NAME:	Eshan Bhuse
UID:	2021300013
SUBJECT	Design and analysis of algorithm
EXPERIMENTNO:	6
AIM:	**To understand and implement Dijkstra's Algorithm.**
ALGORITHM:	function dijkstra(G, S) for each vertex V in G distance[V] <- infinite previous[V] <- NULL If V != S, add V to Priority Queue Q distance[S] <- 0 while Q IS NOT EMPTY U <- Extract MIN from Q for each unvisited neighbour V of U tempDistance <- distance[U] + edge_weight(U, V) if tempDistance < distance[V] distance[V] <- tempDistance previous[V] <- U return distance[], previous[]

THEORY:

Introduction to Dijkstra's Algorithm

Now that you know the basic concepts of graphs, let's start diving into this amazing algorithm.

- Purpose and Use Cases
- History
- Basics of the Algorithm
- Requirements

Purpose and Use Cases:

With Dijkstra's Algorithm, you can find the shortest path between nodes in a graph. Particularly, you can **find the shortest path from a node (called the "source node") to all other nodes in the graph**, producing a shortest-path tree.

This algorithm is used in GPS devices to find the shortest path between the current location and the destination. It has broad applications in industry, specially in domains that require modeling networks.

History:

This algorithm was created and published by Dr. Edsger W. Dijkstra, a brilliant Dutch computer scientist and software engineer.
In 1959, he published a 3-page article titled "A note on two problems in connexion with graphs" where he explained his new algorithm.

Basics of Dijkstra's Algorithm:

Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.

The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.

Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.

The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

Requirements:

Dijkstra's Algorithm can only work with graphs that have **positive** weights. This is because, during the process, the weights of the edges have to be added to find the shortest path.

If there is a negative weight in the graph, then the algorithm will not work properly. Once a node has been marked as "visited", the current path to that node is marked as the shortest path to reach that node. And negative weights can alter this if the total weight can be decremented after this step has occurred.

Working of Dijkstra's Algorithm:

Highlights:

1. Greedy Algorithm
2. Relaxation

Dijkstra's Algorithm requires a graph and source vertex to work. The algorithm is purely based on greedy approach and thus finds the locally optimal choice(local minima in this case) at each step of the algorithm.

In this algorithm each vertex will have two properties defined for it-

- **Visited property**:-
  - This property represents whether the vertex has been visited or not.
  - We are using this property so that we don't revisit a vertex.
  - A vertex is marked visited only after the shortest path to it has been found.
- **Path property**:-
  - This property stores the value of the current minimum path to the vertex. Current minimum path means the shortest way in which we have reached this vertex till now.
  - This property is updated whenever any neighbour of the vertex is visited.
  - The path property is important as it will store the final answer for each vertex.

Initially all the vertices are marked unvisited as we have not visited any of them. Path to all the vertices is set to infinity excluding the source vertex. Path to the source vertex is set to zero (0).

Then we pick the source vertex and mark it visited. After that all the neighbours of the source vertex are accessed and relaxation is performed on each vertex. Relaxation is the process of trying to lower the cost of reaching a vertex using another vertex.

In relaxation, the path of each vertex is updated to the minimum value amongst the current path of the node and the sum of the path to the previous node and the path from the previous node to this node.

Assume that p[v] is the current path value for node v, p[n] is the path

value upto the previously visited node n, and w is the weight of the edge between the current node and previously visited node(edge weight between v and n)

Mathematically, relaxation can be represented as: **p[v]=minimum(p[v],p[n]+w)**

Then in every subsequent step, an unvisited vertex with the least path value is marked visited and its neighbour's paths updated.

The above process is repeated till all the vertices in the graph are marked visited.

Whenever a vertex is added to the visited set, the path to all of its neighbouring vertices is changed according to it.

If any of the vertex is not reachable (disconnected component), its path remains infinity. If the source itself is a disconnected component, then the path to all other vertices remains infinity.

Dijkstra's Algorithm Complexity:

Time Complexity: O(E log(V))

Where, E is the number of edges and V is the number of vertices.

Space Complexity: O(V)

Dijkstra's Algorithm Applications:

- To find the shortest path
- In social networking applications
- In a telephone network
- To find the locations in the map

PROGRAM:

```
#include<bits/stdc++.h>
using namespace std;
int V;
int minDistance(int distance[],bool sptSet[]){

    int minDist=INT_MAX;
    int minVertex=0;
    for(int i=0;i<V;i++)
    {
        if(sptSet[i]==false && distance[i]<=minDist){
            minDist=distance[i];
            minVertex=i;
        }
    }
    return minVertex;
}
void printSolution(int dist[])
{
    cout << "Vertex \t\t\t Distance from Source" << endl;
    for (int i = 0; i < V; i++)
        cout << i << " \t\t\t\t" << dist[i] << endl;
}
void dijkstra(int **graph, int src)
{
    int dist[V];
    bool sptSet[V];
    for (int i = 0; i < V; i++){
        dist[i] = INT_MAX;
        sptSet[i] = false;
    }
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++)
    {
        int u = minDistance(dist, sptSet);
        sptSet[u]=true;
        for (int v = 0; v < V; v++)
        if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
```

```
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++)
    {
        int u = minDistance(dist, sptSet);
        sptSet[u]=true;
        for (int v = 0; v < V; v++)
        if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    printSolution(dist);
}
  int main()
  {
    cout<<"Enter the number of vertices :";
    cin>>V;
    int **graph=new int*[V];
    for(int i=0;i<V;i++)
    {
        graph[i]=new int[V];
    }
    for(int i=0;i<V;i++){
        for(int j=0;j<V;j++){
            graph[i][j]=0;
        }
    }
    cout<<"Enter the number of edges :";
    int e; cin >> e;
    for(int i=0;i<e;i++)
    {
        cout<<"\nEnter the Vertices of the edge "<<i<<" :";
        int a,b,w;
        cin>>a>>b;
        cout<<"Enter the Weight of the edge "<<i<<" :";
        cin>>w;
        graph[a][b]=w;
        graph[b][a]=w;
    }
    dijkstra(graph,0);
    return 0;
}

```

```
Enter the number of vertices :3
Enter the number of edges :3

Enter the Vertices of the edge 0 :0 2
Enter the Weight of the edge 0 :5

Enter the Vertices of the edge 1 :0 1
Enter the Weight of the edge 1 :3

Enter the Vertices of the edge 2 :2 1
Enter the Weight of the edge 2 :4
Vertex                  Distance from Source
0                               0
1                               3
2                               5


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

OBSERVATION:	After implementing dijkstra's algorithm, I observed that it guarantees finding the shortest path if the graph does not have negative weighted edges and it works best when the number of edges are less than the number of vertices. It is also the best algorithm for finding the shortest job and it can be made efficient using priority queues.
CONCLUSION:	Successfully performed the experiment to implement Dijkstra Algorithm to find the shortest path between source vertex and destination vertex.