Graph

A graph is a data structure that describes a binary relation between elements and has a webby looking graphical representation. Graph plays a significant role in solving a rich class of problems.

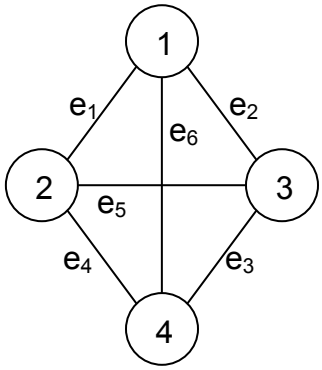
Definition:

A graph G = (V, E) consists of a finite set of non-empty set of vertices V and set of edges E.

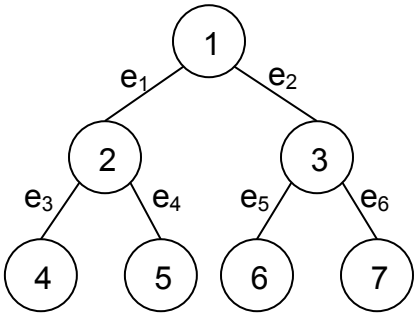
$V = \{ v_1, v_2, \ldots, v_n\}$
$E = \{e_1, e_2, \ldots e_n\}$

Each edge e is a pair (v, w) where v, w ϵ V. The edge is also called arc.

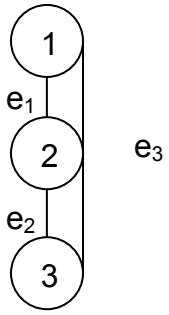
Eg, of graphs:



G_1



G_2



G_3

$V(G_1) = \{1, 2, 3, 4\}$
$E(G_1) = \{e_1, e_2, e_3, e_4, e_5, e_6\}$
$e_1 = (1, 2)$
$e_2 = (1, 3)$
$e_3 = (3, 4)$
$e_4 = (2, 4)$
$e_5 = (2, 3)$
$e_6 = (1, 4)$

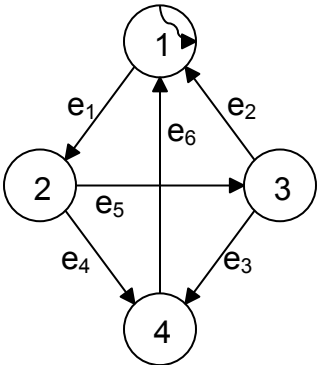
$V(G_3) = \{1, 2, 3\}$
$E(G_3) = \{e_1, e_2, e_3\}$
$e_1 = (1, 2)$
$e_2 = (2, 3)$
$e_3 = (1, 3)$

Now, specify the vertices and edges for G_2.

The vertices are represented by points or circles and the edges are line segments connecting the vertices. If the graph is directed, then the line segments have arrow heads indicating the direction.

Directed Graphs:

If every edge (i, j), in E(G) of a graph G is marked by a direction from i to j, then the graph is called directed graph. It is often called diagraph.



G_4

In edge e = (i, j), we say, e leaves i and enters j. In diagraphs, self loops are allowed. The indegree of a vertex v is the number of edge entering v. The outdegree of a vertex v is the number of edges leaving v. The sum of the indegrees of all the vertices in a graph equals to the sum of outdegree of all the vertices.

$$\sum_{i=0}^{n} d_{in}(v_i) = \sum_{i=0}^{n} d_{out}(v_i) \quad \text{n = no. of vertices}$$

Undirected Graph:

If the directions are not marked for any edge, the graph is called undirected graph. The graphs G_1, G_2, G_3 are undirected graphs. In an undirected graph, we say that an edge e = (u, v) is incident on u and v (u and v are connected).

Undirected graph don't have self loops. Incidence is a symmetric relation i.e. if e = (u, v)

Then u is a neighbor of v and vice versa. The degree of a vertex, d(v), is the total number of edges incident on it.

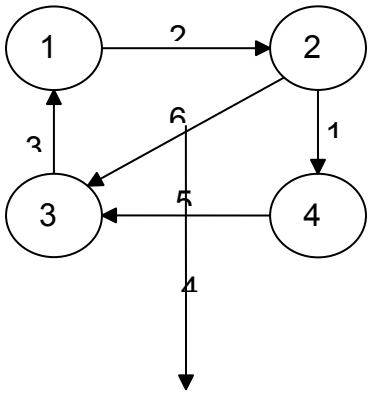
$$\sum_{i=0}^{n} d(v_i) = 2|E|$$

The sum of the degrees of all the vertices in a graph equals twice the number of edges if d(v) = 0, v is isolated. If d(v) = 1, v is pendent.

Weighted Graphs:

A graph is said to be weighted graph if every edge in the graph is assigned some weight or value. The weight is a positive value that may represent the cost of moving along the edge, distance between the vertices etc.

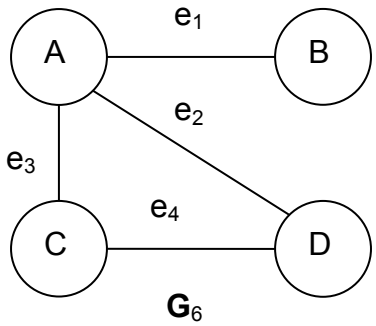
The two vertices with no edge (path) between them can be thought of having an edge (path) with weight infinite.



Additional Terminology in Graph

Adjacent and Incident

Two vertices i and j are called adjacent if there is an edge between the two. Eg. The vertices adjacent to vertex A, in the fig below are B, C, D. The adjacent vertices of C are A and D.



G6

If e(i, j) is an edge on E(G), then we say that the edge e(i, j) is incident on vertices i and j. Eg. in the above figure, e_4 is incident on C and D. If (i, j) is directed edge, then i is adjacent to j and j is adjacent from i.

Path:

A path is a sequence or distinct vertices, each adjacent to the next. For e.g. the sequence of vertices e_1, e_3, e_4 (i.e (B,A), (A,C), (C,D)) of the above graph form a path from B to D. The length of the path is the number of edges in the path. So, the path from B to D has length equal to three.

In a weighted graph, the cost of a path is the sum of the costs of its edges. Loops have path length of 1.

Cycle:

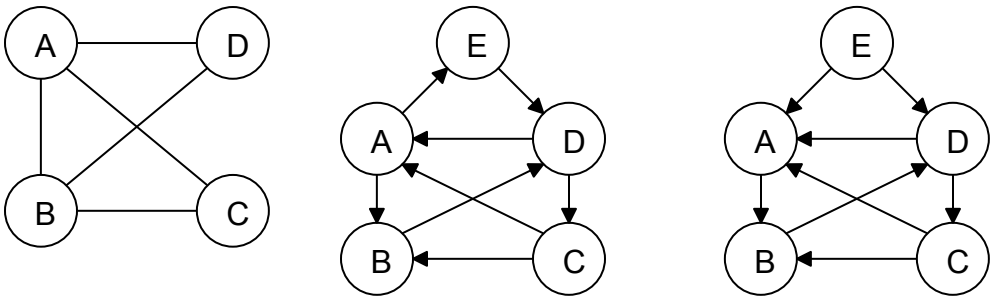
A cycle is a path containing at least three vertices such that the last vertex on the path is adjacent to the first. In the above graph G_6, A, C, D, A is a cycle.

Connected:

Any graph is *connected* provided there exists a path (directed or undirected) between any two nodes. A digraph is said to be *strongly connected* if, for any two vertices there is a direct path from i to j.

A digraph is said to be weakly connected if, for any two vertices i and j, there is a direct path from i to j or j to i. or A *weakly-connected graph* is a directed graph for which its underlying undirected graph is connected.

Eg.



Connecte d

Strongly Connected

Weakly Connected

Complete graph

A *complete graph* is a graph in which there is an edge between every pair of vertices.

Trees and Graph

A tree is a connected graph with no cycle. A tree has **|E| = |V| - 1** edges. Since, it is connected, there is a path between any two vertices.

Representation of Graphs

A graph can be represented in many ways. Some of them are described in this section.

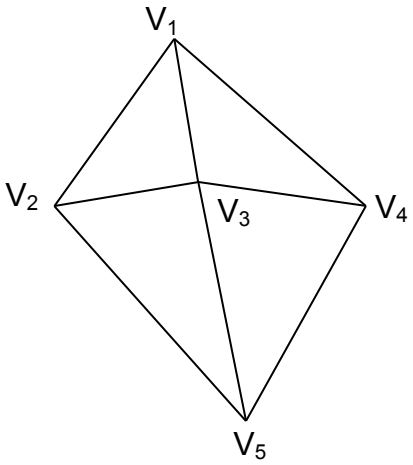
Adjancency matrix

Adjancency matrix A for a graph G = (V, E) with n vertices, is n x n matrix, such that
$A_{ij} = 1$, if there is an edge from v_i to v_j
$A_{ij} = 0$, if there is no such edge

We can also write,

$$A(i, j) = \begin{cases} 1 \text{ if and only if } (v_i, v_j) \text{ exists} \\ 0 \text{ otherwise} \end{cases}$$

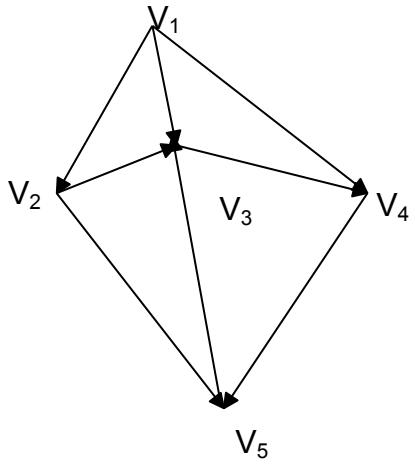
Eg; An adjacency matrix for the following undirected graph is



A =

	V_1	**V_2**	**V_3**	**V_4**	**V_5**
V_1	0	1	1	1	0
V_2	1	0	1	0	1
V3	1	1	0	1	1
V_4	1	0	1	0	1
V_5	0	1	1	1	0

If the graph is directed, then the adjacency matrix will be as follows. The number in each row tells the outdegree of that vertex.



A =

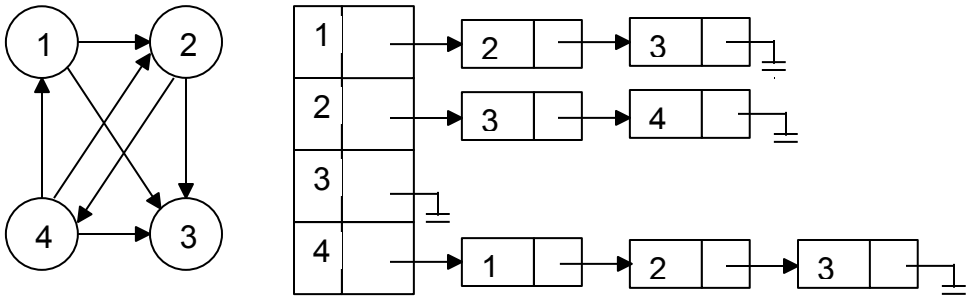
	V_1	V_2	V_3	V_4	V_5
V_1	0	1	1	1	0
V_2	0	0	1	0	1
V3	0	0	0	1	0
V_4	0	0	0	0	1
V_5	0	0	1	0	0

In this representation we require n^2 bits to represent a graph with n nodes. It is a simple way to represent a graph, but it has following disadvantages:

- it takes $O(n^2)$ space
- it takes $O(n^2)$ time to solve most of the problems.

Adjacency List Representation

The n rows of an adjacency matrix can be represented as n linked lists. This is one list for each node in a graph. Each list will contain adjacent nodes. Each node has two fields, *Vertex* and *Link*. The Vertex field of a node p will contain the nodes that are adjacent to the node p.



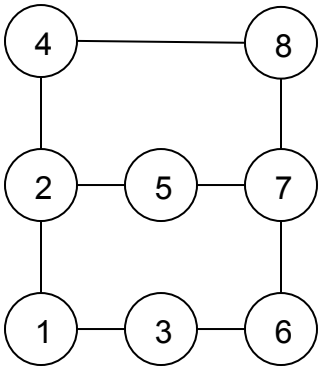
Graph Traversals

A graph traversal means visiting all the nodes of the graph. Basically, there are two methods of graph traversals.

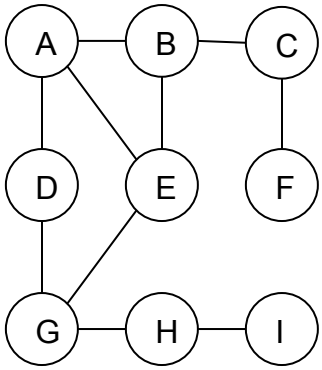
1. Breadth First Traversal
2. Depth First Traversal

1. **Breadth First Traversal**

   The *Breadth First Traversal* begins with a given vertex and then it next visits all the vertices adjacent to v, putting the vertices adjacent to these in a waiting list to be traversed after all vertices adjacent to v have visited. Breadth First Traversal uses queue.



Starting from 1 Breadth First Traversal yields 1, 2, 3, 4, 5, 6, 8, 7



Starting from A Breadth First Traversal yields A, B, D, E, C, G, F,

Algorithm:

```
void BFTraversal(Graph G)
{
  for each vertex v in G
      visited[v] = false;
  for each vertex v in G
      if(visited[v]==false)
       {
            enqueue(V, Q);
            do
            {
                 v=dequeue(q);
                 visited[v]=true;
                 visited(v);
                 for each vertex w adjacent to v
                 if(visited[w]==false)
                 enqueue(w, q);
            }while(isempty(Q)==false)
       }
}
```

2. **Depth First Traversal**

   The *Depth First Traversal* algorithm is roughly analogous to preorder tree traversal, the algorithm follows:

   suppose that the traversal has just visited a vertex v, and let w_0, w_1, …w_k be the vertices adjacent to v

   - next, visit w_0 and keep w_1, …w_k waiting
   - after visiting w_0, we traverse all the vertices to which it is adjacent before returning to traverse w_1, …w_k.

Depth first traversal uses stack to store the adjacent vertices but one.

For above 2 figures Depth First Traversal yields

	Adjacency list		Adjacency list
1	2, 3	A	B, D, E
2	1, 4, 5	B	A, C, E
3	1, 6	C	B, F
4	2, 8	D	A, G
5	2, 7	E	A, B, G
6	3, 7	F	C
7	5, 6, 8	G	D, E, H
8	4, 7	H	G, I
		I	H
DF Traversal:		**DF Traversal:**	
	1, 2, 4, 8, 7, 5, 6, 3		**A, B, C, F, E, G, D, H, I**

Algorithm:

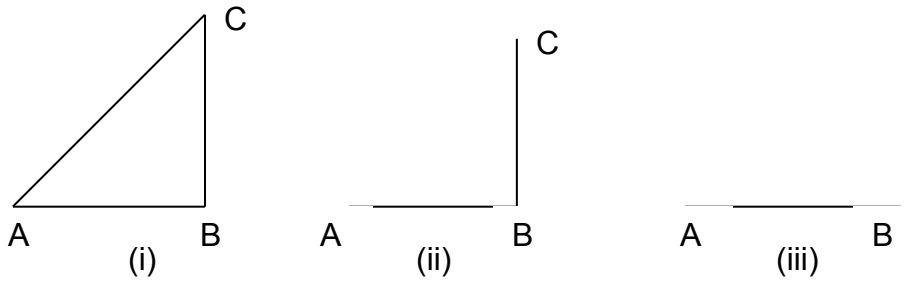
```
void DFTraversal(Graph G)
{
    for each vertex v in G
      Visited[v]=false;

    for each vertex v in G
      If(visited[v]==false)
      Traverse(v);
}
void Traverse(Vertex v)
{
    Visited[v]=true;
    Visited(v);
    For each vertex w adjacent to v
        {
    If(visited[w]==false)
    Traverse(w);
        }
}
```

Subgraph:

Let H be a graph with vertex set v(H) and edge set E(H) and, similarly, let G be a graph with vertex set V(G) and edge set E(G). Then H is said

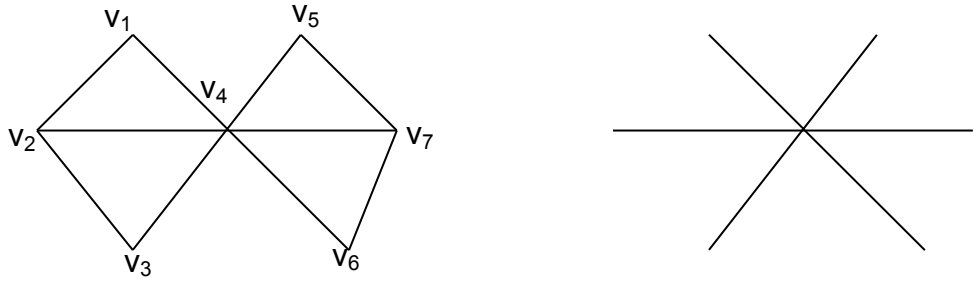
to be a subgraph of G, written as H ⩽ G if V(H) ⩽ V(G) and E(H) ⩽ E(G).



Here, second and third graph are subgraph of the first.

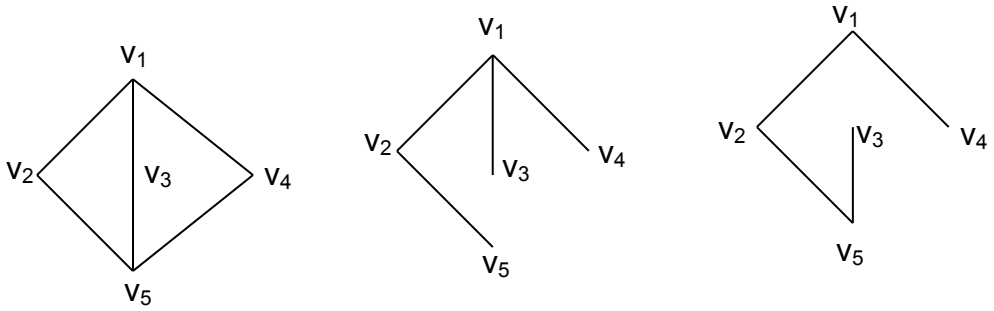
Spanning Tree:
Let a graph G=(V,E) be a graph. If T is a subgraph of G and contains all the vertices but no cycles/circuits, then 'T' is said to be spanning tree of G.

In other words, the spanning tree of an undirected graph G is the free tree formed from graph edges which connects all the vertices of G.



Undirected Graph G □□□□Spanning Tree of G

For a given graph G=(V, E), if Breadth-First search and Depth-First search call is made, then the resultant tree that is generated is the spanning tree of the graph G. Thus, by making a Depth-First search on G, we get the Depth-First spanning tree. Similarly, on applying Breadth-First search on graph 'G' we get Breadth First spanning tree.



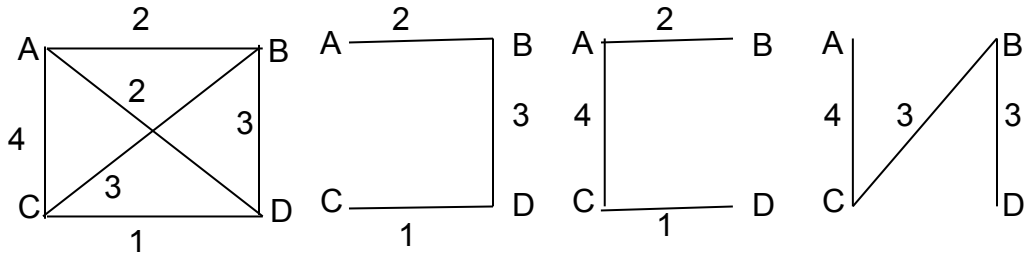
Graph G □□BS Spanning Tree and DF Spanning Tree of G

Minimum Spanning Tree

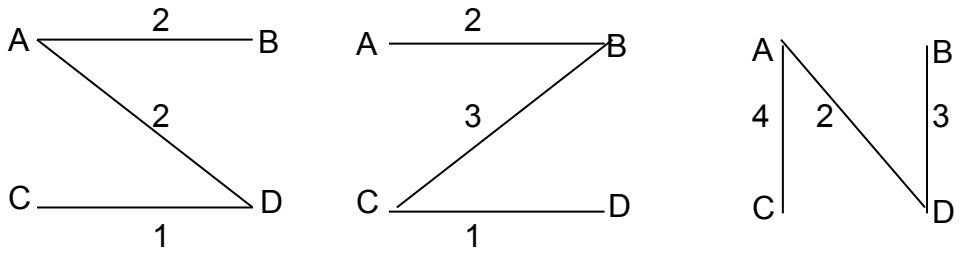
The cost of a spanning tree of a weighted, undirected graph is the sum of the costs (weights) of the edges in tree.

The minimum spanning tree of a weighted undirected graph is the spanning tree that connects all the vertices in G at lowest cost. So, if T is the minimum spanning tree of graph G and T' is any other spanning tree of G then,

$$W(T) \leqslant W(T')$$



G□□□T1□□ **T2□□** **T3□□**



T4□□ **T5□□** **T6□□**

Here, $\mathbf{T_1}$ to $\mathbf{T_6}$ are the *spanning trees* of **G**. Among them, **T4** has minimum cost (i. e. 5). So, $\mathbf{T_4}$ is the minimum spanning tree of graph G

There are several algorithms available to determine the *minimal spanning tree* of a given weighted graph.

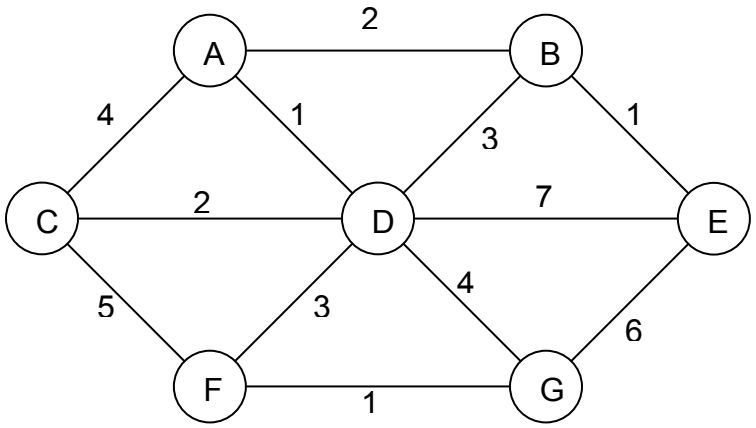
Kruskal's Algorithm

To determine minimum spanning tree consider a graph G with n vertices.

Step 1: List all the edges of the graph G with increasing weights

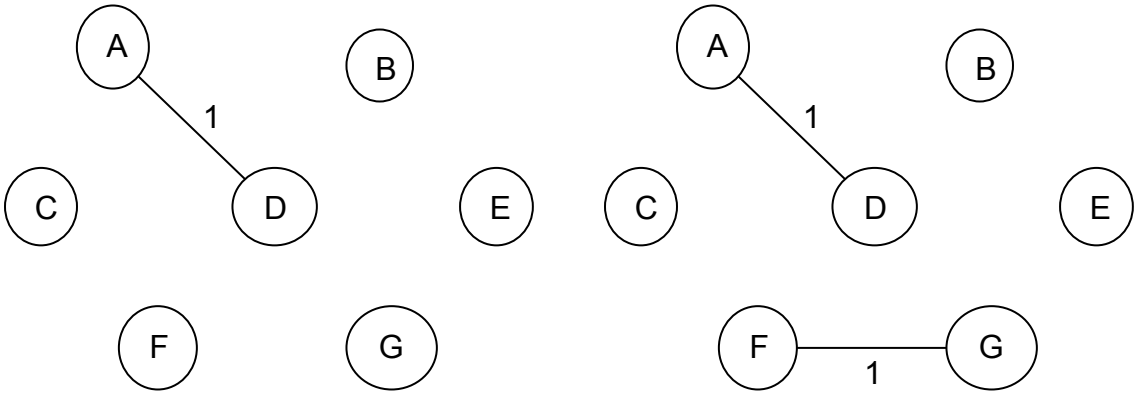
Step 2: Proceed sequentially to select one edge at a time joining n vertices of G such that no cycle is formed until n-1 edges are selected.

Step 3: Draw the n-1 edges that were selected forming a minimal spanning tree T of G.

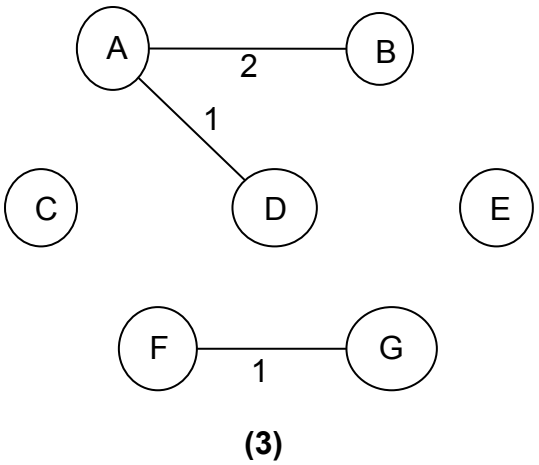


We have,

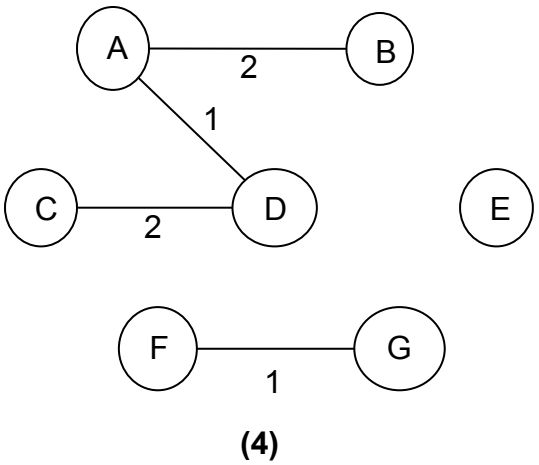
Edges:	AD	FG	AB	CD	BD	AC	DG	CF	EG	DE	DF	BE
Weights	1	1	2	2	3	4	4	5	6	7	8	10



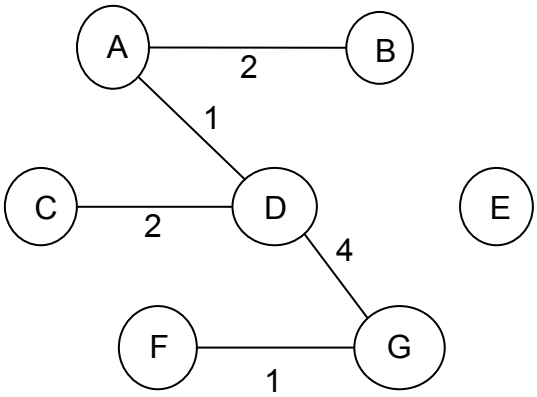
(1) **(2)**



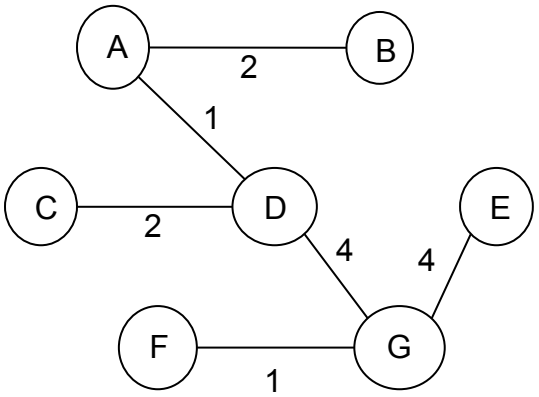
(3)



(4)



(5)



(6)

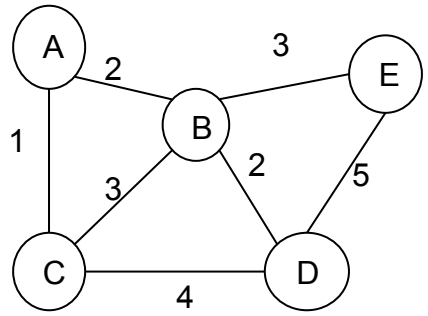
Hence, the minimum spanning tree is generated.

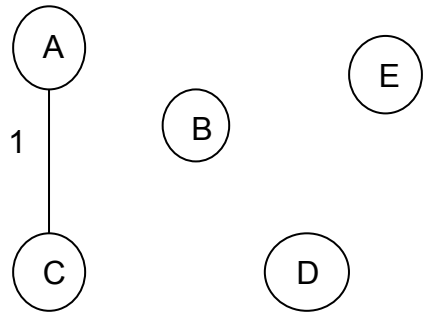
Round Robin Algorithm

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.

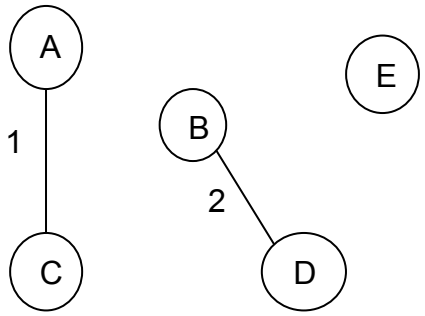
The algorithm proceeds by removing a partial tree, T1, from the front of Q, finding the minimum weight arc a in T1; deleting from Q, the tree T2, at the other end of arc a; combining T1 and T2 into a single new tree T3 and at the same time combining priority queues of T1 and T2 and adding T3 at the rear 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

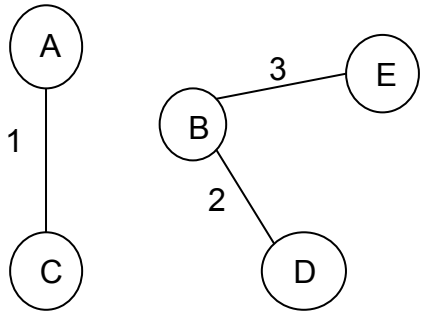




Q	Priority queue
{B}	2, 2, 3, 3
{D}	2, 4, 5
{E}	3, 5
{A, C}	2, 3, 4

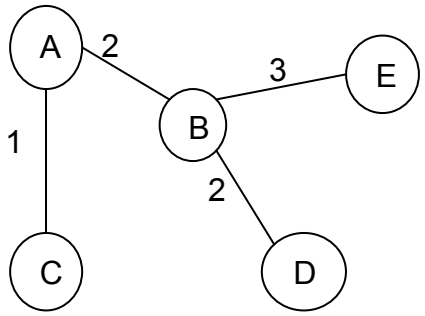


Q	Priority queue
{E}	3, 5
{A, C}	2, 3, 4
{B, D}	2, 3, 3, 4, 5



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



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

Shortest Path Algorithm

There are many problems that can be modeled using graph with weight assigned to their edges. Eg, we may set up the basic graph model by representing cities by vertices and flights by edges. Problems involving distances can be modeled by assigning distances between cities to the edges. Problems involving flight time can be modeled by assigning flight times to edges. Problem involving fares can be modeled by assigning fares to the edges.

Thus, we can model airplane or other mass transit routes by graphs and use shortest path algorithm to compute the best route between two points.

Similarly, if the vertices represent computers; the edges represent a link between computers; and the costs represent communication costs, delay costs, then we can use the shortest-path algorithm to find the cheapest way to send electronic news, data from one computer to a set of other computers.

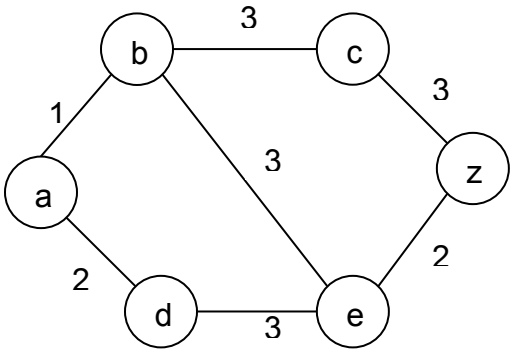
Basically, there are three types of shortest path problems:

Single path: Given two vertices, s and d, find the shortest path from s to d and its length (weights).

Single source: Given a vertex, s find the shortest path to all other vertices.

All pairs: Find the shortest path from all pair of vertices.

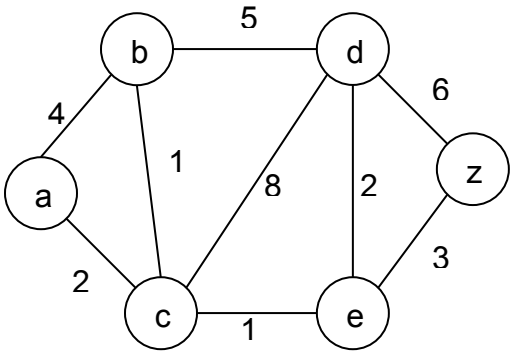
Let source vertex be a. We will find the shortest path to all the other vertices.



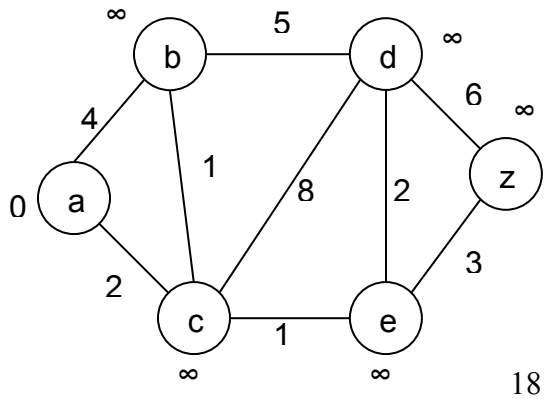
Vertex	Cost	Path
b	1	a, b
c	4	a, b, c
d	2	a, d
e	4	a, b, e
z	6	a, b, e, z

Dijkstra's Algorithm to find the shortest path

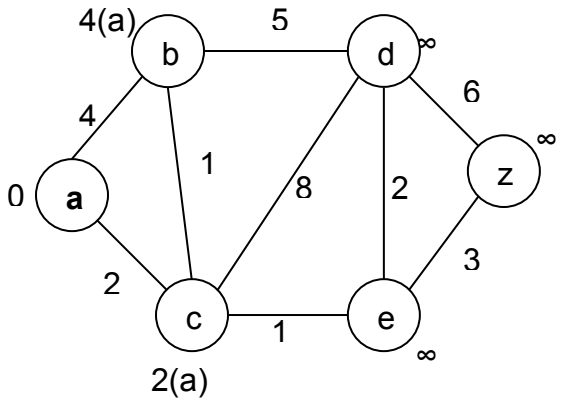
1. Mark all the vertices as unknown
2. for each vertex v keep a distance d_v from source vertex s to v initially set to infinity except for s which is set to $d_s = 0$
3. repeat these steps until all vertices are known
   - i. select a vertex v, which has the smallest d_v among all the unknown vertices
   - ii. mark v as known
   - iii. for each vertex w adjacent to v
     - i. if w is unknown and $d_v + cost(v, w) < d_w$
       update d_w to $d_v + cost(v, w)$



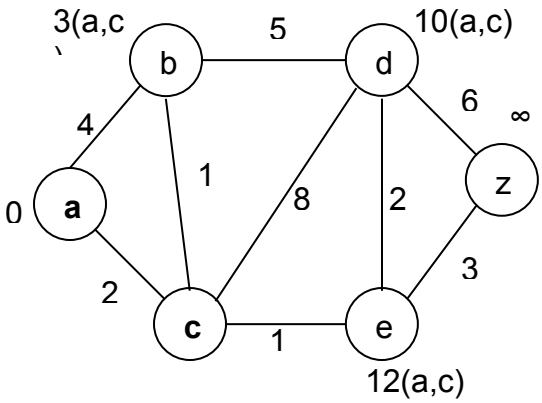
Given graph with weights



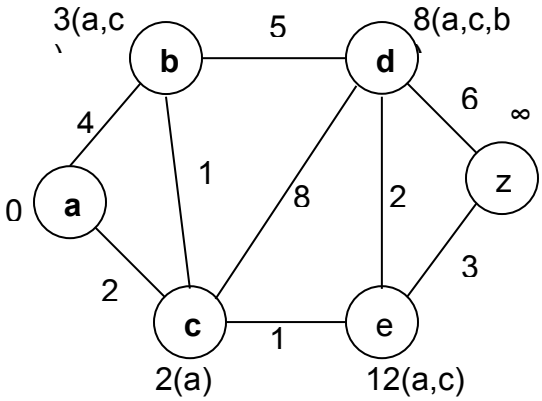
(a)


4(a)
b
5
d
∞
6
4
1
0
a
8
2
z
∞
2
3
c
1
e
∞
2(a)

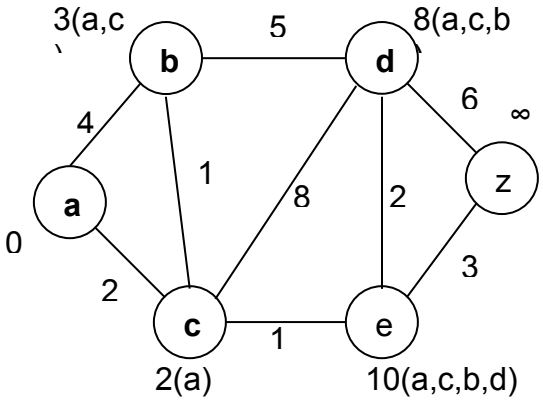
(b)


3(a,c
`
b
5
d
10(a,c)
6
4
∞
1
0
a
8
2
z
2
3
c
1
e
12(a,c)

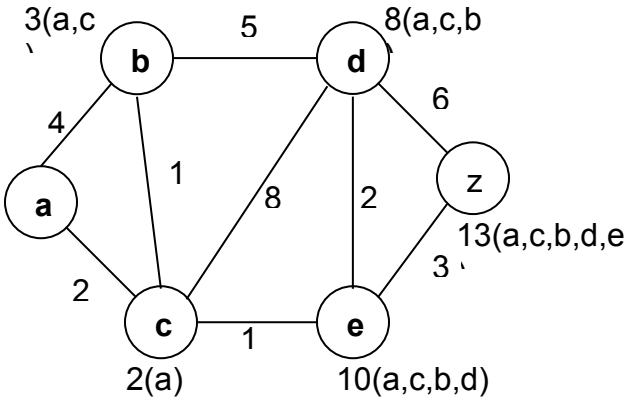
(c)


3(a,c
`
b
5
d
8(a,c,b
6
4
∞
1
0
a
8
2
z
2
3
c
1
e
2(a)
12(a,c)

(d)


3(a,c
`
b
5
d
8(a,c,b
6
4
∞
1
0
a
8
2
z
2
3
c
1
e
2(a)
10(a,c,b,d)

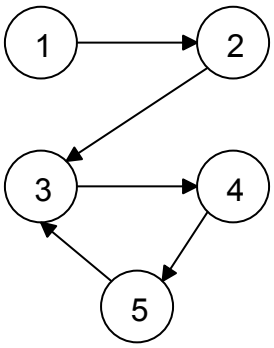
(e)


3(a,c
`
b
5
d
8(a,c,b
6
4
1
a
8
2
z
13(a,c,b,d,e
3
`
2
c
1
e
2(a)
10(a,c,b,d)

(f)

Greedy Algorithm
One of the simplest approaches that often leads to a solution of an optimization problem. This approach selects the best choice at each step, instead of considering all sequences of steps that may lead to an optimal solution. Algorithm that make what seems to be the "best" choice at each step are called greedy algorithm.

Transitive Closure
In all shortest path, we noticed that every time we have to determine whether there is a path from vertex i to j. This can be accomplished by observing a matrix, say $d^{(n)}$. The path exists if and only if the entry does not have the value infinity. In matrix $d^{(n)}$, we consider only path (not the weight). Thus, (d^k) can be regarded as Boolean matrix.
If there is a path from i to j, then (i, j)th entry in d^k is 1 (which means true) otherwise the value is 0. If 'A' is the adjacency matrix of graph 'G', then the transitive closure is defined to be a Boolean matrix with the above property.

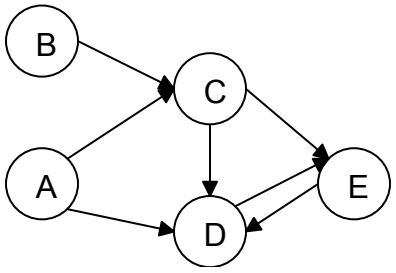


Directed

$$A^+ = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

A^+ Matrix has the following property:
1 if there is a path from i to j
0 otherwise
The matrix A^+ is called the transitive closure. The Transitive closure helps to determine which pair or vertices in a digraph are connected by a path. It takes $O(n^4)$ time complexity.

Let's consider the following directed graph



Now, let adj2 is a matrix that stores information about the existence of path of length 2. The adj2 matrix (path matrix of length 2) can be represented as:

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

Similarly, adj3 can be represented as:

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

And adj 4 as:

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

To calculate the path matrix, we just multiply the matrix itself. For eg, to calculate path matrix 3, the matrix of adj1 is multiplied 3 times itself. While multiplying, the addition operatin is replaced by '||' operation and multiplication is replaced by '&&' operation.

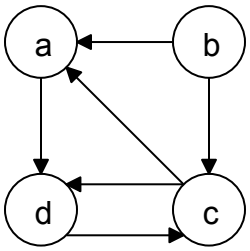
To find at least one path between two nodes, we can check adj1, adj2, adj3, …,$adj_{maxnodes-1}$. If any of the adjacency matrix give true value for node 'i' and 'j', then there exist at least a path between nodes 'i' and 'j'. To store the adjacency values for path, we use a path matrix as:

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

The length of path can be 1, 2, 3 or 4… and there is no termination of the order of the length. Let 'm' is the highest order of length and 'n' is the no. of total nodes. If the graph supports m>n, then at least one of the node is visited twice or more times in the path. It means, we can discard the cycling path from the node to the same node. The removal process is repeated until no node is repeated in the path. This ensures that the paths that exist in the graph be less than or equal to n. The final matrix path in which no path exists that has length greater than 'n' is known as transitive closure of the matrix adj.

Warshall's Algorithm

Transitive closure method of finding path between nodes is quite inefficient because it processes the adjacency matrix lots of times. An Efficient method for calculating transitive closure is by Warshall's algorithm whose time complexity is only $O(n^3)$.



We first find the matrices d_0, d_1, d_2, d_3, d_4, and d_4 is the transitive closure.

$$d_0 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad d_1 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1^* \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

d_1 has 1 as its $(i,j)^{th}$ entry if there is a path from vi to vj that has v1 = a as an interior vertex.

$$d_2 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad d_3 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1^* & 0 & 1 & 1^* \end{bmatrix}$$

d2 has 1 as its $(i,j)^{th}$ entry if there is a path from v_i to v_j that has v_1 and v_2 as interior vertex. d_3 has 1 as its $(i,j)^{th}$ entry if there is a path fro v_i to v_j that has v_1 v_2 and v_3 as interior vertex.

$$d_4 = \begin{bmatrix} 1^* & 0 & 1^* & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1^* & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

Finally, d_4 has 1 as its $(i,j)^{th}$ entry if there is a path from v_i to v_j that has v_1, v_2, v_3 and v_4 as interior vertex.

The matrix d_4 is the transitive closure.

Warshall's Algorithm
procedure warshall (M_R : n x n zero one matrix)
{
D = M_R
for k = 1 to n
begin
for i = 1 to n
begin
for j = 1 to n
$d_{ij} = d_{ij} \vee (d_{ik} \wedge d_{kj})$
end
}
D = $[d_{ij}]$ is the transitive closure.

Topological Sorting

A topological sort is an ordering of vertices in a directed acyclic (graph with no cycle) graph (dag), such that if there is a path from v_i to v_j, then vi appears before v_j in the ordering. A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.

Applications:
Consider the course available at university as the vertices of a directed graph; where there is an edge from one course to another if the first is a prerequisite for the second.

A topological ordering is then a listing of all the courses such that all prerequisite for a course appear before it does.

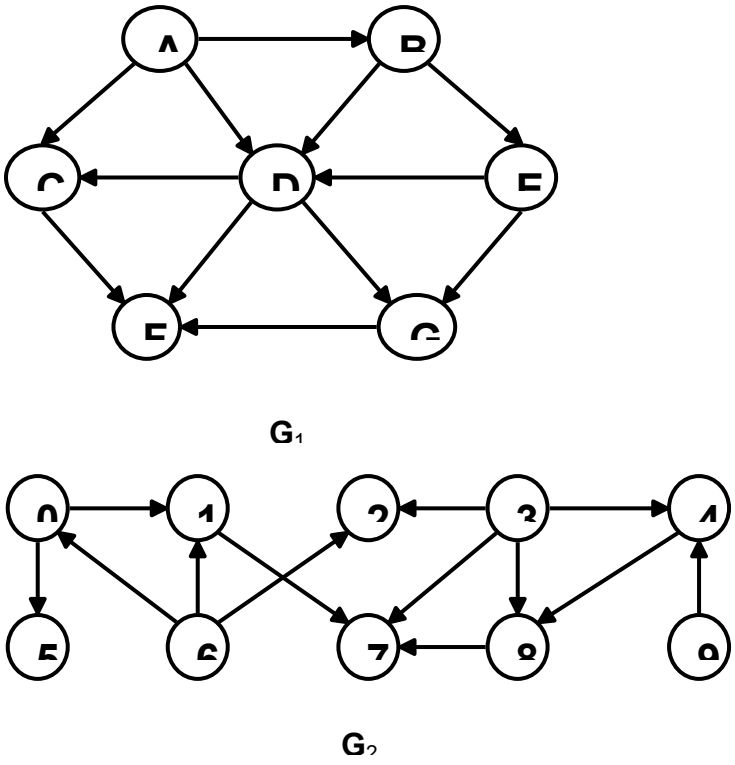


Fig: Directed Graph with no Directed

Algorithms to find out topological ordering

1. Breadth First Ordering
2. Depth First Ordering

1. Breadth First Ordering

In Breadth First Topological ordering of a directed graph with no cycles, we start by finding the vertices that should be first in the topological order and then apply the fact that every vertex must come before its successors in the topological order. The vertices that come first are that are not successors of any other vertex.

Determining the Breadth First Topological ordering:

1. Fill the information from the graph into the table(Vertices, Predecessors and Predecessor count)
2. Visit the vertex that have *Predecessor count* zero (0).
3. Decrement the *Predecessor count* value by 1 where ever the visited vertex appears in the *Adjacent Predecessor vertices* column.
4. Repeat steps 2 and 3 until all the vertices has been visited.

From the graph G_1 we get the following information:

Vertex	Adjacent predecessor vertices	Predecessor count
A	-	~~0~~
B	A	~~1~~ 0
C	A, D	~~2~~ ~~1~~ 0
D	A, B, E	~~3~~ ~~2~~ ~~1~~ 0
E	B	~~1~~ 0
F	C, D, G	~~3~~ ~~2~~ ~~1~~ 0
G	D, E	~~2~~ ~~1~~ 0

The breadth first topological ordering gives the following

A	B	E	D	C	G	F

From the graph G_2 we get following information:

Vertex	Adjacent predecessor vertices	Predecessor count
0	6	~~1~~ 0
1	0, 6	~~2~~ ~~1~~ 0
2	3,6	~~2~~ ~~1~~ 0
3	-	0
4	3, 9	~~2~~ ~~1~~ 0
5	0	~~1~~ 0
6	-	0
7	1, 3, 8	~~3~~ ~~2~~ ~~1~~ 0
8	3, 4	~~2~~ ~~1~~ 0
9	-	0

The breadth first topological ordering gives the following:

3	6	9	0	2	4	1	5	8	7

2. Depth First Ordering:
Start by finding a vertex that has no successors and place it last in the order.
After, we have, by recursion placed all the successors of a vertex into the topological order, then place the vertex itself in the order (before any of its successors).

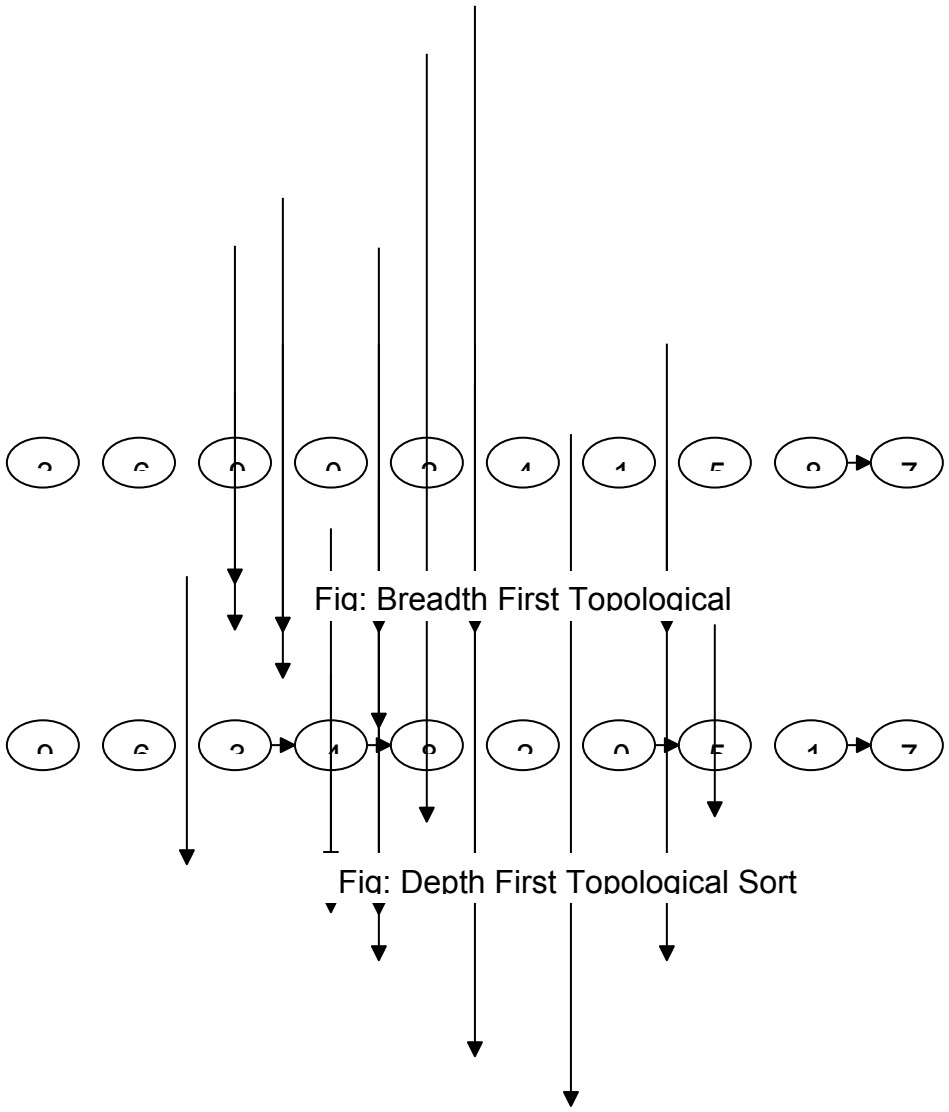


Fig: Breadth First Topological

Fig: Depth First Topological Sort

Determining the Depth First Topological ordering:

1. Fill the information from the graph into the table(Vertices, Successors and Successor count)
2. Push all the vertices into stack and visit the vertex that is on the top.
3. Decrement the *Successor count* value by 1 where ever the visited vertex appears in the *Adjacent Successor vertices* column.
4. Repeat steps 2 and 3 until all the vertices has been visited.

From the graph G_1 we get the following information:

Vertex	**Adjacent successor vertices**	**Successor count**
A	B, C, D	~~3~~ ~~2~~ ~~1~~ 0
B	D, E	~~2~~ ~~1~~ 0
C	F	~~1~~ 0
D	C,F, G	~~3~~ ~~2~~ ~~1~~ 0
E	D, G	~~2~~ ~~1~~ 0
F	-	0
G	F	~~1~~ 0

The depth first topological ordering gives the following

A	B	E	D	C	G	**F**

From the graph G_2 we get the following information:

Vertex	**Adjacent successor vertices**	**Predecessor count**
0	1, 5	~~2~~ ~~1~~ 0
1	7	~~1~~ 0
2	-	0
3	2, 4, 7, 8	~~4~~ ~~3~~ ~~2~~ ~~1~~ 0
4	8	~~1~~ 0
5	-	0
6	0, 1, 2	~~3~~ ~~2~~ ~~1~~ 0
7	-	~~0~~
8	7	~~1~~ 0
9	4	~~1~~ 0

The depth first topological ordering gives the following:

9	6	3	4	8	2	0	5	1	**7**