LAB SESSION 14

Graph Traversals – BFS and DFS

THEORY

A **graph** is a non-linear data structure consisting of vertices (nodes) and edges (connections). Graphs can be **directed** or **undirected**, **weighted** or **unweighted**. They are used to model networks such as social connections, road maps, or communication systems.
Graph traversal refers to visiting all the vertices of a graph in a systematic manner. The two most common traversal methods are:

1. **Breadth-First Search (BFS)**
    - Traverses level by level, starting from a given source vertex.
    - Uses a **queue** data structure to track the vertices to visit.
    - Finds the shortest path (minimum edges) in an unweighted graph.
    - Time Complexity: **O(V + E)**, where V = number of vertices, E = number of edges.
2. **Depth-First Search (DFS)**
    - Traverses as far as possible along each branch before backtracking.
    - Uses a **stack** (can be implemented recursively).
    - Suitable for detecting cycles, topological sorting, and solving maze problems.
    - Time Complexity: **O(V + E)**.

Table 14.1 Key Differences

BFS	DFS
Explores nearest neighbors first.	Explores as far as possible along a path first.
Uses a queue.	Uses a stack or recursion.
Finds shortest path in unweighted graphs.	Does not guarantee shortest path.

PROCEDURE

1. Represent the graph using adjacency lists.
2. Implement BFS using a queue.
3. Implement DFS using recursion.
4. Input the number of vertices, edges, and their connections.
5. Perform BFS and DFS from a given starting vertex.
6. Display the order of visited vertices for each algorithm.

CODE

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

class Graph {
```

```
    int V;
    vector<vector<int>> adj;
public:
    Graph(int v) : V(v) { adj.resize(v); }

    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u); // For undirected graph
    }

    void BFS
        vector<bool> visited(V, false);
        queue<int> q;
        visited[start] = true;
        q.push(start);

        cout << "BFS Traversal: ";
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            cout << node << " ";
            for (int neighbor : adj[node]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
        cout << endl;
    }

    void DFSUtil(int node, vector<bool>& visited) {
        visited[node] = true;
        cout << node << " ";
        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                DFSUtil(neighbor, visited);
            }
        }
    }

    void DFS(int start) {
        vector<bool> visited(V, false);
        cout << "DFS Traversal: ";
        DFSUtil(start, visited);
        cout << endl;
    }
};

int main() {
    Graph g(6);
```

```
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 4);
    g.addEdge(3, 5);
    g.addEdge(4, 5);

    g.BFS(0);
    g.DFS(0);

    return 0;
}
```

EXPECTED OUTPUT

BFS Traversal: 0 1 2 3 4 5
DFS Traversal: 0 1 3 5 4 2

EXERCISE

1. Modify the BFS implementation to store and print the shortest path from the source to every vertex.
2. Extend DFS to detect cycles in an undirected graph.
3. Modify the program for directed graphs.
4. Apply BFS on a weighted graph by ignoring weights (treating as unweighted).