Graph Theory

A Complete 10-Chapter Maths for Programmers Course

Topics covered:

Graph representations & terminology · BFS & DFS traversal

Connectivity & cycle detection · topological sorting

Dijkstra's & Bellman-Ford shortest paths · Minimum Spanning Trees

Trees as special graphs · structure & traversal

Capstone: a full worked infrastructure project

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Why Graph Theory Matters for Programmers

Graph Theory

Chapter 1 · Why Graph Theory Matters for Programmers

Algorithms & Complexity's own Chapter 1 promised this course would build directly on its recursion and Master Theorem material. Discrete Mathematics Fundamentals' own Chapter 1 promised a graph was "technically also a kind of relation," reserved for here. Both promises land in the same place: a **graph** is simply things (**vertices**) and the connections between them (**edges**) — one of the most general, most reused modeling tools in all of computer science.

What a Graph Actually Is

Strip away whatever the vertices and edges represent — cities and roads, people and friendships, packages and dependencies, web pages and links — and what's left is pure structure: which things connect to which. Graph theory studies that structure directly, which is exactly why the same handful of algorithms this course covers work identically whether the vertices are cities, users, or build targets.

> **THIS IS EXACTLY A RELATION, MADE VISUAL**
>
> Discrete Mathematics Fundamentals Chapter 5 defined a relation as a set of ordered pairs describing which elements connect to which — a database foreign key, "same team as," "owns." A graph is that exact same idea, just drawn: each pair in the relation is an edge, each element is a vertex. Nothing new is being invented here — it's a familiar structure, given its own dedicated toolkit.

A Real Space Comparison: Representing a Social Network

A graph can be stored two obvious ways: an **adjacency matrix** (a V×V grid marking which vertex pairs connect) or an **adjacency list** (each vertex keeping a list of just its own neighbors). For a realistic social network — 1,000 users, each with roughly 10 connections:

REPRESENTATION	SPACE NEEDED
Adjacency matrix	$V^2 = 1{,}000^2 =$ **1,000,000** cells
Adjacency list	$V + 2E \approx 1{,}000 + 10{,}000 =$ **11,000** entries

A **90.9×** difference in space, for representing the exact same graph — because real-world graphs like social networks are *sparse*: each user connects to a tiny fraction of all other users, not roughly half of them. Chapter 2 formalizes this tradeoff directly, reusing Algorithms & Complexity Chapter 8's own auxiliary-space accounting.

Five Concrete Connections to Real Code

GRAPH THEORY TOPIC	WHERE IT ACTUALLY SHOWS UP
Dependency resolution (Ch.4–5)	Package managers and build systems detecting circular dependencies and computing a valid install/build order
Shortest paths (Ch.6–7)	Map routing, network packet routing, "fastest way from A to B" problems of every kind
Connectivity (Ch.3–4)	Social networks — "how are these two people connected," friend/follower graphs, recommendation systems
Minimum spanning trees (Ch.8)	Network design — connecting every location with the least total cable, road, or connection cost
Trees (Ch.9)	File systems, org charts, decision trees — hierarchical structures are graphs with one extra constraint

What This Course Won't Cover

Graph theory as a full field is enormous. This course deliberately covers the core representations and the classic named algorithms every working programmer eventually needs, not a comprehensive catalog:

- **Specialized algorithms** — network flow (max-flow/min-cut), graph coloring, planarity testing, and similar advanced topics stay out of scope; each is substantial enough to deserve its own future treatment
- **Graph database technology** — this course covers the mathematical structure and algorithms, not specific tools like Neo4j or graph-query languages
- **Advanced network flow theory** — genuinely deep optimization territory, beyond this course's own scope

> **WHY DRAW THE LINE HERE INSTEAD OF COVERING EVERYTHING AT ONCE**
>
> Each of those areas is substantial enough to deserve its own real depth rather than a rushed final section. This course stays tightly scoped to representations, traversal, and the small set of named algorithms (topological sort, Dijkstra's, Bellman-Ford, Kruskal's/Prim's) that come up constantly in ordinary engineering work — the direct foundation any of those more specialized topics would build on.

Where This Course Is Headed

CHAPTER	TOPIC
2	Graph Representations & Terminology
3	Graph Traversal: BFS & DFS
4	Connectivity & Cycle Detection
5	Topological Sorting
6	Shortest Path Algorithms: Dijkstra's Algorithm
7	Shortest Path Algorithms: Bellman-Ford & Negative Weights
8	Minimum Spanning Trees: Kruskal's & Prim's Algorithms
9	Trees as Special Graphs: Structure & Traversal
10	Capstone — Modeling and Solving a Real Graph Problem

THIS COURSE'S THROUGHLINE

Every chapter answers a version of the same question: given a structure made only of things and the connections between them, what can be determined systematically — is everything reachable, what's the cheapest path, is there a valid order, is there a hidden cycle? These are genuinely the same handful of questions asked repeatedly across wildly different domains, which is exactly why learning the structure once pays off everywhere it shows up again.

Hands-On Exercises

EXERCISE 1

A road network graph has 500 vertices (intersections), each connected to an average of 4 other intersections. Compute the adjacency matrix space (V^2) and the adjacency list space ($V + 2E$) for this graph, and compute the ratio between them.

View solution

EXERCISE 2

A colleague says "graphs are just an academic topic — I've never needed one in real code." Using this chapter's own five connections, name two genuinely different real systems (not variations of the same idea) that are secretly graph problems, and explain the connection for each.

View solution

EXERCISE 3

Using this chapter's own relation-to-graph connection, explain how a database table with a self-referencing foreign key (for example, an `employees` table where each row has a `manager_id` pointing to another row in the same table) can be understood as a graph. What are the vertices, and what are the edges?

View solution

CHAPTER 1 QUICK REFERENCE

- A **graph** is vertices (things) and edges (connections) — pure structure, independent of what it represents
- A graph is exactly a relation (Discrete Mathematics Fundamentals Chapter 5), drawn — nothing new is being invented
- Adjacency matrix (V^2) vs. adjacency list ($V+2E$): a 90.9× space difference for a realistic 1,000-user sparse social network
- Five direct connections: dependency resolution, shortest paths, connectivity, minimum spanning trees, and hierarchical trees
- Deliberately out of scope: network flow, graph coloring, planarity, and graph-database-specific tooling
- **Next chapter:** Graph representations and terminology

CHAPTER 2 OF 10

Graph Representations & Terminology

Graph Theory

Chapter 2 · Graph Representations & Terminology

Chapter 1 showed the space gap between two representations in the abstract. This chapter builds both, in full, for the exact same small graph — and pins down the vocabulary every later chapter leans on.

Core Terminology

TERM	MEANING
Vertex (node)	A single "thing" in the graph
Edge	A connection between two vertices
Degree	The number of edges touching a given vertex
Directed edge	A one-way connection, `u → v` — an asymmetric relation, in Discrete Mathematics Fundamentals Chapter 5's own terms
Undirected edge	A two-way connection, `u – v` — a symmetric relation
Weighted edge	An edge carrying a numeric cost (distance, time, capacity)
Path	A sequence of edges connecting one vertex to another

A Worked Example: One Graph, Two Representations

Five vertices, five undirected, unweighted edges: `A-B, A-C, B-C, C-D, D-E`.

Adjacency Matrix

```
    A  B  C  D  E
A  [0, 1, 1, 0, 0]
B  [1, 0, 1, 0, 0]
C  [1, 1, 0, 1, 0]
D  [0, 0, 1, 0, 1]
E  [0, 0, 0, 1, 0]
```

Adjacency List

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

Reading degree directly off the adjacency list (its own length per vertex): `degree(A)=2, degree(B)=2, degree(C)=3, degree(D)=2, degree(E)=1`. For this graph, `V=5`, `E=5`: matrix needs `25` cells, the list needs `15` entries — a smaller gap than Chapter 1's own 1,000-vertex example, since the space advantage of the list grows with graph size.

The Time/Space Tradeoff

THE SAME TRADEOFF ALGORITHMS & COMPLEXITY CHAPTER 8 ALREADY ESTABLISHED

Just as that chapter found merge sort's better time came with a real space cost against bubble sort, these two graph representations trade space for a different operation's speed — neither is simply "better."

OPERATION	ADJACENCY MATRIX	ADJACENCY LIST
Space	$O(V^2)$	$O(V+E)$
Check if edge (u,v) exists	O(1) — direct lookup	O(degree(v)) — must scan the list
List all neighbors of v	O(V) — scan the whole row	O(degree(v)) — already just that list
Add a new vertex	$O(V^2)$ — matrix must grow in both dimensions	O(1) — just add a new empty list

WHEN EACH REPRESENTATION ACTUALLY WINS

For **sparse** graphs (E much smaller than V^2, the overwhelmingly common real-world case — social networks, road networks, dependency graphs), the adjacency list wins on both space *and* the neighbor-listing operations traversal algorithms need most. The matrix only pulls ahead for **dense** graphs (E close to V^2) or when "does this specific edge exist" needs to be checked very frequently, where its O(1) lookup genuinely matters more than the space cost.

Representations in Code

```
vertices = ['A', 'B', 'C', 'D', 'E']
edges = [('A','B'), ('A','C'), ('B','C'), ('C','D'), ('D','E')]

# Adjacency list — the standard choice for sparse graphs
adj_list = {v: [] for v in vertices}
for u, v in edges:
    adj_list[u].append(v)
    adj_list[v].append(u)  # undirected: add both directions

print(adj_list['C'])          # ['A', 'B', 'D']
print(len(adj_list['C']))     # 3 -- degree(C)
```

Hands-On Exercises

EXERCISE 1

Build the adjacency matrix and adjacency list for the undirected graph with vertices `{P, Q, R, S}` and edges `P-Q, P-R, Q-R, R-S`. State the degree of each vertex.

View solution

EXERCISE 2

A graph has 200 vertices and 190 edges (a sparse graph). Compute the adjacency matrix space (V^2) and the adjacency list space (V+2E). Which representation would you choose for this graph, and why, using this chapter's own sparse-vs-dense reasoning?

View solution

EXERCISE 3

A system needs to check, very frequently, whether a specific pair of users is directly connected — but rarely needs to list all of a user's connections. Using this chapter's own time/space tradeoff table, explain which representation better fits this specific access pattern, even if the underlying social graph is sparse.

View solution

CHAPTER 2 QUICK REFERENCE

- **Vertex, edge, degree** — the core vocabulary; directed edges are asymmetric relations, undirected are symmetric (Discrete Mathematics Fundamentals Chapter 5)
- **Adjacency matrix:** $O(V^2)$ space, O(1) edge lookup, O(V) to list neighbors
- **Adjacency list:** O(V+E) space, O(degree) edge lookup, O(degree) to list neighbors

- Sparse graphs (the common real-world case) favor the list on nearly every dimension; dense graphs or frequent-edge-lookup workloads favor the matrix
- This is the same time/space tradeoff pattern Algorithms & Complexity Chapter 8 already established for sorting algorithms, now applied to graph storage
- **Next chapter:** Graph traversal — BFS and DFS

CHAPTER 3 OF 10

Graph Traversal: BFS & DFS

Graph Theory

Chapter 3 · Graph Traversal: BFS & DFS

Every algorithm this course covers from here on — cycle detection, topological sort, shortest paths — is a variation on one of two ways to systematically visit every reachable vertex in a graph. Both visit the same vertices in the end; the *order* they visit them in is what makes each one suited to genuinely different problems.

Breadth-First Search (BFS) — Level by Level

THE MECHANISM

BFS uses a **queue** (first-in, first-out): visit a vertex, add all its unvisited neighbors to the queue, then process the queue in the order they were added — visiting every vertex at distance 1 from the start before any vertex at distance 2.

Depth-First Search (DFS) — Follow One Path to the End

THE MECHANISM

DFS uses a **stack** (last-in, first-out) — or, equivalently, recursion (Algorithms & Complexity Chapter 5's own recursion machinery, applied directly): visit a vertex, immediately dive into its first unvisited neighbor, and keep diving until forced to backtrack.

A Worked Comparison: Genuinely Different Orders

A branching graph: `A` connects to `B` and `C`; `B` connects to `D` and `E`; `C` connects to `F`.

TRAVERSAL	VISIT ORDER FROM A
BFS	`A, B, C, D, E, F` — both level-1 vertices (B, C) before any level-2 vertex
DFS	`A, B, D, E, C, F` — B's entire branch (D, E) fully explored before C is even touched

WHY BFS FINDS THE SHORTEST PATH IN AN UNWEIGHTED GRAPH

Because BFS processes strictly level by level, the *first* time it reaches any vertex is guaranteed to be via the fewest possible edges — there's no way to reach a vertex "early" through a longer path, since every shorter path from the same level would have been explored first. DFS has no such guarantee: it happily commits to a long, winding path before ever backtracking to check a much shorter one.

Complexity — And Why the Representation Choice Matters

O(V+E) — ON AN ADJACENCY LIST

Both BFS and DFS visit every vertex exactly once (O(V)) and, for each vertex, examine every one of its edges exactly once (O(E) total across the whole traversal) — Θ(V+E) altogether, using Chapter 2's own adjacency list.

THE REPRESENTATION CHOICE DIRECTLY CHANGES THIS

Using an **adjacency matrix** instead, finding a vertex's neighbors means scanning its entire row — O(V) per vertex — making the whole traversal $O(V^2)$, not O(V+E). For a sparse graph, this is a real, measurable slowdown, not just a storage inconvenience: Chapter 2's own representation choice propagates directly into every traversal algorithm built on top of it.

Real Use Cases

TRAVERSAL	USED FOR
BFS	Shortest path in unweighted graphs, "nearest" queries, level-order/web-crawling-by-distance
DFS	Cycle detection (Chapter 4), topological sort (Chapter 5), path-existence/maze-solving, finding connected components

Both Traversals in Code

```
from collections import deque

adj = {'A': ['B','C'], 'B': ['A','D','E'], 'C': ['A','F'],
       'D': ['B'], 'E': ['B'], 'F': ['C']}

def bfs(start):
    visited, order, q = {start}, [start], deque([start])
    while q:
        u = q.popleft()
        for v in adj[u]:
            if v not in visited:
                visited.add(v); order.append(v); q.append(v)
    return order

def dfs(start, visited=None, order=None):
    if visited is None:
        visited, order = set(), []
    visited.add(start); order.append(start)
    for v in adj[start]:
        if v not in visited:
            dfs(v, visited, order)
    return order

print(bfs('A'))  # ['A', 'B', 'C', 'D', 'E', 'F']
print(dfs('A'))  # ['A', 'B', 'D', 'E', 'C', 'F']
```

Hands-On Exercises

EXERCISE 1

A graph has vertex `1` connected to `2` and `3`; `2` connected to `4`; `3` connected to `5` and `6`. Trace the BFS visit order starting from vertex `1`, following this chapter's own level-by-level method.

View solution

EXERCISE 2

Using the exact same graph as Exercise 1, trace the DFS visit order starting from vertex `1` (visiting each vertex's neighbors in the order given: for vertex 1, visit 2 before 3; for vertex 3, visit 5 before 6). Compare the resulting order directly against Exercise 1's BFS order.

View solution

EXERCISE 3

Explain, in your own words, why BFS is guaranteed to find the shortest (fewest-edge) path to every reachable vertex in an unweighted graph, while DFS gives no such guarantee — grounding your answer in this chapter's own level-by-level

vs. dive-then-backtrack distinction.

View solution

CHAPTER 3 QUICK REFERENCE

- **BFS:** queue-based, visits level by level — guarantees the shortest (fewest-edge) path in unweighted graphs
- **DFS:** stack/recursion-based, dives down one branch fully before backtracking
- Both are $\Theta(V+E)$ on an adjacency list — but $O(V^2)$ on an adjacency matrix, since Chapter 2's own representation choice directly affects traversal speed
- BFS → shortest unweighted paths, nearest-neighbor queries; DFS → cycle detection, topological sort, path existence, connected components
- **Next chapter:** Connectivity and cycle detection

CHAPTER 4 OF 10

Connectivity & Cycle Detection

Graph Theory

Chapter 4 · Connectivity & Cycle Detection

Chapter 3 built two ways to explore a graph. This chapter puts them to work answering two genuinely practical questions: is everything actually reachable from everything else, and does this graph loop back on itself somewhere?

Connected Components

A **connected component** is a maximal group of vertices all reachable from each other. Finding every component in a graph is just Chapter 3's own traversal, repeated: run DFS (or BFS) from any unvisited vertex, mark everything it reaches, then repeat from the next still-unvisited vertex.

A graph with three separate "islands": `{A,B,C}` (a triangle), `{D,E}`, `{F,G}` — no edges connecting the groups.

VERIFIED DIRECTLY

Running the repeated-traversal method produces exactly three components: `['A','B','C']`, `['D','E']`, `['F','G']` — each vertex accounted for exactly once, matching the graph's own visibly disconnected structure.

Cycle Detection in Undirected Graphs

During a DFS, encountering an edge to an already-visited vertex normally signals a cycle — **except** the edge leading straight back to the vertex you just came from, which is always there in an undirected graph and doesn't count.

THE PARENT-TRACKING RULE

While doing DFS, track the vertex each call arrived *from* (its "parent"). If DFS reaches an already-visited vertex that **isn't** the current parent, a genuine cycle exists.

Testing two small graphs directly:

GRAPH	STRUCTURE	HAS A CYCLE?
Triangle	A-B, B-C, C-A	**True**
Path/tree	A-B, B-C, C-D	False

A USEFUL FACT, FORWARD-REFERENCING CHAPTER 9

An undirected tree with `V` vertices always has exactly `V-1` edges and contains no cycles — the second row above (4 vertices, 3 edges) is exactly a tree, Chapter 9's own subject.

Cycle Detection in Directed Graphs — A Genuinely Different Problem

The parent-tracking trick doesn't work for directed graphs — an edge can point to an already-visited vertex without forming a cycle at all, if that vertex was finished processing along a completely different branch. The real signal is a **back edge**: an edge to a vertex still *currently on the DFS call stack*, not just visited at some point in the past.

THE THREE-COLOR DFS METHOD

Mark each vertex **white** (unvisited), **gray** (currently being explored — on the active DFS path), or **black** (fully finished). An edge to a **gray** vertex is a genuine back edge — a cycle. An edge to a black vertex is safe.

Real Relevance: Detecting Circular Dependencies

Modeling packages as vertices and "depends on" as directed edges — exactly Chapter 1's own dependency-resolution connection:

DEPENDENCY GRAPH	EDGES	HAS A CYCLE?
Circular	A→B, B→C, C→A	**True — A depends on B depends on C depends on A, impossible to install**
Valid	A→B, A→C, B→C	False — a valid order exists: C, B, A

This is exactly what a real package manager runs before attempting any installation — a directed cycle means the dependency requirements are genuinely impossible to satisfy, not just difficult.

Connectivity & Cycle Detection in Code

```python
def find_components(adj):
    visited, components = set(), []
    for v in adj:
        if v not in visited:
            comp, stack = [], [v]
            while stack:
                u = stack.pop()
                if u in visited: continue
                visited.add(u); comp.append(u)
                stack.extend(w for w in adj[u] if w not in visited)
            components.append(comp)
    return components

def has_cycle_directed(adj):
    WHITE, GRAY, BLACK = 0, 1, 2
    color = {v: WHITE for v in adj}
    def dfs(u):
        color[u] = GRAY
        for v in adj[u]:
            if color[v] == GRAY:
                return True
            if color[v] == WHITE and dfs(v):
                return True
        color[u] = BLACK
        return False
    return any(dfs(v) for v in adj if color[v] == WHITE)

cyclic_deps = {'A': ['B'], 'B': ['C'], 'C': ['A']}
print(has_cycle_directed(cyclic_deps))  # True
```

Hands-On Exercises

EXERCISE 1

Find all connected components of the undirected graph with edges `P-Q, Q-R, S-T, U` (U has no edges at all — an isolated vertex).

View solution

EXERCISE 2

Using this chapter's own parent-tracking method, determine whether the undirected graph with edges `A-B, B-C, C-D, D-B` contains a cycle. Show which edge creates it, if one exists.

View solution

EXERCISE 3

A package dependency graph has: `app → libA`, `app → libB`, `libA → libC`, `libB → libC`, `libC → libA`. Using this chapter's own three-color method, determine whether this dependency graph is installable, and if not, identify the specific cycle.

View solution

CHAPTER 4 QUICK REFERENCE

- **Connected components:** repeated DFS/BFS from every unvisited vertex — each run finds one full component
- **Undirected cycle detection:** a DFS edge to an already-visited, non-parent vertex signals a cycle
- **Directed cycle detection:** the three-color method — an edge to a currently-gray (on-the-active-path) vertex is a genuine back edge/cycle
- A tree with V vertices has exactly V−1 edges and no cycles (Chapter 9 forward reference)
- Directed cycle detection is exactly how real package managers detect impossible circular dependencies before attempting installation
- **Next chapter:** Topological sorting

CHAPTER 5 OF 10

Topological Sorting

Graph Theory

Chapter 5 · Topological Sorting

Chapter 4 answered a yes/no question: does this directed graph contain a cycle? When the answer is no — the graph is a **DAG** (Directed Acyclic Graph) — a much more useful question opens up: what's a valid *order* to process every vertex in, so that nothing is ever handled before something it depends on?

What a Topological Order Actually Is

A **topological sort** is a linear ordering of a DAG's vertices such that for every directed edge `u → v`, `u` comes before `v` in the ordering. It only exists for DAGs — Chapter 4's own cycle check is the precondition this chapter builds on. If `A → B → A` existed, no ordering could ever put both `A` before `B` and `B` before `A`.

TWO GENUINELY DIFFERENT WAYS TO BUILD ONE

This chapter covers both, since they reuse the two traversal paradigms Chapter 3 already established: a **DFS-based** method (reusing Chapter 4's own three-color machinery directly) and **Kahn's algorithm**, a BFS-style method built on counting incoming edges.

Method 1: DFS-Based — Order by Finish Time, Reversed

Run a DFS exactly like Chapter 4's cycle check, but this time record each vertex the moment it turns **black** (fully finished — every descendant already processed). Push it onto a stack at that moment. Once the whole DFS is done, popping the stack — i.e. reading it in *reverse* finish order — is a valid topological order.

Why reverse: a vertex only finishes after all of its descendants have already finished, so it always finishes *later* than everything it points to — meaning it sits *later* in the raw finish-order stack. Reversing puts it back in front, ahead of everything it points to. Tracing it directly removes any doubt:

Graph: `A→B`, `A→C`, `B→C`.

VERIFIED DIRECTLY

DFS from A dives to B, then to C. C has no outgoing edges, so it finishes first (stack: `[C]`). Back at B, nothing left to explore, B finishes (stack: `[C, B]`). Back at A, its other neighbor C is already finished (black), so it's skipped — A

finishes last (stack: `[C, B, A]`). Reversing gives the topological order `[A, B, C]` — exactly matching every edge pointing forward: A before B, A before C, B before C.

Method 2: Kahn's Algorithm — Process What's Ready

Kahn's algorithm works from the opposite direction, and maps much more directly onto how a real scheduler behaves: repeatedly find a vertex with no remaining unprocessed dependencies, output it, then "unlock" whatever becomes newly ready as a result.

THE MECHANISM

Compute every vertex's **in-degree** (number of incoming edges). Put every vertex with in-degree 0 into a queue. Repeatedly: pop a vertex, output it, then decrement the in-degree of each of its neighbors — any neighbor that drops to 0 joins the queue. Stop when the queue is empty.

Same graph, traced with Kahn's algorithm:

VERIFIED DIRECTLY

In-degrees: A=0, B=1 (from A), C=2 (from A and B). Only A starts in the queue. Processing A outputs A and decrements B to 0 and C to 1 — B joins the queue. Processing B outputs B and decrements C to 0 — C joins the queue. Processing C outputs C. Result: `[A, B, C]` — identical to the DFS-based result on this graph.

A Topological Order Usually Isn't Unique

When two vertices have no dependency relationship between them at all, either can legally come first. A slightly larger graph makes this visible: `A→C`, `B→C`, `C→D` — both A and B feed independently into C, which feeds into D.

METHOD	ORDER PRODUCED
DFS-based	`[B, A, C, D]`
Kahn's algorithm	`[A, B, C, D]`

BOTH ARE CORRECT — VERIFIED DIRECTLY

A and B never have an edge between them, so neither ordering violates any dependency: every edge (A→C, B→C, C→D) still points forward in *both* lists. The two methods disagree only on the order of independent vertices, which is genuinely unconstrained — there is no single "the" topological order for a DAG with any parallelism in it, only *a* valid one.

Real Relevance: Build Systems & Package Installation

A compiler deciding which source files to build first, or a package manager deciding install order, is running a topological sort on exactly this kind of dependency DAG. Kahn's own framing maps directly onto the real process: "install everything with no remaining unmet dependencies, then see what that unlocks next" is precisely how tools like `npm`, `pip`, and `make` resolve a build order in practice — and it's the natural next step after Chapter 4's own cycle check, which confirms an order exists at all before this chapter finds one.

Topological Sort in Code

```
def topo_sort_dfs(adj):
    WHITE, GRAY, BLACK = 0, 1, 2
    color = {v: WHITE for v in adj}
    finish_stack = []
    def dfs(u):
        color[u] = GRAY
        for v in adj[u]:
            if color[v] == WHITE:
                dfs(v)
        color[u] = BLACK
        finish_stack.append(u)
    for v in adj:
        if color[v] == WHITE:
            dfs(v)
    return finish_stack[::-1]

from collections import deque
def topo_sort_kahn(adj):
    in_degree = {v: 0 for v in adj}
    for u in adj:
        for v in adj[u]:
            in_degree[v] += 1
    queue = deque([v for v in adj if in_degree[v] == 0])
    order = []
    while queue:
        u = queue.popleft()
        order.append(u)
        for v in adj[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
                queue.append(v)
    return order

deps = {'A': ['C'], 'B': ['C'], 'C': ['D'], 'D': []}
print(topo_sort_kahn(deps))  # ['A', 'B', 'C', 'D']
```

Hands-On Exercises

EXERCISE 1

Using this chapter's own DFS-based method, find a topological order for the DAG with edges `X→Y`, `X→Z`, `Y→W`, `Z→W`. Show the finish-order stack before it's reversed.

View solution

EXERCISE 2

A build system has files with dependencies: `utils.o` is needed by both `main.o` and `app`, and `main.o` is needed by `app`. Using Kahn's algorithm, find a valid build order. Show the in-degree of each file at the start.

View solution

EXERCISE 3

This chapter's own worked example (`A→C`, `B→C`, `C→D`) produced two different valid orders from its two methods. Explain, in terms of what a topological order is actually required to guarantee, why both `[B, A, C, D]` and `[A, B, C, D]` are correct answers rather than one being a mistake.

View solution

CHAPTER 5 QUICK REFERENCE

- **Topological sort:** a linear vertex ordering where every directed edge points forward — only possible on a DAG (Chapter 4's cycle check is the precondition)
- **DFS-based method:** push each vertex when it turns black (finishes); reverse the finish-order stack
- **Kahn's algorithm:** repeatedly output a vertex with in-degree 0, then decrement its neighbors' in-degrees
- A topological order is usually **not unique** — vertices with no dependency relationship can appear in either order
- Kahn's "process what's ready, unlock what's next" framing is exactly how build tools and package managers resolve install/compile order
- **Next chapter:** Weighted graphs and shortest-path algorithms

CHAPTER 6 OF 10

Shortest Path Algorithms: Dijkstra's Algorithm

Graph Theory

Chapter 6 · Shortest Path Algorithms: Dijkstra's Algorithm

Every graph so far has been unweighted — Chapter 3's own finding-box already established that BFS finds the shortest path by *edge count*. Real-world graphs are usually weighted: a road has a distance, a network link has latency, a flight has a price. "Fewest edges" and "cheapest total weight" are genuinely different questions once weights enter the picture, and this chapter builds the algorithm that answers the second one.

Weighted Graphs: One Small Change to the Representation

A weighted graph attaches a number to every edge. The adjacency list from Chapter 2 barely changes — instead of storing just a neighbor, each entry stores a `(neighbor, weight)` pair: `adj['A'] = [('B', 4), ('C', 1)]` means A connects to B with weight 4 and to C with weight 1.

Dijkstra's Algorithm: Always Expand the Closest Unfinished Vertex

Dijkstra's algorithm is greedy: it repeatedly finalizes whichever not-yet-finalized vertex currently has the smallest known distance from the source, then uses that vertex to try to improve ("relax") the distances of its neighbors.

THE MECHANISM

Keep a running **distance table** (source = 0, everything else = infinity) and a **priority queue** keyed by current known distance. Repeatedly pop the smallest-distance unfinished vertex, finalize it, then for each neighbor: if `(finalized vertex's distance) + (edge weight)` is smaller than the neighbor's current recorded distance, update it and push the neighbor back onto the queue.

Graph: `A→B` (4), `A→C` (1), `C→B` (2), `B→D` (1), `C→D` (5), `D→E` (3). Finding shortest distances from A:

TRACED AND VERIFIED DIRECTLY

Pop A(0) → finalize A. Relax B: 0+4=4. Relax C: 0+1=1.
Pop C(1) → finalize C. Relax B: 1+2=3, which beats B's current 4 — **update B to 3**. Relax D: 1+5=6.
Pop B(3) → finalize B. Relax D: 3+1=4, which beats D's current 6 — **update D to 4**.
Pop D(4) → finalize D. Relax E: 4+3=7.

Pop E(7) → finalize E.
Final distances from A: `{A:0, B:3, C:1, D:4, E:7}`.

Notice B was updated twice — first to 4 (direct edge from A), then to a cheaper 3 once C was finalized and revealed the shorter route A→C→B. This is the entire point of the greedy strategy: finalize the closest vertex first, because nothing still in the queue can possibly offer it a cheaper route later — every other candidate is already at least as far away.

BFS Was Secretly Dijkstra All Along

CONNECTING BACK TO CHAPTER 3

If every edge weight is exactly `1`, Dijkstra's algorithm reduces exactly to Chapter 3's own BFS — the priority queue's "always pop the smallest distance" behavior becomes identical to a plain FIFO queue's "always process the next level," since every step increases distance by the same fixed amount. BFS is a special case of Dijkstra, not a separate idea.

Why It Breaks With Negative Weights

Dijkstra's greedy step assumes that once a vertex is finalized, nothing seen later can ever offer it (or anything already routed through it) a cheaper path. A negative edge weight breaks that assumption directly — a big negative edge discovered *after* a vertex has already been finalized can retroactively make an already-"settled" distance wrong, and Dijkstra never revisits a finalized vertex to fix it.

Graph: `A→B` (1), `A→C` (4), `C→B` (−10), `B→D` (100).

A VERIFIED COUNTEREXAMPLE

Dijkstra pops B(1) first (smaller than C's 4) and **finalizes B at distance 1**, immediately using it to set D's distance to 1+100=101. Only afterward does it pop C(4) and discover C→B=4−10=−6, a genuinely shorter route to B — but B is already finalized, so this correction never propagates onward to D. Dijkstra reports D=101. The true shortest A→D, found by checking every actual path, is **94** (via A→C→B→D = 4−10+100). Dijkstra's own final table even shows B=−6 — updated in the raw data even though the vertex was already "settled" — making the error easy to miss unless D is checked directly.

This is exactly why Dijkstra requires non-negative weights, and exactly the gap Chapter 7's own Bellman-Ford algorithm is built to close.

Real Relevance: Routing & Navigation

GPS and mapping software runs some form of Dijkstra (or a close relative) to find a shortest route by distance or time — road segments as edges, intersections as vertices, travel time or distance as weight. Network

routers use a close cousin (OSPF/link-state routing) to find the cheapest path for data by latency and hop cost, with "cheapest" playing the same role weight plays here.

Dijkstra's Algorithm in Code

```
import heapq

def dijkstra(adj, source):
    dist = {v: float('inf') for v in adj}
    dist[source] = 0
    visited = set()
    pq = [(0, source)]
    while pq:
        d, u = heapq.heappop(pq)
        if u in visited:
            continue              # stale queue entry, already finalized
        visited.add(u)
        for v, w in adj[u]:
            if d + w < dist[v]:
                dist[v] = d + w
                heapq.heappush(pq, (dist[v], v))
    return dist

roads = {
    'A': [('B',4), ('C',1)],
    'B': [('D',1)],
    'C': [('B',2), ('D',5)],
    'D': [('E',3)],
    'E': [],
}
print(dijkstra(roads, 'A'))  # {'A':0, 'B':3, 'C':1, 'D':4, 'E':7}
```

Hands-On Exercises

EXERCISE 1

Using Dijkstra's algorithm, trace the shortest distances from source P for the weighted graph: `P→Q` (2), `P→R` (9), `Q→R` (3), `R→S` (1). Show each pop-and-relax step, including any distance updates.

View solution

EXERCISE 2

Explain, in terms of this chapter's own greedy assumption, why Dijkstra's algorithm never needs to reconsider a vertex once it has been popped and finalized — as long as every edge weight is non-negative.

View solution

EXERCISE 3

This chapter's negative-weight counterexample showed Dijkstra reporting `D=101` when the true shortest distance is `94`, while the same run's final table showed `B=-6`, the actually-correct value. Explain why the algorithm ends up with a correct number for B but an incorrect one for D in the very same run.

View solution

CHAPTER 6 QUICK REFERENCE

- **Weighted adjacency list:** each entry is a `(neighbor, weight)` pair rather than just a neighbor
- **Dijkstra's algorithm:** repeatedly finalize the closest unfinished vertex, then relax its neighbors' distances
- BFS is exactly Dijkstra's algorithm when every edge weight is `1`
- **Requires non-negative weights** — a negative edge can retroactively invalidate an already-finalized distance, which Dijkstra never revisits
- Real use: GPS/mapping route-finding and link-state network routing
- **Next chapter:** Bellman-Ford — shortest paths that work with negative weights

CHAPTER 7 OF 10

Shortest Path Algorithms: Bellman-Ford & Negative Weights

Graph Theory

Chapter 7 · Shortest Path Algorithms: Bellman-Ford & Negative Weights

Chapter 6 ended with a genuine failure: Dijkstra reported `D=101` for a graph where the true shortest distance was `94`, because a negative edge corrected a vertex's distance *after* that vertex had already been greedily finalized. This chapter fixes exactly that failure — with an algorithm that never finalizes anything early enough to get caught out.

The Idea: Stop Trusting Any Distance Until Enough Rounds Have Passed

Where Dijkstra finalizes one vertex at a time and never looks back, **Bellman-Ford** takes the opposite approach: it relaxes *every single edge in the graph*, repeatedly, for a fixed number of rounds — giving corrections like the one that broke Dijkstra time to propagate all the way through, no matter how many extra hops away they start.

THE MECHANISM

Initialize distances exactly as before (source = 0, everything else = infinity). Repeat `V - 1` times (V = number of vertices): for every edge `(u, v, w)` in the graph, if `dist[u] + w < dist[v]`, update `dist[v]`. No priority queue, no "finalizing" — just brute-force relaxation, over and over.

Why Exactly V − 1 Rounds?

A shortest path that doesn't repeat any vertex (a "simple" path) can use at most `V - 1` edges — there are only `V` vertices to visit at all. Each full round of relaxing every edge guarantees that any shortest path using up to that many edges gets found: round 1 finds every shortest path of 1 edge, round 2 extends those into every shortest path of up to 2 edges, and so on. After `V - 1` rounds, every possible simple shortest path — however many edges it needs — has been fully accounted for.

Fixing Chapter 6's Own Counterexample

Same graph that broke Dijkstra: `A→B` (1), `A→C` (4), `C→B` (−10), `B→D` (100). 4 vertices, so 3 rounds.

TRACED AND VERIFIED DIRECTLY

Relaxing every edge, in order, during round 1: A→B gives B=1. A→C gives C=4. C→B: 4−10=−6, beats B's current 1 — **B corrected to −6**. B→D: −6+100=**94**. Round 2 finds nothing left to improve — the algorithm has already converged. Final distances: `{A:0, B:-6, C:4, D:94}`. D is now correctly **94**, not Dijkstra's wrong 101 — because B's correction happened *before* B→D was ever relied on again, simply by relaxing every edge in the same pass rather than committing to B's value early.

THE EDGE PROCESSING ORDER STILL MATTERS WITHIN A ROUND

Processing the edges in a less convenient order (e.g. `B→D` listed before `C→B`) means D doesn't get its correct value until round 2 instead of round 1 — verified directly. Either way, by the time all `V−1` rounds are done, the answer is guaranteed correct regardless of what order the edges happen to be listed in — only the number of rounds needed to get there changes, not the final result.

The Extra Round: Detecting Negative Cycles

Everything above assumes a shortest *simple* path exists at all. If the graph contains a **negative cycle** — a cycle whose total edge weight sums to less than zero — there is no shortest path at all, since going around the cycle again and again keeps reducing the total cost forever. Bellman-Ford detects this directly: run one extra (a `V`-th) round of relaxation — if *any* edge can still be relaxed after the normal `V−1` rounds finished, a negative cycle reachable from the source must exist.

Graph: `A→B` (1), `B→C` (−1), `C→A` (−1) — a triangle whose total weight is `1 + (-1) + (-1) = -1`.

VERIFIED DIRECTLY

After the normal 2 rounds (V=3), distances are still shrinking. Running the extra check round finds edge `A→B` still relaxable (`dist[A]+1 < dist[B]`) — the negative-cycle signature. Correctly flagged: **negative cycle detected**.

Real Relevance: Currency Arbitrage Detection

This is Bellman-Ford's single most-cited real application: model each currency as a vertex, each exchange rate as an edge weighted `-log(rate)`. A cycle of exchanges is profitable exactly when the product of its rates exceeds 1 — which, because `-log` turns multiplication into addition, is exactly when the sum of `-log(rate)` around that cycle is **negative**. A negative cycle in this graph is a real, mechanically-detectable arbitrage opportunity.

VERIFIED DIRECTLY

USD→EUR at 0.9, EUR→GBP at 0.8, GBP→USD at 1.5. Product: `0.9 × 0.8 × 1.5 = 1.08` — trading through the full cycle turns $1 into $1.08, an 8% profit. Sum of `-log(rate)` around the same cycle: `-0.077` — negative, exactly as the theory predicts, and exactly what Bellman-Ford's negative-cycle check would flag.

When to Reach for Bellman-Ford Instead of Dijkstra

	DIJKSTRA	BELLMAN-FORD
Negative weights	Unsupported — silently wrong (Chapter 6)	Fully supported
Negative cycles	No detection at all	Detects them directly
Time complexity	O((V+E) log V) with a priority queue	O(V × E) — every edge, every round
Use when	All weights are known to be non-negative (the common case — distances, times, costs)	Negative weights are possible, or a negative cycle itself needs detecting

Bellman-Ford is strictly more capable, but genuinely slower on large graphs — the right default is still Dijkstra whenever negative weights are known to be impossible, reaching for Bellman-Ford specifically when they aren't.

Bellman-Ford in Code

```
def bellman_ford(vertices, edges, source):
    dist = {v: float('inf') for v in vertices}
    dist[source] = 0

    for _ in range(len(vertices) - 1):
        for u, v, w in edges:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # extra round: still-relaxable edge = negative cycle
    for u, v, w in edges:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            return dist, True   # negative cycle found

    return dist, False

vertices = ['A','B','C','D']
edges = [('A','B',1), ('A','C',4), ('C','B',-10), ('B','D',100)]
print(bellman_ford(vertices, edges, 'A'))  # ({'A':0,'B':-6,'C':4,'D':94}, False)
```

Hands-On Exercises

EXERCISE 1

Using Bellman-Ford, find the shortest distances from source S for: `S→T` (6), `S→U` (7), `T→U` (−3), `U→V` (2). Show each round, and how many rounds it actually takes to stop changing.

View solution

EXERCISE 2

Run Bellman-Ford's negative-cycle check on the graph `P→Q` (2), `Q→R` (2), `R→P` (−5). Determine whether a negative cycle exists, and if so, state its total weight.

View solution

EXERCISE 3

A trader finds three exchange rates: USD→JPY at 110, JPY→GBP at 0.0068, GBP→USD at 1.3. Using this chapter's own arbitrage-detection method, determine whether trading through this full cycle is profitable, and by how much.

View solution

CHAPTER 7 QUICK REFERENCE

- **Bellman-Ford:** relax every edge in the graph, `V−1` times — no priority queue, no early finalizing
- `V−1` rounds is exactly enough because a simple shortest path can use at most `V−1` edges
- **Negative cycle detection:** if any edge is still relaxable after `V−1` rounds, a negative cycle exists
- Correctly solves Chapter 6's own counterexample — D=94, not Dijkstra's wrong 101
- **Currency arbitrage:** weight edges by `-log(rate)`; a negative cycle is a real, detectable profit opportunity
- O(V×E), slower than Dijkstra's O((V+E) log V) — use Dijkstra by default, Bellman-Ford only when negative weights are possible
- **Next chapter:** Minimum Spanning Trees — Kruskal's & Prim's algorithms

CHAPTER 8 OF 10

Minimum Spanning Trees: Kruskal's & Prim's Algorithms

Graph Theory

Chapter 8 · Minimum Spanning Trees: Kruskal's & Prim's Algorithms

Chapters 6 and 7 answered "what's the cheapest way from one vertex to the others?" This chapter asks a genuinely different question: given a weighted, undirected, connected graph, what's the cheapest possible way to connect **every** vertex to every other, using as few edges as it takes and nothing more?

What a Minimum Spanning Tree Actually Is

A **spanning tree** of a connected graph with `V` vertices is a subgraph that connects all `V` vertices using exactly `V - 1` edges, with no cycles — precisely the tree definition Chapter 4's own finding-box already established. A **Minimum Spanning Tree (MST)** is the spanning tree whose edge weights sum to the smallest possible total, out of every spanning tree the graph could have.

TWO GENUINELY DIFFERENT GREEDY STRATEGIES

Kruskal's algorithm is greedy on *edges* — always consider the cheapest remaining edge, anywhere in the graph. **Prim's algorithm** is greedy on the *frontier* — always grow the tree by whichever cheapest edge connects something already in the tree to something not yet in it (a close cousin of Chapter 6's own Dijkstra).

Method 1: Kruskal's Algorithm

Sort every edge in the graph by weight, ascending. Go through them one at a time: add an edge to the MST unless both of its endpoints are *already connected* to each other through edges already added — adding it anyway would create a cycle, which a tree can never have.

CHECKING "ALREADY CONNECTED" WITH UNION-FIND

A **Union-Find** (Disjoint Set) structure tracks which component each vertex currently belongs to. `find(x)` returns x's component; `union(x, y)` merges two components into one. An edge is safe to add exactly when `find(u) != find(v)` — its two endpoints are still in different components.

Graph: `A-B` (4), `A-C` (2), `B-C` (1), `B-D` (5), `C-D` (8), `C-E` (10), `D-E` (2), `B-E` (6).

TRACED AND VERIFIED DIRECTLY, IN ASCENDING WEIGHT ORDER

`B-C` (1): different components — **add**. `A-C` (2): different components — **add**. `D-E` (2): different components — **add**. `A-B` (4): A and B already connected via B-C-A — **skip**, would form a cycle. `B-D` (5): different components (the {A,B,C} group and the {D,E} group) — **add**. Every remaining edge (`B-E` , `C-D` , `C-E`) now connects vertices already in the same component — all skipped. Final MST: `{B-C, A-C, D-E, B-D}` , total weight **10**.

Method 2: Prim's Algorithm

Start from any single vertex — it's the entire "tree" so far. Repeatedly find the cheapest edge connecting a vertex already in the tree to one that isn't, add that edge and vertex, and repeat until every vertex is included. A priority queue does the "cheapest available edge" lookup, exactly like Dijkstra's own priority queue found the cheapest available distance.

Same graph, starting from A:

TRACED AND VERIFIED DIRECTLY

Frontier from {A}: cheapest is `A-C` (2) — add C. Frontier from {A,C}: cheapest is `C-B` (1) — add B. Frontier from {A,B,C}: cheapest is `B-D` (5) — add D. Frontier from {A,B,C,D}: cheapest is `D-E` (2) — add E. Final MST: `{A-C, C-B, B-D, D-E}` , total weight **10** — the exact same total, and in fact the exact same set of edges, as Kruskal's algorithm found, just discovered in a different order.

SAME TOTAL WEIGHT IS GUARANTEED — THE EXACT EDGES USUALLY AREN'T

Both algorithms are provably correct, so they always agree on the **total** MST weight. When a graph has no weight ties among the edges that matter, as here, they typically land on the identical edge set too — but with ties present, they can legitimately pick *different* edges of equal weight and still both be correct, exactly like the non-unique topological orders from Chapter 5.

Real Relevance: Network Design

Laying fiber or cable to connect a set of offices, data centers, or towns at minimum total cost is precisely the MST problem — vertices are sites, edge weight is the cost of a direct link between two sites, and the MST is the cheapest possible way to make sure every site can reach every other (even if only indirectly, through other sites on the tree). The same shape shows up in circuit board wiring and in clustering algorithms that group data points by cutting the most expensive edges out of an MST.

Kruskal's & Prim's in Code

```
def kruskal(vertices, edges):
    parent = {v: v for v in vertices}
    def find(x):
        while parent[x] != x:
            x = parent[x]
        return x
    def union(x, y):
        parent[find(x)] = find(y)

    mst, total = [], 0
    for u, v, w in sorted(edges, key=lambda e: e[2]):
        if find(u) != find(v):
            union(u, v)
            mst.append((u, v, w))
            total += w
    return mst, total

import heapq
def prim(vertices, adj, start):
    visited = {start}
    heap = list(adj[start])   # (weight, u, v) tuples
    heapq.heapify(heap)
    mst, total = [], 0
    while heap and len(visited) < len(vertices):
        w, u, v = heapq.heappop(heap)
        if v in visited:
            continue
        visited.add(v)
        mst.append((u, v, w))
        total += w
        for w2, u2, v2 in adj[v]:
            if v2 not in visited:
                heapq.heappush(heap, (w2, v, v2))
    return mst, total
```

Hands-On Exercises

EXERCISE 1

Using Kruskal's algorithm, find the MST of the graph with edges `P-Q` (3), `P-R` (1), `Q-R` (4), `Q-S` (2), `R-S` (5). Show which edges are added and which are skipped, and the total weight.

View solution

EXERCISE 2

Four offices need network cabling: `Office1-Office2` (8), `Office1-Office3` (5), `Office2-Office3` (3), `Office2-Office4` (9), `Office3-Office4` (4). Using Prim's algorithm starting from Office1, find the minimum-cost cabling plan and its total cost.

View solution

EXERCISE 3

Explain why Kruskal's algorithm needs a Union-Find structure to detect cycles, while Prim's algorithm never risks creating a cycle at all — even without any cycle-checking step of its own.

View solution

CHAPTER 8 QUICK REFERENCE

- **Spanning tree:** connects all V vertices using exactly V−1 edges, no cycles (Chapter 4's own tree definition)
- **Kruskal's algorithm:** sort all edges by weight; add each one unless it connects two vertices already in the same component (Union-Find)
- **Prim's algorithm:** grow one tree from a start vertex, always adding the cheapest edge on the current frontier — a close cousin of Dijkstra
- Both algorithms always agree on the total MST weight; they agree on the exact edges too unless weight ties allow more than one valid MST
- Real use: minimum-cost network/cable design, circuit wiring, clustering
- **Next chapter:** Trees as a special case of graphs — structure and traversal

CHAPTER 9 OF 10

Trees as Special Graphs: Structure & Traversal

Graph Theory

Chapter 9 · Trees as Special Graphs: Structure & Traversal

Trees have already shown up twice without ever being the main subject — Chapter 4's own finding-box noted that a tree with `V` vertices has exactly `V−1` edges and no cycles, and Chapter 8 built an entire algorithm family around finding one. This chapter finally gives trees their own spotlight: what they're made of, and how the traversal ideas from Chapters 3 and 5 turn out to already be exactly what a tree needs.

A Tree Is Just a Connected, Acyclic Graph

A **(free) tree** is an undirected graph that is both **connected** (every vertex reachable from every other) and **acyclic** (no cycles at all — not even the parent-tracking exception from Chapter 4's undirected cycle check, since a tree has no cycles whatsoever). Any two of the following three properties automatically guarantee the third, for a graph with `V` vertices:

GIVEN	THEN AUTOMATICALLY
Connected + acyclic	Exactly `V−1` edges
Connected + exactly `V−1` edges	Acyclic
Acyclic + exactly `V−1` edges	Connected

VERIFIED ON THIS CHAPTER'S OWN WORKED EXAMPLE

The tree below has `V=7` vertices and `E=6` edges — exactly `V−1`, matching every one of the equivalences above.

Rooted Trees: Adding a Direction

A free tree has no designated starting point. Picking one vertex as the **root** turns it into a **rooted tree**, which unlocks a whole vocabulary: the root's neighbors become its **children**, and it becomes their **parent**; a vertex with no children is a **leaf**; a vertex's **depth** is its distance from the root; a tree's **height** is the greatest depth any vertex reaches; the vertex and everything reachable below it form a **subtree**.

Worked example, rooted at A: `A→{B,C,D}`, `B→{E,F}`, `D→{G}` (C, E, F, G are leaves).

VERIFIED DIRECTLY

Depths: A=0, B=1, C=1, D=1, E=2, F=2, G=2. Height of the tree: **2** (the deepest leaves, E/F/G, are 2 steps from the root).

SCOPE NOTE: BINARY TREES

A **binary tree** restricts every node to at most 2 children (conventionally "left" and "right") — the structure behind binary search trees and heaps. Balancing, searching, and insertion into binary trees are their own substantial topic and stay out of this course's scope; what matters here is that they're still just rooted trees, and everything below still applies to them directly.

Traversal Orders: Not New Algorithms, Just Named Moments

A tree has exactly one simple path between any two vertices, so DFS never needs Chapter 4's back-edge check at all — there's nothing to loop back on. On a *rooted* tree, the only real choice left is **when** to process ("visit") a node relative to visiting its children:

ORDER	WHEN THE NODE IS VISITED	ALREADY MET AS
Pre-order	Before any of its children	The order Chapter 3's DFS naturally visits vertices in
Post-order	After all of its children	Exactly Chapter 5's own DFS **finish order** — the stack topological sort was built on
Level-order	By depth, shallowest first	Exactly Chapter 3's own BFS

TRACED AND VERIFIED DIRECTLY, ON THE SAME ROOTED TREE ABOVE

Pre-order: `A, B, E, F, C, D, G` — each node written down the instant it's first reached.
Post-order: `E, F, B, C, G, D, A` — each node written down only once every child beneath it is done, the root inevitably last.
Level-order (BFS): `A, B, C, D, E, F, G` — depth 0, then all of depth 1, then all of depth 2.

No new machinery was needed for any of these — pre-order is DFS with no back-edges to worry about, post-order is Chapter 5's finish-order stack read directly instead of reversed (a tree, being acyclic, never needs the reversal that made a topological order valid), and level-order is Chapter 3's BFS applied to a graph that happens to be a tree.

Real Relevance: File Systems, Org Charts & the DOM

A file system is a rooted tree — folders as internal nodes, files as leaves; walking it top-down before descending into each subfolder (exactly how a tool like `os.walk` or a recursive directory listing behaves) is pre-order traversal. An org chart is a rooted tree of reporting relationships. A web page's DOM is a rooted tree of elements, and a JSON or XML document parses into one too — nested objects and arrays as subtrees. Anywhere a "contains" or "reports to" relationship never loops back on itself, the natural model is a tree, and one of these three traversal orders is almost always the right way to walk it.

Tree Traversal in Code

```
def preorder(tree, node, out=None):
    if out is None: out = []
    out.append(node)                      # visit BEFORE children
    for child in tree[node]:
        preorder(tree, child, out)
    return out

def postorder(tree, node, out=None):
    if out is None: out = []
    for child in tree[node]:
        postorder(tree, child, out)
    out.append(node)                      # visit AFTER children
    return out

from collections import deque
def level_order(tree, root):
    out, q = [], deque([root])
    while q:
        node = q.popleft()
        out.append(node)
        for child in tree[node]:
            q.append(child)
    return out

org_chart = {'A': ['B','C','D'], 'B': ['E','F'], 'C': [], 'D': ['G'], 'E': [], 'F': [], 'G': []}
print(preorder(org_chart, 'A'))      # ['A','B','E','F','C','D','G']
print(postorder(org_chart, 'A'))     # ['E','F','B','C','G','D','A']
print(level_order(org_chart, 'A'))   # ['A','B','C','D','E','F','G']
```

Hands-On Exercises

EXERCISE 1

A file system has: `root→{docs, src}`, `docs→{readme.txt}`, `src→{main.py, utils.py}`. Write out the pre-order and post-order traversals starting from `root`, and state which one matches how a real directory listing tool typically prints folders before descending into them.

View solution

EXERCISE 2

For the same file system tree from Exercise 1, compute the depth of every node and state the tree's height.

View solution

EXERCISE 3

Explain, using this chapter's own definitions, why post-order traversal always visits the root last, and why pre-order traversal always visits the root first — for *any* rooted tree, not just this chapter's specific examples.

View solution

CHAPTER 9 QUICK REFERENCE

- **Tree:** a connected, acyclic graph — any two of {connected, acyclic, `V−1` edges} imply the third
- **Rooted tree terms:** root, parent, child, leaf, depth (distance from root), height (max depth), subtree
- **Pre-order:** visit before children — Chapter 3's DFS order
- **Post-order:** visit after children — exactly Chapter 5's DFS finish-order stack
- **Level-order:** visit by depth — exactly Chapter 3's BFS
- Real use: file systems (pre-order), org charts, the DOM, JSON/XML structure
- **Next chapter:** Capstone — modeling and solving a real graph problem

CHAPTER 10 OF 10

Capstone — Modeling and Solving a Real Graph Problem

Graph Theory

Chapter 10 · Capstone — Modeling and Solving a Real Graph Problem

One continuous scenario, touching every chapter of this course in the order a real engineer would actually reach for each idea: **Nimbus**, a small startup, is standing up its own office and infrastructure — and every genuinely different problem that comes up along the way turns out to be a graph problem already covered by this course.

STEP	PROBLEM	CHAPTER(S) USED
1	Modeling the office & checking it's fully wired	Ch.2 (representation), Ch.3-4 (traversal, connectivity)
2	Cheapest dedicated cable run for one camera line	Ch.6 (Dijkstra)
3	Cheapest way to wire every room, not just one line	Ch.8 (Kruskal's MST)
4	Is the deployment plan even valid? What order?	Ch.4-5 (cycle check, topological sort)
5	Fastest request path through a caching layer	Ch.6-7 (Dijkstra fails, Bellman-Ford fixes it)
6	Reusing the org chart three different ways	Ch.9 (pre/post/level-order)

Step 1 — The Office Network: Representation & Connectivity

CH.2 · CH.3-4

Nimbus has six rooms — Lobby, Eng, Design, Sales, Server (the server closet), Kitchen — with possible direct cable runs between some pairs, weighted by meters of cable: `Lobby-Eng` (12), `Lobby-Sales` (8), `Lobby-Kitchen` (5), `Eng-Design` (6), `Eng-Server` (15), `Design-Server` (20), `Sales-Kitchen` (10), `Server-Kitchen` (18). Modeled exactly per Chapter 2: a weighted adjacency list, one entry per room.

VERIFIED DIRECTLY

Running Chapter 3's own BFS from Lobby reaches all six rooms — `{Lobby, Eng, Design, Sales, Server, Kitchen}` — confirming Chapter 4's connectivity check: the network is a single connected component before any wiring decisions are even made.

Step 2 — One Dedicated Line: Cheapest Cable Run to the Server

CH.6

A security camera needs its own dedicated line from the Lobby to the Server closet. Every weight here is real cable length — genuinely non-negative — so Dijkstra is not just usable but the right, efficient default per Chapter 6's own guidance.

VERIFIED DIRECTLY

Dijkstra from Lobby: `{Lobby:0, Sales:8, Kitchen:5, Eng:12, Design:18, Server:23}`. Shortest route to Server: `Lobby → Kitchen → Server`, 5+18=**23m** — cheaper than the direct-looking `Lobby → Eng → Server` route (12+15=27m).

Step 3 — Wiring Every Room: A Genuinely Different Question

CH.8

Step 2 optimized a single point-to-point connection. Wiring the *whole office* is a different problem entirely — a Minimum Spanning Tree, per Chapter 8, minimizes total cable across every room at once, with no requirement that any particular pair's own route be the cheapest possible.

VERIFIED DIRECTLY, VIA KRUSKAL'S ALGORITHM

Sorted edges, skipping any that would close a cycle: `Lobby-Kitchen` (5), `Eng-Design` (6), `Lobby-Sales` (8), `Lobby-Eng` (12), `Eng-Server` (15) — total **46m**.

THE MST DOESN'T EVEN USE STEP 2'S OWN SHORTEST ROUTE

The MST connects Server via `Eng-Server` (15), reached through `Lobby-Eng` (12) — a combined 27m path to Server, worse than Step 2's dedicated 23m route through Kitchen. That's not a mistake: the MST is minimizing the total cost of connecting *everything*, and re-using the already-cheap Lobby-Eng-Design backbone costs less overall than also paying for Kitchen-Server(18) on top of Lobby-Kitchen(5). Shortest path and MST optimize genuinely different quantities, and can disagree on individual routes even while both being correct for what they're actually solving.

Step 4 — The Deployment Graph: Is the Plan Even Valid?

CH.4 (DIRECTED) · CH.5

Nimbus's services depend on each other: `auth→api`, `database→api`, `cache→api`, `api→frontend`, `api→notifications`. Before deploying anything, Chapter 4's directed cycle check confirms this is actually possible.

VERIFIED DIRECTLY

Three-color DFS finds no back edge — **no cycle**. Chapter 5's own Kahn's algorithm then gives a valid deployment order: `auth, cache, database, api, frontend, notifications` (the three zero-dependency services first, in whatever order, then `api` once all three of its dependencies are satisfied, then its own two dependents last).

A PROPOSED CHANGE, CORRECTLY REJECTED

A teammate proposes routing login alerts through notifications, adding `notifications→auth`. Re-running the cycle check catches it immediately: `auth → api → notifications → auth` is a genuine back edge — **cycle detected**, exactly Chapter 4's own package-manager scenario played out for real. The proposal is reworked to decouple the alert (e.g. via a message queue) rather than adding a direct dependency edge.

Step 5 — Request Latency: Where Dijkstra Actually Breaks

CH.6-7

Modeling one request's path in milliseconds: `gateway→auth` (5), `gateway→api` (12), `auth→api` (3), `api→database` (2), `api→cache` (5), `cache→database` (−15), `database→response` (4). The cache edge is negative — a cache hit genuinely saves more time than it costs to check, exactly the kind of real negative weight Chapter 7 introduced.

DIJKSTRA, TRACED DIRECTLY — GENUINELY WRONG HERE

Dijkstra finalizes `database=10` (via `api`) before ever reaching `cache`, so `response` gets relaxed using that stale value: `10+4=14ms`. Only afterward does `cache` (13) get processed, correcting `database`'s own table entry to `13−15=−2` — but `database` is already finalized, so `response` never benefits. Dijkstra reports **14ms** — the exact same self-correcting-vertex, stuck-downstream-value failure pattern as Chapter 6's own B/D counterexample, playing out again here.

BELLMAN-FORD, VERIFIED DIRECTLY

Relaxing every edge for `V−1=5` rounds: `{gateway:0, auth:5, api:8, cache:13, database:−2, response:2}`. True fastest path: **2ms**, not Dijkstra's wrong 14ms. The extra check round confirms **no negative cycle** — a real safety check here, since a negative cycle in a request-latency graph would mean a bug allowing infinite, unbounded "savings."

Step 6 — The Org Chart, Walked Three Different Ways

CH.9

Nimbus's reporting tree: `CEO→{CTO, VP_Sales}`, `CTO→{Eng_Lead, Design_Lead}`, `Eng_Lead→{Dev1, Dev2}`, `VP_Sales→{Sales_Rep}`. The same tree, walked three genuinely different ways for three genuinely different real needs.

VERIFIED DIRECTLY

Pre-order (onboarding checklist, top managers introduced before their reports): `CEO, CTO, Eng_Lead, Dev1, Dev2, Design_Lead, VP_Sales, Sales_Rep`.
Level-order (budget-approval escalation, by seniority): `CEO, CTO, VP_Sales, Eng_Lead, Design_Lead, Sales_Rep, Dev1, Dev2`.
Post-order, used for a headcount rollup (`team_size[node] = 1 + sum of children's team_size`, each computed only after its children are done): `Dev1=1, Dev2=1, Eng_Lead=3, Design_Lead=1, CTO=5, Sales_Rep=1, VP_Sales=2, CEO=8` — the CEO's own total of **8** correctly matches the tree's actual 8 people, a direct payoff of post-order's own "children before parent" guarantee from Chapter 9.

What This Course Doesn't Cover

As stated honestly back in Chapter 1: network flow, graph coloring, planarity, and graph-database technology specifically were named as out of scope, and stayed out of scope through all ten chapters. This course built the core representational and algorithmic toolkit — traversal, connectivity, ordering, shortest paths, spanning trees, and tree structure — not an exhaustive catalog of every named graph algorithm.

Where This Course Connects

This course leaned directly on Algorithms & Complexity's own recursion and Big-O machinery throughout — Chapter 1 named this destination explicitly, and Chapters 3 through 8 all reused it for analyzing traversal and shortest-path cost. Discrete Mathematics Fundamentals' own relations material (Chapter 5 there) is the formal ancestor of a graph itself — this course is, in a real sense, what happens when a relation gets a name and a picture. Within Technical Support, `netdiag1`'s own network-troubleshooting material and this course's shortest-path/connectivity chapters describe the same underlying kind of structure from two different angles — one diagnostic, one mathematical.

Hands-On Exercises

EXERCISE 1

Nimbus adds a seventh room, Storage, connected only by `Design-Storage` (7). Using this chapter's own Step 3 MST as a starting point, state the new MST's total weight and explain why adding one new leaf edge to an existing MST never requires re-examining any of the other edges already chosen.

View solution

EXERCISE 2

A new service, `logging`, needs to depend on `api` (edge `api→logging`), and nothing needs to depend on `logging` in return. Using this chapter's own Step 4 deployment graph plus this new edge, give a full valid deployment order via Kahn's algorithm, showing the in-degree of every service at the start.

View solution

EXERCISE 3

Step 5 showed Dijkstra reporting 14ms for `gateway→response` when the true value is 2ms. Explain specifically why Dijkstra's own `cache` distance (13) and `database` distance (−2, after correction) both end up numerically correct in the final table, despite the overall `response` result being wrong — reusing this chapter's own explanation pattern from Step 5, not just restating that Dijkstra "doesn't work" with negative weights.

View solution

CHAPTER 10 QUICK REFERENCE

- **Full worked project:** representation & connectivity (Ch.2-4) → Dijkstra for one route (Ch.6) → Kruskal's MST for every room (Ch.8) → cycle check & topological deployment order (Ch.4-5) → Dijkstra fails, Bellman-Ford fixes it (Ch.6-7) → one tree, three traversal purposes (Ch.9)
- Shortest path and MST are genuinely different questions — they can and did disagree on Server's own best route
- The same negative-weight failure pattern from Chapter 6 (a self-correcting vertex whose correction never reaches a downstream neighbor) reproduced exactly in a completely different scenario
- Post-order's "children before parent" guarantee (Ch.9) is what makes a bottom-up rollup like a headcount total actually correct
- Out of scope: network flow, graph coloring, planarity, graph-database technology
- **Course complete** — Graph Theory, 10 chapters, from representation to a full worked infrastructure project