Date-

Assignment No. :

Problem Statement:

Program in C to find the shortest path between the nodes of a given graph by Dijkstra’s algorithm.

Theory:

Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. Like Prim’s MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.
For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities.

Dijkstra's algorithm initially marks the distance (from the starting point) to every other intersection on the map with infinity. This is done not to imply there is an infinite distance, but to note that those intersections have not yet been visited; some variants of this method simply leave the intersections' distances unlabeled. Now, at each iteration, select the current intersection. For the first iteration, the current intersection will be the starting point, and the distance to it (the intersection's label) will be zero. For subsequent iterations (after the first), the current intersection will be a closest unvisited intersection to the starting point (this will be easy to find).

From the current intersection, update the distance to every unvisited intersection that is directly connected to it. This is done by determining the sum of the distance between an unvisited intersection and the value of the current intersection, and relabeling the unvisited intersection with this value (the sum), if it is less than its current value. In effect, the intersection is relabeled if the path to it through the current intersection is shorter than the previously known paths.

Algorithm:

Input specification:

1. I : The incidence matrix of dimension (n x n) of the given graph
2. v_S : The source vertex to start the search from

Output specification:

1. A two dimensional array I[1..n][1..n] whose starting index is 1 and ending index is n, size of the array being (n x n).
2. A stack to store the intermediate vertices, say S.

Steps:

Algorithm for method main():

```
Step 1 :    Print "Enter the no. of vertices: "
Step 2 :    Input n
Step 3 :    If(n < 2 || n>10) Then
Step 4 :    Print "Number of vertices must be >= 2 and <=10"
Step 5 :    return 1
            [End of If structure]
Step 6 :    Print "Enter the adjacency matrix:"
Step 7 :    Repeat through step 8 to step 12 For i=0 to n-1
Step 8 :    Repeat through step 8 to step 11 For j=0 to n-1
Step 9 :    Print "Enter the element of row"(i+1)" and column "(j+1)
Step 10 :   Input G[i][j])
Step 11 :   Set j=j+1
            [End of inner For loop]
Step 12 :   Set i=i+1
            [End of outer For loop]
Step 13 :   Print "The entered matrix is: "
Step 14 :   Print "Enter the starting node: "
Step 15 :   Input u
Step 16 :   If(u < 1 || u > n) Then
Step 17 :   Print "Bad starting vertex : " u " Should be between 1 and " n
Step 18 :   return 1
            [End of If structure]
Step 19 :   dijkstra(G,n,u-1) //Calling of the function dijkstra
            [End of method main()]
```

Algorithm for method dijkstra(G[MAX][MAX], n, startnode):

```
Step 1 :    Set k=0
Step 2 :    Repeat through step 3 to step 7 For i=0 to n-1
Step 3 :    Repeat through step 4 to step 7 For j=0 to n-1
Step 4 :    If(G[i][j]==0) Then
Step 5 :    cost[i][j]=INFINITY //INFINITY is a constant with an extreme value
Step 6 :    Else
Step 7 :    cost[i][j]=G[i][j]
            [End of If-Else structure]
            [End of inner For loop]
            [End of outer For loop]
Step 8 :    Repeat through step 9 to step 12 For i=0 to n-1
Step 9 :    Set distance[i]=cost[startnode][i]
Step 10 :   Set pred[i]=startnode
Step 11 :   Set visited[i]=0
Step 12 :   Set i=i+1
            [End of For loop]
Step 13 :   Set distance[startnode]=0
Step 14 :   Set visited[startnode]=1
Step 15 :   Set count=1
Step 16 :   Repeat through step 16 to step 29 While(count < n-1)
Step 17 :    Set mindistance=INFINITY
Step 18 :    Repeat through step 18 to step 22 For i=0 to n-1
Step 19 :    If(distance[i] < mindistance&&!visited[i]) Then
Step 20 :    Set mindistance=distance[i]
Step 21 :    Set nextnode=i
             [End of If structure]
Step 22 :    Set i=i+1
             [End of For loop]
Step 23 :    Set visited[nextnode]=1;
Step 24 :    Repeat through step 25 to step 12 For i=0 to n-1
Step 25 :    If(!visited[i]) Then
Step 26 :    If(mindistance+cost[nextnode][i] < distance[i]) Then
Step 27 :    Set distance[i]=mindistance+cost[nextnode][i]
Step 28 :    Set pred[i]=nextnode
             [End of inner If structure]
             [End of outer If structure]
Step 29 :    Set i=i+1
             [End of For loop]
Step 30 :    Count=count+1
             [End of While loop]
Step 31 :    Repeat through step 32 to step 45 For i=0 to n-1
Step 32 :    If(i!=startnode) Then
Step 33 :    Print "Distance of "i" = "distance[i])
Step 34 :    Set j=i
Step 35 :    Print "Path = "
             [Starting of Do-While loop]
Step 36 :    Set j=pred[j]
Step 37 :    a[k]=j
Step 38 :    k=k+1
Step 39 :    While(j!=startnode)
Step 40 :    s=k-1
Step 41 :    Repeat through step 42 to step 43 For k=s to 0
Step 42 :    Print a[k] "->"
Step 43 :    Set k=k+1
             [End of For loop]
Step 44 :    Print i
Step 45 :    Set k=0
             [End of If structure]
Step 46 :    Set i=i+1
             [End of For loop]
             [End of method dijkstra()]
```

Source Code:

```
#include<stdio.h>
#define INFINITY 9999
#define MAX 10
void dijkstra(int G[MAX][MAX], int n, int startnode);

int main()
{
	int G[MAX][MAX], i, j, n, u;
	printf("Enter the no. of vertices: ");
	scanf("%d", &n);
	if(n < 2 || n>10){//Checking for vertex no
		printf("\nNumber of vertices must be >= 2 and <=10");
		return 1;
	}
	printf("\nEnter the adjacency matrix:\n");
	for(i=0;i < n;i++)
	{
		for(j=0;j < n;j++)
		{
			printf("Enter the element of row %d and column %d:
			",(i+1),(j+1));
			scanf("%d", &G[i][j]);
		}
	}
	printf("The entered matrix is:\n");//Traversing the entered vertex
	for(i=0;i < n;i++)\
	{
		for(j=0;j<n;j++)
		{
```

```
                printf("%d ", G[i][j]);
            }
            printf("\n");
        }
        printf("\nEnter the starting node: ");
        scanf("%d", &u);
   if(u < 1 || u > n){
      printf("\nBad starting vertex : %d\nShould be between 1 and %d!", u, n);
      return 1;
   }
        dijkstra(G,n,u-1);//Calling the function
        return 0;
}

void dijkstra(int G[MAX][MAX], int n, int startnode) //Main operation
{
        int cost[MAX][MAX], distance[MAX], pred[MAX],k=0,a[MAX],s;
        int visited[MAX], count, mindistance, nextnode, i,j;
        for(i=0;i < n;i++)//Setting the matrix
        {
            for(j=0;j < n;j++)
            {
                if(G[i][j]==0)
                    cost[i][j]=INFINITY;
                else
                    cost[i][j]=G[i][j];
            }
        }
        for(i=0;i< n;i++)//Changing the matrix
        {
            distance[i]=cost[startnode][i];
            pred[i]=startnode;
            visited[i]=0;
        }
        distance[startnode]=0;
        visited[startnode]=1;
        count=1;
```

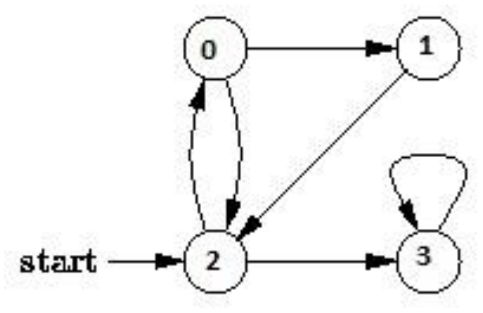
```
while(count < n-1){
        mindistance=INFINITY;
        for(i=0;i < n;i++)
        {
                if(distance[i] < mindistance&&!visited[i])
                {
                        mindistance=distance[i];
                        nextnode=i;
                }
        }
        visited[nextnode]=1;
        for(i=0;i < n;i++)
        {
                if(!visited[i])
                {
                        if(mindistance+cost[nextnode][i] < distance[i])
                        {
                                distance[i]=mindistance+cost[nextnode][i];
                                pred[i]=nextnode;
                        }
                }
        }
        count++;
}

for(i=0;i<n;i++)
{
        if(i!=startnode)//Finding the path
        {
                printf("\nDistance of %d = %d", i, distance[i]);
                j=i;
                printf("\nPath = ");
                do
                {
                        j=pred[j];
                        a[k++]=j;//Auxiliary storage to produce right sequence
                }while(j!=startnode);
```

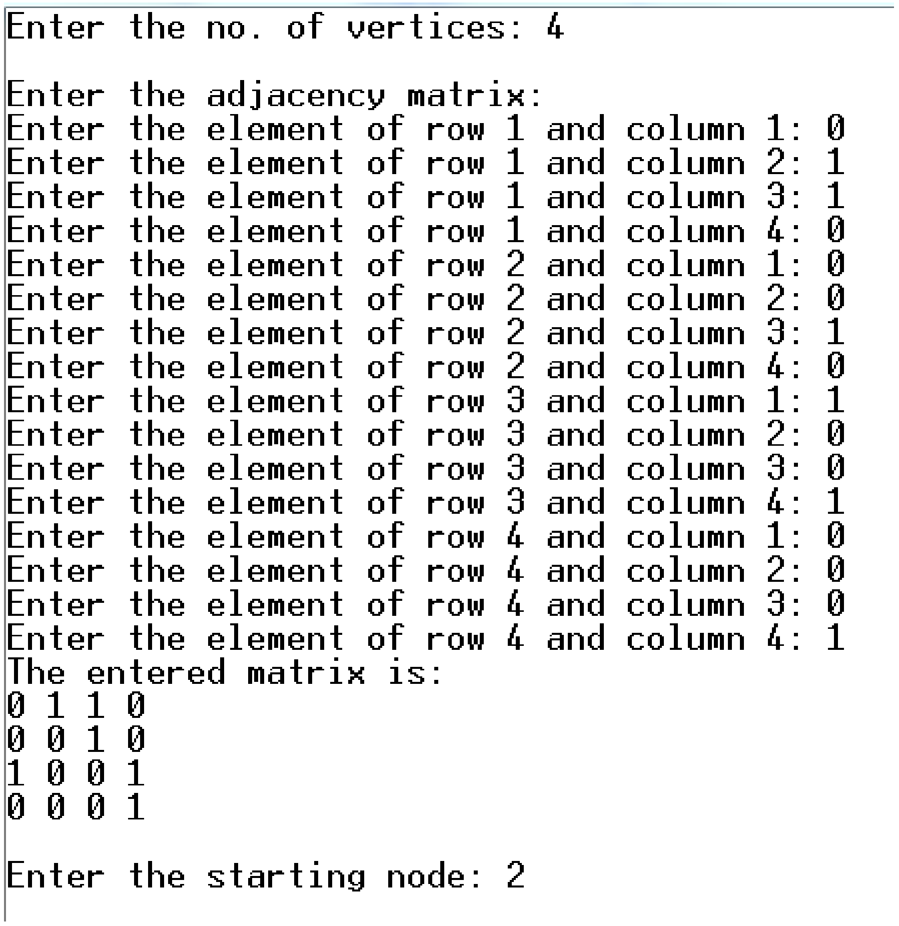
```
                        s=k-1;
                        for(k=s;k>=0;k--)
                        {
                                printf("%d->",a[k]);//Producing right sequence from the
storage
                        }
                        printf("%d", i);
                        k=0;
                }
        }
}
```

Input & Output:

Input graph:



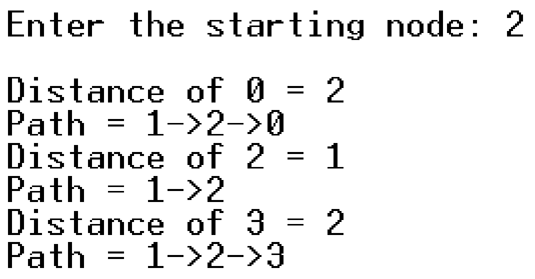
Output of program:



```
Enter the no. of vertices: 4

Enter the adjacency matrix:
Enter the element of row 1 and column 1: 0
Enter the element of row 1 and column 2: 1
Enter the element of row 1 and column 3: 1
Enter the element of row 1 and column 4: 0
Enter the element of row 2 and column 1: 0
Enter the element of row 2 and column 2: 0
Enter the element of row 2 and column 3: 1
Enter the element of row 2 and column 4: 0
Enter the element of row 3 and column 1: 1
Enter the element of row 3 and column 2: 0
Enter the element of row 3 and column 3: 0
Enter the element of row 3 and column 4: 1
Enter the element of row 4 and column 1: 0
Enter the element of row 4 and column 2: 0
Enter the element of row 4 and column 3: 0
Enter the element of row 4 and column 4: 1
The entered matrix is:
0 1 1 0
0 0 1 0
1 0 0 1
0 0 0 1

Enter the starting node: 2
```



```
Enter the starting node: 2

Distance of 0 = 2
Path = 1->2->0
Distance of 2 = 1
Path = 1->2
Distance of 3 = 2
Path = 1->2->3
```

Discussion:

Many more problems than you might at first think can be cast as shortest path problems, making Dijkstra’s algorithm a powerful and general tool. For example:

1. Dijkstra’s algorithm is applied to automatically find directions between physical locations, such as driving directions on websites like Mapquest or Google Maps.
2. In a networking or telecommunication applications, Dijkstra’s algorithm has been used for solving the min-delay path problem (which is the shortest path problem). For example in data network routing, the goal is to find the path for data packets to go through a switching network with minimal delay.
3. It is also used for solving a variety of shortest path problems arising in plant and facility layout, robotics, transportation, and VLSI∗ design.
4. Dijkstra’s algorithm solves such a problem:
    - It finds the shortest path from a given node s to all other nodes in the network.
    - Node s is called a starting node or an initial node.
5. It is a Greedy algorithm.
6. It works by maintaining a set *S* of ``special'' vertices whose shortest distance from the source is already known. At each step, a ``non-special'' vertex is absorbed into *S*.
7. The absorption of an element of *V* - *S* into *S* is done by a greedy strategy.