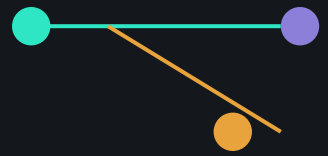


LECTURE 25

Graphs and Representation

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

LECTURE 25

Agenda

1. Core graph terminology: vertices, edges, directed/undirected, weighted/unweighted
2. Degree, path, cycle, and connectivity
3. Directed graphs in practice: in-degree vs. out-degree
4. Adjacency matrix vs. adjacency list representation
5. Converting between representations, and why sparsity matters

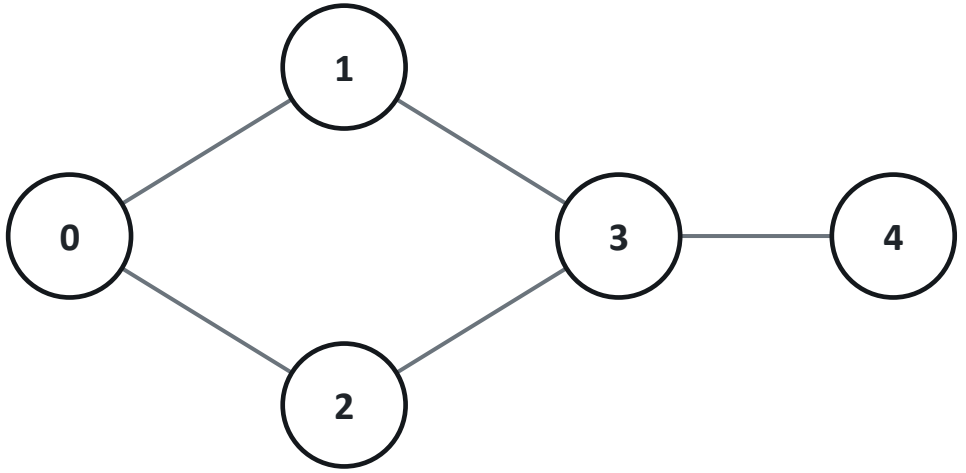
CORE CONCEPTS

What Is a Graph?

A graph G = (V, E) is a set of **vertices** connected by **edges** — no restriction on cycles or a single parent, unlike a tree.

Vertex (node)	A single point in the graph
Edge	A connection between two vertices
Degree	Number of edges touching a vertex
Path	A sequence of edges vertex-to-vertex
Cycle	A path that returns to its start
Connected graph	Every vertex can reach every other

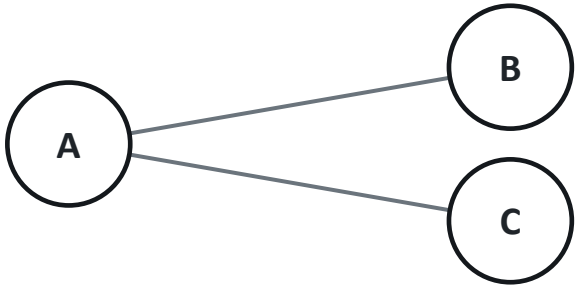
Undirected graph — Lectures 25-28's running example



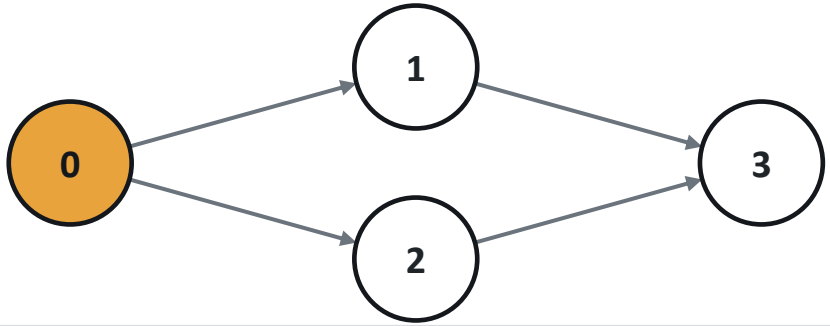
TYPES OF GRAPHS

Directed, Undirected, Weighted, Unweighted

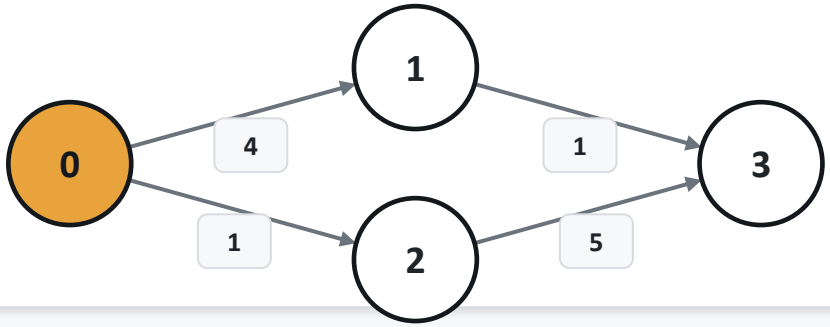
Undirected



Directed (DAG) — task dependencies



Weighted + directed — task durations (hrs)



Directed / undirected and weighted / unweighted are independent choices — four combinations total.

A road map with one-way streets + distances is directed AND weighted. A friendship graph is typically undirected and unweighted.

DIRECTED GRAPHS IN PRACTICE

Directed Edges: In-Degree vs. Out-Degree

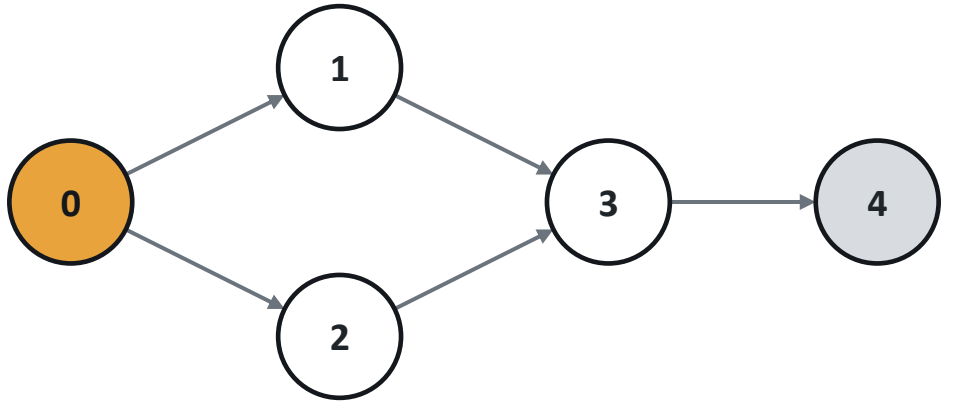
A directed addEdge(u, v) writes only into adjList[u] — "u points to v" says nothing about v pointing back.

directed_graph.cpp — excerpt

```
void addEdge(int u, int v) {
  adjList[u].push_back(v);
  only u -> v
}

// in-degree can't be read off adjList[i] --
// scan every OTHER vertex's list for i.
void printDegrees() const {
  vector<int> inDegree(numVertices, 0);
  for (int i = 0; i < numVertices; i++)
    for (int nb : adjList[i]) inDegree[nb]++;
  ...
}
```

Vertex 0: source (in-degree 0) · Vertex 4: sink (out-degree 0)

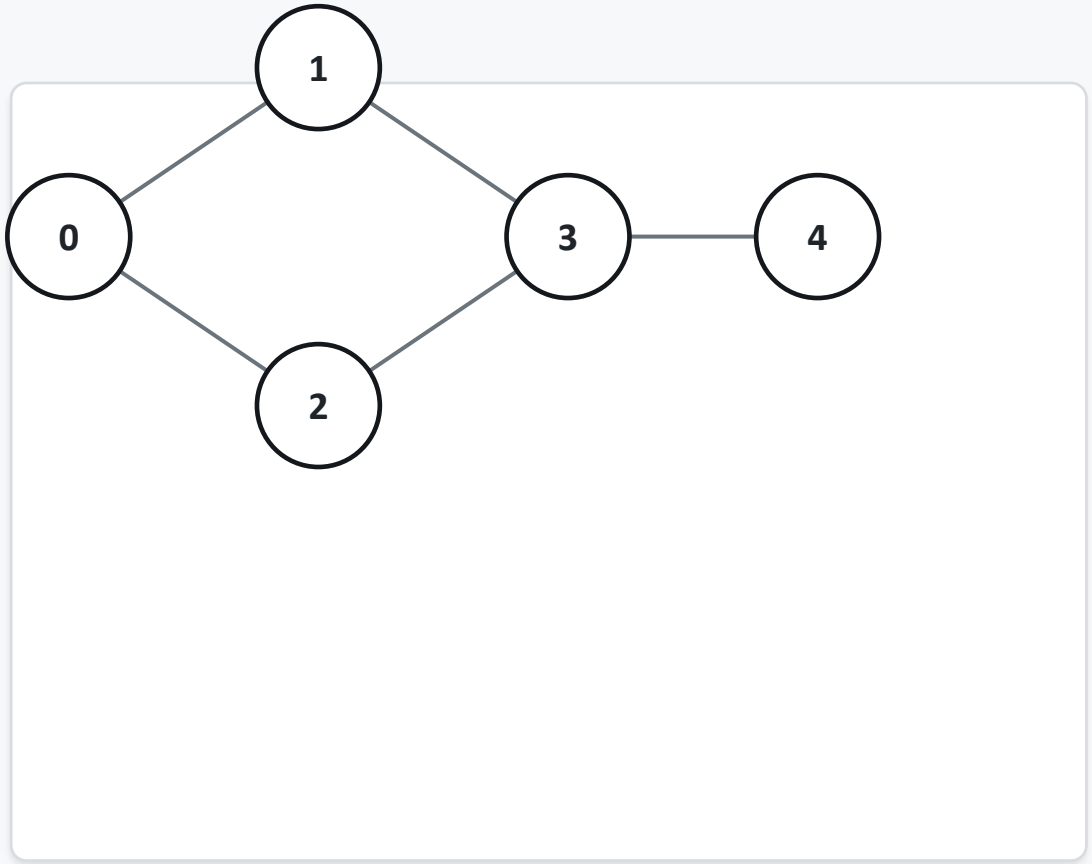


v	out-deg	in-deg
0	2	0
1	1	1
2	1	1
3	1	2
4	0	1

REPRESENTATION #1

Adjacency Matrix

A **V × V** 2D array. matrix[i][j] = 1 (or the weight) if an edge connects i and j, 0 otherwise.



matrix[i][j]

	0	1	2	3	4
0	0	1	1	0	0
1	1	0	0	1	0
2	1	0	0	1	0
3	0	1	1	0	1
4	0	0	0	1	0

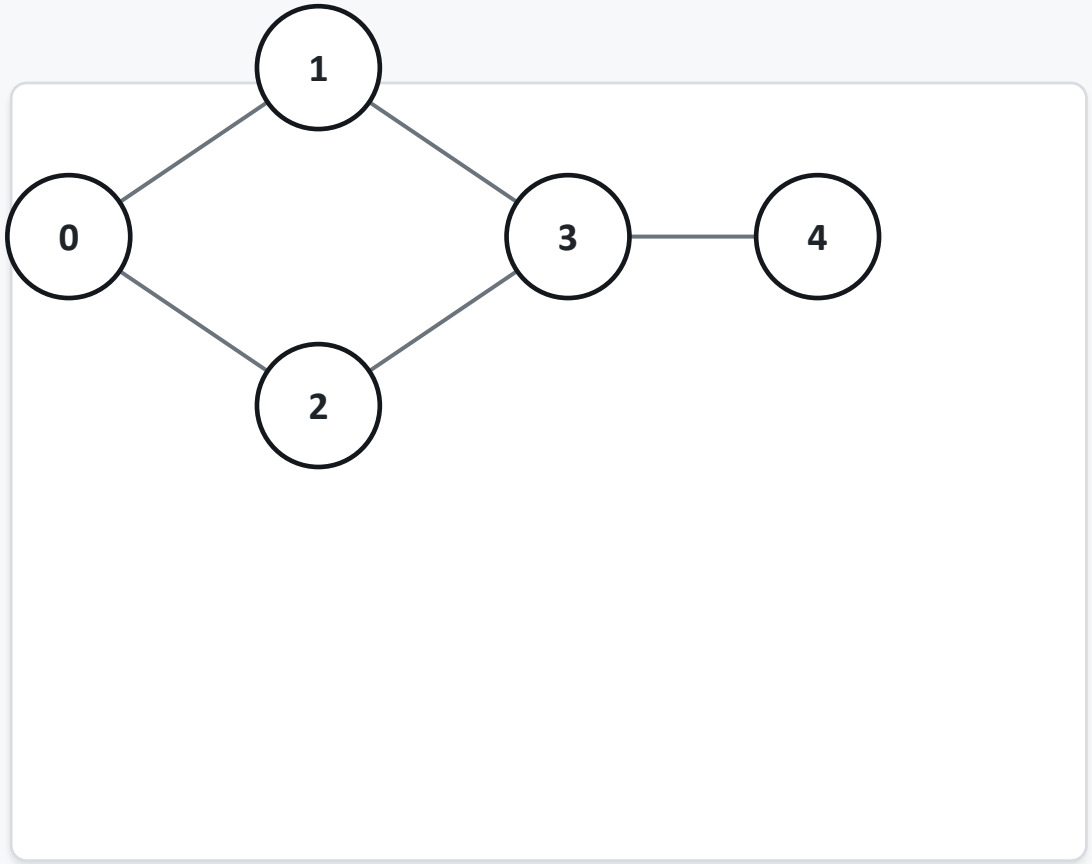
O(1)
edge lookup

O(V²)
space, regardless of edges

REPRESENTATION #2

Adjacency List

For each vertex, store only the vertices it's actually connected to — using the linked structures from Units 2-4.



adjList[i] (same graph)

0	1 -> 2
1	0 -> 3
2	0 -> 3
3	1 -> 2 -> 4
4	3

O(deg)
scan a neighbor list

O(V+E)
total space used

CONVERTING BETWEEN REPRESENTATIONS

Same Facts, Two Structures

Both representations describe the **same underlying graph** — building both from one shared edge list and cross-checking a fact proves they agree.

graph_conversion.cpp — excerpt

```
struct Edge { int u, v, weight; };

void buildMatrix(..., vector<vector<int>>& matrix) {
  for (auto& e : edges) {
    matrix[e.u][e.v] = e.weight;
    matrix[e.v][e.u] = e.weight;
  }
}

void buildList(..., vector<list<pair<int,int>>>&
adjList) {
  for (auto& e : edges) {
    adjList[e.u].push_back({e.v, e.weight});
    adjList[e.v].push_back({e.u, e.weight});
  }
}
```

Cross-check: weight of edge 3-1

```
matrix[3][1]  =  1

scan adjList[3] for 1  =  1
```

Same fact, retrieved two different ways:

O(1)
matrix index jumps straight to the answer

O(deg)
list scans neighbors one at a time

COMPARISON

Matrix vs. List: A Direct Comparison

	Adjacency Matrix	Adjacency List
Space	$O(V^2)$, regardless of edges	O(V + E) — proportional to reality
Check edge (u, v) exists	O(1)	O(degree of u)
Visit all neighbors of v	O(V) — scan whole row	O(degree of u) — exactly the neighbors
Best for	Dense graphs	Sparse graphs (most real-world graphs)

Real-world graphs are almost always sparse — this is why the adjacency list is the representation Lectures 26-28 build on.

AT REAL SCALE

Why Sparsity Matters, Concretely

A city road network: V = 1,000,000 intersections, E ≈ 3,000,000 roads.

~1 trillion
matrix cells (V^2)

~4 million
list entries (V + E)

250,000x
smaller footprint

Add a new vertex	Rebuild entire matrix (row + col)	O(1) — append an empty list
Iterate all edges	$O(V^2)$ — scan whole matrix	O(V + E) — walk each list once

BFS, DFS, Dijkstra's algorithm, and both MST algorithms (Lectures 26-28) all need to visit every edge — O(V+E) vs. $O(V^2)$ is the difference between instant and never-finishes.

KEY TAKEAWAYS

- A graph generalizes a tree completely: any vertex can connect to any number of others, with no restriction on cycles or a single parent.
- Directed edges only appear in their source vertex's adjacency list — which is why directed graphs need both in-degree and out-degree.
- Adjacency matrix: O(1) edge lookup at $O(V^2)$ space. Adjacency list: compact O(V + E) space at the cost of scanning neighbors.
- Both representations encode the same facts about a graph — they just trade space for lookup speed in opposite directions.
- Most real-world graphs are sparse, which is why the adjacency list is the representation used throughout the rest of this unit.