ASSIGNMENT – 2

QUESTION 1: Write a program to find the k(th) shortest path from source to destination in a graph?

Code:

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

using namespace std;

typedef pair<int, int> pii;

// A vector of vectors of pairs to represent the graph's adjacency list
vector<vector<pii>> graph;

// Function to perform Dijkstra's algorithm and return the shortest distances from the start node
vector<int> dijkstra(int start) {
    int n = graph.size();
    vector<int> dist(n, INT_MAX); // Initialize all distances to infinity
    dist[start] = 0; // Distance to the start node is set to 0
    priority_queue<pii, vector<pii>, greater<pii>> pq; // Min-heap priority queue
    pq.push({0, start}); // Push the start node with distance 0

    // Dijkstra's algorithm loop
    while (!pq.empty()) {
        int cost = pq.top().first; // Current cost to reach this node
        int node = pq.top().second; // Current node
        pq.pop(); // Remove the node from the queue

        // Skip if this node has been visited with a shorter path
        if (cost > dist[node]) {
            continue;
        }

        // Explore neighbors of the current node
        for (pii edge : graph[node]) {
            int neighbor = edge.first; // Neighbor node
            int weight = edge.second; // Weight of the edge to the neighbor
            // Update distance and push to queue if a shorter path is found
            if (cost + weight < dist[neighbor]) {
                dist[neighbor] = cost + weight;
                pq.push({dist[neighbor], neighbor});
            }
        }
    }

    return dist; // Return the array of shortest distances
```

```
}

int main() {
    int n, m, k;
    cin >> n >> m >> k; // Input the number of nodes, edges, and kth element

    graph.resize(n); // Resize the graph vector to have 'n' nodes

    // Input the edges and their weights
    for (int i = 0; i < m; ++i) {
        int x, y, w; // Source node, destination node, edge weight
        cin >> x >> y >> w;
        graph[x - 1].push_back({y - 1, w}); // Add edge to adjacency list
        graph[y - 1].push_back({x - 1, w}); // Add edge (graph is undirected)
    }

    vector<vector<int>> dist(n); // 2D vector to store shortest distances from all nodes
    for (int i = 0; i < n; ++i) {
        dist[i] = dijkstra(i); // Run Dijkstra's algorithm for each node
    }

    vector<int> shortest_paths; // Vector to store all shortest paths
    // Generate all shortest path lengths between pairs of nodes
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            shortest_paths.push_back(dist[i][j]); // Add shortest path length to vector
        }
    }

    sort(shortest_paths.begin(), shortest_paths.end()); // Sort the vector of shortest paths

    cout<<"Shortest_paths[k - 1]:   " << shortest_paths[k - 1] << endl; // Output the kth shortest
path

    return 0; // Return 0 to indicate successful completion
}


/*
Input:
6 10 5
2 5 1
5 3 9
6 2 2
1 3 1
5 1 8
6 5 10
1 6 5
6 4 6
3 6 2
3 4 5
output: 3
```

Input:
7 15 18
2 6 3
5 7 4
6 5 4
3 6 9
6 7 7
1 6 4
7 1 6
7 2 1
4 3 2
3 2 8
5 3 6
2 5 5
3 7 9
4 1 8
2 1 1
output: 9
*/

```
PS C:\Users\RAMAVATH SANTHOSH\OneDrive\Desktop\ALL SEMs\SEM3\CCN\Day2>  & '
\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In
-MIEngine-Error-ryesd3av.3kv' '--pid=Microsoft-MIEngine-Pid-txxdnlp4.hfl' '
6 10 5
2 5 1
5 3 9
6 2 2
1 3 1
5 1 8
6 5 10
1 6 5
6 4 6
3 6 2
3 4 5

*****************************
 Shortest_paths[k - 1]:   3

*****************************
PS C:\Users\RAMAVATH SANTHOSH\OneDrive\Desktop\ALL SEMs\SEM3\CCN\Day2>
```

****
****

QUESTION-2: Write a program to generate random graph and find the disjoint paths in the graph?

Code:

```
 #include <bits/stdc++.h>
#include <iostream>
#include <limits.h>
#include <string.h>
#include <queue>
#include <vector>
using namespace std;

#define V 8 // Number of vertices in the graph

// Function to perform Breadth-First Search to find if there is a path from source to sink
bool bfs(int rGraph[V][V], int s, int t, int parent[]) {
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    queue<int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v = 0; v < V; v++) {
            if (!visited[v] && rGraph[u][v] > 0) {
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }

    return visited[t];
}

int main() {
    // Define the capacity matrix of the graph
    int graph[V][V] = {
        {0, 1, 1, 1, 0, 0, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 1, 0},
        {0, 0, 0, 0, 0, 0, 1, 0},
        {0, 0, 1, 0, 0, 0, 0, 1},
        {0, 1, 0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 1, 0, 1},
        {0, 0, 0, 0, 0, 0, 0, 0}
    };

    // Define source-destination pairs for testing
```

```
vector<pair<int, int>> sourceDestPairs = {
    {0, 7},
    {1, 6},
    {3, 5},
    {0, 5},
    {1, 7}
    // You can add more source-destination pairs here
};

// Iterate through each source-destination pair
for (const auto &pair : sourceDestPairs) {
    int s = pair.first; // Source vertex
    int t = pair.second; // Sink vertex

    int rGraph[V][V]; // Residual graph

    // Initialize residual graph with the same capacities as the original graph
    for (int u = 0; u < V; u++)
        for (int v = 0; v < V; v++)
            rGraph[u][v] = graph[u][v];

    int maxFlow = 0;
    int parent[V];
    vector<vector<int>> paths;

    // Calculate edge-disjoint paths using BFS and augmenting paths
    while (bfs(rGraph, s, t, parent)) {
        vector<int> path;
        int pathFlow = INT_MAX;

        // Find the minimum residual capacity along the path
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            pathFlow = min(pathFlow, rGraph[u][v]);
        }

        // Update residual capacities of the edges and reverse edges along the path
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            rGraph[u][v] -= pathFlow;
            rGraph[v][u] += pathFlow;
        }

        maxFlow += pathFlow;

        // Store the path in reverse order
        for (int v = t; v != s; v = parent[v]) {
            path.push_back(v);
        }
        path.push_back(s);
        reverse(path.begin(), path.end());
        paths.push_back(path);
    }

    // Print the number of edge-disjoint paths and each path
```

```
        cout << "There can be a maximum of " << paths.size() << " edge-disjoint paths from vertex
" << s << " to vertex " << t << "." << endl;
        for (const auto &p : paths) {
            cout << "Path: ";
            for (int vertex : p) {
                cout << vertex << " -> ";
            }
            cout << endl;
        }
        cout << endl;
    }

    cout << "****************  End  ******************" << endl;

    return 0;
}
```

```
PS C:\Users\RAMAVATH SANTHOSH\OneDrive\Desktop\ALL SEMs\SEM3\CCN\Day2>  & 'c:\
\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-bc
-MIEngine-Error-ski13qet.uaw' '--pid=Microsoft-MIEngine-Pid-xkcx1rlm.zxt' '--d
There can be a maximum of 2 edge-disjoint paths from vertex 0 to vertex 7.
Path: 0 -> 2 -> 6 -> 7 ->
Path: 0 -> 3 -> 6 -> 5 -> 7 ->

There can be a maximum of 1 edge-disjoint paths from vertex 1 to vertex 6.
Path: 1 -> 2 -> 6 ->

There can be a maximum of 1 edge-disjoint paths from vertex 3 to vertex 5.
Path: 3 -> 6 -> 5 ->

There can be a maximum of 1 edge-disjoint paths from vertex 0 to vertex 5.
Path: 0 -> 2 -> 6 -> 5 ->

There can be a maximum of 1 edge-disjoint paths from vertex 1 to vertex 7.
Path: 1 -> 2 -> 6 -> 7 ->

****************  End  ******************
PS C:\Users\RAMAVATH SANTHOSH\OneDrive\Desktop\ALL SEMs\SEM3\CCN\Day2>
```