Graph Algorithms

5A. Breadth First Search Implement BFS and show the adjacency matrix of the spanning tree.

Algorithm:

Step 1: Create a queue Q to store the vertices.
Step 2: Push the source vertex S in the queue Q.
Step 3: Mark **S** as visited.
Step 4: While the queue Q is not empty
Step 5: Remove vertex U from the front of the queue.
Step 6: For every vertex V adjacent to the vertex U**,** If the vertex V is not visited Then Explore the vertex V and mark V as visited. Push the vertex V in the queue Q.

Program: [In next page]

```cpp
// Exp5A: Breadth First Search
// Author: Pranjal Timsina
#include <iostream>
#include <vector>

using namespace std;

class Graph {

    int v, e; // vertices, edges
    int** adj; // Adjacency matrix

public:
    Graph(int v, int e); // constructor

    void addEdge(int start, int e); // insert new edge

    void BFS(int start); // BFS traversal
    void printMatrix() {
        for (int row = 0; row < v; row++) {
            cout << "\n";
            for (int c= 0;c< v;c++) {
                cout << adj[row][c] << " ";
            }
        }
    }
};
Graph::Graph(int v, int e) {
    // fill adjacency matrix
    this->v = v;
    this->e = e;
    adj = new int*[v];
    for (int row = 0; row < v; row++) {
        adj[row] = new int[v];
        for (int column = 0; column < v; column++) {
            adj[row][column] = 0;
        }
    }
}
```

```
void Graph::addEdge(int start, int e) {
    // add an edge to the graph
    adj[start][e] = 1;
    adj[e][start] = 1;
}

void Graph::BFS(int start) {
    vector<bool> visited(v, false);
    vector<int> q;
    q.push_back(start);
    visited[start] = true;

    int vis;
    while (!q.empty()) {
        vis = q[0];

        // Print the current node
        cout << vis << " ";
        q.erase(q.begin());

        // For every adjacent vertex to the current vertex
        for (int i = 0; i < v; i++) {
            if (adj[vis][i] == 1 && (!visited[i])) {

                // Push the adjacent node to the queue
                q.push_back(i);
                visited[i] = true;
            }
        }
    }
}
```

```
int main() {
    int v = 5, e = 4;
    cout << "Enter vertices: ";
    cin >> v;
    cout <<  "Enter edges: ";
    cin >> e;

    Graph G(v, e);
    for (int i = 0; i < e; i++) {
        int a, b;
        cout << "Enter pair (0 1): ";
        cin >> a >> b;
        G.addEdge(a, b);

    }
    G.BFS(0);
    G.printMatrix();
}
```

Output:

```
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 5\src> g++ .\BFS2.cpp
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 5\src> .\a.exe
Enter vertices: 5
Enter edges: 4
Enter pair (0 1): 0 1
Enter pair (0 1): 1 2
Enter pair (0 1): 1 3
Enter pair (0 1): 3 4
0 1 2 3 4
0 1 0 0 0
1 0 1 1 0
0 1 0 0 0
0 1 0 0 1
0 0 0 1 0
```

Results:

Thus, the breadth first search algorithm has been implemented.

5B. Depth First Search Implement DFS and show the adjacency matrix of the spanning tree.

Algorithm:

```
Initialize an empty stack for storage of nodes, S.
For each vertex u, define u.visited to be false.
Push the root (first node to be visited) onto S.
While S is not empty:
    Pop the first element in S, u.
    If u.visited = false, then:
        U.visited = true
        for each unvisited neighbor w of u:
            Push w into S.
End process when all nodes have been visited.
```

Program:

```
// Exp 5B: DFS
// Author: Pranjal Timsina; 20BDS0392
#include <iostream>
#include <vector>
using namespace std;

class Graph {

    int v, e; // vertices, edges
    int** adj; // Adjacency matrix

public:
    Graph(int v, int e); // constructor

    void addEdge(int start, int e); // insert new edge

    // dfs traversal
    void DFS(int start, vector<bool>& visited, Graph T);

    void printMatrix() {
        for (int row = 0; row < v; row++) {
            cout << "\n";
            for (int c= 0;c< v;c++) {
                cout << adj[row][c] << " ";
            }
        }
```

```
    }
};

// Function to fill the empty adjacency matrix
Graph::Graph(int v, int e) {
    // fill adjacency matrix
    this->v = v;
    this->e = e;
    adj = new int*[v];
    for (int row = 0; row < v; row++) {
        adj[row] = new int[v];
        for (int column = 0; column < v; column++) {
            adj[row][column] = 0;
        }
    }
}

void Graph::addEdge(int start, int e) {
    // add an edge to the graph
    adj[start][e] = 1;
}

// Function to perform DFS on the graph
void Graph::DFS(int start, vector<bool>& visited, Graph T) {

    visited[start] = true;

    for (int i = 0; i < v; i++) {
        // go to the depths of a node
        if (adj[start][i] == 1 && (!visited[i])) {
            T.addEdge(start, i);
            DFS(i, visited, T);
        }
    }
}

// Driver code
int main()
{
```

```
    int v = 5, e = 4;
    cout << "Enter vertices: ";
    cin >> v;
    cout <<  "Enter edges: ";
    cin >> e;

    Graph G(v, e);
    Graph T(v, e);
    for (int i = 0; i < e; i++) {
        int a, b;
        cout << "Enter pair (0 1): ";
        cin >> a >> b;
        G.addEdge(a, b);

    }
    vector<bool> visited(v, false);

    G.DFS(0, visited, T);
    T.printMatrix();
}
```

Output:

```
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 5\src> .\a.exe
Enter vertices: 6
Enter edges: 7
Enter pair (0 1): 0 1
Enter pair (0 1): 1 2
Enter pair (0 1): 1 3
Enter pair (0 1): 2 4
Enter pair (0 1): 3 4
Enter pair (0 1): 4 5
Enter pair (0 1): 0 5

0 1 0 0 0 0
0 0 1 1 0 0
0 0 0 0 1 0
0 0 0 0 0 0
0 0 0 0 0 1
0 0 0 0 0 0
```

Results:

Thus, the depth first search algorithm has been implemented.

5C. Dijkstra's Algorithms Find the shortest path from a given graph

Algorithm:

1. Create a set of all unvisited nodes, mark all nodes as unvisited.
2. Set distance of all nodes to infinity and set the distance of origin to zero Set the initial node as current.
3. Calculate the distance for all the neighbors of the current node with current node as the intermediate node. Compare the new distance to the current distance of a particular node and assign the smaller one.
4. After considering all the neighbors of current, node mark the current node as visited.
5. Repeat till all the nodes are marked visited, or the only node left are not connected to the graph.
6. The resulting distances are the shortest path lengths

Program:

```
// Exp 5C: Dijkstra's Algorithm
// Author: Pranjal Timsina; 20BDS0392
#include <bits/stdc++.h>
#define INF 0x3f3f3f3f
using namespace std;

typedef pair<int,int> myPair;

class Graph{
    int V;
    list<myPair> *adj;
public:
    Graph(int V);
    void addEdge(int u,int v,int w);
    void shortestPath(int src);
};
Graph::Graph(int V){
    this->V = V;
    adj = new list<myPair>[this->V];
}
void Graph::addEdge(int u,int v,int w){
    adj[u].push_back({v,w});
}
//Dijkstra
```

```cpp
void Graph::shortestPath(int src){
    priority_queue<myPair,vector<myPair>,greater<myPair> > pq;
    vector<int> dist(this->V,INF);
    dist[src] = 0;
    list<myPair>::iterator it;

    pq.push({0,src});
    while(!pq.empty()){
        int u = pq.top().second;
        pq.pop();

        for(it = adj[u].begin();it!=adj[u].end();++it){
            int v = it->first;
            int w = it->second;
            if(dist[v] > dist[u] + w){
                dist[v] = dist[u] + w;
                pq.push({dist[v],v});
            }
        }
    }

    for(int i=0;i<this->V;i++){
        cout << "Distance to " << i << ":" << dist[i] << endl;
    }
}
int main(){
    int V = 5, e = 4;
    cout << "Enter vertices: ";
    cin >> V;
    cout <<  "Enter edges: ";
    cin >> e;

    Graph g(V);

    for (int i = 0; i < e; i++) {
        int a, b, w;
        cout << "Enter pair (from to weight): ";
        cin >> a >> b >> w;
        g.addEdge(a,b,w);
```

```
    }
    int src = 0;
    g.shortestPath(src);
    return 0;
}
```

Output:

```
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 5\src> g++ .\Dijkstra.cpp
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 5\src> .\a.exe
Enter vertices: 5
Enter edges: 8
Enter pair (from to weight): 0 1 1
Enter pair (from to weight): 0 2 3
Enter pair (from to weight): 1 2 1
Enter pair (from to weight): 2 3 4
Enter pair (from to weight): 0 3 2
Enter pair (from to weight): 1 3 100
Enter pair (from to weight): 2 4 70
Enter pair (from to weight): 1 4 73
Distance to 0:0
Distance to 1:1
Distance to 2:2
Distance to 3:2
Distance to 4:72
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 5\src>
```

Results:

Thus, the Dijkstra's algorithm has been implemented.