Dijkstra's Algorithm

Dijkstra's algorithm, published in 1959, is named after its discoverer Edsger Dijkstra, who was a Dutch computer scientist. This algorithm aims to find the shortest-path in a directed or undirected graph with non-negative edge weights.

Before, we look into the details of this algorithm, let’s have a quick overview about the following:

- **Graph**: A graph is a non-linear data structure defined as G=(V,E) where V is a finite set of vertices and E is a finite set of edges, such that each edge is a line or arc connecting any two vertices.
- **Weighted graph**: It is a special type of graph in which every edge is assigned a numerical value, called weight
- **Connected graph**: A path exists between each pair of vertices in this type of graph
- **Spanning tree** for a graph G is a subgraph G’ including all the vertices of G connected with minimum number of edges. Thus, for a graph G with n vertices, spanning tree G’ will have n vertices and maximum n-1 edges.

Problem Statement

Given a weighted graph G, the objective is to find the shortest path from a given source vertex to all other vertices of G. The graph has the following characteristics-

- Set of vertices `V`
- Set of weighted edges `E` such that `(q,r)` denotes an **edge** between **vertices** q and r and cost(q,r) denotes its weight

Dijkstra's Algorithm:

- This is a single-source shortest path algorithm and aims to find solution to the given problem statement
- This algorithm works for both directed and undirected graphs
- It works only for connected graphs
- The graph should not contain negative edge weights
- The algorithm predominantly follows Greedy approach for finding locally optimal solution. But, it also uses Dynamic Programming approach for building globally optimal solution, since the previous solutions are stored and further added to get final distances from the source vertex
- The main logic of this algorithm is basedon the following formula- `dist[r]=min(dist[r], dist[q]+cost[q][r])`

This formula states that distance vertex r, which is adjacent to vertex q, will be updated if and only if the value of `dist[q]+cost[q][r] is less than dist[r]`. Here-

- dist is a 1-D array which, at every step, keeps track of the shortest distance from source vertex to all other vertices, and
- cost is a 2-D array, representing the cost adjacency matrix for the graph

- This formula uses both Greedy and Dynamic approaches. The Greedy approach is used for finding the minimum distance value, whereas the Dynamic approach is used for combining the previous solutions (**dist[q]** is already calculated and is used to calculate **dist[r]**)

Algorithm-

Input Data-

- Cost Adjacency Matrix for Graph G, say cost
- Source vertex, say s

Output Data-

- Spanning tree having shortest path from s to all other vertices in G

Following are the steps used for finding the solution-

Step 1; Set dist[s]=0, S=ϕ // s is the source vertex and S is a 1-D array having all the visited vertices

Step 2: For all nodes v except s, set dist[v]= ∞

Step 3: find q not in S such that dist[q] is minimum // vertex q should not be visited

Step 4: add q to S // add vertex q to S since it has now been visited

Step 5: update dist[r] for all r adjacent to q such that r is not in S //vertex r should not be visited $dist[r]=min(dist[r], dist[q]+cost[q][r])$ //Greedy and Dynamic approach

Step 6: Repeat Steps 3 to 5 until all the nodes are in S // repeat till all the vertices have been visited

Step 7: Print array dist having shortest path from the source vertex u to all other vertices

Step 8: Exit

Let's try and understand the working of this algorithm using the following example-

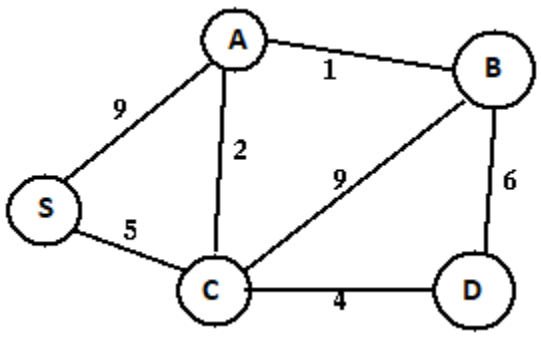

Fig 1: Input Graph (Weighted and Connected)

Given the above weighted and connected graph and source vertex s, following steps are used for finding the tree representing shortest path between s and all other vertices-

Step A- Initialize the distance array (dist) using the following steps of algorithm –

- **Step 1**- Set dist[s]=0, S=ϕ // u is the source vertex and S is a 1-D array having all the visited vertices
- **Step 2**- For all nodes v except s, set dist[v]= ∞

Set of visited vertices (S)	S	A	B	C	D

	0	∞	∞	∞	∞

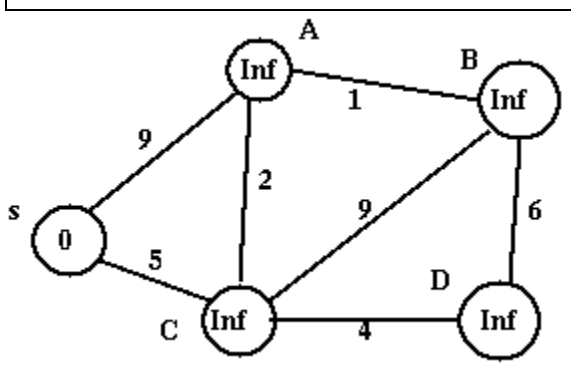

Fig 2: Graph after initializing dist[]

Step B- a)Choose the source vertex s as dist[s] is minimum and s is not in S.

Step 3- find q not in S such that dist[q] is minimum // vertex should not be visited

Visit s by adding it to S

Step 4- add q to S // add vertex q to S since it has now been visited

Step c) For all adjacent vertices of s which have not been visited yet (are not in S) i.e A and C, update the distance array using the following steps of algorithm -

Step 5- update dist[r] for all r adjacent to q such that r is not in S //vertex r should not be visited $dist[r]=min(dist[r], dist[q]+cost[q][r])$ //Greedy and Dynamic approach

$dist[A]= min(dist[A], dist[s]+cost(s, A)) = min(\infty, 0+9) = 9$ $dist[C] = min(dist[C], dist[s]+cost(s, C)) = min(\infty, 0+5) = 5$

Thus dist[] gets updated as follows-

Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞

Step C- Repeat Step B by

a. Choosing and visiting vertex C since it has not been visited (not in S) and dist[C] is minimum
b. Updating the distance array for adjacent vertices of C i.e. A, B and D

Step 6- Repeat Steps 3 to 5 until all the nodes are in S

$dist[A]=min(dist[A], dist[C]+cost(C,A)) = min(9, 5+2)= 7$

$dist[B]= min(dist[B], dist[C]+cost(C,B)) = min(\infty, 5+9)= 14$

$dist[D]= min(dist[D], dist[C]+cost(C,D))= min((\infty,5+4)=9$

This updates dist[] as follows-

Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞

[s,C]	0	7	14	5	9

Continuing on similar lines, Step B gets repeated till all the vertices are visited (added to S). dist[] also gets updated in every iteration, resulting in the following –

Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞
[s,C]	0	7	14	5	9
[s, C, A]	0	7	8	5	9
[s, C, A, B]	0	7	8	5	9
[s, C, A, B, D]	0	7	8	5	9

The last updation of dist[] gives the shortest path values from s to all other vertices

The resultant shortest path spanning tree for the given graph is as follows-

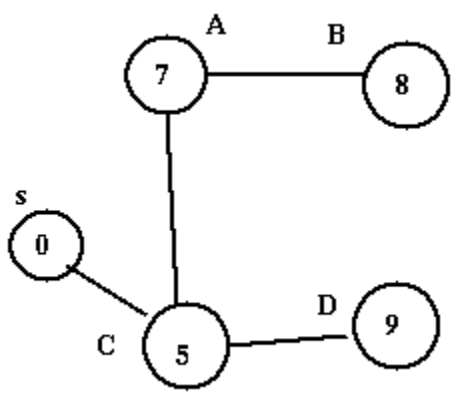

Fig 3: Shortest path spanning tree

Note-

- There can be multiple shortest path spanning trees for the same graph depending on the source vertex

Implementation-
Following is the C++ implementation for Dijkstra's Algorithm

Note : The algorithm can be mapped to any programming language as per the requirement.

```
#include<iostream>

using namespace std;

#define V 5  //Defines total number of vertices in the graph

#define INFINITY 999
```

```
int min_Dist(int dist[], bool visited[])

//This method used to find the vertex with minimum distance and is not
yet visited

{

     int min=INFINITY,index;                    //Initialize min with
infinity

     for(int v=1;v<=V;v++)

     {

          if(visited[v]==false &&dist[v]<=min)

          {

               min=dist[v];

               index=v;

          }

     }

     return index;

}

void Dijkstra(int cost[V][V],int src) //Method to implement shortest
path algorithm

{

     int dist[V];

     bool visited[V];

     for(int i=1;i<=V;i++)                      //Initialize dist[] and
visited[]

     {

          dist[i]=INFINITY;

          visited[i]=false;

     }

     //Initialize distance of the source vertec to zero

     dist[src]=0;

     for(int c=2;c<=V;c++)

     {

          //u is the vertex that is not yet included in visited and is
having minimum
```

```
          int u=min_Dist(dist,visited);                distance

          visited[u]=true;                              //vertex u is now
visited

          for(int v=1;v<=V;v++)

//Update dist[v] for vertex v which is not yet included in visited[]
and

//there is a path from src to v through u that has smaller distance
than

// current value of dist[v]

          {

               if(!visited[v] && cost[u][v]
&&dist[u]+cost[u][v]<dist[v])

                    dist[v]=dist[u]+cost[u][v];

               }

     }

      //will print the vertex with their distance from the source

     cout<<"The shortest path  "<<src<<" to all the other vertices is:
\n";

     for(int i=1;i<=V;i++)

     {

        if(i!=src)

        cout<<"source:"<<src<<"\t destination:"<<i<<"\t MinCost
is:"<<dist[i]<<"\n";

     }

}

int main()

{

     int cost[V][V], i,j, s;

     cout<<"\n Enter the cost matrix weights";

     for(i=1;i<=V;i++)       //Indexing ranges from 1 to n

          for(j=1;j<=V;j++)

          {

cin>>cost[i][j];
```

```
                //Absence of edge between vertices i and j is
represented by INFINITY

              if(cost[i][j]==0)

                cost[i][j]=INFINITY;

            }

cout<<"\n Enter the Source Vertex";

cin>>s;



     Dijkstra(cost,s);

     return 0;

}
```

Copy

The program is executed using same input graph as in Fig.1.This will help in verifying the resultant solution set with actual output.

```
 Enter the cost matrix weights
0 9 999 5 999
9 0 1 2 999
999 1 0 9 6
5 2 9 0 4
999 999 6 4 0

 Enter the Source Vertex1
The shortest path from source 1 to all the other vertices is:
source: 1        destination: 2  MinCost is: 7
source: 1        destination: 3  MinCost is: 8
source: 1        destination: 4  MinCost is: 5
source: 1        destination: 5  MinCost is: 9
```

Fig 4: Output

Time Complexity Analysis-

Following are the cases for calculating the time complexity of Dijkstra's Algorithm-

- **Case1**- When graph G is represented using an adjacency matrix -This scenario is implemented in the above **C++** based program. Since the implementation contains two nested for loops, each of complexity **O(n)**, the complexity of Dijkstra's algorithm is **O(n2)**. Please note that n here refers to total number of vertices in the given graph
- **Case 2**- When graph G is represented using an adjacency list - The time complexity, in this scenario reduces to **O(|E| + |V| log |V|)** where |E|represents number of edges and |V| represents number of vertices in the graph

Disadvantages of Dijkstra's Algorithm-

Dijkstra's Algorithm cannot obtain correct shortest path(s)with weighted graphs having negative edges. Let's consider the following example to explain this scenario-

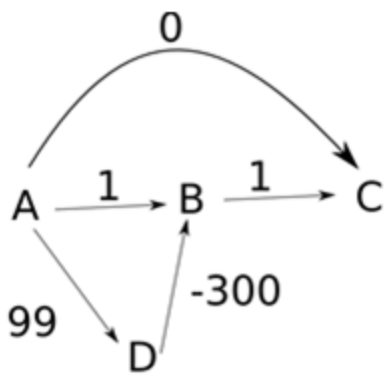

Fig 5: Weighted graph with negative edges

Choosing source vertex as A, the algorithm works as follows-

Step A- Initialize the distance array (dist)-

Set of visited vertices (S)	A	B	C	D
	0	∞	∞	∞

Step B- Choose vertex A as dist[A] is minimum and A is not in S. Visit A and add it to S. For all adjacent vertices of A which have not been visited yet (are not in S) i.e C, B and D, update the distance array

$dist[C] = \min(dist[C], dist[A]+cost(A, C)) = \min(\infty, 0+0) = 0$

$dist[B] = \min(dist[B], dist[A]+cost(A, B)) = \min(\infty, 0+1) = 1$

$dist[D] = \min(dist[D], dist[A]+cost(A, D)) = \min(\infty, 0+99) = 99$

Thus dist[] gets updated as follows-

Set of visited vertices (S)	A	B	C	D
[A]	0	1	0	99

Step C- Repeat Step B by

a. Choosing and visiting vertex C since it has not been visited (not in S) and dist[C] is minimum
b. The distance array does not get updated since there are no adjacent vertices of C

Set of visited vertices (S)	A	B	C	D
[A]	0	1	0	99
[A, C]	0	1	0	99

Continuing on similar lines, Step B gets repeated till all the vertices are visited (added to S). dist[] also gets updated in every iteration, resulting in the following –

Set of visited vertices (S)	A	B	C	D

[A]	0	1	0	99
[A, C]	0	1	0	99
[A, C, B]	0	1	0	99
[A, C, B, D]	0	1	0	99

Thus, following are the shortest distances from A to B, C and D-

A->C = 0 A->B = 1 A->D = 99

But these values are not correct, since we can have another path from **A** to **C, A->D->B->C** having **total cost= -200** which is smaller than 0. This happens because once a vertex is visited and is added to the set S, it is never "looked back" again. Thus, Dijkstra's algorithm does not try to find a shorter path to the vertices which have already been added to S.

- It performs a blind search for finding the shortest path, thus, consuming a lot of time and wasting other resources

Applications of Dijkstra's Algorithm-

- Traffic information systems use Dijkstra's Algorithm for tracking destinations from a given source location
- **Open Source Path First (OSPF)**, an Internet-based routing protocol, uses Dijkstra's Algorithm for finding best route from source router to other routers in the network
- It is used by **Telephone and Cellular networks** for routing management
- It is also used by **Geographic Information System (GIS)**, such as Google Maps, for finding shortest path from point A to point B
