ASSIGNMENT-8
Implementation of Dijikstra's algorithm

AIM: Represent a given graph using adjacency matrix /adjacency list and find the shortest path using Dijkstra's algorithm (single source all destination).

OBJECTIVE:

1. To understand the application of Dijkstra's algorithm

THEORY:

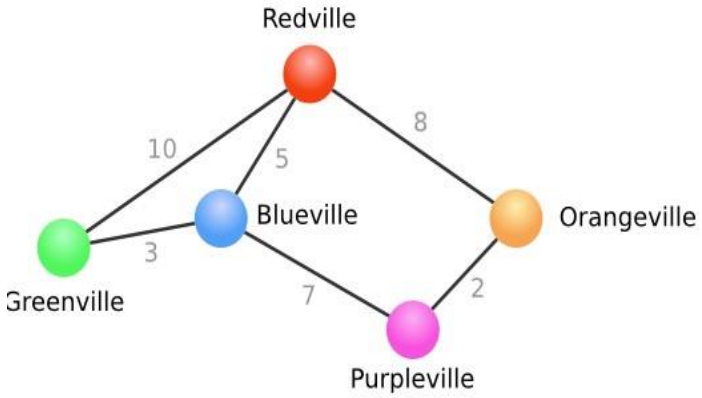
1. Explain in brief with examples how to find the shortest path using Dijkstra's algorithm.

Definition of Dijkstra's Shortest Path

1. To find the shortest path between points, the weight or length of a path is calculated as the sum of the weights of the edges in the path.
2. A path is a shortest if there is no path from x to y with lower weight.
3. Dijkstra's algorithm finds the shortest path from x to y in order of increasing distance from x. That is, it chooses the first minimum edge, stores this value and adds the next minimum value from the next edge it selects.
4. It starts out at one vertex and branches out by selecting certain edges that lead to new vertices.
5. It is similar to the minimum spanning tree algorithm, in that it is "greedy", always choosing the closest edge in hopes of an optimal solution.

Example:

It is easiest to think of the geographical distances, with the vertices being places, such as cities.



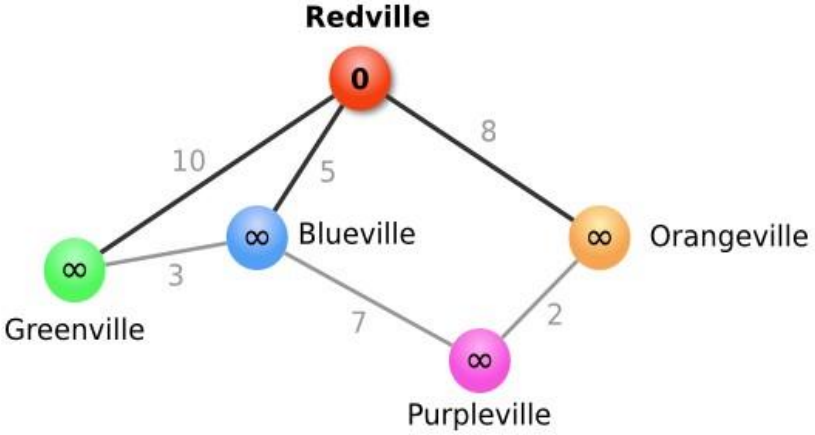
Imagine you live in Redville, and would like to know the shortest way to get to the surrounding towns: Greenville, Blueville, Orangeville, and Purpleville. You would be confronted with problems like: Is it faster to go through Orangeville or Blueville to get to Purpleville? Is it

faster to take a direct route to Greenville, or to take the route that goes through Blueville? As long as you knew the distances of roads going directly from one city to another, Dijkstra's algorithm would be able to tell you what the best route for each of the nearby towns would be.

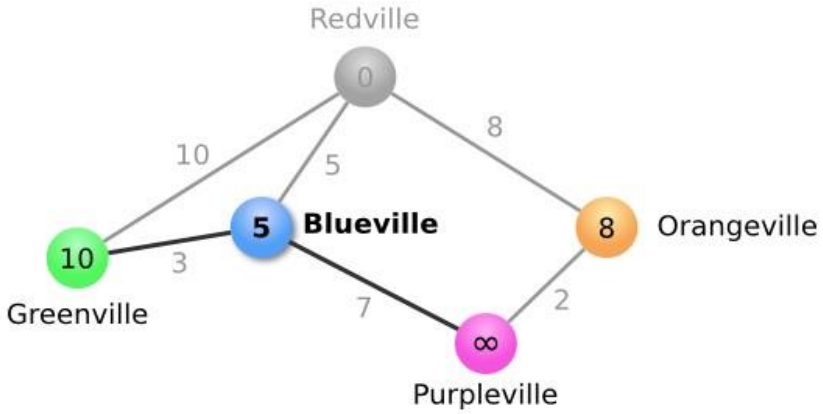
- Begin with the source node (city), and call this the current node. Set its value to 0. Set the value of all other nodes to infinity. Mark all nodes as unvisited.
- For each unvisited node that is adjacent to the current node (i.e. a city there is a direct route to from the present city), do the following. If the value of the current node plus the value of the edge is less than the value of the adjacent node, change the value of the adjacent node to this value. Otherwise leave the value as is.
- Set the current node to visited. If there are still some unvisited nodes, set the unvisited node with the smallest value as the new current node, and go to step 2. If there are no unvisited nodes, then we are done.

In other words, we start by figuring out the distance from our hometown to all of the towns we have a direct route to. Then we go through each town, and see if there is a quicker route through it to any of the towns it has a direct route to. If so, we remember this as our current best route.

Step I:
We set Redville as our current node. We give it a value of 0, since it doesn't cost anything to get to it from our starting point. We assign everything else a value of infinity, since we don't yet know of a way to get to them.

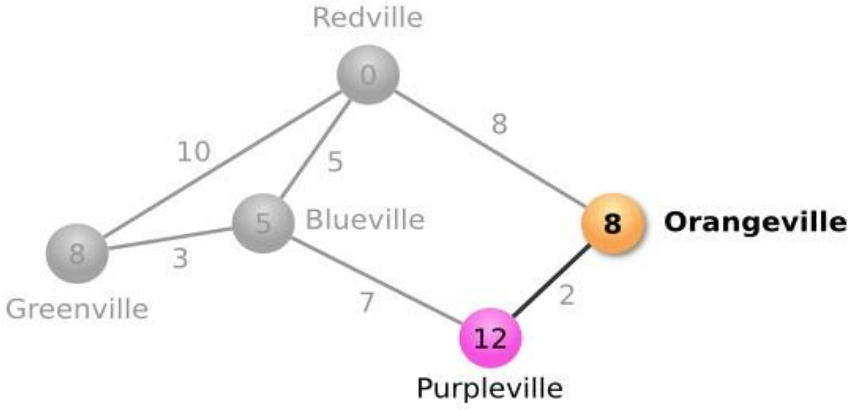


Step II:
Next, we look at the unvisited cities our current node is adjacent to. This means Greenville, Blueville and Orangeville. We check whether the value of the connecting edge, plus the value of our current node, is less than the value of the adjacent node, and if so we change the value. In this case, for all three of the adjacent nodes we should be changing the value, since all of the adjacent nodes have the value infinity. We change the value to the value of the current node (zero) plus the value of the connecting edge (10 for Greenville, 5 for Blueville, 8 for Orangeville). We now mark Redville as visited, and set Blueville as our current node since it has the lowest value of all unvisited nodes.



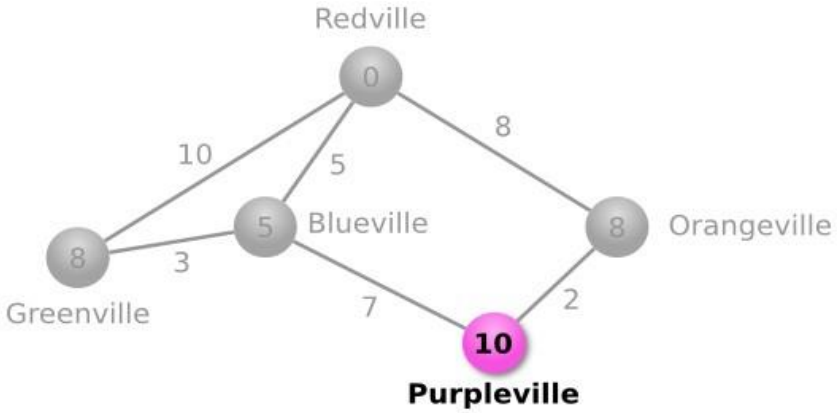
Step III:

The unvisited nodes adjacent to Blueville, our current node, are Purpleville and Greenville. So we want to see if the value of either of those cities is less than the value of Blueville plus the value of the connecting edge. The value of Blueville plus the value of the road to Greenvile is 5 + 3 = 8. This is less than the current value of Greenville (10), so it is shorter to go through Blueville to get to Greenville. We change the value of Greenville to 8, showing we can get there with a cost of 8. For Purpleville, 5 + 7 = 12, which is less than Purpleville‘s current value of infinity, so we change its value as well? We mark Blueville as visited. There are now two unvisited nodes with the lowest value (both Orangeville and Greenville have value 8). We can arbitrarily choose Greenville to be our next current node. However, there are no unvisited nodes adjacent to Greenville! We can mark it as visited without making any other changes, and make Orangeville our next current node.



Step IV:

There is only one unvisited node adjacent to Orangeville. If we check the values, Orangeville plus the connecting road is 8 + 2 = 10, Purpleville's value is 12, and so we change Purpleville's value to 10. We mark Orangeville as visited, and Purpleville is our last unvisited node, so we make it our current node. There are no unvisited nodes adjacent to Purpleville, so we're done!



All above steps can be simply put in a tabular form like this:

Current	Visited	Red	Green	Blue	Orange	Purple	Description
Red		0	Infinity	Infinity	Infinity	Infinity	Initialize Red as current, set initial values
Red		0	10	5	8	Infinity	Change values for Green, Blue, Orange
Blue	Red	0	10	5	8	Infinity	Set Red as visited, Blue as current
Blue	Red	0	8	5	8	12	Change value for Purple
Green	Red, Blue	0	8	5	8	12	Set Blue as visited, Green as current
Orange	Red, Blue, Green	0	8	5	8	12	Set Green as visited, Orange as current
Orange	Red, Blue, Green	0	8	5	8	10	Change value for Purple
Purple	Red, Blue, Green, Orange	0	8	5	8	10	Set Orange as visited, Purple as current
	Red, Blue, Green, Orange, Purple	0	8	5	8	10	Set Purple as visited

ALGORITHM:

College Area represented by Graph.
A graph G with N nodes is maintained by its adjacency matrix Cost. Dijkstra's algorithm find shortest path matrix D of Graph G.
Staring Node is 1.

Step 1: Repeat Step 2 for I = 1 to N
D[I] = Cost[1][I].
Step 2: Repeat Steps 3 & 4 for I = 1 to N

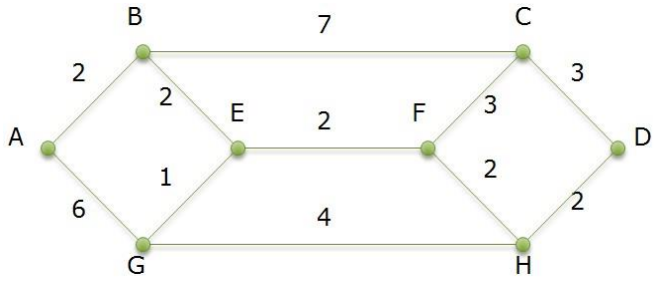
Step 3: Repeat Steps 4 for J = 1 to N
Step 4: If D[J] > D[I] + D[I][J]
Then D[J] = D[I] + D[I][J] Step
5: Stop.

INPUT:

The graph in the form of edges, nodes and corresponding weights, the source node and destination node.

OUTPUT:

The shortest path and length of the shortest path

INPUT	**OUTPUT**	Remark
	Shortest Path is A-B-E-F-H-D Cost - 10	Consider Source Vertex as node A and Destination Vertex as node D

FAQs:

1. What is shortest path?
2. What are the graph representation techniques?
3. What is adjacency Matrix?
4. What is adjacency list?
5. What is adjacency Multi-list?

PROGRAM CODE-

```
#include <iostream>
using namespace std;
class graph{
    int g[20][20];
    int e,v;
    public:
    void accept();
    void display();
    void dijkstra(int start);
};
void graph:: accept(){
    int src, dest, cost, i,j;
    cout<<"Enter the number of vertices:  ";
    cin>>v;
    cout<<"Enter the number of edges:  ";
    cin>>e;
    for(i=0; i<v; i++){
        for(j=0; j<v;j++){
            g[i][j]=0;
        }
    }
    for(i=0; i<e; i++){
        cout<<"\nEnter source vertex:  ";
        cin>>src;
        cout<<"Enter destination vertex:  ";
        cin>>dest;
        cout<<"Enter the cost of the edge:  ";
        cin>>cost;
        g[src][dest]=cost;
        //g[dest][src]=cost;
    }
}
void graph::display(){
    int i,j;
    for(i=0; i<v; i++){
        cout<<endl;
        for(j=0; j<v; j++){
            cout<<g[i][j]<<"\t";
        }
    }
}
void graph::dijkstra(int start){
    int r[20][20], visited[20],distance[20],from[20],i,j,cnt,mindst,next;
    for(i=0; i<v; i++){
        for(j=0; j<v; j++){
```

```
            if(g[i][j]==0){
                r[i][j]=999;
            }
            else{
                r[i][j]=g[i][j];
            }
        }
    }
    for(i=0; i<v; i++){
        visited[i]=0;
        from[i]=start;
        distance[i]=r[start][i];
    }
    distance[start]=0;
    visited[start]=1;
    cnt=v;
    while(cnt>0){
        mindst=999;
        for(i=0; i<v; i++){
            if((mindst>distance[i]) && visited[i]==0){
                mindst=distance[i];
                next=i;
            }
        }
        visited[next]=1;
        for(i=0; i<v; i++){
            if(visited[i]==0 && distance[i]>(mindst+r[next][i])){
                distance[i]=mindst+r[next][i];
                from[i]=next;
            }
        }
        cnt--;
    }
    for(i=0; i<v; i++){
        cout<<"\nDistance of "<<i<< " from "<<start<<" is
"<<distance[i]<<endl<<"Path "<<i;
        j=i;
        do{
            j=from[j];
            cout<<"<-"<<j;
        }
        while(j!=start);
    }
}
int main()
{
    graph g;
```

```
    int s;
    g.accept();
    g.display();
    cout<<"\nEnter the starting vertex: ";
    cin>>s;
    g.dijkstra(s);
return 0;
}
```

OUTPUT-

Enter the number of vertices: 7
Enter the number of edges: 12

Enter source vertex: 1
Enter destination vertex: 2
Enter the cost of the edge: 2

Enter source vertex: 2
Enter destination vertex: 5
Enter the cost of the edge: 10

Enter source vertex: 1
Enter destination vertex: 4
Enter the cost of the edge: 1

Enter source vertex: 2
Enter destination vertex: 4
Enter the cost of the edge: 3

Enter source vertex: 4
Enter destination vertex: 5
Enter the cost of the edge: 2

Enter source vertex: 5
Enter destination vertex: 7
Enter the cost of the edge: 6

Enter source vertex: 4
Enter destination vertex: 7
Enter the cost of the edge: 4

```
Enter source vertex:  4
Enter destination vertex:  6
Enter the cost of the edge:  8

Enter source vertex:  7
Enter destination vertex:  6
Enter the cost of the edge:  1

Enter source vertex:  4
Enter destination vertex:  3
Enter the cost of the edge:  2

Enter source vertex:  3
Enter destination vertex:  6
Enter the cost of the edge:  5

Enter source vertex:  3
Enter destination vertex:  1
Enter the cost of the edge:  4

0    0    0    0    0    0    0
0    0    2    0    1    0    0
0    0    0    0    3    10    0
0    4    0    0    0    0    5
0    0    0    2    0    2    8
0    0    0    0    0    0    0
0    0    0    0    0    0    0
Enter the starting vertex: 1

Distance of 0 from 1 is 999
Path 0<-1
Distance of 1 from 1 is 0
Path 1<-1
Distance of 2 from 1 is 2
Path 2<-1
Distance of 3 from 1 is 3
Path 3<-4<-1
Distance of 4 from 1 is 1
Path 4<-1
Distance of 5 from 1 is 3
Path 5<-4<-1
Distance of 6 from 1 is 8
Path 6<-3<-4<-1
```