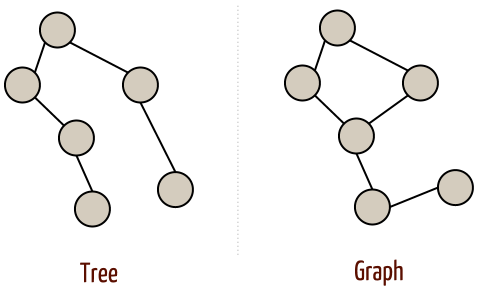
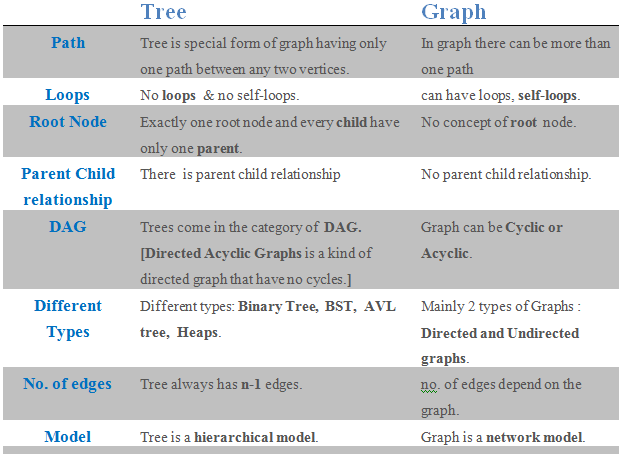
**GRAPHS**

**Contents:**

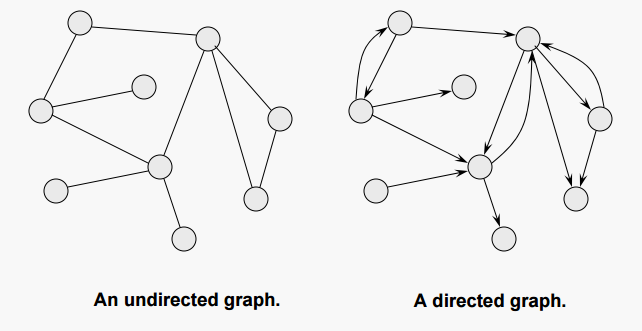
* Tree Vs Graph
* Directed Graphs
* Undirected Graphs
* Weighted Graphs
* Graph Representations:
* Incidence Matrix
* Adjacency Matrix Representation
* Adjacency List Representation
* Graph Traversals
* DFS
* BFS
* Some standard algorithms





Graph is a pair of sets (V, E) [where : V is the set of vertices E is the set of edges, connecting the pairs of vertices.

If the pairs of vertices are un-ordered, G is an undirected graph. If the pairs of vertices are ordered, G is a directed graph .



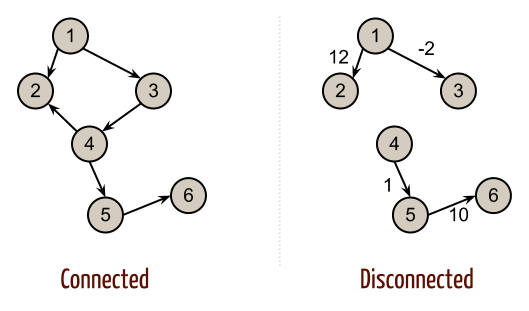
Connected Vs DisConnected Graph

**Connected Graph:**

From every vertex to any other vertex, there should be some path to traverse. That is called the connectivity of a graph.

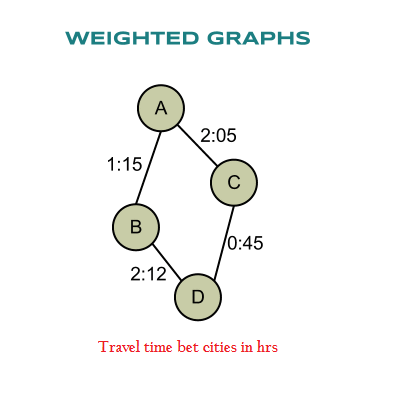
**Disconnected Graph:**

A graph with multiple disconnected vertices and edges is said to be disconnected.

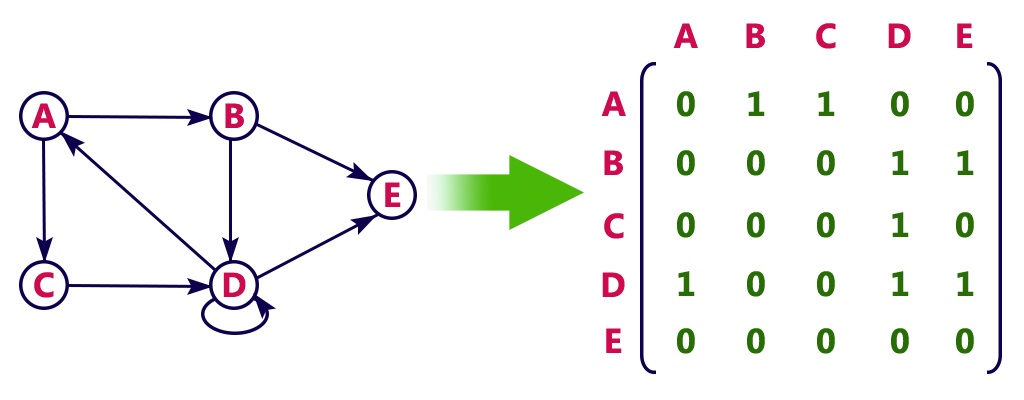


**Weighted Graphs**

We may also want to associate some cost or weight to the traversal of an edge. When we add this information, the graph is called weighted. Directed and undirected graphs may both be weighted.



What DS used for the representation of graphs:



**Incidence matrix**

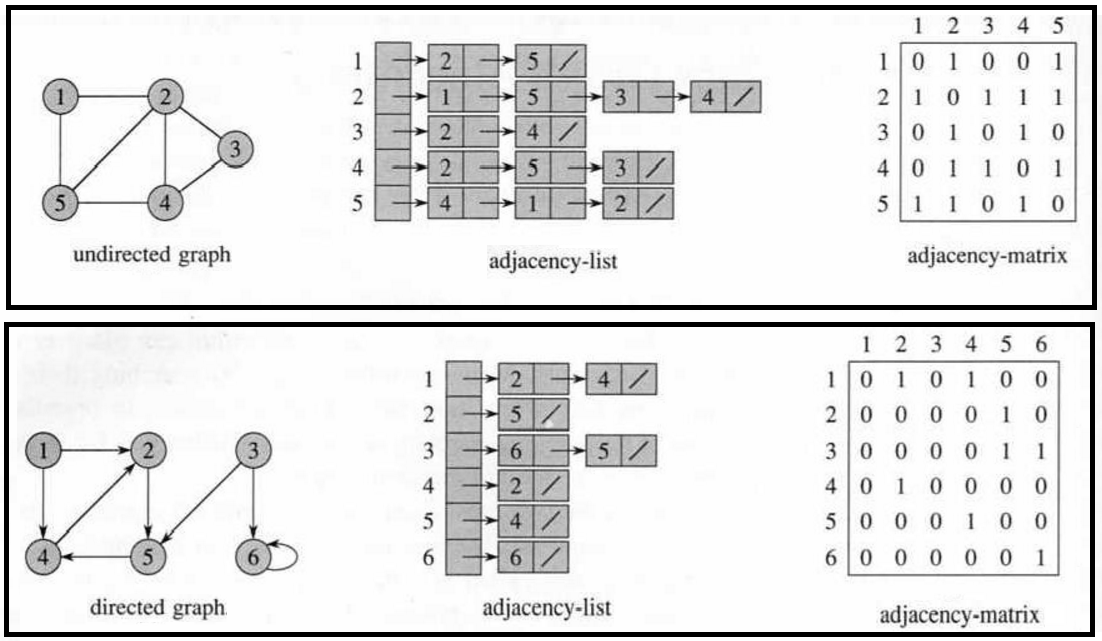
2-D Boolean matrix, in which the rows represent the vertices and columns represent the edges The entries indicate whether the vertex at a row is incident to the edge at a column.

**Adjacency list**

A more space-efficient way to implement a sparsely connected graph is to use an adjacency list. It keep a master list of all the vertices in the Graph object and then each vertex object in the graph maintains a list of the other vertices that it is connected to.

**Adjacency matrix**

A 2-D matrix, in which the rows represent source vertices and columns represent destination vertices.



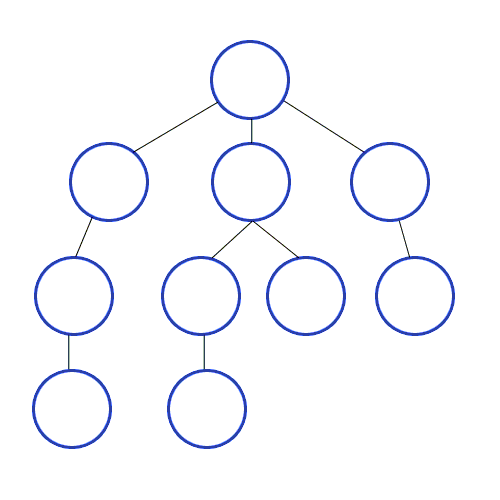
**Graph Traversal**

process of visiting each vertex in a graph

DFS:

Depth first search is a recursive algorithm that uses the idea of backtracking. Basically, it involves exhaustive searching of all the nodes by going ahead - if it is possible, otherwise it will backtrack. By backtrack, here we mean that when we do not get any further node in the current path then we move back to the node, from where we can find the further nodes to traverse. In other words, we will continue visiting nodes as soon as we find an unvisited node on the current path and when current path is completely traversed we will select the next path.

Implement using stack.



**Applications of DFS:**

**1) Check whether graph is connected or not**

We will start from a source node(can be any node) and mark all the nodes connected to source node as visited until no further node can be visited (stack gets empty). Now we can count the number of visited nodes and compare it with total number of nodes in the graph (entered at the time of input).

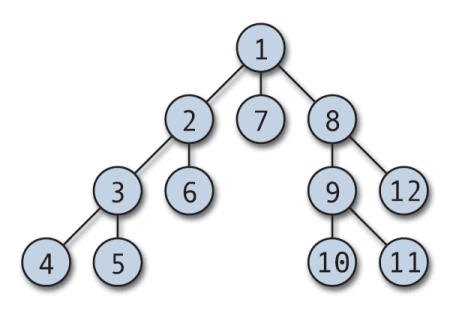
**2) Check whether a connected graph is cyclic or acyclic**

You can check for cycles in a connected component of a graph or connected graph as follows. Find a node which has only outgoing edges. If there is no such node, then there is a cycle. Start a DFS at that node. When traversing each edge, check whether the edge points back to a node already on your stack(already visited). This indicates the existence of a cycle. If you find no such edge, there are no cycles in that connected component(graph is acyclic).

BFS:

Its a traversing algorithm, where we start traversing from selected node (source or starting node) and traverse the graph layerwise which means it explores the neighbour nodes (nodes which are directly connected to source node) and then move towards the next level neighbour nodes. As the name suggests, we move in breadth of the graph, i.e., we move horizontally first and visit all the nodes of the current layer and then we move to the next

Implement using queue.



**Applications of BFS:**

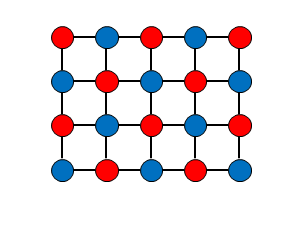
**1) Find level of a node in tree**

As we know in BFS, we traverse level wise (starting from a source node , in this case root node), i.e first we visit all the nodes of one level and then visit to nodes of another level. We can use BFS to determine level of each node (as shown in above gif).

**2) 0-1 BFS**

In this BFS 1 is used to show that two nodes are connected and 0 shows otherwise. No weight/cost is given to edges. This can be used to find shortest distance between two nodes in an unweighted graph. This is done by moving away from source node one level at a time in all directions (this is nothing but normal BFS traversal using queue) and stopping when destination node is encountered, the number of levels traversed will be the shortest distance.

Bipartite Graph



Graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U.

**Some standard algorithms:**

Dijkstra's algorithm (Single-source shortest paths)

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph.

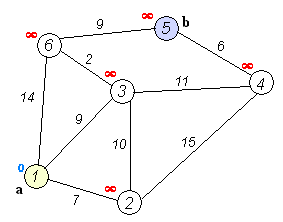
In Dijkstra's algorithm we pick the unvisited vertex with the lowest distance, calculate the distance through it to each unvisited neighbor, and update the neighbor's distance if smaller. Mark visited when done with neighbors.

Drawback: The algorithm won't work for a graph with negative edges. This is because once a shortest path is set for a destination node (marked visited) it should not be updated again but if there is a path with negative edge to the destination node then the path values have to be calculated again for visited nodes as it might be the new shortest path.

Example :

Source node : a

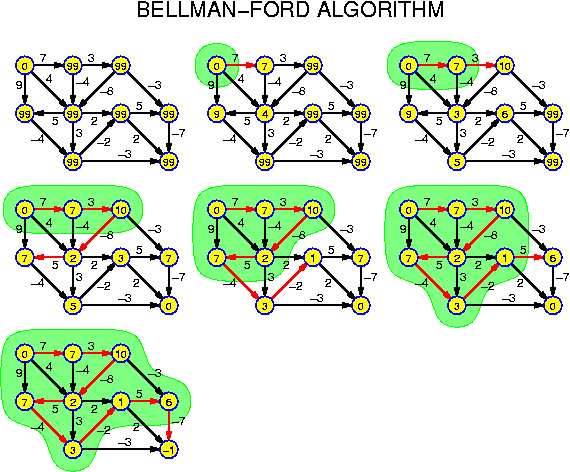
Destination node : b



Bellman-Ford's algorithm (Single-source shortest paths)

The Bellman–Ford algorithm is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.

Example :



Prim's algorithm (minimum spanning tree)

Prim's algorithm finds a minimum spanning tree (MST) for a connected weighted undirected graph. It is similar to Dijkstra's algorithm.

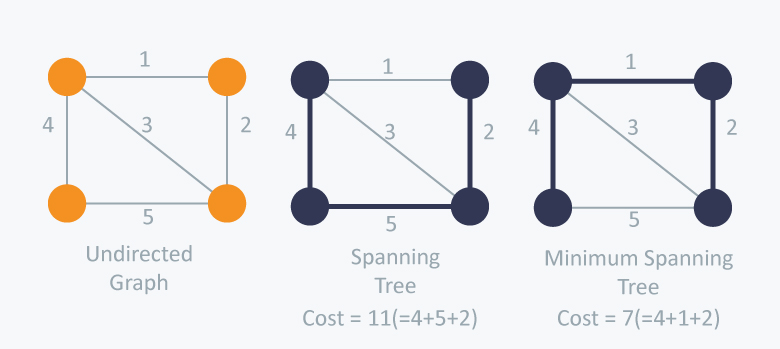
**What is a Spanning Tree?**

It is a tree that spans through a given graph G and connects every vertex of G. It is a subset of graph G.

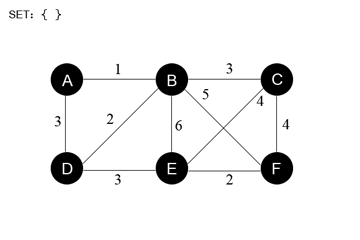
**What is a Minimum Spanning Tree?**

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:



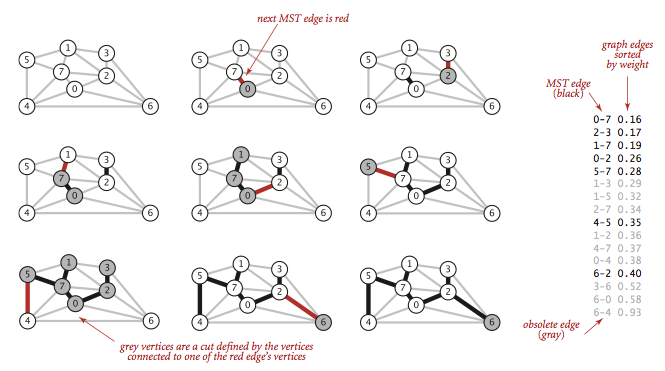
Example of Prim's algorithm :



Kruskal's algorithm (minimum spanning tree)

Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest.

Example :



Note: As it is a tree any cycle formation must be avoided.