Introduction to Graphs

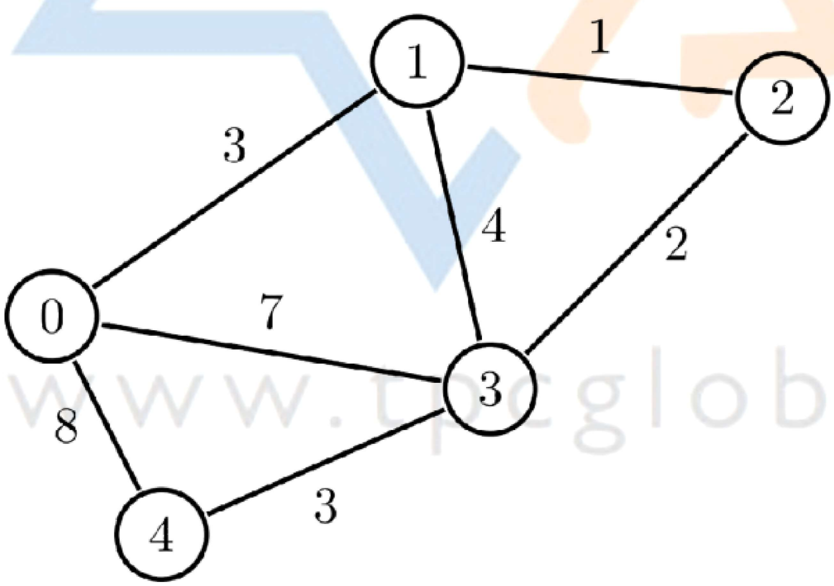
A **graph** is a non-linear data structure consisting of **nodes (vertices)** and **connections (edges)** between them. It is widely used to represent real-world systems like **networks (social, computer), maps, circuits,** etc.

A graph **G** is defined as:

G=(V,E)G = (V, E)G=(V,E)

Where:

- **V** is the set of vertices (nodes)
- **E** is the set of edges (connections between vertices)



Advantages of Graphs

Graphs offer several advantages, especially when modeling relationships or networks. Here are key benefits:

1. Efficient Representation of Complex Relationships

Graphs can represent many-to-many relationships (like users connected to multiple users, cities connected to multiple cities), which are difficult to handle with linear data structures.

2. Flexible Structure

Graphs can be **directed or undirected**, **weighted or unweighted**, **cyclic or acyclic**, allowing flexibility in modeling various problems.

3. Real-world Applicability

They closely mirror real-world systems such as transportation networks, social media, internet infrastructure, etc.

4. Support for Algorithms

Graphs support powerful algorithms like:

- Dijkstra's and Bellman-Ford (shortest path)
- Prim's and Kruskal's (minimum spanning tree)
- DFS, BFS (searching/traversing)
- Topological Sorting
- Cycle Detection, etc.

5. Data Navigation

Graphs allow **efficient navigation**, such as finding the shortest path, all reachable nodes, or detecting connectivity components.

6. Supports Both Static and Dynamic Scenarios

Graphs can be:

- **Static** (predefined structure like a map)
- **Dynamic** (where edges/vertices change, like a live social network)

7. Can Handle Sparse and Dense Connections

Using adjacency list or matrix, graphs can adapt based on data density:

- **Sparse graphs** (few edges): Adjacency List
- **Dense graphs** (many edges): Adjacency Matrix

Uses / Applications of Graphs

Graphs are used across **computer science**, **engineering**, **business**, **networking**, and **real life**.

1. Computer Networks

- Nodes = Computers/Routers
- Edges = Physical or logical connections
- Used for **routing**, **packet transfer**, and **network topology**

2. Social Networks

- Vertices = People
- Edges = Friendships, followers, messages
- Used in platforms like Facebook, Instagram, LinkedIn

3. Google Maps / GPS

- Vertices = Places (cities, intersections)
- Edges = Roads (with weights = distance or time)
- Algorithms: Dijkstra's for **shortest path**, A* for **heuristic path**

4. Web Crawlers

- Websites as nodes
- Hyperlinks as edges
- DFS/BFS used to crawl and index pages

5. Course Scheduling / Prerequisites

- Courses as nodes
- Edges show prerequisites
- **Topological Sort** is used for correct scheduling

6. Recommendation Engines

- Products/Users as nodes
- Connections show preferences or behaviors
- Graph algorithms detect similar users or items

7. Electric Circuits

- Components as vertices
- Connections as edges
- Used in **circuit simulation and analysis**

8. Airline Flight Systems

- Airports = Vertices
- Flights = Edges with weights like fare or time
- Helps in **route planning**, **minimum fare path**, etc.

9. Image Processing / Computer Vision

- Pixels or regions as vertices
- Edges connect similar regions
- Used in **segmentation**, **object detection**, etc.

10. Blockchain / Crypto

- Blocks/transactions as vertices
- Graphs used in **transaction mapping**, **dependency resolution**

Graph Terminology

Understanding **graph terminology** is essential before moving to algorithms.

🔹 1. Vertex (Node)

- A **vertex** is a point in the graph.
- Represents an object or entity (like a city, person, or computer).
- Example: In a social network graph, each person = one vertex.

Notation: `V = {v1, v2, v3, …, vn}`

🔹 2. Edge (Link)

- An **edge** connects two vertices.
- Represents a relationship between the two vertices.

Example:

- In a road map, an edge = road between two cities.
- In a social network, an edge = friendship/following.

Notation: `E = {(v1, v2), (v2, v3), …}`

◆ 3. Degree of a Vertex

The **degree** of a vertex = number of edges connected to it.

- **Undirected Graph:**
  Degree = number of incident edges.
- **Directed Graph (Digraph):**
  - **In-degree:** Number of incoming edges.
  - **Out-degree:** Number of outgoing edges.

Formula (Undirected Graph):

Sum of degrees of all vertices=2×Number of edges\text{Sum of degrees of all vertices} = 2 \times \text{Number of edges}Sum of degrees of all vertices=2×Number of edges

◆ 4. Path

- A **path** is a sequence of vertices connected by edges.
- Length of a path = number of edges in it.

Example: In graph A → B → C → D, the path length from A to D is 3.

◆ 5. Cycle

- A **cycle** is a path where the first and last vertex are the same, and no edge is repeated.
- Example: A → B → C → A.

◆ 6. Connected Graph

- **Connected Graph (Undirected):** Every vertex can be reached from any other vertex.
- **Disconnected Graph:** At least one vertex is not reachable.

Example:

```
Connected: A—B—C
Disconnected:  A—B   C
```

7. Connected Components

- A **connected component** is a set of vertices in which each vertex is reachable from the others.
- Example: In a disconnected graph, each isolated subgraph is a connected component.

8. Weighted vs. Unweighted Graph

- **Weighted Graph:** Each edge has a weight (e.g., distance, cost, time).
- **Unweighted Graph:** All edges are equal (weight = 1).

9. Directed vs. Undirected Graph

- **Directed Graph (Digraph):** Edges have direction (A → B).
- **Undirected Graph:** Edges don't have direction (A — B).

10. Subgraph

- A **subgraph** is a smaller part of a graph, containing some vertices and edges of the main graph.