("Nodes reachable from node " + start + ":");
```

```
        while (!queue.isEmpty()) {
            int node = queue.poll();
            System.out.print(node + " ");
            // Get neighbors of the current node
            Vector<Integer> neighbors = graph.get(node);
            for (int neighbor : neighbors) {
                if (!visited.get(neighbor)) {
                    visited.set(neighbor, true);
                    queue.offer(neighbor);
                }
            }
        }
        System.out.println();
    }
    public static void main(String[] args) {
        Vector<Vector<Integer>> graph = new Vector<>();
        graph.add(new Vector<>(Arrays.asList(1, 2)));  // Node 0
        graph.add(new Vector<>(Arrays.asList(2)));     // Node 1
        graph.add(new Vector<>(Arrays.asList(0, 3)));  // Node 2
        graph.add(new Vector<>(Arrays.asList(3)));     // Node 3
        int startNode = 2;
        bfs(startNode, graph);
    }

}
```

Task 2: Check whether a given graph is connected or not using DFS method.

```
class DFSCheckConnected {
    public static void dfs(int node, Vector<Vector<Integer>> graph, Vector<Boolean>
visited) {
        visited.set(node, true);
        // Get neighbors of the current node
        Vector<Integer> neighbors = graph.get(node);
        for (int neighbor : neighbors) {
            if (!visited.get(neighbor)) {
                dfs(neighbor, graph, visited);
            }
        }
    }
    public static boolean isConnected(Vector<Vector<Integer>> graph) {
        Vector<Boolean> visited = new Vector<>(Collections.nCopies(graph.size(), false));
```

```java
        dfs(0, graph, visited);
        for (boolean visitStatus : visited) {
            if (!visitStatus) {
                return false;
            }
        }
        return true;
    }
    public static void main(String[] args) {
        Vector<Vector<Integer>> graph = new Vector<>();
        // Initialize the graph (Example: Directed graph)
        graph.add(new Vector<>(Arrays.asList(1, 2)));
        graph.add(new Vector<>(Arrays.asList(2)));
        graph.add(new Vector<>(Arrays.asList(0, 3)));
        graph.add(new Vector<>(Collections.emptyList()));
        if (isConnected(graph)) {
            System.out.println("The following graph is connected.");
        } else {
            System.out.println("The graph is not connected.");
        }
    }
}
```

Output:

Task 1:

```
Starting Node: 2
Nodes reachable from node 2:
2 0 3 1

Process finished with exit code 0
```

Task 2:

```
The following graph is connected.

Process finished with exit code 0
```

Learning Outcome:

- **Use of Vectors for Dynamic Graph Representation:**
  - Learn how to represent a graph using Vector<Vector<Integer>> for adjacency lists, gaining flexibility in storing neighbors dynamically.
  - Understand the advantage of using Vector for dynamic sizing, which simplifies graph operations such as adding or removing nodes and edges**.**
- **BFS and DFS Implementation:**
  - Gain experience with implementing BFS using a queue, where nodes are explored level by level.
  - Learn to implement DFS recursively, visiting nodes deeply in one path before backtracking.
  - Understand the differences between BFS and DFS and when to use each approach.
- **Graph Traversal and Connectivity:**
  - Understand how BFS can be used to find all reachable nodes from a given node in a directed graph.
  - Learn how DFS can be applied to check if a graph is connected by traversing through all nodes.
  - Strengthen problem-solving skills by applying graph traversal techniques to solve real-world graph-related problems.
- **Efficient Use of Data Structures:**
  - Learn how to use Vector<Boolean> to track visited nodes efficiently.
  - Understand the use of Queue and Vector for implementing BFS and DFS in a directed graph.