

Graph and its representations

A graph is a data structure that consists of the following two components:

1. A finite set of vertices also called as **nodes**.
2. A finite set of ordered pair of the form (u, v) called as **edge**.

The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph

Directed/Undirected graph
Weighted/unweighted graph

(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v . The edges may contain weight/value/cost.

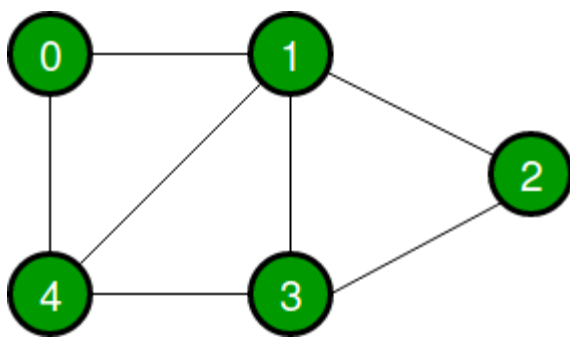
Graphs are used to represent many real-life applications:

- Graphs are used to represent networks.
- paths in a city or telephone network or circuit network.
- Graphs are also used in social networks like linkedIn, Facebook.

For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, and locale. See [this](#) for more applications of graph.

Node: (**User_id**, email_id, name, phone)

Following is an example of an undirected graph with 5 vertices.



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

1. Adjacency Matrix
2. Adjacency List

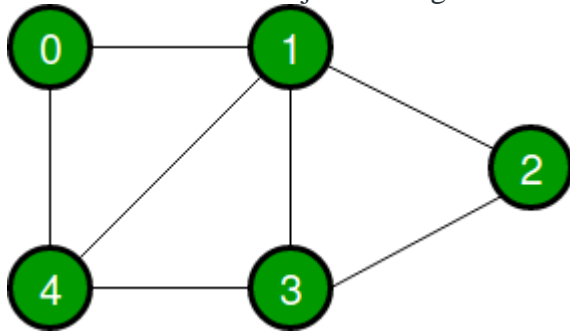
There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size $V \times V$ 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 for undirected graph is always symmetric.

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:

	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

Pros: Representation is easier to implement and follow.

Removing an edge takes $O(1)$ time.

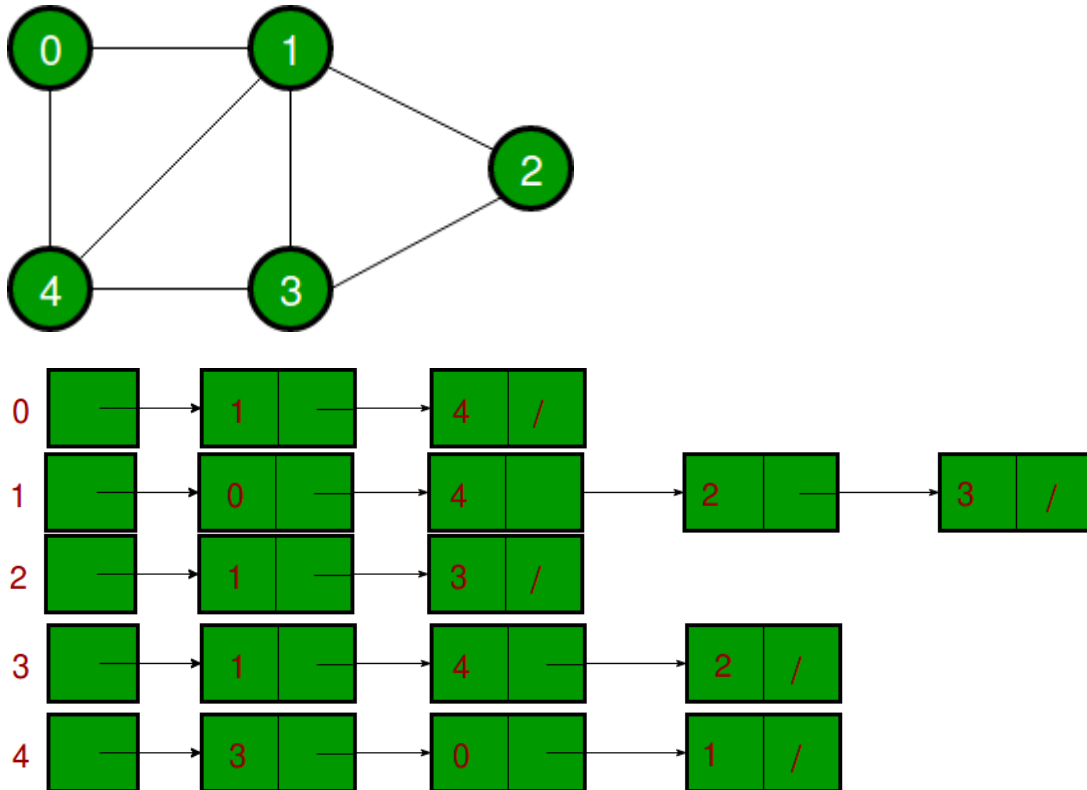
Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done $O(1)$.

Cons: Consumes more space $O(V^2)$.

Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is $O(V^2)$ time.

Adjacency List:

An array of 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 *i*th 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.



0: 1,4

1:0,4,2,3

2:1,3

3:1,4,2

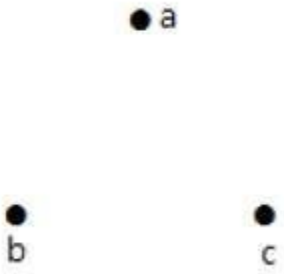
Node *User[5]

Some Information about Graph

Null Graph

A **graph having no edges** is called a Null Graph.

Example



In the above graph, there are three vertices named 'a', 'b', and 'c', but there are no edges among them. Hence it is a Null Graph.

Trivial Graph

A **graph with only one vertex** is called a Trivial Graph.

Example

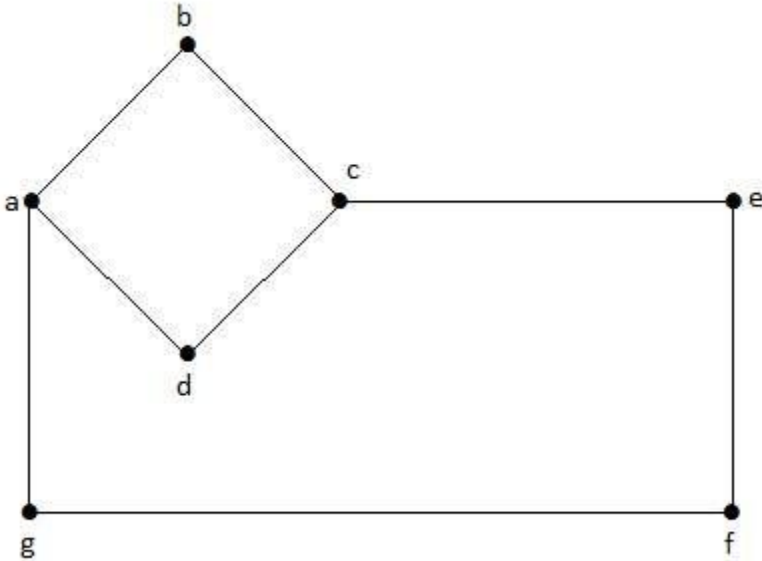


In the above shown graph, there is only one vertex 'a' with no other edges. Hence it is a Trivial graph.

Non-Directed Graph

A non-directed graph contains edges but the edges are not directed ones.

Example

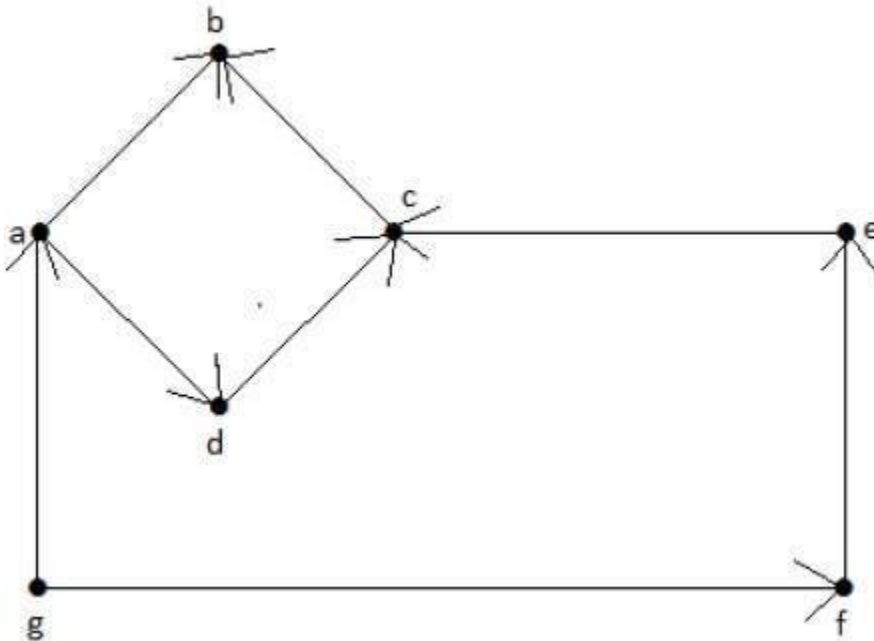


In this graph, 'a', 'b', 'c', 'd', 'e', 'f', 'g' are the vertices, and 'ab', 'bc', 'cd', 'da', 'ag', 'gf', 'ef' are the edges of the graph. Since it is a non-directed graph, the edges 'ab' and 'ba' are same. Similarly other edges also considered in the same way.

Directed Graph

In a directed graph, each edge has a direction.

Example



In the above graph, we have seven vertices 'a', 'b', 'c', 'd', 'e', 'f', and 'g', and eight edges 'ab', 'cb', 'dc', 'ad', 'ec', 'fe', 'gf', and 'ga'. As it is a directed graph, each edge bears an arrow mark that shows its direction. Note that in a directed graph, 'ab' is different from 'ba'.

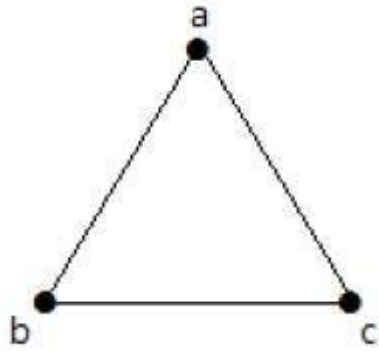
Simple Graph

A graph **with no loops** and **no parallel edges** is called a simple graph.

- The maximum number of edges possible in a simple graph with 'n' vertices is nC_2 where ${}^nC_2 = n(n-1)/2$.
- The number of simple graphs possible with 'n' vertices = $2^{{}^nC_2} = 2^{n(n-1)/2}$.

Example

In the following graph, there are 3 vertices with 3 edges which is maximum excluding the parallel edges and loops. This can be proved by using the above formulae.



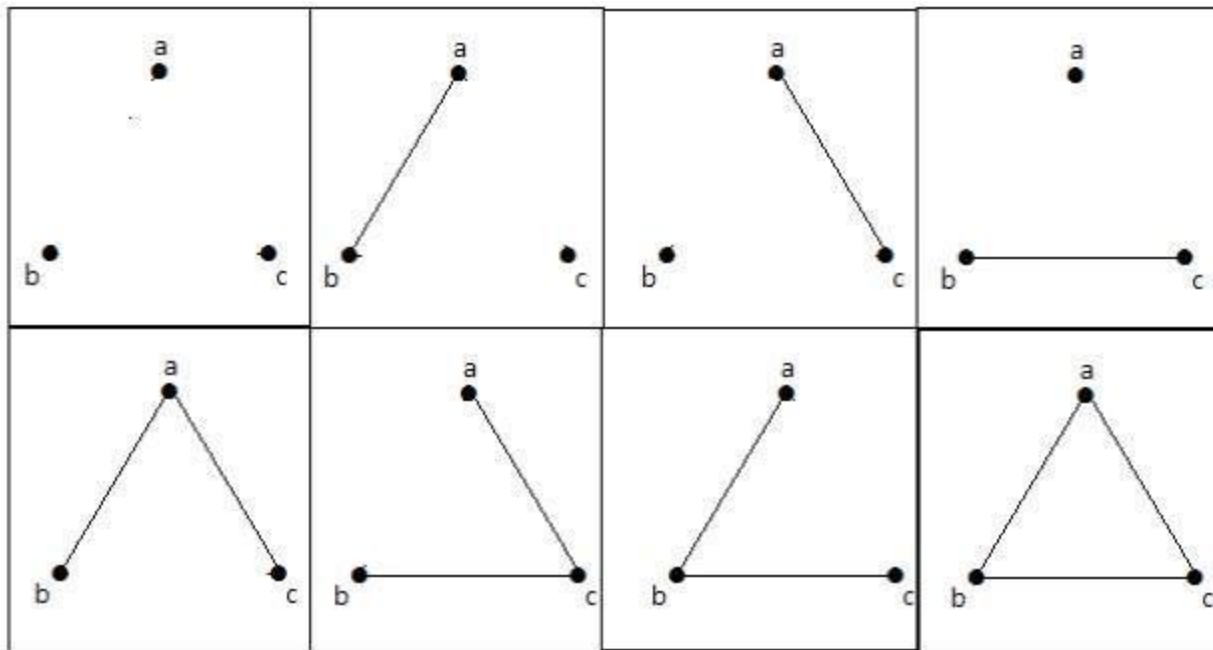
The maximum number of edges with $n=3$ vertices –

$$\begin{aligned}
 {}^nC_2 &= n(n-1)/2 \\
 &= 3(3-1)/2 \\
 &= 6/2 \\
 &= 3 \text{ edges}
 \end{aligned}$$

The maximum number of simple graphs with $n=3$ vertices –

$$\begin{aligned}
 2^n C_2 &= 2^{n(n-1)/2} \\
 &= 2^{3(3-1)/2} \\
 &= 2^3 \\
 &= 8
 \end{aligned}$$

These 8 graphs are as shown below –

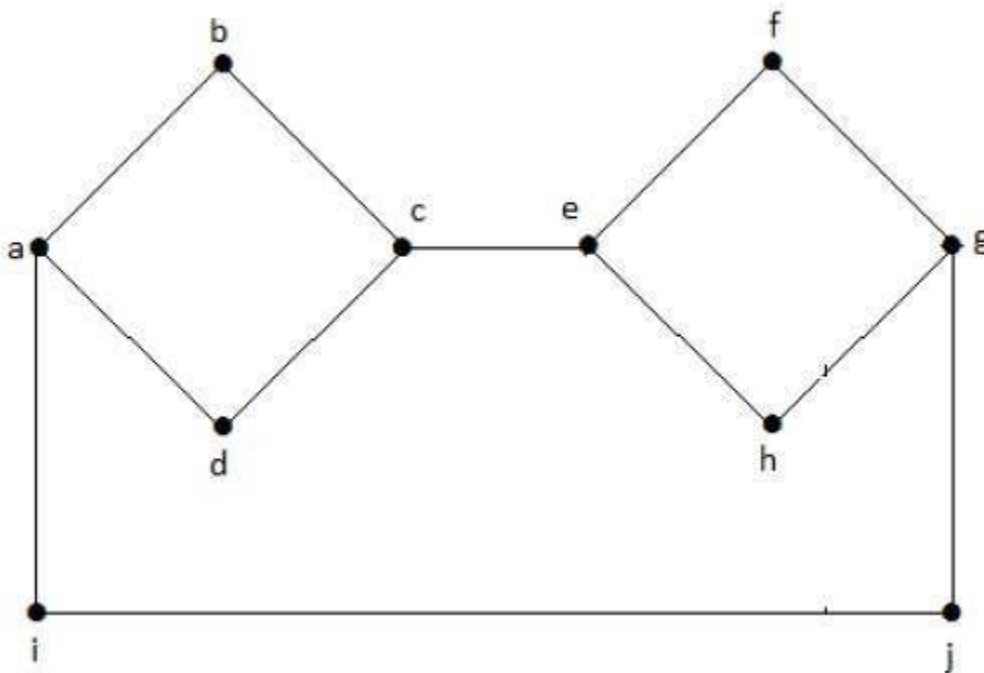


Connected Graph

A graph G is said to be connected **if there exists a path between every pair of vertices**. There should be at least one edge for every vertex in the graph. So that we can say that it is connected to some other vertex at the other side of the edge.

Example

In the following graph, each vertex has its own edge connected to other edge. Hence it is a connected graph.

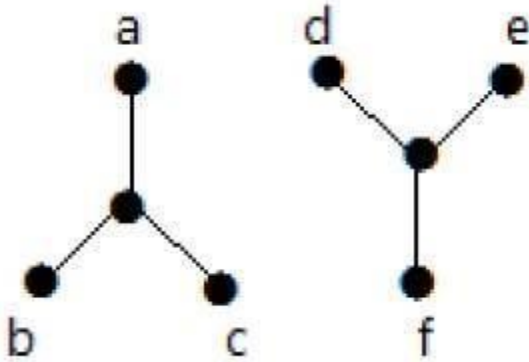


Disconnected Graph

A graph G is disconnected, if it does not contain at least two connected vertices.

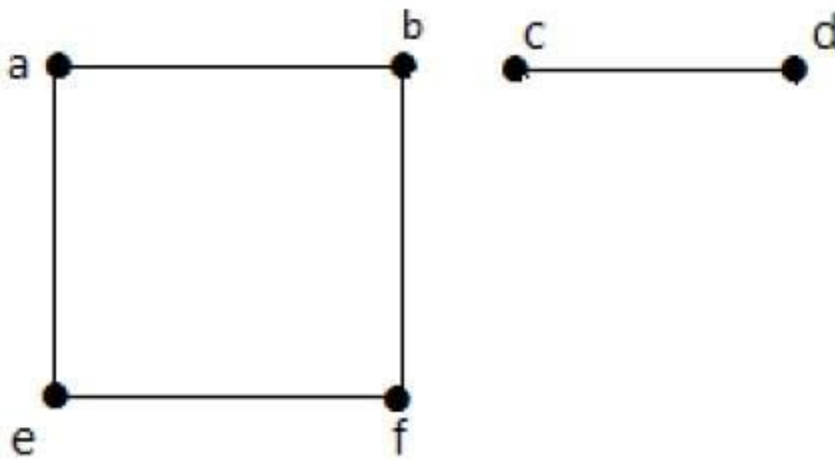
Example 1

The following graph is an example of a Disconnected Graph, where there are two components, one with 'a', 'b', 'c' vertices and another with 'd', 'e', 'f', 'g' vertices.



The two components are independent and not connected to each other. Hence it is called disconnected graph.

Example 2



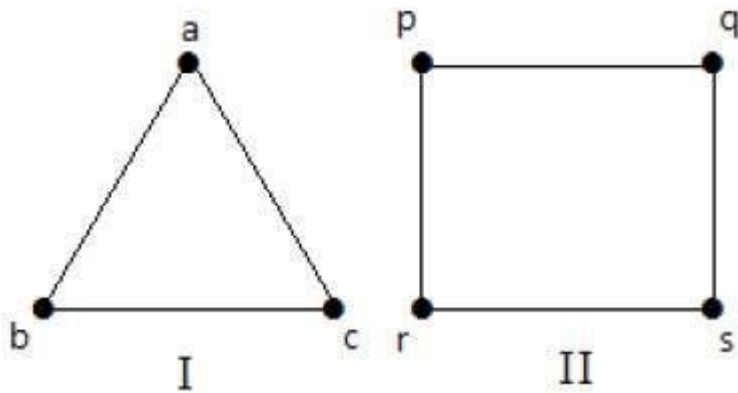
In this example, there are two independent components, a-b-f-e and c-d, which are not connected to each other. Hence this is a disconnected graph.

Regular Graph

A graph G is said to be regular, **if all its vertices have the same degree**. In a graph, if the degree of each vertex is ' k ', then the graph is called a ' k -regular graph'.

Example

In the following graphs, all the vertices have the same degree. So these graphs are called regular graphs.



In both the graphs, all the vertices have degree 2. They are called 2-Regular Graphs.

Complete Graph

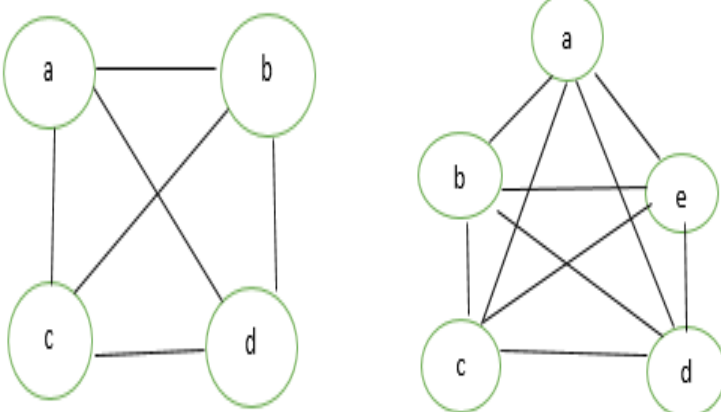
A simple graph with 'n' mutual vertices is called a complete graph and it is **denoted by ' K_n '**. In the graph, **a vertex should have edges with all other vertices**, then it called a complete graph.

In other words, if a vertex is connected to all other vertices in a graph, then it is called a complete graph.

All complete graphs are regular but vice versa is not possible.

Example

In the following graphs, each vertex in the graph is connected with all the remaining vertices in the graph except by itself.



Cycle Graph

A simple graph with ' n ' vertices ($n \geq 3$) and ' n ' edges is called a cycle graph if all its edges form a cycle of length ' n '.

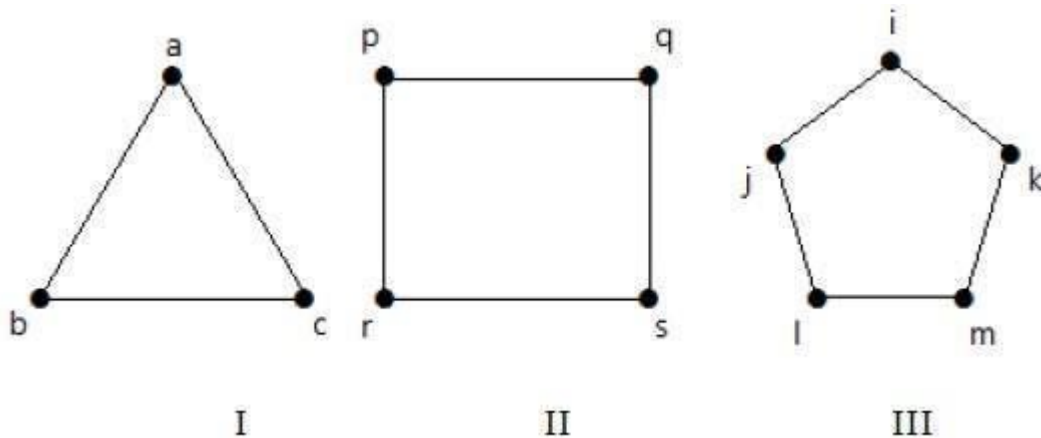
If the **degree of each vertex in the graph is two**, then it is called a Cycle Graph.

Notation – C_n

Example

Take a look at the following graphs –

- Graph I has 3 vertices with 3 edges which is forming a cycle 'ab-bc-ca'.
- Graph II has 4 vertices with 4 edges which is forming a cycle 'pq-qs-sr-rp'.
- Graph III has 5 vertices with 5 edges which is forming a cycle 'ik-km-ml-lj-ji'.

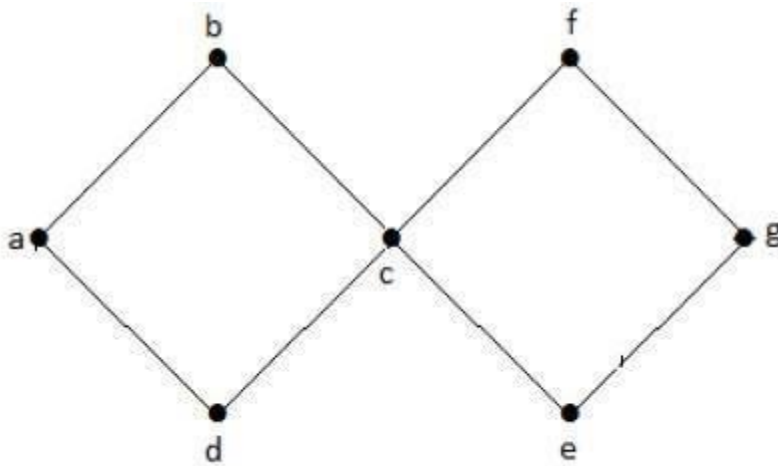


Hence all the given graphs are cycle graphs.

Cyclic Graph

A graph **with at least one** cycle is called a cyclic graph.

Example

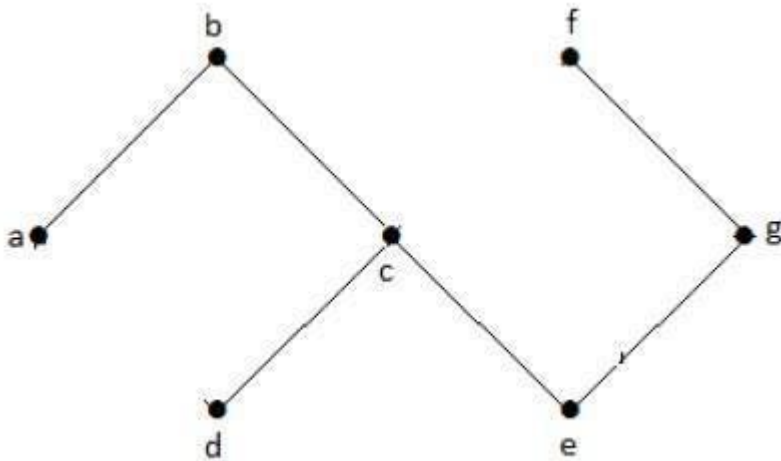


In the above example graph, we have two cycles a-b-c-d-a and c-f-g-e-c. Hence it is called a cyclic graph.

Acyclic Graph

A graph **with no cycles** is called an acyclic graph.

Example

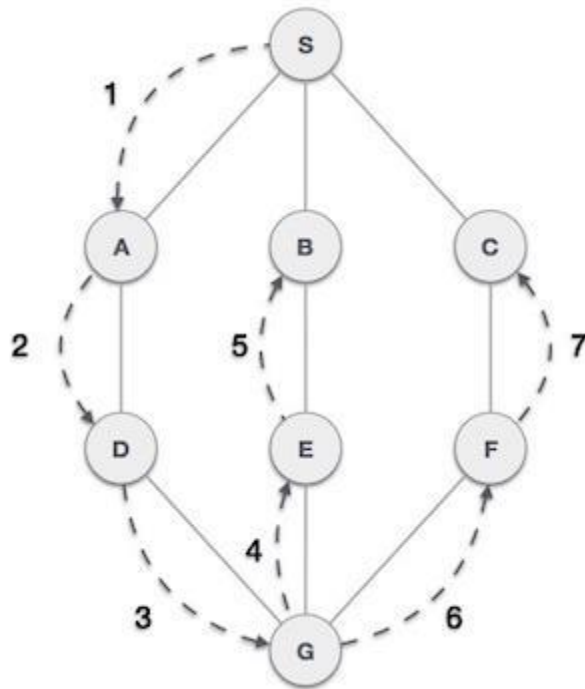


In the above example graph, we do not have any cycles. Hence it is a non-cyclic graph.

https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm

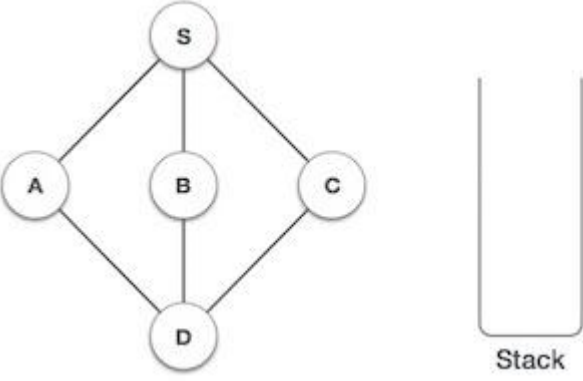
DFS

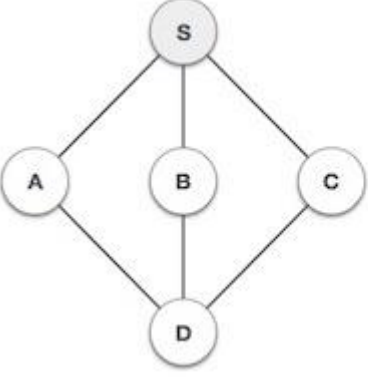
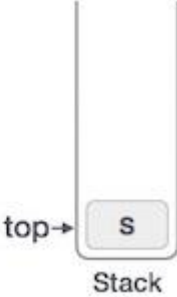
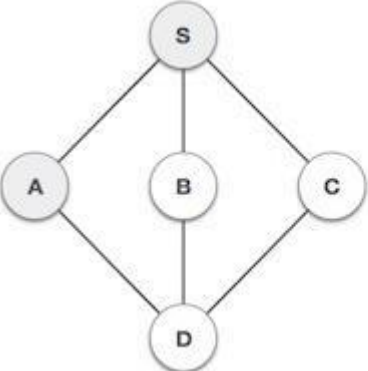
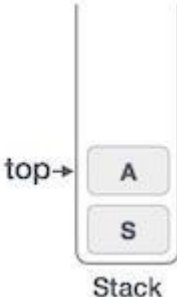
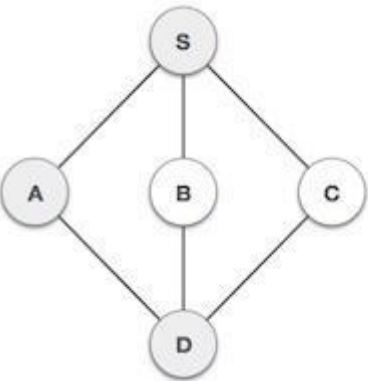
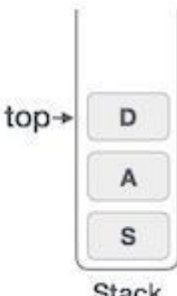
Depth First Search (DFS) algorithm traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

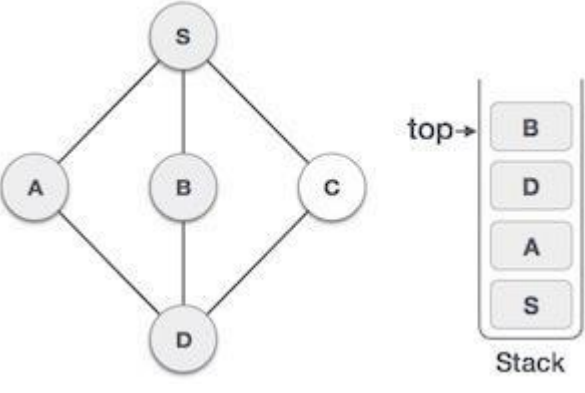
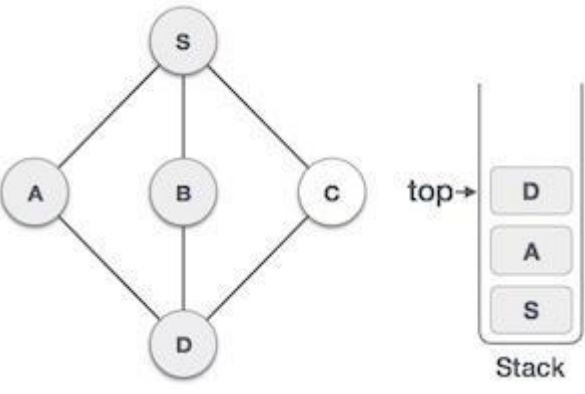
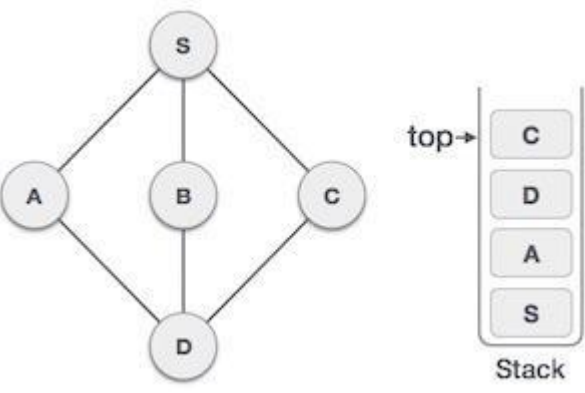


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.

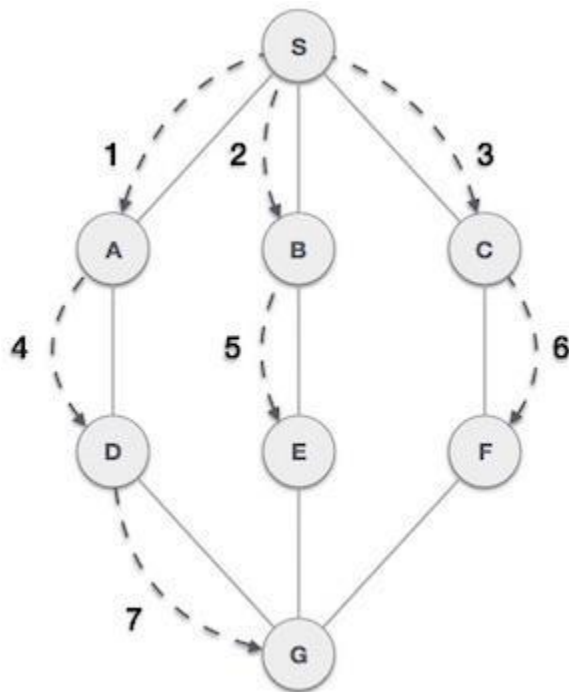
2	 	<p>Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.</p>
3	 	<p>Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.</p>
4	 	<p>Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.</p>

5		<p>We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.</p>
6		<p>We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.</p>
7		<p>Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.</p>

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

BFS

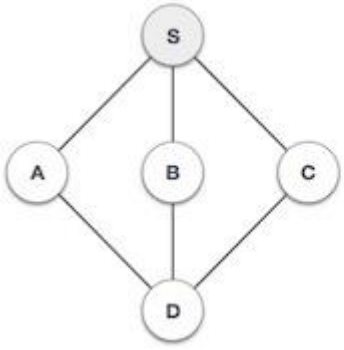
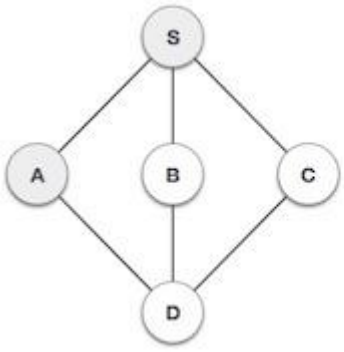
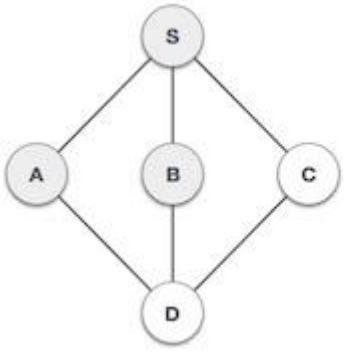
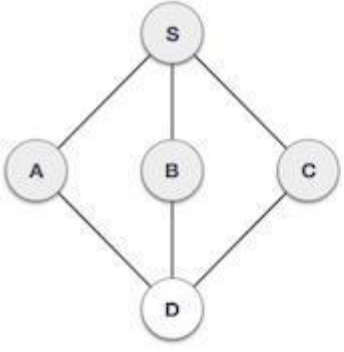
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



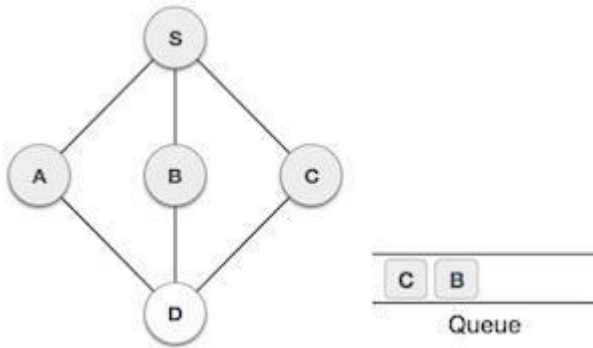
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1	<pre>graph TD; S((S)) --- A((A)); S --- B((B)); S --- C((C)); A --- D((D)); B --- D; C --- D;</pre>	Initialize the queue.

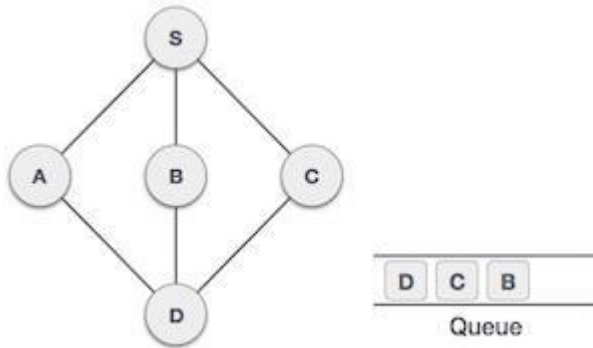
2	 <div data-bbox="644 449 872 543"><hr/><hr/><p>Queue</p></div>	We start from visiting S (starting node), and mark it as visited.
3	 <div data-bbox="644 848 872 940"><div data-bbox="656 858 704 900">A</div><hr/><hr/><p>Queue</p></div>	We then see an unvisited adjacent node from S . In this example, we have three nodes but alphabetically we choose A , mark it as visited and enqueue it.
4	 <div data-bbox="644 1245 872 1339"><div data-bbox="656 1257 704 1297">B</div><div data-bbox="709 1257 758 1297">A</div><hr/><hr/><p>Queue</p></div>	Next, the unvisited adjacent node from S is B . We mark it as visited and enqueue it.
5	 <div data-bbox="644 1644 872 1738"><div data-bbox="656 1654 704 1696">C</div><div data-bbox="709 1654 758 1696">B</div><div data-bbox="763 1654 813 1696">A</div><hr/><hr/><p>Queue</p></div>	Next, the unvisited adjacent node from S is C . We mark it as visited and enqueue it.

6



Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.

7



From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

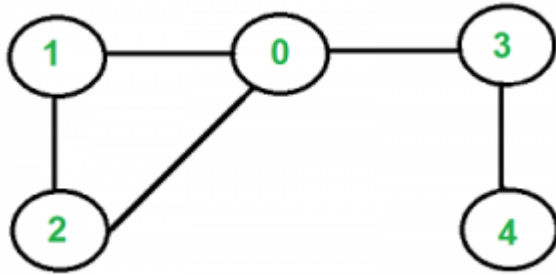
At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Cycle in **Undirected** graph (Use BFS)

You need maintain an array $\text{par}[i]$ = parent of node i ,

start bfs from node 1 and go on level wise.

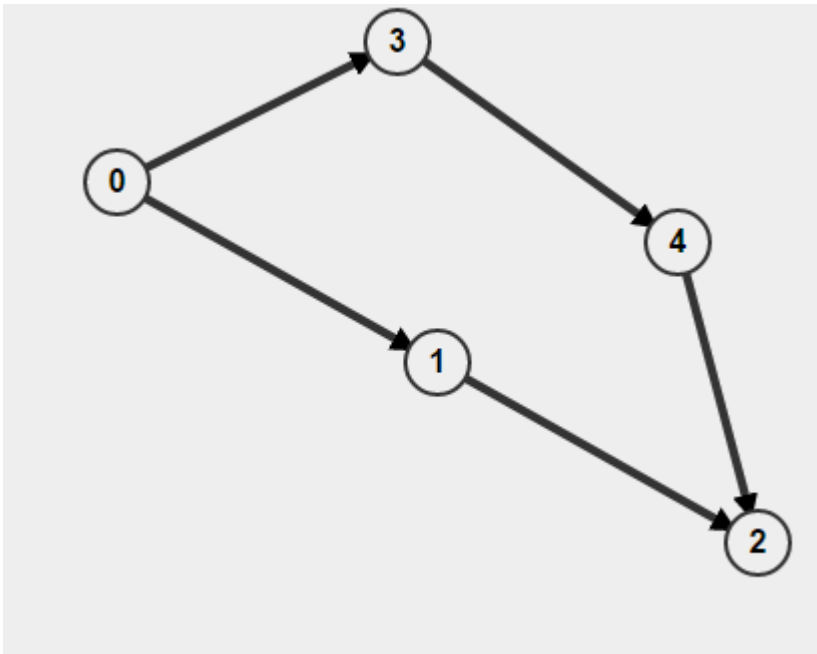
If a condition occurs when we are exploring neighbours of a node u and it is visited node but is not parent of u , then this is certainly a edge leading cycle.



1-0-2-1 is a cycle

Cycle in Directed Graph (Use DFS)

Also if your graph is directed then you have to not just remember if you have visited a node or not, but also how you got there. Otherwise you might think you have found a cycle but in reality all you have is two separate paths A->B but that doesn't mean there is a path B->A. For example



If you do BFS starting from 0, it will detect as cycle is present but actually there is no cycle.

With a depth first search you can mark nodes as visited as you descend and unmark them as you backtrack. See comments for a performance improvement on this algorithm.

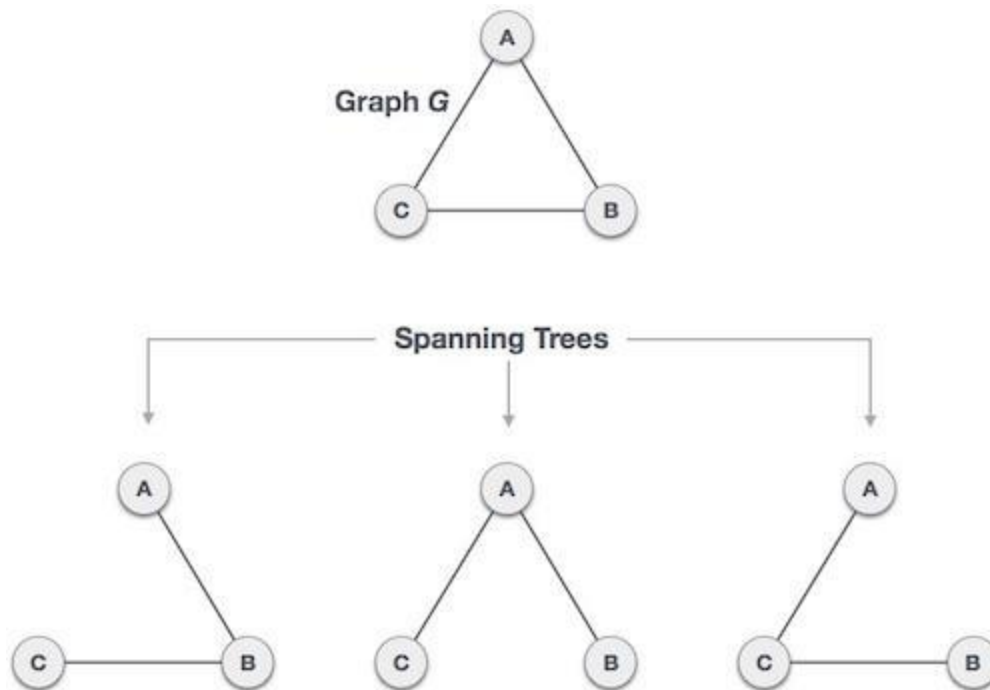
Good Algorithm (<https://iq.opengenus.org/cycle-using-degree-of-nodes-graph/>)

Spanning Tree

(https://www.tutorialspoint.com/data_structures_algorithms/spanning_tree.htm)

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence $3^{3-2} = 3$ spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree

- Spanning tree has $n-1$ edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum $e - n + 1$ edges, we can construct a spanning tree.
- A complete graph can have maximum n^{n-2} number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are –

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here –

- Kruskal's Algorithm
- Prim's Algorithm

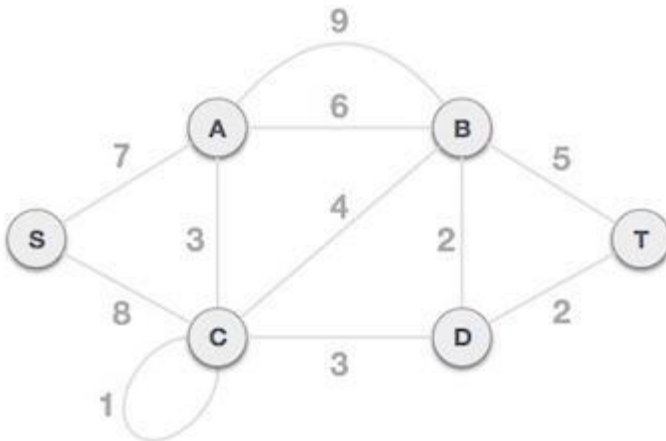
Both are greedy algorithms.

Prim's Spanning Tree Algorithm

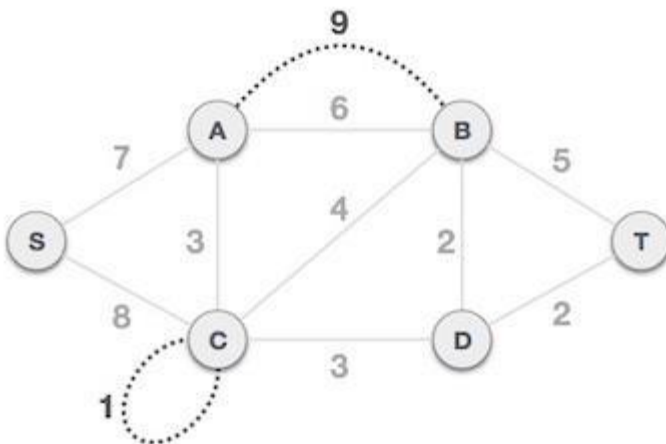
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

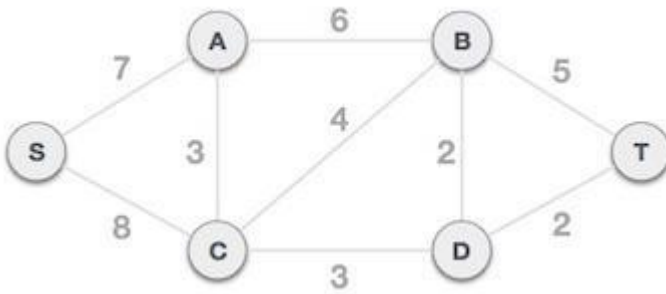
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

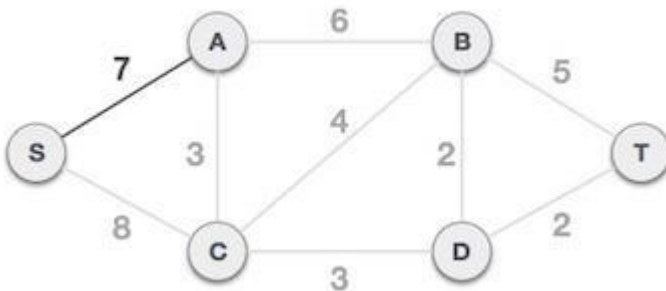


Step 2 - Choose any arbitrary node as root node

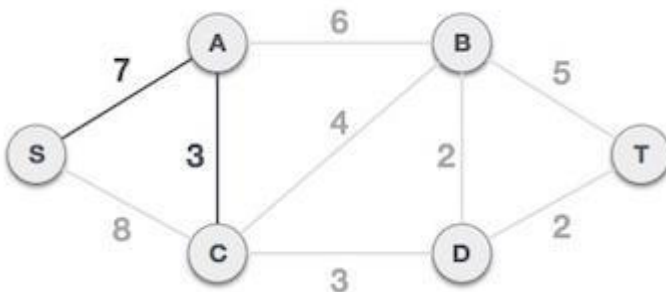
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any node can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

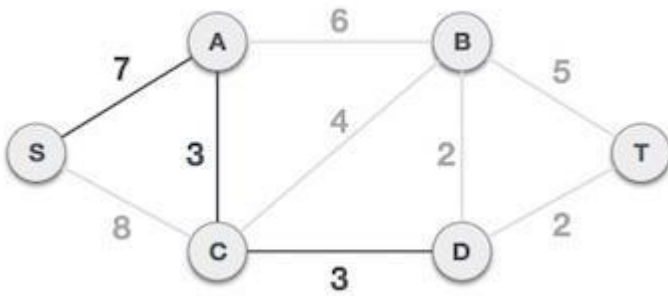
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.

