Ricerca Operativa

Lecture Notes

Lucas Simonetti

Università degli Studi di Udine · A.A. 2025–2026

Contents

CHAPTER 1

Introduction to OR & Modeling

This chapter sets the stage for the entire course: what Operations Research is, why it matters, and the language we use to talk about it.

Operations Research is, at its heart, about making *good decisions*. Not vague, gut-feeling decisions, but decisions backed by mathematical models, algorithms, and rigorous analysis. In this chapter we lay the groundwork: we will see what OR is, learn the vocabulary of problems and instances, define optimization and decision problems formally, touch on computational difficulty, survey the breadth of applications, and introduce the mathematical programming framework that will serve us for the rest of the course.

Along the way, we will also introduce three running examples—a small linear program, a directed graph, and a knapsack instance—that will reappear in virtually every subsequent chapter.

1.1 What is Operations Research?

Suppose you run a small furniture workshop. You can produce chairs and tables. Each product requires wood, labour, and finishing time—all of which are limited. Chairs bring in a certain profit per unit, tables another. *How many chairs and tables should you produce to maximize profit?*

This "furniture workshop" will be our recurring 2-variable LP. We will solve it graphically in Chapter 2 and with the Simplex method in Chapter 4.

That question is an Operations Research question. More precisely:

> **Definition 1.1.1** (Operations Research — informal)**.** Operations Research (OR) is the discipline that applies **mathematical models**, **analytical methods**, and **algorithmic design** to help make better decisions in complex real-world systems.

Let's unpack this piece by piece. "Mathematical models" means we translate a messy real-world situation (machines, deadlines, budgets) into equations and inequalities. "Analytical methods" means we use tools from linear algebra, graph theory, probability, and combinatorics to study those equations. "Algorithmic design" means we build step-by-step procedures that a computer can execute to find—or approximate—the best solution.

The overall process follows a cycle that we will revisit many times:

Step 1. Real-world problem. Identify the decisions, constraints, and objective in a concrete scenario.

Step 2. Formalization. Strip away irrelevant details, define variables, parameters, constraints, and objective function.

Step 3. Mathematical model. Write down the resulting optimization (or decision) problem.

Step 4. Algorithm. Choose or design a method to solve the model—exactly or approximately.

Step 5. Solution & interpretation. Translate the mathematical answer back into real-world terms, validate, and implement.

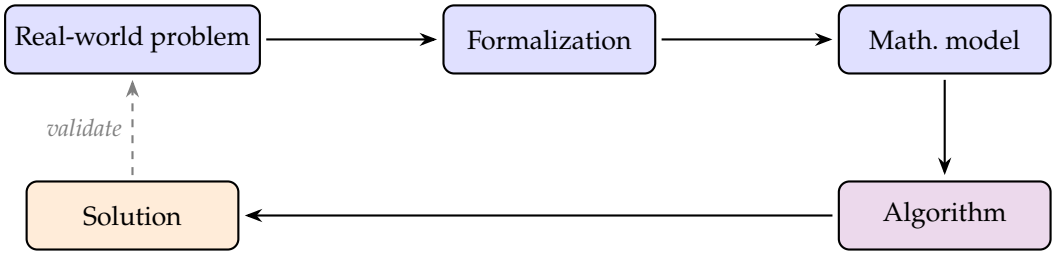


Figure 1.1: The OR modelling cycle.

OR draws on a broad blend of mathematical and computational skills: **graph theory** (shortest paths, matchings, flows), **linear algebra** (the Simplex method lives in matrix land), **computational complexity** (which problems can we solve efficiently?), and **combinatorics** (counting, enumeration, structural arguments).

Intermezzo — A Brief History of Operations Research

The name "Operations Research" dates back to the Second World War. In the late 1930s, the British military established teams of scientists to study *military operations*—radar deployment, convoy routing, anti-submarine tactics—using quantitative analysis. The term *Operational Research* (still the preferred name in British English) stuck.

After the war, the same ideas migrated to industry. George Dantzig developed the **Simplex method** for linear programming in 1947, initially for US Air Force planning. The RAND Corporation, founded in 1948, became a major hub. By the 1950s, OR was applied to oil refining, airline scheduling, and telecommunications.

Today, OR is everywhere—from supply chain optimization at Amazon to radiation therapy planning in oncology—and the algorithmic toolkit has grown enormously. But the core cycle shown in fig. 1.1 remains unchanged.

1.2 Problems and Instances

Before we can talk about solving anything, we need to be precise about what we mean by a "problem."

Definition 1.2.1 (Problem)**.** A **problem** π is an abstract question described in terms of *generic parameters*. It specifies:

- the structure of the input (what data is given), and
- the structure of the desired output (what kind of answer we are looking for).

Think of a problem as a *template*: it tells you the shape of the question, but not the specific numbers.

Definition 1.2.2 (Instance)**.** An **instance** i of a problem π is a *concrete realization* obtained by assigning specific values to all generic parameters.

Problem = the question template. Instance = a specific question.

An analogy: the problem is like a fill-in-the-blank form; an instance is what you get after filling in every blank.

Example 1.2.3 (TSP — problem vs. instance)**.** Consider the **Travelling Salesman Problem (TSP)**.

Problem (TSP). Given a set of cities and pairwise distances, find the shortest route that visits every city exactly once and returns to the starting city.

In graph-theory language: given a weighted complete graph K_n, find a minimum-weight Hamiltonian circuit.

An instance. Four cities $\{A, B, C, D\}$ with the following distance matrix:

	A	B	C	D
A	0	10	15	20
B	10	0	35	25
C	15	35	0	30
D	20	25	30	0

One feasible tour is $A \to B \to D \to C \to A$ with cost $10+25+30+15 = 80$. Is that the best we can do? We will answer this question—and learn to do so systematically—as the course progresses.

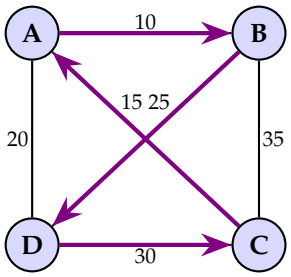


Figure 1.2: A 4-city TSP instance. The violet arrows show the tour $A \to B \to D \to C \to A$ of cost 80.

Notice the distinction: the *problem* says "find the shortest Hamiltonian circuit in a weighted complete graph"; the *instance* says "... where the graph is K_4 with these specific edge weights."

1.3 Optimization Problems

Most of OR is about *optimization*: among all feasible options, find the best one. Let's make this precise.

Definition 1.3.1 (Optimization problem)**.** An **optimization problem** π is described by a triple

$$\left(I_\pi,\ s(\cdot),\ f_\pi\right)$$

where:

- I_π is the **set of all instances** of π.
- For each instance $i \in I_\pi$, $s(i)$ is the **set of feasible solutions**.
- $f_\pi : \{(x, i) : i \in I_\pi,\ x \in s(i)\} \to \mathbb{R}$ is the **objective function**, assigning a real-valued "score" to each feasible solution of each instance.

In plain language: an optimization problem tells us (a) what the inputs can look like, (b) for any given input, what solutions are allowed, and (c) how to measure the quality of each solution.

I_π = all possible inputs.
$s(i)$ = what's allowed for input i.
f_π = how good a solution is.

Definition 1.3.2 (Optimal solution — minimization)**.** For a **minimization** problem, an optimal solution for instance i is a feasible solution $x^* \in s(i)$ such that

$$f_\pi(x^*, i) \ \leq\ f_\pi(x, i) \qquad \text{for all } x \in s(i).$$

Definition 1.3.3 (Optimal solution — maximization)**.** For a **maximization** problem, an optimal solution for instance i is a feasible solution $x^* \in s(i)$ such that

$$f_\pi(x^*, i) \ \geq\ f_\pi(x, i) \qquad \text{for all } x \in s(i).$$

Two important subtleties. First, there can be **multiple optimal solutions**—different feasible solutions that all achieve the same best objective value. The *optimal value* is unique (for a given instance), but the *optimal solution* need not be. Second, an optimal solution might not exist at all, for reasons we will explore in section 1.13.

Example 1.3.4 (TSP as an optimization problem)**.** For the TSP:
- I_π = all weighted complete graphs K_n (for any $n \geq 3$).
- For instance $i = (K_n, w)$, $s(i)$ = the set of all Hamiltonian circuits of K_n.
- $f_\pi(x, i)$ = total weight of the circuit x under weights w.
- Goal: *minimize* f_π.

For the 4-city instance of theorem 1.2.3, the feasible set $s(i)$ contains $(4-1)!/2 = 3$ distinct tours (up to direction), and we seek the one with the smallest total weight.

1.4 Decision Problems

Not every problem in OR involves optimization. Sometimes we just want to know: *can this be done at all?*

Definition 1.4.1 (Decision problem)**.** A **decision problem** is specified by:
- A set of instances I_π.
- A characterization of which instances are **"yes" instances** (i.e., instances for which the answer is *yes*).

The output is binary: either **yes** or **no**.

Notice what is missing compared to theorem 1.3.1: there is *no objective function*. We are not looking for the best solution; we are simply asking whether *any* solution satisfying certain criteria exists.

Example 1.4.2 (Hamiltonian Circuit — decision problem)**. Input:** An undirected graph $G = (V, E)$.
Question: Does G contain a Hamiltonian circuit (a circuit that visits every vertex exactly once)?

Here the answer is simply *yes* or *no*. We do not care about edge weights, and we do not ask for the shortest such circuit.

Example 1.4.3 (3-SAT — decision problem)**. Input:** A Boolean formula in conjunctive normal form (CNF) where each clause has exactly three literals.
Question: Is there a truth assignment to the variables that satisfies *all* clauses simultaneously?

For instance, the formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$ is a yes-instance: setting $x_1 = x_2 = x_3 = \text{TRUE}$ satisfies both clauses.

Decision problems and optimization problems are deeply related. Any optimization problem can be turned into a family of decision problems by adding a budget parameter. For instance: "Is there a TSP tour of cost at most k?" is the decision version of the TSP. We will use this connection in section 1.5.

1.5 Computational Difficulty and NP-Hardness

We will not build the full theory of complexity classes here; that belongs to a course on theoretical CS. We give the essentials needed for OR.

Let's return to the TSP and ask a naive question: why not just try all possible tours and pick the shortest?

For n cities, the number of distinct tours (ignoring direction) is $(n-1)!/2$. Let's tabulate a few values:

Cities (n)	Tours
5	12
10	181 440
15	$\approx 4.4 \times 10^{10}$
20	$\approx 6.1 \times 10^{16}$
30	$\approx 4.4 \times 10^{30}$

Even at a speed of 10^{15} tours per second—about the throughput of a modern supercomputer—30 cities would take roughly 4.4×10^{15} seconds, which is over **100 million years**. Exhaustive enumeration is hopeless for all but the smallest instances.

This is not a shortcoming of our hardware; it is a fundamental property of the *problem itself*.

Definition 1.5.1 (NP-complete — informal)**.** A decision problem is **NP-complete** if:

1. a proposed solution can be *verified* in polynomial time, but
2. no algorithm is known that *finds* a solution in polynomial time, and there is strong theoretical evidence that none exists.

The Hamiltonian Circuit problem (theorem 1.4.2) and 3-SAT (theorem 1.4.3) are both NP-complete. Given a candidate circuit, we can check in $O(n)$ time that it visits every vertex exactly once; but *finding* such a circuit—or proving none exists—seems to require super-polynomial time.

Definition 1.5.2 (NP-hard — informal)**.** A problem (decision or optimization) is **NP-hard** if it is "at least as hard" as every NP-complete problem. That is, a polynomial-time algorithm for it would imply polynomial-time algorithms for all NP-complete problems.

In plain language: NP-hard problems are the "hardest of the hard." An optimization problem can be NP-hard without being NP-complete, because NP-completeness is formally defined only for decision problems. The TSP (optimization version) is NP-hard.

Formal details — Reducing Hamiltonian Circuit to TSP

This is a problem reduction*: we show that solving TSP would let us solve Hamiltonian Circuit.*

To see that TSP is NP-hard, we show that the Hamiltonian Circuit (HC) problem **reduces** to TSP. Here is the reduction:

1. Start with an HC instance: a graph $G = (V, E)$.
2. Construct a TSP instance on the same vertex set V, defining edge weights as:

$$w(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E, \\ 2 & \text{if } (u, v) \notin E. \end{cases}$$

3. Solve the TSP instance. If the optimal tour has cost $|V|$, then every edge in the tour has weight 1, so every edge belongs to E—meaning G has a Hamiltonian circuit. If the optimal cost is $> |V|$, then G has no Hamiltonian circuit.

The construction takes $O(n^2)$ time. So if we could solve TSP in polynomial time, we could solve HC in polynomial time as well. Since HC is NP-complete, TSP must be at least as hard: it is NP-hard.

The key OR insight is this: even though TSP and many other problems are NP-hard in the worst case, **smart algorithms can still solve practical instances** of impressive size. The modern TSP record stands at an *exact* solution for 85 900 cities. The secret is not brute force, but clever mathematical structure: cutting planes, branch and bound, decomposition, and heuristics. These are exactly the tools we will develop in the rest of this course.

1.6 Applications of Operations Research

OR is not a purely theoretical subject; it lives in factories, hospitals, airports, and data centres. Here is a brief survey of application areas. Each hides an optimization or decision problem.

1. **Scheduling & production planning.** A factory has m machines and n jobs. Each job requires a sequence of operations on specific machines. Objective: minimize the total completion time (makespan).

Job-shop scheduling is NP-hard even for 3 machines.

2. **Timetabling.** A hospital must assign nurses to shifts over a month, respecting labour regulations (maximum hours, minimum rest between shifts) and skill requirements.

3. **Vehicle routing.** A logistics company must deliver goods from a depot to n customers, each with a time window, using a fleet of vehicles with limited capacity. This is a generalization of the TSP.
4. **Network design.** A telecommunications company must build a backbone network connecting n cities. The network must be resilient to single-link failures (2-edge-connected). Objective: minimize total cable cost.
5. **Assignment problems.** Assign n tasks to n workers, each with different efficiencies, so that total cost is minimized. A classic combinatorial optimization problem solvable in polynomial time.
6. **Portfolio optimization.** Select a mix of financial assets to maximize expected return subject to a risk budget. The Markowitz model (1952) was one of the earliest OR applications in finance.
7. **Radiation therapy planning.** Design beam angles and intensities for cancer treatment so that the tumour receives a lethal dose while healthy tissue is spared.
8. **Protein folding.** Given an amino acid sequence, predict the three-dimensional structure that minimizes free energy. Hugely combinatorial—a frontier problem at the intersection of OR and computational biology.

What unites these diverse problems is the modelling cycle of fig. 1.1: identify decisions, formalize constraints and objectives, build a model, design or apply an algorithm, interpret the result.

1.7 Mathematical Programming Foundations

The main language of OR is **mathematical programming**. Despite the name, this has nothing to do with writing software. The word "programming" was adopted in the 1940s (by Dantzig and others) to mean *planning* or *scheduling*.

"Programming" here means planning *or* modelling*, not writing code. The word predates the computer-science usage.*

The idea is simple: express every decision as a numerical variable, every constraint as an equation or inequality, and the goal as a function to optimize.

Definition 1.7.1 (Mathematical programming problem)**.** A **mathematical programming problem** (in standard form) is:

$$\begin{aligned} &\text{maximize} && f(x) \\ &\text{subject to} && x \in X, \end{aligned}$$

where:

- $x = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ is the vector of **decision variables**,
- $X \subseteq \mathbb{R}^n$ is the **feasible set**, defined by the constraints,
- $f : \mathbb{R}^n \to \mathbb{R}$ is the **objective function**.

Let's dwell on each piece, because these three ingredients—variables, feasible set, objective—are the building blocks of *every* model we will write.

Decision variables. These represent the choices under our control. In the furniture workshop example, x_1 might be the number of chairs and x_2 the number of tables. Variables are the *unknowns* we solve for.

Data (parameters). These are the known quantities: resource availability, costs, demands. We have no control over them. In the furniture workshop, the profit per chair, the profit per table, the amount of wood available—these are data.

Constraints. Constraints define the feasible set X. They encode physical limits, logical requirements, and resource capacities. For example: $3x_1 + 5x_2 \leq 150$ might say "wood usage cannot exceed 150 units."

Objective function. This is the single number that tells us how good a solution is. We either maximize it (profit, revenue, satisfaction) or minimize it (cost, time, waste).

1.8 Running Example 1: A Small Production LP

Let's establish a concrete example that will follow us through the rest of this book.

This problem will reappear in the graphical method (Ch. 2), Simplex (Ch. 4), and duality (Ch. 5).

Example 1.8.1 (Furniture workshop LP)**.** A workshop produces **chairs** (x_1) and **tables** (x_2). The data:

	Wood (units)	**Labour (hours)**	**Profit (€)**
Chair (x_1)	2	4	30
Table (x_2)	5	2	50
Available	40	32	—

Model:

$$\begin{aligned} \text{maximize} \quad & 30\,x_1 + 50\,x_2 \\ \text{subject to} \quad & 2\,x_1 + 5\,x_2 \leq 40 \quad \text{(wood)} \\ & 4\,x_1 + 2\,x_2 \leq 32 \quad \text{(labour)} \\ & x_1, x_2 \geq 0. \end{aligned}$$

Let's walk through the modelling steps explicitly:

1. **Variables.** x_1 = number of chairs; x_2 = number of tables. Both are continuous (for now—we relax this assumption later).
2. **Data.** Wood per chair = 2, wood per table = 5, total wood = 40; labour per chair = 4, labour per table = 2, total labour = 32; profit per chair = 30, profit per table = 50.
3. **Constraints.** Wood usage ≤ wood supply; labour usage ≤ labour supply; non-negativity.
4. **Objective.** Total profit = $30x_1 + 50x_2$; maximize.

The feasible region is a polygon in $\mathbb{R}^2$:

By inspection (or by the graphical method we will formalize in Chapter 2), the optimum sits at the vertex $(5, 6)$, where both constraints are tight. The optimal profit is $30 \cdot 5 + 50 \cdot 6 = 150 + 300 = 450$.

We will prove in Chapter 2 that the optimum of an LP always occurs at a vertex of the feasible region.

1.9 Running Example 2: A Small Directed Graph

This graph reappears in BFS/DFS (Ch. 8), shortest paths (Ch. 10), and network flow (Ch. 11).

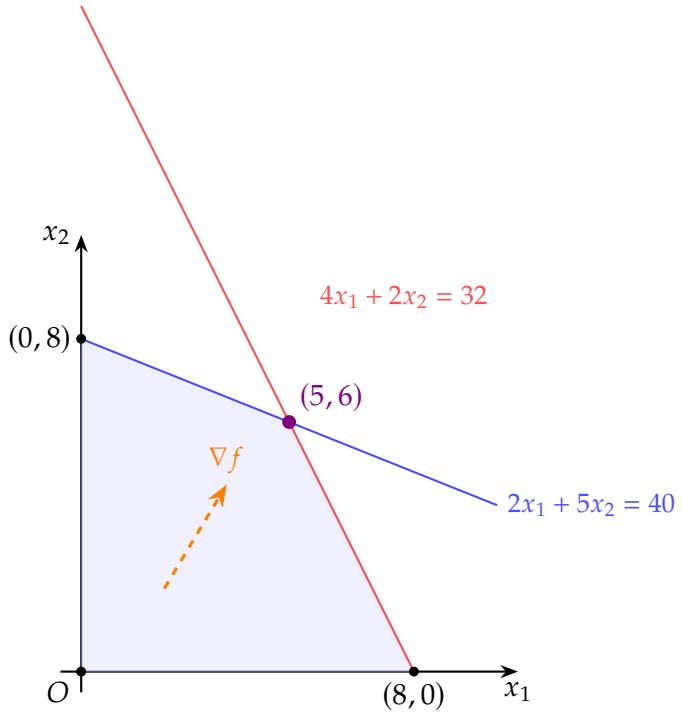


Figure 1.3: Feasible region for the furniture workshop LP (theorem 1.8.1). The violet vertex $(5, 6)$ is the optimal solution, with profit $30(5) + 50(6) = 450$. The orange arrow shows the gradient direction of the objective.

Many OR problems are naturally stated on *graphs*: networks of nodes connected by edges (or arcs). Let's establish a small directed graph that will accompany us throughout the course.

Example 1.9.1 (A 7-node directed graph)**.** Consider the directed graph $G = (V, A)$ shown in fig. 1.4, with $|V| = 7$ nodes and $|A| = 11$ arcs. Each arc (u, v) carries a weight $w(u, v)$ representing, depending on context, a distance, a cost, or a capacity.

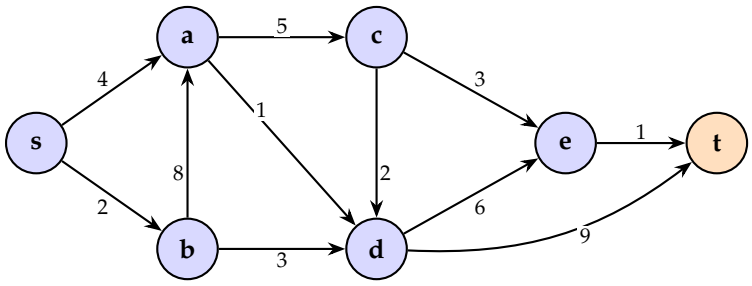


Figure 1.4: Running example: a 7-node directed graph. Node **s** is the source, node **t** (orange) the sink. Arc labels are weights.

In later chapters, this graph will serve as the test bed for:

- BFS and DFS traversals (Chapter 8),
- Dijkstra's shortest-path algorithm from s to t (Chapter 10),
- Maximum flow from s to t (Chapter 11), where the weights become arc capacities.

1.10 Running Example 3: A Small Knapsack Instance

The knapsack problem reappears in ILP modelling (Ch. 3), branch & bound (Ch. 6), and cutting planes (Ch. 6).

The **knapsack problem** is perhaps the most intuitive optimization problem: you have a backpack with limited capacity, and you want to pack the most valuable subset of items.

Example 1.10.1 (Knapsack instance)**.** You have a knapsack with weight capacity $W = 15$. Six items are available:

Item	Weight w_j	Value v_j
1	5	8
2	3	5
3	7	11
4	4	6
5	2	4
6	6	9

Model. Let $x_j \in \{0, 1\}$ indicate whether item j is packed.

$$\begin{aligned} \text{maximize} \quad & 8\,x_1 + 5\,x_2 + 11\,x_3 + 6\,x_4 + 4\,x_5 + 9\,x_6 \\ \text{subject to} \quad & 5\,x_1 + 3\,x_2 + 7\,x_3 + 4\,x_4 + 2\,x_5 + 6\,x_6 \le 15 \\ & x_j \in \{0, 1\}, \quad j = 1, \dots, 6. \end{aligned}$$

Notice three things about this model:

1. The variables are *binary*: each x_j is either 0 (leave the item) or 1 (take the item). This makes it an **integer linear program** (ILP), which is generally harder to solve than a continuous LP.
2. There is only one constraint (besides integrality): the total weight must not exceed capacity.
3. With 6 items there are $2^6 = 64$ possible subsets. We could check them all. But with 100 items there would be $2^{100} \approx 10^{30}$ subsets—exhaustive search is out of the question. We will need smarter methods.

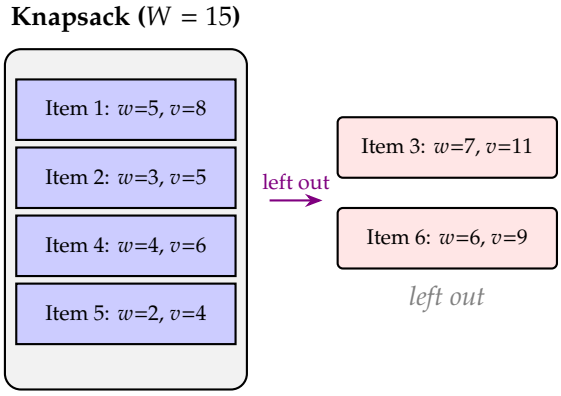


Figure 1.5: A feasible packing: items $\{1, 2, 4, 5\}$ have total weight $5+3+4+2 = 14 \le 15$ and total value $8 + 5 + 6 + 4 = 23$. Items 3 and 6 are left out. Is this packing optimal? We will find out in Chapter 6.

1.11 Domain Transformations: From Decisions to Vectors

One of the elegant ideas in mathematical programming is that very different-looking decisions can all be encoded as *vectors in* $\mathbb{R}^n$.

Continuous decisions. Some decisions are naturally real-valued: how many tons of steel to produce, how much to invest in each asset. These are directly represented by variables $x_j \in \mathbb{R}$.

Binary decisions. Many practical decisions are yes/no: build a factory or not, assign a worker to a task or not. We encode these as $x_j \in \{0, 1\}$.

The hypercube perspective. When all n variables are binary, the feasible solutions are a subset of the vertices of the n-dimensional hypercube $\{0, 1\}^n$. The hypercube has 2^n vertices—one for every possible combination of yes/no decisions.

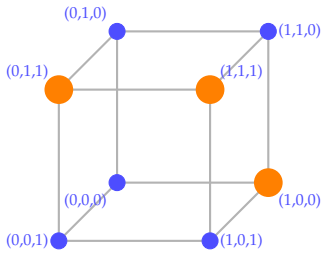


Figure 1.6: The 3-dimensional hypercube $\{0, 1\}^3$. Each vertex represents a binary decision vector. The orange vertices might be the feasible solutions of a particular constraint set.

The power of this perspective is that it unifies discrete and continuous optimization under the same mathematical framework. A binary knapsack problem lives on the vertices of a hypercube; a continuous production problem lives in a polyhedron. Both are subsets of $\mathbb{R}^n$, and many of the same tools apply.

1.12 Maximization vs. Minimization

Some textbooks use maximization as the default; others prefer minimization. In practice, the choice is irrelevant, thanks to a simple duality.

Theorem 1.12.1 (Max–Min duality). *For any function $f : X \to \mathbb{R}$,*

$$\max_{x \in X} f(x) \;=\; -\min_{x \in X} \big(-f(x)\big).$$

Moreover, x^* maximizes f over X if and only if x^* minimizes $-f$ over X.

In plain language: to turn a maximization problem into a minimization problem, just flip the sign of the objective. The optimal *solution* is the same; only the optimal *value* changes sign.

Throughout this course, we will use whichever formulation is more natural for the problem at hand: maximization for profit/revenue, minimization for cost/distance. You should always feel free to convert between the two.

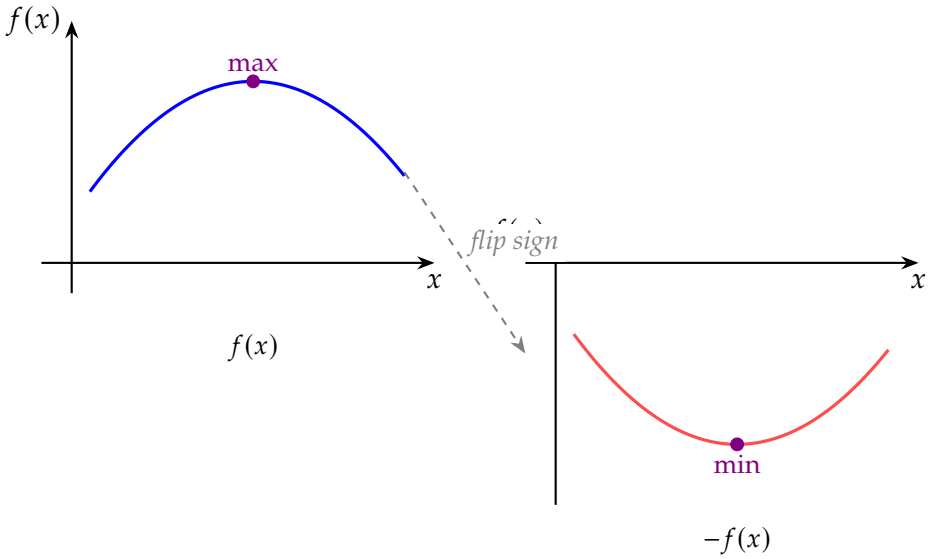


Figure 1.7: Geometric interpretation of max–min duality. Flipping the objective vertically turns a maximum into a minimum, but the optimizer x^* stays the same.

1.13 Feasibility, Optimality, and Unboundedness

Given a mathematical programming problem, there are exactly three possible outcomes. Understanding these is crucial before we attempt to solve anything.

Definition 1.13.1 (Feasibility)**.** A problem is **feasible** if its feasible set is non-empty: $X \neq \emptyset$. A problem is **infeasible** if $X = \emptyset$—that is, no point satisfies all constraints simultaneously.

Infeasibility means the constraints are contradictory. For example: $x_1 \geq 5$, $x_1 \leq 3$—no value of x_1 can satisfy both. In practice, infeasibility often signals a modelling error or over-constrained specifications.

Definition 1.13.2 (Bounded and unbounded)**.** A feasible maximization problem is **bounded** if there exists $M \in \mathbb{R}$ such that $f(x) \leq M$ for all $x \in X$. It is **unbounded** if for every M, there exists $x \in X$ with $f(x) > M$—the objective can grow without limit.

An analogous definition holds for minimization (replace "$\leq M$" with "$\geq M$" and "$>$" with "$<$").

Unbounded problems in practice usually indicate a missing constraint or a modelling mistake.

Definition 1.13.3 (Optimal solution)**.** A feasible problem has an **optimal solution** $x^* \in X$ if it is bounded and $f(x^*) \geq f(x)$ for all $x \in X$ (for maximization), or $f(x^*) \leq f(x)$ for all $x \in X$ (for minimization).

Theorem 1.13.4 (Existence of optimal solutions for LPs)**.** *For a linear program with a non-empty polyhedral feasible region, if the objective is bounded above (for maximization) or below (for minimization), then an optimal solution exists.*

The closedness of polyhedra matters here. For a completely general mathematical program, feasibility and boundedness alone are not enough: for example, maximize x subject to $x < 1$ is feasible and bounded, but it has no optimal solution because the value 1 is approached and never reached. LP feasible regions are closed polyhedra, so this escape cannot happen.

For LPs we can summarize the three cases in a table:

Case	Feasible?	Bounded?	Conventional opt. value
Optimal	Yes	Yes	$f(x^*)$
Unbounded	Yes	No	$+\infty$ (max) / $-\infty$ (min)
Infeasible	No	—	$-\infty$ (max) / $+\infty$ (min)

The conventional values for infeasible and unbounded problems are chosen so that useful inequalities hold. For instance, for maximization:

$$-\infty \;\le\; f(x^*) \;\le\; +\infty$$

and "infeasible ≤ optimal ≤ unbounded."

Example 1.13.5 (The three cases illustrated)**.** Consider three small problems in $\mathbb{R}^1$.

(a) Optimal. maximize x subject to $0 \le x \le 10$.
Feasible ($X = [0, 10]$), bounded. Optimal solution: $x^* = 10$, value = 10.

(b) Unbounded. maximize x subject to $x \ge 0$.
Feasible ($X = [0, +\infty)$), but unbounded: x can grow indefinitely. Conventional value: $+\infty$.

(c) Infeasible. maximize x subject to $x \ge 5$, $x \le 3$.
$X = \emptyset$. No feasible point exists. Conventional value: $-\infty$.

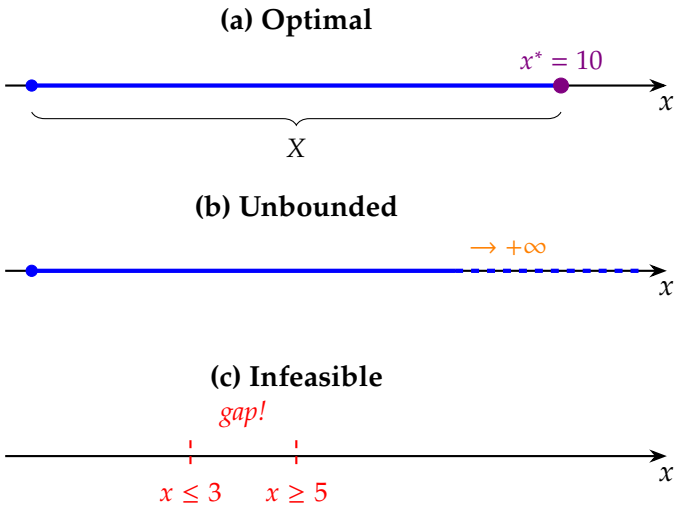


Figure 1.8: The three possible outcomes of an optimization problem: (a) an optimal solution exists, (b) unbounded—the objective grows indefinitely, (c) infeasible—no feasible point exists.

1.14 The Model Formulation Process

Building a good model is an art as much as a science. Here is a systematic process.

Step 1. Understand the scenario. Read the problem description carefully. What is the real-world context? Who makes the decisions? What is the goal?

Step 2. Identify the decisions (variables). What can be controlled? Each independent decision becomes a variable. Choose meaningful notation: x_j for production quantities, y_{ij} for assignments, etc.

Step 3. Identify the data (parameters). What is given and fixed? Costs, capacities, demands, time horizons—these are constants in the model.

Step 4. Write the constraints. Translate physical, logical, and regulatory restrictions into mathematical inequalities or equalities. Don't forget non-negativity and integrality where appropriate.

Step 5. Write the objective function. Express the goal as a function of the decision variables. Decide whether to maximize or minimize.

Step 6. Validate. Check the model on a small instance. Does it produce sensible answers? Are the units consistent? Are there redundant or missing constraints?

Remark 1.14.1. A model is always a *simplification* of reality. We ignore some factors, approximate others, and assume precise knowledge of the data. When data is uncertain (as it often is), we enter the realm of **stochastic programming** and **robust optimization**—topics beyond the scope of this course, but important to keep in mind. The models we build here are most powerful when the parameters (costs, capacities, demands) are known with reasonable precision.

Let's walk through the formulation process one more time, on a fresh example.

Example 1.14.2 (Diet problem — formulation walkthrough)**.** A student wants to meet daily nutritional requirements at minimum cost. They can buy three foods: rice (x_1 kg), beans (x_2 kg), and chicken (x_3 kg).

	Protein (g/kg)	Calories (kcal/kg)	Cost (€/kg)
Rice	25	1300	1.50
Beans	80	1200	2.00
Chicken	250	1650	6.00
Daily requirement:		$\geq$ 50 g protein	$\geq$ 2000 kcal

Step 1. The student decides how much of each food to buy.
Step 2. Variables: $x_1, x_2, x_3 \geq 0$ (kg of each food).
Step 3. Data: nutritional content per kg, costs per kg, daily requirements.

Step 4. Constraints:

$$\begin{aligned} 25\,x_1 + 80\,x_2 + 250\,x_3 &\geq 50 && \text{(protein)} \\ 1300\,x_1 + 1200\,x_2 + 1650\,x_3 &\geq 2000 && \text{(calories)} \\ x_1, x_2, x_3 &\geq 0. \end{aligned}$$

Step 5. Objective: minimize $1.50\,x_1 + 2.00\,x_2 + 6.00\,x_3$.

Step 6. Quick check: buying only chicken, $x_3 = 2000/1650 \approx 1.21\,\text{kg}$ would cost about 7.27 € and provide $\approx 303\,\text{g}$ protein (more than enough). So the problem is certainly feasible. Can we do cheaper with a mix? That's what solving the model will tell us.

1.15 A Taxonomy of Mathematical Programs

We will study LP in Chapters 2–5 and ILP in Chapters 3 and 6.

Mathematical programs come in many flavours, depending on the nature of the objective function, the constraints, and the variables. Here is a brief taxonomy of the types we will encounter in this course:

Linear Program (LP)
Both the objective and all constraints are *linear* functions of continuous variables:

$$\text{maximize } c^\top x \quad \text{subject to } Ax \leq b,\ x \geq 0.$$

This is the workhorse of OR. Chapters 2–5 are devoted to LP.

Integer Linear Program (ILP)
Like an LP, but some or all variables are restricted to integer values: $x_j \in \mathbb{Z}$ (or $x_j \in \{0, 1\}$ for binary).

$$\text{maximize } c^\top x \quad \text{subject to } Ax \leq b,\ x \geq 0,\ x \in \mathbb{Z}^n.$$

ILPs are generally NP-hard. We study them in Chapters 3 and 6.

Mixed-Integer Program (MIP)
Some variables are continuous, others integer. This is the most general linear model:

$$\text{maximize } c^\top x + d^\top y \quad \text{subject to } Ax + By \leq b,\ x \geq 0,\ y \in \mathbb{Z}^p.$$

Nonlinear Program (NLP)
The objective or constraints (or both) are nonlinear. For example, minimize $x_1^2 + x_2^2$ subject to $x_1 + x_2 \geq 1$. We will not study NLPs in depth, but they arise naturally in portfolio optimization and engineering design.

1.16 Summary and Outlook

Let's recap what we have established in this opening chapter.

- **Operations Research** applies mathematical models and algorithmic thinking to real-world decision problems, following the cycle: real problem → formalization → model → algorithm → solution (fig. 1.1).

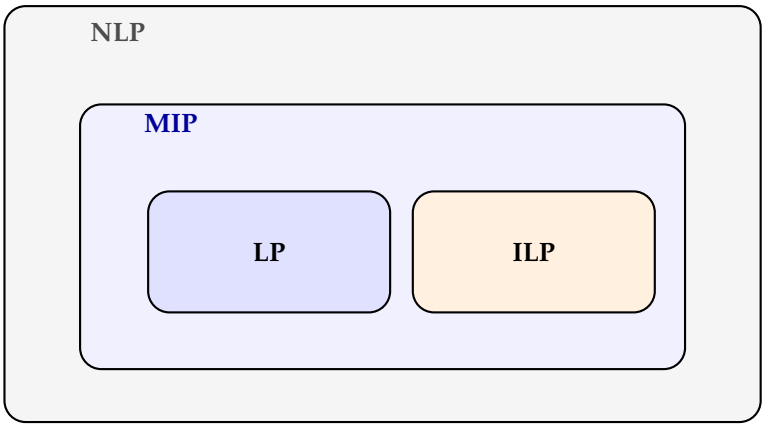


Figure 1.9: Taxonomy of mathematical programs. LP and ILP are special cases of MIP, which is itself a special case of NLP.

- A **problem** is an abstract template; an **instance** is a specific realization with concrete data (section 1.2).
- An **optimization problem** is a triple $(I_\pi, s(\cdot), f_\pi)$, and we seek feasible solutions that maximize (or minimize) the objective function (section 1.3).
- **Decision problems** have binary output (yes/no) and no objective function (section 1.4).
- Many important problems are **NP-hard**: no known polynomial-time algorithm exists. But smart OR algorithms can still solve large instances in practice (section 1.5).
- **Mathematical programming** provides a unified framework: variables, constraints, objective function. Maximization and minimization are interchangeable via sign-flip (sections 1.7 and 1.12).
- Every problem falls into one of three cases: **optimal**, **unbounded**, or **infeasible** (section 1.13).
- We established three **running examples** that will reappear throughout the course:
  1. the furniture workshop LP (theorem 1.8.1),
  2. the 7-node directed graph (theorem 1.9.1),
  3. the 6-item knapsack (theorem 1.10.1).

In the next chapter, we dive into **Linear Programming** in earnest: the graphical method for 2-variable problems, standard form, and the geometry of polyhedra. The furniture workshop LP will be our first test subject.

■ **Summary & Key Takeaways**

- **Operations Research:** Systematic application of mathematical modeling and quantitative methods to support decision-making processes.
- **Modelling Components:** Decision variables (what to choose), Objective function (what to optimize), and Constraints (limits on feasibility).

- **Problem Classes:** Linear Programming (LP - continuous variables), Integer Linear Programming (ILP - discrete variables), Mixed-Integer Linear Programming (MILP), and Nonlinear Programming (NLP).
- **NP-Hardness:** Combinatorial problems like the Knapsack Problem (KP) and Travelling Salesman Problem (TSP) are NP-hard; they lack polynomial-time exact algorithms, requiring heuristics or structured search.

Exercises

Exercise 1. Define *Operations Research* in your own words. What distinguishes it from pure mathematics and from computer science?

Exercise 2. True or false: "Operations Research always requires a computer to find a solution." Justify your answer.

Exercise 3. List three real-world domains in which OR techniques are routinely applied, and for each domain name one concrete decision that OR helps to optimize.

Exercise 4. What are the three main components of a *mathematical programming model*? Briefly explain the role of each component.

Exercise 5. Define the term *instance* of a problem and explain how it differs from the problem itself. Give one example from operations research.

Exercise 6. The *Shortest Path Problem* asks, given a weighted directed graph $G = (V, A)$ and two nodes $s, t \in V$, for a minimum-cost directed path from s to t.

(a) Write down a specific instance of this problem (you may use a small graph with $|V| \leq 5$).

(b) Identify which part of your description belongs to the *problem* and which part belongs to the *instance*.

Exercise 7. Explain why a single algorithm is said to *solve a problem* rather than solve an instance, even though the algorithm operates on a specific input each time it is run.

Exercise 8. True or false: "Two instances of the same problem can have different feasible sets." Justify your answer with an example.

Exercise 9. Formally define an *optimization problem*. Your definition must include: the feasible set X, the objective function f, and the notion of an optimal solution.

Exercise 10. Consider the following optimization problem:

$$\max\ 3x_1 + 2x_2 \quad \text{s.t.} \quad x_1 + x_2 \leq 4,\ x_1, x_2 \geq 0.$$

(a) Write the feasible set X in set-builder notation.

(b) Find an optimal solution by inspection and state the optimal value.

Exercise 11. The *furniture workshop* produces tables (x_1) and chairs (x_2). Each table requires 3 hours of carpentry and 2 hours of finishing; each chair requires 2 hours of carpentry and 1 hour of finishing. Available capacity is 120 carpentry hours and 70 finishing hours per week. Profit is \$90 per table and \$60 per chair.

(a) Write the LP formulation (decision variables, objective, constraints).

(b) Identify the type of the model (LP, ILP, MILP, or NLP).

Exercise 12. What is the difference between a *feasible solution* and an *optimal solution*? Is every optimal solution feasible? Is every feasible solution optimal?

Exercise 13. True or false: "An optimization problem always has at least one feasible solution." Justify with a brief example or counterexample.

Exercise 14. Define a *decision problem*. How does it differ structurally from an optimization problem?

Exercise 15. Given the optimization problem

$$\min \; c^\top x \quad \text{s.t.} \quad Ax \le b, \; x \ge 0,$$

formulate the corresponding *decision version* parameterized by a threshold $k \in \mathbb{R}$.

Exercise 16. True or false: "If the decision version of an optimization problem can be solved in polynomial time, then the optimization version can also be solved in polynomial time." Justify your answer.

Exercise 17. Explain informally what it means for a problem to be *NP-hard*. Why does NP-hardness not mean a problem is unsolvable in practice?

Exercise 18. The *Knapsack Decision Problem* asks: given items with weights w_i and values v_i, a capacity W, and a threshold K, is there a subset of items with total weight $\le W$ and total value $\ge K$?

(a) Is this problem in NP? Justify your answer by describing a polynomial-time verification procedure.

(b) State whether this problem is known to be NP-hard.

Exercise 19. True or false: "NP-hard problems cannot be solved to optimality for any instance." Justify your answer.

Exercise 20. Explain the relationship between *polynomial-time solvability* and *practical tractability*. Give an example of a problem that is polynomial-time solvable but may still be slow on very large instances.

Exercise 21. Suppose you are told that the optimization version of a problem is NP-hard. Describe two strategies that OR practitioners commonly use to still obtain good solutions in reasonable time.

Exercise 22. Define each of the following model classes and state the key structural property that distinguishes it from the others:

(a) Linear Program (LP)

(b) Integer Linear Program (ILP)

(c) Mixed-Integer Linear Program (MILP)

(d) Nonlinear Program (NLP)

Exercise 23. Classify each of the following models as LP, ILP, MILP, or NLP. Justify each classification.

(a) $\min \; x_1^2 + x_2$ s.t. $x_1 + x_2 \ge 1$, $x_1, x_2 \ge 0$.

(b) max $5x_1 + 3x_2$ s.t. $2x_1 + x_2 \leq 10$, $x_1, x_2 \geq 0$, $x_1 \in \mathbb{Z}$.

(c) min $4x_1 + 7x_2$ s.t. $x_1 + 2x_2 = 6$, $x_1, x_2 \geq 0$.

(d) max $x_1 x_2$ s.t. $x_1 + x_2 \leq 4$, $x_1, x_2 \geq 0$.

Exercise 24. Why is the integrality constraint ($x \in \mathbb{Z}^n$) alone enough to make an otherwise linear program potentially NP-hard to solve?

Exercise 25. Show that the maximization problem

$$\max \ f(x) \quad \text{s.t.} \quad x \in X$$

is equivalent to the minimization problem

$$\min \ -f(x) \quad \text{s.t.} \quad x \in X.$$

Specifically, prove that x^* is optimal for the max problem if and only if it is optimal for the min problem.

Exercise 26. Convert the following maximization LP to an equivalent minimization LP without changing the feasible set:

$$\max \ 5x_1 - 3x_2 + x_3 \quad \text{s.t.} \quad x_1 + x_2 \leq 4, \ 2x_2 + x_3 \leq 6, \ x_1, x_2, x_3 \geq 0.$$

What is the relationship between the optimal values of the two formulations?

Exercise 27. An optimization problem is in one of three states: *infeasible*, *unbounded*, or *has an optimal solution*. Define each state precisely and give a simple 1-variable example of each.

Exercise 28. Consider the LP:

$$\max \ x_1 + x_2 \quad \text{s.t.} \quad x_1 - x_2 \leq 1, \ x_1, x_2 \geq 0.$$

Determine whether this problem is infeasible, unbounded, or has an optimal solution. Justify your answer.

Exercise 29. Consider the LP:

$$\min \ x \quad \text{s.t.} \quad x \geq 3, \ x \leq 1.$$

Determine whether this problem is infeasible, unbounded, or has an optimal solution. Justify your answer.

Exercise 30. True or false: "A feasible problem always has an optimal solution." Justify with a brief argument or counterexample.

Exercise 31. Prove or disprove: "If a minimization problem is unbounded, then its feasible set must be infinite."

Exercise 32. A directed graph (*digraph*) $G = (V, A)$ has node set $V = \{1, 2, 3, 4, 5, 6, 7\}$ and arc set

$$A = \{(1,2), (1,3), (2,4), (3,4), (3,5), (4,6), (5,6), (6,7), (5,7)\}.$$

(a) Draw the graph (a sketch suffices).

(b) List all directed paths from node 1 to node 7.

(c) How many arcs does the graph have? How many nodes?

Exercise 33. For the directed graph in the previous exercise, assign the following arc costs: $c_{12} = 4$, $c_{13} = 2$, $c_{24} = 5$, $c_{34} = 1$, $c_{35} = 3$, $c_{46} = 2$, $c_{56} = 4$, $c_{67} = 3$, $c_{57} = 6$.

(a) Compute the total cost of the path $1 \to 3 \to 4 \to 6 \to 7$.

(b) Compute the total cost of the path $1 \to 3 \to 5 \to 7$.

(c) Which of these two paths is cheaper?

Exercise 34. Formulate the *minimum-cost flow* problem on a directed graph $G = (V, A)$ as a linear program. Define all notation you introduce (variables, parameters, constraints).

Exercise 35. True or false: "Every directed graph on n nodes has at most $n(n-1)$ arcs." Justify your answer.

Exercise 36. Six items are available for the knapsack with capacity $W = 10$. Their weights and values are:

Item	1	2	3	4	5	6
Weight	2	3	4	5	1	3
Value	6	5	8	9	3	7

(a) Write the 0-1 Knapsack ILP formulation for this instance.

(b) Evaluate the objective value for the selection $\{1, 3, 6\}$. Is it feasible?

(c) Evaluate the objective value for the selection $\{2, 4, 5\}$. Is it feasible?

Exercise 37. For the 6-item knapsack instance in the previous exercise, compute the *fractional relaxation* bound by sorting items by value-to-weight ratio and applying the greedy fractional algorithm.

Exercise 38. Formulate the general 0-1 Knapsack Problem as an integer linear program. Use n items with weights w_i, values v_i, capacity W, and binary variables $x_i \in \{0, 1\}$.

Exercise 39. Why is the 0-1 Knapsack Problem NP-hard while its *fractional relaxation* (where $0 \le x_i \le 1$) can be solved in $O(n \log n)$ time?

Exercise 40. A production plant manufactures two products P_1 and P_2. Each unit of P_1 requires 4 kg of raw material and 1 machine-hour; each unit of P_2 requires 2 kg and 3 machine-hours. Available resources are 120 kg of raw material and 90 machine-hours. Unit profits are \$10 for P_1 and \$15 for P_2.

(a) Write the LP formulation.

(b) State whether the model is an LP, ILP, MILP, or NLP.

(c) Write the decision version of this problem with threshold K.

Exercise 41. For the LP formulation you wrote in the previous exercise:

(a) Is the point $(x_1, x_2) = (20, 10)$ feasible? Check all constraints.

(b) Is the point $(x_1, x_2) = (30, 0)$ feasible?

(c) Which of the two feasible points (if any) has the larger objective value?

Exercise 42. A decision variable x is required to satisfy $l \le x \le u$. Show how to substitute $x' = x - l$ to obtain an equivalent formulation in which the new variable satisfies $0 \le x' \le u - l$.

Exercise 43. A variable x is *free* (unrestricted in sign). Show how to replace x with two non-negative variables x^+ and x^- such that $x = x^+ - x^-$, $x^+, x^- \ge 0$. Why might this substitution be useful in LP solvers?

Exercise 44. Consider a binary variable $y \in \{0, 1\}$ that models whether a factory is open ($y = 1$) or closed ($y = 0$). A fixed cost of F is incurred if the factory is open, and a variable cost of c per unit of production $p \ge 0$ is incurred. Write a mixed-integer formulation of the total cost, including a constraint that production is only possible when the factory is open (assume an upper bound M on production).

Exercise 45. A small airline must assign pilots to routes. There are m routes and n pilots; each pilot j is qualified for a subset $Q_j \subseteq \{1, \dots, m\}$ of routes. Each route must be covered by exactly one pilot. Formulate a *set-cover* integer program for this assignment.

Exercise 46. A company ships goods from two warehouses (W_1, W_2) to three customers (C_1, C_2, C_3). Supply at W_1 is 80 units and at W_2 is 60 units. Demand at C_1, C_2, C_3 is 50, 40, and 50 units respectively. Shipping cost per unit from W_i to C_j is given by the matrix:

$$\begin{pmatrix} 2 & 3 & 1 \\ 5 & 4 & 8 \end{pmatrix}.$$

Formulate this as a linear program (the *transportation problem*).

Exercise 47. Formulate the *maximum weight independent set* problem on an undirected graph $G = (V, E)$ with node weights $w_v \ge 0$ as an integer linear program. (A set $S \subseteq V$ is *independent* if no two nodes in S are adjacent.)

Exercise 48. A project consists of n tasks. Task j has a duration d_j and a set P_j of *predecessor tasks* that must all finish before task j can start. Define start time variables $s_j \ge 0$ and formulate the problem of minimizing the project makespan as a linear program.

Exercise 49. Explain the difference between *continuous* and *discrete* optimization. Give one example of each type from the running examples in this chapter.

Exercise 50. An objective function $f(x) = c^\top x$ is *linear*. Explain why replacing it with $f(x) = \|x\|_2^2$ changes the problem class from LP to NLP. Does the feasible set change?

Exercise 51. Consider the following claim: "Solving the decision version of an NP-hard problem is easier than solving its optimization version." Argue for or against this claim, providing a formal justification.

Exercise 52. Let z^*_{LP} denote the optimal value of the LP relaxation of an ILP, and z^*_{ILP} the optimal value of the ILP itself (both are maximization problems). Prove or disprove: $z^*_{\text{LP}} \ge z^*_{\text{ILP}}$.

Exercise 53. Suppose an optimization problem over a finite feasible set has 2^{50} feasible solutions. Explain why exhaustive enumeration is infeasible in practice, and discuss what OR provides as an alternative.

Exercise 54. Define what it means for a constraint to be *binding* (or *active*) at a

feasible solution x^0. For the LP

$$\max\ 2x_1 + x_2 \quad \text{s.t.} \quad x_1 + x_2 \leq 5,\ x_1 \leq 3,\ x_1, x_2 \geq 0,$$

identify which constraints are binding at the point $(3, 2)$.

Exercise 55. Explain how OR can be used to support *multi-objective* decisions where a decision-maker wants to simultaneously minimize cost and maximize service level. Why is there generally no single optimal solution in such settings?

CHAPTER 2

Linear Programming

In chapter 1 we met the furniture workshop (theorem 1.8.1) and saw, almost by inspection, that its optimal solution lies at a corner of the feasible region. This chapter asks *why* that happens and turns the observation into a general theory.

This chapter formalises the LP framework introduced informally in Chapter 1 and develops the geometry that underpins every algorithm we will study.

We will proceed in four stages. First, we give a formal definition of linear programming and its various forms. Second, we develop the **graphical method**—a visual technique that works for two-variable problems and builds powerful geometric intuition. Third, we study the **geometry** of LP: polyhedra, polytopes, faces, and vertices. Fourth, we show that every LP can be rewritten in a **standard form** that the Simplex method (Chapter 4) requires. Along the way we will meet new modelling examples that illustrate the breadth and power of the LP framework.

2.1 What is Linear Programming?

The furniture workshop LP of theorem 1.8.1 had two variables, a linear objective, and linear constraints. Let's generalise.

"Linear" refers to the objective and constraints; "programming" means planning, *not coding.*

Definition 2.1.1 (Linear Programming problem)**.** A **Linear Programming (LP)** problem is an optimization problem of the form

$$\begin{aligned} &\text{maximize or minimize} \\ &\quad c_1x_1 + c_2x_2 + \cdots + c_nx_n \\ &\text{subject to} \\ &\quad a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \begin{cases} \leq \\ = \\ \geq \end{cases} b_i, \quad i = 1, \ldots, m, \\ &\quad x_j \geq 0 \text{ (for some or all } j), \quad x_j \in \mathbb{R}. \end{aligned}$$

In compact **matrix notation**, writing $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$:

$$\text{maximize} \quad c^\top x \qquad \text{subject to} \quad Ax \leq b, \quad x \geq 0.$$

Three features make an LP *linear*:

1. The **objective function** $c^\top x = \sum_j c_j x_j$ is linear in x.
2. Every **constraint** is a linear inequality or equality.

3. Every **variable** x_j is continuous—it takes values in $\mathbb{R}$ (or $\mathbb{R}_{\geq 0}$).

In particular, there are no products of variables ($x_1 x_2$), no powers (x_j^2), and no integrality requirements ($x_j \in \mathbb{Z}$). If the objective or a constraint involves such terms, the problem is a *nonlinear program*; if variables must be integers, it is an *integer linear program* (ILP, Chapter 3).

No $x_1 x_2$ terms, no x_j^2, no $\lfloor x_j \rfloor$. If any of these appear, it is no longer an LP.

In the most general form we may have a mix of $\leq$, $=$, and $\geq$ constraints, and some variables may be unrestricted in sign. It is convenient to partition the constraints and variables into groups.

Definition 2.1.2 (Generic LP form)**.** Let M_1 index the "$\leq$" rows, M_2 the "$=$" rows, and M_3 the "$\geq$" rows. Let N_1 index the variables restricted to be non-negative, and N_2 those that are unrestricted. A generic LP is:

$$
\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{n} c_j \, x_j \\
\text{subject to} \quad & \sum_{j=1}^{n} a_{ij} \, x_j \leq b_i \quad i \in M_1 \\
& \sum_{j=1}^{n} a_{ij} \, x_j = b_i \quad i \in M_2 \\
& \sum_{j=1}^{n} a_{ij} \, x_j \geq b_i \quad i \in M_3 \\
& x_j \geq 0 \qquad\qquad j \in N_1 \\
& x_j \in \mathbb{R} \text{ (free)} \qquad j \in N_2
\end{aligned}
$$

with $M_1 \cup M_2 \cup M_3 = \{1, \ldots, m\}$ and $N_1 \cup N_2 = \{1, \ldots, n\}$.

The furniture workshop LP (theorem 1.8.1) has $M_1 = \{1, 2\}$, $M_2 = M_3 = \emptyset$, $N_1 = \{1, 2\}$, $N_2 = \emptyset$: all constraints are $\leq$ and all variables are non-negative. This is the simplest and most common pattern.

Intermezzo — Why "Linear" Programming?

Linear programming was born in 1947, when George Dantzig—then a mathematical advisor to the US Air Force—developed both the LP formulation and the Simplex algorithm to solve logistics planning problems. The word "programming" meant *planning* (as in "program of activities"), not computer programming. Leonid Kantorovich had independently formulated linear optimization problems in the Soviet Union as early as 1939, but his work was suppressed and not widely known in the West until decades later. Kantorovich shared the 1975 Nobel Prize in Economics for his contributions.

2.2 The Graphical Method

When an LP has only two decision variables, we can solve it by hand on a two-dimensional plot. The method is simple:

The graphical method only works for 2 (at most 3) variables, but it builds the geometric intuition that guides the Simplex method.

Step 1. Plot each constraint as a **boundary line** and shade the feasible side.

Step 2. The **feasible region** is the intersection of all shaded half-planes (plus non-negativity).

Step 3. Draw the **objective function isolines** (level sets $c^\top x = k$ for various k).

Step 4. Slide the isoline in the direction of the **gradient** $\nabla(c^\top x) = c$ until it is about to leave the feasible region.

Step 5. The last point(s) of contact with the feasible region give the **optimal solution**.

Let's walk through this carefully on our recurring example.

Example 2.2.1 (Graphical method — Step 1: plotting constraints)**.** Recall the furniture workshop LP (theorem 1.8.1):

$$\begin{aligned} \text{maximize} \quad & 30\,x_1 + 50\,x_2 \\ \text{subject to} \quad & 2\,x_1 + 5\,x_2 \le 40 \quad \text{(wood)} \\ & 4\,x_1 + 2\,x_2 \le 32 \quad \text{(labour)} \\ & x_1, x_2 \ge 0. \end{aligned}$$

Each inequality constraint defines a **half-plane**. Its boundary is the line obtained by replacing $\le$ with $=$:

- **Wood:** $2x_1 + 5x_2 = 40$. Intercepts: $(20, 0)$ and $(0, 8)$.
- **Labour:** $4x_1 + 2x_2 = 32$. Intercepts: $(8, 0)$ and $(0, 16)$.

The feasible side is always the side that contains the origin (since $0 \le 40$ and $0 \le 32$). Figure 2.1 shows both lines with the feasible side shaded.

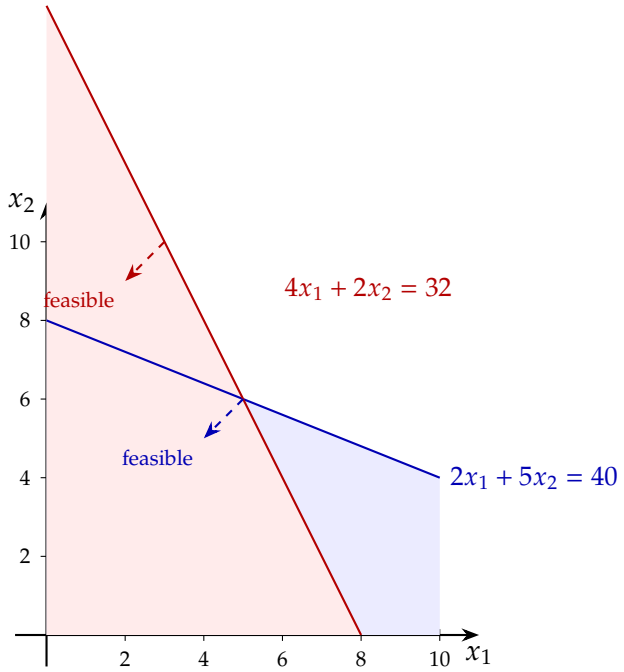


Figure 2.1: Step 1 of the graphical method: each constraint is a line that divides the plane into two half-planes. The arrows point toward the feasible side.

Example 2.2.2 (Graphical method — Step 2: feasible region)**.** The feasible region is the intersection of all four half-planes (two resource constraints plus the two non-negativity constraints $x_1 \geq 0$ and $x_2 \geq 0$). Geometrically, this is a **convex polygon** in the first quadrant.

Every point inside the polygon satisfies all constraints simultaneously.

To find its vertices, we solve every pair of boundary-line equations and keep only those intersections that satisfy all constraints:

- $x_1 = 0,\ x_2 = 0$: the origin $(0, 0)$.
- $4x_1 + 2x_2 = 32,\ x_2 = 0$: gives $(8, 0)$. Check wood: $2(8) + 5(0) = 16 \leq 40$. ✓
- $2x_1 + 5x_2 = 40,\ 4x_1 + 2x_2 = 32$: solving the system yields $(x_1, x_2) = (5, 6)$. Both constraints are tight. ✓
- $2x_1 + 5x_2 = 40,\ x_1 = 0$: gives $(0, 8)$. Check labour: $4(0) + 2(8) = 16 \leq 32$. ✓

The feasible region is the quadrilateral with vertices $(0, 0)$, $(8, 0)$, $(5, 6)$, $(0, 8)$, shown in fig. 2.2.

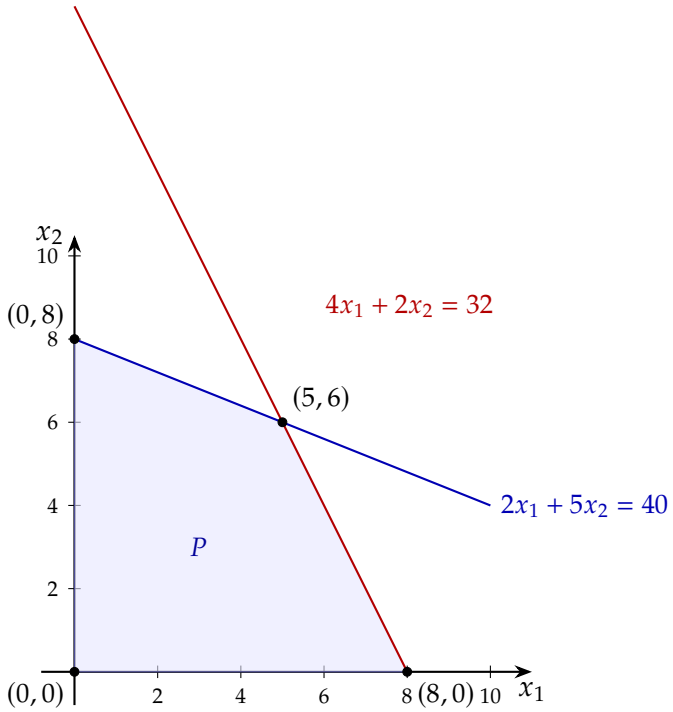


Figure 2.2: Step 2: the feasible region P (shaded) is the convex polygon with four vertices. Every point in P satisfies all constraints.

Example 2.2.3 (Graphical method — Step 3: objective isolines)**.** The objective function is $z = 30x_1 + 50x_2$. An **isoline** (or *level set*) is the set of all points (x_1, x_2) where the objective takes a fixed value k:

$$30x_1 + 50x_2 = k.$$

This is a family of parallel lines (one for each value of k) with slope $-30/50 = -3/5$. As k increases, the line shifts in the direction of the

gradient

$$\nabla z = \begin{pmatrix} 30 \\ 50 \end{pmatrix}.$$

Figure 2.3 shows several isolines for increasing values of k.

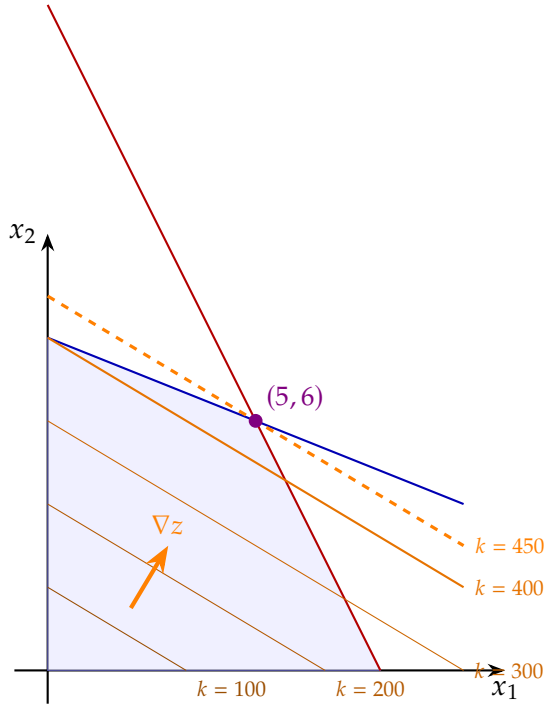


Figure 2.3: Step 3: objective isolines $30x_1 + 50x_2 = k$ for $k = 100, 200, 300, 400, 450$. The gradient $\nabla z = (30, 50)$ points in the direction of increasing profit. The last isoline touching the feasible region (dashed, $k = 450$) passes through the vertex $(5, 6)$.

Example 2.2.4 (Graphical method — Steps 4–5: the optimum)**.** We "slide" the isoline in the gradient direction. As long as it intersects the feasible region, we can increase k. The largest k for which the isoline still touches the feasible region gives the **optimal value**.

For the furniture workshop, the last point of contact is the vertex $(5, 6)$, which yields

$$z^* = 30(5) + 50(6) = 150 + 300 = 450.$$

This confirms the answer we previewed in chapter 1.

What could happen in general? Three scenarios:

1. **Unique vertex optimum.** The isoline exits at a single vertex. This is the generic case.
2. **Edge optimum (multiple optima).** The objective gradient is perpendicular to a binding constraint, so the isoline lies along an edge of the feasible region. Every point on that edge is optimal; the optimal value is unique, but the optimal solution is not.
3. **Unbounded.** The isoline can slide indefinitely without leaving the feasible region. The LP has no finite optimum.

If the isoline is parallel *to a facet, the optimum is an entire edge—infinitely many optimal solutions sharing the same optimal value.*

Example 2.2.5 (Multiple optimal solutions)**.** Consider the LP

$$\text{maximize} \quad 2x_1+5x_2 \qquad \text{subject to} \quad 2x_1+5x_2 \le 40,\ 4x_1+2x_2 \le 32,\ x_1, x_2 \ge 0.$$

The objective $2x_1 + 5x_2$ has the same coefficients as the wood constraint. Its isolines are parallel to the wood boundary line. Every point on the edge from $(5, 6)$ to $(0, 8)$ achieves $2x_1 + 5x_2 = 40$. The optimal value is 40, but there are infinitely many optimal solutions.

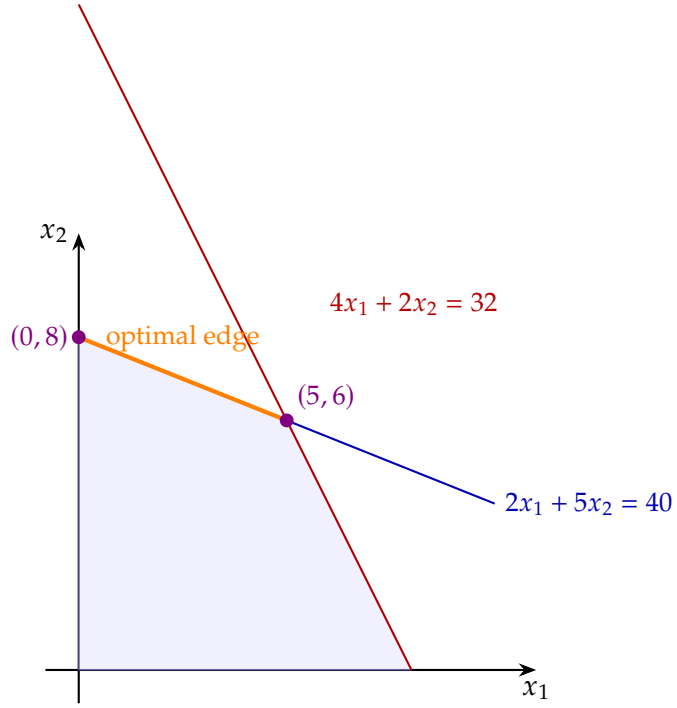


Figure 2.4: When the objective is parallel to a binding constraint, the optimum is an entire edge (thick orange segment).

Exercises 1–6 provide hands-on graphical method practice across all constraint types; Exercises 31–33 ask you to construct your own examples of the infeasible, unbounded, and multiple-optima cases just discussed.

2.3 Geometry of LP: Polyhedra and Polytopes

The graphical method gave us a polygon. In higher dimensions, the analogous object is a **polyhedron**. Understanding polyhedra is essential—the Simplex method (Chapter 4) will walk along the edges of a polyhedron from vertex to vertex.

2.3.1 Half-spaces and hyperplanes

Definition 2.3.1 (Hyperplane)**.** A **hyperplane** in $\mathbb{R}^n$ is a set of the form

$$H = \left\{ x \in \mathbb{R}^n : a^\top x = b \right\},$$

where $a \in \mathbb{R}^n \setminus \{0\}$ and $b \in \mathbb{R}$.

A hyperplane divides $\mathbb{R}^n$ into two **closed half-spaces**:

$$H^+ = \{ x : a^\top x \le b \}, \qquad H^- = \{ x : a^\top x \ge b \}.$$

In the furniture workshop, the wood constraint $2x_1 + 5x_2 \le 40$ defines a half-space whose boundary hyperplane (a line in $\mathbb{R}^2$) is $2x_1 + 5x_2 = 40$. The feasible side is the half-space $H^+ = \{x : 2x_1 + 5x_2 \le 40\}$.

In $\mathbb{R}^2$, a hyperplane is a line; in $\mathbb{R}^3$, it is a plane.

2.3.2 Polyhedra

Definition 2.3.2 (Polyhedron)**.** A **polyhedron** is the intersection of finitely many closed half-spaces:

$$P = \{ x \in \mathbb{R}^n : Ax \le b \}$$

for some matrix $A \in \mathbb{R}^{m \times n}$ and vector $b \in \mathbb{R}^m$.

This is the **external representation** (or *H-representation*) of a polyhedron: we describe it by the half-spaces that bound it.

In plain language, a polyhedron is the region "inside" a collection of flat walls. In two dimensions, polyhedra are polygons (possibly unbounded). In three dimensions, they are the familiar solid shapes with flat faces—like a cube, a tetrahedron, or a prism.

Example 2.3.3 (The furniture feasible region as a polyhedron)**.** The feasible region of the furniture workshop LP is

$$P = \left\{ \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in \mathbb{R}^2 \; : \; \begin{pmatrix} 2 & 5 \\ 4 & 2 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \le \begin{pmatrix} 40 \\ 32 \\ 0 \\ 0 \end{pmatrix} \right\},$$

where the last two rows encode $x_1 \ge 0$ and $x_2 \ge 0$ (written as $-x_1 \le 0$ and $-x_2 \le 0$). This is a bounded polygon with four vertices and four edges.

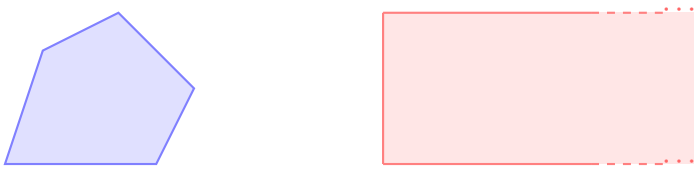
Exercises 12 and 13 ask you to enumerate vertices and edges for other polyhedra by hand; Exercise 17 invites you to construct a polyhedron that has no vertices at all.

2.3.3 Polytopes

Definition 2.3.4 (Polytope)**.** A **polytope** is a **bounded** polyhedron: a polyhedron P such that there exists $R > 0$ with $P \subseteq \{x : \|x\| \le R\}$.

Equivalently, a polytope is a polyhedron that can be enclosed in a finite ball. The furniture feasible region is a polytope (it fits inside a circle of radius 20). An example of an *unbounded* polyhedron is $\{(x_1, x_2) : x_1 \ge 0,\ x_2 \ge 0\}$—the entire first quadrant, which extends to infinity.

Every polytope is a polyhedron, but not every polyhedron is a polytope. A polyhedron can extend to infinity.



Polytope (bounded) **Polyhedron (may be unbounded)**

Figure 2.5: Left: a polytope—a polyhedron that is bounded. Right: a polyhedron that happens to be unbounded; polyhedra in general allow this. Solid lines are finite constraint boundaries; dashed lines and $\cdots$ indicate the region extends to infinity.

2.3.4 Dimension of a polyhedron

Definition 2.3.5 (Dimension of a polyhedron)**.** The **dimension** of a polyhedron $P \subseteq \mathbb{R}^n$ is

$$\dim(P) = n - \operatorname{rank}(A_{\text{eq}}),$$

where A_{eq} is the matrix of *implicit equalities*—those constraints $a_i^\top x \le b_i$ that are satisfied with equality by **every** point in P.

Intuitively, each linearly independent equality "uses up" one degree of freedom. A polyhedron $P \subseteq \mathbb{R}^n$ with no implicit equalities has dimension n; we say it is **full-dimensional**.

"Implicit equality" = a constraint that holds as = everywhere in P, even though it was written as $\le$.

Example 2.3.6 (Dimension of the furniture polytope)**.** The furniture polytope lives in $\mathbb{R}^2$. None of its four constraints holds as equality for *every* point in P (for instance, the origin satisfies all constraints as strict inequalities). So $\operatorname{rank}(A_{\text{eq}}) = 0$ and $\dim(P) = 2 - 0 = 2$. The polytope is full-dimensional.

Example 2.3.7 (A lower-dimensional polyhedron)**.** Consider $P = \{(x_1, x_2) \in \mathbb{R}^2 : x_1 + x_2 = 1,\ x_1 \ge 0,\ x_2 \ge 0\}$. The constraint $x_1 + x_2 = 1$ is an implicit equality. So $\dim(P) = 2 - 1 = 1$: the polyhedron is the line segment from $(1, 0)$ to $(0, 1)$, which is one-dimensional even though it lives in $\mathbb{R}^2$.

Exercise 19 extends this to the standard simplex in $\mathbb{R}^3$; Exercises 35 and 52 explore dimension, vertices, and facets for the unit cube and a related polytope in $\mathbb{R}^4$.

2.4 Faces, Facets, Edges, and Vertices

We now look at the *boundary structure* of a polyhedron. When you stare at a cube, you see 8 corners, 12 edges, and 6 flat sides. These are all examples of **faces**.

The face hierarchy—from vertices up to facets—organises the combinatorial structure of a polyhedron.

2.4.1 Supporting hyperplanes and faces

Definition 2.4.1 (Supporting hyperplane)**.** A hyperplane $H = \{x : a^\top x = b\}$ is a **supporting hyperplane** of the polyhedron P if:

1. $P \subseteq H^+$ or $P \subseteq H^-$ (i.e., P lies entirely on one side of H), and
2. $H \cap P \neq \emptyset$ (i.e., H "touches" P).

In two dimensions, a supporting hyperplane is a line that touches the polygon without crossing into its interior. Think of balancing a ruler against one edge of the polygon.

Definition 2.4.2 (Face)**.** A **face** of a polyhedron P is the intersection of P with a supporting hyperplane:

$$F = P \cap H,$$

where H is a supporting hyperplane of P.

By convention, P itself and $\emptyset$ are also considered faces (the *improper* faces).

2.4.2 Types of faces

Definition 2.4.3 (Facet, edge, vertex)**.** Let P be a polyhedron of dimension $d = \dim(P)$.

- A **facet** is a face of dimension $d - 1$.
- An **edge** is a face of dimension 1.
- A **vertex** is a face of dimension 0 (a single point).

Facets are the "walls," edges are the "ridges," and vertices are the "corners."

For the furniture polytope ($d = 2$):

- **Facets** = edges of the polygon (dimension 1): the four sides connecting consecutive vertices.
- **Edges** = also the sides (since $d - 1 = 1$; in 2D, facets and edges coincide).
- **Vertices** = the four corners: $(0, 0)$, $(8, 0)$, $(5, 6)$, $(0, 8)$.

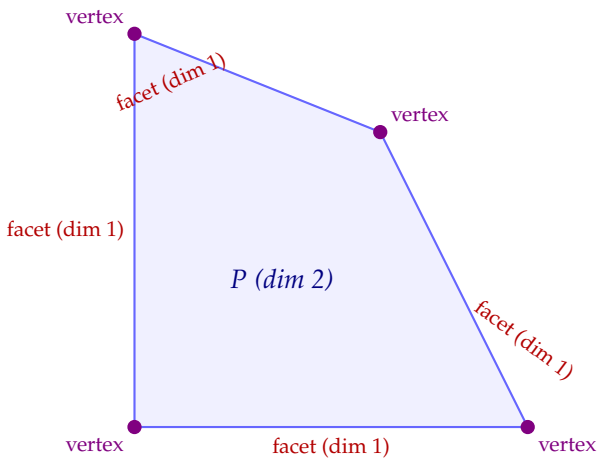


Figure 2.6: The face structure of the furniture polytope: four vertices (dimension 0), four facets/edges (dimension 1), and the full polytope (dimension 2).

Exercise 16 carries out the same face analysis for the unit square $[0, 1]^2$. Exercises 43 and 46 ask you to prove or disprove further structural properties

of polyhedra and their faces.

2.4.3 Vertices as extreme points

Vertices have a beautiful characterisation that does not refer to supporting hyperplanes at all.

Definition 2.4.4 (Extreme point)**.** A point $v \in P$ is an **extreme point** (or vertex) of the polyhedron P if it **cannot** be written as a strict convex combination of two other points in P:

$$v = \lambda u + (1-\lambda) w \quad \text{with } u, w \in P,\ 0 < \lambda < 1 \quad \Longrightarrow \quad u = w = v.$$

In plain language: a vertex is a point that does not lie strictly between any two other feasible points. You cannot "average away" a vertex—it sticks out.

Example 2.4.5 (Extreme points of the furniture polytope)**.** The point $(5, 6)$ is a vertex because there is no way to express it as $\lambda u + (1-\lambda)w$ with u, w in P and $0 < \lambda < 1$. On the other hand, the point $(3, 3)$—which lies inside P—*can* be written as a convex combination; for instance, take

$$(3,3) = \tfrac{1}{2}(2,2) + \tfrac{1}{2}(4,4).$$

Both points are feasible: for $(4, 4)$ we have $2(4) + 5(4) = 28 \leq 40$ and $4(4) + 2(4) = 24 \leq 32$, and the check for $(2, 2)$ is even easier. Thus $(3, 3)$ lies between two feasible points and is not an extreme point.

Exercises 15 and 27 ask you to verify the extreme-point condition directly for specific points in explicit polyhedra.

2.4.4 The fundamental theorem of Linear Programming

Here is the key result that makes LP tractable.

Theorem 2.4.6 (Fundamental theorem of LP)**.** *If a linear program* maximize $c^\top x$ *subject to* $x \in P$ *(where* P *is a non-empty polytope) has a finite optimum, then* ***at least one optimal solution is a vertex of*** P*.*

This is the theorem that makes the entire LP machinery work. Instead of searching over the *infinitely many* points inside a polytope, we only need to check the *finitely many* vertices. The Simplex method (Chapter 4) is a smart way to walk from vertex to vertex, improving the objective at each step, until it reaches an optimum.

This theorem reduces an infinite search (over all points in P) to a finite search (over the vertices of P). The Simplex method exploits this.

Formal details — Proof of the Fundamental Theorem

Proof. Let x^* be an optimal solution. If x^* is a vertex, we are done. Otherwise, x^* is not an extreme point, so there exist $u, w \in P$, $u \neq w$, and $0 < \lambda < 1$ such that $x^* = \lambda u + (1-\lambda)w$.

Since $c^\top x^*$ is optimal, we have

$$c^\top x^* \geq c^\top u \quad \text{and} \quad c^\top x^* \geq c^\top w.$$

But also

$$c^\top x^* = \lambda\, c^\top u + (1-\lambda)\, c^\top w.$$

If $c^\top u < c^\top x^*$ or $c^\top w < c^\top x^*$, then the weighted average would be strictly less than $c^\top x^*$, a contradiction. So $c^\top u = c^\top w = c^\top x^*$: both u and w are also optimal.

Now repeat the argument with u in place of x^*. Since P is a polytope (bounded and with finitely many vertices), this process must terminate at a vertex. □

Exercises 28–30 probe your understanding with batteries of true/false statements about LP optimality, feasibility, and polyhedra. Exercise 34 asks you to show that a finite optimum can exist even when the feasible region is unbounded.

2.5 Convexity and the Minkowski–Weyl Theorem

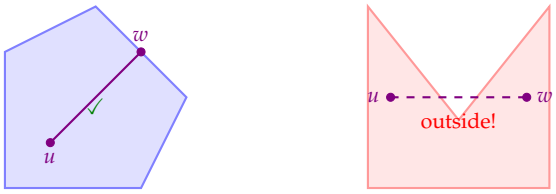
Convexity is the geometric property that makes LP "easy" compared to general optimisation.

The geometry of LP rests on one fundamental property: **convexity**. Every polyhedron is convex, and convexity is what guarantees that local optima are global optima.

2.5.1 Convex sets and convex combinations

Definition 2.5.1 (Convex set)**.** A set $S \subseteq \mathbb{R}^n$ is **convex** if, for every pair of points $u, w \in S$, the entire line segment joining them lies in S:

$$\lambda\, u + (1-\lambda)\, w \in S \qquad \text{for all } \lambda \in [0,1].$$



Convex **Not convex**

Figure 2.7: Left: a convex set—every line segment between two points lies inside. Right: a non-convex set—the dashed segment exits the set.

Exercise 22 asks you to exhibit a non-convex set in $\mathbb{R}^2$ and pinpoint which condition it violates; Exercises 14, 18, 20 and 21 develop fluency with convexity proofs.

Definition 2.5.2 (Convex combination)**.** A **convex combination** of points $v_1, \dots, v_k \in \mathbb{R}^n$ is

$$x = \sum_{i=1}^{k} \lambda_i\, v_i \qquad \text{where} \quad \lambda_i \geq 0 \;\; \forall\, i, \quad \sum_{i=1}^{k} \lambda_i = 1.$$

A convex combination is a "weighted average" where the weights are non-negative and sum to one. For two points, it gives every point on the line segment; for three points, every point in the triangle; and so on.

Definition 2.5.3 (Convex hull)**.** The **convex hull** of a set $S \subseteq \mathbb{R}^n$, denoted $\operatorname{conv}(S)$, is the set of all convex combinations of points in S. Equivalently, it is the smallest convex set containing S.

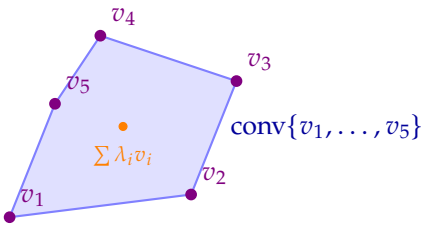


Figure 2.8: The convex hull of five points is the smallest convex polygon containing all of them. Every interior point is a convex combination of the vertices.

Exercise 23 asks you to describe the convex hull of three specific points using half-space inequalities.

2.5.2 The Minkowski–Weyl theorem

There are two natural ways to describe a polytope:

1. **External (H-representation):** $P = \{x : Ax \le b\}$—the intersection of half-spaces.
2. **Internal (V-representation):** $P = \operatorname{conv}\{v_1, \dots, v_k\}$—the convex hull of vertices.

The remarkable fact is that these descriptions are *equivalent*: every polytope can be described either way.

Theorem 2.5.4 (Minkowski–Weyl)**.** *A set $P \subseteq \mathbb{R}^n$ is a bounded polyhedron (polytope) if and only if it is the convex hull of finitely many points:*

$$P = \{x : Ax \le b\} \quad \Longleftrightarrow \quad P = \operatorname{conv}\{v_1, \dots, v_k\}$$

for some $v_1, \dots, v_k \in \mathbb{R}^n$.

The H- and V-representations are dual descriptions *of the same object. Converting between them is computationally expensive in general.*

Remark 2.5.5 (Full Minkowski–Weyl for unbounded polyhedra). The key idea is that any unbounded polyhedron can be split into a bounded "core" (the convex hull of its vertices) and a set of directions one can travel to infinity (its extreme rays) — so every point is just a vertex combination plus a non-negative combination of those directions. The theorem above covers the bounded case. For a general (possibly unbounded) polyhedron, every point $x \in P$ can be decomposed as

$$x = v + r, \qquad v \in \operatorname{conv}(V), \quad r \in \operatorname{cone}(R),$$

where $V = \{v_1, \dots, v_k\}$ is the finite set of **vertices** of P and $R = \{r_1, \dots, r_\ell\}$ is a finite set of **extreme rays** of P. The **conic hull** is

$$\operatorname{cone}(R) = \left\{ \textstyle\sum_{i=1}^{\ell} \lambda_i r_i : \lambda_i \ge 0 \right\}.$$

In short, $P = \text{conv}(V) + \text{cone}(R)$. The polytope case (theorem 2.5.4) is the special case $R = \emptyset$, i.e. $\text{cone}(R) = \{0\}$.

Think of it this way: you can describe a diamond either by listing its facets (the flat cuts made by a jeweller) or by listing its corners (vertices). Both descriptions uniquely determine the same shape.

Example 2.5.6 (Two representations of the furniture polytope). **H-representation** (external):

$$P = \left\{x \in \mathbb{R}^2 : 2x_1 + 5x_2 \le 40,\ 4x_1 + 2x_2 \le 32,\ x_1 \ge 0,\ x_2 \ge 0\right\}.$$

V-representation (internal):

$$P = \text{conv}\big\{(0,0),\ (8,0),\ (5,6),\ (0,8)\big\}.$$

Both describe exactly the same quadrilateral in $\mathbb{R}^2$.

Exercise 24 asks you to prove that every polytope equals the convex hull of its vertices (the V-direction of Minkowski–Weyl); Exercises 25 and 26 work through specific convex combinations and the decomposition of an unbounded polyhedron.

Intermezzo — Minkowski and Weyl

Hermann Minkowski (1864–1909) was a German mathematician who made fundamental contributions to geometry and number theory. He introduced the concept of a convex body and proved that every convex polytope is the convex hull of its vertices. Hermann Weyl (1885–1955), one of the great mathematicians of the twentieth century, later proved the converse: every convex hull of finitely many points is a polyhedron (i.e., can be described by linear inequalities). Together, these results form the Minkowski–Weyl theorem, a cornerstone of polyhedral combinatorics.

2.6 Canonical and Standard Forms

Different forms of LP are mathematically equivalent but serve different algorithmic purposes.

So far, we have seen LPs written with a mix of $\le$, $\ge$, and $=$ constraints, and variables that may or may not have sign restrictions. Algorithms, however, prefer a *fixed* format. In this section we define two privileged forms and show that every LP can be converted to either one.

2.6.1 Canonical form

Definition 2.6.1 (Canonical form). An LP is in **canonical form** if:

Maximization:

$$\text{maximize} \quad c^\top x \qquad \text{subject to} \quad Ax \le b, \quad x \ge 0.$$

Minimization:

$$\text{minimize} \quad c^\top x \qquad \text{subject to} \quad Ax \ge b, \quad x \ge 0.$$

The pattern is: for maximization, all functional constraints are $\leq$; for minimization, all functional constraints are $\geq$. In both cases, all variables are non-negative.

The furniture workshop LP (theorem 1.8.1) is already in canonical (maximization) form.

2.6.2 Standard form

Definition 2.6.2 (Standard form)**.** An LP is in **standard form** if:

$$\text{maximize} \quad c^\top x \qquad \text{subject to} \quad Ax = b, \quad x \geq 0.$$

All constraints are **equalities** and all variables are **non-negative**.

The Simplex method requires standard form. Every LP can be brought to standard form by adding slack/surplus variables.

Standard form is what the Simplex method needs. At first glance, it seems restrictive—how can we enforce inequalities if we only have equalities? The answer is **slack** and **surplus** variables, which we introduce next.

2.7 Conversion to Standard Form

Any LP—regardless of its original form—can be rewritten in standard form. The transformations are purely mechanical.

2.7.1 Handling $\leq$ constraints (slack variables)

Definition 2.7.1 (Slack variable)**.** Given a $\leq$ constraint $a^\top x \leq b$, introduce a new variable $s \geq 0$ and rewrite the constraint as the equality

$$a^\top x + s = b, \qquad s \geq 0.$$

The variable s is called a **slack variable**. Its value represents the "slack" (unused resource).

For instance, the wood constraint $2x_1 + 5x_2 \leq 40$ becomes $2x_1 + 5x_2 + s_1 = 40$ with $s_1 \geq 0$. If $(x_1, x_2) = (5, 6)$, then $s_1 = 40 - 2(5) - 5(6) = 0$: all wood is used. If $(x_1, x_2) = (3, 3)$, then $s_1 = 40 - 6 - 15 = 19$: we have 19 units of wood left over.

2.7.2 Handling $\geq$ constraints (surplus variables)

Definition 2.7.2 (Surplus variable)**.** Given a $\geq$ constraint $a^\top x \geq b$, introduce a new variable $s \geq 0$ and rewrite the constraint as

$$a^\top x - s = b, \qquad s \geq 0.$$

The variable s is called a **surplus variable**. Its value represents the amount by which the left-hand side exceeds b.

2.7.3 Handling unrestricted variables

If a variable x_j is unrestricted in sign ($x_j \in \mathbb{R}$), we **split** it into two non-negative variables:

$$x_j = x_j^+ - x_j^-, \qquad x_j^+, x_j^- \geq 0.$$

We replace every occurrence of x_j in the model with $(x_j^+ - x_j^-)$.

2.7.4 Converting minimization to maximization

If the original LP minimizes $c^\top x$, we equivalently **maximize** $(-c)^\top x$. The optimal solution is the same; only the sign of the optimal value changes.

2.7.5 Summary of transformations

Original form	Standard-form equivalent	New variable
$a^\top x \leq b$	$a^\top x + s = b,\ s \geq 0$	slack s
$a^\top x \geq b$	$a^\top x - s = b,\ s \geq 0$	surplus s
x_j free	replace x_j with $x_j^+ - x_j^-;\ x_j^+, x_j^- \geq 0$	split pair
minimize $c^\top x$	maximize $(-c)^\top x$	—

Exercises 7–11 practise all four conversion moves on a range of LP instances. Exercise 47 explores the max/min equivalence further; Exercise 48 shows how to linearise an absolute-value constraint; Exercise 53 introduces an auxiliary variable to handle a min-max objective.

2.7.6 Step-by-step example: furniture workshop to standard form

Example 2.7.3 (Furniture workshop LP in standard form)**.** Starting from the canonical form (theorem 1.8.1):

$$\begin{aligned} \text{maximize} \quad & 30\,x_1 + 50\,x_2 \\ \text{subject to} \quad & 2\,x_1 + 5\,x_2 \leq 40 \\ & 4\,x_1 + 2\,x_2 \leq 32 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Step 1. The first constraint is $\leq$. Introduce slack $s_1 \geq 0$:

$$2x_1 + 5x_2 + s_1 = 40.$$

Step 2. The second constraint is $\leq$. Introduce slack $s_2 \geq 0$:

$$4x_1 + 2x_2 + s_2 = 32.$$

Step 3. Both x_1, x_2 are already ≥ 0. No splitting needed.
Step 4. Already a maximization. No sign flip needed.

Result (standard form):

$$\begin{aligned} \text{maximize} \quad & 30\,x_1 + 50\,x_2 + 0\,s_1 + 0\,s_2 \\ \text{subject to} \quad & 2\,x_1 + 5\,x_2 + s_1 = 40 \\ & 4\,x_1 + 2\,x_2 + s_2 = 32 \\ & x_1, x_2, s_1, s_2 \geq 0. \end{aligned}$$

In matrix notation, with $\hat{x} = (x_1, x_2, s_1, s_2)^\top$:

$$\hat{c} = \begin{pmatrix} 30 \\ 50 \\ 0 \\ 0 \end{pmatrix}, \quad \hat{A} = \begin{pmatrix} 2 & 5 & 1 & 0 \\ 4 & 2 & 0 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 40 \\ 32 \end{pmatrix}.$$

We went from 2 variables and 2 inequalities to 4 variables and 2 equalities. In general, each $\leq$ constraint adds one slack variable, so an LP with n original variables and m inequality constraints becomes a standard-form LP with $n + m$ variables and m equalities.

Example 2.7.4 (A mixed LP converted to standard form)**.** Consider the LP:

$$\begin{aligned} \text{minimize} \quad & 3x_1 - 2x_2 + x_3 \\ \text{subject to} \quad & x_1 + x_2 + x_3 \leq 10 \\ & 2x_1 - x_2 \geq 4 \\ & x_1, x_2 \geq 0, \quad x_3 \text{ free}. \end{aligned}$$

Step 1. Convert minimize to maximize: maximize $-3x_1 + 2x_2 - x_3$.
Step 2. First constraint ($\leq$): add slack $s_1 \geq 0$: $x_1 + x_2 + x_3 + s_1 = 10$.
Step 3. Second constraint ($\geq$): subtract surplus $s_2 \geq 0$: $2x_1 - x_2 - s_2 = 4$.
Step 4. Variable x_3 is free: replace with $x_3 = x_3^+ - x_3^-$, both ≥ 0.

Result:

$$\begin{aligned} \text{maximize} \quad & -3x_1 + 2x_2 - x_3^+ + x_3^- \\ \text{subject to} \quad & x_1 + x_2 + x_3^+ - x_3^- + s_1 = 10 \\ & 2x_1 - x_2 - s_2 = 4 \\ & x_1, x_2, x_3^+, x_3^-, s_1, s_2 \geq 0. \end{aligned}$$

We now have 6 variables and 2 equalities, all in standard form.

2.7.7 Basic feasible solutions

Standard form gives us a clean algebraic characterisation of the vertices of the feasible polyhedron—the points where the Simplex method will search for the optimum.

Every vertex of the standard-form polyhedron corresponds to at least one BFS. The Simplex method (Chapter 4) walks from BFS to BFS.

Definition 2.7.5 (Basic feasible solution)**.** Intuitively, a basic feasible solution is what you get when you "use up" exactly m degrees of freedom: you pin $n - m$ variables to zero, solve the resulting square system, and check that the remaining variables are non-negative. Formally, consider an LP in standard form with $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $\operatorname{rank}(A) = m$.

- A **basis** $\mathcal{B} \subseteq \{1, \ldots, n\}$ is a set of m column indices such that the $m \times m$ submatrix $A_{\mathcal{B}}$ is non-singular. The corresponding variables $x_{\mathcal{B}}$ are called **basic**; the remaining $n - m$ variables $x_{\mathcal{N}}$ (indexed by $\mathcal{N} = \{1, \ldots, n\} \setminus \mathcal{B}$) are **non-basic**.
- The **basic solution** associated with $\mathcal{B}$ is obtained by setting $x_{\mathcal{N}} = 0$ and solving $A_{\mathcal{B}}\, x_{\mathcal{B}} = b$, i.e. $x_{\mathcal{B}} = A_{\mathcal{B}}^{-1}\, b$.
- A **basic feasible solution (BFS)** is a basic solution with $x_{\mathcal{B}} \geq 0$.

Key fact: every vertex of the feasible polyhedron $\{x : Ax = b,\ x \geq 0\}$ corresponds to at least one BFS, and every BFS corresponds to a vertex. Hence the Simplex method can restrict its search to the (finitely many) basic feasible solutions.

Exercise 36 asks you to identify the rank condition that guarantees a point is a BFS; Exercises 49–51 work through the slack/surplus interpretation of active constraints in concrete examples.

Chapter 4 develops this algebraic framework in full, showing precisely how the Simplex method moves from one BFS to an adjacent one by swapping one index into $\mathcal{B}$ and one out (a *pivot*).

2.8 LP Modelling: More Examples

The hardest part of LP is not solving it—it is building *the model correctly.*

The furniture workshop is a tiny LP with two variables. Real-world LPs can have millions of variables and constraints. What makes LP powerful is its **modelling flexibility**: an enormous range of problems can be cast as LPs. In this section we present several classic examples.

2.8.1 The diet problem

Example 2.8.1 (Diet problem)**.** A student wants to meet daily nutritional requirements at minimum cost. The following foods are available:

Food	Cost (€/unit)	Protein (g)	Carbs (g)	Fat (g)
Rice	0.30	3	28	0.5
Chicken	1.50	25	0	3
Broccoli	0.80	3	5	0.4
Bread	0.20	4	15	1

Daily requirements: at least 50 g protein, at least 200 g carbs, at most 65 g fat.

Variables: x_1 = units of rice, x_2 = units of chicken, x_3 = units of broccoli, x_4 = units of bread.

Model:

$$\begin{aligned}
\text{minimize} \quad & 0.30\,x_1 + 1.50\,x_2 + 0.80\,x_3 + 0.20\,x_4 & \\
\text{subject to} \quad & 3\,x_1 + 25\,x_2 + 3\,x_3 + 4\,x_4 \geq 50 & \text{(protein)} \\
& 28\,x_1 + 0\,x_2 + 5\,x_3 + 15\,x_4 \geq 200 & \text{(carbs)} \\
& 0.5\,x_1 + 3\,x_2 + 0.4\,x_3 + 1\,x_4 \leq 65 & \text{(fat)} \\
& x_1, x_2, x_3, x_4 \geq 0. &
\end{aligned}$$

This is an LP in canonical minimization form (after treating the fat constraint): the protein and carb constraints are $\geq$, the fat constraint is $\leq$. It can be converted to standard form using the transformations of section 2.7.

Exercises 37 and 38 ask you to formulate and extend a similar diet problem from scratch.

The diet problem nicely illustrates the *dual* nature of LP constraints: some constraints impose lower bounds ("eat *at least* this much"), others impose upper bounds ("eat *at most* this much"). The LP framework handles both seamlessly.

The diet problem was one of the first LP applications, studied by Stigler (1945) and solved by Dantzig using the Simplex method in 1947.

2.8.2 The transportation problem

Example 2.8.2 (Transportation problem)**.** A company has two factories (F1, F2) and three warehouses (W1, W2, W3). Each factory has a supply, each warehouse has a demand, and there is a per-unit shipping cost for each factory-warehouse pair:

	W1	**W2**	**W3**	**Supply**
F1	4	8	1	30
F2	5	2	7	50
Demand	20	30	30	

Variables: x_{ij} = units shipped from factory i to warehouse j (for $i \in \{1,2\}$, $j \in \{1,2,3\}$).

Model:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{2}\sum_{j=1}^{3} c_{ij}\,x_{ij} = 4x_{11} + 8x_{12} + x_{13} + 5x_{21} + 2x_{22} + 7x_{23} & \\
\text{subject to} \quad & \sum_{j=1}^{3} x_{ij} \leq S_i, \quad i = 1,2 & \text{(supply)} \\
& \sum_{i=1}^{2} x_{ij} \geq D_j, \quad j = 1,2,3 & \text{(demand)} \\
& x_{ij} \geq 0 \quad \forall\, i, j. &
\end{aligned}$$

This LP has $2 \times 3 = 6$ variables and $2 + 3 = 5$ functional constraints. When supply equals demand ($\sum S_i = \sum D_j$), the problem is *balanced* and all

constraints can be written as equalities. Note that total supply here is $30 + 50 = 80$ and total demand is $20 + 30 + 30 = 80$, so this instance is balanced.

Exercises 39 and 40 ask you to formulate a transportation instance with different data and check whether it is balanced.

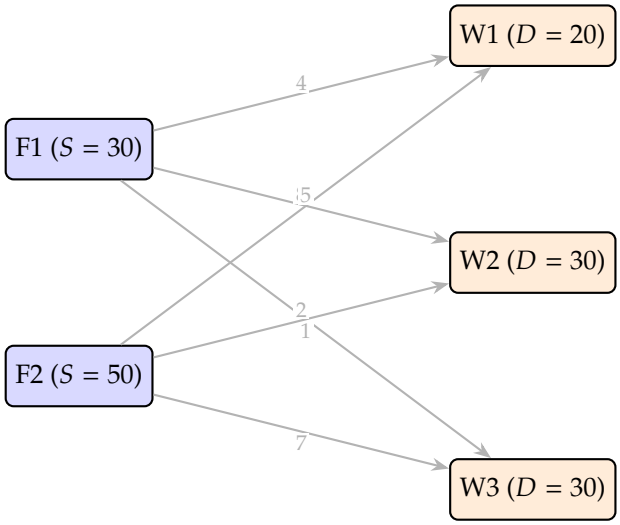


Figure 2.9: The transportation problem as a bipartite network. Factories (blue) on the left, warehouses (orange) on the right. Arc labels are per-unit shipping costs.

2.8.3 A blending / mixing problem

Example 2.8.3 (Oil blending)**.** A refinery blends two crude oils (Light and Heavy) into a single product. The product must have:

- sulphur content $\leq 1.5\%$,
- octane rating ≥ 90.

Data:

Crude	Cost ($/barrel)	Sulphur (%)	Octane
Light	30	0.5	100
Heavy	20	2.5	80

The refinery wants to produce exactly 100 barrels at minimum cost.

Variables: x_1 = barrels of Light, x_2 = barrels of Heavy.

Model:

$$
\begin{aligned}
\text{minimize} \quad & 30\,x_1 + 20\,x_2 && \\
\text{subject to} \quad & x_1 + x_2 = 100 && \text{(total production)} \\
& 0.5\,x_1 + 2.5\,x_2 \leq 1.5 \cdot 100 && \text{(sulphur)} \\
& 100\,x_1 + 80\,x_2 \geq 90 \cdot 100 && \text{(octane)} \\
& x_1, x_2 \geq 0. &&
\end{aligned}
$$

Simplifying:

$$\begin{aligned} \text{minimize} \quad & 30\,x_1 + 20\,x_2 \\ \text{subject to} \quad & x_1 + x_2 = 100 \\ & 0.5\,x_1 + 2.5\,x_2 \le 150 \\ & 100\,x_1 + 80\,x_2 \ge 9000 \\ & x_1, x_2 \ge 0. \end{aligned}$$

The blending problem illustrates a key modelling trick: **quality constraints** (sulphur content, octane rating) are linearised by multiplying both sides by the total quantity. This keeps everything linear. Exercises 41 and 42 present further blending scenarios for you to model.

2.8.4 Production planning over time

Example 2.8.4 (Multi-period production planning)**.** A factory produces a single good over $T = 3$ months. Demand, production capacity, and production cost vary by month. Unsold goods can be stored at a holding cost of €2 per unit per month. Initial inventory is zero.

	Month 1	**Month 2**	**Month 3**
Demand d_t	40	60	30
Capacity C_t	50	50	50
Prod. cost p_t	10	12	11

Variables:

- x_t = units produced in month t ($t = 1, 2, 3$).
- I_t = inventory at the end of month t ($t = 1, 2, 3$).

Model:

$$\begin{aligned} \text{minimize} \quad & \sum_{t=1}^{3} p_t\,x_t + 2\sum_{t=1}^{3} I_t \\ \text{subject to} \quad & I_t = I_{t-1} + x_t - d_t, \quad t = 1,2,3 && \text{(inventory balance)} \\ & 0 \le x_t \le C_t, \quad t = 1,2,3 && \text{(capacity)} \\ & I_t \ge 0, \quad t = 1,2,3 && \text{(no backorders)} \\ & I_0 = 0 && \text{(initial inventory).} \end{aligned}$$

This LP has 6 variables ($x_1, x_2, x_3, I_1, I_2, I_3$) and links periods through the **inventory balance** equations $I_t = I_{t-1} + x_t - d_t$. Such constraints are characteristic of dynamic LP models.

Exercises 54 and 55 ask you to model two further problems as LPs: a two-machine resource-allocation problem and a shortest-path problem.

Remark 2.8.5. All four examples above (diet, transportation, blending, production planning) are *linear programs*: the objective and every constraint are linear functions of the decision variables. Despite their very different real-world contexts, they all share the same mathematical structure and

> can all be solved by the same algorithm (the Simplex method of Chapter 4 or an interior-point method). This universality is what makes LP such a powerful tool.

2.9 Properties of LP: A Summary

Let's collect the key properties we have established in this chapter. These will serve as the foundation for everything that follows.

P1. Feasible region is a polyhedron. The set of all feasible solutions to an LP is a polyhedron $P = \{x : Ax \leq b\}$. If P is bounded, it is a polytope.

P2. Polyhedra are convex. Every polyhedron is a convex set. In particular, the feasible region of any LP is convex.

P3. Vertex optimality (theorem 2.4.6). If an LP has a finite optimum and the feasible region is a polytope, at least one optimal solution is a vertex.

P4. Finite vertex set. A polytope defined by m constraints in n variables has at most $\binom{m}{n}$ vertices. The Simplex method searches this finite set intelligently.

P5. Dual representations (theorem 2.5.4). A polytope can be described externally (half-spaces) or internally (convex hull of vertices).

P6. Form equivalence. Every LP can be converted to canonical or standard form using slack/surplus variables, variable splitting, and sign flips.

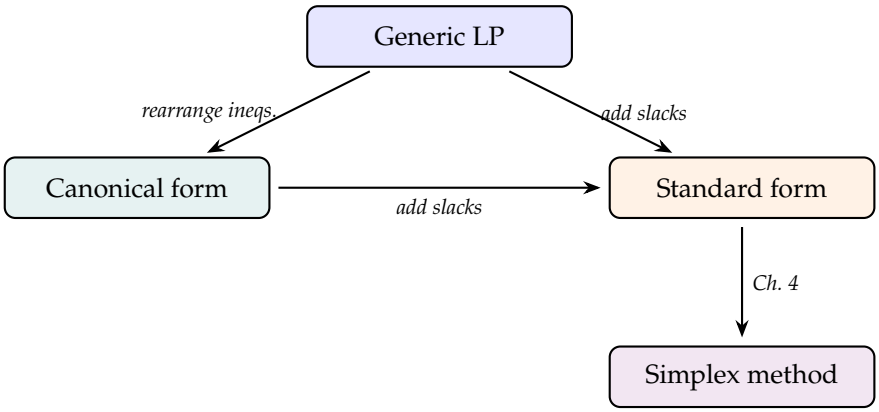


Figure 2.10: The relationships between LP forms. Every LP can be converted to canonical or standard form. The Simplex method operates on the standard form.

2.10 Looking Ahead

We now have a solid geometric and algebraic foundation for linear programming. In the next chapters, we will build on it:

- **Chapter 3 (Integer Linear Programming)** restricts variables to integers, turning the problem from "easy" (polynomial-time) to NP-hard.

- **Chapter 4 (Simplex Method)** gives us an algorithm that walks along the edges of the standard-form polytope, exploiting vertex optimality to find the optimum efficiently.
- **Chapter 5 (Duality)** reveals that every LP has a "mirror" LP whose optimal value equals the original's—a profound fact with algorithmic and economic interpretations.

All of this rests on the geometry we developed here: polyhedra, vertices, convexity, and the fundamental theorem.

Summary & Key Takeaways

- **Linear Programming (LP):** Continuous optimization where objective and constraints are all linear functions.
- **Geometry of LPs:** The set of feasible solutions forms a convex polyhedron $P = \{x \in \mathbb{R}^n : Ax \leq b\}$.
- **Extreme Points & Vertices:** A point $x \in P$ is a vertex if it cannot be represented as a convex combination of two distinct points in P. Vertices correspond to Algebraic Basic Feasible Solutions (BFS).
- **Fundamental Theorem of LP:** If an LP has a finite optimal solution, there exists an extreme point (vertex) of the feasible region that is optimal.

Exercises

Exercise 1. Consider the LP

$$\max\ 3x_1 + 2x_2 \quad \text{s.t.} \quad x_1 + x_2 \leq 4,\ 2x_1 + x_2 \leq 6,\ x_1, x_2 \geq 0.$$

Draw the feasible region, identify all vertices, evaluate the objective at each vertex, and state the optimal solution.

Exercise 2. Solve the following LP graphically:

$$\min\ x_1 - 2x_2 \quad \text{s.t.} \quad x_1 + 2x_2 \leq 8,\ x_1 - x_2 \leq 3,\ x_1, x_2 \geq 0.$$

Identify the optimal vertex and the optimal objective value.

Exercise 3. Consider the LP

$$\max\ 2x_1 + 2x_2 \quad \text{s.t.} \quad x_1 + x_2 \leq 5,\ x_1 \leq 3,\ x_2 \leq 3,\ x_1, x_2 \geq 0.$$

Solve it graphically. How many optimal solutions does it have? Describe the set of all optimal solutions.

Exercise 4. Use the graphical method to determine whether the LP

$$\max\ x_1 + x_2 \quad \text{s.t.} \quad -x_1 + x_2 \leq 1,\ x_1 - 2x_2 \leq 2,\ x_1, x_2 \geq 0$$

is feasible, infeasible, or unbounded. Justify your answer with a sketch.

Exercise 5. Solve graphically:

$$\min\ 4x_1 + 3x_2 \quad \text{s.t.} \quad 2x_1 + x_2 \geq 6,\ x_1 + 2x_2 \geq 6,\ x_1, x_2 \geq 0.$$

Identify the optimal vertex and confirm it is feasible.

Exercise 6. For the LP

$$\max\ 5x_1 + 4x_2 \quad \text{s.t.} \quad 6x_1 + 4x_2 \le 24,\ x_1 + 2x_2 \le 6,\ x_1, x_2 \ge 0,$$

sketch the feasible region, find all corner points algebraically (by solving pairs of binding constraints), and determine the optimal solution.

Exercise 7. Convert the following LP to standard form (minimisation, equality constraints, all variables ≥ 0):

$$\max\ 3x_1 - x_2 + 2x_3 \quad \text{s.t.} \quad x_1 + x_2 \le 5,\ 2x_1 - x_3 \ge 1,\ x_1 + x_2 + x_3 = 4,\ x_1, x_2, x_3 \ge 0.$$

Name every slack or surplus variable you introduce.

Exercise 8. Convert to standard form:

$$\min\ x_1 + 2x_2 \quad \text{s.t.} \quad x_1 - x_2 \ge 3,\ x_1 + 3x_2 \le 9,\ x_1 \ge 0,\ x_2 \text{ unrestricted}.$$

Show the substitution used for the unrestricted variable.

Exercise 9. Convert to standard form:

$$\max\ -2x_1 + x_2 + 3x_3 \quad \text{s.t.} \quad x_1 + x_2 + x_3 \le 10,\ x_1 - x_2 + 2x_3 \ge 2,\ x_1 \ge 0,\ x_2 \ge 0,\ x_3 \le 0.$$

Introduce $x_3' = -x_3 \ge 0$ and write the resulting standard-form problem.

Exercise 10. A linear programme has variables $x_1 \ge 0$, x_2 unrestricted, and $x_3 \le 0$. Write the substitutions needed to express all three variables as non-negative variables, and state how many new variables are introduced in the worst case.

Exercise 11. Convert the following LP to canonical form (all inequality constraints of the same sense, right-hand side ≥ 0, no equality constraints):

$$\min\ x_1 + x_2 \quad \text{s.t.} \quad 2x_1 + x_2 = 4,\ x_1 - x_2 \le 1,\ x_1, x_2 \ge 0.$$

Exercise 12. Let $P = \{(x_1, x_2) \in \mathbb{R}^2 : x_1 + x_2 \le 3,\ x_1 \ge 0,\ x_2 \ge 0\}$.

(a) List all vertices of P.

(b) For each vertex, state which constraints are active (tight).

(c) State the dimension of P.

Exercise 13. Consider the polyhedron

$$P = \{(x_1, x_2) \in \mathbb{R}^2 : x_1 \ge 0,\ x_2 \ge 0,\ x_1 + 2x_2 \le 6,\ 2x_1 + x_2 \le 6\}.$$

(a) Find all vertices of P by solving pairs of binding constraints.

(b) List all edges (1-dimensional faces) and the inequalities that define them.

Exercise 14. Prove that the feasible region of any LP with only inequality constraints (no equality constraints) and non-negative variables is a convex set.

Exercise 15. A point $\bar{x}$ in a polyhedron P is a *vertex* (extreme point) if and only if it cannot be written as $\bar{x} = \lambda y + (1-\lambda) z$ with $y, z \in P$, $y \ne z$, $0 < \lambda < 1$. Using this definition, show that $(0, 0)$ is a vertex of the polyhedron $P = \{x \in \mathbb{R}^2 : x_1 + x_2 \le 1,\ x_1 \ge 0,\ x_2 \ge 0\}$.

Exercise 16. Let $P \subset \mathbb{R}^2$ be the unit square $[0, 1]^2$.

(a) List all faces of P (vertices, edges, the whole square, and the empty face), and for each face give the supporting hyperplane that defines it.

(b) How many facets does P have?

Exercise 17. Give an example of a polyhedron in $\mathbb{R}^2$ that has infinitely many feasible points but *no* vertices. Describe it by its half-space representation and explain why no vertex exists.

Exercise 18. Consider the half-spaces $H_1 = \{x : a^\top x \le b\}$ and $H_2 = \{x : c^\top x \le d\}$ in $\mathbb{R}^n$. Prove that $H_1 \cap H_2$ is a convex set.

Exercise 19. Let $P = \{x \in \mathbb{R}^3 : x_1 + x_2 + x_3 = 1,\ x_1, x_2, x_3 \ge 0\}$.

(a) State the dimension of P.

(b) Find all vertices of P.

(c) Describe the edges of P.

Exercise 20. Prove that the intersection of any finite collection of convex sets is convex.

Exercise 21. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a convex function and $\alpha \in \mathbb{R}$. Show that the sublevel set $S = \{x : f(x) \le \alpha\}$ is a convex set.

Exercise 22. Give an example of a set that is *not* convex and explain which convexity condition it violates. Use a set in $\mathbb{R}^2$ defined by a simple inequality.

Exercise 23. (Convex hull.) Let $S = \{(0,0), (2,0), (1,2)\} \subset \mathbb{R}^2$.

(a) Describe $\mathrm{conv}(S)$ as a set of half-space inequalities.

(b) Is S itself convex? Justify.

Exercise 24. Prove that every polytope (bounded polyhedron) is equal to the convex hull of its vertices. (You may appeal to Minkowski–Weyl without reproving it, but you must state clearly which direction of the theorem you are using.)

Exercise 25. Express every point of the polytope $P = \mathrm{conv}\{(0,0), (4,0), (0,4), (2,2)\}$ as a convex combination of vertices. Use a general point $(x_1, x_2) \in P$ and write the corresponding coefficients $\lambda_1, \lambda_2, \lambda_3, \lambda_4 \ge 0$ with $\sum_i \lambda_i = 1$.

Exercise 26. The Minkowski–Weyl theorem states that every polyhedron P can be written as $P = Q + C$ where Q is a polytope and C is a polyhedral cone. For the polyhedron

$$P = \{(x_1, x_2) : x_1 + x_2 \ge 1,\ x_1 \ge 0,\ x_2 \ge 0\},$$

identify the lineality/recession directions and decompose P into the sum of a polytope and a cone.

Exercise 27. Show that the point $v = (1, 0, 0)$ is an extreme point (vertex) of the simplex $\Delta = \{x \in \mathbb{R}^3 : x_1 + x_2 + x_3 = 1,\ x_1, x_2, x_3 \ge 0\}$ by verifying the active-constraint characterisation.

Exercise 28. For each statement, state whether it is **true** or **false** and give a brief justification or counterexample.

(a) Every LP with a bounded feasible region has a finite optimal solution.

(b) If an LP is feasible and its objective function is bounded below (for a minimisation problem), then an optimal solution exists.

(c) The optimal value of an LP is always attained at a vertex of the feasible region.

(d) An LP can have exactly two optimal solutions.

Exercise 29. State whether each of the following is **true** or **false**, and justify briefly.

(a) The feasible region of every LP is a convex set.

(b) Adding a new constraint to an LP can only decrease or maintain the optimal objective value (for maximisation).

(c) The set of optimal solutions of an LP is always a convex set.

(d) If an LP in $\mathbb{R}^2$ has exactly one optimal solution, it must be a vertex of the feasible region.

Exercise 30. State whether each of the following is **true** or **false**, and justify briefly.

(a) A polyhedron defined by m inequality constraints in $\mathbb{R}^n$ has at most $\binom{m}{n}$ vertices.

(b) Every bounded polyhedron in $\mathbb{R}^n$ is a polytope.

(c) The empty set is a valid polyhedron.

(d) A single equality constraint $a^\top x = b$ in $\mathbb{R}^n$ defines a set of dimension $n - 1$.

Exercise 31. Write down a specific LP in two variables that is *infeasible*. Sketch its (empty) feasible region and explain why no feasible point exists.

Exercise 32. Write down a specific LP in two variables that is *unbounded* (the objective can be made arbitrarily large). Sketch the feasible region and show the direction along which the objective grows without bound.

Exercise 33. Write down an LP in two variables that has *infinitely many optimal solutions* forming a line segment. Identify the two optimal vertices and the entire optimal face.

Exercise 34. Write down an LP that is feasible and whose feasible region is unbounded, but the optimal value is still *finite*. Explain why boundedness of the feasible region is not necessary for the existence of an optimal solution.

Exercise 35. Consider the system $Ax \leq b$ in $\mathbb{R}^3$ given by: $x_1 \leq 2$, $x_2 \leq 2$, $x_3 \leq 2$, $x_1 \geq 0$, $x_2 \geq 0$, $x_3 \geq 0$.

(a) What is the dimension of the feasible polyhedron?

(b) How many vertices does it have?

(c) How many facets does it have?

Exercise 36. A point $x^* \in \mathbb{R}^3$ satisfies four out of six inequality constraints with equality. Under what conditions on the active-constraint matrix is x^* guaranteed to be a vertex (basic feasible solution)?

Exercise 37. (Diet problem.) A nutritionist must design a minimum-cost daily diet using three foods: bread, milk, and eggs. The relevant data are given below.

Nutrient	Bread (per slice)	Milk (per cup)	Eggs (per egg)
Protein (g)	3	8	6
Fat (g)	1	5	5
Carbohydrates (g)	15	12	1
Cost (€)	0.10	0.20	0.15

Minimum daily requirements: 55 g protein, 33 g fat, 70 g carbohydrates. Formulate an LP to minimise total daily food cost subject to the nutritional requirements. Define all variables clearly.

Exercise 38. (Diet problem, extension.) Extend Exercise 37 by adding the constraint that the number of eggs consumed per day may not exceed 3. How does this change the LP formulation? Is the new LP still a linear programme?

Exercise 39. (Transportation problem.) A company has two warehouses (W_1, W_2) and three retail shops (S_1, S_2, S_3). Supplies are $s_1 = 120$ and $s_2 = 80$ units; demands are $d_1 = 70$, $d_2 = 90$, $d_3 = 40$ units. Unit shipping costs c_{ij} (in €) are:

$$C = \begin{pmatrix} 2 & 3 & 1 \\ 5 & 4 & 8 \end{pmatrix}.$$

Formulate an LP that minimises total shipping cost while satisfying all supply and demand constraints.

Exercise 40. (Transportation problem, feasibility.) Using the transportation problem of Exercise 39, determine whether total supply equals total demand. If not, explain how to add a dummy warehouse or dummy shop to make the problem balanced.

Exercise 41. (Blending problem.) A paint manufacturer blends two raw materials, R_1 and R_2, to produce three products: P_1, P_2, P_3. Each litre of P_1 requires 0.4 L of R_1 and 0.6 L of R_2; each litre of P_2 requires 0.7 L of R_1 and 0.3 L of R_2; each litre of P_3 requires 0.5 L of each. Available supply: 600 L of R_1 and 500 L of R_2. Profits per litre: €8 for P_1, €6 for P_2, €7 for P_3. Formulate an LP to maximise total profit.

Exercise 42. (Blending with quality constraints.) An oil company blends two crude oils, A and B, to produce petrol. Crude A costs €30/barrel, crude B costs €20/barrel. The petrol must contain at least 60% of crude A by volume (quality constraint). The company must produce at least 1000 barrels of petrol. Formulate an LP to minimise total blending cost.

Exercise 43. Prove or disprove: if P and Q are polyhedra in $\mathbb{R}^n$, then their union $P \cup Q$ is also a polyhedron.

Exercise 44. Prove or disprove: if P is a polytope and Q is a polytope, then $P + Q = \{x + y : x \in P,\ y \in Q\}$ is also a polytope.

Exercise 45. Prove that every face of a polyhedron is itself a polyhedron.

Exercise 46. Prove or disprove: the projection of a polyhedron onto a coordinate subspace is always a polyhedron.

Exercise 47. Show that $\max\{c^\top x : x \in P\}$ is equivalent to $-\min\{-c^\top x : x \in P\}$. Use this equivalence to convert the maximisation LP

$$\max\ 5x_1 - 3x_2 \quad \text{s.t.} \quad x_1 + x_2 \le 4,\ x_1, x_2 \ge 0$$

into an equivalent minimisation LP in standard form.

Exercise 48. An LP has a constraint $|x_1 - x_2| \le 3$. This is not a linear constraint as written. Show how to replace it with two linear constraints and incorporate them into the LP formulation.

Exercise 49. Given the LP

$$\max\ 2x_1 + x_2 \quad \text{s.t.} \quad x_1 + x_2 + s_1 = 5,\ 2x_1 - x_2 + s_2 = 4,\ x_1, x_2, s_1, s_2 \ge 0,$$

identify which constraints are active at the point $(2, 1)$. Is $(2, 1)$ a basic feasible solution? What are the values of the slack variables?

Exercise 50. Consider the constraint $3x_1 + 2x_2 \ge 12$. Introduce a surplus variable $s \ge 0$ and rewrite it as an equality. If $x_1 = 3$ and $x_2 = 1$, what is the value of s? Is the constraint satisfied?

Exercise 51. How many linearly independent constraints are active at a vertex of a polyhedron in $\mathbb{R}^n$? Why must this number be at least n? Give an example in $\mathbb{R}^3$ where exactly $n = 3$ constraints are active at a vertex.

Exercise 52. Consider the LP in $\mathbb{R}^4$:

$$\max\ x_1 + x_2 + x_3 + x_4 \quad \text{s.t.} \quad x_1 + x_2 + x_3 + x_4 = 1,\ x_1, x_2, x_3, x_4 \ge 0.$$

(a) What is the dimension of the feasible polytope?

(b) List all vertices.

(c) What is the optimal value? Is it attained at a unique vertex?

Exercise 53. Rewrite the following problem as a linear programme (linearise the objective and constraints):

$$\min\ \max(2x_1 + x_2,\ x_1 + 3x_2) \quad \text{s.t.} \quad x_1 + x_2 \le 4,\ x_1, x_2 \ge 0.$$

Hint: introduce an auxiliary variable $t = \max(2x_1 + x_2,\ x_1 + 3x_2)$.

Exercise 54. (Resource allocation.) A factory produces two products A and B using two machines M_1 and M_2. Each unit of A requires 2 hours on M_1 and 1 hour on M_2. Each unit of B requires 1 hour on M_1 and 3 hours on M_2. Available hours: 120 on M_1, 150 on M_2. Profits: €5 per unit of A, €4 per unit of B. Formulate an LP to maximise profit. Identify all vertices of the feasible region and find the optimal solution graphically.

Exercise 55. (Shortest-path as LP.) Let G be a directed graph with nodes $\{1, 2, 3\}$ and arcs $(1, 2)$, $(1, 3)$, $(2, 3)$ with costs $c_{12} = 4$, $c_{13} = 7$, $c_{23} = 2$. Formulate an LP whose solution gives the shortest path from node 1 to node 3. Define the flow variables and the flow-conservation constraints.

CHAPTER 3

Integer Linear Programming

This chapter extends LP to the world of discrete decisions: integer and binary variables, combinatorial models, and the fundamental gap between LP and ILP.

In chapter 2 we developed a rich theory of linear programming, where every variable is free to take any real value in its domain. But the furniture workshop LP (theorem 1.8.1) already hinted at a limitation: the optimal solution $(5, 6)$ happened to be integer, but what if it hadn't been? What if the LP optimum were $(5.3, 5.7)$? We cannot produce 0.3 of a chair.

The knapsack instance of theorem 1.10.1 makes the point even more sharply: each item is either packed or not—there is no such thing as packing "half" of item 3. Many of the most important real-world decisions are inherently discrete: build a factory or don't, assign a nurse to a shift or don't, route a truck through city A or through city B.

This chapter introduces **Integer Linear Programming** (ILP), the framework that handles such decisions. We will see how to model a surprising variety of problems with binary and integer variables, study the relationship between an ILP and its LP relaxation, and begin to understand why integer programming is fundamentally harder than continuous LP.

Road map.

1. Definitions: ILP, MILP, binary programs (section 3.1).
2. Modeling with binary variables (section 3.2).
3. The knapsack problem (section 3.3).
4. The set covering problem (section 3.4).
5. Non-linear costs via ILP tricks (section 3.5).
6. Logical constraints (section 3.6).
7. Classic ILP models (section 3.7).
8. LP relaxation and the integrality gap (section 3.8).

3.1 Definitions: ILP, MILP, and Binary Programs

"Integer programming" does not *mean the data must be integer —only that the* variables *must be.*

Let's start with the formal definition.

Definition 3.1.1 (Integer Linear Program — ILP)**.** An **Integer Linear Pro-**

gram (ILP) is an optimization problem of the form

$$\begin{aligned} \text{maximize} \quad & c^\top x \\ \text{subject to} \quad & Ax \le b, \\ & x \ge 0, \\ & x \in \mathbb{Z}^n, \end{aligned}$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and every variable x_j is required to be a **non-negative integer**.

Intuitively, an ILP is "an LP plus integrality constraints." The feasible region is no longer a polyhedron—it is a *finite* set of lattice points inside a polyhedron. This seemingly small change has profound consequences: while LP can be solved in polynomial time (e.g. via interior-point methods), ILP is **NP-hard** in general.

LP is in P. ILP is NP-hard. This is one of the sharpest complexity jumps in optimization.

Remark 3.1.2. The definition above uses $\le$ constraints and maximisation, but as with LP, we can equivalently use $\ge$, $=$, minimisation, or any mix. The essential feature is that variables must be integers.

In practice, we often encounter two important variants.

Definition 3.1.3 (Mixed Integer Linear Program — MILP)**.** A **Mixed Integer Linear Program (MILP)** is an optimisation problem of the form

$$\begin{aligned} \text{maximize} \quad & c^\top x + d^\top y \\ \text{subject to} \quad & Ax + By \le b, \\ & x \ge 0, \quad y \ge 0, \\ & x \in \mathbb{Z}^p, \quad y \in \mathbb{R}^q, \end{aligned}$$

where x contains the p **integer** variables and y contains the q **continuous** variables, with $p + q = n$.

In a MILP, only *some* variables are required to be integer. This is the most common setting in practice: a production planning model might have integer variables for how many factories to open and continuous variables for how much to produce at each factory.

Definition 3.1.4 (Binary (0–1) Program)**.** A **Binary Program** (or **0–1 Integer Program**) is an ILP in which every variable is restricted to $\{0, 1\}$:

$$\begin{aligned} \text{maximize} \quad & c^\top x \\ \text{subject to} \quad & Ax \le b, \\ & x \in \{0,1\}^n. \end{aligned}$$

Binary programs are the workhorse of combinatorial optimisation. Every yes/no decision—select an item, assign a worker, activate a link—naturally maps to a binary variable. The knapsack instance of theorem 1.10.1 is a binary program.

Intermezzo — Why is ILP NP-hard?

Recall from chapter 1 that a problem is NP-hard if every problem in NP can be reduced to it in polynomial time. The Boolean satisfiability problem (SAT)—a foundational NP-complete problem—can be encoded as a 0–1 ILP (we will see how in section 3.7.3). Since SAT is NP-complete and reduces to ILP, ILP is at least as hard as any problem in NP. This means (assuming P ≠ NP) there is no polynomial-time algorithm that solves every ILP instance.

In contrast, LP is solvable in polynomial time: the ellipsoid method (Khachiyan, 1979) and interior-point methods (Karmarkar, 1984) both achieve this. The Simplex method, while exponential in the worst case, is extremely fast in practice.

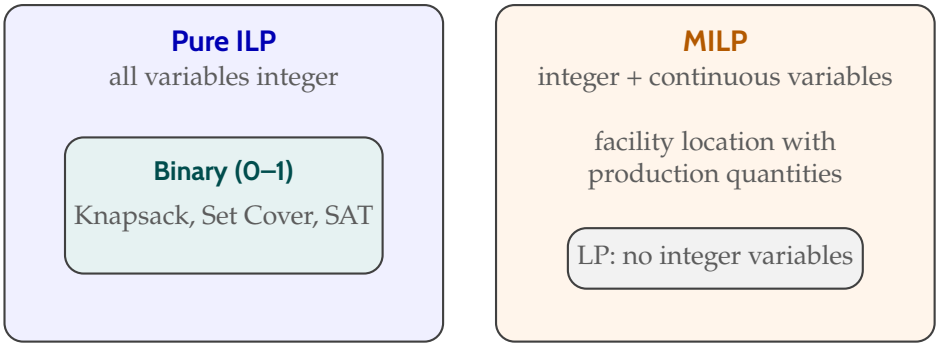


Figure 3.1: Relationships among linear optimisation models with integrality. Binary programs are a special case of pure ILP. A MILP allows both integer and continuous variables; pure ILP and LP appear as boundary cases when all variables, or no variables, are constrained to be integer.

3.2 Modeling with Binary Variables

The power of integer programming comes from the ability to model discrete choices. In this section we develop the key principles.

The art of ILP modeling is largely the art of choosing the right binary variables.

3.2.1 Selection decisions

The simplest pattern: we have n items (projects, locations, routes), and we must *select* a subset. For each item i, introduce

$$x_i = \begin{cases} 1 & \text{if item } i \text{ is selected,} \\ 0 & \text{otherwise.} \end{cases}$$

The knapsack (theorem 1.10.1) is exactly this pattern: $x_j = 1$ means item j is packed.

Example 3.2.1 (Project selection). A company has 5 candidate projects. Each project i requires an investment c_i and yields an expected return r_i.

The total budget is B. We want to maximize total return:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{5} r_i\, x_i \\ \text{subject to} \quad & \sum_{i=1}^{5} c_i\, x_i \le B, \\ & x_i \in \{0,1\}, \quad i = 1,\dots,5. \end{aligned}$$

This is, of course, a knapsack—budget plays the role of capacity, and investment cost plays the role of weight. The knapsack structure appears whenever we select from a set subject to a single resource constraint.

3.2.2 The golden rule: one binary variable per choice

Never encode categorical choices as a single integer variable. Use one binary variable per option.

Suppose we must choose *one* warehouse location among three cities: Milan, Rome, Naples. A common mistake is to introduce a single variable $y \in \{1,2,3\}$, where $y = 1$ means Milan, $y = 2$ means Rome, and $y = 3$ means Naples.

This is **wrong**—or at least extremely poor modeling. The encoding introduces a spurious numerical relationship: it suggests that Rome ($y = 2$) is somehow "between" Milan ($y = 1$) and Naples ($y = 3$), or that Naples costs "three times" as much as Milan. The numbers 1, 2, 3 are just labels with no quantitative meaning.

The correct approach uses **one binary variable per option**:

$$y_{\text{Mi}} \in \{0,1\}, \quad y_{\text{Ro}} \in \{0,1\}, \quad y_{\text{Na}} \in \{0,1\}, \quad y_{\text{Mi}} + y_{\text{Ro}} + y_{\text{Na}} = 1.$$

The constraint $\sum y_i = 1$ ensures exactly one city is chosen. Each city's cost, capacity, or distance enters the model through its own coefficient, without any artificial numerical bias.

Remark 3.2.2. This "one-hot" encoding is the standard approach in ILP whenever choices are categorical. It generalises cleanly: if we must choose exactly k out of n options, we use n binary variables with $\sum_{i=1}^{n} x_i = k$.

3.2.3 Multi-index variables

Many real problems involve decisions with two or more dimensions.

Example 3.2.3 (Worker–job assignment)**.** A company has m workers and n jobs. Worker i performing job j costs c_{ij}. Each worker is assigned to exactly one job, and each job is done by exactly one worker ($m = n$). Define

$$x_{ij} = \begin{cases} 1 & \text{if worker } i \text{ is assigned to job } j, \\ 0 & \text{otherwise.} \end{cases}$$

The assignment problem is:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}\, x_{ij} \\
\text{subject to} \quad & \sum_{j=1}^{n} x_{ij} = 1 \qquad \text{for all } i \quad \text{(each worker gets one job)}, \\
& \sum_{i=1}^{n} x_{ij} = 1 \qquad \text{for all } j \quad \text{(each job gets one worker)}, \\
& x_{ij} \in \{0,1\}.
\end{aligned}$$

The double index x_{ij} captures a two-dimensional decision (who does what). Triple indices x_{ijk} appear in scheduling (worker i, job j, time slot k) and routing (vehicle i, customer j, position k). The modeling principle is always the same: one binary variable for each possible combination.

The assignment problem has a beautiful property: its LP relaxation always gives integer solutions. We will see why in chapter 7, when we study totally unimodular matrices.

3.3 The Knapsack Problem

We already met the knapsack in theorem 1.10.1. Let's now study it systematically.

The knapsack problem is arguably the most fundamental combinatorial optimisation problem.

3.3.1 The binary knapsack

Definition 3.3.1 (Binary Knapsack Problem)**.** Given n items, each with a **weight** $w_j > 0$ and a **value** $v_j > 0$, and a knapsack of **capacity** $W > 0$, the **Binary Knapsack Problem** is:

$$\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{n} v_j\, x_j \\
\text{subject to} \quad & \sum_{j=1}^{n} w_j\, x_j \le W, \\
& x_j \in \{0,1\}, \quad j = 1, \ldots, n.
\end{aligned}$$

Each item is either taken ($x_j = 1$) or left behind ($x_j = 0$). The single capacity constraint makes the knapsack structurally the simplest ILP—yet it is already NP-hard.

Example 3.3.2 (Running knapsack as a binary program)**.** Returning to theorem 1.10.1, we have $n = 6$ items with capacity $W = 15$:

Item j	1	2	3	4	5	6
Weight w_j	5	3	7	4	2	6
Value v_j	8	5	11	6	4	9
Ratio v_j/w_j	1.60	1.67	1.57	1.50	2.00	1.50

The ILP formulation is:

$$\begin{aligned} \text{maximize} \quad & 8x_1 + 5x_2 + 11x_3 + 6x_4 + 4x_5 + 9x_6 \\ \text{subject to} \quad & 5x_1 + 3x_2 + 7x_3 + 4x_4 + 2x_5 + 6x_6 \leq 15, \\ & x_j \in \{0,1\}, \quad j = 1,\ldots,6. \end{aligned}$$

In chapter 1 we found the feasible selection $\{1,2,4,5\}$ with weight 14 and value 23 by greedy reasoning. Is it optimal? We will answer this question shortly using LP relaxation.

3.3.2 The integer knapsack

A natural variant allows *multiple copies* of each item.

Definition 3.3.3 (Integer Knapsack Problem)**.** The **Integer Knapsack Problem** (or **Unbounded Knapsack**) replaces the binary constraint with a non-negative integer constraint:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} v_j\, x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j\, x_j \leq W, \\ & x_j \in \mathbb{Z}_{\geq 0}, \quad j = 1,\ldots,n. \end{aligned}$$

Here x_j represents the **number of copies** of item j packed.

Example 3.3.4 (Integer knapsack variant)**.** With the data from theorem 1.10.1, if we allow multiple copies, we could try packing 7 copies of item 5 (weight 2 each): total weight $14 \leq 15$, total value $7 \times 4 = 28 > 23$. That already beats the binary solution! Item 5 has the best value-to-weight ratio ($v_5/w_5 = 2.00$), so filling the knapsack with copies of it is a natural greedy idea.

3.3.3 LP relaxation of the knapsack

What happens if we *drop* the integrality constraint and allow $x_j \in [0,1]$ instead of $x_j \in \{0,1\}$? We get the **LP relaxation**—a continuous LP that is easy to solve.

Definition 3.3.5 (LP Relaxation)**.** The **LP relaxation** of a binary knapsack is:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} v_j\, x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j\, x_j \leq W, \\ & 0 \leq x_j \leq 1, \quad j = 1,\ldots,n. \end{aligned}$$

The LP relaxation of the knapsack has a beautiful closed-form solution.

Theorem 3.3.6 (Dantzig's greedy solution for the fractional knapsack)**.** *Sort the items by decreasing* ***value-to-weight ratio*** v_j/w_j*. Pack items greedily in this order. When an item does not fit entirely, pack the fraction that fills the remaining capacity. The resulting solution is optimal for the LP relaxation.*

Example 3.3.7 (LP relaxation of the running knapsack)**.** We sort the items of theorem 1.10.1 by v_j/w_j:

Order	1st	2nd	3rd	4th	5th	6th
Item	5	2	1	3	4	6
v_j/w_j	2.00	1.67	1.60	1.57	1.50	1.50
w_j	2	3	5	7	4	6

Pack greedily:

- Item 5: $x_5 = 1$, remaining capacity $15 - 2 = 13$.
- Item 2: $x_2 = 1$, remaining capacity $13 - 3 = 10$.
- Item 1: $x_1 = 1$, remaining capacity $10 - 5 = 5$.
- Item 3: $w_3 = 7 > 5$, so $x_3 = 5/7 \approx 0.714$. Remaining capacity 0.

Items 4 and 6 are not packed: $x_4 = x_6 = 0$.

The LP relaxation value is:

$$4 + 5 + 8 + \tfrac{5}{7} \cdot 11 = 17 + \tfrac{55}{7} = \tfrac{119+55}{7} = \tfrac{174}{7} \approx 24.86.$$

The LP relaxation value ≈ 24.86 *is an* upper bound *on the binary knapsack optimum. Since the optimum must be integer, we even know the optimum* ≤ 24.

Notice what happened: the LP relaxation found a *fractional* solution—it packed 5/7 of item 3. This is not feasible for the original binary knapsack. But the LP relaxation value (≈ 24.86) is an **upper bound** on the ILP optimum, since the LP relaxation has a larger feasible set.

We already know a feasible binary solution with value 23 (the selection $\{1, 2, 4, 5\}$ from theorem 1.10.1). So the ILP optimum is between 23 and 24. We will return to this bounding idea in section 3.8.

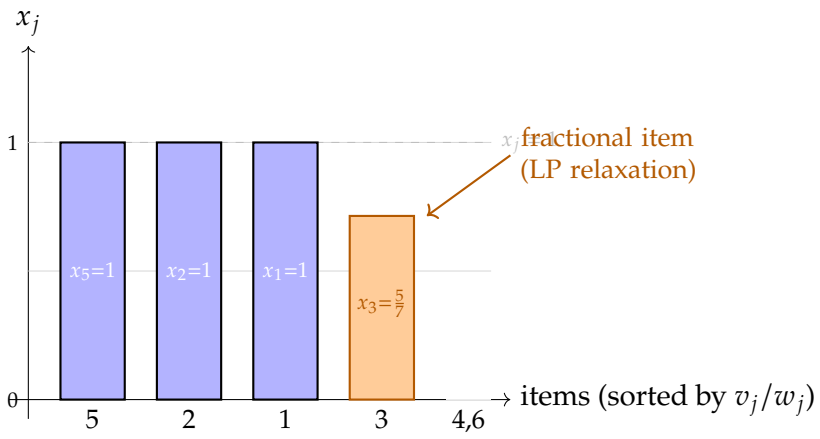


Figure 3.2: LP relaxation of the running knapsack. Items are sorted by value-to-weight ratio. Blue bars show fully packed items; the orange bar shows the fractional item (item 3, packed to 5/7).

3.4 The Set Covering Problem

Set covering is a canonical NP-hard problem with applications in logistics, scheduling, and facility location.

The knapsack asks us to select items subject to a capacity. The **set covering problem** asks us to select sets that together "cover" everything we need.

Definition 3.4.1 (Set Covering Problem)**.** Given a **universe** $U = \{e_1, e_2, \ldots, e_n\}$ of elements, a collection of **subsets** $S_1, S_2, \ldots, S_m \subseteq U$, and **costs** $c_1, c_2, \ldots, c_m \geq 0$, the **Set Covering Problem** asks for a minimum-cost subcollection that covers every element:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{m} c_j\, y_j \\ \text{subject to} \quad & \sum_{j:\, e_i \in S_j} y_j \geq 1 \quad \text{for all } i = 1, \ldots, n, \\ & y_j \in \{0,1\}, \quad j = 1, \ldots, m. \end{aligned}$$

Here $y_j = 1$ means we select subset S_j. The constraint says: for every element e_i, at least one selected subset must contain e_i.

The covering constraint $\sum_{j:\, e_i \in S_j} y_j \geq 1$ is the heart of the model. It says: element e_i must appear in at least one chosen subset.

Example 3.4.2 (Fire station placement)**.** A city has 6 neighbourhoods. Five possible sites for fire stations are being considered. Each site covers certain neighbourhoods (within response time) and has a construction cost:

Site j	Covers neighbourhoods	Cost c_j
1	$\{1,2,3\}$	5
2	$\{2,4\}$	3
3	$\{3,5,6\}$	4
4	$\{1,5\}$	3
5	$\{4,5,6\}$	4

We must place fire stations so every neighbourhood is covered. The ILP:

$$\begin{aligned} \text{minimize} \quad & 5y_1 + 3y_2 + 4y_3 + 3y_4 + 4y_5 \\ \text{subject to} \quad & y_1 + y_4 \geq 1 && \text{(nbhd 1)}, \\ & y_1 + y_2 \geq 1 && \text{(nbhd 2)}, \\ & y_1 + y_3 \geq 1 && \text{(nbhd 3)}, \\ & y_2 + y_5 \geq 1 && \text{(nbhd 4)}, \\ & y_3 + y_4 + y_5 \geq 1 && \text{(nbhd 5)}, \\ & y_3 + y_5 \geq 1 && \text{(nbhd 6)}, \\ & y_j \in \{0,1\},\ j = 1, \ldots, 5. \end{aligned}$$

By inspection, choosing sites $\{1,5\}$ covers all 6 neighbourhoods at cost $5 + 4 = 9$. Can we do better? Sites $\{1,2,3\}$ also cover everything at cost $5 + 3 + 4 = 12$. The selection $\{1,5\}$ is cheaper. In fact, $\{1,5\}$ is optimal (one

can verify by checking all $2^5 = 32$ subsets, or by solving the LP relaxation for a lower bound).

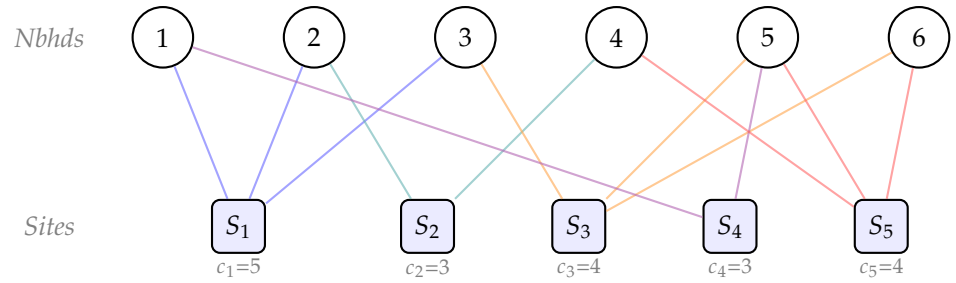


Figure 3.3: Set covering instance for fire station placement (theorem 3.4.2). Each line connects a site to the neighbourhoods it covers. The optimal solution selects sites S_1 and S_5 (cost 9), covering all 6 neighbourhoods.

Remark 3.4.3. Real-world applications of set covering include:

- **Airline crew scheduling**: cover all flight legs with minimum-cost crew pairings.
- **Facility location**: place facilities so all demand points are served.
- **Sensor placement**: cover an area with minimum sensors.

3.5 Modeling Non-Linear Costs with ILP

ILP can handle certain non-linearities by introducing auxiliary binary variables.

So far, our objectives and constraints have been purely linear. But many real-world costs are *not* linear—there are fixed startup costs, economies of scale, or piecewise pricing. Remarkably, ILP can model several of these patterns exactly.

3.5.1 Fixed (startup) costs

Consider a factory that costs nothing to operate if it is closed, but incurs a **fixed cost** V just for being open, plus a **variable cost** c per unit produced. If we produce x units, the total cost is:

$$f(x) = \begin{cases} 0 & \text{if } x = 0, \\ V + cx & \text{if } x > 0. \end{cases}$$

This is *not* a linear function: there is a discontinuity at $x = 0$. We cannot model it directly in an LP.

The trick is to introduce a **binary indicator variable** $y \in \{0, 1\}$, where $y = 1$ means "the factory is open." We then add:

$$\text{cost} = Vy + cx, \tag{3.1}$$

$$x \leq My, \tag{3.2}$$

where M is a large constant (an upper bound on the maximum possible production).

The constant M should be as tight *as possible. A lazy choice of $M = 10^6$ weakens the LP relaxation dramatically.*

How does this work?

- If $y = 0$ (factory closed): constraint (3.2) forces $x \leq 0$, so $x = 0$ and the cost is 0.

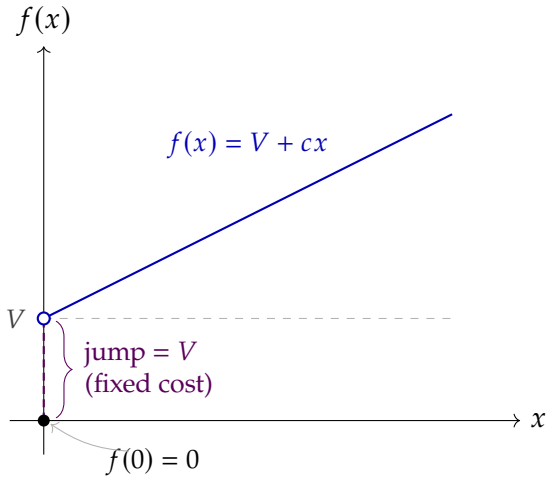


Figure 3.4: Fixed-cost function: $f(0) = 0$, but $f(x) = V + cx$ for $x > 0$. The function jumps by the fixed cost V the instant x becomes positive (filled dot: value attained at the origin; open dot: limit not attained). This discontinuity cannot be modeled by LP, but is captured by a binary variable.

- If $y = 1$ (factory open): $x \le M$ is non-binding (since M is large enough), x is free to be positive, and the cost is $V + cx$.

Example 3.5.1 (Facility opening with fixed costs)**.** A company can open up to 3 factories. Factory i has fixed cost V_i, per-unit cost c_i, and maximum capacity K_i. Total demand is D.

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{3} (V_i y_i + c_i x_i) \\ \text{subject to} \quad & x_i \le K_i\, y_i \qquad\qquad i = 1,2,3, \\ & \sum_{i=1}^{3} x_i \ge D, \\ & x_i \ge 0, \quad y_i \in \{0,1\}. \end{aligned}$$

Here $M = K_i$ is the tightest possible choice for each factory—we use the capacity itself.

3.5.2 Piecewise linear costs

Sometimes the unit cost changes at certain **breakpoints**. For instance, the first 100 units cost €10 each, the next 200 cost €8, and anything beyond 300 costs €6.

We model this by splitting the production into **segments**. Let $b_0 = 0 < b_1 < b_2 < \cdots$ be the breakpoints and c_k the unit cost in segment k. Introduce variables $z_k \ge 0$ for the amount produced in segment k:

$$x = z_1 + z_2 + z_3, \qquad 0 \le z_k \le b_k - b_{k-1}.$$

The cost becomes $c_1 z_1 + c_2 z_2 + c_3 z_3$, which is linear in the z_k variables.

When the slopes are decreasing *(economies of scale, concave cost), we also need binary variables and linking constraints to ensure segments fill in order.*

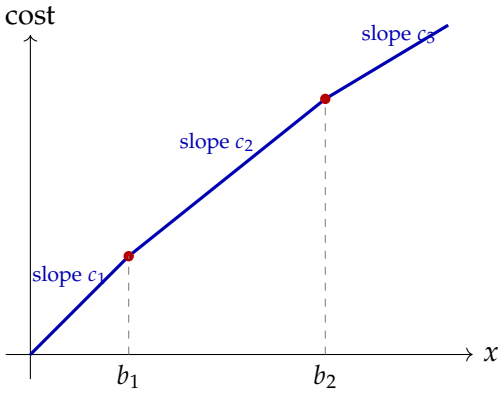


Figure 3.5: A piecewise linear cost function with two breakpoints. The slopes decrease (economies of scale), so the function is *concave*.

When the cost function is **convex** (increasing slopes), the LP solver will automatically fill segments in order (cheapest first), and no binary variables are needed. When the cost function is **concave** (decreasing slopes, as in economies of scale), we must introduce binary indicator variables $\delta_k \in \{0, 1\}$ and linking constraints $z_{k+1} \leq (b_{k+1} - b_k)\,\delta_k$ and $z_k \geq (b_k - b_{k-1})\,\delta_k$ to enforce that segment k is fully used before segment $k + 1$ begins.

3.6 Logical Constraints with Binary Variables

Binary variables turn logic into algebra.

One of the most powerful features of binary variables is their ability to encode **logical relationships**. Since $x \in \{0, 1\}$ naturally represents TRUE/FALSE, we can translate propositional logic into linear inequalities.

3.6.1 Basic logical operations

Let $x_1, x_2, \ldots, x_n \in \{0, 1\}$ be binary decision variables.

Definition 3.6.1 (Logical constraints as linear inequalities)**.**

Logic	Meaning	Constraint
OR (disjunction)	at least one of $x_1, \ldots, x_n$	$x_1 + x_2 + \cdots + x_n \geq 1$
AND (conjunction)	all of $x_1, \ldots, x_n$	$x_i = 1$ for all i
XOR (exclusive or)	exactly one of $x_1, \ldots, x_n$	$x_1 + x_2 + \cdots + x_n = 1$
Implication	if x_1 then x_2	$x_1 \leq x_2$
At most k	at most k selected	$x_1 + x_2 + \cdots + x_n \leq k$
At least k	at least k selected	$x_1 + x_2 + \cdots + x_n \geq k$
Exactly k	exactly k selected	$x_1 + x_2 + \cdots + x_n = k$

Let's verify the implication encoding. "If x_1 then x_2" means: whenever $x_1 = 1$, we must have $x_2 = 1$. The constraint $x_1 \leq x_2$ does exactly this: if $x_1 = 1$ then $x_2 \geq 1$, so $x_2 = 1$. If $x_1 = 0$, then $x_2 \geq 0$, which is always satisfied—x_2 is free.

Example 3.6.2 (Prerequisite constraints)**.** A university must decide which of 5 courses to offer. Course 3 requires course 1 as a prerequisite; course 5 requires both course 2 and course 4.

$$x_3 \leq x_1, \qquad x_5 \leq x_2, \qquad x_5 \leq x_4.$$

If we decide to offer course 5 ($x_5 = 1$), we are forced to also offer courses 2 and 4.

3.6.2 Negation and more complex logic

To encode the negation $\neg x_i$, we use $(1 - x_i)$, which is 1 when $x_i = 0$ and 0 when $x_i = 1$. This lets us handle richer logic:

- **If** x_1 **then NOT** x_2 (incompatibility): $x_1 + x_2 \leq 1$.
- **If NOT** x_1 **then** x_2 (fallback): $(1 - x_1) \leq x_2$, i.e. $x_1 + x_2 \geq 1$.
- $x_3 =$ x_1 **AND** x_2: $x_3 \leq x_1$, $x_3 \leq x_2$, $x_3 \geq x_1 + x_2 - 1$.
- $x_3 =$ x_1 **OR** x_2: $x_3 \geq x_1$, $x_3 \geq x_2$, $x_3 \leq x_1 + x_2$.

3.6.3 The Big-M method for conditional constraints

Big-M turns "if-then" relationships between constraints into linear inequalities.

Sometimes we want a constraint to be active only when a certain condition holds. For example: "*if* we open factory i, *then* the production at factory i must be at least 50 units."

The **Big-M method** uses a large constant M to "switch off" a constraint when the condition is not met.

Theorem 3.6.3 (Big-M conditional constraint)**.** *Let $a^\top x \leq b$ be a constraint that should hold only when $y = 1$ (where $y \in \{0, 1\}$). The inequality*

$$a^\top x \leq b + M(1 - y)$$

enforces $a^\top x \leq b$ when $y = 1$ and is non-binding when $y = 0$ (since M is large enough to make the right-hand side exceed any feasible $a^\top x$).

Example 3.6.4 (Conditional minimum production)**.** Factory i is open if $y_i = 1$. If open, production x_i must be at least 50:

$$x_i \geq 50 - M(1 - y_i), \qquad x_i \leq K_i\, y_i.$$

When $y_i = 1$: first constraint becomes $x_i \geq 50$; second becomes $x_i \leq K_i$. When $y_i = 0$: first becomes $x_i \geq 50 - M$ (non-binding since $x_i \geq 0$); second forces $x_i = 0$.

Remark 3.6.5. Choosing M wisely is critical. A tighter M (closer to the actual maximum of $a^\top x$ over the feasible region) gives a **stronger LP relaxation**. A lazy $M = 10^9$ makes the LP relaxation extremely weak, leading to slow branch-and-bound performance (Chapter 6).

3.6.4 Either/or constraints

Sometimes exactly one of two constraints must hold, but we don't know which in advance. For example, two jobs cannot overlap in time: either job 1 finishes before job 2 starts, or vice versa.

Using a binary variable $y \in \{0, 1\}$:

$$a_1^\top x \le b_1 + M(1 - y), \tag{3.3}$$
$$a_2^\top x \le b_2 + My. \tag{3.4}$$

- If $y = 1$: constraint (3.3) is non-binding, constraint (3.4) is active.
- If $y = 0$: constraint (3.3) is active, constraint (3.4) is non-binding.

Exactly one constraint is enforced, depending on the value of y that the solver chooses.

Example 3.6.6 (Job sequencing)**.** Two jobs with processing times p_1, p_2 and start times s_1, s_2 cannot overlap on the same machine. Either job 1 finishes before job 2 starts, or job 2 finishes before job 1 starts:

$$s_1 + p_1 \le s_2 + M(1 - y), \qquad s_2 + p_2 \le s_1 + My.$$

The solver picks y to enforce whichever ordering is cheaper.

3.7 Classic ILP Models

We now present three classic combinatorial problems that arise frequently in applications and illustrate different modeling techniques.

3.7.1 Weighted Independent Set

Definition 3.7.1 (Independent Set)**.** Given a graph $G = (V, E)$, an **independent set** is a subset $S \subseteq V$ such that no two vertices in S are adjacent: for all $\{i, j\} \in E$, at most one of i, j belongs to S.

Intuitively, an independent set is a collection of vertices with no edges between them—no two selected vertices are "in conflict."

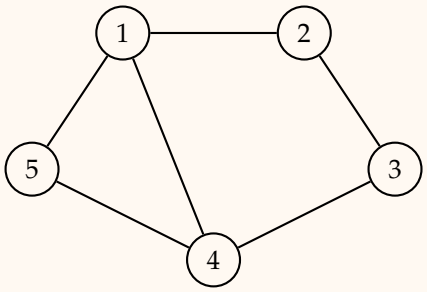
Definition 3.7.2 (Weighted Independent Set Problem)**.** Given a graph $G = (V, E)$ with vertex weights $w_i \ge 0$, the **Maximum Weighted Independent Set Problem** is:

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in V} w_i \, x_i \\ \text{subject to} \quad & x_i + x_j \le 1 \quad \text{for all } \{i, j\} \in E, \\ & x_i \in \{0, 1\} \quad \text{for all } i \in V. \end{aligned}$$

The constraint $x_i + x_j \le 1$ for each edge says: we cannot select both endpoints. At most one of i, j can be in the independent set.

Example 3.7.3 (Independent set on a small graph)**.** Consider the graph below with 5 vertices and 6 edges, where each vertex has weight 1 (unweighted

case).



The maximum independent set has size 2: for instance, $\{2, 4\}$ or $\{2, 5\}$ or $\{3, 5\}$. The edge between 1 and 4 means we cannot take both, and since vertex 1 is adjacent to vertices 2, 4, 5, selecting vertex 1 severely limits our options.

Remark 3.7.4. The Maximum Independent Set problem is NP-hard even on unweighted graphs. It is equivalent to the **Maximum Clique** problem on the complement graph and to the **Minimum Vertex Cover** problem (by complementing the solution: S is independent if and only if $V \setminus S$ is a vertex cover).

3.7.2 Graph Coloring and Timetabling

Definition 3.7.5 (Graph Coloring Problem)**.** Given a graph $G = (V, E)$ and a set of K available **colors** $\{1, 2, \ldots, K\}$, a **proper coloring** assigns a color to each vertex such that no two adjacent vertices share the same color. The **chromatic number** $\chi(G)$ is the minimum number of colors needed.

Graph coloring models conflicts: two vertices sharing an edge cannot receive the same colour (time slot, frequency, exam slot, . . .).

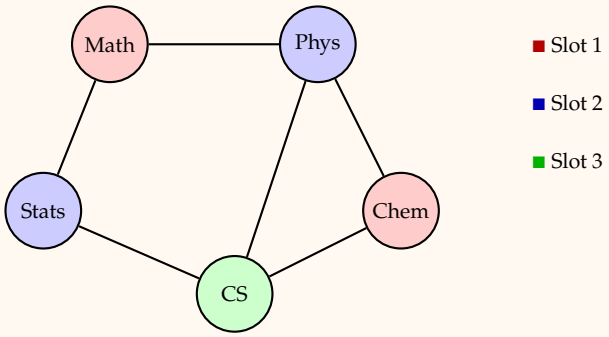
The ILP formulation uses binary variables $x_{v,k} \in \{0, 1\}$, where $x_{v,k} = 1$ means "vertex v gets color k." We also use binary variables $w_k \in \{0, 1\}$, where $w_k = 1$ means "color k is used."

Definition 3.7.6 (Graph Coloring ILP)**.**

$$\begin{aligned}
&\text{minimize} && \sum_{k=1}^{K} w_k && && \\
&\text{subject to} && \sum_{k=1}^{K} x_{v,k} = 1 && \text{for all } v \in V && \text{(each vertex gets one color),} \\
& && x_{v,k} + x_{u,k} \le 1 && \text{for all } \{u, v\} \in E,\ k = 1, \ldots, K && \text{(adjacent} \neq \text{same color),} \\
& && x_{v,k} \le w_k && \text{for all } v \in V,\ k = 1, \ldots, K && \text{(color used if assigned),} \\
& && x_{v,k} \in \{0, 1\},\ w_k \in \{0, 1\}. && &&
\end{aligned}$$

The constraint $\sum_k x_{v,k} = 1$ ensures each vertex gets exactly one color. The constraint $x_{v,k} + x_{u,k} \le 1$ prevents adjacent vertices from sharing a color. The linking constraint $x_{v,k} \le w_k$ activates the "color used" indicator.

Example 3.7.7 (Exam timetabling)**.** A university must schedule exams for 5 courses. Some courses share students, so their exams cannot be at the same time. Model as graph coloring: each course is a vertex, and an edge connects courses with shared students. Colors represent time slots.



Here $\chi(G) = 3$: we need at least 3 time slots. The coloring shown assigns Math and Chem to slot 1, Physics and Stats to slot 2, and CS to slot 3.

Formal details — Strong vs. weak formulations for graph coloring

The formulation in theorem 3.7.6 is sometimes called the **assignment formulation**. Its LP relaxation is rather weak: colors are symmetric, so the LP can spread fractional assignments across many colors.

Symmetry breaking constraints can strengthen the formulation. For instance, we can require that colors are used in order: $w_1 \geq w_2 \geq \cdots \geq w_K$. This eliminates equivalent solutions that differ only in color labeling.

Even stronger formulations are possible using *column generation* (where each column represents an independent set, i.e. a valid color class), but these go beyond our current scope.

3.7.3 SAT as ILP

Encoding SAT as ILP shows why ILP is NP-hard: it can express any problem in NP.

The **Boolean Satisfiability Problem** (SAT) asks: given a Boolean formula in **Conjunctive Normal Form** (CNF), is there an assignment of TRUE/FALSE to the variables that makes the formula true?

A CNF formula is a conjunction (AND) of **clauses**, where each clause is a disjunction (OR) of **literals** (a variable or its negation).

Example 3.7.8 (A CNF formula)**.**

$$(p_1 \vee \neg p_2 \vee p_3) \ \wedge \ (\neg p_1 \vee p_2) \ \wedge \ (p_2 \vee \neg p_3).$$

This formula has 3 variables (p_1, p_2, p_3) and 3 clauses.

To encode this as a 0–1 ILP, map each Boolean variable p_i to a binary variable $x_i \in \{0, 1\}$ (where $x_i = 1$ means TRUE). Each clause becomes a linear inequality:

- A positive literal p_i contributes x_i.
- A negated literal $\neg p_i$ contributes $(1 - x_i)$.
- The clause is satisfied iff the sum ≥ 1.

Example 3.7.9 (SAT instance as ILP)**.** The formula from theorem 3.7.8 becomes:

$$\begin{aligned} x_1 + (1 - x_2) + x_3 \geq 1 \quad &\Longleftrightarrow \quad x_1 - x_2 + x_3 \geq 0, \\ (1 - x_1) + x_2 \geq 1 \quad &\Longleftrightarrow \quad -x_1 + x_2 \geq 0, \\ x_2 + (1 - x_3) \geq 1 \quad &\Longleftrightarrow \quad x_2 - x_3 \geq 0, \\ x_i \in \{0, 1\}, \quad i = 1, 2, 3. & \end{aligned}$$

Any feasible solution is a satisfying assignment. For instance, $(x_1, x_2, x_3) = (0, 1, 1)$: the first clause gives $0 - 1 + 1 = 0 \geq 0$ (satisfied), the second gives $-0 + 1 = 1 \geq 0$ (satisfied), the third gives $1 - 1 = 0 \geq 0$ (satisfied).

To find a satisfying assignment, we can solve the *feasibility* problem (any objective, e.g. maximize 0). To find the assignment satisfying the most clauses (**MAX-SAT**), we introduce slack variables $z_k \in \{0, 1\}$ for each clause k and maximize $\sum_k z_k$, where the clause constraint becomes "sum of literals $\geq z_k$" instead of "≥ 1."

3.8 LP Relaxation and the Integrality Gap

LP relaxation is the single most important tool in integer programming. It provides bounds, guides algorithms, and measures formulation quality.

Throughout this chapter, we have repeatedly mentioned the idea of "dropping integrality constraints." Let's now make this precise and explore its consequences.

3.8.1 The LP relaxation

Definition 3.8.1 (LP Relaxation)**.** Given an ILP

$$z^*_{\text{ILP}} = \max\{c^\top x : Ax \leq b,\ x \geq 0,\ x \in \mathbb{Z}^n\},$$

its **LP relaxation** is the LP obtained by dropping the integrality constraint:

$$z^*_{\text{LP}} = \max\{c^\top x : Ax \leq b,\ x \geq 0\}.$$

The LP relaxation is *easier* to solve (it's just an LP!) and its feasible region is *larger* (every integer feasible point is also LP feasible, but not vice versa). This immediately gives us a bound.

Theorem 3.8.2 (LP relaxation bound)**.** *Let z^*_{ILP} be the optimal value of an ILP and z^*_{LP} the optimal value of its LP relaxation. Then:*

- ***Maximization**: $z^*_{\text{ILP}} \leq z^*_{\text{LP}}$ (the LP relaxation gives an **upper bound**).*
- ***Minimization**: $z^*_{\text{ILP}} \geq z^*_{\text{LP}}$ (the LP relaxation gives a **lower bound**).*

Proof. Every feasible solution of the ILP is also feasible for the LP relaxation (since we only removed constraints). Hence, the ILP optimises over a *subset* of the LP feasible region. Optimising over a larger set can only improve (or maintain) the objective value. □

Example 3.8.3 (Bounding the running knapsack)**.** From theorem 3.3.7, the LP relaxation value is $z^*_{\text{LP}} = 174/7 \approx 24.86$. From theorem 1.10.1, we have the feasible integer solution $\{1, 2, 4, 5\}$ with value 23. Therefore:

$$23 \leq z^*_{\text{ILP}} \leq 24.86.$$

Since the ILP objective has integer coefficients and binary variables, z^*_{ILP} must be an integer (it is a sum of integers). Therefore $z^*_{\text{ILP}} \leq 24$.

Can we achieve 24? Items $\{1, 2, 3\}$ have weight $5 + 3 + 7 = 15 \leq 15$ (feasible) and value $8 + 5 + 11 = 24$—yes, value 24 is attained.

Therefore $z^*_{\text{ILP}} = 24$, and the selection $\{1, 2, 3\}$ *is* optimal. The LP relaxation, combined with a good feasible solution, proved optimality without enumerating all $2^6 = 64$ subsets.

3.8.2 The integrality gap

Definition 3.8.4 (Integrality Gap)**.** The **integrality gap** of an ILP instance is the ratio between the LP relaxation value and the ILP optimal value:

$$\text{Integrality gap} = \frac{z^*_{\text{LP}}}{z^*_{\text{ILP}}} \quad \text{(for maximization, gap} \geq 1\text{)}.$$

For a minimization problem, the ratio is inverted: $z^*_{\text{ILP}}/z^*_{\text{LP}}$.

A small integrality gap means the LP relaxation is a good approximation of the ILP—and solvers can find the optimum quickly. A large gap means the LP relaxation is weak, and the solver must work much harder (more branching, more cuts).

Example 3.8.5 (Integrality gap of the running knapsack)**.** For our knapsack instance:

$$\text{gap} = \frac{z^*_{\text{LP}}}{z^*_{\text{ILP}}} = \frac{174/7}{24} = \frac{174}{168} = \frac{29}{28} \approx 1.036.$$

The gap is about 3.6%—quite small. The LP relaxation is a good approximation in this case.

3.8.3 Strong and weak formulations

The same combinatorial problem can be modeled by *different* ILPs. These formulations have the same set of integer feasible points but different LP relaxations—and hence different integrality gaps.

Definition 3.8.6 (Stronger formulation)**.** Let $P_1 = \{x \in \mathbb{R}^n : A_1 x \leq b_1\}$ and $P_2 = \{x \in \mathbb{R}^n : A_2 x \leq b_2\}$ be the LP relaxation polyhedra of two ILP formulations for the same problem. We say formulation 1 is **stronger** (or **tighter**) than formulation 2 if $P_1 \subseteq P_2$.

A stronger formulation has a smaller LP relaxation polyhedron, which means its LP relaxation bound is at least as good. This leads to faster solving.

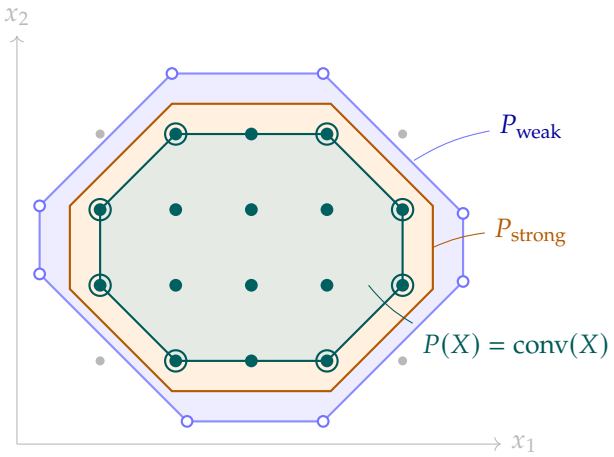


Figure 3.6: Three nested relaxations of the same integer set X (teal dots; gray dots are infeasible integer points, outside all three). The **weak formulation** P_{weak} (blue) is loose: its vertices (hollow circles) are fractional, far from any integer solution. The **strong formulation** P_{strong} (orange) is tighter. The **convex hull** $P(X) = \text{conv}(X)$ (teal) is the tightest possible LP relaxation—every vertex of $P(X)$ (ringed) is an integer point.

3.8.4 The convex hull and ideal formulations

Definition 3.8.7 (Convex hull of integer points)**.** Given the set of integer feasible points $X = \{x \in \mathbb{Z}^n : Ax \leq b,\ x \geq 0\}$, the **convex hull**

$$P(X) = \text{conv}(X) = \left\{ \sum_i \lambda_i x^{(i)} : x^{(i)} \in X,\ \lambda_i \geq 0,\ \sum_i \lambda_i = 1 \right\}$$

is the tightest convex set containing all integer feasible points.

The convex hull $P(X)$ is always a polyhedron (by the Minkowski-Weyl theorem from chapter 2). It is the "ideal" LP relaxation: if we could describe $P(X)$ with linear inequalities, then solving the LP relaxation would give an integer optimal solution directly.

Theorem 3.8.8 (Ideal formulation)**.** *If the LP relaxation polyhedron P equals the convex hull $P(X) = \text{conv}(X)$, then every vertex of P is an integer point, and the LP relaxation solves the ILP exactly.*

Proof. If $P = \text{conv}(X)$, then every vertex of P is a convex combination of points in X that cannot be further decomposed—hence it must be a point in X itself (an integer point). Since the LP optimum is attained at a vertex (by the fundamental theorem of LP from chapter 2), the LP optimum is integer. □

This ideal situation—$P = P(X)$—is rare but extremely valuable. In chapter 7 we will see that when the constraint matrix A is **totally unimodular** (TU), the LP relaxation polyhedron automatically equals the convex hull of integer points, and the LP relaxation solves the ILP. The assignment problem (theorem 3.2.3) is one such case.

Finding $P(X)$ explicitly may require exponentially many inequalities. But for special problem structures, $P = P(X)$ comes "for free." This is the magic of total unimodularity (chapter 7).

3.8.5 Rounding is not enough

A natural idea when the LP relaxation gives a fractional solution is to simply **round** each variable to the nearest integer. This is tempting but dangerous.

Example 3.8.9 (Why rounding fails)**.** Consider the ILP:

$$\begin{aligned} \text{maximize} \quad & x_1 + x_2 \\ \text{subject to} \quad & 2x_1 + 2x_2 \leq 3, \\ & x_1, x_2 \in \{0, 1\}. \end{aligned}$$

The LP relaxation optimum is $(x_1, x_2) = (0.75, 0.75)$ with value 1.5. Rounding both up gives $(1, 1)$, which has $2(1)+2(1) = 4 > 3$—**infeasible**. Rounding both down gives $(0, 0)$ with value 0, which misses the ILP optimum $(1, 0)$ or $(0, 1)$ with value 1.

Rounding can violate feasibility, miss the optimum, or both. Solving ILPs requires more sophisticated methods: **branch and bound**, **cutting planes**, and their combination **branch and cut**. These are the subject of chapter 6.

3.9 A Modelling Checklist

Before moving on, let's collect the modelling principles from this chapter into a practical checklist.

(i) **Identify the decisions.** What are the yes/no or how-many choices? Each one becomes a variable.

(ii) **Choose the right variable type.** Binary for yes/no, general integer for quantities, continuous for amounts.

(iii) **One binary per option** (the golden rule, section 3.2.2). Never encode categorical choices as a single integer.

(iv) **Write the objective.** Is it linear in the variables? If not, can fixed costs (section 3.5.1) or piecewise linearisation (section 3.5.2) help?

(v) **Write the constraints.** Resource limits, covering requirements, logical conditions, linking constraints.

(vi) **Use Big-M carefully** (section 3.6.3). Always choose the tightest possible M.

(vii) **Check the LP relaxation.** Is it strong? Could additional (valid) inequalities tighten it?

(viii) **Exploit structure.** Does the problem have a known polyhedral description? Is the constraint matrix TU?

3.10 Chapter Summary

- **Integer Linear Programming** extends LP by requiring some or all variables to be integers. Pure ILP: all integer. MILP: some integer, some continuous. Binary program: all in $\{0, 1\}$.

- ILP is **NP-hard**, in sharp contrast to LP which is polynomial.
- **Binary variables** model yes/no decisions. The golden rule: use one binary variable per categorical option, never a single integer with arbitrary labels.
- The **knapsack problem** (theorem 3.3.1) is the prototypical binary program. Its LP relaxation is solved greedily by value-to-weight ratio (theorem 3.3.6). For our running instance (theorem 1.10.1), the LP relaxation value ≈ 24.86 and the ILP optimum is 24.
- The **set covering problem** (theorem 3.4.1) selects minimum-cost subsets to cover a universe.
- **Fixed costs** and **piecewise linear costs** can be modeled with auxiliary binary variables and Big-M constraints.
- **Logical constraints** (OR, AND, XOR, implication) translate directly into linear inequalities over binary variables. The **Big-M method** (theorem 3.6.3) enables conditional constraints.
- Classic ILP models include the **weighted independent set**, **graph coloring**, and **SAT**.
- The **LP relaxation** (theorem 3.8.1) provides a *bound* on the ILP optimum: upper bound for max, lower bound for min (theorem 3.8.2).
- The **integrality gap** (theorem 3.8.4) measures how far the LP relaxation is from the ILP. A **stronger formulation** has a tighter LP relaxation and a smaller gap.
- The **convex hull** $\text{conv}(X)$ is the ideal formulation. If $P = \text{conv}(X)$, the LP relaxation solves the ILP exactly (theorem 3.8.8). This happens when A is totally unimodular (chapter 7).
- **Rounding** the LP relaxation solution is unreliable. Proper ILP algorithms (branch and bound, cutting planes) are needed and will be covered in chapter 6.

Looking ahead. In chapter 4 we study the Simplex method for solving LPs—the engine inside every ILP solver. In chapter 5 we develop LP duality, which provides alternative proofs of optimality and connects to pricing in ILP algorithms. Then, in chapter 6, we tackle the ILP models from this chapter with branch and bound, cutting planes, and branch and cut.

Summary & Key Takeaways

- **Integer Linear Programming (ILP):** An LP where some or all variables are restricted to integer values, introducing non-convexity and rendering the problem NP-hard.
- **Logical Formulations:** Binary variables $y \in \{0, 1\}$ are used to model discrete choices:
  - *Big-M Method:* $f(x) \leq My$ models activation of constraint $f(x) \leq 0$ when $y = 1$.
  - *Either-Or (Aut-Aut):* $a^\top x \leq b_1 + M(1 - y)$ and $c^\top x \leq d_1 + My$ ensure at least one is satisfied.

– *Implications:* $x_1 \leq y$ forces $x_1 = 0$ if action $y = 0$ is chosen.

Exercises

Exercise 1 (ILP vs. LP feasible region). Consider the LP relaxation of the following ILP:

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 2x_2 \\ \text{subject to} \quad & 2x_1 + x_2 \leq 7 \\ & x_1 + 2x_2 \leq 7 \\ & x_1, x_2 \geq 0,\ x_1, x_2 \in \mathbb{Z}. \end{aligned}$$

1. Solve the LP relaxation graphically and record the optimal LP value z^*_{LP}.
2. List all integer feasible points in the feasible region.
3. Identify the ILP optimum z^*_{ILP}.
4. Compute the integrality gap.

Exercise 2 (Binary knapsack LP relaxation). A knapsack has capacity $W = 10$. The items are:

Item	1	2	3
w_i	6	5	4
v_i	9	7	5

1. Write the binary knapsack ILP.
2. Formulate and solve the LP relaxation (each $x_i \in [0, 1]$). *Hint:* Sort items by value density v_i/w_i and fill greedily.
3. State z^*_{LP} and z^*_{ILP}, then compute the integrality gap.

Exercise 3 (Integrality gap interpretation). For a *minimisation* ILP instance, the LP relaxation gives $z^*_{\text{LP}} = 12$ and the ILP optimum is $z^*_{\text{ILP}} = 15$.

1. Compute the integrality gap.
2. A colleague claims: "The LP relaxation always underestimates the true optimum for minimisation." Is this correct? Justify.
3. What does it mean for a formulation to be *strong* in terms of the integrality gap?

Exercise 4 (Bounding lemma for maximisation ILP). Let $X = \{x \in \mathbb{Z}^n_{\geq 0} : Ax \leq b\}$ be the set of integer-feasible solutions to an ILP, and let $P = \{x \in \mathbb{R}^n_{\geq 0} : Ax \leq b\}$ be the LP relaxation feasible region.

1. Prove that $X \subseteq P$.
2. Using part (a), prove that $z^*_{\text{LP}} \geq z^*_{\text{ILP}}$ for a maximisation problem.

Exercise 5 (True or false: LP rounding). State whether each claim is **true** or **false** and give a brief justification or counterexample.

1. Rounding down every component of the LP relaxation optimal solution always yields a feasible ILP solution.
2. Rounding down every component of the LP relaxation optimal solution always yields the ILP optimal solution.
3. For a pure binary ILP, every extreme point of the LP relaxation is integer-valued.
4. If the LP relaxation of an ILP is infeasible, then so is the ILP.

Exercise 6 (Convex hull and ideal formulation). Let $X = \{(x_1, x_2) \in \mathbb{Z}^2_{\geq 0} : x_1 + x_2 \leq 3,\ x_1 \leq 2,\ x_2 \leq 2\}$.

1. Enumerate all points in X.

2. Describe conv(X) by listing its extreme points and giving a minimal inequality description.
3. Explain why the LP relaxation over conv(X) solves the ILP exactly.

Exercise 7 (Strong vs. weak formulation). Consider the single-item lot-sizing variable $y \in \{0, 1\}$ indicating whether a machine is switched on, and $x \geq 0$ the production quantity, with $x \leq 10$.

Formulation A: $x \leq 10y$.
Formulation B: $x \leq 10, \quad x \leq 10y$.

1. Show that both formulations have the same integer-feasible set.
2. Solve the LP relaxation of each formulation (for $y \in [0, 1]$, $x \geq 0$) with objective maximize $x - 5y$.
3. Which formulation is stronger? Why does the stronger formulation lead to a tighter LP bound?

Exercise 8 (Comparing two ILP formulations). The following two formulations both model the constraint "at most one of y_1, y_2, y_3 equals 1" (binary variables):

Formulation I: $y_1 + y_2 + y_3 \leq 1$.
Formulation II: $y_1 + y_2 \leq 1, \quad y_1 + y_3 \leq 1, \quad y_2 + y_3 \leq 1$.

1. Show that both formulations are equivalent over $\{0, 1\}^3$.
2. Determine which formulation has a tighter LP relaxation (i.e., a smaller feasible region when $y_i \in [0, 1]$). *Hint:* Check whether $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ is feasible in each.
3. Conclude which formulation is stronger.

Exercise 9 (Binary selection constraints). A project portfolio contains six projects. The binary variable $y_j = 1$ if project j is selected. Formulate each of the following business rules as one or more linear constraints:

1. At most three projects are selected.
2. At least two projects are selected.
3. If project 3 is selected, then project 1 must also be selected.
4. Projects 4 and 5 cannot both be selected.
5. Exactly one of projects 2 and 6 is selected.

Exercise 10 (Logical constraints with binary variables). Let $y_1, y_2, y_3 \in \{0, 1\}$. For each logical condition below, write a *linear* constraint (or set of constraints) that is satisfied if and only if the condition holds.

1. $z = y_1$ AND y_2 (binary auxiliary z).
2. $z = y_1$ OR y_2 (binary auxiliary z).
3. $z =$ NOT y_1 (binary auxiliary z).
4. $(y_1 = 1) \Rightarrow (y_2 = 1)$.
5. $(y_1 = 1) \Rightarrow (y_2 = 0)$.

Exercise 11 (If–then constraint via Big-M). Let $x_1, x_2 \geq 0$ be general (non-binary) variables and $y \in \{0, 1\}$ a binary indicator. The constraint is:

$$\text{If } x_1 \geq 1, \text{ then } x_2 \leq 0.$$

Assume $x_1 \leq M_1$ and $x_2 \leq M_2$ for known big-M values.

1. Introduce a binary variable y and write a Big-M formulation that enforces this implication.
2. Briefly explain why choosing M as tight as possible (i.e., the smallest valid upper bound) yields a stronger formulation.

Exercise 12 (Either–or constraints). A production plan must satisfy *at least one* of the two constraints:

$$(A)\ 3x_1 + 2x_2 \le 12, \qquad (B)\ x_1 + 4x_2 \le 10.$$

Using a binary variable $y \in \{0, 1\}$ and a suitable Big-M, write a single ILP formulation that enforces "constraint A holds **or** constraint B holds (or both)".

Exercise 13 (Big-M conditional constraint). A warehouse is opened at site i only if binary variable $z_i = 1$. When it is open, its throughput q_i satisfies $q_i \le C_i$ (capacity). When it is closed, $q_i = 0$.

1. Write the Big-M constraint linking q_i and z_i.
2. What is the tightest valid choice for M?
3. Is the formulation with $M = C_i$ stronger or weaker than one with $M = 10\,C_i$? Explain.

Exercise 14 (At-most-k binary constraint). You have $n = 8$ binary variables $y_1, \ldots, y_8$ representing whether a sensor is activated.

1. Write a single linear constraint that allows at most $k = 3$ sensors to be active simultaneously.
2. Now add the additional rule: "if sensor 2 is active, then at most $k - 1 = 2$ *other* sensors (excluding sensor 2) may be active." Write the extra constraint.
3. Are your constraints linear? Justify.

Exercise 15 (Linearising a product of binary variables). Let $y_1, y_2 \in \{0, 1\}$ and consider the nonlinear term $w = y_1 \cdot y_2$ that appears in the objective or a constraint.

1. Introduce an auxiliary binary variable w and write three linear constraints that force $w = y_1 y_2$.
2. Verify your formulation by checking all four combinations $(y_1, y_2) \in \{0, 1\}^2$.
3. Generalise: how many auxiliary variables and constraints are needed to linearise the product of k binary variables?

Exercise 16 (Fixed startup cost). A machine incurs a fixed startup cost $F = 200$ whenever it is used. Variable production cost is $c = 5$ per unit, and the machine has capacity $U = 100$ units per period. Let $x \ge 0$ be production quantity.

1. Introduce a binary indicator y and write the ILP formulation of the total cost, including the fixed cost and the constraint that $x = 0$ when the machine is off.
2. Sketch the feasible region in the (x, y) space.
3. Is it ever optimal to set $y = 1$ and $x = 0$? Explain.

Exercise 17 (Piecewise linear objective with binary variables). A cost function is piecewise linear in $x \in [0, 60]$:

$$c(x) = \begin{cases} 2x & 0 \le x \le 20, \\ 40 + 3(x - 20) & 20 < x \le 40, \\ 100 + 5(x - 40) & 40 < x \le 60. \end{cases}$$

1. Introduce binary variables d_1, d_2, d_3 (one per segment) and write an ILP formulation that minimises $c(x)$ subject to $x \ge 45$, ensuring x lies in exactly one segment.
2. Write the linking constraints between the segment variables and x.

Exercise 18 (Uncapacitated facility location). There are m potential facility sites and n customers. Opening facility i costs f_i. Serving customer j from facility i has cost c_{ij}. Each customer must be served by exactly one open facility.

1. Define binary variables y_i (open facility i) and x_{ij} (serve customer j from facility i).
2. Write the complete ILP: objective, assignment constraints, linking constraints, and variable domains.
3. Identify which constraints enforce that a customer can only be assigned to an open facility.

Exercise 19 (Assignment problem ILP). There are n workers and n jobs. Worker i assigned to job j produces profit p_{ij}. Each worker does exactly one job and each job is done by exactly one worker.

1. Define binary decision variables and write the ILP formulation.
2. How many variables and constraints does your model have (as a function of n)?
3. For $n = 3$ and the profit matrix $P = \left(\begin{smallmatrix} 3 & 1 & 4 \\ 2 & 5 & 2 \\ 4 & 3 & 6 \end{smallmatrix}\right)$, write out the full model explicitly.

Exercise 20 (TSP with subtour elimination). The Travelling Salesman Problem (TSP) on n cities minimises the total travel cost of a Hamiltonian cycle. Let $x_{ij} \in \{0, 1\}$ equal 1 iff arc (i, j) is used.

1. Write the degree constraints (each city entered and left exactly once).
2. For $n = 4$, write all *subtour elimination constraints* (SEC) for proper non-empty subsets $S \subsetneq \{1, 2, 3, 4\}$, $2 \le |S| \le n - 1$.
3. How many SECs are there in total for a general n-city TSP?
4. Why are subtour elimination constraints necessary? Give an example of a solution satisfying the degree constraints but containing a subtour.

Exercise 21 (Graph coloring ILP). Given a graph $G = (V, E)$ with $|V| = 5$ vertices and edges $\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}, \{4, 5\}$, and at most $k = 3$ colors.

1. Define binary variables x_{vc} (= 1 iff vertex v receives color c) and w_c (= 1 iff color c is used).
2. Write the complete ILP (each vertex gets exactly one color, adjacent vertices get different colors, objective minimises number of colors used).
3. Check whether the chromatic number of this graph is at most 3 by inspection.

Exercise 22 (Maximum weighted independent set). A graph has vertices $V = \{1, 2, 3, 4, 5\}$ with weights $w = (3, 5, 2, 4, 1)$ and edges $\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 5\}, \{1, 5\}$ (a 5-cycle).

1. Write the ILP for the maximum weight independent set.
2. Solve the LP relaxation. *Hint:* By symmetry, try $x_v = \frac{1}{2}$ for all v.
3. Find the ILP optimum by inspection and compute the integrality gap.

Exercise 23 (Bin packing ILP). Items $1, \ldots, n$ have sizes $s_1, \ldots, s_n$. Each bin has capacity C. Minimise the number of bins used.

1. Define binary variables y_k (bin k is used) and x_{ik} (item i is placed in bin k).
2. Write the complete ILP: objective, assignment constraints, capacity constraints, and linking constraints.
3. For $n = 4$ items with sizes $(4, 3, 3, 2)$ and $C = 6$, what is a feasible solution using 3 bins? Using 2 bins?

Exercise 24 (Set covering model). A city must place emergency response

units so that every neighbourhood is covered within 5 minutes by at least one unit. There are 6 candidate sites and 8 neighbourhoods. The coverage matrix is:

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

where $A_{ji} = 1$ iff site i covers neighbourhood j.

1. Write the set covering ILP (minimise number of sites opened).
2. Write the LP relaxation.
3. Verify that $y = (1, 0, 1, 1, 0, 0)$ is feasible.

Exercise 25 (Set packing ILP). Given a universe $U = \{1, 2, 3, 4, 5\}$ and collection of subsets $S_1 = \{1, 2\}$, $S_2 = \{2, 3, 4\}$, $S_3 = \{1, 3\}$, $S_4 = \{4, 5\}$, $S_5 = \{1, 5\}$, with weights $w = (3, 5, 2, 4, 3)$:

1. Write the set packing ILP: maximise total weight of selected subsets subject to each element of U being covered by at most one selected subset.
2. Enumerate all feasible packings and find the optimal one.
3. Compare set covering and set packing: in which does the constraint direction flip?

Exercise 26 (Multi-index scheduling model). A hospital must assign $n = 3$ nurses to $m = 4$ shifts over a two-day horizon. Let $x_{ijk} \in \{0, 1\}$ be 1 iff nurse i works shift j on day k.

1. Write a constraint ensuring each shift on each day is covered by at least one nurse.
2. Write a constraint ensuring each nurse works at most one shift per day.
3. Write a constraint ensuring no nurse works the last shift of day 1 (shift 4, $k = 1$) *and* the first shift of day 2 (shift 1, $k = 2$). Use a linear constraint (not Big-M).

Exercise 27 (Bounded integer knapsack). Extend the binary knapsack to an *integer* knapsack: item i can be packed up to u_i times. Capacity $W = 15$, and:

Item	1	2	3	4
w_i	4	3	5	2
v_i	7	5	9	3
u_i	2	3	1	4

1. Write the ILP (bounded variable version).
2. Show how the bounded integer knapsack reduces to a binary knapsack by binary expansion of each variable x_i.
3. Compute the LP relaxation bound.

Exercise 28 (SAT encoded as ILP). A propositional formula is in conjunctive normal form (CNF) with binary variables $y_1, y_2, y_3 \in \{0, 1\}$ (1 = true) and clauses:

$$(y_1 \vee \neg y_2), \quad (\neg y_1 \vee y_2 \vee y_3), \quad (\neg y_2 \vee \neg y_3).$$

1. Encode each clause as a linear inequality over $y_1, y_2, y_3 \in \{0, 1\}$. *Hint:* A literal $\neg y_i$ contributes $(1 - y_i)$.

2. Write the ILP feasibility problem (the objective can be maximize 0).
3. Determine by inspection whether the formula is satisfiable.

Exercise 29 (Modelling checklist application). A small factory can produce product A or product B (or both) each week. Production of A requires machine M1 and M2; production of B requires M2 and M3. Each machine has a limited weekly capacity. Starting production of either product incurs a fixed cost.

1. Apply the modelling checklist: identify the decision variables (continuous, binary), the objective, and the constraint types.
2. Write the complete MILP formulation.
3. Identify which constraints are coupling constraints linking binary and continuous variables.

Exercise 30 (Maximum satisfiability as ILP). Given the same three clauses as Exercise 28, now maximise the number of satisfied clauses (MAX-SAT).

1. Introduce binary slack variables $s_1, s_2, s_3 \in \{0, 1\}$, where $s_k = 1$ iff clause k is satisfied.
2. Rewrite each clause constraint as: "(clause k LHS) $+s_k \geq 1$", then transform to allow s_k to indicate satisfaction.
3. Write the complete MAX-SAT ILP.

Exercise 31 (Complement and mutual exclusion). Let $y_1, y_2 \in \{0, 1\}$.

1. Write a linear constraint enforcing $y_2 = 1 - y_1$ (i.e., y_2 is the complement of y_1).
2. Three events E_1, E_2, E_3 are mutually exclusive (at most one occurs). Write a single constraint on binary indicators z_1, z_2, z_3.
3. Exactly one of the three events must occur. Write the corresponding constraint.

Exercise 32 (Linearising a product of binary and continuous). Let $y \in \{0, 1\}$ and $x \in [0, U]$. The term $w = y \cdot x$ appears in a constraint.

1. Introduce an auxiliary variable $w \geq 0$ and write the four linear constraints that enforce $w = yx$ exactly.
2. Verify your formulation when $y = 0$ and when $y = 1$.
3. How does the choice of U affect the tightness of the formulation?

Exercise 33 (Lower bound on chromatic number via cliques). Recall that a *clique* in G is a set of pairwise adjacent vertices. Any proper coloring must assign a distinct color to each vertex in a clique.

1. Prove that the chromatic number $\chi(G) \geq \omega(G)$, where $\omega(G)$ is the clique number (size of the largest clique).
2. For the graph with vertex set $\{1, 2, 3, 4, 5\}$ and edges $\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}, \{4, 5\}$, identify the largest clique and give a lower bound on $\chi(G)$.
3. Exhibit a valid 3-coloring, establishing $\chi(G) \leq 3$.

Exercise 34 (Stepwise fixed cost model). A production line has two modes. Mode 1 (low speed) handles $0 \leq x \leq 50$ units at variable cost $c_1 = 3$ per unit and fixed cost $F_1 = 100$. Mode 2 (high speed) handles $0 \leq x \leq 120$ units at $c_2 = 2$ per unit and fixed cost $F_2 = 250$. Exactly one mode is used per period.

1. Define binary variable $d \in \{0, 1\}$ ($d = 0$: mode 1, $d = 1$: mode 2) and write the MILP.
2. What quantity x^* makes the two modes equally costly?
3. For a demand of $x = 80$ units, which mode minimises cost?

Exercise 35 (Multi-dimensional knapsack). Five items may be selected. Each item j has value v_j, weight w_j, and volume u_j. The knapsack has weight

capacity $W = 10$ and volume capacity $V = 8$:

j	1	2	3	4	5
v_j	5	4	3	7	6
w_j	3	4	2	5	3
u_j	2	3	4	3	5

1. Write the multi-dimensional binary knapsack ILP.
2. Solve the LP relaxation of the weight-only knapsack (ignore volume) and record $z^*_{\text{LP,W}}$.
3. Adding the volume constraint, is $z^*_{\text{LP,WV}}$ larger or smaller than $z^*_{\text{LP,W}}$? Justify without solving.

Exercise 36 (Integrality gap of two formulations). Two formulations F_1 and F_2 both model the same maximisation ILP with optimal value $z^*_{\text{ILP}} = 10$. The LP relaxations give $z^*_{\text{LP}}(F_1) = 14$ and $z^*_{\text{LP}}(F_2) = 11$.

1. Compute the integrality gap for each formulation.
2. Which formulation is stronger?
3. In a branch-and-bound algorithm, which formulation would you prefer and why?

Exercise 37 (LP relaxation of a small ILP). Solve the LP relaxation of:

$$\begin{aligned} \text{maximize} \quad & 5x_1 + 8x_2 \\ \text{subject to} \quad & x_1 + x_2 \le 6 \\ & 5x_1 + 9x_2 \le 45 \\ & x_1, x_2 \ge 0,\ x_1, x_2 \in \mathbb{Z}. \end{aligned}$$

1. Solve the LP relaxation (drop the integrality constraint) graphically or algebraically.
2. Round the LP solution and check feasibility.
3. Find the ILP optimum and compute the integrality gap.

Exercise 38 (Capital budgeting with logical dependencies). A firm considers five investment projects ($j = 1, \dots, 5$) with expected NPVs (in \$M) of $(12, 18, 9, 14, 20)$ and capital requirements $(4, 6, 3, 5, 8)$. Total capital available is 15\$M.

1. Write the binary knapsack ILP for this capital budgeting problem.
2. Add the constraint: project 5 can be selected only if both projects 2 and 4 are selected.
3. Add the constraint: at most one of projects 1 and 3 is selected.
4. With all constraints, find the optimal selection by enumeration (there are at most $2^5 = 32$ combinations to check; eliminate infeasible ones quickly using the budget constraint).

Exercise 39 (Nurse rostering (small instance)). Four nurses (indexed $1, \dots, 4$) must cover three daily shifts (morning, afternoon, night). Each shift requires exactly one nurse. Each nurse works at most one shift per day. Nurse 4 is not available for the night shift. Nurse preferences give a cost matrix:

$$C = \begin{pmatrix} 2 & 3 & 5 \\ 4 & 1 & 3 \\ 3 & 4 & 2 \\ 1 & 2 & \infty \end{pmatrix}$$

(rows = nurses, columns = shifts; ∞ means unavailable).

1. Write the ILP (excluding ∞ entries).
2. Identify the constraint that handles nurse 4's unavailability.
3. Solve by inspection.

Exercise 40 (Capacitated vehicle routing: formulation setup). A depot serves $n = 4$ customers with a single vehicle of capacity Q. Customer i has demand d_i. Binary $x_{ij} = 1$ if the vehicle travels directly from node i to node j (depot is node 0).

1. Write the flow-conservation constraints for each customer node.
2. Write the capacity constraint using auxiliary variables $u_i \geq 0$ (accumulated load after visiting customer i) to eliminate subtours and enforce capacity simultaneously (the *Miller–Tucker–Zemlin* style for the capacitated case).
3. What is the role of the u_i variables beyond subtour elimination?

Exercise 41 (Concave piecewise linear minimisation). The total production cost is a concave piecewise linear function of output $x \in [0, 90]$ with breakpoints at $x = 30$ and $x = 60$:

$$c(x) = \begin{cases} 4x & 0 \leq x \leq 30, \\ 120 + 3(x - 30) & 30 < x \leq 60, \\ 210 + 2(x - 60) & 60 < x \leq 90. \end{cases}$$

1. Introduce SOS2 (Special Ordered Set type 2) variables or binary variables d_1, d_2, d_3 to linearise the objective.
2. Write the constraint that exactly one segment is active.
3. Write the expression for x and $c(x)$ in terms of the segment variables.

Exercise 42 (Strengthening the independent set formulation). The standard independent set ILP uses constraints $x_u + x_v \leq 1$ for each edge $\{u, v\} \in E$.

1. For a triangle (3-clique) $\{u, v, w\}$, what three edge constraints are generated?
2. Write a single *clique inequality* that dominates those three constraints (is tighter over $[0, 1]^n$).
3. Show that the clique inequality cuts off the fractional point $x_u = x_v = x_w = \frac{1}{2}$ that satisfies all three edge constraints.

Exercise 43 (Semi-continuous variable modelling). A heating system must operate either at a flow rate $x \in [L, U]$ or be completely off ($x = 0$), where $0 < L < U$. This is a *semi-continuous* variable.

1. Let $y \in \{0, 1\}$ indicate whether the system is on. Write the two Big-M inequalities enforcing $x \in \{0\} \cup [L, U]$.
2. Verify that when $y = 0$, the constraints force $x = 0$, and when $y = 1$, they allow $x \in [L, U]$.
3. Could you enforce $x \in \{0\} \cup [L, U]$ without a binary variable? Why or why not?

Exercise 44 (TSP vs. Hamiltonian path ILP). The Hamiltonian path problem asks for a simple path visiting every vertex exactly once (not necessarily returning to the start).

1. Starting from the TSP formulation (degree constraints + SECs), describe exactly what changes are needed to model the Hamiltonian path problem instead.
2. For $n = 4$ nodes, how many SECs does the TSP require? How many does the Hamiltonian path require?

3. A directed TSP on $n = 4$ nodes has a cost matrix:

$$C = \begin{pmatrix} \infty & 3 & 1 & 5 \\ 3 & \infty & 4 & 2 \\ 1 & 4 & \infty & 6 \\ 5 & 2 & 6 & \infty \end{pmatrix}.$$

Write out the degree constraints explicitly.

Exercise 45 (True or false: ILP theory). State whether each claim is **true** or **false** and justify briefly.

1. Every LP relaxation of an ILP has an optimal solution.
2. If the LP relaxation is unbounded, then the ILP is also unbounded.
3. A stronger formulation always has fewer constraints than a weaker one.
4. The convex hull formulation $\text{conv}(X)$ always has an exponential number of inequalities.
5. Adding valid inequalities to an ILP formulation cannot change the set of integer-feasible solutions.

Exercise 46 (LP relaxation bound for graph coloring). Consider the graph coloring ILP on a graph G with n vertices and k colors.

1. Solve the LP relaxation of the standard formulation on a triangle (K_3) with $k = 2$ colors. Show that the LP allows each vertex to take value $\frac{1}{2}$ in each of the two colors, achieving LP objective value 0 (minimising the number of colors used), whereas the ILP optimum requires 3 colors.
2. What does this imply about the strength of the standard graph coloring formulation?

Exercise 47 (Radio tower coverage). A telecommunications company wants to place radio towers to cover $n = 7$ zones. There are $m = 5$ candidate tower locations. The binary coverage matrix is:

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

where $A_{ji} = 1$ iff tower i covers zone j. Tower costs are $(3, 2, 4, 2, 3)$.

1. Write the weighted set covering ILP (minimise total cost).
2. Write the LP relaxation.
3. Is $y = (1, 1, 0, 1, 0)$ feasible? Check all coverage constraints.

Exercise 48 (k-median facility location). The k-median problem opens exactly k facilities from a set of m candidates to minimise total client-to-facility assignment cost. There are $m = 4$ candidates and $n = 5$ clients, with assignment costs c_{ij}.

1. Define variables $y_i \in \{0, 1\}$ (open facility i) and $x_{ij} \in \{0, 1\}$ (assign client j to facility i).
2. Write the complete ILP: objective, assignment constraints, linking constraints, and the constraint that exactly k facilities are opened.
3. How does the k-median formulation differ from the uncapacitated facility location formulation of Exercise 18?

Exercise 49 (Stronger formulation implies smaller integrality gap). Let $P_1 \supseteq P_2$ be two LP relaxation polyhedra for the same maximisation ILP (so P_2 is a tighter relaxation). Let $z_i = \max\{c^\top x : x \in P_i\}$ for $i = 1, 2$.

1. Prove that $z_1 \geq z_2 \geq z^*_{\text{ILP}}$.
2. Conclude that formulation 2 has a smaller or equal integrality gap.
3. Give an example where $P_1 \supsetneq P_2$ but $z_1 = z_2$ (i.e., both LP relaxations give the same bound despite different feasible regions).

Exercise 50 (Bin packing LP relaxation lower bound). Four items have sizes $(0.4, 0.5, 0.6, 0.7)$ and bin capacity $C = 1$.

1. Write the bin packing ILP for this instance (using at most 4 bins).
2. Solve the LP relaxation (each $x_{ik} \in [0, 1]$, $y_k \in [0, 1]$).
3. Show that the LP lower bound (obtained by $\sum_i s_i / C$) gives $\lceil 2.2 \rceil = 3$ bins.
4. Exhibit a feasible packing using 3 bins.

Exercise 51 (At-least-one and threshold constraints). Binary variables $y_1, \ldots, y_7$ indicate which of seven tasks are completed. Write linear constraints for each rule:

1. At least one task in the set $\{2, 4, 6\}$ must be completed.
2. If task 1 is completed, then at least one of tasks 3 or 5 must also be completed.
3. Tasks 6 and 7 must both be completed or neither is completed (they are bundled).
4. The total number of completed tasks is either 0, 3, or 6 (a multiple of 3 or zero). *Hint:* introduce auxiliary binary variables z_0, z_1, z_2 with $\sum z_k = 1$.

Exercise 52 (Bounding variables for valid Big-M). A common modelling mistake is to choose M too small, which may cut off integer-feasible solutions, or too large, which weakens the LP relaxation.

1. A variable $x \geq 0$ appears in the constraint $x \leq My$ (binary y). If x is otherwise bounded by $x \leq 50$, what is the tightest valid M?
2. Now suppose x is unbounded. Explain why a valid finite M may not exist and how this complicates the Big-M approach.
3. Given the either-or constraint: "$2x_1 - x_2 \leq 4$ or $x_1 + 3x_2 \leq 9$" with $x_1, x_2 \in [0, 10]$, determine tight Big-M values M_1 and M_2 for each constraint and write the Big-M reformulation.

Exercise 53 (MILP: production with setup). A manufacturer has three products ($j = 1, 2, 3$) and two machines ($i = 1, 2$). Production quantities are $x_{ij} \geq 0$ (continuous). A setup binary $y_{ij} \in \{0, 1\}$ equals 1 iff product j is run on machine i in the current period. Machine i has capacity C_i, and production of product j on machine i requires a_{ij} time units. A fixed setup time s_{ij} is incurred whenever $y_{ij} = 1$.

1. Write the capacity constraints incorporating both production time and setup time.
2. Write the linking constraint ensuring $x_{ij} = 0$ whenever $y_{ij} = 0$ (using Big-$M = C_i / a_{ij}$).
3. Write the complete MILP objective minimising total production cost $\sum_{i,j} c_{ij} x_{ij}$ plus total setup cost $\sum_{i,j} f_{ij} y_{ij}$.

Exercise 54 (Clique inequalities as valid inequalities). For the independent set ILP on a graph $G = (V, E)$, any clique $Q \subseteq V$ yields the valid inequality $\sum_{v \in Q} x_v \leq 1$.

1. Show that every clique inequality is implied by the individual edge constraints when $|Q| = 2$.

2. For $|Q| = 3$ (triangle $\{u, v, w\}$), show that the clique inequality $x_u + x_v + x_w \leq 1$ is *not* implied by the three edge constraints over $[0, 1]^3$. Exhibit a fractional point satisfying all edge constraints but violating the clique inequality.
3. Adding all clique inequalities over all maximal cliques strengthens the LP relaxation. Does this guarantee an integer LP relaxation solution for all graphs? Justify.

Exercise 55 (TSP on four cities: full model and LP relaxation). A symmetric TSP on four cities has cost matrix:

$$C = \begin{pmatrix} 0 & 10 & 8 & 9 \\ 10 & 0 & 7 & 5 \\ 8 & 7 & 0 & 6 \\ 9 & 5 & 6 & 0 \end{pmatrix}.$$

1. Write the complete ILP (degree constraints + subtour elimination constraints for all proper subsets of size 2 and 3).
2. List all $\frac{4!}{2} = 12$ distinct tours and their costs. Identify the optimal tour.
3. The LP relaxation of the degree constraints alone (without SECs) may permit subtours. Show that the solution $x_{12} = x_{23} = x_{34} = x_{14} = \frac{1}{2}$ and $x_{13} = x_{24} = \frac{1}{2}$ satisfies all degree constraints but contains subtours (check the relevant SEC).

CHAPTER 4

The Simplex Method

In chapter 2 we proved that if an LP has a finite optimum, it occurs at a *vertex* of the feasible polyhedron (theorem 2.4.6). We also converted the furniture workshop LP (theorem 1.8.1) to **standard form** (theorem 2.7.3), ending up with equalities and non-negative variables. But a theorem that says "an optimum exists at some vertex" does not tell us *which* vertex. We need an *algorithm*.

This chapter turns the geometric insight of Chapter 2—"the optimum is at a vertex"—into a concrete, step-by-step algorithm.

That algorithm is the **Simplex method**, invented by George Dantzig in 1947. Its idea is beautifully simple:

1. Start at *some* vertex of the feasible polyhedron.
2. Look at the neighbouring vertices (connected by edges).
3. If a neighbour has a strictly better objective value, move there.
4. Repeat until no neighbour improves the objective—then the current vertex is optimal.

Geometrically, the Simplex method is a *greedy walk* along the edges of the polytope, always climbing towards higher profit. Algebraically, it maintains a **basis**—a set of columns of the constraint matrix—and swaps one column in and one column out at each step (a *pivot*). This chapter develops both viewpoints in detail.

Simplex is a greedy walk *on the vertex-edge graph of the polytope.*

Road map.

1. Standard form recap and matrix setup (section 4.1).
2. Bases and basic feasible solutions (section 4.2).
3. The reduced echelon form and reduced costs (section 4.3).
4. The optimality condition (section 4.4).
5. The Simplex iteration: pivoting (section 4.5).
6. Complete worked example on the furniture workshop (section 4.6).
7. Geometric interpretation (section 4.7).
8. Degeneracy and cycling (section 4.8).
9. Finding an initial BFS: the two-phase method (section 4.9).
10. Complexity and history (section 4.10).

4.1 Standard Form Recap

The Simplex method operates on LPs in **standard form** (theorem 2.6.2). Recall:

All of this was set up in section 2.6. We restate it here for easy reference.

$$\text{maximize} \quad c^\top x \qquad \text{subject to} \quad Ax = b, \quad x \geq 0, \tag{4.1}$$

where $A \in \mathbb{R}^{m \times d}$, $c \in \mathbb{R}^d$, $b \in \mathbb{R}^m$, and $x \in \mathbb{R}^d$.

We use d for the total number of variables (including slacks) and m for the number of equality constraints. The *degrees of freedom* are $n = d - m$: these are the variables whose values we get to "choose" once we fix a basis.

> *Remark* 4.1.1. We assume throughout that A has **full row rank**: $\operatorname{rank}(A) = m$. This is not restrictive—if two rows of A are linearly dependent, one of the corresponding equations is redundant and can be dropped.

For concreteness, let us recall the furniture workshop LP in standard form (theorem 2.7.3):

$$\begin{aligned} \text{maximize} \quad & 30\,x_1 + 50\,x_2 + 0\,s_1 + 0\,s_2 \\ \text{subject to} \quad & 2\,x_1 + 5\,x_2 + s_1 = 40 \\ & 4\,x_1 + 2\,x_2 + s_2 = 32 \\ & x_1, x_2, s_1, s_2 \geq 0. \end{aligned} \tag{4.2}$$

Here $d = 4$ (variables x_1, x_2, s_1, s_2), $m = 2$ (two equality constraints), and $n = d - m = 2$. The matrices are:

$d = 4$ variables, $m = 2$ equations, $n = d - m = 2$ degrees of freedom.

$$c = \begin{pmatrix} 30 \\ 50 \\ 0 \\ 0 \end{pmatrix}, \quad A = \begin{pmatrix} 2 & 5 & 1 & 0 \\ 4 & 2 & 0 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 40 \\ 32 \end{pmatrix}.$$

4.2 Bases and Basic Solutions

The key algebraic idea behind the Simplex method is the notion of a *basis*. Think of it this way: we have d variables but only m equations. If we "fix" $n = d - m$ of the variables to zero, the remaining m variables are determined by the m equations—provided the corresponding columns of A are linearly independent.

A basis is a selection of m columns of A that are linearly independent. It determines a vertex.

> **Definition 4.2.1** (Basis and non-basis)**.** Let $A \in \mathbb{R}^{m \times d}$ with $\operatorname{rank}(A) = m$. A **basis** $\mathcal{B}$ is a set of m column indices $\mathcal{B} \subseteq \{1, \ldots, d\}$ such that the $m \times m$ submatrix $A_{\mathcal{B}}$ (formed by the columns indexed by $\mathcal{B}$) is **non-singular** (invertible).
>
> The complementary set $\mathcal{N} = \{1, \ldots, d\} \setminus \mathcal{B}$ is called the **non-basis**. It has $n = d - m$ indices.

Intuitively, picking a basis means deciding which m variables are "active"—they carry values determined by the system $Ax = b$—while the remaining n variables are "shut off" at zero. The active variables are called **basic variables** ($x_{\mathcal{B}}$) and the shut-off ones are **non-basic variables** ($x_{\mathcal{N}}$).

> **Example 4.2.2** (Bases of the furniture workshop)**.** With $d = 4$ columns and $m = 2$ rows, a basis consists of 2 column indices. The total number of possible bases is $\binom{d}{m} = \binom{4}{2} = 6$. Let's list them all, labelling the columns $1 = x_1$, $2 = x_2$, $3 = s_1$, $4 = s_2$:

$\mathcal{B}$	Columns of $A_{\mathcal{B}}$	Invertible?
$\{1,2\}$	$\left(\begin{smallmatrix}2&5\\4&2\end{smallmatrix}\right)$	Yes ($\det = 4-20=-16\neq 0$)
$\{1,3\}$	$\left(\begin{smallmatrix}2&1\\4&0\end{smallmatrix}\right)$	Yes ($\det = 0-4=-4\neq 0$)
$\{1,4\}$	$\left(\begin{smallmatrix}2&0\\4&1\end{smallmatrix}\right)$	Yes ($\det = 2\neq 0$)
$\{2,3\}$	$\left(\begin{smallmatrix}5&1\\2&0\end{smallmatrix}\right)$	Yes ($\det = 0-2=-2\neq 0$)
$\{2,4\}$	$\left(\begin{smallmatrix}5&0\\2&1\end{smallmatrix}\right)$	Yes ($\det = 5\neq 0$)
$\{3,4\}$	$\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$	Yes (identity!)

All six pairs give invertible submatrices, so there are six bases. In general, not all $\binom{d}{m}$ column sets form a basis—some may be singular. Here we are lucky: all six work.

Given a basis $\mathcal{B}$, we can partition the system $Ax = b$ by separating basic and non-basic columns:

$$A_{\mathcal{B}}\, x_{\mathcal{B}} + A_{\mathcal{N}}\, x_{\mathcal{N}} = b.$$

Setting the non-basic variables to zero ($x_{\mathcal{N}} = 0$) gives us a system with a unique solution.

Definition 4.2.3 (Basic solution)**.** Given a basis $\mathcal{B}$, the **basic solution (BS)** associated with $\mathcal{B}$ is obtained by setting $x_{\mathcal{N}} = 0$ and solving

$$A_{\mathcal{B}}\, x_{\mathcal{B}} = b \qquad \Longrightarrow \qquad x_{\mathcal{B}} = A_{\mathcal{B}}^{-1}\, b.$$

A basic solution always satisfies $Ax = b$ (it is an algebraic solution of the system), but it may violate $x \geq 0$. When it doesn't, it earns a special name.

A basic solution may have negative components—it's a solution of $Ax = b$, but it might not satisfy $x \geq 0$.

Definition 4.2.4 (Basic feasible solution — BFS)**.** A basic solution is a **basic feasible solution (BFS)** if all its components are non-negative:

$$x_{\mathcal{B}} = A_{\mathcal{B}}^{-1}\, b \geq 0.$$

(We already have $x_{\mathcal{N}} = 0 \geq 0$ by construction.)

Intuitively, a BFS is a corner point of the feasible polyhedron. The non-basic variables being zero means we are "pushed against" n constraint boundaries (the ones $x_j \geq 0$ for $j \in \mathcal{N}$), and the m equalities pin down the rest.

Example 4.2.5 (Basic solutions of the furniture workshop)**.** Let's compute the basic solution for each of the six bases from theorem 4.2.2:

$\mathcal{B}$	$x_{\mathcal{B}} = A_{\mathcal{B}}^{-1}b$	Full (x_1, x_2, s_1, s_2)	Feasible?	Objective
$\{1,2\}$	$(5,6)$	$(5,6,0,0)$	Yes	**450**
$\{1,3\}$	$(8,24)$	$(8,0,24,0)$	Yes	240
$\{1,4\}$	$(20,-48)$	$(20,0,0,-48)$	No	—
$\{2,3\}$	$(8,-8)$	$(0,8,-8,0)$	No	—
$\{2,4\}$	$(8,16)$	$(0,8,0,16)$	Yes	400
$\{3,4\}$	$(40,32)$	$(0,0,40,32)$	Yes	0

Four of the six basic solutions are feasible (BFS). They correspond to the four vertices $(5,6)$, $(8,0)$, $(0,8)$, and $(0,0)$ of the furniture feasible region. Among these, the one with basis $\mathcal{B} = \{1,2\}$—corresponding to the vertex $(x_1, x_2) = (5,6)$—achieves the highest objective value of 450. This is exactly the optimal solution we found graphically in theorem 2.2.4.

4.2.1 The fundamental correspondence

The connection between BFS and vertices is not a coincidence: it is a theorem.

Theorem 4.2.6 (BFS ⇔ vertex)**.** *Let $P = \{x \in \mathbb{R}^d : Ax = b,\ x \geq 0\}$ be the feasible polyhedron of a standard-form LP with* $\operatorname{rank}(A) = m$*. Then $\bar{x} \in P$ is a* ***vertex*** *(extreme point) of P if and only if $\bar{x}$ is a* ***basic feasible solution****.*

Vertices = BFS. The Simplex method searches over BFS, hence over vertices.

This is the algebraic counterpart of the geometric theorem 2.4.6: since the optimum of a bounded LP occurs at a vertex, and vertices are exactly the BFS, the Simplex method only needs to search through basic feasible solutions.

Formal details — Proof sketch: BFS ⇔ vertex

The key idea is that fixing the non-basic variables to zero pins down a unique point, and that uniqueness is exactly what it means to be an extreme point.

(⇒) BFS ⇒ vertex.

1. We suppose for contradiction that $\bar{x}$ is a BFS that is not extreme, writing $\bar{x} = \lambda u + (1-\lambda)w$ with $u, w \in P$, $0 < \lambda < 1$, $u \neq w$.
2. We deduce that $u_j = w_j = 0$ for all $j \in \mathcal{N}$ because $\bar{x}_j = 0$ and both $u_j, w_j \geq 0$ force each to zero.
3. We conclude $u = w = \bar{x}$ because both u and w must satisfy $A_{\mathcal{B}}u_{\mathcal{B}} = b$ and $A_{\mathcal{B}}w_{\mathcal{B}} = b$, and $A_{\mathcal{B}}$ is invertible, giving $u_{\mathcal{B}} = w_{\mathcal{B}} = A_{\mathcal{B}}^{-1}b = \bar{x}_{\mathcal{B}}$. This contradicts $u \neq w$.

(⇐) Vertex ⇒ BFS.

1. We let $S = \{j : \bar{x}_j > 0\}$ be the support of $\bar{x}$ because these are the only columns that participate non-trivially in $Ax = b$.
2. We argue that the columns $\{A_j : j \in S\}$ are linearly independent because if they were not, $\bar{x}$ could be expressed as a strict convex combination of two distinct feasible points, contradicting that it is a vertex.
3. We extend S to a full basis $\mathcal{B}$ (of size m) because $|S| \leq m$ and we can add indices with $\bar{x}_j = 0$ to complete the set; those variables are already zero, so $\bar{x}$ is the BFS for this basis.

Thus every vertex of P is a BFS and every BFS is a vertex, establishing the one-to-one correspondence between bases and extreme points. □

A useful corollary bounds the search space.

Corollary 4.2.7 (Finite number of BFS)**.** *The number of basic feasible solutions is at most* $\binom{d}{m}$.

For the furniture workshop, $\binom{4}{2} = 6$. For a "real" LP with $d = 10{,}000$ variables and $m = 5{,}000$ constraints, $\binom{d}{m}$ is astronomically large. The genius of the Simplex method is that it visits only a *tiny* fraction of these.

4.3 The Reduced Form and Reduced Costs

The reduced form rewrites the system so that basic variables are expressed in terms of non-basic ones.

Suppose we have a basis $\mathcal{B}$ with the corresponding BFS $\bar{x}$. We want to know: can we *improve* the objective by bringing a non-basic variable into the basis? To answer this, we rewrite the entire LP in terms of the non-basic variables $x_{\mathcal{N}}$.

Starting from $A_{\mathcal{B}} x_{\mathcal{B}} + A_{\mathcal{N}} x_{\mathcal{N}} = b$, we solve for the basic variables:

$$x_{\mathcal{B}} = A_{\mathcal{B}}^{-1} b - A_{\mathcal{B}}^{-1} A_{\mathcal{N}}\, x_{\mathcal{N}} = \bar{b} - \bar{A}_{\mathcal{N}}\, x_{\mathcal{N}}, \tag{4.3}$$

where we define the shorthand

$$\bar{b} \;=\; A_{\mathcal{B}}^{-1} b, \qquad \bar{A}_{\mathcal{N}} \;=\; A_{\mathcal{B}}^{-1} A_{\mathcal{N}}.$$

Now substitute into the objective $z = c^\top x = c_{\mathcal{B}}^\top x_{\mathcal{B}} + c_{\mathcal{N}}^\top x_{\mathcal{N}}$:

$$\begin{aligned} z &= c_{\mathcal{B}}^\top \big(\bar{b} - \bar{A}_{\mathcal{N}}\, x_{\mathcal{N}}\big) + c_{\mathcal{N}}^\top x_{\mathcal{N}} \\ &= c_{\mathcal{B}}^\top \bar{b} + \big(c_{\mathcal{N}}^\top - c_{\mathcal{B}}^\top \bar{A}_{\mathcal{N}}\big)\, x_{\mathcal{N}} \\ &= \bar{z} + \hat{c}_{\mathcal{N}}^\top\, x_{\mathcal{N}}, \end{aligned} \tag{4.4}$$

where we have defined two crucial quantities.

Definition 4.3.1 (Current objective value)**.** The objective value at the current BFS is

$$\bar{z} = c_{\mathcal{B}}^\top A_{\mathcal{B}}^{-1} b = c_{\mathcal{B}}^\top \bar{b}.$$

Definition 4.3.2 (Reduced costs)**.** For each non-basic variable x_j ($j \in \mathcal{N}$), the **reduced cost** is

$$\hat{c}_j = c_j - c_{\mathcal{B}}^\top A_{\mathcal{B}}^{-1} A_j = c_j - c_{\mathcal{B}}^\top \bar{a}_j,$$

where $\bar{a}_j = A_{\mathcal{B}}^{-1} A_j$ is the j-th column of A expressed in the current basis.

In vector form: $\hat{c}_{\mathcal{N}} = c_{\mathcal{N}} - \bar{A}_{\mathcal{N}}^\top c_{\mathcal{B}}$.

The reduced cost $\hat{c}_j$ tells us by how much the objective would change per unit increase of x_j from zero, assuming we adjust $x_{\mathcal{B}}$ to maintain feasibility.

The reduced cost $\hat{c}_j$ has a clear economic interpretation: it measures the *rate of improvement* of the objective when we increase non-basic variable x_j from zero, while adjusting the basic variables to keep $Ax = b$. If $\hat{c}_j > 0$ (in a maximisation problem), then increasing x_j would increase z—so the current BFS is not optimal. If all $\hat{c}_j \le 0$, no non-basic variable can improve the objective, and we should stop.

Example 4.3.3 (Reduced costs at the initial basis)**.** Start the furniture workshop LP with the natural initial basis $\mathcal{B} = \{3, 4\}$ (the slack variables s_1, s_2). Then:

$$A_{\mathcal{B}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I, \qquad c_{\mathcal{B}} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

The current BFS is $\bar{b} = A_{\mathcal{B}}^{-1} b = b = (40, 32)^\top$, which gives $(x_1, x_2, s_1, s_2) = (0, 0, 40, 32)$ and $\bar{z} = c_{\mathcal{B}}^\top \bar{b} = 0$.

The non-basic variables are x_1 and x_2 (indices 1 and 2), with $c_{\mathcal{N}} = (30, 50)^\top$. The reduced costs are:

$$\hat{c}_1 = c_1 - c_{\mathcal{B}}^\top A_{\mathcal{B}}^{-1} A_1 = 30 - (0, 0) \begin{pmatrix} 2 \\ 4 \end{pmatrix} = 30, \qquad \hat{c}_2 = 50 - (0, 0) \begin{pmatrix} 5 \\ 2 \end{pmatrix} = 50.$$

Both are positive: the current BFS $(0, 0)$ is *not* optimal. We can improve the objective by increasing x_1 or x_2 from zero.

4.4 The Optimality Condition

Equation (4.4) makes the optimality condition immediate.

Theorem 4.4.1 (Simplex optimality condition)**.** *Consider a maximisation LP in standard form with current BFS $\bar{x}$ (basis $\mathcal{B}$). If every non-basic reduced cost satisfies*

$$\hat{c}_j \le 0 \qquad \text{for all } j \in \mathcal{N},$$

*then $\bar{x}$ is an **optimal solution**.*

Proof. Intuitively, if every direction we can move from the current vertex decreases or maintains the objective, we cannot improve.

1. We use the reduced-form expression (4.4) because it writes the objective of any feasible point in terms of the non-basic variables: $z = \bar{z} + \sum_{j \in \mathcal{N}} \hat{c}_j x_j$.
2. We bound each term because $x_j \ge 0$ for all j and $\hat{c}_j \le 0$ for all $j \in \mathcal{N}$ together imply $\hat{c}_j x_j \le 0$ for every non-basic index.
3. We conclude that $z \le \bar{z}$ for every feasible x, while the current BFS achieves $z = \bar{z}$ exactly (all non-basics are zero).

Thus, no feasible solution can achieve a higher objective value than the current BFS, confirming its optimality. □

Notice how clean this is: no complicated linear algebra, no calculus—just the observation that non-negative variables multiplied by non-positive coefficients cannot increase the sum.

This is a sufficient *condition. In the non-degenerate case, it is also necessary.*

Remark 4.4.2. For a **minimisation** problem, the condition reverses: optimality holds when $\hat{c}_j \ge 0$ for all $j \in \mathcal{N}$.

4.5 The Simplex Iteration

If the optimality condition is not met, at least one reduced cost $\hat{c}_j > 0$ exists. The Simplex method performs one **pivot** to move to a neighbouring BFS with a higher objective value. Let's describe this step by step.

Each iteration moves from one BFS to an adjacent BFS with a better (or equal) objective value.

4.5.1 Step 1: Check optimality

Compute (or read off) the reduced costs $\hat{c}_j$ for all $j \in \mathcal{N}$. If $\hat{c}_j \leq 0$ for every $j \in \mathcal{N}$, the current BFS is **optimal**—stop.

Otherwise, proceed to Step 2.

4.5.2 Step 2: Select the entering variable

Dantzig's rule: choose the non-basic variable with the largest positive reduced cost.

Choose a non-basic index $j^* \in \mathcal{N}$ with $\hat{c}_{j^*} > 0$. Several selection rules exist:

- **Dantzig's rule** (largest coefficient): $j^* = \arg\max_{j \in \mathcal{N}} \hat{c}_j$. This gives the steepest rate of improvement per unit increase.
- **Bland's rule** (smallest index): $j^* = \min\{j \in \mathcal{N} : \hat{c}_j > 0\}$. This prevents cycling (see section 4.8).

The variable x_{j^*} will **enter the basis**.

4.5.3 Step 3: Check for unboundedness

Compute the column $\bar{a}_{j^*} = A_{\mathcal{B}}^{-1} A_{j^*}$—the entering column expressed in the current basis.

If $\bar{a}_{j^*} \leq 0$ (every component is non-positive), then increasing x_{j^*} will never cause any basic variable to become negative. We can push $x_{j^*} \to +\infty$, and the objective $z = \bar{z} + \hat{c}_{j^*} x_{j^*} \to +\infty$. The LP is **unbounded**—stop.

> *Remark* 4.5.1. Unboundedness means the feasible polyhedron extends to infinity in a direction that keeps improving the objective. In practical problems this usually signals a modelling error (a forgotten constraint).

4.5.4 Step 4: Ratio test (select the leaving variable)

If at least one component $\bar{a}_{ij^*} > 0$, we must limit how much x_{j^*} can increase before a basic variable hits zero. From (4.3), as x_{j^*} increases from 0 to some value $\theta \geq 0$ (with all other non-basic variables staying at 0), the basic variables become:

$$x_{\mathcal{B}(i)} = \bar{b}_i - \bar{a}_{ij^*}\,\theta, \qquad i = 1, \ldots, m.$$

For $x_{\mathcal{B}(i)}$ to remain non-negative, we need $\theta \leq \bar{b}_i / \bar{a}_{ij^*}$ whenever $\bar{a}_{ij^*} > 0$.

Definition 4.5.2 (Ratio test)**.** The maximum increase of the entering variable x_{j^*} is

$$\theta^* = \min_{i : \bar{a}_{ij^*} > 0} \frac{\bar{b}_i}{\bar{a}_{ij^*}}.$$

Let i^* be the index where this minimum is attained. The variable $x_{\mathcal{B}(i^*)}$ is the **leaving variable**: it drops to zero and exits the basis.

The ratio test finds the tightest bottleneck: the basic variable that hits zero first as we increase the entering variable.

Intuitively, we are sliding along an edge of the polytope (increasing x_{j^*}) until we hit the next vertex (some basic variable reaches zero). The ratio test identifies which vertex that is.

4.5.5 Step 5: Pivot

Update the basis: replace the leaving index $\mathcal{B}(i^*)$ with the entering index j^*. The new basis is

$$\mathcal{B}' = \big(\mathcal{B} \setminus \{\mathcal{B}(i^*)\}\big) \cup \{j^*\}.$$

Recompute $\bar{b}$, $\bar{z}$, and the reduced costs for the new basis. Then return to Step 1.

The new objective value is:

$$\bar{z}' = \bar{z} + \hat{c}_{j^*}\,\theta^*.$$

Since $\hat{c}_{j^*} > 0$ and $\theta^* \geq 0$, the objective does not decrease. If $\theta^* > 0$, it strictly increases.

Tableau row operations. The idea is simply Gaussian elimination on the tableau — we want the entering column to become a unit vector (1 in the pivot row, 0 elsewhere), which we achieve by the standard row operations:

Let $r = i^*$ be the leaving row and j^* the entering column, with pivot element $\bar{a}_{rj^*}$.

1. We normalise the pivot row because we need the entering column to show a 1 in row r (reflecting that x_{j^*} is now basic with coefficient 1):

$$a_{rj} \;\leftarrow\; \frac{a_{rj}}{a_{rj^*}} \qquad \text{for all } j.$$

2. We eliminate the entering variable from every other row $i \neq r$ because a basic column must have a zero in all rows except its own:

$$a_{ij} \;\leftarrow\; a_{ij} - \frac{a_{ij^*}}{a_{rj^*}}\,a_{rj} \qquad \text{for all } j,\ i \neq r.$$

3. We apply the same elimination to the objective row because the objective expression must also be re-expressed in terms of the non-basic variables under the new basis.

These three steps together maintain the invariant that basic columns form an identity submatrix and that reduced costs are correctly updated.

Algorithm 1 summarises the complete procedure.

Algorithm 1: Simplex method (maximisation)

Input: LP in standard form: maximize $c^\top x$ s.t. $Ax = b$, $x \geq 0$; initial basis $\mathcal{B}$ with BFS.

Output: Optimal BFS or declaration of unboundedness.

Compute $\bar{b} = A_{\mathcal{B}}^{-1} b$
Compute $\bar{z} = c_{\mathcal{B}}^\top \bar{b}$
while *true* **do**
  Compute reduced costs $\hat{c}_j$ for all $j \in \mathcal{N}$
  if $\hat{c}_j \leq 0$ *for all* $j \in \mathcal{N}$ **then**
    return current BFS is **optimal** with value $\bar{z}$
  Select entering variable x_{j^*} with $\hat{c}_{j^*} > 0$
  Compute $\bar{a}_{j^*} = A_{\mathcal{B}}^{-1} A_{j^*}$
  if $\bar{a}_{ij^*} \leq 0$ *for all* $i = 1, \ldots, m$ **then**
    return LP is **unbounded**
  Ratio test: $\theta^* = \min_{i:\bar{a}_{ij^*}>0} \bar{b}_i / \bar{a}_{ij^*}$
  Let i^* be the minimising index
  Pivot: $\mathcal{B} \leftarrow (\mathcal{B} \setminus \{\mathcal{B}(i^*)\}) \cup \{j^*\}$
  Update $\bar{b}$, $\bar{z}$, reduced costs

4.6 Worked Example: The Furniture Workshop

We now walk through the Simplex method on our familiar furniture workshop LP, step by step.

Let's execute the Simplex method on the furniture workshop LP (4.2), which we repeat for convenience:

$$\begin{aligned} \text{maximize} \quad & 30\,x_1 + 50\,x_2 \\ \text{subject to} \quad & 2\,x_1 + 5\,x_2 + s_1 = 40 \\ & 4\,x_1 + 2\,x_2 + s_2 = 32 \\ & x_1, x_2, s_1, s_2 \geq 0. \end{aligned}$$

We use the **Simplex tableau**—a compact bookkeeping device that tracks all the information we need. Each row corresponds to a basic variable, and the bottom row tracks the objective function (via reduced costs).

Definition 4.6.1 (Simplex tableau)**.** The **Simplex tableau** for a basis $\mathcal{B}$ is the augmented matrix:

$$\left[\begin{array}{c|c} A_{\mathcal{B}}^{-1}A & A_{\mathcal{B}}^{-1}b \\ \hline \hat{c}^{\top} & \bar{z} \end{array}\right]$$

where $\hat{c}_j = c_j - c_{\mathcal{B}}^{\top} A_{\mathcal{B}}^{-1} A_j$ are the reduced costs. The columns corresponding to basic variables contain the identity matrix, and their reduced costs are zero.

Iteration 0: Initial tableau

Initial basis: $\mathcal{B} = \{s_1, s_2\} = \{3, 4\}$. The slack variables start in the basis with $A_{\mathcal{B}} = I$, giving the BFS $(x_1, x_2, s_1, s_2) = (0, 0, 40, 32)$ and $\bar{z} = 0$.

	x_1	x_2	s_1	s_2	RHS
s_1	2	5	1	0	40
s_2	4	2	0	1	32
z	30	50	0	0	0

Reduced costs in the bottom row: both $\hat{c}_1 = 30 > 0$ and $\hat{c}_2 = 50 > 0$. Not optimal.

Check optimality: The bottom row has $\hat{c}_1 = 30 > 0$ and $\hat{c}_2 = 50 > 0$. Not optimal.

Entering variable: By Dantzig's rule, pick the largest reduced cost: $\hat{c}_2 = 50$, so x_2 enters.

Ratio test: The entering column (under x_2) is $(5, 2)^{\top}$. Both entries are positive, so:

$$\text{Row 1: } \frac{40}{5} = 8, \qquad \text{Row 2: } \frac{32}{2} = 16.$$

Minimum ratio is 8 at row 1. So s_1 **leaves** the basis.

Pivot: We pivot on the element in row 1, column x_2 (the value 5). To make this column look like $(1, 0)^{\top}$, we divide row 1 by 5 and subtract 2× the new row 1 from row 2.

Row operations:

- $R_1 \leftarrow R_1/5$
- $R_2 \leftarrow R_2 - 2R_1$
- $R_z \leftarrow R_z - 50R_1$

Iteration 1: After the first pivot

New basis: $\mathcal{B} = \{x_2, s_2\} = \{2, 4\}$.

	x_1	x_2	s_1	s_2	RHS
x_2	$\frac{2}{5}$	1	$\frac{1}{5}$	0	8
s_2	$\frac{16}{5}$	0	$-\frac{2}{5}$	1	16
z	10	0	-10	0	400

The new BFS is $(x_1, x_2, s_1, s_2) = (0, 8, 0, 16)$ with $\bar{z} = 400$.

After one pivot, the BFS is $(0, 8, 0, 16)$ with $z = 400$. Geometrically, we've moved from the origin to the vertex $(0, 8)$.

Let's verify: $x_2 = 8$ means we produce 8 tables. Wood: $5 \times 8 = 40$ (exactly the supply); labour: $2 \times 8 = 16$ (out of 32 available, so $s_2 = 16$ is the labour slack). Profit: $50 \times 8 = 400$. Correct!

Check optimality: $\hat{c}_1 = 10 > 0$, $\hat{c}_{s_1} = -10 \leq 0$. Not yet optimal ($x_1$ still wants to enter).

Entering variable: The only positive reduced cost is $\hat{c}_1 = 10$, so x_1 enters.

Ratio test: The entering column (under x_1) is $(\frac{2}{5}, \frac{16}{5})^\top$. Both positive:

$$\text{Row 1: } \frac{8}{2/5} = 20, \qquad \text{Row 2: } \frac{16}{16/5} = 5.$$

Minimum ratio is 5 at row 2. So s_2 **leaves** the basis.

Pivot: Pivot on row 2, column x_1 (the value $\frac{16}{5}$).

Row operations:

- $R_2 \leftarrow R_2 \cdot \frac{5}{16}$
- $R_1 \leftarrow R_1 - \frac{2}{5}R_2$
- $R_z \leftarrow R_z - 10R_2$

Iteration 2: Optimal tableau

New basis: $\mathcal{B} = \{x_2, x_1\} = \{2, 1\}$.

	x_1	x_2	s_1	s_2	RHS
x_2	0	1	$\frac{1}{4}$	$-\frac{1}{8}$	6
x_1	1	0	$-\frac{1}{8}$	$\frac{5}{16}$	5
z	0	0	$-\frac{35}{4}$	$-\frac{25}{8}$	450

Check optimality: $\hat{c}_{s_1} = -\frac{35}{4} < 0$ and $\hat{c}_{s_2} = -\frac{25}{8} < 0$. **All reduced costs are non-positive.** By theorem 4.4.1, the current BFS is **optimal**.

All reduced costs ≤ 0. The Simplex declares $(x_1, x_2) = (5, 6)$ optimal with profit 450.

The optimal solution is $(x_1, x_2, s_1, s_2) = (5, 6, 0, 0)$, i.e. $x_1 = 5$ chairs, $x_2 = 6$ tables, both slack variables zero (both resources fully used), with optimal profit $z^* = 450$.

This is exactly the vertex $(5, 6)$ found graphically in theorem 2.2.4!

Summary of the Simplex path:

Iteration	Basis	BFS (x_1, x_2)	Objective z	Entering / Leaving
0	$\{s_1, s_2\}$	$(0, 0)$	0	x_2 enters, s_1 leaves
1	$\{x_2, s_2\}$	$(0, 8)$	400	x_1 enters, s_2 leaves
2	$\{x_2, x_1\}$	$(5, 6)$	450	OPTIMAL

The Simplex method visited only 3 of the 4 feasible vertices (it skipped $(8, 0)$) and reached the optimum in just 2 pivots.

4.7 Geometric Interpretation

The algebraic pivots have a beautiful geometric meaning. Each BFS corresponds to a vertex of the feasible polyhedron, and each pivot moves along an **edge** to an adjacent vertex. The Simplex method is therefore a *walk on the vertex-edge graph* of the polytope, always climbing uphill (in the direction of increasing objective).

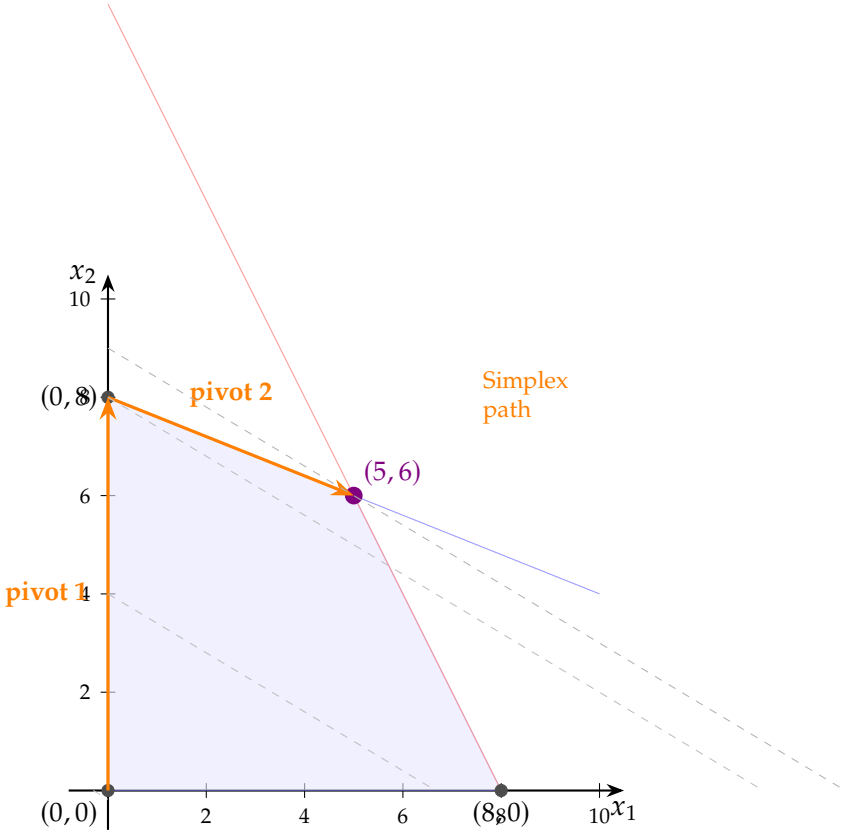


Figure 4.1: The Simplex path on the furniture workshop LP. Starting from the origin $(0, 0)$, the method moves along the x_2-axis to $(0, 8)$, then along an edge to the optimal vertex $(5, 6)$. The dashed gray lines are objective isolines ($z = 200, 400, 450$). Compare with fig. 1.3.

Notice how the Simplex method *does not* cut across the interior of the polytope. It is constrained to walk along edges. This is because, at each step, exactly one non-basic variable changes from zero to a positive value (the entering variable), while one basic variable drops to zero (the leaving variable). The set of variables that are zero defines which "face" of the polytope we are on, and changing exactly one variable at a time means we move along a one-dimensional face—an edge.

4.7.1 Higher dimensions

In two dimensions, the polytope is a polygon and the Simplex path visits corners. In higher dimensions, the polytope has far more edges and vertices, but the principle is the same.

The vertex-edge graph of a polytope can be very large. In standard form with d variables and m independent equality constraints, there are at most $\binom{d}{m}$

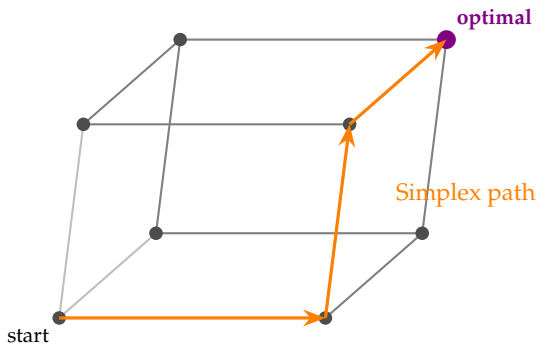


Figure 4.2: Schematic of the Simplex method on a three-dimensional polytope. The orange path follows edges from vertex to vertex, each time improving the objective, until reaching the violet optimal vertex.

candidate bases, hence at most that many BFS. The Simplex method typically visits far fewer than this—but in the worst case, it can visit exponentially many (see section 4.10).

4.8 Degeneracy and Cycling

Degeneracy occurs when a BFS has a basic variable equal to zero. It means the vertex is "over-determined."

So far, everything has gone smoothly: each pivot strictly increased the objective, and after a finite number of pivots, we reached the optimum. But there is a subtle complication that can arise in practice.

Definition 4.8.1 (Degenerate BFS)**.** A basic feasible solution is **degenerate** if one or more basic variables are equal to zero: $\bar{b}_i = 0$ for some i.

Intuitively, a degenerate BFS corresponds to a vertex where *more than* n constraints are active (tight). In two dimensions, a non-degenerate vertex is the intersection of exactly two lines; a degenerate vertex has three or more constraint lines passing through it.

At a degenerate vertex, the same geometric point can correspond to multiple different bases.

Example 4.8.2 (Degeneracy in the furniture workshop)**.** The furniture LP in theorem 4.2.5 is non-degenerate: at each of its four vertices, the associated feasible basis has strictly positive basic variables. For example, the origin uses basis $\mathcal{B} = \{3, 4\}$ with $s_1 = 40 > 0$ and $s_2 = 32 > 0$.

For a clean example of degeneracy, suppose we add the constraint $x_1 + x_2 \leq 8$ to the furniture workshop. At vertex $(0, 8)$, the constraints $x_1 \geq 0$, $2x_1 + 5x_2 \leq 40$, and $x_1 + x_2 \leq 8$ are all tight. That's three active constraints for a two-dimensional problem—one more than needed. The vertex is degenerate.

4.8.1 Degenerate pivots

When the current BFS is degenerate, the ratio test may give $\theta^* = 0$. This happens when a constraint boundary that is not part of our current basis is already tight at the current vertex.

If we choose an entering variable x_{j^*} and attempt to move along an edge (relaxing one of the active constraints in our basis), we immediately hit this extra tight constraint. Since we are already on its boundary, we cannot move in

that direction without violating it. Thus, the maximum step size is restricted to $\theta^* = 0$.

Because we do not move geometrically, the values of all variables remain unchanged. Consequently, the new objective value is:

$$\bar{z}' = \bar{z} + \bar{c}_{j^*} \cdot \theta^* = \bar{z} + \bar{c}_{j^*} \cdot 0 = \bar{z}$$

The algebraic basis changes (we swap the entering variable with the slack variable of the blocking constraint), but we stay at the exact same geometric vertex.

This is visualized in Figure 4.3.

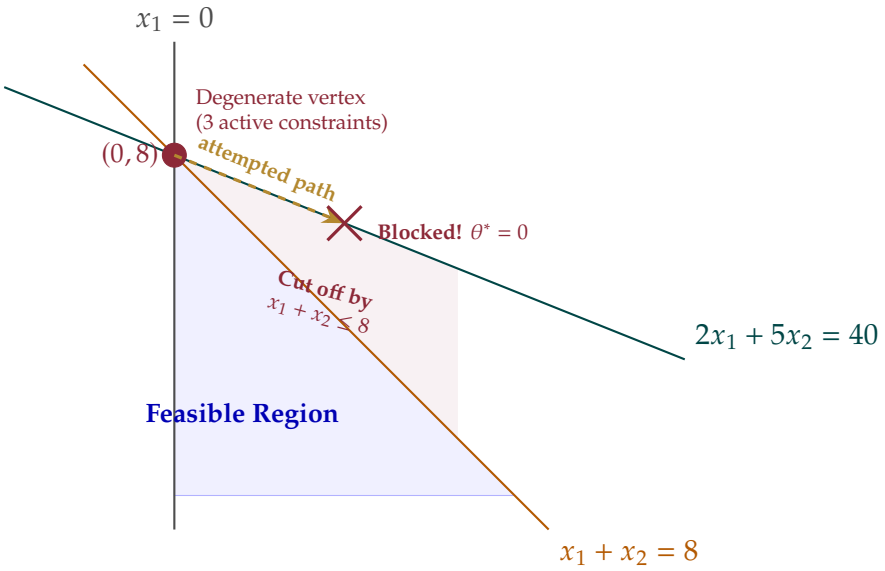


Figure 4.3: Visualization of a degenerate pivot at the vertex $(0, 8)$ in Example 4.8.2. Three constraints meet at $(0, 8)$. If we choose a basis where $x_1 = 0$ and $2x_1 + 5x_2 = 40$ are non-basic, and try to pivot by letting x_1 enter, we must move along the boundary $2x_1 + 5x_2 = 40$. However, any positive step ($x_1 > 0$) immediately violates the tight constraint $x_1 + x_2 \leq 8$. Thus, the step size is restricted to $\theta^* = 0$, resulting in an algebraic basis change but no geometric movement ($\bar{z}' = \bar{z}$).

Such a **degenerate pivot** is not an error—it is just the method exploring different bases at the same vertex. Usually, after one or two degenerate pivots, it finds a productive direction and moves to a new vertex with a better objective.

4.8.2 Cycling

Cycling is theoretically possible but extremely rare in practice.

The danger of degenerate pivots is **cycling**: the method might revisit the same sequence of bases forever, looping without progress. Because any non-degenerate pivot strictly increases the objective value, we can never return to a previously visited basis. Thus, cycling consists entirely of a loop of degenerate pivots ($\theta^* = 0$) at the *same* degenerate vertex.

At a degenerate vertex, there are multiple different bases (combinations of active boundaries) that describe the exact same geometric point. If the pivot rule does not break ties systematically, the Simplex method can transition from one basis to another, circling around the same vertex indefinitely without ever leaving it.

This is visualized in Figure 4.4.

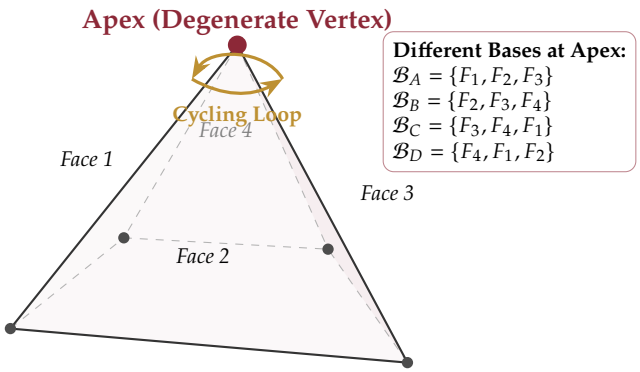


Figure 4.4: Visualization of cycling at a 3D degenerate vertex (the apex of a pyramid). Four constraint faces (F_1, F_2, F_3, F_4) meet at a single point. In 3D, any 3 faces are sufficient to algebraically define a vertex. The Simplex method can perform degenerate pivots ($\theta^* = 0$) that transition between the bases $\mathcal{B}_A \to \mathcal{B}_B \to \mathcal{B}_C \to \mathcal{B}_D \to \mathcal{B}_A$, looping indefinitely without ever moving away from the apex or improving the objective value.

> *Remark* 4.8.3. Cycling *can* happen. Small artificial examples have been constructed to demonstrate it. But in decades of practical use, cycling has almost never been observed in real-world problems.

4.8.3 Bland's rule: guaranteed termination

> **Theorem 4.8.4** (Bland's rule)**.** *If ties in both the entering and leaving variable selection are broken by choosing the variable with the* ***smallest index****, the Simplex method terminates in a finite number of iterations.*

The proof is non-trivial (it proceeds by contradiction, analysing what would happen in a minimal cycling example), but the rule itself is trivially easy to implement: just pick the smallest index among all candidates.

Bland's rule sacrifices some pivot efficiency for guaranteed termination. Dantzig's rule is often faster in practice.

In practice, most implementations use Dantzig's rule (or more sophisticated pricing strategies) and only switch to Bland's rule if degeneracy becomes problematic.

Intermezzo — Is Bland's rule necessary?

For the furniture workshop LP, the Simplex method terminates in two pivots regardless of the pivot rule, because the problem is small and non-degenerate. Bland's rule becomes important for larger problems with many degenerate vertices. In commercial solvers, more sophisticated anti-cycling techniques are used, such as lexicographic pivoting or symbolic perturbation, but they all serve the same purpose: ensuring finite termination even in the presence of degeneracy.

4.9 Finding an Initial BFS: The Two-Phase Method

The Simplex method needs a starting BFS. What if one isn't obvious?

The Simplex method requires an initial basic feasible solution to start. For the furniture workshop, this was easy: the slack variables s_1, s_2 form a natural basis with $A_{\mathcal{B}} = I$ and $\bar{b} = b = (40, 32) \geq 0$.

But not all LPs have such a convenient starting basis. Consider an LP with $\geq$ constraints (surplus variables give negative coefficients in the identity) or = constraints (no slack at all). How do we find an initial BFS?

The **two-phase method** solves this problem elegantly.

4.9.1 Phase I: Find a BFS

The idea is to create an *auxiliary LP* that always has an obvious starting BFS and whose optimal solution (if the value is zero) gives us a BFS for the original problem.

Given the original LP:

$$\text{maximize} \quad c^\top x \qquad \text{subject to} \quad Ax = b, \quad x \geq 0,$$

we first ensure $b \geq 0$ (multiply rows with $b_i < 0$ by -1). Then construct the Phase I problem:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m} a_i \\ \text{subject to} \quad & Ax + Ia = b \\ & x \geq 0, \quad a \geq 0, \end{aligned} \tag{4.5}$$

where $a = (a_1, \ldots, a_m)^\top$ are **artificial variables** and I is the $m \times m$ identity matrix.

The artificial variables a_i provide a trivial starting basis: $x = 0$, $a = b$. Phase I drives them to zero.

Starting BFS for Phase I: Set $x = 0$ and $a = b$. Since $b \geq 0$, this is feasible, and the basis is $\{a_1, \ldots, a_m\}$ with $A_{\mathcal{B}} = I$. We can run the Simplex method on this auxiliary LP.

> **Theorem 4.9.1** (Phase I correctness)**.** *The original LP has a feasible solution if and only if the optimal value of the Phase I problem is zero. Moreover, if the optimal Phase I value is zero, the resulting BFS (after dropping the artificial variables) is a BFS for the original LP.*

Proof. The key idea is that the artificial variables act as a perfect "feasibility gauge": they are zero precisely when the original LP is feasible.

1. We show the Phase I optimum is 0 when the original LP is feasible because any solution $\bar{x}$ with $A\bar{x} = b$, $\bar{x} \geq 0$ gives the Phase I point $(\bar{x}, 0)$ with $\sum a_i = 0$; since $a_i \geq 0$ always, zero is the minimum.
2. We show the original LP is feasible when the Phase I optimum is 0 because at that optimal BFS all artificial variables are zero, so the corresponding x satisfies $Ax = b$ and $x \geq 0$, making it feasible for the original problem.

Thus the Phase I optimal value is zero if and only if the original LP is feasible, and in that case the optimal Phase I BFS provides a valid starting point for Phase II. □

4.9.2 Phase II: Optimise the original objective

Once Phase I has produced a BFS for the original LP, we drop the artificial variables and their columns from the tableau, replace the objective row with the original $c^\top$, and recompute the reduced costs. Then we continue the Simplex method as usual.

Example 4.9.2 (Two-phase method on an LP with ≥ constraints)**.** Consider:

$$\begin{aligned} \text{maximize} \quad & 2x_1 + 3x_2 \\ \text{subject to} \quad & x_1 + x_2 \le 10 \\ & x_1 + 2x_2 \ge 8 \\ & x_1, x_2 \ge 0. \end{aligned}$$

Standard form: Add slack $s_1 \ge 0$ and surplus $s_2 \ge 0$:

$$\begin{aligned} \text{maximize} \quad & 2x_1 + 3x_2 \\ \text{subject to} \quad & x_1 + x_2 + s_1 = 10 \\ & x_1 + 2x_2 - s_2 = 8 \\ & x_1, x_2, s_1, s_2 \ge 0. \end{aligned}$$

We cannot use $\{s_1, s_2\}$ as a starting basis because the s_2 column in A is $(0, -1)^\top$—the matrix $\left(\begin{smallmatrix} 1 & 0 \\ 0 & -1 \end{smallmatrix}\right)$ would give $\bar{b}_2 = -8 < 0$. Instead, we introduce an artificial variable a_1 for the second constraint and solve the Phase I problem:

$$\text{minimize} \quad a_1 \qquad \text{subject to} \quad \begin{cases} x_1 + x_2 + s_1 = 10, \\ x_1 + 2x_2 - s_2 + a_1 = 8, \end{cases} \qquad x_1, x_2, s_1, s_2, a_1 \ge 0.$$

The initial basis is $\{s_1, a_1\}$. After running Phase I (we omit the tableau details), the optimal Phase I value is 0, confirming feasibility. The resulting BFS provides a starting point for Phase II, where we optimise the original objective $2x_1 + 3x_2$.

This complete two-phase process is visualized geometrically in Figure 4.5.

4.9.3 Alternative: The Big-M method

Instead of two phases, we can use a single-phase trick: add artificial variables but penalise them with a very large cost M in the objective:

$$\text{maximize} \quad c^\top x - M \sum_{i=1}^{m} a_i \qquad \text{subject to} \quad Ax + Ia = b, \quad x, a \ge 0.$$

Big-M is simpler to implement but introduces a large parameter M that can cause numerical issues.

If M is large enough, the Simplex method will drive all artificial variables to zero (if the original LP is feasible). The advantage is simplicity—one LP instead of two. The disadvantage is that choosing M too large causes numerical instability, and choosing it too small may give incorrect results. For this reason, the two-phase method is generally preferred in theory, though Big-M is common in practice (with M chosen relative to the problem data).

4.10 Complexity and History

4.10.1 How many pivots does the Simplex method take?

In the worst case, the Simplex method may visit *every* vertex of the polytope.

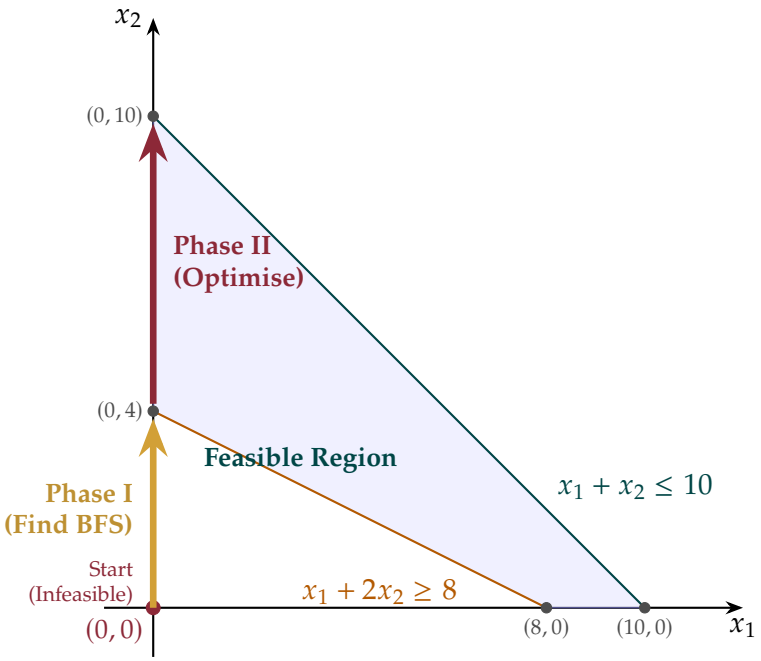


Figure 4.5: Visualization of the Two-Phase Simplex method on Example 4.9.2. The origin $(0, 0)$ is infeasible. Phase I starts at $(0, 0)$, using artificial variables to create a starting basis, and moves along the x_2-axis (driving the artificial variable a_1 to zero) until it hits the boundary at the initial BFS $(0, 4)$. Phase II then starts from $(0, 4)$ and moves along the boundary of the feasible region to find the optimal solution at $(0, 10)$.

Theorem 4.10.1 (Klee–Minty, 1972)**.** *For every $d \geq 1$, there exists a d-dimensional LP with $2d$ constraints for which the Simplex method (with Dantzig's rule) performs $2^d - 1$ pivots—exponential in the dimension.*

The Klee–Minty cube is a "stretched" hypercube that tricks Dantzig's rule into visiting all 2^d vertices.

The **Klee–Minty cube** is a carefully designed distortion of the standard d-dimensional hypercube $[0, 1]^d$. The constraints are chosen so that Dantzig's rule always picks the worst possible pivot, forcing the method to traverse every single vertex before reaching the optimum.

Example 4.10.2 (Klee–Minty cube in 3D)**.** The 3-dimensional Klee–Minty LP is:

$$\begin{aligned} \text{maximize} \quad & 4x_1 + 2x_2 + x_3 \\ \text{subject to} \quad & x_1 \leq 5 \\ & 4x_1 + x_2 \leq 25 \\ & 8x_1 + 4x_2 + x_3 \leq 125 \\ & x_1, x_2, x_3 \geq 0. \end{aligned}$$

The feasible region has $2^3 = 8$ vertices. With Dantzig's rule, the Simplex method visits all 8 vertices before finding the optimum at $(0, 0, 125)$. It takes $2^3 - 1 = 7$ pivots for what should be a trivially small problem.

Despite this worst-case result, the Simplex method performs excellently in practice.

4.10.2 Average-case behaviour

Empirically, on "typical" LPs with m constraints and d variables, the Simplex method takes roughly $O(m)$ to $O(m \log d)$ pivots. Theoretical analyses of average-case complexity (Borgwardt, 1982; Spielman and Teng, 2004, with their "smoothed analysis") confirm that the expected number of pivots is *polynomial*.

> **Theorem 4.10.3** (Smoothed analysis — Spielman & Teng, 2004)**.** *Under small random perturbations of the constraint data, the expected number of Simplex pivots is polynomial in m and d.*

This helps explain why the Simplex method dominates in practice despite its exponential worst case.

Intermezzo — The birth of the Simplex method

George Dantzig developed the Simplex algorithm in 1947 while working for the US Air Force. The problem was to optimise military logistics—scheduling, resource allocation, and deployment planning. Dantzig's method was so effective that it quickly spread to civilian applications: production planning, transportation, diet optimisation, and network design.

The name "Simplex" comes from the geometric *simplex* (a generalisation of a triangle to higher dimensions), though the method doesn't actually operate on simplices. Dantzig himself later admitted the name was somewhat misleading.

In 1979, Leonid Khachiyan proved that LP can be solved in polynomial time using the **ellipsoid method**—a landmark theoretical result. In 1984, Narendra Karmarkar introduced **interior point methods**, which are polynomial-time *and* practical. Despite this, the Simplex method remains the most widely used LP solver for many problem classes: it is fast, it produces vertex solutions (useful for ILP), and it enables warm-starting when solving sequences of related LPs.

4.10.3 Polynomial-time alternatives

Method	Worst-case	Practical
Simplex (Dantzig, 1947)	Exponential	Excellent
Ellipsoid (Khachiyan, 1979)	Polynomial	Poor
Interior point (Karmarkar, 1984)	Polynomial	Excellent

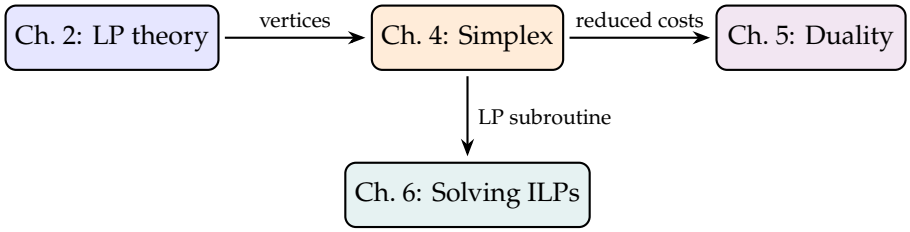
The ellipsoid method was the first proof that LP is in the complexity class P, but it is too slow in practice. Interior point methods are both theoretically efficient and practically fast—they are the main competitor to Simplex in modern solvers. Most commercial solvers (CPLEX, Gurobi, FICO Xpress) implement both and let the user choose.

4.11 Looking Ahead

The Simplex method gives us a practical algorithm for solving LP. But there is more to the story:

- **Chapter 5 (Duality)** will reveal that every LP has a "dual" LP. The reduced costs in the final Simplex tableau have a direct interpretation as dual variables, and the optimality condition $\hat{c}_j \leq 0$ is equivalent to dual feasibility. This connection deepens our understanding of LP and leads to powerful sensitivity analysis.
- **Chapter 6 (Solving ILPs)** will use the Simplex method as a subroutine: branch-and-bound and cutting-plane methods repeatedly solve LP relaxations, often by performing a few Simplex pivots from a "warm start."
- The **dual Simplex method**—which maintains dual feasibility while restoring primal feasibility—is introduced in Chapter 5 and is essential for re-solving after adding cutting planes.

The Simplex method is not just an algorithm—it is the computational engine that powers much of combinatorial optimisation.



Summary & Key Takeaways

- **Standard Form:** Maximize $c^\top x$ subject to $Ax = b, x \geq 0$, where $b \geq 0$.
- **Dictionary & Tableau:** Divides variables into m basic (x_B) and $n - m$ non-basic (x_N) variables.
- **Pivot Selection Rules:**
  - *Entering variable:* Any non-basic variable with a positive coefficient in the z-row.
  - *Leaving variable:* Determined by the Minimum Ratio Test (restricts step size to maintain feasibility).
  - *Bland's Rule:* Chooses variables with the smallest index to prevent cycling under degeneracy.
- **Two-Phase Simplex:** Phase 1 introduces artificial variables to find an initial BFS (or prove infeasibility); Phase 2 optimizes the original objective.

Exercises

Exercise 1 (Converting to standard form). Convert the following LP to standard maximisation form by introducing slack variables. Identify the initial basis and write the corresponding basic feasible solution (BFS).

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 5x_2 \\ \text{subject to} \quad & x_1 \leq 4 \\ & 2x_2 \leq 12 \\ & 3x_1 + 5x_2 \leq 25 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Exercise 2 (Standard form with equality and $\geq$ constraints). Write the following LP in standard form (equalities, non-negative variables, maximise):

$$\begin{aligned} \text{minimize} \quad & 4x_1 + x_2 \\ \text{subject to} \quad & x_1 + x_2 = 5 \\ & 2x_1 - x_2 \geq 3 \\ & x_1, x_2 \geq 0. \end{aligned}$$

State how many variables appear in the standard form and indicate which are slack, surplus, or artificial.

Exercise 3 (Identifying a basis). Consider a system $Ax = b$ with $A \in \mathbb{R}^{3\times 6}$, $x \geq 0$. A proposed basis is $B = \{x_1, x_3, x_5\}$.

1. What condition must the 3×3 submatrix B satisfy for this to be a valid basis?
2. If $B^{-1}b = (2, 0, 7)^\top$, is the corresponding BFS degenerate? Explain.
3. How many basic feasible solutions can exist in total (upper bound)?

Exercise 4 (Unique determination of basic variables). **Prove** the following statement: at any basic feasible solution of a standard-form LP, the values of the basic variables are uniquely determined by the basis matrix B and the right-hand side b.
(*Hint:* Use the fact that B is invertible.)

Exercise 5 (Reading a BFS from a tableau). The following is a simplex tableau for a maximisation LP with variables x_1, x_2, s_1, s_2, s_3:

	x_1	x_2	s_1	s_2	s_3	rhs
s_1	0	1	1	0	0	6
x_1	1	2	0	1	0	8
s_3	0	−1	0	−2	1	2
$\bar{c}$	0	−3	0	4	0	28

1. Identify the current basis and read off the BFS.
2. Is the tableau optimal? Justify using the optimality condition.
3. What is the current objective value?

Exercise 6 (Optimality check). For each tableau below, state whether the current solution is optimal. If not, identify the entering variable using the most-negative reduced cost rule.

(a)

	x_1	x_2	s_1	s_2	rhs
s_1	2	1	1	0	10
s_2	1	3	0	1	12
$\bar{c}$	−5	−4	0	0	0

(b)

	x_1	x_2	s_1	s_2	rhs
x_1	1	0	2	−1	4
x_2	0	1	−1	2	6
$\bar{c}$	0	0	3	1	36

Exercise 7 (Ratio test — entering column given). In the tableau below, x_2 has been chosen as the entering variable. Apply the minimum-ratio test to identify the leaving variable and the pivot element.

	x_1	x_2	s_1	s_2	s_3	rhs
s_1	1	3	1	0	0	12
s_2	2	1	0	1	0	8
s_3	0	2	0	0	1	10
$\bar{c}$	−2	−6	0	0	0	0

Show all ratio computations and state the updated basis after the pivot.

Exercise 8 (Full simplex pivot). Starting from the tableau in Exercise 7, perform the pivot operation:

1. Update the pivot row.
2. Update all other rows (including the objective row).
3. Write the new complete tableau.
4. Read off the new BFS and the new objective value.

Exercise 9 (Two simplex iterations — small LP). Apply the simplex method (most-negative reduced cost rule) for *two* iterations to the LP below. Show the tableau after each pivot.

$$\begin{aligned} \text{maximize} \quad & 2x_1 + 3x_2 \\ \text{subject to} \quad & x_1 + x_2 \le 4 \\ & x_1 + 3x_2 \le 6 \\ & x_1, x_2 \ge 0. \end{aligned}$$

After two pivots, is the solution optimal? If not, what would happen next?

Exercise 10 (Three-variable LP — simplex from scratch). Solve the following LP by the simplex method, showing the full tableau at each iteration. State the optimal solution and optimal value.

$$\begin{aligned} \text{maximize} \quad & 5x_1 + 4x_2 + 3x_3 \\ \text{subject to} \quad & 6x_1 + 4x_2 + 2x_3 \le 240 \\ & 3x_1 + 2x_2 + 5x_3 \le 270 \\ & 5x_1 + 6x_2 + 5x_3 \le 420 \\ & x_1, x_2, x_3 \ge 0. \end{aligned}$$

Exercise 11 (Pivot arithmetic check). Verify that after a simplex pivot with pivot element a_{rs}, the new tableau entry in row $i \neq r$, column j is

$$\bar{a}_{ij} \leftarrow a_{ij} - \frac{a_{is}}{a_{rs}} a_{rj}.$$

Use this formula to redo the pivot of Exercise 8 by hand, verifying at least three entries.

Exercise 12 (Detecting an unbounded LP). Explain the criterion for detecting that an LP is **unbounded** directly from a simplex tableau. Then apply this criterion to the tableau below: is the LP unbounded? If so, exhibit a feasible ray along which the objective tends to $+\infty$.

	x_1	x_2	s_1	s_2	rhs
x_1	1	0	1	-2	3
s_2	0	1	-1	-3	5
$\bar{c}$	0	-4	2	-5	15

Exercise 13 (Constructing an unbounded instance). Construct a two-variable LP (with at least two constraints) that is feasible but unbounded. Draw the feasible region and indicate the direction of unboundedness. Verify algebraically that no finite optimum exists.

Exercise 14 (Degenerate BFS). A standard-form LP has the tableau

	x_1	x_2	s_1	s_2	s_3	rhs
s_1	2	1	1	0	0	0
s_2	1	2	0	1	0	4
s_3	0	1	0	0	1	6
$\bar{c}$	-3	-5	0	0	0	0

1. Identify the current BFS. Is it degenerate? Why?
2. If a pivot is performed with x_2 entering, which row wins the ratio test? What happens to the entering variable's value?
3. Explain informally why degeneracy can cause the simplex method to cycle.

Exercise 15 (True/false: degeneracy and reduced costs). State whether each of the following is **True** or **False**, and give a brief justification or counterexample.

1. "If a reduced cost $\bar{c}_j = 0$ for a non-basic variable, then the current basis is degenerate."
2. "If the current BFS is degenerate, then the next pivot will not improve the objective value."
3. "A non-degenerate LP cannot cycle."
4. "Bland's rule always terminates the simplex method in a finite number of steps."

Exercise 16 (Bland's rule — avoiding cycling). Consider the following three-variable LP, known to induce cycling under the standard most-negative-reduced-cost rule (Beale's example):

$$\begin{aligned} \text{maximize} \quad & \tfrac{3}{4}x_1 - 150x_2 + \tfrac{1}{50}x_3 - 6x_4 \\ \text{subject to} \quad & \tfrac{1}{4}x_1 - 8x_2 - x_3 + 9x_4 \le 0 \\ & \tfrac{1}{2}x_1 - 12x_2 - \tfrac{1}{2}x_3 + 12x_4 \le 0 \\ & x_3 \le 1 \\ & x_1, x_2, x_3, x_4 \ge 0. \end{aligned}$$

Describe Bland's rule precisely. Then show—by listing the sequence of bases—that Bland's rule terminates in at most four pivots on this instance (you may write the tableaux or just state the basis sequences).

Exercise 17 (Bland's rule — small cycling example). Construct a 2×4 standard-form tableau (2 constraints, 4 variables including slacks) that has a degenerate BFS. Show that the standard pivot rule performs a degenerate pivot, and then apply Bland's rule to the same starting tableau and state which variable enters first.

Exercise 18 (Vertices and basic feasible solutions). Consider the LP feasible region defined by $x_1 + x_2 \le 6$, $x_1 \le 4$, $x_2 \le 4$, $x_1, x_2 \ge 0$.

1. List all vertices of the feasible polytope.
2. For each vertex, identify the corresponding basis (which constraints are tight).
3. Indicate which vertex the simplex method visits first if it starts at the origin and uses the most-negative reduced cost rule when maximising $x_1 + 2x_2$.

Exercise 19 (Tracing the simplex path). The feasible region of

$$x_1 + 2x_2 \le 8, \quad 3x_1 + 2x_2 \le 12, \quad x_1, x_2 \ge 0$$

is a quadrilateral.

1. Sketch the feasible region and label all vertices.
2. Solve maximize $3x_1 + x_2$ graphically.
3. Now trace the simplex path from the origin using the most-negative reduced cost rule. List each basis and objective value visited.

Exercise 20 (Adjacent bases and edges). Two bases B_1 and B_2 of a standard-form LP are called *adjacent* if they differ in exactly one element. Explain why an edge of the feasible polytope corresponds to two adjacent basic feasible solutions (or the same BFS if degenerate). Use the LP of Exercise 19 to illustrate with a specific pair of adjacent bases.

Exercise 21 (Reading the optimal tableau). The following is the *final* (optimal) tableau of a maximisation LP with original variables x_1, x_2 and slacks s_1, s_2, s_3:

	x_1	x_2	s_1	s_2	s_3	rhs
x_2	0	1	$\frac{1}{2}$	0	$-\frac{1}{4}$	3
s_2	0	0	-1	1	$\frac{1}{2}$	2
x_1	1	0	$-\frac{1}{2}$	0	$\frac{3}{4}$	5
$\bar{c}$	0	0	2	0	1	40

1. State the optimal solution (x_1^*, x_2^*) and the optimal value.
2. Read off the slack values s_1^*, s_2^*, s_3^*.
3. Which original constraints are tight at the optimum?

Exercise 22 (Dual variables from the optimal tableau). Using the optimal tableau of Exercise 21:

1. Write the formula $y^\top = c_B^\top B^{-1}$ and explain how the reduced costs of the slack variables give the dual variables directly.
2. Read off the dual variables y_1, y_2, y_3 from the tableau.
3. Verify that $y^\top b = z^*$ (strong duality) if $b = (8, 4, 10)^\top$ and $c = (3, 4)^\top$ for the original LP.

Exercise 23 (Reconstructing the original LP). Given the optimal tableau of Exercise 21 and the knowledge that the original LP has two variables (x_1, x_2) and three $\le$ constraints, reconstruct the full original LP.
Hint: The columns of the identity matrix in the initial tableau become B^{-1} in the final tableau; use this to recover B^{-1} and then $b = B \cdot x_B^*$.

Exercise 24 (Phase I formulation). Set up the Phase I LP for the following problem. Introduce artificial variables as needed, write the Phase I objective, and state what conclusion can be drawn from the Phase I optimal value.

$$\begin{aligned} \text{maximize} \quad & 2x_1 + x_2 \\ \text{subject to} \quad & x_1 + x_2 = 4 \\ & x_1 - x_2 \le 2 \\ & x_1, x_2 \ge 0. \end{aligned}$$

Exercise 25 (Phase I — full execution). Execute Phase I for the LP

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 2x_2 \\ \text{subject to} \quad & x_1 + 2x_2 = 6 \\ & 2x_1 + x_2 = 8 \\ & x_1, x_2 \ge 0. \end{aligned}$$

1. Introduce artificial variables a_1, a_2 and write the Phase I tableau.

2. Apply the simplex method to Phase I until the artificials leave the basis (show each tableau).
3. Use the resulting basis to initialise Phase II and find the optimal solution.

Exercise 26 (Phase I detects infeasibility). Apply Phase I to the following LP:

$$\begin{aligned} \text{maximize} \quad & x_1 + x_2 \\ \text{subject to} \quad & x_1 + x_2 = 5 \\ & x_1 + x_2 = 7 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Show that the Phase I optimal value is strictly positive, and explain why this implies the original LP is infeasible.

Exercise 27 (Artificial variable remains in basis). After Phase I, suppose an artificial variable a_i is still in the basis at value zero. Explain two strategies for handling this situation before beginning Phase II:

1. Pivoting the artificial out of the basis.
2. Dropping the corresponding row (detecting redundancy).

Give a small 2×3 example to illustrate the first strategy.

Exercise 28 (Big-M formulation). Convert the Phase I problem of Exercise 24 to a Big-M formulation. Write the penalised objective explicitly, and set up the initial Big-M tableau (with M as a symbolic large constant).

Exercise 29 (Big-M — full solve). Solve the following LP using the Big-M method. Show the initial tableau (with M terms) and all subsequent tableaux.

$$\begin{aligned} \text{maximize} \quad & 2x_1 + 3x_2 \\ \text{subject to} \quad & x_1 + x_2 = 4 \\ & x_1 + 2x_2 \leq 6 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Exercise 30 (Comparing Phase I and Big-M). List two *theoretical* and two *practical* differences between the Two-Phase method and the Big-M method. Under what circumstances might Big-M give incorrect results in floating-point arithmetic?

Exercise 31 (Furniture workshop — Phase II continuation). Recall the furniture workshop LP from the chapter:

$$\begin{aligned} \text{maximize} \quad & 5x_1 + 4x_2 \\ \text{subject to} \quad & 6x_1 + 4x_2 \leq 24 \\ & x_1 + 2x_2 \leq 6 \\ & x_1, x_2 \geq 0. \end{aligned}$$

The simplex method was shown to reach optimum at $(x_1, x_2) = (3, \frac{3}{2})$ with $z^* = 21$.

1. Confirm this by writing the initial tableau, applying two pivots, and verifying the final tableau is optimal.
2. Add the constraint $x_1 + x_2 \leq 4$ and redo the simplex method. Does the optimal solution change?

Exercise 32 (Furniture workshop — sensitivity). Using the optimal tableau of the furniture workshop LP (without the extra constraint), answer the following:

1. By how much can the profit coefficient of x_1 increase before the current basis becomes sub-optimal? (*Hint:* re-examine the reduced costs as a function of c_1.)
2. By how much can the RHS of the first constraint ($b_1 = 24$) decrease while the current basis remains feasible?

Exercise 33 (Reduced cost formula derivation). Starting from the standard-form LP $\max c^\top x$ s.t. $Ax = b$, $x \geq 0$, and a feasible basis B:

1. Derive the formula $\bar{c}_N = c_N - c_B^\top B^{-1} N$ for the reduced costs of non-basic variables.
2. Show that the objective can be written as $z = c_B^\top B^{-1} b + \bar{c}_N^\top x_N$.
3. Conclude that $\bar{c}_N \leq 0$ (componentwise) is sufficient for optimality.

Exercise 34 (Optimality condition — sufficiency and necessity).

1. Show that $\bar{c}_j \leq 0$ for all non-basic j is a *sufficient* condition for optimality at the current BFS.
2. Show that it is also *necessary* if the LP is non-degenerate.
3. Give an example where $\bar{c}_j = 0$ for some non-basic j at an optimal BFS, yet the optimum is unique.

Exercise 35 (Multiple optimal solutions). Suppose the final simplex tableau shows $\bar{c}_j = 0$ for a non-basic variable x_j.

1. What does this imply about the existence of multiple optimal solutions?
2. Perform one more (degenerate) pivot to find a second optimal BFS. Use the tableau

	x_1	x_2	s_1	s_2	rhs
x_1	1	2	1	0	6
s_2	0	1	−1	1	2
$\bar{c}$	0	0	3	0	18

3. Express all optimal solutions as a convex combination of the two optimal BFSs.

Exercise 36 (Worst-case complexity of simplex). The Klee–Minty cube shows that the simplex method can visit $2^n - 1$ vertices on an n-dimensional LP.

1. Write the Klee–Minty LP for $n = 3$ (three variables, three constraints after conversion).
2. How many vertices does the feasible polytope have?
3. Explain why this does *not* contradict the practical efficiency of the simplex method.

Exercise 37 (Polynomial vs. exponential algorithms).

1. State the time complexity of the simplex method in the worst case and compare it to the ellipsoid method and interior-point methods.
2. Despite its exponential worst case, why is the simplex method preferred in practice for most LP instances?
3. What is the complexity class of LP as a decision problem, and who first proved polynomial solvability?

Exercise 38 (Diet problem — simplex setup). A diet must supply at least 2000 kcal and 50 g of protein per day. Food A provides 400 kcal and 20 g protein per unit, costs \$2. Food B provides 300 kcal and 10 g protein per unit, costs \$1.50.

1. Formulate as a minimisation LP.
2. Convert to standard maximisation form (negate objective, convert $\geq$ constraints).

3. Set up the Two-Phase simplex tableau.

Exercise 39 (Transport LP — one simplex pivot). A transportation LP has been partially solved; the current tableau is:

	x_{11}	x_{12}	x_{21}	x_{22}	s_1	rhs
s_1	1	1	0	0	1	50
x_{21}	0	0	1	1	0	30
$\bar{c}$	-2	-3	0	-1	0	90

Identify the entering variable, leaving variable, and perform one pivot. State the new objective value.

Exercise 40 (Production planning — three products). A factory makes products P, Q, R with profit per unit $8, 5, 4$. Resource constraints: $2P+Q+R \le 14$, $P+2Q+R \le 14$, $P+Q+2R \le 14$, $P, Q, R \ge 0$.

1. Set up the initial simplex tableau.
2. Perform the first pivot (most-negative reduced cost rule).
3. After the first pivot, is the solution optimal? Identify the next entering variable if not.

Exercise 41 (Maximisation with four constraints). Solve by the simplex method:

$$\begin{aligned} \text{maximize} \quad & x_1 + 2x_2 + x_3 \\ \text{subject to} \quad & x_1 + x_2 \le 40 \\ & 2x_1 + x_3 \le 60 \\ & x_2 + x_3 \le 30 \\ & x_1, x_2, x_3 \ge 0. \end{aligned}$$

List the basis at each iteration and the objective value.

Exercise 42 (Entering variable tie-breaking). The following tableau has two non-basic variables with the same (most negative) reduced cost:

	x_1	x_2	s_1	s_2	rhs
s_1	3	2	1	0	12
s_2	1	4	0	1	16
$\bar{c}$	-4	-4	0	0	0

1. Apply Bland's rule to break the tie.
2. Apply the largest-coefficient rule (ties broken by smallest index).
3. Do both choices lead to the same optimal solution after the same number of pivots? Investigate.

Exercise 43 (Ratio test tie — perturbation method). When the ratio test yields a tie (two rows give the same minimum ratio), degeneracy occurs. Describe the *perturbation method*: replace b by $b + \epsilon$ for a small $\epsilon > 0$ (formally as a lexicographic ordering). Apply this idea to break the tie in the following tableau when x_1 enters:

	x_1	x_2	s_1	s_2	rhs
s_1	2	1	1	0	4
s_2	2	3	0	1	4
$\bar{c}$	-5	-3	0	0	0

Exercise 44 (Feasibility and the simplex method). An LP is given with three equality constraints and five variables (after adding slacks). After Phase I, the

simplex method reports that the minimum of $\sum a_i$ (sum of artificials) equals 0 but an artificial variable remains in the basis at value 0.

1. Does this mean the original LP is feasible? Why or why not?
2. Give an example where this happens and the redundant constraint can be dropped.
3. Give an example where this happens but *no* constraint is redundant (inconsistent sub-system of rank less than expected).

Exercise 45 (Sensitivity: objective coefficient ranging). Let the optimal tableau of a maximisation LP be given, with objective coefficients $c = (c_1, c_2)$. The basis is $B = \{x_1, s_2\}$ with reduced costs $\bar{c}_2 = c_2 - 3$ and $\bar{c}_{s_1} = 2$.

1. For what range of c_2 does the current basis remain optimal?
2. If c_1 increases by Δ, derive the condition on Δ for optimality to be preserved, in terms of the tableau entries.

Exercise 46 (Sensitivity: RHS ranging). For the optimal tableau with basis B, the current basic solution is $x_B = B^{-1}b$. If b_1 changes to $b_1 + \delta$:

1. Write the new basic solution in terms of the first column of B^{-1}.
2. State the condition on δ for the current basis to remain feasible.
3. Apply this to Exercise 21: for what range of δ can b_1 vary while keeping the same optimal basis?

Exercise 47 (Cycling: explicit three-pivot cycle). Show that the following sequence of bases forms a *cycle* under a specific (non-Bland) pivot rule, i.e., the simplex method returns to a previously visited basis without improving the objective. Construct a 3×6 tableau (3 constraints, 6 variables including slacks) that exhibits cycling in two complete rounds of three pivots each, and explain which pivot rule is being used. Then state why Bland's rule breaks this cycle.

Exercise 48 (Recap: simplex algorithm correctness).

1. State (without proof) the three key facts that together ensure the simplex method is correct: (i) the objective is non-decreasing at each step; (ii) the number of bases is finite; (iii) an optimal BFS satisfies the optimality conditions.
2. Identify which fact fails when cycling occurs and how Bland's rule restores correctness.
3. Prove (i): show that if $\bar{c}_s > 0$ and no degeneracy occurs, the objective strictly increases after the pivot.

Exercise 49 (Simplex on a minimisation LP). Apply the simplex method to the following minimisation LP by negating the objective and solving as a maximisation problem. Show the initial tableau, all intermediate tableaux, and the final optimal solution and minimum value.

$$\begin{aligned} \text{minimize} \quad & x_1 + 2x_2 + 3x_3 \\ \text{subject to} \quad & x_1 + x_2 \le 10 \\ & x_2 + x_3 \le 8 \\ & x_1 + x_3 \le 7 \\ & x_1, x_2, x_3 \ge 0. \end{aligned}$$

Exercise 50 (Characterising all optimal bases). A maximisation LP has the

final tableau

	x_1	x_2	s_1	s_2	s_3	rhs
x_1	1	0	2	0	−1	4
s_2	0	0	−1	1	1	3
x_2	0	1	1	0	0	5
$\bar{c}$	0	0	0	0	2	27

1. Identify all non-basic variables with $\bar{c}_j = 0$ and those with $\bar{c}_j > 0$.
2. Is the optimal solution unique or are there multiple optima? Justify.
3. If there are multiple optimal solutions, describe the optimal face of the feasible polytope.

Exercise 51 (Eta-factorisation and the basis update). When the simplex method moves from basis B to an adjacent basis $B' = (B \setminus \{x_r\}) \cup \{x_s\}$, the new basis inverse satisfies $B'^{-1} = E \cdot B^{-1}$, where E is an elementary matrix.

1. Describe the structure of E: which column differs from the identity, and what does it contain?
2. Verify one pivot step from Exercise 8 using this formula.
3. Explain why eta-factorisations are useful in large-scale implementations.

Exercise 52 (Degenerate optimum). Construct a two-variable LP (with three $\leq$ constraints) whose unique optimal BFS is degenerate (at least one slack is zero even though the constraint is not binding at the optimum in the usual sense). Draw the feasible region, mark the optimal vertex, and identify the degenerate basis. Write the simplex tableau at that vertex and confirm that all reduced costs satisfy the optimality condition.

Exercise 53 (BFSs and extreme points).

1. Define an *extreme point* (vertex) of a convex set.
2. Prove that every basic feasible solution of a standard-form LP is an extreme point of the feasible polyhedron $P = \{x \geq 0 : Ax = b\}$.
3. Show by example (a 2×4 system) that every extreme point of P corresponds to at least one BFS.

Exercise 54 (Geometric interpretation of a pivot). Consider the LP maximize $2x_1 + x_2$ subject to $x_1 + x_2 \leq 6$, $x_1 \leq 4$, $x_2 \leq 5$, $x_1, x_2 \geq 0$.

1. Sketch the feasible region and label all six vertices.
2. Starting at the origin, apply one simplex pivot and identify the new vertex on the sketch.
3. Explain why the simplex pivot corresponds to moving along an edge of the polytope to an adjacent vertex.

Exercise 55 (Choosing the Big-M constant).

1. Explain why M must be large enough relative to the problem data for the Big-M method to correctly identify an infeasible LP.
2. Given that all entries of A are bounded in absolute value by K and all entries of b by R, suggest a safe lower bound for M in terms of K, R, and the problem dimensions.
3. Construct a small numerical example where a finite but insufficiently large M causes the Big-M method to return a wrong (non-infeasibility) answer.

CHAPTER 5

Duality & Valid Inequalities

This chapter reveals the "other side" of every LP: the dual problem. Every LP has a twin, and the relationship between the two is one of the deepest ideas in optimisation.

In chapter 4 we solved the furniture workshop LP (theorem 1.8.1) with the Simplex method, finding the optimum $(x_1, x_2) = (5, 6)$ with profit $z^* = 450$. At the end, every reduced cost was non-positive, and the Simplex declared victory. But a nagging question remains: *how can we be sure 450 really is the maximum?*

One way would be to check all vertices—but that is combinatorially expensive. A much better way is to exhibit an *upper bound* on the profit that happens to *equal* 450. If we can prove that no feasible solution can achieve more than 450, and we have a feasible solution achieving exactly 450, then we are done.

This chapter develops a systematic way to produce such upper bounds. The idea leads us to **valid inequalities**, **Farkas' lemma**, and ultimately to the **dual LP**—a companion optimisation problem that always accompanies the original ("primal") LP.

Road map.

1. Bounding the optimum: valid inequalities (section 5.1).
2. Direct and indirect inequalities (section 5.2).
3. Farkas' lemma: characterising all valid inequalities (section 5.3).
4. Deriving the dual LP (section 5.4).
5. Primal-dual pairs: the general rules (section 5.5).
6. Weak duality (section 5.6).
7. Strong duality (section 5.7).
8. Complementary slackness (section 5.8).
9. Economic interpretation: shadow prices (section 5.9).
10. Worked example: the furniture workshop dual (section 5.10).

5.1 Bounding the Optimum: Valid Inequalities

The key question: can we prove an upper bound on the objective, using only the constraints?

Let's start with a concrete question about our furniture workshop. The primal LP in canonical form (theorem 1.8.1) is:

$$\begin{aligned} \text{maximize} \quad & 30\,x_1 + 50\,x_2 \\ \text{subject to} \quad & 2\,x_1 + 5\,x_2 \le 40 \quad \text{(wood)} \\ & 4\,x_1 + 2\,x_2 \le 32 \quad \text{(labour)} \\ & x_1, x_2 \ge 0. \end{aligned} \tag{5.1}$$

We claim the optimum is 450. Can we prove that no feasible point gives a higher profit?

Here is a clever trick. Take the wood constraint and multiply it by 10:

$$10 \cdot (2\,x_1 + 5\,x_2) \le 10 \cdot 40 \quad \Longrightarrow \quad 20\,x_1 + 50\,x_2 \le 400.$$

This is an inequality that every feasible point must satisfy. Notice that $20\,x_1 + 50\,x_2$ is "almost" the objective function $30\,x_1 + 50\,x_2$, but with a smaller coefficient on x_1. Since $x_1 \ge 0$, we have $30\,x_1 + 50\,x_2 \ge 20\,x_1 + 50\,x_2$, so this particular combination does not directly give an upper bound on the objective—it goes the wrong way.

We need the combination to dominate *the objective coefficients.*

Let's try a different approach: combine *both* constraints. Multiply the wood constraint by u_1 and the labour constraint by u_2, where $u_1, u_2 \ge 0$. Adding:

$$(2u_1 + 4u_2)\,x_1 + (5u_1 + 2u_2)\,x_2 \;\le\; 40\,u_1 + 32\,u_2. \tag{5.2}$$

If the left-hand-side coefficients are *at least as large* as the objective coefficients—i.e. $2u_1 + 4u_2 \ge 30$ and $5u_1 + 2u_2 \ge 50$—then for every feasible point (x_1, x_2) with $x_1, x_2 \ge 0$:

$$30\,x_1 + 50\,x_2 \;\le\; (2u_1 + 4u_2)\,x_1 + (5u_1 + 2u_2)\,x_2 \;\le\; 40\,u_1 + 32\,u_2.$$

The right-hand side $40u_1 + 32u_2$ is an **upper bound** on the profit!

Example 5.1.1 (A first upper bound)**.** Try $u_1 = 10, u_2 = 5$. Then:

- $2(10) + 4(5) = 40 \ge 30$ ✓
- $5(10) + 2(5) = 60 \ge 50$ ✓

Upper bound: $40(10) + 32(5) = 400 + 160 = 560$.

So profit ≤ 560. That's true, but not very tight—we know the optimum is 450. Can we do better?

Example 5.1.2 (A tighter upper bound)**.** Try $u_1 = 10, u_2 = \frac{5}{2}$. Then:

- $2(10) + 4(5/2) = 20 + 10 = 30 \ge 30$ ✓
- $5(10) + 2(5/2) = 50 + 5 = 55 \ge 50$ ✓

Upper bound: $40(10) + 32(5/2) = 400 + 80 = 480$.

Better! Profit ≤ 480. But still above 450. The natural question is: what is the *tightest* upper bound we can get this way?

Finding the tightest upper bound is itself an optimisation problem— the dual *LP.*

The tightest upper bound means: *minimise* $40u_1 + 32u_2$ subject to the domination conditions $2u_1 + 4u_2 \ge 30$, $5u_1 + 2u_2 \ge 50$, and $u_1, u_2 \ge 0$. That's a new LP! We will formalise this observation shortly. But first, let's develop the general theory.

5.2 Direct and Indirect Inequalities

We now work in standard form: equalities and non-negativity.

Let us state the general setup. Consider a polyhedron defined in **standard form**:

$$P = \{x \in \mathbb{R}^n : Ax = b,\ x \ge 0\}, \tag{5.3}$$

where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.

Definition 5.2.1 (Valid inequality)**.** An inequality $\alpha^\top x \le \beta$ (with $\alpha \in \mathbb{R}^n$, $\beta \in \mathbb{R}$) is **valid** for the polyhedron P if it is satisfied by every point in P:

$$x \in P \implies \alpha^\top x \le \beta.$$

In plain language, a valid inequality is a "redundant" constraint: you can add it to the system without changing the feasible set, because every feasible point already satisfies it. This sounds useless—but it is exactly what we need to prove upper bounds on the objective.

If $\alpha = c$ (the objective vector) and the inequality $c^\top x \le \beta$ is valid, then β is an upper bound on the optimum.

5.2.1 Direct inequalities

The simplest way to produce a valid inequality is by taking a linear combination of the constraints.

Definition 5.2.2 (Direct inequality)**.** Let $u \in \mathbb{R}^m$ be any vector of multipliers. Since $Ax = b$ for all $x \in P$, we have $u^\top A\, x = u^\top b$ for all $x \in P$.

A **direct inequality** is any inequality of the form

$$u^\top A\, x \le u^\top b$$

obtained from $u^\top(Ax) = u^\top b$ by simply reading the equality as an inequality. Since the left- and right-hand sides are *equal* on P, the inequality is trivially valid.

Of course, $u^\top A\, x = u^\top b$ is stronger (it's an equality!). But we state it as a "$\le$" because we want to *compare* different valid inequalities in a uniform way, and the relaxation step below needs the $\le$ form.

"Direct" = derived purely from the constraints, with no weakening.

Example 5.2.3 (Direct inequalities from the furniture LP)**.** Consider the standard form of the furniture LP (eq. (4.2)):

$$\underbrace{\begin{pmatrix} 2 & 5 & 1 & 0 \\ 4 & 2 & 0 & 1 \end{pmatrix}}_{A} \begin{pmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \end{pmatrix} = \underbrace{\begin{pmatrix} 40 \\ 32 \end{pmatrix}}_{b}.$$

With $u = (u_1, u_2)^\top$, the direct inequality is:

$$(2u_1 + 4u_2)\, x_1 + (5u_1 + 2u_2)\, x_2 + u_1\, s_1 + u_2\, s_2 \le 40\, u_1 + 32\, u_2.$$

For $u = (1, 0)$: this gives $2x_1 + 5x_2 + s_1 \le 40$ (just the first constraint as a $\le$, but it is actually an equality on P).

For $u = (3, 2)$: $14x_1 + 19x_2 + 3s_1 + 2s_2 \le 184$. Still valid, because it's the sum of 3 times the first constraint and 2 times the second.

5.2.2 Indirect inequalities (relaxations)

Direct inequalities come from exact linear combinations of constraints. We can weaken them to produce a richer family.

Definition 5.2.4 (Indirect inequality / relaxation)**.** An **indirect inequality** (or **relaxation**) of a valid inequality $\alpha^\top x \le \beta$ is any inequality $\alpha'^\top x \le \beta'$ such that

$$\alpha'_j \le \alpha_j \quad \text{for all } j, \qquad \beta' \ge \beta.$$

The idea is simple: if we decrease a coefficient on the left or increase the right-hand side, the inequality becomes *weaker*—easier to satisfy—so it remains valid for all $x \ge 0$.

Decrease the LHS or increase the RHS: the inequality gets weaker but stays valid.

Example 5.2.5 (Relaxing a direct inequality)**.** From theorem 5.2.3 with $u = (10, 0)$, the direct inequality gives:

$$20\,x_1 + 50\,x_2 + 10\,s_1 \le 400.$$

Since $s_1 \ge 0$, we can drop the $10\,s_1$ term (decrease the coefficient from 10 to 0):

$$20\,x_1 + 50\,x_2 \le 400.$$

This is a **relaxation**—still valid, but weaker.

We can relax further by decreasing the x_1 coefficient from 20 to 0: $50\,x_2 \le 400$, i.e. $x_2 \le 8$. Indeed, every feasible point in the furniture LP has $x_2 \le 8$.

The remarkable fact—which we prove next—is that *every* valid inequality for P can be obtained as a relaxation of some direct inequality.

5.3 Farkas' Lemma

We have seen that direct inequalities (and their relaxations) produce valid inequalities. But do they produce *all* valid inequalities? The answer is yes, and this is the content of Farkas' lemma.

Farkas' lemma is the theoretical cornerstone: it tells us exactly which inequalities are valid.

Theorem 5.3.1 (Farkas' Lemma — valid-inequality form)**.** *Let $P = \{x \in \mathbb{R}^n : Ax = b,\ x \ge 0\}$ with $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$. The inequality $\alpha^\top x \le \beta$ is valid for P if and only if there exists a vector $u \in \mathbb{R}^m$ such that*

$$u^\top A \ge \alpha^\top \qquad \text{and} \qquad u^\top b \le \beta.$$

Let's unpack this. The condition says: the inequality $\alpha^\top x \le \beta$ holds for all feasible x if and only if we can "certify" it using multipliers u. The certificate is:

1. Take the linear combination $u^\top(Ax) = u^\top b$.
2. Check that $u^\top A \ge \alpha^\top$ component-wise (so the direct inequality dominates the target on non-negative x).
3. Check that $u^\top b \le \beta$ (the RHS is at most β).

In other words, $\alpha^\top x \le \beta$ is a relaxation of the direct inequality $u^\top A\,x \le u^\top b$.

The "only if" direction is the deep part. It says that if an inequality holds for every feasible point, then there *must* exist multipliers certifying this. There is no "magical" valid inequality that cannot be derived from the constraints—every valid inequality has a proof.

Farkas: every valid inequality has a proof *from the constraints.*

Formal details — Proof sketch of Farkas' Lemma

The key idea is that multipliers u serve as a *certificate*: they let us "price" the constraints to dominate the target inequality, and the Simplex optimality conditions guarantee such a certificate always exists when the inequality is valid.

($\Leftarrow$) Certificate $\Rightarrow$ valid inequality.

1. We assume multipliers u exist with $u^\top A \geq \alpha^\top$ and $u^\top b \leq \beta$ because these are the two conditions that make u a valid certificate.
2. We bound $\alpha^\top x$ from above because $u^\top A \geq \alpha^\top$ and $x \geq 0$ give $\alpha^\top x \leq (u^\top A)\, x = u^\top (Ax)$.
3. We complete the chain because $Ax = b$ and $u^\top b \leq \beta$ yield $u^\top (Ax) = u^\top b \leq \beta$.

($\Rightarrow$) Valid inequality $\Rightarrow$ certificate exists.

1. We consider the LP maximize $\alpha^\top x$ s.t. $Ax = b$, $x \geq 0$ because if $\alpha^\top x \leq \beta$ is valid, this LP's optimal value is at most β (or the LP is infeasible, making the inequality trivially valid).
2. We apply the Simplex method to find an optimum x^* with basis $\mathcal{B}$ because at the optimal tableau the Simplex produces dual multipliers $u^{*\top} = \alpha_{\mathcal{B}}^\top A_{\mathcal{B}}^{-1}$.
3. We verify the certificate because the optimality condition gives $\hat{c}_j = \alpha_j - u^{*\top} A_j \leq 0$ for all j, which rearranges to $u^{*\top} A \geq \alpha^\top$; and the optimal value satisfies $u^{*\top} b = z^* \leq \beta$.

A complete proof without assuming LP duality uses the separating hyperplane theorem, but the Simplex-based argument above is sufficient for our purposes.

Thus, every valid inequality has a linear-combination certificate derived from the constraints.

Intermezzo — Gyula Farkas (1847–1930)

Farkas was a Hungarian mathematician and physicist. His lemma, published in 1902, is one of the foundational results of linear optimisation. It can be viewed as a "theorem of the alternative": either a system of linear inequalities has a solution, or there exists a certificate (a linear combination of the constraints) proving it has none. Many equivalent forms exist in the literature; we use the one most directly connected to LP duality.

5.4 Deriving the Dual LP

We now have all the ingredients to derive the dual LP from first principles. The argument is natural and—once you see it—almost inevitable.

The dual LP is the problem of finding the tightest upper bound.

5.4.1 The question

Consider the primal LP in standard form:

$$(P) \qquad \text{maximize } c^\top x \qquad \text{subject to} \quad Ax = b, \quad x \geq 0. \tag{5.4}$$

We want to bound the optimal value from above. By Farkas' lemma (theorem 5.3.1), the inequality $c^\top x \leq \beta$ is valid for $P = \{x : Ax = b,\, x \geq 0\}$ if and only if there exist multipliers $u \in \mathbb{R}^m$ with

$$u^\top A \geq c^\top \qquad \text{and} \qquad u^\top b \leq \beta.$$

Given such u, the value $\beta = u^\top b$ is an upper bound on $c^\top x$ for all feasible x.

5.4.2 The tightest bound

The tightest such upper bound is:

$$\min \{u^\top b : u^\top A \geq c^\top\}.$$

Since u comes from $\mathbb{R}^m$ (no sign restriction from the equality constraints), this is itself a linear programme. Rewriting in column form (transposing so that u becomes the variable vector), we get:

Definition 5.4.1 (Dual LP — standard form primal)**.** Given the primal LP (P): maximize $c^\top x$ s.t. $Ax = b$, $x \geq 0$, the **dual** LP is

$$(D) \qquad \text{minimize } b^\top u \qquad \text{subject to } A^\top u \geq c, \quad u \in \mathbb{R}^m \text{ (unrestricted).} \tag{5.5}$$

Notice: u is unrestricted because the primal constraints are equalities.

Let's make sure the dimensions make sense. The primal has n variables and m constraints (equalities). The dual has m variables $(u_1, \ldots, u_m)$ and n constraints (one for each primal variable). The cost vectors and constraint matrices are "transposed": the primal objective c becomes the dual RHS, and the primal RHS b becomes the dual objective.

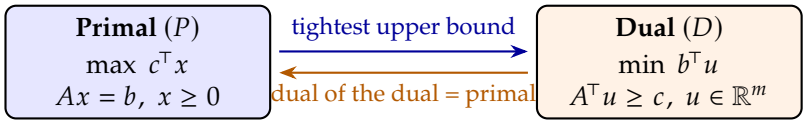


Figure 5.1: The primal-dual relationship. The dual seeks the tightest upper bound on the primal; the dual of the dual recovers the primal.

Example 5.4.2 (Dual of the furniture workshop LP)**.** The standard-form primal (eq. (4.2)) has:

$$A = \begin{pmatrix} 2 & 5 & 1 & 0 \\ 4 & 2 & 0 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 40 \\ 32 \end{pmatrix}, \quad c = \begin{pmatrix} 30 \\ 50 \\ 0 \\ 0 \end{pmatrix}.$$

The dual (D): minimize $40\,u_1 + 32\,u_2$ subject to $A^\top u \geq c$, i.e.:

$$\begin{aligned} \text{minimize} \quad & 40\,u_1 + 32\,u_2 \\ \text{subject to} \quad & 2\,u_1 + 4\,u_2 \geq 30 \\ & 5\,u_1 + 2\,u_2 \geq 50 \\ & u_1 \geq 0 \\ & u_2 \geq 0 \\ & u_1, u_2 \text{ unrestricted.} \end{aligned} \tag{5.6}$$

The third and fourth constraints say $u_1 \geq 0$, $u_2 \geq 0$, but we also said u is unrestricted. There's no contradiction: the constraints $u_1 \geq 0$ and $u_2 \geq 0$ come from the rows of $A^\top u \geq c$ corresponding to the slack variables s_1, s_2 (whose objective coefficients are 0). The "unrestricted" statement means

there is no *additional* sign constraint on u beyond what $A^\top u \geq c$ imposes. In this case, the derived constraints already force $u \geq 0$.

Deriving the dual from canonical form is cleaner—no slack variables to worry about.

The standard-form derivation works perfectly but involves the slack variables. In practice, it is cleaner to derive the dual directly from the *canonical* form. We develop systematic rules for this next.

5.5 Primal-Dual Pairs: The General Rules

Mechanical rules: each primal feature (constraint type, variable sign) maps to a dual feature.

In practice, we rarely convert to standard form just to write the dual. Instead, we apply a set of mechanical rules that map each feature of the primal to a corresponding feature of the dual.

5.5.1 The correspondence table

Primal (max)	⟷	Dual (min)
PRIMAL CONSTRAINT		**DUAL VARIABLE**
$\leq$ constraint	⟷	$u_i \geq 0$
$=$ constraint	⟷	u_i unrestricted
$\geq$ constraint	⟷	$u_i \leq 0$
PRIMAL VARIABLE		**DUAL CONSTRAINT**
$x_j \geq 0$	⟷	$\geq$ constraint
x_j unrestricted	⟷	$=$ constraint
$x_j \leq 0$	⟷	$\leq$ constraint
objective: $c^\top x$	⟷	*objective:* $b^\top u$

Figure 5.2: The primal-dual correspondence rules. Each primal constraint maps to a dual variable (and its sign), and each primal variable maps to a dual constraint (and its type). The arrows are bidirectional: applying them twice recovers the original problem.

The logic behind these rules comes from Farkas' lemma and the derivation in section 5.4:

- A $\leq$ constraint in the primal contributes a non-negative slack. When we multiply by u_i, the slack's contribution to the bound is $u_i \cdot s_i \geq 0$ only if $u_i \geq 0$. Hence: $\leq$ constraint $\leftrightarrow u_i \geq 0$.
- An equality constraint has no slack—the multiplier is free (unrestricted).
- A $\geq$ constraint can be written as $\leq$ after flipping the sign, giving $u_i \leq 0$.
- A non-negative primal variable $x_j \geq 0$ allows the domination $u^\top A_j \geq c_j$ to hold as a "$\geq$" inequality. Hence: $x_j \geq 0 \leftrightarrow \geq$ constraint in the dual.
- An unrestricted primal variable forces *equality* in the dual constraint: if $u^\top A_j > c_j$, we could exploit $x_j \to -\infty$ to violate the bound; if $u^\top A_j < c_j$, we could exploit $x_j \to +\infty$. So we need $u^\top A_j = c_j$.

Theorem 5.5.1 (Dual of the dual). *The dual of the dual LP is the primal LP.*

This follows from the correspondence rules: applying them twice brings us back to the original problem. This symmetry is fundamental—neither problem is "more natural" than the other.

Duality is symmetric: the relationship is perfectly reciprocal.

5.5.2 Canonical form shortcut

The most common case in practice is the **canonical form**:

Primal (canonical max)	$\longleftrightarrow$	**Dual (canonical min)**
maximize $c^\top x$	$\longleftrightarrow$	minimize $b^\top u$
$Ax \le b$	$\longleftrightarrow$	$A^\top u \ge c$
$x \ge 0$	$\longleftrightarrow$	$u \ge 0$

This is very clean: all $\le$ constraints in the primal give $u \ge 0$ in the dual, and all $x_j \ge 0$ give $\ge$ constraints in the dual.

Example 5.5.2 (Furniture workshop — canonical-form dual)**.** Starting from the canonical primal (5.1):

$$\begin{aligned} \text{maximize} \quad & 30\,x_1 + 50\,x_2 \\ \text{subject to} \quad & 2\,x_1 + 5\,x_2 \le 40 \\ & 4\,x_1 + 2\,x_2 \le 32 \\ & x_1, x_2 \ge 0, \end{aligned}$$

both constraints are $\le$ and both variables are ≥ 0, so we are in canonical form. The dual is:

$$\begin{aligned} \text{minimize} \quad & 40\,u_1 + 32\,u_2 \\ \text{subject to} \quad & 2\,u_1 + 4\,u_2 \ge 30 \quad \text{(chair bound)} \\ & 5\,u_1 + 2\,u_2 \ge 50 \quad \text{(table bound)} \\ & u_1, u_2 \ge 0. \end{aligned} \tag{5.7}$$

This is exactly the "tightest upper bound" problem we discovered informally in section 5.1!

Example 5.5.3 (A mixed-form LP)**.** Consider the primal:

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 2x_2 + x_3 \\ \text{subject to} \quad & x_1 + x_2 + x_3 \le 10 \\ & 2x_1 - x_2 = 4 \\ & x_1 \ge 0, \quad x_2 \ge 0, \quad x_3 \text{ unrestricted.} \end{aligned}$$

Applying the rules from fig. 5.2:

- First constraint ($\le$) $\Rightarrow u_1 \ge 0$.
- Second constraint ($=$) $\Rightarrow u_2$ unrestricted.
- $x_1 \ge 0 \Rightarrow$ dual constraint $u_1 + 2u_2 \ge 3$ (type $\ge$).
- $x_2 \ge 0 \Rightarrow$ dual constraint $u_1 - u_2 \ge 2$ (type $\ge$).
- x_3 unrestricted $\Rightarrow$ dual constraint $u_1 = 1$ (type $=$).

The dual is:

$$\begin{aligned} \text{minimize} \quad & 10\,u_1 + 4\,u_2 \\ \text{subject to} \quad & u_1 + 2u_2 \geq 3 \\ & u_1 - u_2 \geq 2 \\ & u_1 = 1 \\ & u_1 \geq 0, \quad u_2 \text{ unrestricted.} \end{aligned}$$

From the third constraint, $u_1 = 1$. Then $1 + 2u_2 \geq 3$ gives $u_2 \geq 1$, and $1 - u_2 \geq 2$ gives $u_2 \leq -1$. These are contradictory: the **dual is infeasible**. By weak duality (section 5.6), this means the primal is either infeasible or unbounded.

5.6 Weak Duality

Weak duality: any dual-feasible solution gives an upper bound on the primal optimum.

Now that we have both the primal and dual LPs, what is the relationship between their optimal values?

Theorem 5.6.1 (Weak Duality). *Let $\bar{x}$ be any primal-feasible solution and $\bar{u}$ any dual-feasible solution. Then*

$$c^\top \bar{x} \;\leq\; b^\top \bar{u}.$$

In words: every feasible dual solution provides an upper bound on every feasible primal objective value.

Formal details — Proof of Weak Duality

The key idea is that the dual multipliers $\bar{u} \geq 0$ let us "price" the primal constraints and sandwich the primal objective between two quantities that can be compared.

We prove it for the canonical form (all $\leq$, all $x \geq 0$, all $u \geq 0$); the general case is analogous.

1. We record feasibility: $\bar{x}$ is primal-feasible ($A\bar{x} \leq b$, $\bar{x} \geq 0$), and $\bar{u}$ is dual-feasible ($A^\top \bar{u} \geq c$, $\bar{u} \geq 0$).
2. We bound $c^\top \bar{x}$ from above because $A^\top \bar{u} \geq c$ and $\bar{x} \geq 0$ together imply $c^\top \bar{x} \leq (A^\top \bar{u})^\top \bar{x} = \bar{u}^\top A\bar{x}$.
3. We bound $\bar{u}^\top A\bar{x}$ from above because $A\bar{x} \leq b$ and $\bar{u} \geq 0$ give $\bar{u}^\top A\bar{x} \leq \bar{u}^\top b = b^\top \bar{u}$.

Chaining the two bounds yields the full inequality:

$$c^\top \bar{x} \;\leq\; (A^\top \bar{u})^\top \bar{x} \;=\; \bar{u}^\top A\,\bar{x} \;\leq\; \bar{u}^\top b \;=\; b^\top \bar{u}.$$

Thus every dual-feasible $\bar{u}$ provides a valid upper bound on every primal objective value $c^\top \bar{x}$.

Weak duality has several immediate and important consequences.

Corollary 5.6.2 (Optimality certificate). *If $\bar{x}$ is primal-feasible, $\bar{u}$ is dual-feasible, and $c^\top \bar{x} = b^\top \bar{u}$, then $\bar{x}$ is primal-optimal and $\bar{u}$ is dual-optimal.*

Proof. Intuitively, matching objective values lets us use weak duality as a two-sided pincer that forces both solutions to be optimal.

1. We show $\bar{x}$ is primal-optimal because for any primal-feasible x, weak duality gives $c^\top x \leq b^\top \bar{u} = c^\top \bar{x}$, so $\bar{x}$ achieves the highest possible primal value.
2. We show $\bar{u}$ is dual-optimal because for any dual-feasible u, weak duality gives $b^\top u \geq c^\top \bar{x} = b^\top \bar{u}$, so $\bar{u}$ achieves the lowest possible dual value.

Thus, equal primal and dual objective values certify simultaneous optimality of both solutions without any further computation. □

This is extremely useful. To *certify* that a solution is optimal, we just need to exhibit a dual solution with the same objective value. No need to enumerate vertices or run Simplex again.

Corollary 5.6.3 (Unbounded ⇒ infeasible)**.** 1. *If the primal is unbounded ($c^\top x \to +\infty$), then the dual is infeasible.*
2. *If the dual is unbounded ($b^\top u \to -\infty$), then the primal is infeasible.*

Proof. Intuitively, a feasible solution on one side of the duality relationship always imposes a finite bound on the other side, so unboundedness forces infeasibility across.

1. We prove the first item by contradiction: if the dual had a feasible $\bar{u}$, weak duality would give $c^\top x \le b^\top \bar{u}$ for all primal-feasible x, a finite upper bound that contradicts primal unboundedness.
2. We prove the second item analogously: a feasible primal $\bar{x}$ would give $b^\top u \ge c^\top \bar{x}$ for all dual-feasible u, a finite lower bound contradicting dual unboundedness.

Thus, unboundedness on either side forces infeasibility on the other. □

Caution: both can be infeasible simultaneously! Weak duality does not rule this out.

Remark 5.6.4 (Both infeasible is possible). It is possible for both the primal and the dual to be infeasible. For example: max $x_1 + x_2$ s.t. $x_1 - x_2 \le 0$, $-x_1 + x_2 \le -1$, $x_1, x_2 \ge 0$ has an empty feasible set (the constraints imply $x_1 \le x_2$ and $x_2 \le x_1 - 1$, a contradiction for $x_1, x_2 \ge 0$), and so does its dual. Weak duality does not prevent this.

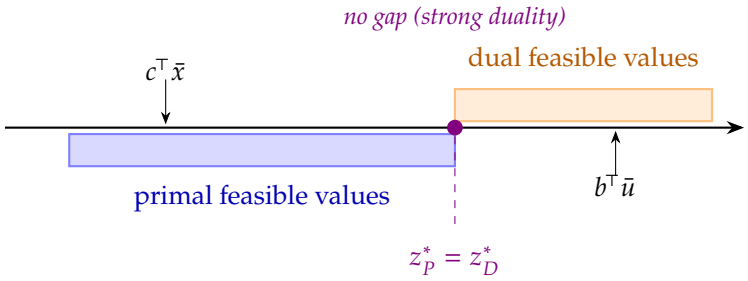


Figure 5.3: Weak duality says primal values ≤ dual values. Strong duality (next section) says the optima meet: no duality gap.

5.7 Strong Duality

Strong duality: at optimality, the primal and dual objectives are equal. This is special to LP—it fails for ILP!

Weak duality gives an inequality: $z_P^* \le z_D^*$. Could there be a gap? For linear programming, the answer is **no**.

Theorem 5.7.1 (Strong Duality)**.** *If the primal LP has an optimal solution x^*, then the dual LP also has an optimal solution u^*, and*

$$c^\top x^* = b^\top u^*.$$

This is one of the most powerful results in LP theory. It says that the primal and dual optima are equal—there is **no duality gap** for linear programmes.

Formal details — Proof of Strong Duality via the Simplex method

The key idea is that the optimal Simplex basis itself manufactures a dual-feasible solution whose objective value matches the primal optimum, so the optimality-certificate corollary closes the gap.

Suppose the Simplex method has found a primal optimum x^* with basis $\mathcal{B}$. At the optimal tableau:

- $x^*_{\mathcal{B}} = A_{\mathcal{B}}^{-1} b = \bar{b} \geq 0, \quad x^*_{\mathcal{N}} = 0.$
- $\hat{c}_j = c_j - c_{\mathcal{B}}^\top A_{\mathcal{B}}^{-1} A_j \leq 0$ for all $j \in \mathcal{N}$.
- $z^* = c_{\mathcal{B}}^\top A_{\mathcal{B}}^{-1} b$.

1. We define a dual candidate using the optimal basis because the formula $u^{*\top} = c_{\mathcal{B}}^\top A_{\mathcal{B}}^{-1}$ is the unique vector that prices resources so that the basis is "shadow-price optimal":
$$u^{*\top} \;=\; c_{\mathcal{B}}^\top A_{\mathcal{B}}^{-1}.$$
2. We verify dual feasibility because we need $u^{*\top} A_j \geq c_j$ for all j:
   - For $j \in \mathcal{B}$: $u^{*\top} A_j = c_{\mathcal{B}}^\top A_{\mathcal{B}}^{-1} A_j = c_j$ (equality, since A_j is a basis column).
   - For $j \in \mathcal{N}$: $\hat{c}_j = c_j - u^{*\top} A_j \leq 0$ rearranges to $u^{*\top} A_j \geq c_j$.
3. We confirm that the dual objective matches z^* because this will let the optimality-certificate corollary fire:
$$b^\top u^* = b^\top A_{\mathcal{B}}^{-\top} c_{\mathcal{B}} = (A_{\mathcal{B}}^{-1} b)^\top c_{\mathcal{B}} = \bar{b}^\top c_{\mathcal{B}} = z^*.$$

Since u^* is dual-feasible and $b^\top u^* = c^\top x^*$, theorem 5.6.2 confirms that both x^* and u^* are optimal, proving that the primal and dual optima coincide with no duality gap.

Remark 5.7.2 (Reduced costs and dual variables). The proof reveals a fundamental connection. At the Simplex optimum, the dual optimal solution is
$$u^{*\top} = c_{\mathcal{B}}^\top A_{\mathcal{B}}^{-1},$$
and the reduced costs are
$$\hat{c}_j = c_j - u^{*\top} A_j.$$
In other words, the **reduced costs are exactly the dual slacks** (with a sign flip). The Simplex optimality condition $\hat{c}_j \leq 0$ is exactly **dual feasibility**: $u^{*\top} A_j \geq c_j$.

This observation is profound: the Simplex method has been solving the dual problem all along, without us realising it! Every time it checks the reduced costs, it is checking dual feasibility. When all reduced costs are ≤ 0, the current basis yields both a primal-optimal and a dual-optimal solution.

Reduced costs $\hat{c}_j \leq 0 \iff$ dual feasibility. The Simplex solves primal and dual simultaneously!

Intermezzo — Duality gap in integer programming

Strong duality is special to LP. For *integer* linear programmes, the LP relaxation dual gives an upper bound (for maximisation), but the gap between the LP optimum and the integer optimum

can be strictly positive. This **integrality gap** is a central difficulty in combinatorial optimisation and is the reason techniques like branch and bound (chapter 6) are needed.

5.8 Complementary Slackness

Complementary slackness gives necessary and sufficient conditions for optimality of a primal-dual pair.

Strong duality tells us that optimal primal and dual values are equal. Complementary slackness sharpens this into a *structural* condition on optimal solutions.

Theorem 5.8.1 (Complementary Slackness)**.** *Let x^* be primal-feasible and u^* be dual-feasible (for the canonical-form pair). Then x^* and u^* are* both *optimal if and only if the following two conditions hold:*

1. ***Primal complementarity:*** *For each $j = 1, \dots, n$,*

$$x_j^* \cdot \left[(A^\top u^*)_j - c_j\right] = 0.$$

   (If $x_j^* > 0$, the dual constraint is tight.)

2. ***Dual complementarity:*** *For each $i = 1, \dots, m$,*

$$u_i^* \cdot \left[b_i - (Ax^*)_i\right] = 0.$$

   (If $u_i^* > 0$, the primal constraint is tight.)

In plain language:

- If a primal variable $x_j^* > 0$, the corresponding dual constraint must be *binding*: $(A^\top u^*)_j = c_j$. Conversely, if the dual constraint has slack, then $x_j^* = 0$.
- If a dual variable $u_i^* > 0$, the corresponding primal constraint must be binding: $(Ax^*)_i = b_i$. Conversely, if the primal constraint has slack, then $u_i^* = 0$.

"If a resource is not fully used, its shadow price is zero."

The economic intuition is clean: if a resource is not fully used (primal constraint has slack), then adding more of it has no value (dual variable is zero). If a product is produced ($x_j > 0$), then the "implied cost" of the resources it consumes exactly equals its profit (dual constraint is tight).

Formal details — Proof of Complementary Slackness

Intuitively, strong duality forces the two inequalities in the weak duality chain to be *equalities*, and the only way a sum of non-negative terms can be zero is if every term is individually zero — that is precisely the complementarity conditions.

1. We recall the weak duality chain because it contains exactly the two inequalities we will force to equality:

$$c^\top x^* \;\le\; (A^\top u^*)^\top x^* \;=\; u^{*\top} A\, x^* \;\le\; u^{*\top} b \;=\; b^\top u^*.$$

2. We apply strong duality because $c^\top x^* = b^\top u^*$ forces both $\le$ signs to be equalities.
3. We derive primal complementarity because the first equality $(A^\top u^* - c)^\top x^* = 0$, combined with $A^\top u^* \ge c$ (dual feasibility) and $x^* \ge 0$ (primal feasibility), means each term $[(A^\top u^*)_j - c_j] \cdot x_j^* \ge 0$ sums to zero, so every term vanishes:

$$x_j^* \cdot \left[(A^\top u^*)_j - c_j\right] = 0 \qquad \text{for all } j.$$

4. We derive dual complementarity because the second equality $u^{*\top}(b - Ax^*) = 0$, combined with $b - Ax^* \geq 0$ (primal feasibility) and $u^* \geq 0$ (dual feasibility), gives each term $u_i^*(b_i - (Ax^*)_i) \geq 0$ summing to zero, so:
$$u_i^* \cdot \left[b_i - (Ax^*)_i\right] = 0 \qquad \text{for all } i.$$
5. For the *converse*, we note that if both complementarity conditions hold together with feasibility, the chain of weak duality collapses to equality $c^\top x^* = (A^\top u^*)^\top x^* = u^{*\top} A x^* = u^{*\top} b = b^\top u^*$, so theorem 5.6.2 confirms both are optimal.

Thus, a primal-dual feasible pair is simultaneously optimal if and only if complementary slackness holds at every primal variable and every dual variable.

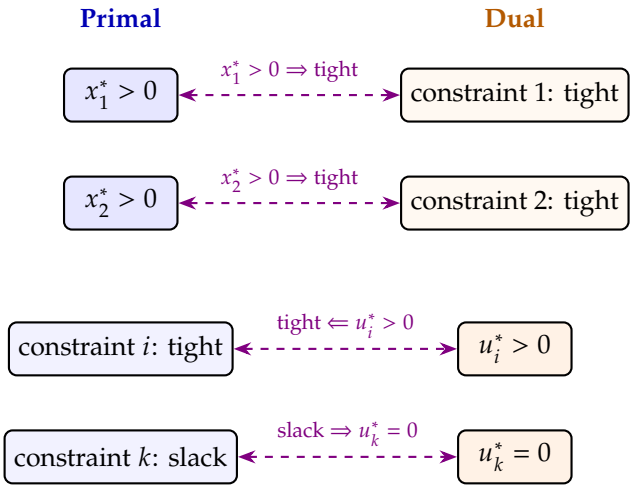


Figure 5.4: Complementary slackness: a primal variable is positive only if its dual constraint is tight, and a dual variable is positive only if its primal constraint is tight.

Complementary slackness gives us a powerful tool for *verifying* optimality without solving the LP from scratch.

Observation 5.8.2 (Checking optimality via complementary slackness). Given a candidate primal solution x^* and dual solution u^*, to verify that both are optimal, check three things:

1. x^* is primal-feasible.
2. u^* is dual-feasible.
3. Complementary slackness holds.

If all three conditions hold, then both x^* and u^* are optimal (and $c^\top x^* = b^\top u^*$). This check requires no pivoting—just arithmetic.

5.9 Economic Interpretation: Shadow Prices

Dual variables are the shadow prices *of resources.*

The dual variables have a beautiful economic interpretation that makes duality much more than a mathematical curiosity.

Consider the furniture workshop again. The primal constraints are resource limits:

- $2x_1 + 5x_2 \leq 40$ (wood: 40 units available)
- $4x_1 + 2x_2 \leq 32$ (labour: 32 hours available)

Now suppose we could get one extra unit of wood—the RHS of the first constraint changes from 40 to 41. **How much would the optimal profit increase?**

Definition 5.9.1 (Shadow price)**.** The **shadow price** of the i-th constraint is the rate of change of the optimal objective value with respect to the i-th RHS coefficient:

$$u_i^* = \frac{\partial z^*}{\partial b_i}.$$

It measures how much the optimal profit would increase if we had one additional unit of the i-th resource.

This is exactly the dual variable u_i^*. To see why, recall from the strong duality proof (section 5.7) that $z^* = c_{\mathcal{B}}^\top A_{\mathcal{B}}^{-1} b = u^{*\top} b$. As long as the optimal *basis* does not change (i.e. for small perturbations of b), the optimal value is

$$z^*(b) = u^{*\top} b = \sum_{i=1}^{m} u_i^* \, b_i.$$

Differentiating with respect to b_i gives $\partial z^* / \partial b_i = u_i^*$.

Remark 5.9.2 (Validity range of shadow prices). The shadow price interpretation $\Delta z^* \approx u_i^* \cdot \Delta b_i$ is exact only as long as the *optimal basis* remains the same. For larger changes in b_i, a different basis may become optimal, and the shadow price changes. The range of b_i values for which the current basis remains optimal is called the **sensitivity range** (or **allowable increase/decrease**). We will not develop full sensitivity analysis here, but the key idea is that $z^*(b)$ is a *piecewise linear* function of b, with the slope in each piece given by the dual variables of the corresponding optimal basis.

5.10 Worked Example: The Furniture Workshop Dual

We now bring everything together on our recurring example.

Let's put all the pieces together on the furniture workshop LP. We will:

1. Write the dual.
2. Compute the dual optimal solution from the Simplex basis.
3. Verify complementary slackness.
4. Interpret the shadow prices.

5.10.1 Step 1: Write the dual

The canonical primal (eq. (5.1)) is:

$$\begin{aligned} \text{maximize} \quad & 30\,x_1 + 50\,x_2 \\ \text{subject to} \quad & 2\,x_1 + 5\,x_2 \le 40 \quad \text{(wood)} \\ & 4\,x_1 + 2\,x_2 \le 32 \quad \text{(labour)} \\ & x_1, x_2 \ge 0. \end{aligned}$$

By the canonical-form shortcut (section 5.5.2), the dual is:

$$\begin{aligned} \text{minimize} \quad & 40\,u_1 + 32\,u_2 \\ \text{subject to} \quad & 2\,u_1 + 4\,u_2 \ge 30 \quad \text{(chair)} \\ & 5\,u_1 + 2\,u_2 \ge 50 \quad \text{(table)} \\ & u_1, u_2 \ge 0. \end{aligned} \tag{5.8}$$

Here u_1 is the dual variable (shadow price) for the wood constraint and u_2 for the labour constraint.

5.10.2 Step 2: Compute the dual solution from the Simplex basis

The dual solution is read from the Simplex optimal basis using $u^{*\top} = c_{\mathcal{B}}^\top A_{\mathcal{B}}^{-1}$.

In section 4.6, the Simplex method terminated with the optimal basis $\mathcal{B} = \{x_2, x_1\}$ (row 1 corresponds to x_2, row 2 to x_1). From theorem 5.7.2, the dual optimal solution is $u^{*\top} = c_{\mathcal{B}}^\top A_{\mathcal{B}}^{-1}$.

The optimal tableau (section 4.6) has the following body rows:

	x_1	x_2	s_1	s_2	RHS
x_2	0	1	$\frac{1}{4}$	$-\frac{1}{8}$	6
x_1	1	0	$-\frac{1}{8}$	$\frac{5}{16}$	5

Since the slack columns of A are the identity ($A_{s_1} = e_1$, $A_{s_2} = e_2$), the columns under s_1 and s_2 in the tableau give $A_{\mathcal{B}}^{-1}$ directly:

$$A_{\mathcal{B}}^{-1} = \begin{pmatrix} \frac{1}{4} & -\frac{1}{8} \\ -\frac{1}{8} & \frac{5}{16} \end{pmatrix}.$$

With $c_{\mathcal{B}} = (50, 30)^\top$ (profits for x_2, x_1):

$$u^{*\top} = (50, 30) \begin{pmatrix} \frac{1}{4} & -\frac{1}{8} \\ -\frac{1}{8} & \frac{5}{16} \end{pmatrix} = \left(\tfrac{50}{4} - \tfrac{30}{8},\ -\tfrac{50}{8} + \tfrac{150}{16} \right) = \left(\tfrac{100-30}{8},\ \tfrac{-100+150}{16} \right) = \left(\tfrac{70}{8},\ \tfrac{50}{16} \right).$$

Therefore:

$$\boxed{u_1^* = \frac{35}{4} = 8.75, \qquad u_2^* = \frac{25}{8} = 3.125.} \tag{5.9}$$

Dual objective value:

$$40 \cdot \frac{35}{4} + 32 \cdot \frac{25}{8} = 350 + 100 = 450 = z_P^*. \quad \checkmark$$

Strong duality holds, as expected.

Dual feasibility check:

- Chair: $2 \cdot \frac{35}{4} + 4 \cdot \frac{25}{8} = \frac{70}{4} + \frac{100}{8} = \frac{140+100}{8} = \frac{240}{8} = 30 \geq 30$. ✓
- Table: $5 \cdot \frac{35}{4} + 2 \cdot \frac{25}{8} = \frac{175}{4} + \frac{50}{8} = \frac{350+50}{8} = \frac{400}{8} = 50 \geq 50$. ✓
- $u_1^* = 8.75 \geq 0$, $u_2^* = 3.125 \geq 0$. ✓

5.10.3 Step 3: Verify complementary slackness

The primal optimal solution is $x^* = (5, 6)$ and the dual optimal solution is $u^* = (35/4, 25/8)$.

Dual complementarity (primal constraints):

- Wood: $2(5) + 5(6) = 40 = b_1$ (tight), and $u_1^* = 35/4 > 0$. ✓
- Labour: $4(5) + 2(6) = 32 = b_2$ (tight), and $u_2^* = 25/8 > 0$. ✓

Both resources are fully used, which is consistent with both dual variables being strictly positive.

Primal complementarity (dual constraints):

- Chair: $2u_1^* + 4u_2^* = 30 = c_1$ (tight), and $x_1^* = 5 > 0$. ✓
- Table: $5u_1^* + 2u_2^* = 50 = c_2$ (tight), and $x_2^* = 6 > 0$. ✓

Both dual constraints are tight, consistent with both primal variables being positive.

All four complementary slackness conditions hold—both solutions are confirmed optimal.

All complementary slackness conditions are satisfied. This confirms (independently of the Simplex method) that $x^* = (5, 6)$ and $u^* = (35/4, 25/8)$ are optimal for the primal and dual respectively.

5.10.4 Step 4: Interpret the shadow prices

The dual optimal values tell us the marginal worth of each resource:

Resource	Dual var.	Shadow price	Interpretation
Wood ($b_1 = 40$)	u_1^*	$\frac{35}{4} = 8.75$	+1 unit wood $\Rightarrow$ +€8.75 profit
Labour ($b_2 = 32$)	u_2^*	$\frac{25}{8} = 3.125$	+1 hour labour $\Rightarrow$ +€3.125 profit

Wood is nearly three times more valuable than labour at the margin.

Managerial insight: Each extra unit of wood is worth €8.75 in additional profit, while each extra hour of labour is worth only €3.125. If the workshop manager can purchase additional resources, wood should be the priority—it delivers nearly three times the marginal return of labour.

Let's verify with a small perturbation. If b_1 increases from 40 to 41 (one extra unit of wood), the new right-hand side is $b' = (41, 32)^\top$. If the same basis $\{x_2, x_1\}$ remains optimal:

$$\bar{b}' = A_{\mathcal{B}}^{-1} b' = \begin{pmatrix} 1/4 & -1/8 \\ -1/8 & 5/16 \end{pmatrix} \begin{pmatrix} 41 \\ 32 \end{pmatrix} = \begin{pmatrix} 41/4 - 4 \\ -41/8 + 10 \end{pmatrix} = \begin{pmatrix} 25/4 \\ 39/8 \end{pmatrix}.$$

Both components are positive (so the basis is still feasible), and the reduced costs are unchanged (they depend only on the basis, not on b), so the basis is still optimal. The new profit is:

$$z^* = 50 \cdot \frac{25}{4} + 30 \cdot \frac{39}{8} = \frac{1250}{4} + \frac{1170}{8} = \frac{2500 + 1170}{8} = \frac{3670}{8} = 458.75.$$

Change in profit: $458.75 - 450 = 8.75 = u_1^*$. ✓

Example 5.10.1 (When a resource has zero shadow price)**.** Suppose we add a third constraint to the furniture LP: a *finishing* constraint $x_1 + x_2 \leq 20$ (at most 20 items can be finished per period). At the optimum $(5, 6)$, the finishing constraint gives $5 + 6 = 11 \leq 20$—it has slack of 9. By complementary slackness, the corresponding dual variable $u_3^* = 0$: finishing capacity is not a bottleneck, so an extra unit has no value.

Exercise 5.10.2 (Dual of a three-constraint LP). A factory produces products A and B. The LP is:

$$\begin{aligned} \text{maximize} \quad & 20x_1 + 30x_2 \\ \text{subject to} \quad & x_1 + 2x_2 \leq 14 \\ & 3x_1 + 2x_2 \leq 18 \\ & x_1 + x_2 \leq 8 \\ & x_1, x_2 \geq 0. \end{aligned}$$

1. Write the dual LP.
2. The primal optimum is at $(x_1, x_2) = (2, 6)$ with profit 220. Use complementary slackness to find the dual optimal solution.
3. Interpret the shadow prices.

Hint: First check which primal constraints are tight at $(2, 6)$. Constraint 1: $2 + 12 = 14 = b_1$ (tight). Constraint 2: $6 + 12 = 18 = b_2$ (tight). Constraint 3: $2 + 6 = 8 = b_3$ (tight). All three are tight, so all dual variables could

be positive. But we have only two primal variables, both positive, giving two dual-constraint equalities—together with $u_i \geq 0$, this determines the solution.

5.11 The Dual Simplex Method

Motivation

Think of the primal Simplex as walking along the boundary of the feasible region toward the optimum, always keeping primal feasibility. The dual Simplex does the opposite: it starts from a point that is *optimal for the LP relaxation* (dual feasible) but possibly infeasible (some $\bar{b}_i < 0$), and walks toward primal feasibility while keeping optimality.

More precisely, the primal Simplex method maintains *primal feasibility* ($\bar{b} \geq 0$) at every iteration while improving the objective, terminating when dual feasibility is achieved (all reduced costs $\hat{c}_j \leq 0$). The **dual Simplex method** starts from a basis that is *dual feasible* (all $\hat{c}_j \leq 0$, so the current solution would be optimal if it were primal feasible) and iterates by restoring primal feasibility one row at a time, while preserving dual feasibility throughout.

When to use it

The dual Simplex is particularly valuable after adding a **cutting plane** (Chapter 6): the cut is violated by the current LP optimum, so appending it as a new row makes the current basis primal-infeasible. Because the reduced costs are unchanged, dual feasibility is still satisfied. Dual Simplex re-optimises from the current basis without discarding any prior work.

The algorithm

Formal details — Dual Simplex iteration

Intuitively, the dual Simplex is the mirror image of the primal: it keeps the objective looking optimal (dual feasibility intact) while driving away primal infeasibility one row at a time.

Assumption: current basis $\mathcal{B}$ is dual feasible ($\hat{c}_j \leq 0$ for all $j \notin \mathcal{B}$); some $\bar{b}_r < 0$ (primal infeasible).

1. **Check for optimality** because if $\bar{b}_i \geq 0$ for all i then the current basis is both primal and dual feasible — **optimal**. Stop.
2. **Select the leaving variable** by choosing row r with $\bar{b}_r < 0$ (e.g., the most negative) because the basic variable $x_{\mathcal{B}(r)}$ is the one violating primal feasibility that we want to expel from the basis.
3. **Check for primal infeasibility** because if every $\bar{a}_{rj} \geq 0$ for $j \notin \mathcal{B}$, no entering variable can repair row r: the primal LP is **infeasible** (equivalently, the dual is unbounded). Stop.
4. **Select the entering variable** using the minimum-ratio test on the *reduced costs* (not the RHS) because we must preserve dual feasibility $\hat{c}_j \leq 0$ after the pivot — the ratio test ensures no reduced cost becomes positive:
$$j^* = \operatorname*{argmin}_{j:\bar{a}_{rj}<0} \left| \frac{\hat{c}_j}{\bar{a}_{rj}} \right|.$$
5. **Pivot** using the same row operations as in the primal Simplex (divide row r by $\bar{a}_{rj^*}$, then eliminate the entering column from all other rows) because this is standard Gaussian

elimination, which keeps the tableau consistent; update $\mathcal{B}$ accordingly.

6. Go to Step 1.

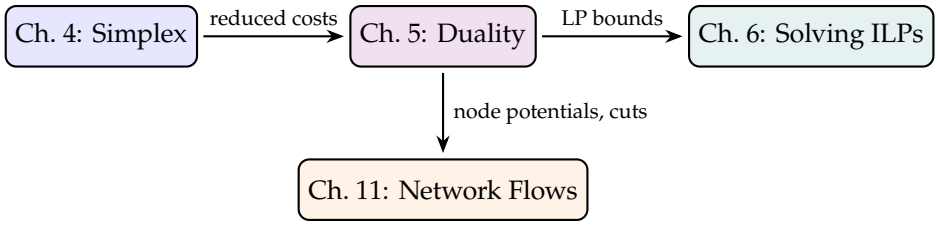
Each iteration repairs one primal infeasibility while preserving dual feasibility, so the method terminates at a point that is feasible and optimal for both the primal and the dual.

Remark 5.11.1. The dual Simplex terminates in finitely many steps under non-degeneracy; cycling-prevention rules analogous to Bland's rule exist for the degenerate case. This implies that the cutting-plane algorithm of Chapter 6 is well-founded: each cut added after an LP optimum is fractional induces at most finitely many dual Simplex iterations before a new (integer or fractional) optimum is found, so the overall branch-and-cut procedure makes measurable progress at every node.

Looking Ahead

Duality is not just an elegant theoretical result—it is a workhorse throughout optimisation.

- **Chapter 6 (Solving ILPs)** will use LP duality to provide bounds in branch-and-bound and to derive cutting planes. The LP relaxation bound is precisely the dual bound.
- **Chapter 7 (TUM)** will show that when the constraint matrix has a special structure (total unimodularity), the LP optimum is automatically integer—and the dual variables are integer too.
- In **network optimisation** (chapter 11), dual variables appear as *node potentials* in shortest-path problems and as *cuts* in max-flow problems. The celebrated max-flow min-cut theorem is a special case of LP strong duality.



Summary & Key Takeaways

- **Duality Correspondence:** Every primal LP has a symmetric dual LP where primal constraints correspond to dual variables, and primal variables to dual constraints.
- **Weak Duality Theorem:** For any primal feasible x and dual feasible y, $c^\top x \leq b^\top y$.
- **Strong Duality Theorem:** If either the primal or the dual has an optimal solution, both have optimal solutions and $c^\top x^* = b^\top y^*$.
- **Complementary Slackness:** Feasible primal x and dual y are optimal if and only if $x_j(a_j^\top y - c_j) = 0$ for all j and $y_i(b_i - a_i^\top x) = 0$ for all i.
- **Shadow Prices:** Optimal dual variables y_i^* represent the marginal rate of change of the objective value per unit increase in RHS resource b_i.

Exercises

Exercise 1 (Dual of a standard max LP). Consider the primal LP:

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 5x_2 \\ \text{subject to} \quad & x_1 + x_2 \le 4 \\ & x_1 + 3x_2 \le 6 \\ & x_1, x_2 \ge 0. \end{aligned}$$

1. Write the dual LP using dual variables $u_1, u_2 \ge 0$.
2. Identify the dual objective direction and the type (min or max) of the dual.
3. How many dual variables are there? How many dual constraints? Explain the general pattern.

Exercise 2 (Dual of a min LP with $\ge$ constraints). Consider:

$$\begin{aligned} \text{minimize} \quad & 2x_1 + 4x_2 + x_3 \\ \text{subject to} \quad & x_1 + 2x_2 + x_3 \ge 5 \\ & 2x_1 + x_2 \ge 3 \\ & x_1, x_2, x_3 \ge 0. \end{aligned}$$

1. Write the dual LP.
2. What sign restrictions apply to the dual variables?
3. Verify that the dimensions (number of constraints / variables) in the dual match the general primal–dual correspondence table.

Exercise 3 (Dual with equality constraints). Consider the primal LP:

$$\begin{aligned} \text{maximize} \quad & 4x_1 + 6x_2 + 2x_3 \\ \text{subject to} \quad & x_1 + x_2 + x_3 = 10 \\ & 2x_1 + x_2 \le 14 \\ & x_1 + 3x_3 \le 12 \\ & x_1, x_2, x_3 \ge 0. \end{aligned}$$

1. Write the dual LP. What sign restriction does the dual variable associated with the equality constraint carry?
2. How do equality primal constraints manifest in the dual compared with inequality primal constraints?

Exercise 4 (Dual with a free variable). Consider:

$$\begin{aligned} \text{maximize} \quad & x_1 - x_2 + 2x_3 \\ \text{subject to} \quad & x_1 + x_2 + x_3 \le 6 \\ & x_1 - x_2 \le 2 \\ & x_1, x_2 \ge 0, \quad x_3 \text{ free}. \end{aligned}$$

1. Write the dual LP. What dual constraint corresponds to the free primal variable x_3?
2. Explain in one sentence why a free primal variable forces an equality in the corresponding dual constraint.

Exercise 5 (Dual with non-positive variable). Consider the primal:

$$\begin{aligned} \text{maximize} \quad & 5x_1 + 3x_2 \\ \text{subject to} \quad & 2x_1 + x_2 \le 8 \\ & x_1 + 2x_2 \le 7 \\ & x_1 \ge 0, \quad x_2 \le 0. \end{aligned}$$

1. Write the dual LP, stating the sign of each dual variable and the direction of each dual constraint.
2. What happens to the dual constraint corresponding to x_2 compared with a standard non-negative primal variable?

Exercise 6 (Mixed constraint types). Consider the following LP:

$$\begin{aligned} \text{minimize} \quad & x_1 + 2x_2 + 3x_3 \\ \text{subject to} \quad & x_1 + x_2 \ge 4 \\ & x_2 + x_3 = 3 \\ & x_1 + x_3 \le 5 \\ & x_1, x_2, x_3 \ge 0. \end{aligned}$$

1. Write the dual LP, carefully stating sign restrictions for each dual variable based on the type of primal constraint.
2. Verify the dimensions: primal has $n = 3$ variables and $m = 3$ constraints; confirm the dual has $m = 3$ variables and $n = 3$ constraints.

Exercise 7 (Diet LP dual). A minimum-cost diet must supply at least $R_1 = 2000$ kcal, $R_2 = 50$ g protein, and $R_3 = 30$ mg iron per day using two foods whose nutritional content and costs are:

	kcal	protein (g)	iron (mg)
Food 1 (cost \$3/unit)	400	10	5
Food 2 (cost \$5/unit)	200	15	8

1. Formulate the primal LP (minimize cost).
2. Write the dual LP and give an economic interpretation: what do the dual variables represent?
3. What is the dual objective, and what does its optimal value equal (by strong duality)?

Exercise 8 (Dual of the furniture workshop LP). Consider the following LP (a variant of the furniture workshop introduced in chapter 1; compare with the standard formulation in theorem 1.8.1):

$$\begin{aligned} \text{maximize} \quad & 40x_1 + 30x_2 \\ \text{subject to} \quad & x_1 + x_2 \le 40 \\ & 2x_1 + x_2 \le 60 \\ & x_1 + x_2 \le 50 \\ & x_1, x_2 \ge 0. \end{aligned}$$

1. Write the dual LP.
2. The primal optimum is $(x_1^*, x_2^*) = (20, 20)$ with $z^* = 1400$. Identify which primal constraints are tight.
3. Use complementary slackness to derive the dual optimal solution (u_1^*, u_2^*, u_3^*).

4. Verify that the dual optimal value equals 1400.

Exercise 9 (Proving weak duality from first principles). Let the primal be $\max\{c^\top x : Ax \le b,\ x \ge 0\}$ and the dual be $\min\{b^\top u : A^\top u \ge c,\ u \ge 0\}$.

Prove weak duality in at most four lines: starting from x feasible for the primal and u feasible for the dual, show $c^\top x \le b^\top u$.

Hint: Multiply $Ax \le b$ (componentwise) by $u \ge 0$, then use $A^\top u \ge c$ and $x \ge 0$.

Exercise 10 (Verifying weak duality numerically). Consider the LP:

$$\begin{aligned} \text{maximize} \quad & 5x_1 + 4x_2 \\ \text{subject to} \quad & 6x_1 + 4x_2 \le 24 \\ & x_1 + 2x_2 \le 6 \\ & x_1, x_2 \ge 0. \end{aligned}$$

The dual is $\min\{24u_1 + 6u_2 : 6u_1 + u_2 \ge 5,\ 4u_1 + 2u_2 \ge 4,\ u_1, u_2 \ge 0\}$.

1. Verify that $(x_1, x_2) = (3, 1)$ is primal feasible and compute its objective value.
2. Verify that $(u_1, u_2) = (0.5, 2)$ is dual feasible and compute its objective value.
3. Does weak duality hold for these two solutions?
4. Are either of these solutions optimal? How do you know?

Exercise 11 (Using weak duality to bound without solving). Consider the LP:

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 2x_2 + 5x_3 \\ \text{subject to} \quad & x_1 + x_2 + x_3 \le 10 \\ & 2x_1 + x_2 + 3x_3 \le 18 \\ & x_1, x_2, x_3 \ge 0. \end{aligned}$$

1. Write the dual LP.
2. Find, by inspection or trial, a dual feasible solution (u_1, u_2) with objective value as small as possible. This value is an upper bound on the primal optimum.
3. State precisely what weak duality guarantees about the relationship between any primal feasible value and any dual feasible value.

Exercise 12 (Weak duality and infeasibility/unboundedness). State whether each of the following is **True** or **False**, and give a one-sentence justification:

1. "If the primal is infeasible, then the dual must be unbounded."
2. "If the dual is infeasible, then the primal must be unbounded."
3. "If the primal is unbounded, then the dual must be infeasible."
4. "It is impossible for both the primal and the dual to be infeasible simultaneously."
5. "Weak duality implies $z^*_{\text{primal}} \le z^*_{\text{dual}}$ for any max–min primal–dual pair."

Exercise 13 (Gap between primal and dual). Suppose you are given a maximisation LP and you find a primal feasible solution with value $v_P = 120$ and a dual feasible solution with value $v_D = 135$.

1. What can you conclude immediately about the optimal primal value z^*?
2. If you then find a primal feasible solution with value 128, how tight is your bound now?
3. What additional condition would allow you to certify that a given primal solution is optimal without using the Simplex final tableau?

Exercise 14 (Strong duality: statement and conditions). State the Strong Duality Theorem precisely. Your statement should specify:

1. The exact condition under which strong duality holds.
2. What equality is asserted between the optimal objective values.
3. What happens if the primal is unbounded?
4. What happens if the primal is infeasible but the dual is feasible?

Exercise 15 (CS to find dual optimal — two-variable LP). The primal LP is:

$$\begin{aligned} \text{maximize} \quad & 6x_1 + 8x_2 \\ \text{subject to} \quad & 3x_1 + x_2 \le 9 \\ & x_1 + 2x_2 \le 8 \\ & x_1, x_2 \ge 0. \end{aligned}$$

The primal optimum is $(x_1^*, x_2^*) = (2, 3)$ with $z^* = 36$.

1. Write the dual LP.
2. Identify which primal constraints are tight at $(2, 3)$ and which primal variables are positive.
3. Write out all complementary slackness conditions.
4. Solve the resulting system to find (u_1^*, u_2^*).
5. Verify $b^\top u^* = 36$.

Exercise 16 (CS to find dual optimal — three-constraint LP). The following LP has three constraints:

$$\begin{aligned} \text{maximize} \quad & 2x_1 + 3x_2 \\ \text{subject to} \quad & x_1 + x_2 \le 4 \\ & x_1 + 3x_2 \le 6 \\ & 2x_1 + x_2 \le 7 \\ & x_1, x_2 \ge 0. \end{aligned}$$

The primal optimum is $(x_1^*, x_2^*) = (3, 1)$ with $z^* = 9$.

1. Write the dual LP.
2. Determine which primal constraints are active at $(3, 1)$.
3. Write and solve the complementary slackness system to obtain (u_1^*, u_2^*, u_3^*).
4. Verify dual feasibility: check all dual constraints.

Exercise 17 (CS to find dual optimal — three-variable LP). Consider:

$$\begin{aligned} \text{maximize} \quad & x_1 + 2x_2 + 3x_3 \\ \text{subject to} \quad & x_1 + x_2 + x_3 \le 6 \\ & 2x_1 + x_3 \le 8 \\ & x_1, x_2, x_3 \ge 0. \end{aligned}$$

The primal optimum is $(x_1^*, x_2^*, x_3^*) = (0, 0, 6)$ with $z^* = 18$.

1. Write the dual LP.
2. Identify which primal variables are positive and which primal constraints are tight.
3. Apply complementary slackness to determine (u_1^*, u_2^*).
4. Verify: $c^\top x^* = b^\top u^*$.

Exercise 18 (Strong duality verification — production LP). A factory makes

three products using two machines. The LP is:

$$\begin{aligned} \text{maximize} \quad & 10x_1 + 12x_2 + 8x_3 \\ \text{subject to} \quad & 2x_1 + 3x_2 + x_3 \le 120 \\ & x_1 + x_2 + 2x_3 \le 80 \\ & x_1, x_2, x_3 \ge 0. \end{aligned}$$

The primal optimum is $(x_1^*, x_2^*, x_3^*) = (0, 40, 0)$ with $z^* = 480$.

1. Write the dual LP.
2. Use complementary slackness to find the dual optimal (u_1^*, u_2^*).
3. Verify dual feasibility for all three dual constraints.
4. Confirm $b^\top u^* = 480$.

Exercise 19 (Degenerate primal optimum). Suppose a maximisation LP has a primal optimal BFS x^* that is *degenerate* (i.e. some basic variable equals zero at the optimum).

1. Write out the complementary slackness conditions for both primal and dual slack.
2. Explain why a degenerate primal optimal BFS does *not* uniquely determine the dual optimal solution through CS alone.
3. Give a one-sentence description of how the dual may respond (multiple dual optima, or a dual optimal face of positive dimension).

Exercise 20 (Dual of dual equals primal). Let the primal be P: $\max\{c^\top x : Ax \le b,\ x \ge 0\}$ with dual D: $\min\{b^\top u : A^\top u \ge c,\ u \ge 0\}$.

1. Write the dual of D (treat D as a minimisation LP with $\ge$ constraints).
2. Show that the dual of D is equivalent to P.
3. What does this symmetry imply about the roles of primal and dual? State whether the following is True or False: "The dual of the dual is always equivalent to the primal."

Exercise 21 (Writing CS conditions). For the LP pair below, write out *all* complementary slackness conditions explicitly (both primal CS and dual CS):

$$\begin{array}{llll} P: \text{ maximize} & 4x_1 + 5x_2 & D: \text{ minimize} & 8u_1 + 7u_2 \\ \text{subject to} & 2x_1 + x_2 \le 8 & \text{subject to} & 2u_1 + u_2 \ge 4 \\ & x_1 + 2x_2 \le 7 & & u_1 + 2u_2 \ge 5 \\ & x_1, x_2 \ge 0. & & u_1, u_2 \ge 0. \end{array}$$

List each condition in the form "(slack or variable) × (dual variable or slack) = 0".

Exercise 22 (Checking a primal-dual pair via CS). Using the LP pair from Exercise 21:

1. Check whether $(x_1, x_2) = (3, 2)$ and $(u_1, u_2) = (1, 2)$ satisfy all complementary slackness conditions.
2. Determine whether this pair is optimal. Justify your answer using the CS theorem.

Exercise 23 (CS: given primal solution, find dual). For the LP:

$$\begin{aligned} \text{maximize} \quad & 7x_1 + 5x_2 + 3x_3 \\ \text{subject to} \quad & x_1 + x_2 + x_3 \le 10 \\ & 2x_1 + x_2 \le 14 \\ & x_1, x_2, x_3 \ge 0. \end{aligned}$$

The primal optimal solution is $(x_1^*, x_2^*, x_3^*) = (4, 6, 0)$.

1. Write the dual LP.
2. Apply the complementary slackness conditions to determine which dual variables must be zero and which constraints must be tight in the dual.
3. Solve for the dual optimal (u_1^*, u_2^*).
4. Verify the optimal value agrees.

Exercise 24 (CS: non-optimal pair detection). Consider the LP pair:

$$\begin{array}{llll} P: & \max & 3x_1 + 2x_2 & \\ \text{subject to} & & x_1 + x_2 \le 4 & \\ & & x_1 + 2x_2 \le 6 & \\ & & x_1, x_2 \ge 0. & \end{array} \qquad \begin{array}{llll} D: & \min & 4u_1 + 6u_2 & \\ \text{subject to} & & u_1 + u_2 \ge 3 & \\ & & u_1 + 2u_2 \ge 2 & \\ & & u_1, u_2 \ge 0. & \end{array}$$

1. Verify that $(x_1, x_2) = (2, 2)$ is primal feasible and $(u_1, u_2) = (2, 1)$ is dual feasible.
2. Check all CS conditions for this pair.
3. Is this pair optimal? Compute both objective values and state what the duality gap is.

Exercise 25 (CS with equality constraints). Suppose a primal LP has an equality constraint $a_i^\top x = b_i$.

1. What does the complementary slackness condition say about the corresponding dual variable u_i?
2. Suppose instead a primal variable x_j is free (unrestricted in sign). What does CS say about the j-th dual constraint?
3. Summarise in a table the four combinations: (primal constraint $\le$ or $=$) $\times$ (primal variable ≥ 0 or free) and the corresponding CS implications.

Exercise 26 (CS optimality certificate). The following LP:

$$\begin{array}{ll} \text{maximize} & 5x_1 + 4x_2 + 3x_3 \\ \text{subject to} & 6x_1 + 4x_2 + 2x_3 \le 240 \\ & 3x_1 + 2x_2 + 5x_3 \le 270 \\ & 5x_1 + 6x_2 + 5x_3 \le 420 \\ & x_1, x_2, x_3 \ge 0 \end{array}$$

has claimed primal optimal $(x_1^*, x_2^*, x_3^*) = (30, 0, 0)$ and dual optimal $(u_1^*, u_2^*, u_3^*) = (5/6, 0, 0)$.

1. Verify primal feasibility.
2. Verify dual feasibility (check all three dual constraints).
3. Check all CS conditions.
4. Compute $c^\top x^*$ and $b^\top u^*$. Do they match?

Exercise 27 (CS and reduced costs). At a Simplex optimum, the reduced costs of non-basic variables are all ≤ 0 (for maximisation).

1. Rewrite the condition "reduced cost of $x_j \le 0$" in terms of dual variables u and coefficients c_j, A_j.
2. Show that this condition is exactly the dual feasibility constraint for variable j.
3. Explain why the Simplex optimality criterion (all reduced costs ≤ 0) is equivalent to dual feasibility.
4. What does this tell you about the dual variables in the final Simplex tableau?

Exercise 28 (Farkas' Lemma: statement). State Farkas' Lemma precisely in the following form: *For a matrix $A \in \mathbb{R}^{m\times n}$ and vector $b \in \mathbb{R}^m$, exactly one of the following holds:*

(i) ...
(ii) ...

Then explain in one paragraph the geometric intuition: what does it mean that b is, or is not, in the cone generated by the columns of A?

Exercise 29 (Farkas' Lemma: proving infeasibility). Use Farkas' Lemma to show that the following system has no non-negative solution:

$$x_1 + 2x_2 = 5, \quad 2x_1 + x_2 = 5, \quad x_1 - x_2 = 2, \quad x_1, x_2 \geq 0.$$

Hint: Find a vector y such that $A^\top y \leq 0$ and $b^\top y > 0$.

Exercise 30 (Farkas alternative for inequalities). The inequality form of Farkas' Lemma states: either $Ax \leq b$ has a solution $x \geq 0$, or there exists $y \geq 0$ with $A^\top y \geq 0$ and $b^\top y < 0$, but not both.

Apply this to show that the system

$$x_1 + x_2 \leq -1, \quad x_1, x_2 \geq 0$$

is infeasible, by explicitly finding the Farkas certificate y.

Exercise 31 (LP infeasibility certificate via Farkas). Consider the LP $\max\{2x_1 + x_2 : x_1 - x_2 \geq 3,\ -x_1 + x_2 \geq 1,\ x_1, x_2 \geq 0\}$.

1. Show graphically or algebraically that the feasible region is empty.
2. Use Farkas' Lemma to produce a certificate of infeasibility (a non-negative vector y satisfying the Farkas alternative).
3. Verify your certificate satisfies the required conditions.

Exercise 32 (Connecting Farkas to duality). Consider a maximisation LP $\max\{c^\top x : Ax \leq b,\ x \geq 0\}$.

1. Suppose the primal is infeasible. Write the system of constraints that would need to be solved, and apply Farkas' Lemma to characterise infeasibility.
2. The Farkas certificate of infeasibility is related to the dual. Identify what relationship exists between the Farkas vector and a dual solution when the primal is infeasible.
3. True or False: "If the primal LP is infeasible, Farkas' Lemma guarantees the dual is either unbounded or infeasible." Justify your answer.

Exercise 33 (Shadow prices — production LP). A factory produces two products with the following LP:

$$\begin{aligned} \text{maximize} \quad & 50x_1 + 40x_2 & \\ \text{subject to} \quad & x_1 + x_2 \leq 100 & \text{(raw material)} \\ & 2x_1 + x_2 \leq 150 & \text{(machine hours)} \\ & x_1 + 2x_2 \leq 120 & \text{(labour hours)} \\ & x_1, x_2 \geq 0. & \end{aligned}$$

The optimal dual variables are $u_1^* = 30$, $u_2^* = 10$, $u_3^* = 0$.

1. Give an economic interpretation of $u_1^* = 30$: what does it mean if one additional unit of raw material becomes available?
2. Why is $u_3^* = 0$? What does this say about labour hours?

3. If the available machine hours increase from 150 to 152, by how much does the optimal profit change (to first order)?

Exercise 34 (Shadow prices — diet problem). A nutritionist formulates a minimum-cost diet (LP) to meet daily requirements. At the optimum, the dual variables (shadow prices) for the nutrition constraints are $u_1^* = 0.04$ (\$/kcal), $u_2^* = 0.10$ (\$/g protein), $u_3^* = 0$ (\$/mg iron).

1. Interpret $u_2^* = 0.10$: if the protein requirement increases by 5 g/day, what happens to the minimum cost?
2. Interpret $u_3^* = 0$: what does it mean for the iron constraint?
3. If a new food becomes available that contributes 300 kcal and 12 g of protein per unit at cost \$2.50, how would you use the shadow prices to quickly assess whether it should be included in the optimal diet?

Exercise 35 (Shadow prices — transportation LP). A transportation LP minimises shipping costs from two warehouses to three stores. At the optimum:

- Shadow price of supply constraint at Warehouse 1: $u_1^* = -3$.
- Shadow price of supply constraint at Warehouse 2: $u_2^* = -2$.
- Shadow price of demand constraint at Store 1: $v_1^* = 5$.

1. Why are supply shadow prices negative in a minimisation LP?
2. Interpret $v_1^* = 5$: what happens to the total cost if Store 1's demand increases by one unit?
3. The unit shipping cost from Warehouse 1 to Store 1 is currently \$8. Use the dual variables to compute the "reduced cost" for this route. What does it mean if the reduced cost is zero?

Exercise 36 (Identifying binding constraints from shadow prices). An LP has five constraints with optimal shadow prices $u^* = (0, 12, 0, 7, 0)$.

1. Which constraints are definitely binding at the optimum?
2. Which constraints are definitely not binding?
3. Could a constraint with zero shadow price be binding? Give a one-sentence explanation.
4. If you want to increase the optimal objective value, which RHS values should you increase?

Exercise 37 (Negative shadow prices). In a minimisation LP, a constraint $a_i^\top x \geq b_i$ has shadow price $u_i^* > 0$.

1. What does a positive shadow price mean for a $\geq$ constraint in a minimisation LP?
2. If b_i increases by one unit (the requirement gets tighter), does the optimal cost increase or decrease?
3. Sketch a two-variable example to illustrate your answer.

Exercise 38 (Shadow price of an equality constraint). A resource-allocation LP has an equality constraint $x_1 + x_2 = B$ (total budget must be *exactly* spent). The shadow price at the optimum is $u^* = 15$.

1. What does $u^* = 15$ mean economically?
2. Unlike a $\leq$ constraint, an equality constraint's shadow price can be negative. Explain why.
3. If the budget increases from B to $B + 1$, must the optimal value increase, decrease, or could it go either way? Justify using the sign of u^*.

Exercise 39 (RHS ranging — single constraint). For the LP:

$$\begin{aligned} \text{maximize} \quad & 5x_1 + 4x_2 \\ \text{subject to} \quad & 6x_1 + 4x_2 \leq b_1 \\ & x_1 + 2x_2 \leq 6 \\ & x_1, x_2 \geq 0, \end{aligned}$$

suppose the current value is $b_1 = 24$ and the optimal basis is $B = \{x_1, x_2\}$ (both structural variables basic). The optimal solution is $(x_1^*, x_2^*) = (3, 1.5)$.

1. How is the optimal solution affected if b_1 changes to 26?
2. For what range of b_1 does the current basis remain optimal (i.e. primal feasibility is maintained)?

Hint: Express the basic variable values as functions of b_1 and require all to remain non-negative.

Exercise 40 (RHS ranging — which constraint is most valuable?). A production LP has three resources with shadow prices $u_1^* = 8$, $u_2^* = 0$, $u_3^* = 5$.

1. Which resource should the manager invest in to increase profit most efficiently?
2. Resource 3 has a current capacity of 100 units. If an additional 4 units are available at a cost of \$18 total, is this economically worthwhile?
3. What happens if the capacity of Resource 2 increases by 10?

Exercise 41 (Objective coefficient ranging — basic variable). For the LP:

$$\begin{aligned} \text{maximize} \quad & c_1 x_1 + 4x_2 \\ \text{subject to} \quad & 2x_1 + x_2 \leq 8 \\ & x_1 + 3x_2 \leq 9 \\ & x_1, x_2 \geq 0, \end{aligned}$$

suppose the current optimal basis is $B = \{x_1, x_2\}$ with $c_1 = 6$ and optimal solution $(x_1^*, x_2^*) = (3, 2)$.

1. For what range of c_1 does the current basis remain optimal?
2. What happens to the optimal *solution* (not just the objective value) as c_1 changes within this range?
3. What happens if c_1 exceeds the upper limit of the range?

Hint: The current basis remains optimal as long as all reduced costs of non-basic variables stay ≤ 0.

Exercise 42 (Objective coefficient ranging — non-basic variable). At the optimum of a maximisation LP, variable x_3 is non-basic (at zero) with reduced cost $\bar{c}_3 = -4$.

1. The objective coefficient c_3 can increase by how much before x_3 becomes attractive to enter the basis?
2. If c_3 increases by more than this amount, what happens?
3. If c_3 decreases, does the current basis change? Why or why not?

Exercise 43 (Adding a new variable — sensitivity perspective). At the optimum of:

$$\begin{aligned} \text{maximize} \quad & 4x_1 + 5x_2 \\ \text{subject to} \quad & x_1 + x_2 \leq 6 \\ & 2x_1 + x_2 \leq 10 \\ & x_1, x_2 \geq 0 \end{aligned}$$

with optimal dual $(u_1^*, u_2^*) = (3, 1)$, a new product x_3 is proposed with profit \$7 and resource usage $(a_{13}, a_{23}) = (2, 1)$.

1. Compute the reduced cost of x_3 using the dual variables.
2. Should x_3 be produced? Justify using duality.
3. If the profit of x_3 were \$5 instead, what would the answer be?

Exercise 44 (Sensitivity analysis: simultaneous RHS changes). The 100% rule for simultaneous RHS changes states that if each change is within $\theta_i \in [0, 1]$ of its individual range, and $\sum_i \theta_i \leq 1$, then the current basis remains optimal.

1. State the rule precisely and explain its use.
2. Suppose resource 1 can range from $b_1 \in [20, 40]$ (current $b_1 = 30$) and resource 2 can range from $b_2 \in [10, 25]$ (current $b_2 = 15$) while maintaining the current basis. You want to change b_1 to 38 and b_2 to 22 simultaneously. Apply the 100% rule.

Exercise 45 (Interpretation of the dual problem as pricing). An entrepreneur rents out capacity on machines owned by a factory. The factory's LP is $\max\{c^\top x : Ax \leq b,\ x \geq 0\}$, with dual $\min\{b^\top u : A^\top u \geq c,\ u \geq 0\}$.

1. Explain why the dual objective $b^\top u$ represents the total rental revenue if the entrepreneur charges u_i per unit of resource i.
2. The dual constraint $A^\top u \geq c$ says that for every product j, the cost of the resources used to make one unit of product j is at least c_j. What does this mean for the factory's decision to produce product j?
3. Why does the entrepreneur want to minimise $b^\top u$? What is the entrepreneur's incentive relative to the factory's?

Exercise 46 (True/False: duality fundamentals). For each statement, write **True** or **False** and provide a brief justification (one or two sentences, or a counterexample):

(a) "The dual of a maximisation LP is always a minimisation LP."
(b) "If the primal optimal value is $z^* = 0$, then the dual optimal value is also 0."
(c) "A primal LP and its dual always have the same number of variables."
(d) "Strong duality holds for every LP."
(e) "If both the primal and dual are feasible, then both are bounded and strong duality holds."

Exercise 47 (True/False: complementary slackness). Classify each statement as **True** or **False**:

(a) "If $x_j^* > 0$ in the primal optimal, then the j-th dual constraint must be tight."
(b) "If the i-th primal constraint is not binding, the dual variable u_i^* must be zero."
(c) "A primal-dual pair (x, u) is optimal if and only if both are feasible and all CS conditions hold."
(d) "Complementary slackness conditions are necessary but not sufficient for optimality."
(e) "If all primal constraints are equalities, then CS imposes no restriction on the dual variables."

Exercise 48 (True/False: shadow prices and sensitivity). Classify each statement as **True** or **False**:

(a) "A shadow price of zero means the corresponding constraint can be removed without changing the optimal solution."
(b) "Shadow prices are valid only for marginal (infinitesimally small) changes

in the RHS."

(c) "In a maximisation LP, all shadow prices of $\leq$ constraints are non-negative."

(d) "The shadow price of a non-binding constraint is always zero."

(e) "Sensitivity ranges for objective coefficients and RHS values are independent of each other."

Exercise 49 (True/False: infeasibility and unboundedness). Classify each statement as **True** or **False**:

(a) "If the primal is infeasible, the dual must be unbounded."

(b) "If the primal is unbounded, the dual must be infeasible."

(c) "Both the primal and the dual can be simultaneously infeasible."

(d) "If the dual is unbounded, the primal is infeasible."

(e) "Farkas' Lemma provides a certificate of infeasibility for any infeasible system $Ax \leq b$, $x \geq 0$."

Exercise 50 (True/False: valid inequalities and Farkas). Classify each statement as **True** or **False**:

(a) "Every non-negative combination of the constraints of an LP yields a valid inequality."

(b) "A valid inequality that is tight at the optimum is always a constraint of the original LP."

(c) "The dual constraints can be interpreted as valid inequalities for the primal."

(d) "Farkas' Lemma implies that if a linear system has no solution, there is always a *linear* certificate of this fact."

(e) "Valid inequalities are only useful for integer programming, not for LP."

Exercise 51 (Bounding the optimal value via duality). Consider the LP:

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 4x_2 + 2x_3 \\ \text{subject to} \quad & x_1 + x_2 + x_3 \leq 8 \\ & 2x_1 + x_2 \leq 10 \\ & x_1 + 2x_3 \leq 7 \\ & x_1, x_2, x_3 \geq 0. \end{aligned}$$

1. Without solving the LP, find a feasible primal solution and compute a lower bound on z^*.
2. Write the dual and find, by inspection, a dual feasible solution. This gives an upper bound on z^*.
3. Can you tighten either bound? Report the best bounds you can find without running Simplex.
4. What would it mean if your lower and upper bounds coincide?

Exercise 52 (Primal-dual relationship: correspondence table). Fill in the following correspondence table for a max primal / min dual LP pair. For each row, state the condition in the dual that corresponds to the given primal feature:

Primal (max)	Dual (min)
$\leq$ constraint i	$u_i \geq 0$
$\geq$ constraint i	?
$=$ constraint i	?
Variable $x_j \geq 0$	$\geq$ dual constraint j
Variable $x_j \leq 0$	?
Variable x_j free	?

Complete the missing entries and verify with a specific example.

Exercise 53 (Degeneracy and the dual). A maximisation LP has the following primal optimal BFS: $x^* = (4, 0, 2, 0, 0)$ where x_1, x_3 are basic but $x_3^* = 0$ (degenerate). The problem has two constraints.

1. How many dual variables are there?
2. At the degenerate primal BFS, write out the complementary slackness conditions. How many equations do you obtain from the dual CS? How many from the primal CS?
3. Can you uniquely determine the dual optimal solution from CS alone? Explain why or why not.
4. Describe what the set of dual optimal solutions looks like in this case (geometrically).

Exercise 54 (Verifying dual feasibility). For the LP:

$$\begin{aligned} \text{maximize} \quad & 8x_1 + 5x_2 \\ \text{subject to} \quad & x_1 \leq 6 \\ & 2x_2 \leq 8 \\ & x_1 + x_2 \leq 9 \\ & x_1, x_2 \geq 0, \end{aligned}$$

a claimed dual optimal solution is $(u_1^*, u_2^*, u_3^*) = (5, 1, 3)$.

1. Write all dual constraints explicitly.
2. Verify that $(5, 1, 3)$ satisfies each dual constraint.
3. Compute $b^\top u^*$.
4. The primal optimal is $(x_1^*, x_2^*) = (6, 3)$. Verify $c^\top x^* = b^\top u^*$ and check all CS conditions.

Exercise 55 (Constructing a valid inequality upper bound). The furniture workshop LP is:

$$\begin{aligned} \text{maximize} \quad & 60x_1 + 50x_2 \\ \text{subject to} \quad & 2x_1 + x_2 \leq 100 \quad \text{(wood, m}^2\text{)} \\ & x_1 + x_2 \leq 60 \quad \text{(labour, h)} \\ & x_1 \leq 40 \quad \text{(demand for A)} \\ & x_1, x_2 \geq 0. \end{aligned}$$

1. Take a non-negative linear combination $y_1 \cdot$(constraint 1)$+ y_2 \cdot$(constraint 2)$+$ $y_3 \cdot$ (constraint 3) and choose $y_1, y_2, y_3 \geq 0$ to obtain a valid inequality of the form $60x_1 + 50x_2 \leq M$ for some constant M.
2. What is the tightest (smallest) M you can produce this way?
3. The dual LP will give exactly the minimum such M. Write the dual LP and verify this claim.

CHAPTER 6

Solving ILPs

This chapter moves from formulating *ILPs to actually* solving *them. Two big ideas: divide the problem (B&B) or tighten the relaxation (cutting planes).*

In chapter 3 we formulated the running knapsack (theorem 1.10.1) as an ILP. Its LP relaxation (theorem 3.3.7) has the fractional optimum $z^*_{\text{LP}} = 174/7 \approx 24.86$, whereas the integer optimum is $z^*_{\text{ILP}} = 24$ (theorem 3.8.3). There we certified optimality by combining the LP bound with a matching feasible solution. Here we develop systematic algorithms that do not require an optimal integer solution to be known in advance.

In chapter 5 we learned that duality is the systematic engine behind such bounds. Now we ask the next question: *given* that LP relaxation bounds are useful, how do we actually close the gap and find a provably optimal integer solution?

This chapter presents two complementary strategies:

1. **Branch and Bound**: partition the feasible region into smaller subproblems, using LP bounds to prune branches that cannot improve.
2. **Cutting Planes**: tighten the LP relaxation by adding valid inequalities that cut off fractional solutions, without removing any integer point.

Modern ILP solvers (CPLEX, Gurobi, SCIP) combine both in a framework called **Branch and Cut**. By the end of this chapter we will understand why.

Road map.

1. Why ILP is hard (section 6.1).
2. Branch and Bound: the idea (section 6.2).
3. The B&B algorithm (section 6.3).
4. Worked example: B&B on the running knapsack (section 6.4).
5. Branching and search strategies (section 6.5).
6. Cutting Planes: the idea (section 6.6).
7. Chvátal–Gomory inequalities (section 6.7).
8. Gomory cuts from the Simplex tableau (section 6.8).
9. The Cutting Plane algorithm (section 6.9).
10. Branch and Cut (section 6.10).

6.1 The ILP Challenge

LP is in P. ILP is NP-hard. That gap drives everything in this chapter.

Let us recall the fundamental difficulty. As we noted in chapter 3, ILP is **NP-hard**: no polynomial-time algorithm is known, and most researchers believe none exists. This means that, unlike the Simplex method for LP, we cannot hope for an algorithm that is fast on *every* instance.

But "NP-hard" does not mean "unsolvable." It means we need algorithms

that are smart about which parts of the search space to explore. The two main paradigms are:

- **Enumerate intelligently**: explore parts of the feasible region, but use bounds to skip large portions that provably contain no improving solution. This is *Branch and Bound*.
- **Tighten the relaxation**: add constraints ("cuts") to the LP relaxation so that its optimum moves closer to the ILP optimum—ideally until the LP solution *is* integer. This is *Cutting Planes*.

Both strategies rely on the LP relaxation bound from theorem 3.8.2: for a maximisation ILP,

$$z^*_{\text{ILP}} \;\leq\; z^*_{\text{LP}}.$$

Branch and Bound uses this bound to prune; Cutting Planes uses it as a target to tighten. Let us start with Branch and Bound.

6.2 Branch and Bound: The Idea

6.2.1 Divide and conquer

Branch and Bound = divide and conquer + bounding. The "bound" part is what makes it practical.

The brute-force approach to an ILP with n binary variables is to enumerate all 2^n possible solutions. Branch and Bound (B&B) is a *structured enumeration* that avoids exploring most of them.

The core idea has two components:

1. **Branching** (divide): split the current problem into two or more *subproblems* with disjoint feasible regions whose union covers the original region. No integer solution is lost.

2. **Bounding** (conquer): at each subproblem, solve the LP relaxation. The LP optimum is an upper bound on the best integer solution in that subproblem. If the bound is not promising, we discard the subproblem entirely.

6.2.2 Branching on a fractional variable

The most common branching rule works as follows. Suppose the LP relaxation of the current subproblem has optimal solution $\hat{x}$, and some variable x_j has a fractional value $\hat{x}_j \notin \mathbb{Z}$. We create two child subproblems:

- **Left child**: add the constraint $x_j \leq \lfloor \hat{x}_j \rfloor$.
- **Right child**: add the constraint $x_j \geq \lceil \hat{x}_j \rceil$.

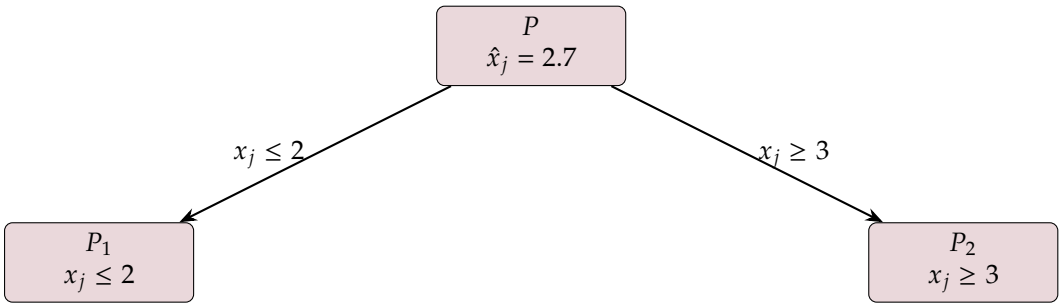


Figure 6.1: Branching on a fractional variable $x_j = 2.7$. The left child forces $x_j \leq 2$; the right child forces $x_j \geq 3$. Every integer value of x_j is in exactly one child.

Since every integer lies either at or below $\lfloor \hat{x}_j \rfloor$ or at or above $\lceil \hat{x}_j \rceil$, no integer feasible point is lost. But the fractional solution $\hat{x}$ is excluded from *both* children (since $\lfloor 2.7 \rfloor = 2 < 2.7 < 3 = \lceil 2.7 \rceil$). This guarantees progress.

6.2.3 The B&B tree

Repeated branching creates a **binary tree** of subproblems. Each node corresponds to a subproblem (the original ILP plus extra bound constraints from branching decisions above it in the tree). At each node we solve the LP relaxation.

Definition 6.2.1 (B&B tree)**.** The **Branch and Bound tree** is a rooted binary tree where:

- The **root** is the original ILP.
- Each **internal node** is a subproblem obtained by adding bound constraints from branching.
- Each **edge** represents a branching constraint ($x_j \leq k$ or $x_j \geq k+1$).
- Each node stores the LP relaxation bound for its subproblem.

6.2.4 Incumbent and fathoming

Two more concepts complete the picture.

Definition 6.2.2 (Incumbent)**.** The **incumbent** is the best integer feasible solution found so far during the B&B search. Its objective value serves as a **global lower bound** (for maximisation) on the optimum: $z^*_{\text{ILP}} \geq z_{\text{inc}}$.

Intuitively, the incumbent is our "best answer so far." Every time we discover a new integer feasible solution, we compare it to the incumbent and keep the better one. At the start, if no feasible solution is known, we set $z_{\text{inc}} = -\infty$.

Definition 6.2.3 (Fathoming (pruning))**.** A node in the B&B tree is **fathomed** (pruned) if we can guarantee it contains no integer solution better than the incumbent. There are three fathoming rules:

1. **Infeasibility**: the LP relaxation of the subproblem is infeasible. Then no integer solution exists in this branch.
2. **Bound**: the LP relaxation optimal value $\bar{z} \leq z_{\text{inc}}$. Even the best possible solution in this branch cannot beat the incumbent.
3. **Integrality**: the LP relaxation optimal solution $\hat{x}$ is integer. No further branching is needed; update the incumbent if $c^\top \hat{x} > z_{\text{inc}}$.

The power of B&B comes from rule 2. A single LP relaxation solve can eliminate an entire subtree containing exponentially many integer solutions—none of which need to be examined. The tighter the LP relaxation bounds, the more aggressive the pruning.

Three pruning rules: infeasible, bounded, integer. *A good mnemonic: IBI.*

6.3 The Branch and Bound Algorithm

Let us now state the full B&B algorithm for a maximisation ILP.

Algorithm 2: Branch and Bound (maximisation)

Input : ILP: maximize $c^\top x$ subject to $Ax \le b,\ x \ge 0,\ x \in \mathbb{Z}^n$
Output: Optimal integer solution x^*, or proof of infeasibility

```
1  Initialise: z_inc ← −∞, x* ← null
2  Solve the LP relaxation at the root
3  if root LP is infeasible then
4      return ILP is infeasible
5  if root LP solution x̂ is integer then
6      return x̂ is optimal with value c^⊤ x̂
7  Add root node to the open list ℒ
8  while ℒ ≠ ∅ do
9      Select a node N from ℒ (by search strategy)
10     Choose a fractional variable x_j in N's LP solution
11     for each child N' (with x_j ≤ ⌊x̂_j⌋ or x_j ≥ ⌈x̂_j⌉) do
12         Solve the LP relaxation of N'
13         if LP is infeasible then
14             Fathom N' (infeasibility)
15         else if LP optimum z̄ ≤ z_inc then
16             Fathom N' (bound)
17         else if LP solution x̂ is integer then
18             Update: z_inc ← c^⊤ x̂, x* ← x̂
19             Fathom N' (integrality)
20             Prune any open node with bound ≤ z_inc
21         else
22             Add N' to ℒ
23 return x* with value z_inc
```

Remark 6.3.1. When the incumbent is updated (line 16), we also scan the open list and prune any node whose bound is now $\le z_{\text{inc}}$. This retroactive pruning can eliminate many nodes at once.

Remark 6.3.2 (Mixed-Integer Programs). Branch and Bound applies directly to **mixed-integer programs (MIPs)**, where only a subset of variables are required to be integer and the rest are continuous. The only modification is that branching is performed exclusively on variables that carry an integrality constraint *and* are currently fractional in the LP relaxation; continuous variables are never branched on and are relaxed freely throughout the entire tree. All correctness and termination results carry over unchanged: the algorithm explores a finite tree, every integer-feasible region is covered, and the optimal MIP solution is found.

■ **Formal details — Correctness of Branch and Bound**

Theorem 6.3.3 (Correctness of B&B). *algorithm 2 terminates finitely and returns an optimal integer solution (or correctly reports infeasibility).*

Proof. **Termination.** Each branching step partitions an integer variable's domain into two strictly smaller parts. Since variables are bounded (by the LP constraints) and integer, each branch of the tree has finite depth. The tree is binary and finitely deep, hence finite. Every node is either fathomed or branched upon, so the algorithm explores at most finitely many nodes.

Optimality. Suppose the algorithm returns x^* with value z_{inc}. Every fathomed node either (i) contains no integer solution (infeasibility), (ii) contains no integer solution better than z_{inc} (bound pruning), or (iii) was already accounted for (integrality). Since every integer solution lives in at least one subproblem (branching is a partition), and all subproblems have been fathomed, no integer solution can have value $> z_{\text{inc}}$. Hence x^* is optimal. □

6.4 Worked Example: B&B on the Running Knapsack

We return to the 6-item knapsack from theorem 1.10.1. We now solve it with B&B.

Let us now walk through a complete B&B execution on our running knapsack instance (theorem 1.10.1). Recall the data: 6 items with weights $(5, 3, 7, 4, 2, 6)$, values $(8, 5, 11, 6, 4, 9)$, and capacity $W = 15$. The ILP is:

$$\begin{aligned} \text{maximize} \quad & 8\,x_1 + 5\,x_2 + 11\,x_3 + 6\,x_4 + 4\,x_5 + 9\,x_6 \\ \text{subject to} \quad & 5\,x_1 + 3\,x_2 + 7\,x_3 + 4\,x_4 + 2\,x_5 + 6\,x_6 \le 15 \\ & x_j \in \{0,1\}, \quad j = 1, \dots, 6. \end{aligned}$$

To illustrate the complete search process, we run B&B from scratch, without supplying an initial incumbent.

Example 6.4.1 (B&B on the running knapsack)**. Node 0 (Root).** We solve the LP relaxation (replacing $x_j \in \{0,1\}$ with $0 \le x_j \le 1$). From theorem 3.3.7, the LP optimum is:

$$x_1 = 1,\ x_2 = 1,\ x_3 = \tfrac{5}{7},\ x_4 = 0,\ x_5 = 1,\ x_6 = 0, \qquad \bar{z}_0 = \tfrac{174}{7} \approx 24.86.$$

The solution is fractional ($x_3 = 5/7$), so we cannot fathom. We set $z_{\text{inc}} = -\infty$ and branch on x_3.

Node 1 (Left child: $x_3 = 0$). We add $x_3 \le 0$, i.e., $x_3 = 0$. The subproblem becomes: maximize $8x_1 + 5x_2 + 6x_4 + 4x_5 + 9x_6$ subject to $5x_1 + 3x_2 + 4x_4 + 2x_5 + 6x_6 \le 15$.

Solving the LP relaxation by the greedy ratio method (items sorted by v_j/w_j: item 5, 2, 1, 6, 4 with ratios 2.00, 1.67, 1.60, 1.50, 1.50):

- Item 5: $x_5 = 1$, remaining $15 - 2 = 13$.
- Item 2: $x_2 = 1$, remaining $13 - 3 = 10$.
- Item 1: $x_1 = 1$, remaining $10 - 5 = 5$.
- Item 6: $w_6 = 6 > 5$, so $x_6 = 5/6$.
- Item 4: $x_4 = 0$.

LP solution: $x_1 = 1, x_2 = 1, x_5 = 1, x_6 = 5/6, x_3 = x_4 = 0$.

$$\bar{z}_1 = 8 + 5 + 4 + \tfrac{5}{6} \cdot 9 = 17 + \tfrac{45}{6} = 17 + 7.5 = 24.5.$$

Still fractional ($x_6 = 5/6$). Since $24.5 > z_{\text{inc}} = -\infty$, we cannot prune. We will branch on x_6 later.

Node 2 (Right child: $x_3 = 1$). We add $x_3 \ge 1$, i.e., $x_3 = 1$. The remaining capacity is $15 - 7 = 8$. Subproblem: maximize $8x_1 + 5x_2 + 6x_4 + 4x_5 + 9x_6 + 11$ subject to $5x_1 + 3x_2 + 4x_4 + 2x_5 + 6x_6 \le 8$.

Greedy LP relaxation (same ratio ordering among remaining items: 5, 2, 1, 6, 4):

- Item 5: $x_5 = 1$, remaining $8 - 2 = 6$.
- Item 2: $x_2 = 1$, remaining $6 - 3 = 3$.
- Item 1: $w_1 = 5 > 3$, so $x_1 = 3/5$.
- Items 6, 4: $x_6 = x_4 = 0$.

LP solution: $x_3 = 1, x_5 = 1, x_2 = 1, x_1 = 3/5, x_4 = x_6 = 0$.

$$\bar{z}_2 = 11 + 4 + 5 + \tfrac{3}{5} \cdot 8 = 20 + 4.8 = 24.8.$$

Still fractional ($x_1 = 3/5$). Not prunable.

We have two open nodes. Let us explore **Node 2** first (best-first strategy: $\bar{z}_2 = 24.8 > \bar{z}_1 = 24.5$). Branch on x_1.

Node 3 (Child of Node 2: $x_3 = 1, x_1 = 0$). Remaining capacity: $15 - 7 = 8$. Subproblem: maximize $5x_2 + 6x_4 + 4x_5 + 9x_6 + 11$ subject to $3x_2 + 4x_4 + 2x_5 + 6x_6 \leq 8$.

Greedy LP: items sorted by ratio: 5 (2.00), 2 (1.67), 6 (1.50), 4 (1.50):

- Item 5: $x_5 = 1$, remaining $8 - 2 = 6$.
- Item 2: $x_2 = 1$, remaining $6 - 3 = 3$.
- Item 6: $w_6 = 6 > 3$, so $x_6 = 3/6 = 1/2$.
- Item 4: $x_4 = 0$.

$$\bar{z}_3 = 11 + 4 + 5 + \tfrac{1}{2} \cdot 9 = 20 + 4.5 = 24.5.$$

Fractional ($x_6 = 1/2$). Not yet prunable.

Node 4 (Child of Node 2: $x_3 = 1, x_1 = 1$). Remaining capacity: $15 - 7 - 5 = 3$. Subproblem: maximize $5x_2 + 6x_4 + 4x_5 + 9x_6 + 19$ subject to $3x_2 + 4x_4 + 2x_5 + 6x_6 \leq 3$.

Greedy LP:

- Item 5: $x_5 = 1$, remaining $3 - 2 = 1$.
- Item 2: $w_2 = 3 > 1$, so $x_2 = 1/3$.
- Items 6, 4: $x_6 = x_4 = 0$.

$$\bar{z}_4 = 19 + 4 + \tfrac{1}{3} \cdot 5 = 23 + \tfrac{5}{3} \approx 24.67.$$

Fractional. Not yet prunable.

Let us continue with the best bound. We have open nodes: Node 1 (24.5), Node 3 (24.5), Node 4 (24.67). Explore Node 4. Branch on x_2.

Node 5 (Child of Node 4: $x_3 = 1, x_1 = 1, x_2 = 0$). Remaining capacity: $15 - 7 - 5 = 3$. Subproblem: maximize $6x_4 + 4x_5 + 9x_6 + 19$ subject to $4x_4 + 2x_5 + 6x_6 \leq 3$.

Greedy LP:

- Item 5: $x_5 = 1$, remaining $3 - 2 = 1$.
- Items 4 ($w_4 = 4$), 6 ($w_6 = 6$): neither fits. $x_4 = x_6 = 0$.

$$\hat{x} = (1, 0, 1, 0, 1, 0), \qquad \bar{z}_5 = 19 + 4 = 23.$$

Integer solution found! Update incumbent: $z_{\text{inc}} = 23$, $x^* = (1, 0, 1, 0, 1, 0)$.

Fathom Node 5 (integrality).

Node 6 (Child of Node 4: $x_3 = 1, x_1 = 1, x_2 = 1$). Remaining capacity: $15 - 7 - 5 - 3 = 0$. All remaining items forced to 0: $x_4 = x_5 = x_6 = 0$.

$$\hat{x} = (1, 1, 1, 0, 0, 0), \qquad \bar{z}_6 = 8 + 5 + 11 = 24.$$

Integer solution found! Since $24 > z_{\text{inc}} = 23$, update: $z_{\text{inc}} = 24$, $x^* = (1,1,1,0,0,0)$.

Fathom Node 6 (integrality).

Now we check the remaining open nodes against the new incumbent $z_{\text{inc}} = 24$:

- Node 1: $\bar{z}_1 = 24.5 > 24$. Not prunable; keep open.
- Node 3: $\bar{z}_3 = 24.5 > 24$. Not prunable; keep open.

Explore Node 3 (bound 24.5). Branch on x_6.

Node 7 (Child of Node 3: $x_3 = 1, x_1 = 0, x_6 = 0$). Remaining capacity: $15 - 7 = 8$. Subproblem: maximize $5x_2 + 6x_4 + 4x_5 + 11$ subject to $3x_2 + 4x_4 + 2x_5 \leq 8$.

Greedy LP:

- Item 5: $x_5 = 1$, remaining $8 - 2 = 6$.
- Item 2: $x_2 = 1$, remaining $6 - 3 = 3$.
- Item 4: $w_4 = 4 > 3$, so $x_4 = 3/4$.

$$\bar{z}_7 = 11 + 4 + 5 + \tfrac{3}{4} \cdot 6 = 20 + 4.5 = 24.5.$$

Fractional ($x_4 = 3/4$). Bound $24.5 > 24 = z_{\text{inc}}$. Branch on x_4.

Node 9 (Child of Node 7: $x_3 = 1, x_1 = 0, x_6 = 0, x_4 = 0$). Remaining capacity: 8. Maximize $5x_2 + 4x_5 + 11$ s.t. $3x_2 + 2x_5 \leq 8$.

- $x_5 = 1$ (remaining 6), $x_2 = 1$ (remaining 3).
- Capacity left: no more items.

$$\hat{x} = (0,1,1,0,1,0), \qquad \bar{z}_9 = 11 + 5 + 4 = 20.$$

Integer solution, but $20 < 24 = z_{\text{inc}}$. Fathom (bound: $20 \leq 24$).

Node 10 (Child of Node 7: $x_3 = 1, x_1 = 0, x_6 = 0, x_4 = 1$). Remaining capacity: $8 - 4 = 4$. Maximize $5x_2 + 4x_5 + 17$ s.t. $3x_2 + 2x_5 \leq 4$.

- $x_5 = 1$ (remaining 2), x_2: $w_2 = 3 > 2$, so $x_2 = 2/3$.

$$\bar{z}_{10} = 17 + 4 + \tfrac{2}{3} \cdot 5 = 21 + \tfrac{10}{3} \approx 24.33.$$

Fractional, but $24.33 > 24$. Branch on x_2.

Node 11 ($x_3 = 1, x_1 = 0, x_6 = 0, x_4 = 1, x_2 = 0$). Remaining capacity 4: $x_5 = 1$ (remaining 2). No more items fit.

$$\hat{x} = (0,0,1,1,1,0), \qquad \bar{z}_{11} = 11 + 6 + 4 = 21.$$

Integer, but $21 < 24$. Fathom (bound).

Node 12 ($x_3 = 1, x_1 = 0, x_6 = 0, x_4 = 1, x_2 = 1$). Remaining capacity $4 - 3 = 1$: x_5: $w_5 = 2 > 1$, so LP gives $x_5 = 1/2$.

$$\bar{z}_{12} = 17 + 5 + \tfrac{1}{2} \cdot 4 = 24.$$

Fractional, but $\bar{z}_{12} = 24 \leq 24 = z_{\text{inc}}$. **Fathom** (bound: cannot improve upon incumbent).

Node 8 (Child of Node 3: $x_3 = 1, x_1 = 0, x_6 = 1$). Remaining capacity: $15 - 7 - 6 = 2$. Subproblem: maximize $5x_2 + 6x_4 + 4x_5 + 20$ subject to $3x_2 + 4x_4 + 2x_5 \leq 2$.

Greedy LP: item 5 ($x_5 = 1$, remaining 0), all others = 0.

$$\hat{x} = (0,0,1,0,1,1), \qquad \bar{z}_8 = 20 + 4 = 24.$$

Integer solution with value 24! But $24 \leq z_{\text{inc}} = 24$, so it matches (not strictly better). This is an alternative optimum. Fathom.

Now turn to **Node 1** ($x_3 = 0$, bound 24.5). Branch on x_6.

Node 13 (Child of Node 1: $x_3 = 0, x_6 = 0$). Capacity 15. Maximize $8x_1 + 5x_2 + 6x_4 + 4x_5$ s.t. $5x_1 + 3x_2 + 4x_4 + 2x_5 \leq 15$.
Greedy LP: $x_5 = 1$ (rem. 13), $x_2 = 1$ (rem. 10), $x_1 = 1$ (rem. 5), $x_4 = 1$ (rem. 1). All integer!

$$\hat{x} = (1, 1, 0, 1, 1, 0), \qquad \bar{z}_{13} = 8 + 5 + 6 + 4 = 23.$$

Integer, but $23 < 24$. Fathom (bound).

Node 14 (Child of Node 1: $x_3 = 0, x_6 = 1$). Remaining capacity: $15 - 6 = 9$. Maximize $8x_1 + 5x_2 + 6x_4 + 4x_5 + 9$ s.t. $5x_1 + 3x_2 + 4x_4 + 2x_5 \leq 9$.
Greedy LP: $x_5 = 1$ (rem. 7), $x_2 = 1$ (rem. 4), x_1: $w_1 = 5 > 4$, so $x_1 = 4/5$.

$$\bar{z}_{14} = 9 + 4 + 5 + \tfrac{4}{5} \cdot 8 = 18 + 6.4 = 24.4.$$

Fractional. $\bar{z}_{14} = 24.4 > 24$. Branch on x_1.

Node 15 ($x_3 = 0, x_6 = 1, x_1 = 0$). Remaining capacity 9. $x_5 = 1$ (rem. 7), $x_2 = 1$ (rem. 4), $x_4 = 1$ (rem. 0).

$$\hat{x} = (0, 1, 0, 1, 1, 1), \qquad \bar{z}_{15} = 9 + 4 + 5 + 6 = 24.$$

Integer, $24 \leq z_{\text{inc}} = 24$. Another alternative optimum. Fathom.

Node 16 ($x_3 = 0, x_6 = 1, x_1 = 1$). Remaining capacity $9 - 5 = 4$. $x_5 = 1$ (rem. 2), x_2: $w_2 = 3 > 2$, so $x_2 = 2/3$.

$$\bar{z}_{16} = 9 + 8 + 4 + \tfrac{2}{3} \cdot 5 = 21 + \tfrac{10}{3} \approx 24.33.$$

Fractional, $24.33 > 24$. Branch on x_2.

Node 17 ($x_3 = 0, x_6 = 1, x_1 = 1, x_2 = 0$). Remaining capacity 4. $x_5 = 1$ (rem. 2), x_4: $w_4 = 4 > 2$, $x_4 = 1/2$.

$$\bar{z}_{17} = 9 + 8 + 4 + \tfrac{1}{2} \cdot 6 = 24.$$

$\bar{z}_{17} = 24 \leq z_{\text{inc}}$. Fathom (bound).

Node 18 ($x_3 = 0, x_6 = 1, x_1 = 1, x_2 = 1$). Remaining capacity $4 - 3 = 1$. No item fits ($w_4 = 4, w_5 = 2$): x_5 gets $x_5 = 1/2$ fractionally.

$$\bar{z}_{18} = 9 + 8 + 5 + \tfrac{1}{2} \cdot 4 = 24.$$

$\bar{z}_{18} = 24 \leq z_{\text{inc}}$. Fathom (bound).

All nodes fathomed. The open list is empty. The optimal solution is:

$$x^* = (1, 1, 1, 0, 0, 0), \quad z^* = 8 + 5 + 11 = 24.$$

Items 1, 2, and 3 are packed (total weight $5 + 3 + 7 = 15$, total value 24). The search also found the alternative optimal solutions $\{3, 5, 6\}$ and $\{2, 4, 5, 6\}$, both with value 24.

The B&B explored 19 nodes out of $2^6 = 64$ possible leaves. That is a factor-3 speedup even on this tiny instance.

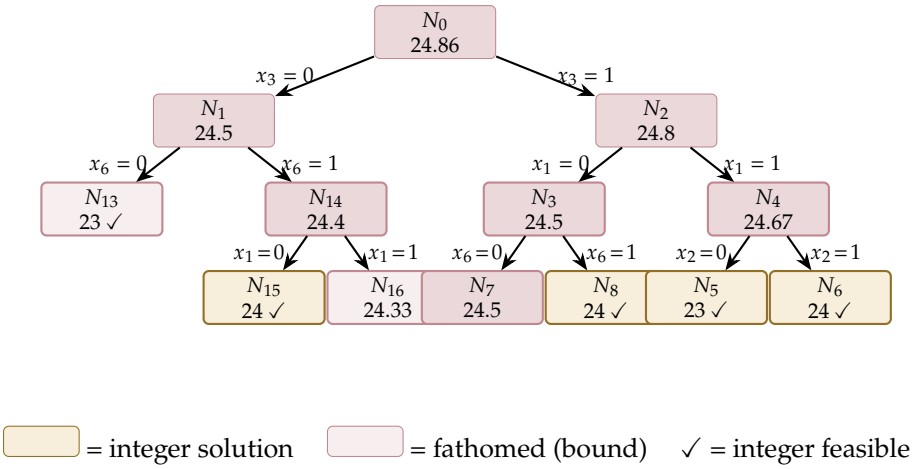


Figure 6.2: Partial B&B tree for the running knapsack. Node labels show the LP relaxation bound. Integer-solution nodes and fathomed nodes are highlighted with distinct palette tints. The optimal solution $\{1, 2, 3\}$ with value 24 is found at Node 6.

6.5 Branching and Search Strategies

The B&B algorithm leaves two choices unspecified: *which variable to branch on* and *which node to explore next*. These choices dramatically affect performance.

6.5.1 Variable selection (branching rules)

The branching variable determines the shape of the tree. A good choice leads to more pruning.

When the LP solution has multiple fractional variables, we must choose one to branch on. Common strategies include:

1. **Most fractional**: pick the variable whose fractional part is closest to 0.5. This maximises the "disruption" to the LP solution.
2. **Most impactful** (strong branching): for each candidate variable, tentatively solve both child LPs and pick the variable that yields the largest bound decrease. Expensive per node but can dramatically reduce the tree size.
3. **Pseudocost branching**: estimate the bound change per unit change in each variable, based on historical branching data. A practical compromise between quality and cost.
4. **Reliability branching**: use strong branching for the first few branchings on each variable; then switch to pseudocosts once estimates are reliable. This is the default in most modern solvers.

6.5.2 Node selection (search strategies)

Search strategy = the order in which we visit nodes in the B&B tree.

The *open list* $\mathcal{L}$ can be ordered in several ways:

1. **Depth-first (LIFO)**: always explore the most recently created node.
   - \+ Low memory (at most $O(n)$ open nodes for n variables).
   - \+ Finds feasible solutions quickly (goes deep fast).
   - – May explore many nodes before finding a good bound.

2. **Breadth-first (FIFO)**: explore nodes level by level.
   + Systematic: explores all nodes at depth d before depth $d+1$.
   – Exponential memory.
   – Slow to find feasible solutions.

3. **Best-first**: always explore the node with the best (highest, for maximisation) LP bound.
   + Minimises the total number of nodes explored (optimal in a certain sense).
   + Closes the gap fastest.
   – Higher memory than depth-first.
   – May delay finding integer solutions.

> *Remark* 6.5.1. In practice, solvers use a **hybrid** strategy: start with depth-first to find a good incumbent quickly, then switch to best-first to close the gap efficiently. The initial incumbent obtained by depth-first diving enables aggressive pruning in subsequent best-first exploration.

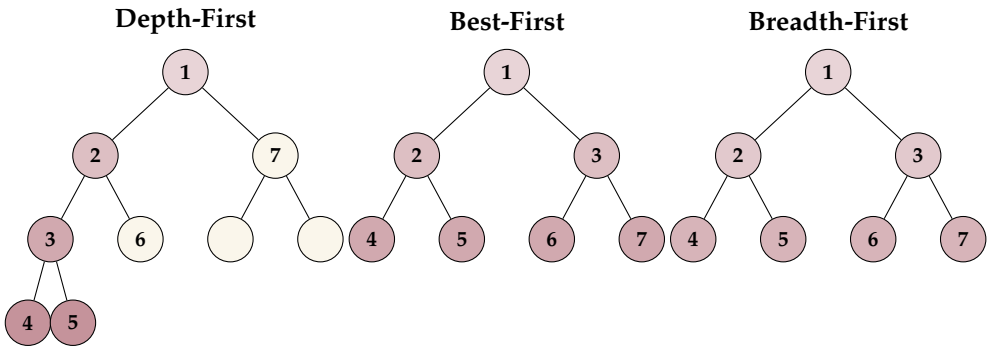


Figure 6.3: Three node selection strategies. Numbers indicate exploration order. Depth-first dives deep immediately; best-first follows the tightest bounds; breadth-first is systematic but slow.

6.5.3 B&B performance

> *Observation* 6.5.2. In the worst case, B&B explores $O(2^n)$ nodes (all possible subsets). In practice, good bounds and branching heuristics typically reduce this to a manageable number. The quality of the LP relaxation bound is the single most important factor: a tighter bound means more pruning and fewer nodes.

This observation motivates the second half of the chapter: if we can tighten the LP relaxation—by adding constraints that cut off fractional solutions without eliminating any integer point—then B&B becomes faster. This is exactly what cutting planes achieve.

6.6 Cutting Planes: The Idea

Instead of partitioning the problem (B&B), we tighten *the relaxation until the LP solution is integer.*

Branch and Bound works by *dividing* the feasible region. Cutting Planes take a completely different approach: instead of splitting the problem, we *strengthen* the LP relaxation by adding new constraints—called **cuts**—that are valid for all integer solutions but violated by the current fractional LP solution.

6.6.1 The convex hull ideal

Recall from chapter 3 that the **convex hull** $P(X) = \text{conv}(X)$ of the integer feasible set X is the "ideal" LP relaxation. If we could write down $P(X)$ explicitly, then solving the LP over $P(X)$ would give an integer optimal solution directly (theorem 3.8.8).

The problem is that $P(X)$ typically requires exponentially many inequalities to describe. We cannot write them all down. But we do not *need* all of them—we only need enough to force the LP optimum to be integer. The cutting plane approach adds these inequalities one at a time, driven by the current fractional solution.

6.6.2 Valid cuts

Definition 6.6.1 (Valid inequality)**.** Given an ILP with integer feasible set $X = \{x \in \mathbb{Z}^n : Ax \leq b,\ x \geq 0\}$, an inequality $\pi^\top x \leq \pi_0$ is a **valid inequality** (or **valid cut**) for X if every $x \in X$ satisfies it:

$$\pi^\top x \leq \pi_0 \quad \forall\, x \in X.$$

A valid cut is *useful* if it is violated by the current LP relaxation solution $\hat{x}$: $\pi^\top \hat{x} > \pi_0$. Adding such a cut tightens the LP relaxation without removing any integer feasible point.

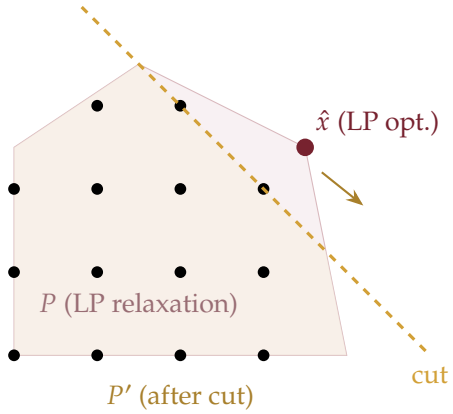


Figure 6.4: A cutting plane in action. The light wine region is the LP relaxation polyhedron P. The gold dashed line is a valid cut that removes the fractional LP optimum $\hat{x}$ without excluding any integer point (black dots). The tightened region P' uses a matching warm tint and has a better LP bound.

6.6.3 The separation problem

Definition 6.6.2 (Separation problem)**.** Given a point $\hat{x} \in \mathbb{R}^n$ and an integer

feasible set X, the **separation problem** asks: either certify that $\hat{x} \in \text{conv}(X)$, or find a valid inequality $\pi^\top x \le \pi_0$ that separates $\hat{x}$ from $\text{conv}(X)$ (i.e., $\pi^\top \hat{x} > \pi_0$).

The separation problem is the engine of cutting plane methods: each iteration, we solve it to find a violated cut, add the cut, and re-solve the LP. The key question is: how do we systematically *generate* valid cuts? The next two sections answer this.

Separation is the dual of optimisation: if you can separate efficiently, you can optimise efficiently, and vice versa (the ellipsoid method makes this precise).

6.7 Chvátal–Gomory Inequalities

We now develop a systematic method for generating valid inequalities for any ILP. The construction, due to Chvátal, is beautifully simple.

Chvátal's elegant idea: combine constraints, round coefficients, round the RHS. Three steps to a valid cut.

6.7.1 A motivating example

Consider a small ILP:

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 2x_2 \\ \text{subject to} \quad & 2x_1 + 3x_2 \le 14 \\ & 4x_1 + x_2 \le 16 \\ & x_1, x_2 \ge 0, \quad x_1, x_2 \in \mathbb{Z}. \end{aligned} \tag{6.1}$$

The LP relaxation optimum is at $x_1 = 34/10 = 3.4$, $x_2 = 24/10 = 2.4$ with $z^*_{\text{LP}} = 3(3.4) + 2(2.4) = 15$.

Example 6.7.1 (Deriving a Chvátal cut)**.** Take $u_1 = 1/2$ times the first constraint:

$$x_1 + \tfrac{3}{2}x_2 \le 7.$$

This is valid for all (real) feasible points.

Now **round down** the LHS coefficients. Since $x_1, x_2 \ge 0$ and integer, replacing $3/2$ by $\lfloor 3/2 \rfloor = 1$ only makes the LHS smaller:

$$x_1 + x_2 \le 7.$$

Can we tighten the RHS too? No. Since $x_1 + x_2$ is an *integer* (sum of two integers), and $x_1 + x_2 \le 7$, this already has an integer RHS, so no further rounding helps here.

Let's try a more interesting combination. Take $u_1 = 1/3$ of constraint 1 and $u_2 = 1/3$ of constraint 2:

$$\tfrac{2}{3}x_1 + x_2 + \tfrac{4}{3}x_1 + \tfrac{1}{3}x_2 = 2x_1 + \tfrac{4}{3}x_2 \le \tfrac{14}{3} + \tfrac{16}{3} = 10.$$

Round LHS coefficients down: $\lfloor 2 \rfloor = 2$, $\lfloor 4/3 \rfloor = 1$:

$$2x_1 + x_2 \le 10.$$

Now $2x_1 + x_2$ is integer for integer (x_1, x_2), and $2x_1 + x_2 \le 10$. Since the RHS is already integer, this is our cut: $2x_1 + x_2 \le 10$.

Check: at the LP optimum $(3.4, 2.4)$: $2(3.4) + 2.4 = 9.2 \le 10$. This cut is not violated—we need a different combination.

Take $u_1 = 1/2, u_2 = 1/4$:

$$(1+1)\,x_1 + (\tfrac{3}{2} + \tfrac{1}{4})\,x_2 = 2\,x_1 + \tfrac{7}{4}\,x_2 \le 7 + 4 = 11.$$

Round LHS: $2x_1 + x_2 \le 11$. Round RHS: already integer. At LP optimum: $2(3.4) + 2.4 = 9.2 \le 11$. Still not violated.

Let's try $u_1 = 1/3, u_2 = 0$:

$$\tfrac{2}{3}\,x_1 + x_2 \le \tfrac{14}{3} \approx 4.67.$$

Round LHS: $0 \cdot x_1 + x_2 \le 14/3$. Actually, let's keep $\lfloor 2/3 \rfloor = 0$: $x_2 \le 14/3$. Round RHS: $x_2 \le \lfloor 14/3 \rfloor = 4$.

At LP optimum: $x_2 = 2.4 \le 4$. Not violated.

The point is the *method*, not finding the perfect cut in every attempt. Let's now state the general procedure.

6.7.2 The Chvátal–Gomory procedure

Theorem 6.7.2 (Chvátal–Gomory rounding)**.** *Consider an ILP with constraints $Ax \le b$, $x \ge 0$, $x \in \mathbb{Z}^n$. Let $u \ge 0$ be any vector of non-negative multipliers. Then the inequality*

$$\lfloor u^\top A \rfloor\, x \;\le\; \lfloor u^\top b \rfloor$$

is valid for all integer feasible points.

Proof. Let x^* be any integer feasible point. Since $Ax^* \le b$ and $u \ge 0$:

$$u^\top A\, x^* \le u^\top b.$$

Since $x^* \ge 0$ and $\lfloor u^\top A \rfloor_j \le (u^\top A)_j$ for every j:

$$\lfloor u^\top A \rfloor\, x^* \le u^\top A\, x^* \le u^\top b.$$

Since $x^* \in \mathbb{Z}^n$, the LHS $\lfloor u^\top A \rfloor\, x^*$ is an integer. An integer $\le u^\top b$ is also $\le \lfloor u^\top b \rfloor$. □

The rounding trick works because the LHS is guaranteed to be integer, so we can safely round down the RHS.

Intuitively, the procedure works in three steps:

1. **Combine**: take a non-negative combination $u^\top A\, x \le u^\top b$ of the original constraints.
2. **Round the LHS**: replace each coefficient by its floor. Since $x \ge 0$, this only weakens the LHS.
3. **Round the RHS**: since the LHS is now integer-valued (for $x \in \mathbb{Z}^n$), round down the RHS. This is the powerful step that creates inequalities not implied by the LP relaxation.

Definition 6.7.3 (Chvátal closure)**.** The **Chvátal closure** of a polyhedron $P = \{x \in \mathbb{R}^n : Ax \le b,\ x \ge 0\}$ is the set $P' = P \cap \{\text{all rank-1 CG inequalities}\}$. That is, P' is obtained by adding *every* inequality derivable by one application of theorem 6.7.2 (for all $u \ge 0$).

The Chvátal closure P' is itself a polyhedron (this is non-trivial to prove), and $\text{conv}(X) \subseteq P' \subseteq P$. By iterating—taking the Chvátal closure of P', then of P'', and so on—we get a nested sequence of polyhedra that converges to $\text{conv}(X)$.

Theorem 6.7.4 (Chvátal's theorem)**.** *For any rational polyhedron P and integer feasible set $X = P \cap \mathbb{Z}^n$, the iterated Chvátal closure procedure converges to* $\mathrm{conv}(X)$ *after finitely many rounds.*

Chvátal's theorem says CG cuts are complete *in the sense that repeated closure reaches the integer hull. The number of rounds can be very large.*

This is a remarkable completeness result: repeated CG closure reaches the integer hull, so it recovers exactly the inequalities that are valid for the ILP. In practice, however, we do not apply the full closure; we generate specific cuts as needed.

6.8 Gomory Cuts from the Simplex Tableau

Gomory's insight: the optimal Simplex tableau already contains all the information needed to generate a cut.

The CG procedure of section 6.7 tells us that valid cuts *exist*, but it does not tell us how to find one that is violated by the current LP solution. Gomory's **fractional cutting plane** solves this problem elegantly: it reads a violated cut directly from the optimal Simplex tableau.

6.8.1 Setup: the Simplex tableau

Recall from chapter 4 that the optimal Simplex tableau (theorem 4.6.1) expresses each basic variable as a linear combination of the non-basic variables. For basic variable $x_{\mathcal{B}(i)}$, the i-th row of the tableau reads:

$$x_{\mathcal{B}(i)} + \sum_{j \in \mathcal{N}} \bar{a}_{ij}\, x_j = \bar{b}_i, \tag{6.2}$$

where $\bar{a}_{ij}$ and $\bar{b}_i$ are the updated coefficients after Simplex pivoting. At the optimal BFS, all non-basic variables are zero ($x_j = 0$ for $j \in \mathcal{N}$), so $x_{\mathcal{B}(i)} = \bar{b}_i$.

If $\bar{b}_i$ is not integer, then the current BFS is fractional. This is exactly the row we will use to generate a Gomory cut.

6.8.2 Deriving the Gomory cut

Let $f_0 = \bar{b}_i - \lfloor \bar{b}_i \rfloor$ denote the fractional part of $\bar{b}_i$, and for each non-basic variable j, let $f_j = \bar{a}_{ij} - \lfloor \bar{a}_{ij} \rfloor$ denote the fractional part of $\bar{a}_{ij}$. Note that $0 < f_0 < 1$ (since $\bar{b}_i$ is fractional).

Theorem 6.8.1 (Gomory's fractional cut)**.** *The inequality*

$$\sum_{j \in \mathcal{N}} f_j\, x_j \;\geq\; f_0 \tag{6.3}$$

is valid for all integer solutions of the ILP, and is violated by the current fractional BFS.

Proof. From the tableau row (6.2):

$$x_{\mathcal{B}(i)} = \bar{b}_i - \sum_{j \in \mathcal{N}} \bar{a}_{ij}\, x_j.$$

Decompose each $\bar{a}_{ij} = \lfloor \bar{a}_{ij} \rfloor + f_j$ and $\bar{b}_i = \lfloor \bar{b}_i \rfloor + f_0$:

$$x_{\mathcal{B}(i)} = \lfloor \bar{b}_i \rfloor + f_0 - \sum_{j \in \mathcal{N}} \big(\lfloor \bar{a}_{ij} \rfloor + f_j \big)\, x_j.$$

Rearranging:

$$\underbrace{x_{\mathcal{B}(i)} - \lfloor \bar{b}_i \rfloor + \sum_{j \in \mathcal{N}} \lfloor \bar{a}_{ij} \rfloor \, x_j}_{\text{integer}} = f_0 - \sum_{j \in \mathcal{N}} f_j \, x_j.$$

The LHS is an integer (since $x_{\mathcal{B}(i)} \in \mathbb{Z}$ and all $\lfloor \bar{a}_{ij} \rfloor \in \mathbb{Z}$). Therefore the RHS must also be an integer. We have $0 < f_0 < 1$ and $0 \le f_j < 1$, and all $x_j \ge 0$.

If $\sum_j f_j x_j < f_0$, then $0 < f_0 - \sum_j f_j x_j < 1$ (since $f_0 < 1$ and $\sum_j f_j x_j \ge 0$). But the RHS must be integer, and the only integer in $(0, 1)$ is. . . none. Contradiction.

Therefore $\sum_j f_j x_j \ge f_0$.

Violated by current BFS: at the BFS, $x_j = 0$ for all $j \in \mathcal{N}$, so $\sum_j f_j x_j = 0 < f_0$. The cut is violated. □

The Gomory cut is both valid *(no integer point violates it) and* effective *(the current LP optimum violates it). Perfect.*

6.8.3 Worked example: Gomory cut

Example 6.8.2 (Gomory cut for a 2-variable ILP)**.** Consider the ILP:

$$\begin{aligned} &\text{maximize} && 5\,x_1 + 4\,x_2 \\ &\text{subject to} && 3\,x_1 + 2\,x_2 \le 14 \\ &&& x_1 + 4\,x_2 \le 12 \\ &&& x_1, x_2 \ge 0, \quad x_1, x_2 \in \mathbb{Z}. \end{aligned} \tag{6.4}$$

Step 1: Solve the LP relaxation. Adding slacks $s_1, s_2 \ge 0$:

$$3\,x_1 + 2\,x_2 + s_1 = 14, \quad x_1 + 4\,x_2 + s_2 = 12.$$

The optimal Simplex tableau (after pivoting) is:

	x_1	x_2	s_1	s_2	RHS
x_1	1	0	2/5	−1/5	16/5
x_2	0	1	−1/10	3/10	11/5
z	0	0	−8/5	−1/5	124/5

Let us verify: from the first row, $x_1 = 16/5 = 3.2$; from the second row, $x_2 = 11/5 = 2.2$. Objective: $5(16/5) + 4(11/5) = 16 + 44/5 = 124/5 = 24.8$.

Both $x_1 = 3.2$ and $x_2 = 2.2$ are fractional. We generate a Gomory cut from row 1 (x_1 row).

Step 2: Identify fractional parts. Row 1: $x_1 + \frac{2}{5}\,s_1 - \frac{1}{5}\,s_2 = \frac{16}{5}$.

Non-basic variables: s_1, s_2.

- $\bar{b}_1 = 16/5 = 3.2$, so $f_0 = 0.2$.
- $\bar{a}_{1,s_1} = 2/5 = 0.4$, so $f_{s_1} = 0.4 - \lfloor 0.4 \rfloor = 0.4$.
- $\bar{a}_{1,s_2} = -1/5 = -0.2$, so $f_{s_2} = -0.2 - \lfloor -0.2 \rfloor = -0.2 - (-1) = 0.8$.

Step 3: Write the Gomory cut.

$$0.4\,s_1 + 0.8\,s_2 \ge 0.2, \quad \text{or equivalently} \quad 2\,s_1 + 4\,s_2 \ge 1.$$

Step 4: Verify violation. At the current BFS: $s_1 = s_2 = 0$, so LHS $= 0 < 0.2 = f_0$. The cut is violated. ✓

Step 5: Express in original variables. Since $s_1 = 14 - 3x_1 - 2x_2$ and $s_2 = 12 - x_1 - 4x_2$:

$$2(14 - 3x_1 - 2x_2) + 4(12 - x_1 - 4x_2) \geq 1$$

$$28 - 6x_1 - 4x_2 + 48 - 4x_1 - 16x_2 \geq 1$$

$$76 - 10x_1 - 20x_2 \geq 1 \quad \Longrightarrow \quad 10x_1 + 20x_2 \leq 75.$$

Simplifying: $2x_1 + 4x_2 \leq 15$.

At the LP optimum $(3.2, 2.2)$: $2(3.2) + 4(2.2) = 6.4 + 8.8 = 15.2 > 15$ — the LP optimum *violates* the cut, confirming it is effective.

After adding the Gomory cut and re-solving the LP, the new optimum will have x_1 closer to an integer.

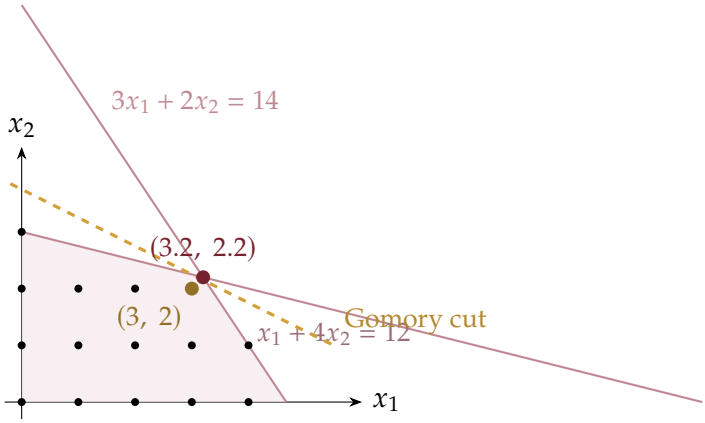


Figure 6.5: Gomory cut for theorem 6.8.2. The LP relaxation optimum is at $(3.2, 2.2)$. The Gomory cut removes the fractional optimum. The ILP optimum at $(3, 2)$ with value 23 lies inside the tightened region.

Remark 6.8.3. In theorem 6.8.2, the ILP optimum is at $(3, 2)$ with value $5(3) + 4(2) = 23$. After adding the Gomory cut and re-solving, the LP solution moves toward this point. One or two more cuts would close the gap entirely.

6.9 The Cutting Plane Algorithm

We now have all the ingredients for a complete cutting plane algorithm.

6.9.1 Why dual Simplex?

Adding a cut keeps dual feasibility but destroys primal feasibility—exactly the setup for dual Simplex.

When we add a Gomory cut $\sum_j f_j x_j \geq f_0$, we introduce it as $\sum_j f_j x_j - s = f_0$ with surplus variable $s \geq 0$. In the current BFS, $s = -f_0 < 0$—the new constraint is violated, so primal feasibility is lost. However, the reduced costs from the previous optimal tableau are still non-positive (for maximisation), so **dual feasibility is maintained**. This is exactly the setup where the **dual Simplex method** excels: it restores primal feasibility while maintaining dual feasibility, typically in very few pivots.

Algorithm 3: Gomory's Cutting Plane Algorithm

Input : Pure ILP: maximize $c^\top x$ s.t. $Ax \le b$, $x \ge 0$, $x \in \mathbb{Z}^n$
Output: Optimal integer solution x^*

```
Solve the LP relaxation (by Simplex)
while LP solution x̂ is not integer do
    Select a row i with fractional b̄_i
    Compute the Gomory cut: Σ_{j∈N} f_j x_j ≥ f_0
    Add the cut as a new constraint (with a surplus variable)
    Re-solve the LP (use dual Simplex for efficiency)
return x̂
```

6.9.2 Convergence

Theorem 6.9.1 (Gomory's convergence theorem)**.** *For a **pure ILP** (all variables integer) with rational data, the cutting plane algorithm (algorithm 3) terminates after finitely many iterations, yielding an optimal integer solution (or proving infeasibility).*

The proof is technically involved and relies on lexicographic pivot rules to prevent cycling. We state it without proof.

Remark 6.9.2. In practice, pure Gomory cuts can converge slowly—the LP bound may improve by tiny amounts each iteration, requiring hundreds of cuts. This motivates combining cuts with Branch and Bound, leading to Branch and Cut.

Intermezzo — Gomory's discovery

Ralph Gomory developed his cutting plane method in 1958 while working at IBM. He was trying to solve scheduling problems for the U.S. Navy. The idea of deriving cuts from the Simplex tableau was strikingly original—it showed that the LP solver itself contains the information needed to enforce integrality. Although pure cutting planes proved slow in practice, Gomory's cuts became a cornerstone of modern ILP solvers when combined with Branch and Bound in the 1990s.

6.9.3 Worked example: full cutting plane execution

Example 6.9.3 (Cutting plane algorithm on the 2-variable ILP)**.** We continue with the ILP from theorem 6.8.2:

$$\text{maximize } 5x_1{+}4x_2 \quad \text{subject to} \quad 3x_1{+}2x_2 \le 14,\ \ x_1{+}4x_2 \le 12,\ \ x_1, x_2 \ge 0,\ \ x_1, x_2 \in \mathbb{Z}.$$

Iteration 0. Solve the LP relaxation. Optimal: $x_1 = 16/5 = 3.2$, $x_2 = 11/5 = 2.2$, $z = 124/5 = 24.8$. Not integer (both x_1 and x_2 are fractional).

Iteration 1. From the x_1 row of the tableau, we derived the Gomory cut $0.4\,s_1 + 0.8\,s_2 \ge 0.2$, or equivalently $2x_1 + 4x_2 \le 15$.

Add this cut and re-solve. The new LP includes

$$3x_1 + 2x_2 \leq 14, \quad x_1 + 4x_2 \leq 12, \quad 2x_1 + 4x_2 \leq 15.$$

The new LP optimum: $x_1 = 3, x_2 = 9/4 = 2.25, z = 5(3) + 4(2.25) = 24$.

Hmm, still not integer ($x_2 = 2.25$).

Iteration 2. After adding the first Gomory cut, the LP at this point is:

$$3x_1 + 2x_2 \leq 14, \quad x_1 + 4x_2 \leq 12, \quad 2x_1 + 4x_2 \leq 15.$$

The current LP optimum is $x_1 = 3$, $x_2 = 9/4$, $z = 24$. At this optimum the slack variables are $s_1 = 1/2$ (basic), $s_2 = 0$ (non-basic), $s_3 = 0$ (non-basic, where s_3 is the slack of the first Gomory cut). The basis is $\{x_1, x_2, s_1\}$.

The x_2 row of the optimal tableau reads:

$$x_2 + \tfrac{1}{2}\, s_2 - \tfrac{1}{4}\, s_3 = \tfrac{9}{4}.$$

Gomory cut from the x_2 row. The right-hand side is $\bar{b} = 9/4$, so $f_0 = 9/4 - 2 = 1/4$. Fractional parts of the non-basic coefficients:

- $\bar{a}_{s_2} = 1/2$, so $f_{s_2} = 1/2$.
- $\bar{a}_{s_3} = -1/4$, so $f_{s_3} = -1/4 - \lfloor -1/4 \rfloor = -1/4 + 1 = 3/4$.

The Gomory cut is $\frac{1}{2}\, s_2 + \frac{3}{4}\, s_3 \geq \frac{1}{4}$, or equivalently $2\, s_2 + 3\, s_3 \geq 1$.

Substituting $s_2 = 12 - x_1 - 4x_2$ and $s_3 = 15 - 2x_1 - 4x_2$:

$$2(12 - x_1 - 4x_2) + 3(15 - 2x_1 - 4x_2) \geq 1 \implies 8x_1 + 20x_2 \leq 68,$$

i.e., $2x_1 + 5x_2 \leq 17$. We can verify the cut is violated at the current LP optimum: $2(3) + 5(9/4) = 6 + 45/4 = 69/4 = 17.25 > 17$. ✓

Add $2x_1 + 5x_2 \leq 17$ and re-solve. New LP optimum: $x_1 = 3$, $x_2 = 2$, $z = 5(3) + 4(2) = 23$. **Integer!**

The optimal ILP solution is $(x_1, x_2) = (3, 2)$ with value $z^* = 23$. The algorithm terminates after 2 Gomory cuts.

6.10 Branch and Cut

Branch and Cut = B&B + cutting planes. This is what every serious ILP solver does.

We have now seen two complete methods:

- **Branch and Bound**: guaranteed to find the optimum but may explore many nodes if the LP relaxation is weak.
- **Cutting Planes**: tightens the LP relaxation systematically but can converge slowly on its own.

The natural idea is to *combine* them: at each node of the B&B tree, before branching, add cutting planes to tighten the LP relaxation. This is **Branch and Cut**.

Definition 6.10.1 (Branch and Cut)**. Branch and Cut** is an algorithm that combines Branch and Bound with cutting plane generation:

1. At each node of the B&B tree, solve the LP relaxation.
2. Attempt to generate violated cuts (Gomory cuts, problem-specific cuts, or other families).
3. If cuts are found, add them and re-solve the LP. Repeat the cut generation until no more violated cuts are found (or a limit is reached).
4. If the LP solution is still fractional, branch.

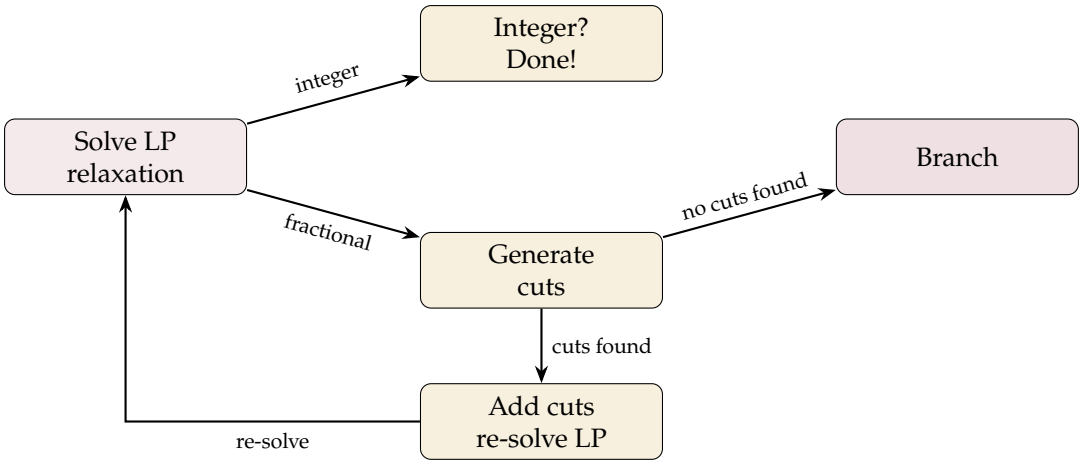


Figure 6.6: Branch and Cut at a single node. After solving the LP relaxation, we attempt to add cuts. Only when no more useful cuts can be found do we branch.

6.10.1 Why it works

The synergy between B&B and cutting planes is powerful:

1. **Cuts reduce the B&B tree.** Tighter LP relaxations at each node produce better bounds, leading to more pruning and fewer nodes.
2. **B&B compensates for slow cut convergence.** Pure cutting planes can take many iterations to close the gap. B&B can branch before the gap is fully closed, handling the remaining integrality gap by enumeration.
3. **Problem-specific cuts.** Beyond generic Gomory cuts, modern solvers exploit structure-specific cuts: cover inequalities for knapsacks, clique cuts for independent set, subtour elimination for TSP. These cuts are often much more effective than generic ones.

Intermezzo — Inside a modern ILP solver

When you call `model.optimize()` in Gurobi or CPLEX, the solver runs a sophisticated Branch and Cut algorithm with many enhancements:

- **Presolve**: simplify the model (fix variables, tighten bounds, remove redundant constraints) before solving.
- **Root node processing**: spend extra time at the root adding many families of cuts (Gomory, MIR, clique, cover, flow cover, . . .).
- **Heuristics**: at each node, run heuristics (rounding, diving, feasibility pump) to find good incumbents early.
- **Parallel B&B**: explore multiple nodes simultaneously on multiple cores.
- **Conflict analysis**: learn from infeasible subproblems to generate additional cuts (inspired by SAT solvers).

These techniques, combined, allow modern solvers to handle ILPs with millions of variables and constraints. The mathematical foundations are exactly what we studied in this chapter: LP relaxation bounds, branching, and cutting planes.

Summary

- ILP is **NP-hard**: no polynomial algorithm is known. The two main exact approaches are Branch and Bound and Cutting Planes.
- **Branch and Bound** (sections 6.2 and 6.3) is a divide-and-conquer strategy. It maintains a tree of subproblems, using LP relaxation bounds to prune branches that cannot improve the incumbent. Three fathoming rules: infeasibility, bounding, integrality (theorem 6.2.3).
- On the running knapsack (theorem 6.4.1), B&B found the optimum $\{1, 2, 3\}$ with value 24 in 19 nodes (vs. 64 possible leaves). The LP relaxation bound at the root (≈ 24.86) guided the search.
- **Search strategies** (section 6.5): depth-first (fast incumbents, low memory), best-first (fewest nodes), breadth-first (systematic but expensive). Modern solvers use hybrids.
- **Cutting planes** (section 6.6) tighten the LP relaxation by adding valid inequalities that are violated by the fractional LP solution but satisfied by all integer points.
- **Chvátal–Gomory rounding** (theorem 6.7.2): combine constraints with non-negative multipliers, round down coefficients, round down the RHS. The resulting inequality is valid for all integer points. Repeated application converges to the integer hull (theorem 6.7.4).
- **Gomory cuts** (theorem 6.8.1) are read directly from the optimal Simplex tableau. For a fractional basic variable, the cut $\sum_j f_j x_j \geq f_0$ (where f_j and f_0 are fractional parts) is valid and violated by the current BFS.
- The **Cutting Plane Algorithm** (algorithm 3) repeatedly adds Gomory cuts and re-solves (via dual Simplex). It converges finitely for pure ILPs (theorem 6.9.1).
- **Branch and Cut** (section 6.10) combines B&B with cutting planes: at each node, add cuts to tighten the LP relaxation before branching. This is what modern solvers (CPLEX, Gurobi, SCIP) actually do.

In the next chapter, we will study **totally unimodular matrices** (chapter 7), a structural property that makes the LP relaxation automatically integer—eliminating the need for B&B or cuts entirely.

Summary & Key Takeaways

- **LP Relaxation:** Dropping integrality constraints yields an upper bound (for max problems) $z^*_{\text{LP}} \geq z^*_{\text{ILP}}$.
- **Branch & Bound:** Systematically partitions the feasible region into a tree. Pruning rules:
  - *Optimality:* The subproblem yields an integer solution.
  - *Infeasibility:* The subproblem relaxation is infeasible.
  - *Bound:* The relaxation value is worse than the best integer solution found so far (incumbent).
- **Cutting Planes:** Iteratively tightens the LP relaxation by adding hyperplanes (e.g., Gomory

fractional cuts) that slice off non-integer vertices without cutting any feasible integer points.

Exercises

Exercise 1 (Why rounding is not enough). Consider the ILP:

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 5x_2 \\ \text{subject to} \quad & x_1 + 2x_2 \leq 7 \\ & 2x_1 + x_2 \leq 7 \\ & x_1, x_2 \geq 0, \quad x_1, x_2 \in \mathbb{Z}. \end{aligned}$$

1. Solve the LP relaxation and record its optimal value z^*_{LP} and optimal solution (x_1^*, x_2^*).
2. Round each fractional component *down* (floor) to obtain a candidate integer solution. Compute its objective value.
3. Round each fractional component *to the nearest integer*. Compute its objective value and check feasibility.
4. Find the true ILP optimum by inspection or enumeration of nearby integer points. How large is the gap between z^*_{LP} and the true integer optimum?

Exercise 2 (LP relaxation upper bound). Let P be a maximisation ILP with feasible region $S \subseteq \mathbb{Z}^n$ and LP relaxation P_{LP} with feasible region S_{LP}.

1. Explain in one sentence why $z^*_{\text{LP}} \geq z^*_{\text{ILP}}$.
2. Give a concrete 2-variable example where the gap $z^*_{\text{LP}} - z^*_{\text{ILP}}$ is exactly 1.
3. Give a concrete 2-variable example where the gap is 0 (i.e. the LP relaxation optimum is already integer).

Exercise 3 (Branching and the feasible set). You are solving the ILP $\max\{c^\top x : Ax \leq b,\ x \geq 0,\ x \in \mathbb{Z}^n\}$. The LP relaxation at the root node has optimal solution $\bar{x} = (2.7,\ 1.0,\ 3.4)$.

1. Which variables are fractional? Name one branching variable.
2. If you branch on x_1, write the two sub-problems (constraints only).
3. Explain why the union of the two sub-problems' feasible sets contains every *integer* feasible point of the parent, but not every *LP* feasible point.
4. What is an *incumbent* in a B&B tree?

Exercise 4 (Fathoming rules). A B&B node can be fathomed (pruned) for three reasons. For each scenario below, state which rule applies and why:

1. The LP relaxation of the node is infeasible.
2. The LP relaxation optimal value is 18.3 and the current incumbent has value 22. (Maximisation problem.)
3. The LP relaxation optimal solution is $\bar{x} = (3, 0, 2)$, which is integer and has objective value 15. The incumbent value is 12.

Exercise 5 (B&B trace — Instance I). Solve the following ILP by hand using Branch and Bound. At each node, solve the LP relaxation (you may quote the optimum without showing simplex steps), apply fathoming rules, and record your B&B tree.

$$\begin{aligned} \text{maximize} \quad & 5x_1 + 4x_2 \\ \text{subject to} \quad & 6x_1 + 4x_2 \leq 24 \\ & x_1 + 2x_2 \leq 6 \\ & x_1, x_2 \geq 0, \quad x_1, x_2 \in \mathbb{Z}. \end{aligned}$$

1. Solve the LP relaxation at the root. What is z^*_{LP}?
2. Branch on the most-fractional variable. List the two child nodes.
3. For each child, solve the LP relaxation and apply fathoming rules.
4. State the optimal ILP solution and its objective value.
5. Draw the B&B tree, labelling each node with its LP bound and status (open / fathomed-bound / fathomed-integer / fathomed-infeasible).

Exercise 6 (B&B trace — Instance II). Apply Branch and Bound to:

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 7x_2 \\ \text{subject to} \quad & x_1 + 4x_2 \le 9 \\ & 3x_1 + 2x_2 \le 10 \\ & x_1, x_2 \ge 0, \quad x_1, x_2 \in \mathbb{Z}. \end{aligned}$$

1. Solve the LP relaxation at the root. Identify the fractional variable.
2. Branch on the fractional variable. For each child node, solve the LP relaxation and record the bound.
3. Update the incumbent whenever an integer-feasible leaf is found.
4. State the optimal ILP solution and prove no better integer point exists by citing the fathomed bounds.

Exercise 7 (B&B trace — Instance III). Use Branch and Bound to solve:

$$\begin{aligned} \text{maximize} \quad & 2x_1 + 3x_2 \\ \text{subject to} \quad & 4x_1 + 6x_2 \le 13 \\ & 2x_1 + x_2 \le 5 \\ & x_1, x_2 \ge 0, \quad x_1, x_2 \in \mathbb{Z}. \end{aligned}$$

Record every node in your B&B tree: its LP relaxation value, the branching decision, and the reason each node is closed. State the optimal solution.

Exercise 8 (B&B trace — Instance IV (minimisation)). Apply Branch and Bound to the *minimisation* ILP:

$$\begin{aligned} \text{minimize} \quad & 4x_1 + 3x_2 \\ \text{subject to} \quad & 2x_1 + x_2 \ge 5 \\ & x_1 + 2x_2 \ge 5 \\ & x_1, x_2 \ge 0, \quad x_1, x_2 \in \mathbb{Z}. \end{aligned}$$

Note that the LP bound is now a *lower* bound and fathoming by bound occurs when $z_{\text{LP}} \ge z_{\text{incumbent}}$. Trace the tree and state the optimal solution.

Exercise 9 (Classify nodes in a given B&B tree). The following describes a B&B tree for a *maximisation* ILP. The current incumbent value is 14.

Node	LP relaxation value	LP solution
Root	18.4	(3.2, 1.5)
N_1 (branch $x_1 \le 3$)	17.0	(3.0, 1.75)
N_2 (branch $x_1 \ge 4$)	infeasible	—
N_3 (branch $x_2 \le 1$, child of N_1)	14.8	(3.0, 1.0)
N_4 (branch $x_2 \ge 2$, child of N_1)	13.5	(2.0, 2.0)

1. For each of N_1, N_2, N_3, N_4, state whether it should be fathomed and if so by which rule (infeasibility / bound / integer feasibility).

2. After processing all four nodes, what is the incumbent?
3. Could the optimal ILP value be 15? Justify your answer.

Exercise 10 (Fathoming by bound vs. integer feasibility). For each scenario, identify the correct fathoming rule and explain whether the node *must* be the last node explored on that branch.

1. $z_{\text{LP}} = 22$, incumbent = 23, LP solution is integer.
2. $z_{\text{LP}} = 22.5$, incumbent = 23, LP solution is fractional.
3. $z_{\text{LP}} = 25$, incumbent = 23, LP solution is integer.
4. $z_{\text{LP}} = 25$, incumbent = 23, LP solution is fractional.

Exercise 11 (Most-fractional vs. nearest-to-integer branching). An LP relaxation has optimal solution $\bar{x} = (1.1, 2.5, 3.9, 0.5)$.

1. Using the *most-fractional* rule, which variable do you branch on? (Recall: choose the variable whose fractional part is closest to 0.5.)
2. Using the *nearest-to-integer* rule, which variable do you branch on? (Recall: choose the variable whose fractional part is closest to 0 or 1.)
3. Argue intuitively why most-fractional branching may produce more balanced subtrees while nearest-to-integer branching may lead to faster fathoming.
4. Give a small example (even a 2-variable one) where the two rules branch on different variables.

Exercise 12 (Best-first vs. depth-first search). A B&B tree has open nodes with LP bounds 21.3, 19.8, 22.7, 20.0. The current incumbent is 18.

1. Under *best-first* search, which node is explored next?
2. Under *depth-first* search, the node with bound 19.8 is the deepest. Is it pruned immediately? Why or why not?
3. Which strategy typically uses less memory? Which typically finds a good incumbent sooner?

Exercise 13 (Valid cuts do not remove integer points). A *valid inequality* for an ILP is one satisfied by every feasible integer point.

1. Consider the ILP $\max\{x_1 + x_2 : x_1 + x_2 \le 1.5,\ x_1, x_2 \ge 0,\ x_1, x_2 \in \mathbb{Z}\}$. The LP relaxation optimum is $(0.75, 0.75)$ with value 1.5. Show that the cut $x_1 + x_2 \le 1$ is valid (check all integer points in the relaxation's feasible region) and that it cuts off the fractional LP optimum.
2. Define "separation problem" in one sentence.
3. Why is it important that a cut does *not* remove any integer feasible point, even if it is not the current LP optimum?

Exercise 14 (Chvátal–Gomory cut — Example I). Derive a Chvátal–Gomory inequality from the system below.

$$2x_1 + 3x_2 \le 7, \qquad x_1, x_2 \ge 0, \quad x_1, x_2 \in \mathbb{Z}.$$

1. Multiply the constraint by $\lambda = \frac{1}{2}$. Write the resulting scaled inequality.
2. Round down all coefficients on the left-hand side to obtain a new inequality with integer coefficients.
3. Round down the right-hand side to obtain the final Chvátal–Gomory inequality.
4. Verify that the LP relaxation optimum $(0, 7/3)$ violates your cut, while the integer point $(0, 2)$ satisfies it.

Exercise 15 (Chvátal–Gomory cut — Example II). Consider the two con-

straints:

$$x_1 + x_2 \le 5, \qquad 2x_1 - x_2 \le 4, \qquad x_1, x_2 \ge 0, \quad x_1, x_2 \in \mathbb{Z}.$$

1. Combine the two constraints with multipliers $\lambda_1 = \frac{2}{3}$ and $\lambda_2 = \frac{1}{3}$. Write the resulting inequality.
2. Apply the Chvátal–Gomory rounding (floor all coefficients, floor the RHS) to obtain a valid integer inequality.
3. Check that the fractional point $(x_1, x_2) = (3, 2)$ satisfies both original constraints and your new cut.

Exercise 16 (Chvátal–Gomory cut — Example III). Given:

$$3x_1 + 5x_2 \le 11, \qquad x_1 + x_2 \le 3, \qquad x_1, x_2 \ge 0, \quad x_1, x_2 \in \mathbb{Z}.$$

1. Use multiplier $\lambda_1 = \frac{1}{3}$ on the first constraint and $\lambda_2 = 0$ on the second. Apply Chvátal–Gomory rounding and state the resulting cut.
2. Now use $\lambda_1 = 0$ and $\lambda_2 = \frac{2}{3}$. State the resulting cut.
3. Which of the two cuts is tighter (eliminates more of the LP relaxation's feasible region)? Justify briefly.

Exercise 17 (Chvátal–Gomory cut — Example IV). A knapsack LP relaxation has constraint $7x_1 + 4x_2 + 3x_3 \le 10$ with $x_i \in \{0, 1\}$ (relax to $0 \le x_i \le 1$). The LP optimum is $x^* = (0, 1, 1)$ rounded to $(0, 10/7, \ldots)$—more precisely the LP gives a fractional solution.

1. Apply the Chvátal–Gomory procedure with $\lambda = \frac{1}{7}$ to derive a cut.
2. Verify your cut is satisfied by all 0-1 points feasible for the original constraint.
3. State why this cut is sometimes called a *cover inequality* in this context.

Exercise 18 (Gomory cut from simplex tableau — Instance I). After solving an LP relaxation, the optimal simplex tableau contains the following row for basic variable x_1:

$$x_1 + \tfrac{3}{4}\, s_1 - \tfrac{1}{4}\, s_2 = \tfrac{7}{4}.$$

Here s_1, s_2 are slack variables (non-basic, equal to zero at the optimum).

1. Identify the fractional part $f_0 = \{7/4\}$ and the fractional parts f_j of each coefficient.
2. Write the Gomory cut $\sum_j f_j\, x_j \ge f_0$ in terms of s_1 and s_2.
3. Verify that the current BFS (where $s_1 = s_2 = 0$, $x_1 = 7/4$) violates your cut.
4. Verify that the integer point $x_1 = 2$, $s_1 = s_2 = 0$ satisfies your cut.

Exercise 19 (Gomory cut from simplex tableau — Instance II). The optimal tableau for a two-variable ILP relaxation is:

	x_1	x_2	s_1	s_2	s_3	RHS
x_1	1	0	$\frac{1}{2}$	$-\frac{1}{4}$	0	$\frac{5}{2}$
x_2	0	1	$\frac{1}{4}$	$\frac{3}{4}$	0	$\frac{7}{4}$
s_3	0	0	-1	1	1	2
$-z$	0	0	$\frac{3}{4}$	$\frac{1}{4}$	0	11

1. Which basic variables are fractional?
2. Derive a Gomory cut from the row for x_1.
3. Derive a Gomory cut from the row for x_2.
4. Add the cut from x_2's row to the tableau as a new row. What is the new right-hand-side value, and is the new basic variable feasible (non-negative)?

Exercise 20 (Gomory cut from simplex tableau — Instance III). A single-row optimal tableau is:

$$x_3 + \tfrac{2}{3}\, x_1 + \tfrac{5}{3}\, s_1 = \tfrac{8}{3}.$$

Here x_1 is non-basic.

1. Write the Gomory cut derived from this row.
2. Introduce a slack variable s_{new} to convert the cut to an equality and add it as a new row to the tableau.
3. The new BFS has $s_{\text{new}} = -f_0 < 0$. Explain why a *dual* simplex pivot (not a primal pivot) is required to restore feasibility.

Exercise 21 (Gomory cut from simplex tableau — Instance IV). The fractional basic variable row reads:

$$x_2 + \tfrac{1}{3}\, s_1 + \tfrac{2}{3}\, s_2 + \tfrac{4}{3}\, s_3 = \tfrac{10}{3}.$$

1. Compute the fractional parts of all coefficients and the RHS.
2. Write the Gomory cut.
3. After adding the cut as a constraint, the dual simplex must choose an entering variable. The reduced costs (objective row) for s_1, s_2, s_3 are 0.5, 1.2, 0.8 respectively. Apply the dual simplex ratio test to identify the entering variable.

Exercise 22 (One iteration of the cutting plane algorithm). Consider:

$$\begin{aligned} \text{maximize} \quad & 2x_1 + x_2 \\ \text{subject to} \quad & 4x_1 + 2x_2 \le 9 \\ & 2x_1 + 3x_2 \le 8 \\ & x_1, x_2 \ge 0, \quad x_1, x_2 \in \mathbb{Z}. \end{aligned}$$

1. Solve the LP relaxation. Is the optimal solution integer?
2. Identify a fractional basic variable row in the optimal tableau.
3. Derive the Gomory cut for that row.
4. Add the Gomory cut to the LP and re-solve using the dual simplex. (You may describe the dual pivot step without performing full simplex—just identify the leaving and entering variables and state the new BFS.)
5. Is the new LP optimal solution integer? If yes, is it also ILP-optimal? If no, explain what you would do next.

Exercise 23 (Dual simplex pivot after adding a Gomory cut). After solving an LP relaxation, the optimal tableau (relevant rows) is:

	x_1	x_2	s_1	s_2	RHS
x_1	1	0	$\frac{1}{2}$	$-\frac{1}{4}$	$\frac{5}{2}$
x_2	0	1	$-\frac{1}{4}$	$\frac{3}{4}$	$\frac{3}{4}$
$-z$	0	0	$\frac{3}{4}$	$\frac{1}{4}$	$\frac{13}{4}$

A Gomory cut derived from the x_2-row gives (after introducing slack s_3): $\frac{3}{4}s_1 + \frac{1}{4}s_2 - s_3 = \frac{3}{4}$, equivalently $s_3 = \frac{3}{4}s_1 + \frac{1}{4}s_2 - \frac{3}{4}$.

1. Add the Gomory cut row to the tableau. What is the RHS of the new row? Is the current basis (extended with s_3 basic) primal feasible? Is it dual feasible?
2. Identify the leaving variable for the dual simplex pivot (the most negative RHS).
3. Apply the dual simplex ratio test to find the entering variable.

4. Perform one dual simplex pivot and write the updated tableau.

Exercise 24 (Convergence of the Gomory cutting plane algorithm).

1. State Gomory's finite convergence theorem: the cutting plane algorithm terminates in a finite number of iterations for any bounded pure ILP.
2. Explain informally why each Gomory cut strictly tightens the LP relaxation (i.e. cuts off the current fractional optimum).
3. Give one practical reason why the pure cutting-plane algorithm is rarely used in isolation in modern solvers, even though it is theoretically convergent.

Exercise 25 (Branch and Cut on a small ILP). Consider the ILP:

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 2x_2 \\ \text{subject to} \quad & 2x_1 + x_2 \leq 7 \\ & x_1 + 2x_2 \leq 7 \\ & x_1, x_2 \geq 0, \quad x_1, x_2 \in \mathbb{Z}. \end{aligned}$$

1. Solve the LP relaxation at the root.
2. Derive one Gomory cut from the fractional LP optimum and add it to the root node LP. Re-solve.
3. If the new LP optimum is integer, you are done. If not, branch on a fractional variable. Describe the two child nodes.
4. State the optimal ILP solution found by Branch and Cut.
5. Compare the number of B&B nodes needed with and without the Gomory cut.

Exercise 26 (How Branch and Cut uses cuts at each node). Describe the Branch and Cut framework by answering the following:

1. At which point in the B&B node processing are cutting planes added? (Before branching, after, or during?)
2. Why is it useful to add cuts at an *internal* B&B node rather than only at the root?
3. What happens to a cut added at a node N? Is it valid for all other nodes in the tree, or only for the subtree rooted at N? Explain.
4. Name two types of cuts commonly used in Branch and Cut solvers (e.g. Gomory cuts and one other class).

Exercise 27 (Comparing LP relaxation strength). Two different formulations of the same ILP have LP relaxation optimal values $z_1^{\text{LP}} = 14.5$ and $z_2^{\text{LP}} = 13.0$. The true ILP optimal value is 12.

1. Which formulation has the stronger LP relaxation? Why?
2. A stronger LP relaxation generally leads to a smaller B&B tree. Explain the mechanism: how does a tighter LP bound lead to more pruning?
3. Give a concrete 2-variable example of two different LP formulations of the same ILP (i.e. add a redundant LP constraint to one of them) such that the two LP relaxations have different optimal values.

Exercise 28 (Tightening a formulation with variable bounds). An ILP has binary variable $x_1 \in \{0, 1\}$ but is formulated with only the bound $0 \leq x_1 \leq 1$ (i.e. the integrality constraint is relaxed). A second formulation explicitly adds $x_1 \leq 1$ as a constraint and also adds the cover inequality $x_1 + x_2 \leq 1$.

1. Explain why the second formulation has a tighter (smaller) LP relaxation feasible region.
2. Under what condition does the additional cover inequality become redundant for the LP relaxation?

3. Why does a tighter LP relaxation reduce the number of nodes explored in B&B?

Exercise 29 (Effect of Big-M on the B&B tree). A mixed-integer program uses a Big-M formulation for a disjunctive constraint: $x \le My$ where $y \in \{0, 1\}$ and M is a large constant.

1. With $M = 100$ and $M = 10{,}000$, how does the LP relaxation bound change at the root? (Assume x is bounded above by 5 in any feasible integer solution.)
2. Explain why a very large M weakens the LP relaxation.
3. What practical advice would you give to a modeller who must use Big-M in a B&B-based solver?
4. Suggest an alternative to Big-M for the specific constraint $x_1 + x_2 \le 10y$ where you know $x_1, x_2 \le 5$.

Exercise 30 (True or False: Gomory cuts and integer feasibility). Determine whether each statement is true or false. If true, give a brief justification. If false, give a counterexample or explain the error.

1. "Adding a Gomory cut can cut off an integer-feasible point."
2. "Every Gomory cut derived from an optimal simplex tableau is violated by the current LP optimum."
3. "Gomory cuts can only be derived from rows where the basic variable is required to be integer."
4. "After adding a Gomory cut, the dual simplex always terminates in one pivot."

Exercise 31 (True or False: B&B completeness and correctness). For each statement, write **True** or **False** and justify.

1. "Branch and Bound with LP bounds is complete: it always finds the optimal integer solution if one exists."
2. "If the LP relaxation at a node is feasible, the ILP restricted to that node must also be feasible."
3. "The B&B incumbent after processing all nodes is a feasible integer solution."
4. "B&B is guaranteed to terminate in polynomial time."

Exercise 32 (True or False: Cutting planes and LP relaxations). Write **True** or **False** and justify each claim.

1. "A valid inequality that is not violated by the current LP optimum is useless and should not be added."
2. "The cutting plane algorithm is equivalent to B&B in the sense that they explore the same nodes."
3. "Every valid inequality for an ILP is a Chvátal–Gomory inequality for some choice of multipliers."
4. "Adding more valid cuts always speeds up the overall B&B algorithm."

Exercise 33 (Proof: Gomory cut is valid for all integer points). Let the optimal simplex tableau row for basic variable x_i be

$$x_i + \sum_{j \in N} \bar{a}_{ij}\, x_j = \bar{b}_i,$$

where N is the set of non-basic variable indices and $\bar{b}_i \notin \mathbb{Z}$ (i.e. x_i is fractional at the LP optimum). Write $\bar{b}_i = \lfloor \bar{b}_i \rfloor + f_0$ and $\bar{a}_{ij} = \lfloor \bar{a}_{ij} \rfloor + f_j$ with $f_0, f_j \in (0, 1)$. The Gomory cut is $\sum_{j \in N} f_j\, x_j \ge f_0$.

Prove, in two steps, that this cut is satisfied by every integer feasible point:

1. Rewrite the tableau row as
$$x_i + \sum_{j \in N} \lfloor \bar{a}_{ij} \rfloor x_j - \lfloor \bar{b}_i \rfloor = f_0 - \sum_{j \in N} f_j x_j.$$
2. Argue that if x_i and all x_j are integers, the left-hand side is an integer, so the right-hand side must also be an integer. Since $f_0 \in (0, 1)$, conclude that $\sum_{j \in N} f_j x_j \geq f_0$.

Exercise 34 (Proof: Gomory cut is violated by the current LP optimum). Using the same notation as Exercise 33, prove that the current LP optimal BFS $\bar{x}$ violates the Gomory cut.

Hint: At the LP optimum, all non-basic variables $x_j = 0$ for $j \in N$. Substitute into $\sum_{j \in N} f_j x_j \geq f_0$ and use the fact that $f_0 > 0$.

Exercise 35 (Chvátal closure and iterated cuts). The *Chvátal closure* P' of an LP polyhedron P is the intersection of all Chvátal–Gomory cuts derivable from the constraints of P.

1. For the LP $\{(x_1, x_2) : x_1 + x_2 \leq 3.5,\ x_1, x_2 \geq 0\}$, write all non-trivial Chvátal–Gomory cuts obtainable with $\lambda = 1$ applied to the single inequality.
2. Is P' equal to the integer hull of P in general? What is the relationship?
3. How many rounds of Chvátal–Gomory cuts are needed to reach the integer hull of a rational polyhedron?

Exercise 36 (B&B on a three-item knapsack). A knapsack has capacity $W = 8$. Three items have weights $w = (5, 3, 2)$ and values $v = (7, 4, 3)$. Variables $x_i \in \{0, 1\}$.

1. Solve the LP relaxation (greedy by value/weight ratio). What is z^*_{LP}?
2. Branch on the fractional item. For each child:
   (a) Set the item to 0 or 1 and solve the remaining LP relaxation.
   (b) Apply fathoming rules.
3. State the optimal integer solution and its value.

Exercise 37 (Optimality gap and termination criterion). A B&B solver maintains a best lower bound $\underline{z}$ (the incumbent) and a best upper bound $\bar{z}$ (the best open-node LP bound). The relative optimality gap is defined as $(\bar{z} - \underline{z})/|\bar{z}|$.

1. After exploring some nodes, $\underline{z} = 47$ and $\bar{z} = 51.3$. Compute the relative optimality gap (as a percentage).
2. A solver is set to terminate when the gap is below 1%. Is it safe to stop now? What can you say about the quality of the current incumbent?
3. Explain why allowing a small optimality gap (e.g. 0.5%) is common in practice even for exact solvers.

Exercise 38 (B&B for a mixed-integer program). A mixed-integer program has one integer variable $y \in \mathbb{Z}$ and one continuous variable $x \in \mathbb{R}$:

$$\begin{aligned} \text{maximize} \quad & 3y + x \\ \text{subject to} \quad & 2y + x \leq 7 \\ & y + 2x \leq 6 \\ & y, x \geq 0, \quad y \in \mathbb{Z}. \end{aligned}$$

1. Solve the LP relaxation.
2. Branch on y (the only integer variable). For each child, solve the LP (now a pure LP) and record the bound.
3. Fathom nodes as appropriate and state the optimal MIP solution.

Exercise 39 (Infeasible node in B&B). Consider an ILP with constraints $x_1 + x_2 \leq 3$ and $x_1 + x_2 \geq 4$, $x_1, x_2 \geq 0$.

1. Show that the LP relaxation (and hence the ILP) is infeasible.
2. Suppose this infeasible system arises as a child node in a B&B tree after branching $x_1 \geq 4$ on the parent which had feasible region $0 \leq x_1 \leq 3$. Why does infeasibility arise?
3. When a B&B node is infeasible, is there any need to continue exploring its subtree? Why?

Exercise 40 (Warm-start with dual simplex in B&B). When B&B adds a branching constraint (e.g. $x_1 \leq 2$) to a node whose parent LP has already been solved, it is efficient to use the parent's optimal basis as a starting basis for the child.

1. Adding $x_1 \leq 2$ may make the current basis *infeasible* (but dual feasible). Which simplex method (primal or dual) is appropriate to re-optimise? Why?
2. Adding $x_1 \geq 3$ similarly may require a dual simplex restart. State the condition on the parent's basis that makes this possible without an artificial variable.
3. Why is warm-starting at child nodes computationally important in practice?

Exercise 41 (LP bound quality and the integrality gap). The *integrality gap* of an ILP instance is $\mathrm{IG} = z^*_{\mathrm{LP}}/z^*_{\mathrm{ILP}}$ (for maximisation; assume $z^*_{\mathrm{ILP}} > 0$).

1. For the ILP $\max\{x_1 + x_2 : x_1 + x_2 \leq k,\ x_1, x_2 \geq 0, x_1, x_2 \in \mathbb{Z}\}$ with $k = 1.5$, compute IG.
2. Is IG always ≥ 1 for maximisation ILPs? Why?
3. Give an example of an ILP with IG $= 1$ (no gap). What structural property ensures this?

Exercise 42 (Design a Branch and Cut strategy for a set-cover ILP). A set-cover ILP is: $\min\{c^\top x : Ax \geq \mathbf{1},\ x \in \{0,1\}^n\}$, where $A \in \{0,1\}^{m \times n}$.

1. Describe the LP relaxation and explain why the LP optimum is often fractional (e.g. $x_j = 0.5$ for many j).
2. Name two classes of cuts that are commonly used for set-cover ILPs (e.g. cover cuts, odd-cycle cuts).
3. Outline a Branch and Cut algorithm for this problem:
   (a) What cuts do you add at the root node?
   (b) How do you branch (which variable)?
   (c) How do you fathom nodes?
4. Explain why adding cuts at each B&B node can dramatically reduce the number of nodes in the tree.

Exercise 43 (Sensitivity of B&B to objective perturbation). For the ILP from Exercise 5, suppose the objective changes from $5x_1 + 4x_2$ to $4x_1 + 5x_2$.

1. Solve the new LP relaxation at the root.
2. Does the B&B tree change (different branching decisions, different number of nodes)?
3. State the new optimal ILP solution. Is the optimal basis of the LP relaxation the same?

Exercise 44 (Strengthening Gomory cuts). The standard Gomory fractional cut derived from a tableau row may be weak when some variables are bounded (e.g. binary). For a row

$$x_i + \bar{a}\, y = \bar{b}, \quad y \in \{0,1\},$$

with $f_0 = \{\bar{b}\}$ and $f = \{\bar{a}\}$:

1. Write the standard Gomory cut.
2. Write the *mixed-integer* Gomory cut, which uses the bound $y \leq 1$:

$$\text{if } f \leq f_0: \quad f\,y \geq f_0, \qquad \text{if } f > f_0: \quad (1-f)\,y \geq (1-f_0).$$

   With $\bar{b} = 7/4$ (so $f_0 = 3/4$) and $\bar{a} = 5/4$ (so $f = 1/4$), apply both formulas and compare the two cuts.
3. State which cut is tighter and why.

Exercise 45 (Enumerate all integer points in an LP feasible region). Consider the LP relaxation:

$$\{(x_1, x_2) \in \mathbb{R}^2 : 2x_1 + x_2 \leq 6,\ x_1 + 2x_2 \leq 6,\ x_1, x_2 \geq 0\}.$$

1. Find the vertices of the LP feasible region.
2. List all integer points $(x_1, x_2) \in \mathbb{Z}^2_{\geq 0}$ inside or on the boundary of the region.
3. Among those, which maximises $2x_1 + 3x_2$?
4. What is the LP relaxation optimum of $\max 2x_1 + 3x_2$? Compute the integrality gap.

Exercise 46 (Correctness of the B&B upper bound update). Explain, using the definition of the B&B tree, why the following procedure maintains a valid upper bound throughout the algorithm:

1. Initially, $\bar{z} = z^*_{\text{LP}}$ at the root node.
2. After branching, $\bar{z}$ is updated to max of the LP bounds of all open nodes.
3. Justify: for every unexplored integer solution x^*, there exists an open node whose LP relaxation contains x^*, so $\bar{z}$ is a valid upper bound on the ILP optimum.

Exercise 47 (Chvátal rank). The *Chvátal rank* of a polyhedron P is the minimum number of rounds of Chvátal–Gomory cuts needed to obtain the integer hull.

1. For the polytope $P = \{(x_1, x_2) : x_1 + x_2 \leq 1.5,\ x_1, x_2 \geq 0\}$, show that one round of cuts is sufficient to reach the integer hull. What is the Chvátal rank?
2. Give an example (even informal) of a family of polytopes for which the Chvátal rank grows with the dimension.
3. What does a high Chvátal rank imply about the difficulty of solving the ILP by cutting planes alone?

Exercise 48 (Separation oracle concept). A *separation oracle* for a class of valid inequalities takes a point $\bar{x}$ and either confirms $\bar{x}$ satisfies all inequalities in the class, or returns a violated one.

1. For Gomory cuts derived from the optimal simplex tableau, describe a simple separation oracle: given the current LP optimal $\bar{x}$, how do you find a violated Gomory cut?
2. Why is an efficient separation oracle important for the practical application of cutting planes in large-scale ILPs?
3. For the class of all valid inequalities of a given ILP, is a polynomial-time separation oracle always available? Explain.

Exercise 49 (Node selection and the B&B lower bound). In a maximisation B&B, the *best lower bound* $\underline{z}$ equals the best integer-feasible solution found so far (the incumbent).

1. Explain why exploring a node with LP bound 21 before a node with LP bound 25 might cause you to find a better incumbent sooner.
2. A B&B tree has three open nodes with LP bounds 18.0, 21.5, 19.3. The current incumbent is 16. Under best-first search, list the order in which nodes are explored.
3. After exploring the node with bound 21.5, its LP solution is integer with value 21. Update the incumbent and state which, if any, open nodes can now be fathomed.

Exercise 50 (Cutting plane algorithm — two iterations). Apply two iterations of the cutting plane algorithm to:

$$\begin{aligned} \text{maximize} \quad & x_1 + x_2 \\ \text{subject to} \quad & 3x_1 + 2x_2 \leq 7 \\ & x_1 + 3x_2 \leq 6 \\ & x_1, x_2 \geq 0, \quad x_1, x_2 \in \mathbb{Z}. \end{aligned}$$

1. Iteration 1: solve the LP relaxation, identify a fractional basic variable, derive the Gomory cut, add it to the LP.
2. Iteration 2: solve the augmented LP (use dual simplex), check integrality. If still fractional, derive another Gomory cut.
3. After two cuts, has the LP optimum become integer? If yes, verify it is also ILP-feasible.

Exercise 51 (Comparing B&B and cutting plane efficiency). For a given ILP, a pure B&B solver explores 120 nodes, while a pure cutting plane solver requires 8 Gomory cuts to reach the integer optimum.

1. Is it possible to compare these two approaches directly in terms of "work done"? What are the relevant cost metrics?
2. Describe one scenario where B&B is clearly preferable to a pure cutting plane approach.
3. Describe one scenario where adding cuts (before or during B&B) is clearly beneficial.
4. Why do modern solvers use Branch and Cut rather than pure B&B or pure cutting planes?

Exercise 52 (LP relaxation of a binary ILP). A binary ILP has $x_i \in \{0, 1\}$ for all i.

1. Write the LP relaxation (replace $x_i \in \{0, 1\}$ with $0 \leq x_i \leq 1$).
2. Give an example where the LP relaxation optimum has all $x_i \in \{0, 0.5, 1\}$ (half-integer solution).
3. What is a *half-integer* LP solution, and why do they arise frequently in binary ILP relaxations?
4. For a half-integer solution, which variable would the most-fractional branching rule choose?

Exercise 53 (Gomory cut for a pure binary program). For a binary ILP, all variables satisfy $x_j \in \{0, 1\}$. Suppose the optimal tableau row for basic variable x_1 is:

$$x_1 + \tfrac{1}{2}\, x_2 + \tfrac{3}{4}\, x_3 = \tfrac{5}{4},$$

where x_2, x_3 are non-basic binary variables.

1. Write the standard Gomory fractional cut.
2. Since $x_2, x_3 \in \{0, 1\}$, verify your cut directly by checking all four combinations $(x_2, x_3) \in \{0, 1\}^2$. Which are feasible for the original row and satisfy the cut?

3. Is the current BFS (with $x_2 = x_3 = 0$, $x_1 = 5/4$) integer feasible? Does it violate the cut?

Exercise 54 (Recognise valid and invalid cuts). An ILP has feasible integer points $S = \{(0,0),(1,0),(0,1),(2,1),(1,2)\}$ and LP relaxation optimum $(1.5, 1.5)$ with objective value 6. For each proposed cut below, state whether it is a *valid* cut (satisfied by all points in S) and whether it is *useful* (violated by the LP optimum):

1. $x_1 + x_2 \leq 3$.
2. $x_1 + x_2 \leq 2$.
3. $x_1 \leq 1$.
4. $x_1 + x_2 \leq 4$.

Exercise 55 (ILP infeasibility certificate via B&B). A B&B algorithm terminates having explored all nodes without finding any integer-feasible solution.

1. What does this outcome imply about the ILP?
2. Suppose all leaf nodes were fathomed by infeasibility of the LP relaxation. What does this imply about the LP relaxation of the original problem?
3. Suppose some leaf nodes were fathomed by infeasibility and some by LP bound being below the incumbent—but the incumbent was never updated from $-\infty$. What does this imply?
4. Give a 2-variable example of an ILP that is infeasible but whose LP relaxation is feasible. Show how B&B detects infeasibility.

CHAPTER 7

TUM & Integral Polyhedra

This chapter answers a question from chapter 3: when does the LP relaxation solve the ILP for free? The answer lies in a remarkable property of the constraint matrix.

In chapter 6 we developed two powerful strategies for solving ILPs: Branch and Bound, which partitions the search space and prunes with LP bounds, and Cutting Planes, which tightens the LP relaxation until the fractional solution becomes integer. Both are needed when the LP relaxation polyhedron P is strictly larger than the convex hull of integer points $P(X) = \text{conv}(X)$.

But what if $P = P(X)$ automatically? Then the LP relaxation already gives an integer optimum, and we need neither branching nor cutting. We first encountered this possibility in chapter 3 (theorem 3.8.8) and noted that the assignment problem (theorem 3.2.3) enjoys this property. Now we finally explain *why*.

The key turns out to be a structural property of the constraint matrix A, called **total unimodularity** (TUM). When A is totally unimodular and b is integral, every vertex of the polyhedron $\{x : Ax \leq b, x \geq 0\}$ is automatically integer. This single theorem connects LP to a wealth of combinatorial problems—shortest paths, network flows, matching on bipartite graphs, transportation—and sets the stage for the graph optimisation chapters that follow.

Road map.

1. Integral polyhedra and naturally integer ILPs (section 7.1).
2. Total unimodularity: definition and first examples (section 7.2).
3. The main theorem: TUM $\Rightarrow$ integral polyhedra (section 7.3).
4. Properties preserved under operations (section 7.5).
5. Sufficient conditions for TUM (section 7.6).
6. Directed graph incidence matrices (section 7.7).
7. Bipartite graph incidence matrices (section 7.8).
8. Applications: where TUM solves the ILP for free (section 7.9).

7.1 Integral Polyhedra

An integral polyhedron is one whose vertices "land" on the integer lattice. No rounding needed.

Before we introduce total unimodularity, let us formalise the ideal situation that we are aiming for.

> **Definition 7.1.1** (Integral polyhedron)**.** A polyhedron $P \subseteq \mathbb{R}^n$ is **integral** if every vertex (extreme point) of P has integer coordinates, i.e. every vertex belongs to $\mathbb{Z}^n$.

Why does this matter? Recall the fundamental theorem of linear programming from chapter 2: if an LP has a finite optimum, it is attained at a vertex of the feasible polyhedron. So if P is integral, the LP optimum is automatically an integer point —exactly what we want when solving an ILP by its LP relaxation.

Example 7.1.2 (An integral vs. a non-integral polyhedron)**.** Consider two polyhedra in $\mathbb{R}^2$:

$$P_1 = \{(x_1, x_2) : x_1 + x_2 \leq 3,\ x_1,\ x_2 \geq 0\},$$

$$P_2 = \{(x_1, x_2) : 2x_1 + 2x_2 \leq 3,\ x_1,\ x_2 \geq 0\}.$$

The vertices of P_1 are $(0,0)$, $(3,0)$, and $(0,3)$—all integer. So P_1 is integral.

The vertices of P_2 are $(0,0)$, $(3/2,0)$, and $(0,3/2)$. The last two are fractional, so P_2 is *not* integral.

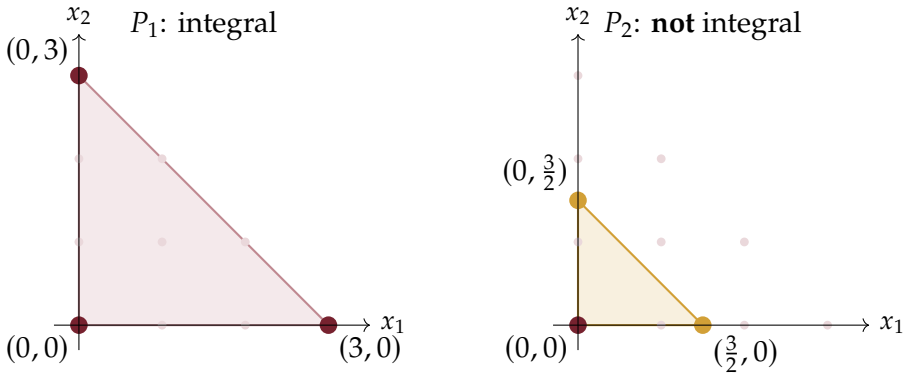


Figure 7.1: Left: the polyhedron P_1 is integral—all three vertices lie on the integer lattice. Right: the polyhedron P_2 is *not* integral—the vertices $(\frac{3}{2},0)$ and $(0,\frac{3}{2})$ are fractional.

Now, when we solve an ILP by its LP relaxation, the feasible polyhedron depends on A, b, and the sign constraints. We want the polyhedron to be integral not just for one particular b, but for *any* integer right-hand side. This motivates the following notion.

Definition 7.1.3 (Naturally integer ILP)**.** An ILP is **naturally integer** (or **naturally integral**) if, for every integer vector b, the LP relaxation polyhedron $P = \{x \in \mathbb{R}^n : Ax \leq b,\ x \geq 0\}$ is integral.

In other words, when an ILP is naturally integer, integrality comes for free. We never need Branch and Bound or cutting planes. The Simplex method (or any LP algorithm) suffices. This is a tremendous computational advantage.

"Naturally integer" means the integrality constraints $x \in \mathbb{Z}^n$ are redundant: the LP already gives integer solutions for any integer b.

The question becomes: *which constraint matrices A make the ILP naturally integer?* The answer is total unimodularity.

7.2 Total Unimodularity: Definition

Let's build up to the definition through a concrete example.

TUM is a property of the matrix *A alone—it does not depend on b, c, or any particular ILP instance.*

Example 7.2.1 (Checking determinants of submatrices)**.** Consider the matrix

$$A = \begin{pmatrix} 1 & 0 & 1 \\ -1 & 1 & 0 \end{pmatrix}.$$

Let us compute the determinant of every square submatrix:

- 1×1 **submatrices** (single entries): $1, 0, 1, -1, 1, 0$. All in $\{0, +1, -1\}$. ✓
- 2×2 **submatrices** (there are $\binom{3}{2} = 3$ ways to choose 2 columns):

$$\det \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} = 1, \quad \det \begin{pmatrix} 1 & 1 \\ -1 & 0 \end{pmatrix} = 1, \quad \det \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = -1.$$

  All in $\{0, +1, -1\}$. ✓

Every square submatrix of A has determinant in $\{0, +1, -1\}$. As we will see next, this makes A totally unimodular.

Definition 7.2.2 (Totally Unimodular Matrix — TUM)**.** A matrix $A \in \mathbb{R}^{m \times n}$ is **totally unimodular** (TU or TUM) if every square submatrix of A has determinant equal to 0, +1, or −1.

Let us unpack this carefully. A *square submatrix* of A is obtained by selecting any k rows and any k columns ($1 \leq k \leq \min(m, n)$) and forming the resulting $k \times k$ matrix. The definition requires that *every* such submatrix—of *every* size—has determinant in $\{0, +1, -1\}$.

Observation 7.2.3 (Entries of a TUM matrix). Since 1×1 submatrices are simply individual entries, a TUM matrix can only contain the values 0, +1, and −1. No entry can be 2, −3, or 0.5.

Having entries in $\{0, \pm 1\}$ is necessary but not sufficient. The constraint applies to all sizes simultaneously.

A matrix with entries in $\{0, \pm 1\}$ is not necessarily TUM. TUM requires the determinant condition for submatrices of all *sizes, not just 1×1.*

Example 7.2.4 (A $\{0, \pm 1\}$-matrix that is NOT TUM)**.** Consider

$$B = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Every entry is in $\{0, +1, -1\}$. But $\det(B) = (1)(-1) - (1)(1) = -2$. Since $-2 \notin \{0, +1, -1\}$, the matrix B is **not** TUM.

Example 7.2.5 (The identity matrix is TUM)**.** The $n \times n$ identity matrix I_n is TUM. Every entry is 0 or 1, and every square submatrix of I_n is a permutation matrix (possibly with some zero rows/columns), which has determinant 0 or ±1.

7.3 The Main Theorem: TUM Implies Integral Polyhedra

This is the central result of the chapter. It is the bridge between matrix structure and the LP-solves-ILP miracle.

We now state and prove the theorem that connects total unimodularity to integral polyhedra. This is the result promised in chapter 3 and chapter 6: it explains when and why the LP relaxation solves an ILP for free.

Theorem 7.3.1 (TUM implies integral polyhedra)**.** *Let $A \in \mathbb{R}^{m \times n}$ be a totally unimodular matrix and let $b \in \mathbb{Z}^m$ be an integer vector. Then the polyhedron*

$$P = \{x \in \mathbb{R}^n : Ax \leq b,\ x \geq 0\}$$

is integral: every vertex of P has integer coordinates.

Before proving this, let us appreciate what it says. If we formulate an ILP with constraint matrix A and A happens to be TUM, then for *any* integer right-hand side b, the LP relaxation polyhedron has only integer vertices. Since the Simplex method finds a vertex solution, it automatically finds an integer solution. The ILP is naturally integer in the sense of theorem 7.1.3.

In the language of chapter 3: the LP relaxation polyhedron P equals the convex hull $P(X)$, so the LP relaxation solves the ILP exactly (theorem 3.8.8).

Formal details — Proof of theorem 7.3.1

We prove that every vertex of P is integer.

Setup. The polyhedron is $P = \{x : Ax \leq b,\ x \geq 0\}$. We can rewrite this in standard form by introducing slack variables, but it is more direct to work with the vertex characterisation. A vertex $\bar{x}$ of P is a basic feasible solution (BFS): there exists a set $\mathcal{B}$ of n linearly independent constraints that are active (satisfied with equality) at $\bar{x}$.

Active constraints. The constraints $Ax \leq b$ and $x \geq 0$ give us $m + n$ inequalities. At a vertex, possibly more than n constraints are tight; choose n linearly independent tight constraints, which uniquely determine the vertex. Let us partition the chosen constraints into two groups:

- The rows of $Ax \leq b$ that are tight: say the rows indexed by $I \subseteq \{1, \ldots, m\}$, giving $A_I \bar{x} = b_I$.
- The non-negativity constraints that are tight: say $\bar{x}_j = 0$ for $j \in J \subseteq \{1, \ldots, n\}$.

Together, $|I| + |J| = n$ and the system

$$A_I \bar{x} = b_I, \qquad \bar{x}_J = 0$$

uniquely determines $\bar{x}$.

Reducing to a square system. The variables $\bar{x}_j$ with $j \in J$ are zero. Let $\bar{J} = \{1, \ldots, n\} \setminus J$ be the indices of the *non-zero* (or potentially non-zero) variables. Then $|\bar{J}| = n - |J| = |I|$. Writing $A_{\mathcal{B}}$ for the submatrix of A with rows I and columns $\bar{J}$, we get:

$$A_{\mathcal{B}}\, \bar{x}_{\bar{J}} = b_I.$$

This is a square system of size $|I| \times |I|$, and $A_{\mathcal{B}}$ is a square submatrix of A.

Using TUM. Since A is TUM, $\det(A_{\mathcal{B}}) \in \{0, +1, -1\}$. Because $\bar{x}$ is a vertex (hence the n tight constraints are linearly independent), $A_{\mathcal{B}}$ is non-singular, so $\det(A_{\mathcal{B}}) \in \{+1, -1\}$.

Cramer's rule. The unique solution is $\bar{x}_{\bar{J}} = A_{\mathcal{B}}^{-1} b_I$. By Cramer's rule,

$$(\bar{x}_{\bar{J}})_k = \frac{\det(A_{\mathcal{B}}^{(k)})}{\det(A_{\mathcal{B}})},$$

where $A_{\mathcal{B}}^{(k)}$ is $A_{\mathcal{B}}$ with column k replaced by b_I.

Now:

- The denominator is $\det(A_{\mathcal{B}}) = \pm 1$.
- The numerator $\det(A_{\mathcal{B}}^{(k)})$ is an integer, because $A_{\mathcal{B}}$ has integer entries (all entries of a TUM matrix are in $\{0, \pm 1\}$) and b_I is integer (given).

Therefore each component of $\bar{x}_{\bar{J}}$ is integer/(± 1) = integer. Combined with $\bar{x}_J = 0$, the entire vertex $\bar{x}$ is integer. □

> *Remark* 7.3.2. The proof shows that TUM + integer b is a sufficient condition for integrality. It is worth noting that the converse does not hold in general: a particular polyhedron can be integral even if A is not TUM (the integrality may depend on the specific b). What TUM gives us is integrality for **every** integer b.

The practical consequence is immediate and powerful.

Corollary 7.3.3 (LP relaxation solves the ILP when A is TUM)**.** *If A is TUM and $b \in \mathbb{Z}^m$, then the ILP*

$$\text{maximize} \quad c^\top x \quad \textit{subject to} \quad Ax \le b,\ x \ge 0,\ x \in \mathbb{Z}^n$$

has the same optimal value as its LP relaxation. In particular, any vertex solution returned by an LP solver (for example, the Simplex method, or an interior-point method followed by crossover) is an integer optimum.

When A is TUM: no B&B, no cuts, no rounding. Just solve the LP.

7.4 Recognising TUM: The Equality Form

In many applications the ILP is stated with *equality* constraints rather than inequalities. A useful extension of theorem 7.3.1 covers this case.

Theorem 7.4.1 (TUM with equality constraints)**.** *Let A be TUM and $b \in \mathbb{Z}^m$. Then the polyhedron*

$$P_= = \{x \in \mathbb{R}^n : Ax = b,\ x \ge 0\}$$

is integral.

The equality form is the one we will ent, ork flow ated as

Formal details — Proof of theorem 7.4.1

The key idea: at any vertex of $P_=$, the basis matrix $A_{\mathcal{B}}$ is a square submatrix of A, hence has determinant ± 1 by TUM. Cramer's rule then forces integer values.

Let $\bar{x}$ be any vertex (basic feasible solution) of $P_= = \{x \ge 0 : Ax = b\}$. By the theory of LP, there exists a **basis** $\mathcal{B} \subseteq \{1, \dots, n\}$ with $|\mathcal{B}| = m$ such that the basis matrix $A_{\mathcal{B}}$ (the $m \times m$ submatrix of A formed by the columns in $\mathcal{B}$) is square and invertible, and

$$\bar{x}_{\mathcal{B}} = A_{\mathcal{B}}^{-1} b, \qquad \bar{x}_{\mathcal{N}} = 0,$$

where $\mathcal{N} = \{1, \dots, n\} \setminus \mathcal{B}$ is the set of non-basic variables.

TUM gives $\det(A_\mathcal{B}) = \pm 1$**.** Since A is TUM, every square submatrix has determinant in $\{0, \pm 1\}$. Because $A_\mathcal{B}$ is invertible, $\det(A_\mathcal{B}) \neq 0$, so $\det(A_\mathcal{B}) = \pm 1$.

Cramer's rule gives integer components. For each $i \in \mathcal{B}$, Cramer's rule gives

$$\left(A_\mathcal{B}^{-1} b\right)_i = \frac{\det(\hat{A}_i)}{\det(A_\mathcal{B})},$$

where $\hat{A}_i$ is the matrix obtained from $A_\mathcal{B}$ by replacing column i with b. Since $b \in \mathbb{Z}^m$ and all entries of $A_\mathcal{B}$ are integers (every TUM entry is in $\{0, \pm 1\}$), the matrix $\hat{A}_i$ has integer entries, so $\det(\hat{A}_i) \in \mathbb{Z}$. Dividing by $\det(A_\mathcal{B}) = \pm 1$ yields an integer. Thus $\bar{x}_\mathcal{B} \in \mathbb{Z}^{|\mathcal{B}|}$, and together with $\bar{x}_\mathcal{N} = 0 \in \mathbb{Z}^{|\mathcal{N}|}$ we get $\bar{x} \in \mathbb{Z}^n$. Since $\bar{x}$ was an arbitrary vertex of $P_=$, the polyhedron is integral.

This equality form is particularly natural for network problems, where the constraints express flow conservation at each node.

7.5 Properties of TUM Matrices

These properties tell us that TUM is robust: standard matrix operations do not destroy it.

Total unimodularity is preserved by a number of natural matrix operations. These properties are useful for recognising TUM matrices and for building new TUM matrices from known ones.

Theorem 7.5.1 (Closure properties of TUM)**.** *If A is a totally unimodular matrix, then:*

(i) $A^\top$ *is TUM.*
(ii) Deleting any row or column of A yields a TUM matrix.
(iii) Multiplying any row or column of A by -1 yields a TUM matrix.
(iv) Adding a row or column of all zeros yields a TUM matrix.
(v) Permuting rows or columns of A yields a TUM matrix.
(vi) Stacking an identity matrix below A: the matrix $\binom{A}{I}$ is TUM.

Proof. Each property follows directly from the definition.

(i): Every square submatrix of $A^\top$ is the transpose of a square submatrix of A. Since $\det(M^\top) = \det(M)$, the determinant stays in $\{0, \pm 1\}$.

(ii): Every square submatrix of a submatrix of A is also a square submatrix of A.

(iii): Multiplying a row or column of a submatrix by -1 multiplies the determinant by -1. Since $\{0, \pm 1\}$ is closed under sign change, the property is preserved.

(iv): A row/column of zeros does not change the set of square submatrices (any submatrix including the zero row/column has determinant 0).

(v): Permuting rows or columns of a submatrix changes the determinant by a sign at most.

(vi): Consider a square submatrix S of $\binom{A}{I}$. Each row of S comes either from A or from I. If a row from I is included, it has a single 1 in some column. Expanding by that row (Laplace expansion), we reduce to a smaller submatrix—either of A alone or of $\binom{A}{I}$ again. By induction on the size, $\det(S) \in \{0, \pm 1\}$. □

Property (vi) is particularly useful: it says that if A is TUM, then the augmented system $Ax \leq b$, $x \geq 0$ (which becomes $\binom{A}{-I} x \leq \binom{b}{0}$) still has a TUM coefficient matrix.

7.6 Sufficient Conditions for TUM

Checking the definition directly is impractical: a 5×5 matrix already has hundreds of square submatrices. We need shortcuts.

Verifying TUM from the definition requires checking the determinant of every square submatrix—an exponential number. Fortunately, there is a clean sufficient condition that covers most matrices arising in practice.

Theorem 7.6.1 (Sufficient conditions for TUM)**.** *Let A be an $m \times n$ matrix with entries in $\{0, +1, -1\}$. Suppose:*

*(a) Each column of A has **at most two** non-zero entries.*

(b) The rows of A can be partitioned into two sets R_1 and R_2 such that:

- *if a column has two non-zero entries with the **same sign**, the corresponding rows are in **different sets** (R_1 and R_2);*
- *if a column has two non-zero entries with **different signs**, the corresponding rows are in the **same set**.*

Then A is totally unimodular.

Let us make this concrete before proving it.

Example 7.6.2 (Applying the sufficient conditions)**.** Consider the matrix

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & -1 & 1 \end{pmatrix}.$$

Check condition (a): Column 1 has entries $1, -1, 0$ (two non-zeros). Column 2: $0, 1, -1$ (two). Column 3: $1, 0, -1$ (two). Column 4: $0, 0, 1$ (one). All columns have at most two non-zero entries. ✓

Check condition (b): Let us try $R_1 = \{\text{row 1, row 2}\}$ and $R_2 = \{\text{row 3}\}$.

- Column 1: entries +1 (row 1) and −1 (row 2) → different signs → rows must be in the *same set*. Both in R_1. ✓
- Column 2: entries +1 (row 2) and −1 (row 3) → different signs → rows must be in the *same set*. Row 2 in R_1, row 3 in R_2. ×

That partition does not work. Let us try $R_1 = \{\text{row 1, row 3}\}$ and $R_2 = \{\text{row 2}\}$:

- Column 1: +1 (row 1), −1 (row 2) → different signs → same set? Row 1 in R_1, row 2 in R_2. ×

Also fails. Let us try $R_1 = \{\text{row 1}\}$ and $R_2 = \{\text{row 2, row 3}\}$:

- Column 1: +1 (row 1), −1 (row 2) → different signs → same set? Row 1 in R_1, row 2 in R_2. ×

Finally, try $R_1 = \{\text{row 1, row 2, row 3}\}$, $R_2 = \emptyset$:

- Column 1: $+1, -1$ → different signs → same set. Both in R_1. ✓
- Column 2: $+1, -1$ → different signs → same set. Both in R_1. ✓
- Column 3: $+1, -1$ → different signs → same set. Both in R_1. ✓
- Column 4: only one non-zero → no constraint. ✓

All conditions are met. Therefore A is TUM.

(We will see shortly that this matrix is precisely the node-arc incidence matrix of a directed graph.)

Formal details — Proof of theorem 7.6.1

We prove by induction on k that every $k \times k$ submatrix S of A has $\det(S) \in \{0, \pm 1\}$.

Base case ($k = 1$). A 1×1 submatrix is a single entry, which is in $\{0, +1, -1\}$ by hypothesis. ✓

Inductive step. Let S be a $k \times k$ submatrix of A ($k \geq 2$). We consider three cases based on the columns of S.

Case 1: S has a column with no non-zero entries. Then $\det(S) = 0$. ✓

Case 2: S has a column with exactly one non-zero entry. That entry is ± 1. Expanding $\det(S)$ along that column (Laplace expansion) gives $\det(S) = \pm \det(S')$, where S' is a $(k-1) \times (k-1)$ submatrix of A. By the inductive hypothesis, $\det(S') \in \{0, \pm 1\}$, so $\det(S) \in \{0, \pm 1\}$. ✓

Case 3: Every column of S has exactly two non-zero entries. Consider the partition R_1, R_2 from condition (b). Let r_S denote the rows of S. Define

$$\sigma = \sum_{i \in r_S \cap R_1} (\text{row } i \text{ of } S) - \sum_{i \in r_S \cap R_2} (\text{row } i \text{ of } S).$$

We claim that $\sigma = \mathbf{0}$ (the zero vector). Consider any column j of S. It has exactly two non-zero entries in rows i_1 and i_2.

- If the two entries have the **same sign** ($+1, +1$ or $-1, -1$), then by condition (b), i_1 and i_2 are in *different* sets. One contributes $+a_{i_1 j}$ and the other $-a_{i_2 j}$ to σ_j. Since $a_{i_1 j} = a_{i_2 j}$, we get $\sigma_j = 0$.
- If the two entries have **different signs**, then i_1 and i_2 are in the *same* set. Both contribute with the same sign to σ_j. Since $a_{i_1 j} = -a_{i_2 j}$, we again get $\sigma_j = 0$.

So the rows of S are linearly dependent, and $\det(S) = 0$. ✓

In all cases, $\det(S) \in \{0, \pm 1\}$, completing the induction. □

7.7 Directed Graph Incidence Matrices

Directed graphs are the natural habitat of network flow problems: shortest paths, maximum flow, minimum-cost flow.

The sufficient conditions from theorem 7.6.1 are not just a theoretical curiosity—they are satisfied by some of the most important matrices in combinatorial optimisation. The first major class is the incidence matrix of a directed graph.

Definition 7.7.1 (Node-arc incidence matrix)**.** Let $G = (V, E)$ be a **directed graph** with n nodes and m arcs. The **node-arc incidence matrix** $A \in \{0, +1, -1\}^{n \times m}$ is defined by:

$$a_{ie} = \begin{cases} +1 & \text{if arc } e \textit{ leaves} \text{ node } i, \\ -1 & \text{if arc } e \textit{ enters} \text{ node } i, \\ 0 & \text{otherwise.} \end{cases}$$

In words: each column of A corresponds to an arc, and it has exactly one $+1$ (at the tail node) and one -1 (at the head node). All other entries are 0.

Example 7.7.2 (Node-arc incidence matrix of a small digraph)**.** Consider the directed graph with $V = \{1, 2, 3, 4\}$ and arcs $e_1 = (1, 2)$, $e_2 = (1, 3)$, $e_3 = (2, 3)$, $e_4 = (2, 4)$, $e_5 = (3, 4)$:

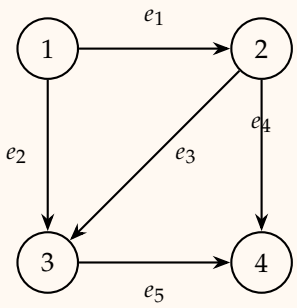


Figure 7.2: A directed graph on 4 nodes and 5 arcs.

The node-arc incidence matrix is:

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{pmatrix} +1 & +1 & 0 & 0 & 0 \\ -1 & 0 & +1 & +1 & 0 \\ 0 & -1 & -1 & 0 & +1 \\ 0 & 0 & 0 & -1 & -1 \end{pmatrix} \end{array}$$

Each column has exactly one +1 and one −1. The column sum is always zero (which reflects the fact that every arc leaves one node and enters another).

Theorem 7.7.3 (Node-arc incidence matrices are TUM)**.** *The node-arc incidence matrix of any directed graph is totally unimodular.*

Proof. We verify the sufficient conditions of theorem 7.6.1.

- **Entries:** All entries are in $\{0, +1, -1\}$. ✓
- **Condition (a):** Each column has exactly two non-zero entries (one +1, one −1), so at most two. ✓
- **Condition (b):** Choose $R_1 = \{\text{all rows}\}$ and $R_2 = \emptyset$. In every column, the two non-zero entries are +1 and −1 (different signs). The condition requires that they be in the *same* set. Since both rows are in R_1, this is satisfied. ✓

By theorem 7.6.1, the matrix is TUM. □

This result has far-reaching consequences, which we explore in section 7.9. But first, let us consider undirected bipartite graphs.

The trivial partition R_1 = all rows, $R_2 = \emptyset$ works because every column has entries of opposite *sign.*

7.8 Bipartite Graph Incidence Matrices

Directed graph incidence matrices have entries +1 and −1, which made the sufficient conditions easy to satisfy (different signs → same set). What about *undirected* graphs, where the incidence matrix has only 0s and +1s?

Undirected bipartite graphs arise in matching and assignment problems. Their incidence matrices are TUM too.

For a general undirected graph, the incidence matrix is *not* necessarily TUM. But for **bipartite** graphs, it is.

Definition 7.8.1 (Incidence matrix of an undirected graph)**.** Let $G = (V, E)$ be an undirected graph with n vertices and m edges. The **vertex-edge**

incidence matrix $A \in \{0,1\}^{n\times m}$ is defined by:

$$a_{ie} = \begin{cases} 1 & \text{if vertex } i \text{ is an endpoint of edge } e, \\ 0 & \text{otherwise.} \end{cases}$$

Each column has exactly two 1s (one for each endpoint of the edge), and all entries are non-negative.

Example 7.8.2 (Incidence matrix of a triangle)**.** Consider the complete graph K_3 on vertices $\{1,2,3\}$ with edges $e_1 = \{1,2\}$, $e_2 = \{1,3\}$, $e_3 = \{2,3\}$:

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}.$$

Is A TUM? Let us check: $\det(A) = 1(0-1) - 1(1-0) + 0 = -2$. So A is **not** TUM. The triangle K_3 is not bipartite, and indeed this is the obstruction.

Example 7.8.3 (Incidence matrix of a bipartite graph)**.** Consider the bipartite graph with parts $U = \{1,2\}$ and $W = \{3,4\}$, and edges $e_1 = \{1,3\}$, $e_2 = \{1,4\}$, $e_3 = \{2,3\}$, $e_4 = \{2,4\}$:

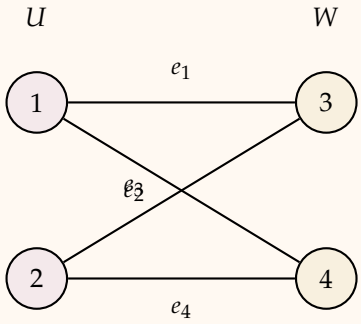


Figure 7.3: A complete bipartite graph $K_{2,2}$. The two vertex classes are highlighted with distinct palette tints. Each edge connects one vertex in U to one in W.

The incidence matrix is:

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} e_1 & e_2 & e_3 & e_4 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

Each column has two 1s (same sign). The partition $R_1 = \{1,2\} = U$ and $R_2 = \{3,4\} = W$ works: in every column, the two 1s come from vertices on *different sides* of the bipartition, hence in *different sets*. This satisfies the sufficient conditions.

Theorem 7.8.4 (Bipartite incidence matrices are TUM)**.** *The vertex-edge incidence matrix of an undirected graph G is totally unimodular if and only if G is bipartite.*

Proof. **(⇒) Bipartite implies TUM.** Let G be bipartite with bipartition $V = U \cup W$. We verify the sufficient conditions of theorem 7.6.1.

- **Entries:** All entries are in $\{0, 1\} \subseteq \{0, +1, -1\}$. ✓
- **Condition (a):** Each column (edge) has exactly two non-zero entries. ✓
- **Condition (b):** Set $R_1 = U$, $R_2 = W$. Every edge connects a vertex in U to a vertex in W, so the two +1 entries (same sign) are in *different* sets. ✓

(⇐) Non-bipartite implies not TUM. If G is not bipartite, it contains an odd cycle $v_1, v_2, \ldots, v_{2k+1}, v_1$. The submatrix of A formed by the $2k+1$ rows (vertices) and $2k+1$ columns (edges) of this cycle is a square matrix where each column has exactly two 1s (the cycle's edges), and each row has exactly two 1s (the cycle's vertices). One can verify (by row operations) that the determinant of this matrix is ± 2. Since $2 \notin \{0, \pm 1\}$, A is not TUM.

We verify this for the smallest odd cycle, the triangle C_3, to build intuition for the general case.

Explicit check for $k = 1$ (the 3-cycle C_3). Label the vertices $1, 2, 3$ and the edges e_{12}, e_{23}, e_{13}. The vertex–edge incidence submatrix is

$$M = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix},$$

and $\det(M) = 1(1 \cdot 1 - 0 \cdot 1) - 0 + 1(1 \cdot 1 - 0 \cdot 1) = 1 + 1 = 2$. (Compare theorem 7.8.2, where a different column ordering gives $\det = -2$.) Hence M is not TUM, confirming the claim for the smallest odd cycle. □

The triangle example

s the "only
ycle (odd),

Intermezzo — Bipartiteness as the dividing line

The theorem above gives an exact characterisation: among undirected graphs, exactly the bipartite ones yield TUM incidence matrices. This is intimately connected to why matching on bipartite graphs is polynomial (solvable via LP), while matching on general graphs requires more sophisticated techniques (Edmonds' blossom algorithm, covered in chapter 12).

Interestingly, for *directed* graphs, the incidence matrix is always TUM regardless of the graph structure—no bipartiteness condition needed. The asymmetry comes from the signs: directed incidence matrices have $+1/-1$ pairs (different signs → trivial partition), while undirected ones have $+1/+1$ pairs (same sign → need a valid bipartition).

7.9 Applications: Where TUM Solves the ILP for Free

We now survey the major problem classes whose constraint matrices are TUM. For all of these, the LP relaxation automatically gives integer solutions, and no Branch and Bound or cutting planes are needed.

7.9.1 Network flow problems

Network flow problems are covered in depth in chapter 11. Here we establish why they are "easy" from the integrality perspective.

The general **minimum-cost network flow** problem on a directed graph $G = (V, E)$ is:

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c_e \, x_e \\ \text{subject to} \quad & \sum_{e \in \delta^+(i)} x_e - \sum_{e \in \delta^-(i)} x_e = b_i \qquad \text{for all } i \in V, \\ & l_e \le x_e \le u_e \qquad \text{for all } e \in E, \end{aligned}$$

where $\delta^+(i)$ and $\delta^-(i)$ are the arcs leaving and entering node i, b_i is the supply/demand at node i, and l_e, u_e are capacity bounds.

In matrix form, the flow conservation constraints are $Ax = b$, where A is the node-arc incidence matrix. By theorem 7.7.3, A is TUM. If b and the capacity bounds l, u are integer, then by theorem 7.3.1 (and its equality variant, theorem 7.4.1), the LP has an integer optimal solution.

Corollary 7.9.1 (Integrality of network flows)**.** *If the supplies/demands b_i and the arc capacities l_e, u_e are all integers, then the minimum-cost network flow LP has an integer optimal solution.*

This single result explains why many classical network problems have integer solutions:

- **Shortest path**: a network flow problem with one unit of flow from source to sink. The LP solution assigns $x_e \in \{0, 1\}$—the arcs of a shortest path.
- **Maximum flow**: the dual of a min-cost flow. The max flow is integer when capacities are integer (the celebrated **max-flow min-cut theorem** also follows from LP duality and TUM).
- **Transportation problem**: flow on a bipartite network. Supplies and demands are integer, so the LP gives integer shipments.

7.9.2 The assignment problem

This fulfils the promise from theorem 3.2.3 in chapter 3: the assignment LP relaxation is always integer.

The assignment problem (theorem 3.2.3) assigns n workers to n jobs to minimise total cost:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} \, x_{ij} \\ \text{subject to} \quad & \sum_{j=1}^{n} x_{ij} = 1 \quad \text{for all } i, \\ & \sum_{i=1}^{n} x_{ij} = 1 \quad \text{for all } j, \\ & x_{ij} \ge 0. \end{aligned}$$

The constraint matrix is the incidence matrix of the complete bipartite graph $K_{n,n}$: rows correspond to workers and jobs (the two sides of the bipartition), and columns correspond to worker-job pairs (edges). Each column has exactly two 1s—one in the worker's row and one in the job's row.

By theorem 7.8.4, this matrix is TUM. Since $b = \mathbf{1}$ is integer, the LP relaxation polyhedron is integral. Therefore:

Corollary 7.9.2 (Assignment LP relaxation is integer)**.** *The LP relaxation of the assignment problem has an integer optimal solution $x_{ij} \in \{0,1\}$. The integrality constraint $x_{ij} \in \{0,1\}$ is redundant.*

This is why the assignment problem can be solved efficiently: we can drop the integrality requirement and solve the resulting LP directly. Specialised algorithms (the Hungarian method) exploit the structure further, but the fact that "LP is enough" is the TUM miracle.

Example 7.9.3 (A 3×3 assignment)**.** Three workers, three jobs, with cost matrix:

$$C = \begin{pmatrix} 5 & 9 & 3 \\ 8 & 7 & 2 \\ 6 & 4 & 1 \end{pmatrix}.$$

The LP relaxation (dropping $x_{ij} \in \{0,1\}$, keeping only $x_{ij} \geq 0$ and the row/column sum constraints) automatically gives an integer solution. In this case, the optimal assignment is worker 1 $\to$ job 1 (cost 5), worker 2 $\to$ job 3 (cost 2), worker 3 $\to$ job 2 (cost 4), with total cost 11.

The LP relaxation gives exactly this assignment—no rounding, no branching needed.

7.9.3 Matching on bipartite graphs

The maximum matching problem on a bipartite graph $G = (U \cup W, E)$ asks for the largest set of edges with no shared endpoints:

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in E} x_e \\ \text{subject to} \quad & \sum_{e \in \delta(v)} x_e \leq 1 \quad \text{for all } v \in U \cup W, \\ & x_e \geq 0. \end{aligned}$$

Here $\delta(v)$ denotes the edges incident to vertex v. The constraint matrix is the incidence matrix of the bipartite graph G (with inequality constraints ≤ 1). By theorem 7.8.4, this matrix is TUM. Since $b = \mathbf{1}$ is integer, the LP gives an integer solution with $x_e \in \{0,1\}$.

Corollary 7.9.4 (Bipartite matching LP is integer)**.** *The LP relaxation of the maximum matching problem on a bipartite graph has an integer optimal solution.*

This is one of the most important consequences of TUM in combinatorial optimisation. It explains why bipartite matching is solvable in polynomial time, while general matching requires more sophisticated combinatorial techniques.

For non-bipartite graphs, the matching LP relaxation can give $x_e = 1/2$ on odd cycles. Edmonds' blossom algorithm handles this case.

7.9.4 The shortest path LP

The shortest path from a source node s to a sink node t in a directed graph with arc costs $c_e \geq 0$ can be formulated as a network flow:

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c_e \, x_e \\ \text{subject to} \quad & \sum_{e \in \delta^+(i)} x_e - \sum_{e \in \delta^-(i)} x_e = \begin{cases} 1 & i = s, \\ -1 & i = t, \\ 0 & \text{otherwise}, \end{cases} \\ & x_e \geq 0. \end{aligned}$$

The constraint matrix is the node-arc incidence matrix (TUM by theorem 7.7.3) and the right-hand side is integer. The LP optimal solution is integer, with $x_e \in \{0, 1\}$: the arcs with $x_e = 1$ form a shortest path from s to t.

7.9.5 The transportation problem

The transportation problem ships goods from m suppliers to n consumers to minimise total shipping cost:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} \, x_{ij} \\ \text{subject to} \quad & \sum_{j=1}^{n} x_{ij} = s_i \qquad \text{for all suppliers } i, \\ & \sum_{i=1}^{m} x_{ij} = d_j \qquad \text{for all consumers } j, \\ & x_{ij} \geq 0. \end{aligned}$$

The constraint matrix has the same structure as the assignment matrix but with general supplies s_i and demands d_j instead of all 1s. It is the incidence matrix of the complete bipartite graph $K_{m,n}$, hence TUM by theorem 7.8.4. When supplies and demands are integer, the LP gives integer shipments.

7.10 A Non-Example: Why the Knapsack Matrix Is Not TUM

TUM is powerful but limited. The knapsack problem, which drove chapter 3 and chapter 6, is not *covered by TUM.*

After this parade of successes, let us understand the limits of TUM. Not every ILP has a TUM constraint matrix. The running knapsack from chapter 3 is a prime counterexample.

Recall the knapsack constraint:

$$5x_1 + 3x_2 + 7x_3 + 4x_4 + 2x_5 + 6x_6 \leq 15.$$

The coefficient 5 is not in $\{0, \pm 1\}$, so the constraint matrix cannot be TUM (theorem 7.2.3). This is why the knapsack LP relaxation gave a fractional solution in chapter 3, and we needed Branch and Bound (chapter 6) to find the integer optimum.

More generally, TUM applies to problems with a **combinatorial structure** (networks, bipartite graphs, assignments) where the constraint matrix is sparse and has $\{0, \pm 1\}$ entries. Problems with arbitrary integer coefficients (knapsack, general ILP) require the full machinery of B&B and cutting planes.

Observation 7.10.1 (When to look for TUM). A constraint matrix is a good candidate for TUM when:

1. All entries are in $\{0, +1, -1\}$.
2. Each column has at most two non-zero entries.
3. The constraints arise from a network or bipartite graph structure.

If the matrix has entries outside $\{0, \pm 1\}$, it cannot be TUM.

7.11 The Big Picture: From ILP Hardness to TUM Tractability

Let us step back and see where TUM fits in our developing understanding of integer programming.

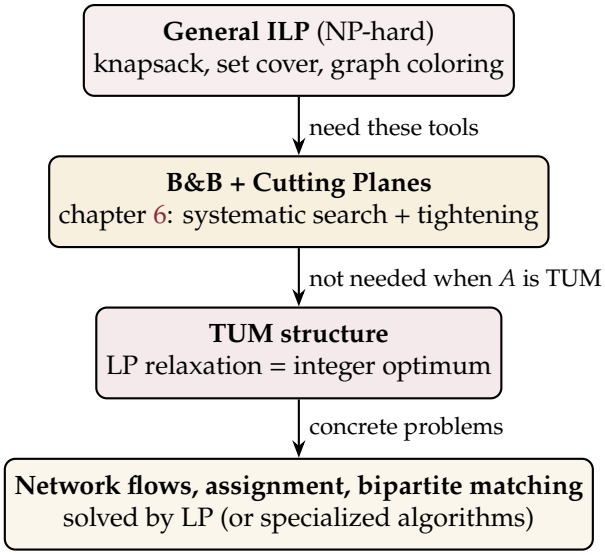


Figure 7.4: The landscape of ILP solvability. General ILPs are NP-hard and require Branch & Bound or cutting planes. When the constraint matrix is TUM, the LP relaxation solves the ILP directly, yielding a class of efficiently solvable problems.

In chapter 3 we learned that ILP is NP-hard in general. In chapter 6 we developed Branch and Bound and cutting planes to tackle this hardness. Now we have identified a structural condition —total unimodularity—under which the hardness vanishes: the LP relaxation is exact, and we can solve the ILP in polynomial time.

This is not just a theoretical curiosity. The problems in the TUM class (network flows, shortest paths, assignment, transportation, bipartite matching) are among the most frequently solved optimisation problems in practice. Millions of routing, scheduling, and assignment decisions are made daily, and TUM is the reason these computations are fast.

7.12 Chapter Summary

- A polyhedron is **integral** (theorem 7.1.1) if every vertex has integer coordinates. When the LP feasible region is integral, the LP optimum is automatically integer.
- An ILP is **naturally integer** (theorem 7.1.3) if its LP relaxation polyhedron is integral for every integer b. No B&B or cuts needed.

- A matrix A is **totally unimodular** (TUM, theorem 7.2.2) if every square submatrix has determinant in $\{0, +1, -1\}$. This forces all entries to be in $\{0, \pm 1\}$.
- **Main theorem** (theorem 7.3.1): if A is TUM and b is integer, the polyhedron $\{x : Ax \leq b,\ x \geq 0\}$ is integral. Proof via Cramer's rule: vertex coordinates are ratios of integer determinants with denominator ± 1.
- TUM is preserved under transpose, deletion, sign change, zero-padding, and permutation (theorem 7.5.1).
- **Sufficient conditions** (theorem 7.6.1): entries in $\{0, \pm 1\}$, at most two non-zeros per column, and a row partition respecting sign patterns $\Rightarrow$ TUM.
- **Directed graph** incidence matrices are always TUM (theorem 7.7.3). Consequence: network flow problems (shortest path, max flow, min-cost flow) have integer LP solutions when data is integer.
- **Undirected graph** incidence matrices are TUM if and only if the graph is bipartite (theorem 7.8.4). Consequence: matching and assignment on bipartite graphs are naturally integer.
- The **assignment problem** LP relaxation is always integer (theorem 7.9.2), fulfilling the promise from chapter 3.
- Problems with arbitrary coefficients (knapsack, general ILP) do **not** have TUM matrices and still require B&B / cuts.

Looking ahead. In chapter 8 we begin the graph optimisation part of the course. The TUM results from this chapter guarantee that the LP formulations of network problems have integer solutions; in the graph chapters, we develop *combinatorial algorithms* (BFS, Dijkstra, Ford–Fulkerson, Hungarian) that exploit the graph structure to solve these problems even more efficiently than a general LP solver.

Summary & Key Takeaways

- **Total Unimodularity (TUM):** A matrix A is TUM if the determinant of every square submatrix is $0, +1$, or -1.
- **Integrality Property:** If A is TUM and $b \in \mathbb{Z}^m$, the polyhedron $P = \{x \geq 0 : Ax \leq b\}$ has only integer vertices. Hence, the LP relaxation solves the ILP exactly.
- **Poincaré-Ghouila-Houri Theorem:** A matrix A with entries in $\{0, 1, -1\}$ is TUM if its rows can be partitioned into two sets R_1, R_2 such that for any column, the sum of entries in R_1 minus the sum of entries in R_2 lies in $\{0, 1, -1\}$.
- **Incidence Matrices:** The node-vertex incidence matrix of a bipartite graph (or node-arc incidence matrix of a directed graph) is always TUM.

Exercises

Exercise 1 (TU check: a 2×3 matrix). Let

$$A = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & 1 \end{pmatrix}.$$

1. List every square submatrix of A (sizes 1×1 and 2×2).
2. Compute all subdeterminants and verify that each belongs to $\{-1, 0, 1\}$.
3. Conclude whether A is totally unimodular.

Exercise 2 (TU check: a 2×3 matrix with a violation). Let

$$B = \begin{pmatrix} 1 & 1 & 0 \\ 1 & -1 & 1 \end{pmatrix}.$$

1. Compute all 2×2 subdeterminants.
2. Find a 2×2 submatrix whose determinant lies outside $\{-1, 0, 1\}$.
3. Conclude that B is *not* totally unimodular.

Exercise 3 (TU check: another 2×3 matrix). Let

$$C = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \end{pmatrix}.$$

Determine whether C is totally unimodular by computing all subdeterminants of sizes 1×1 and 2×2.

Exercise 4 (TU check: a 3×3 identity-like matrix). Let

$$D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & -1 \end{pmatrix}.$$

1. Compute $\det(D)$.
2. For each 2×2 submatrix, compute its determinant.
3. Is D totally unimodular? Justify.

Exercise 5 (TU check: a 3×3 matrix with off-diagonal entries). Let

$$E = \begin{pmatrix} 1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & -1 \end{pmatrix}.$$

1. Compute $\det(E)$.
2. Examine every 2×2 minor.
3. Determine whether E is totally unimodular.

Exercise 6 (Non-TU 3×3 matrix). Let

$$F = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Find a square submatrix of F whose determinant is not in $\{-1, 0, 1\}$, and conclude that F is not totally unimodular.

Exercise 7 (Incidence matrix of a directed path). Consider the directed graph $G = (V, A)$ with vertices $V = \{1, 2, 3, 4\}$ and arcs $A = \{(1, 2), (2, 3), (3, 4), (1, 3)\}$.

1. Write the node-arc incidence matrix $M \in \{-1, 0, 1\}^{4 \times 4}$.
2. Compute $\det(M)$ (after choosing any three rows if convenient).
3. Verify that every 2×2 submatrix has determinant in $\{-1, 0, 1\}$.
4. State the general theorem that guarantees M is totally unimodular.

Exercise 8 (Incidence matrix of a directed triangle). Let G be the directed triangle with vertices $\{1, 2, 3\}$ and arcs $(1, 2), (2, 3), (3, 1)$.

1. Write the node-arc incidence matrix $M \in \mathbb{R}^{3\times 3}$.
2. Compute $\det(M)$.
3. Is M totally unimodular?
4. Note: all rows of M sum to the zero vector. What does this imply about the rank of M?

Exercise 9 (Applying TUM to a min-cost flow LP). A directed network has n nodes and m arcs. The min-cost flow LP is

$$\min\ c^\top x \quad \text{s.t.} \quad Bx = b,\ 0 \le x \le u,$$

where B is the node-arc incidence matrix, $b \in \mathbb{Z}^n$, $c, u \in \mathbb{Z}^m$.

1. Why is B totally unimodular?
2. Explain why the LP always has an integer optimal solution when b and u are integer. Which theorem do you need?
3. Does the same argument apply if we drop the upper-bound constraints $x \le u$? Briefly explain.

Exercise 10 (Bipartite graph: incidence matrix is TU). Let $G = (U \cup W, E)$ be the complete bipartite graph $K_{2,2}$ with $U = \{u_1, u_2\}$, $W = \{w_1, w_2\}$, and edges $E = \{u_1w_1,\ u_1w_2,\ u_2w_1,\ u_2w_2\}$.

1. Write the node-edge incidence matrix $N \in \{0,1\}^{4\times 4}$.
2. Verify that every 2×2 subdeterminant lies in $\{-1, 0, 1\}$.
3. Identify the row partition (one set per bipartition side) and verify the Ghouila-Houri condition for each column.

Exercise 11 (Non-bipartite graph: incidence matrix is NOT TU). Let G be the undirected triangle with vertices $\{1, 2, 3\}$ and edges $\{12,\ 23,\ 13\}$.

1. Write the node-edge incidence matrix $N \in \{0,1\}^{3\times 3}$.
2. Compute $\det(N)$.
3. Explain why $|\det(N)| \notin \{0, 1\}$ implies N is not totally unimodular.
4. State the general theorem characterising which undirected graphs have a TU incidence matrix.

Exercise 12 (Odd cycle implies non-TU). Let C_5 be the undirected cycle on vertices $\{1, 2, 3, 4, 5\}$ with edges $\{12, 23, 34, 45, 51\}$.

1. Write the 5×5 node-edge incidence matrix N.
2. Without computing the full determinant, argue using the bipartite-TUM theorem that N is not totally unimodular.
3. Exhibit explicitly a square submatrix of N with determinant ± 2. (*Hint:* use a 3×3 submatrix corresponding to three vertices and three edges forming a triangle within C_5 is not possible; instead use a 3×3 submatrix of your choice and compute.)

Exercise 13 (Ghouila-Houri on a network matrix). Consider the matrix

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & -1 & 1 \\ 0 & 0 & 0 & -1 \end{pmatrix}.$$

Each column has exactly two non-zero entries: one $+1$ and one -1.

1. Partition the rows into two sets R_1 and R_2 such that for every column the sum of entries in R_1 minus the sum in R_2 belongs to $\{-1, 0, 1\}$.
2. State the Ghouila-Houri theorem and explain why your partition verifies that A is totally unimodular.

Exercise 14 (Applying Ghouila-Houri to a $0/1$ matrix). Let

$$M = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}.$$

1. State the Ghouila-Houri sufficient condition in full.
2. Attempt to find a row partition satisfying the condition for every column of M.
3. If no such partition exists, find a 2×2 submatrix with $|\det| = 2$ to confirm M is not TU.

Exercise 15 (Ghouila-Houri: constructive verification). Suppose a $0/\pm1$ matrix A has the property that every column contains at most one $+1$ and at most one -1, with all other entries zero.

1. Show that the row partition $R_1 = \{\text{all rows}\}$, $R_2 = \emptyset$ satisfies the Ghouila-Houri condition for every column.
2. Conclude that A is totally unimodular.
3. Give a 3×3 example with entries in $\{0, \pm1\}$, at most two non-zeros per column, that illustrates your conclusion.

Exercise 16 (True/False: basic TUM statements). For each statement, decide whether it is true or false and give a brief justification or counterexample.

1. Every $0/1$ matrix is totally unimodular.
2. If M is totally unimodular, then $2M$ is totally unimodular.
3. If M is totally unimodular, then $M^\top$ is totally unimodular.
4. Every submatrix of a totally unimodular matrix is totally unimodular.
5. The matrix $\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$ is totally unimodular.

Exercise 17 (True/False: TUM and operations). Decide true or false, with justification:

1. Adding a column of zeros to a TU matrix preserves total unimodularity.
2. Multiplying a single row of a TU matrix by -1 preserves total unimodularity.
3. If A is TU and B is TU, then $\begin{pmatrix} A \\ B \end{pmatrix}$ is TU.
4. If A is TU and $b \in \mathbb{Z}^m$, then every vertex of $\{x \geq 0 : Ax \leq b\}$ is integer.
5. The transpose of the node-arc incidence matrix of a directed graph is TU.

Exercise 18 (True/False: integral polyhedra). Decide true or false, with justification:

1. If $P = \{x : Ax = b,\ x \geq 0\}$ is an integral polyhedron, then A must be totally unimodular.
2. A polyhedron P is integral if and only if every LP over P with integer objective has an integer optimal.
3. If A is TU and b is integer, the LP $\min\{c^\top x : Ax \leq b, x \geq 0\}$ always has an integer optimal solution (assuming it is bounded).
4. The knapsack polytope $\{x \in [0,1]^n : \sum_j a_j x_j \leq b\}$ is integral for all non-negative integer a_j, b.

Exercise 19 (Integral vertices via TUM). Let $A \in \{0, \pm1\}^{m\times n}$ be totally unimodular and $b \in \mathbb{Z}^m$. Define $P = \{x \in \mathbb{R}^n : Ax \leq b,\ x \geq 0\}$.

1. Let x^* be a vertex of P. Write the system of equations that characterises x^* as the unique solution to n linearly independent tight constraints.

2. Express x^* as the solution to a square linear system $\hat{A}x = \hat{b}$ where $\hat{A}$ is a square submatrix of the augmented constraint matrix and $\hat{b}$ is the corresponding right-hand side.
3. Use Cramer's rule and the fact that $|\det(\hat{A})| \in \{0, 1\}$ to conclude that $x^* \in \mathbb{Z}^n$.

Exercise 20 (Integral polyhedron: small example). Consider

$$P = \left\{ x \in \mathbb{R}^2 \,:\, x_1 + x_2 \le 3,\ x_1 - x_2 \le 1,\ x_1, x_2 \ge 0 \right\}.$$

1. Write the constraint matrix A and check that it is totally unimodular by computing all subdeterminants.
2. Find all vertices of P by solving the equality systems from pairs of tight constraints.
3. Verify that every vertex is integer.

Exercise 21 (When does LP = ILP?). Let A be an $m \times n$ totally unimodular matrix and $b \in \mathbb{Z}^m$. Consider the integer program

$$z^* = \max\{c^\top x : Ax \le b,\ x \in \mathbb{Z}^n_{\ge 0}\}.$$

1. State precisely what it means for the LP relaxation to *solve the ILP for free*.
2. Using the TUM $\Rightarrow$ integral polyhedra theorem, explain why z^* equals the LP relaxation optimum z^*_{LP}.
3. Give a concrete 2×2 example (matrix, right-hand side, objective) illustrating this.

Exercise 22 (Transpose preserves TU). Let A be a totally unimodular matrix.

1. Prove that $A^\top$ is also totally unimodular. (*Hint:* Every square submatrix of $A^\top$ is the transpose of a square submatrix of A; use $\det(B^\top) = \det(B)$.)
2. Give a 2×3 example: write A, verify TU for A, then verify TU for $A^\top$.

Exercise 23 (Adding a zero row preserves TU). Let A be a totally unimodular matrix. Let A' be the matrix obtained by appending a row of zeros to A.

1. Prove that A' is totally unimodular. (*Hint:* Consider any square submatrix S' of A'. If S' contains the zero row, expand along that row; otherwise S' is a submatrix of A.)
2. Give an explicit 3×2 example.

Exercise 24 (Adding a standard basis row preserves TU). Let A be a totally unimodular matrix and let $e_j^\top$ be the j-th standard basis row (one 1 in position j, zeros elsewhere). Let A' be the matrix formed by appending $e_j^\top$ to A.

1. Prove that A' is totally unimodular.
2. Explain the consequence for the constraint matrix of an LP whose constraints include $x_j \le u_j$ for integer upper bound u_j.

Exercise 25 (Negating a row preserves TU). Let A be totally unimodular and let A' be obtained from A by multiplying one row by -1.

1. Prove that A' is totally unimodular.
2. Use this to show that the matrix $\begin{pmatrix} A \\ -A \end{pmatrix}$ is totally unimodular whenever A is.
3. What LP constraint set does $\begin{pmatrix} A \\ -A \end{pmatrix} x \le \begin{pmatrix} b \\ -\ell \end{pmatrix}$ represent?

Exercise 26 (Shortest-path LP and TU). The shortest-path LP on a directed

graph $G = (V, A)$ from source s to sink t is

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij} \quad \text{s.t.} \quad \sum_{j} x_{ji} - \sum_{j} x_{ij} = b_i \;\; \forall i \in V, \quad x \geq 0,$$

where $b_s = -1$, $b_t = 1$, $b_i = 0$ otherwise.

1. Identify the constraint matrix and state why it is totally unimodular.
2. Why does TUM guarantee that the LP has an integer (hence 0/1) optimal solution when $c \in \mathbb{Z}^{|A|}$?
3. Interpret the integer solution as a path from s to t.

Exercise 27 (Network flow: TU and integrality). Consider a max-flow instance on a directed graph $G = (V, A)$ with integer capacities $u_{ij} \in \mathbb{Z}_{\geq 0}$. The LP relaxation is

$$\max \; v \quad \text{s.t.} \quad Bx + (\text{source/sink columns}) = 0, \; 0 \leq x \leq u,$$

where B is the node-arc incidence matrix.

1. Argue that the full constraint matrix (including upper-bound constraints) is TU.
2. Conclude that the max-flow LP always has an integer optimum.
3. State Ford–Fulkerson's integrality theorem and identify it as a consequence of TUM.

Exercise 28 (Assignment problem: LP formulation and TU). The assignment problem on a complete bipartite graph $K_{n,n}$ (workers $I = \{1, \dots, n\}$, jobs $J = \{1, \dots, n\}$, cost c_{ij}) has LP relaxation

$$\min \sum_{i,j} c_{ij} x_{ij} \quad \text{s.t.} \quad \sum_{j} x_{ij} = 1 \;\forall i, \quad \sum_{i} x_{ij} = 1 \;\forall j, \quad x_{ij} \geq 0.$$

1. Write the constraint matrix A explicitly for the case $n = 2$ (four variables $x_{11}, x_{12}, x_{21}, x_{22}$).
2. Identify A as the node-edge incidence matrix of which graph?
3. Explain why A is TU and why this guarantees an integer optimal solution to the LP.
4. What does an integer optimal x^* represent combinatorially?

Exercise 29 (Bipartite matching: LP = ILP). The maximum bipartite matching LP on $G = (U \cup W, E)$ is

$$\max \sum_{e\in E} x_e \quad \text{s.t.} \quad \sum_{e \ni v} x_e \leq 1 \;\forall v \in U \cup W, \quad x_e \geq 0.$$

1. Identify the constraint matrix with the node-edge incidence matrix of a bipartite graph.
2. State why this matrix is TU.
3. Conclude that the LP optimal value equals the maximum matching size.
4. Contrast with a non-bipartite graph: why does the argument fail?

Exercise 30 (Transportation problem: constraint matrix and TU). The transportation problem has supply nodes $\{s_1, \dots, s_m\}$, demand nodes $\{d_1, \dots, d_n\}$, and a variable $x_{ij} \geq 0$ for each (i, j) pair. The constraints are $\sum_j x_{ij} = a_i$ (supply) and $\sum_i x_{ij} = b_j$ (demand).

1. For $m = n = 2$, write the full constraint matrix $A \in \{0, 1\}^{4\times 4}$ (rows = supply/demand constraints, columns = variables in order $x_{11}, x_{12}, x_{21}, x_{22}$).

2. Verify that every column of A has exactly two 1s (one supply, one demand row) and all other entries 0.
3. Apply the Ghouila-Houri condition (with the natural row partition: supply rows vs. demand rows) to confirm A is TU.
4. Conclude that the transportation LP always has an integer optimal when $a_i, b_j \in \mathbb{Z}_{\geq 0}$.

Exercise 31 (Knapsack is NOT TU). The single-constraint knapsack problem has constraint matrix $A = (a_1, \ldots, a_n)$, a single row vector with positive integer entries.

1. Take $n = 2$ and $a_1 = 3, a_2 = 2$. Write the 1×2 matrix A and check whether it is TU.
2. Now consider the 2×2 matrix formed by stacking A with a row that bounds x_1: $A' = \begin{pmatrix} 3 & 2 \\ 1 & 0 \end{pmatrix}$. Compute $\det(A')$ and verify $|\det(A')| \notin \{0, 1\}$.
3. Conclude that A' is not TU and hence the knapsack LP relaxation need not have an integer optimum.
4. Give a specific right-hand side b for which the LP relaxation optimum is fractional while the ILP optimum is strictly smaller.

Exercise 32 (General knapsack: finding a non-TU submatrix). Let $A = \begin{pmatrix} 5 & 3 \\ 1 & 0 \end{pmatrix}$.

1. Compute $\det(A)$.
2. Is A totally unimodular?
3. Provide the 2×2 knapsack-type constraint matrix $\begin{pmatrix} a_1 & a_2 \end{pmatrix}$ with $a_1, a_2 \geq 2$ for which a 2×2 augmented matrix has $|\det| = 2$; show the computation explicitly.

Exercise 33 (ILP solved by LP relaxation: 2-variable example). Consider the integer program

$$\max\ 3x_1 + 2x_2 \quad \text{s.t.} \quad x_1 + x_2 \leq 4,\ x_1 - x_2 \leq 2,\ x_1, x_2 \in \mathbb{Z}_{\geq 0}.$$

1. Write the constraint matrix A and verify it is totally unimodular.
2. Solve the LP relaxation graphically or by the simplex method.
3. Verify that the LP optimum is integer and hence also optimal for the ILP.

Exercise 34 (ILP solved by LP relaxation: assignment instance). Three workers $\{1, 2, 3\}$ must be assigned to three jobs $\{A, B, C\}$ (one each). The costs are:

$$c = \begin{pmatrix} 9 & 2 & 7 \\ 3 & 6 & 4 \\ 1 & 8 & 5 \end{pmatrix}.$$

The assignment LP has constraint matrix that is the node-edge incidence matrix of $K_{3,3}$.

1. Why is this matrix totally unimodular?
2. Formulate the LP (6 equality constraints, 9 variables) and state that its optimal solution is integer.
3. Find the minimum-cost assignment by inspection or the Hungarian method and verify it is also the LP optimum.

Exercise 35 (ILP: checking TU before solving). You are given the ILP

$$\max\ x_1 + 2x_2 + x_3 \quad \text{s.t.} \quad \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix},\ x \in \mathbb{Z}^3_{\geq 0}.$$

1. Compute all 2×2 and the 3×3 subdeterminants of the constraint matrix.
2. Is the matrix totally unimodular?
3. If yes, solve the LP relaxation and conclude it also solves the ILP. If no, explain why TUM cannot be used here.

Exercise 36 (Recognising TU: equality-form system). The equality-form LP $\min\{c^\top x : Ax = b,\ x \geq 0\}$ has constraint matrix A. A sufficient condition for TU in equality form is that every square submatrix of A has determinant in $\{-1, 0, 1\}$.

1. Verify this condition for the matrix $A = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.
2. Is A totally unimodular?

Exercise 37 (Big picture: which problems are "naturally integer"?). For each of the following problems, state (a) the constraint matrix, (b) whether it is TU, and (c) whether the LP relaxation always yields an integer solution for integer data. Give a one-sentence justification for each.

1. Shortest path on a directed graph.
2. Maximum matching on a bipartite graph.
3. Maximum matching on a general (non-bipartite) graph.
4. Knapsack.
5. Transportation problem with integer supplies and demands.

Exercise 38 (Column scaling does NOT preserve TU in general). Let A be the 2×2 identity matrix I_2, which is trivially TU.

1. Multiply the first column of A by 2 to obtain A'. Is A' TU?
2. Multiply the first column of A by -1 to obtain A''. Is A'' TU?
3. Summarise: which column operations preserve TU, and which do not?

Exercise 39 (TU and box constraints). Let A be a totally unimodular matrix, and let $\ell, u \in \mathbb{Z}^n$ with $\ell \leq u$. Consider the polyhedron

$$P = \{x \in \mathbb{R}^n : Ax \leq b,\ \ell \leq x \leq u\}.$$

1. Rewrite the box constraints $\ell \leq x \leq u$ as two sets of inequality constraints using identity matrices.
2. Show that the augmented constraint matrix (stacking A, I, $-I$) is TU.
3. Conclude that every vertex of P is integer when b, ℓ, u are integer.

Exercise 40 (Definition drill: TU vs. unimodular).

1. State the definition of a *unimodular* matrix.
2. State the definition of a *totally unimodular* matrix.
3. Give an example of a matrix that is unimodular but not TU.
4. Give an example of a square TU matrix. Is every square TU matrix also unimodular? Justify.
5. True or false: a matrix A is TU if and only if $\det(A) \in \{-1, 0, 1\}$.

Exercise 41 (TU check: a 3×4 directed-arc-type matrix). Let

$$A = \begin{pmatrix} 1 & -1 & 0 & 1 \\ 0 & 1 & -1 & 0 \\ -1 & 0 & 1 & -1 \end{pmatrix}.$$

Each column has exactly one $+1$ and one -1 (and one 0).

1. Verify this pattern for every column.

2. Without computing all 3×3 subdeterminants, explain why the Ghouila-Houri condition (partition rows $R_1 = \{1, 3\}$, $R_2 = \{2\}$) confirms that A is TU.
3. Compute two 3×3 subdeterminants explicitly to confirm.

Exercise 42 (Rank and TU: low-rank example). Let $A = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$.

1. Compute $\det(A)$.
2. Is A totally unimodular? Justify.
3. What is the rank of A? Does low rank imply TU? Give a counterexample or proof.

Exercise 43 (Stacking two TU matrices). Let A and B be two TU matrices with the same number of columns. Is the matrix $\begin{pmatrix} A \\ B \end{pmatrix}$ necessarily TU?

1. State and prove (or disprove with a counterexample) the general claim.
2. Apply your conclusion to the constraint matrix of the LP $\{x : Ax \leq b,\ Bx \leq d,\ x \geq 0\}$ when both A and B are TU.

Exercise 44 (Max-weight bipartite matching via LP). Given the bipartite graph $K_{2,3}$ with weight matrix

$$W = \begin{pmatrix} 4 & 1 & 3 \\ 2 & 5 & 2 \end{pmatrix}$$

(rows = workers $\{u_1, u_2\}$, columns = jobs $\{v_1, v_2, v_3\}$):

1. Write the LP relaxation of the maximum-weight matching problem (each worker matched to at most one job, each job to at most one worker).
2. Identify the constraint matrix and explain why it is TU.
3. Solve the LP (the matrix has six variables); verify the optimum is integer and identify the optimal matching.

Exercise 45 (Network simplex and integer bases). A basic feasible solution of $\min\{c^\top x : Bx = b,\ x \geq 0\}$ corresponds to choosing a basis $\mathcal{B}$ of m linearly independent columns of B.

1. If B is the node-arc incidence matrix of a connected directed graph on n nodes, explain why a basis of B corresponds to a spanning tree.
2. Show that the basis matrix $B_{\mathcal{B}}$ (the $n - 1$ columns of a spanning tree) satisfies $|\det(B_{\mathcal{B}})| = 1$.
3. Conclude that the basic feasible solution $x_{\mathcal{B}} = B_{\mathcal{B}}^{-1} b$ is always integer when b is integer.

Exercise 46 (Sensitivity of integral polyhedra to perturbations). Let A be TU and $b \in \mathbb{Z}^m$. Consider perturbing b to $b' = b + \delta$ where $\delta \in \mathbb{Z}^m$.

1. Is the perturbed polyhedron $P' = \{x \geq 0 : Ax \leq b'\}$ still integral? Justify.
2. Now perturb b to $b'' = b + \varepsilon \mathbf{1}$ for small $\varepsilon \in (0, 1)$. Is $P'' = \{x \geq 0 : Ax \leq b''\}$ necessarily integral?
3. Summarise: integrality of the polyhedron depends on A being TU *and* b being integer. Give a concrete 1×1 example showing the polyhedron is not integral when $b \notin \mathbb{Z}$.

Exercise 47 (Incidence matrix of a directed complete graph K_3). Orient all edges of K_3 (three vertices, three arcs forming a tournament): arcs $(1, 2)$, $(1, 3)$, $(2, 3)$.

1. Write the 3×3 node-arc incidence matrix M.
2. Compute $\det(M)$.

3. Verify TU by checking all 1×1, 2×2, and 3×3 subdeterminants.

Exercise 48 (Constructing an integral polyhedron from data). You are given the system

$$x_1 + x_2 \le 5, \quad x_1 \le 3, \quad x_2 \le 4, \quad x_1, x_2 \ge 0.$$

1. Write the constraint matrix $A \in \{0,1\}^{3\times 2}$ and right-hand side b.
2. Check that A is TU by listing all square submatrices.
3. Find all vertices of the polyhedron $P = \{x \ge 0 : Ax \le b\}$ and verify they are all integer.
4. State the LP $\max\{3x_1 + 2x_2 : x \in P\}$ and identify the optimal vertex.

Exercise 49 (Min-cost assignment: TU guarantees optimality). Prove from first principles (without invoking the Hungarian algorithm) that the minimum-cost assignment LP always has an integer optimal solution.

1. Identify the LP constraint matrix as the node-edge incidence matrix of $K_{n,n}$.
2. State why $K_{n,n}$ is bipartite.
3. Apply the bipartite-TUM theorem and the TUM $\Rightarrow$ integral polyhedra theorem.
4. Conclude: every vertex of the assignment polytope is $0/1$, hence every LP optimal solution is integer, hence the LP equals the ILP.

Exercise 50 (Two-commodity flow and loss of TU). A two-commodity flow network routes two commodities simultaneously. The constraint matrix of a two-commodity flow LP is a stack of two node-arc incidence matrices (one per commodity), interleaved with capacity constraints.

1. Explain why a single node-arc incidence matrix B is TU.
2. Show by example that the matrix $\begin{pmatrix} B & 0 \\ 0 & B \\ I & I \end{pmatrix}$ (for a simple 2-node, 1-arc graph, so $B = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$) may fail to be TU by exhibiting a 2×2 submatrix with $|\det| > 1$.
3. Discuss: why does multi-commodity flow generally require integer programming techniques instead of pure LP?

Exercise 51 (Recognising TU from column structure). A matrix $A \in \{0,1\}^{m\times n}$ has the *consecutive-ones property* (each row has its 1s in a consecutive block of columns) and each column sum is at most 2.

1. Give a 3×4 example of such a matrix.
2. Check whether your example is TU by computing all 2×2 and 3×3 subdeterminants.
3. State which of the known sufficient conditions (Ghouila-Houri or directed-graph theorem) might apply, if any, and verify it.

Exercise 52 (Interval scheduling: constraint matrix and TU). In an interval scheduling LP, each job j is active during time slots $S_j \subseteq \{1, \dots, T\}$ forming a contiguous interval. The LP relaxation is

$$\max \sum_j x_j \quad \text{s.t.} \quad \sum_{j : t \in S_j} x_j \le 1 \ \forall t, \quad 0 \le x_j \le 1.$$

The constraint matrix A has $A_{t,j} = 1$ if $t \in S_j$, 0 otherwise.

1. For $T = 3$ and jobs with intervals $[1,2]$, $[2,3]$, $[1,3]$, $[1,1]$, write $A \in \{0,1\}^{3\times 4}$.

2. Identify a row partition satisfying Ghouila-Houri, or find a submatrix with $|\det| = 2$ to show A is not TU.
3. Does the LP relaxation always have an integer optimum for interval scheduling? Justify.

Exercise 53 (TU under augmentation with slacks). Converting $Ax \leq b$ to standard form introduces slack variables: the equality-form matrix is $[A \mid I_m]$.

1. Prove that if A is TU, then $[A \mid I_m]$ is also TU. (*Hint:* Consider a square submatrix S of $[A \mid I_m]$. If S selects some slack columns, expand along those columns.)
2. Illustrate with the 2×2 TU matrix $A = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$.

Exercise 54 (TU and the LP integrality gap). The *integrality gap* of an LP relaxation is $\text{gap} = z^*_{\text{LP}}/z^*_{\text{ILP}}$ (for maximisation).

1. Prove that if the constraint matrix is TU and b is integer, the integrality gap equals 1.
2. Give a 1×1 example (knapsack with a single item) where A is not TU and the integrality gap is > 1.
3. Explain why minimising the integrality gap is one motivation for developing the theory of TUM.

Exercise 55 (Summary: TUM proof roadmap). Without looking at your notes, reconstruct the proof chain that shows: "If A is TU and $b \in \mathbb{Z}^m$, then every vertex of $P = \{x \geq 0 : Ax \leq b\}$ is integer."

1. State how a vertex x^* is characterised (as a basic feasible solution).
2. Write x^* using the inverse of a basis matrix $\hat{A}$.
3. Invoke Cramer's rule: write the formula for each component $(x^*)_j$.
4. Use TU to bound $|\det(\hat{A})|$ and $|\det(\hat{A}_j)|$.
5. Conclude $x^* \in \mathbb{Z}^n$ and state why this implies P is an integral polyhedron.

CHAPTER 8

Graph Algorithms

This chapter shifts gears: from the algebraic world of LP/ILP to the combinatorial world of graphs and efficient algorithms.

In chapter 7 we saw that many integer programmes with totally unimodular constraint matrices can be solved as linear programmes. That algebraic shortcut is elegant, but it is not the only route to efficient solutions. A large class of optimisation problems—shortest paths, minimum spanning trees, network flows, matching—possess a rich *graph structure* that specialised algorithms can exploit directly, often outperforming general-purpose LP solvers by orders of magnitude.

This chapter lays the groundwork. We introduce the language of graphs, study three fundamental data structures for representing them, and develop the two workhorse traversal algorithms—Breadth-First Search (BFS) and Depth-First Search (DFS). We then turn to Directed Acyclic Graphs (DAGs), show how to compute a topological ordering, use it to solve shortest-path problems on DAGs in linear time, and close with the transitive closure, which makes all reachability information explicit.

Road map.

1. Graph basics and representations (section 8.1).
2. Breadth-First Search (section 8.2).
3. Depth-First Search (section 8.3).
4. Directed Acyclic Graphs (section 8.4).
5. Topological sorting (section 8.5).
6. Shortest paths on DAGs (section 8.6).
7. Transitive closure (section 8.7).

Throughout, we illustrate every algorithm on the running directed graph from theorem 1.9.1, reproduced in fig. 8.1 for convenience.

8.1 Graph Basics and Representations

Combinatorial optimisation seeks an optimal subset of a finite ground set—paths, trees, matchings are all examples.

Before we can design algorithms on graphs, we need to agree on terminology and choose efficient data structures. The representation we pick has a direct impact on running time: an algorithm that is $O(n + m)$ with adjacency lists may degrade to $O(n^2)$ with an adjacency matrix on a sparse graph.

8.1.1 Basic Definitions

Definition 8.1.1 (Graph). A **directed graph** (or *digraph*) is a pair $G = (V, A)$

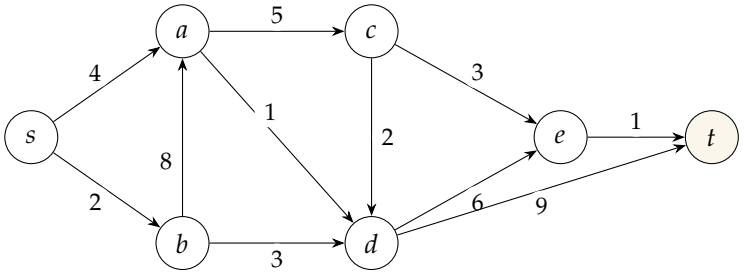


Figure 8.1: The running directed graph $\mathcal{G} = (\mathcal{V}, A)$ with $n = 7$ nodes and $m = 11$ arcs. Node t is highlighted as the sink. Arc weights are shown alongside each arc. This graph is a DAG (it contains no directed cycle), a fact we will verify formally in section 8.4.

where $\mathcal{V} = \{1, \dots, n\}$ is a finite set of **vertices** (or *nodes*) and $A \subseteq \mathcal{V} \times \mathcal{V}$ is a set of **arcs** (directed edges). We write $m = |A|$.

An **undirected graph** is a pair $\mathcal{G} = (\mathcal{V}, E)$ where E is a set of unordered pairs $\{u, v\}$ with $u, v \in \mathcal{V}$. We write $m = |E|$.

In plain language, a graph is a collection of dots (nodes) connected by lines (edges) or arrows (arcs). When we say "graph" without qualification, we mean a directed graph unless stated otherwise. A **weighted** graph associates a real number $w(e)$ to each edge or arc e; in our running graph (fig. 8.1), the weights represent costs or distances.

Definition 8.1.2 (Degree)**.** For an undirected graph, the **degree** of a vertex v, written $\deg(v)$ or $|\delta(v)|$, is the number of edges incident to v. For a directed graph we distinguish:

- **out-degree** $d^+(v) = |\delta^+(v)|$: number of arcs leaving v,
- **in-degree** $d^-(v) = |\delta^-(v)|$: number of arcs entering v.

The **neighbourhood** $\delta(v)$ is the set of edges/arcs incident to v; $\delta^+(v)$ and $\delta^-(v)$ are the forward and backward stars, respectively.

Example 8.1.3 (Degrees in the running graph)**.** In the running graph of fig. 8.1, node s has $d^+(s) = 2$ (arcs to a and b) and $d^-(s) = 0$ (no arc enters s). Node d has $d^+(d) = 2$ (arcs to e and t) and $d^-(d) = 3$ (arcs from a, b, and c). The out-degrees are $d^+(s) = 2$, $d^+(a) = 2$, $d^+(b) = 2$, $d^+(c) = 2$, $d^+(d) = 2$, $d^+(e) = 1$, $d^+(t) = 0$. The sum is $2+2+2+2+2+1+0 = 11 = m$, confirming the handshaking lemma for directed graphs: $\sum_v d^+(v) = \sum_v d^-(v) = m$.

Remark 8.1.4 (Sparse vs. dense). A graph is called **dense** when $m = \Theta(n^2)$ (a constant fraction of all possible arcs exist) and **sparse** when $m = O(n)$. Most real-world networks—road maps, social networks, dependency graphs—are sparse. The distinction matters because algorithmic complexity is expressed in terms of both n and m, and the choice of data structure should match the density.

Definition 8.1.5 (Walks, paths, and cycles)**.** Let $\mathcal{G} = (\mathcal{V}, A)$ be a directed graph.

- A **walk** of *length* k is a sequence of nodes $v_0, v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in A$ for all $0 \leq i < k$.
- A **trail** is a walk in which no arc is repeated.
- A **(simple) path** is a walk in which no node is repeated (which implies no arc is repeated).
- A **cycle** is a walk $v_0, v_1, \ldots, v_k = v_0$ with $k \geq 1$ in which no node other than the first/last is repeated.

For an undirected graph $\mathcal{G} = (\mathcal{V}, E)$, the definitions are analogous with each condition "$(v_i, v_{i+1}) \in A$" replaced by "$\{v_i, v_{i+1}\} \in E$".

Convention. Throughout this text, "path" always means a simple path unless explicitly stated otherwise.

Definition 8.1.6 (Connectivity).

- An undirected graph is **connected** if there exists a path between every pair of vertices.
- A directed graph is **strongly connected** if for every ordered pair of vertices (u, v) there exists a directed path from u to v and a directed path from v to u.
- A directed graph is **weakly connected** if its underlying undirected graph (obtained by ignoring arc directions) is connected.
- The **strongly connected components (SCCs)** of a directed graph are its maximal strongly connected subgraphs.

8.1.2 Adjacency Matrix

The most straightforward representation stores an $n \times n$ boolean matrix M where $M[i, j] = 1$ if arc $(i, j) \in A$ and $M[i, j] = 0$ otherwise.

For undirected graphs the matrix is symmetric: $M[i, j] = M[j, i]$.

- **Space:** $\Theta(n^2)$.
- **Edge test** "does (i, j) exist?": $O(1)$—a single array lookup.
- **Enumerate neighbours of** v: $\Theta(n)$—scan the entire row, regardless of $d^+(v)$.

The adjacency matrix is optimal when the graph is dense, because the $\Theta(n^2)$ memory is proportional to the actual graph size. On sparse graphs, however, most entries are zero, wasting both space and time when scanning neighbours.

8.1.3 Edge List

An edge list simply stores all m arcs as pairs (i, j) in an array of length m.

- **Space:** $O(m)$.
- **Edge test:** $O(m)$—linear scan.
- **Enumerate neighbours of** v: $O(m)$—must scan the entire list.

The edge list is memory-efficient but operationally slow. It is useful as an intermediate format (e.g. reading input) but rarely the primary structure for algorithm execution.

8.1.4 Adjacency List (List of Stars)

The **adjacency list** (or *list of stars*) stores, for each node v, a list of its forward neighbours $\delta^+(v)$. For directed graphs, we typically maintain both the **forward star** $\delta^+(v)$ (outgoing arcs) and the **backward star** $\delta^-(v)$ (incoming arcs).

"List of Stars" is the standard term in the Italian OR tradition.

- **Space:** $O(n + m)$.
- **Enumerate neighbours of** v: $O(d^+(v))$—we touch only the actual neighbours.
- **Edge test:** $O\big(\min\{d^+(i),\, d^-(j)\}\big)$—scan the shorter list.

For sparse graphs ($m = O(n)$), the adjacency list uses $O(n)$ space—dramatically better than the $\Theta\big(n^2\big)$ of the matrix. For this reason, the adjacency list is the preferred representation for virtually all graph algorithms we will study.

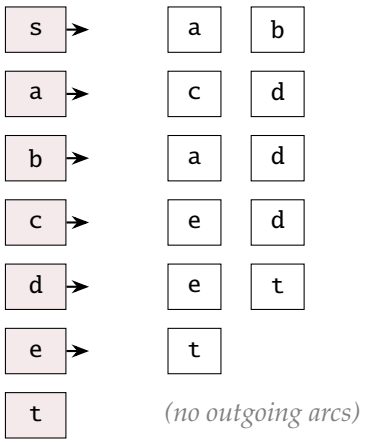


Figure 8.2: Adjacency list (forward star) representation of the running graph. Each row stores the out-neighbours of a node. For instance, node d points to the list $[e, t]$ because $\delta^+(d) = \{e, t\}$. The total storage is $n + m = 7 + 11 = 18$ entries.

8.1.5 Comparison of Representations

Table 8.1: Comparison of graph representations. Here $n = |\mathcal{V}|$, $m = |A|$, and $d = d^+(v)$ is the out-degree of the queried node.

Operation	Adjacency matrix	Edge list	Adjacency list
Space	$\Theta\big(n^2\big)$	$O(m)$	$O(n + m)$
Edge test (i, j)	$O(1)$	$O(m)$	$O(d)$
Enumerate $\delta^+(v)$	$\Theta(n)$	$O(m)$	$O(d)$
Insert edge	$O(1)$	$O(1)$	$O(1)$
Best when	dense	input I/O	sparse

> *Remark* 8.1.7 (Hybrid approaches). In practice, one may store the same graph in multiple formats—for example an adjacency matrix for $O(1)$ edge tests alongside an adjacency list for fast neighbour enumeration. This doubles the memory but gives the best of both worlds.

8.2 Breadth-First Search

BFS explores the graph "layer by layer," like ripples expanding from a stone dropped in water.

Breadth-First Search (BFS) is the simplest algorithm for systematically exploring a graph from a source node. It visits nodes in order of their distance (hop

count) from the source, making it the algorithm of choice for finding shortest paths in unweighted graphs.

8.2.1 The Three-Colour Scheme

During BFS (and later DFS), every node is in exactly one of three states:

1. **White** (undiscovered): the node has not yet been reached.
2. **Gray** (frontier): the node has been discovered and is waiting in the queue; its neighbours have not all been examined.
3. **Black** (finished): the node has been fully processed; all its neighbours have been examined.

Initially every node is white. When a white node is first encountered, it turns gray and enters the queue. When it is dequeued and all its neighbours have been inspected, it turns black and never changes colour again.

8.2.2 The BFS Algorithm

Formal details — BFS pseudocode

Given a directed graph $\mathcal{G} = (\mathcal{V}, A)$ stored as an adjacency list and a source node $s \in \mathcal{V}$, BFS computes for every node v the distance $\mathrm{dist}[v]$ (number of arcs on a shortest s–v path) and a parent pointer $\mathrm{par}[v]$ that encodes a shortest-path tree.

1. **Initialise:** for all $v \in \mathcal{V}$, set $\mathrm{colour}[v] \leftarrow \textsc{white}$, $\mathrm{dist}[v] \leftarrow \infty$, $\mathrm{par}[v] \leftarrow \textsc{nil}$.
2. Set $\mathrm{colour}[s] \leftarrow \textsc{gray}$, $\mathrm{dist}[s] \leftarrow 0$. Enqueue s into FIFO queue Q.
3. **While** $Q \neq \emptyset$:
   3.1. Dequeue $v \leftarrow Q.\mathrm{dequeue}()$.
   3.2. **For each** $w \in \delta^+(v)$:
      3.2.1. **If** $\mathrm{colour}[w] = \textsc{white}$:
         - $\mathrm{colour}[w] \leftarrow \textsc{gray}$,
         - $\mathrm{dist}[w] \leftarrow \mathrm{dist}[v] + 1$,
         - $\mathrm{par}[w] \leftarrow v$,
         - $Q.\mathrm{enqueue}(w)$.
   3.3. $\mathrm{colour}[v] \leftarrow \textsc{black}$.

Let us unpack this step by step. The queue Q is the *frontier*: it holds all gray nodes. At each iteration we take the node that has been waiting longest (FIFO discipline), inspect all its outgoing arcs, and for any white neighbour w we record its distance and add it to the frontier. Because the queue processes nodes in the order they were discovered, nodes at distance k are all handled before any node at distance $k + 1$. This "layer-by-layer" expansion is the defining feature of BFS.

8.2.3 BFS on the Running Graph

Example 8.2.1 (BFS from s)**.** We run BFS on the running graph (fig. 8.1) starting from s. The queue contents and distances evolve as follows:

Step	Dequeue	Queue after step	Discovery
0	—	$[s]$	s: dist = 0
1	s	$[a, b]$	a: dist = 1, b: dist = 1
2	a	$[b, c, d]$	c: dist = 2, d: dist = 2
3	b	$[c, d]$	(a and d already gray/black)
4	c	$[d, e]$	e: dist = 3 (d already discovered)
5	d	$[e, t]$	t: dist = 3 (e already discovered)
6	e	$[t]$	(t already discovered)
7	t	$[\,]$	—

The BFS tree has edges $s \to a$, $s \to b$, $a \to c$, $a \to d$, $c \to e$, $d \to t$. Nodes $\{a, b\}$ form layer 1 (distance 1 from s), $\{c, d\}$ form layer 2, and $\{e, t\}$ form layer 3.

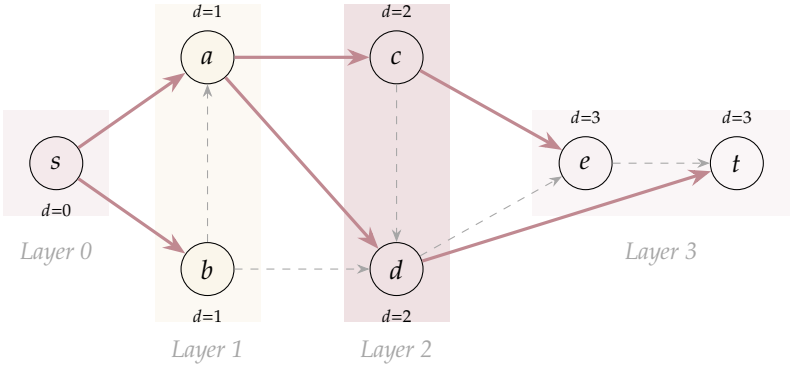


Figure 8.3: BFS tree (thick accent edges) rooted at s for the running graph. Shaded bands indicate BFS layers. Non-tree arcs are shown as thin dashed arrows. Every node is labelled with its BFS distance from s.

8.2.4 Properties of BFS

Theorem 8.2.2 (BFS shortest paths)**.** *After BFS from source s in a (possibly directed) graph with adjacency-list representation, for every vertex v reachable from s:*

1. dist[v] *equals the length (hop count) of a shortest s–v path.*
2. *The parent pointers encode a shortest-path tree rooted at s.*

The key insight is that the FIFO queue guarantees nodes are processed in non-decreasing order of distance. When we discover a white node w via a gray node v at distance k, the distance $k + 1$ assigned to w is optimal, because any shorter path would have caused w to be discovered earlier.

Formal details — Proof of theorem 8.2.2

Proof. The key idea is that BFS explores vertices in non-decreasing order of distance: the FIFO queue ensures all vertices at distance k are processed before any at distance $k + 1$. A simple induction on distance then shows each vertex receives the correct label.

Let $\delta(s, v)$ denote the true shortest-hop distance from s to v (and $\delta(s, v) = \infty$ if v is unreachable).

We prove by induction on $k = \delta(s, v)$ that $\text{dist}[v] = \delta(s, v)$ for every vertex v after BFS terminates.

Base case ($k = 0$). Only $v = s$ satisfies $\delta(s, s) = 0$, and the algorithm initialises $\text{dist}[s] = 0$. ✓

Inductive step. Assume every vertex at distance $< k$ receives the correct label. Let v be a vertex with $\delta(s, v) = k$. There exists a shortest path $s \rightsquigarrow u \to v$ where $\delta(s, u) = k - 1$. By the inductive hypothesis, $\text{dist}[u] = k - 1$. When u is dequeued, the algorithm examines each $w \in \delta^+(u)$; in particular it reaches v. If v is still white at that moment, BFS sets $\text{dist}[v] = k$. If v was already coloured gray, it was discovered via some other node u' with $\text{dist}[u'] \leq k - 1$, so $\text{dist}[v] \leq k$. Since no path of length $< k$ from s to v exists, we have $\text{dist}[v] = k = \delta(s, v)$. Unreachable vertices retain $\text{dist}[v] = \infty$. ✓ □

Formal details — BFS complexity

Proposition 8.2.3. *BFS runs in $O(n + m)$ time when the graph is stored as an adjacency list.*

Proof. Every node is enqueued and dequeued at most once: $O(n)$ total queue operations. For each dequeued node v, we scan $\delta^+(v)$, spending $O(d^+(v))$ time. Summing over all nodes: $\sum_v d^+(v) = m$. Therefore the total time is $O(n) + O(m) = O(n + m)$. □

BFS gives shortest paths only when all edges have unit weight. *For general weights we need Dijkstra (chapter 10).*

8.3 Depth-First Search

DFS explores "as deep as possible" before backtracking, like navigating a maze by always taking the first unexplored corridor.

Where BFS fans out layer by layer, Depth-First Search (DFS) plunges as deep as possible along a single path before backtracking. Conceptually, it replaces the FIFO queue with a LIFO stack—or, equivalently, uses recursion. DFS is the basis for many advanced graph algorithms: cycle detection, topological sorting, strongly connected components, and more.

8.3.1 Timestamps and the DFS Algorithm

DFS uses the same white/gray/black colouring scheme as BFS, but adds two **timestamps** to each node:

- $d[v]$: **discovery time**—the moment v turns gray.
- $f[v]$: **finish time**—the moment v turns black (all descendants fully explored).

A global clock, starting at 0, is incremented before each colour change.

Formal details — DFS pseudocode

DFS($\mathcal{G}$):
1. **Initialise:** for all $v \in \mathcal{V}$, set $\text{colour}[v] \leftarrow$ WHITE, $\text{par}[v] \leftarrow$ NIL. Set clock $\leftarrow 0$.
2. **For each** $v \in \mathcal{V}$ (in some fixed order):
   2.1. **If** $\text{colour}[v] =$ WHITE: call **DFS-Visit**(v).

DFS-Visit(v):
1. clock $\leftarrow$ clock $+ 1$; $d[v] \leftarrow$ clock; $\text{colour}[v] \leftarrow$ GRAY.
2. **For each** $w \in \delta^+(v)$:
   2.1. **If** $\text{colour}[w] =$ WHITE: set $\text{par}[w] \leftarrow v$ and call **DFS-Visit**(w).

3. clock ← clock + 1; $f[v] \leftarrow$ clock; colour$[v] \leftarrow$ BLACK.

The recursive structure means DFS naturally produces a **DFS forest**: a collection of rooted trees, one per connected component (or one per "restart" in a directed graph).

8.3.2 Edge Classification

As DFS traverses a graph, it doesn't just visit nodes; it also implicitly classifies every examined arc (v, w) into one of four categories based on the state of the destination node w at the moment the arc is explored. This classification reveals critical structural properties of the graph, such as the presence of cycles or reachability relations.

When DFS is at node v and examines the outgoing arc (v, w), the colour of w at that precise moment determines its type:

Definition 8.3.1 (DFS edge classification)**.** Let (v, w) be an arc examined during DFS. We classify it as:

- **Tree edge**: w is WHITE. The arc is traversed to discover w, becoming a parent-child link in the DFS forest.
- **Back edge**: w is GRAY. The node w is currently active (on the recursion stack) and is an ancestor of v in the DFS tree. A back edge indicates the presence of a cycle.
- **Forward edge**: w is BLACK and $d[v] < d[w]$. The node w is a descendant of v in the DFS tree that has already been finished. The arc (v, w) represents a shortcut pointing forward in the tree.
- **Cross edge**: w is BLACK and $d[v] > d[w]$. The node w is not an ancestor or descendant of v; it lies in a different subtree or was finished before v was discovered. The arc goes "across" branches.

Observation 8.3.2 (Dynamic vs. static classification). This classification is defined *dynamically on-the-fly* at the exact step when DFS explores the arc (v, w). At that specific moment:

- A WHITE target indicates a tree edge.
- A GRAY target indicates a back edge.
- A BLACK target indicates a forward or cross edge.

However, the same classification can be recovered *statically a posteriori* after the search is complete using only the final timestamps d and f:

- Tree edges form the parent-child structure of the DFS forest.
- Back edges satisfy $d[w] < d[v] < f[v] < f[w]$.
- Forward edges satisfy $d[v] < d[w] < f[w] < f[v]$ (and are not tree edges).
- Cross edges satisfy $d[w] < f[w] < d[v] < f[v]$.

Back edges are the key: a directed graph has a cycle if and only if *DFS discovers a back edge.*

In an undirected graph, only tree edges and back edges can occur; forward and cross edges are exclusive to directed graphs.

To build an intuitive understanding, we can think of the DFS forest as a family tree and the node colours as progress indicators:

Tree Edge (Parent to Child): You encounter a node that has never been seen before (WHITE). You immediately visit it, establishing a direct parental relationship.

Back Edge (Descendant to Ancestor): You encounter a node that is currently active (GRAY), meaning we are still exploring its descendants (including you). Connecting to it forms a loop in the lineage, proving the graph contains a cycle.

Forward Edge (Ancestor to Descendant): You encounter a node that is already finished (BLACK) and was discovered after you ($d[v] < d[w]$). It is a descendant reached through another path. This edge is a "shortcut" to the future.

Cross Edge (Cousin to Cousin): You encounter a node that is already finished (BLACK) but was discovered before you ($d[v] > d[w]$). Since it was already finished when you were discovered, it cannot be a descendant, and since it is finished, it cannot be an ancestor (which would still be gray on the stack). Thus, it must reside in an entirely different branch.

This classification is visualised schematically in fig. 8.4.

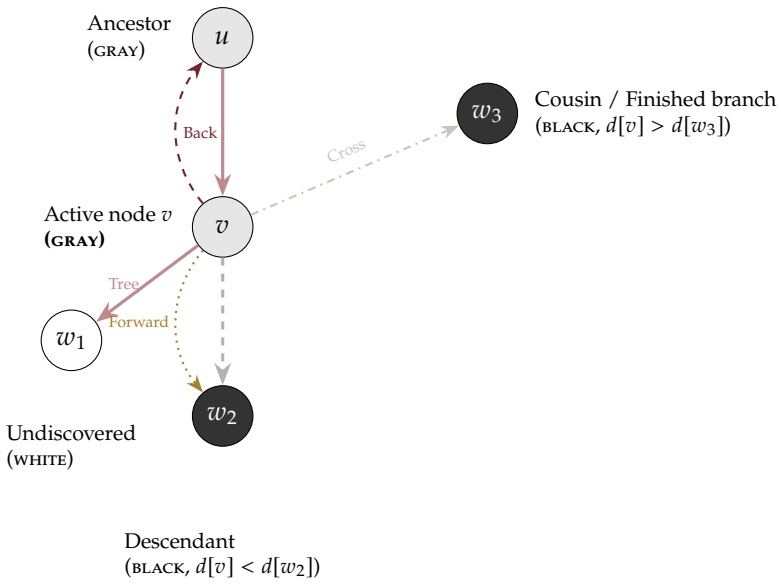


Figure 8.4: Schematic classification of an arc (v, w) examined from the active node v. Node colors represent the state of w at the moment (v, w) is evaluated.

Intermezzo — A brief history of graph search

BFS and DFS are among the oldest algorithms in computer science. Depth-first search traces back to the 19th-century work of Trémaux on maze-solving. The modern algorithmic formulation is due to Tarjan (1972), who also showed how DFS leads to linear-time algorithms for biconnected components and strongly connected components. BFS in its queue-based form was formalised by Moore (1959) in the context of shortest-path routing.

8.3.3 DFS on the Running Graph

Example 8.3.3 (DFS from s)**.** We run DFS on the running graph from s, exploring neighbours in alphabetical order when there is a choice. The

execution proceeds as follows:

We trace step by step, exploring neighbours in alphabetical order:

1. Call **DFS-Visit**(s): $d[s] = 1$, s turns gray. Neighbours of s: $\{a, b\}$. First neighbour a is white.
2. Call **DFS-Visit**(a): $d[a] = 2$, a turns gray. Neighbours of a: $\{c, d\}$. First neighbour c is white.
3. Call **DFS-Visit**(c): $d[c] = 3$, c turns gray. Neighbours of c: $\{e, d\}$. First neighbour e is white.
4. Call **DFS-Visit**(e): $d[e] = 4$, e turns gray. Neighbours of e: $\{t\}$. t is white.
5. Call **DFS-Visit**(t): $d[t] = 5$, t turns gray. Neighbours of t: $\emptyset$ (no outgoing arcs). $f[t] = 6$, t turns black.
6. Return to e: no more white neighbours. $f[e] = 7$, e turns black.
7. Return to c: next neighbour d is white.
8. Call **DFS-Visit**(d): $d[d] = 8$, d turns gray. Neighbours of d: $\{e, t\}$. e is black (cross edge $d \to e$: $d[d] = 8 > d[e] = 4$, so e is not a descendant of d); t is black (cross edge $d \to t$). $f[d] = 9$, d turns black.
9. Return to c: no more white neighbours. $f[c] = 10$, c turns black.
10. Return to a: next neighbour d is now black (forward edge $a \to d$: $d[a] = 2 < d[d] = 8$ and d is a descendant of a in the DFS tree). $f[a] = 11$, a turns black.
11. Return to s: next neighbour b is white.
12. Call **DFS-Visit**(b): $d[b] = 12$, b turns gray. Neighbours of b: $\{a, d\}$. a is black (cross edge $b \to a$); d is black (cross edge $b \to d$). $f[b] = 13$, b turns black.
13. Return to s: no more white neighbours. $f[s] = 14$, s turns black.

Summary of timestamps:

v	s	a	b	c	d	e	t
$d[v]$	1	2	12	3	8	4	5
$f[v]$	14	11	13	10	9	7	6

8.3.4 Properties of DFS

Theorem 8.3.4 (Parenthesis theorem)**.** *For any two vertices u and v in a DFS forest, exactly one of the following holds:*

1. *$[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint.*
2. *$[d[u], f[u]] \subset [d[v], f[v]]$ (so v is an ancestor of u).*
3. *$[d[v], f[v]] \subset [d[u], f[u]]$ (so u is an ancestor of v).*

In words, the discovery/finish intervals either nest (ancestor–descendant relationship) or are disjoint (no relationship in the DFS tree). This is why we call them "parentheses": they open and close like matched brackets.

Theorem 8.3.5 (White-path theorem — informal)**.** *Vertex v is a descendant of u in the DFS forest if and only if, at the instant just before u is discovered, there is a path from u to v whose vertices are all white.*

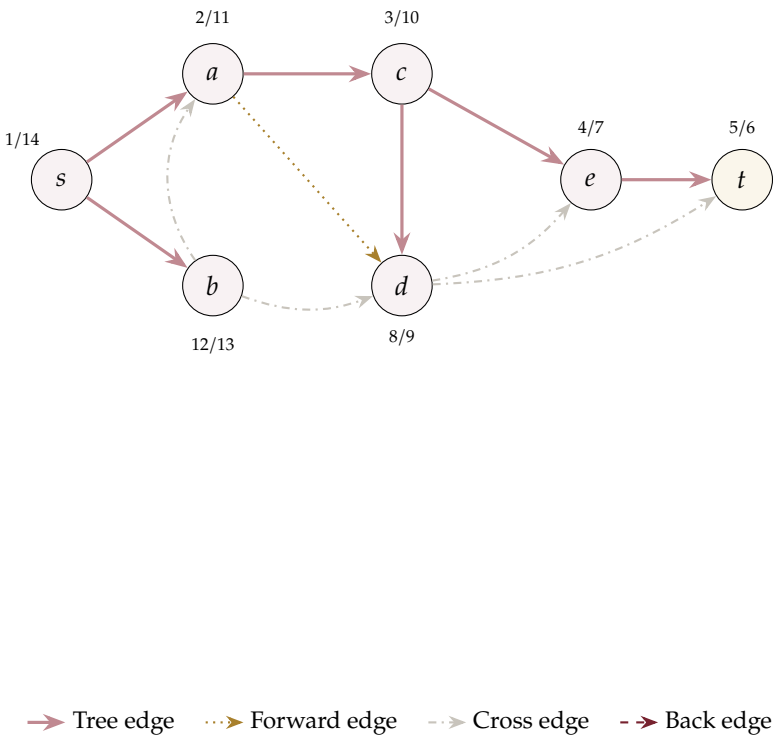


Figure 8.5: DFS tree for the running graph with timestamps $d[v]/f[v]$. Tree edges use the main accent; cross edges are dash-dotted warm lines ($d \to e$, $d \to t$, $b \to a$, $b \to d$); the forward edge $a \to d$ is dotted gold. No back edges appear, confirming that the graph is a DAG.

■ **Formal details — DFS complexity**

Proposition 8.3.6. *DFS runs in $O(n + m)$ time when the graph is stored as an adjacency list.*

Proof. The argument mirrors BFS: each node is discovered and finished exactly once ($O(n)$ total), and each arc is examined exactly once during the **for each** loop of the node that owns it, for a total of $O(m)$. Hence $O(n + m)$ overall. □

8.4 Directed Acyclic Graphs

Many optimisation problems have an inherent ordering: tasks must be completed before others can begin, courses have prerequisites, software modules

DAGs encode "something comes before something else" without circularity—the natural language of prerequisites and dependencies.

depend on libraries. The right abstraction for such structures is the *directed acyclic graph*.

Definition 8.4.1 (Directed Acyclic Graph)**.** A **directed acyclic graph (DAG)** is a directed graph $\mathcal{G} = (\mathcal{V}, A)$ that contains no directed cycle.

That is, there is no sequence of arcs $v_0 \to v_1 \to \cdots \to v_k = v_0$ with $k \geq 1$.

Example 8.4.2 (The running graph is a DAG)**.** The running graph of fig. 8.1 is a DAG: all arcs point "roughly from left to right," and there is no way to return to a previously visited node by following arc directions. Our DFS in theorem 8.3.3 confirmed this: no back edge was found.

The connection between DAGs and DFS is captured by the following characterisation:

Theorem 8.4.3 (DAG characterisation via DFS)**.** *A directed graph $\mathcal{G}$ is a DAG if and only if every DFS on $\mathcal{G}$ produces no back edges.*

Formal details — Proof of theorem 8.4.3

Proof. ($\Rightarrow$) Suppose $\mathcal{G}$ is a DAG and, for contradiction, DFS finds a back edge (v, w). Then w is an ancestor of v, meaning there is a tree path from w to v. Together with the back edge $v \to w$, this forms a directed cycle—contradiction.

($\Leftarrow$) Suppose $\mathcal{G}$ contains a directed cycle $v_0 \to v_1 \to \cdots \to v_k = v_0$. Let v_i be the first node of the cycle discovered by DFS. By the white-path theorem, all subsequent cycle nodes become descendants of v_i. In particular, v_{i-1} (the predecessor of v_i on the cycle) is a descendant of v_i, so the arc $v_{i-1} \to v_i$ is a back edge. □

8.4.1 Partial Orders and DAGs

A DAG naturally induces a **partial order** $\prec$ on its vertices: we write $u \prec v$ if there is a directed path from u to v. This relation is:

- **Antisymmetric**: if $u \prec v$ then $v \not\prec u$ (otherwise there would be a cycle).
- **Transitive**: if $u \prec v$ and $v \prec w$, then $u \prec w$ (concatenate the paths).

It is a *partial* order because some pairs of vertices may be incomparable: neither $u \prec v$ nor $v \prec u$. If the DAG contains a Hamiltonian path (visiting every vertex exactly once), then the order is *total*: every pair is comparable.

8.4.2 Sources and Sinks

Definition 8.4.4 (Source and sink)**.** A **source** is a vertex v with $d^-(v) = 0$ (no incoming arcs). A **sink** is a vertex v with $d^+(v) = 0$ (no outgoing arcs).

Proposition 8.4.5. *Every non-empty DAG has at least one source and at least one sink.*

Proof. Suppose for contradiction that every vertex has $d^+(v) \geq 1$. Start at any vertex v_0 and follow an outgoing arc to v_1, then from v_1 to v_2, and so on. After n steps we have visited $n+1$ vertices, but there are only n vertices in the graph. By the pigeonhole principle, some vertex was visited twice, forming a directed cycle—contradicting the assumption that the graph is a DAG. Hence at least one vertex must have $d^+(v) = 0$ (a sink). A symmetric argument shows at least one source exists. □

Example 8.4.6 (Sources and sinks in the running graph)**.** In our running graph, s is the unique source ($d^-(s) = 0$) and t is the unique sink ($d^+(t) = 0$).

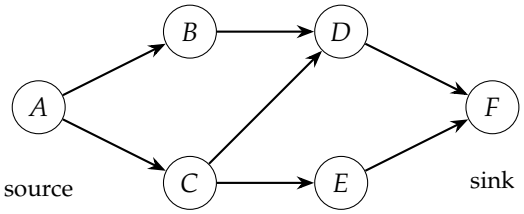


Figure 8.6: A small DAG with 6 nodes, representing task dependencies. Node A is the unique source; node F is the unique sink. This DAG will be used in section 8.5 to illustrate topological sorting.

8.5 Topological Sort

Topological sorting is one of the most useful operations on a DAG. It produces a linear ordering of the vertices that respects all directed arcs—a "flattening" of the partial order into a sequence.

Topological sort answers: "In what order should I process these tasks so that every prerequisite is met?"

Definition 8.5.1 (Topological sort)**.** A **topological sort** of a DAG $\mathcal{G} = (\mathcal{V}, A)$ is a bijective labelling $\ell : \mathcal{V} \to \{1, \ldots, n\}$ such that for every arc $(u, v) \in A$:

$$\ell(u) < \ell(v).$$

Equivalently, it is a permutation of the vertices such that every arc points from an earlier vertex to a later one.

The definition says: if there is an arc from u to v, then u must appear before v in the ordering. Topological sorts are not unique in general—a DAG may admit many valid orderings.

Theorem 8.5.2. *A directed graph admits a topological sort if and only if it is a DAG.*

Proof. ($\Rightarrow$) If a topological sort exists and there were a directed cycle $v_0 \to v_1 \to \cdots \to v_k = v_0$, then $\ell(v_0) < \ell(v_1) < \cdots < \ell(v_k) = \ell(v_0)$, which is a contradiction.

($\Leftarrow$) We give a constructive proof in the next subsection. □

8.5.1 Algorithm: DFS-Based Topological Sort

The simplest approach uses the finish times from DFS:

Formal details — DFS-based topological sort

1. Run DFS on the entire graph $\mathcal{G}$.
2. Output the vertices in **decreasing order of finish time** $f[v]$.

Correctness: For any arc $(u, v) \in A$, we need $f[u] > f[v]$. Since $\mathcal{G}$ is a DAG, there are no back edges (theorem 8.4.3). For a tree, forward, or cross edge (u, v), v finishes before u, so $f[u] > f[v]$.

Example 8.5.3 (DFS-based topological sort on the running graph)**.** From theorem 8.3.3, the finish times are:

$$f[s] = 14,\ f[b] = 13,\ f[a] = 11,\ f[c] = 10,\ f[d] = 9,\ f[e] = 7,\ f[t] = 6.$$

Sorting by decreasing f gives: s, b, a, c, d, e, t. Assigning labels $\ell(s) = 1,\ \ell(b) = 2,\ \ell(a) = 3,\ \ell(c) = 4,\ \ell(d) = 5,\ \ell(e) = 6,\ \ell(t) = 7$. We can verify that every arc goes from a lower to a higher label. For instance, $s \to a$: $1 < 3\checkmark$; $b \to a$: $2 < 3\checkmark$; $c \to d$: $4 < 5\checkmark$; $d \to t$: $5 < 7\checkmark$.

8.5.2 Algorithm: Kahn's Algorithm (Indegree-Based)

An alternative that avoids DFS entirely is **Kahn's algorithm** (forward labelling). The idea is simple: repeatedly find a source (a node with zero in-degree), label it next, and "remove" it by decrementing the in-degree of its successors.

Formal details — Kahn's algorithm

1. **Initialise:** compute $\deg[v] \leftarrow d^-(v)$ for all v. Set $\text{cnt} \leftarrow 0$. Push every node with $\deg[v] = 0$ onto a frontier S (stack or queue).
2. **While** $S \neq \emptyset$:
   2.1. Extract v from S.
   2.2. $\text{cnt} \leftarrow \text{cnt} + 1$; $\ell[v] \leftarrow \text{cnt}$.
   2.3. **For each** $w \in \delta^+(v)$:
      2.3.1. $\deg[w] \leftarrow \deg[w] - 1$.
      2.3.2. **If** $\deg[w] = 0$: push w onto S.
3. **If** $\text{cnt} < n$: report "graph contains a cycle."

The key insight is that we never physically remove nodes or arcs. Instead, the counter $\deg[w]$ tracks the *residual* in-degree: the number of predecessors of w that have not yet been labelled. When this counter reaches zero, w has no remaining dependencies and can safely be labelled next.

Example 8.5.4 (Kahn's algorithm on the small DAG)**.** We apply Kahn's algorithm to the DAG from fig. 8.6 with nodes $\{A, B, C, D, E, F\}$ and arcs $\{A{\to}B,\ A{\to}C,\ B{\to}D,\ C{\to}D,\ C{\to}E,\ D{\to}F,\ E{\to}F\}$.

Initial in-degrees: $\deg[A] = 0$, $\deg[B] = 1$, $\deg[C] = 1$, $\deg[D] = 2$, $\deg[E] = 1$, $\deg[F] = 2$.

Only A has deg = 0; push A onto S.

Step	Extract	Label	Deg updates	New sources
1	A	$\ell[A] = 1$	$\deg[B] : 1 \to 0,\ \deg[C] : 1 \to 0$	push B, C
2	B	$\ell[B] = 2$	$\deg[D] : 2 \to 1$	—
3	C	$\ell[C] = 3$	$\deg[D] : 1 \to 0,\ \deg[E] : 1 \to 0$	push D, E
4	D	$\ell[D] = 4$	$\deg[F] : 2 \to 1$	—
5	E	$\ell[E] = 5$	$\deg[F] : 1 \to 0$	push F
6	F	$\ell[F] = 6$	—	—

Result: A, B, C, D, E, F with cnt = $6 = n$. This is a valid topological sort: every arc points from a smaller label to a larger one. For instance, $C \to D$: $3 < 4$✓; $E \to F$: $5 < 6$✓.

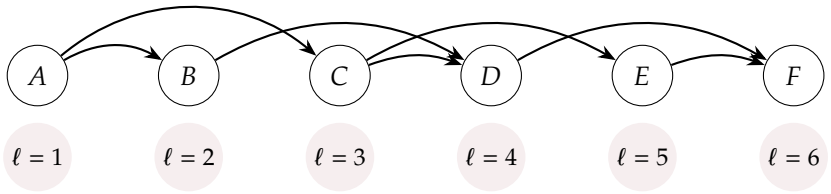


Figure 8.7: Topological sort of the DAG from fig. 8.6. The nodes are arranged left to right in topological order, and every arc points from left to right, confirming the validity of the ordering.

Formal details — Complexity of topological sort

Proposition 8.5.5. *Both the DFS-based and Kahn's algorithm compute a topological sort in $O(n + m)$ time.*

Proof. The DFS-based method runs DFS ($O(n + m)$) and then sorts by finish time ($O(n)$ with a counting sort or simply by outputting in reverse order).

Kahn's algorithm initialises in-degrees in $O(n + m)$ (scan all arcs once), and the main loop processes each node exactly once ($O(n)$ frontier operations) and each arc exactly once (when decrementing in-degrees), for a total of $O(n + m)$. □

8.6 Shortest Paths on DAGs

On a DAG, shortest paths are easier than on a general graph: no need for Dijkstra's priority queue. Negative weights are also handled effortlessly.

One of the most attractive properties of DAGs is that shortest-path problems can be solved in $O(n + m)$ time—faster than Dijkstra's algorithm ($O\big((n + m) \log n\big)$ with a binary heap) and, unlike Dijkstra, with no restriction on edge weights (negative weights are allowed).

The idea is simple: process the nodes in topological order and relax each outgoing arc.

8.6.1 The Algorithm

Formal details — Shortest paths on a DAG

Given a weighted DAG $\mathcal{G} = (\mathcal{V}, A, w)$ and a source s:

1. Compute a topological sort $v_1, v_2, \ldots, v_n$.
2. **Initialise:** $\text{dist}[s] \leftarrow 0$; $\text{dist}[v] \leftarrow +\infty$ for all $v \neq s$; $\text{par}[v] \leftarrow \text{NIL}$ for all v.
3. **For** $i = 1$ **to** n:
   3.1. Let $u \leftarrow v_i$.
   3.2. **For each** $(u, w) \in \delta^+(u)$:
      3.2.1. **If** $\text{dist}[u] + w(u, w) < \text{dist}[w]$:
         - $\text{dist}[w] \leftarrow \text{dist}[u] + w(u, w)$,
         - $\text{par}[w] \leftarrow u$.

Correctness: When we process node u in topological order, all predecessors of u have already been processed, so $\text{dist}[u]$ is already optimal. The relaxation of outgoing arcs then propagates this optimal value forward.

Complexity: Topological sort takes $O(n + m)$. The main loop processes each node once and each arc once, for another $O(n + m)$. Total: $O(n + m)$.

8.6.2 Worked Example

Example 8.6.1 (Shortest paths from s on the running graph)**.** We use the topological order s, b, a, c, d, e, t from theorem 8.5.3 (any valid topological order works).

Process	$\text{dist}[s]$	$\text{dist}[a]$	$\text{dist}[b]$	$\text{dist}[c]$	$\text{dist}[d]$	$\text{dist}[e]$	$\text{dist}[t]$
Init	0	∞	∞	∞	∞	∞	∞
s	0	4	2	∞	∞	∞	∞
b	0	4	2	∞	5	∞	∞
a	0	4	2	9	5	∞	∞
c	0	4	2	9	5	12	∞
d	0	4	2	9	5	11	14
e	0	4	2	9	5	11	12
t	0	4	2	9	5	11	12

Let us trace the key relaxations. When we process s, arcs $s \to a$ (weight 4) and $s \to b$ (weight 2) update $\text{dist}[a] = 4$ and $\text{dist}[b] = 2$. When we process b, arc $b \to a$ would set $\text{dist}[a] = 2 + 8 = 10 > 4$, so no update; arc $b \to d$ sets $\text{dist}[d] = 2 + 3 = 5$. When we process a, arc $a \to c$ sets $\text{dist}[c] = 4+5 = 9$; arc $a \to d$ would give $4+1 = 5 = \text{dist}[d]$, no improvement. Processing d updates $\text{dist}[e] = 5 + 6 = 11$ and $\text{dist}[t] = 5 + 9 = 14$. Finally, processing e relaxes $e \to t$: $11 + 1 = 12 < 14$, improving $\text{dist}[t]$ to 12.

The shortest s–t path is $s \to b \to d \to e \to t$ with cost $2 + 3 + 6 + 1 = 12$.

Remark 8.6.2 (Longest paths on DAGs). By negating all weights and running the same algorithm, we can find *longest* paths on a DAG in $O(n + m)$ time. This is the basis of the **critical path method (CPM)** in project scheduling. On general graphs, longest path is NP-hard; on DAGs it is solvable in linear time.

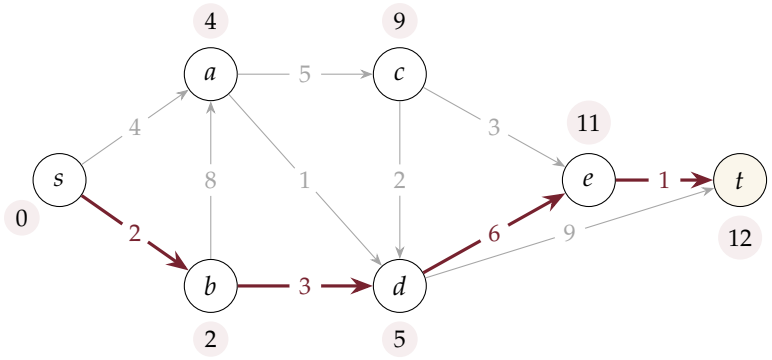


Figure 8.8: Shortest paths from s in the running DAG. The shortest-path tree is highlighted with the main accent. The optimal s–t path $s \to b \to d \to e \to t$ has total weight 12.

8.7 Transitive Closure

The transitive closure answers every reachability query in $O(1)$ time, at the cost of $O(n^2)$ space.

Given a directed graph $G = (V, A)$, we often want to know which vertices can reach which other vertices. The **transitive closure** packages all reachability information into a single structure.

Definition 8.7.1 (Transitive closure)**.** The **transitive closure** of a directed graph $G = (V, A)$ is the graph $G^* = (V, A^*)$ where

$$(u, v) \in A^* \iff \text{there exists a positive-length directed walk from } u \text{ to } v \text{ in } G.$$

Thus A^* is the transitive, generally non-reflexive, closure of the arc relation. A diagonal pair (v, v) belongs to A^* precisely when v lies on a directed cycle.

In practice, we represent the transitive closure as a boolean $n \times n$ matrix T where $T[i][j] = 1$ if and only if j is reachable from i.

8.7.1 Naïve Algorithm: BFS/DFS from Each Vertex

The most straightforward approach is to run a BFS or DFS from each vertex i, starting the search from the out-neighbours of i. Set $T[i][j] = 1$ whenever the search reaches j. In particular, $T[i][i]$ becomes true only if the search returns to i through a positive-length directed cycle.

- Each BFS/DFS takes $O(n + m)$.
- We run it n times.
- **Total:** $O(n(n + m))$.

For sparse graphs ($m = O(n)$), this gives $O(n^2)$—better than the cubic algorithm below. For dense graphs ($m = \Theta(n^2)$), it gives $O(n^3)$, which matches the cubic algorithm.

8.7.2 Warshall's Algorithm

Intermezzo — Warshall and Floyd

Stephen Warshall published his transitive closure algorithm in 1962. In the same year, Robert

Floyd independently described the closely related all-pairs shortest-path algorithm. Both are based on the same dynamic programming recurrence over intermediate nodes. The shortest-path version (Floyd–Warshall) replaces boolean OR/AND with min/addition.

Warshall's algorithm computes the transitive closure using a dynamic programming approach that considers increasingly larger sets of intermediate nodes.

■ **Formal details — Warshall's algorithm**

Input: Directed graph $\mathcal{G} = (\mathcal{V}, A)$ with $n = |\mathcal{V}|$.
Output: Boolean matrix T where $T[i][j]$ = true iff there is a directed path from i to j.

1. **Initialise:** for all $i, j \in \{1, \ldots, n\}$:

$$T[i][j] \leftarrow \begin{cases} \text{true} & \text{if } (i,j) \in A, \\ \text{false} & \text{otherwise.} \end{cases}$$

2. **For** $k = 1$ **to** n: **for** $i = 1$ **to** n: **for** $j = 1$ **to** n:

$$T[i][j] \leftarrow T[i][j] \ \vee \ \big(T[i][k] \ \wedge \ T[k][j]\big).$$

Recursive property: At the start of iteration k, $T[i][j]$ = true iff there is a path from i to j using only intermediate vertices from $\{1, \ldots, k-1\}$. After the update, the set of allowed intermediate vertices expands to $\{1, \ldots, k\}$. After all n iterations, $T[i][j]$ reflects unrestricted reachability.

Let us unpack the update rule. At step k, we ask: "Can i reach j through the intermediate node k?" This requires i to reach k *and* k to reach j, both via nodes $\{1, \ldots, k-1\}$. If either path already existed ($T[i][j]$ was already true), the entry remains true. The OR captures both possibilities.

Example 8.7.2 (Warshall's algorithm on a small graph)**.** Consider a directed graph on $\{1, 2, 3\}$ with arcs $\{1 \to 2,\ 2 \to 3\}$.

Initialisation:

$$T^{(0)} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

$k = 1$**:** No path through node 1 is new (row 1 can reach node 2, but no one can reach node 1 except node 1 itself, and $T[1][1] = 0$). Matrix unchanged.

$k = 2$**:** Check $T[i][2] \wedge T[2][j]$ for all i, j. $T[1][2] = 1$ and $T[2][3] = 1$, so $T[1][3] \leftarrow T[1][3] \vee (1 \wedge 1) = 1$.

$$T^{(2)} = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

$k = 3$**:** $T[2][3] = 1$ and $T[1][3] = 1$, but $T[3][j] = 0$ for all j, so no new paths through node 3. Final matrix unchanged.

The result confirms: node 1 can reach nodes 2 and 3; node 2 can reach node 3; node 3 cannot reach anyone.

8.7.3 Complexity Comparison

Table 8.2: Comparison of transitive closure algorithms.

Algorithm	Time	Best for
BFS/DFS from each vertex	$O(n(n+m))$	Sparse graphs
Warshall's algorithm	$\Theta(n^3)$	Dense graphs

Remark 8.7.3 (Cycle detection via transitive closure). If we initialise $T[i][i] =$ false for all i (rather than true), then after running Warshall's algorithm, $T[i][i] =$ true if and only if node i lies on a directed cycle. Thus the diagonal of the transitive closure matrix provides a cycle-detection mechanism. For a DAG, all diagonal entries remain false.

8.7.4 Strongly Connected Components

Two vertices u and v are **mutually reachable** if there exist directed paths $u \to \cdots \to v$ and $v \to \cdots \to u$. This is an equivalence relation, and its equivalence classes are called **strongly connected components (SCCs)**.

If we *contract* each SCC into a single "super-node," the resulting graph is always a DAG, called the **condensation** of $\mathcal{G}$. The condensation captures the "skeleton" of the directed graph's reachability structure: reachability between super-nodes follows the DAG structure, while within each super-node everything is mutually reachable.

SCCs and the condensation DAG are studied in detail in many algorithms courses. Here we mention them briefly to connect the concepts.

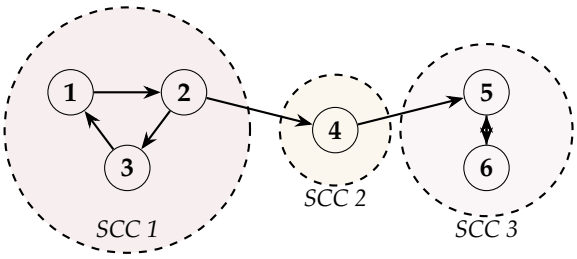


Figure 8.9: A directed graph with three strongly connected components (dashed boxes). Contracting each SCC to a single node yields a DAG: SCC 1 → SCC 2 → SCC 3.

8.8 Summary

This chapter introduced the foundational tools for working with graphs in operations research. Table 8.3 collects the main algorithms and their complexities.

Looking ahead. With these tools in hand, we are ready to tackle the classical optimisation problems on graphs:

- **Chapter 9**: Minimum Spanning Trees (Kruskal's and Prim's algorithms).

Table 8.3: Summary of graph algorithms from this chapter.

Problem	Algorithm	Complexity
Shortest paths (unweighted)	BFS	$O(n+m)$
Traversal / cycle detection	DFS	$O(n+m)$
Topological sort	DFS finish-time / Kahn's	$O(n+m)$
Shortest paths (DAG, weighted)	Topological relaxation	$O(n+m)$
Transitive closure (sparse)	$n\times$ BFS/DFS	$O(n(n+m))$
Transitive closure (dense)	Warshall	$\Theta(n^3)$

- **Chapter 10**: Shortest paths with general weights (Dijkstra, Bellman–Ford, Floyd–Warshall).
- **Chapter 11**: Network flows and the max-flow/min-cut theorem.

Each of these builds on the graph representations and traversal algorithms developed here. BFS reappears in augmenting-path algorithms for flows, DFS underlies strongly connected component decompositions, and topological sorting is used in CPM scheduling and in preprocessing steps for many network algorithms.

Summary & Key Takeaways

- **Graph Representation:** Adjacency Matrix ($O(V^2)$ space, $O(1)$ edge lookup) and Adjacency List ($O(V+E)$ space, $O(\deg(v))$ edge lookup).
- **BFS (Breadth-First Search):** Explores nodes layer-by-layer using a FIFO queue. Finds shortest path in unweighted graphs in $O(V+E)$ time.
- **DFS (Depth-First Search):** Explores as deep as possible before backtracking using a LIFO stack. Identifies connected components and topological sort in $O(V+E)$ time.

Exercises

Exercise 1. Let $G=(V,E)$ with $V=\{1,2,3,4,5\}$ and $E=\{(1,2),(1,3),(2,4),(3,4),(4,5)\}$ (undirected).

(a) Write the adjacency matrix of G.

(b) Write the adjacency list of G.

(c) State the degree of each vertex.

(d) Verify the handshaking lemma: $\sum_{v\in V}\deg(v)=2|E|$.

Exercise 2. Consider the directed graph $G=(V,A)$ with $V=\{a,b,c,d\}$ and $A=\{(a,b),(a,c),(b,d),(c,b),(c,d)\}$.

(a) Write the adjacency matrix.

(b) State the in-degree and out-degree of each vertex.

(c) Is G strongly connected? Justify.

Exercise 3. For a simple undirected graph G with n vertices, what is the maximum number of edges it can have? Express your answer as a formula in n and explain why it is an upper bound.

Exercise 4. A graph is called *sparse* when $m = O(n)$ and *dense* when $m = \Theta(n^2)$.

(a) For which class does the adjacency matrix representation use memory proportional to the number of edges rather than to n^2?

(b) For sparse graphs, compare the time to iterate over all neighbours of a vertex using an adjacency matrix versus an adjacency list.

(c) Give a concrete example of a sparse graph family and a dense graph family.

Exercise 5. True or false. For each statement give a short justification or counterexample.

(a) Every tree on n vertices has exactly $n - 1$ edges.

(b) A connected undirected graph has at least $n - 1$ edges.

(c) Every vertex in a k-regular graph has the same degree k.

(d) The sum of all out-degrees in a directed graph equals the number of arcs.

Exercise 6. Let G be the undirected graph with $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (4, 5)\}$. Perform BFS starting from vertex 1, breaking ties by visiting lower-numbered vertices first.

(a) List the order in which vertices are coloured grey (first discovered).

(b) Draw the BFS tree (describe it as a set of tree edges).

(c) State the BFS distance $d[v]$ for each vertex v.

(d) Identify all non-tree edges and classify them as cross edges.

Exercise 7. Consider the directed graph $G = (V, A)$ with $V = \{s, a, b, c, d, e\}$ and $A = \{(s, a), (s, b), (a, c), (b, c), (b, d), (c, e), (d, e)\}$. Perform BFS from source s (visit adjacency lists in alphabetical order).

(a) List the BFS queue contents at each step (show enqueue and dequeue operations).

(b) Give the BFS tree as a set of tree arcs.

(c) State the shortest-path distance from s to each vertex.

(d) Is the shortest s-e path unique? Find all shortest s-e paths.

Exercise 8. Explain, using the BFS layer structure, why BFS on an unweighted graph correctly computes shortest-path distances. Your argument should refer to the invariant maintained at each layer and show by induction that $d[v]$ equals the true shortest-path distance for every discovered vertex v.

Exercise 9. Let G be an undirected graph and let $s \in V$. Prove that every edge $(u, v) \in E$ satisfies $|d[u] - d[v]| \leq 1$, where d denotes BFS distances from s. What does this imply about the type of non-tree edges that can appear in a BFS of an undirected graph?

Exercise 10. A graph G is *bipartite* if and only if it contains no odd-length cycle. Describe an $O(n + m)$ algorithm based on BFS that decides whether a connected undirected graph is bipartite, and outline a correctness argument.

Exercise 11. Count the connected components of the undirected graph G with $V = \{1, 2, 3, 4, 5, 6, 7\}$ and $E = \{(1, 2), (1, 3), (2, 3), (4, 5), (6, 7)\}$ using BFS. List the vertices in each component and state how many components there are.

Exercise 12. Analyse the time complexity of BFS when the graph is represented by

(a) an adjacency list, and

(b) an adjacency matrix.

For which graph densities (in terms of m as a function of n) does the adjacency-matrix representation lead to the same asymptotic running time as the adjacency-list representation?

Exercise 13. Let G be the undirected graph with $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (4, 6)\}$. Perform DFS starting from vertex 1, visiting neighbours in increasing numerical order.

(a) Assign discovery time $d[v]$ and finish time $f[v]$ to each vertex (use the convention that the global clock starts at 1 and increments by 1 at each discovery and finish event).

(b) List the DFS tree edges.

(c) Classify every non-tree edge as a back edge, forward edge, or cross edge. (Recall: in an undirected DFS there are only tree edges and back edges.)

Exercise 14. Consider the directed graph $G = (V, A)$ with $V = \{1, 2, 3, 4, 5\}$ and $A = \{(1, 2), (1, 3), (2, 4), (3, 4), (4, 5), (3, 5), (5, 2)\}$. Perform DFS from vertex 1 (explore arcs in increasing order of head vertex).

(a) Record the discovery and finish times for every vertex.

(b) List tree arcs, back arcs, forward arcs, and cross arcs separately.

(c) Does the presence of a back arc imply anything about the graph? State the relevant theorem and verify it here.

Exercise 15. State the *white-path theorem* for DFS. Use it to show that vertex v is a descendant of vertex u in the DFS forest if and only if, at the instant just before u is discovered, there exists a path from u to v consisting entirely of white vertices.

Exercise 16. Prove or disprove: "In a DFS of an undirected graph there are no cross edges." Your argument should use discovery and finish times and the fact that edges are undirected.

Exercise 17. True or false. Justify each answer.

(a) A DFS tree of a connected undirected graph is a spanning tree.

(b) In a DFS of a directed graph, every back arc (u, v) satisfies $f[v] > f[u]$.

(c) In a DFS of a directed graph, every forward arc (u, v) satisfies $d[u] < d[v]$.

(d) Cross arcs can appear in a DFS of an undirected graph.

Exercise 18. Explain why DFS on a directed graph detects a cycle if and only if it encounters a back arc. Your answer should prove both directions of the equivalence.

Exercise 19. Determine which of the following directed graphs are DAGs. For each graph that is not a DAG, exhibit a directed cycle.

(a) $V = \{1, 2, 3, 4\}$, $A = \{(1, 2), (2, 3), (3, 4), (1, 4)\}$.

(b) $V = \{1, 2, 3, 4\}$, $A = \{(1, 2), (2, 3), (3, 1), (1, 4)\}$.

(c) $V = \{a, b, c, d, e\}$, $A = \{(a, b), (a, c), (b, d), (c, d), (d, e)\}$.

Exercise 20. A directed graph G has a topological ordering if and only if it is a DAG. Prove the forward direction: if G has a topological ordering then G is a DAG. (Hint: show that any cycle would contradict the ordering.)

Exercise 21. Prove the converse: if G is a DAG then it has at least one source (a vertex with in-degree 0). Use this fact to argue by induction on $|V|$ that every DAG has a topological ordering.

Exercise 22. Let G be a DAG. A vertex v is a *source* if in-deg$(v) = 0$ and a *sink* if out-deg$(v) = 0$.

(a) Must every DAG have at least one source? Prove or give a counterexample.

(b) Must every DAG have at least one sink? Prove or give a counterexample.

(c) Can a DAG have a vertex that is both a source and a sink? Under what conditions?

Exercise 23. Apply the **DFS-based topological sort** to the DAG with $V = \{a, b, c, d, e, f\}$ and $A = \{(a, b), (a, c), (b, d), (c, d), (c, e), (d, f), (e, f)\}$. Visit adjacency lists in alphabetical order.

(a) Record discovery and finish times for each vertex.

(b) List the vertices in decreasing order of finish time: this gives the topological order.

(c) Verify that every arc (u, v) satisfies $f[u] > f[v]$.

Exercise 24. Apply **Kahn's algorithm** (remove sources repeatedly) to the same DAG as the previous exercise: $V = \{a, b, c, d, e, f\}$, $A = \{(a, b), (a, c), (b, d), (c, d), (c, e), (d, f), (e, f)\}$.

(a) Initialise the in-degree of each vertex and the source queue.

(b) Simulate the algorithm step by step, showing which vertex is removed and how in-degrees are updated at each step.

(c) Write down the topological order produced.

(d) Is the output the same as that of the DFS-based algorithm? Must it be?

Exercise 25. Describe how Kahn's algorithm can be modified to detect whether a directed graph is a DAG. What condition on the algorithm's output signals the presence of a cycle? Argue correctness.

Exercise 26. Consider the DAG with $V = \{1, 2, 3, 4, 5, 6\}$ and $A = \{(1, 2), (1, 3), (2, 4), (3, 4), (3, 5), (4, 6), (5, 6)\}$.

(a) How many distinct topological orderings does this DAG have? List them all.

(b) Identify the unique vertex that must appear last in every topological ordering, and the set of vertices that can appear first.

Exercise 27. Prove that the DFS-based topological sort runs in $O(n + m)$ time when the graph is stored as an adjacency list. Identify which operations dominate the running time.

Exercise 28. Consider the weighted DAG with $V = \{s, a, b, c, t\}$ and arcs:

Arc	w
(s, a)	3
(s, b)	6
(a, b)	2
(a, c)	7
(b, c)	1
(b, t)	5
(c, t)	2

(a) Find a topological order of the vertices.

(b) Apply the DAG shortest-path algorithm (relax arcs in topological order) to compute $d[v]$ for all v, starting from s.

(c) Recover the shortest path from s to t by tracing predecessor pointers.

(d) What is the total weight of the shortest s-t path?

Exercise 29. Explain why the DAG shortest-path algorithm is correct even when arc weights are negative. Why does this not contradict the fact that Dijkstra's algorithm requires non-negative weights?

Exercise 30. Formulate the **longest path** problem on a DAG. Show how to reduce it to a shortest-path problem by negating weights, and apply your reduction to the DAG with $V = \{s, a, b, c, t\}$ and arcs:

Arc	w
(s, a)	2
(s, b)	5
(a, c)	4
(b, c)	1
(a, t)	3
(c, t)	6

Find the longest path from s to t and its length.

Exercise 31. Describe an $O(n + m)$ algorithm that counts the number of distinct shortest paths from a source s to every other vertex in a DAG with unit arc weights. Illustrate on the DAG: $V = \{s, a, b, c, d\}$, $A = \{(s, a), (s, b), (a, c), (b, c), (a, d), (c, d)\}$.

Exercise 32. Let $G = (V, A)$ with $V = \{1, 2, 3, 4\}$ and $A = \{(1, 2), (2, 3), (3, 4), (4, 2)\}$.

(a) Compute the transitive closure $G^* = (V, A^*)$ by running a BFS/DFS from each vertex and recording which vertices are reachable.

(b) Write the 4×4 reachability matrix R where $R_{ij} = 1$ iff j is reachable from i.

(c) What is the time complexity of this "n BFS" approach in terms of n and m?

Exercise 33. Apply **Warshall's algorithm** to the directed graph with $V = \{1, 2, 3, 4\}$ and $A = \{(1, 2), (2, 3), (3, 4), (4, 1)\}$.

(a) Write the initial reachability matrix $R^{(0)}$.

(b) Show the matrix $R^{(k)}$ after each iteration $k = 1, 2, 3, 4$, explaining which new entries become 1 and why.

(c) Compare the final matrix with the one you would obtain by multiplying the adjacency matrix by itself repeatedly.

Exercise 34. Apply **Warshall's algorithm** to the DAG with $V = \{1, 2, 3, 4\}$ and $A = \{(1, 2), (1, 3), (2, 4), (3, 4)\}$.

(a) Write $R^{(0)}$ and show the matrices $R^{(1)}, R^{(2)}, R^{(3)}, R^{(4)}$.

(b) Verify that the result matches the reachability computed by inspection.

(c) In this case, after which iteration k does the matrix stop changing?

Exercise 35. Compare the time complexities of computing the transitive closure by

(a) running BFS/DFS from each of the n vertices, and

(b) Warshall's algorithm.

For what density $m = \Theta(n^\alpha)$ do the two approaches have the same asymptotic cost? Find α.

Exercise 36. Warshall's algorithm can be viewed as a special case of the Floyd–Warshall all-pairs shortest-path algorithm. Describe the correspondence: what "distances" does Warshall's algorithm compute, and what is the "plus" and "min" of the tropical semiring replaced by in Warshall's setting?

Exercise 37. Let G be an undirected connected graph on n vertices and m edges.

(a) Give the tight asymptotic cost of BFS and DFS when G is stored as an adjacency list.

(b) Now suppose G is a tree. What is m in terms of n? Simplify the BFS/DFS cost.

(c) Suppose G is a complete graph K_n. What is m? Simplify the cost.

Exercise 38. A directed graph G is given by its adjacency matrix A (Boolean entries).

(a) Describe how to perform BFS using the matrix A. What is the time cost per layer?

(b) How does the matrix-based BFS compare to an adjacency-list BFS on a sparse graph?

(c) Describe an alternative $O(n^3)$ method to compute reachability using repeated matrix squaring over the Boolean semiring, and compare it to Warshall's.

Exercise 39. True or false. Justify each answer with a proof sketch or counterexample.

(a) A directed graph has a topological ordering if and only if it is a DAG.

(b) In BFS on an unweighted undirected graph, the BFS tree is always a minimum spanning tree.

(c) If DFS on a directed graph produces no back arcs, then the graph is a DAG.

(d) The transitive closure of a DAG is also a DAG.

(e) Every directed graph has a unique transitive closure.

Exercise 40. Let G be a directed graph and let T be a DFS tree rooted at vertex r.

(a) For a forward arc (u, v), how do the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ relate?

(b) For a cross arc (u, v), how do the same intervals relate?

(c) Use these relationships to explain why cross arcs in a DFS of an undirected graph would lead to a contradiction.

Exercise 41. Consider a directed graph $G = (V, A)$ with $V = \{1, 2, 3, 4, 5, 6\}$ and $A = \{(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 6), (6, 4)\}$.

(a) Is G a DAG? Identify all directed cycles.

(b) Identify the strongly connected components (SCCs) of G by inspection.

(c) How many sources and sinks does the condensation DAG (the DAG of SCCs) have?

Exercise 42. Describe an algorithm that, given an undirected graph G, outputs the number of connected components and assigns each vertex a component label. State its time complexity when the graph is stored as an adjacency list, and trace it on the graph with $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $E = \{(1, 2), (2, 3), (4, 5), (6, 7), (7, 8)\}$.

Exercise 43. The *diameter* of a connected undirected graph G is $\operatorname{diam}(G) = \max_{u,v \in V} d(u, v)$, where $d(u, v)$ is the shortest-path distance. Describe an algorithm that computes the exact diameter using BFS, state its time complexity, and discuss why a single BFS from an arbitrary vertex does not suffice to find the diameter in general.

Exercise 44. A directed graph G is given by $V = \{s, a, b, c, d, t\}$ and $A = \{(s, a), (s, c), (a, b), (b, t), (c, d), (d, b), (d, t)\}$ with unit weights.

(a) Is G a DAG? Give a topological order.

(b) Use the DAG shortest-path algorithm to find $d[v]$ for all v from source s.

(c) Is the shortest path from s to t unique?

Exercise 45. Let G be a DAG representing a project network, where each arc (u, v) has a duration $w(u, v) \geq 0$. The *earliest start time* of vertex v is the length of the longest path from the unique source s to v.

(a) Write a recurrence for the earliest start time $ES[v]$.

(b) Describe an $O(n + m)$ algorithm to compute $ES[v]$ for all v, using topological order.

(c) Apply your algorithm to the DAG with $V = \{s, a, b, c, t\}$ and arcs shown below. Find the critical path length.

Arc	w
(s, a)	3
(s, b)	2
(a, c)	4
(b, c)	5
(a, t)	9
(c, t)	2

Exercise 46. Describe how to modify DFS to check whether a given undirected graph contains a cycle, without computing discovery and finish times. State the key invariant maintained and argue correctness. What is the time complexity?

Exercise 47. Let $G = (V, E)$ be a connected undirected graph and let T be a BFS tree rooted at s.

(a) Prove that T is a spanning tree of G.

(b) Prove that for every non-tree edge $(u, v) \in E \setminus T$, the BFS layers of u and v differ by at most 1.

(c) Give an example showing that the BFS tree need not be a minimum spanning tree even for unit-weight graphs.

Exercise 48. Consider the weighted DAG with $V = \{1, 2, 3, 4, 5\}$ and arcs:

Arc	w
$(1, 2)$	4
$(1, 3)$	2
$(2, 4)$	3
$(3, 2)$	1
$(3, 4)$	8
$(4, 5)$	2
$(2, 5)$	6

(a) Find a topological ordering.

(b) Compute shortest distances from vertex 1 to all other vertices using topological relaxation. Show the relaxation steps.

(c) Find the shortest path from 1 to 5.

Exercise 49. Describe the relationship between DFS finish times and topological order: prove that if G is a DAG then listing vertices in decreasing order of DFS finish time yields a valid topological ordering.

Exercise 50. Let G be a directed graph with n vertices and m arcs. You want to decide, for every pair (i, j), whether there is a directed path from i to j.

(a) What data structure do you output? How many bits does it require?

(b) Give the complexity of computing it using n DFS runs versus Warshall's algorithm.

(c) When $m = O(n \log n)$, which method is faster asymptotically? Justify.

Exercise 51. A DAG is given by $V = \{1, 2, 3, 4, 5, 6, 7\}$ and $A = \{(1, 2), (1, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4, 7), (5, 7), (

(a) Apply Kahn's algorithm, choosing the smallest available vertex at each step. List the output order and the queue contents at each step.

(b) How many valid topological orderings does this DAG have? You do not need to list them all; provide an argument for the count.

(c) What is the length (number of arcs) of the longest path in this DAG? Find one such path.

Exercise 52. Prove that the following two statements are equivalent for a directed graph G:

(i) G is a DAG.

(ii) G has a topological ordering.

Your proof must address both directions. (You may use DFS or Kahn's algorithm as tools in your argument.)

Exercise 53. Explain why BFS and DFS both run in $O(n + m)$ time on adjacency-list graphs. In your explanation, identify exactly which operations are charged to vertices and which are charged to edges/arcs, and verify that each vertex and each edge is processed a constant number of times.

Exercise 54. A *bipartite graph* $G = (L \cup R, E)$ has its vertex set partitioned into two independent sets L and R.

(a) State the maximum number of edges a bipartite graph on n vertices can have (over all choices of the partition $L \cup R = V$).

(b) Given the graph G with $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 4), (1, 5), (2, 4), (2, 6), (3, 5), (3, 6)\}$, verify that it is bipartite by exhibiting a two-colouring produced by BFS from vertex 1.

(c) How does BFS detect that a graph is *not* bipartite during the colouring process? Give an example of a non-bipartite graph and show where BFS would discover the contradiction.

Exercise 55. Let G be a directed graph with $V = \{1, 2, 3, 4, 5\}$ and arcs $(1, 2), (2, 3), (3, 5), (1, 3), (3, 4), (4, 5)$ with weights $w(1, 2) = 2$, $w(2, 3) = 2$, $w(3, 5) = 1$, $w(1, 3) = 5$, $w(3, 4) = 3$, $w(4, 5) = 1$.

(a) Check that G is a DAG and give a topological ordering.

(b) Apply topological relaxation from source 1 to compute shortest distances $d[v]$ for all v. Show every relaxation step.

(c) List all shortest paths from 1 to 5 and state their weight.

(d) Find the *longest* path from 1 to 5 by negating all arc weights and re-applying the algorithm.

CHAPTER 9

Minimum Spanning Trees

MST is one of the few combinatorial optimisation problems where greedy algorithms provably find the global optimum.

In chapter 8 we learned how to explore graphs systematically using BFS and DFS, and how to compute shortest paths on DAGs. We now turn to a different kind of question: *how should we connect all the nodes of a network at minimum total cost?*

This is the **Minimum Spanning Tree** (MST) problem, a cornerstone of network design and one of the best-studied problems in combinatorial optimisation. Its practical importance is matched by its theoretical elegance: unlike most combinatorial problems, the MST can be solved optimally by simple greedy algorithms. The key insight that makes this possible—the *cut property*—is both intuitive and beautiful.

Road map.

1. Spanning trees: definitions and structural properties (section 9.1).
2. The greedy approach and the cut property (section 9.2).
3. Prim's algorithm (section 9.3).
4. Kruskal's algorithm (section 9.4).
5. Comparing Prim and Kruskal (section 9.5).
6. Variations: maximum spanning trees, negative weights, uniqueness (section 9.6).
7. MST and integer programming (section 9.7).
8. Summary and forward pointers (section 9.8).

Throughout, we illustrate every algorithm on the *undirected* version of the running graph from theorem 1.9.1, shown in fig. 9.1.

> *Remark* 9.0.1 (No source or sink is required). The labels s and t are inherited from the earlier directed running graph; here they are just ordinary vertex names. An MST instance needs only a connected undirected weighted graph. It has no distinguished source or sink. Prim's algorithm chooses an arbitrary starting vertex (a *root*), while Kruskal's algorithm does not even require a starting vertex.

9.1 Spanning Trees: Basics

9.1.1 Definitions

A spanning tree is the leanest possible connected subgraph—removing any edge disconnects it.

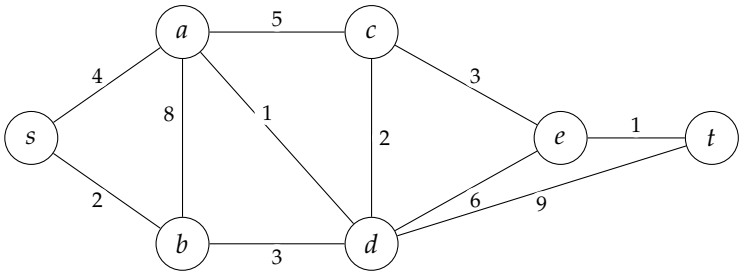


Figure 9.1: The undirected version of the running graph from theorem 1.9.1, with $n = 7$ nodes and $m = 11$ edges. The symbols s and t are ordinary vertices, not a source and a sink. Edge weights are identical to the arc weights of the directed version. We will compute the MST using both Prim's and Kruskal's algorithms.

Definition 9.1.1 (Spanning tree)**.** Let $G = (V, E)$ be a connected undirected graph with $n = |V|$ vertices and $m = |E|$ edges. A **spanning tree** of G is a subgraph $T = (V, E_T)$ such that:

1. T is **connected**: there is a path in T between every pair of vertices;
2. T is **acyclic**: T contains no cycle;
3. $E_T \subseteq E$ and $|E_T| = n - 1$.

In plain language, a spanning tree touches every node of G and uses the fewest possible edges to keep everything connected—exactly $n - 1$ of them. Any fewer and the graph becomes disconnected; any more and a cycle appears. Think of it as the "skeleton" of the network: every node is reachable, but there is no redundancy whatsoever.

Definition 9.1.2 (Minimum spanning tree)**.** Given a connected undirected graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$, a **minimum spanning tree** (MST) is a spanning tree T^* whose total weight

$$w(T^*) = \sum_{e \in E_{T^*}} w(e)$$

is minimum over all spanning trees of G.

The MST problem asks: *among the (possibly astronomically many) spanning trees of G, find one whose total edge weight is as small as possible.*

9.1.2 Why Spanning Trees Matter: Network Design

MST models the one-time construction *cost of a network, not ongoing usage costs.*

Imagine you need to connect seven cities with fibre-optic cable. Building a direct link between cities i and j costs $w(i, j)$ euros. You want every city reachable from every other, but you want to minimise the total construction cost. The answer is the MST of the graph whose nodes are cities and whose edge weights are construction costs.

This "network design" motivation appears in many guises: laying pipelines, building road networks, wiring circuit boards, or connecting servers in a data centre. In each case the MST gives the cheapest connected infrastructure.

Remark 9.1.3 (Robustness caveat). An MST has no redundancy: removing *any* single edge disconnects the network. In practice, engineers add extra links for fault tolerance. The MST is therefore a theoretical lower bound on construction cost, not always a practical design.

9.1.3 Key Properties of Trees

Before diving into algorithms, let us record three structural facts about spanning trees that will be used repeatedly.

Proposition 9.1.4 (Fundamental properties of spanning trees)**.** *Let T be a spanning tree of a connected graph $G = (V, E)$.*

1. ***Edge count.*** *T has exactly $n - 1$ edges.*
2. ***Unique cycle.*** *Adding any edge $e \in E \setminus E_T$ to T creates exactly one cycle, denoted $C(T, e)$.*
3. ***Unique cut.*** *Removing any edge $f \in E_T$ from T disconnects it into exactly two components, defining a cut $(S, V \setminus S)$. We denote this cut $D(T, f)$.*

Property (2) is intuitive: since T already connects every pair of vertices, adding a new edge creates a second path between its endpoints, closing a loop. Property (3) follows because a tree has no alternate route: if we break the unique path between two nodes by removing an edge, those nodes land in different components.

Example 9.1.5 (Spanning trees of the running graph)**.** The undirected running graph (fig. 9.1) has $n = 7$ nodes and $m = 11$ edges, so any spanning tree must use exactly 6 edges. Here are two spanning trees with different total weights:

- $T_1 = \{s\text{-}a, a\text{-}d, d\text{-}b, d\text{-}c, c\text{-}e, e\text{-}t\}$ with weight $4 + 1 + 3 + 2 + 3 + 1 = 14$.
- $T_2 = \{s\text{-}b, b\text{-}d, d\text{-}a, d\text{-}c, c\text{-}e, e\text{-}t\}$ with weight $2 + 3 + 1 + 2 + 3 + 1 = 12$.

Let us verify property (2) on T_2. Adding edge s–a (weight 4) to T_2 creates the cycle s–a–d–b–s. Adding edge d–e (weight 6) creates the cycle d–c–e–d (going via c), so the cycle is d–e–c–d. In T_2 the path from d to e goes d–c–e, so adding d–e creates exactly the cycle d–e–c–d.

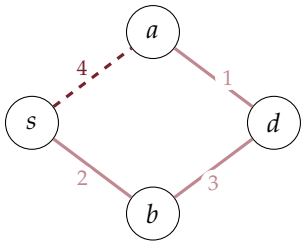
For property (3), removing edge b–d from T_2 disconnects the tree into $\{s, b\}$ and $\{a, d, c, e, t\}$. The cut edges crossing this partition are s–a (4), b–a (8), and b–d (3)—exactly the edges that could "repair" the cut.

Definition 9.1.6 (Cut)**.** A **cut** in a graph $G = (V, E)$ is a partition of V into two non-empty sets $(S, V \setminus S)$. The **cut edges** are $\delta(S) = \{\{u, v\} \in E : u \in S,\ v \in V \setminus S\}$.

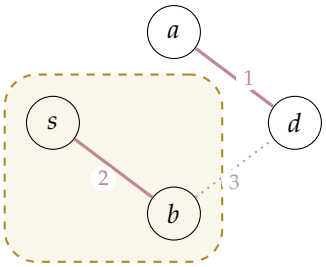
Intermezzo — Cycle–cut duality

A beautiful structural fact underpins spanning tree theory: *every cycle and every cut in a graph share an even number of edges.* This is sometimes called the *cycle–cut intersection property*.

The intuition is simple: imagine walking around a cycle. Every time you cross from S to $V \setminus S$



Adding the dashed edge s–a creates the cycle s–a–d–b–s.



Removing b–d (dotted) creates the cut $\{s, b\}$ vs. $\{a, d\}$.

Figure 9.2: Left: adding one dashed edge to a spanning tree creates exactly one cycle. Right: removing an edge creates exactly one cut, indicated by the dashed region.

(entering the cut), you must eventually cross back (leaving the cut) to complete the loop. So the crossings come in pairs, and the count is even.

This parity property explains why adding an edge to a tree and removing a different edge from the resulting cycle can yield another spanning tree. It is the structural engine behind the exchange arguments used in MST proofs.

9.2 The Greedy Approach and the Cut Property

Most combinatorial optimisation problems cannot be solved by greedy algorithms making locally optimal choices typically leads to globally suboptimal solutions. The MST problem is a remarkable exception. The reason greedy works here is captured by a single theorem: the *cut property*.

~~The~~ cut property is the theoretical backbone of all *classical MST algorithms.*

9.2.1 Extendable Sets and Safe Edges

To state the result precisely, we need two definitions.

Definition 9.2.1 (Extendable set)**.** A set of edges $A \subseteq E$ is **extendable** if there exists some MST T^* with $A \subseteq E_{T^*}$. In other words, the edges we have chosen so far can be extended into a full optimal solution.

The empty set is trivially extendable (it is a subset of every MST). A necessary condition for extendability is acyclicity: if A contains a cycle, it cannot be part of any tree, let alone an MST.

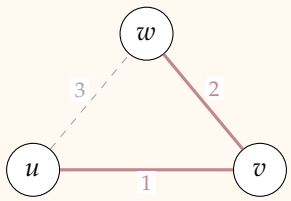
Definition 9.2.2 (Safe edge)**.** Given an extendable set A, an edge $e \notin A$ is

safe for A if $A \cup \{e\}$ is also extendable.

To build an intuitive model of these concepts, consider a puzzle-building analogy:

- **Extendability** means you haven't made any mistakes yet. The edges in A are a subset of a final optimal tree. You are on the right track, and it is still possible to reach the global optimum without dismantling any edges you've already placed.
- **Safety** is a property of the next edge you select. It is a "no-regrets" choice. By adding a safe edge, you are guaranteed that you haven't painted yourself into a corner; the expanded set $A \cup \{e\}$ still has a valid path to an optimal tree.

Example 9.2.3 (Extendable sets and safe edges)**.** Consider the triangle graph shown below with vertices $\{u, v, w\}$ and edge weights:



The unique MST for this graph has edges $T^* = \{\{u, v\}, \{v, w\}\}$ (solid lines) with a total cost of $1 + 2 = 3$. The edge $\{u, w\}$ (dashed line) is not in the MST. Let us evaluate some subsets of edges:

- $A_1 = \emptyset$ is extendable because $\emptyset \subseteq T^*$. For this set, the edge $e = \{u, v\}$ is *safe* because $A_1 \cup \{e\} = \{\{u, v\}\} \subseteq T^*$.
- $A_2 = \{\{u, w\}\}$ is *not* extendable. Although it is acyclic, no MST can contain $\{u, w\}$ because its weight is too high; any spanning tree containing it would be suboptimal (for instance, the tree with $\{u, v\}$ and $\{u, w\}$ has cost $4 > 3$). Therefore, $\{u, w\}$ was not a safe edge for A_1.
- If we start from the extendable set $A_3 = \{\{u, v\}\}$, the edge $e' = \{v, w\}$ is *safe* because $A_3 \cup \{e'\} = T^*$ is the MST itself. In contrast, the edge $e'' = \{u, w\}$ is *not* safe for A_3: adding it would form the set {{u, v}, {u, w}}, which is not a subset of any MST.

The generic MST algorithm is now almost trivial to state:

1. Start with $A = \emptyset$.
2. Repeat $n - 1$ times: find a safe edge e for A and set $A \leftarrow A \cup \{e\}$.
3. Return A.

The entire challenge lies in step 2: how do we efficiently identify safe edges? The cut property gives the answer.

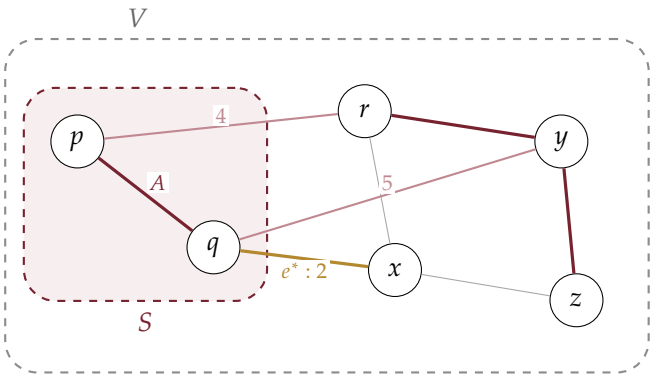
9.2.2 The Cut Property

Theorem 9.2.4 (Cut property)**.** *Let A be an extendable set, and let S be the vertex set of any connected component of the forest (V, A). If e^* is an edge of minimum weight in the cut $\delta(S)$, then e^* is safe for A.*

Before we prove this, let us unpack it. The forest (V, A) consists of the edges we have selected so far—it may have several connected components.

Pick any one component; its vertices form a set S. Look at all edges that cross the boundary between S and the rest of the graph. The cut property says: *the cheapest such edge is guaranteed to belong to some MST.* We can add it without fear.

This is a remarkably powerful statement. It works for *any* component and *any* extendable set, giving algorithms a great deal of freedom in choosing which cut to exploit at each step.



V: all vertices of the graph $G = (V, E)$.

$A \subseteq E$: the thick selected edges; (V, A) is the current forest.

$S \subset V$: the vertices of one connected component of (V, A).

$\delta(S)$: *all* edges with exactly one endpoint in S (here, the three edges leaving the shaded region).

$e^* \in \arg\min_{e \in \delta(S)} w(e)$: a lightest cut edge, highlighted in gold.

Figure 9.3: Notation in the cut property. The sets V, A, and S play different roles: V is the fixed vertex set, A is the selected edge set, and S is the vertex set of one component of the forest (V, A). The symbol $\delta(S)$ denotes a set of edges, not a vertex set.

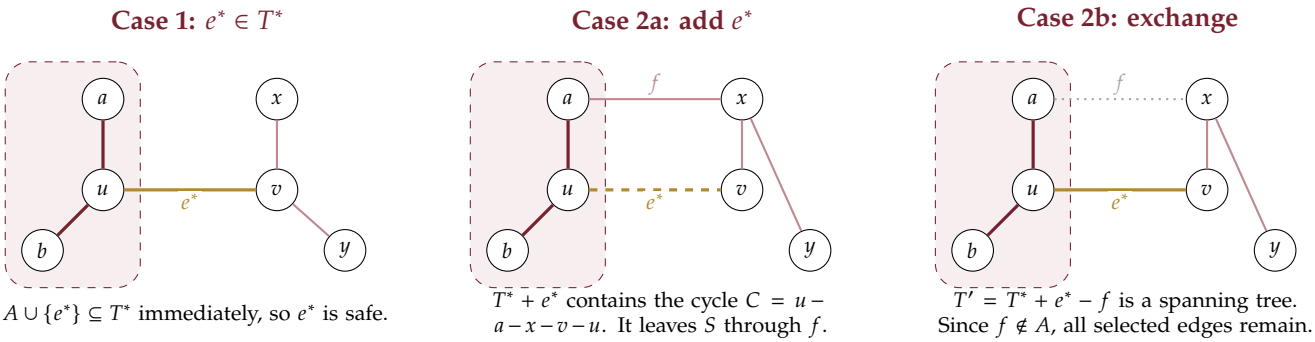


Figure 9.4: The two cases in the exchange proof. Thick burgundy edges belong to A; gold marks the candidate e^*; the dotted edge f is removed. The diagram deliberately shows that an edge of A may lie on the cycle—the essential fact is only $f \notin A$.

Formal details — Proof of the cut property

Proof. Let $e^* = \{u, v\}$ with $u \in S$ and $v \notin S$, and suppose e^* is a minimum-weight edge in $\delta(S)$. Since A is extendable, there exists an MST T^* with $A \subseteq E_{T^*}$.

Case 1: $e^* \in E_{T^*}$ (left panel of fig. 9.4). Then $A \cup \{e^*\} \subseteq E_{T^*}$, so e^* is safe. Done.

Case 2: $e^* \notin E_{T^*}$. Adding e^* to T^* creates a unique cycle C (middle panel of fig. 9.4, and theorem 9.1.4). The cycle contains e^*, which crosses the cut $\delta(S)$. A closed cycle that leaves S must later re-enter S; equivalently, a cycle and a cut intersect in an even number of edges. Therefore C contains another edge $f \neq e^*$ with $f \in E_{T^*} \cap \delta(S)$.

Now consider $T' = T^* \cup \{e^*\} \setminus \{f\}$. This is still a spanning tree (right panel of fig. 9.4): removing one edge of the unique cycle restores acyclicity without destroying connectivity. Its

weight is

$$w(T') = w(T^*) + w(e^*) - w(f) \;\leq\; w(T^*),$$

since $w(e^*) \leq w(f)$ (because e^* is the minimum-weight cut edge and f also crosses the cut). Since T^* was an MST, we have $w(T') = w(T^*)$, so T' is also an MST.

Finally, because S is a connected component of (V, A), no edge of A crosses $\delta(S)$. But f does cross $\delta(S)$, so $f \notin A$. The exchange therefore removes none of the previously selected edges:

$$A \cup \{e^*\} \subseteq E_{T'}.$$

Since T' is an MST, this proves that e^* is safe. □

Corollary 9.2.5. *Any algorithm that repeatedly adds a minimum-weight cut edge (for any cut respecting the current forest) produces a minimum spanning tree.*

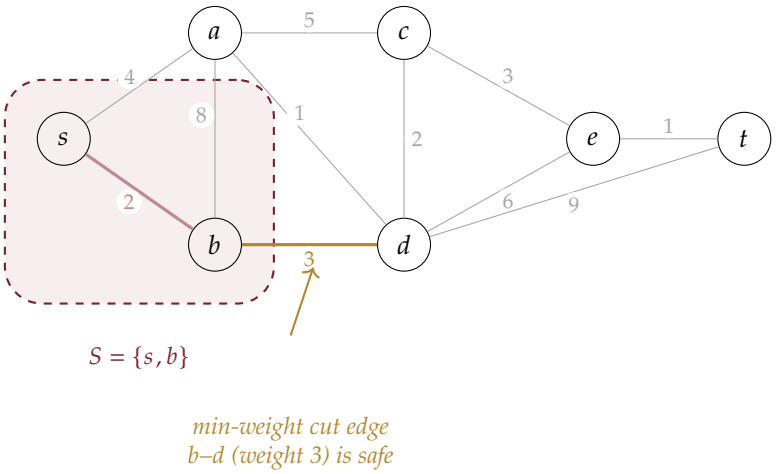


Figure 9.5: Illustrating the cut property. The component $S = \{s, b\}$ (dashed region) is already connected by the highlighted edge s–b. Three edges cross the cut: s–a (4), b–a (8), and b–d (3). The minimum-weight cut edge b–d (weight 3) is highlighted as safe.

9.3 Prim's Algorithm

Prim's algorithm grows one tree *from a root, always adding the cheapest edge to a non-tree vertex.*

Prim's algorithm is a direct application of the cut property. It maintains a single growing tree $\mathcal{T}$, starting from an arbitrary root vertex r. At each step, the cut is always $(\mathcal{T}, V \setminus \mathcal{T})$—the set of vertices already in the tree versus everything else. The algorithm picks the cheapest edge crossing this cut, adds the new vertex to $\mathcal{T}$, and repeats.

9.3.1 Algorithm Description

The algorithm uses three arrays:

- `inTree`$[v]$: boolean flag indicating whether $v \in \mathcal{T}$.

- `key`$[v]$: for $v \notin \mathcal{T}$, the weight of the cheapest edge connecting v to some vertex in $\mathcal{T}$. Initially $+\infty$.
- `parent`$[v]$: the vertex in $\mathcal{T}$ that achieves `key`$[v]$.

Formal details — Prim's algorithm — pseudocode

Algorithm 4: Prim's MST Algorithm

Input : Connected undirected graph $G = (V, E)$ with weights w; root r
Output: Set of MST edges E_T

```
1  foreach v ∈ V do
2      key[v] ← +∞
3      inTree[v] ← false
4      parent[v] ← nil
5  key[r] ← 0
6  parent[r] ← r
7  for i = 1 to n do
8      u ← arg min_{v: inTree[v]=false} key[v]
9      inTree[u] ← true
10     foreach edge {u, v} ∈ E with inTree[v] = false do
11         if w(u, v) < key[v] then
12             key[v] ← w(u, v)
13             parent[v] ← u
14 return E_T = {{parent[v], v} : v ∈ V \ {r}}
```

9.3.2 Correctness

Correctness follows immediately from the cut property (theorem 9.2.4). At each iteration, the set $\mathcal{T}$ is a connected component of the partial MST. The algorithm selects the minimum-weight edge in the cut $\delta(\mathcal{T})$, which is safe by the cut property. After $n-1$ such additions, $\mathcal{T}$ spans all of V.

Intuitively: at every step, Prim's algorithm grows its tree from the inside out. The set V_A of already-covered nodes defines a natural cut, and the minimum-weight edge crossing it is always safe to add.

More explicitly, the **loop invariant** is: at the start of each iteration, A (the set of edges selected so far) is a subtree of some MST, and V_A is the set of vertices it spans. The cut considered is $(V_A,\ V \setminus V_A)$: one side contains all tree vertices, the other contains all remaining vertices. The edge extracted in line 8 is precisely the minimum-weight edge crossing this cut, so the cut property guarantees it is safe to add. After $n-1$ additions the invariant gives a complete spanning tree that is optimal.

9.3.3 Complexity

The running time depends on how we implement the "extract minimum" operation in line 8 of algorithm 4.

- **Array scan (naïve):** Scanning all n entries of `key` to find the minimum takes $O(n)$ per iteration, for $\Theta(n^2)$ overall. Updating keys costs $O(\deg(u))$ per vertex, summing to $O(m)$ over all iterations. Total: $\Theta(n^2)$.

The $\Theta(n^2)$ version is optimal for dense graphs: just reading the adjacency matrix already costs $\Theta(n^2)$.

- **Binary heap:** A min-heap supports extract-min in $O(\log n)$ and decrease-key in $O(\log n)$. We perform n extract-min operations and at most m decrease-key operations, giving $O((n+m)\log n) = O(m \log n)$.
- **Fibonacci heap:** Decrease-key is amortised $O(1)$, so the total becomes $O(m + n \log n)$—the best known for Prim's algorithm.

Fibonacci heaps are rarely used in practice because the constant factors are large. Binary heaps are the usual choice.

9.3.4 Walkthrough on the Running Graph

We run Prim's algorithm on the undirected running graph (fig. 9.1) starting from root $r = s$. Recall the edges:

s-a:4, s-b:2, a-c:5, a-d:1, a-b:8, b-d:3, c-e:3, c-d:2, d-e:6, d-t:9, e-t:1.

Example 9.3.1 (Prim's algorithm step by step)**.** We track $\mathcal{T}$, the key values, and the parent pointers.

Step	Add vertex (edge)	Updated keys	$\mathcal{T}$
0	s (root, key = 0)	a:4, b:2	$\{s\}$
1	b via s–b (key = 2)	d:3, a: $\min(4,8) = 4$	$\{s, b\}$
2	d via b–d (key = 3)	a: $\min(4,1) = 1$, c:2, e:6, t:9	$\{s, b, d\}$
3	a via d–a (key = 1)	c: $\min(2,5) = 2$, e:6	$\{s, b, d, a\}$
4	c via d–c (key = 2)	e: $\min(6,3) = 3$	$\{s, b, d, a, c\}$
5	e via c–e (key = 3)	t: $\min(9,1) = 1$	$\{s, b, d, a, c, e\}$
6	t via e–t (key = 1)	—	$\{s, b, d, a, c, e, t\}$

MST edges: $\{s\text{-}b, b\text{-}d, d\text{-}a, d\text{-}c, c\text{-}e, e\text{-}t\}$ with weights $\{2, 3, 1, 2, 3, 1\}$. Total weight: $2 + 3 + 1 + 2 + 3 + 1 = 12$.

Let us narrate the key moments.

Step 0. We start with $\mathcal{T} = \{s\}$. Node s has two neighbours: a (weight 4) and b (weight 2). We set $\texttt{key}[a] = 4$ and $\texttt{key}[b] = 2$.

Step 1. The minimum key among non-tree nodes is $\texttt{key}[b] = 2$, so we add b. We scan b's neighbours: a has edge weight 8 (no improvement over the current key 4) and d has edge weight 3 (new, sets $\texttt{key}[d] = 3$).

Step 2. The minimum key is $\texttt{key}[d] = 3$. Adding d triggers a cascade of updates: d is adjacent to a (weight 1, improving $\texttt{key}[a]$ from 4 to 1), c (weight 2, new), e (weight 6, new), and t (weight 9, new).

Step 3. Now $\texttt{key}[a] = 1$ is the smallest. Adding a reveals edges to c (weight 5, no improvement over 2) and b (already in tree). No keys change.

Steps 4–6. The remaining vertices join in order c, e, t, each time via the cheapest available connection to the growing tree. The crucial update in step 5 is that adding e drops $\texttt{key}[t]$ from 9 (via d) to 1 (via e).

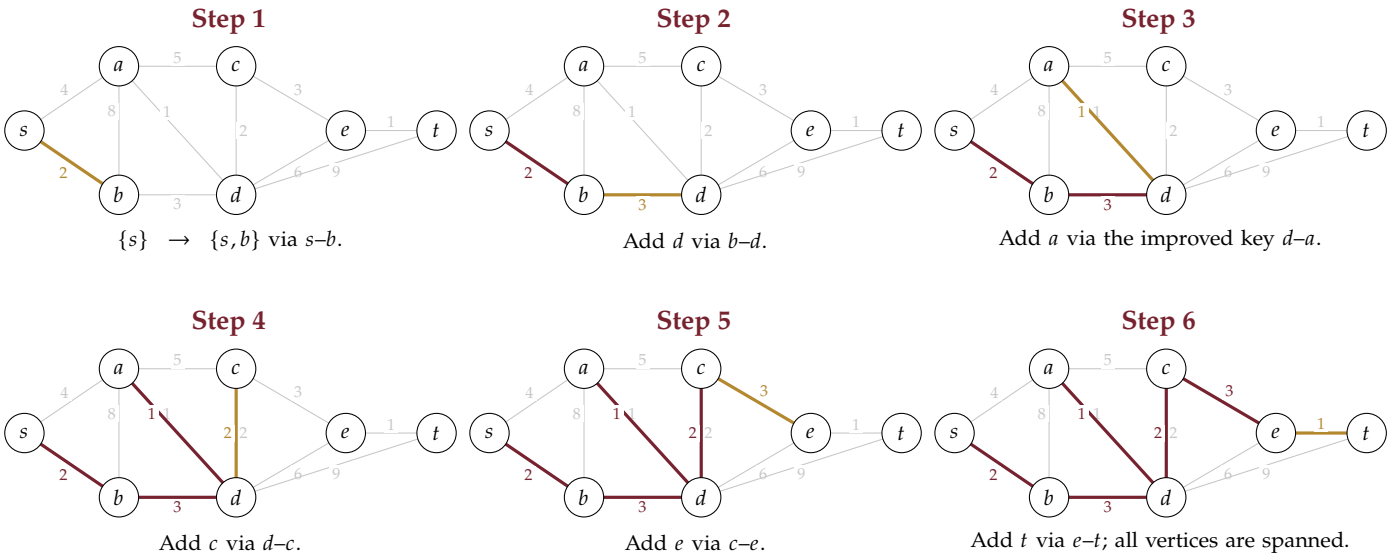


Figure 9.6: Prim's algorithm on the running graph, starting from s. Each panel shows the cumulative tree after one edge addition. The newest edge is gold, earlier selected edges are burgundy, and unselected edges are light gray. Total weight: $2 + 3 + 1 + 2 + 3 + 1 = 12$.

Intermezzo — Prim, Jarník, and Dijkstra

The algorithm we call "Prim's" was discovered independently at least three times. Vojtěch Jarník described it in 1930 in a Czech journal, Robert Prim published it in 1957 at Bell Labs, and Edsger Dijkstra rediscovered it in 1959. Dijkstra's 1959 paper actually contains *two* algorithms: one for MST (identical to Prim's) and one for shortest paths (chapter 10). The structural similarity between the two is no accident—both grow a tree by repeatedly selecting the "best" edge from a cut.

9.4 Kruskal's Algorithm

Kruskal's algorithm sorts all edges and adds them greedily, using union-find to detect cycles.

Kruskal's algorithm takes a different approach. Instead of growing a single tree, it considers edges in order of increasing weight and adds each edge unless it would create a cycle. The partial solution is a forest—a collection of trees—that gradually merges into a single spanning tree.

9.4.1 Algorithm Description

Formal details — Kruskal's algorithm — pseudocode

Algorithm 5: Kruskal's MST Algorithm

Input : Connected undirected graph $G = (V, E)$ with weights w
Output: Set of MST edges E_T

Sort edges E by weight: $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$
$E_T \leftarrow \emptyset$
Initialise union-find: each vertex is its own component
for $i = 1$ **to** m **do**
    Let $e_i = \{u, v\}$
    if $\textsc{Find}(u) \neq \textsc{Find}(v)$ **then**
        $E_T \leftarrow E_T \cup \{e_i\}$
        $\textsc{Union}(u, v)$
    if $|E_T| = n - 1$ **then**
        break
return E_T

9.4.2 Correctness

Correctness again follows from the cut property. When we consider edge $e = \{u, v\}$ and find that u and v are in different components, edge e crosses the cut $\delta(S)$ where S is the component containing u. Moreover, e is a minimum-weight edge crossing this component cut: any lighter crossing edge would have been processed earlier and would already have merged S with another component. By theorem 9.2.4, e is safe.

Intuitively: Kruskal's processes edges from lightest to heaviest. When it considers edge $e = \{u, v\}$, all lighter edges have already been accepted or rejected. The accepted ones form the current forest, and e crosses a cut between the component of u and the rest — making it safe by the cut property.

The **loop invariant** makes this precise: at every iteration, A is a forest each of whose trees is a subtree of some MST. When edge $e = \{u, v\}$ is examined with u and v in different components C_u and C_v, take the cut $(C_u,\ V \setminus C_u)$. All edges lighter than e were considered earlier and either added (extending some component) or skipped (both endpoints already in the same component, so they could not cross this cut). Hence e is the minimum-weight edge crossing $(C_u,\ V \setminus C_u)$, and the cut property certifies it as safe. Edges rejected because they form a cycle are correctly discarded, since any spanning tree must be cycle-free. After $n - 1$ safe additions the forest is a spanning tree, and the invariant implies it is an MST.

9.4.3 Union-Find Data Structure

The efficiency of Kruskal's algorithm hinges on how quickly we can answer the question "are u and v in the same component?" and merge two components. The **union-find** (or *disjoint-set*) data structure handles both operations efficiently.

Each component is represented as a rooted tree. Every element stores a pointer to its parent; the root of each tree is the component's **representative**. Two operations are supported:

- FIND(x): follow parent pointers from x to the root. With **path compression**—making every visited node point directly to the root—future queries on the same elements are nearly instantaneous.
- UNION(x, y): link the root of one tree to the root of the other. With **union by rank**—always attaching the shorter tree to the taller one—the trees stay shallow.

Remark 9.4.1 (Amortised complexity). With both path compression and union by rank, each operation runs in $O(\alpha(n))$ amortised time, where α is the inverse Ackermann function. For all practical purposes, $\alpha(n) \leq 4$ for any n that fits in the observable universe. We can treat union-find operations as effectively constant-time.

9.4.4 Complexity

- **Sorting:** $O(m \log m) = O(m \log n)$ (since $m \leq n^2$, so $\log m \leq 2 \log n$).
- **Union-find operations:** $O(m \cdot \alpha(n)) \approx O(m)$.
- **Total:** $O(m \log n)$, dominated by sorting.

9.4.5 Walkthrough on the Running Graph

Example 9.4.2 (Kruskal's algorithm step by step)**.** We sort the 11 edges by weight and process them in order until $n - 1 = 6$ edges have been accepted.

#	Edge	Weight	Action	Components after
1	a–d	1	Accept	$\{a,d\},\{s\},\{b\},\{c\},\{e\},\{t\}$
2	e–t	1	Accept	$\{a,d\},\{s\},\{b\},\{c\},\{e,t\}$
3	s–b	2	Accept	$\{a,d\},\{s,b\},\{c\},\{e,t\}$
4	c–d	2	Accept	$\{a,d,c\},\{s,b\},\{e,t\}$
5	b–d	3	Accept	$\{a,d,c,s,b\},\{e,t\}$
6	c–e	3	Accept	$\{a,d,c,s,b,e,t\}$
7	s–a	4	Not reached	(would form a cycle)
8	a–c	5	Not reached	(would form a cycle)
9	d–e	6	Not reached	(would form a cycle)
10	a–b	8	Not reached	(would form a cycle)
11	d–t	9	Not reached	(would form a cycle)

MST edges: $\{a\text{-}d, e\text{-}t, s\text{-}b, c\text{-}d, b\text{-}d, c\text{-}e\}$. Total weight: $1+1+2+2+3+3 = 12$.

This is the same MST found by Prim's algorithm in theorem 9.3.1, as expected.

Let us highlight the interesting moments.

Edges 1–2. The two cheapest edges (a–d and e–t, both weight 1) are in different parts of the graph. Unlike Prim, Kruskal happily creates two disconnected trees at the start.

Edges 3–4. Edges s–b and c–d (both weight 2) each create or extend a small component. After step 4, we have three components: $\{a,d,c\}$, $\{s,b\}$, and $\{e,t\}$.

Edge 5. Edge b–d (weight 3) merges the two largest components: $\{s,b\}$ joins $\{a,d,c\}$ to form $\{a,d,c,s,b\}$. Notice that b and d were in different

components (verified by FIND(b) ≠ FIND(d)).

Edge 6. Edge c–e (weight 3) bridges the last gap, connecting $\{e, t\}$ to the main component. We now have $n - 1 = 6$ edges and the algorithm terminates.

Edges 7–11. The pseudocode stops after edge 6 because the forest already has $n - 1$ edges. If the scan continued, every remaining edge would be rejected because its endpoints are already in the single spanning component; each would create a cycle.

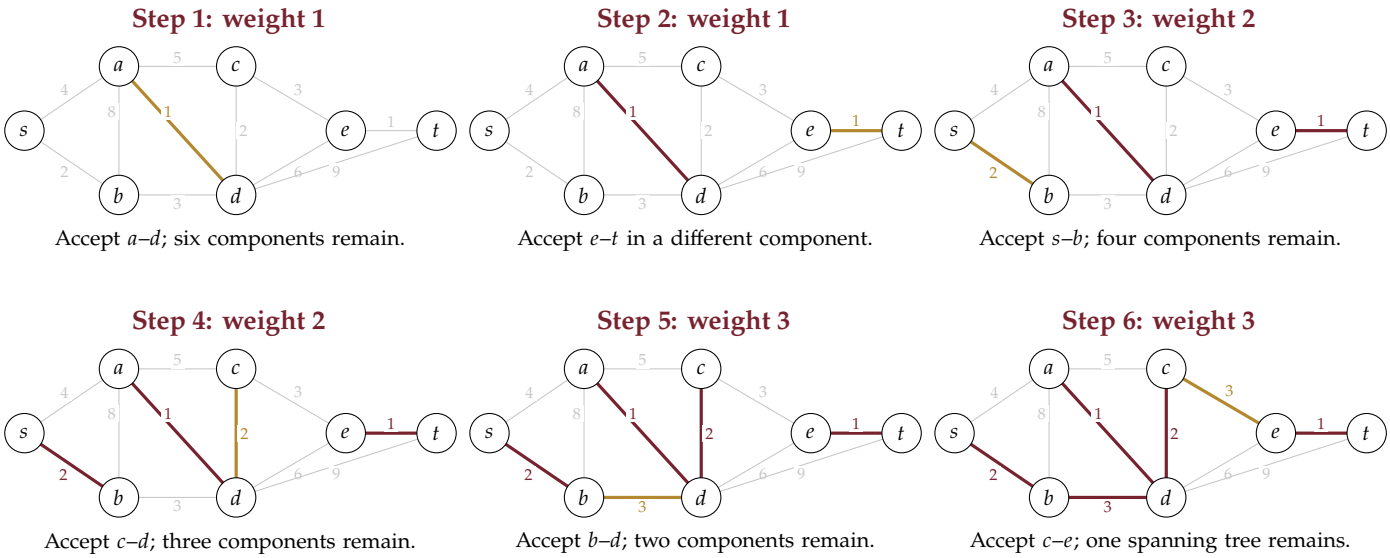


Figure 9.7: Kruskal's algorithm on the running graph. Each panel shows the cumulative forest after one accepted edge. The newest edge is gold, earlier accepted edges are burgundy, and unprocessed edges are light gray. The algorithm stops after step 6. Total weight: 1 + 1 + 2 + 2 + 3 + 3 = 12.

Intermezzo — Kruskal and Borůvka

Joseph Kruskal published his algorithm in 1956. But the earliest MST algorithm is due to Otakar Borůvka, who described it in 1926 to solve the problem of electrifying Moravia. Borůvka's algorithm works in rounds: in each round, every component simultaneously adds its cheapest outgoing edge. The number of components at least halves each round, so the algorithm terminates in $O(\log n)$ rounds, each costing $O(m)$, for a total of $O(m \log n)$. It is particularly well-suited for parallel computation.

9.5 Comparing Prim and Kruskal

Both algorithms are correct (by the cut property) and efficient, but their characteristics differ in ways that matter in practice.

When to prefer which:

- **Dense graphs** ($m = \Theta(n^2)$): the naïve $\Theta(n^2)$ Prim is unbeatable—it matches the cost of simply reading the input. Kruskal would need $O(n^2 \log n)$ just for the sorting step.
- **Sparse graphs** ($m = O(n)$): both algorithms achieve $O(n \log n)$. Kruskal is often preferred because its implementation is straightforward: sort, loop, union-find.

Rule of thumb: use naïve Prim for dense graphs, Kruskal for everything else.

Table 9.1: Comparison of Prim's and Kruskal's algorithms.

	Prim	**Kruskal**
Strategy	Grow one tree from root	Merge forest components
Key data structure	Priority queue (heap)	Union-find
Dense graphs ($m \approx n^2$)	$\Theta(n^2)$ (array scan)	$O(n^2 \log n)$
Sparse graphs ($m \approx n$)	$O(n \log n)$ (binary heap)	$O(n \log n)$
General	$O(m \log n)$ (binary heap)	$O(m \log n)$
Implementation ease	Moderate	Simple

- **General case**: both run in $O(m \log n)$. Prim with a Fibonacci heap achieves $O(m + n \log n)$, which is theoretically superior but rarely faster in practice.

9.6 Variations

9.6.1 Maximum Spanning Tree

Sometimes we want the *most expensive* spanning tree rather than the cheapest. For example, in a reliability network where edge weights represent link strength, we may want to maximise the total strength.

To find a maximum spanning tree, just negate all weights and run your favourite MST algorithm.

Proposition 9.6.1 (Maximum spanning tree)**.** *A maximum spanning tree of G with weights w is a minimum spanning tree of G with weights $-w$.*

Proof. Every spanning tree has exactly $n - 1$ edges, so negating all weights reverses the ordering of trees by total weight without changing *which subsets* are spanning trees. Hence the maximum of $\sum w(e)$ equals $-\min \sum(-w(e))$. □

Equivalently, we can adapt the algorithms directly: in Prim's, select the *maximum*-weight cut edge; in Kruskal's, sort edges in *decreasing* order and add each edge if it connects two different components.

9.6.2 Negative Edge Weights

A natural question: do Prim's and Kruskal's algorithms still work when some edge weights are negative?

Proposition 9.6.2. *Both Prim's and Kruskal's algorithms correctly find the MST even when the graph contains edges with negative weights.*

The key observation is that the cut property (theorem 9.2.4) never assumes non-negativity. The exchange argument in the proof only requires that $w(e^*) \leq w(f)$, which holds regardless of sign.

Alternatively, we can reduce negative-weight MST to the non-negative case. Let $c_{\min} = \min_{e \in E} w(e)$. Define new weights $w'(e) = w(e) - c_{\min} \geq 0$. Since every spanning tree has exactly $n - 1$ edges, the total weight changes by the same constant $(n-1) \cdot (-c_{\min})$ for every tree. The relative ordering is preserved, so the MST under w is the same as under w'.

This contrasts sharply with shortest paths: Dijkstra's algorithm (chapter 10) fails with negative weights, but MST algorithms do not.

9.6.3 Uniqueness of the MST

When can the MST fail to be unique? Only when there are ties in edge weights.

Proposition 9.6.3 (MST uniqueness)**.** *If all edge weights in G are distinct, then the MST is unique.*

Formal details — Proof sketch of uniqueness

Proof. Suppose for contradiction that there are two distinct MSTs T_1 and T_2, both with the same (minimum) total weight. Let e be the lightest edge in the symmetric difference $E_{T_1} \triangle E_{T_2}$. Without loss of generality, $e \in E_{T_1} \setminus E_{T_2}$. Adding e to T_2 creates a unique cycle C. The cycle must contain an edge $f \in E_{T_2} \setminus E_{T_1}$ (otherwise the cycle would be entirely in T_1, contradicting T_1 being a tree). Since e is the lightest edge in the symmetric difference, $w(f) > w(e)$ (weights are distinct). Then $T_2' = T_2 \cup \{e\} \setminus \{f\}$ has $w(T_2') < w(T_2)$, contradicting the optimality of T_2. □

Remark 9.6.4. When edge weights are not all distinct, there may be multiple MSTs with the same total weight. Both Prim's and Kruskal's algorithms will find *one* of them; which one depends on tie-breaking rules.

9.6.4 Bottleneck Spanning Trees

Definition 9.6.5 (Minimum bottleneck spanning tree)**.** A **minimum bottleneck spanning tree** (MBST) is a spanning tree that minimises the weight of its heaviest edge: $\min_T \max_{e \in E_T} w(e)$.

Proposition 9.6.6. *Every MST is also a minimum bottleneck spanning tree.*

Proof. The idea: if T^* were not a minimum bottleneck tree, we could swap its heaviest edge for a lighter one, contradicting its optimality as an MST.

Let T^* be an MST with bottleneck value $b^* = \max_{e \in T^*} w(e)$. Suppose for contradiction that T^* is not an MBST, so there exists a spanning tree T' with $\max_{e \in T'} w(e) < b^*$. In particular, *every* edge of T' has weight strictly less than b^*.

Let $e^* \in T^*$ be a heaviest edge of T^* (so $w(e^*) = b^*$). Removing e^* from T^* splits V into two parts S and $V \setminus S$. Since T' is a spanning tree, it must contain at least one edge f crossing the cut $(S, V \setminus S)$. By assumption, $w(f) < b^* = w(e^*)$.

Now replace e^* with f in T^*: the resulting graph $T^* - e^* + f$ is still a spanning tree (it is connected and has $n - 1$ edges), and its total weight is $w(T^*) - w(e^*) + w(f) < w(T^*)$. This contradicts the minimality of T^*. Hence no such T' exists, and T^* is an MBST. □

The converse does not hold: an MBST need not be an MST (it might have a larger total weight while keeping the maximum edge light). However, since computing an MST also gives us an MBST "for free," there is rarely a need for specialised MBST algorithms.

9.7 MST and Integer Programming

The connection to chapter 7 closes a nice loop: the MST also has an integral LP description, although here the reason is the graphic-matroid structure rather than TUM.

The MST problem can be formulated as an integer linear programme. Introduce a binary variable $x_e \in \{0,1\}$ for each edge $e \in E$, where $x_e = 1$ means edge e is included in the spanning tree.

Definition 9.7.1 (Edges induced by a vertex subset)**.** For any vertex subset $S \subseteq V$, define

$$E(S) = \big\{\{u,v\} \in E : u \in S,\ v \in S\big\}.$$

Thus, $E(S)$ contains the edges with *both* endpoints inside S. This differs from the cut

$$\delta(S) = \big\{\{u,v\} \in E : |\{u,v\} \cap S| = 1\big\},$$

whose edges have exactly one endpoint in S. The set S is not a decision variable: it ranges over every proper vertex subset. Each choice of S generates one constraint.

$$\begin{aligned}
\text{minimize} \quad & \sum_{e \in E} w(e)\, x_e \\
\text{subject to} \quad & \sum_{e \in E} x_e = n - 1 \\
& \sum_{e \in E(S)} x_e \le |S| - 1 \quad \text{for all } S \subsetneq V, \quad 2 \le |S| \le n-1 \\
& x_e \in \{0,1\} \qquad\qquad \text{for all } e \in E
\end{aligned}$$

The first constraint ensures we select exactly $n-1$ edges, but that count alone does *not* ensure a tree: those edges could contain a cycle in one part of the graph while leaving another part disconnected. The cycle-elimination constraints (often called "subtour elimination" constraints)

$$x(E(S)) := \sum_{e \in E(S)} x_e \le |S| - 1$$

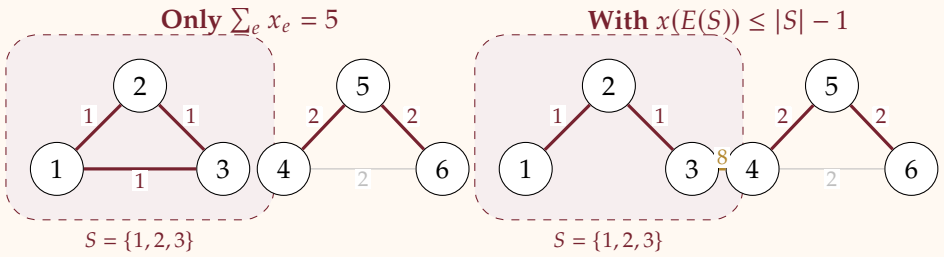
forbid this. Among the vertices of any set S, an acyclic selected subgraph can use at most $|S|-1$ internal edges. We omit $|S| \le 1$ because those constraints are trivial.

Once all subset constraints hold, the selected edges are acyclic. Together with the equation $\sum_e x_e = n-1$, this also forces connectivity: a forest on n vertices with $n-1$ edges has exactly one component and is therefore a spanning tree.

Let us unpack this formulation. The number of subtour elimination constraints is exponential—essentially one for every proper subset of V—so we could never list them all for a large graph. In a general-purpose solver they are usually generated only when violated (*cut generation*). The formulation nevertheless precisely describes the spanning tree polytope; for the basic MST, Prim and Kruskal remain far more efficient.

Example 9.7.2 (Why the subset constraints are necessary)**.** Consider six vertices arranged as two triangles, joined only by edge $\{3,4\}$. The three left-triangle edges have weight 1, the three right-triangle edges have weight 2,

and the bridge has weight 8.



Five cheapest edges, cost 7: disconnected, with a cycle in S.

Feasible MST, cost 14: two internal edges per triangle plus the bridge.

Figure 9.8: Why selecting exactly $n-1$ edges is insufficient. For $S = \{1,2,3\}$, the induced edge set is $E(S) = \{\{1,2\},\{2,3\},\{1,3\}\}$. The left solution selects all three, violating $x(E(S)) \le |S| - 1 = 2$. The bridge $\{3,4\}$ belongs to $\delta(S)$, not to $E(S)$.

If we kept only the edge-count equation, the model would choose the five cheapest edges:

$$x_{12} = x_{23} = x_{13} = x_{45} = x_{56} = 1, \qquad \text{cost } 1+1+1+2+2 = 7.$$

This vector has five selected edges, as required, but it is not a spanning tree: vertices $\{1,2,3\}$ form a cycle and the two triangles are disconnected. For $S = \{1,2,3\}$, the model includes

$$x_{12} + x_{23} + x_{13} \le 2.$$

The cheap vector has left-hand side 3 and is cut off. Symmetrically, $S = \{4,5,6\}$ prevents selecting all three edges of the right triangle. Any feasible five-edge solution must therefore use at most two edges inside each triangle and must select the only connecting edge $\{3,4\}$. The optimal tree has cost $1+1+8+2+2 = 14$, as shown on the right of fig. 9.8.

Remark 9.7.3 (LP relaxation is integral). The LP relaxation of the spanning tree formulation (replacing $x_e \in \{0,1\}$ with $0 \le x_e \le 1$) always has an integer optimal solution. This follows from the theory of *matroids*: the feasible spanning trees are exactly the *bases* of the *graphic matroid* of G, and the base polytope of any matroid has only integral extreme points (Edmonds, 1970). Consequently, any linear function over spanning trees is optimised at an integer vertex of the LP relaxation, so there is no integrality gap. In other words, the greedy algorithms of Prim and Kruskal are not just heuristics—they provably solve the LP to integrality.

The TUM theory from chapter 7 gives integrality for flow *and* matching *polytopes via totally unimodular constraint matrices. For spanning trees the analogous result goes through matroids rather than TUM directly.*

Remark 9.7.4 (NP-hard variants). Adding structural constraints to the spanning tree problem can make it NP-hard. For instance:

- **Degree-constrained MST**: find a minimum-weight spanning tree where no vertex has degree exceeding a given bound k.
- **Leaf-constrained MST**: find a minimum-weight spanning tree with

at most (or at least) ℓ leaves.
Both problems are NP-hard because they generalise the Hamiltonian path problem: a Hamiltonian path is a spanning tree with maximum degree 2 and exactly 2 leaves.

9.8 Summary

Table 9.2: Summary of MST algorithms and their complexities.

Algorithm	Key idea	Complexity
Prim (array)	Grow one tree; array scan	$\Theta(n^2)$
Prim (binary heap)	Grow one tree; heap extract-min	$O(m \log n)$
Prim (Fibonacci heap)	Grow one tree; $O(1)$ decrease-key	$O(m + n \log n)$
Kruskal	Sort edges; union-find	$O(m \log n)$

Key takeaways:

- The **cut property** guarantees that greedily adding the cheapest cut edge is always safe—the foundation of all MST algorithms.
- **Prim's algorithm** grows a single tree, selecting the cheapest edge to a non-tree vertex. The $\Theta(n^2)$ implementation is optimal for dense graphs.
- **Kruskal's algorithm** sorts all edges and adds them if they connect different components, using union-find for cycle detection. Its $O(m \log n)$ complexity is dominated by sorting.
- Both algorithms handle **negative weights** correctly. The MST is **unique** when all edge weights are distinct.
- The MST polytope is integral because it is the base polytope of the graphic matroid. This is analogous to, but not a consequence of, the TUM results in chapter 7.

Looking ahead.

- **chapter 10:** Dijkstra's shortest-path algorithm has a structure strikingly similar to Prim's—both grow a tree from a root using a priority queue. The difference lies in what the key values represent (edge weight for Prim, path length for Dijkstra).
- **chapter 11:** Spanning trees appear as the basic feasible solutions of network flow problems. The simplex method on networks pivots between spanning tree bases.

Summary & Key Takeaways

- **Minimum Spanning Tree (MST):** An acyclic subgraph connecting all vertices with minimum total edge weight.
- **Cut Property:** For any cut, the edge of minimum weight crossing the cut belongs to some MST.
- **Cycle Property:** For any cycle, the edge of maximum weight on the cycle does not belong to any MST.
- **Algorithms:**
  - *Kruskal's:* Sorts edges and greedily adds them if they don't form a cycle. Uses Disjoint Set Union (DSU) in $O(E \log E)$ time.

– *Prim's:* Grows a tree from a start vertex, adding the cheapest outgoing edge. Uses Min-Priority Queue in $O(E + V \log V)$ time.

Exercises

Exercise 1. Let $G = (V, E)$ be a connected undirected graph with n vertices.

(a) State the definition of a *spanning tree* of G.

(b) Prove that every spanning tree of G has exactly $n - 1$ edges.

(c) Give an example of a graph on 5 vertices and exhibit two distinct spanning trees of it.

Exercise 2. Let T be a spanning tree of $G = (V, E)$.

(a) State the *unique path property*: for any two vertices $u, v \in V$ there is a unique path in T between u and v.

(b) Use this property to prove that adding any non-tree edge $e \in E \setminus T$ to T creates exactly one cycle.

(c) State the *unique cycle property* for an MST: if e is a non-tree edge, then $w(e) \geq w(f)$ for every edge f on the unique cycle created by e in T.

Exercise 3. Let $G = (V, E, w)$ be a connected weighted graph and let T be a spanning tree.

(a) Define the *fundamental cut* of a tree edge $e \in T$.

(b) State the *unique cut property* for an MST: if e is a tree edge, then $w(e) \leq w(f)$ for every edge f in the fundamental cut of e (assuming distinct edge weights).

(c) Explain why the cut property and the cycle property are in a sense dual to each other.

Exercise 4. True or false? For each statement, either prove it or give a counter-example.

(a) A minimum spanning tree always contains the globally minimum-weight edge of the graph.

(b) If all edge weights are distinct, the MST is unique.

(c) If two edges have the same weight, the MST is not unique.

(d) Every edge of a graph belongs to at least one spanning tree of that graph.

(e) Removing the heaviest edge from any cycle in a spanning tree yields a spanning tree of smaller or equal total weight.

Exercise 5. Let $G = (V, E, w)$ with $V = \{a, b, c, d, e\}$ and edges $ab(4)$, $ac(2)$, $bc(5)$, $bd(3)$, $cd(1)$, $ce(6)$, $de(7)$. Consider the cut $(S, V \setminus S)$ with $S = \{a, c, d\}$.

(a) List all edges that cross this cut.

(b) Identify the minimum-weight crossing edge.

(c) Using the cut property, argue that this edge must belong to every MST of G.

Exercise 6. State and prove the *cut property*: if $e = (u, v)$ is the unique minimum-weight edge crossing some cut $(S, V \setminus S)$ of a connected weighted graph $G = (V, E, w)$, then e belongs to every MST of G.

Exercise 7. Let $G = (V, E, w)$ with $V = \{1, 2, 3, 4, 5, 6\}$ and edges 12(8), 13(5), 14(9), 23(3), 24(7), 34(6), 35(2), 45

(a) Consider the cut $(\{1, 2, 3\}, \{4, 5, 6\})$. List all crossing edges and identify the minimum one.

(b) Now consider the cut $(\{1\}, \{2, 3, 4, 5, 6\})$. Which edge does the cut property force into the MST?

(c) Find the MST by hand and verify both edges appear in it.

Exercise 8. State and prove the *cycle property*: if e is the unique maximum-weight edge on some cycle C of a connected weighted graph $G = (V, E, w)$, then e does not belong to any MST of G.

Exercise 9. Let $G = (V, E, w)$ with $V = \{a, b, c, d\}$ and edges $ab(3)$, $ac(1)$, $ad(6)$, $bc(4)$, $bd(2)$, $cd(5)$. Suppose the MST T contains edges $\{ac, bd, ab\}$.

(a) Verify that T is indeed a spanning tree and compute its total weight.

(b) Add the non-tree edge $cd(5)$ to T. Identify the unique cycle created.

(c) Using the cycle property, confirm that cd cannot be in any MST.

Exercise 10. Run Kruskal's algorithm on the graph $G = (V, E, w)$ with $V = \{a, b, c, d, e\}$ and edges $ac(1)$, $de(1)$, $bd(2)$, $ab(3)$, $bc(4)$, $cd(5)$, $ce(6)$. Show each step: the edge considered, whether it is accepted or rejected, and the current forest after each decision. State the final MST and its total weight.

Exercise 11. Run Kruskal's algorithm on the graph $G = (V, E, w)$ with $V = \{1, 2, 3, 4, 5, 6\}$ and edges 56(1), 23(2), 14(3), 35(4), 12(5), 46(6), 16(7), 24(8), 36(9), 45(10). List edges in sorted order, then execute the algorithm step by step. State which edges are accepted and which are rejected (with reason), and give the final MST.

Exercise 12. Run Kruskal's algorithm on the graph $G = (V, E, w)$ with $V = \{p, q, r, s, t, u, v\}$ and edges $pq(2)$, $qr(3)$, $rs(1)$, $st(4)$, $tu(2)$, $uv(5)$, $pv(6)$, $qu(3)$, $rt(7)$, $sv(2)$.

(a) Sort the edges by weight.

(b) Execute the algorithm, recording the union-find structure (components) after each accepted edge.

(c) Report the MST edges and total weight.

Exercise 13. Run Kruskal's algorithm on the graph $G = (V, E, w)$ with $V = \{A, B, C, D, E, F\}$ and edges $AB(4)$, $AC(2)$, $BC(6)$, $BD(5)$, $CD(3)$, $CE(7)$, $DE(1)$, $DF(8)$, $EF(9)$, $AF(10)$.

(a) List edges sorted by non-decreasing weight.

(b) Trace the algorithm showing each edge in order, whether it is added (and why it does not create a cycle) or skipped (and which cycle it would create).

(c) State the MST and its total weight.

Exercise 14. Run Prim's algorithm on the graph $G = (V, E, w)$ with $V = \{a, b, c, d, e\}$ and edges $ab(3)$, $ac(1)$, $bc(4)$, $bd(2)$, $cd(5)$, $de(1)$, starting from source vertex a. At each step show the current tree T, the key values of all non-tree vertices, and the vertex extracted from the priority queue. State the final MST and its weight.

Exercise 15. Repeat the Prim trace from **the previous exercise** but start from source vertex d. Do you obtain the same MST? Explain why the choice of starting vertex does or does not affect the final result.

Exercise 16. Run Prim's algorithm on the graph $G = (V, E, w)$ with $V = \{1, 2, 3, 4, 5, 6\}$ and edges 12(6), 13(1), 14(5), 23(4), 24(2), 34(3), 35(7), 45(8), 46(9), 56(10) from source 1, then repeat from source 4. Compare the two traces and verify that both produce a tree of the same total weight.

Exercise 17. In Prim's algorithm with a binary min-heap:

(a) Identify the two main operations performed on the heap and state their time complexity.

(b) Derive the overall time complexity of Prim's with a binary heap in terms of $n = |V|$ and $m = |E|$.

(c) Explain when Prim's with a Fibonacci heap is preferred and state its time complexity.

Exercise 18. Compare the time complexities of Prim's algorithm and Kruskal's algorithm.

(a) State the complexity of each algorithm (Prim with binary heap, Prim with Fibonacci heap, Kruskal with union-find).

(b) For a *dense* graph ($m = \Theta(n^2)$), which algorithm is asymptotically faster?

(c) For a *sparse* graph ($m = \Theta(n)$), which algorithm is asymptotically faster?

(d) Explain the bottleneck in Kruskal's complexity when union-find is *not* used.

Exercise 19. Explain how the union-find data structure is used in Kruskal's algorithm.

(a) Describe the FIND and UNION operations.

(b) How is FIND used to check whether adding an edge would create a cycle?

(c) State the time complexity of m FIND/UNION operations with union-by-rank and path compression.

(d) Without union-find, what is the simplest way to check for cycles, and what is its complexity per operation?

Exercise 20. A set $F \subseteq E$ of edges is called *extendable* if F can be extended to a minimum spanning tree of G.

(a) Show that the empty set $\emptyset$ is always extendable.

(b) State the greedy property: if F is extendable and e is a safe edge for F (minimum-weight edge crossing some cut respected by F), then $F \cup \{e\}$ is also extendable.

(c) Use parts (a) and (b) to give a high-level correctness argument for both Prim's and Kruskal's algorithms.

Exercise 21. An edge $e = (u, v)$ is called *safe* for a partial tree F if there exists a cut $(S, V \setminus S)$ such that F has no edge crossing the cut and e is the minimum-weight edge crossing the cut.

(a) Give an example of a safe edge and a non-safe edge on a small graph $G = (V, E, w)$ of your choice with at least 5 vertices.

(b) Explain why Kruskal's algorithm only adds safe edges.

(c) Explain why Prim's algorithm only adds safe edges.

Exercise 22.

(a) Prove: if all edge weights in G are distinct, then G has a unique MST.

(b) Give an example of a graph on 4 vertices where not all weights are distinct and the MST is *not* unique.

(c) Give an example of a graph on 4 vertices where not all weights are distinct but the MST *is* unique.

Exercise 23. Let T be an MST of a connected weighted graph $G = (V, E, w)$. Prove that T is the unique MST if and only if, for every non-tree edge $e = \{u, v\} \notin T$,

$$w(e) > \max\{w(f) : f \text{ lies on the unique } u\text{–}v \text{ path in } T\}.$$

Also show by example that having a unique minimum-weight edge across every cut is sufficient, but not necessary, for uniqueness of the MST.

Exercise 24. Let $G = (V, E, w)$ with $V = \{a, b, c, d\}$ and edges $ab(1)$, $ac(1)$, $bc(2)$, $bd(3)$, $cd(3)$.

(a) Find all spanning trees of G and compute their total weights.

(b) How many MSTs does G have? Identify them.

(c) Which cut witnesses the non-uniqueness (i.e., has two minimum-weight crossing edges)?

Exercise 25. Describe two methods for finding a *maximum* spanning tree of a weighted graph $G = (V, E, w)$.

(a) Method 1: negate all edge weights and run a standard MST algorithm. Prove this is correct.

(b) Method 2: run Kruskal's algorithm but sort edges in non-increasing order. Prove this is correct by stating the analogous cut property for maximum spanning trees.

(c) Apply Method 2 to the graph $G = (V, E, w)$ with $V = \{a, b, c, d, e\}$ and edges $ab(5)$, $ac(3)$, $bc(6)$, $bd(4)$, $cd(2)$, $de(7)$, $ce(1)$. Show each step and state the maximum spanning tree.

Exercise 26. The *bottleneck spanning tree* problem asks for a spanning tree T of G that minimises the weight of the maximum-weight edge in T.

(a) Prove that every MST is also a bottleneck spanning tree.

(b) Is every bottleneck spanning tree an MST? Prove or give a counter-example.

(c) Apply Kruskal's algorithm to the graph $G = (V, E, w)$ with $V = \{1, 2, 3, 4, 5\}$ and edges 12(9), 13(3), 14(7), 23(5), 24(6), 34(2), 35(8), 45(4) and identify the bottleneck value of the MST.

Exercise 27. Suppose all edge weights of G are negative.

(a) Does Prim's algorithm still correctly find the MST? Justify your answer by examining where the correctness proof uses the sign of the weights.

(b) Does Kruskal's algorithm still correctly find the MST? Justify.

(c) What is the total weight of an MST when all weights are negative? Is it the smallest (most negative) or the largest (least negative) of all spanning tree weights?

Exercise 28. Let $G = (V, E, w)$ be a connected graph with n vertices and m edges. Introduce binary variables $x_e \in \{0, 1\}$ for each edge $e \in E$.

(a) Write an integer linear program (ILP) whose optimal solution is the MST. Your formulation must include the subtour-elimination (or connectivity) constraints.

(b) Show that the constraint "exactly $n - 1$ edges are selected" is necessary but not sufficient for a feasible spanning tree.

(c) Explain the role of the cut constraints: for every non-empty $S \subsetneq V$, at least one edge in $\delta(S)$ must be selected. Compare this alternative integer formulation with the cycle-elimination formulation on $E(S)$ used in section 9.7.

Exercise 29. The MST polytope is the convex hull of the incidence vectors of all spanning trees of G.

(a) Explain what it means for a matrix to be *totally unimodular* (TUM) and why TUM implies that the LP relaxation of an ILP has an integral optimal solution.

(b) Explain why the integrality of the spanning-tree formulation is instead obtained by viewing it as the base polytope of the graphic matroid; it is not a direct application of the TUM argument.

(c) What is the practical consequence: why can a greedy algorithm optimise a linear objective over this LP relaxation without branch-and-bound?

Exercise 30. The *second-best MST* is the spanning tree with the second smallest total weight among all spanning trees of G.

(a) Argue that the second-best MST differs from the MST in exactly one edge swap: some tree edge e is removed and some non-tree edge f is inserted.

(b) Describe an $O(n^2)$ algorithm to find the second-best MST given the MST T.

(c) Apply your algorithm to the graph $G = (V, E, w)$ with $V = \{a, b, c, d, e\}$ and edges $ab(1)$, $bc(2)$, $cd(3)$, $de(4)$, $ae(5)$, $ac(6)$, $bd(7)$. First find the MST, then find the second-best MST.

Exercise 31. Let $G = (V, E, w)$ with $V = \{a, b, c, d, e\}$ and edges $ab(2)$, $ac(4)$, $bc(1)$, $bd(5)$, $cd(3)$, $de(6)$.

(a) Find the MST T of G.

(b) For the MST edge $cd(3)$: by how much can its weight increase before a different spanning tree becomes optimal? (*Hint:* consider the minimum-weight non-tree edge in the fundamental cut of cd.)

(c) For the non-tree edge $ac(4)$: by how much can its weight decrease before it enters the MST? (*Hint:* consider the maximum-weight tree edge on the path from a to c in T.)

Exercise 32. Suppose T is the unique MST of $G = (V, E, w)$ and let $e \in T$ be a tree edge with weight $w(e)$. Let δ be the minimum difference $w(f) - w(e)$ over all non-tree edges f in the fundamental cut of e.

(a) Prove that increasing $w(e)$ by any amount strictly less than δ keeps T optimal.

(b) Prove that increasing $w(e)$ by exactly δ or more may cause T to no longer be the unique MST.

(c) Explain the analogous result for a non-tree edge f: by how much can $w(f)$ decrease before f enters the MST?

Exercise 33. A telecommunications company wants to connect 6 cities $\{C_1, C_2, C_3, C_4, C_5, C_6\}$ at minimum total cable cost. The possible direct links and their costs (in thousands of euros) are: $C_1C_2(15)$, $C_1C_3(10)$, $C_2C_3(8)$, $C_2C_4(12)$, $C_3C_4(7)$, $C_3C_5(9)$, $C_4C_5(5)$, $C_4C_6(11)$, $C_5C_6(6)$.

(a) Model this as an MST problem (identify V, E, and w).

(b) Run Kruskal's algorithm to find the minimum-cost network.

(c) What is the total installation cost?

Exercise 34. An electrical grid must connect n substations. Each pair of substations can be linked at a known cost. Formulate the problem of finding the cheapest connected network (with no redundant lines) as an MST instance.

(a) Why is it valid to restrict to spanning trees (i.e., why are cycles wasteful)?

(b) If the grid must tolerate the failure of any single link (i.e., the network must remain connected after removing any one edge), is an MST sufficient? If not, what structure is needed?

Exercise 35. Prove the correctness of Kruskal's algorithm using the cut property. Your proof should:

(a) Show that every edge added by Kruskal is safe (belongs to some MST).

(b) Show that the algorithm terminates with a spanning tree.

(c) Conclude that the result is an MST.

Exercise 36. Prove the correctness of Prim's algorithm using the cut property. At each iteration, Prim adds the minimum-weight edge from the current tree T to a non-tree vertex.

(a) Identify the cut that Prim's edge selection satisfies at each step.

(b) Show that the selected edge is safe.

(c) Conclude by induction that the final tree is an MST.

Exercise 37. Let T_1 and T_2 be two distinct spanning trees of $G = (V, E, w)$ with the same total weight.

(a) Are T_1 and T_2 both MSTs? Explain.

(b) Is it possible that T_1 is an MST but T_2 is not? Prove or give a counter-example.

(c) Describe a procedure that, given T_1 and T_2 with equal total weight, either confirms both are MSTs or finds one that is not.

Exercise 38. Let $G = (V, E, w)$ and suppose we add a new edge $e^* = (u, v)$ with weight $w(e^*)$ to G, obtaining $G' = (V, E \cup \{e^*\}, w)$. Let T be an MST of G.

(a) If $w(e^*)$ is larger than every edge on the unique path from u to v in T, is T still an MST of G'? Prove your answer.

(b) If $w(e^*)$ is smaller than some edge f on the path from u to v in T, describe how to update T to get an MST of G'.

Exercise 39. Suppose we delete an edge $e \in T$ from the MST T of $G = (V, E, w)$. This splits T into two subtrees T_1 and T_2.

(a) Describe how to find a new MST of $G \setminus \{e\}$ if G remains connected.

(b) What is the minimum-weight edge that can reconnect T_1 and T_2?

(c) State the time complexity of this update operation.

Exercise 40. For each statement below, state whether it is true or false and give a brief justification (one or two sentences, or a small counter-example).

(a) "The MST always contains the globally minimum-weight edge of the graph."

(b) "If all edge weights are distinct, the MST is unique."

(c) "Prim's algorithm and Kruskal's algorithm always produce the same MST."

(d) "The MST of a graph is also a shortest-path tree from any root."

(e) "Adding a constant $c > 0$ to every edge weight does not change the MST."

Exercise 41. Kruskal's algorithm requires sorting the edges.

(a) Show that sorting dominates the runtime when $m = \omega(n\,\alpha(n))$, where α is the inverse-Ackermann function.

(b) Suppose all edge weights are integers in the range $[1, W]$. Suggest a sorting algorithm that runs in $O(m + W)$ time and state the resulting complexity of Kruskal's.

(c) For a complete graph on n vertices, compare the runtimes of Prim (binary heap), Prim (Fibonacci heap), and Kruskal (comparison sort).

Exercise 42. Prim's algorithm can be implemented with different priority-queue data structures.

(a) Implement Prim's with a simple array (no heap). State its time complexity and argue that it is optimal for dense graphs.

(b) Implement Prim's with a binary min-heap. Count the number of ExtractMin and DecreaseKey operations and derive the total time complexity.

(c) State the time complexity of Prim's with a Fibonacci heap and explain which operation becomes $O(1)$ amortized.

Exercise 43. Let $G = (V, E, w)$ be a complete graph on $n = 5$ vertices with edge weights $w(ij) = i + j$ for all $i < j$ (i.e., vertices are labelled $1, 2, 3, 4, 5$ and edge ij has weight $i + j$).

(a) List all edges and their weights.

(b) Run Kruskal's algorithm and state the MST.

(c) Verify the total weight of the MST.

Exercise 44. The set of forests of a graph $G = (V, E)$ forms a *graphic matroid* $\mathcal{M} = (E, \mathcal{I})$, where $\mathcal{I}$ is the collection of all acyclic subsets of E.

(a) State the three axioms that $\mathcal{I}$ must satisfy to be a matroid (non-emptiness, hereditary property, augmentation property).

(b) Verify that forests satisfy all three axioms.

(c) Explain why the greedy algorithm (sort edges by weight, add edge if it is independent) finds the maximum-weight basis of any matroid, and how this immediately implies the correctness of Kruskal's algorithm for the MST.

Exercise 45. Run Prim's algorithm on the graph $G = (V, E, w)$ with $V = \{A, B, C, D, E, F, G\}$ and edges $AB(4)$, $AC(8)$, $BC(11)$, $BD(8)$, $CD(7)$, $CE(1)$, $CF(2)$, $DE(6)$, $DF(2)$, $EG(9)$, $FG($ starting from vertex A. At each step, record the vertex added to the tree, the edge used, and the updated key values of all remaining vertices. State the final MST and its total weight.

Exercise 46. Run Kruskal's algorithm on the same graph as the previous exercise ($V = \{A, B, C, D, E, F, G\}$, same edges and weights). List edges in non-decreasing order of weight and show which are accepted or rejected at each step, indicating the reason for each rejection. Verify that you obtain the same total MST weight as Prim's algorithm.

Exercise 47. Let $G = (V, E, w)$ be a weighted graph with non-negative edge weights.

(a) Give an example where the MST is not a shortest-path tree from any fixed root s.

(b) Give an example (if one exists) where the MST *is* also a shortest-path tree from some root s.

(c) Explain the structural difference between Prim's algorithm and Dijkstra's algorithm that causes them to compute different objects despite their similar structure.

Exercise 48. Let $G = K_4$ (the complete graph on 4 vertices $\{1, 2, 3, 4\}$) with unit edge weights.

(a) Use Cayley's formula to determine the number of spanning trees of K_4.

(b) List all distinct spanning trees and verify the count.

(c) Since all weights are equal, every spanning tree is an MST. What does this imply for the uniqueness of the MST of K_4?

Exercise 49. A city planner wants to lay water pipes to connect 7 districts. Denote the districts $D_1, \ldots, D_7$. The possible pipe segments and their construction costs (in million euros) are: $D_1D_2(3)$, $D_1D_3(6)$, $D_2D_3(4)$, $D_2D_4(5)$, $D_3D_4(2)$, $D_3D_5(8)$, $D_4D_5(1)$, $D_4D_6(7)$, $D_5D_6($

(a) Run Kruskal's algorithm to find the minimum-cost pipe network.

(b) State the total construction cost.

(c) If the cost of segment D_3D_4 increases to 6, does the MST change? Show your reasoning using the sensitivity analysis framework.

Exercise 50. A researcher needs to cluster n data points by building a minimum spanning tree on the complete weighted graph where edge weights represent distances.

(a) Explain why removing the $k-1$ heaviest MST edges produces a partition of the data into k clusters.

(b) For $k = 2$, describe what the two clusters represent in terms of the cut corresponding to the removed edge.

(c) Discuss one advantage and one disadvantage of MST-based clustering compared to k-means clustering.

Exercise 51. Let $G = (V, E, w)$ be a connected graph. Suppose we run both Prim's and Kruskal's algorithms and obtain spanning trees T_P and T_K respectively.

(a) Must $T_P = T_K$ as sets of edges? Under what conditions are they guaranteed to be identical?

(b) Prove that $w(T_P) = w(T_K)$ always holds.

(c) If G has exactly two edges with the same minimum weight, describe a scenario where $T_P \neq T_K$ yet both are MSTs.

Exercise 52. Let $G = (V, E, w)$ with $V = \{s, a, b, c, d, t\}$ and edges $sa(2)$, $sb(4)$, $ab(1)$, $ac(3)$, $bc(5)$, $bd(6)$, $cd(2)$, $($

(a) Find the MST using either Prim's (from s) or Kruskal's algorithm. Show all steps.

(b) Identify the fundamental cycle for each non-tree edge and verify the cycle property holds.

(c) Identify the fundamental cut for each tree edge and verify the cut property holds.

Exercise 53. Consider a weighted graph $G = (V, E, w)$ in which all edge weights are equal (say $w(e) = 1$ for all $e \in E$).

(a) How many MSTs does G have? Express your answer in terms of the number of spanning trees of G.

(b) Describe what Kruskal's algorithm does in this case (which edges are accepted/rejected and why).

(c) Is it possible to design a graph with n vertices and m edges (with $m > n - 1$) that has exactly one spanning tree? Explain.

Exercise 54. Recall that the MST polytope is defined as $P_{\text{MST}} = \text{conv}\{\chi^T : T \text{ is a spanning tree of } G\}$, where $\chi^T \in \{0,1\}^E$ is the incidence vector of T.

(a) Write the LP relaxation of the MST ILP (replace $x_e \in \{0,1\}$ with $0 \leq x_e \leq 1$) and state all constraints.

(b) Explain why this LP relaxation has integral extreme points by viewing it as the base polytope of the graphic matroid. Why is the TUM argument from network-flow formulations not the relevant justification here?

(c) What is the consequence for the practical solving of MST instances: do we need branch-and-bound or cutting planes?

Exercise 55. Prove or disprove each of the following claims about how global transformations of edge weights affect the MST of $G = (V, E, w)$.

(a) If every edge weight is multiplied by the same positive constant $\lambda > 0$, then any MST of G is also an MST of the rescaled graph.

(b) If the same constant c (possibly negative) is added to every edge weight, then any MST of G is also an MST of the shifted graph. (*Hint:* consider what happens to the total weight of every spanning tree, and whether the relative order of spanning trees by total weight is preserved.)

CHAPTER 10

Shortest Paths

Shortest path algorithms power every GPS navigator and every routing protocol on the Internet.

In chapter 8 we saw how to traverse graphs (BFS, DFS) and how to compute shortest paths on DAGs using topological sorting. In chapter 9 we solved the minimum-cost *connectivity* problem. We now tackle a different fundamental question: *what is the cheapest way to travel from one node to another?*

This is the **Shortest Path Problem** (SPP), arguably the single most important optimisation problem on networks. Its applications range from route planning in navigation systems to scheduling in project management. The problem is surprisingly rich: depending on the sign of the arc weights and the structure of the graph, it can be solved in linear time or require the more careful Bellman–Ford/Floyd–Warshall machinery. If we additionally force paths to be elementary in graphs with arbitrary negative cycles, the problem becomes NP-hard. Understanding *why* these cases differ—and choosing the right algorithm for each situation—is the central theme of this chapter.

Road map.

1. Problem definition, complexity, and tractable cases (section 10.1).
2. ILP formulation and total unimodularity (section 10.2).
3. Dijkstra's algorithm for non-negative weights (section 10.3).
4. Bellman–Ford algorithm for general weights (section 10.4).
5. Shortest paths in DAGs and the Critical Path Method (section 10.5).
6. Floyd–Warshall algorithm for all-pairs shortest paths (section 10.6).
7. Summary and comparison (section 10.7).

Throughout, we illustrate the algorithms on the *directed* version of the running graph from theorem 1.9.1, reproduced in fig. 10.1.

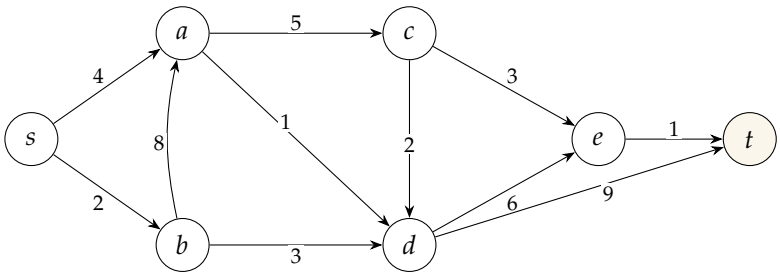


Figure 10.1: The directed running graph with $n = 7$ nodes and $m = 11$ arcs. Arc weights represent costs c_{ij}. The shortest path from s to t is $s \to b \to d \to e \to t$ with cost $2 + 3 + 6 + 1 = 12$.

10.1 Introduction and Complexity

10.1.1 Problem Definition

Standard SPP is polynomial when no relevant negative cycle is present. The elementary variant with arbitrary negative costs is hard.

Definition 10.1.1 (Path and path length)**.** Let $G = (V, A)$ be a directed graph with arc weights $c\colon A \to \mathbb{R}$. A **path** from s to t is a sequence of vertices $P = (v_0, v_1, \dots, v_k)$ with $v_0 = s$, $v_k = t$, and $(v_i, v_{i+1}) \in A$ for $i = 0, \dots, k-1$. The **length** (or **cost**) of P is

$$c(P) \;=\; \sum_{i=0}^{k-1} c(v_i, v_{i+1}).$$

A path is **simple** if no arc is repeated, and **elementary** if no node is repeated.

Remark 10.1.2 (Terminology warning). The terms *simple* and *elementary* as used above follow the convention common in operations research and network-flow texts. Note that the more common convention in graph theory is the reverse: a *simple* path has no repeated *vertex*, while a path with no repeated *arc* is called a *trail*. Throughout this chapter we use the OR convention: **simple** = no repeated arc, **elementary** = no repeated node.

Every elementary path is simple (no repeated nodes implies no repeated arcs), but the converse need not hold. In shortest-path algorithms we usually allow walks while computing distances. If the optimum is finite, an optimal walk can always be simplified to an elementary path by deleting cycles of non-negative total cost. Negative cycles are the only obstacle to this simplification.

Definition 10.1.3 (Shortest Path Problem)**.** Given a directed graph $G = (V, A)$ with arc costs $c\colon A \to \mathbb{R}$, a source $s \in V$ and a target $t \in V$, the **Shortest Path Problem** (SPP) asks for a minimum-cost s–t route. If a negative cycle is reachable from s and can reach t, the standard problem has value $-\infty$; otherwise an optimal route exists and can be chosen elementary.

Two important variants exist:

- **Single-source shortest paths (SSSP):** find shortest paths from a fixed source s to *all* other vertices.
- **All-pairs shortest paths (APSP):** find shortest paths between *every* pair of vertices.

Remark 10.1.4 (Undirected graphs). An undirected graph can be converted to a directed one by replacing each edge $\{i, j\}$ with two anti-parallel arcs (i, j) and (j, i), both having the same weight. All algorithms in this chapter therefore apply to undirected graphs as well.

10.1.2 Negative Cycles and NP-Hardness

If a directed **negative cycle** is reachable from s and can itself reach t, then one can traverse it arbitrarily many times and drive the s–t walk cost to $-\infty$. In this case, no minimum-cost s–t walk exists unless we enforce the elementary constraint, which makes the problem much harder. Negative cycles that cannot lie on an s–t walk do not affect that particular pair.

Theorem 10.1.5 (Elementary SPP with arbitrary negative costs is NP-hard)**.** *The problem of finding a minimum-cost elementary s–t path in a graph with arbitrary arc costs is NP-hard.*

Formal details — NP-hardness via Hamiltonian path reduction

We reduce from the Hamiltonian Path (HP) problem, which is NP-complete. Given an undirected graph $\hat{G} = ([n], E)$, construct a directed graph $G = (V, A)$ as follows:

1. $V = [n] \cup \{s, t\}$, where s and t are two new nodes.
2. For each edge $\{h, k\} \in E$, add two arcs (h, k) and (k, h), each with cost -1.
3. Add arcs (s, i) and (i, t) with cost 0 for every $i \in [n]$.

A Hamiltonian path in $\hat{G}$ uses $n - 1$ edges. The corresponding $s \to t$ elementary path in G traverses $n-1$ internal arcs of cost -1 and two terminal arcs of cost 0, giving a total cost of $-(n-1)$. Conversely, any s–t elementary path in G of cost $-(n-1)$ visits all n internal nodes and thus defines a Hamiltonian path in $\hat{G}$.

Therefore $\hat{G}$ has a Hamiltonian path if and only if the shortest elementary s–t path in G has cost $-(n-1)$. Since HP is NP-complete, the elementary shortest-path problem is NP-hard.

10.1.3 Tractable Cases

Non-negative weights, DAGs, or no negative cycles — each yields polynomial time.

Fortunately, the SPP becomes tractable whenever negative cycles are excluded. The following cases are all solvable in polynomial time:

1. **Non-negative weights** ($c_{ij} \geq 0$ for all arcs): negative cycles cannot form. Solved by Dijkstra's algorithm in $O(n^2)$ or $O(m \log n)$.
2. **DAGs:** no cycles at all, so no negative cycles. Solved via dynamic programming in topological order in $O(n + m)$.
3. **General weights, no negative cycles:** solved by Bellman–Ford in $O(nm)$.
4. **All-pairs shortest paths:** solved by Floyd–Warshall in $O(n^3)$; with non-negative weights, one may instead run Dijkstra from every source.

Observation 10.1.6. The key complexity driver is not the presence of negative weights per se, but the combination of negative weights *and* cycles. A DAG with negative weights is easy; a cyclic graph with only non-negative weights is also easy. When a relevant negative cycle can appear, the standard SPP has value $-\infty$ or must be treated with an explicit elementary-path restriction; that restricted variant is NP-hard.

10.2 ILP Formulation and Total Unimodularity

10.2.1 Flow-Based ILP

The SPP can be modelled as a minimum-cost flow of one unit from s to t.

We introduce binary variables $x_{ij} \in \{0, 1\}$ for each arc $(i, j) \in A$, indicating whether the arc belongs to the shortest path.

$$\begin{aligned} \text{minimize} \quad & \sum_{(i,j)\in A} c_{ij}\, x_{ij} \\ \text{subject to} \quad & \sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = b_i, \quad \forall\, i \in V \\ & x_{ij} \in \{0, 1\}, \quad \forall\, (i, j) \in A \end{aligned}$$

where $b_s = 1$, $b_t = -1$, and $b_i = 0$ for all $i \neq s, t$.

The constraints are standard *flow conservation*: the source sends out one unit of flow, the target absorbs it, and every intermediate node balances inflow with outflow.

10.2.2 Subtour Elimination

Subtour elimination constraints prevent disconnected negative-cost cycles from appearing in the solution.

Without additional constraints, the LP relaxation may include disconnected negative-cost cycles alongside the s–t path. These **subtours** reduce the objective value artificially. To eliminate them, we can add:

$$\sum_{\substack{(i,j)\in A:\\ i\in S,\ j\in S}} x_{ij} \ \leq\ |S| - 1, \qquad \forall\, S \subseteq V \setminus \{s,t\},\ S \neq \emptyset.$$

These constraints are exponential in number. However, when no negative cycles exist, they are *unnecessary*, as shown next.

10.2.3 TUM and Integrality

Theorem 10.2.1 (Integrality of the SPP relaxation)**.** *If the graph G contains no negative-cost cycles, the LP relaxation of the flow-based formulation (without subtour elimination constraints) always has an optimal integer solution.*

Proof sketch. The constraint matrix of the flow conservation constraints is the *node–arc incidence matrix* of the directed graph. This matrix is totally unimodular (TUM), as shown in chapter 7. By the Hoffman–Kruskal theorem, any LP with a TUM constraint matrix and integer right-hand side has integer optimal vertices. Since b is integer, the LP relaxation has an integer optimal extreme point.

When no negative cycles exist, any directed cycles in such an integer solution have non-negative cost and can be removed without increasing the objective. What remains is the incidence vector of an s–t path. Zero-cost cycles may therefore occur in some optimal solutions, but an optimal path solution always exists. □

Remark 10.2.2. This result connects shortest paths to the theory of totally unimodular matrices from chapter 7: the integrality "comes for free" from the network structure, allowing us to solve the problem as a simple LP rather than an ILP.

10.3 Dijkstra's Algorithm

Dijkstra's algorithm is the workhorse for non-negative weights—analogous to Prim's for MST.

Dijkstra's algorithm solves the SSSP problem on graphs with *non-negative* arc weights. It builds the shortest-path tree greedily, extending it one vertex at a time—always choosing the nearest unvisited vertex.

10.3.1 Correctness Argument

Let T be the set of vertices whose shortest-path distance from s is known and final, with $d[v]$ denoting the distance for each $v \in T$. Initially $T = \{s\}$ and $d[s] = 0$.

Lemma 10.3.1 (Greedy extension)**.** *Let (v, w) be an arc with $v \in T$ and $w \notin T$ such that*

$$d[v] + c(v, w) \;=\; \min_{(i,j)\in\delta^+(T)} \big\{ d[i] + c(i, j) \big\}.$$

Then $d[w] = d[v] + c(v, w)$ is the shortest-path distance from s to w.

Proof. Consider any s–w path P. Since $s \in T$ and $w \notin T$, the path must cross the cut $\delta^+(T)$ at some arc (u, u'). Split P into: a subpath P' from s to u, the arc (u, u'), and a subpath P'' from u' to w. Then:

$$c(P) = c(P') + c(u, u') + c(P'') \;\geq\; d[u] + c(u, u') + 0 \;\geq\; d[v] + c(v, w),$$

where the first inequality uses $c(P') \geq d[u]$ (since $u \in T$) and $c(P'') \geq 0$ (non-negative weights), and the second uses the minimality of (v, w). Hence no path can be cheaper than $d[v] + c(v, w)$. □

Remark 10.3.2. This proof *crucially* relies on $c(P'') \geq 0$. If negative weights were allowed, the subpath P'' could have negative cost, invalidating the greedy choice. This is precisely why Dijkstra's algorithm requires non-negative weights.

10.3.2 Algorithm

Algorithm 6: Dijkstra's algorithm (single source, non-negative weights)

Input: Directed graph $G = (V, A)$ with $c_{ij} \geq 0$; source s
Output: Distance array $d[\cdot]$; predecessor array $\pi[\cdot]$

```
d[s] ← 0; d[v] ← +∞ for all v ≠ s
π[v] ← nil for all v ∈ V
T ← ∅                                   // set of finalised vertices
while T ≠ V do
    w ← arg min_{v∉T} d[v]              // select nearest
    T ← T ∪ {w}
    foreach arc (w, k) with k ∉ T do
        if d[w] + c_wk < d[k] then
            d[k] ← d[w] + c_wk          // relax
            π[k] ← w
```

10.3.3 Worked Example on the Running Graph

Example 10.3.3 (Dijkstra from s on the running graph)**.** We apply Dijkstra's algorithm to the running graph (fig. 10.1) with source s. Table 10.1 shows the state after each extraction. The final distances are: $d[s] = 0$, $d[b] = 2$, $d[a] = 4$, $d[d] = 5$, $d[c] = 9$, $d[e] = 11$, $d[t] = 12$. The shortest path to t is $s \to b \to d \to e \to t$ with cost $2 + 3 + 6 + 1 = 12$.

Step-by-step detail (continuing theorem 10.3.3):

Table 10.1: Dijkstra's algorithm trace on the running graph. At each step, the vertex w with minimum d-value is extracted and its neighbours are relaxed.

Step	Extract w	$d[s]$	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$	$d[t]$
Init	—	0	∞	∞	∞	∞	∞	∞
1	s (d=0)	0	4	2	∞	∞	∞	∞
2	b (d=2)	0	4	2	∞	5	∞	∞
3	a (d=4)	0	4	2	9	5	∞	∞
4	d (d=5)	0	4	2	9	5	11	14
5	c (d=9)	0	4	2	9	5	11	14
6	e (d=11)	0	4	2	9	5	11	12
7	t (d=12)	0	4	2	9	5	11	12

1. **Extract s ($d = 0$):** Relax $s \to a$: $d[a] = 0 + 4 = 4$; relax $s \to b$: $d[b] = 0 + 2 = 2$.
2. **Extract b ($d = 2$):** Relax $b \to a$: $2 + 8 = 10 > 4$, no update; relax $b \to d$: $d[d] = 2 + 3 = 5$.
3. **Extract a ($d = 4$):** Relax $a \to c$: $d[c] = 4 + 5 = 9$; relax $a \to d$: $4 + 1 = 5$, no update ($d[d]$ already 5).
4. **Extract d ($d = 5$):** Relax $d \to e$: $d[e] = 5 + 6 = 11$; relax $d \to t$: $d[t] = 5 + 9 = 14$.
5. **Extract c ($d = 9$):** Relax $c \to e$: $9 + 3 = 12 > 11$, no update; relax $c \to d$: $9 + 2 = 11 > 5$, no update.
6. **Extract e ($d = 11$):** Relax $e \to t$: $11 + 1 = 12 < 14$, update $d[t] = 12$.
7. **Extract t ($d = 12$):** no outgoing arcs. Done.

Shortest path to t: trace predecessors $t \leftarrow e \leftarrow d \leftarrow b \leftarrow s$, giving $s \to b \to d \to e \to t$ with cost $2 + 3 + 6 + 1 = 12$.

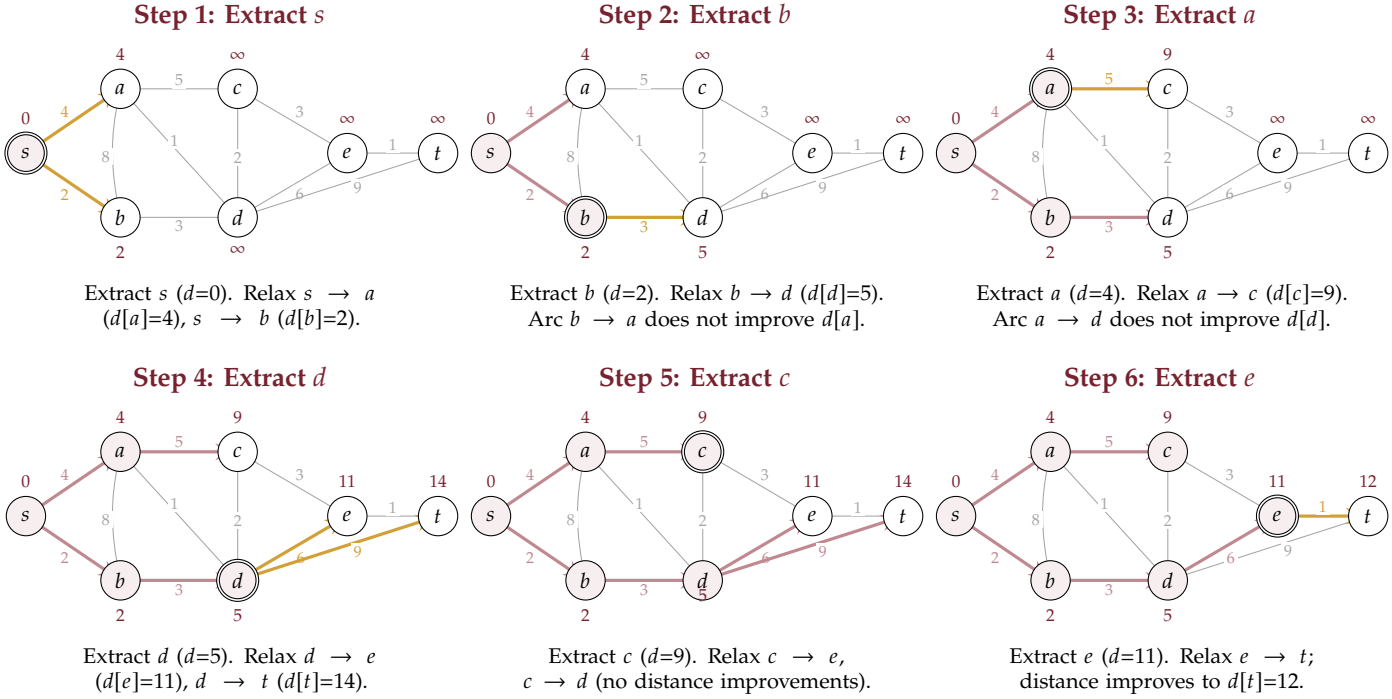


Figure 10.2: Dijkstra's algorithm execution on the directed version of the running graph, starting from s. Finalized vertices are shaded. At each step, the highlighted gold arc(s) show the relaxation(s) that successfully update distances. Total path cost to t: $2 + 3 + 6 + 1 = 12$.

10.3.4 Complexity

Proposition 10.3.4 (Dijkstra complexity)**.** *Dijkstra's algorithm runs in:*

- $O(n^2)$ *with a simple array (scanning all vertices for the minimum); ideal for* dense *graphs* ($m \approx n^2$)*.*
- $O(m \log n)$ *with a binary min-heap; ideal for* sparse *graphs* ($m \ll n^2$)*.*

Proof. In the array-based version, each of the n iterations requires an $O(n)$ scan to find the minimum, giving $O(n^2)$ total.

With a binary heap, extracting the minimum costs $O(\log n)$ per iteration (n extractions total: $O(n \log n)$), and each of the m relaxations may require a decrease-key operation at $O(\log n)$ cost. The total is $O((n+m)\log n) = O(m \log n)$ since the graph is connected. □

Remark 10.3.5 (BFS as a special case). When all arc costs equal 1, the shortest path from s minimises the number of arcs. In this case, a simple **breadth-first search** (BFS) using a FIFO queue finds all shortest-path distances in $O(n+m)$—no priority queue is needed. This follows from the BFS shortest-path property proved in chapter 8.

Intermezzo — Edsger W. Dijkstra (1930–2002)

Dijkstra published his shortest-path algorithm in a two-page paper in 1959. He reportedly conceived it in about twenty minutes while sitting at a café in Amsterdam. The algorithm became one of the most cited results in computer science. Dijkstra went on to make seminal contributions to structured programming, concurrent computing, and formal verification, earning the Turing Award in 1972.

10.4 Bellman–Ford Algorithm

Bellman–Ford trades speed for generality: it handles negative weights (but not negative cycles).

When arc weights may be negative (but no negative cycles exist), Dijkstra's greedy strategy fails because a longer path through a negative-weight arc might turn out cheaper. The **Bellman–Ford** algorithm handles this case by performing $n-1$ rounds of *relaxation* over all arcs.

10.4.1 Algorithm

Algorithm 7: Bellman–Ford algorithm

Input: Directed graph $G = (V, A)$ with weights c; source s
Output: Distance array $d[\cdot]$; predecessor array $\pi[\cdot]$; negative-cycle flag

```
1  d[s] ← 0; d[v] ← +∞ for all v ≠ s
2  π[v] ← nil for all v ∈ V
3  for i = 1 to n − 1 do
4      foreach arc (u, v) ∈ A do
5          if d[u] + c_uv < d[v] then                    // relax
6              d[v] ← d[u] + c_uv
7              π[v] ← u
   // Negative-cycle detection
8  foreach arc (u, v) ∈ A do
9      if d[u] + c_uv < d[v] then
10         return negative cycle detected
```

10.4.2 Correctness and Complexity

Theorem 10.4.1 (Bellman–Ford correctness)**.** *If G contains no negative-cost cycle reachable from s, then after $n - 1$ relaxation passes, $d[v]$ equals the shortest-path distance from s to v for every $v \in V$.*

Formal details — Bellman–Ford correctness proof

The proof proceeds by induction on the number of arcs in the shortest path.

Claim: after the k-th pass, $d[v]$ is at most the cost of the shortest path from s to v using at most k arcs.

Base case ($k = 0$): $d[s] = 0$ and $d[v] = +\infty$ for $v \neq s$. The only path with zero arcs is the empty path from s to itself, so the claim holds.

Inductive step: assume the claim holds after pass $k - 1$. Consider a shortest path P from s to v using exactly k arcs, with last arc (u, v). The subpath from s to u uses $k - 1$ arcs, so by the inductive hypothesis, $d[u]$ is at most its cost after pass $k - 1$. During pass k, the relaxation of arc (u, v) sets $d[v] \leq d[u] + c_{uv} \leq c(P)$.

Since any elementary shortest path uses at most $n - 1$ arcs, after $n - 1$ passes all shortest-path distances are correct.

Lower bound. For the lower bound, notice that $d[v]$ is never set to a value that is not the length of some actual s-v path in G (every relaxation follows a real arc). Therefore $d[v] \geq \delta(s, v)$ throughout, since $\delta(s, v)$ is the minimum over all such paths. Together with the upper-bound argument above, $d[v] = \delta(s, v)$ after $n - 1$ passes.

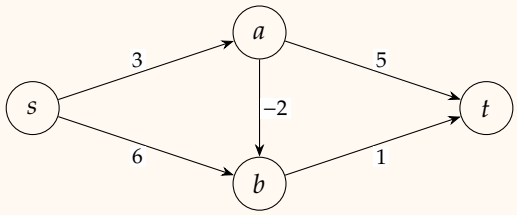
Proposition 10.4.2 (Bellman–Ford complexity)**.** *Bellman–Ford runs in $O(nm)$: $n - 1$ passes, each scanning all m arcs.*

The extra pass after the main loop detects negative cycles: if any arc can still be relaxed, some shortest path requires n or more arcs, which is only

possible if a negative cycle is reachable from s.

Example 10.4.3 (Bellman–Ford on the running graph)**.** On the running graph with source s, Bellman–Ford produces the same final distances as Dijkstra (since all weights are non-negative): $d[s] = 0$, $d[b] = 2$, $d[a] = 4$, $d[d] = 5$, $d[c] = 9$, $d[e] = 11$, $d[t] = 12$. However, it takes up to $n - 1 = 6$ full passes over all 11 arcs, rather than Dijkstra's more targeted approach. The extra pass confirms no negative cycle exists.

Example 10.4.4 (Bellman–Ford with a negative arc)**.** The following example shows why Dijkstra cannot handle negative arcs, and how Bellman-Ford succeeds. Consider the 4-node graph with nodes $\{s, a, b, t\}$ and arcs:



Why Dijkstra fails here. Dijkstra would finalise b at distance 6 (directly from s) before discovering the cheaper path $s \to a \to b$ of cost $3 + (-2) = 1$. Since non-negative weights are required, Dijkstra's greedy choice is invalid on this graph.

Bellman–Ford trace (source s, processing arcs in the order $s{\to}a$, $s{\to}b$, $a{\to}b$, $b{\to}t$, $a{\to}t$):

Pass	Event	$d[s]$	$d[a]$	$d[b]$	$d[t]$
Init	—	0	∞	∞	∞
1	Relax all	0	3	1	8
2	Relax all	0	3	1	2
3	Relax all	0	3	1	2

After pass 1: $d[a] = 3$ (via $s{\to}a$), $d[b] = \min(6, 3{-}2) = 1$ (via $s{\to}a{\to}b$), $d[t] = \min(\infty, 3{+}5) = 8$ (via $s{\to}a{\to}t$). After pass 2: $d[t] = \min(8, 1{+}1) = 2$ (via $s{\to}a{\to}b{\to}t$). Passes 3 and beyond yield no further improvement.

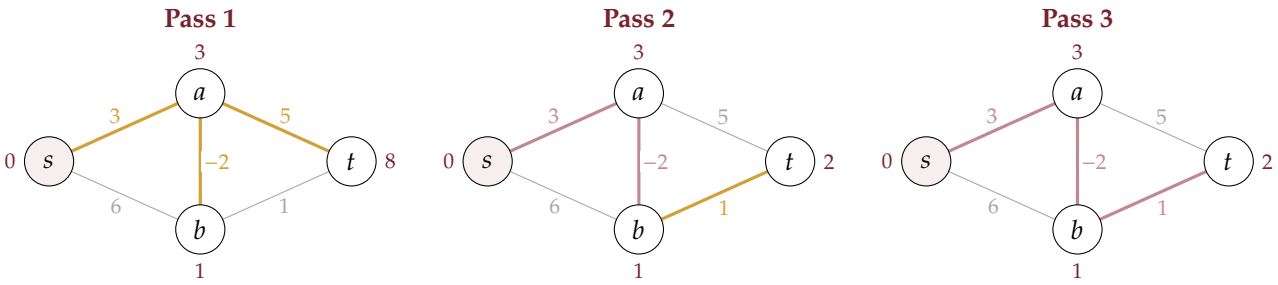
Final distances: $d[s] = 0$, $d[a] = 3$, $d[b] = 1$, $d[t] = 2$. The shortest path to t is $s \to a \to b \to t$ with cost $3 + (-2) + 1 = 2$.

Intermezzo — Richard Bellman (1920–1984) & Lester Ford Jr. (1927–2017)

Bellman, the father of *dynamic programming*, published the algorithm in 1958. Ford independently described a similar method. Their combined work provides the most general single-source shortest-path algorithm for graphs without negative cycles. Bellman's "principle of optimality"—that subpaths of shortest paths are themselves shortest paths—is the conceptual backbone of every DP-based shortest-path method.

10.5 Shortest Paths in DAGs and the CPM

DAGs allow both shortest and *longest paths in linear time.*



$d[a]$=3 (via $s \to a$), $d[b]$=1 (via $a \to b$), and $d[t]$=8 (via $a \to t$).

$d[t]$ improves to 2 (via $b \to t$, using $d[b]$=1 from pass 1).

No distances change, indicating convergence to the SSSP tree.

Figure 10.3: Bellman–Ford algorithm passes on the negative arc instance. Gold arcs show improvements in the current pass; burgundy arcs show the resulting predecessor paths.

Since a DAG has no cycles whatsoever, negative-weight arcs pose no difficulty. Processing vertices in topological order (see section 8.5) allows a single-pass dynamic programming solution.

10.5.1 Algorithm

Algorithm 8: Shortest paths in a DAG

Input: DAG $G = (V, A)$ with weights c; source s
Output: Distance array $d[\cdot]$; predecessor array $\pi[\cdot]$

Compute a topological ordering $v_1, v_2, \ldots, v_n$ of V (see section 8.5)
$d[s] \leftarrow 0$; $d[v] \leftarrow +\infty$ for all $v \neq s$
$\pi[v] \leftarrow \texttt{nil}$ for all $v \in V$
for *each* v_i *in topological order* **do**
  foreach *arc* $(v_i, v_j) \in A$ **do**
    if $d[v_i] + c_{v_i v_j} < d[v_j]$ **then**
      $d[v_j] \leftarrow d[v_i] + c_{v_i v_j}$
      $\pi[v_j] \leftarrow v_i$

This is essentially the algorithm presented in section 8.6, now placed in the broader context of shortest-path methods. It runs in $O(n + m)$—one pass through every node and every arc.

Example 10.5.1 (Shortest paths in a DAG)**.** We apply the DAG shortest-path algorithm to the directed version of the running graph, which has no cycles and is therefore a DAG. We use the topological sort order s, b, a, c, d, e, t to process the nodes. Table 10.2 shows the state of the distance labels $d[v]$ and predecessor labels $\pi[v]$ after processing each node.

Notice that while the distance updates are identical to Dijkstra's algorithm, the order in which nodes are processed is determined purely by the topological sorting of the DAG, completely independent of the actual edge weights.

10.5.2 Longest Paths and Scheduling

Longest path in a DAG = critical path in project scheduling.

Table 10.2: DAG shortest-path algorithm trace on the running graph.

Step	Process v_i	$d[s]$	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$	$d[t]$
Init	—	0	∞	∞	∞	∞	∞	∞
1	s	0	4 (s)	2 (s)	∞	∞	∞	∞
2	b	0	4 (s)	2 (s)	∞	5 (b)	∞	∞
3	a	0	4 (s)	2 (s)	9 (a)	5 (b)	∞	∞
4	c	0	4 (s)	2 (s)	9 (a)	5 (b)	12 (c)	∞
5	d	0	4 (s)	2 (s)	9 (a)	5 (b)	11 (d)	14 (d)
6	e	0	4 (s)	2 (s)	9 (a)	5 (b)	11 (d)	12 (e)
7	t	0	4 (s)	2 (s)	9 (a)	5 (b)	11 (d)	12 (e)

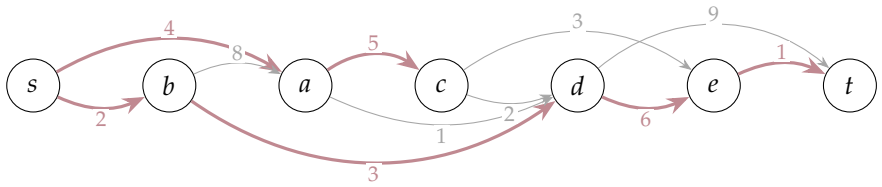


Figure 10.4: The directed running graph laid out in topological sorting order $s \to b \to a \to c \to d \to e \to t$. All arcs point strictly from left to right. Bold arcs form the shortest-path tree rooted at s.

Since a DAG has no cycles, the *longest* path problem is also tractable. Simply negate all arc weights and find the shortest path, or equivalently, replace min with max in the DP recurrence:

$$d_{\max}[v_j] \;=\; \max_{(v_i,v_j)\in A} \bigl\{ d_{\max}[v_i] + c_{v_i v_j} \bigr\}.$$

This leads directly to the **Critical Path Method (CPM)**, one of the most important applications of graph algorithms in operations research.

10.5.3 The Critical Path Method

Consider a project consisting of n **jobs** (tasks), each with a processing time $p_j > 0$ and subject to **precedence constraints**: job i must finish before job j can start. We model this as a DAG:

- Nodes represent jobs, plus a dummy source (start, $p = 0$) and a dummy sink (end, $p = 0$).
- An arc (i, j) with weight p_i means "job j cannot start until p_i time units after job i starts."

Definition 10.5.2 (Makespan, EST, LST, critical job)**.**

- The **makespan** M is the minimum total project duration, equal to the length of the longest path from the source to the sink.
- The **earliest starting time** $\mathrm{EST}[j]$ is the length of the longest path from the source to job j.
- The **latest starting time** $\mathrm{LST}[j]$ is the latest time job j can start without increasing the makespan: $\mathrm{LST}[j] = M - \lambda(j, \text{sink})$, where $\lambda(j, \text{sink})$ is the longest path from j to the sink.
- A job is **critical** if $\mathrm{EST}[j] = \mathrm{LST}[j]$ (zero slack).
- A **critical path** is a longest source–sink path; all jobs on it are critical.

Example 10.5.3 (CPM for a small project)**.** Consider a project with 6 jobs and the following data:

Table 10.3: Job data for the CPM example.

Job	Description	Duration p_j	Predecessors
A	Design	3	—
B	Procure parts	5	—
C	Build frame	4	A
D	Build engine	6	B
E	Assemble	3	C, D
F	Test	2	E

We construct a project DAG with dummy start node 0 and dummy end node 7 (both with $p = 0$).

Forward pass (EST):

$$\begin{aligned}
&\mathrm{EST}[0] = 0\\
&\mathrm{EST}[A] = 0, \qquad \mathrm{EST}[B] = 0\\
&\mathrm{EST}[C] = \mathrm{EST}[A] + p_A = 0 + 3 = 3\\
&\mathrm{EST}[D] = \mathrm{EST}[B] + p_B = 0 + 5 = 5\\
&\mathrm{EST}[E] = \max(\mathrm{EST}[C] + p_C,\ \mathrm{EST}[D] + p_D) = \max(3+4,\ 5+6) = \max(7, 11) = 11\\
&\mathrm{EST}[F] = \mathrm{EST}[E] + p_E = 11 + 3 = 14\\
&\mathrm{EST}[7] = \mathrm{EST}[F] + p_F = 14 + 2 = 16
\end{aligned}$$

The **makespan** is $M = 16$.

Backward pass (LST):

$$\begin{aligned}
\mathrm{LST}[7] &= 16\\
\mathrm{LST}[F] &= \mathrm{LST}[7] - p_F = 16 - 2 = 14\\
\mathrm{LST}[E] &= \mathrm{LST}[F] - p_E = 14 - 3 = 11\\
\mathrm{LST}[D] &= \mathrm{LST}[E] - p_D = 11 - 6 = 5\\
\mathrm{LST}[C] &= \mathrm{LST}[E] - p_C = 11 - 4 = 7\\
\mathrm{LST}[B] &= \mathrm{LST}[D] - p_B = 5 - 5 = 0\\
\mathrm{LST}[A] &= \mathrm{LST}[C] - p_A = 7 - 3 = 4
\end{aligned}$$

Slack and critical jobs:

The **critical path** is $B \to D \to E \to F$, with total duration 16. Delaying any of these jobs by even one time unit would increase the makespan. Jobs A and C have a slack of 4: they can be delayed by up to 4 units without affecting the project deadline.

10.6 Floyd–Warshall Algorithm

Floyd–Warshall: all-pairs shortest paths via dynamic programming in $O(n^3)$.

The algorithms above solve SSSP. For the **all-pairs shortest paths** (APSP) problem, we could run Dijkstra (or Bellman–Ford) from each vertex, but the **Floyd–Warshall** algorithm offers a simple, elegant $O(n^3)$ alternative based

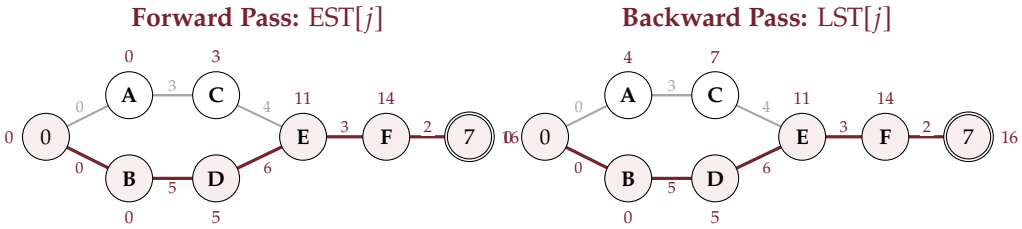


Figure 10.5: Forward and backward passes for the CPM example. Numbers next to nodes represent earliest starting times EST[j] (left) and latest starting times LST[j] (right). Critical nodes (zero slack) are shaded in burgundy; the critical path $0 \to B \to D \to E \to F \to 7$ determines the makespan of 16.

Table 10.4: EST, LST, and slack for each job. Critical jobs (zero slack) are highlighted.

Job	EST	LST	Slack	Critical?
A	0	4	4	No
B	0	0	0	**Yes**
C	3	7	4	No
D	5	5	0	**Yes**
E	11	11	0	**Yes**
F	14	14	0	**Yes**

on dynamic programming. It handles both positive and negative weights, provided no negative cycles exist.

10.6.1 DP Recurrence

Definition 10.6.1 (Floyd–Warshall recurrence)**.** Let $d_{ij}^{(k)}$ denote the length of the shortest path from i to j using only intermediate vertices from $\{1, 2, \ldots, k\}$.

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ c_{ij} & \text{if } (i,j) \in A, \\ +\infty & \text{otherwise.} \end{cases}$$

$$d_{ij}^{(k)} = \min\bigl(d_{ij}^{(k-1)},\; d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\bigr), \quad k = 1, \ldots, n.$$

After n iterations, $d_{ij}^{(n)}$ is the shortest-path distance from i to j.

This recurrence is visualised in the following two figures:

- fig. 10.6 shows a schematic view of the decision made at step k. We compare the shortest path from i to j that only uses intermediate nodes in $T = \{1, \ldots, k-1\}$ (Case 1) with the concatenated path that goes from i to k and then from k to j using only intermediate nodes in T (Case 2).
- fig. 10.7 shows a concrete example of this decision on a small graph.

The idea is simple: at each stage we ask, "does routing through vertex k improve the best known path from i to j?"

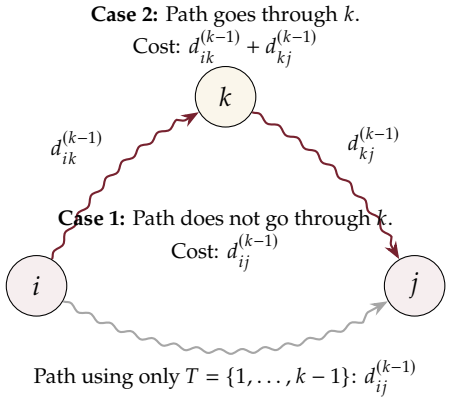


Figure 10.6: Schematic representation of the Floyd–Warshall dynamic programming decision at step k. We choose the minimum cost between not using k (Case 1) and using k as an intermediate node (Case 2).

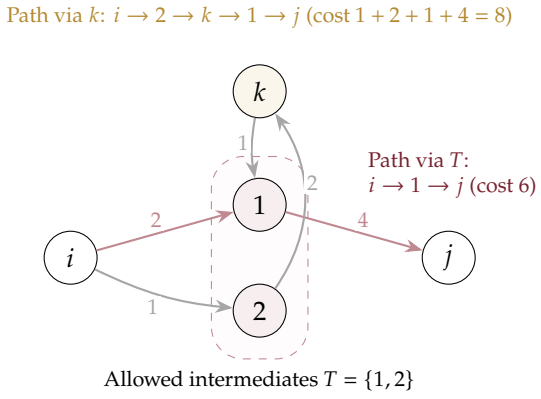


Figure 10.7: A concrete graph example illustrating the Floyd–Warshall DP step. The set of allowed intermediate vertices $T = \{1, 2\}$ is enclosed in the dashed bubble. When vertex k is added to the allowed set, we compare the shortest path using only $\{1, 2\}$ (cost 6) with the path using k (cost 8). In this case, routing through k does not improve the distance.

Algorithm 9: Floyd–Warshall algorithm

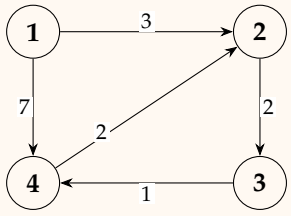
Input: Directed graph G with n vertices, weight matrix c
Output: Distance matrix D; predecessor matrix Π

```
// Initialise
for i = 1 to n do
    for j = 1 to n do
        if i = j then
            D[i][j] ← 0
        else
            if (i, j) ∈ A then
                D[i][j] ← c_ij;
                Π[i][j] ← i
            else
                D[i][j] ← +∞;
                Π[i][j] ← nil

// Main DP loop
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            if D[i][k] + D[k][j] < D[i][j] then
                D[i][j] ← D[i][k] + D[k][j]
                Π[i][j] ← Π[k][j]
```

10.6.2 Worked Example

Example 10.6.2 (Floyd–Warshall on a 4-node graph)**.** Consider the directed graph with 4 nodes and the following arcs:



Initial matrix $D^{(0)}$:

$$D^{(0)} = \begin{pmatrix} 0 & 3 & \infty & 7 \\ \infty & 0 & 2 & \infty \\ \infty & \infty & 0 & 1 \\ \infty & 2 & \infty & 0 \end{pmatrix}$$

After $k = 1$ (using vertex 1 as intermediate): No improvement, since vertex 1 has no incoming arcs from other vertices. $D^{(1)} = D^{(0)}$.

After $k = 2$ (using vertices $\{1, 2\}$):

- $D[1][3]$: $\min(\infty,\ 3 + 2) = 5$ (via $1 \to 2 \to 3$)
- $D[4][3]$: $\min(\infty,\ 2 + 2) = 4$ (via $4 \to 2 \to 3$)

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 5 & 7 \\ \infty & 0 & 2 & \infty \\ \infty & \infty & 0 & 1 \\ \infty & 2 & 4 & 0 \end{pmatrix}$$

After $k = 3$ (using vertices $\{1, 2, 3\}$):

- $D[1][4]$: $\min(7,\ 5 + 1) = 6$ (via $1 \to 2 \to 3 \to 4$)
- $D[2][4]$: $\min(\infty,\ 2 + 1) = 3$ (via $2 \to 3 \to 4$)
- $D[4][4]$: $\min(0,\ 4 + 1) = 0$ (no improvement)

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 5 & 6 \\ \infty & 0 & 2 & 3 \\ \infty & \infty & 0 & 1 \\ \infty & 2 & 4 & 0 \end{pmatrix}$$

After $k = 4$ (using all vertices):

- $D[1][2]$: $\min(3,\ 6 + 2) = 3$ (no improvement)
- $D[2][2]$: $\min(0,\ 3 + 2) = 0$ (no improvement)
- $D[3][2]$: $\min(\infty,\ 1 + 2) = 3$ (via $3 \to 4 \to 2$)
- $D[3][1]$: remains ∞ (no path)
- $D[2][1]$: remains ∞ (no path)

$$D^{(4)} = \begin{pmatrix} 0 & 3 & 5 & 6 \\ \infty & 0 & 2 & 3 \\ \infty & 3 & 0 & 1 \\ \infty & 2 & 4 & 0 \end{pmatrix}$$

The final matrix gives all-pairs shortest-path distances. For example, the shortest path from 1 to 4 costs 6, routed $1 \to 2 \to 3 \to 4$. See fig. 10.8 for a step-by-step visual trace of the algorithm's execution and path updates.

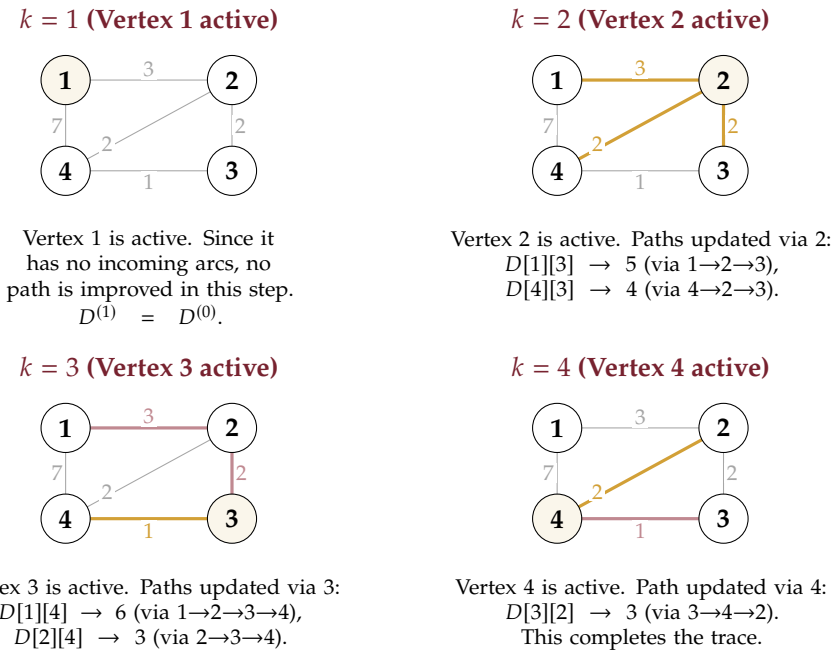


Figure 10.8: Step-by-step execution of the Floyd–Warshall algorithm on the 4-node directed graph. The active intermediate vertex k at each step is shaded in gold. Gold arcs highlight edge relaxations that improve paths in the current step; burgundy arcs show previously established predecessor paths that are used to build the new paths.

10.6.3 Negative Cycle Detection

After the algorithm completes, check the diagonal of D: if $D[i][i] < 0$ for any i, then a negative cycle involving vertex i exists. In a graph without negative cycles, $D[i][i] = 0$ for all i (the "shortest path" from i to itself is the empty path).

10.6.4 Space Optimisation

The DP recurrence formally requires n separate $n \times n$ matrices $D^{(0)}, D^{(1)}, \ldots, D^{(n)}$. However, one can update D *in place*: the calculation of $D_{ij}^{(k)}$ reads values $D_{ik}^{(k-1)}$ and $D_{kj}^{(k-1)}$. Since neither of these entries changes when k is the current intermediate vertex (the k-th row and k-th column remain invariant during iteration k), in-place update is correct. This reduces space from $O(n^3)$ to $O(n^2)$.

Proposition 10.6.3 (Floyd–Warshall complexity)**.** *Floyd–Warshall runs in $O(n^3)$ time and $O(n^2)$ space (with in-place update).*

Intermezzo — Robert Floyd (1936–2001) & Stephen Warshall (1935–2006)

Floyd published the all-pairs shortest-paths algorithm in 1962. Warshall independently described a closely related algorithm for computing transitive closures in 1962—the same DP scheme we saw in chapter 8. The two approaches are essentially the same recurrence applied to different semirings. Floyd received the Turing Award in 1978 for his contributions to programming language semantics and program verification.

10.7 Summary and Comparison

Remark 10.7.1 (Johnson's algorithm). When the graph is sparse ($m \ll n^2$), Floyd-Warshall's $O(n^3)$ cost is wasteful. Johnson's algorithm avoids this by cleverly reweighting arcs so that Dijkstra can be applied from each source. Concretely, **Johnson's algorithm** (1977) solves APSP in $O(nm + n^2 \log n)$ time—substantially better than Floyd–Warshall's $O(n^3)$ when $m = o(n^2)$. The idea is to run Bellman–Ford *once* from a new auxiliary source to compute node potentials $h[v]$, then reweight each arc by $c'_{ij} = c_{ij} + h[i] - h[j] \geq 0$. Because these reweighted costs are non-negative, Dijkstra can be run from each of the n sources on the reweighted graph; original distances are recovered by reversing the reweighting. The algorithm handles general graphs with no negative cycles, so it subsumes both Dijkstra (fast but non-negative weights only) and Floyd–Warshall (general but cubic).

Table 10.5 summarises the shortest-path algorithms discussed in this chapter.

How to choose.

- **All weights non-negative:** use Dijkstra (array-based for dense graphs, heap-based for sparse graphs).
- **Unit weights:** use BFS.

Table 10.5: Comparison of shortest-path algorithms.

Algorithm	Problem	Weights	Complexity	Neg. cycles?
BFS	SSSP	unit ($c = 1$)	$O(n + m)$	N/A
DAG DP	SSSP	any (DAG)	$O(n + m)$	impossible
Dijkstra	SSSP	$c \geq 0$	$O(n^2)$ or $O(m \log n)$	N/A
Bellman–Ford	SSSP	any	$O(nm)$	detects
Floyd–Warshall	APSP	any	$O(n^3)$	detects
Johnson	APSP	general, no neg. cyc.	$O(nm + n^2 \log n)$	N/A

- **DAG:** use topological-sort DP—it handles negative weights and runs in linear time.
- **Negative weights, no negative cycles:** use Bellman–Ford for SSSP.
- **All pairs needed:** use Floyd–Warshall; for sparse graphs use repeated Dijkstra when weights are non-negative, or Johnson's algorithm when negative edges but no negative cycles are present.
- **Negative cycles possible:** Bellman–Ford or Floyd–Warshall will *detect* them; a relevant negative cycle makes the corresponding distance unbounded below (the elementary variant is NP-hard).

Remark 10.7.2 (Looking ahead). The shortest-path problem is a special case of *minimum-cost network flow*: it corresponds to sending one unit of flow from s to t at minimum cost. We will see this connection in chapter 11, where the algorithmic ideas from this chapter generalise to much richer flow problems.

Summary & Key Takeaways

- **Dijkstra's Algorithm:** Finds shortest paths from a single source. Requires non-negative edge weights. Runs in $O(E + V \log V)$ time using a Fibonacci heap.
- **Bellman-Ford Algorithm:** Handles negative edge weights. Runs in $O(VE)$ time. Detects negative cycles (if a relaxation step succeeds in the V-th iteration).
- **DAG Shortest Paths:** Computes shortest paths on Directed Acyclic Graphs by relaxing edges according to a topological sort in $O(V + E)$ time.

Exercises

Exercise 1. Apply Dijkstra's algorithm to the following directed graph with source node $s = 1$. Nodes: $\{1, 2, 3, 4, 5\}$.

Arc	w
$(1, 2)$	2
$(1, 3)$	5
$(2, 3)$	1
$(2, 4)$	7
$(3, 4)$	3
$(3, 5)$	6
$(4, 5)$	1

For each iteration state the node extracted from the priority queue, its definitive distance, and the updated tentative distances of its neighbours. Report the final distance vector $d[1..5]$.

Exercise 2. Apply Dijkstra's algorithm with source $s = A$ to the directed graph with nodes $\{A, B, C, D, E, F\}$ and arcs:

Arc	w
(A, B)	4
(A, C)	2
(C, B)	1
(B, D)	5
(C, D)	8
(C, E)	10
(B, E)	2
(D, F)	2
(E, F)	3

Trace every iteration (extracted node, current distances). Give the shortest-path tree by listing the predecessor of each node.

Exercise 3. Consider the directed graph with nodes $\{1, 2, 3, 4, 5, 6\}$ and weighted arcs:

Arc	w
$(1, 2)$	7
$(1, 3)$	9
$(1, 6)$	14
$(2, 3)$	10
$(2, 4)$	15
$(3, 4)$	11
$(3, 6)$	2
$(4, 5)$	6
$(5, 6)$	9
$(6, 5)$	9

Run Dijkstra from source 1. After the algorithm terminates, list the shortest path (sequence of nodes) from 1 to 5 and its total cost.

Exercise 4. Run Dijkstra from source $s = 1$ on the directed graph with nodes $\{1, 2, 3, 4, 5\}$ and arcs:

Arc	w
$(1, 2)$	10
$(1, 3)$	3
$(3, 2)$	4
$(3, 4)$	8
$(2, 4)$	1
$(2, 5)$	2
$(4, 5)$	7

After the algorithm terminates, give the predecessor array $\pi[1..5]$ that encodes the shortest-path tree rooted at 1. Use $\pi[s] = \text{NIL}$.

Exercise 5. Apply Dijkstra from source A to the directed graph with nodes $\{A, B, C, D, E\}$ and arcs:

Arc	w
(A, B)	1
(A, C)	4
(B, C)	2
(B, D)	5
(C, D)	1
(C, E)	3
(D, E)	2

Determine which arcs belong to the shortest-path tree. Draw (or describe) the tree and verify that $d[v] = d[u] + c(u, v)$ holds for every tree arc (u, v).

Exercise 6. After running Dijkstra from source s on a directed graph you obtain the distance vector $d = [0, 3, 7, 5, 10]$ and predecessor array $\pi = [\text{NIL}, 1, 4, 2, 3]$ (1-indexed).

(a) Reconstruct the shortest path from $s = 1$ to node 5.

(b) Verify the optimality condition $c_{ij} - d[j] + d[i] \geq 0$ (reduced cost $\bar{c}_{ij} \geq 0$) for at least three arcs of your choice.

Exercise 7 (Dijkstra correctness). Prove that when Dijkstra's algorithm extracts a node u from the priority queue, the tentative distance $d[u]$ is already the true shortest-path distance $\delta(s, u)$. Your proof should use the fact that all arc weights are non-negative and reason about any hypothetical shorter path through an unvisited node.

Exercise 8. Consider the following claim: "Dijkstra's algorithm (with a binary heap) runs in $O(n^2)$ time on any directed graph with n nodes and m arcs." Is the claim correct? Justify your answer by stating the correct worst-case complexity and explaining the role of m.

Exercise 9 (True or False). For each statement below, state whether it is **true** or **false** and give a brief justification (one or two sentences, or a counterexample).

(a) Dijkstra's algorithm produces correct shortest paths on any directed graph that has negative arcs but no negative-weight cycle.

(b) Bellman–Ford has worst-case time complexity $O(n^2)$ on a graph with n nodes and m arcs.

(c) Floyd–Warshall can detect the presence of a negative-weight cycle in a directed graph.

(d) On a DAG the shortest-path problem can be solved in $O(n + m)$ time even when some arc weights are negative.

(e) If all arc weights are non-negative, Bellman–Ford gives the same distance vector as Dijkstra.

Exercise 10. Apply the Bellman–Ford algorithm with source $s = 1$ to the directed graph with nodes $\{1, 2, 3, 4, 5\}$ and arcs:

Arc	w
$(1,2)$	6
$(1,3)$	7
$(2,3)$	8
$(2,4)$	-4
$(2,5)$	5
$(3,4)$	9
$(4,1)$	2
$(4,5)$	7
$(5,4)$	4

There is no negative cycle. Trace the distance array d after each of the $n-1=4$ relaxation passes. Report the final distances and predecessor array.

Exercise 11. Run Bellman–Ford from source $s = A$ on the directed graph with nodes $\{A, B, C, D\}$ and arcs: $(A, B, 1)$, $(A, C, 4)$, $(B, C, -3)$, $(B, D, 2)$, $(C, D, 3)$. Show the distance array after each relaxation pass. What is the shortest path from A to D and its cost?

Exercise 12. Apply Bellman–Ford with source 1 to the directed graph with nodes $\{1, 2, 3, 4, 5, 6\}$ and arcs:

Arc	w
$(1,2)$	2
$(1,4)$	1
$(2,3)$	-3
$(3,5)$	1
$(4,2)$	4
$(4,5)$	2
$(5,6)$	1
$(6,3)$	-2

There is no negative cycle. After the algorithm terminates, list the complete shortest-path tree by giving, for each node, its predecessor and its distance from 1.

Exercise 13. Given the directed graph with nodes $\{1, 2, 3, 4\}$ and arcs: $(1, 2, 3)$, $(2, 3, 4)$, $(3, 4, -2)$, $(4, 2, -5)$, $(1, 4, 10)$.

(a) Identify the negative cycle by inspection (state the cycle and compute its total weight).

(b) Describe how Bellman–Ford detects this cycle: what happens when you attempt an extra (n-th) relaxation pass after the standard $n-1$ passes?

Exercise 14. Apply Bellman–Ford from source $s = 1$ to the graph with nodes $\{1, 2, 3, 4, 5\}$ and arcs:

Arc	w
$(1,2)$	1
$(2,3)$	2
$(3,4)$	-4
$(4,2)$	1
$(4,5)$	3
$(1,5)$	10

(a) Run $n - 1 = 4$ relaxation passes and record the distance array after each pass.

(b) Perform a 5th relaxation pass and explain which distance value decreases, thereby confirming the presence of a negative cycle.

(c) Identify the negative cycle and compute its total weight.

Exercise 15. After running Floyd–Warshall on a directed graph with $n = 4$ nodes you obtain the following distance matrix D:

$$D = \begin{pmatrix} 0 & 3 & 5 & 2 \\ 7 & 0 & 2 & -1 \\ 4 & -3 & 0 & -4 \\ 6 & 1 & 3 & 0 \end{pmatrix}.$$

(a) Explain the rule used to detect a negative cycle from the diagonal of D.

(b) Apply the rule: does this matrix indicate a negative cycle? Justify.

Exercise 16. The following directed acyclic graph has nodes $\{1, 2, 3, 4, 5, 6\}$ with a topological order $1, 2, 3, 4, 5, 6$.

Arc	w
$(1, 2)$	3
$(1, 3)$	6
$(2, 3)$	4
$(2, 4)$	4
$(2, 5)$	11
$(3, 4)$	8
$(3, 5)$	5
$(4, 5)$	1
$(4, 6)$	7
$(5, 6)$	2

Apply the DAG shortest-path dynamic programming algorithm from source 1 (process nodes in topological order). Show the distance array after processing each node and report the final shortest distances.

Exercise 17. Consider the DAG with nodes $\{A, B, C, D, E\}$ in topological order A, B, C, D, E and arcs:

Arc	w
(A, B)	2
(A, C)	5
(B, C)	1
(B, D)	4
(C, D)	-2
(C, E)	3
(D, E)	2

(a) Find shortest distances from A to all other nodes using the DP recurrence $d[v] = \min_{(u,v)\in A}\{d[u] + c(u, v)\}$.

(b) Verify that Dijkstra *cannot* be directly applied here (state which arc causes the problem) and explain why the DAG-DP approach works despite the negative arc.

Exercise 18. Apply the DAG shortest-path algorithm from source $s = 1$ to the DAG with nodes $\{1, 2, 3, 4, 5\}$ in topological order $1, 2, 3, 4, 5$ and arcs:

Arc	w
$(1, 2)$	4
$(1, 3)$	2
$(2, 4)$	3
$(3, 2)$	-1
$(3, 4)$	5
$(3, 5)$	7
$(4, 5)$	1

Report the distance vector and shortest-path tree (predecessors).

Exercise 19. Define the *longest-path* problem on a DAG (i.e., find the maximum-weight path from a source to every other node).

(a) Explain how to reduce longest-path on a DAG to shortest-path on a DAG by negating all arc weights.

(b) Apply this approach to the DAG with nodes $\{1, 2, 3, 4\}$, topological order $1, 2, 3, 4$, and arcs: $(1, 2, 3)$, $(1, 3, 1)$, $(2, 3, 2)$, $(2, 4, 5)$, $(3, 4, 4)$. Find the longest path from 1 to 4 and its length.

Exercise 20. Given the DAG with nodes $\{A, B, C, D, E, F\}$ in topological order A, B, C, D, E, F and arc weights:

Arc	w
(A, B)	3
(A, C)	1
(B, D)	2
(B, E)	4
(C, D)	5
(C, E)	1
(D, F)	3
(E, F)	2

Find the longest path from A to F using the max-DP recurrence $L[v] = \max_{(u,v)\in A}\{L[u] + c(u, v)\}$ with $L[A] = 0$. State the longest path length and the corresponding path.

Exercise 21 (CPM — Basic project network). A project consists of 7 activities with the following durations and precedence constraints (activity: duration, must follow): A: 3, –; B: 5, –; C: 2, A; D: 4, B; E: 6, A, B; F: 3, C, D; G: 2, E, F.

(a) Draw the project network (activity-on-arc or activity-on-node, state your choice).

(b) Compute the earliest start time for each activity using a forward pass.

(c) Identify the critical path and state the total project duration.

Exercise 22 (CPM — Forward and backward pass). Consider the project network with activities $A(4)$, $B(3)$, $C(6)$, $D(2)$, $E(5)$, $F(4)$, $G(3)$ and precedence constraints: $A \to C$, $A \to D$, $B \to D$, $B \to E$, $C \to F$, $D \to F$, $E \to G$, $F \to G$.

(a) Compute earliest start times (forward pass).

(b) Compute latest start times (backward pass).

(c) Compute total float (slack) for each activity.

(d) List all critical activities (zero float) and state the critical path.

Exercise 23 (CPM as longest path). Explain why the Critical Path Method is equivalent to solving a longest-path problem on a DAG. Define precisely what the nodes and arcs represent in the project network (activity-on-arc model), and state the recurrence used in the forward pass. Why is this DAG guaranteed to be acyclic in any well-defined project?

Exercise 24 (CPM — Float and resource management). A project has six activities with durations and dependencies as follows: $A(2)$: start activity; $B(4)$: after A; $C(3)$: after A; $D(5)$: after B; $E(2)$: after C; $F(3)$: after D, after E.

(a) Find the critical path and project duration.

(b) Which non-critical activities have the largest float?

(c) If activity C is delayed by 2 time units, does the project completion date change? Justify using your float computation.

Exercise 25 (Floyd-Warshall — Full trace on 4×4 matrix). Apply the Floyd–Warshall algorithm to the directed graph with nodes $\{1, 2, 3, 4\}$ and the following arc weights (∞ means no arc):

$$W = \begin{pmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{pmatrix}.$$

Compute the intermediate matrices $D^{(0)}, D^{(1)}, D^{(2)}, D^{(3)}, D^{(4)}$, showing which entries change at each step. Report the final all-pairs distance matrix.

Exercise 26 (Floyd-Warshall — Intermediate node argument). State and prove the correctness of the Floyd–Warshall recurrence

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right).$$

In particular, explain why using $d_{ik}^{(k-1)}$ and $d_{kj}^{(k-1)}$ (rather than the updated values at step k) is safe.

Exercise 27 (Floyd-Warshall — Path reconstruction). Extend the Floyd–Warshall algorithm with a predecessor matrix Π, where $\Pi[i][j]$ stores the last intermediate node on the current best path from i to j.

(a) State the update rule for Π within the algorithm.

(b) Write pseudocode for a ReconstructPath(i, j) procedure that uses Π to print the shortest path from i to j.

Exercise 28 (Floyd-Warshall on a 4×4 graph with a negative arc). Apply Floyd–Warshall to the graph with nodes $\{1, 2, 3, 4\}$ and arcs:

Arc	w
$(1, 2)$	2
$(2, 3)$	3
$(3, 1)$	4
$(1, 4)$	8
$(2, 4)$	-1
$(3, 4)$	5

There is no negative cycle. Show the distance matrix after each intermediate-node iteration and verify that no diagonal entry becomes negative.

Exercise 29 (ILP formulation for SSSP). Let $G = (V, A)$ be a directed graph with arc costs c_{ij} and a source $s \in V$. Write a binary integer program for the SSSP problem (find a minimum-cost s-t path) using flow variables $x_{ij} \in \{0, 1\}$ for each arc $(i, j) \in A$.

(a) Write the objective function and the flow-conservation constraints.

(b) Identify the coefficient matrix A of the constraint system.

(c) Explain why A is Totally Unimodular (TUM) and state what consequence this has for the LP relaxation.

Exercise 30 (TUM and integrality). Recall that a matrix is *Totally Unimodular* (TUM) if every square submatrix has determinant 0, $+1$, or -1.

(a) State the sufficient condition for TUM in terms of $\{0, \pm 1\}$ matrices with at most one $+1$ and one -1 per column (node–arc incidence matrix of a directed graph).

(b) Prove that the node–arc incidence matrix of any directed graph satisfies this condition.

(c) Conclude that the LP relaxation of the shortest-path ILP always has an integral optimal solution.

Exercise 31 (Reduced costs and optimality). Let G be a directed graph with arc costs c_{ij} and let $d : V \to \mathbb{R}$ be a vector of node potentials (distances from a source). The *reduced cost* of arc (i, j) is defined as $\bar{c}_{ij} = c_{ij} - d[j] + d[i]$.

(a) State the optimality condition for shortest paths in terms of reduced costs.

(b) Consider the directed graph with nodes $\{1, 2, 3, 4\}$, arcs $(1, 2, 3)$, $(1, 3, 5)$, $(2, 4, 2)$, $(3, 4, 1)$, $(2, 3, 1)$, and the distance vector $d = [0, 3, 4, 5]$ from source 1. Compute $\bar{c}_{ij}$ for all arcs and verify that the optimality conditions hold.

Exercise 32 (Optimality conditions: counterexample). Consider the directed graph with nodes $\{1, 2, 3\}$, arcs $(1, 2, 4)$, $(1, 3, 2)$, $(2, 3, 1)$, and the proposed distance vector $d = [0, 4, 4]$ from source 1.

(a) Compute the reduced cost of each arc.

(b) Identify which arc has a negative reduced cost and explain what that tells you about d.

(c) Correct the distance vector so that all reduced costs are non-negative and state the true shortest distances.

Exercise 33 (Complexity comparison). Fill in the table below and answer the questions that follow.

Algorithm	Problem	Negative weights	Negative cycle	Complexity
Dijkstra (heap)	SSSP	?	?	?
Bellman–Ford	SSSP	?	?	?
DAG-DP	SSSP	?	N/A	?
Floyd–Warshall	APSP	?	?	?

(a) For which algorithm is there the largest asymptotic difference between a dense graph ($m = \Theta(n^2)$) and a sparse graph ($m = \Theta(n)$)? Quantify the difference.

(b) When $m = O(n \log n)$, which SSSP algorithm is fastest?

(c) If $n = 1000$ and $m = 500\,000$, estimate the number of operations for Dijkstra (heap) vs Bellman–Ford.

Exercise 34 (Dense vs sparse). Compare the complexities of Dijkstra implemented with (i) a simple array (*array Dijkstra*, $O(n^2)$) and (ii) a binary heap ($O((n + m) \log n)$).

(a) For which regime ($m \ll n^2$ or $m \approx n^2$) is each implementation preferable? Give a formal argument.

(b) Show that for $m = n^2/\log n$ both implementations have the same asymptotic complexity.

Exercise 35. Explain the connection between the Bellman–Ford algorithm and dynamic programming. Specifically, define the subproblem $d^{(k)}[v]$ as the length of a shortest path from source s to v using at most k arcs, and state the recurrence. Show that after $n - 1$ iterations the values $d^{(n-1)}[v]$ equal the true shortest distances (assuming no negative cycle).

Exercise 36 (Shortest path with at most k arcs). Given a directed graph with $n = 5$ nodes, source $s = 1$, and arcs:

Arc	w
$(1, 2)$	1
$(1, 3)$	4
$(2, 3)$	2
$(2, 4)$	5
$(3, 4)$	1
$(3, 5)$	3
$(4, 5)$	2

(a) Compute $d^{(k)}[v]$ for $k = 1, 2, 3, 4$ using the Bellman–Ford DP formulation.

(b) Verify that $d^{(4)}$ equals the true shortest distances.

(c) State, for each node, the minimum number of arcs required to achieve the shortest distance.

Exercise 37 (Shortest vs longest path complexity).

(a) Explain why the shortest-path problem on a general directed graph (with no negative cycles) is solvable in polynomial time.

(b) Explain why the longest *simple* path problem on a general directed graph is NP-hard (hint: reduction from Hamiltonian path).

(c) Why does the longest-path problem become easy when the graph is a DAG? What structural property makes DP applicable?

Exercise 38 (Negative arc weights and Dijkstra failure). Construct a small directed graph with 3 nodes and at least one negative arc (but no negative cycle) on which Dijkstra's algorithm produces an incorrect result.

(a) Describe the graph (nodes, arcs, weights).

(b) Trace Dijkstra's execution and show the incorrect distance it computes.

(c) Explain why the greedy argument breaks down in the presence of a negative arc.

Exercise 39 (Re-weighting and Johnson's idea). Johnson's algorithm makes all arc weights non-negative by using node potentials, so that Dijkstra can be run from every source.

(a) Given potentials $h : V \to \mathbb{R}$, define the re-weighted cost $\hat{c}_{ij} = c_{ij} + h[i] - h[j]$ and show that $\hat{c}_{ij} \geq 0$ if and only if h satisfies the shortest-path optimality conditions (i.e., h is a feasible potential).

(b) Explain how Bellman–Ford is used once to compute the potentials h.

(c) State the total complexity of Johnson's algorithm for computing all-pairs shortest paths on a sparse graph and compare it with Floyd–Warshall.

Exercise 40 (Dijkstra on an undirected graph). An undirected graph can be treated as a directed graph by replacing each undirected edge $\{u, v\}$ with two directed arcs (u, v) and (v, u), both with the same weight.

(a) Apply Dijkstra from source 1 to the undirected graph with nodes $\{1, 2, 3, 4, 5\}$ and edges: $\{1, 2\} = 4$, $\{1, 3\} = 2$, $\{2, 3\} = 1$, $\{2, 4\} = 5$, $\{3, 4\} = 8$, $\{3, 5\} = 10$, $\{4, 5\} = 2$.

(b) Show the shortest-path tree and the distance vector.

(c) Does Dijkstra on the directed version of an undirected graph always give the same result as running it directly on the undirected graph? Justify.

Exercise 41 (SSSP uniqueness).

(a) Give a directed graph whose arc weights are all distinct but which has two different shortest paths from a source s to some vertex. Conclude that distinct arc weights do not imply a unique shortest-path tree.

(b) Prove that the shortest-path tree rooted at s is unique if every reachable vertex $v \neq s$ has a unique shortest s–v path.

(c) Describe a deterministic predecessor tie-breaking rule that makes an algorithm return one reproducible shortest-path tree when several shortest paths exist.

Exercise 42 (Sensitivity analysis). Let $d[v]$ be the shortest distances from s computed by Dijkstra on a directed graph G with non-negative weights. Suppose the weight of a single arc (u, v) is *decreased* from c_{uv} to $c_{uv} - \delta$ for some $\delta > 0$.

(a) Under what condition does the distance vector d remain optimal after this decrease? Express the condition in terms of reduced costs.

(b) Under what condition does the distance $d[v]$ change, and by how much?

(c) Illustrate with a concrete graph of your choice (4 or 5 nodes, state all arcs).

Exercise 43 (Floyd-Warshall and transitive closure). The Floyd–Warshall algorithm can be adapted to compute the *transitive closure* of a directed graph: a matrix T where $T[i][j] = 1$ if there is a path from i to j (and 0 otherwise).

(a) State the modified recurrence using Boolean operations ($\vee$ and $\wedge$) instead of min and +.

(b) Apply this algorithm to the directed graph with nodes $\{1, 2, 3, 4\}$ and arcs $(1, 2)$, $(2, 3)$, $(3, 4)$, $(4, 2)$. Show the transitive closure matrix.

(c) State the time complexity and compare with computing reachability by BFS/DFS from each node.

Exercise 44 (Shortest path duality / complementary slackness). The LP relaxation of the shortest-path ILP has a natural dual whose variables are node potentials.

(a) Write the dual LP of the flow-based shortest-path LP (primal variables x_{ij} for each arc, dual variables π_i for each node).

(b) State the complementary slackness conditions.

(c) Interpret the complementary slackness conditions in terms of reduced costs: which arcs can carry flow in an optimal solution?

Exercise 45 (Dijkstra with a Fibonacci heap). Dijkstra's algorithm can be implemented with a Fibonacci heap to achieve $O(m + n \log n)$ time.

(a) Compare this bound with the binary-heap implementation $O((n + m) \log n)$: for which values of m (as a function of n) is the Fibonacci-heap version asymptotically faster?

(b) Identify the two key operations (DecreaseKey and ExtractMin) and state their amortised costs in a Fibonacci heap.

(c) Explain why the improvement matters for dense graphs in practice.

Exercise 46 (Bellman-Ford: early termination). The standard Bellman–Ford algorithm always performs exactly $n - 1$ relaxation passes. An improved version stops early if a full pass produces no change in any distance value.

(a) Prove that early termination is correct: if no distance changes in pass k, then no distance will change in any subsequent pass.

(b) Construct a family of graphs (parametrised by n) for which early termination saves a constant fraction of the passes.

(c) Does early termination change the worst-case complexity? Justify.

Exercise 47 (Dijkstra trace with tie-breaking). Apply Dijkstra from source $s = 1$ to the directed graph with nodes $\{1, 2, 3, 4, 5\}$ and arcs:

Arc	w
$(1, 2)$	4
$(1, 3)$	4
$(2, 4)$	3
$(3, 4)$	3
$(2, 5)$	7
$(4, 5)$	2
$(3, 5)$	8

(a) Trace the algorithm, noting any iteration where two nodes have equal tentative distances.

(b) Show that the final distance vector is the same regardless of which tied node is extracted first.

(c) Give two distinct shortest-path trees that the algorithm may produce depending on tie-breaking, and verify both are valid.

Exercise 48 (Bellman-Ford on a complete graph). Consider the complete directed graph on nodes $\{1, 2, 3, 4\}$ where the weight of arc (i, j) is $|i - j|$ (absolute difference).

(a) List all arcs and their weights.

(b) Run Bellman–Ford from source 1 for $n - 1 = 3$ passes. Show the distance array after each pass.

(c) Are there any negative cycles? Verify using the n-th pass criterion.

Exercise 49 (DAG DP: counting paths). Consider the DAG with nodes $\{1, 2, 3, 4, 5\}$ in topological order $1, 2, 3, 4, 5$ and arcs: $(1, 2)$, $(1, 3)$, $(2, 3)$, $(2, 4)$, $(3, 4)$, $(3, 5)$, $(4, 5)$.

(a) Using a DP similar to shortest-path (but with + instead of min), compute the number of distinct paths from node 1 to each other node.

(b) What is the total number of paths from 1 to 5?

(c) Modify the recurrence to count only paths of *even* total number of arcs.

Exercise 50 (CPM: crashing activities). A project has four activities $A(6)$, $B(4)$, $C(5)$, $D(3)$ with dependencies $A \to C$, $A \to D$, $B \to C$, $C \to$ end, $D \to$ end. The critical path has duration 11. Each activity can be crashed (duration reduced) at a cost per unit time: A: \$3, B: \$2, C: \$4, D: \$1.

(a) Identify the critical path.

(b) By how much can you reduce the project duration by crashing a single activity at minimum cost? Which activity do you crash?

(c) If the target duration is 9 time units, what is the minimum total crashing cost?

Exercise 51 (Negative cycle: weight computation). A directed graph has nodes $\{1, 2, 3, 4, 5\}$ and arcs:

Arc	w
$(1, 2)$	5
$(2, 3)$	3
$(3, 4)$	-2
$(4, 5)$	-4
$(5, 3)$	2
$(1, 5)$	9
$(2, 4)$	6

(a) Identify all cycles in the graph (list each cycle as a sequence of nodes).

(b) Compute the total weight of each cycle.

(c) Which cycle(s), if any, are negative?

(d) Explain how Bellman–Ford would flag this graph.

Exercise 52 (Floyd-Warshall: asymmetric graph). Apply Floyd–Warshall to the directed graph with nodes $\{1, 2, 3, 4\}$ and arcs (note: some arcs exist only in one direction):

Arc	w
$(1, 2)$	1
$(2, 1)$	4
$(1, 3)$	3
$(3, 2)$	2
$(2, 4)$	5
$(3, 4)$	1
$(4, 1)$	2

(a) Show the initial distance matrix $D^{(0)}$ (use ∞ for missing arcs, 0 on diagonal).

(b) Compute $D^{(1)}$ and $D^{(2)}$ explicitly.

(c) Report the final all-pairs distance matrix $D^{(4)}$.

(d) Is there a pair (i, j) for which $d_{ij} \neq d_{ji}$?

Exercise 53 (ILP and LP relaxation on a small graph). Consider the directed graph with nodes $\{s, 1, 2, t\}$ and arcs $(s, 1, 2)$, $(s, 2, 5)$, $(1, t, 3)$, $(2, t, 1)$, $(1, 2, 1)$. Write the ILP for the shortest s-t path:

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij}, \quad x_{ij} \in \{0, 1\},$$

subject to the flow-conservation constraints.

(a) Write out all constraints explicitly.

(b) Solve the LP relaxation (allow $x_{ij} \in [0, 1]$) and verify the optimal LP solution is integral.

(c) State the shortest path and its cost.

Exercise 54 (Shortest path on a grid graph). A 3×3 grid has nodes (i, j) for $i, j \in \{1, 2, 3\}$. Directed arcs go only rightward or downward: from (i, j) to $(i, j + 1)$ with weight $i + j$, and from (i, j) to $(i + 1, j)$ with weight $i \cdot j$.

(a) This grid graph is a DAG. State a valid topological order.

(b) Apply the DAG shortest-path DP from source $(1, 1)$ to find the shortest path to $(3, 3)$.

(c) How many distinct paths from $(1, 1)$ to $(3, 3)$ exist in this grid (moving only right or down)?

Exercise 55 (Bellman-Ford vs Dijkstra on the same instance). Consider the directed graph with nodes $\{1, 2, 3, 4, 5\}$ and arcs:

Arc	w
$(1, 2)$	6
$(1, 3)$	4
$(3, 2)$	1
$(2, 4)$	5
$(3, 4)$	9
$(4, 5)$	3
$(2, 5)$	2

All weights are non-negative.

(a) Run Dijkstra from source 1 and record the order in which nodes are finalized together with their distances.

(b) Run Bellman–Ford from source 1 for $n - 1 = 4$ passes; record the distance array after each pass.

(c) Verify that both algorithms produce the same final distance vector $d[1..5]$.

(d) Count the total number of arc relaxations performed by each algorithm and relate the counts to the theoretical complexity bounds.

CHAPTER 11

Network Flows

Network flow models lie at the heart of logistics, telecommunications, and transportation planning.

In chapter 10 we studied the problem of finding cheapest paths in a weighted graph. We now move to a richer family of problems: *how much* of a commodity can we push through a network, and at what cost? This is the realm of **network flow theory**, which models the movement of goods, data, or fluids through a capacitated directed graph.

The importance of network flows can hardly be overstated. Every time a logistics company routes parcels, a telecommunications provider allocates bandwidth, or a water utility manages its distribution system, a flow problem is being solved—often many times per second. The theory is also remarkable from a structural perspective: flow problems are among the few combinatorial optimisation problems that can be solved in polynomial time, thanks to the total unimodularity of the underlying constraint matrix (already studied in chapter 7).

Road map.

1. Flow network definitions: capacity, flow, divergence, and circulations (section 11.1).
2. Flow problems: max-flow and min-cost flow (section 11.2).
3. ILP formulation and total unimodularity (section 11.3).
4. The Max-Flow Min-Cut Theorem (section 11.4).
5. The Ford–Fulkerson algorithm (section 11.5).
6. Complexity analysis and the Edmonds–Karp improvement (section 11.6).
7. Summary and comparison (section 11.7).

11.1 Flow Networks

11.1.1 Definitions

A flow network adds capacity *and* cost *information on top of a directed graph.*

Definition 11.1.1 (Flow network)**.** A **flow network** is a triple (G, k, c) where $G = (V, A)$ is a directed graph, $k \colon A \to \mathbb{R}_{\geq 0}$ assigns a **capacity** k_{ij} to each arc $(i, j) \in A$, and $c \colon A \to \mathbb{R}$ assigns a **cost** c_{ij} (per unit of flow) to each arc.

Definition 11.1.2 (Feasible flow)**.** A **flow** (or **exchange**) in the network

(G, k, c) is a vector $x \in \mathbb{R}^{|A|}$ satisfying the **capacity constraints**:

$$0 \;\le\; x_{ij} \;\le\; k_{ij} \qquad \forall (i,j) \in A.$$

A flow that satisfies these bounds is called **feasible** (or **admissible**). An arc with $x_{ij} = k_{ij}$ is **saturated**; an arc with $x_{ij} = 0$ is **empty**.

11.1.2 Divergence

Definition 11.1.3 (Divergence)**.** The **divergence** of a flow x at a node v is

$$\Delta_x(v) \;=\; \sum_{(v,j)\in\delta^+(v)} x_{vj} - \sum_{(i,v)\in\delta^-(v)} x_{iv},$$

where $\delta^+(v)$ and $\delta^-(v)$ denote the sets of outgoing and incoming arcs of v, respectively. A node with $\Delta_x(v) > 0$ is a **source**, a node with $\Delta_x(v) < 0$ is a **sink**, and a node with $\Delta_x(v) = 0$ is a **conservation node**.

Theorem 11.1.4 (Divergence theorem)**.** *For any feasible flow x in a network $G = (V, A)$,*

$$\sum_{v\in V} \Delta_x(v) \;=\; 0.$$

Proof. Each arc (i, j) contributes $+x_{ij}$ to $\Delta_x(i)$ and $-x_{ij}$ to $\Delta_x(j)$. Summing over all nodes, each arc flow cancels, giving zero. □

The divergence theorem says "no flow is created or destroyed in the network."

11.1.3 Circulations and s-t Flows

Definition 11.1.5 (Circulation)**.** A feasible flow x is a **circulation** if $\Delta_x(v) = 0$ for every $v \in V$.

Definition 11.1.6 (s-t flow and flow value)**.** Given two distinguished nodes s (source) and t (sink), an **s-t flow** is a feasible flow x such that $\Delta_x(v) = 0$ for all $v \in V \setminus \{s, t\}$ (flow conservation at intermediate nodes). The **value** of the flow is

$$\varphi(x) \;=\; \Delta_x(s) \;=\; -\Delta_x(t).$$

The equality $\Delta_x(s) = -\Delta_x(t)$ follows directly from the divergence theorem: since all intermediate divergences vanish, $\Delta_x(s) + \Delta_x(t) = 0$.

Example 11.1.7 (Divergence computation)**.** In the network of fig. 11.1:

- $\Delta_x(s) = 4 + 3 = 7$ (source),
- $\Delta_x(a) = (3 + 1) - 4 = 0$ (conservation),
- $\Delta_x(b) = 6 - (3 + 3) = 0$ (conservation),
- $\Delta_x(c) = 1 - 1 = 0$ (conservation),

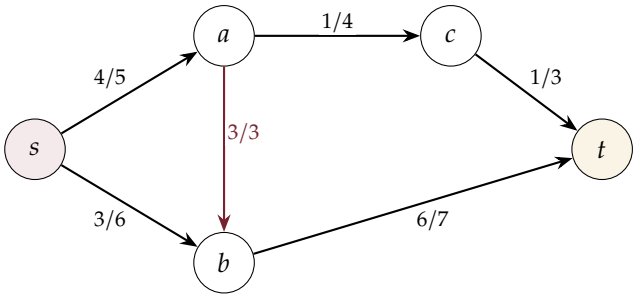


Figure 11.1: A flow network with five nodes. Arc labels show x_{ij}/k_{ij} (flow/capacity). The arc $a \to b$ (drawn highlighted with the main accent) is *saturated* ($x_{ab} = k_{ab} = 3$). The flow value is $\varphi(x) = \Delta_x(s) = (4+3) - 0 = 7$.

- $\Delta_x(t) = 0 - (6+1) = -7$ (sink).

The flow value is $\varphi(x) = 7$, and indeed $\sum_v \Delta_x(v) = 7 + 0 + 0 + 0 - 7 = 0$.

11.2 Flow Problems

11.2.1 Maximum Flow

Max-flow and min-cost flow are the two canonical network flow optimisation problems.

Definition 11.2.1 (Maximum flow problem)**.** Given a flow network (G, k) with source s and sink t, the **maximum flow problem** asks for a feasible s-t flow x^* that maximises the flow value:

$$\text{maximize} \quad \varphi(x) \;=\; \Delta_x(s) \qquad \text{subject to} \quad 0 \le x_{ij} \le k_{ij} \;\; \forall (i,j) \in A, \quad \Delta_x(v) = 0 \;\; \forall v \neq s, t.$$

11.2.2 Minimum-Cost Flow

Definition 11.2.2 (Minimum-cost flow problem)**.** Given a flow network (G, k, c) and a **demand vector** $b \in \mathbb{R}^{|V|}$ with $\sum_v b_v = 0$, the **minimum-cost flow problem** asks:

$$\text{minimize} \quad \sum_{(i,j)\in A} c_{ij}\, x_{ij} \qquad \text{subject to} \quad \Delta_x(v) = b_v \;\; \forall v \in V, \quad 0 \le x_{ij} \le k_{ij} \;\; \forall (i,j) \in A.$$

If we set $b_s = B$, $b_t = -B$, and $b_v = 0$ for all other nodes, the problem asks for the cheapest way to ship B units from s to t.

Remark 11.2.3 (Max-flow as a special min-cost flow). The max-flow problem can be reformulated as a min-cost flow by adding an artificial return arc (t, s) with $c_{ts} = -1$, $k_{ts} = \infty$, and setting all other arc costs to zero. Minimising total cost then maximises the flow on the return arc, which equals the s-t flow value.

11.3 ILP Formulation and Total Unimodularity

The incidence matrix of a directed graph is TU, so flow LPs with integer capacities and balances have integer optimal vertices.

11.3.1 LP Formulation

Consider the min-cost flow problem. Let M denote the **node-arc incidence matrix** of G, defined by

$$M_{v,(i,j)} = \begin{cases} +1 & \text{if } v = i, \\ -1 & \text{if } v = j, \\ 0 & \text{otherwise.} \end{cases}$$

Example 11.3.1 (Node-arc incidence matrix)**.** Consider a directed graph with 3 nodes and 4 arcs: $V = \{1, 2, 3\}$ and $A = \{(1, 2), (1, 3), (2, 3), (3, 1)\}$. The node-arc incidence matrix M has 3 rows (corresponding to the nodes) and 4 columns (corresponding to the arcs). Ordering the rows as $(1, 2, 3)$ and the columns as $((1, 2), (1, 3), (2, 3), (3, 1))$, the matrix M is:

$$M = \begin{pmatrix} 1 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & -1 & 1 \end{pmatrix}$$

Each column corresponds to a directed arc and contains exactly one $+1$ (at its starting node/tail) and one -1 (at its ending node/head), with all other entries being 0. Multiplying M by the flow vector $x = (x_{12}, x_{13}, x_{23}, x_{31})^\top$ yields the net outflow (divergence) at each node:

$$Mx = \begin{pmatrix} x_{12} + x_{13} - x_{31} \\ -x_{12} + x_{23} \\ -x_{13} - x_{23} + x_{31} \end{pmatrix} = \begin{pmatrix} \Delta_x(1) \\ \Delta_x(2) \\ \Delta_x(3) \end{pmatrix}$$

Then the min-cost flow LP is

$$\text{minimize} \quad c^\top x \qquad \text{subject to} \quad M'x = b', \quad 0 \le x \le k, \tag{11.1}$$

where M' is M with one row removed (any one row, typically the sink row, since the rows sum to zero) and b' is the corresponding subvector of b.

11.3.2 Total Unimodularity and Integrality

Theorem 11.3.2 (TU of incidence matrices)**.** *The node-arc incidence matrix M of any directed graph is **totally unimodular** (TU).*

This was proved in chapter 7. Since M is TU, so is M', and so is the full constraint matrix of (11.1) (after adding the identity blocks for the capacity constraints). By the integrality theorem for TU matrices:

Corollary 11.3.3 (Integer flows from LP)**.** *If the capacities k and demands b are integer, every vertex of the LP polyhedron defined by (11.1) is integral. In particular, the LP relaxation yields an optimal* integer *flow—no integrality constraints are needed.*

This result has a profound practical consequence: network flow problems with integer data can be solved exactly using polynomial-time LP algorithms.

A vertex-returning method such as the simplex method gives an integer optimum directly; an interior-point method may return a fractional point on an optimal face and then use crossover or a similar cleanup step to recover an integer optimal vertex. In either case we do not need ILP techniques such as branch-and-bound. It is one of the key reasons why network flow problems are among the "easiest" combinatorial optimisation problems.

11.4 Max-Flow Min-Cut Theorem

11.4.1 Cuts and Cut Capacity

Definition 11.4.1 (s-t cut)**.** An **s-t cut** in a network (G, k) is a partition (S, T) of V with $s \in S$ and $t \in T$, where $T = V \setminus S$. The **capacity** of the cut is

$$k(S) = \sum_{\substack{(i,j)\in A \\ i\in S,\ j\in T}} k_{ij},$$

i.e. the sum of capacities of arcs crossing from S to T.

Note that for a network with n nodes, there are 2^{n-2} possible s-t cuts (since s and t are fixed, and each of the remaining $n-2$ nodes can be placed in S or T).

Only arcs leaving S count towards the cut capacity; arcs entering S from T are not *included.*

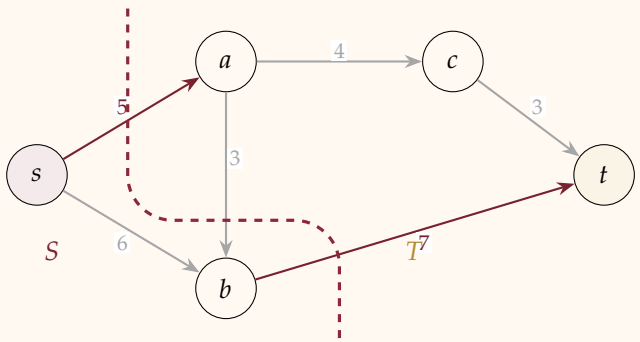
Example 11.4.2 (s-t cut and capacity)**.** Consider the network from fig. 11.1. Let the node set be $V = \{s, a, b, c, t\}$. Define the s-t cut (S, T) by choosing $S = \{s, b\}$ and $T = \{a, c, t\}$. The partition is valid because $s \in S$ and $t \in T$.

To compute the capacity $k(S)$, we look at all directed arcs that go from S to T. Let's examine all arcs in the graph:

- $s \to a$ crosses from S to T (capacity $k_{sa} = 5$).
- $b \to t$ crosses from S to T (capacity $k_{bt} = 7$).
- $a \to b$ crosses from T to S (capacity $k_{ab} = 3$). This arc is *not* counted because it enters S instead of leaving it.
- $a \to c$ (internal to T, capacity 4), $c \to t$ (internal to T, capacity 3), and $s \to b$ (internal to S, capacity 6) are not counted.

Therefore, the capacity of the cut is:

$$k(S) = k_{sa} + k_{bt} = 5 + 7 = 12.$$



The highlighted arcs ($s \to a$ and $b \to t$) are the only ones crossing from S

to T. The arc $a \to b$ goes in the opposite direction (from T to S) and does not count towards $k(S)$.

11.4.2 Flow Through a Cut

A key observation relates the flow value to any cut. For any s-t flow x and any s-t cut (S, T), define the **net flow through the cut** as

$$\Delta_x(S) = \sum_{\substack{(i,j)\in A \\ i\in S,\, j\in T}} x_{ij} - \sum_{\substack{(i,j)\in A \\ i\in T,\, j\in S}} x_{ij}.$$

Theorem 11.4.3 (Flow value equals net flow through any cut)**.** *For any s-t flow x and any s-t cut (S, T),*

$$\varphi(x) = \Delta_x(S).$$

Proof. Summing the divergences over all $v \in S$: $\sum_{v\in S} \Delta_x(v) = \Delta_x(s)$ since $\Delta_x(v) = 0$ for all $v \in S \setminus \{s\}$. When expanding the sum, arcs internal to S cancel in pairs, leaving exactly $\Delta_x(S)$. □

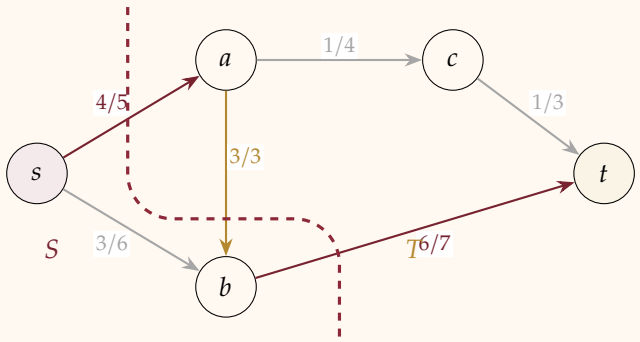
Example 11.4.4 (Net flow through a cut)**.** Using the flow from fig. 11.1 (with value $\varphi(x) = 7$) and the cut (S, T) from theorem 11.4.2 where $S = \{s, b\}$ and $T = \{a, c, t\}$, we calculate the net flow through the cut $\Delta_x(S)$:

- Arcs crossing from S to T:
  - $s \to a$ carrying flow $x_{sa} = 4$.
  - $b \to t$ carrying flow $x_{bt} = 6$.
- Arcs crossing from T to S:
  - $a \to b$ carrying flow $x_{ab} = 3$.

Thus, the net flow crossing the cut boundary from S to T is:

$$\Delta_x(S) = (x_{sa} + x_{bt}) - x_{ab} = (4 + 6) - 3 = 7.$$

This matches the flow value $\varphi(x) = 7$, illustrating theorem 11.4.3.



The highlighted forward arcs ($s \to a$ and $b \to t$ crossing from S to T) carry $4 + 6 = 10$ units of flow. The backward arc ($a \to b$ crossing from T to S) carries 3 units. The net flow is $10 - 3 = 7$.

11.4.3 Weak Duality

Theorem 11.4.5 (Weak max-flow min-cut)**.** *For any feasible s-t flow x and any s-t cut (S, T),*

$$\varphi(x) \leq k(S).$$

In particular, $\max_x \varphi(x) \leq \min_S k(S)$.

Proof. By theorem 11.4.3, $\varphi(x) = \Delta_x(S) = \sum_{i \in S, j \in T} x_{ij} - \sum_{i \in T, j \in S} x_{ij}$. Since $x_{ij} \geq 0$, the second sum is non-negative, so $\varphi(x) \leq \sum_{i \in S, j \in T} x_{ij} \leq \sum_{i \in S, j \in T} k_{ij} = k(S)$. □

Weak duality: every feasible flow is bounded by every cut capacity. Finding a matching pair certifies optimality.

Observation 11.4.6 (Certificate of optimality). If a feasible flow x and a cut (S, T) satisfy $\varphi(x) = k(S)$, then x is a *maximum flow* and (S, T) is a *minimum cut*.

Example 11.4.7 (Weak Duality illustration)**.** Let's compare the flow x of value $\varphi(x) = 7$ from fig. 11.1 with two different cuts to verify the weak duality inequality $\varphi(x) \leq k(S)$:

- **Cut 1**: $S_1 = \{s\}$ and $T_1 = \{a, b, c, t\}$. The capacity is:

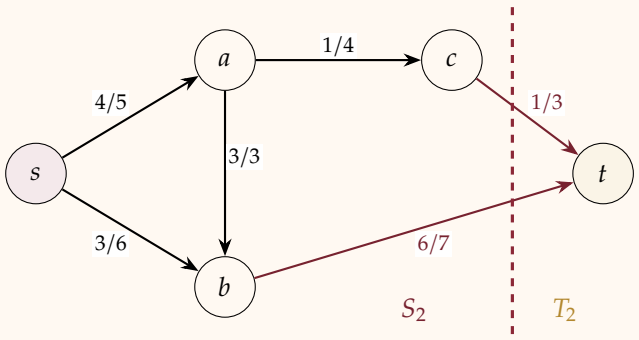
$$k(S_1) = k_{sa} + k_{sb} = 5 + 6 = 11.$$

  We have $\varphi(x) = 7 \leq 11 = k(S_1)$.
- **Cut 2**: $S_2 = \{s, a, b, c\}$ and $T_2 = \{t\}$. The capacity is:

$$k(S_2) = k_{bt} + k_{ct} = 7 + 3 = 10.$$

  We have $\varphi(x) = 7 \leq 10 = k(S_2)$.

Since $\varphi(x) \leq k(S)$ for all cuts, the capacity of any cut provides an upper bound on what flow value can possibly be achieved. In this case, the existence of Cut 2 tells us that we can never push more than 10 units of flow through the network.



The cut $S_2 = \{s, a, b, c\}$ separates the source and intermediate nodes from the sink t. The arcs crossing from S_2 to T_2 ($b \to t$ and $c \to t$) carry a combined flow of $6 + 1 = 7$, which is strictly less than their combined capacity of $7 + 3 = 10$.

11.4.4 Residual Network and Augmenting Paths

To prove strong duality, we need a way to mathematically characterize when a flow cannot be increased further. This is done by constructing an auxiliary graph called the residual network, which represents the remaining capacities and the ability to cancel flow.

Definition 11.4.8 (Residual network)**.** Given a feasible flow x in (G, k), the **residual network** $R_x = (V, A_x)$ is a directed graph on the same node set with:

- a **forward arc** (i, j) with residual capacity $r_{ij} = k_{ij} - x_{ij}$ whenever $x_{ij} < k_{ij}$;
- a **backward arc** (j, i) with residual capacity $r_{ji} = x_{ij}$ whenever $x_{ij} > 0$.

Forward arcs represent room to *increase* flow; backward arcs represent the option to *decrease* ("push back") existing flow.

Definition 11.4.9 (Augmenting path and bottleneck)**.** An **augmenting path** is a directed path P from s to t in the residual network R_x. Its **bottleneck capacity** is

$$\varepsilon(P) = \min_{(u,v)\in P} r_{uv}.$$

Example 11.4.10 (Residual network and augmenting path)**.** Consider the flow x of value $\varphi(x) = 7$ from fig. 11.1. We construct the residual network R_x and identify an augmenting path.

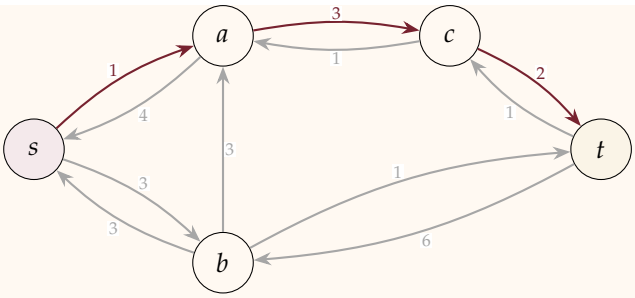
For each arc in the original graph:

- $s \to a$ (4/5): since $x_{sa} < k_{sa}$, we have a forward arc $s \to a$ with capacity $r_{sa} = 5 - 4 = 1$. Since $x_{sa} > 0$, we have a backward arc $a \to s$ with capacity $r_{as} = 4$.
- $s \to b$ (3/6): forward arc $s \to b$ (capacity 3), backward arc $b \to s$ (capacity 3).
- $a \to b$ (3/3): since the arc is saturated ($x_{ab} = k_{ab}$), there is no forward arc. Since $x_{ab} > 0$, we have a backward arc $b \to a$ with capacity $r_{ba} = 3$.
- $a \to c$ (1/4): forward arc $a \to c$ (capacity 3), backward arc $c \to a$ (capacity 1).
- $b \to t$ (6/7): forward arc $b \to t$ (capacity 1), backward arc $t \to b$ (capacity 6).
- $c \to t$ (1/3): forward arc $c \to t$ (capacity 2), backward arc $t \to c$ (capacity 1).

An augmenting path is a path from s to t in R_x. We can find two such paths:

1. $P_1 = s \to a \to c \to t$ with bottleneck capacity $\varepsilon(P_1) = \min(1, 3, 2) = 1$.
2. $P_2 = s \to b \to t$ with bottleneck capacity $\varepsilon(P_2) = \min(3, 1) = 1$.

Below is the residual network R_x, with the augmenting path $P_1 = s \to a \to c \to t$ highlighted:



11.4.5 Strong Duality

Theorem 11.4.11 (Max-flow min-cut theorem). *In any flow network with source s and sink t,*

$$\max_x \varphi(x) \;=\; \min_S k(S).$$

Formal details — Proof sketch of the Max-Flow Min-Cut Theorem

The proof shows that if no augmenting path exists in the residual network R_x, then the current flow is optimal.

Define $S^* = \{v \in V$: there is a directed path from s to v in $R_x\}$. Since no augmenting path from s to t exists, $t \notin S^*$, so $(S^*, V \setminus S^*)$ is a valid s-t cut.

For any arc $(i,j) \in A$ with $i \in S^*$ and $j \notin S^*$: if $x_{ij} < k_{ij}$, a forward arc would exist in R_x, making j reachable from s, contradicting $j \notin S^*$. Hence $x_{ij} = k_{ij}$ (all outgoing arcs are saturated).

For any arc $(j,i) \in A$ with $j \notin S^*$ and $i \in S^*$: if $x_{ji} > 0$, a backward arc (i,j) would exist in R_x, again making j reachable. Hence $x_{ji} = 0$ (all return arcs are empty).

Therefore:

$$\varphi(x) = \Delta_x(S^*) = \sum_{\substack{(i,j)\in A \\ i\in S^*,\, j\notin S^*}} k_{ij} \;-\; 0 = k(S^*).$$

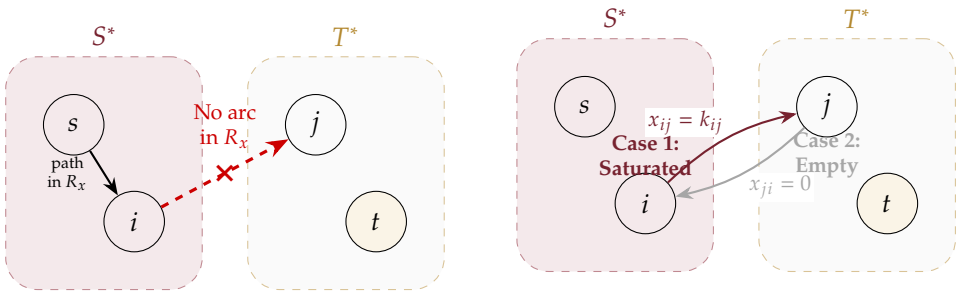
By weak duality (theorem 11.4.5), x is a maximum flow and $(S^*, V \setminus S^*)$ is a minimum cut.

Intermezzo — Strong Duality in Layman's Terms

To understand the intuition behind the Max-Flow Min-Cut Theorem and its proof, imagine a pipeline network carrying water from a reservoir (s) to a city (t). Each pipe has a maximum capacity. The theorem asserts that the maximum amount of water we can send to the city is limited by the tightest bottleneck (the minimum cut) in the network.

The proof explains how to find this bottleneck once we have pushed as much water as possible:

1. **Coloring reachable nodes**: Once the flow is blocked and no more water can be sent (no augmenting path exists), we color "blue" all nodes that can still receive a trickle of additional water from the reservoir in the residual network (this is the set S^*). The reservoir s is blue. The city t must remain white, because we cannot send any more water to it.
2. **Inspecting the boundary**: Now look at the boundary between the blue nodes and the white nodes:
   - **Outgoing pipes must be full**: Any pipe going from a blue node to a white node must be running at full capacity (saturated). If there were any spare capacity, water would flow through it, making the white node reachable, so it would have been colored blue.
   - **Incoming pipes must be empty**: Any pipe going from a white node to a blue node



(a) Residual network R_x: no arc can cross from S^* to T^*, otherwise j would be reachable and thus in S^*.

(b) Original network G: outgoing arcs must be fully saturated ($x_{ij} = k_{ij}$), and incoming arcs must be empty ($x_{ji} = 0$).

Figure 11.2: Visual representation of the Strong Duality proof. (a) By definition, S^* contains all nodes reachable from s in the residual network R_x. (b) In the original graph G, any outgoing arc $i \to j$ crossing the cut must be saturated (otherwise a forward arc $i \to j$ would exist in R_x), and any incoming arc $j \to i$ must carry zero flow (otherwise a backward arc $i \to j$ would exist in R_x).

must carry zero flow. If it carried water, we could reduce its flow, which is equivalent to pushing water backward, making the white node reachable from the blue node.

3. **The bottleneck is found**: Since all pipes crossing the boundary from blue to white are completely full, and none of the returning pipes are carrying water back, the total water flowing from the blue zone to the white zone is exactly the sum of the capacities of the boundary pipes. This boundary is the bottleneck (the minimum cut), and the water flowing through it is the maximum flow.

In the running example (fig. 11.3), the maximum flow has value 10. The cut $(S', T') = (\{s, a, b, c\}, \{t\})$ has capacity $k_{bt} + k_{ct} = 7 + 3 = 10 = \varphi(x)$, so by theorem 11.4.6 the flow is optimal and the cut is minimum.

11.5 Ford–Fulkerson Algorithm

11.5.1 Flow Augmentation

The key insight: if we cannot reach t from s in the residual network, the current flow is optimal.

Given a feasible flow x and an augmenting path P in R_x (as defined in section 11.4), we can increase the flow value by updating the flow along the path. The flow is updated as follows:

$$x'_{ij} = \begin{cases} x_{ij} + \varepsilon(P) & \text{if } (i,j) \in P^+ \text{ (forward arc)}, \\ x_{ij} - \varepsilon(P) & \text{if } (j,i) \in P^- \text{ (backward arc on original arc } (i,j)), \\ x_{ij} & \text{otherwise.} \end{cases}$$

One can verify that x' is feasible and $\varphi(x') = \varphi(x) + \varepsilon(P)$.

11.5.2 The Ford–Fulkerson Scheme

Intermezzo — Ford and Fulkerson (1956)

L. R. Ford, Jr. and D. R. Fulkerson introduced the augmenting-path method in their seminal

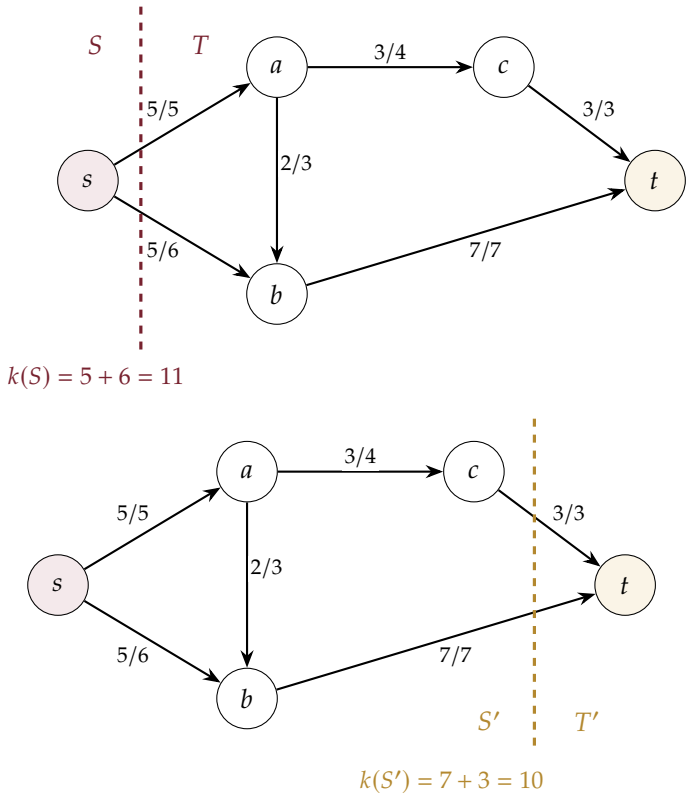
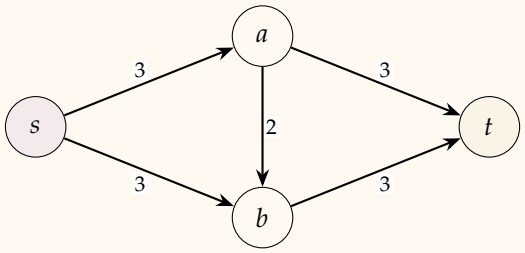


Figure 11.3: The max-flow on the running example network. **Top:** cut $S = \{s\}$ has capacity $k(S) = 5+6 = 11 > \varphi(x)$. **Bottom:** cut $S' = \{s, a, b, c\}$ has capacity $k(S') = 7 + 3 = 10 = \varphi(x)$, certifying that this is the min-cut and the flow value 10 is maximum.

1956 paper. Technically, their method is a *scheme* rather than an algorithm, because it does not specify *how* to choose the augmenting path. Different choices lead to different running times—from pseudo-polynomial to strongly polynomial.

11.5.3 Worked Example

Example 11.5.1 (Ford–Fulkerson on a 4-node network)**.** Consider the network below with s, a, b, t and capacities as shown. We trace the Ford–Fulkerson algorithm.



Iteration 1. Augmenting path: $s \to a \to b \to t$ (all forward arcs in R_x). Bottleneck: $\varepsilon = \min(3, 2, 3) = 2$.

Algorithm 10: Ford–Fulkerson scheme

Input: Flow network (G, k) with source s, sink t
Output: Maximum s-t flow x

$x_{ij} \leftarrow 0$ for all $(i, j) \in A$
while *there exists an* s-t *path* P *in* R_x **do**
  $\varepsilon \leftarrow \min\{r_{uv} : (u, v) \in P\}$
  foreach *forward arc* $(i, j) \in P^+$ **do**
    $x_{ij} \leftarrow x_{ij} + \varepsilon$
  foreach *backward arc* $(j, i) \in P^-$ (original arc (i, j)) **do**
    $x_{ij} \leftarrow x_{ij} - \varepsilon$
return x

Update: $x_{sa} = 2$, $x_{ab} = 2$, $x_{bt} = 2$. Flow value: $\varphi = 2$.

Iteration 2. Augmenting path: $s \to b \to a \to t$.
The arc $b \to a$ is a *backward arc* in R_x (since $x_{ab} = 2 > 0$, backward arc has capacity 2).
Bottleneck: $\varepsilon = \min(r_{sb}, r_{ba}, r_{at}) = \min(3, 2, 3) = 2$.
Update: $x_{sb} = 0 + 2 = 2$, $x_{ab} = 2 - 2 = 0$ (decreased via backward arc), $x_{at} = 0 + 2 = 2$. Flow value: $\varphi = 4$.

Iteration 3. Augmenting path: $s \to a \to t$ (forward arcs).
Bottleneck: $\varepsilon = \min(r_{sa}, r_{at}) = \min(1, 1) = 1$.
Update: $x_{sa} = 2 + 1 = 3$, $x_{at} = 2 + 1 = 3$. Flow value: $\varphi = 5$.

Iteration 4. Augmenting path: $s \to b \to t$ (forward arcs).
Bottleneck: $\varepsilon = \min(r_{sb}, r_{bt}) = \min(1, 1) = 1$.
Update: $x_{sb} = 2 + 1 = 3$, $x_{bt} = 2 + 1 = 3$. Flow value: $\varphi = 6$.

Termination. No s-t path exists in R_x. The arcs $s \to a$ and $s \to b$ are both saturated, so s is isolated in the residual network.

Final flow: $x_{sa} = 3$, $x_{sb} = 3$, $x_{ab} = 0$, $x_{at} = 3$, $x_{bt} = 3$. Maximum flow $\varphi^* = 6$.
Min-cut: $S^* = \{s\}$ (only s is reachable in R_x). Capacity: $k(S^*) = k_{sa} + k_{sb} = 3 + 3 = 6 = \varphi^*$.

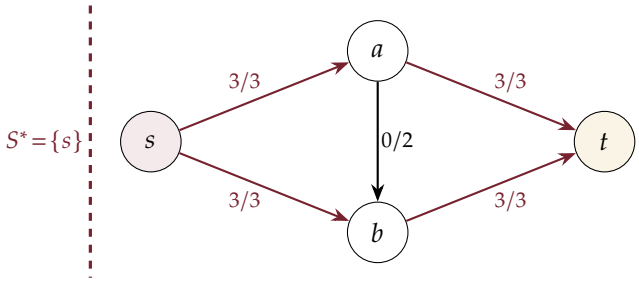


Figure 11.4: Final maximum flow from theorem 11.5.1. Saturated arcs are highlighted with the main accent. The dashed line shows the minimum cut $S^* = \{s\}$, with capacity $k(S^*) = 6 = \varphi^*$.

11.6 Complexity and Edmonds–Karp

11.6.1 Integer Capacities: Termination

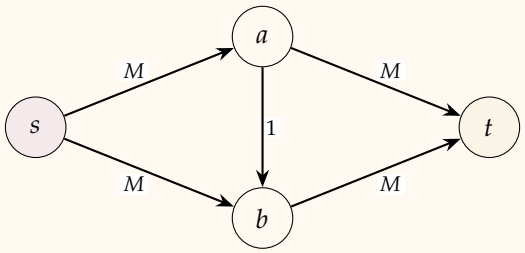
With integer capacities, Ford–Fulkerson terminates but may be very slow.

When all capacities are integers, the flow value starts at 0 and increases by at least 1 per iteration (since every bottleneck $\varepsilon(P) \geq 1$). Thus the scheme terminates after at most φ^* iterations, yielding a total running time of $O\big(|A| \cdot \varphi^*\big)$, where each iteration requires $O(|A|)$ time to find a path (e.g. via DFS or BFS).

This complexity is **pseudo-polynomial**: it depends on the *value* of the data (the max-flow φ^*), not just on the *size* of the input. Since φ^* can be exponential in the input encoding length ($\varphi^* \leq \sum k_{ij}$, which can be represented in $O\big(\log \varphi^*\big)$ bits), this is not polynomial in the traditional sense.

11.6.2 The Zig-Zag Worst Case

Example 11.6.1 (Pseudo-polynomial behaviour)**.** Consider the "diamond" network with four nodes:



The maximum flow is clearly $2M$ (send M through each of $s \to a \to t$ and $s \to b \to t$). But if the algorithm alternates between $s \to a \to b \to t$ and $s \to b \to a \to t$ (using the backward arc on $a \to b$), each iteration sends only 1 unit. This requires $2M$ iterations—a number that grows with the capacity M, not with the graph size.

11.6.3 Irrational Capacities: Non-Termination

If arc capacities are allowed to be irrational, the Ford–Fulkerson scheme may *fail to terminate* and can even converge to a flow value strictly less than the optimum. A well-known counter-example uses capacities based on the golden ratio. This pathology motivates the need for a more careful path-selection rule.

11.6.4 Edmonds–Karp Algorithm

Intermezzo — Edmonds and Karp (1972)

Jack Edmonds and Richard Karp showed that a simple modification turns the Ford–Fulkerson scheme into a polynomial-time algorithm: always choose a *shortest* augmenting path (fewest arcs), found via BFS.

Theorem 11.6.2 (Edmonds–Karp complexity)**.** *If the augmenting path is always chosen as a shortest s-t path in the residual network (found via BFS), the total number of augmentations is at most $O(|V| \cdot |A|)$, and the overall running*

time is

$$O\left(|V| \cdot |A|^2\right).$$

The key insight is that with BFS, the shortest-path distance from s to any node in the residual network never decreases across iterations. This limits the total number of "saturating" augmentations per edge to $O(|V|)$, bounding the overall iteration count.

The Edmonds–Karp modification is surprisingly simple—just use BFS instead of DFS—yet it transforms pseudo-polynomial into strongly polynomial.

Since $O\left(|V| \cdot |A|^2\right)$ depends only on the graph structure (not on capacities), the Edmonds–Karp algorithm is **strongly polynomial**. This also means that the minimum cut can be found in polynomial time, contrasting with the NP-hard *maximum* cut problem.

Remark 11.6.3 (Better algorithms exist). While Edmonds–Karp was groundbreaking, faster max-flow algorithms have since been developed. Notable examples include Dinic's blocking-flow algorithm ($O\left(|V|^2 \cdot |A|\right)$) and push-relabel methods ($O\left(|V|^3\right)$). These are beyond the scope of this course.

11.7 Summary

Table 11.1: Summary of max-flow algorithms.

Algorithm	Path selection	Complexity	Type
Ford–Fulkerson	arbitrary	$O\left(\lvert A\rvert \cdot \varphi^*\right)$	pseudo-polynomial
Edmonds–Karp	shortest (BFS)	$O\left(\lvert V\rvert \cdot \lvert A\rvert^2\right)$	strongly polynomial

The main results of this chapter are:

- Network flow problems (max-flow, min-cost flow) can be formulated as LPs whose constraint matrix is totally unimodular (section 11.3). With integer data, LP relaxation yields integer solutions automatically.
- The **Max-Flow Min-Cut Theorem** (theorem 11.4.11): the maximum s-t flow value equals the minimum s-t cut capacity. Finding one solves the other.
- The **Ford–Fulkerson scheme** (algorithm 10) iteratively augments flow along paths in the residual network until no augmenting path remains.
- With arbitrary path selection, Ford–Fulkerson is pseudo-polynomial. The **Edmonds–Karp** variant (BFS) achieves $O\left(|V| \cdot |A|^2\right)$ — strongly polynomial.
- Min-cost flow algorithms (e.g. successive shortest paths, cycle cancelling) are not covered here but rely on similar residual-network ideas; see the remark below.

Remark 11.7.1 (Min-cost flow — further reading). Network flow problems can be enriched with arc costs: we want not just *any* flow, but the *cheapest* one. This is the **min-cost flow problem**, which encompasses shortest paths, transportation, and assignment as special cases. Formally, it asks for a feasible flow of a given value φ^* that minimises the total cost $\sum_{(i,j)\in A} c_{ij} x_{ij}$, where c_{ij} is the cost per unit of flow on arc (i, j). Two classical algorithms are:

- **Successive shortest paths**: repeatedly augment flow along a *minimum-cost* augmenting path in the residual network (found, e.g., with

Dijkstra after re-weighting via Johnson's technique to handle reduced costs).

- **Cycle cancelling**: start from any feasible flow of value φ^*, then repeatedly cancel negative-cost directed cycles in the residual network until none remain; the resulting flow is optimal.

Both algorithms rely on the same residual-network structure used for max-flow. Because the constraint matrix of the min-cost flow LP is totally unimodular (section 11.3 and chapter 7), the TU structure guarantees that all basic feasible solutions are integer whenever the supply/demand vector b and capacity vector k are integer—no branch-and-bound is needed. The running time of successive shortest paths depends on the number of augmentations. With integer data and one unit sent per augmentation, a basic bound is $O(\varphi^*(m + n \log n))$ using Dijkstra with reduced costs; stronger bounds require more sophisticated min-cost-flow algorithms.

Summary & Key Takeaways

- **Flow Network:** Directed graph with source s, sink t, and capacities $c(u, v) \geq 0$. Flow conservation requires inflow = outflow for all $v \notin \{s, t\}$.
- **Max-Flow Min-Cut Theorem:** The maximum value of an s-t flow equals the minimum capacity of an s-t cut.
- **Ford-Fulkerson Method:** Repeatedly augments flow along paths in the residual graph G_f. Complexity is $O\left(E|f^*|\right)$.
- **Edmonds-Karp Algorithm:** Implements Ford-Fulkerson using BFS to select shortest augmenting paths, guaranteeing polynomial time execution of $O\left(VE^2\right)$.
- **Integrality Theorem:** If all capacities are integers, there exists an integer maximum flow.

Exercises

Exercise 1. Consider the flow network with nodes $\{s, a, b, t\}$ and arc capacities $k_{sa} = 5$, $k_{sb} = 3$, $k_{ab} = 2$, $k_{at} = 4$, $k_{bt} = 5$. A proposed flow assigns $f_{sa} = 4$, $f_{sb} = 2$, $f_{ab} = 1$, $f_{at} = 3$, $f_{bt} = 3$.

(a) Verify that every capacity constraint $0 \leq f_e \leq k_e$ holds.

(b) Check flow conservation at every internal node (a and b).

(c) Compute the value $|f|$ of this flow (net flow out of s).

(d) Is this flow feasible? If not, identify the violated constraint.

Exercise 2. A *circulation* is a feasible flow on a network with no source or sink, i.e. the divergence equals zero at every node. Consider the directed graph with nodes $\{1, 2, 3, 4\}$, arc capacities $k_{12} = 6$, $k_{23} = 4$, $k_{34} = 5$, $k_{41} = 3$, $k_{13} = 2$, $k_{24} = 4$.

(a) Write the flow-conservation constraint for each node.

(b) Find a feasible circulation of total arc flow as large as possible.

(c) Is the feasibility of a circulation related to the existence of directed cycles? Explain.

Exercise 3. Let $G = (V, A)$ be a flow network with source s and sink t. Recall that the *divergence* of a flow f at node v is $\operatorname{div}_f(v) = \sum_{(v,w)\in A} f_{vw} - \sum_{(u,v)\in A} f_{uv}$.

(a) Prove that $\sum_{v\in V} \operatorname{div}_f(v) = 0$ for any feasible flow f.

(b) Conclude that $\operatorname{div}_f(s) = -\operatorname{div}_f(t)$ when $\operatorname{div}_f(v) = 0$ for all $v \notin \{s, t\}$.

(c) If $|f|$ denotes $\operatorname{div}_f(s)$, explain why $|f|$ is well-defined as the *value* of the flow.

Exercise 4. Consider the flow network with nodes $\{s, a, b, c, t\}$ and arcs with capacities $k_{sa} = 4$, $k_{sb} = 6$, $k_{ac} = 3$, $k_{bc} = 5$, $k_{at} = 2$, $k_{ct} = 7$. Suppose we assign $f_{sa} = 3$, $f_{sb} = 4$, $f_{ac} = 2$, $f_{bc} = 4$, $f_{at} = 1$, $f_{ct} = 6$.

(a) Check capacity feasibility for every arc.

(b) Check flow conservation at a, b, and c.

(c) Is this a feasible s-t flow? What is its value?

(d) If the flow is infeasible, correct the assignment for exactly one arc to restore feasibility and state the new flow value.

Exercise 5. State and prove the *flow decomposition lemma*: for any feasible s-t flow f of value $v > 0$, there exist s-t paths and directed cycles in G such that f can be written as a non-negative combination of their characteristic vectors, where the s-t path components together carry exactly v units of flow. (You may assume integer-valued flows and proceed by induction on the number of arcs with $f_e > 0$.)

Exercise 6. Consider the flow network with nodes $\{s, a, b, t\}$ and arc capacities $k_{sa} = 3$, $k_{sb} = 2$, $k_{at} = 4$, $k_{bt} = 3$, $k_{ab} = 2$.

(a) List all s-t cuts and compute the capacity of each.

(b) Identify the minimum cut(s).

(c) Find a feasible s-t flow whose value equals the minimum cut capacity, thereby verifying the Max-Flow Min-Cut Theorem.

Exercise 7. Consider the flow network with nodes $\{s, a, b, c, t\}$ and arc capacities $k_{sa} = 10$, $k_{sb} = 8$, $k_{ac} = 6$, $k_{bc} = 7$, $k_{at} = 4$, $k_{ct} = 9$, $k_{bt} = 3$.

(a) Write the LP formulation of the max-flow problem on this network.

(b) Identify the constraint matrix and argue why it is totally unimodular (TU).

(c) Conclude that the LP optimal solution is always integer-valued when all capacities are integers.

Exercise 8. Let (S, T) be an s-t cut of a flow network G. Let f be any feasible s-t flow.

(a) Prove the *weak duality* result: $|f| \le \operatorname{cap}(S, T)$.

(b) Explain why the Max-Flow Min-Cut Theorem constitutes the corresponding *strong duality* result.

(c) Give an example where strict inequality $|f| < \text{cap}(S, T)$ holds for a particular cut but the flow is still maximum.

Exercise 9. Consider the flow network with nodes $\{s, 1, 2, 3, t\}$ and arc capacities $k_{s1} = 5$, $k_{s2} = 4$, $k_{12} = 3$, $k_{13} = 2$, $k_{23} = 4$, $k_{1t} = 3$, $k_{3t} = 5$, $k_{2t} = 2$. Find *all* minimum s-t cuts by:

(a) Computing the maximum flow (state the flow on each arc).

(b) Constructing the residual network G_f at the optimal flow.

(c) Identifying the set R of nodes reachable from s in G_f.

(d) Showing that every minimum cut corresponds to a partition (S, T) with S containing s and T containing t such that every arc from S to T is saturated and every arc from T to S carries zero flow.

Exercise 10. True or False. For each statement, write T or F and give a brief justification or counterexample.

(a) The maximum s-t flow is unique.

(b) If all arc capacities are positive integers, then there exists a maximum flow that is integer-valued.

(c) The minimum cut is always unique.

(d) Every feasible flow can be augmented along some path in the residual network if and only if the flow is not maximum.

(e) The value of the maximum flow equals the capacity of the minimum cut in every flow network.

Exercise 11. Consider the flow network with nodes $\{s, a, b, t\}$ and arc capacities $k_{sa} = 4$, $k_{sb} = 3$, $k_{ab} = 2$, $k_{at} = 3$, $k_{bt} = 4$. Suppose the current flow is $f_{sa} = 2$, $f_{sb} = 1$, $f_{ab} = 1$, $f_{at} = 3$, $f_{bt} = 0$.

(a) Verify that this flow is feasible (check capacity and conservation).

(b) Construct the residual network G_f: for each arc (u, v) list the forward residual capacity $k_{uv} - f_{uv}$ and the backward residual capacity f_{uv}.

(c) Identify all s-t paths in G_f and their bottleneck capacities.

(d) Augment along the best path and update the flow and the residual network.

Exercise 12. Consider the following intermediate state of a Ford–Fulkerson execution on a network with nodes $\{s, a, b, c, t\}$. Arc capacities: $k_{sa} = 6$, $k_{sc} = 4$, $k_{ab} = 5$, $k_{cb} = 3$, $k_{bt} = 7$, $k_{at} = 2$. Current flow: $f_{sa} = 4$, $f_{sc} = 3$, $f_{ab} = 4$, $f_{cb} = 3$, $f_{bt} = 6$, $f_{at} = 1$.

(a) Describe the residual network G_f by listing, for each original arc, the forward and backward residual arcs and their capacities.

(b) Find an augmenting path in G_f (if one exists) and state its bottleneck capacity.

(c) If no augmenting path exists, identify the minimum cut and verify its capacity equals the current flow value.

Exercise 13. Prove that a feasible s-t flow f is maximum if and only if the residual network G_f contains no directed s-t path. *Hint:* Use the Max-Flow Min-Cut Theorem and the cut defined by the set of nodes reachable from s in G_f.

Exercise 14. Ford–Fulkerson trace (4 nodes). Consider the flow network with nodes $\{s, a, b, t\}$ and arc capacities $k_{sa} = 4$, $k_{sb} = 3$, $k_{ab} = 2$, $k_{at} = 5$, $k_{bt} = 4$. Starting from the zero flow, execute the Ford–Fulkerson algorithm, choosing augmenting paths in lexicographic order (smallest node label first).

(a) For each iteration, state: the augmenting path, its bottleneck capacity, and the updated flow on every arc.

(b) Draw (describe textually) the residual network after each augmentation.

(c) When the algorithm terminates, state the maximum flow value.

(d) Identify a minimum cut and verify that its capacity equals the maximum flow value.

Exercise 15. Ford–Fulkerson trace (5 nodes). Consider the flow network with nodes $\{s, a, b, c, t\}$ and arc capacities $k_{sa} = 7$, $k_{sb} = 5$, $k_{ac} = 4$, $k_{bc} = 3$, $k_{at} = 3$, $k_{ct} = 8$, $k_{bt} = 2$. Starting from the zero flow, execute Ford–Fulkerson using any valid sequence of augmenting paths.

(a) Show at least three distinct augmenting paths used across the execution, with their bottleneck capacities.

(b) State the final flow on every arc.

(c) Compute the maximum flow value.

(d) Find the minimum cut and verify the Max-Flow Min-Cut Theorem.

Exercise 16. Ford–Fulkerson trace (5 nodes, backward arc used). Consider the flow network with nodes $\{s, a, b, c, t\}$ and arc capacities $k_{sa} = 5$, $k_{sc} = 3$, $k_{ab} = 4$, $k_{cb} = 2$, $k_{ba} = 1$, $k_{bt} = 5$, $k_{ct} = 4$. Suppose after two augmentations the partial flow is: $f_{sa} = 3$, $f_{sc} = 3$, $f_{ab} = 3$, $f_{cb} = 2$, $f_{ba} = 0$, $f_{bt} = 5$, $f_{ct} = 0$.

(a) Verify that this partial flow is feasible.

(b) Construct the residual network and identify an augmenting path that uses a *backward* arc.

(c) Complete the Ford–Fulkerson execution: augment, update the flow, and repeat until no augmenting path remains.

(d) State the maximum flow value and exhibit a minimum cut.

Exercise 17. Ford–Fulkerson trace (6 nodes). Consider the flow network with nodes $\{s, a, b, c, d, t\}$ and arc capacities $k_{sa} = 8$, $k_{sb} = 6$, $k_{ac} = 5$, $k_{bc} = 4$, $k_{ad} = 3$, $k_{bd} = 5$, $k_{ct} = 7$, $k_{dt} = 6$.

(a) Find the maximum s-t flow by any method (you may use Ford–Fulkerson or inspection). State the flow on every arc.

(b) List every minimum cut (S, T) of this network.

(c) For each minimum cut, verify that every forward arc (u, v) with $u \in S$, $v \in T$ is saturated ($f_{uv} = k_{uv}$) and every backward arc (u, v) with $u \in T$, $v \in S$ carries zero flow ($f_{uv} = 0$).

Exercise 18. Ford–Fulkerson trace (4 nodes, multiple min cuts). Consider the flow network with nodes $\{s, a, b, t\}$ and arc capacities $k_{sa} = 3$, $k_{sb} = 3$, $k_{at} = 3$, $k_{bt} = 3$, $k_{ab} = 0$ (i.e. no arc between a and b).

(a) Run Ford–Fulkerson from the zero flow and state every augmenting path used.

(b) Find the maximum flow value.

(c) List *all* minimum cuts, compute their capacities, and verify they are all equal to the maximum flow value.

(d) What structural property of this network causes the minimum cut not to be unique?

Exercise 19. Consider the following "zig-zag" network used to show that Ford–Fulkerson with arbitrary path selection can require many iterations when capacities are large. Nodes: $\{s, a, b, t\}$. Arc capacities: $k_{sa} = 100$, $k_{sb} = 100$, $k_{ab} = 1$, $k_{at} = 100$, $k_{bt} = 100$.

(a) Show that Ford–Fulkerson, if it alternately selects the path $s \to a \to b \to t$ and then $s \to b \to a \to t$ (using the backward arc $b \to a$ in the residual network), requires 200 augmentations to reach the maximum flow.

(b) What is the maximum flow value in this network?

(c) Explain how Edmonds–Karp (BFS shortest augmenting path) avoids this behaviour and find the max flow in at most 2 BFS iterations.

Exercise 20. Edmonds–Karp complexity. Edmonds–Karp selects, at each iteration of Ford–Fulkerson, a *shortest* augmenting path (fewest arcs) using BFS.

(a) State the complexity of Edmonds–Karp in terms of $|V|$ and $|A|$ and compare it with the complexity of plain Ford–Fulkerson.

(b) Explain why the distance $d_{G_f}(s, v)$ from s to any node v in the residual network is non-decreasing across iterations.

(c) Use this monotonicity to argue that each arc can become *critical* (i.e. saturated) at most $O(|V|)$ times, giving a total of $O(|V| \cdot |A|)$ augmentations.

(d) Conclude that Edmonds–Karp runs in $O(|V| \cdot |A|^2)$ time.

Exercise 21. True or False — termination and complexity.

(a) Ford–Fulkerson always terminates if all arc capacities are positive rational numbers.

(b) Ford–Fulkerson always terminates if all arc capacities are positive integers.

(c) Edmonds–Karp always terminates, regardless of whether capacities are rational or irrational.

(d) If all capacities are integers, Ford–Fulkerson terminates in at most φ^* augmentations, where φ^* is the maximum flow value.

(e) The Edmonds–Karp algorithm is strongly polynomial.

Justify each answer with one or two sentences.

Exercise 22. Irrational capacities — non-termination of Ford–Fulkerson. Let $\phi = (1 + \sqrt{5})/2$ be the golden ratio. Consider the flow network with nodes $\{s, a, b, c, d, t\}$ and the following arc capacities designed to make Ford–Fulkerson cycle: $k_{sa} = 1$, $k_{sd} = 1$, $k_{ab} = \phi$, $k_{bc} = 1$, $k_{cd} = \phi$, $k_{at} = 1$, $k_{bt} = \phi$, $k_{ct} = 1$, $k_{dt} = \phi$. (This is a schematic version of the Zwick–Papadimitriou example.)

(a) Explain informally why, with an adversarial path-selection strategy and irrational capacities, the sequence of flow values can converge to a limit strictly less than the true maximum flow.

(b) Why does Edmonds–Karp avoid this problem?

(c) What is the practical implication for implementations of Ford–Fulkerson?

Note: exercises in Parts 6–7 on min-cost flow algorithms go beyond the main text and are provided for advanced reading.

Exercise 23. Consider the flow network with nodes $\{s, a, b, t\}$, arc capacities and unit costs: arc (s, a) with $k = 5$, $c = 2$; arc (s, b) with $k = 4$, $c = 3$; arc (a, t) with $k = 4$, $c = 1$; arc (b, t) with $k = 3$, $c = 2$; arc (a, b) with $k = 2$, $c = 1$.

(a) Formulate the min-cost flow problem of sending $k = 4$ units from s to t as a linear program. Write all decision variables, the objective, the supply/demand constraints, and the capacity constraints.

(b) Find the min-cost flow of value 4 by inspection or by solving the LP, and state the cost.

(c) Is there a cheaper way to send 4 units if the capacity of arc (a, b) is increased to 4? Verify.

Exercise 24. Consider the flow network with nodes $\{s, a, b, c, t\}$, arc capacities and unit costs: (s, a): $k = 6$, $c = 1$; (s, b): $k = 5$, $c = 2$; (a, c): $k = 4$, $c = 3$; (b, c): $k = 3$, $c = 1$; (a, t): $k = 3$, $c = 4$; (c, t): $k = 6$, $c = 2$.

(a) Write the LP formulation of the min-cost flow problem for a required flow value of $k = 5$.

(b) Identify the cheapest s-t path (in terms of cost per unit of flow) and use it to route as much flow as possible.

(c) Continue routing flow along successive cheapest paths until $k = 5$ units have been sent. State the total cost.

Exercise 25. Prove that the constraint matrix of the min-cost flow LP (with flow-conservation equalities and capacity bounds) is totally unimodular. *Hint:* Show that it is the node-arc incidence matrix of a directed graph, and recall that such matrices are always TU.

Exercise 26. LP formulation of min-cost flow. Let $G = (V, A)$ be a directed graph. Each arc $(i, j) \in A$ has capacity $u_{ij} \geq 0$ and cost $c_{ij} \in \mathbb{R}$. Each node i has a supply/demand value b_i (positive = supply, negative = demand, $\sum_i b_i = 0$).

(a) Write the standard LP formulation of the min-cost flow problem in matrix form $\min\{c^\top x : Ex = b,\ 0 \leq x \leq u\}$, where E is the node-arc incidence matrix.

(b) Explain why an optimal basic feasible solution of this LP corresponds to a spanning tree of G (the *network simplex* basis structure).

(c) State the dual of this LP and interpret the dual variables as *node potentials*.

Exercise 27. SSSP as min-cost flow. Let $G = (V, A, w)$ be a directed graph with non-negative arc weights w_{ij} and a designated source s. Show how to reduce the Single-Source Shortest Path (SSSP) problem to a min-cost flow problem.

(a) For each node $t \neq s$, describe a min-cost flow network whose optimal cost equals $d(s, t)$ (the shortest-path distance from s to t).

(b) Set up the supply/demand values, arc capacities, and arc costs explicitly for a small example: $V = \{s, a, b, t\}$, $w_{sa} = 2$, $w_{sb} = 5$, $w_{ab} = 1$, $w_{at} = 4$, $w_{bt} = 1$.

(c) Solve the resulting min-cost flow LP and verify the answer against Dijkstra's algorithm applied directly to G.

Exercise 28. Explain why the *successive shortest paths* algorithm for min-cost flow generalises Dijkstra's algorithm, in the sense that each augmentation routes flow along a shortest (minimum-cost) path in the residual network. What role do *reduced costs* (using node potentials) play in keeping all arc costs non-negative in the residual network?

Exercise 29. Bipartite matching as max-flow. Let $G = (U \cup W, E)$ be a bipartite graph with $|U| = |W| = n$.

(a) Construct a directed flow network $G' = (V', A')$ as follows: add a supersource s and a supersink t; orient every edge $\{u, w\} \in E$ from $u \in U$ to $w \in W$; add arcs (s, u) for all $u \in U$ and arcs (w, t) for all $w \in W$; set all arc capacities to 1.

(b) Show that every integer-valued s-t flow of value k in G' corresponds to a matching of size k in G, and vice versa.

(c) Conclude that the maximum matching in G equals the maximum s-t flow in G'.

(d) Apply this construction to the bipartite graph with $U = \{u_1, u_2, u_3\}$, $W = \{w_1, w_2, w_3\}$, and edges u_1w_1, u_1w_2, u_2w_2, u_2w_3, u_3w_1, u_3w_3. Find the maximum matching by running Ford–Fulkerson on G'.

Exercise 30. Kőnig's theorem via max-flow min-cut. Let $G = (U \cup W, E)$ be a bipartite graph and let G' be the flow network constructed in the previous exercise.

(a) State Kőnig's theorem: in a bipartite graph, the size of the maximum matching equals the size of the minimum vertex cover.

(b) Using the Max-Flow Min-Cut Theorem applied to G', show that the minimum cut of G' corresponds to a minimum vertex cover of G.

(c) Illustrate with the bipartite graph from the previous exercise: find a minimum vertex cover and verify it has the same size as the maximum matching.

Exercise 31. Assignment problem as min-cost flow. The assignment problem asks for a minimum-cost perfect matching in a complete bipartite graph $K_{n,n}$ with edge costs c_{ij}.

(a) Model the assignment problem as a min-cost flow problem: describe the flow network, the supply/demand at each node, the arc capacities, and the arc costs.

(b) Write the LP formulation and note that the constraint matrix is totally unimodular, so an integer optimum always exists.

(c) Consider $n = 3$ with cost matrix $\begin{pmatrix} 4 & 2 & 8 \\ 3 & 9 & 1 \\ 7 & 6 & 5 \end{pmatrix}$ (rows = workers, columns = tasks). Formulate the min-cost flow network explicitly and find the optimal assignment by solving the LP or by inspection.

Exercise 32. A shipping company has three warehouses W_1, W_2, W_3 with supplies $10, 15, 20$ units respectively, and four customers C_1, C_2, C_3, C_4 with demands $8, 12, 18, 7$ units respectively. Unit shipping costs c_{ij} are given by the matrix

$$\begin{pmatrix} 2 & 3 & 1 & 5 \\ 4 & 1 & 6 & 2 \\ 3 & 5 & 2 & 4 \end{pmatrix}$$

(rows = warehouses, columns = customers).

(a) Model this as a min-cost flow problem by constructing a bipartite flow network with one node per warehouse (supply) and one node per customer (demand), plus a supersource and supersink if needed.

(b) Write the LP and compute its optimal solution (the transportation plan of minimum cost).

(c) What is the minimum total shipping cost?

Exercise 33. Consider the max-flow problem on the network with nodes $\{s, a, b, t\}$ and arc capacities $k_{sa} = 5$, $k_{sb} = 3$, $k_{ab} = 2$, $k_{at} = 4$, $k_{bt} = 3$.

(a) Write the ILP formulation: maximise $f_{sa} + f_{sb}$ subject to flow conservation and capacity constraints, with $f_e \in \mathbb{Z}_{\geq 0}$.

(b) Write the LP relaxation (drop the integrality constraints).

(c) Argue that the node-arc incidence matrix of this network is TU, and therefore the LP relaxation has an integer optimal solution.

(d) Solve the LP relaxation and verify that the solution is integral.

Exercise 34. The node-arc incidence matrix B of a directed graph has one row per node and one column per arc; entry $B_{v,e} = +1$ if v is the tail of arc e, $B_{v,e} = -1$ if v is the head, and 0 otherwise.

(a) Prove that B is totally unimodular by showing that every square submatrix has determinant in $\{-1, 0, +1\}$. *Hint:* use induction on the size of the submatrix, expanding along a column that has at most one non-zero entry, or use the characterisation of TU via the Ghouila-Houri condition.

(b) Explain why TU of B implies that the max-flow LP always has an integer optimal solution when capacities are integers.

Exercise 35. Let f^* be a maximum s-t flow in a network G and let (S^*, T^*) be the minimum cut defined by the set of nodes reachable from s in the residual network G_{f^*}.

(a) Prove directly (without invoking the Max-Flow Min-Cut Theorem) that every arc (u, v) with $u \in S^*$ and $v \in T^*$ is saturated in f^*.

(b) Prove that every arc (u, v) with $u \in T^*$ and $v \in S^*$ carries zero flow in f^*.

(c) Conclude that $|f^*| = \mathrm{cap}(S^*, T^*)$, thereby completing a self-contained proof of the Max-Flow Min-Cut Theorem.

Exercise 36. Consider a flow network G where every arc has capacity 1.

(a) What can you say about the structure of the maximum flow?

(b) Show that in this case Ford–Fulkerson terminates in at most $|A|$ augmentations (each augmentation saturates at least one arc).

(c) Show that the max-flow equals the maximum number of arc-disjoint s-t paths in G.

(d) Give an example with 4 nodes where the maximum number of arc-disjoint s-t paths is strictly less than the number of s-t paths in G.

Exercise 37. A directed graph $G = (V, A)$ is said to be *k-arc-connected* if removing any $k - 1$ arcs leaves the graph strongly connected.

(a) Formulate the problem of finding the minimum number of arcs whose removal disconnects s from t as a max-flow problem.

(b) Use Menger's theorem (arc version) to state that the minimum number of arcs in an s-t separating set equals the maximum number of arc-disjoint s-t paths.

(c) Verify Menger's theorem on the network with nodes $\{s, a, b, t\}$ and arcs $(s, a), (s, b), (a, t), (b, t), (a, b)$, all with capacity 1.

Exercise 38. Adding a node with bounded throughput. Suppose in a flow network you want to model a node v that can process at most B units of flow (a *node capacity*).

(a) Describe a standard graph transformation that replaces v with two nodes v^{in} and v^{out} connected by a single arc of capacity B, and redirects all incoming and outgoing arcs of v appropriately.

(b) Show that the resulting network has the same max s-t flow as the original (with node capacity at v).

(c) Apply this transformation to a network with nodes $\{s, a, b, t\}$, arc capacities $k_{sa} = 5$, $k_{sb} = 4$, $k_{at} = 6$, $k_{bt} = 3$, and node capacity $B_a = 3$ at node a. Find the max flow.

Exercise 39. Ford–Fulkerson on a network with a unique augmenting path. Consider the flow network with nodes $\{s, a, t\}$ and arc capacities $k_{sa} = 10$, $k_{at} = 7$.

(a) How many augmenting paths does Ford–Fulkerson find? State them and their bottleneck capacities.

(b) What is the maximum flow value? Identify the bottleneck arc.

(c) What is the minimum cut? Verify that its capacity equals the max flow.

Exercise 40. Parallel arcs and multiple source/sink. Consider a logistics network with two sources s_1, s_2 and two sinks t_1, t_2. Supplies: s_1 produces 6 units, s_2 produces 4 units. Demands: t_1 needs 5 units, t_2 needs 5 units. Arc capacities: $k_{s_1a} = 5$, $k_{s_2a} = 4$, $k_{s_1b} = 3$, $k_{s_2b} = 3$, $k_{at_1} = 6$, $k_{at_2} = 4$, $k_{bt_1} = 3$, $k_{bt_2} = 5$.

(a) Add a supersource S with arcs (S, s_1) of capacity 6 and (S, s_2) of capacity 4, and a supersink T with arcs (t_1, T) of capacity 5 and (t_2, T) of capacity 5.

(b) Find the maximum S-T flow in the extended network.

(c) Determine whether all supply and demand requirements can be met simultaneously.

Exercise 41. Min-cut duality in LP. Consider the max-flow LP:

$$\max\ v \quad \text{s.t.} \quad \sum_{j:(i,j)\in A} f_{ij} - \sum_{j:(j,i)\in A} f_{ji} = \begin{cases} v & i = s \\ -v & i = t \\ 0 & \text{otherwise,} \end{cases} \qquad 0 \le f_{ij} \le k_{ij}.$$

(a) Write the dual of this LP.

(b) Interpret the dual variables as cut indicator variables and show that the dual is equivalent to the minimum cut problem.

(c) Explain how LP strong duality implies the Max-Flow Min-Cut Theorem.

Exercise 42. Sensitivity analysis. Consider the flow network with nodes $\{s, a, b, t\}$ and arc capacities $k_{sa} = 4$, $k_{sb} = 3$, $k_{ab} = 2$, $k_{at} = 3$, $k_{bt} = 4$. Suppose the maximum flow is f^*.

(a) By how much can the capacity of arc (s, a) be decreased before the maximum flow value decreases?

(b) By how much can the capacity of arc (a, b) be increased before the maximum flow value increases?

(c) Which single arc, if its capacity is increased by 1, increases the maximum flow value by the largest amount?

Exercise 43. Max-flow with lower bounds. Suppose each arc (i, j) has both a lower bound $\ell_{ij} \geq 0$ and an upper bound $k_{ij} \geq \ell_{ij}$. A feasible flow must satisfy $\ell_{ij} \leq f_{ij} \leq k_{ij}$.

(a) Describe the standard reduction to a max-flow problem with only upper bounds (the "circulation with lower bounds" trick): define a modified network where the transformed flow $f'_{ij} = f_{ij} - \ell_{ij}$ satisfies $0 \leq f'_{ij} \leq k_{ij} - \ell_{ij}$, and adjust the supply/demand at each node accordingly.

(b) Apply this reduction to the network with nodes $\{s, a, b, t\}$ and arc data: (s, a): $\ell = 2$, $k = 5$; (a, b): $\ell = 1$, $k = 3$; (b, t): $\ell = 2$, $k = 4$; (s, b): $\ell = 0$, $k = 3$; (a, t): $\ell = 1$, $k = 4$.

(c) Find a feasible flow in the original network satisfying all lower bounds, or determine that none exists.

Exercise 44. Circulation and feasibility with demands. Consider a directed graph with nodes $\{1, 2, 3, 4\}$ and arc capacities $k_{12} = 4$, $k_{23} = 5$, $k_{34} = 3$, $k_{41} = 6$, $k_{13} = 2$. Node demands: $d_1 = -4$ (supply 4), $d_2 = 1$, $d_3 = 2$, $d_4 = 1$.

(a) Verify that $\sum_i d_i = 0$ (a necessary condition for feasibility).

(b) Add a supersource S and supersink T to convert demands into supply/demand arcs and find whether a feasible circulation exists.

(c) If feasible, exhibit a feasible flow.

Exercise 45. Project scheduling as min-cost flow. A project consists of n tasks. Each task i has a normal duration d_i and can be *crashed* (accelerated) at a cost of c_i per unit of time saved, down to a minimum duration $d_i^{\min}$.

(a) Describe informally how the problem of minimising total crashing cost subject to a project deadline can be modelled as a min-cost flow problem on the project's activity-on-arc network.

(b) In a simple two-task network with nodes $\{s, a, t\}$, task $s \to a$ with $d = 3$, $d^{\min} = 1$, $c = 5$, and task $a \to t$ with $d = 4$, $d^{\min} = 2$, $c = 3$: formulate the min-cost flow LP for reducing the project duration by 2 time units.

Exercise 46. Ford–Fulkerson: integer capacities and integrality theorem. Prove the following:

(a) If all arc capacities are positive integers, then Ford–Fulkerson terminates in at most φ^* augmentations, where φ^* is the value of the maximum flow.

(b) Every augmentation in this case increases the flow value by at least 1.

(c) Conclude the *integrality theorem*: there exists an optimal s-t flow that is integer-valued whenever all capacities are integers.

Exercise 47. Equivalent flow decompositions. Consider the flow network with nodes $\{s, a, b, c, t\}$ and arc capacities $k_{sa} = 4$, $k_{sb} = 3$, $k_{ac} = 2$, $k_{bc} = 3$, $k_{ct} = 4$, $k_{at} = 2$, $k_{bt} = 1$. A maximum flow f^* is given by $f_{sa} = 3$, $f_{sb} = 2$, $f_{ac} = 1$, $f_{bc} = 3$, $f_{ct} = 4$, $f_{at} = 2$, $f_{bt} = 0$.

(a) Verify that f^* is feasible and state its value.

(b) Decompose f^* into a sum of s-t path flows using the flow decomposition algorithm (repeatedly find a path with positive flow and subtract).

(c) Is the decomposition unique? Provide two distinct decompositions if possible.

Exercise 48. Augmenting paths and cycle flows.

(a) Explain why, in the Ford–Fulkerson algorithm, augmenting along a cycle in the residual network does not change the flow value but may change the flow on individual arcs.

(b) Give an example of a flow network and a feasible flow f such that the residual network G_f contains a directed cycle, yet f is maximum.

(c) Why does the presence of a directed cycle in G_f imply that the current flow is *not* a min-cost flow (assuming strictly positive costs)?

Exercise 49. Shortest augmenting paths and BFS layers. Consider the flow network with nodes $\{s, a, b, c, d, t\}$ and arc capacities $k_{sa} = 5$, $k_{sb} = 4$, $k_{ac} = 3$, $k_{bd} = 3$, $k_{bc} = 2$, $k_{cd} = 2$, $k_{ct} = 4$, $k_{dt} = 5$. Apply the Edmonds–Karp algorithm (BFS augmenting paths):

(a) Perform BFS on the initial residual network (= the original network, since $f = 0$) and identify the shortest s-t path.

(b) Augment along this path, update the residual network, and repeat BFS.

(c) After each augmentation, record the BFS distance $d(s, t)$ in the residual network and verify that it is non-decreasing.

(d) Terminate when no augmenting path exists and state the max flow.

Exercise 50. Network flow modelling: traffic routing. A city road network can be modelled as a directed graph where each arc (i, j) has a capacity k_{ij} (maximum vehicles per hour).

(a) Model the problem of routing the maximum number of vehicles per hour from district s to district t as a max-flow problem.

(b) Explain why the minimum cut in this model corresponds to the *bottleneck* set of roads whose congestion limits overall throughput.

(c) In a small example with nodes $\{s, a, b, c, t\}$ and capacities $k_{sa} = 10$, $k_{sc} = 8$, $k_{ab} = 6$, $k_{cb} = 5$, $k_{at} = 4$, $k_{bt} = 9$, $k_{ct} = 3$: find the max flow and the minimum cut, and identify the bottleneck roads.

Exercise 51. Planar graphs and dual cuts. In a planar flow network, every s-t cut corresponds to a path in the *planar dual* graph from the face containing s to the face containing t.

(a) State this planar duality result precisely.

(b) Explain how, for planar graphs, computing the minimum cut reduces to computing a shortest path in the dual graph.

(c) Verify the duality on the network with nodes $\{s, a, b, t\}$, arcs (s, a), (s, b), (a, t), (b, t), (a, b), all with unit capacities, by drawing (describing) the planar dual and finding the minimum-cost path in it.

Exercise 52. Hospital-resident matching as max-flow. Consider a hospital-resident assignment problem with 3 residents $\{r_1, r_2, r_3\}$ and 2 hospitals $\{h_1, h_2\}$. Hospital h_1 has capacity 2 and hospital h_2 has capacity 2. Each resident has applied to a subset of hospitals: $r_1 \to \{h_1, h_2\}$, $r_2 \to \{h_1\}$, $r_3 \to \{h_2\}$.

(a) Model this as a max-flow problem: add a supersource s with arcs to each resident of capacity 1, add arcs from residents to the hospitals they applied to (capacity 1), and add arcs from hospitals to a supersink t equal to the hospital capacity.

(b) Find the maximum flow and the resulting assignment.

(c) Interpret the minimum cut: which hospitals or residents form the bottleneck?

Exercise 53. Supply chain as min-cost flow. A manufacturer has two plants P_1 and P_2 producing 30 and 20 units per day. There are three distribution centres D_1, D_2, D_3 with demands 20, 15, 15. Shipping costs and capacities (units per day) are: (P_1, D_1): cost 2, capacity 25; (P_1, D_2): cost 4, capacity 20; (P_2, D_2): cost 3, capacity 15; (P_2, D_3): cost 1, capacity 20; (P_1, D_3): cost 6, capacity 10.

(a) Verify that total supply equals total demand.

(b) Model as a min-cost flow problem (no supersource/sink needed).

(c) Find the minimum-cost shipping plan.

Exercise 54. Inverse max-flow problem. Given a flow network $G = (V, A, k)$ and a target flow value v, the *inverse max-flow problem* asks: what is the minimum total increase in arc capacities needed so that the maximum flow becomes at least v?

(a) Explain why this problem reduces to finding a minimum-cost s-t cut augmentation.

(b) Formulate the inverse problem as an LP.

(c) Solve the inverse problem for the network with nodes $\{s, a, b, t\}$, current capacities $k_{sa} = 2$, $k_{sb} = 2$, $k_{at} = 2$, $k_{bt} = 2$, $k_{ab} = 1$, and target value $v = 5$. (All capacity increases have unit cost.)

Exercise 55. Parametric max-flow. Consider a family of flow networks $G(\lambda)$ where the capacity of a single arc (s, a) is $k_{sa}(\lambda) = \lambda$ and all other capacities are fixed: nodes $\{s, a, b, t\}$, $k_{ab} = 3$, $k_{sb} = 4$, $k_{at} = 5$, $k_{bt} = 2$.

(a) Express the maximum s-t flow value $\varphi^*(\lambda)$ as a function of $\lambda \geq 0$.

(b) For which values of λ does increasing $k_{sa}(\lambda)$ by 1 actually increase $\varphi^*(\lambda)$ by 1, and for which values does it have no effect?

(c) Identify the threshold λ^* beyond which the arc (s, a) is no longer in any minimum cut, and explain what this implies for the minimum cut structure.

CHAPTER 12

Matching

Matching theory connects combinatorial optimisation with real-world assignment and pairing problems.

In the previous chapters we studied how to push flow through a capacitated network. We now turn to a closely related but structurally distinct question: given an undirected graph, what is the largest set of edges we can select so that no vertex is used twice? This is the **matching problem**, one of the cornerstones of combinatorial optimisation. Matching models arise whenever we need to pair objects in a one-to-one fashion—workers to jobs, organs to patients, or dominoes to grid squares.

The theory is also elegant from a mathematical standpoint: Berge's theorem gives a clean optimality condition, and for bipartite graphs the matching problem reduces to the network flow machinery developed in chapter 11. For general (non-bipartite) graphs, Edmonds' celebrated *blossom algorithm* is needed, but we will not cover it in detail.

Road map.

1. Matching definitions: matchings, exposed nodes, perfect and maximum matchings (section 12.1).
2. Alternating and augmenting paths (section 12.2).
3. Berge's theorem: the optimality condition (section 12.3).
4. Bipartite matching via maximum flow and the labelling algorithm (section 12.4).
5. König's theorem: maximum matching equals minimum vertex cover (section 12.6).
6. Applications: assignment, domino tiling, scheduling (section 12.7).
7. Summary and comparison (section 12.8).

12.1 Matching Definitions

Italian: accoppiamento.

Definition 12.1.1 (Matching)**.** Let $G = (V, E)$ be an undirected graph. A **matching** is a subset $M \subseteq E$ such that no two edges in M share an endpoint; equivalently, every vertex $v \in V$ is incident to *at most one* edge of M.

Definition 12.1.2 (Exposed and saturated nodes)**.** Given a matching M:

- A vertex v is **saturated** (or **covered**) by M if there exists an edge $e \in M$ incident to v.
- A vertex v is **exposed** (or **uncovered**, **free**) if no edge of M is incident

to v.

Definition 12.1.3 (Perfect matching)**.** A matching M is **perfect** if every vertex of G is saturated. A necessary condition for the existence of a perfect matching is that $|V|$ is even, since each edge of M covers exactly two vertices. This condition is necessary but *not* sufficient.

Definition 12.1.4 (Maximum matching)**.** A matching M is a **maximum (cardinality) matching** if there is no matching M' with $|M'| > |M|$.

In a weighted graph one may seek the maximum weight *matching rather than maximum cardinality.*

Definition 12.1.5 (Weighted matching)**.** In a weighted graph $G = (V, E)$ with edge weights $w\colon E \to \mathbb{R}$, the **weight** of a matching M is $w(M) = \sum_{e \in M} w(e)$. A **maximum weight matching** maximises $w(M)$ over all matchings. When all weights equal 1, this reduces to maximum cardinality matching.

Remark 12.1.6. If weights are non-negative and we seek to minimise $w(M)$, the trivial solution $M = \emptyset$ is optimal. Hence we typically *maximise* weight. For the assignment problem (see section 12.7) we seek a perfect matching of maximum weight.

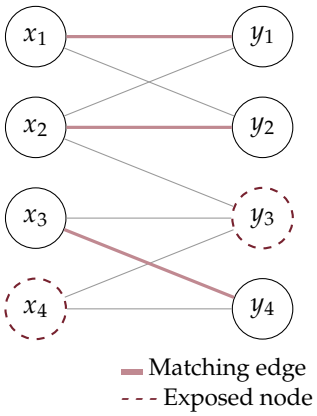


Figure 12.1: A bipartite graph with partition $X = \{x_1, x_2, x_3, x_4\}$ and $Y = \{y_1, y_2, y_3, y_4\}$. The matching $M = \{x_1y_1,\ x_2y_2,\ x_3y_4\}$ is shown with thick accent edges. Nodes x_4 and y_3 are exposed (dashed outline).

12.2 Alternating and Augmenting Paths

Alternating paths are the key tool for improving a matching.

Definition 12.2.1 (Alternating path)**.** Given a matching M, an **alternating path** is a path whose edges alternate between edges in M and edges in $E \setminus M$.

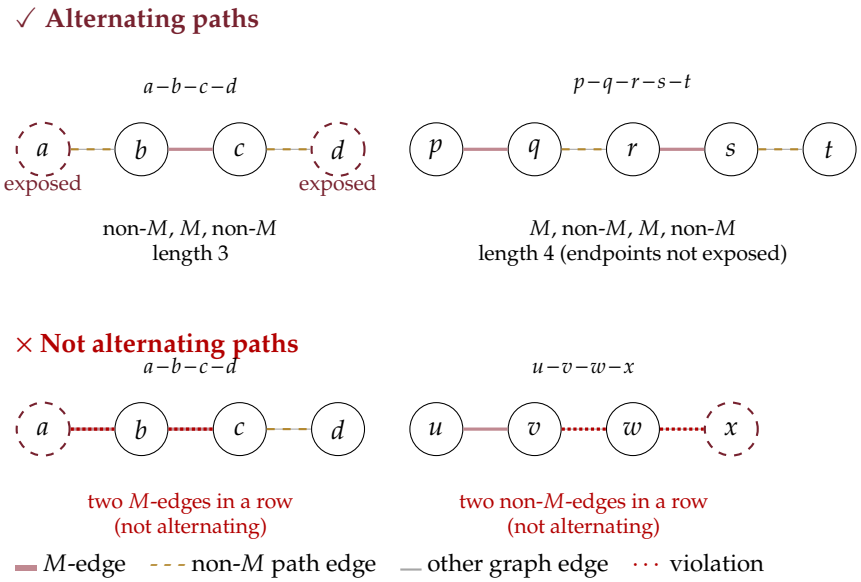


Figure 12.2: Examples of alternating and non-alternating paths. An alternating path must *strictly alternate* between M-edges and non-M-edges. Dashed circles denote exposed nodes.

Definition 12.2.2 (Augmenting path)**.** An **augmenting path** (with respect to M) is an alternating path whose *two endpoints are both exposed*. Since the path starts and ends at an exposed node, its first and last edges belong to $E \setminus M$. Consequently, an augmenting path has odd length $2k+1$ and contains $k+1$ edges from $E \setminus M$ and k edges from M.

Theorem 12.2.3 (Augmentation)**.** *If P is an augmenting path with respect to a matching M, then $M' = M \oplus E(P)$ (the symmetric difference) is a matching with $|M'| = |M| + 1$.*

Formal details — Proof of the augmentation property

Let $P = v_0\, v_1\, v_2 \cdots v_{2k+1}$ be an augmenting path. The edges of P alternate:

$$\underbrace{v_0v_1}_{\notin M},\ \underbrace{v_1v_2}_{\in M},\ \underbrace{v_2v_3}_{\notin M},\ \ldots,\ \underbrace{v_{2k}v_{2k+1}}_{\notin M}.$$

The symmetric difference $M' = M \oplus E(P)$ removes the k edges of P that were in M and inserts the $k+1$ edges that were not. It remains a valid matching: at each interior vertex v_i ($1 \le i \le 2k$), exactly one of its two incident path-edges is in M', so the degree-at-most-one constraint is preserved. At the endpoints v_0 and v_{2k+1}, which were previously exposed, exactly one path-edge enters M'. Hence M' is a matching with $|M'| = |M| - k + (k+1) = |M| + 1$.

12.3 Berge's Theorem

Berge's theorem is the matching analogue of the max-flow min-cut duality.

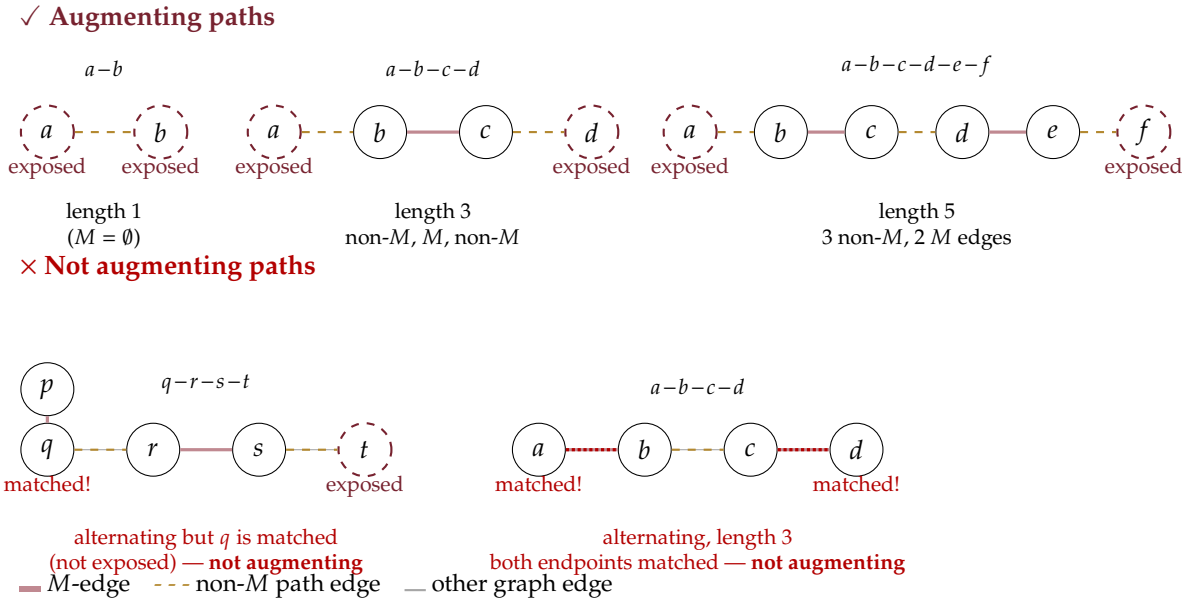


Figure 12.3: Examples of augmenting and non-augmenting paths. An augmenting path must be alternating *and* have both endpoints exposed. Row 1: valid augmenting paths of lengths 1, 3, and 5. Row 2: alternating paths that fail the exposed-endpoint condition.

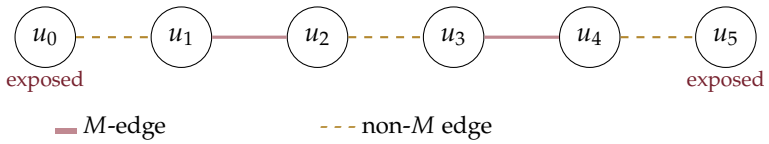


Figure 12.4: An augmenting path $P = u_0\,u_1\,u_2\,u_3\,u_4\,u_5$ of length 5. It contains 3 dashed non-M edges and 2 solid M-edges. Both endpoints u_0 and u_5 are exposed. After augmenting, $M' = M \oplus E(P)$ has $|M| + 1$ edges.

The following classical result gives a necessary and sufficient condition for a matching to be of maximum cardinality.

Theorem 12.3.1 (Berge, 1957)**.** *A matching M in a graph $G = (V, E)$ is a* ***maximum matching*** *if and only if there is no augmenting path with respect to M.*

Formal details — Proof of Berge's theorem

($\Rightarrow$) Necessity. Suppose an augmenting path P exists. By theorem 12.2.3, $M \oplus E(P)$ is a matching of size $|M| + 1$, contradicting the maximality of M. Hence no augmenting path can exist.

($\Leftarrow$) Sufficiency. We prove the contrapositive: if M is *not* maximum, then an augmenting path exists.

Let M^* be a matching with $|M^*| > |M|$. Consider the symmetric difference $H = M \oplus M^*$, viewed as a subgraph of G. Every vertex in H has degree at most 2 (at most one edge from M and at most one from M^*), so each connected component of H is either:

1. an **even cycle**, alternating between M-edges and M^*-edges (equal count of each), or
2. a **path**, alternating between M-edges and M^*-edges.

Since $|M^*| > |M|$, the total number of M^*-edges in H exceeds the number of M-edges. Even cycles contribute equally, so at least one path component must have more M^*-edges than M-edges. Such a path has odd length, starts and ends with M^*-edges, and its endpoints are exposed with respect to M. This path is therefore an augmenting path for M.

Intermezzo — Claude Berge (1926–2002)

Claude Berge was a French mathematician who made fundamental contributions to combinatorics and graph theory. His 1957 theorem on augmenting paths became the basis for virtually all matching algorithms. Berge was also a founding member of the *Oulipo* literary group and an accomplished sculptor—a remarkable blend of mathematical rigour and artistic creativity.

Berge's theorem immediately yields a generic algorithm for maximum matching:

Observation 12.3.2 (Generic augmenting-path algorithm). Start with any matching M (possibly $M = \emptyset$). Repeatedly search for an augmenting path P; if one is found, set $M \leftarrow M \oplus E(P)$. When no augmenting path exists, M is maximum by theorem 12.3.1. Since each augmentation increases $|M|$ by 1, and $|M| \leq \lfloor |V|/2 \rfloor$, the algorithm terminates in at most $\lfloor |V|/2 \rfloor$ augmentations.

The key question is: *how do we find augmenting paths efficiently?* For bipartite graphs the answer is elegant and ties back to network flows.

12.4 Bipartite Matching and Maximum Flow

Bipartite matching reduces to max flow in a network with unit capacities.

A graph $G = (V, E)$ is **bipartite** if its vertex set can be partitioned as $V = X \cup Y$ with $X \cap Y = \emptyset$ and every edge having one endpoint in X and one in Y. (Recall from chapter 8 that a graph is bipartite if and only if it contains no odd cycle.)

12.4.1 Reduction to maximum flow

Given a bipartite graph $G = (X \cup Y, E)$, we construct a flow network $\hat{G} = (\hat{V}, \hat{A})$ as follows:

1. Add a **super-source** s and a **super-sink** t, so $\hat{V} = X \cup Y \cup \{s, t\}$.
2. For each $x \in X$, add a directed arc (s, x) with capacity 1.
3. For each $y \in Y$, add a directed arc (y, t) with capacity 1.
4. For each edge $\{x, y\} \in E$, add a directed arc (x, y) with capacity 1.

Theorem 12.4.1 (Matching–flow equivalence)**.** *The value of the maximum flow in $\hat{G}$ equals the size of the maximum matching in G.*

Formal details — Proof sketch

From matching to flow. Given a matching M in G, define a flow: for each edge $\{x, y\} \in M$, send one unit along $s \to x \to y \to t$. The capacity constraints are satisfied (all capacities are 1 and each vertex has at most one matched edge). The flow value equals $|M|$.

From flow to matching. Since all capacities are integers, by the integrality theorem (chapter 11), there exists an integer maximum flow. All flow values are 0 or 1. The set $M = \{\{x, y\} : x_{xy} = 1\}$ is a matching (each $x \in X$ has at most one unit entering from s, and each $y \in Y$ has at most one unit leaving to t). Its size equals the flow value.

Since the incidence matrix of a bipartite graph is totally unimodular (see chapter 11), the LP relaxation of the matching ILP always yields an integer optimal solution, confirming the equivalence.

Since the maximum flow value in a network with unit capacities is at most $\min(|X|, |Y|)$, and each augmentation in Ford–Fulkerson takes $O(|E|)$ time, the overall complexity is $O(|V| \cdot |E|)$.

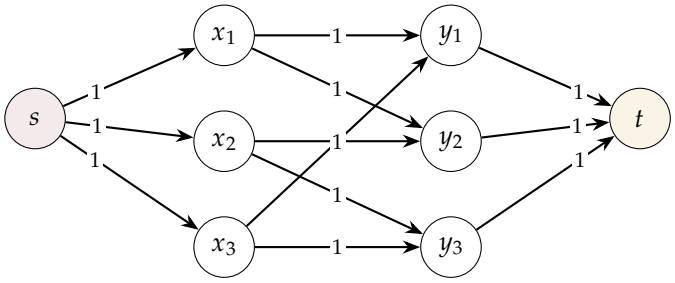


Figure 12.5: Flow network $\hat{G}$ constructed from a bipartite graph with $X = \{x_1, x_2, x_3\}$ and $Y = \{y_1, y_2, y_3\}$. Every arc has capacity 1. The maximum flow (here 3) gives the maximum matching size.

12.4.2 Labelling algorithm for bipartite matching

Instead of building the full flow network, we can work directly on the bipartite graph using a **labelling procedure**—essentially Ford–Fulkerson specialised to bipartite matching.

The idea is to grow alternating trees from exposed nodes in X, trying to reach an exposed node in Y. Labels record the predecessor of each node so that the augmenting path can be reconstructed.

Algorithm 11: Labelling algorithm for maximum bipartite matching

Input: Bipartite graph $G = (X \cup Y, E)$; initial matching M (possibly $\emptyset$)
Output: Maximum matching M

```
repeat
    foreach node x ∈ X do
        if x is exposed then label x with ⟨∗⟩
    Initialise queue Q with all labelled, unscanned nodes in X
    found ← false
    while Q ≠ ∅ and not found do
        Dequeue node x from Q
        foreach y ∈ Y adjacent to x via edge {x, y} ∉ M do
            if y is not yet labelled then
                Label y with ⟨x⟩
                if y is exposed then
                    found ← true
                    Reconstruct augmenting path from y via labels
                    M ← M ⊕ E(P)
                    Remove all labels
                else
                    Let x′ be the mate of y in M
                    Label x′ with ⟨y⟩
                    Enqueue x′ into Q
until no augmenting path is found
return M
```

Remark 12.4.2 (Correctness and complexity). Each execution of the outer loop either finds an augmenting path (increasing $|M|$ by 1) or certifies that none exists, proving M is maximum by Berge's theorem. Each inner BFS visits each edge at most once, taking $O(|E|)$ time. There are at most $\lfloor |V|/2 \rfloor$ augmentations, so the total time is $O(|V| \cdot |E|)$.

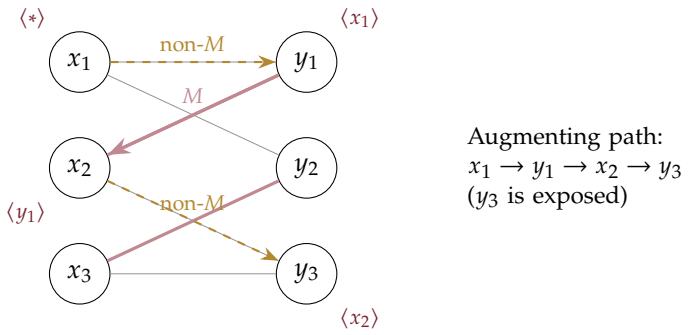


Figure 12.6: One step of the labelling algorithm. The current matching is $M = \{x_2y_1,\ x_3y_2\}$. Starting from the exposed node x_1, labels propagate along the alternating path $x_1 \to y_1 \to x_2 \to y_3$, reaching the exposed node y_3. After augmenting, $M' = \{x_1y_1,\ x_2y_3,\ x_3y_2\}$.

12.5 Hall's Theorem

Hall's theorem gives a necessary and sufficient condition for a bipartite graph to have a matching that covers one side completely.

Let $G = (X \cup Y, E)$ be a bipartite graph. For a subset $S \subseteq X$, define the **neighbourhood** of S as

$$N(S) = \{\, y \in Y : \exists\, x \in S,\ \{x, y\} \in E \,\}.$$

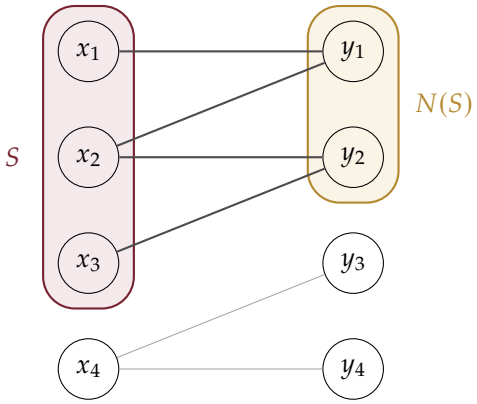


Figure 12.7: The neighborhood $N(S)$ of a subset $S \subseteq X$ in a bipartite graph. Here, the subset $S = \{x_1, x_2, x_3\}$ (left highlighted region) has neighborhood $N(S) = \{y_1, y_2\}$ (right highlighted region). Since $|N(S)| = 2 < 3 = |S|$, the subset S violates Hall's condition, making a matching that saturates X impossible.

Theorem 12.5.1 (Hall's Marriage Theorem). *A bipartite graph $G = (X \cup Y, E)$ has a matching that saturates every vertex of X (a **perfect matching on** X) if and only if*

$$|N(S)| \geq |S| \qquad \text{for every } S \subseteq X.$$

The condition "$|N(S)| \geq |S|$ for all $S \subseteq X$" is called **Hall's condition**.

Formal details — Proof sketch

The forward direction is immediate: a perfect matching injects S into $N(S)$, so $|N(S)| \geq |S|$. The reverse direction is deeper: we derive a contradiction from Hall's condition using König's theorem.

(⇒) Suppose a matching M saturates every $x \in X$. For any $S \subseteq X$, each vertex of S is matched to a distinct vertex in $N(S)$, so $|N(S)| \geq |S|$.

(⇐) Suppose Hall's condition holds but no matching saturates all of X. Then the maximum matching size satisfies $\nu(G) < |X|$. By König's theorem (theorem 12.6.3), $\tau(G) = \nu(G) < |X|$, so there exists a minimum vertex cover C with $|C| < |X|$. Let $S = X \setminus C$ (the X-vertices not in C). Since C is a cover, every neighbour of a vertex in S must lie in $C \cap Y$, so $N(S) \subseteq C \cap Y$. Therefore:

$$|N(S)| \leq |C \cap Y| \leq |C| - |C \cap X| \leq |C| - \big(|X| - |S|\big) = |C| + |S| - |X| < |S|.$$

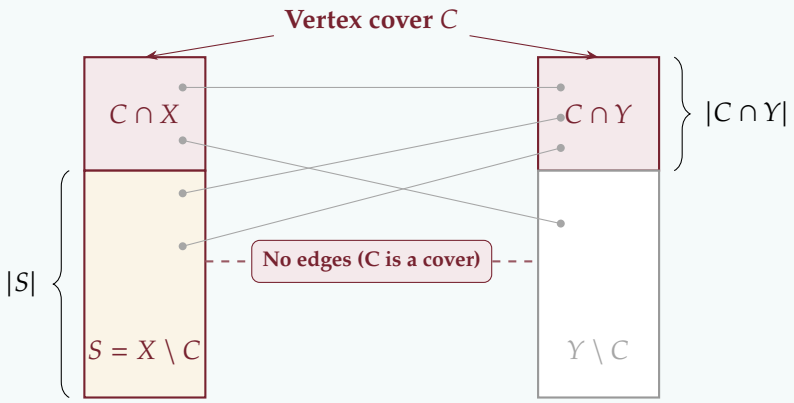


Figure 12.6: Partition structure under a minimum vertex cover C with $|C| < |X|$. Since C covers all edges, no edges can connect $S = X \setminus C$ to $Y \setminus C$. Hence all neighbors of S lie in $C \cap Y$ ($N(S) \subseteq C \cap Y$), meaning $|N(S)| \leq |C \cap Y| < |S|$.

This violates Hall's condition — contradiction.

Corollary 12.5.2 (Regular bipartite graphs). *Every k-regular bipartite graph with $k \geq 1$ has a perfect matching.*

Formal details — Proof sketch

Let $G = (X \cup Y, E)$ be k-regular. Note first that $k|X| = |E| = k|Y|$, so $|X| = |Y|$. For any $S \subseteq X$, count the edges leaving S: there are $k|S|$ such edges, all landing in $N(S)$. Each vertex of $N(S)$ has degree at most k, so $k|S| \leq k|N(S)|$, giving $|N(S)| \geq |S|$. Hall's condition holds, so a perfect

matching on X exists; since $|X| = |Y|$ it is a perfect matching of G.

12.6 König's Theorem

König's theorem is the bipartite counterpart of the max-flow min-cut theorem.

König's theorem is a striking min–max result for bipartite graphs that parallels the Max-Flow Min-Cut theorem of chapter 11.

Definition 12.6.1 (Vertex cover)**.** A **vertex cover** of a graph $G = (V, E)$ is a set $C \subseteq V$ such that every edge in E has at least one endpoint in C. A **minimum vertex cover** is a vertex cover of minimum cardinality.

Observation 12.6.2. For any matching M and any vertex cover C, we have $|M| \leq |C|$: each edge of M requires at least one endpoint in C, and since matching edges share no endpoints, distinct edges demand distinct cover vertices.

Theorem 12.6.3 (König, 1931)**.** *In a bipartite graph, the size of the maximum matching equals the size of the minimum vertex cover:*

$$\nu(G) \;=\; \tau(G),$$

where $\nu(G)$ denotes the maximum matching size and $\tau(G)$ the minimum vertex cover size.

Formal details — Proof via max-flow min-cut

Consider the flow network $\hat{G}$ from the reduction in section 12.4, with one small adjustment for the cut argument: arcs $s \to x$ and $y \to t$ have capacity 1, while arcs $x \to y$ corresponding to original edges have capacity $M = |V| + 1$. This does not change the maximum matching value, because the source arcs already limit the flow to at most $|X|$. By the Max-Flow Min-Cut theorem (chapter 11):

$$\text{max flow} \;=\; \text{min cut}.$$

We showed that the max flow equals $\nu(G)$. It remains to show that the min cut equals $\tau(G)$.

An s-t cut in $\hat{G}$ partitions $\hat{V}$ into sets S (containing s) and $T = \hat{V} \setminus S$ (containing t). Define:

$$X_T = X \cap T, \quad Y_S = Y \cap S.$$

An arc from S to T is one of three types:

1. (s, x) for $x \in X_T$ — capacity 1 each, contributing $|X_T|$;
2. (y, t) for $y \in Y_S$ — capacity 1 each, contributing $|Y_S|$;
3. (x, y) for $x \in X \cap S$, $y \in Y \cap T$ — capacity M each.

Any cut that uses a type-3 arc has capacity at least $M > |V|$, while there is always a cut of capacity at most $|X| \leq |V|$ (put only s on the source side). Hence a minimum cut has no type-3 arcs. Therefore every original edge $\{x, y\}$ must have $x \in X_T$ or $y \in Y_S$, so $C = X_T \cup Y_S$ is a vertex cover. Its cut capacity equals $|X_T| + |Y_S| = |C|$.

Conversely, every vertex cover C defines a cut of capacity $|C|$: put $X \setminus C$ and $Y \cap C$ on the source side, and put the remaining vertices on the sink side. Therefore min cut $= \tau(G)$, and the result follows.

Intermezzo — Dénes König (1884–1944)

Dénes König was a Hungarian mathematician and a pioneer of graph theory. His 1936 textbook *Theorie der endlichen und unendlichen Graphen* was one of the first monographs on the subject. The theorem bearing his name, proved in 1931, is one of the earliest min–max results in combinatorial optimisation and can be seen as a precursor to the duality theory of linear programming.

Remark 12.6.4 (Computing the minimum vertex cover). Once we have a maximum matching M, we can read off a minimum vertex cover directly by a BFS from unmatched X-nodes. Here is the construction: Given a maximum matching M in a bipartite graph $G = (X \cup Y, E)$, the following procedure constructs a minimum vertex cover C of size $|M|$:

1. Let $U \subseteq X$ be the set of exposed (unmatched) vertices of X.
2. Run BFS/DFS from U along *alternating paths*: from an X-vertex follow non-matching edges into Y; from a Y-vertex follow the unique matching edge back into X.
3. Let $Z_X \subseteq X$ be the X-vertices reached and $Z_Y \subseteq Y$ be the Y-vertices reached.
4. Set $C = (X \setminus Z_X) \cup Z_Y$.

Every matching edge has exactly one endpoint in C (so $|C| = |M|$), and C is a vertex cover because any edge not covered by C would yield an augmenting path, contradicting the maximality of M.

12.7 Applications

12.7.1 The assignment problem

The Hungarian algorithm solves the assignment problem in $O(|V|^3)$ time.

The **assignment problem** arises when n workers must be assigned to n jobs in a one-to-one fashion. Given a bipartite graph $G = (X \cup Y, E)$ with $|X| = |Y| = n$ and profit w_{ij} for assigning worker x_i to job y_j, we seek a *perfect matching of maximum weight*. The ILP formulation is:

$$\text{maximize} \quad \sum_{(i,j)\in E} w_{ij}\, z_{ij}$$

$$\text{subject to} \quad \sum_{j:(i,j)\in E} z_{ij} = 1 \quad \forall\, i \in X, \qquad \sum_{i:(i,j)\in E} z_{ij} = 1 \quad \forall\, j \in Y, \qquad z_{ij} \in \{0,1\}.$$

Since the constraint matrix is the incidence matrix of a bipartite graph (which is totally unimodular), the LP relaxation with $z_{ij} \geq 0$ always has an integer optimal solution. The **Hungarian algorithm** (Kuhn, 1955) solves this in $O(n^3)$ time by exploiting the dual structure of the LP.

Remark 12.7.1 (The Hungarian algorithm). The Hungarian algorithm works on the LP *dual* of the assignment problem. Instead of searching the primal for a better matching, it adjusts *dual variables* (node potentials) to create new tight edges, then augments on those. It is usually described for the *minimum-cost* assignment problem. The dual variables—called **node potentials** (or **labels**) u_i for each worker $x_i \in X$ and v_j for each job $y_j \in Y$—

must satisfy $u_i + v_j \le c_{ij}$ for every edge $\{i, j\}$. An edge is **tight** (or belongs to the **equality graph**) when $u_i + v_j = c_{ij}$; only tight edges can appear in the current complementary-slackness certificate.

The algorithm augments the current matching along augmenting paths in the equality graph. When no such path exists, it adjusts the potentials: compute $\delta = \min_{i \in Z_X,\, j \notin Z_Y}(c_{ij} - u_i - v_j)$, then set $u_i \leftarrow u_i + \delta$ for $i \in Z_X$ and $v_j \leftarrow v_j - \delta$ for $j \in Z_Y$. This creates at least one new tight edge while preserving dual feasibility and the current matching. For a maximum-profit assignment one first converts profits to costs (for example, $c_{ij} = C - w_{ij}$ with C large enough). The three phases—augment on tight edges, adjust potentials, repeat—run in $O(n^3)$ time overall.

Example 12.7.2 (Worker–job assignment)**.** A company has 3 workers and 3 jobs with profitability matrix:

$$W = \begin{pmatrix} 5 & 7 & 2 \\ 3 & 9 & 6 \\ 8 & 4 & 1 \end{pmatrix}.$$

The optimal assignment is $x_1 \to y_2$, $x_2 \to y_3$, $x_3 \to y_1$, yielding total profit $7 + 6 + 8 = 21$.

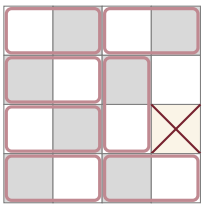
12.7.2 Domino tiling and maximum placement

Consider an $n \times n$ chessboard with some squares removed. We wish to place the maximum number of dominoes, where each domino covers exactly two adjacent squares. This is a maximum bipartite matching problem.

Reduction. Colour the board in the usual checkerboard pattern (black and white). Every domino covers one black and one white square. Define a bipartite graph:

- X = non-removed white squares, Y = non-removed black squares.
- Place an edge $\{x, y\}$ whenever squares x and y are horizontally or vertically adjacent.

A matching in this graph corresponds to a valid domino placement; the maximum matching gives the maximum number of dominoes. If that matching saturates every non-removed square, then we have a complete tiling; otherwise the board can only be packed partially.



4×4 grid with one cell removed (15 cells). Maximum matching: 7 dominoes (one cell remains uncovered).

Figure 12.8: Maximum domino placement on a 4×4 grid with one cell removed. The checkerboard colouring defines a bipartite graph; each domino (highlighted rectangle) corresponds to a matched edge. The maximum matching of 7 leaves exactly one cell uncovered, so this instance has no complete tiling.

12.7.3 Scheduling and resource assignment

Bipartite matching models many scheduling scenarios:

- **Workers to time slots:** each worker has availability constraints; edges connect workers to their available slots. Maximum matching assigns the most workers.
- **Students to projects:** each student ranks a subset of projects; a maximum weight matching optimises overall satisfaction.
- **Non-attacking rooks:** placing the maximum number of non-attacking rooks on a chessboard with forbidden squares is equivalent to maximum bipartite matching, where rows and columns form the two partitions and available squares form the edges.

12.8 Summary

Table 12.1 compares the main matching problems and algorithms discussed in this chapter and beyond.

General matching requires Edmonds' blossom algorithm, which handles odd cycles via "shrinking."

Table 12.1: Summary of matching problems and algorithms.

Problem	Graph type	Algorithm	Complexity
Max cardinality matching	Bipartite	Labelling / max-flow	$O(\|V\| \cdot \|E\|)$
Max weight matching	Bipartite	Hungarian (Kuhn 1955)	$O\left(\|V\|^3\right)$
Max cardinality matching	General	Blossom (Edmonds 1965)	$O\left(\|V\|^3\right)$

For bipartite graphs, the reduction to maximum flow (section 12.4) provides an efficient $O(|V| \cdot |E|)$ algorithm via Ford–Fulkerson with BFS (Edmonds–Karp). König's theorem (section 12.6) gives the powerful duality $\nu(G) = \tau(G)$ between maximum matching and minimum vertex cover.

For general (non-bipartite) graphs, augmenting paths may pass through odd cycles, creating so-called *blossoms*. Edmonds' blossom algorithm (1965) handles this by "shrinking" odd cycles into single nodes, running in $O\left(|V|^3\right)$ time. While we do not cover the blossom algorithm in this course, it is important to know that efficient polynomial-time algorithms exist for general matching as well.

Key take-aways:

1. A matching M is maximum $\iff$ no augmenting path exists (Berge's theorem).
2. Bipartite matching = max flow in a unit-capacity network.
3. In bipartite graphs: max matching = min vertex cover (König's theorem).
4. The labelling algorithm finds augmenting paths in $O(|E|)$ per iteration, $O(|V| \cdot |E|)$ overall.
5. The assignment problem (max weight perfect matching in bipartite graphs) is solved in $O\left(|V|^3\right)$ by the Hungarian algorithm.

Summary & Key Takeaways

- **Bipartite Matching:** Finding a subset of pairwise non-adjacent edges in a bipartite graph.
- **Hall's Marriage Theorem:** A bipartite graph $G = (L \cup R, E)$ has a matching covering L if and only if for all subsets $S \subseteq L$, the size of the neighborhood $|N(S)| \geq |S|$.

- **Max Flow Reduction:** Can solve bipartite matching by constructing a flow network with source s connected to L (capacity 1), and R connected to sink t (capacity 1).
- **Augmenting Path Algorithm:** A matching M is maximum if and only if there is no augmenting path (an alternating path between two unmatched vertices).

Exercises

Exercise 1. Define a *matching* in an undirected graph $G = (V, E)$. What does it mean for a vertex to be *saturated* (matched) by a matching M? What does it mean for a vertex to be *exposed* (free)?

Exercise 2. Let $G = (V, E)$ be a graph with $|V| = 8$ and $|E| = 10$. Is it possible for G to have a perfect matching? State the necessary condition on $|V|$ and explain why it is necessary (but not sufficient).

Exercise 3. Give the definition of a *maximum matching* and a *perfect matching*. Provide a small example (5–6 vertices) showing that a maximum matching need not be a perfect matching.

Exercise 4. Define an *alternating path* and an *augmenting path* with respect to a matching M. Explain the key structural difference between the two concepts.

Exercise 5. Let $G = (X \cup Y, E)$ with $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2, y_3\}$, and edges $x_1y_1, x_1y_2, x_2y_1, x_2y_3, x_3y_2, x_3y_3$. Start with the matching $M = \{x_1y_1, x_2y_3\}$.

(a) Identify all exposed vertices under M.

(b) Find an augmenting path with respect to M, listing the vertices in order and indicating which edges are in M and which are not.

(c) Apply the symmetric difference $M' = M \oplus E(P)$ to obtain a larger matching. Write down M' explicitly.

Exercise 6. Let $G = (X \cup Y, E)$ with $X = \{x_1, x_2, x_3, x_4\}$, $Y = \{y_1, y_2, y_3, y_4\}$, and edges $x_1y_1, x_1y_3, x_2y_1, x_2y_2, x_3y_2, x_3y_4, x_4y_3, x_4y_4$. Starting from the empty matching $M = \emptyset$, apply the augmenting-path method iteratively until a maximum matching is reached. At each step, exhibit the augmenting path used and the updated matching.

Exercise 7. Let $G = (X \cup Y, E)$ with $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2, y_3, y_4\}$, and edges $x_1y_1, x_1y_2, x_2y_2, x_2y_3, x_3y_3, x_3y_4$. The current matching is $M = \{x_1y_2, x_2y_3\}$.

(a) Run the BFS-based labelling algorithm to search for an augmenting path starting from every exposed vertex in X. Record which vertices receive a label and in which order.

(b) Write down the augmenting path found (if any) and the resulting matching M' after augmentation.

Exercise 8. Let $G = (X \cup Y, E)$ with $X = \{x_1, x_2, x_3, x_4\}$, $Y = \{y_1, y_2, y_3\}$, and edges $x_1y_1, x_2y_1, x_2y_2, x_3y_2, x_3y_3, x_4y_3$. The current matching is $M = \{x_1y_1, x_3y_2\}$.

(a) Find an augmenting path.

(b) Apply augmentation to obtain M'.

(c) Is M' a maximum matching? Justify without citing Berge's theorem.

Exercise 9. State Berge's theorem precisely. Identify the two implications that constitute the "if and only if" and explain informally why each direction holds.

Exercise 10. Let $G = (X \cup Y, E)$ with $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2, y_3\}$, and edges x_1y_1, x_2y_2, x_3y_3. The matching is $M = \{x_1y_1, x_2y_2, x_3y_3\}$.

(a) Attempt to find an augmenting path with respect to M.

(b) Using Berge's theorem, conclude whether M is a maximum matching.

(c) What is the size of a maximum matching in G?

Exercise 11. Prove one direction of Berge's theorem: *if M is a maximum matching then no augmenting path exists*. (Hint: assume for contradiction that an augmenting path P exists and construct a strictly larger matching.)

Exercise 12. Prove the other direction of Berge's theorem: *if no augmenting path exists then M is a maximum matching*. Your proof should explicitly use the structure of the symmetric difference $M \oplus M^*$ for any matching M^*.

Exercise 13. Let $G = (X \cup Y, E)$ with $X = \{x_1, x_2, x_3, x_4\}$, $Y = \{y_1, y_2, y_3, y_4\}$, and edges $x_1y_1, x_1y_4, x_2y_2, x_2y_4, x_3y_3, x_4y_3$. Let $M = \{x_1y_1, x_2y_2, x_3y_3\}$. Use Berge's theorem to determine whether M is maximum: run the labelling algorithm to exhaustion and report your conclusion.

Exercise 14. Define a *vertex cover* of a graph G. State König's theorem for bipartite graphs. Give a small bipartite example illustrating that König's theorem does *not* hold for general graphs.

Exercise 15. Let $G = (X \cup Y, E)$ with $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2, y_3\}$, and edges $x_1y_1, x_1y_2, x_2y_2, x_2y_3, x_3y_1, x_3y_3$.

(a) Find a maximum matching M^* using the augmenting-path method.

(b) Apply the König construction (alternating-path BFS from exposed X-vertices) to obtain a minimum vertex cover C.

(c) Verify that $|M^*| = |C|$.

Exercise 16. Let $G = (X \cup Y, E)$ with $X = \{x_1, x_2, x_3, x_4\}$, $Y = \{y_1, y_2, y_3, y_4\}$, and edges $x_1y_1, x_1y_2, x_2y_2, x_2y_3, x_3y_3, x_3y_4, x_4y_1, x_4y_4$.

(a) Find a maximum matching.

(b) Find a minimum vertex cover and verify König's equality.

Exercise 17. Using König's theorem, argue that the *minimum vertex cover problem* in bipartite graphs is solvable in polynomial time. What is the corresponding claim for general graphs, and why is it harder?

Exercise 18. Let $G = (X \cup Y, E)$ with $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2, y_3, y_4\}$, and edges $x_1y_1, x_1y_3, x_2y_2, x_2y_3, x_3y_2, x_3y_4$.

(a) Find a maximum matching M^* (state its size $\nu(G)$).

(b) Exhibit a vertex cover C with $|C| = \nu(G)$.

(c) Write down both the matching LP and the vertex-cover LP for G and identify the LP dual relationship between them.

Exercise 19. State Hall's theorem (Marriage Theorem) for a bipartite graph $G = (L \cup R, E)$. Define Hall's condition precisely using the neighbourhood $N(S)$ for $S \subseteq L$.

Exercise 20. Let $G = (L \cup R, E)$ with $L = \{l_1, l_2, l_3\}$, $R = \{r_1, r_2, r_3\}$, and edges $l_1r_1, l_1r_2, l_2r_2, l_2r_3, l_3r_1, l_3r_3$. Check Hall's condition for every subset $S \subseteq L$ and determine whether a perfect matching on L exists.

Exercise 21. Let $G = (L \cup R, E)$ with $L = \{l_1, l_2, l_3, l_4\}$, $R = \{r_1, r_2, r_3\}$, and edges $l_1r_1, l_2r_1, l_3r_2, l_4r_3$.

(a) Check Hall's condition for all subsets of L that could violate it. Identify a violating set S if one exists.

(b) Conclude whether a matching saturating all of L exists.

Exercise 22. Let $G = (L \cup R, E)$ with $L = \{l_1, l_2, l_3\}$, $R = \{r_1, r_2, r_3, r_4\}$, and edges $l_1r_1, l_1r_2, l_2r_2, l_2r_3, l_3r_3, l_3r_4$. Verify Hall's condition for all subsets $S \subseteq L$. Does a perfect matching (saturating all of L) exist? If yes, exhibit one.

Exercise 23. Let $G = (L \cup R, E)$ with $L = \{l_1, l_2, l_3\}$, $R = \{r_1, r_2, r_3\}$, and edges l_1r_1, l_2r_1, l_3r_1. Explicitly identify the subset $S \subseteq L$ that violates Hall's condition and explain why no perfect matching on L can exist.

Exercise 24. Let $G = (L \cup R, E)$ with $L = \{l_1, l_2, l_3, l_4\}$, $R = \{r_1, r_2, r_3, r_4\}$, and edges $l_1r_1, l_1r_2, l_2r_1, l_2r_3, l_3r_2, l_3r_4, l_4r_3, l_4r_4$.

(a) Check Hall's condition.

(b) Find a perfect matching on L (one that saturates every vertex of L).

Exercise 25. Suppose Hall's condition holds for a bipartite graph $G = (L \cup R, E)$ with $|L| = |R|$. Does it follow that the perfect matching is *unique*? Provide either a proof or a counterexample.

Exercise 26. Describe how to reduce bipartite matching to maximum flow. Given $G = (X \cup Y, E)$, define the directed flow network N (with source s and sink t), specifying all arc capacities. Explain the correspondence between integer s-t flows and matchings.

Exercise 27. Let $G = (X \cup Y, E)$ with $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2, y_3\}$, and edges $x_1y_1, x_1y_2, x_2y_2, x_2y_3, x_3y_1, x_3y_3$.

(a) Write down all arcs and capacities of the flow network N.

(b) Find a maximum s-t flow in N (by inspection or Ford–Fulkerson).

(c) Translate the flow into a matching M^* and state $|M^*|$.

Exercise 28. Let $G = (X \cup Y, E)$ with $X = \{x_1, x_2, x_3, x_4\}$, $Y = \{y_1, y_2, y_3, y_4\}$, and edges $x_1y_1, x_2y_1, x_2y_2, x_3y_3, x_3y_4, x_4y_2, x_4y_4$. Construct the corresponding flow network and find a maximum flow. What is the size of the maximum matching?

Exercise 29. In the reduction of bipartite matching to max-flow, a minimum s-t cut corresponds to a minimum vertex cover (König's theorem viewed through the max-flow min-cut lens). For the graph $G = (X \cup Y, E)$ with $X = \{x_1, x_2\}$, $Y = \{y_1, y_2\}$, and edges x_1y_1, x_1y_2, x_2y_1,

(a) Build the flow network and find a minimum cut.

(b) Translate the cut into a vertex cover and verify König's equality.

Exercise 30. Write the LP relaxation of the bipartite matching problem for a graph $G = (V, E)$. Use variables $x_e \in [0, 1]$ for each edge $e \in E$. State all constraints explicitly.

Exercise 31. Explain why the LP relaxation of bipartite matching always has an integer optimal solution. Your explanation should reference the concept of *total unimodularity* (TUM) and the constraint matrix of the LP.

Exercise 32. Consider the bipartite matching LP for $G = (X \cup Y, E)$ with $X = \{x_1, x_2\}$, $Y = \{y_1, y_2\}$, and edges x_1y_1, x_1y_2, x_2y_1.

(a) Write the LP explicitly with all constraints.

(b) Write the dual LP.

(c) Identify a primal optimal solution and a dual optimal solution and verify complementary slackness.

Exercise 33. True or false — justify each statement:

(a) "Every bipartite graph with $|X| = |Y|$ has a perfect matching."

(b) "If a bipartite graph has a perfect matching, the LP relaxation has a unique optimal solution."

(c) "The constraint matrix of the bipartite matching LP is totally unimodular."

(d) "For non-bipartite graphs, the matching LP relaxation can have fractional optimal solutions."

Exercise 34. True or false — justify each statement:

(a) "Every bipartite graph has a perfect matching."

(b) "If Hall's condition holds for every subset of L, the matching that saturates L is unique."

(c) "König's theorem holds for all graphs."

(d) "A maximum matching in a bipartite graph can be found in polynomial time."

Exercise 35. Define the *assignment problem* as a weighted bipartite matching problem. Write the ILP formulation with binary variables $x_{ij} \in \{0, 1\}$ for assigning worker i to job j, given cost matrix (c_{ij}).

Exercise 36. Write the LP relaxation of the assignment problem defined in the previous exercise. Explain why the LP relaxation always yields an integer solution (citing TUM or the doubly stochastic matrix argument).

Exercise 37. Three workers $\{w_1, w_2, w_3\}$ must be assigned to three jobs $\{j_1, j_2, j_3\}$ with cost matrix

$$C = \begin{pmatrix} 3 & 1 & 2 \\ 2 & 4 & 3 \\ 1 & 2 & 5 \end{pmatrix},$$

where C_{ij} is the cost of assigning worker w_i to job j_j. Write the assignment LP for this instance. Identify by inspection or reasoning (not the full Hungarian algorithm) an optimal assignment and compute its total cost.

Exercise 38. Explain the Hungarian algorithm at a high level: what are the main phases (row and column reduction, zero-covering, augmentation) and what is its worst-case time complexity?

Exercise 39. Model the problem of tiling an $m \times n$ grid by 1×2 dominoes as a bipartite matching problem. Define the bipartite graph $G = (B \cup W, E)$ where B and W are the black and white squares in the standard chessboard colouring, and describe the edges.

Exercise 40. Show that a standard 8×8 chessboard can be tiled perfectly by 32 dominoes by constructing the bipartite graph and arguing that Hall's condition is satisfied.

Exercise 41. Consider an 8×8 chessboard with the two diagonally opposite corner squares removed (one black corner and one white corner).

(a) How many black squares and how many white squares remain?

(b) Does the bipartite graph of the modified board satisfy the size condition $|B| = |W|$ necessary for a perfect matching?

(c) Using a Hall-condition or colouring argument, determine whether the board can be tiled by dominoes.

Exercise 42. Consider an 8×8 chessboard with two squares of the *same colour* removed.

(a) After removal, how many black and white squares remain?

(b) Argue (without finding a specific configuration) whether a domino tiling is possible in general.

Exercise 43. A 4×4 grid has the four corner squares removed, leaving 12 squares. Model the tiling problem as bipartite matching. Write down the bipartite graph explicitly and determine whether a perfect tiling by dominoes exists.

Exercise 44. Five jobs $\{j_1, \ldots, j_5\}$ must be assigned to five machines $\{m_1, \ldots, m_5\}$, with each job assigned to exactly one machine and each machine handling at most one job. Feasible assignments are: $j_1 \to \{m_1, m_2\}$, $j_2 \to \{m_2, m_3\}$, $j_3 \to \{m_3, m_4\}$, $j_4 \to \{m_1, m_4\}$, $j_5 \to \{m_3, m_5\}$.

(a) Model this as a bipartite matching problem: define $G = (J \cup M, E)$ explicitly.

(b) Find a maximum matching and state how many jobs can be scheduled.

(c) If a job cannot be scheduled (exposed), which job is it and why?

Exercise 45. Four students $\{s_1, s_2, s_3, s_4\}$ wish to be assigned to four lab-session slots $\{t_1, t_2, t_3, t_4\}$. Availability: $s_1 \to \{t_1, t_3\}$, $s_2 \to \{t_1, t_2\}$, $s_3 \to \{t_2, t_4\}$, $s_4 \to \{t_3, t_4\}$.

(a) Model as bipartite matching.

(b) Apply the augmenting-path method to find a perfect matching (if one exists).

(c) Verify your answer using Hall's theorem.

Exercise 46. A project manager has n tasks and n employees. Each employee can perform a subset of tasks, and each task requires exactly one employee. Formulate the feasibility question ("can all tasks be assigned?") as a bipartite matching problem and state the condition (Hall's theorem) under which the answer is "yes".

Exercise 47. Explain why the augmenting-path method for bipartite graphs fails for general (non-bipartite) graphs. What is a *blossom* (odd cycle) and why does it cause problems when searching for augmenting paths?

Exercise 48. Describe at a high level what Edmonds' blossom algorithm does differently from the bipartite labelling algorithm. What is the "shrinking" step and what property does it preserve?

Exercise 49. Give an example of a non-bipartite graph on 5 vertices and 5 edges that has a maximum matching of size 2 but for which a naïve augmenting-path search (without blossom handling) would fail to find an augmenting path from a particular starting configuration. Explain the failure.

Exercise 50. Compare the computational complexities of the following matching algorithms: (a) the bipartite labelling algorithm (BFS-based augmenting paths), (b) the Hopcroft–Karp algorithm for bipartite graphs, and (c) Edmonds' blossom algorithm for general graphs. For each, state the time complexity and briefly explain the main idea.

Exercise 51. Let $G = (X \cup Y, E)$ with $X = \{x_1, x_2, x_3, x_4\}$, $Y = \{y_1, y_2, y_3, y_4\}$, and edges $x_1y_2, x_1y_3, x_2y_1, x_2y_3, x_3y_1, x_3y_4, x_4y_2, x_4y_4$.

(a) Find a maximum matching M^* using augmenting paths.

(b) Apply the König construction to find a minimum vertex cover.

(c) Check Hall's condition and confirm whether a perfect matching exists.

(d) Write the matching LP and identify an integer optimal solution.

Exercise 52. A bipartite graph $G = (L \cup R, E)$ with $|L| = |R| = n$ is *k-regular* (every vertex has degree exactly k). Prove that G has a perfect matching. (Hint: use Hall's theorem together with a double-counting argument on edges.)

Exercise 53. Let $G = (X \cup Y, E)$ with $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2, y_3, y_4\}$, and edges $x_1y_1, x_1y_4, x_2y_2, x_2y_4, x_3y_3, x_3y_4$.

(a) Find a maximum matching.

(b) Apply the max-flow reduction: build the flow network, find a maximum flow, and translate it back to a matching.

(c) Identify the minimum cut in the flow network and the corresponding minimum vertex cover.

Exercise 54. Explain the relationship between matching and *independent sets* in a graph. In particular, for a bipartite graph G, relate the size of a maximum independent set $\alpha(G)$ to the size of a minimum vertex cover $\tau(G)$ and to the maximum matching $\nu(G)$. State the relevant theorems (Gallai's identities and König's theorem).

Exercise 55. Consider the following claim: "In a bipartite graph, if $|M| < |L|$ for every matching M, then there exists a set $S \subseteq L$ with $|N(S)| < |S|$." Prove this claim. (This is the converse direction of Hall's theorem.)

Appendix: Quick Reference Card

How to use this card

For every exercise, answer three questions before computing:

1. **Structure:** continuous or integer variables? weighted graph? capacities? bipartition?
2. **Hypotheses:** signs of weights, integer data, tableau convention, graph type.
3. **Certificate:** reduced costs, dual solution, cut, or absence of augmenting paths.

Linear Programming

Canonical and standard form

$$\begin{aligned} &\text{Canonical max:} && \max\{c^\top x : Ax \le b,\ x \ge 0\},\\ &\text{Standard max:} && \max\{c^\top x : Ax = b,\ x \ge 0\}. \end{aligned}$$

- $a^\top x \le b$: add slack $s \ge 0$.
- $a^\top x \ge b$: subtract surplus $s \ge 0$.
- x_j unrestricted: $x_j = x_j^+ - x_j^-, x_j^\pm \ge 0$.
- $\min f(x) = \max[-f(x)]$.

Basis, BFS, and reduced costs

For a basis B with nonsingular A_B:

$$\bar{b} = A_B^{-1} b, \qquad x_B = \bar{b} - A_B^{-1} A_N x_N.$$

A basic solution is feasible (a **BFS**) iff $\bar{b} \ge 0$. For each nonbasic variable x_j:

$$\hat{c}_j = c_j - c_B^\top A_B^{-1} A_j, \qquad z = \bar{z} + \sum_{j \in N} \hat{c}_j x_j.$$

For a maximisation problem in the convention used in these notes:

Test	Conclusion
$\hat{c}_j \le 0$ for all $j \in N$	current BFS is optimal
some $\hat{c}_j > 0$	x_j may enter the basis
$A_B^{-1} A_j \le 0$ componentwise	LP is unbounded
some $\bar{a}_{ij} > 0$	apply the ratio test

$$\theta^* = \min_{i : \bar{a}_{ij} > 0} \frac{\bar{b}_i}{\bar{a}_{ij}}.$$

The row attaining the minimum identifies the leaving basic variable. If $\theta^* = 0$, the pivot is **degenerate**.

Check: Some tableaux store $-\hat{c}_j$ in the objective row. In that convention all sign tests reverse. Read the equation for z before applying a memorised pivot rule.

Simplex decision procedure

1. Optimality: stop if every nonbasic $\hat{c}_j \le 0$. **2. Entering:** choose any $\hat{c}_j > 0$ (Dantzig: largest; Bland: smallest index). **3. Unboundedness:** if entering column $A_B^{-1} A_j \le 0$, stop: LP is unbounded. **4. Leaving:** run min ratio test over positive entries only. **5. Pivot:** normalise pivot row, eliminate entering column elsewhere.

Bland's rule

Among eligible variables, use the smallest index. On a ratio-test tie, the basic variable with smallest index leaves. This prevents cycling.

Dual Simplex

Optimality: stop if every $\bar{b}_i \ge 0$. **Leaving:** choose any $\bar{b}_r < 0$. **Infeasibility:** if row r has no $\bar{a}_{rj} < 0$, primal is infeasible. **Entering:** min ratio $|\hat{c}_j / \bar{a}_{rj}|$ over $\bar{a}_{rj} < 0$.

Two-phase & Big-M

- **Phase I:** introduce artificials $a \ge 0$, minimise $\sum a$. If opt > 0, LP is infeasible. If $= 0$, restore original obj.
- **Big-M:** penalise artificials in the original objective (e.g. $\max z - M \sum a$).

Outcomes

Infeasible: empty feasible region. **Unbounded:** objective improves infinitely. **Multiple optima:** an optimal tableau has an eligible nonbasic variable with zero reduced cost.

LP Duality

Canonical pair

$$\begin{array}{ccc} \max\ c^\top x & \longleftrightarrow & \min\ b^\top y \\ Ax \le b & \longleftrightarrow & A^\top y \ge c \\ x \ge 0 & \longleftrightarrow & y \ge 0 \end{array}$$

General max-to-min correspondence

Primal max	Dual min
constraint $\le$	$y_i \ge 0$
constraint $=$	y_i free
constraint $\ge$	$y_i \le 0$
$x_j \ge 0$	dual constraint $\ge$
x_j free	dual constraint $=$
$x_j \le 0$	dual constraint $\le$

Optimality certificates

Weak duality gives $c^\top x \le b^\top y$ for every feasible pair. If equality holds, both solutions are optimal. Equivalently:

$$y_i (b_i - a_i^\top x) = 0 \quad \forall i, \qquad x_j (a_j^\top y - c_j) = 0 \quad \forall j.$$

Thus $y_i > 0$ forces primal constraint i to be tight, and $x_j > 0$ forces dual constraint j to be tight.

Binary and ILP modelling

Let $y, y_A, y_B \in \{0,1\}$. Use the following patterns only after writing their intended logic in words.

Logic	Linear formulation
$A \Rightarrow B$	$y_A \le y_B$
at least one	$\sum_j y_j \ge 1$
at most one	$\sum_j y_j \le 1$
exactly one	$\sum_j y_j = 1$
x active only if $y = 1$	$0 \le x \le Uy$
$y = 1 \Rightarrow a^\top x \le b$	$a^\top x \le b + M(1-y)$
$y = 0 \Rightarrow a^\top x \le b$	$a^\top x \le b + My$

Classic Models

Knapsack: $\max\{\sum v_j x_j : \sum w_j x_j \le W\}$. **Set Cover:** $\min\{\sum c_j x_j : \sum_{j:e_i \in S_j} x_j \ge 1\}$. **Set Packing/Partitioning:** use ≤ 1 or $= 1$. **TSP (SECs):** $\sum_{i,j\in S} x_{ij} \le |S| - 1 \; \forall S \subsetneq V, |S| \ge 2$.

Either-or constraint

To impose at least one of $a^\top x \le b$ and $d^\top x \le e$:

$$a^\top x \le b + My, \qquad d^\top x \le e + M(1-y).$$

Fixed-charge activation

If $y = 1$ means that activity x is open:

$$Ly \le x \le Uy.$$

Use $L = 0$ when opening permits, but does not require, positive production.

Check: M is a valid upper bound on the largest possible violation, not an arbitrary "very large number." Too small removes feasible points; too large weakens the LP relaxation and harms numerical stability.

Relaxation and integrality gap

Dropping $x \in \mathbb{Z}^n$ gives the LP relaxation. For maximisation:

$$z^*_{\text{LP}} \ge z^*_{\text{ILP}}.$$

A tighter formulation gives a better bound and usually a smaller search tree.

ILP Solving Methods

Method	Branching	Cuts	LP Engine
B&B	On $x_j \notin \mathbb{Z}$	None	Primal Simplex
Cuts	None	Gomory / CG	Dual Simplex
B&C	On $x_j \notin \mathbb{Z}$	At nodes	Dual Simplex

Branch and Bound (B&B)

Divide-and-conquer on fractional variables. LP relaxations yield **upper bounds** ($z_{\text{LP}} \ge z_{\text{ILP}}$ for max). Keep the best integer solution as **incumbent** ($\underline{z}$). Evaluate a node by solving its LP:

- **Prune (infeasibility):** Node LP is infeasible.
- **Prune (optimality):** LP solution is integer. If $z_{\text{LP}} > \underline{z}$, update incumbent ($\underline{z} \leftarrow z_{\text{LP}}$).
- **Prune (bound):** $z_{\text{LP}} \le \underline{z}$ (cannot improve incumbent).
- **Branch:** If x^*_j is fractional, split into:

$$x_j \le \lfloor x^*_j \rfloor \quad \text{or} \quad x_j \ge \lceil x^*_j \rceil.$$

Cutting Plane Method

Iteratively refines the LP relaxation without branching.

1. Solve the current LP relaxation.
2. If the solution is fractional, find a **valid cut** (excludes the fractional point, preserves all integer feasible points).
3. Add the cut. Re-optimize using the **Dual Simplex** (since the old basis becomes primal-infeasible but remains dual-feasible).
4. Repeat until the LP optimum is integer.

Chvátal–Gomory cut: For $\lambda \ge 0$ where $\lambda^\top A \in \mathbb{Z}^n$:

$$(\lambda^\top A)x \le \lfloor \lambda^\top b \rfloor.$$

Gomory Tableau Cut: From a fractional row $x_i + \sum_{j\in N} \bar{a}_{ij} x_j = \bar{b}_i$ (with $f_i = \bar{b}_i - \lfloor \bar{b}_i \rfloor > 0$ and $f_{ij} = \bar{a}_{ij} - \lfloor \bar{a}_{ij} \rfloor$):

$$\sum_{j\in N} f_{ij} x_j \ge f_i.$$

Branch and Cut (B&C)

Integrates B&B and Cutting Planes. At each node of the B&B tree, we solve the LP relaxation and run a separation algorithm to add violated cuts. We only branch when no more useful cuts can be found.

Total Unimodularity

Definition and consequence

A is **totally unimodular (TUM)** if every square submatrix has determinant in $\{-1, 0, 1\}$. If A is TUM and $b \in \mathbb{Z}^m$, every vertex of

$$P = \{x \ge 0 : Ax \le b\}$$

is integral. The LP relaxation then solves the corresponding ILP.

Ghouila–Houri criterion

For every subset R of rows, there must exist a partition $R = R_1 \dot\cup R_2$ such that, for every column j,

$$\sum_{i\in R_1} a_{ij} - \sum_{i\in R_2} a_{ij} \in \{-1, 0, 1\}.$$

The quantifier "for every subset of rows" is essential.

Course shortcut: at most two nonzeros per column

For $A \in \{0, \pm 1\}^{m\times n}$, it is sufficient that:

- every column has at most two nonzero entries;
- equal signs occur in rows belonging to different parts;
- opposite signs occur in rows belonging to the same part.

Classes to recognise

- Node–arc incidence matrix of a directed graph: TUM.
- Vertex–edge incidence matrix of a bipartite graph: TUM.
- A triangle's undirected incidence matrix is not TUM: it has a subdeterminant of magnitude 2.
- Transposition, row/column permutations, and sign changes preserve TUM; appending identity rows or columns also preserves TUM.

Check: TUM alone is not enough: the right-hand side must be integral. Also check that the full constraint matrix, including added bounds, preserves the TUM structure.

Graph representations and visits

Write $n = |V|$ and $m = |E|$.

Tool	Cost	Use
Adjacency list	space $O(n+m)$	sparse graphs, scan neighbours
Adjacency matrix	space $O(n^2)$	edge test in $O(1)$
BFS	$O(n+m)$	unit distances, components
DFS	$O(n+m)$	cycles, SCC, topological order
Kahn / topo DFS	$O(n+m)$	topological order in a DAG
Warshall	$O(n^3)$	transitive closure

BFS

Use a FIFO queue. The first time v is discovered, $d[v] = d[u] + 1$ and $\pi[v] = u$. BFS finds shortest paths only when every edge has the same cost (usually one).

DFS

Use discovery/finishing times. In a directed graph, a back edge certifies a cycle. A directed graph is acyclic iff it admits a topological ordering. Reverse finishing order gives a topological order only for a DAG.

Minimum Spanning Trees

Assume an undirected, connected, weighted graph. Every spanning tree has $n - 1$ edges.

- **Cut property:** a minimum-weight edge crossing a cut is safe for some MST.
- **Cycle property:** an edge strictly heavier than every other edge on a cycle belongs to no MST.

Kruskal

Sort edges by nondecreasing weight. Add (u, v) iff $\textsc{Find}(u) \neq \textsc{Find}(v)$; then union the components. Stop after $n - 1$ accepted edges.

$$T_{\text{Kruskal}} = O(m \log m) = O(m \log n).$$

Prim

Grow one tree from a root. Repeatedly extract the outside vertex with minimum connection key and update its neighbours.

$$T_{\text{Prim}} = O(m + n \log n)$$

with a Fibonacci heap; $O(m \log n)$ with a binary heap.

Check: Negative edge weights are allowed for MST. Distinct edge weights imply a unique MST, but an MST may still be unique when some weights tie.

Shortest paths

Relaxation

For an arc (u, v) with weight $w(u, v)$:

$$\text{if } d[v] > d[u] + w(u,v), \quad d[v] \leftarrow d[u] + w(u,v), \quad \pi[v] \leftarrow u.$$

Initialise $d[s] = 0$ and $d[v] = +\infty$ for $v \neq s$.

Choose the algorithm from the hypotheses

Case	Algorithm	Time
unit weights	BFS	$O(n+m)$
DAG, any signs	topo + relax	$O(n+m)$
nonnegative weights	Dijkstra	$O(m \log n)$
negative edges	Bellman–Ford	$O(nm)$
all pairs, dense	Floyd–Warshall	$O(n^3)$

Dijkstra

Extract the unsettled vertex with minimum tentative distance, then relax its outgoing arcs. The extracted distance becomes final only because all edge weights are nonnegative.

Bellman–Ford

Relax every edge for $n - 1$ passes. If any edge is still relaxable on pass n, a negative cycle is reachable from the source.

DAG shortest paths

Process vertices in topological order and relax outgoing arcs. Negative weights are allowed because a DAG has no directed cycle.

Floyd–Warshall

$$D_{ij}^{(k)} = \min\{D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\}.$$

Initialise $D_{ii} = 0$, direct-edge weights where present, and $+\infty$ otherwise. At termination, $D_{ii} < 0$ reveals a negative cycle through i.

Critical Path Method (DAG)

For activities with duration p_v and precedence arcs:

$$ES(v) = \max_{u \in \text{pred}(v)} \{ES(u) + p_u\}.$$

The project duration is the longest source–sink path. Zero-slack activities form a critical path.

Check: Never run Dijkstra merely because most edges are nonnegative: one negative edge is enough to invalidate its correctness proof.

Network flow

Feasibility

For every arc (u, v):

$$0 \le f(u,v) \le c(u,v).$$

For every internal vertex $v \notin \{s, t\}$:

$$\sum_u f(u,v) = \sum_w f(v,w).$$

The flow value is the net flow out of s (equivalently, into t).

Residual network

$$c_f(u,v) = c(u,v) - f(u,v), \qquad c_f(v,u) = f(u,v).$$

For an augmenting path P:

$$\Delta = \min_{e \in P} c_f(e).$$

Add Δ on forward arcs and subtract it on backward arcs. Backward arcs are how the algorithm revises previous choices.

Cuts and the optimality certificate

For a cut (S, T) with $s \in S$ and $t \in T$:

$$c(S,T) = \sum_{\substack{u \in S \\ v \in T}} c(u,v), \qquad |f| \le c(S,T).$$

If no residual s–t path exists, let S be the vertices reachable from s in the residual graph. Then $|f| = c(S,T)$, proving:

$$\max |f| = \min c(S,T).$$

Algorithms

Method	Path choice	Time
Ford–Fulkerson	arbitrary	$O(m\lvert f^*\rvert)$ for integer capacities
Edmonds–Karp	BFS in residual	$O(nm^2)$

With integer capacities, there exists an integer maximum flow.

Check: Flow feasibility requires both capacity constraints and conservation. A set of arc values satisfying capacities alone is not necessarily a flow.

Matching

A matching M is a set of edges with no shared endpoint.

- An **alternating path** alternates edges outside/inside M.
- An **augmenting path** is alternating and has free endpoints.
- Flipping membership along an augmenting path increases $|M|$ by one.
- **Berge:** M is maximum iff no augmenting path exists.

Bipartite matching as max flow

For $G = (L \cup R, E)$, create

$$s \to L, \qquad L \to R, \qquad R \to t$$

with unit capacities. An integer flow of value k corresponds to a matching of cardinality k.

Hall and König

$$G \text{ has a matching covering } L \iff |N(S)| \ge |S| \quad \forall S \subseteq L.$$

For every bipartite graph:

$$\text{maximum matching size} = \text{minimum vertex-cover size}.$$

Matching certificates

- Maximum matching: no augmenting path.
- Bipartite maximum matching: a vertex cover of the same cardinality.
- Perfect matching covering L: Hall's condition.

Algorithm selector

Goal / hypothesis	Use	Time
unweighted SSSP	BFS	$O(n+m)$
DAG SSSP	topo relax	$O(n+m)$
nonnegative SSSP	Dijkstra	$O(m \log n)$
negative-edge SSSP	Bellman–Ford	$O(nm)$
dense APSP	Floyd–Warshall	$O(n^3)$
MST, edge-oriented	Kruskal	$O(m \log n)$
MST, vertex-oriented	Prim	$O(m + n \log n)$
max flow, integer data	Ford–Fulkerson	$O(m\lvert f^*\rvert)$
max flow, polynomial	Edmonds–Karp	$O(nm^2)$
bipartite matching	augment / max flow	$O(nm)$
assignment	Hungarian	$O(n^3)$

Final exam checklist

1. State the optimisation direction and variable domains.
2. Write the algorithm's hypotheses beside its name.
3. Track the invariant: feasibility, settled distances, acyclicity, conservation, or matching validity.
4. Finish with a certificate, not only a candidate solution.