Graph Theory (Practical)

What Problem This Solves

Graphs model relationships between things.

Any time you have:

- **Connections**: users following users, pages linking to pages
- **Dependencies**: packages requiring packages, tasks depending on tasks
- **Networks**: servers, routers, services communicating
- **Hierarchies**: org charts, file systems, component trees

...you're working with graphs.

Graphs make invisible relationships visible and computable.

Intuition & Mental Model

Think: Social Network, Not Bar Chart

A **graph** is just:

- **Nodes** (vertices): The things
- **Edges** (links): The connections between things

```
    Alice ←→ Bob
      ↓        ↓
    Carol → David

Nodes: {Alice, Bob, Carol, David}
Edges: {Alice-Bob, Alice-Carol, Bob-David, Carol-David}
```

Not this kind of graph:

```
  ↑
  |  /\
  | /  \
  |/    \___
  └──────────→
```

Core Concepts

1. Directed vs Undirected Graphs

Undirected: Connection works both ways

```
Friendship graph:
    A ─── B
    |     |
    C ─── D
```

```
If A is friends with B, then B is friends with A
```

```
const friendships = {
  'A': ['B', 'C'],
  'B': ['A', 'D'],
  'C': ['A', 'D'],
  'D': ['B', 'C']
};
```

Directed: Connection has a direction

```
Twitter follows:
    A ——→ B
    ↑       ↓
    C ←—— D

A follows B, but B might not follow A back
```

```
const follows = {
  'A': ['B'],
  'B': ['D'],
  'C': ['A'],
  'D': ['C']
};
```

Real-world examples:

Type	Example
Undirected	Friendships, network cables, chemical bonds
Directed	Twitter follows, web links, function calls, task dependencies

---

2. Representing Graphs in Code

Adjacency List (most common):

```
// Memory efficient, fast for sparse graphs
const graph = {
  'A': ['B', 'C'],
  'B': ['D'],
  'C': ['D'],
  'D': []
};

// Check if edge exists: O(degree)
graph['A'].includes('B'); // true
```

Adjacency Matrix:

```
// Fast lookups, uses more memory
const nodes = ['A', 'B', 'C', 'D'];
const matrix = [
//   A  B  C  D
  [ 0, 1, 1, 0 ], // A connects to B, C
  [ 0, 0, 0, 1 ], // B connects to D
  [ 0, 0, 0, 1 ], // C connects to D
  [ 0, 0, 0, 0 ]  // D connects to nothing
];

// Check if edge exists: O(1)
const hasEdge = (from, to) => matrix[from][to] === 1;
```

Edge List:

```
// Simple, good for storing/serializing
const edges = [
  ['A', 'B'],
  ['A', 'C'],
  ['B', 'D'],
  ['C', 'D']
];
```

When to use each:

Representation	Best For	Space
Adjacency List	Sparse graphs (few edges)	O(V + E)
Adjacency Matrix	Dense graphs, fast lookups	$O(V^2)$
Edge List	Serialization, algorithms that iterate edges	O(E)

---

3. Graph Terminology

Degree: Number of connections

```
Undirected graph:
    A ─── B ─── C

degree(A) = 1
degree(B) = 2
degree(C) = 1
```

In-degree / Out-degree (directed):

```
Directed graph:
    A ───→ B ───→ C
    ↑             ↓
    └─────────────┘
```

```
in-degree(A) = 1  (one arrow pointing to A)
out-degree(A) = 1 (one arrow pointing from A)
```

Path: Sequence of connected nodes

```
    A → B → C → D

Path from A to D: [A, B, C, D]
Length: 3 (number of edges)
```

Cycle: Path that returns to starting node

```
    A → B
    ↑   ↓
    D ← C

Cycle: [A, B, C, D, A]
```

Connected: All nodes reachable from any node

```
Connected:          Not connected:
    A — B               A — B    C — D
    |   |
    C — D
```

---

4. Special Graph Types

Tree: Connected graph with no cycles

```
        A          ← Root
       / \
      B   C        ← Children
     / \   \
    D   E   F      ← Leaves

Properties:
- Exactly one path between any two nodes
- |edges| = |nodes| - 1
- No cycles
```

```
// File system (tree)
const fileSystem = {
  'root': ['home', 'etc', 'var'],
  'home': ['user1', 'user2'],
  'user1': ['documents', 'downloads'],
  'documents': [],
  'downloads': [],
  'user2': [],
  'etc': [],
```

```
  'var': []
};
```

DAG (Directed Acyclic Graph): Directed graph, no cycles

```
    A → B → D
    ↓   ↓
    C → E

Can't get back to where you started following arrows
```

Use cases:

- **Task dependencies**: Task B depends on A
- **Build systems**: File B depends on A
- **Git commit history**: Commit B comes after A
- **React component trees**: Child component depends on parent

```
// Build dependencies (DAG)
const buildOrder = {
  'parse': [],
  'transform': ['parse'],
  'bundle': ['transform'],
  'minify': ['bundle'],
  'deploy': ['minify', 'test'],
  'test': ['bundle']
};

// Topological sort gives valid build order
```

Weighted Graph: Edges have values

```
    A --5-- B
    |      /
   10    3
    |  /
    C

Edge (A,B) has weight 5
Edge (A,C) has weight 10
Edge (B,C) has weight 3
```

```
const graph = {
  'A': [{ node: 'B', weight: 5 }, { node: 'C', weight: 10 }],
  'B': [{ node: 'C', weight: 3 }],
  'C': []
};
```

Applications:

- **Road networks**: weight = distance/time

- **Network topology**: weight = latency/bandwidth
- **Cost optimization**: weight = cost/priority

---

5. Graph Traversal

Depth-First Search (DFS): Go deep, then backtrack

```
Graph:
    A → B → D
    ↓   ↓
    C   E

DFS from A: A → B → D (dead end) → E (dead end) → C

Visit order: [A, B, D, E, C]
```

```
function dfs(graph, start, visited = new Set()) {
  if (visited.has(start)) return;

  console.log(start); // Visit node
  visited.add(start);

  for (const neighbor of graph[start] || []) {
    dfs(graph, neighbor, visited); // Recursive
  }
}

// Like diving deep into folders before moving to next one
```

Breadth-First Search (BFS): Visit level by level

```
Graph:
    A → B → D
    ↓   ↓
    C   E

BFS from A: A → B → C (level 1) → D → E (level 2)

Visit order: [A, B, C, D, E]
```

```
function bfs(graph, start) {
  const visited = new Set();
  const queue = [start];
  visited.add(start);

  while (queue.length > 0) {
    const node = queue.shift();
    console.log(node); // Visit node

    for (const neighbor of graph[node] || []) {
```

```
      if (!visited.has(neighbor)) {
        visited.add(neighbor);
        queue.push(neighbor);
      }
    }
  }
}

// Like exploring floor by floor in a building
```

When to use each:

Algorithm	Best For	Space	Use Case
DFS	Finding paths, detecting cycles	O(h)	File system, topological sort
BFS	Shortest path (unweighted), level-order	O(w)	Social network distance, routing

---

6. Cycles and DAGs

Detecting cycles (critical for dependency management):

```
function hasCycle(graph) {
  const visiting = new Set(); // Currently exploring
  const visited = new Set();  // Fully explored

  function dfs(node) {
    if (visiting.has(node)) return true; // Cycle!
    if (visited.has(node)) return false; // Already checked

    visiting.add(node);
    for (const neighbor of graph[node] || []) {
      if (dfs(neighbor)) return true;
    }
    visiting.delete(node);
    visited.add(node);
    return false;
  }

  for (const node of Object.keys(graph)) {
    if (dfs(node)) return true;
  }
  return false;
}

// Example: package.json circular dependency check
```

Topological Sort: Order nodes so all edges point forward

```
Dependencies:
    A → B → D
```

```
    ↓    ↓
    C    E

Valid topological order: [A, B, C, D, E] or [A, C, B, E, D]
Invalid: [B, A, C, D, E] (B comes before A, but A→B exists)
```

```
function topologicalSort(graph) {
  const visited = new Set();
  const stack = [];

  function dfs(node) {
    if (visited.has(node)) return;
    visited.add(node);

    for (const neighbor of graph[node] || []) {
      dfs(neighbor);
    }

    stack.unshift(node); // Add to front
  }

  for (const node of Object.keys(graph)) {
    dfs(node);
  }

  return stack;
}

// Build systems use this to determine compilation order
```

Software Engineering Connections

1. React Component Trees

```
<App>                   ← Root
  <Header />            ← Child
  <Main>                ← Child
    <Sidebar />         ← Grandchild
    <Content />         ← Grandchild
  </Main>
  <Footer />            ← Child
</App>

Tree graph:
App → [Header, Main, Footer]
Main → [Sidebar, Content]
```

Virtual DOM diffing = graph comparison

2. Package Dependencies

```
// package.json (DAG)
{
  "dependencies": {
    "express": "^4.0.0",     // Express depends on other packages
    "react": "^18.0.0"
  }
}

// npm/yarn resolves dependency graph
// Detects circular dependencies
// Determines installation order (topological sort)
```

3. Database Relationships

```sql
-- Users following users (directed graph)
CREATE TABLE follows (
  follower_id INT,
  following_id INT
);

-- Query: mutual follows (undirected subgraph)
SELECT f1.follower_id, f1.following_id
FROM follows f1
JOIN follows f2
  ON f1.follower_id = f2.following_id
  AND f1.following_id = f2.follower_id;
```

4. API Route Graph

```
// Express routes (DAG of middleware)
app.use(authMiddleware);     // A
app.use(loggingMiddleware);  // B
app.get('/api/users',
  validateMiddleware,         // C
  getUserHandler              // D
);

// Execution graph: A → B → C → D
```

5. Git Commit History

```
     A ← B ← C ← D     (main)
          ↖
            E ← F      (feature branch)

DAG:
```

```
- Each commit points to parent(s)
- Merges create nodes with multiple parents
- No cycles (can't commit before your parent)
```

6. Network Topology

```javascript
// Service mesh (weighted directed graph)
const services = {
  'api-gateway': [
    { service: 'auth', latency: 10 },
    { service: 'users', latency: 15 }
  ],
  'auth': [
    { service: 'database', latency: 5 }
  ],
  'users': [
    { service: 'database', latency: 5 },
    { service: 'cache', latency: 2 }
  ]
};

// Shortest path = fastest route through services
```

Common Misconceptions

❌ "Graphs are always visualized"

Graphs are **abstract data structures**. Visualization helps understanding, but code works with adjacency lists/matrices.

❌ "Trees are not graphs"

Trees **are graphs** (special case: connected, acyclic, undirected). All trees are graphs, but not all graphs are trees.

❌ "Adjacency matrix is always better"

Not for sparse graphs. If you have 1000 nodes but only 1500 edges, matrix uses 1,000,000 cells (99.85% waste).

❌ "BFS always finds the shortest path"

Only in unweighted graphs. In weighted graphs, use Dijkstra's algorithm (we'll cover in optimization).

❌ "Cycles are always bad"

In **directed graphs**, cycles often indicate problems (circular dependencies). In **undirected graphs**, cycles are common and fine (social networks, road networks).

Practical Mini-Exercises

Exercise 1: Social Network

Build a directed graph of Twitter-like follows:

```
const follows = {
  'alice': ['bob', 'charlie'],
  'bob': ['charlie'],
  'charlie': ['alice'],
  'david': []
};
```

Implement:

1. `getFollowers(user)` - who follows this user?
2. `getMutualFollows(user1, user2)` - do they follow each other?
3. `suggestFollows(user)` - friends of friends

▶ Solution

Exercise 2: Build Dependency Checker

Given build dependencies, check for circular dependencies:

```
const deps = {
  'app': ['utils', 'api'],
  'api': ['utils', 'models'],
  'models': ['utils'],
  'utils': []
};
```

▶ Solution

Exercise 3: Shortest Connection

Find shortest path between two users (unweighted):

```
const friendships = {
  'alice': ['bob', 'charlie'],
  'bob': ['alice', 'david'],
  'charlie': ['alice', 'david', 'eve'],
  'david': ['bob', 'charlie'],
  'eve': ['charlie']
};
```

Find path from 'alice' to 'eve'.

▶ Solution

Summary Cheat Sheet

Graph Types

Type	Definition	Example

Undirected	Edges work both ways	Friendships
Directed	Edges have direction	Twitter follows
Tree	Connected, no cycles	File system
DAG	Directed, no cycles	Task dependencies
Weighted	Edges have values	Road distances

Representations

```
// Adjacency list (most common)
const graph = {
  'A': ['B', 'C'],
  'B': ['D']
};

// Adjacency matrix
const matrix = [
  [0, 1, 1, 0],
  [0, 0, 0, 1]
];

// Edge list
const edges = [['A', 'B'], ['A', 'C'], ['B', 'D']];
```

Algorithms

```
// DFS (recursive, go deep)
function dfs(graph, node, visited = new Set()) {
  if (visited.has(node)) return;
  visited.add(node);
  for (const neighbor of graph[node] || []) {
    dfs(graph, neighbor, visited);
  }
}

// BFS (iterative, level by level)
function bfs(graph, start) {
  const queue = [start];
  const visited = new Set([start]);
  while (queue.length > 0) {
    const node = queue.shift();
    for (const neighbor of graph[node] || []) {
      if (!visited.has(neighbor)) {
        visited.add(neighbor);
        queue.push(neighbor);
      }
    }
```

```
    }
}
```

Next Steps

Graphs are everywhere in software. You now understand how to model and traverse relationships.

Next, we'll explore **Boolean algebra**—the mathematical foundation of logic, conditionals, and decision-making in code.

Continue to: 03-boolean-algebra.md