**📘 Graph – Data Structures & Algorithms (DSA)**

**🔹 What is a Graph?**

A **Graph** is a non-linear data structure consisting of:

* **Nodes (Vertices)**: Points in the graph
* **Edges**: Connections between the nodes

It can represent relationships, paths, and networks.

**🔹 Real-Life Examples of Graphs**

* Social networks (users as nodes, connections as edges)
* Google Maps (places as nodes, roads as edges)
* Web page links
* Airline routes
* Network of computers

**🔹 Types of Graphs**

| **Type** | **Description** |
| --- | --- |
| **Undirected** | Edges have no direction (bi-directional) |
| **Directed (Digraph)** | Edges have direction (from → to) |
| **Weighted** | Edges have a weight or cost |
| **Unweighted** | Edges are equal (no weight) |
| **Cyclic** | Contains at least one cycle |
| **Acyclic** | No cycles |
| **Connected** | Path exists between all pairs of nodes |
| **Disconnected** | Not all nodes are connected |

**🔹 Graph Representation in Python**

**✅ 1. Adjacency List (Most common)**

graph = {

"A": ["B", "C"],

"B": ["A", "D"],

"C": ["A", "D"],

"D": ["B", "C"]

}

**✅ 2. Adjacency Matrix**

# For nodes A, B, C

# A B C

# 0 1 1 ← A connected to B, C

# 1 0 0 ← B connected to A

# 1 0 0 ← C connected to A

matrix = [

[0, 1, 1],

[1, 0, 0],

[1, 0, 0]

]

**🔹 Graph Traversal Algorithms**

**✅ 1. Breadth First Search (BFS) – Level-wise (Queue)**

from collections import deque

def bfs(graph, start):

visited = set()

queue = deque([start])

while queue:

vertex = queue.popleft()

if vertex not in visited:

print(vertex, end=" ")

visited.add(vertex)

for neighbor in graph[vertex]:

if neighbor not in visited:

queue.append(neighbor)

**✅ 2. Depth First Search (DFS) – Path-wise (Stack / Recursion)**

def dfs(graph, vertex, visited=None):

if visited is None:

visited = set()

print(vertex, end=" ")

visited.add(vertex)

for neighbor in graph[vertex]:

if neighbor not in visited:

dfs(graph, neighbor, visited)

**🔹 Example Graph**

graph = {

'A': ['B', 'C'],

'B': ['A', 'D', 'E'],

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

'F': ['C', 'E']

}

**Output:**

bfs(graph, 'A') # A B C D E F

dfs(graph, 'A') # A B D E F C

**🔹 Applications of Graphs**

✅ Graphs are used in:

* Social networks (friends/followers)
* GPS navigation (shortest path)
* Web crawling
* Computer networks
* Scheduling & dependency resolution
* Game AI/pathfinding
* Knowledge graphs

**🔹 Advanced Graph Concepts**

| **Concept** | **Description** |
| --- | --- |
| Topological Sort | Sorting of DAG (Directed Acyclic Graph) |
| Dijkstra's Algorithm | Shortest path from one source (weighted graph) |
| Bellman-Ford Algorithm | Handles negative weights too |
| Floyd-Warshall Algorithm | All-pairs shortest path |
| Minimum Spanning Tree (MST) | Connect all nodes with min weight (Prim/Kruskal) |
| Cycle Detection | DFS with back edge detection or Union-Find |
| Disjoint Set (Union-Find) | For cycle detection & Kruskal’s MST |

**🔹 Time Complexity Summary**

| **Operation** | **Adjacency List** | **Adjacency Matrix** |
| --- | --- | --- |
| Storage | O(V + E) | O(V²) |
| Add Edge | O(1) | O(1) |
| Remove Edge | O(E) | O(1) |
| Check Edge Existence | O(E) | O(1) |
| Traverse Neighbors | O(V) | O(V) |

V = vertices, E = edges