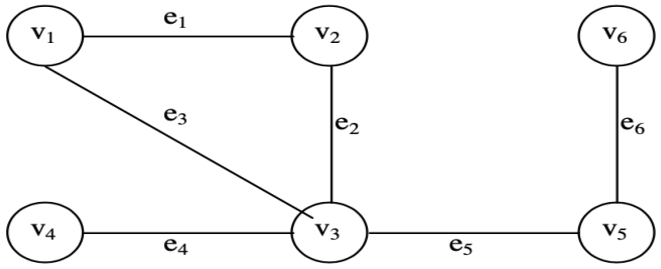
Unit 7: Graphs

- A simple graph, denoted by G = (V, E), is composed of a finite set of vertices and a finite set of edges.
- The elements of V are the vertices and those of E are the edges. The vertex set of V are denoted by V_G and edge set is denoted by E_G. Thus G = (V_G, E_G).
- The number of vertices is denoted by |V| and number of edges is denoted by |E|.

Example of a Graph



The set of vertices V and set of edges E in the above graph are as below:

- V = {*v*1, *v*2, *v*3, *v*4, *v*5, *v*6}
- E = {*e*1, *e*2, *e*3, *e*4, *e*5, *e*6} or E = {(*v*1, *v*2) (*v*2, *v*3) (*v*1, *v*3) (*v*3, *v*4), (*v*3, *v*5) (*v*5, *v*6)}.

Basic Terminologies in a Graph:

1. Edge:

- An edge (*a*, *b*), in said to the incident with the vertices it joints, *i.e., a*, *b*. We can also say that the edge (*a*, *b*) is incident from *a* to *b*. The vertex a is called the *initial vertex* and the vertex *b* is called the *terminal(or final) vertex* of the edge (*a*, *b*).

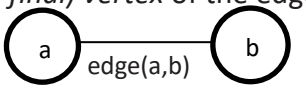


Figure 1

2. Loop: If an edge that is incident from and into the same vertex, is called a *loop*.

Figure 2

3. Adjacent Vertices:

- Two vertices are said to be adjacent if they are joined by an edge.
- Consider edge (*a*, *b*), the vertex *a* is said to be adjacent to the vertex *b*, and the vertex *b* is said to be adjacent from the vertex *a*.

4. Isolated Vertex:

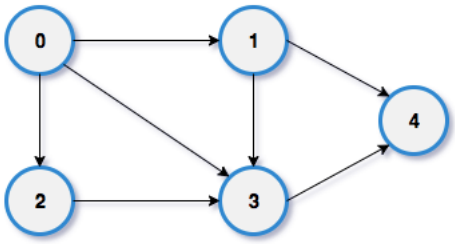
- A vertex is said to be an *isolated vertex* if there is no edge incident with it. In Figure 2 vertex a is an isolated vertex.

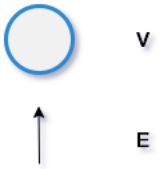
5. Degree of a vertex:

- The number of edges incident on a vertex is its *degree*.
- The degree of vertex *a*, is written as deg(*a*).
- If the degree of vertex a is zero, then vertex *a* is called isolated vertex.
- If the degree of vertex a is one, then vertex *a* is called pendant vertex.

Indegree and Outdegree of a vertex:

- Indegree of a vertex is the number of edges coming to this vertex.
- Outdegree of a vertex is the number of edges which are coming out from this vertex.





In-degree of vertex 0 = 0	Out-degree of vertex 0 = 3
In-degree of vertex 1 = 1	Out-degree of vertex 1 = 2
In-degree of vertex 2 = 1	Out-degree of vertex 2 = 1
In-degree of vertex 3 = 3	Out-degree of vertex 3 = 1
In-degree of vertex 4 = 2	Out-degree of vertex 4 = 0

6. Path:

- A path from v_i to v_j is a sequence of edges edge (v_1, v_2), edge(v_2,v_3),….edge(v_{n-1},v_n) and is denoted as path v_1,v_2,v_3,… v_{n-1},v_n.
- A path in a graph is a sequence of edges, each one incident to the next.
- If $v_1 = v_n$ and no edge is repeated, then the path is called a circuit.
- A path is a **circuit** if it begins and ends at the same vertex and has length ≥1.
- A path or circuit is **simple** if it does not include the same edge more than once.

7. Cycle:

- A *path* from a node to itself is called a *cycle*.
- A circuit that doesn't repeat vertices is called a cycle.

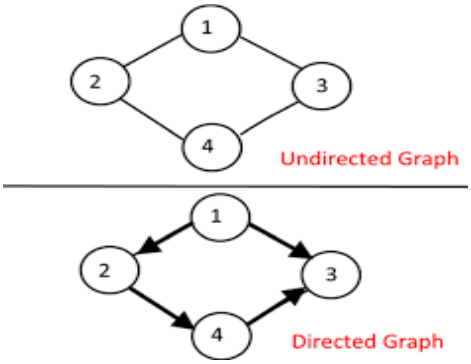
Types of Graph:

1. Undirected Graph:

- In an undirected graph, the pair of vertices representing any edge is unordered. Thus *(u, v)* and *(v, u)* represent the same edge.

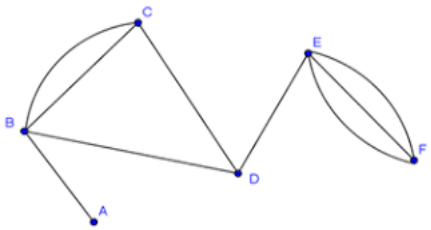
2. Directed Graph:

- In a directed graph, each edge is an ordered pair of vertices i.e., each edge is represented by a directed pair. We use angled brackets to represent an ordered pair. If ***e=<v, w>***, then ***v*** is initial vertex and ***w*** is the final vertex. Thus ***<v, w>*** and <***w, v>*** represent two different edges.
- In directed graph $(v_i, v_j) \neq (v_j, v_i)$



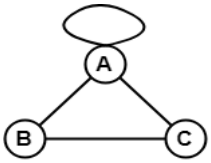
3. Multigraph:

- If there exists multiple edges between a pair of vertices in a graph, then it is called a multi graph.



4. Pseudograph:

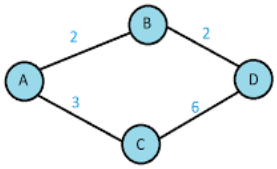
- A graph that contains at least one loop is called a pseudo graph.



Pseudo Graph

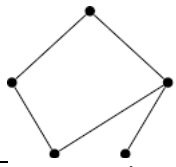
5. Weighted Graph:

- The graph is called a weighted graph if each edge of graph is assigned with some weight or value.

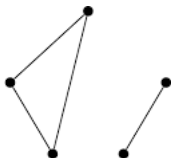


6. Connected and Disconnected Graph:

- An undirected graph is said to be *connected* if there exist a path from any vertex to any other vertex. Otherwise it is said to be *disconnected*.

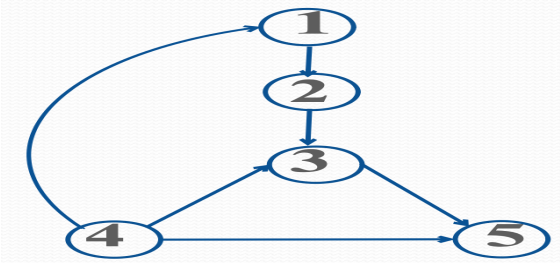


(a) Connected Graph

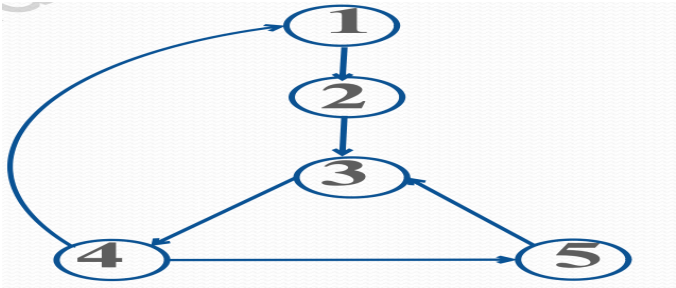


(b) Disconnected Graph

A directed graph is ***strongly connected*** if there exists a **directed path** from any vertex to any other vertex. A directed graph is ***weakly connected*** if there exists a path between any vertex to any other vertex in the underlying undirected graph (i.e. when the directions of the edges are disregarded).
The below graph is weakly connected since there is no direct path exists from vertex 1 to vertex 4 and also from vertex 5 to any other vertex.



The below graph is strongly connected:



7. <u>Complete Graph or Fully Connected Graph:</u>

- A graph is said to be complete or fully connected, if there is exactly one edge between each pair of distinct vertices.
- A complete graph with ***n*** vertices has ***n*(n-1)/2** edges.
- It is denoted by K_n.

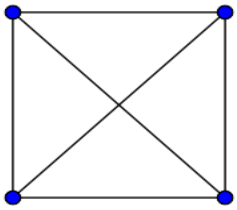
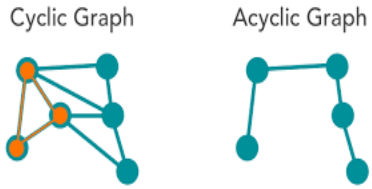
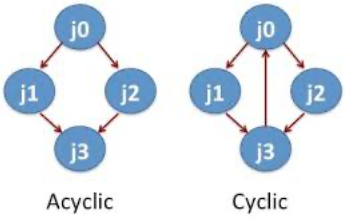


Figure: A K_4 graph

8. <u>Cyclic Graph and Acyclic Graph:</u>

- If a graph contains a cycle, it is a cyclic graph; otherwise, it is an acyclic graph.
- *A directed acyclic graph is called a "dag" from its acronym.*





Graph as an ADT/ Operations on Graph:

1. Creating a Graph:

To create a graph, first adjacency list array is created to store the vertices name, dynamically at the run time. Then the node is created and linked to the list array if an edge is there to the vertex.

Step 1: Input the total number of vertices in the graph, say *n*.

Step 2: Allocate the memory dynamically for the vertices to store in list array.

Step 3: Input the first vertex and the vertices through which it has edge(s) by linking the node from list array through nodes.

Step 4: Repeat the process by incrementing the list array to add other vertices and edges.

Step 5: Exit.

2. Searching and Deleting from A Graph:

Step 1: Input an edge to be searched

Step 2: Search for an initial vertex of edge in list arrays by incrementing the array index.

Step 3: Once it is found, search through the link list for the terminal vertex of the edge.

Step 4: If found display "the edge is present in the graph".

Step 5: Then delete the node where the terminal vertex is found and rearrange the link list.

Step 6: Exit.

3. Traversing A Graph:

A graph traversal which means visiting all the nodes of the graph. There are two graph traversal methods which are: Breadth First Traversal and Depth First Traversal.

Graph Representation:

There are various ways to represent a graph.

- A simple representation is given by an **adjacency list** which specifies all vertices adjacent to each vertex of the graph. This list can be implemented as table, in which case it is called a star representation which can be forward or reverse.
- Another representation is a **matrix**, which comes in two forms: an adjacency matrix and an incidence matrix.

1. An adjacency matrix of graph g = (V, E) is a binary |V|×|V| matrix such that each entry of this matrix:

a_{ij} = 1 if there exists an edge (v_i, v_j)

0 Otherwise.

Generalization of this definition to multigraph is obtained by transforming the definition into the following form:

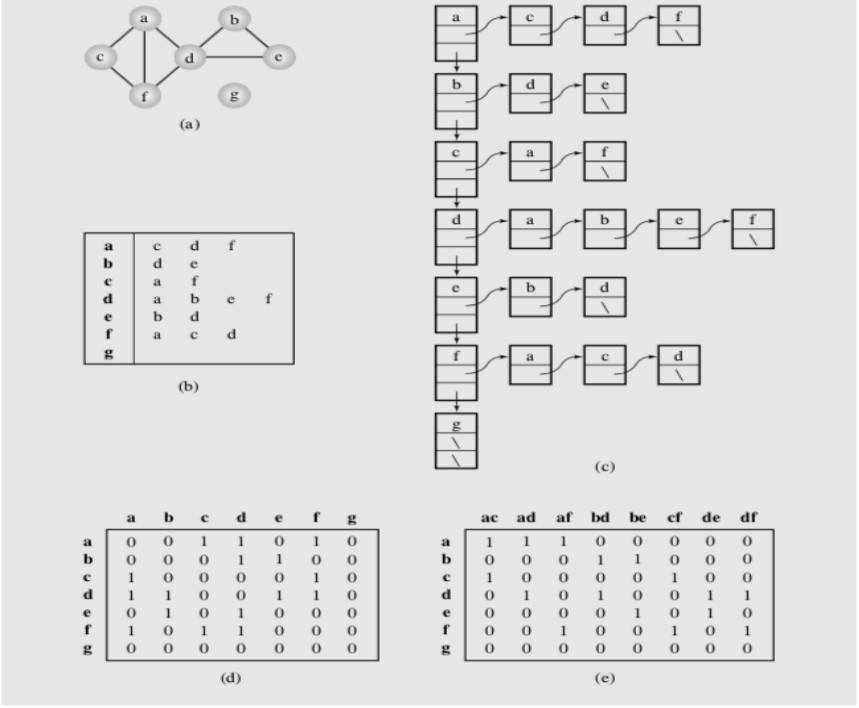
a_{ij} = number of edges between v_i and v_j.

2. Another matrix representation of a graph is based on the incidents of vertices and edges and is called an incidence matrix. An incidence matrix of graph G = (V, E) is a |V|×|E| matrix such that:

a_{ij} = 1 if edge e_j is incident with vertex v_i

0 otherwise

FIGURE 8.2 Graph representations. (a) A graph represented as (b–c) an adjacency list, (d) an adjacency matrix, and (e) an incidence matrix.



Graph Traversal

As in the trees, traversing a graph consists of visiting each vertex only one time. The simple traversal algorithms used for trees cannot be applied because graphs may include cycles; hence tree traversal algorithms would result in infinite loops. To prevent that from happening, each visited vertex can be marked to avoid revisiting it.

There are two types of traversals.

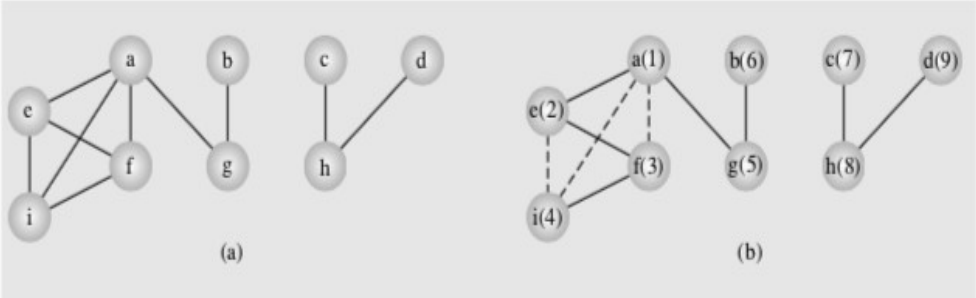
Depth First Traversal: This algorithm was developed by John Hop croft and Robert Tarjan.

- In this algorithm, each vertex v is visited and then unvisited vertex adjacent to v is visited.
- If a vertex v has no adjacent vertices or all of its adjacent vertices have been visited, we backtrack to the predecessor of v.
- The traversal is finished if this visiting and backtracking process leads to the first vertex where the traversal started.
- If there are still some unvisited vertices in the graph, the traversal continues restarting for one ofthe unvisited vertices.
- The algorithm assigns a unique number to each accessed vertex so that vertices are now renumbered.

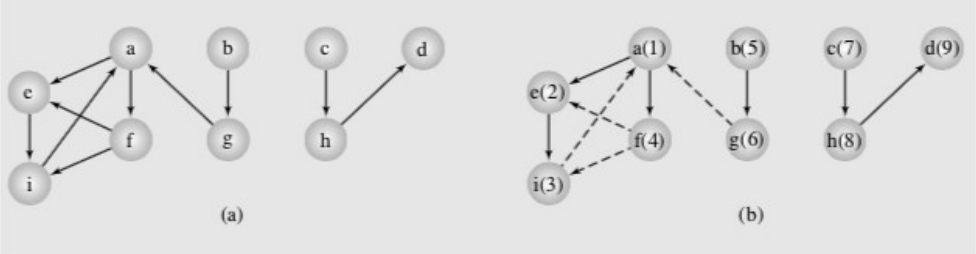
```
DFS(v)
    num(v) = i++;
    for all vertices u adjacent
       to vif num(u) is 0
          attach edge (uv) to
          edges;DFS(u);
DepthFirstSearch
()for all vertices v
    num(v) =
    0; edges =
    null;i = 1;
    while there is a vertex v such that num
       (v) is 0DFS(v)
    output edges;
```

⇒ The complexity of depth First Search is O(|V|+|E|) because initializing num(v) for each vertex v requires |V| steps

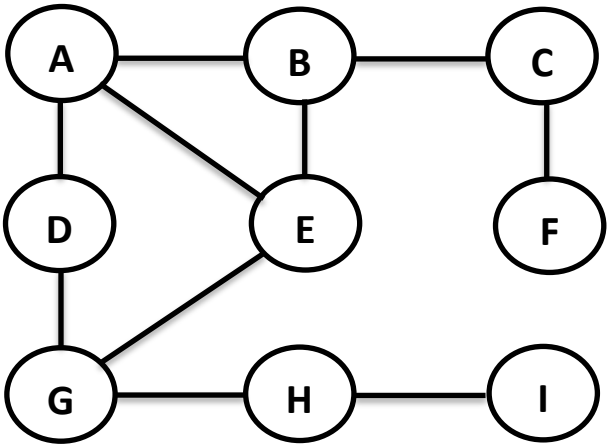
Example 1



DFT For Digraph:

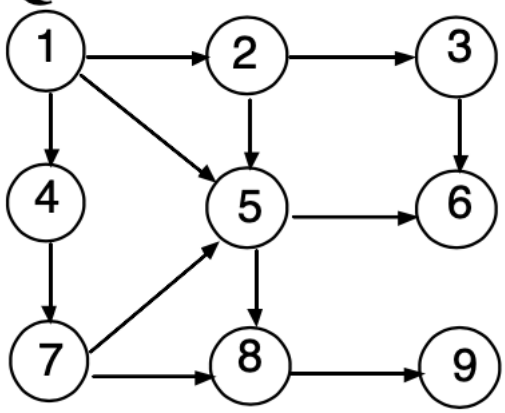


Example 2:



Depth First Traversal: A, B, C, F, E, G, D, H, I

Question :-



Depth First Traversal: 1, 4, 7, 5, 8, 9, 6, 2, 3

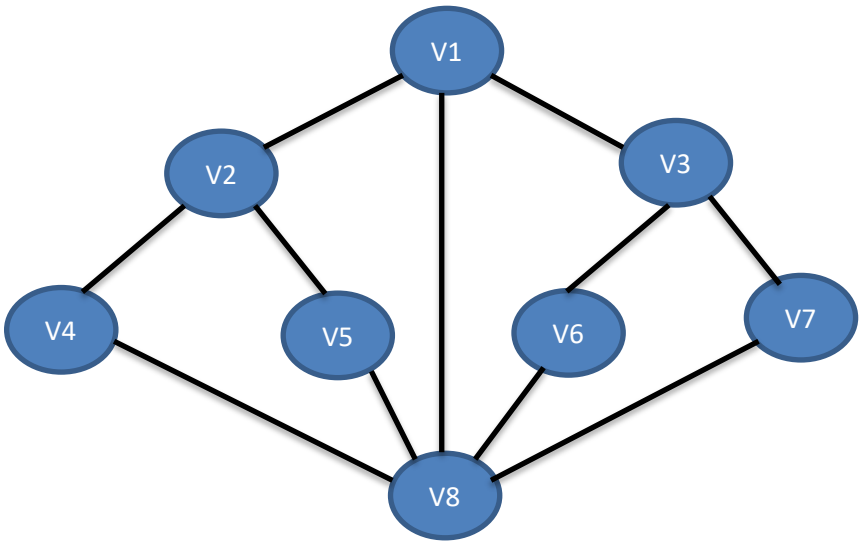
Breadth First Traversal (BFS):

- breadth-first traversal uses a queue as the basic data structure.
- first tries to mark all neighbors of a vertex *v* before proceeding to other vertices
- whereas DFS () picks one neighbor of a *v* and then proceeds to a neighbor of this neighbor beforeprocessing any other neighbors of *v*

Pseudocode for Breadth First Traversal:

```
BFS(G)
    add start vertex to Q
    mark start as visited
    while (Q is not empty)
          v = dequeue(Q);
          for each adjacent vertex av of v
              if av is not visited
                    add av to Q;
                    mark av as visited
```

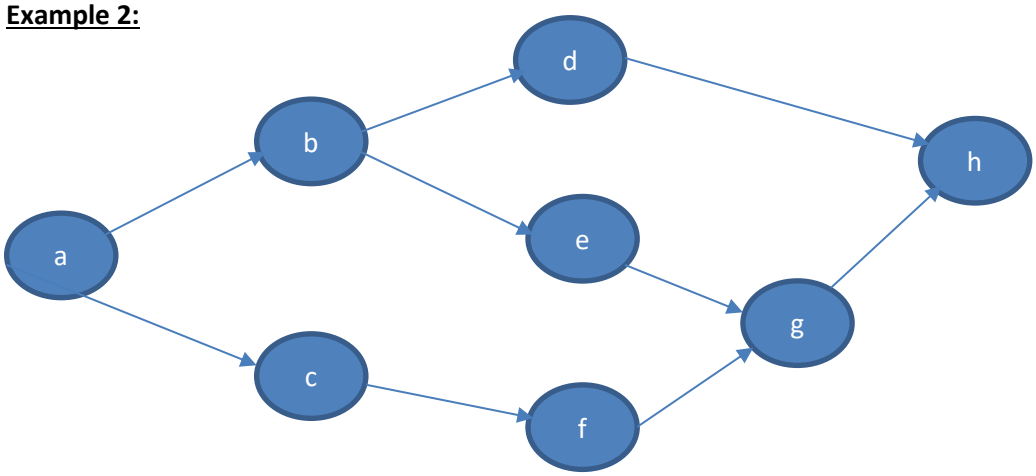
Example 1:



- We start with vertex V1. Its adjacent vertices are V2, V8 and V3. We visit all one-by-one and order these vertices say the order be V2, V3, V8.
- Then we pick on V2. The unvisited adjacent vertices to V2 are V4 and V5. We visit both and then go back to the remaining visited vertices of V1.
- Then we pick on V3. The unvisited adjacent vertices to V3 are V6 and V7. Now, there are no more adjacent vertices of V8, V4, V5, V6 and V7.

Thus, the breadth-first search generates the sequence: V1, V2, V8, V3, V4, V5, V6, V7.

Example 2:



Breadth First Traversal of given digraph starting from a is: a, b, c, d, e, f, h, g
: a, c, b, f, d, e, h, g
: a, c, b, f, e, d, g, h

Greedy Algorithm:

- A greedy algorithm **always makes the choice that seems to be the best at that moment**. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.
- Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

Example: Counting Coins:

This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ₹ 1, 2, 5 and 10 and we are asked to count ₹ 18 then the greedy procedure will be −

- Select one ₹ 10 coin, the remaining count is 8
- Then select one ₹ 5 coin, the remaining count is 3
- Then select one ₹ 2 coin, the remaining count is 1
- And finally, the selection of one ₹ 1 coin solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.

For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1)

Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

Examples of Greedy Algorithm:

1. Dijkstra's Shortest Path Algorithm
2. Kruskal's Minimum Spanning Tree Algorithm
3. Prim's Minimum Spanning Tree Algorithm
4. Travelling Salesman Problem

Shortest Path Algorithm

- It is used to find the shortest path form one node to another node.
- A single source vertex and seek shortest path to all other vertices.
- Shortest path is that path in which the sum of weight of included edges is minimum.
- Dijkstra's Algorithm is used find shortest path.

Dijkstra's Algorithm:

DijkstraAlgorithm (*weighted simple* graph, *vertex* first)

for *all vertices* v

currDist(v) = ∞;

currDist(first) = 0;

toBeChecked = *all vertices*;

while toBeChecked *is not empty*

v = *a vertex in* toBeChecked *with minimal currDist*(v);

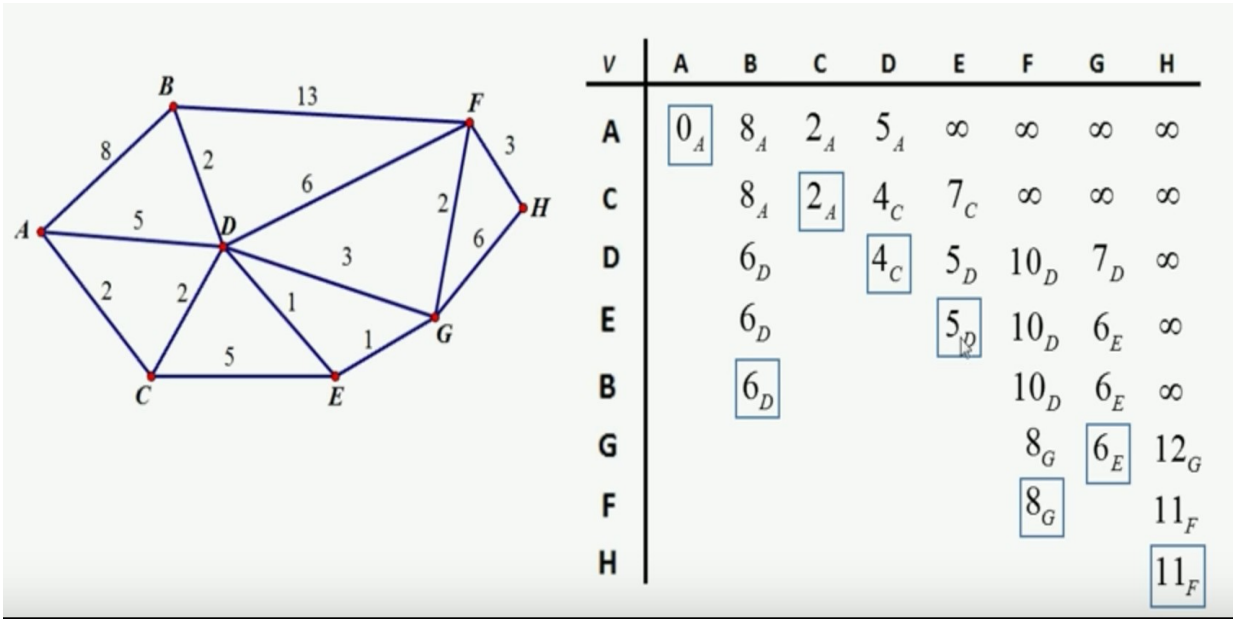
remove v *from* toBeChecked;

for *all vertices* u *adjacent to* v *and in* toBeChecked

if *currDist*(u) > *currDist*(v)+ *weight*(*edge*(vu))

currDist(u) = *currDist*(v)+ *weight*(*edge*(vu));

predecessor(u) = v;



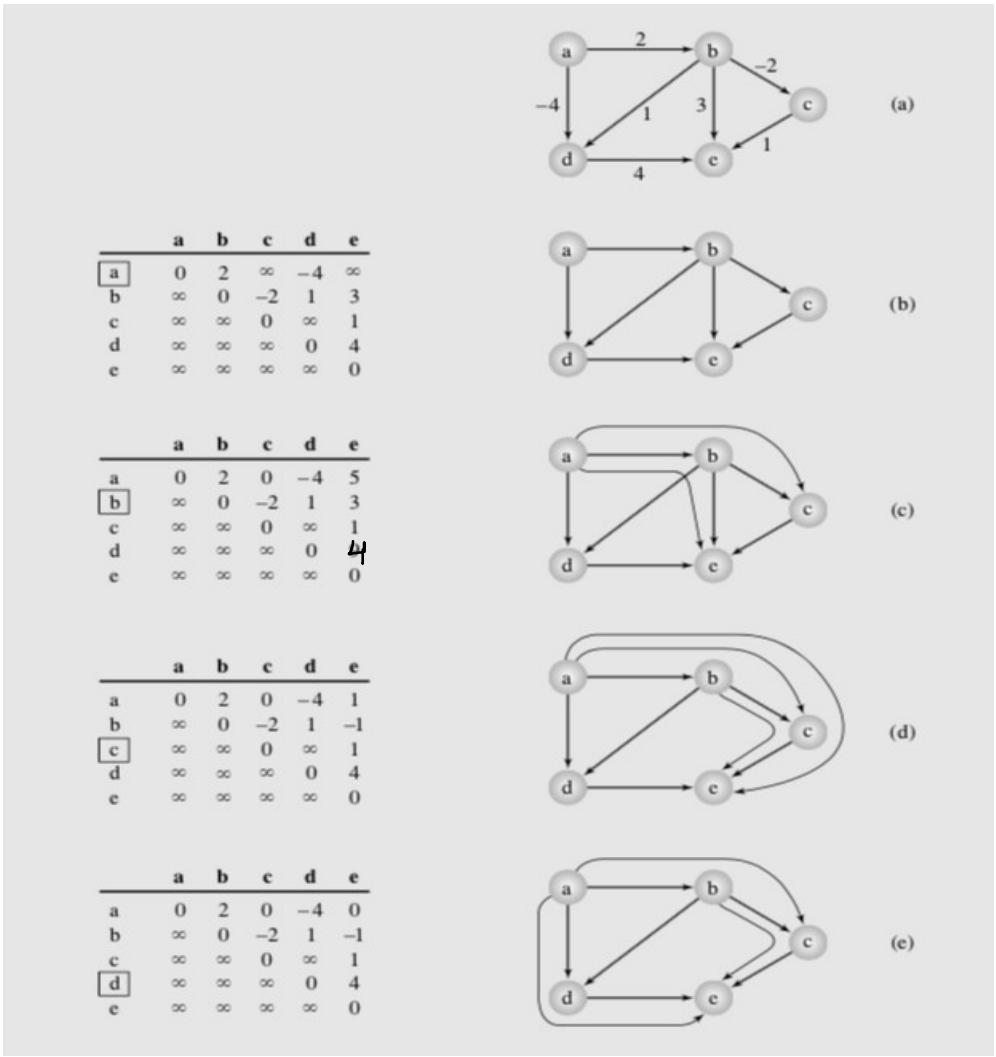
V	A	B	C	D	E	F	G	H
A	0_A	8_A	2_A	5_A	∞	∞	∞	∞
C		8_A	2_A	4_C	7_C	∞	∞	∞
D		6_D		4_C	5_D	10_D	7_D	∞
E		6_D			5_D	10_D	6_E	∞
B		6_D				10_D	6_E	∞
G						8_G	6_E	12_G
F						8_G		11_F
H								11_F

All to All Shortest Path Problem (Floyd Warshall Algorithm)

The task of finding all shortest paths from any vertex to any other vertex is more complex than the task of dealing with the one source only. But the method was designed by Stephen Warshall and implemented by Robert W Floyd and P, Z using adjacency matrix. The graph can include negative weights. The algorithm is as follows:

```
WFIalgorithm  (matrix  weight)
    for i=1 to |V|
        for j=1 to |V|
            for k = 1 to |V|
                if  weight[j][k]> weight[j][i]  +  weight[i][k]
                    weight[j][k] = weight[j][I ]+ weight[i][k];
```

The outermost loop refers to vertices that may be on a path between the vertex with index j and the vertex k.



	a	b	c	d	e
[a]	0	2	∞	−4	∞
b	∞	0	−2	1	3
c	∞	∞	0	∞	1
d	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

	a	b	c	d	e
a	0	2	0	−4	5
[b]	∞	0	−2	1	3
c	∞	∞	0	∞	1
d	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

	a	b	c	d	e
a	0	2	0	−4	1
b	∞	0	−2	1	−1
[c]	∞	∞	0	∞	1
d	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

	a	b	c	d	e
a	0	2	0	−4	0
b	∞	0	−2	1	−1
c	∞	∞	0	∞	1
[d]	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

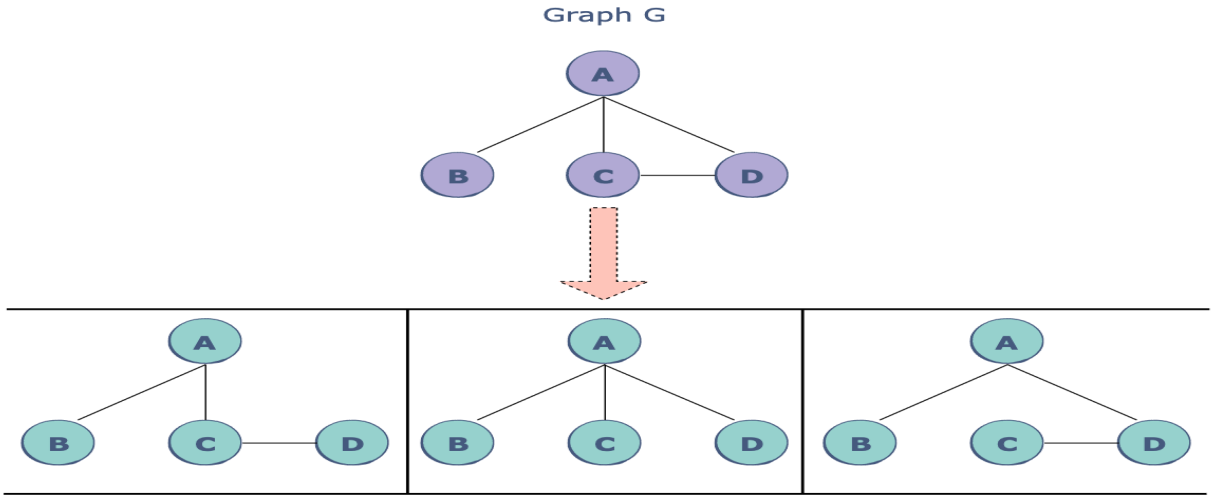
Spanning Tree:

The **spanning tree** of a graph (*G*) is a subset of *G* that covers all of its vertices using the minimum number of edges.

Some properties of a spanning tree can be deduced from this definition:

1. Since “a spanning tree covers all of the vertices”, it cannot be disconnected.
2. A spanning tree cannot have any cycles.
3. In a spanning tree, number of edges will be one less than number of vertices.

A graph can have more than one spanning trees.



Different possible spanning trees

Minimum Spanning Tree:

A minimum spanning tree in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges. In this part we study one algorithm that is used to construct the minimum spanning tree from the given connected weighted graph.

Algorithms to find minimum spanning trees are:

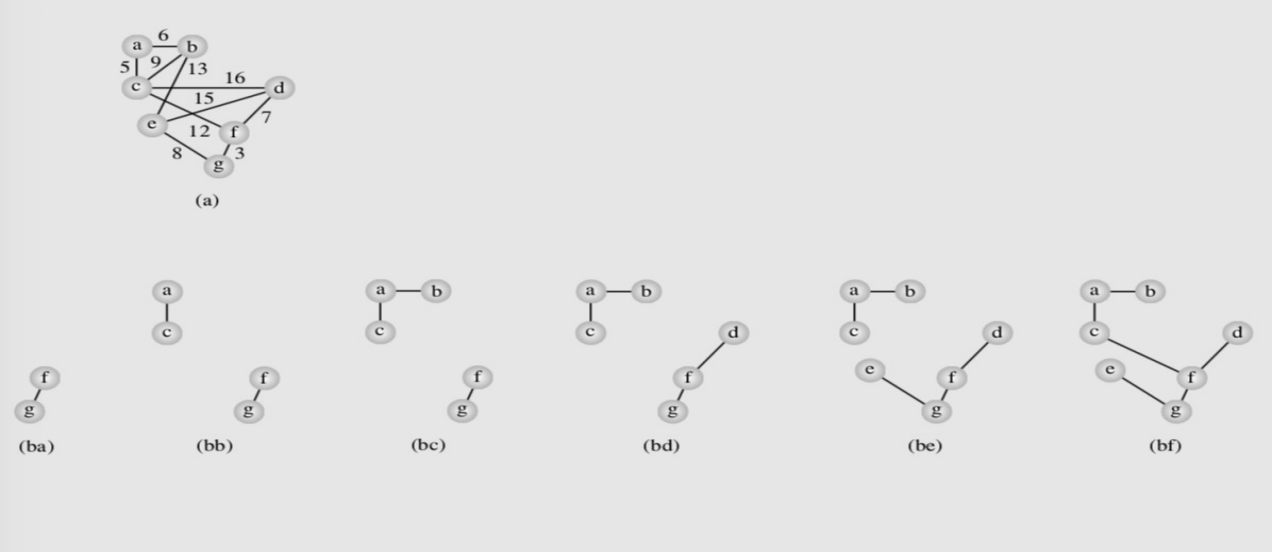
1. Kruskal’s Algorithm
2. Prim’s Algorithm

Spanning Trees: Consider the graph representing the airline's connections between seven cities. If the economic situation forces this airline to shut down as many connections as possible, which of them should be retained to make sure that it is still possible to reach any city from any other city, if only indirected? One popular algorithm was devised by Joseph Kruskal. In this, all edges are ordered by weight, and then each edge in this ordered sequence is checked to see whether it can be considered part of the tree under construction. It is added to the tree if no cycle arises after its inclusion.

1. Kruskal's Algorithm

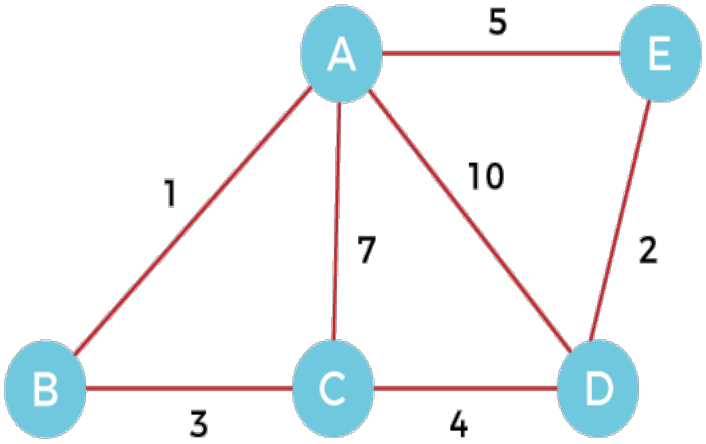
KruskalAlgorithm *(weighted connected undirected graph)*

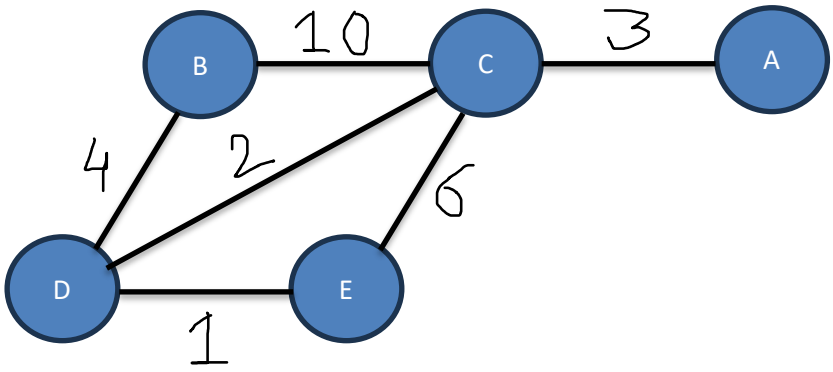
tree = null;
edges = sequence of all edges of graph sorted by weight;
for (i=1; i ≤|E| and |tree| < |V|-1; i++)
if e_i from edges does not form a cycle with edges in tree
add e_i to tree;



The complexity of this algorithm is determined by the complexity of the sorting method applied, which for an efficient sorting is O(|E|log|E|). It also depends on the complexity of the method used for cycle detection.

Question: Draw a minimum Spanning Tree from the graphs below:





2. Prim's Algorithm:

- It is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.
- Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

Algorithm:

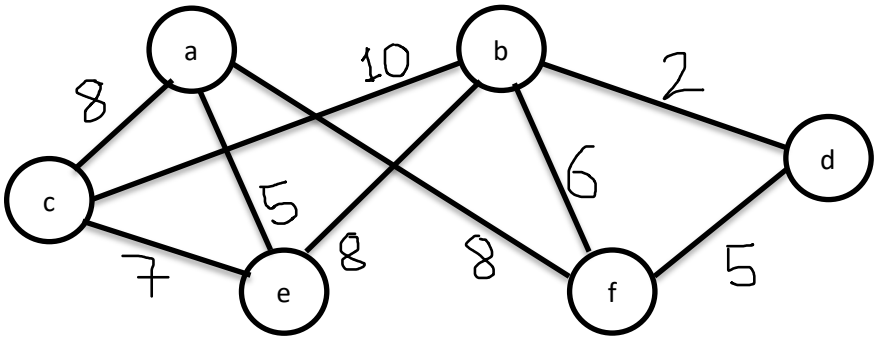
Step 1: Randomly choose any vertex.

Step 2: Find all the edges that connect the tree to new vertices.

Step 3: Find the least weight edge among those edges and include it in the existing tree if it does not form cycle. If it forms a cycle, look for the next least weight edge

Step 4: Keep repeating step 2 and step 3 until all the vertices are included

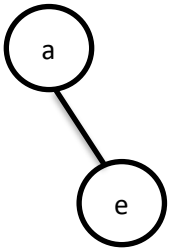
Example 1:



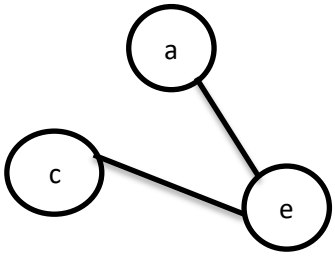
Let's start from vertex e:

Edges incident on e = {ea, eb, ec}

- Choose an edge **ea** since it has the least weight



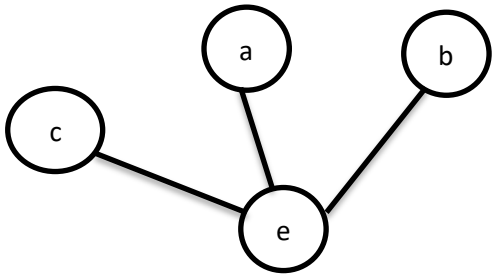
Now list of edges = {eb, ec, ac, af}



- Choose an edge **ec** since it has the least weight

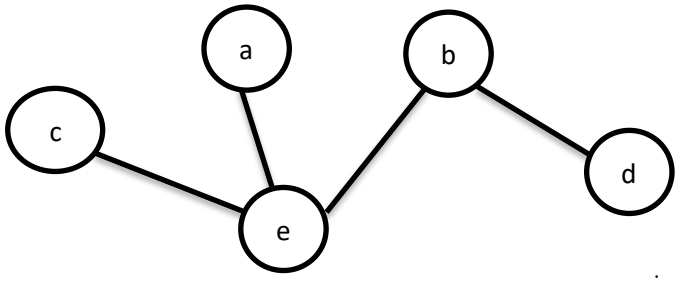
Now list of edges = {**eb**, **af**, **cb**} **[removed ac since it forms cycle]**

- Choose an edge **eb** since it has the least weight



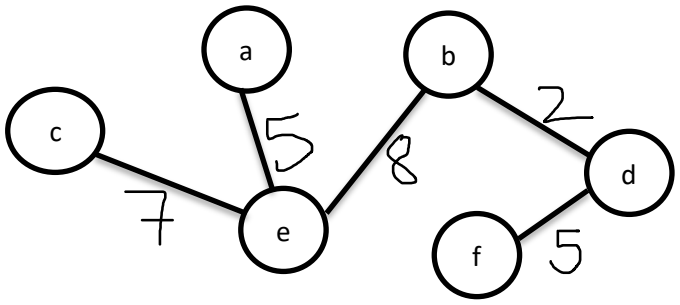
Now list of edges = {af, bf, bd} **[removed cb since it forms cycle]**

- Choose an edge bd since it has the least weight



Now list of edges = {af, bf, df}

- Choose an edge df since it has the least weight



Since all the vertices are included in the above graph, we stop the process.
Total weight of minimum spanning tree = 7+5+8+2+5 = 27 units

Topological Sort

In many situations, there is a set of tasks to be performed. For some pairs of tasks, it matters which task is performed first, where for other pairs, the order of execution is unimportant.
The dependencies between tasks can be shown in the form of a diagraph. A topological sort linearizes a digraph; that is, it labels all its vertices with numbers 1..... |V| so that i<j only if there is a path from vertex vi to vertex vj. The diagraph must not include a cycle; otherwise, a topological sort is impossible.
The algorithm for topological sort is rather simple. We have to find a vertex v with no outgoing edges, called a sink or a minimal vertex, and then disregard all edges leading from any vertex to v. The summary of the topological sort algorithm is as follows:

```
topologicalSort(digraph)
    for i=1 to |V|
        find a minimal vertex v;
        num(v)=i;
        remove from diagraph vertex v and all edges incident wi
```