📍

Graphs

📘 Graphs & Traversal — Exam Notes

1. **What is a Graph?**

- A **graph** is a pair G=(V,E)G = (V, E)G=(V,E) where:
  - VVV = set of vertices (nodes)
  - EEE = set of edges (connections between vertices)
- Graphs model networks: computers, transport, social links, logic, etc.

2. **Types of Graphs**

- **Undirected Graph**:
  - Edges have no direction: {u,v}={v,u}\{u, v\} = \{v, u\}{u,v}={v,u}
  - Example: friendships (A–B is same as B–A)
  - ![Undirected Example: {A,B}, {A,C}, {B,C}, {C,D}]lecture06_graphs
- **Directed Graph (Digraph)**:
  - Edges are ordered pairs: (u,v)≠(v,u)(u, v) \neq (v, u)(u,v)=(v,u)
  - Example: Twitter follows (A → B doesn't mean B → A)
- **Weighted Graph**:
  - Each edge has a weight (cost, distance, time, etc.).
  - Example: Road network where edges store distances.

3. **Graph Representations**

A. **Edge Set / Map**

- Store edges explicitly. With weights → use a map.

```
V = {A, B, C, D}
E = { (A,B):0, (B,C):2, (C,A):-1, (C,B):5, (C,D):4 }
```

- **Complexity**:
  - Memory: O(V+E)O(V + E)O(V+E)
  - Easy to list vertices/edges.
  - Listing adjacents: O(E)O(E)O(E).

B. **Adjacency Matrix**

- 2D array, entry [i][j] = weight if edge exists.

```
  A  B  C  D
A [ -, 0, -, - ]
B [ -, -, 2, - ]
C [ -1, 5, -, 4 ]
D [ -, -, -, - ]
```

- **Complexity**:
  - Find weight: O(1)O(1)O(1) (direct lookup).
  - Add/edit edge: O(1)O(1)O(1).
  - List adjacents: O(V)O(V)O(V) (must scan row).
  - Memory: O(V2)O(V^2)O(V2) (bad for sparse graphs).

C. **Adjacency List**

- Each vertex has a list of its neighbors.

```
A: [(B,0)]
B: [(C,2)]
C: [(A,-1), (B,5), (D,4)]
D: []
```

- **Complexity**:
  - List outgoing: O(degree(v))O(\text{degree}(v))O(degree(v)).
  - Find edge: O(degree(v))O(\text{degree}(v))O(degree(v)).
  - Add edge: O(1)O(1)O(1).
  - Memory: O(V+E)O(V+E)O(V+E).
  - Very good for sparse graphs.

D. **Adjacency Map**

- Like adjacency list, but neighbors stored in a map.

```
A: {B:0}
B: {C:2}
C: {A:-1, B:5, D:4}
D: {}
```

- **Complexity**:
  - Find edge: O(1)O(1)O(1) (map lookup).
  - Outgoing neighbors: O(degree(v))O(\text{degree}(v))O(degree(v)).
  - Memory: O(V+E)O(V+E)O(V+E).
  - Incoming neighbors: costly (O(V)O(V)O(V)) unless transpose stored.

---

4. **Graph Structures**

- **Path Graph**: sequence of vertices where edges connect consecutively.
    - Exactly 2 vertices have degree 1, rest degree 2.
- **Directed Path Graph**: same, but edges have direction.
- **Path Length**:
    - Unweighted: number of edges.
    - Weighted: sum of weights.
- **Spanning Tree**:
    - Subgraph that is a tree and contains all vertices.
    - Used in traversal (BFS/DFS build spanning trees).

---

5. **Graph Traversals**

A. **Breadth-First Search (BFS)**

- **General Idea**:
    - Explore level by level.
    - Finds **shortest path** (unweighted).
- **Process (lower-level)**:
    1. Start at root, mark it seen.
    2. Put root into a **queue**.
    3. Pop from queue → visit.
    4. For each unvisited neighbor: mark seen, push to queue.
    5. Repeat until queue empty.
- **Complexity**: O(V+E)O(V + E)O(V+E).
- **When to use**: shortest path, level-order traversal, connectivity check.

B. **Depth-First Search (DFS)**

- **General Idea**:

- Explore as deep as possible before backtracking.
- Useful for **cycle detection, topological sort, backtracking problems**.

- **Process (lower-level)**:
  1. Start at root, push it onto **stack**.
  2. Pop top of stack.
  3. If not visited → mark visited.
  4. Push all neighbors (even duplicates).
  5. Continue until stack empty.
- **Complexity**: O(V+E)O(V + E)O(V+E).
- **When to use**: explore paths, detect cycles, topo sort, puzzles.

C. **Preorder vs Postorder (DFS Variants)**

- **Preorder**: Visit node **before** children (root → children → ...).
- **Postorder**: Visit node **after** children (children → root).
- Example in DFS:
  - Preorder: push node, process immediately.
  - Postorder: push node, process only after exploring its neighbors.

✅ **Summary of Purpose**:

- **BFS** → shortest paths, level exploration.
- **DFS** → explore fully, analyze structure, cycle detection, backtracking.

BFS vs DFS in Reachability

- If you only need to know *which vertices are reachable*, **BFS and DFS are interchangeable**.
- If you care about *how far away* a vertex is (fewest edges) → only **BFS** works.
- If you care about *path existence or structure* → **DFS** is usually better.

🌳 Tree vs 🔄 DAG (Directed Acyclic Graph)

Feature	Tree (rooted, directed downward)	DAG (general)
Definition	Connected, acyclic graph	Directed graph with no directed cycles
Connectivity	Always connected	May be disconnected
Roots / Sources	Exactly **1 root**	Can have **multiple sources**
Parents per node	Every node (except root) has **1 parent**	Nodes may have **multiple parents**
Edges	Exactly V−1V-1V−1	Between 0 and V(V−1)2\frac{V(V-1)}{2}2V(V−1)
Paths	Exactly 1 unique path between any two nodes	May have multiple paths between nodes
Special case?	A tree is a special case of a DAG	Not every DAG is a tree

---

✅ Key takeaway

- **Every tree is a DAG** (if you direct edges from parent → child).
- **Not every DAG is a tree** (because DAGs allow multiple parents, multiple roots, may be disconnected, and can have many more edges).