NAME: Meet Raut

DIV: S21

ROLL.NO: 2201084

Experiment 4:

- **AIM: To study and implement Single source shortest path using Dijkstra's Algorithm.**

- **THEORY:**

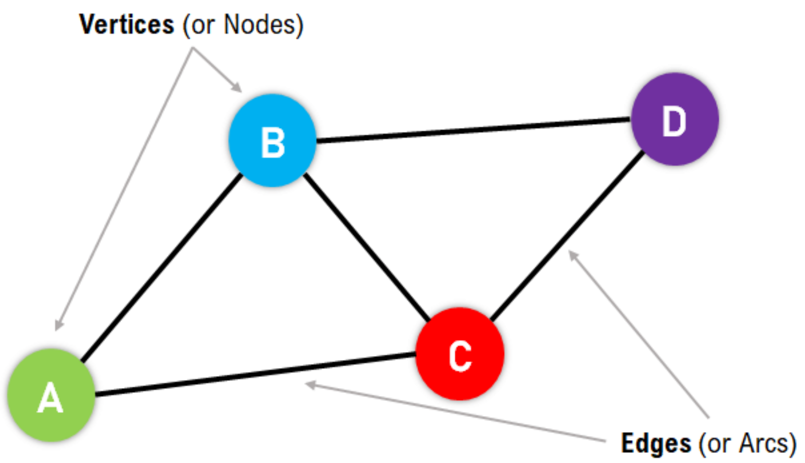
A Brief Introduction to Graphs:

Graphs are non-linear data structures representing the "connections" between the elements. These elements are known as the **Vertices**, and the lines or arcs that connect any two vertices in the graph are known as the **Edges**. More formally, a Graph comprises **a set of Vertices (V)** and **a set of Edges (E)**. The Graph is denoted by **G(V, E)**.

Components of a Graph:

1. **Vertices:**Vertices are the basic units of the graph used to represent real-life objects, persons, or entities. Sometimes, vertices are also known as Nodes.
2. **Edges:**Edges are drawn or used to connect two vertices of the graph. Sometimes, edges are also known as Arcs.

The following figure shows a graphical representation of a Graph:

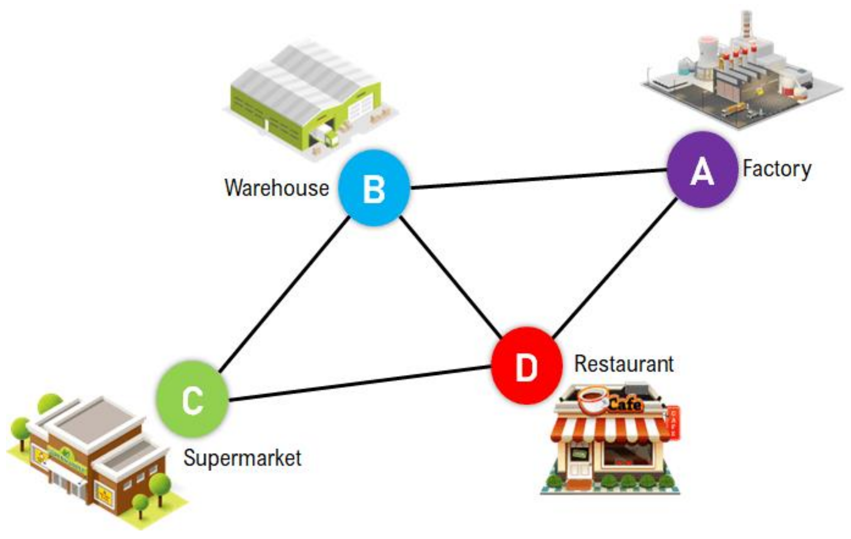


In the above figure, the Vertices/Nodes are denoted with Colored Circles, and the Edges are denoted with the lines connecting the nodes.

Applications of the Graphs:

Graphs are used to solve many real-life problems. Graphs are utilized to represent the networks. These networks may include telephone or circuit networks or paths in a city.

For example, we could use Graphs to design a transportation network model where the vertices display the facilities that send or receive the products, and the edges represent roads or paths connecting them. The following is a pictorial representation of the same:



Graphs are also utilized in different Social Media Platforms like LinkedIn, Facebook, Twitter, and more. For example, Platforms like Facebook use Graphs to store the data of their users where every person is indicated with a vertex, and each of them is a structure containing information like Person ID, Name, Gender, Address, etc.

Types of Graphs:

The Graphs can be categorized into two types:

1. Undirected Graph
2. Directed Graph

Undirected Graph: A Graph with edges that do not have a direction is termed an Undirected Graph. The edges of this graph imply a two-way relationship in which each edge can be traversed in both directions. The following figure displays a simple undirected graph with four nodes and five edges.

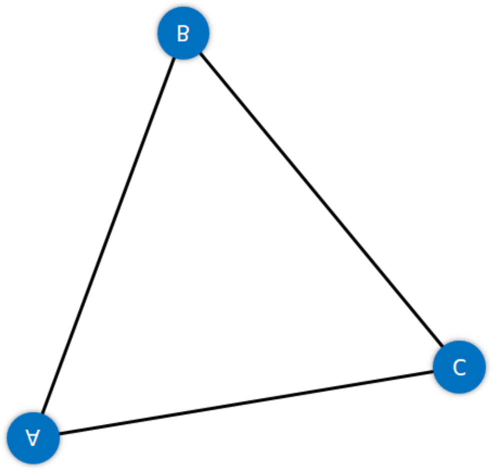


Figure 3: A Simple Undirected Graph

Directed Graph: A Graph with edges with direction is termed a Directed Graph. The edges of this graph imply a one-way relationship in which each edge can only be traversed in a single direction. The following figure displays a simple directed graph with four nodes and five edges.

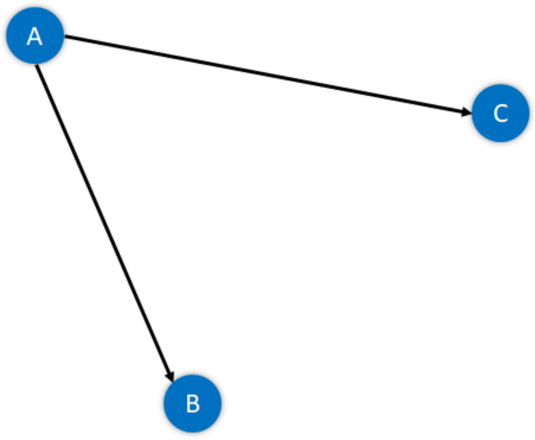


Figure 4: A Simple Directed Graph

The absolute length, position, or orientation of the edges in a graph illustration characteristically does not have meaning. In other words, we can visualize the same graph in different ways by rearranging the vertices or distorting the edges if the underlying structure of the graph does not alter.

What are Weighted Graphs?

A Graph is said to be Weighted if each edge is assigned a 'weight'. The weight of an edge can denote distance, time, or anything that models the 'connection' between the pair of vertices it connects.

For instance, we can observe a blue number next to each edge in the following figure of the Weighted Graph. This number is utilized to signify the weight of the corresponding edge.

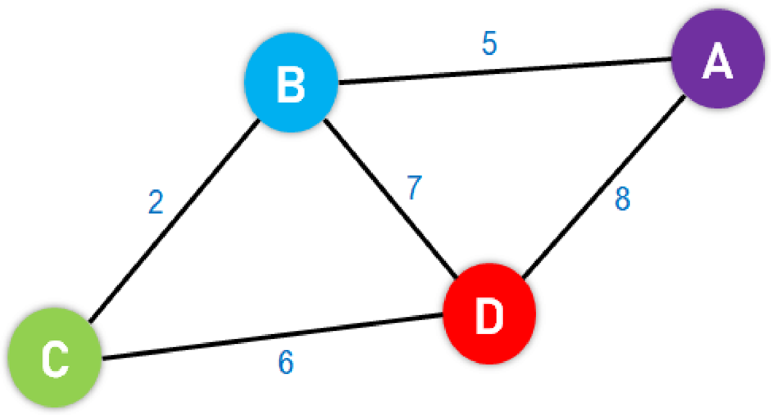


Figure 5: An Example of a Weighted Graph

➢ Dijkstra's Algorithm:

Dijkstra's Algorithm is a Graph algorithm **that finds the shortest path** from a source vertex to all other vertices in the Graph (single source shortest path). It is a type of Greedy Algorithm that only works on Weighted Graphs having positive weights. The time complexity of Dijkstra's Algorithm is **O(V^2)** with the help of the adjacency matrix representation of the graph. This time complexity can be reduced to **O((V + E) log V)** with the help of an adjacency list representation of the graph, where **V** is the number of vertices and **E** is the number of edges in the graph.

History of Dijkstra's Algorithm:

Dijkstra's Algorithm was designed and published by **Dr. Edsger W. Dijkstra**, a Dutch Computer Scientist, Software Engineer, Programmer, Science Essayist, and Systems Scientist.

During an Interview with Philip L. Frana for the Communications of the ACM journal in the year 2001, Dr. Edsger W. Dijkstra revealed:

"What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city? It is the algorithm for the shortest path, which I designed in about twenty minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention. In fact, it was published in '59, three years later. The publication is still readable, it is, in fact, quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities. Eventually, that algorithm became to my great amazement, one of the cornerstones of my fame."

Dijkstra thought about the shortest path problem while working as a programmer at the Mathematical Centre in Amsterdam in 1956 to illustrate the capabilities of a new computer known as ARMAC. His goal was to select both a problem and a solution (produced by the computer) that people with no computer background could comprehend. He developed the shortest path algorithm and later executed it for ARMAC for a vaguely shortened transportation map of 64 cities in the Netherlands (64 cities, so 6 bits would be sufficient to encode the city number). A year later, he came across another issue from hardware engineers operating the next computer of the institute: Minimize the amount of wire required to connect the pins on the machine's back panel. As a solution, he re-discovered the algorithm called Prim's minimal spanning tree algorithm and published it in the year 1959.

Fundamentals of Dijkstra's Algorithm:

The following are the basic concepts of Dijkstra's Algorithm:

1. Dijkstra's Algorithm begins at the node we select (the source node), and it examines the graph to find the shortest path between that node and all the other nodes in the graph.
2. The Algorithm keeps records of the presently acknowledged shortest distance from each node to the source node, and it updates these values if it finds any shorter path.
3. Once the Algorithm has retrieved the shortest path between the source and another node, that node is marked as 'visited' and included in the path.
4. The procedure continues until all the nodes in the graph have been included in the path. In this manner, we have a path connecting the source node to all other nodes, following the shortest possible path to reach each node.

Understanding the Working of Dijkstra's Algorithm:

A **graph** and **source vertex** are requirements for Dijkstra's Algorithm. This Algorithm is established on Greedy Approach and thus finds the locally optimal choice (local minima in this case) at each step of the Algorithm.

Each Vertex in this Algorithm will have two properties defined for it:

1. Visited Property
2. Path Property

Let us understand these properties in brief.

Visited Property:

1. The 'visited' property signifies whether or not the node has been visited.
2. We are using this property so that we do not revisit any node.
3. A node is marked visited only when the shortest path has been found.

Path Property:

1. The 'path' property stores the value of the current minimum path to the node.
2. The current minimum path implies the shortest way we have reached this node till now.
3. This property is revised when any neighbor of the node is visited.

4. This property is significant because it will store the final answer for each node.

Initially, we mark all the vertices, or nodes, unvisited as they have yet to be visited. The path to all the nodes is also set to infinity apart from the source node. Moreover, the path to the source node is set to zero (0).

We then select the source node and mark it as visited. After that, we access all the neighboring nodes of the source node and perform relaxation on every node. Relaxation is the process of lowering the cost of reaching a node with the help of another node.

In the process of relaxation, the path of each node is revised to the minimum value amongst the node's current path, the sum of the path to the previous node, and the path from the previous node to the current node.

Let us suppose that p[n] is the value of the current path for node n, p[m] is the value of the path up to the previously visited node m, and w is the weight of the edge between the current node and previously visited one (edge weight between n and m).

In the mathematical sense, relaxation can be exemplified as:

p[n] = minimum(p[n], p[m] + w)

We then mark an unvisited node with the least path as visited in every subsequent step and update its neighbor's paths.

We repeat this procedure until all the nodes in the graph are marked visited.

Whenever we add a node to the visited set, the path to all its neighboring nodes also changes accordingly.

If any node is left unreachable (disconnected component), its path remains 'infinity'. In case the source itself is a separate component, then the path to all other nodes remains 'infinity'.

Understanding Dijkstra's Algorithm with an Example

The following is the step that we will follow to implement Dijkstra's Algorithm:

Step 1: First, we will mark the source node with a current distance of 0 and set the rest of the nodes to INFINITY.

Step 2: We will then set the unvisited node with the smallest current distance as the current node, suppose X.

Step 3: For each neighbor N of the current node X: We will then add the current distance of X with the weight of the edge joining X-N. If it is smaller than the current distance of N, set it as the new current distance of N.

Step 4: We will then mark the current node X as visited.

Step 5: We will repeat the process from **'Step 2'** if there is any node unvisited left in the graph.

Let us now understand the implementation of the algorithm with the help of an example:

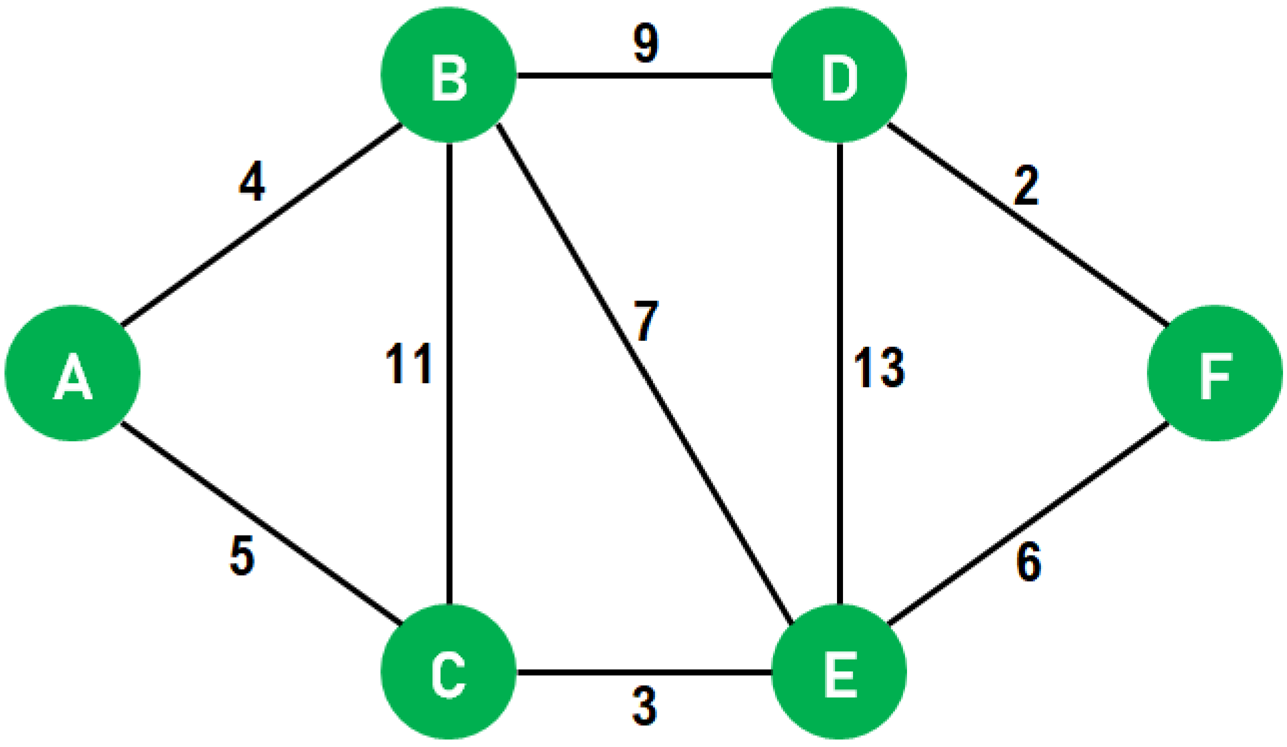
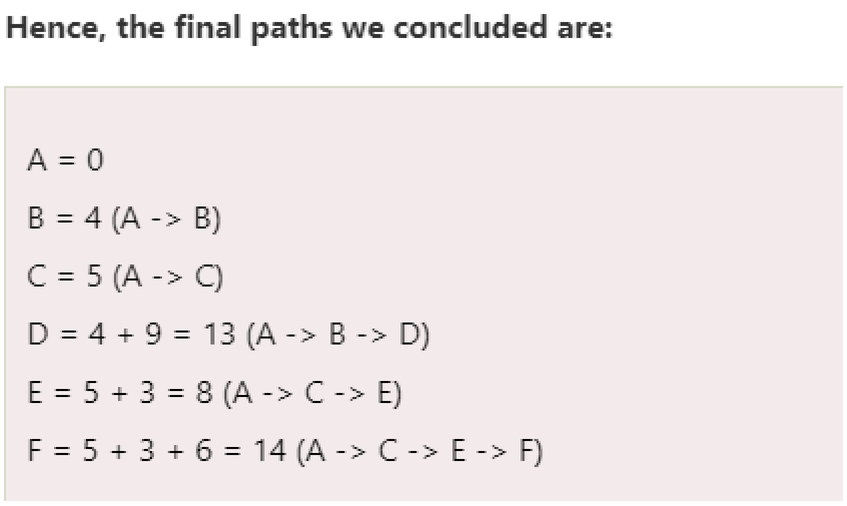


Figure 6: The Given Graph

1. We will use the above graph as the input, with node **A** as the source.
2. First, we will mark all the nodes as unvisited.
3. We will set the path to **0** at node **A** and **INFINITY** for all the other nodes.
4. We will now mark source node **A** as visited and access its neighboring nodes. **Note:** We have only accessed the neighboring nodes, not visited them.
5. We will now update the path to node **B** by **4** with the help of relaxation because the path to node **A** is **0** and the path from node **A** to **B** is **4**, and the **minimum((0 + 4), INFINITY)** is **4**.
6. We will also update the path to node **C** by **5** with the help of relaxation because the path to node **A** is **0** and the path from node **A** to **C** is **5**, and the **minimum((0 + 5), INFINITY)** is **5**. Both the neighbors of node **A** are now relaxed; therefore, we can move ahead.
7. We will now select the next unvisited node with the least path and visit it. Hence, we will visit node **B** and perform relaxation on its unvisited neighbors. After performing

relaxation, the path to node **C** will remain **5**, whereas the path to node **E** will become **11**, and the path to node **D** will become **13**.

8. We will now visit node **E** and perform relaxation on its neighboring nodes **B, D**, and **F**. Since only node **F** is unvisited, it will be relaxed. Thus, the path to node **B** will remain as it is, i.e., **4**, the path to node **D** will also remain **13**, and the path to node **F** will become **14 (8 + 6)**.
9. Now we will visit node **D**, and only node **F** will be relaxed. However, the path to node **F** will remain unchanged, i.e., **14**.
10. Since only node **F** is remaining, we will visit it but not perform any relaxation as all its neighboring nodes are already visited.
11. Once all the nodes of the graphs are visited, the program will end.


Hence, the final paths we concluded are:

```
A = 0
B = 4 (A -> B)
C = 5 (A -> C)
D = 4 + 9 = 13 (A -> B -> D)
E = 5 + 3 = 8 (A -> C -> E)
F = 5 + 3 + 6 = 14 (A -> C -> E -> F)
```

Pseudocode for Dijkstra's Algorithm:

We will now understand a pseudocode for Dijkstra's Algorithm.

- We have to maintain a record of the path distance of every node. Therefore, we can store the path distance of each node in an array of size n, where n is the total number of nodes.
- Moreover, we want to retrieve the shortest path along with the length of that path. To overcome this problem, we will map each node to the node that last updated its path length.
- Once the algorithm is complete, we can backtrack the destination node to the source node to retrieve the path.
- We can use a minimum Priority Queue to retrieve the node with the least path distance in an efficient way.

Let us now implement a pseudocode of the above illustration:

Pseudocode:

```
function Dijkstra_Algorithm(Graph, source_node)
    // iterating through the nodes in Graph and set their distances to INFINITY
    for each node N in Graph:
        distance[N] = INFINITY
        previous[N] = NULL
        If N != source_node, add N to Priority Queue G
    // setting the distance of the source node of the Graph to 0
    distance[source_node] = 0

    // iterating until the Priority Queue G is not empty
    while G is NOT empty:
        // selecting a node Q having the least distance and marking it as visited
        Q = node in G with the least distance[]
        mark Q visited

        // iterating through the unvisited neighboring nodes of the node Q and perform
ing relaxation accordingly
        for each unvisited neighbor node N of Q:
            temporary_distance = distance[Q] + distance_between(Q, N)

            // if the temporary distance is less than the given distance of the path to the
Node, updating the resultant distance with the minimum value
            if temporary_distance < distance[N]
                distance[N] := temporary_distance
                previous[N] := Q

    // returning the final list of distance
    return distance[], previous[]
```

Explanation:

In the above pseudocode, we have defined a function that accepts multiple parameters - the Graph consisting of the nodes and the source node. Inside this function, we have iterated through each node in the Graph, set their initial distance to **INFINITY**, and set the previous node value to **NULL**. We have also checked whether any selected node is not a source node and added the same into the Priority Queue. Moreover, we have set the distance of the source node to **0**. We then iterated through the nodes in the priority queue, selected the node with the least distance, and marked it as visited. We then iterated through the unvisited neighboring nodes of the selected node and performed relaxation accordingly. At last, we have compared both the distances (original and temporary distance) between the source node and the destination node, updated the resultant distance with the minimum value and previous node information, and returned the final list of distances with their previous node information.

Dijkstra's Algorithm using Adjacency Matrix:

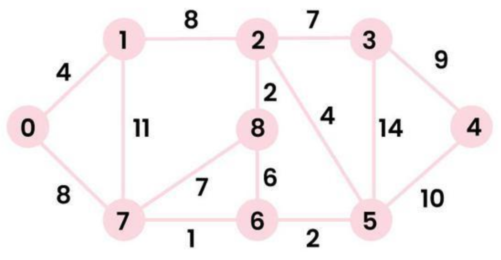
The idea is to generate a SPT (shortest path tree) with a given source as a root. Maintain an Adjacency Matrix with two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, find a vertex that is in the other set (set not yet included) and has a minimum distance from the source.

Algorithm:

- Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign the distance value as 0 for the source vertex so that it is picked first.
- While sptSet doesn't include all vertices
- Pick a vertex u that is not there in sptSet and has a minimum distance value.
- Include u to sptSet.
- Then update the distance value of all adjacent vertices of u.
- To update the distance values, iterate through all adjacent vertices.
- For every adjacent vertex v, if the sum of the distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Note: We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store the shortest distance values of all vertices.

Examples:
Input: *src = 0, the graph is shown below.*

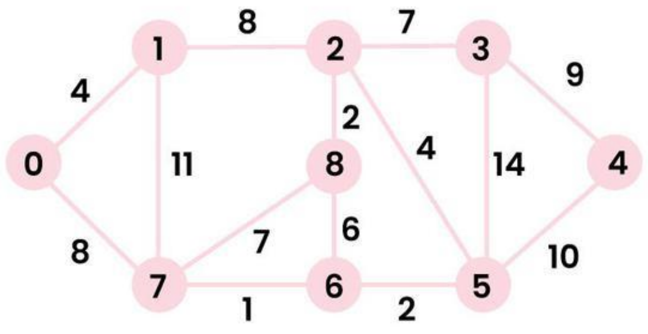


Working of Dijkstra's Algorithm

Output: 0 4 12 19 21 11 9 8 14
Explanation: The distance from 0 to 1 = 4.
The minimum distance from 0 to 2 = 12. 0->1->2
The minimum distance from 0 to 3 = 19. 0->1->2->3
The minimum distance from 0 to 4 = 21. 0->7->6->5->4
The minimum distance from 0 to 5 = 11. 0->7->6->5
The minimum distance from 0 to 6 = 9. 0->7->6
The minimum distance from 0 to 7 = 8. 0->7
The minimum distance from 0 to 8 = 14. 0->1->2->8

Illustration of Dijkstra Algorithm:

To understand the Dijkstra's Algorithm lets take a graph and find the shortest path from source to all nodes.
Consider below graph and **src = 0**

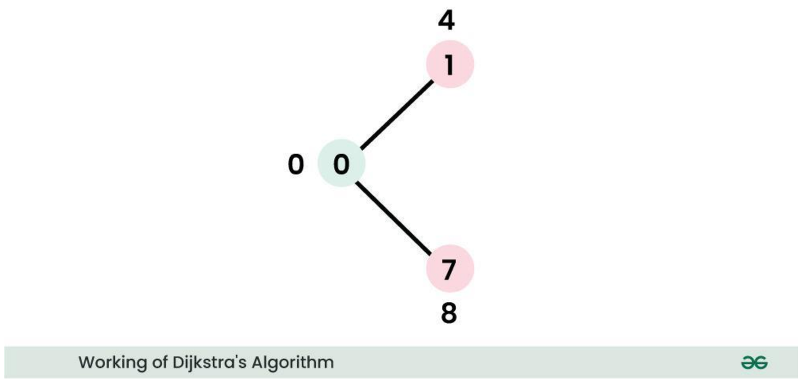


Working of Dijkstra's Algorithm

Step 1:

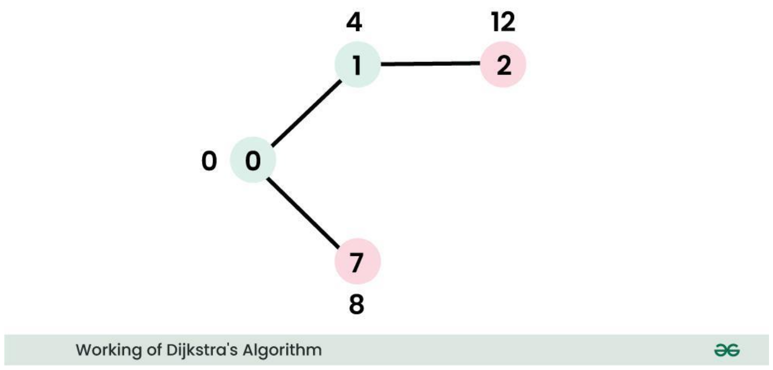
- The set **sptSet** is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where **INF** indicates infinite.
- Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in **sptSet**. So **sptSet** becomes {0}. After including 0 to **sptSet**, update distance values of its adjacent vertices.
- Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8.

The following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in **SPT** are shown in **green** colour.



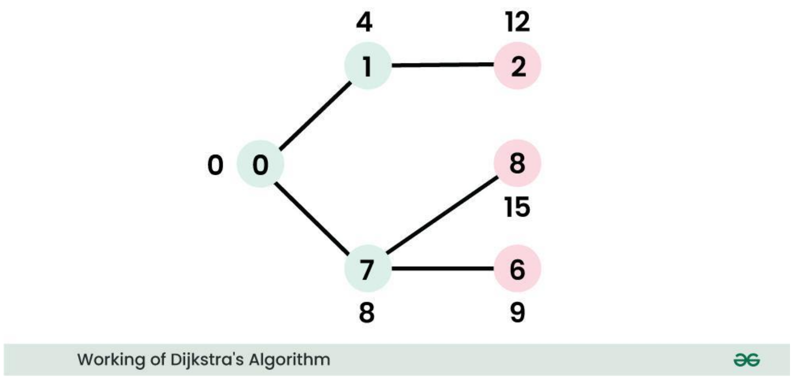
Step 2:

- Pick the vertex with minimum distance value and not already included in **SPT** (not in **sptSET**). The vertex 1 is picked and added to **sptSet**.
- So **sptSet** now becomes {0, 1}. Update the distance values of adjacent vertices of 1.
- The distance value of vertex 2 becomes **12**.



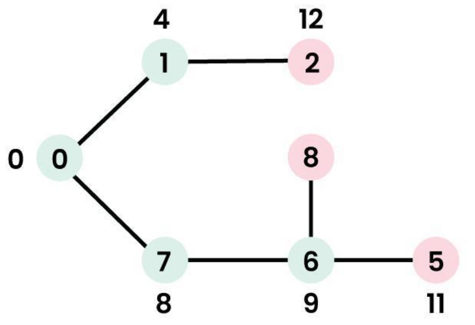
Step 3:

- Pick the vertex with minimum distance value and not already included in **SPT** (not in **sptSET**). Vertex 7 is picked. So **sptSet** now becomes **{0, 1, 7}.**
- Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (**15 and 9** respectively).



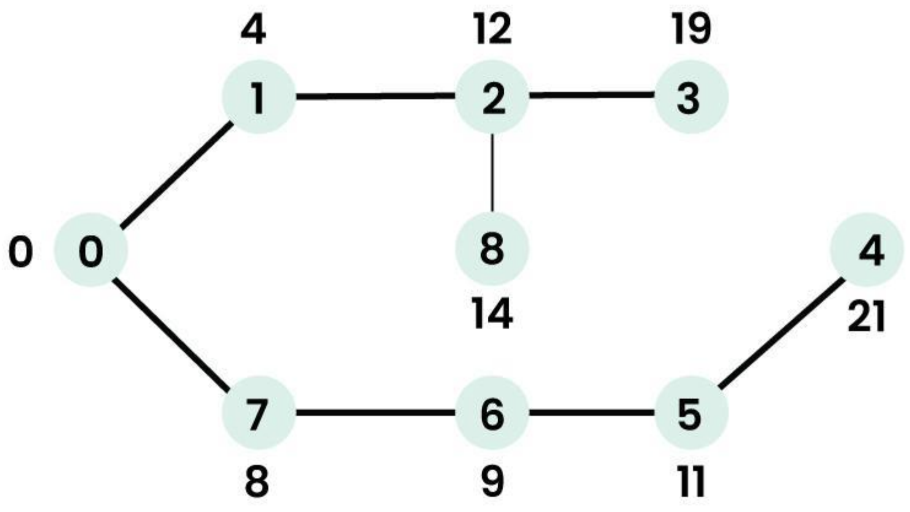
Step 4:

- Pick the vertex with minimum distance value and not already included in **SPT** (not in **sptSET**). Vertex 6 is picked. So **sptSet** now becomes **{0, 1, 7, 6}**.
- Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



Working of Dijkstra's Algorithm

We repeat the above steps until **sptSet** includes all vertices of the given graph. Finally, we get the following S**hortest Path Tree (SPT).**



Working of Dijkstra's Algorithm

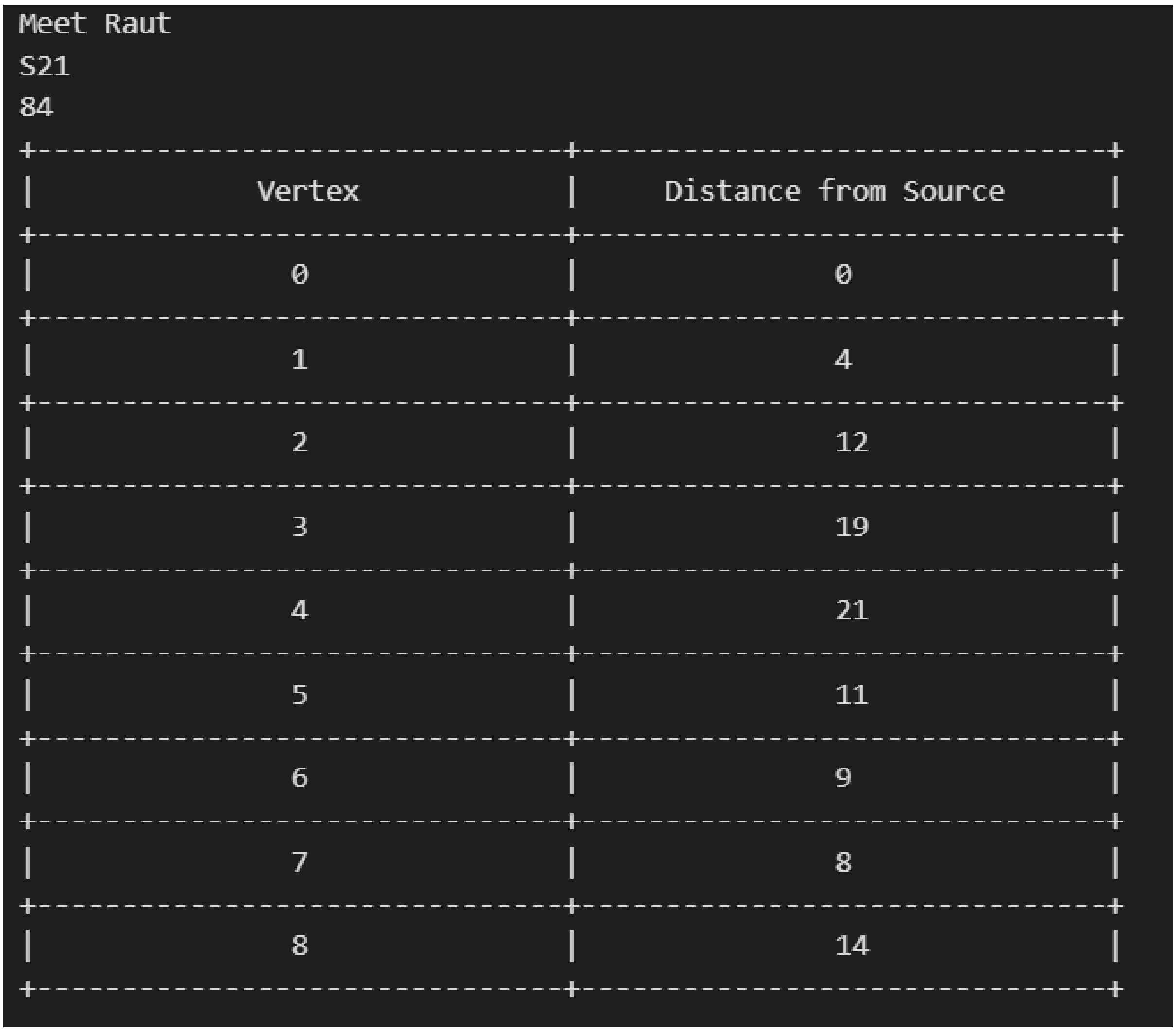
➢ **C PROGRAM:**

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
// Number of vertices in the graph
#define V 9
int minDistance(int dist[], bool sptSet[])
{
// Initialize min value
int min = INT_MAX, min_index;
for (int v = 0; v < V; v++)
if (sptSet[v] == false && dist[v] <= min)
min = dist[v], min_index = v;
return min_index;
}
// A utility function to print the constructed distance arr
void printSolution(int dist[])
{
   printf("Meet Raut\nS21 \n84\n");
   printf("+------------------------------+------------------------------+\n");
   printf("|              Vertex              |     Distance from Source      |\n");
   printf("+------------------------------+------------------------------+\n");
   for (int i = 0; i < V; i++)
   {
      printf("|\t\t\t\t%d\t\t\t\t|\t\t\t\t%d\t\t\t\t|\n", i, dist[i]);
      printf("+------------------------------+------------------------------+\n");
   }
}

void dijkstra(int graph[V][V], 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];
    }
    // print the constructed distance array
    printSolution(dist);
}
// driver's code
int main()
{
/* Let us create the example graph discussed above */
int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
{ 4, 0, 8, 0, 0, 0, 0, 11, 0 },
{ 0, 8, 0, 7, 0, 4, 0, 0, 2 },
{ 0, 0, 7, 0, 9, 14, 0, 0, 0 },
{ 0, 0, 0, 9, 0, 10, 0, 0, 0 },
{ 0, 0, 4, 14, 10, 0, 2, 0, 0 },
{ 0, 0, 0, 0, 0, 2, 0, 1, 6 },
{ 8, 11, 0, 0, 0, 0, 1, 0, 7 },
{ 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
// Function call
dijkstra(graph, 0);
return 0;
}
```

- OUTPUT:



```
Meet Raut
S21
84
+------------------------------+------------------------------+
|            Vertex            |     Distance from Source     |
+------------------------------+------------------------------+
|              0               |              0               |
+------------------------------+------------------------------+
|              1               |              4               |
+------------------------------+------------------------------+
|              2               |              12              |
+------------------------------+------------------------------+
|              3               |              19              |
+------------------------------+------------------------------+
|              4               |              21              |
+------------------------------+------------------------------+
|              5               |              11              |
+------------------------------+------------------------------+
|              6               |              9               |
+------------------------------+------------------------------+
|              7               |              8               |
+------------------------------+------------------------------+
|              8               |              14              |
+------------------------------+------------------------------+
```

- **<u>CONCLUSION:</u> Hence, we have successfully implemented Single source shortest path using Dijkstra’s Algorithm.**