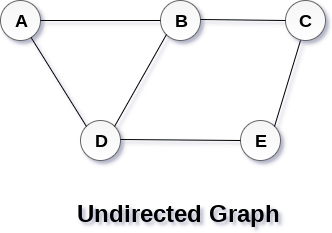
**Graph**

A graph can be defined as a group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having a parent relationship.

**Definition**

A graph G can be defined as an ordered set G (V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

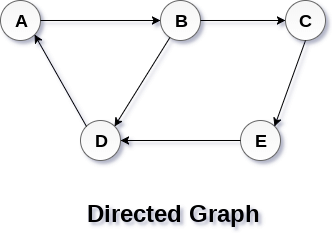
A Graph G (V, E) with 5 vertices (A, B, C, D, E) and six edges ((A, B), (B, C), (C, E), (E, D), (D, B), (D, A)) is shown in the following figure.



**Directed and Undirected Graph**

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

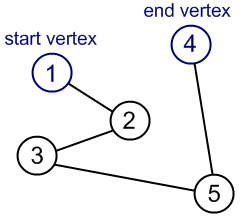
In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

A directed graph is shown in the following figure  


**Graph Terminology**

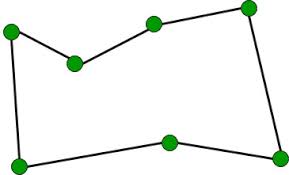
**Path**

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



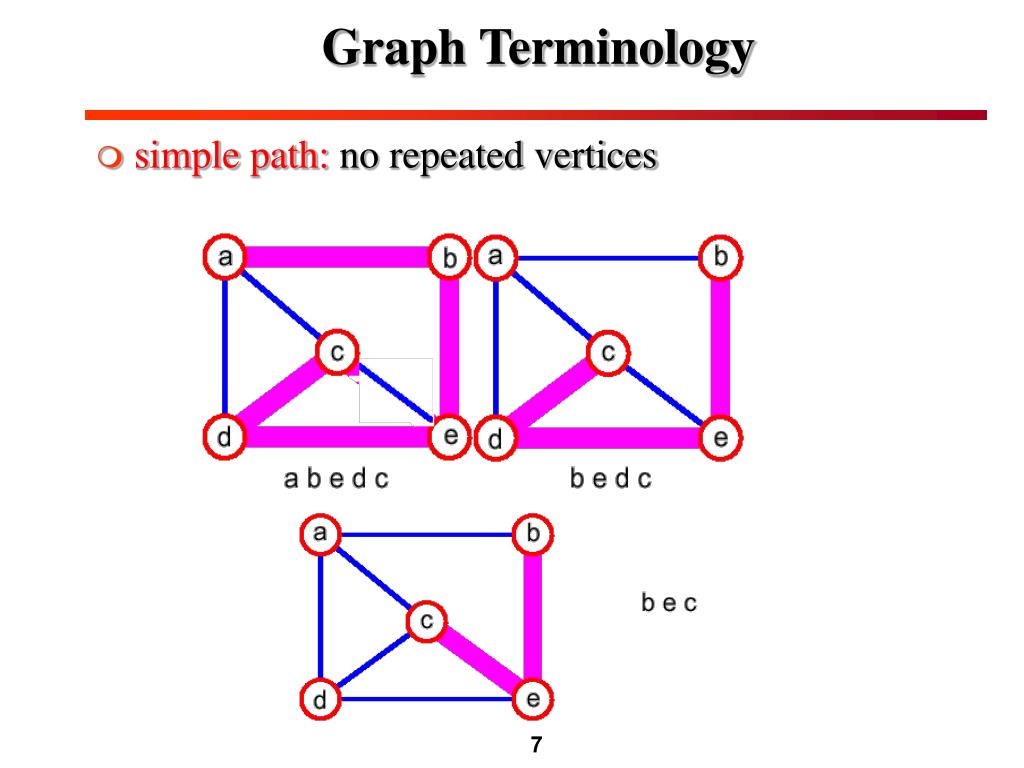
**Closed Path**

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if V0=VN.



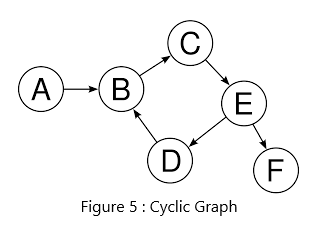
**Simple Path**

If all the nodes of the graph are distinct with an exception V0=VN, then such path P is called as closed simple path.



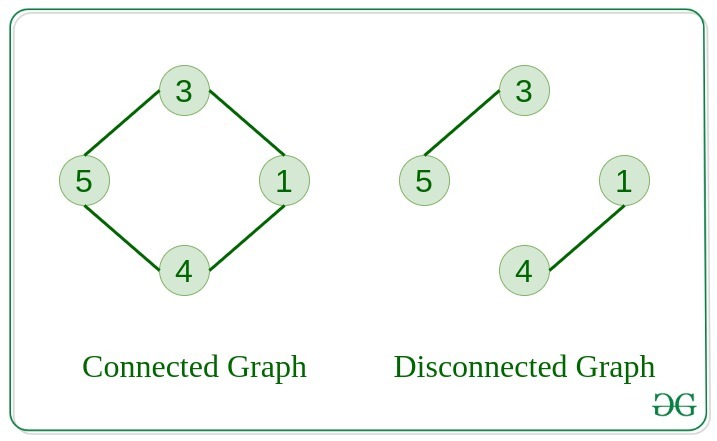
**Cycle**

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



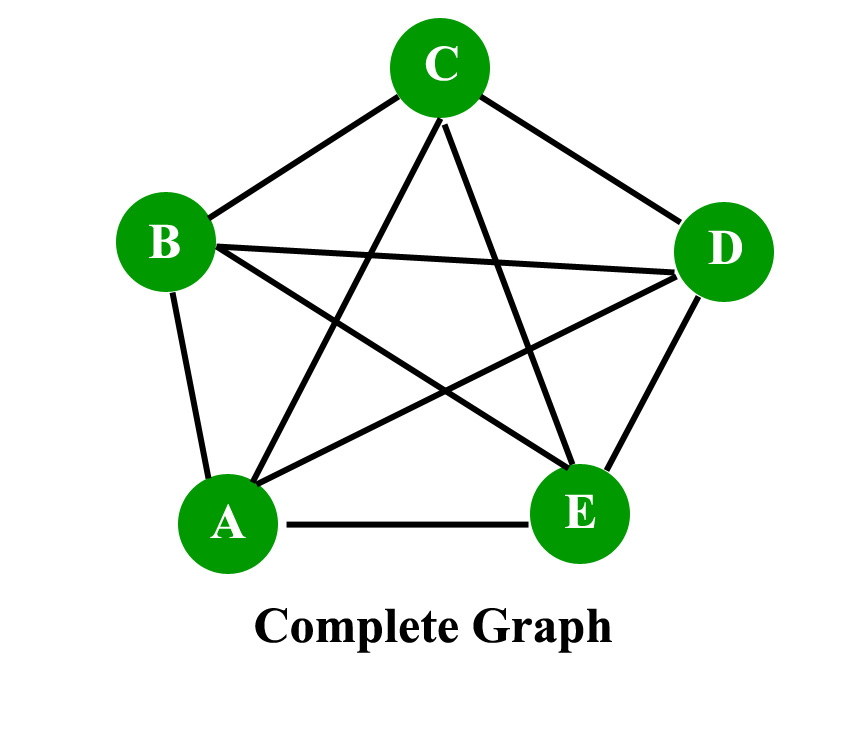
**Connected Graph**

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



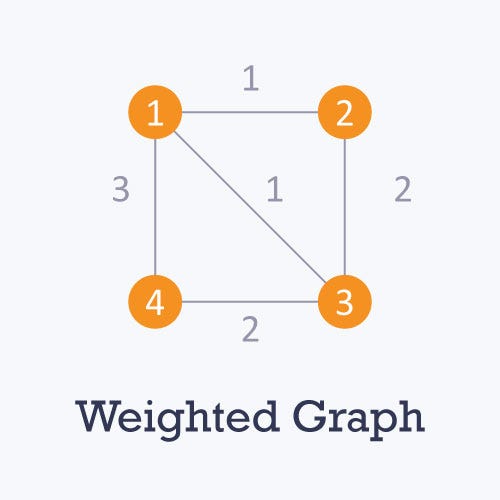
**Complete Graph**

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



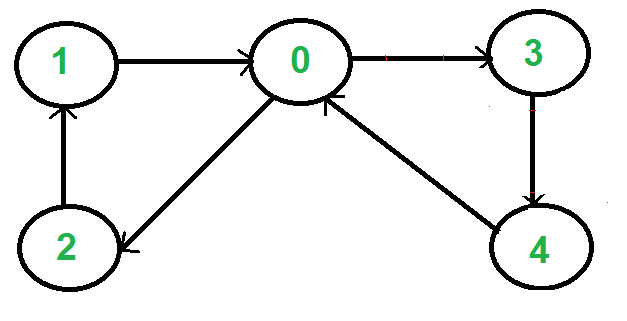
**Weighted Graph**

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



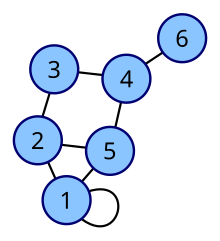
**Digraph**

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



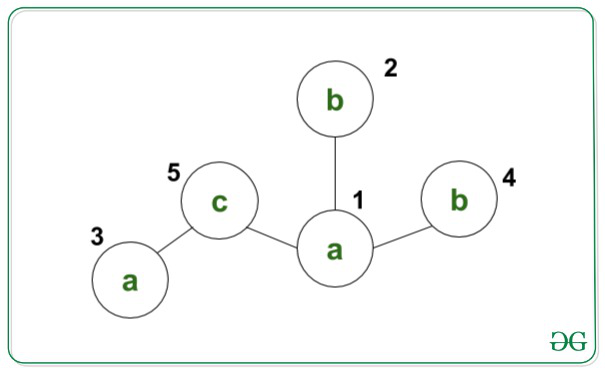
**Loop**

An edge that is associated with the similar endpoints can be called a Loop.



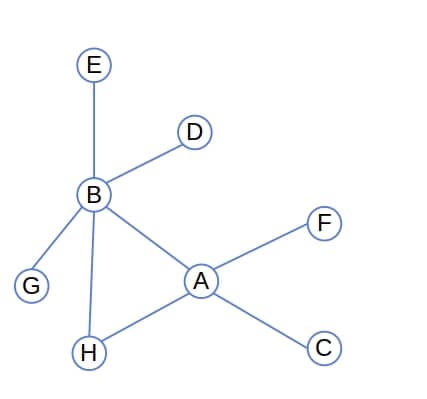
**Adjacent Nodes**

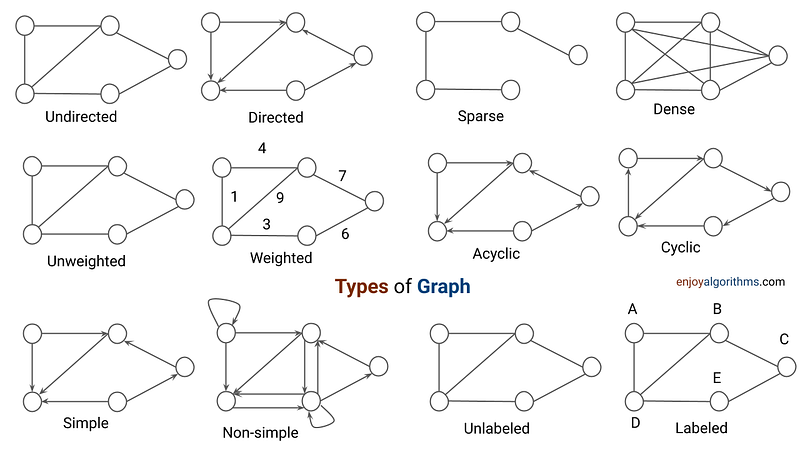
If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.



**Degree of the Node**

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





**Fundamentals of Graph Data Structure**

1. **Definition**:
   * A graph is a collection of nodes (vertices) and edges (arcs) that connect pairs of nodes.
2. **Types of Graphs**:
   * **Undirected Graph**: Edges have no direction.
   * **Directed Graph (Digraph)**: Edges have a direction.
   * **Weighted Graph**: Edges have weights (values).
3. **Terminology**:
   * **Vertex (Node)**: A fundamental unit of the graph.
   * **Edge (Link)**: A connection between two vertices.
   * **Degree**: Number of edges connected to a vertex.
     + **In-degree**: Number of incoming edges (directed graph).
     + **Out-degree**: Number of outgoing edges (directed graph).
   * **Path**: A sequence of vertices connected by edges.
   * **Cycle**: A path that starts and ends at the same vertex.
4. **Graph Representation**:
   * **Adjacency Matrix**: A 2D array where matrix[i][j] is 1 (or weight) if there's an edge from vertex i to vertex j, otherwise 0.
     + Pros: Fast edge lookup.
     + Cons: Space inefficient for sparse graphs.
   * **Adjacency List**: An array of lists. Each list contains all vertices adjacent to the vertex.
     + Pros: Space efficient.
     + Cons: Slower edge lookup compared to adjacency matrix.
5. **Graph Traversal**:
   * **Depth-First Search (DFS)**: Explores as far as possible along each branch before backtracking.
     + Uses: Finding connected components, topological sorting, cycle detection.
   * **Breadth-First Search (BFS)**: Explores all neighbors at the present depth prior to moving on to nodes at the next depth level.
     + Uses: Shortest path in unweighted graphs, level-order traversal.
6. **Special Graphs**:
   * **Tree**: A connected acyclic undirected graph.
   * **Binary Tree**: A tree where each node has at most two children.
   * **Binary Search Tree (BST)**: A binary tree where for each node, the left subtree has smaller values, and the right subtree has larger values.
7. **Common Problems and Algorithms**:
   * **Shortest Path**:
     + **Dijkstra's Algorithm**: For graphs with non-negative weights.
     + **Bellman-Ford Algorithm**: For graphs with negative weights (handles negative cycles).
     + **Floyd-Warshall Algorithm**: For all-pairs shortest paths.
   * **Minimum Spanning Tree (MST)**:
     + **Kruskal's Algorithm**: Uses a union-find structure.
     + **Prim's Algorithm**: Greedy algorithm.
   * **Cycle Detection**:
     + DFS for directed and undirected graphs.
   * **Topological Sorting**:
     + Ordering of vertices in a Directed Acyclic Graph (DAG).
8. **Complexity Considerations**:
   * **Time Complexity**:
     + Adjacency Matrix: O(V2)O(V^2)O(V2) for traversal.
     + Adjacency List: O(V+E)O(V + E)O(V+E) for traversal.
   * **Space Complexity**:
     + Adjacency Matrix: O(V2)O(V^2)O(V2).
     + Adjacency List: O(V+E)O(V + E)O(V+E).

Understanding these fundamentals is essential for solving graph-related problems in data structures and algorithms.

**Graph Representation Basics**

A **graph** consists of vertices (nodes) and edges (connections between nodes). There are two common ways to represent a graph in programming:

1. **Adjacency List**: A list where each node points to a list of its neighbors.
2. **Adjacency Matrix**: A 2D array (matrix) where rows and columns represent nodes. The value at matrix[i][j] indicates whether there is an edge from node i to node j.

**Adjacency Matrix**

An adjacency matrix is a square matrix used to represent a finite graph. Here's how it works:

* The matrix is a 2D array matrix where the element matrix[i][j] is:
  + 1 if there is an edge from node i to node j
  + 0 if there is no edge from node i to node j

A -- B

| |

C -- D

The adjacency matrix for this graph would be :

A B C D

A 0 1 1 0

B 1 0 0 1

C 1 0 0 1

D 0 1 1 0