Weighted Graph:

A weighted graph, G is a graph comprised of three entities V a set of vertices and E a set of edges and W is a function from E into Z, where Z is a set of all positive integers or negative integers. That is , W: E -> Z.

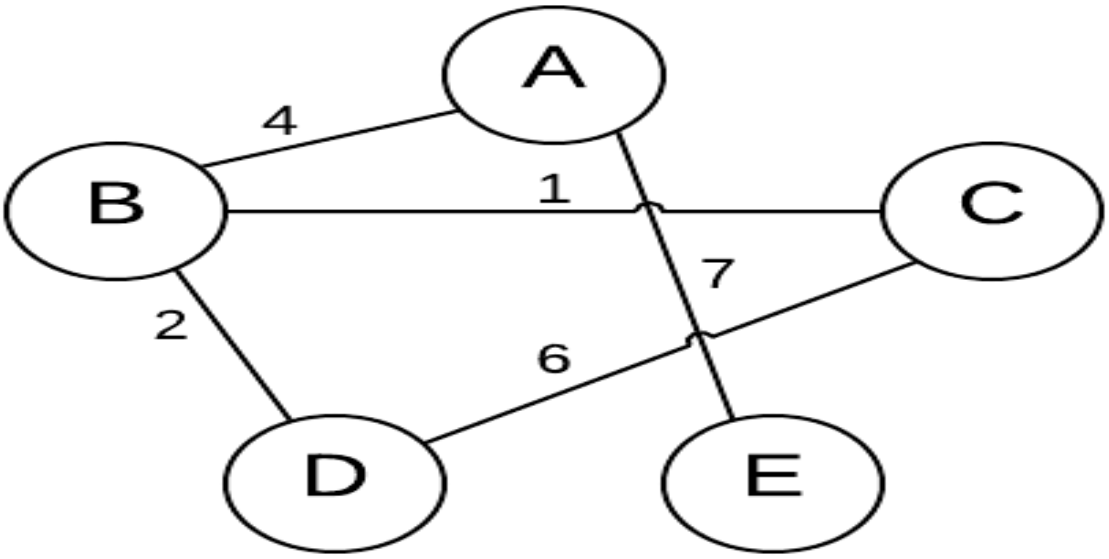
A **weighted graph** refers to a simple graph that has weighted edges. These weighted edges can be used to compute shortest path. It consists of:

- A set of vertices V.
- A set of edges E.
- A number w (**weight**) that is assigned to each edge. Weights might represent things such as costs, lengths or capacities.
- Additional operation on weighted graphs are as follows

Add (v1, v2, weight)	adds the given edge
Remove (v1, v3, weight)	removes the given edge
Weight (v1, v2)	returns the weight of the edge

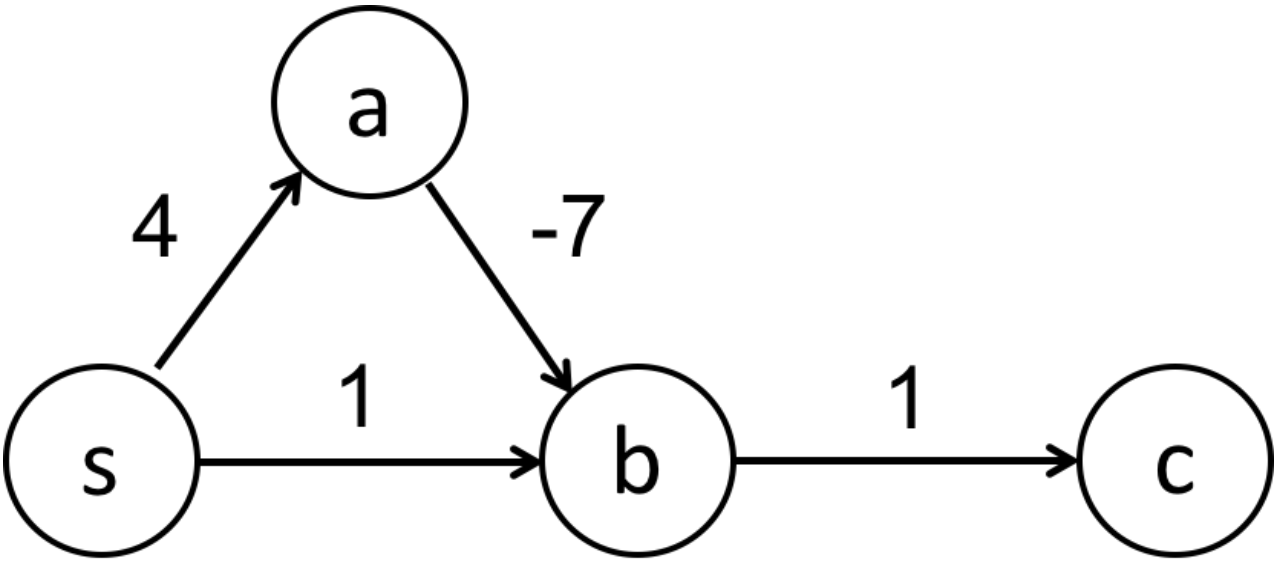
In weighted graphs there are two types of graphs.

- **Undirected Graph**, where direction of edges is not specified between the vertices. Two vertices are connected to each other irrespective of the direction. One can iterate from one vertex to another or vice versa only if they are connected through an edge.



In the above weighted graph, there are five vertices [A, B, C, D, E] and five edges with the weights [1, 2, 4, 6, 7]. For example, weight of the edge between vertices A-B is 4, weight of the edge between vertices B-D is 2, weight of the edge between vertices B-C is 1, weight of the edge between vertices D-C is 6 and weight of the edge between vertices A-E is 7. And one can iterate from a vertex to another and vice versa if they are connected.

- **Directed Graph**, where direction of edges is specified between the vertices. Edge is represented in the form of an arrow having a tail on one vertex and head on the other vertex. While iterating through the graph the direction of the edge decides the flow. One starts from the tail of an edge and ends on the head of the edge.



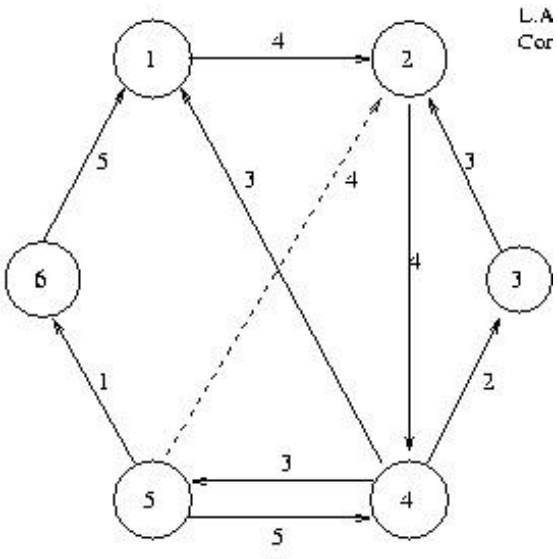
In the above graph, there are four vertices [a, b, s, c] and four edges a -> b (-7), s -> a (4), s -> b (1), b -> c (1). For example, if one starts iterating from "s" then the path either will be (s -> a-> b -> c) or (s -> b -> c). But in an case iteration from b vertex to a or b vertex to s is not feasible.

Adjacency List:

An adjacency list consists of an array of vertices (V) and an array of edges (E), where each element in the vertex array stores the starting index (in the edge array) of the edges outgoing from each node. The edge array stores the destination vertices of each edge.

Adjacency List for Directed Graph:

Consider the following case:



An edge <vi, vj> is placed in a list associated with vi. The edge is represented by the destination vj and the weight.

1 -> 2, 4: nil

2 -> 4, 4: nil

3 -> 2, 3: nil

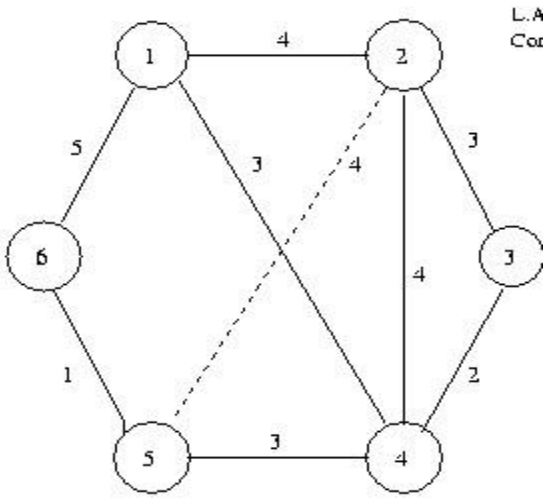
4 -> 1, 3: -> 3, 2 -> 5, 3: nil

5 -> 4, 5: -> 6, 1: nil

6 -> 1, 5: nil

Adjacency List for Undirected Graph:

Consider the following case:



1 -> 2, 4: -> 4, 3: -> 6, 5: nil

2 -> 1, 4: -> 3, 3: -> 4, 4: -> 5, 4: nil

3 -> 2, 3: -> 4, 2: nil

4 -> 1, 3: -> 2, 4: -> 3, 2: -> 5, 3: nil

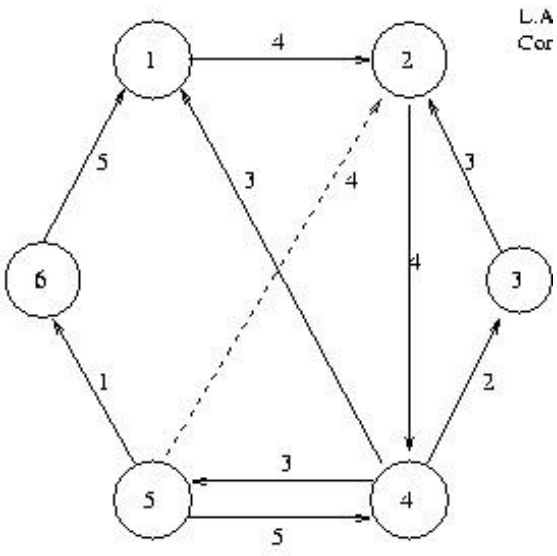
5-> 2, 4: -> 4, 3: -> 6, 1: nil

6 -> 1, 5: -> 5, 1: nil

Adjacency Matrix:

An adjacency matrix is easily implemented as an array. Both directed and undirected may be weighted. A weight is attached to each edge. The weight is attached to each edge. The may be used to represent the distance between two cities, the flight time, the cost of the fare, the electrical capacity of a cable or some other quantity associated with the edge. The weight is sometimes called the length of the edge, particularly when the graph represents a map of some kind. The weight or length of a path or a cycle is the sum of the weights or lengths of its component edges. Algorithms to find shortest paths in a graph are given later.

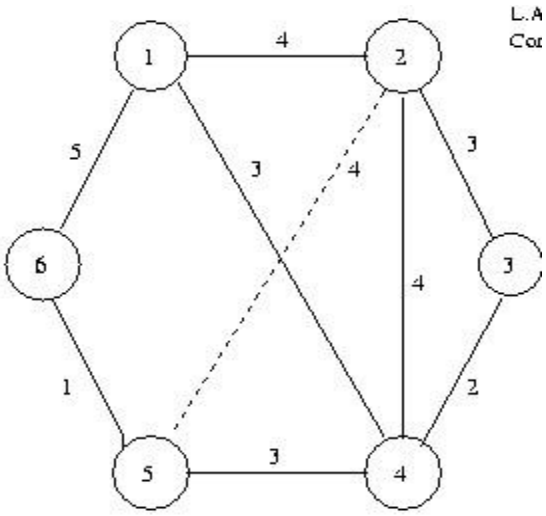
The Adjacency Matrix of Weighted Directed Graph:



The adjacency matrix of a weighted graph can be used to store the weights of the edges. If an edge is missing a special value, perhaps a negative value, zero or a large value to represent “infinity”, indicates this fact.

	1	2	3	4	5	6
1		4				
2				4		
3		3				
4	3		2		3	
5		4		5		1
6	5					

The Adjacency Matrix of Weighted Undirected Graph:



	1	2	3	4	5	6
1		4		3		5
2	4		3	4	4	
3		3		2		
4	3	4	2		3	
5		4		3		1
6	5				1	

Dijkstra Algorithm:

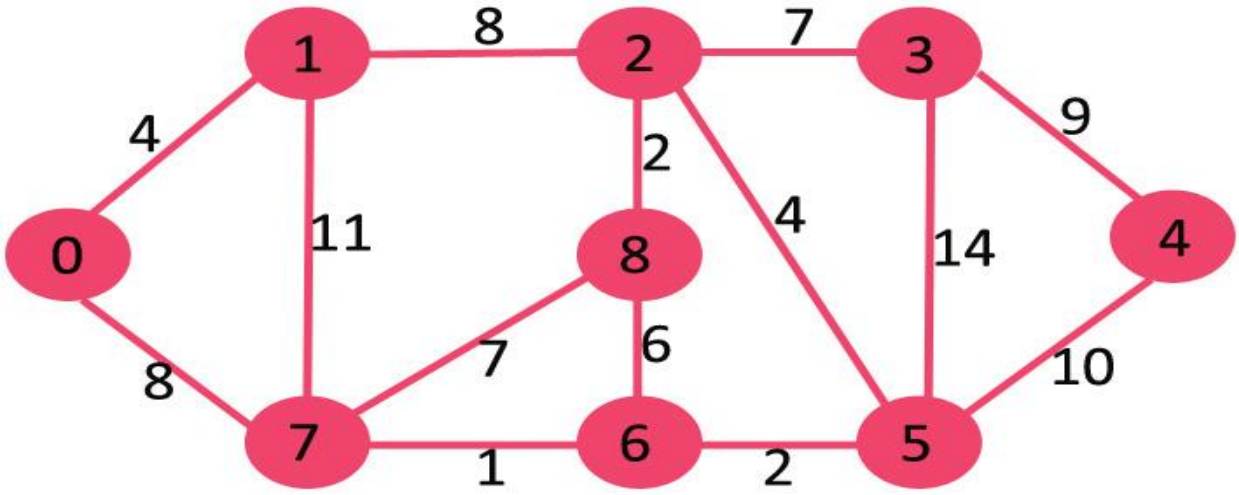
Dijkstra's Algorithm works on the basis that any sub path B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D. Djikstra used this

property in the opposite direction i.e. we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest sub path to those neighbors.

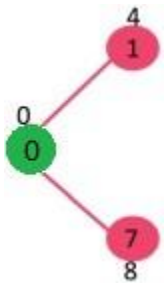
Dijkstra's Algorithm solves the single source shortest path problem in **O ((E + V)logV)** time, which can be improved to **O(E + VlogV)** when using a Fibonacci heap. This note requires that you understand basic graph theory terminology and concepts.

The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem. Algorithm is explained in the images given below

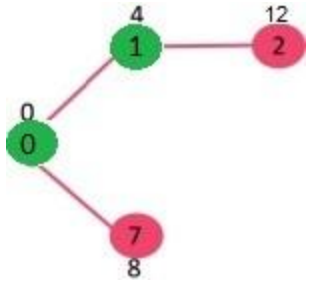
Let us understand with the following example:



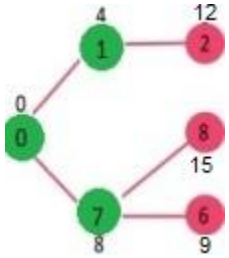
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update 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. 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 color.



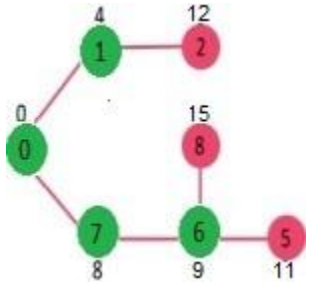
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



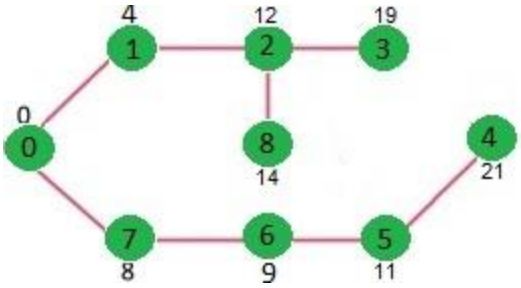
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



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.



We repeat the above steps until *sptSet* doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



Code for Dijkstra Algorithm:

```
# Python program for Dijkstra's single
# source shortest path algorithm. The program is
# for adjacency matrix representation of the graph

# Library for INT_MAX
import sys

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printSolution(self, dist):
        print "Vertex tDistance from Source"
        for node in range(self.V):
            print node,"t",dist[node]

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minDistance(self, dist, sptSet):

        # Initilaize minimum distance for next node
        min = sys.maxint

        # Search not nearest vertex not in the
        # shortest path tree
        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
```

```
                min = dist[v]
                min_index = v

        return min_index

    # Funtion that implements Dijkstra's single source
    # shortest path algorithm for a graph represented
    # using adjacency matrix representation
    def dijkstra(self, src):

        dist = [sys.maxint] * self.V
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):

            # Pick the minimum distance vertex from
            # the set of vertices not yet processed.
            # u is always equal to src in first iteration
            u = self.minDistance(dist, sptSet)

            # Put the minimum distance vertex in the
            # shotest path tree
            sptSet[u] = True

            # Update dist value of the adjacent vertices
            # of the picked vertex only if the current
            # distance is greater than new distance and
            # the vertex in not in the shotest path tree
            for v in range(self.V):
                if self.graph[u][v] > 0 and sptSet[v] == False and
                  dist[v] > dist[u] + self.graph[u][v]:
                     dist[v] = dist[u] + self.graph[u][v]

        self.printSolution(dist)

# Driver program
g  = Graph(9)
g.graph = [[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]
       ];

g.dijkstra(0);
```