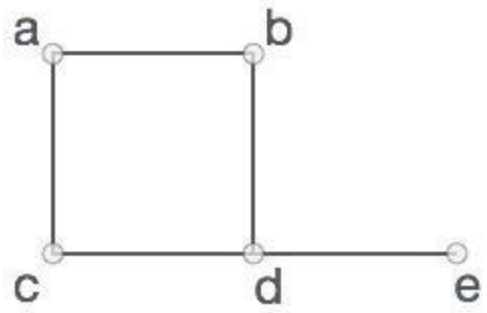
Unit 9 (Graphs)

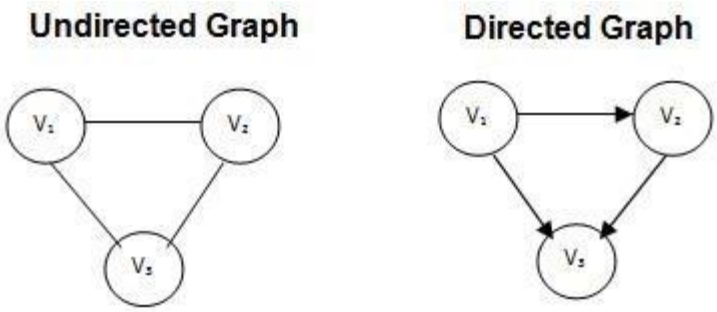
1. Representation and applications
2. Transitive closure
3. Warshall's algorithm
4. Graphs type
5. Graph traversal and Spanning forests
    - Depth First Traversal and Breadth First Traversal
    - Topological sorting: Depth first, Breadth first topological sorting
    - Minimum spanning trees, Prim's, Kruskal's and Round- Robin algorithms
6. Shortest-path algorithm
    - Greedy algorithm
    - Dijkstra's Algorithm

Representation and Application

- A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.
- Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices.
- Take a look at the following graph.

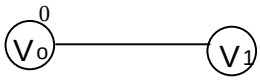


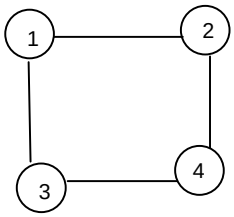
- In the above graph, ○ V = {a, b, c, d, e} ○ E = {ab, ac, bd, cd, de}



Undirected:-

A graph which has unordered pair of vertices is called undirected graph. Suppose there is an edge between V_0 & V_1 then it can be represented as (V , V_1) or (V_1, V_0).



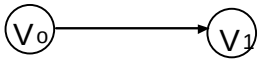


V(G) = { 1, 2, 3, 4}

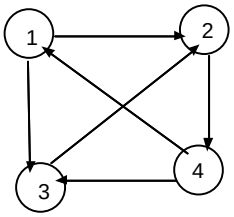
E(G) = {(1, 2), (1, 3), (2, 4), (3, 4)}

Directed:-

A directed graph or digraph is a graph which has ordered pair of vertices (V_1, V_2) where V_1 is the tail & V_2 is the head of the edge.



(V_1, V_2)



V(G) = { 1, 2, 3, 4}

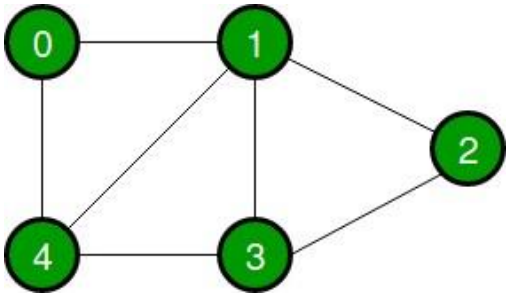
E(G) = {(1, 2), (1, 3), (2, 4),(3,2), (4, 3),(4,1)}

Representation of graphs:

The following two are the most commonly used representations of a graph.

1. Adjacency Matrix
2. Adjacency List

Let us consider an undirected graph.

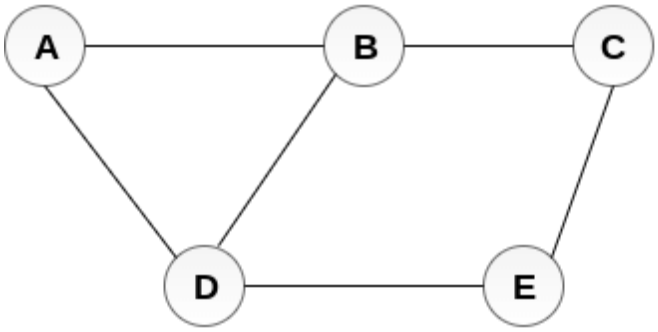


Adjacency Matrix:

- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.
- Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

- The adjacency matrix for the above example graph is:
- Adjacency Matrix Representation

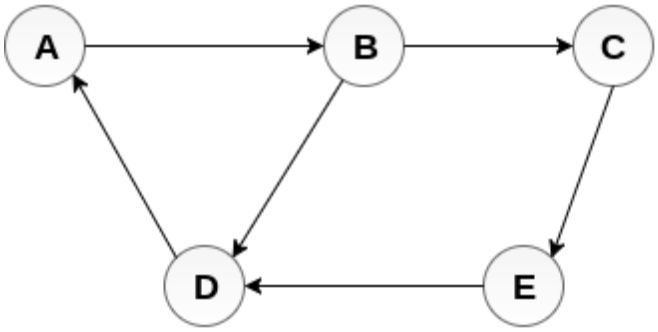
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0



Undirected Graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

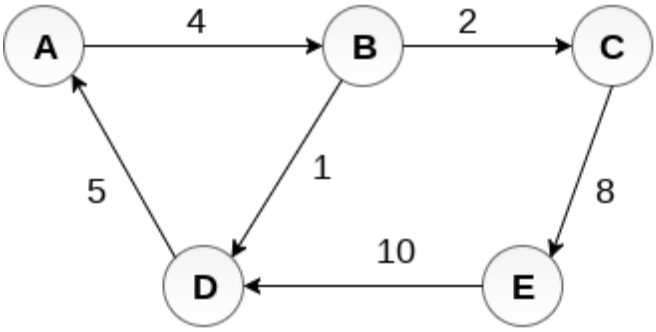
Adjacency Matrix



Directed Graph

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

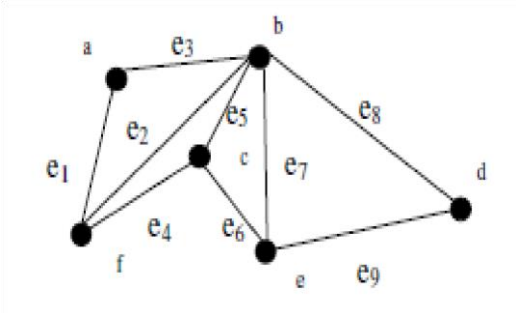
Adjacency Matrix



Weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

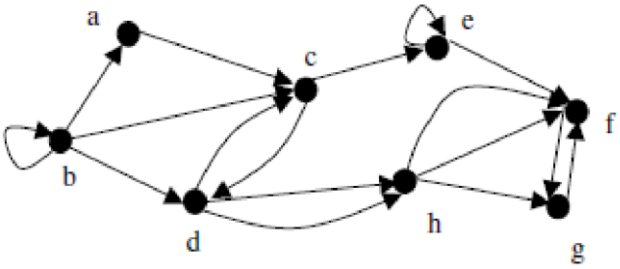


Solution:Let the order of the vertices be a, b, c, d, e, f

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Adjacency Matrix

Let us take a directed graph



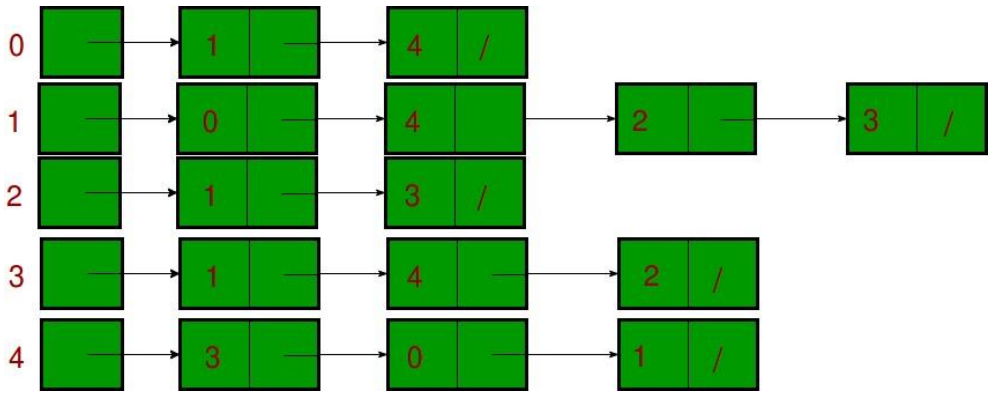
Solution:

Let the order of the vertices be a, b, c, d, e, f, g

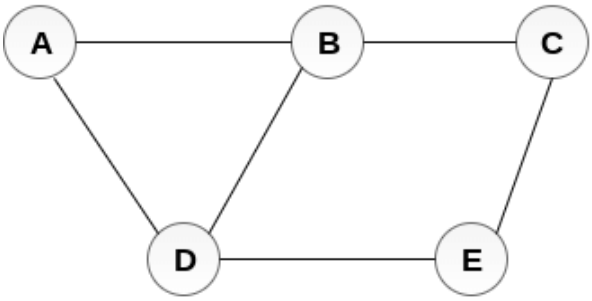
$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 1 & 0 \end{bmatrix}$$

Adjacency List:

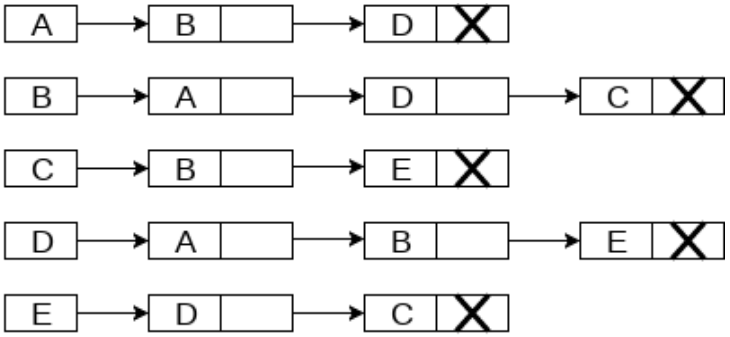
- An array of linked lists is used.
- The size of the array is equal to the number of vertices. Let the array be an array[].
- An entry array[i] represents the list of vertices adjacent to the ith vertex.
- This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.
- Following is the adjacency list representation of the above graph.



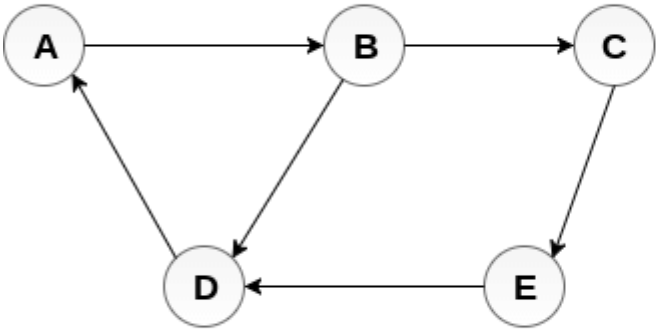
Compiled by: Er. Sandesh S Poudel



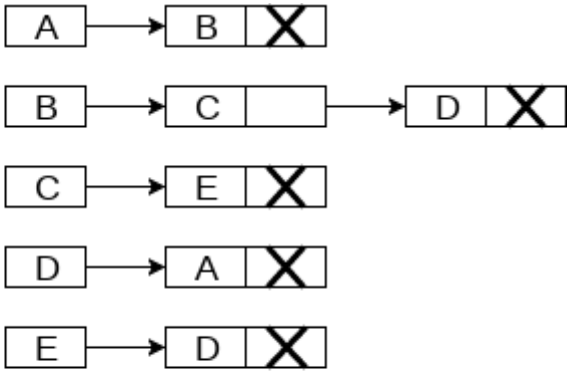
Undirected Graph



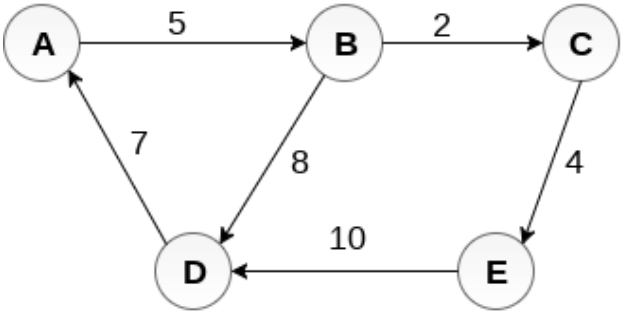
Adjacency List



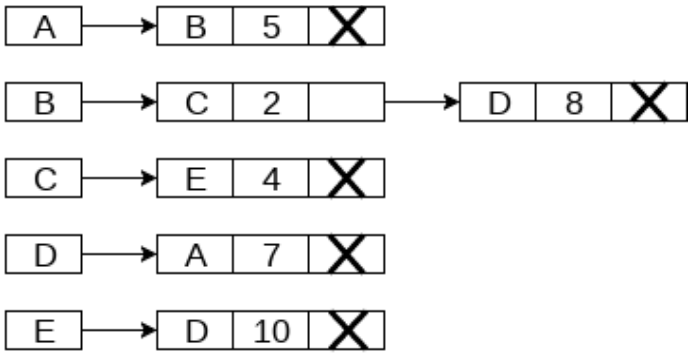
Directed Graph



Adjacency List



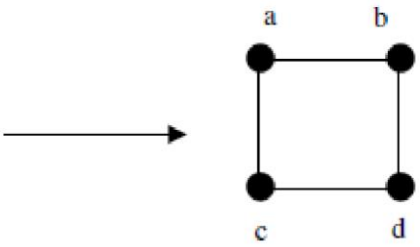
Weighted Directed Graph



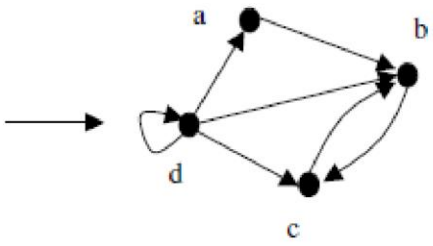
Adjacency List

Compiled by: Er. Sandesh S Poudel

Edge List for Simple Graph	
Vertex	Adjacent Vertices
a	b, c
b	a, d
c	a, d
d	b, c



Edge List for Directed Graph	
Initial Vertex	End Vertices
a	b
b	c
c	b
d	a, b, c, d



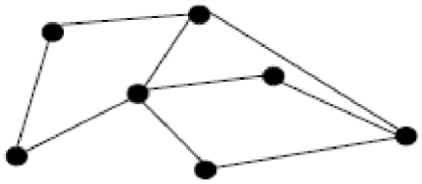
Application of Graphs

- In Computer science graphs are used to represent the flow of computation.
- Google maps uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In Facebook, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of undirected graph.
- In World Wide Web, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of Directed graph. It was the basic idea behind Google Page Ranking Algorithm.
- In Operating System, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.
- LAN
- Communication, Transport Network
- Job assignment
    - Circuit design in the Planer Circuit Board (PCB)

Types of Graph:

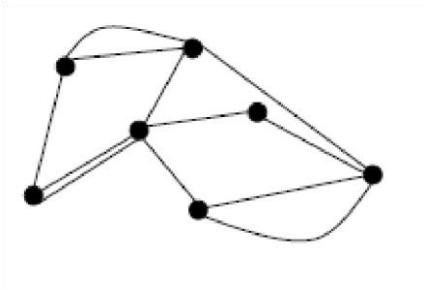
Simple Graph:

We define a simple graph as 2 – tuple consists of a non-empty set of vertices V and a set of unordered pairs of distinct elements of vertices called edges. We can represent graph as G = (V, E). This kind of graph has no loops and can be used for modeling networks that do not have connection to themselves but have both ways connection when two vertices are connected but no two vertices have more than one connection. The figure below is an example of simple graph.



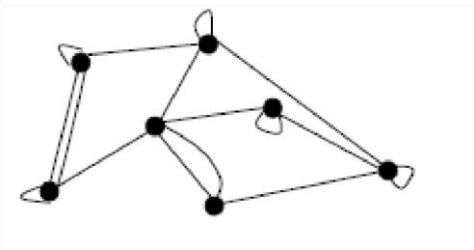
Multigraph:

A multigraph G =(V, E) consists of a set of vertices V, a set of edges E, and a function *f* from E to {{u, v}|u, v Î V, u ¹ v}. The edges e_1 and e_2 are called multiple or parallel edges if $f(e_1) = f(e_2)$. In this representation of graph also loops are not allowed. Since simple graph has single edges every simple graph is a multigraph. The figure below is an example of a multigraph.



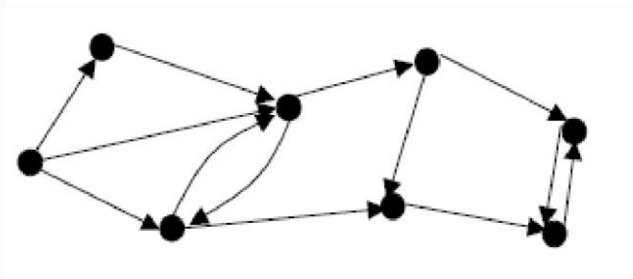
Pseudograph:

A pseudograph G =(V, E) consists of a set of vertices V, a set of edges E, and a function *f* from E to {{u, v}|u, v ∈ V}. An edge is a loop if $f(e) = \{u, u\} = \{u\}$ for some u ∈ V. The figure below is an example of a multigraph



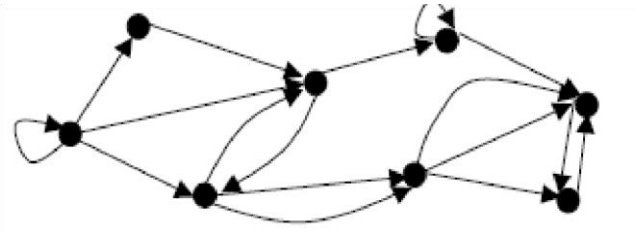
Directed Graph:

A directed graph (V, E) consists of a set V of vertices, a set E of edges that are ordered pairs of elements of V. The below figure is a directed graph. In this graph loop is allowed but no two vertices van have multiple edges in same direction.



Directed Multigraph:

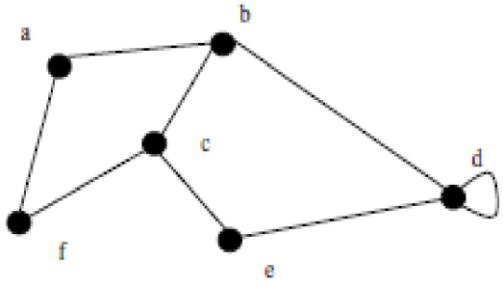
A directed multigraph G =(V, E) consists of a set of vertices V, a set of edges E, and a function *f* from E to {(u, v)|u, v ∈ V}. The edges e_1 and e_2 are called multiple edges if $f(e_1) = f(e_2)$. The figure below is an example of a directed multigraph.



Terminologies:

Two vertices u, v are adjacent vertices of a graph if {u, v} is an edge. The edge e is called incident with the vertices u and v if e = {u, v}. This edge is also said to connect u and v. where u and v are end points of the edge. **Degree of a vertex** in an undirected graph is the number of edges incident with it, except a loop at a vertex. Loop in a vertex counts twice to the degree. Degree of a vertex v is denoted by deg (v).A vertex of degree zero is called isolated vertex and a vertex with degree one is called pendant vertex.

Example: Find the degrees of the vertices in the following graph.



Solution:

deg(a) = deg(f) = deg(e) = 2 ; deg(b) = deg(c) = 3; deg(d) = 4

Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0=V_N$.

Simple Path

If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simple path.

Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop

An edge that is associated with the similar end points can be called as Loop.

Compiled by: Er. Sandesh S Poudel

Adjacent Nodes

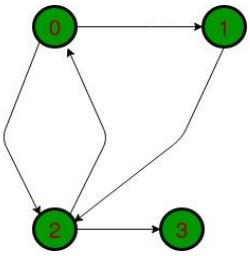
If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Transitive closure

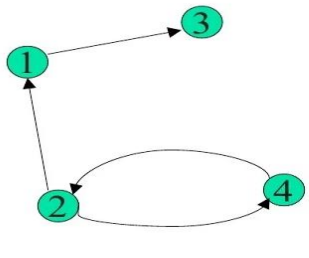
- Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j. The reach-ability matrix is called the transitive closure of a graph.
- For example, consider below graph (with loop)



Transitive closure of above graphs is

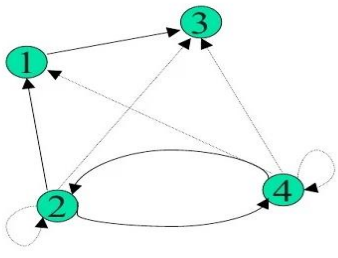
	0	1	2	3
0	1	1	1	1
1	1	1	1	1
2	1	1	1	1
3	0	0	0	1

Alternative method: (without loop)



Adjacency matrix

0	0	1	0
1	0	0	1
0	0	0	0
0	1	0	0



Transitive closure

0	0	1	0
1	**1**	**1**	1
0	0	0	0
1	1	**1**	**1**

Compiled by: Er. Sandesh S Poudel

Warshall's algorithm for finding transitive closure from diagraph

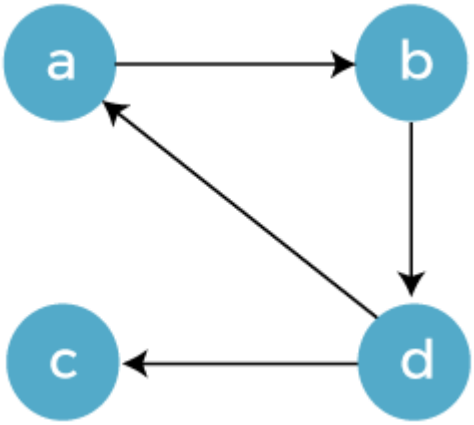
- In wars hall's algorithm we construct a sequence of Boolean matrices A = W[0], W[1], W[2],….W[n] =T, where A is adjacency matrix and T is its transitive closure. This can be done from digraph D as follows.
- [W[1]]i,j = 1 if and only if there is a path from vi to vj with elements of a subset of {v1} as interior vertices.
- [W[2]]i,j =1 if and only if there is a path from vi to vj with elements of a subset of {v1 ,v2} as interior vertices.

  Continuing this process, we generalized to

- [W[k]]i,j = 1 if and only if there is a path from vi to vj with elements of a subset of {v1,v2,….vk} as interior vertices.

Application of Warshall's algorithm to the directed graph

In order to understand this, we will use a graph, which is described as follow:



For this graph R(0) will be looked like this:

$$R^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Here R(0) shows the adjacency matrix. In R(0), we can see the existence of a path, which has no intermediate vertices. We will get R(1) with the help of a boxed row and column in R(0).

Compiled by: Er. Sandesh S Poudel

In R(1), we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 1, which means just a vertex. It contains a new path from d to b. We will get R(2) with the help of a boxed row and column in R(1).

$R^{(1)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

In R(2), we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 2, which means a and b. It contains two new paths. We will get R(3) with the help of a boxed row and column in R(2).

$R^{(2)} =$

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

In R(3), we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 3, which means a, b and c. It does not contain any new paths. We will get R(4) with the help of a boxed row and column in R(3).

$R^{(3)} =$

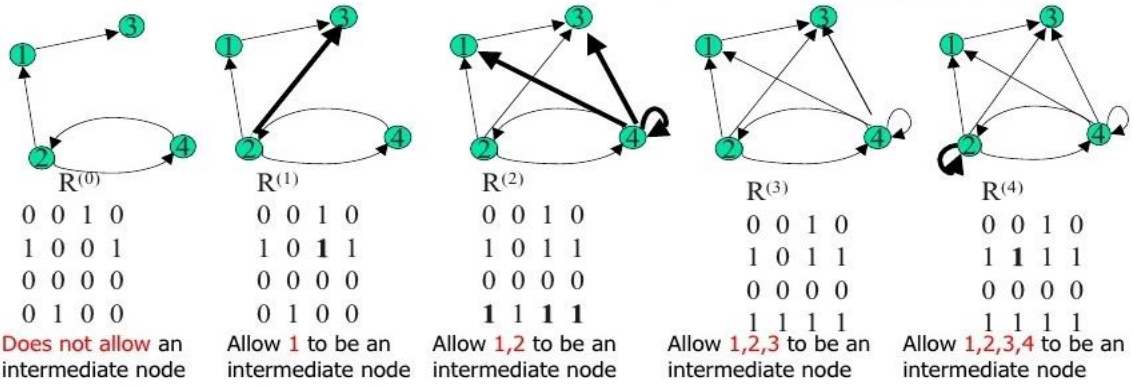
	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

In R(4), we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 4, which means a, b, c and d. It contains five new paths.

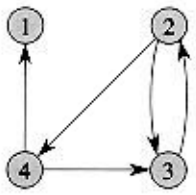
Compiled by: Er. Sandesh S Poudel

$$R^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} \mathbf{1} & 1 & \mathbf{1} & 1 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Example:



Example:(with loop)

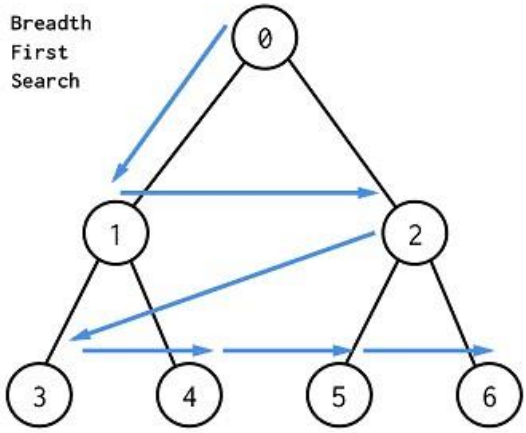


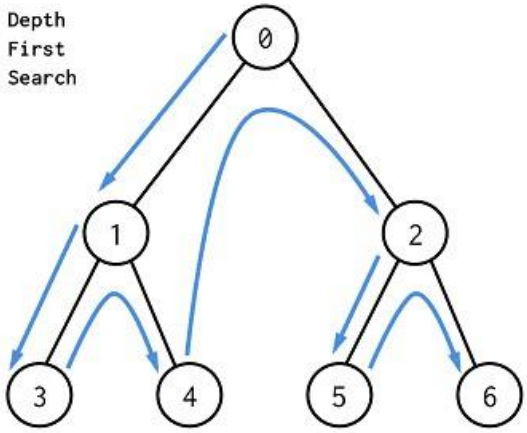
$$T^{k}(i,j) = T^{k-1}(i,j) \vee \left[T^{k-1}(i,k) \wedge T^{k-1}(k,j) \right]$$

$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

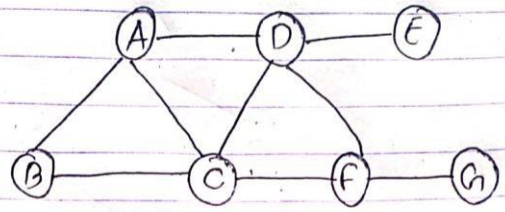
Graph Traversals





Breadth First Search (BFS)

- Given an input graph G = (V, E) and a source vertex S, from where the searching starts. The breadth first search systematically traverses the edges of G to explore every vertex that is reachable from S.
- Then we examine the entire vertices neighbour to source vertex S. Then we traverse all the neighbours of the neighbours of source vertex S and so on. A queue is used to keep track of the progress of traversing the neighbour nodes.
- Considered the following Figure for breadth first search traversing.



Let the starting vertex be 'B'

Initially

B	

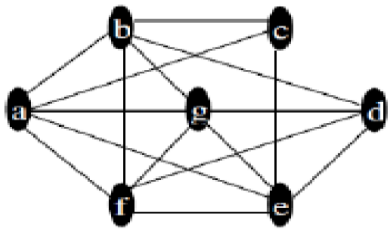
Queue

			Display
(1)	A	C	B
(2)	C	D	A
(3)	D	F	C
(4)	F	E	D
(5)	E	G	F
(6)	G		E
(7)			G

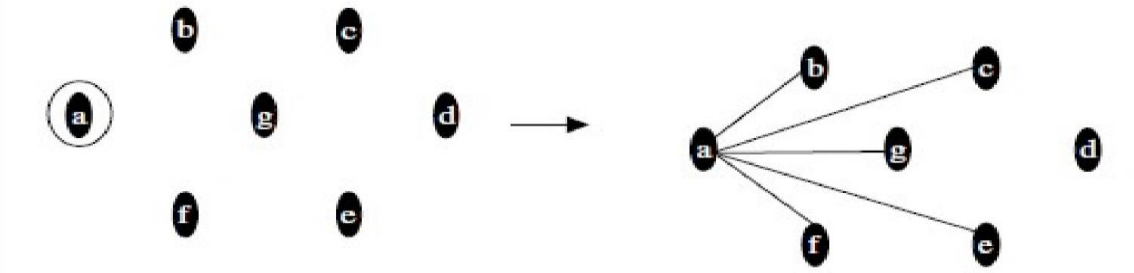
∴ Result of BFS on above Graph is BACDFEG.

Example:

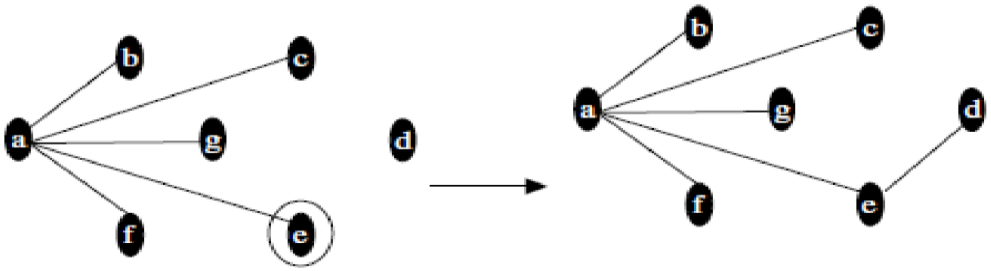
Use breadth first search to find a BFS tree of the following graph



Solution:
Choose a as initial vertex then we have



Order the vertices of level 1 i.e. {b, c, g, e, f}. Say order be {e, f, g, b, c}.



Algorithm:

```
BFS(G,s) //s is start vertex
{
    T = {s};
    L =Φ; //an empty queue Enqueue(L,s);
    while (L != Φ )
    {
        v = dequeue(L); for each neighbor w to v
        if ( w∉ L and w ∉ T )
            { enqueue( L,w);
                T = T U {w}; //put edge {v,w} also
            }
    }
        }
```

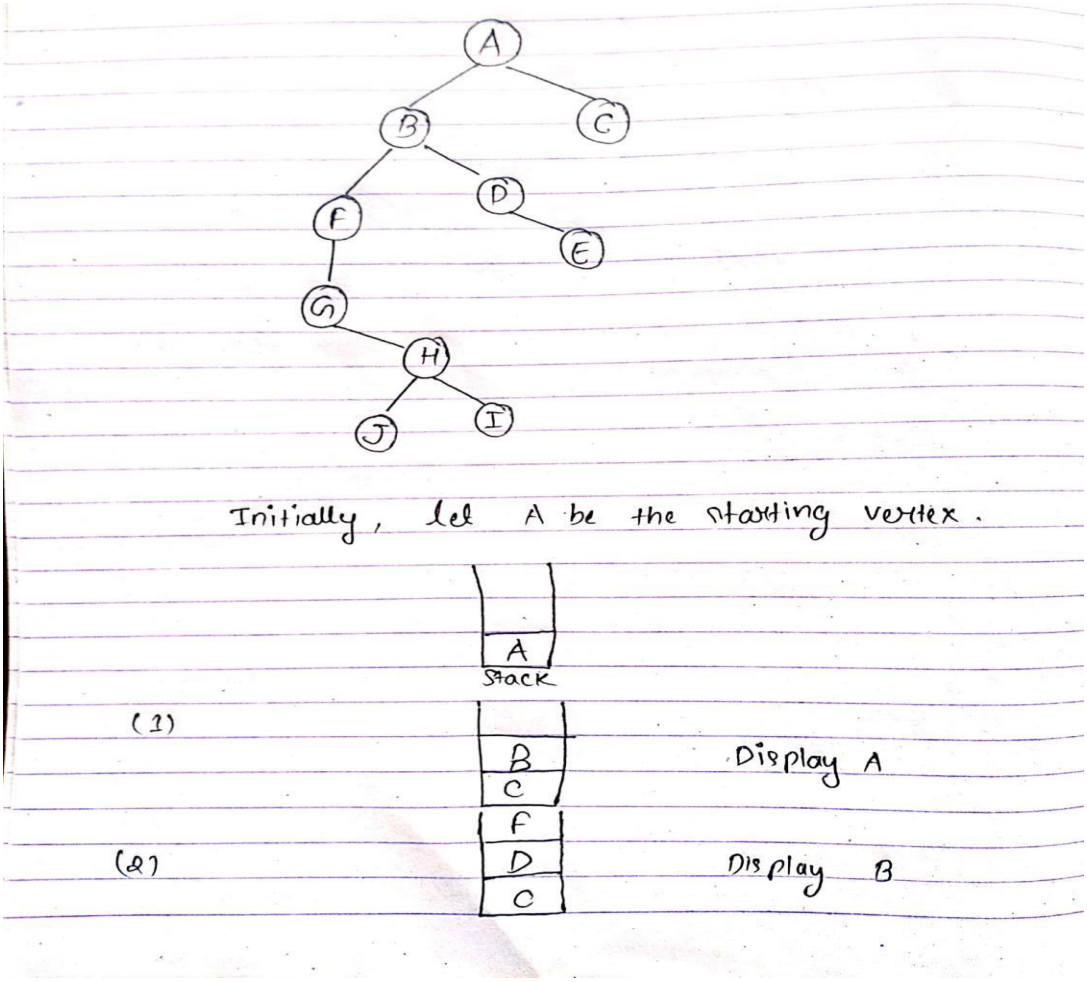
Analysis

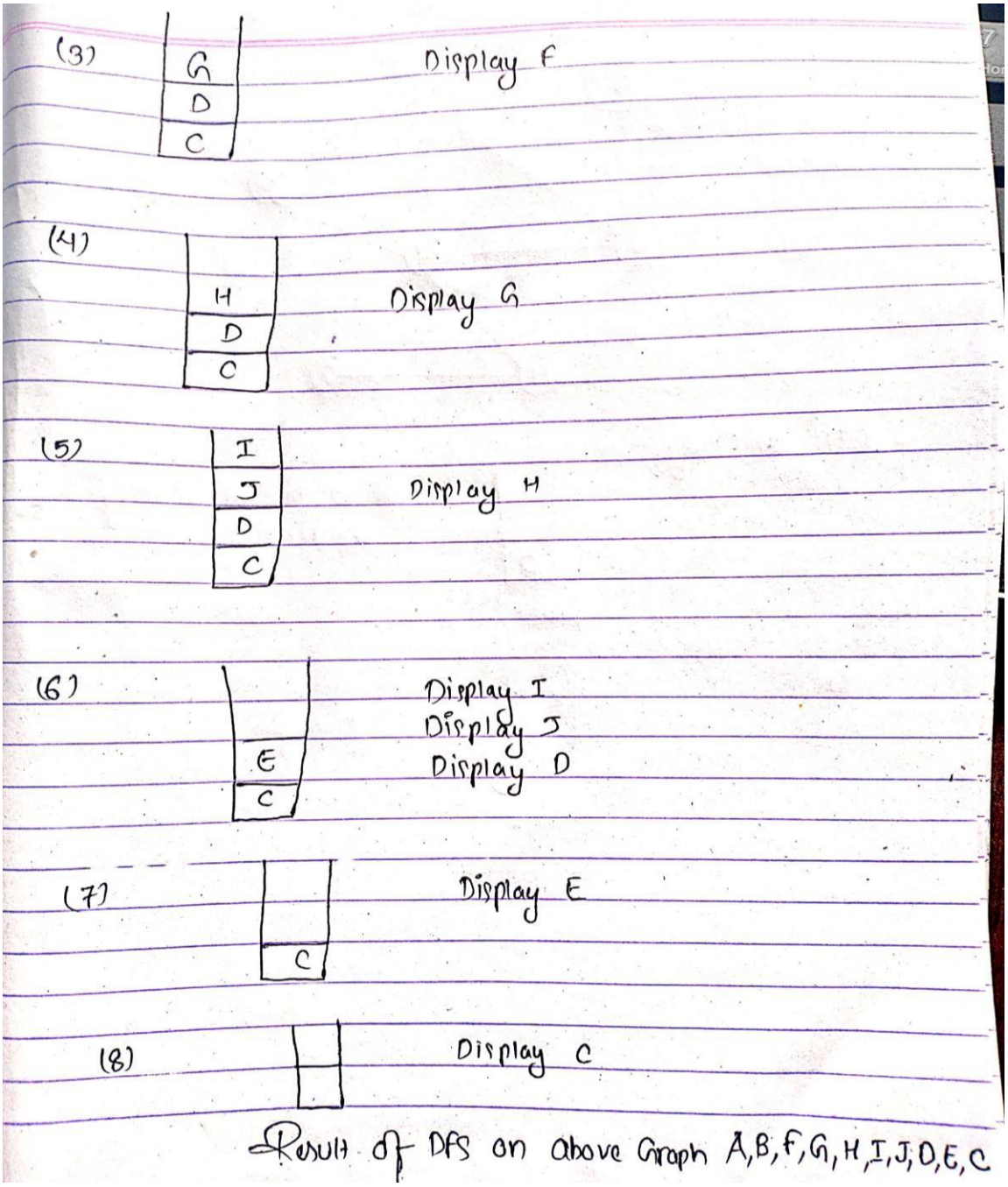
From the algorithm above all the vertices are put once in the queue and they are accessed. For each accessed vertex from the queue their adjacent vertices are looked for and this can be done in O(n) time(for the worst case the graph is complete). This computation for all the possible vertices that may be in the queue i.e. n, produce complexity of an algorithm as O(n2).

Depth First Search (DFS)

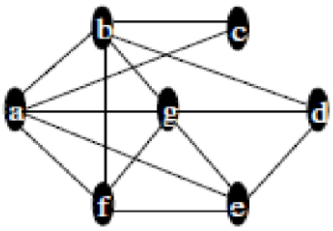
- The depth first search (DFS), as its name suggest, is to search deeper in the graph, whenever possible.
- Given an input graph G = (V, E) and a source vertex S, from where the searching starts. First we visit the starting node. Then we travel through each node along a path, which begins at S. That is we visit a neighbour vertex of S and again a neighbour of a neighbour of S, and so on. The implementation of
- BFS is almost same except a stack is used instead of the queue.

The following steps will illustrate the DFS:



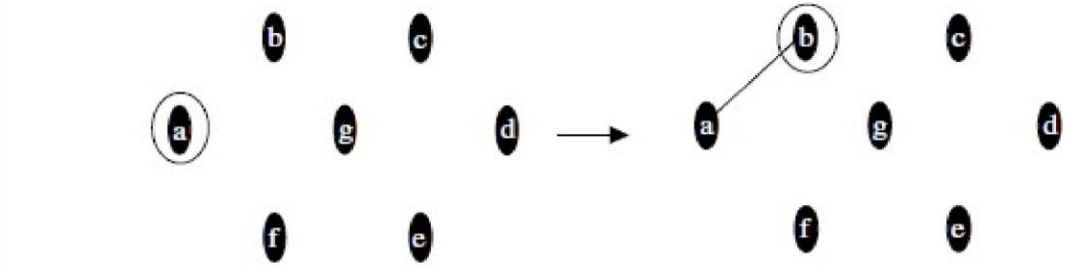


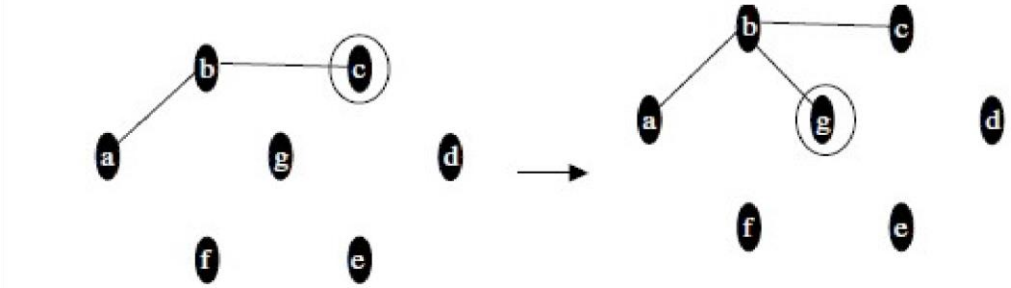
Example: Use depth first search to find a spanning tree of the following graph.

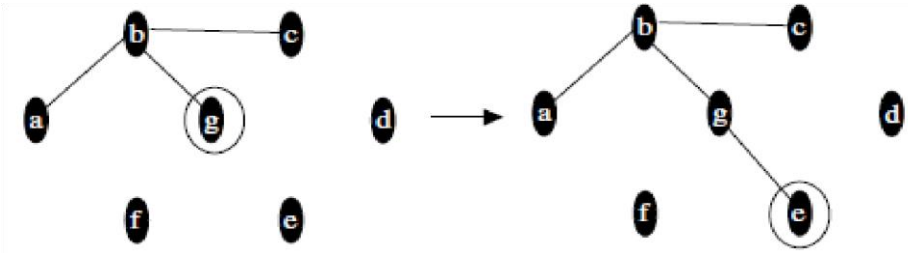


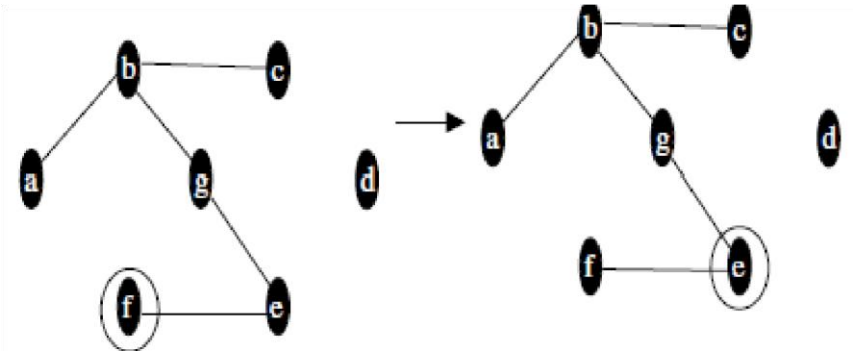
Solution:

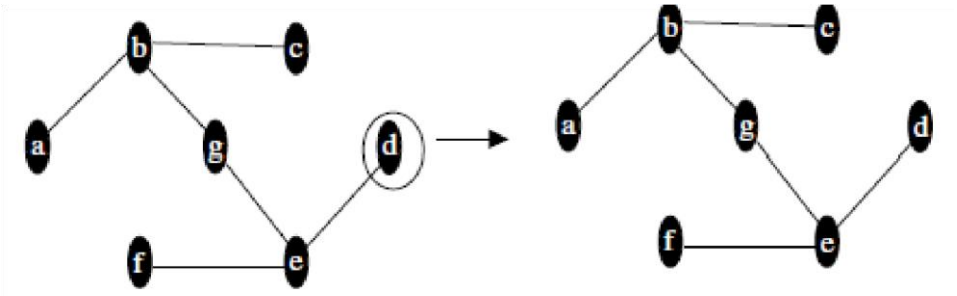
Choose a as initial vertex then we have











Algorithm:

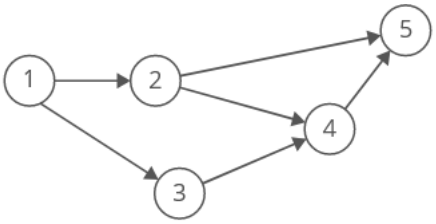
```
DFS(G,s)
{
        T = {s};
        Traverse(s);
}
Traverse(v)
{
        for each w adjacent to v and not yet in T
        {
                T = T U {w}; //put edge {v,w} also
                Traverse (w);
        }
}
```

Analysis:

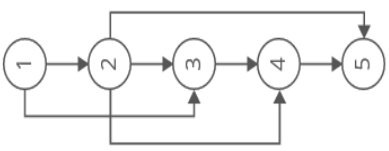
The complexity of the algorithm is greatly affected by **Traverse** function we can write its running time in terms of the relation T(n) = T(n-1) + O(n), here O(n) is for each vertex at most all the vertices are checked (for loop). At each recursive call a vertex is decreased. Solving this we can find that the complexity of an algorithm is $O(n_2)$.

Topological Sort

Topological sorting for Directed Acyclic Graph (DAG)(**directed graph with no directed cycles)** is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG. Two Different algorithm are used: DFS and BFS. Here's an example:

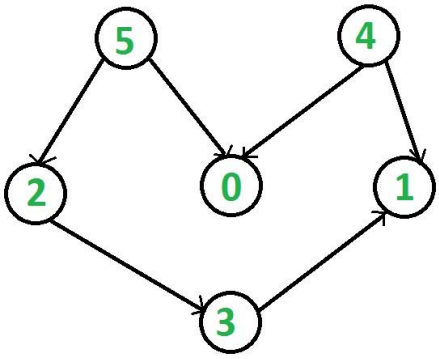


Since node 1 points to nodes 2 and 3, node 1 appears before them in the ordering. And, since nodes 2 and 3 both point to node 4, they appear before it in the ordering.



So [1, 2, 3, 4, 5] would be a topological ordering of the graph.

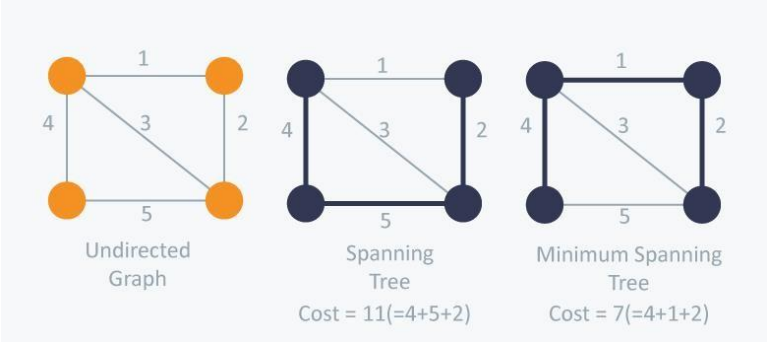
Example:



- For example, a topological sorting of the following graph is “5 4 2 3 1 0”.
- There can be more than one topological sorting for a graph.
- For example, another topological sorting of the following graph is “4 5 2 3 1 0”.

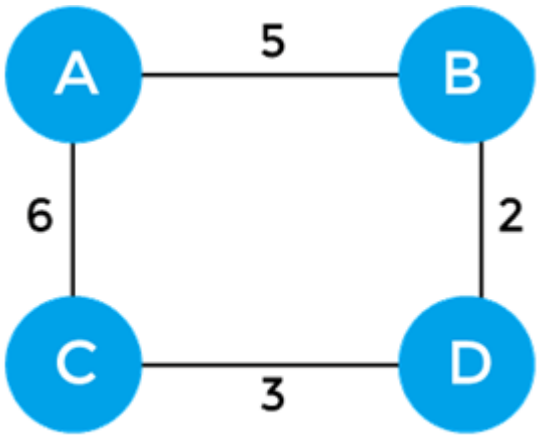
Minimum Spanning Tree

- Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together.
- A single graph can have many different spanning trees.
- A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree.
- The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.



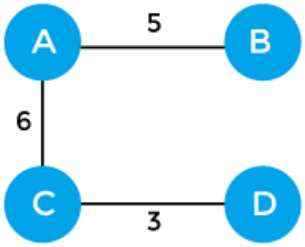
Example of minimum spanning tree

Let's understand the minimum spanning tree with the help of an example.

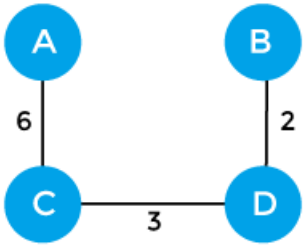


Weighted graph

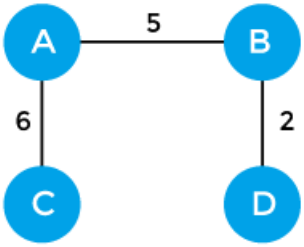
The sum of the edges of the above graph is 16. Now, some of the possible spanning trees created from the above graph are -



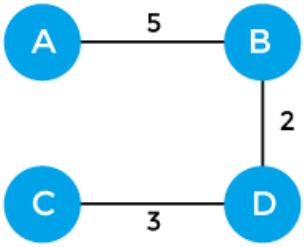
Sum = 14
Minimum spanning tree - 1



Sum = 11
Minimum spanning tree - 2

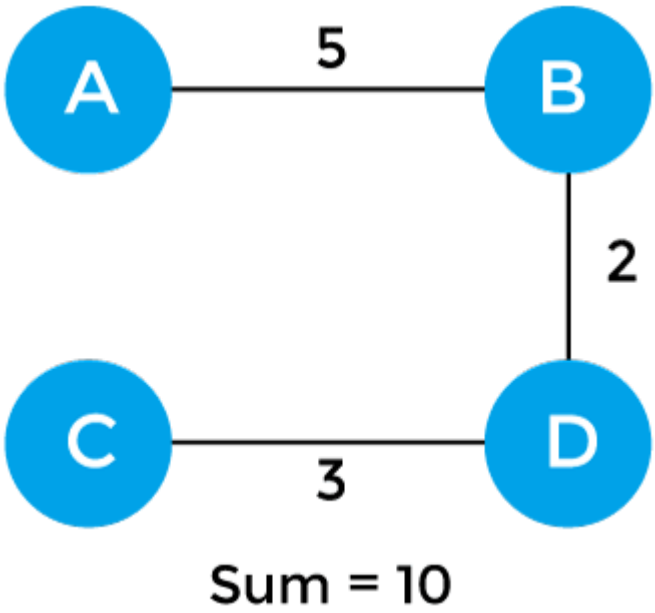


Sum = 13
Minimum spanning tree - 3



Sum = 10
Minimum spanning tree - 4

So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is -



Sum = 10

Compiled by: Er. Sandesh S Poudel

Kruskal’s algorithm

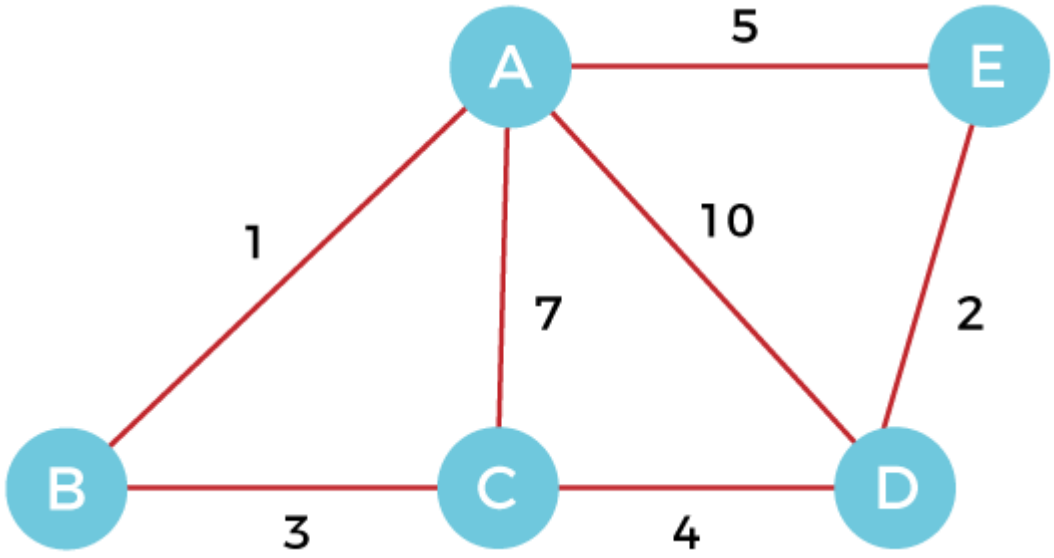
The problem of finding MST can be solved by using Kruskal’s algorithm. The idea behind this algorithm is that you put the set of edges form the given graph G = (V,E) in nondecreasing order of their weights. The selection of each edge in sequence then guarantees that the total cost that would from will be the minimum. Note that we have G as a graph, V as a set of n vertices and E as set of edges of graph G.

- Below are the steps for finding MST using Kruskal’s algorithm
  1. Sort all the edges in non-decreasing order of their weight.
  2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
  3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Example of Kruskal's algorithm

Now, let's see the working of Kruskal's algorithm using an example. It will be easier to understand Kruskal's algorithm using an example.

Suppose a weighted graph is -



The weight of the edges of the above graph is given in the below table -

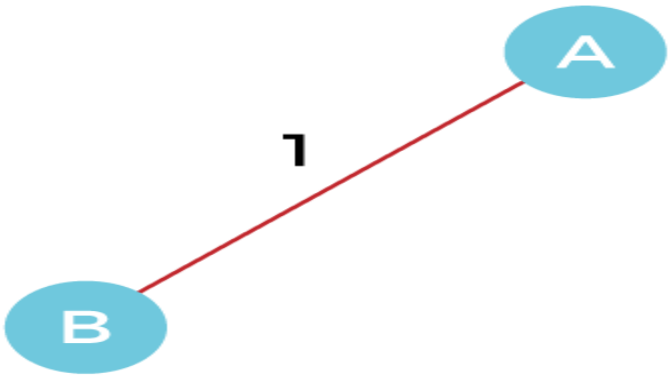
Edge	AB	AC	AD	AE	BC	CD	DE
Weight	1	7	10	5	3	4	2

Now, sort the edges given above in the ascending order of their weights.

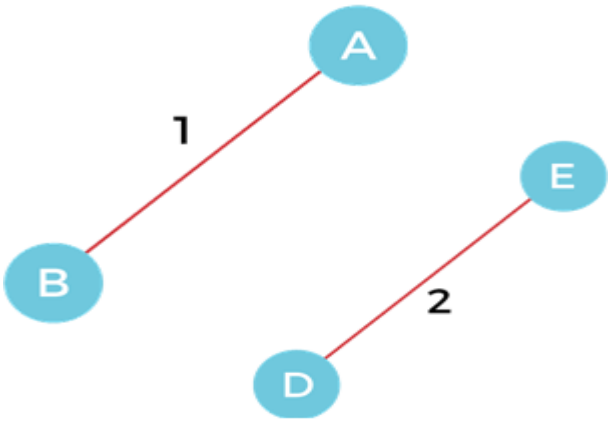
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Now, let's start constructing the minimum spanning tree.

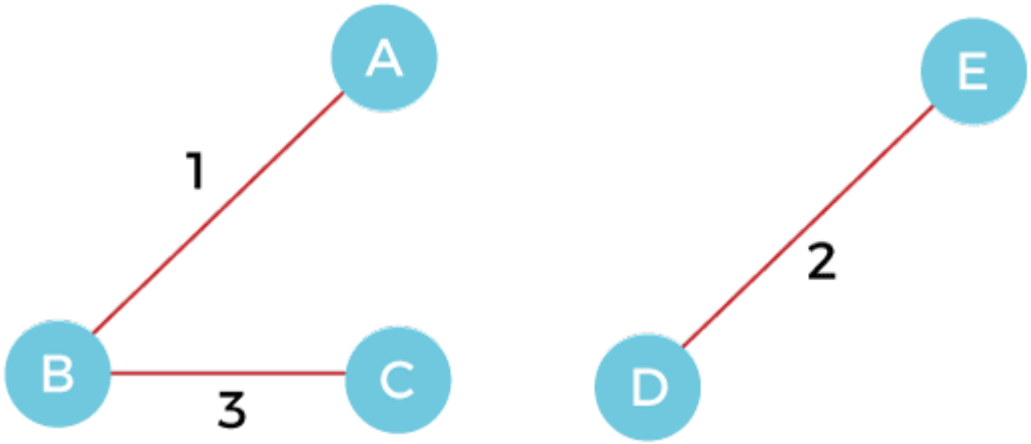
Step 1 - First, add the edge **AB** with weight **1** to the MST.



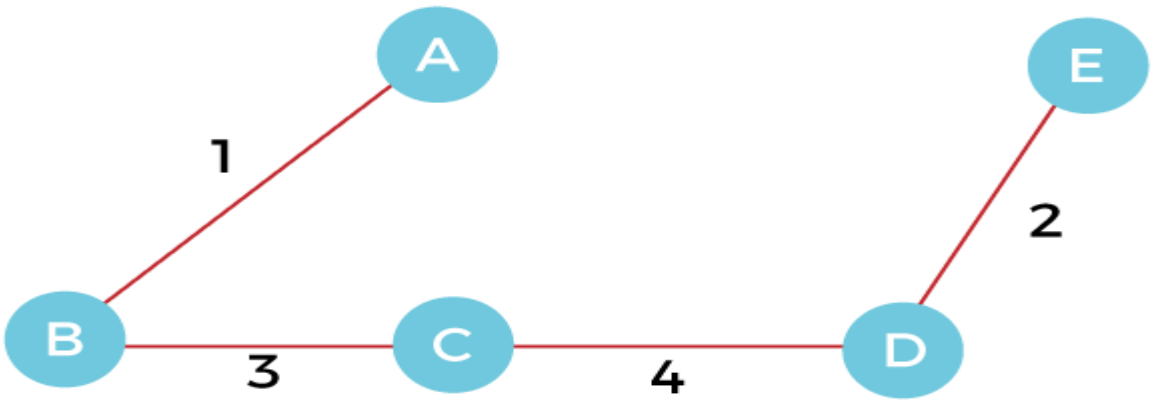
Step 2 - Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.



Step 3 - Add the edge **BC** with weight **3** to the MST, as it is not creating any cycle or loop.



Step 4 - Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle.

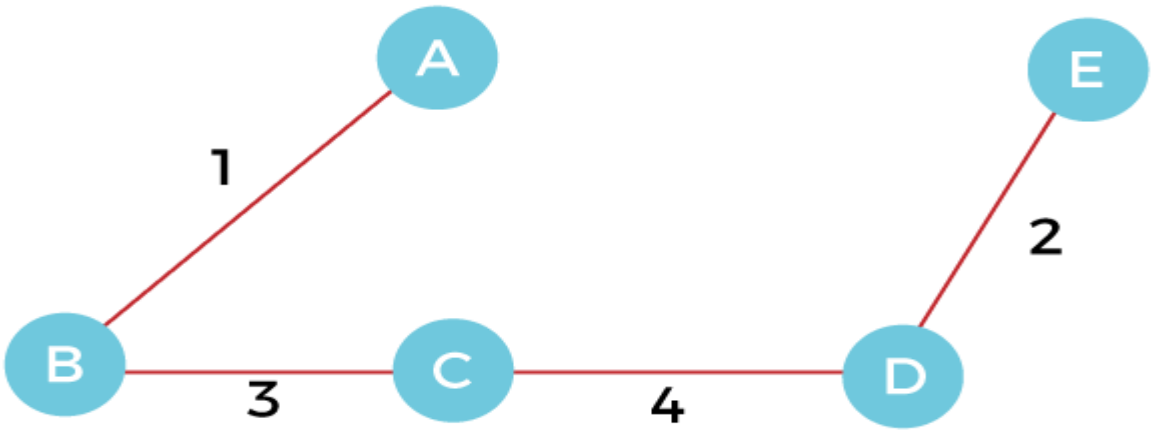


Step 5 - After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.

Step 6 - Pick the edge **AC** with weight **7.** Including this edge will create the cycle, so discard it.

Step 7 - Pick the edge **AD** with weight **10.** Including this edge will also create the cycle, so discard it.

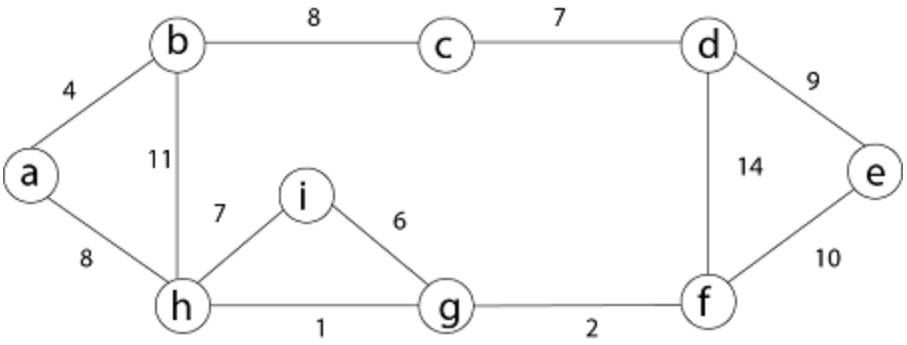
So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is -



The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.

Now, the number of edges in the above tree equals the number of vertices minus 1. So, the algorithm stops here.

Example: Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.



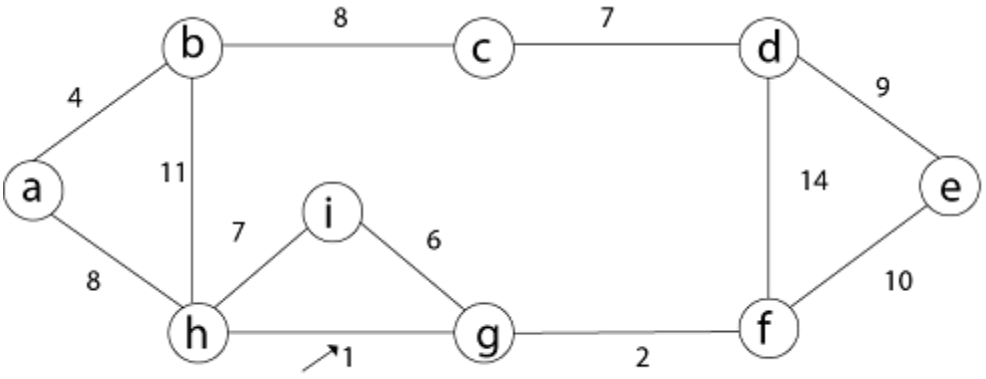
Solution: First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.

There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

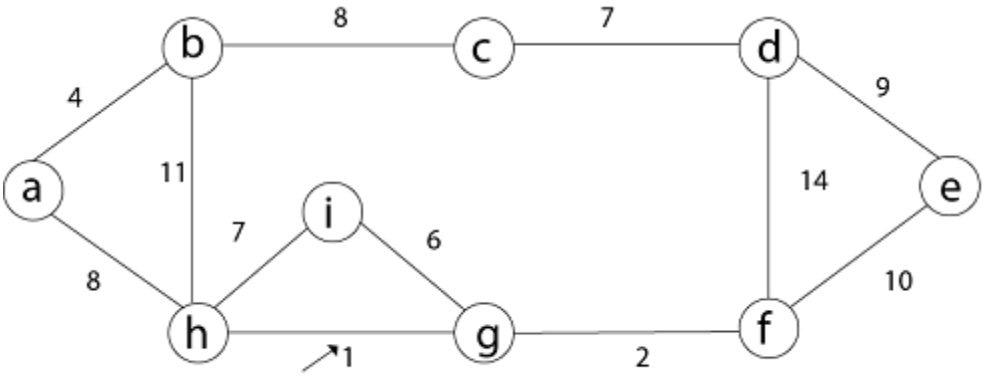
Weight	Source	Destination
1	h	g
2	g	f
4	a	b
6	i	g
7	h	i
7	c	d
8	b	c
8	a	h
9	d	e
10	e	f
11	b	h
14	d	f

Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.

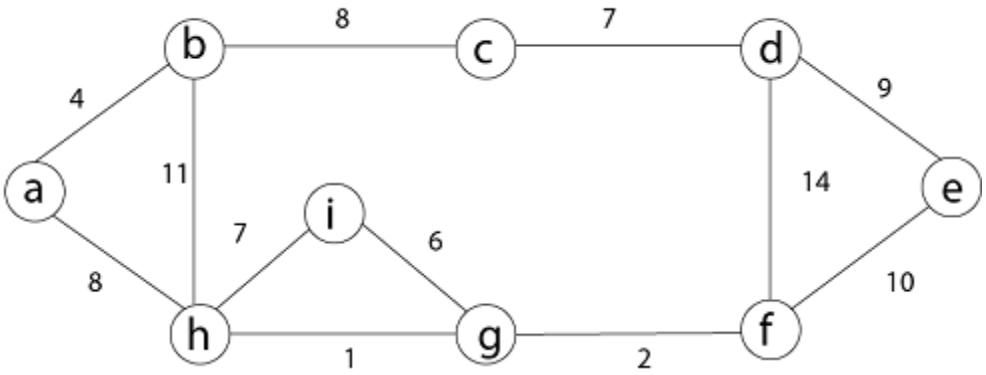
Step1: So, first take (h, g) edge



Step 2: then (g, f) edge.

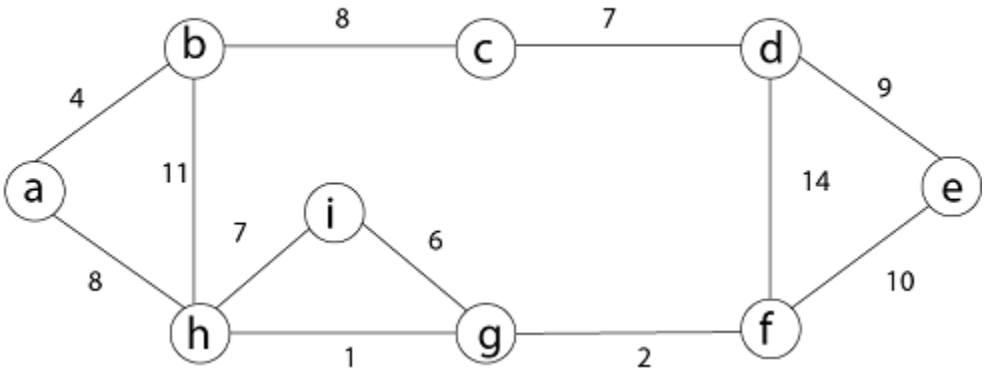


Step 3: then (a, b) and (i, g) edges are considered, and the forest becomes



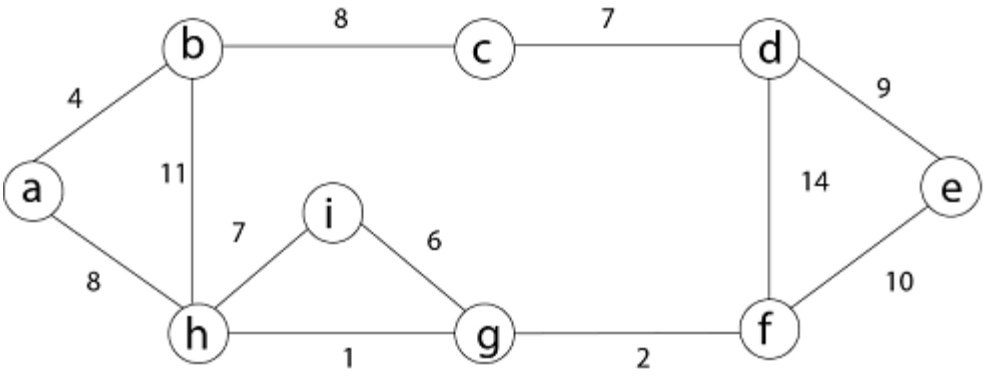
Step 4: Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded.

Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes.



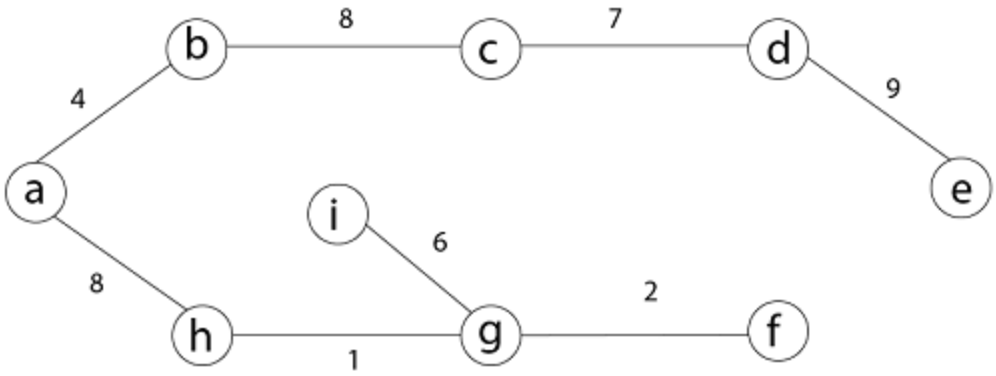
Step 5: In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.

Step 6: After that edge (d, f) and the final spanning tree is shown as in dark lines.



Step 7: This step will be required Minimum Spanning Tree because it contains all the 9 vertices and (9 - 1) = 8 edges

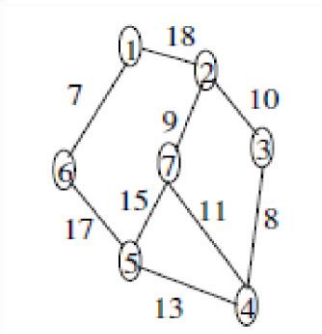
e → f, b → h, d → f [cycle will be formed]



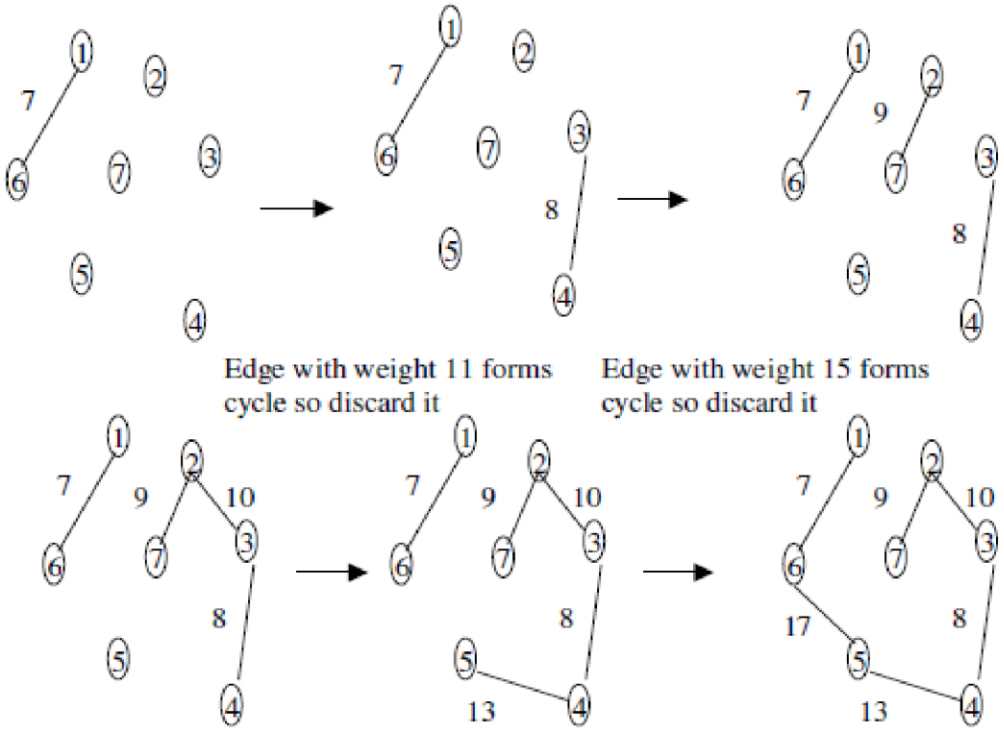
Minimum Cost MST=1+2+6+8+4+8+7+9=45

Example:

Find the MST and its weight of the graph.

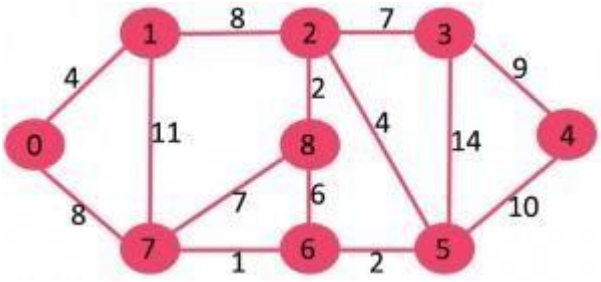


Solution:



The total weight of MST is 64.

Example 2: Let us understand it with an example: Consider the below input graph.



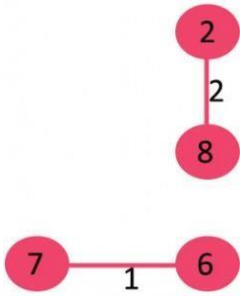
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

After sorting:

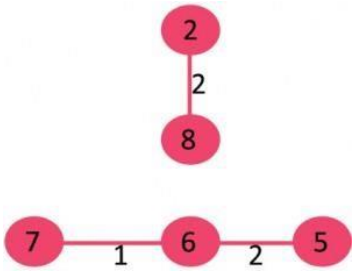
Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Step 1:

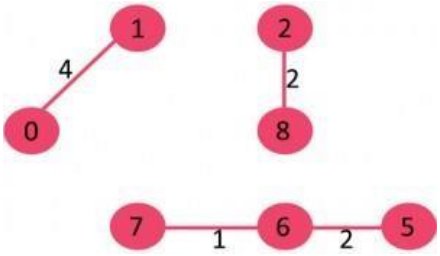
Step 2:



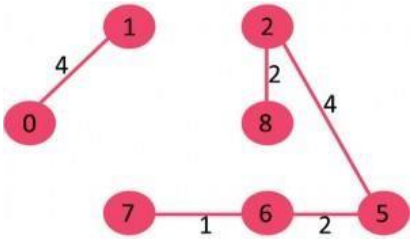
Step 3:



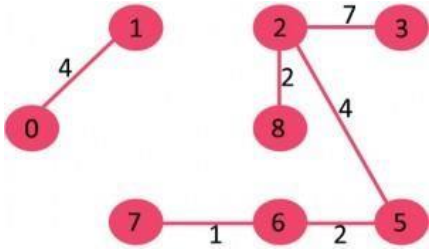
Step 4:



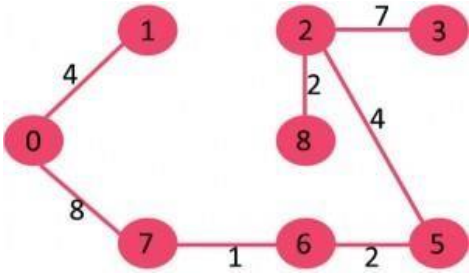
Step 5:



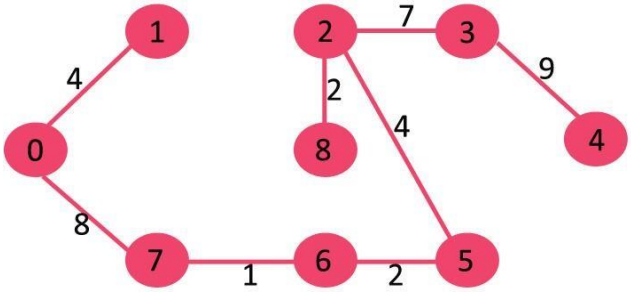
Step 6:



Step 7:



Step 8:



Compiled by: Er. Sandesh S Poudel

Algorithm:

```
KruskalMST(G)
{
    T = {V} // forest of n nodes
    S = set of edges sorted in nondecreasing order of weightwhile(|T| < n-1 and E !=Æ)
    {
        Select (u,v) from S in order Remove (u,v)
        from E
        if((u,v) doesnot create a cycle in T))
        T = T È {(u,v)}
    }

}
```

Analysis:

In the above algorithm the n tree forest at the beginning takes (V) time, the creation of set S takes O(ElogE) time and while loop execute O(n) times and the steps inside the loop take almost linear time (see disjoint set operations; find and union). So the total time taken is O(ElogE)

Prim's Algorithm

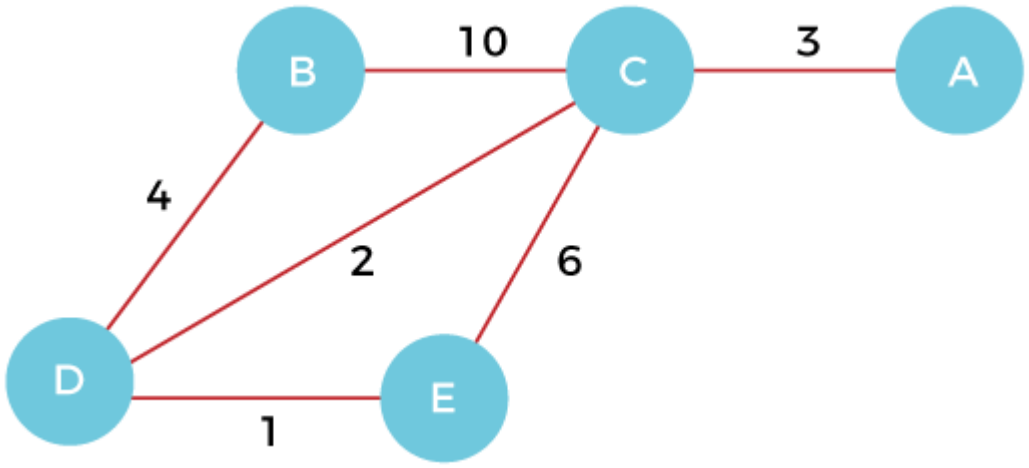
Algorithm

1. Create a set mstSet that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3. While mstSet doesn't include all vertices
    - Pick a vertex u which is not there in mstSet and has minimum key value.
    - Include u to mstSet.
    - Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

Example of prim's algorithm

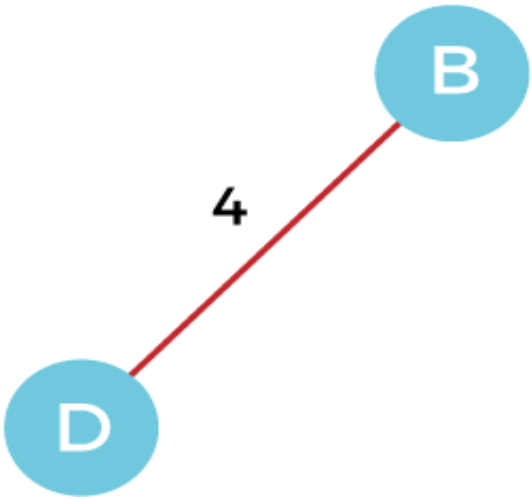
Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.

Suppose, a weighted graph is -

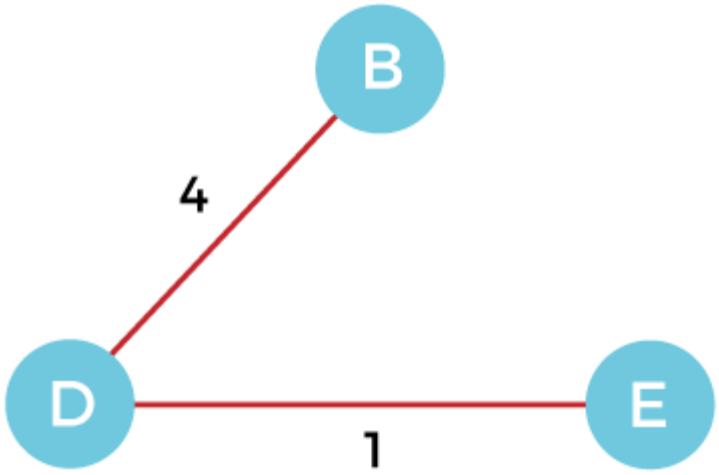


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

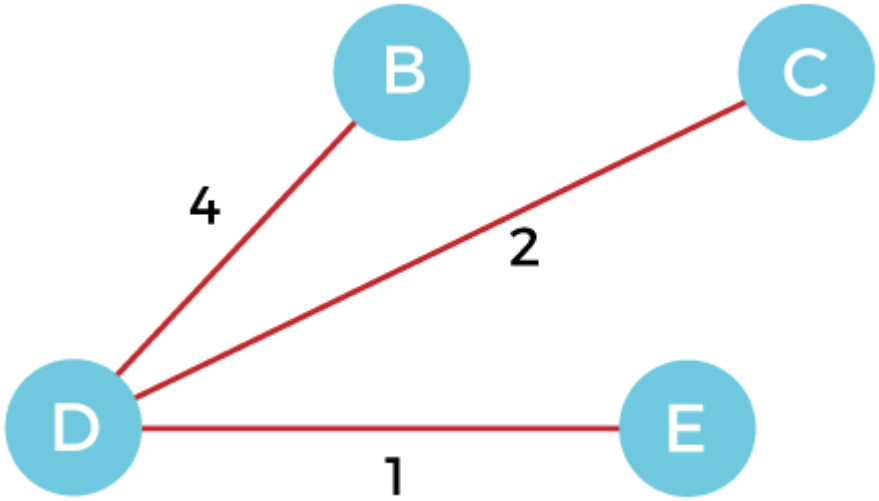
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



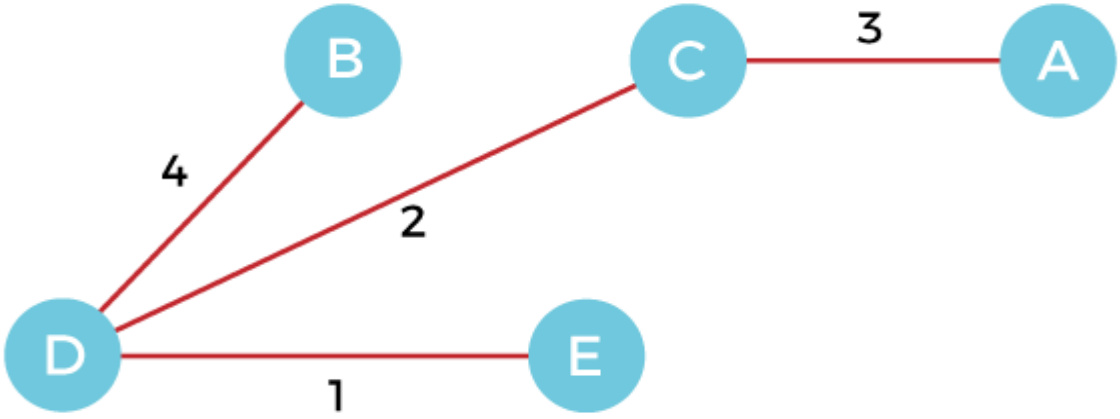
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4 - Now, select the edge CD, and add it to the MST.



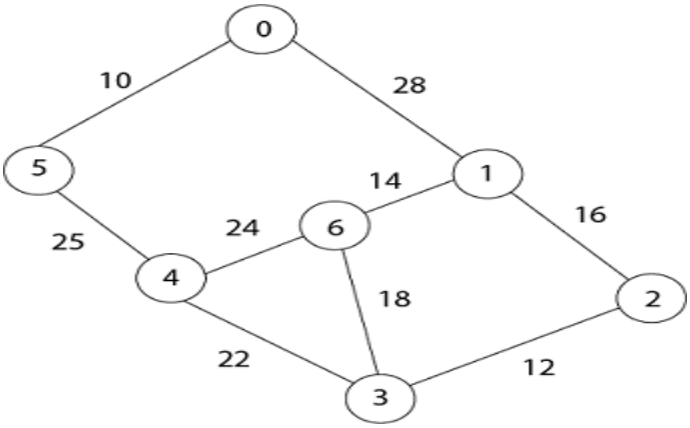
Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



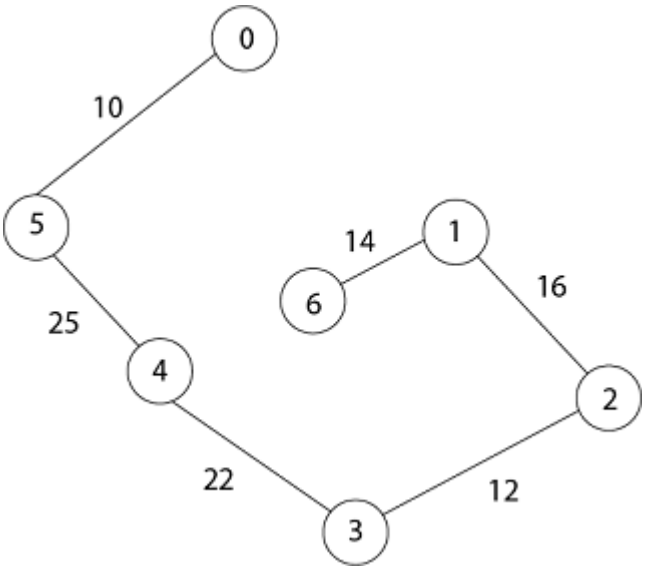
So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

Question: Generate minimum cost spanning tree for the following graph using Prim's algorithm.

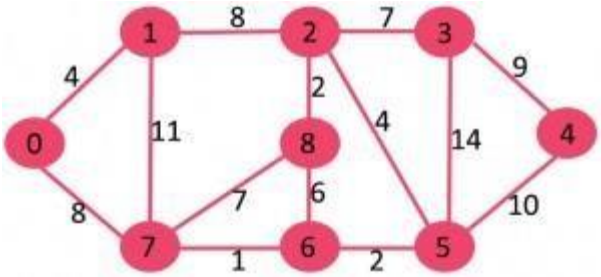


Answer: Thus the final spanning Tree is

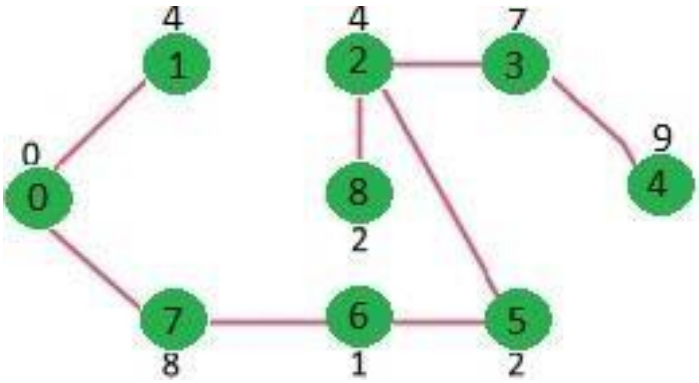


Total Cost = 10 + 25 + 22 + 12 + 16 + 14 = 99

Question:



Answer:

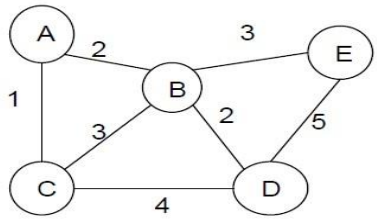


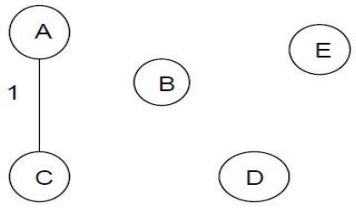
Total cost: 4+8+1+2+4+2+7+9=37

Round Robin Algorithm

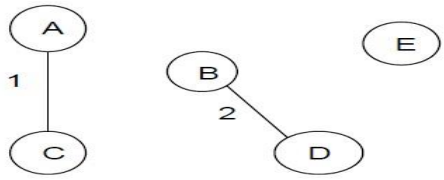
- This method provides better performance when the number of edges is low. Initially each node is considered to be a partial tree. Each partial tree is maintained in a queue Q.
- Priority queue is associated with each partial tree, which contains all the arcs ordered by their weights.
- The algorithm proceeds by removing a partial tree, T1, from the front of Q, finding the minimum weight arc a in T1; deleting from Q, the tree T2, at the other end of arc a; combining T1 and T2 into a single new tree T3 and at the same time combining priority queues of T1 and T2 and adding T3 at the rear of priority queue.
    - This continues until Q contains a single tree, the minimum spanning tree.

Q	Priority queue
{A}	1, 2
{B}	2, 2, 3, 3
{C}	1, 3, 4
{D}	2, 4, 5
{E}	3, 5

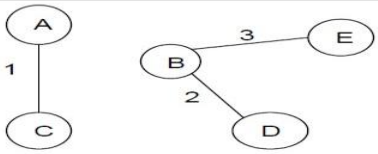




Q	Priority queue
{B}	2, 2, 3, 3
{D}	2, 4, 5
{E}	3, 5
{A, C}	2, 3, 4

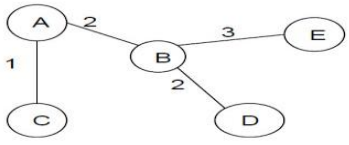


Q	Priority queue
{E}	3, 5
{A, C}	2, 3, 4
{B, D}	2, 3, 3, 4, 5



Q	Priority queue
{A, C}	2, 3, 4
{E, B, D}	2, 3, 4, 5, 5

Q	Priority queue
{A, C, E, B, D}	3, 3, 4, 4, 5, 5



Compiled by: Er. Sandesh S Poudel

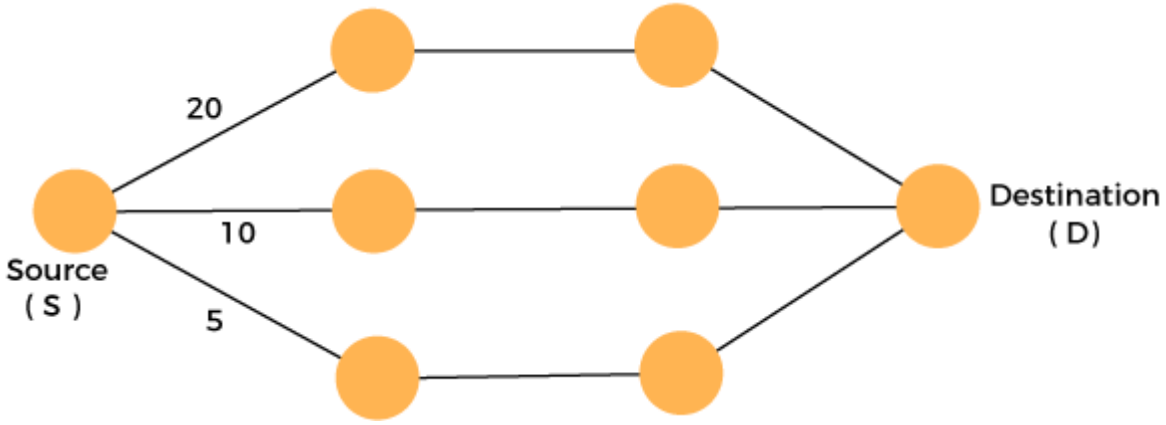
- Only 1 partial tree is left in the queue, which is the required minimum spanning tree.

Shortest path algorithm

Greedy Algorithm

- A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem.
- Greedy algorithms are quite successful in some problems, such as Huffman encoding which is used to compress data, or Dijkstra's algorithm, which is used to find the shortest path through a graph.

Consider the graph which is given below:



We have to travel from the source to the destination at the minimum cost. Since we have three feasible solutions having cost paths as 10, 20, and 5. 5 is the minimum cost path so it is the optimal solution. This is the local optimum, and in this way, we find the local optimum at each stage in order to calculate the global optimal solution.

Dijkstra's shortest path algorithm

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

Algorithm

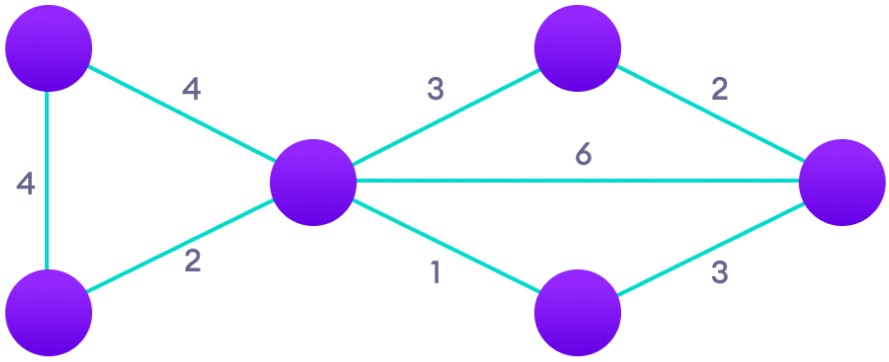
1. Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3. While sptSet doesn't include all vertices

a. Pick a vertex u which is not there in sptSet and has minimum distance value.
b. Include u to sptSet.
c. Update distance value of all adjacent vertices of u.

To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of 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.

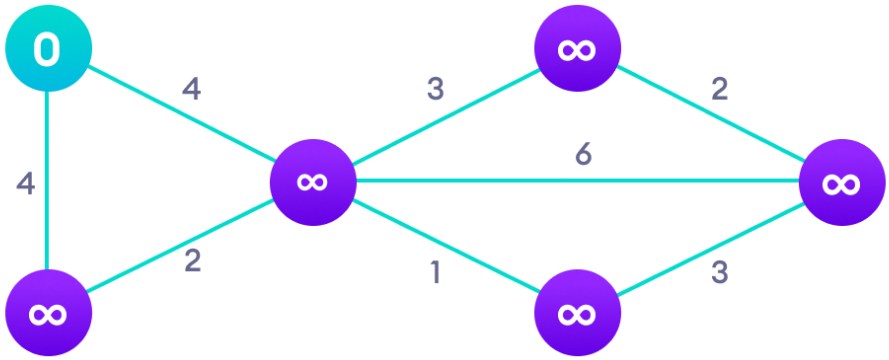
Example of Dijkstra's algorithm

It is easier to start with an example and then think about the algorithm.



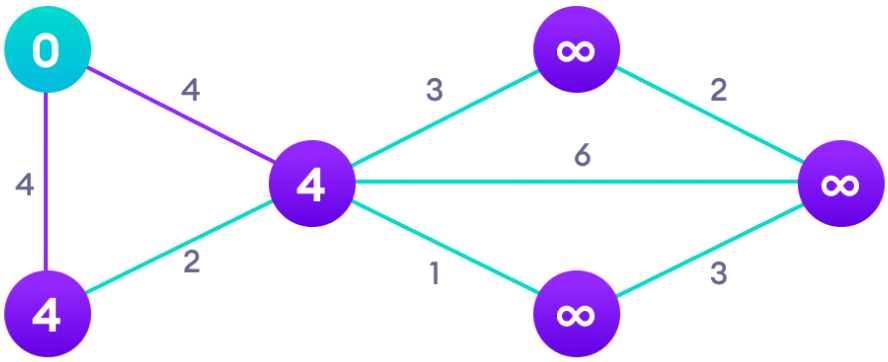
Step: 1

Start with a weighted graph



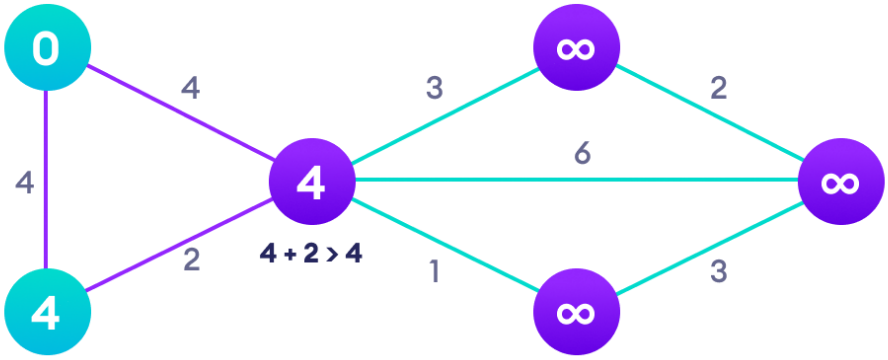
Step: 2

Choose a starting vertex and assign infinity path values to all other devices



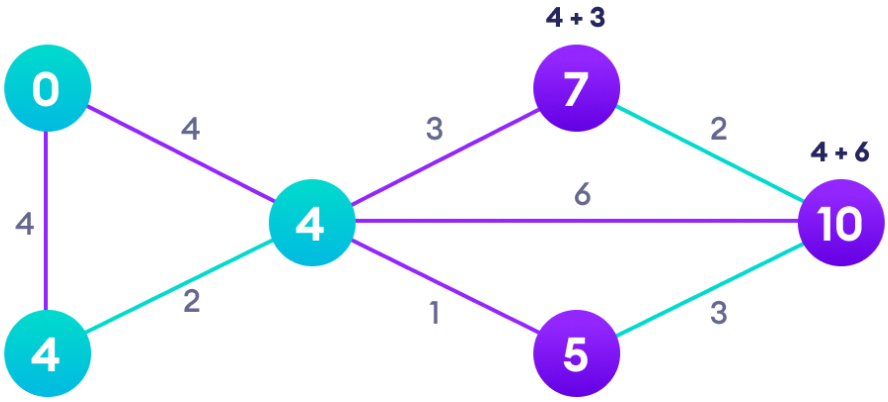
Step: 3

Go to each vertex and update its path length



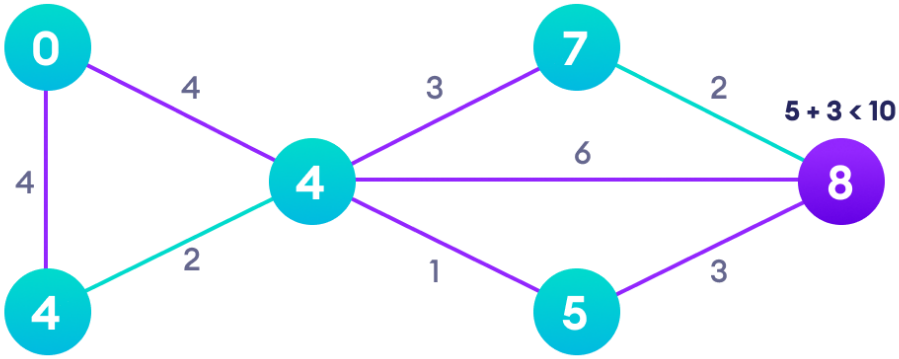
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



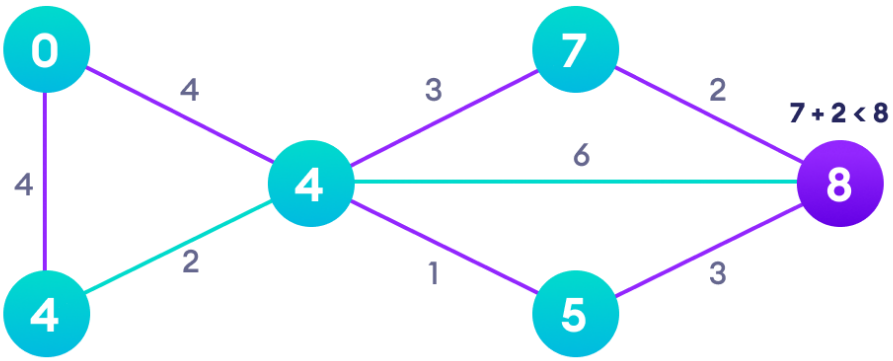
Step: 5

Avoid updating path lengths of already visited vertices



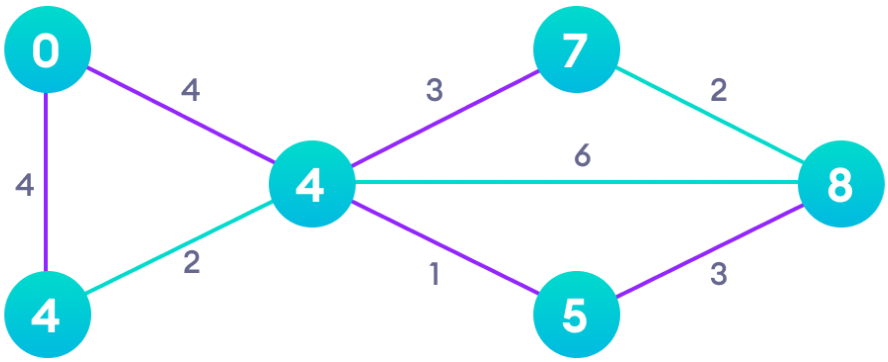
Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



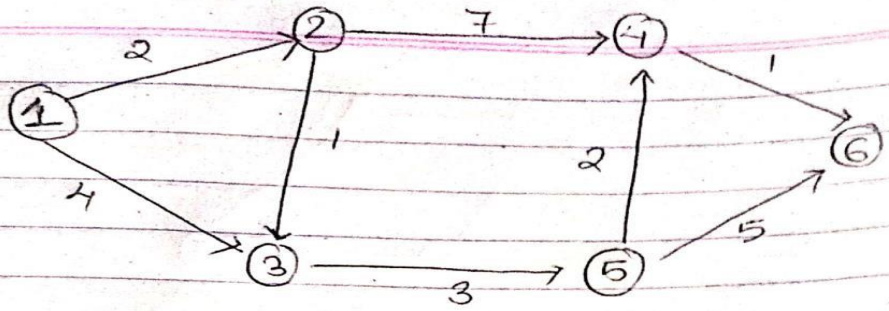
Step: 7

Notice how the rightmost vertex has its path length updated twice

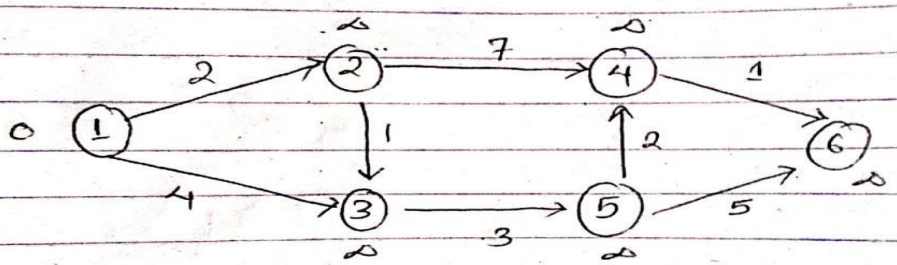


Step: 8

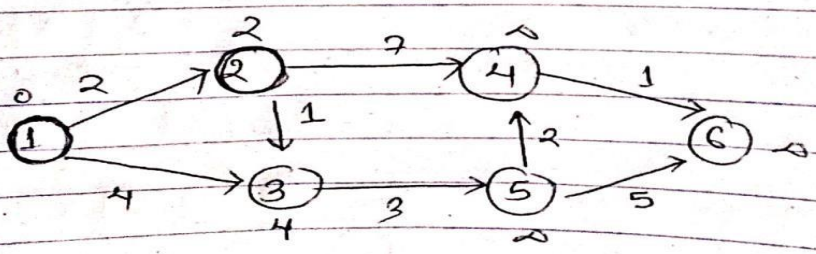
Repeat until all the vertices have been visited



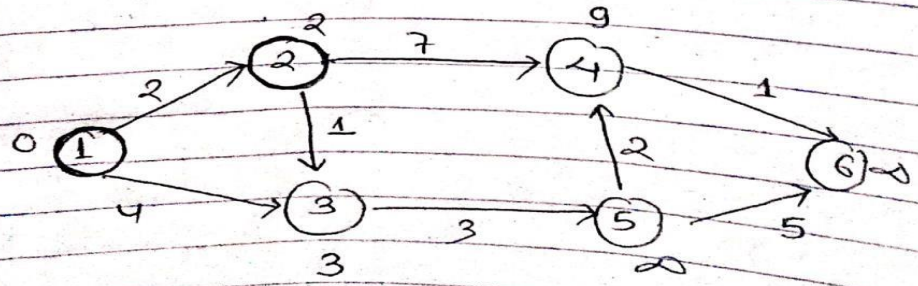
Step (1) let Source vertex is 1



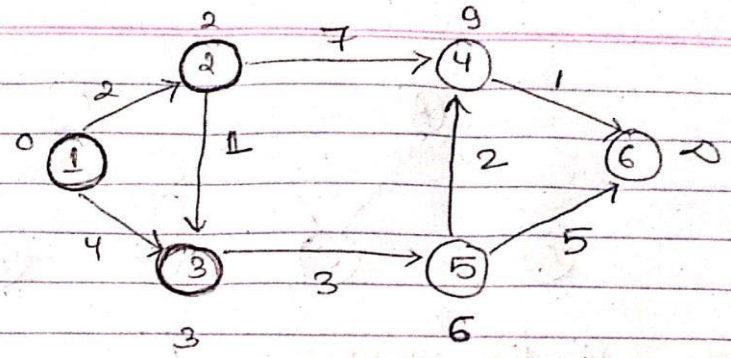
Step (2)



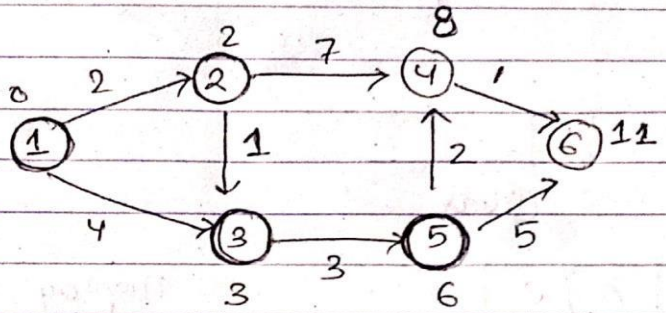
Step (3)



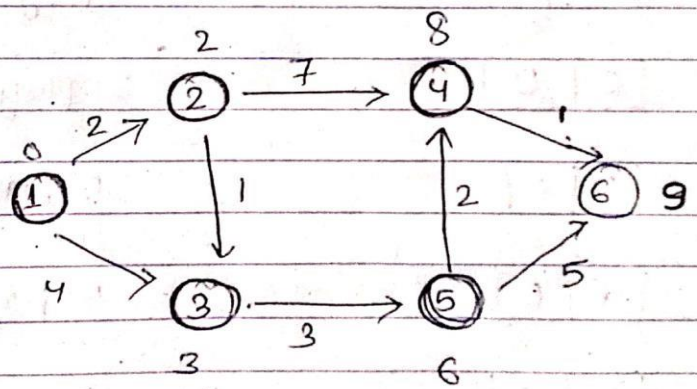
Step (4)



Step (5)



Step (6)



Step (7)