Hello there, aspiring data structure experts!

We've explored linear structures like lists, stacks, and queues, and then branched out into the hierarchical world of trees. Trees are great for parent-child relationships, but what if the relationships are more complex? What if any item can be connected to any other item, without strict hierarchy?

Imagine a network of friends on social media, a map with cities and roads, or even the connections between web pages. These aren't neatly hierarchical. They are **networks** – and in DSA, we model these networks using **Graphs**!

🌐 What Exactly is a Graph in DSA?

A **Graph** is a non-linear data structure that consists of a finite set of entities called **nodes** (or **vertices**) and a set of connections between them called **edges** (or **links**).

Think of it as a collection of "dots" (nodes) and "lines" (edges) connecting some of those dots.

Common Analogies:

- **Social Networks:** People are nodes, friendships are edges.
- **Road Maps:** Cities are nodes, roads are edges.
- **Airline Routes:** Airports are nodes, flights are edges.
- **Web Pages:** Pages are nodes, hyperlinks are edges.
- **Electrical Circuits:** Components are nodes, wires are edges.

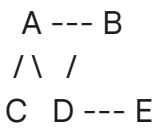
The Core Idea: Relationships Between Entities

Graphs are incredibly powerful because they allow us to model complex relationships where any item can be related to any other item, not just in a parent-child fashion.

🗣️ Graph Terminology: Speaking the Language of Networks

To understand graphs, we need to learn their specific vocabulary.

Let's imagine a simple graph for our definitions:



1. **Vertex (Node):** The fundamental entity in a graph. It represents an item or a

point.
    - In our example: **A, B, C, D, E** are vertices.
2. **Edge (Link/Arc):** A connection between two vertices. It represents a relationship.
    - In our example: (A, B), (A, C), (A, D), (B, D), (D, E) are edges.
3. **Adjacent Vertices (Neighbors):** Two vertices are adjacent if they are connected by an edge.
    - In our example: A is adjacent to B, C, and D. B is adjacent to A and D.
4. **Degree of a Vertex:** The number of edges connected to that vertex.
    - Degree of A = 3 (edges to B, C, D)
    - Degree of B = 2 (edges to A, D)
    - Degree of C = 1 (edge to A)
    - Degree of D = 3 (edges to A, B, E)
    - Degree of E = 1 (edge to D)
5. **Path:** A sequence of distinct vertices where each consecutive pair is connected by an edge.
    - In our example: A -> D -> E is a path. A -> B -> D -> C is NOT a path because B and C are not directly connected.
6. **Length of a Path:** The number of edges in the path.
    - Path A -> D -> E has length 2.
7. **Cycle:** A path that starts and ends at the same vertex, and has at least 3 vertices (to avoid trivial cycles like A-B-A).
    - In our example: A -> B -> D -> A is a cycle.
8. **Connected Graph:** A graph where there is a path between every pair of vertices.
    - Our example graph is connected. If we removed the edge (A,D) and (B,D), D and E would become disconnected from A, B, C.
9. **Disconnected Graph:** A graph that is not connected. It consists of two or more **connected components**.
10. **Loop (Self-loop):** An edge that connects a vertex to itself. (e.g., A to A).
11. **Parallel Edges (Multiple Edges):** More than one edge connecting the same pair of vertices.
12. **Simple Graph:** A graph that has no loops and no parallel edges. (Most graphs in DSA are assumed to be simple unless stated otherwise).

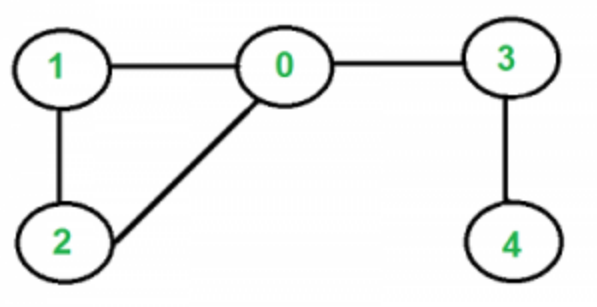
📊 Types of Graphs: Categorizing Connections

Graphs come in various flavors, each suited for different modeling needs.

1. Undirected Graph

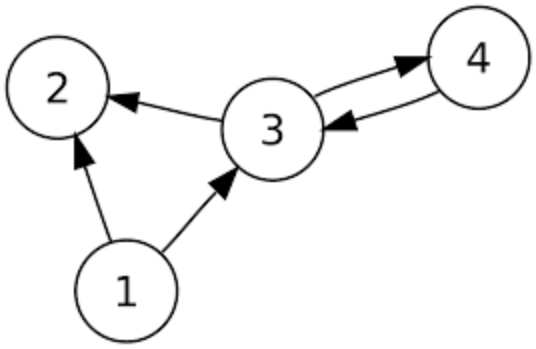
- **Concept:** Edges have no direction. If A is connected to B, then B is also connected to A. Think of a two-way street.

- **Notation:** Edges are represented as unordered pairs (u, v).
    - **Degree:** Simple count of edges connected.



2. Directed Graph (Digraph)

- **Concept:** Edges have a direction. If an edge goes from A to B, it doesn't necessarily mean there's an edge from B to A. Think of a one-way street.
- **Notation:** Edges are represented as ordered pairs (u, v), meaning an edge from u to v. u is the **tail**, v is the **head**.
- **In-degree:** Number of incoming edges to a vertex.
- **Out-degree:** Number of outgoing edges from a vertex.



3. Weighted Graph

- **Concept:** Each edge has an associated numerical value (a "weight" or "cost"). This weight can represent distance, time, cost, capacity, etc.
- Can be directed or undirected.

4. Unweighted Graph

- **Concept:** Edges have no associated weight. All edges are considered to have a weight of 1 or simply represent existence.

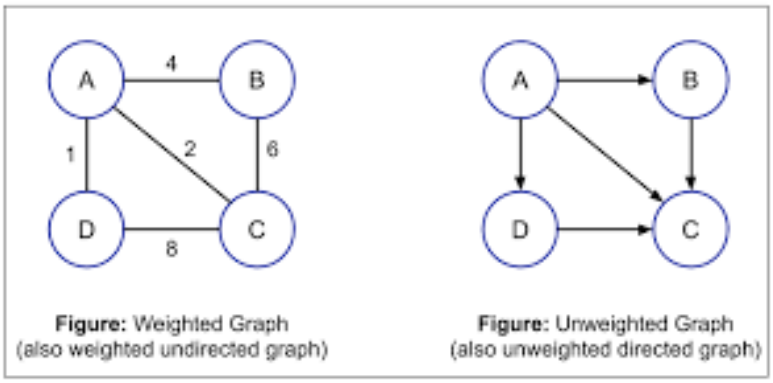


Figure: Weighted Graph
(also weighted undirected graph)

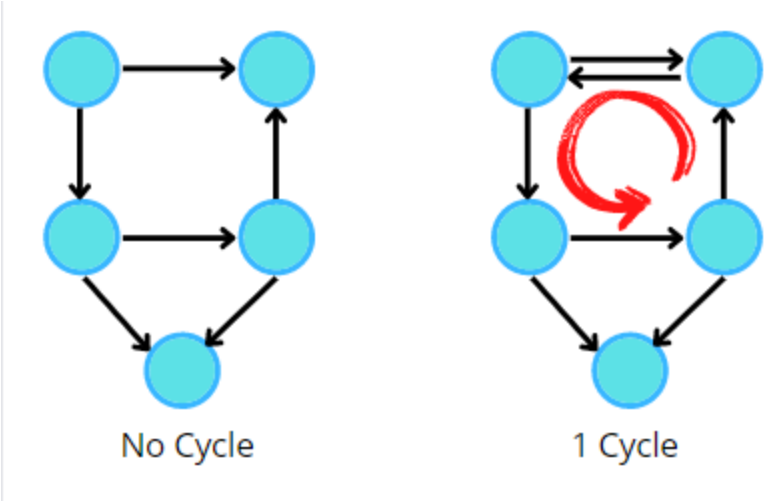
Figure: Unweighted Graph
(also unweighted directed graph)

5. Cyclic Graph

- **Concept:** Contains at least one cycle.

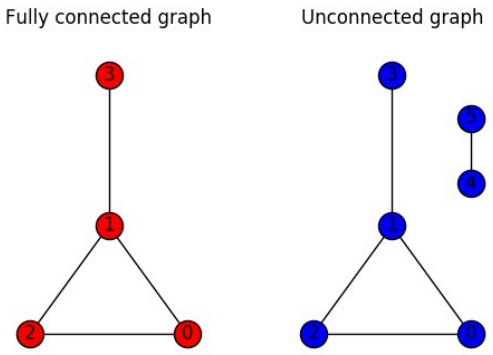
6. Acyclic Graph

- **Concept:** Contains no cycles.
- **Directed Acyclic Graph (DAG):** A very important type of directed graph with no cycles. Used for task scheduling, dependency resolution, etc.



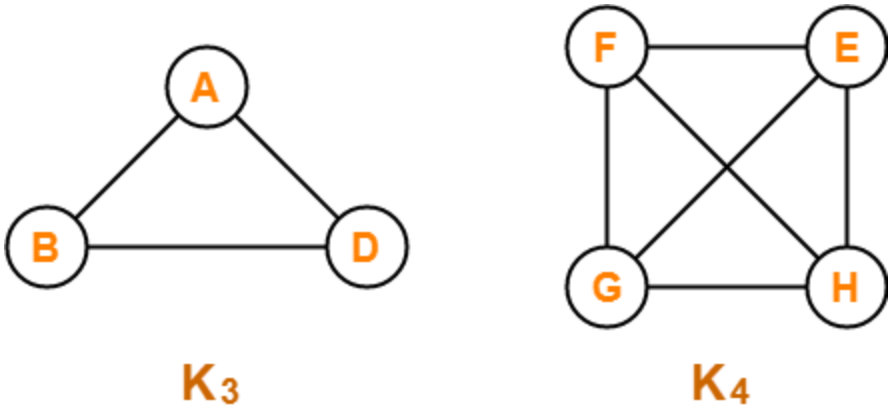
7. Connected Graph (for Undirected) / Strongly Connected (for Directed)

- **Connected (Undirected):** A path exists between every pair of vertices.
- **Strongly Connected (Directed):** A path exists from every vertex to every other vertex.
- **Weakly Connected (Directed):** The underlying undirected graph (ignoring edge directions) is connected.



8. Complete Graph

- **Concept:** Every distinct pair of vertices is connected by a unique edge.
- A complete graph with n vertices has n * (n-1) / 2 edges (for undirected).



Examples of Complete Graph

9. Sparse vs. Dense Graphs

- **Sparse Graph:** A graph with relatively few edges compared to the maximum possible number of edges.
- **Dense Graph:** A graph with many edges, close to the maximum possible.

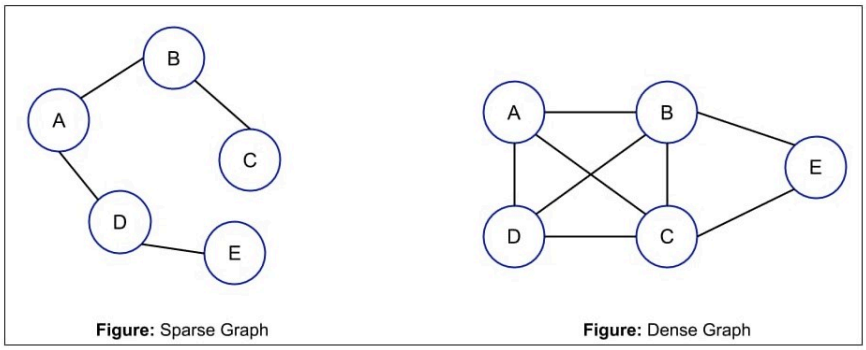


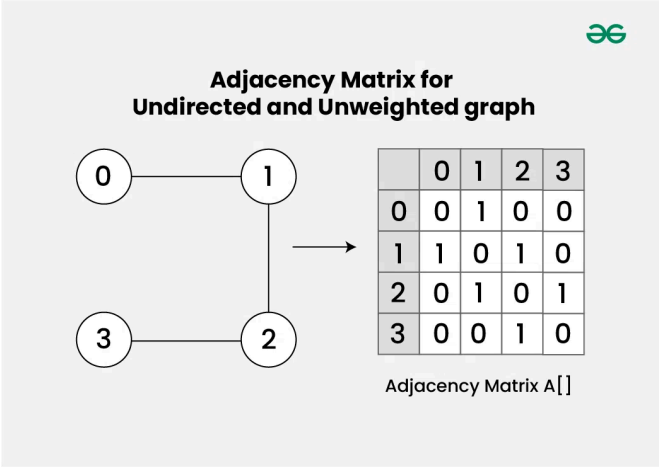
Figure: Sparse Graph **Figure:** Dense Graph

🗺️ Graph Representations: How to Store a Graph in Memory

Before we can perform operations, we need to decide how to store the graph's vertices and edges in a computer's memory. The two most common ways are:

1. Adjacency Matrix

- **Concept:** A 2D array (matrix) where rows and columns represent vertices.
  - matrix[i][j] = 1 (or true) if there's an edge from vertex i to vertex j.
  - matrix[i][j] = 0 (or false) if there's no edge.
  - For weighted graphs, matrix[i][j] would store the weight.
  - For undirected graphs, the matrix is symmetric (matrix[i][j] == matrix[j][i]).
- **Example (Undirected Graph):**



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

- **Advantages:**
  - **Fast isAdjacent(u, v) check:** O(1) (just look up matrix[u][v]).
  - Simple to implement.
  - Good for **dense graphs**.
- **Disadvantages:**
  - **Space Complexity:** O(V2) where V is the number of vertices. Can waste a lot

of space for sparse graphs (mostly zeros).
    - **Finding all neighbors of a vertex:** O(V) (must iterate through a whole row/column).
    - **Adding/Removing vertices:** O(V2) (requires resizing the whole matrix).

2. Adjacency List

- **Concept:** An array (or hash map) where each index (or key) represents a vertex. At each index, there's a list (e.g., LinkedList, ArrayList) of vertices that are adjacent to it.
    - For weighted graphs, the list would store pairs (neighbor, weight).
- **Advantages:**
    - **Space Complexity:** O(V+E) where E is the number of edges. Much more space-efficient for **sparse graphs**.
    - **Finding all neighbors of a vertex:** O(degree(V)) (just iterate through its list).
    - Good for most graph algorithms.
- **Disadvantages:**
    - **isAdjacent(u, v) check:** O(degree(u)) in worst case (must iterate through u's list).
    - Slightly more complex to implement than an adjacency matrix.

🚀 Basic Operations on a Graph

Regardless of representation, these are the fundamental things you do with a graph:

1. addVertex(vertex): Adding a New Vertex

- **Concept:** Introduce a new node to the graph.
- **Adjacency Matrix:** Add a new row and column (often involves resizing the matrix).
- **Adjacency List:** Add a new empty list for the new vertex.

2. removeVertex(vertex): Removing a Vertex

- **Concept:** Remove a node and all edges connected to it.
- **Adjacency Matrix:** Delete the row and column corresponding to the vertex.
- **Adjacency List:** Remove the vertex's list and remove all occurrences of that vertex from other lists.

3. addEdge(u, v): Adding an Edge

- **Concept:** Create a connection between two existing vertices.
- **Adjacency Matrix:** Set matrix[u][v] = 1 (and matrix[v][u] = 1 for undirected).
- **Adjacency List:** Add v to u's list (and u to v's list for undirected).

4. removeEdge(u, v): Removing an Edge

- **Concept:** Break the connection between two vertices.
- **Adjacency Matrix:** Set matrix[u][v] = 0 (and matrix[v][u] = 0 for undirected).
- **Adjacency List:** Remove v from u's list (and u from v's list for undirected).

5. isAdjacent(u, v): Checking for Adjacency

- **Concept:** Determine if an edge exists between two vertices.
- **Adjacency Matrix:** O(1) (direct lookup).
- **Adjacency List:** O(degree(u)) (iterate u's list).

6. Graph Traversal: Visiting Every Vertex

Just like trees, we need ways to systematically visit every vertex in a graph. Since graphs can have cycles, we need to be careful not to get stuck in an infinite loop and to visit each vertex only once. This usually involves keeping track of "visited" vertices.

a. Breadth-First Search (BFS)

- **Concept:** Explores the graph layer by layer (or level by level). It visits all neighbors of a starting vertex, then all their unvisited neighbors, and so on.
- **Analogy:** Ripples in a pond.
- **Data Structure Used: Queue**.
- **Time Complexity:** O(V+E) (for both Adjacency List and Matrix, but list is better for sparse graphs).

b. Depth-First Search (DFS)

- **Concept:** Explores as far as possible along each branch before backtracking. It goes deep into a path before exploring other branches.
- **Analogy:** Exploring a maze by always going forward until you hit a dead end, then backtracking.
- **Data Structure Used: Stack** (explicitly or implicitly via recursion).
- **Time Complexity:** O(V+E).