data structures and algorithm project

## Project name: mutuals

## Member details:

## 1. Shagil Islam:

## Stream: Computer Science and Engineering (CSE)

## Section: A, Roll no.: 33

## Enrolment no.: 12020002002051

## 2. Sruti Patwari:

## Stream: Computer Science and Engineering (CSE)

## Section: A, Roll no.: 28

## Enrolment no.: 12020002002036

## 3. Chaitak gorai:

## Stream: Computer Science and Engineering (CSE)

## Section: A, Roll no.: 21

## Enrolment no.: 12020002002027

## 4. chirag aggarwal:

## Stream: Computer Science and Engineering (CSE

## Section: A, Roll no.: 38

## Enrolment no.: 12020002002056

## platform used: visual studio code

## language used: c++

### objective

We have created a Social Media Network with the help Data Structures and Algorithms which helps in providing friend-suggestions to any specified user and has a built-in map feature which helps in giving the shortest distance from the set-location of a particular user to all other users which are directly, or indirectly are in contact with the user.

### features, their functions and their working (data structures and algorithms used)

1. **Storing the user details and connectivity**: This Social-media uses Graph Data Structure to store the details of users (with the help of its nodes/vertices) and connectivity and distance, i.e., whether any two users are friends or not and if they are friends then what is the distance between their set locations (with the help of weighted undirected graph edges).
2. **Friend Suggestion**: This social-media makes use of the very popular Breadth-First Search (BFS) Algorithm to traverse the network in “layers” and suggest friends by printing the names of all user which are not friends of the user but are friends of his/her friends. It uses the Queue Data Structure.

Algorithm:

Step 1: SET STATUS = 1 (ready state)

for each node in G

Step 2: Enqueue the starting node A

and set its STATUS = 2

(waiting state)

Step 3: Repeat Steps 4 and 5 until

QUEUE is empty

Step 4: Dequeue a node N. Process it

and set its STATUS = 3

(processed state).

Step 5: Enqueue all the neighbours of

N that are in the ready state

(whose STATUS = 1) and set

their STATUS = 2

(waiting state)

[END OF LOOP]

Step 6: EXIT

1. **Shortest Distance between any two users**: To compute the shortest distance between any two users, the famous Dijkstra’s Shortest Path Graph Algorithm is used.

The Data structures employed wherein are: Priority-Queues or the Min-Heap Data Structure.

Algorithm:

1. While min-heap Q is not empty, pop the node v, that is not already in set S, from Q with the smallest dist (v). In the first run, source node s will be chosen because dist (s) was initialized to 0. In the next run, the next node with the smallest dist value is chosen.
2. Add node v to S, to indicate that v has been visited
3. Update dist values of adjacent nodes of the current node v as follows: for each new adjacent node u,

if dist (v) + weight(u,v) < dist (u), there is a new minimal distance found for u, so update dist (u) to the new minimal distance value;

otherwise, no updates are made to dist (u).

1. The algorithm has visited all nodes in the graph and found the smallest distance to each node. dist now contains the shortest path tree from source s.

Note: The weight of an edge (u,v) is taken from the value associated with (u,v) on the graph.

### code

#include<bits/stdc++.h>

using namespace std;

class WeightedUndirectedGraph

{

    int N;

    vector<pair<int,int>> \*adj;

    priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    public:

    int \*distTo;

    WeightedUndirectedGraph(int n)

    {

        N = n;

        adj = new vector<pair<int,int>>[N+1];

        distTo = new int[N+1];

        for(int i = 0; i<=N; i++)

        {

            distTo[i] = INT\_MAX;

        }

    }

    void addEdge(int u, int v, int wt)

    {

        adj[u].push\_back(make\_pair(v, wt));

        adj[v].push\_back(make\_pair(u, wt));

    }

    void DijkstraShortestPath(int s)

    {

        distTo[s] = 0;

        pq.push(make\_pair(0, s));

        while(!pq.empty())

        {

            int node = pq.top().second;

            int dist = pq.top().first;

            pq.pop();

            for(auto it: adj[node])

            {

                int nextNode = it.first;

                int nextDist = it.second;

                if(distTo[nextNode]>dist + nextDist)

                {

                    distTo[nextNode] = dist + nextDist;

                    pq.push(make\_pair(distTo[nextNode], nextNode));

                }

            }

        }

    }

    void BFS(int s, string name[])

    {

        int friends = adj[s].size();

        bool \*visited = new bool[N+1];

        for(int i = 0; i <= N; i++)

            visited[i] = false;

        queue<int> q;

        // Mark the current node as visited and enqueue it

        visited[s] = true;

        q.push(s);

        // 'i' will be used to get all adjacent

        // vertices of a vertex

        vector<pair<int,int>>::iterator i;

        while(!q.empty())

        {

            // Dequeue a vertex from queue and print it

            int node = q.front();

            q.pop();

            if(friends<0)

            {

                cout << name[node] << " ";

            }

            else

            {

                friends--;

            }

            // Get all adjacent vertices of the dequeued

            // vertex s. If a adjacent has not been visited,

            // then mark it visited and enqueue it

            for (i = adj[node].begin(); i != adj[node].end(); i++)

            {

                if (!visited[i->first])

                {

                    visited[i->first] = true;

                    q.push(i->first);

                }

            }

        }

    }

    /\*void DFS(int s)

    {

        bool \*explored = new bool[N+1];

        for(int i = 0; i <= N; i++)

            explored[i] = false;

        stack<int> st;

        // Mark the current node as visited and push it into stack

        explored[s] = true;

        st.push(s);

        // 'i' will be used to get all adjacent

        // vertices of a vertex

        vector<int>::iterator i;

        while(!st.empty())

        {

            // Pop a vertex from stack and print it

            int node = st.top();

            st.pop();

            cout << node << " ";

            // Get all adjacent vertices of the destacked

            // vertex s. If a adjacent has not been visited,

            // then mark it visited and enstack it

            for (i = adj[node].begin(); i != adj[node].end(); i++)

            {

                if (!explored[\*i])

                {

                    explored[\*i] = true;

                    st.push(\*i);

                }

            }

        }

    }\*/

};

int main()

{

    int n;

    cout<<"Enter the number of nodes: ";

    cin>>n;

    cout<<endl;

    WeightedUndirectedGraph fb(n);

    string name[n+1];

    name[0]="";

    cout<<"Enter the name of the users: \n";

    cin.clear();

    cin.sync();

    for(int i=1; i<=n; i++)

    {

        getline(cin, name[i]);

    }

    cout<<endl;

    for(int i=1; i<=n; i++)

    {

        cout<<i<<": "<<name[i]<<endl;

    }

    cout<<"\nEnter the edges and edge weights(u,v,wt): \n";

    while(true)

    {

        int u, v, wt;

        cin>>u>>v>>wt;

        if(u==-1)

        {

            break;

        }

        else

        {

            fb.addEdge(u,v,wt);

        }

    }

    cout<<"\nNode of the person whose friend suggestion is required: ";

    int node;

    cin>>node;

    cout<<"\nFriend Suggestion:\n";

    fb.BFS(node, name);

    fb.DijkstraShortestPath(node);

    cout<<"\n\nShortest Distance to all other user's location:\n";

    for(int i=1; i<=n; i++)

    {

        if(i==node)

        {

            continue;

        }

        else

        {

            cout<<name[i]<<"\t"<<fb.distTo[i]<<endl;

        }

    }

    return 0;

}

### input and output





