Topics to cover:

1. Graph in Java

1. Graph

In Java, the **Graph** is a data structure that stores a certain type of data. The concept of the **graph** has been stolen from the mathematics that fulfills the need of the computer science field. It represents a network that connects multiple points to each other. In this section, we will learn Java Graph data structure in detail. Also, we will learn the **types of Graph**, their implementation, and **traversal** over the graph.

In other words, A **graph** is a data structure that stores connected data. In other words, a graph G (or g) is defined as a set of vertices (V) and edges (E) that connects vertices. The examples of graph are a social media network, computer network, Google Maps, etc.

Each graph consists of **edges** and **vertices** (also called nodes). Each vertex and edge have a relation. Where vertex represents the data and edge represents the relation between them. Vertex is denoted by a circle with a label on them. Edges are denoted by a line that connects nodes (vertices).

Graph Terminology

Vertex: Vertices are the point that joints edges. It represents the data. It is also known as a node. It is denoted by a circle and it must be labeled. To construct a graph there must be at least a node. For example, house, bus stop, etc.

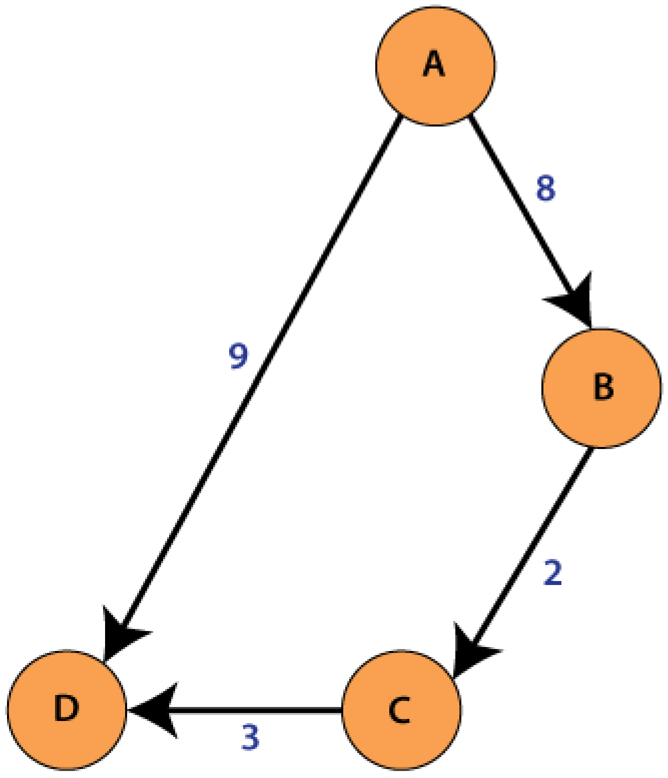
Edge: An edge is a line that connects two vertices. It represents the relation between the vertices. Edges are denoted by a line. For example, a path to the bus stop from your house.

Weight: It is labeled to edge. For example, the distance between two cities is 100 km, then the distance is called weight for the edge.

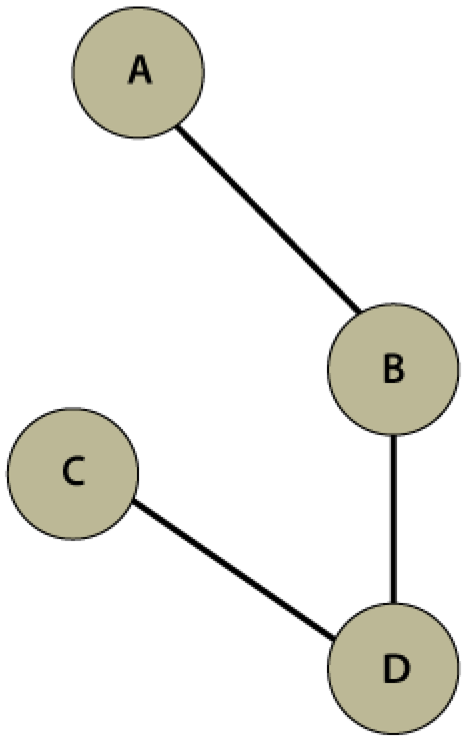
Path: The path is a way to reach a destination from the initial point in a sequence.

Types of Graph

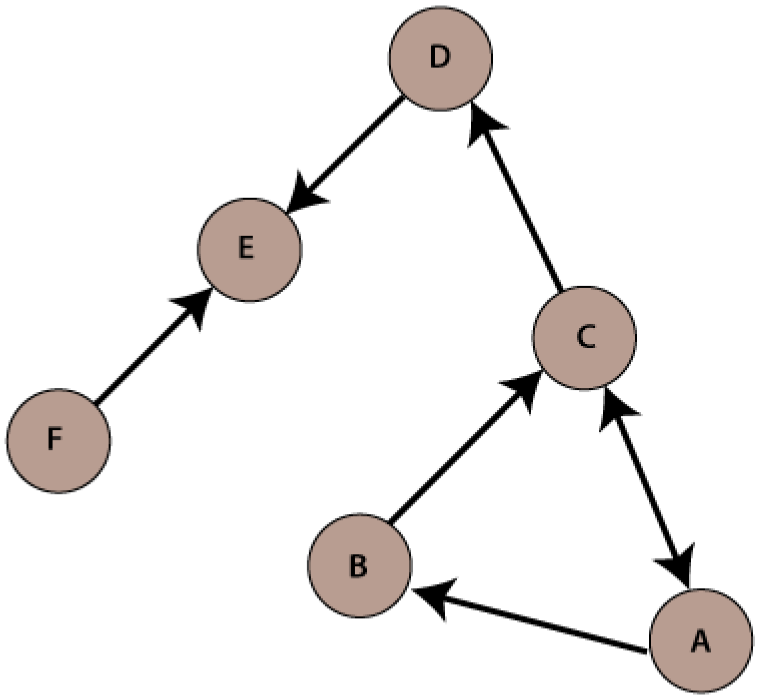
Weighted Graph: In a weighted graph, each edge contains some **data** (weight) such as distance, weight, height, etc. It is denoted as w(e). It is used to calculate the cost of traversing from one vertex to another. The following figure represents a weighted graph.



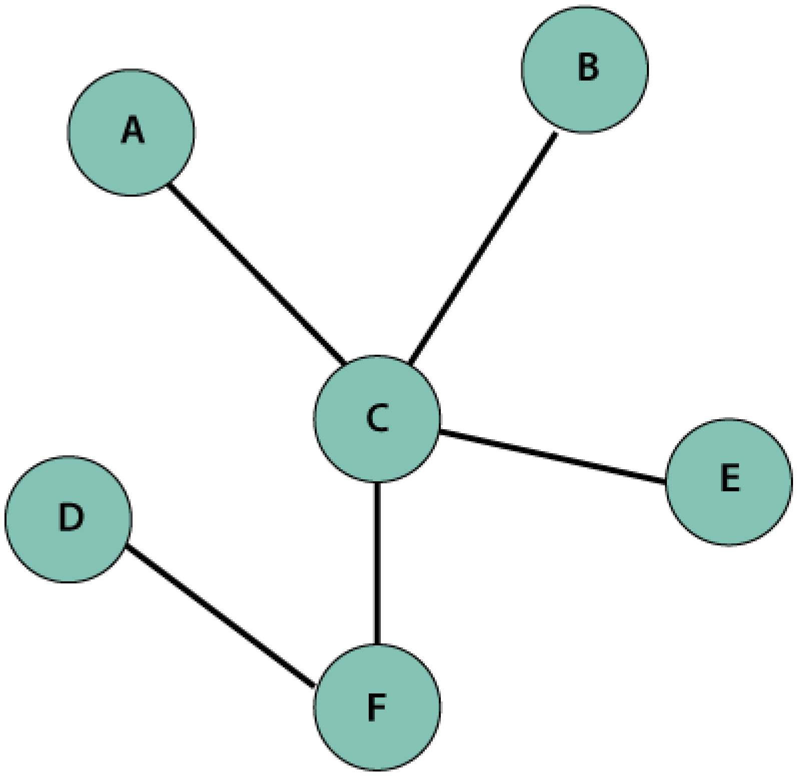
Unweighted Graph: A graph in which edges are not associated with any value is called an unweighted graph. The following figure represents an unweighted graph.



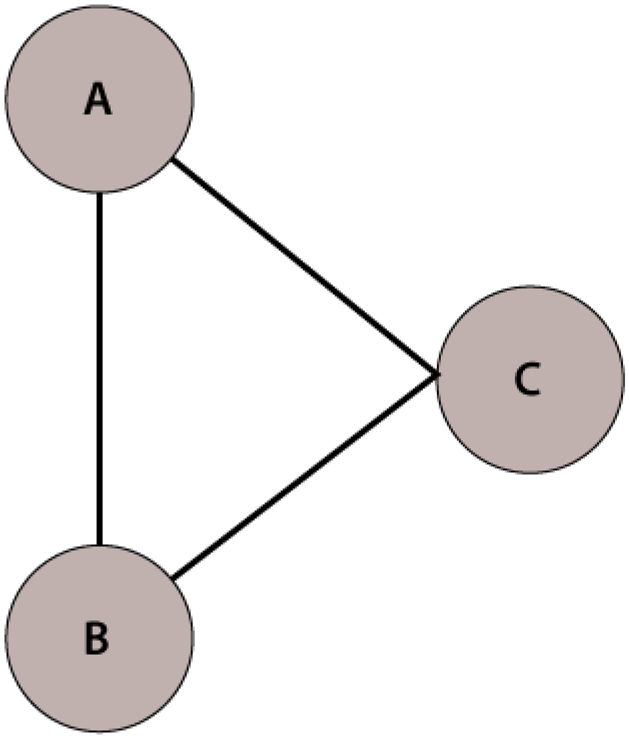
- **Directed Graph:** A graph in which edges represent direction is called a directed graph. In a directed graph, we use arrows instead of lines (edges). Direction denotes the way to reach from one node to another node. Note that in a directed graph, we can move either in one direction or in both directions. The following figure represents a directed graph.



- **Undirected Graph:** A graph in which edges are bidirectional is called an undirected graph. In an undirected graph, we can traverse in any direction. Note that we can use the same path for return through which we have traversed. While in the directed graph we cannot return from the same path.

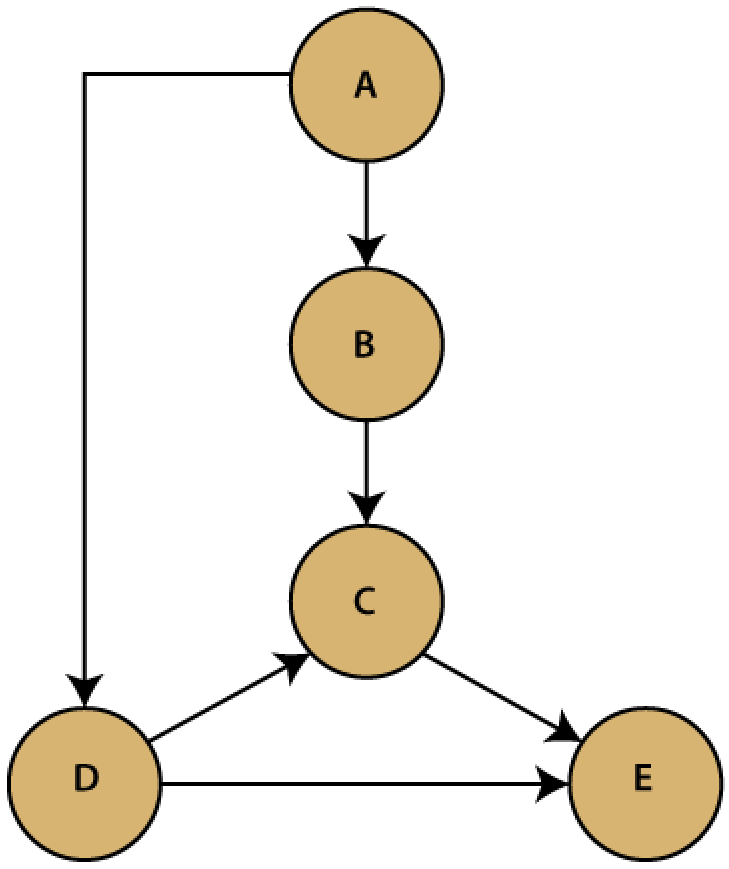


- **Connected Graph:** A graph is said to be connected if there exists at least one path between every pair of vertices. Note that a graph with only a vertex is a connected graph.

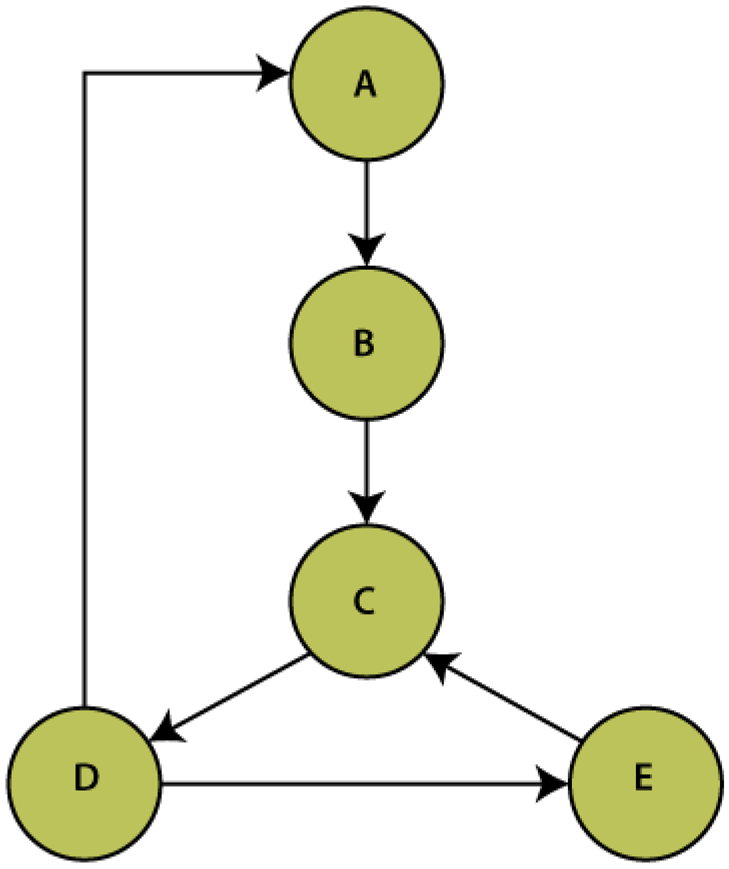


There are two types of connected graphs.

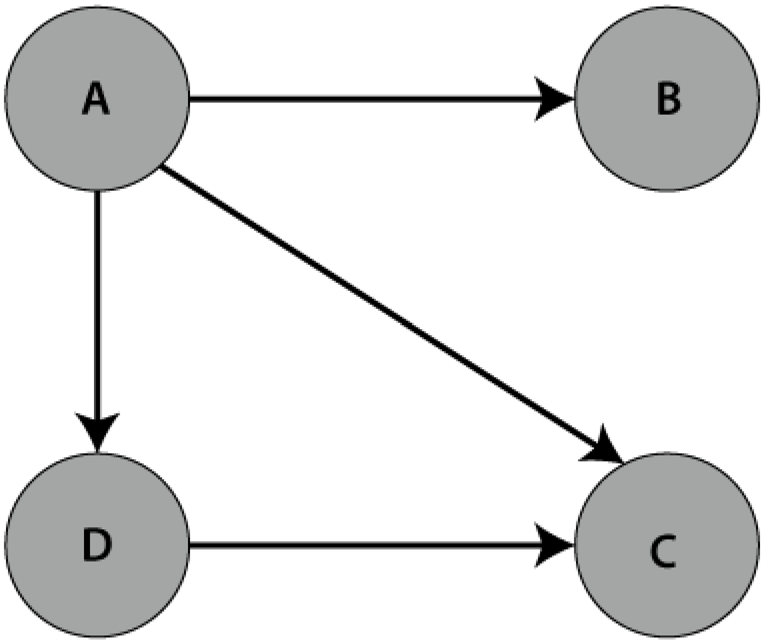
1. **Weekly Connected Graph:** A graph in which nodes cannot be visited by a single path is called a weekly connected graph.



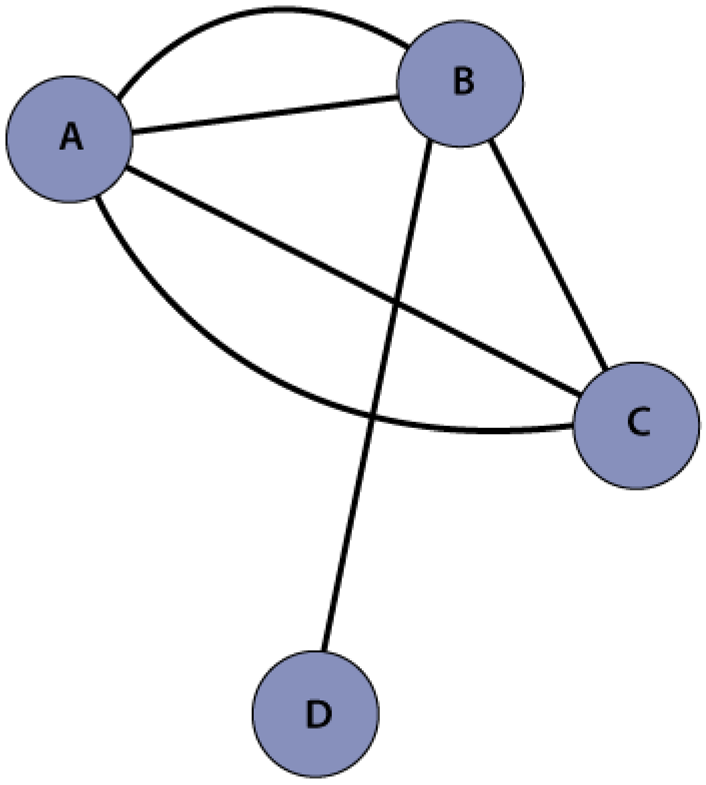
2. **Strongly Connected Graph:** A graph in which nodes can be visited by a single path is called a strongly connected graph.



- **Disconnected Graph:** A graph is said to be disconnected if there is no path between a pair of vertices is called a disconnected graph. A disconnected graph may consist of two or more connected graphs.

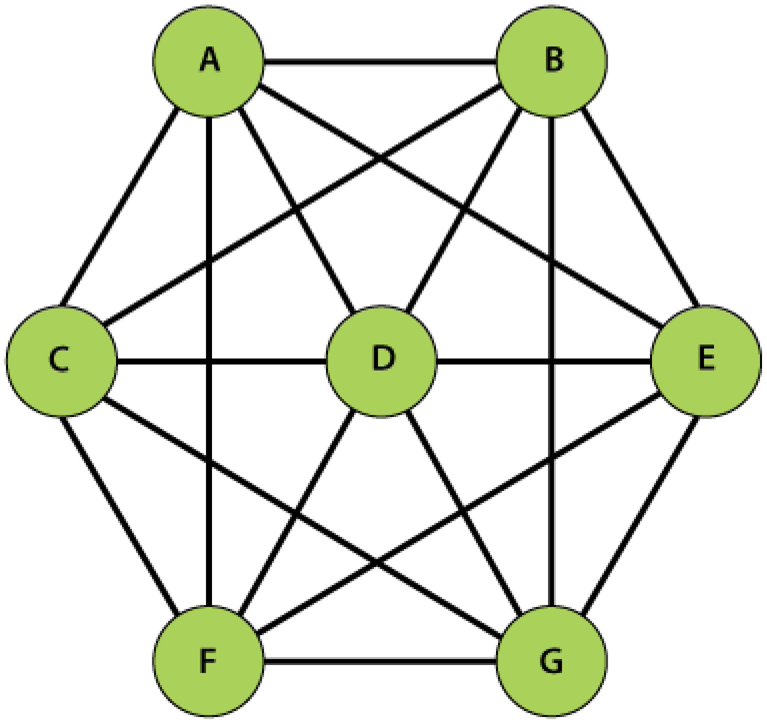


- **Multi Graph:** A graph that has multiple edges connecting the same pair of nodes. The following figure represents a multi-graph.



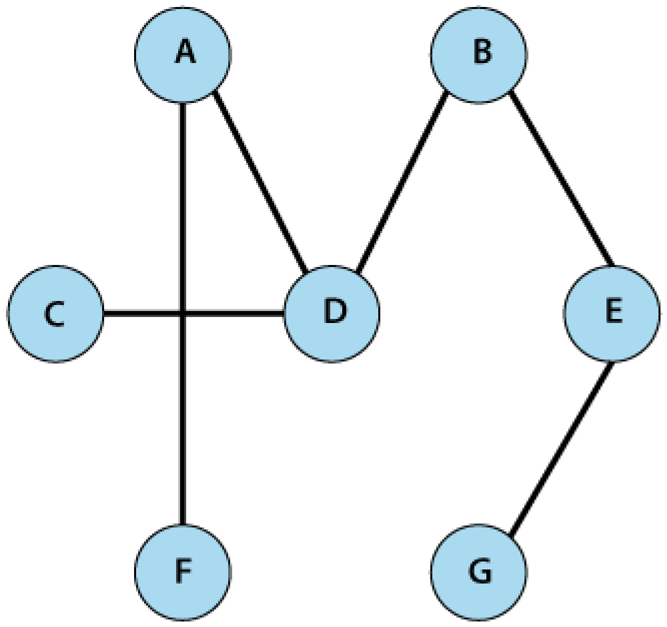
Multigraph

- **Dense Graph:** A graph in which the number of edges is close to the maximal number of edges, the graph is called the dense graph. The following figure represents a dense graph.



Dense

- **Sparse Graph:** A graph in which the number of edges is close to the minimal number of edges, the graph is called the sparse graph. It can be a disconnected graph. The following figure represents a sparse graph.



Sparse

Graph Implementation in Java

Following are most common data-structures for the implementation of graphs in Java.

- Adjacency list
- Adjacency matrix

Adjacency list:

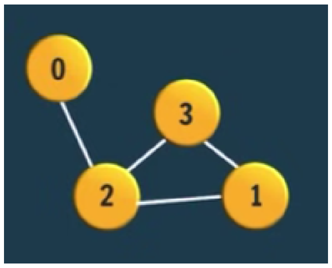
An adjacency list is a data structure used to represent a graph where each node in the graph stores a list of its neighboring vertices.

Adjacency matrix:

It is a connection matrix containing rows and columns used to represent a simple labelled graph.

In this tutorial, we will use Adjacency list which is probably the most popular way of creating graphs.

Example: Create a graph as follows



We can present the above graph vertex-wise through the list of edges as follows;

Vertex(source) -> {source, dest-1}, {source, dest-2}, ….. {source, dest-n}

0 -> {0, 2}

1 -> {1, 2}, {1, 3}

2 -> {2, 0}, {2, 1}, {2, 3}

3 -> {3, 1}, {3, 2}

Program 1: From the above edge information, write a program to create an undirected graph. Show the neighbours of any vertex.

```
import java.util.*;

public class Graph2 {
        static class Edge{
                int src,dst;

                Edge(int s, int d){
                        this.src=s;
                        this.dst=d;
                }
        }

        public static void createGraph(ArrayList<Edge> graph[]) {
                for(int i=0;i<graph.length;i++) {
                        graph[i]=new ArrayList<Edge>();
                }
                graph[0].add(new Edge(0,2));
                graph[1].add(new Edge(1,2));
                graph[1].add(new Edge(1,3));
                graph[2].add(new Edge(2,0));
                graph[2].add(new Edge(2,1));
                graph[2].add(new Edge(2,3));
                graph[3].add(new Edge(3,1));
                graph[3].add(new Edge(3,2));
        }

        public static void main(String[] args) {
                int V=4;
                ArrayList<Edge> graph[] = new ArrayList[V];
                createGraph(graph);


                for(int i=0;i<graph[2].size();i++) {
                        Edge e = graph[2].get(i);
                        System.out.println(e.src+" "+e.dst);
                }
        }
}
```

Output :

```
2 0
2 1
2 3
```

In the above output, you can see the neighbours of vertex 2.