Experiment 3.2

Student Name: Pranjal Kumar **UID:** 20BCS3504
Branch: CSE **Section/Group:** 607 /B
Semester: 05 **Date of Performance:** 04/11/2022
Subject Name: Design & Analysis Algorithm **Subject Code:** 20CSP-312

1.AIM

Code and analyze to find shortest paths in a graph with positive edge weights using Dijkstra's algorithm.

2.TASK TO BE DONE

Implementing and analyzing to find shortest paths in a graph with positive edge weights using Dijkstra's algorithm.

3.ALGORITHM/FLOWCHART

1) Initialize distances of all vertices as infinite.
2) Create an empty priority_queue pq. Every item of pq is a pair (weight, vertex). Weight (or distance) is used as first item of pair as first item is by default used to compare two pairs.
3) Insert source vertex into pq and make its distance as 0.
4) While either pq doesn't become empty
5) Extract minimum distance vertex from pq. Let the extracted vertex be u.
6) Loop through all adjacent of u and do. following for every vertex v.If there is a shorter path to v through u. If dist[v] > dist[u] + weight(u, v)
(i) Update distance of v, i.e., do dist[v] = dist[u] + weight(u, v)
(ii) Insert v into the pq (Even if v is already there)
7) Print distance array dist[] to print all shortest paths.

4.STEPS FOR EXPIREMENT/PRACTICAL/CODE

```
#include <bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f

typedef pair<int, int> iPair;

class Graph {
int V;
list<pair<int, int> >* adj;

public:
Graph(int V);
void addEdge(int u, int v, int w);
void shortestPath(int s);
};

Graph::Graph(int V)
{
this->V = V;
adj = new list<iPair>[V];
}

void Graph::addEdge(int u, int v, int w)
{
adj[u].push_back(make_pair(v, w));
adj[v].push_back(make_pair(u, w));
}
void Graph::shortestPath(int src)
{
```

```
priority_queue<iPair, vector<iPair>, greater<iPair> >
        pq;

vector<int> dist(V, INF);

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

while (!pq.empty()) {

        int u = pq.top().second;
        pq.pop();

        list<pair<int, int> >::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i) {

                int v = (*i).first;
                int weight = (*i).second;

                if (dist[v] > dist[u] + weight) {

                        dist[v] = dist[u] + weight;
                        pq.push(make_pair(dist[v], v));
                }
        }
}

printf("Vertex Distance from Source\n");
for (int i = 0; i < V; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
```

```
}

int main()
{

int V = 9;
Graph g(V);

g.addEdge(0, 1, 4);
g.addEdge(0, 7, 8);
g.addEdge(1, 2, 8);
g.addEdge(1, 7, 11);
g.addEdge(2, 3, 7);
g.addEdge(2, 8, 2);
g.addEdge(2, 5, 4);
g.addEdge(3, 4, 9);
g.addEdge(3, 5, 14);
g.addEdge(4, 5, 10);
g.addEdge(5, 6, 2);
g.addEdge(6, 7, 1);
g.addEdge(6, 8, 6);
g.addEdge(7, 8, 7);

g.shortestPath(0);

return 0;
}
```

```cpp
#include <bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f

typedef pair<int, int> iPair;

class Graph {
    int V;
    list<pair<int, int> >* adj;

public:
    Graph(int V);
    void addEdge(int u, int v, int w);
    void shortestPath(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<iPair>[V];
}

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

void Graph::shortestPath(int src)
{
    priority_queue<iPair, vector<iPair>, greater<iPair> >
        pq;

```

```
    vector<int> dist(V, INF);

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

    while (!pq.empty()) {

        int u = pq.top().second;
        pq.pop();

        list<pair<int, int> >::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i) {

            int v = (*i).first;
            int weight = (*i).second;

            if (dist[v] > dist[u] + weight) {

                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
            }
        }
    }

    printf("Vertex Distance from Source\n");
    for (int i = 0; i < V; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

int main()
{

    int V = 9;
    Graph g(V);

    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);

    g.shortestPath(0);

    return 0;
}
```

4.OBSERVATIONS/DISCUSSIONS/COMPLEXITY ANALYSIS

The time complexity remains O(ELogV)) as there will be at most O(E) vertices in priority queue and O(Log E) is same as O(Log V)

5.OUTPUT/RESULT

```
Vertex Distance from Source
0                 0
1                 4
2                 12
3                 19
4                 21
5                 11
6                 9
7                 8
8                 14


...Program finished with exit code 0
Press ENTER to exit console.
```

LEARNING OUTCOMES

1.Learnt about Dijkstra's algorithm and its implementation.

2. Also, learnt about how to analyze time and space complexity.

EVALUATION GRID (To be created as per the SOP and Assessment guidelines by the faculty):

Sr. No.	Parameters	Marks Obtained	Maximum Marks
1.			
2.			
3.			