Lab Experiment No. 4

Objective

To Implementation and analysis of Graph based single source shortest distance algorithms.

Task:

T-4.1 Breadth first search
T-4.2 Depth first search
T-4.3 Dijkstra's algorithm
T-4.4 Topological Sort
T-4.5 Floyd–Warshall algorithm

Theory

The shortest path from a single source to all other nodes in a graph.

T-4.1. Breadth-First Search (BFS)

- BFS is a graph traversal algorithm that explores all the vertices of a graph level by level.
- It uses a queue data structure to keep track of the nodes to be explored.
- BFS can be used to find the shortest path in an unweighted graph.

T-4.2. Depth-First Search (DFS)

- DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking.
- It uses a stack data structure (or recursion) to keep track of the nodes to be explored.
- DFS is not typically used for finding the shortest path but is useful for topological sorting and cycle detection.

T-4.3. Dijkstra's Algorithm

- Dijkstra's algorithm is used to find the shortest path from a single source to all other nodes in a weighted graph with non-negative edge weights.
- It uses a priority queue (min-heap) to greedily select the node with the smallest distance.

T-4.4. Topological Sort

- Topological sort is used for Directed Acyclic Graphs (DAGs) to linearly order the vertices such that for every directed edge (u, v), vertex u comes before v in the ordering.
- It is implemented using DFS.

T-4.5. Floyd-Warshall Algorithm

- The Floyd-Warshall algorithm is used to find the shortest paths between all pairs of vertices in a weighted graph.
- It works for both directed and undirected graphs and can handle negative edge weights (but not negative cycles).

Code

T-4.1. Breadth-First Search (BFS)

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

class Graph{
        int V;
        int t = 0;
        vector<int> path;
        vector< vector<int> > adj;

        void BFS_util(int s, vector<bool>& visited){

                queue<int> q;
                visited[s]=true;
                q.push(s);
                while(!q.empty()){
                        int curr = q.front();
                        q.pop();
                        path.push_back(curr);

                        for(int i:adj[curr]){
                                ++t;
                                if(!visited[i]){
visited[i]=true;
                                        q.push(i);
                                }

                        }
                }
        }

        public:
        Graph(int v){
                V=v;
```

```
void BFS(int s){
                vector <bool> visited(V,false);
                BFS_util(s,visited);
        }

        void printPath(){
                for(int i:path){
                        cout<<i<<" ";
                }
        }

        int T(){return t;}

};

 int main(){
        cout<<"No of Vertices: ";
        int v;cin>>v;

        Graph g(v);
        cout<<"No of Edges: ";

        int e;cin>>e;
        cout<<"Ënter "<<e<<" edges: "<<endl;
        for(int i=0;i<e;++i){
                int v,u;cin>>v>>u;

                g.addEdge(v,u);
        }

        cout<<"Enter source vertex: ";
        int s;cin>>s;
        g.BFS(s);
        cout<<"Path: ";
        g.printPath();
        cout<<endl;
```

<pre>   adj.resize(v);
        //path.resize(v);
    }
 	
    void addEdge(int v, int u){
        adj[v].push_back(u);
        adj[u].push_back(v);
    }</pre>
	<pre> cout<<endl<<"Total time complexity
	O(V+E)= "<<g.T();
	    cout<<endl;
	    cout<<endl<<"Auxiliary space complexity
	O(V)= "<<v;
	    cout<<endl;
	
	    return 0;
	}</pre>

T-4.2. Depth-First Search (DFS)

```
#include<iostream>
#include<vector>

using namespace std;

class Graph{
    int V;
    int t = 0;
    vector<int> path;
    vector< vector<int> > adj;

    void DFS_util(int s, vector<bool>& visited){

        visited[s]=true;
        path.push_back(s);
        for(int i: adj[s]){
            ++t;
            if(!visited[i])
                DFS_util(i,visited);
        }
    }

    public:
    Graph(int v){
        V=v;
        adj.resize(v);
    }

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

    void DFS(int s){
        vector <bool> visited(V,false);
        DFS_util(s,visited);
    }
```

```
void printPath(){
        for(int i:path){
            cout<<i<<" ";
        }
    }

    int T(){return t;}

};
int main(){
    cout<<"No of Vertices: ";
    int v;cin>>v;

    Graph g(v);
    cout<<"No of Edges: ";

    int e;cin>>e;
    cout<<"Enter "<<e<<" edges: "<<endl;
    for(int i=0;i<e;++i){
        int v,u;cin>>v>>u;

        g.addEdge(v,u);
    }

    cout<<"Enter source vertex: ";
    int s;cin>>s;
    g.DFS(s);
    cout<<"Path: ";
    g.printPath();
    cout<<endl;
    cout<<endl<<"Total time complexity
O(V+E)= "<<g.T();
    cout<<endl;
    cout<<endl<<"Auxiliary space complexity
O(V+E)= "<<v+e;
    cout<<endl;

    return 0;
}
```

T-4.3. Dijkstra's Algorithm

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>

using namespace std;

class Graph {
    int V;
    vector<vector<pair<int, int>>> adj; // Adjacency list
(node, weight)

public:
    Graph(int v) {
        V = v;
        adj.resize(v);
    }

    void addEdge(int u, int v, int weight) {
        adj[u].push_back({v, weight});
        adj[v].push_back({u, weight}); // Remove this for
a directed graph
    }

    void dijkstra(int src) {
        vector<int> dist(V, INT_MAX);
        priority_queue<pair<int, int>, vector<pair<int,
int>>, greater<pair<int, int>>> pq;

        dist[src] = 0;
        pq.push({0, src});

        while (!pq.empty()) {
            int u = pq.top().second;
            pq.pop();

            for (auto &[v, weight] : adj[u]) {
                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.push({dist[v], v});
                }
            }
        }
```

```
        cout << "Shortest distances from source " << src
<< ":\n";
        for (int i = 0; i < V; i++) {
            cout << "Node " << i << " -> Distance: " <<
dist[i] << endl;
        }
    }
};

int main() {
    cout << "Enter number of vertices: ";
    int V, E;
    cin >> V;
    Graph g(V);

    cout << "Enter number of edges: ";
    cin >> E;

    cout << "Enter " << E << " edges (u v weight):\n";
    for (int i = 0; i < E; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        g.addEdge(u, v, w);
    }

    cout << "Enter source vertex: ";
    int src;
    cin >> src;

    g.dijkstra(src);

    return 0;
}
```

Sample Output

T-4.1. Breadth-First Search (BFS)

No of Vertices: 5
No of Edges: 5
Enter 5 edges:
0 1

0 2
1 2
2 3
2 4
Enter source vertex: 2
Path: 2 0 1 3 4

Total time complexity O(V+E)= 10
Auxiliary space complexity O(V)= 5

T-4.2. Depth-First Search (DFS)

No of Vertices: 5
No of Edges: 5
Enter 5 edges:
2 4
2 3
1 2
0 2
0 1
Enter source vertex: 2
Path: 2 4 3 1 0

Total time complexity O(V+E)= 10
Auxiliary space complexity O(V+E)= 10

T-4.3. Dijkstra's Algorithm

Enter number of vertices: 5
Enter number of edges: 7
Enter 7 edges (u v weight):
0 1 2
0 2 4
1 2 1
1 3 7
2 4 3
3 4 1
3 2 2
Enter source vertex: 0
Shortest distances from source 0:
Node 0 -> Distance: 0
Node 1 -> Distance: 2
Node 2 -> Distance: 3
Node 3 -> Distance: 5
Node 4 -> Distance: 6

Complexity Analysis

Algorithm	Time Complexity	Space Complexity
BFS	$O(V + E)$	$O(V)$
DFS	$O(V + E)$	$O(V)$
Dijkstra's Algorithm	$O((V + E) \log V)$	$O(V)$
Topological Sort	$O(V + E)$	$O(V)$
Floyd-Warshall	$O(V^3)$	$O(V^2)$

Conclusion

In this lab, we implemented and analyzed five graph-based algorithms for finding the shortest path or traversing a graph. Each algorithm has its strengths and weaknesses, and the choice of algorithm depends on the problem requirements (e.g., weighted vs. unweighted graphs, single-source vs. all-pairs shortest paths).

1. **BFS** is efficient for unweighted graphs and guarantees the shortest path.
2. **DFS** is useful for topological sorting and cycle detection but not for shortest paths.
3. **Dijkstra's Algorithm** is optimal for weighted graphs with non-negative edges.
4. **Topological Sort** is applicable only for Directed Acyclic Graphs (DAGs).
5. **Floyd-Warshall** is ideal for finding all-pairs shortest paths but has a higher time complexity.

These algorithms form the foundation for solving more complex graph problems in computer science.