EAST WEST UNIVERSITY

Department of CSE

Course Code: CSE246
Course Title: Algorithms
Lab Assignment Report

Assignment: 1
Assignment Title: Dijkstra of all possible shortest path with cost

Submitted by-
Noushin Pervez
Id: 2020-1-60-189
Section: 3
Summer 2022

Submitted to-
Jesan Ahammed Ovi
Senior Lecturer
Department of Computer Science and Engineering
East West University

Submission Date: 07-09-2022

Problem Statement: Dijkstra of all possible shortest paths with cost

Description:
Given a graph and a source vertex of a graph, we find shortest paths from the source to all vertices in the given graph.

The Dijkstra algorithm is an iterative algorithm that provides us with the shortest path from one particular starting node to all other nodes in the graph. It is similar to the results of Breadth First Search(BFS).

Explanation:
At first, we initialize a structure to store the nodes and costs.

```
struct node{
    int nd, weight;
}tmp;
```

We initialize a min priority queue to pop the minimum cost and vertex each time until the priority queue is empty, a vector to store the information from the structure, two arrays to store the cost and the parent of each node. Every item in the priority queue is a pair (cost, vertex). Cost is used as the first item. By default, the first item is used to compare.

```
typedef pair<int, int>pi;

vector<node>adj[100];
int dis[100], par[100];
priority_queue<pi, vector<pi>, greater<pi>>q;
```

We take the number of vertices and edges as input. Now, we start a for loop from 0 to edges which input the edges of two nodes and their corresponding weights of them. A node type variable is used to store the information and then the variable is pushed to the vector.

```
int main(){
    int n, edge, u, v, w;
    cout << "Enter number of vertex: ";
    cin >> n;
    cout << "Enter number of edge: ";
    cin >> edge;

    for(int i = 0; i < edge; i++){
        cin >> u >> v >> w;
        tmp.nd = v;
        tmp.weight = w;
        adj[u].push_back(tmp);
    }
```

We initialize the cost of all nodes as infinity(99999) and the parent of all nodes as -1.

```
for(int i = 0; i < n; i++){
    dis[i] = 99999;
    par[i] = -1;
}
```

Now we input the source vertex and make the cost of the source vertex 0. Now we pair the cost of source and source vertex and push it to the min priority queue.

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

dis[src] = 0;
q.push(make_pair(dis[src], src));
```

We use a function to apply the Dijkstra algorithm to the given graph.
While the priority queue does not become empty, we extract the minimum cost vertex from the priority queue. Let the extracted vertex be u.
We loop through all adjacent vertices of u. We initialize a node type variable to store the information of adjacent vertices. Now, we get two integer type variables (cost and vertex) from it.
We check if there is a shorter path to v through u and update the cost and parent of v. We insert this new information into the priority queue.

```
void dijkstra(){
    while(!q.empty()){
        int u = q.top().second;
        q.pop();

        for(int i = 0; i < adj[u].size(); i++){
            node t = adj[u][i];
            int v = t.nd;
            int w = t.weight;

            if(dis[v] > dis[u] + w){
                dis[v] = dis[u] + w;
                par[v] = u;

                q.push(make_pair(dis[v], v));
            }
        }
    }
}
```

To print the path from the source to all other vertices, we use a function with the parameter of an integer target and a parent array. We initialize a stack to print the paths. At first, we push the target vertex into the stack. Now, while the parent of the target vertex does not become 0, we update the target variable to the parent of the target vertex and push the new target vertex to the stack.

To print the path, while the stack does not become empty, we print the stack elements and pop each vertex. Now, we get the path from the source to all other vertices.

```
void path(int target, int par[]){
    stack<int>st;

    st.push(target);

    while(par[target] != -1){
        target = par[target];
        st.push(target);
    }

    while(!st.empty()){
        if(st.size() == 1)
            cout << st.top() << " ";
        else
            cout << st.top() << " -> ";

        st.pop();
    }
}
```

In the main function, we call the dijkstra() function. We iterate from 1 to n and call the path(i, par) function to print the path and print the cost corresponding to it.

```
dijkstra();

printf("Vertex Distance from Source:\n");
for(int i = 1; i < n; i++){
    path(i, par);
    cout << dis[i] << endl;
}
```

Time Complexity: O(ElogV)

Code:

```
#include<bits/stdc++.h>

using namespace std;

struct node{
   int nd, weight;
}tmp;

typedef pair<int, int>pi;

vector<node>adj[100];
int dis[100], par[100];
priority_queue<pi, vector<pi>, greater<pi>>q;

void path(int target, int par[]){
   stack<int>st;

   st.push(target);

   while(par[target] != -1){
      target = par[target];
      st.push(target);
   }

   while(!st.empty()){
      if(st.size() == 1)
         cout << st.top() << " ";
      else
         cout << st.top() << " -> ";

      st.pop();
   }
}

void dijkstra(int n, int src){
   while(!q.empty()){
      int u = q.top().second;
      q.pop();
```

```
        for(int i = 0; i < adj[u].size(); i++){
            node t = adj[u][i];
            int v = t.nd;
            int w = t.weight;

            if(dis[v] > dis[u] + w){
                dis[v] = dis[u] + w;
                par[v] = u;

                q.push(make_pair(dis[v], v));
            }
        }
    }
}

int main(){
    int n, edge, u, v, w;
    cout << "Enter number of vertex: ";
    cin >> n;
    cout << "Enter number of edge: ";
    cin >> edge;

    for(int i = 0; i < edge; i++){
        cin >> u >> v >> w;
        tmp.nd = v;
        tmp.weight = w;
        adj[u].push_back(tmp);
    }

    for(int i = 0; i < n; i++){
        dis[i] = 99999;
        par[i] = -1;
    }

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

    dis[src] = 0;
```

```
    q.push(make_pair(dis[src], src));

    dijkstra(n, src);

    printf("Vertex Distance from Source:\n");
    for(int i = 1; i < n; i++){
        path(i, par);
        cout << dis[i] << endl;
    }
}
```

Output:

We input the number of vertices and edges. Now, we input two vertices that are connected and the costs. We enter the source vertex to get the paths from source node to all other nodes.