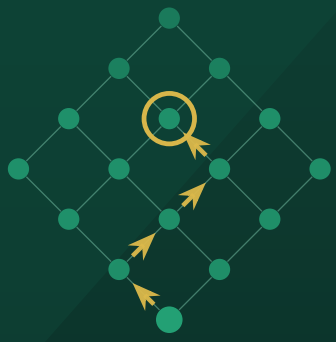
A Systematic Approach to

Sequential Algorithms



Vijay K. Garg

Department of Electrical and Computer Engineering
The University of Texas at Austin

To my family

©2019, 2026 Vijay K. Garg

Contents

List of Figures

List of Algorithms

Preface

Approach

This book presents algorithms from two complementary perspectives. The *traditional* perspective develops each algorithm in its classical form: greedy choices, divide-and-conquer recursions, dynamic-programming table fills, network-flow augmenting paths, and so on. The *lattice-linear predicate* (LLP) perspective recasts each problem as a search for an extremal element—typically the least element—of an appropriate distributive lattice that satisfies a Boolean predicate B. An algorithm in the second form maintains a single state vector G, identifies any component that is *forbidden* under B, and *advances* it according to a rule derived from the problem's structure.

The lattice-search perspective brings several advantages:

- **Simpler statements.** Almost every algorithm in this book uses a single state vector G. Any auxiliary state, when present, is syntactic sugar over the underlying lattice machinery.
- **Nondeterministic schedules.** The LLP loop fixes only that *some* forbidden index advances next; the choice of *which* is left open. A proof that holds for the nondeterministic schema applies uniformly to every concrete schedule, including the textbook sequential ones.
- **Vector-based reasoning.** Reasoning about G as a vector in a lattice replaces case analysis with monotonicity arguments.
- **Natural parallelism.** Because the schedule is open, every forbidden index is a candidate for advance in the same round. Many of the algorithms in this book are parallel by inheritance, requiring no separate derivation.
- **Easier correctness proofs.** The standard recipe—prove lattice-linearity of B, exhibit the advance rule, and apply the LLP fixed-point theorem—replaces the algorithm-specific invariant work found in classical presentations.

Guiding principles

The book is organized around three principles.

Derive, do not present. The goal of each chapter is to *derive* the algorithm from the feasibility and optimality structure of the underlying problem, rather than to present a finished algorithm and then verify that it works. The lattice formulation makes this derivation systematic: identify the lattice, write down B, check lattice-linearity, and read off the advance rule.

Prefer nondeterminism. Whenever possible, algorithms are stated nondeterministically. A nondeterministic statement is typically simpler than any of its deterministic refinements, and—crucially—a single proof applies to the entire class of refinements. The reader who wants a sequential schedule, a parallel schedule, or a distributed schedule can specialize the same algorithm to her needs.

Parallel by default. Many algorithms in this book admit a simple parallel form: start from a feasible point and improve it monotonically toward the optimum. The LLP framework exposes this structure explicitly—every forbidden index can advance in the same round—so parallelism need not be retrofitted onto a sequential description.

Companion material

Every algorithm in the book ships as runnable code on the companion website. Each algorithm is written once in a small lattice-linear DSL and compiled to Java, Python, C++, and JavaScript. The website also hosts interactive demos that step through the LLP loop on small concrete instances. Both the source code and the demos are intended to help the reader develop intuition for what an LLP algorithm *does*, beyond what the printed page alone can convey.

A solution manual covering the end-of-chapter problems is available to instructors on request.

Acknowledgments

I thank my co-authors on papers whose results appear in this book: David Alves, Bharath Balasubramanian, John Bridgman, Craig Chase, Himanshu Chauhan, Rohan Garg, Milos Gligoric, Changyong Hu, Neeraj Mittal, Vinit Ogale, Abdullah Rasheed, Alper Sen, Robert Streit, Brian Waldecker, and Xiong Zheng.

I thank the Department of Electrical and Computer Engineering at The University of Texas at Austin, where I have had the opportunity to develop and teach a course on algorithms. The students in that course have, over many years, sharpened the presentation of nearly every chapter.

This work has been supported in part by many grants from the National Science Foundation; this book would not have been possible without that support.

Finally, I thank my family. Without their love and support, this book would not have been conceived.

The list of known errors and supplementary material for the book is maintained at:

`http://www.ece.utexas.edu/~garg`

Vijay K. Garg

Austin, Texas

Chapter 1

Introduction to Algorithms

1.1 Introduction

Computers are fast, but algorithms are what make them useful. The same problem can admit algorithms whose running times differ by factors of millions on inputs that already fit comfortably in memory: sorting a billion records takes minutes with a good algorithm but centuries with a bad one. Understanding what makes one algorithm faster than another, and being able to predict how an algorithm scales before running it, is the central skill of algorithm design.

A single perspective runs through the chapters that follow, and it is worth stating up front. Most of the algorithms in this book can be cast as a *search*: the candidate solutions form a structured space, the desired answer is the smallest (or largest) element of that space satisfying a property, and the algorithm walks through the space toward that element. The novelty is in the structure we use to organize the search and the discipline it imposes on what counts as a step toward the answer.

The algorithms in this book all work with a vector of values — a vector of distances from a source vertex, a vector of prices on items, an assignment of one set of agents to another. Two such vectors are compared *componentwise*: one vector is smaller than another if it is smaller (or equal) at every coordinate. Two vectors can also be combined componentwise, either by taking the componentwise minimum or the componentwise maximum. These two combining operations — minimum and maximum coordinate by coordinate — are at the heart of how the framework reasons about progress toward the answer. We will refer to this vector as the algorithm's *state vector*.

Denote the state vector by G. What we ask of G is captured by a Boolean predicate B, and we want the smallest G such that $B(G)$ holds. The framework's central observation is that for many problems B has a closure property called *lattice-linearity*: whenever $B(G)$ is false, some component j of G is provably *forbidden* — every vector that agrees with G at coordinate j and is otherwise componentwise larger still fails B. The algorithm then writes itself: maintain a current state G, find any forbidden coordinate j, advance G at j by the minimum amount that removes the violation, and repeat. The loop terminates with the unique smallest G satisfying B.

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

This is the *lattice-linear predicate* (LLP) framework, developed formally in Chapter 2. Its appeal is that a single short proof — that B is lattice-linear — replaces the case-by-case correctness arguments traditionally given for individual algorithms. As we will show in later chapters, many classical algorithms turn out to be instances of the LLP schema under appropriate choices of the lattice and the predicate: the Gale–Shapley algorithm for stable marriage, Dijkstra's algorithm for shortest paths, Kruskal's and Prim's algorithms for minimum spanning trees, Bellman–Ford for shortest paths with negative edges, and Kuhn's Hungarian method for the assignment problem all fit the same template. The lattice differs from problem to problem — proposal vectors in stable marriage, distance vectors in shortest paths, edge-inclusion booleans in MST — but the *forbidden* / *advance* loop is the same.

Even Euclid's algorithm for the greatest common divisor, our first concrete example in Section 1.2 below, fits the template. The state is the pair $G = (G[1], G[2])$, initialized to (a, b), on a *descending* lattice (the algorithm decreases coordinates toward the answer). The predicate is that $G[1]$ and $G[2]$ are equal, and each divides its original input — formally, $B \equiv (G[1] \mid a) \wedge (G[2] \mid b) \wedge (G[1] = G[2])$. If $G[1] > G[2]$, then $G[1]$ is forbidden: it is too large to equal $G[2]$, and must be reduced. Each step replaces the larger value by its remainder modulo the smaller (or by the smaller itself when the remainder is zero), and the loop terminates with $G[1] = G[2] = \gcd(a, b)$. This is a tiny instance, but it is a faithful one — and once you can see Euclid's GCD as "identify the lattice, identify the predicate, advance the forbidden coordinate," you can see the same shape in Gale–Shapley, in Dijkstra, in Kruskal, and in every algorithm that comes after.

Adopting this perspective brings three practical benefits. First, a single correctness argument carries across a whole family of algorithms — once you have shown that an instance of the framework is lattice-linear, the rest of the analysis is bookkeeping. Second, the same algorithm reused on different lattices solves problems that look superficially unrelated: the same loop computes shortest paths, market-clearing prices, and stable matchings. Third, the LLP recasting often exposes a natural parallel version of a sequential algorithm: any number of independently forbidden coordinates can advance in the same round, so a sequential LLP algorithm typically has a free parallel sibling without any new algorithmic insight required. We make use of all three observations throughout the book. The reader is encouraged to ask, on encountering each new algorithm: *what is the lattice, what is the predicate, and what is the advance rule?*

This chapter is organized as follows. Section 1.2 defines what an algorithm is, using Euclid's GCD as a running example. Section 1.3 introduces asymptotic notation (Big-O, Big-Omega, Big-Theta) for describing growth rates. Section 1.4 analyzes the running time of several familiar algorithms (linear search, binary search, the Tower of Hanoi) to make the asymptotic vocabulary concrete. Section 1.5 reviews the basic data structures (linked lists, stacks, queues, binary search trees) that later chapters take as primitives. Section 1.6 describes the heap data structure and the linear-time Build-Heap algorithm.

1.2 What is an Algorithm?

An algorithm is a finite, unambiguous sequence of instructions that solves a problem or computes a function, transforming an input into an output in a finite number of steps. It must exhibit *finiteness* (terminates), *definiteness* (each step is precisely specified), and *effectiveness*

(each operation is basic and executable). Algorithms are the foundation of computer science, solving problems from simple arithmetic to complex optimization.

We start with one of the earliest algorithms due to Euclid. Suppose that we are required to find the greatest common divisor (GCD) of two natural numbers a and b (integers greater than or equal to 1). If they are both equal, then we have the answer: a. Suppose that a is greater than b, then we claim that it is safe to reduce a to a mod b — the remainder when a is divided by b. If b divides a, the remainder is 0; in that case we set a to b, and the two values are now equal. Similarly, if b is greater than a, then we reduce b to b mod a, or to a if a divides b. The algorithm terminates with both numbers identical and equal to the GCD.

Algorithm GCD: Euclidean algorithm for greatest common divisor

```
   input: a, b: positive integers
   output: greatest common divisor of a and b
 1 while a ≠ b do
 2 |  if a > b then
 3 |  |  if a mod b = 0 then a := b;
 4 |  |  else a := a mod b;
 5 |  else
 6 |  |  if b mod a = 0 then b := a;
 7 |  |  else b := b mod a;
 8 |  end
 9 end
10 return a
```

Example 1.1 For $a = 48$, $b = 18$: since $a > b$, we compute 48 mod 18 = 12, so a becomes 12. In the next step, $b > a$, so 18 mod 12 = 6, and b becomes 6. In the following step, $a > b$ and 12 mod 6 = 0, so a is set to $b = 6$. Now $a = b = 6$, and the algorithm returns 6.

It is easy to see that the algorithm always terminates. After each iteration, $\max(a, b)$ strictly decreases and both values remain positive integers, so a must equal b after finitely many steps.

We use the *mod* function to update the larger value. This is what makes the algorithm fast: every two iterations shrink $\max(a, b)$ by at least a factor of two, so the running time is $O(\log(\min(a, b)))$.

Suppose that we want to compute the factorial of a non-negative integer n. Algorithm Factorial shows a method that uses *recursion*. A recursive approach requires specification of the *base case*, when n equals 0. For a higher value of n, we use the value of Factorial$(n-1)$ to compute Factorial(n).

As a final simple example of an algorithm, suppose that we are required to find the largest element in an array A. Algorithm MaxElement uses a *for* loop to find this element.

Algorithm Factorial: Recursive factorial computation

input: a non-negative integer n
output: $n!$

```
1 if n = 0 then return 1 ;
2 return n · Factorial(n − 1)
```

Algorithm MaxElement: Finding the maximum element in an array

input: $A[0..n-1]$: array of int
output: maximum element in A

```
1 max := A[0] ;
2 for i = 1 to n − 1 do
3 |   if A[i] > max then max := A[i] ;
4 end
5 return max
```

1.3 Asymptotic Notation (Big O, Big Omega, Big Theta)

Asymptotic notation quantifies runtime growth, abstracting constants and lower-order terms to focus on scalability. It is the cornerstone of algorithmic comparison.

- $T(n) = O(f(n))$ if $\exists c, n_0 > 0$ such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$. For example, suppose that $T(n) = 2n + 5$. Since $2n + 5 \leq 3n$ for $n \geq 5$, we conclude that $T(n)$ equals $O(n)$.

- $T(n) = \Omega(f(n))$ if $\exists c, n_0 > 0$ such that $T(n) \geq c \cdot f(n)$ for all $n \geq n_0$. For example, suppose that $T(n) = n^2 - n$. Since $n^2 - n \geq 0.5n^2$ for $n \geq 2$, we conclude that $T(n)$ equals $\Omega(n^2)$.

- $T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$. For example, suppose that $T(n) = 4n^2 + 3n$, $3n^2 \leq 4n^2 + 3n \leq 5n^2$ for $n \geq 3$. We conclude that $T(n)$ equals $\Theta(n^2)$.

Example 1.2 We first give an example, where $T(n)$ is a linearly growing function of n. Let $T(n) = 7n + 10$. Then, it is easy to see that $T(n)$ equals $O(n)$: $7n + 10 \leq 8n$ for $n \geq 10$. In addition, $T(n)$ equals $\Omega(n)$: $7n + 10 \geq 7n$. Thus, $T(n)$ equals $\Theta(n)$.

Example 1.3 As another example, suppose $T(n) = 2n^3 + 5n^2 + n$. Then $T(n)$ equals $O(n^3)$ because $2n^3 + 5n^2 + n \leq 3n^3$ for $n \geq 6$. Similarly, $T(n)$ equals $\Omega(n^3)$ because $2n^3 + 5n^2 + n \geq 2n^3$. Hence, $T(n)$ equals $\Theta(n^3)$.

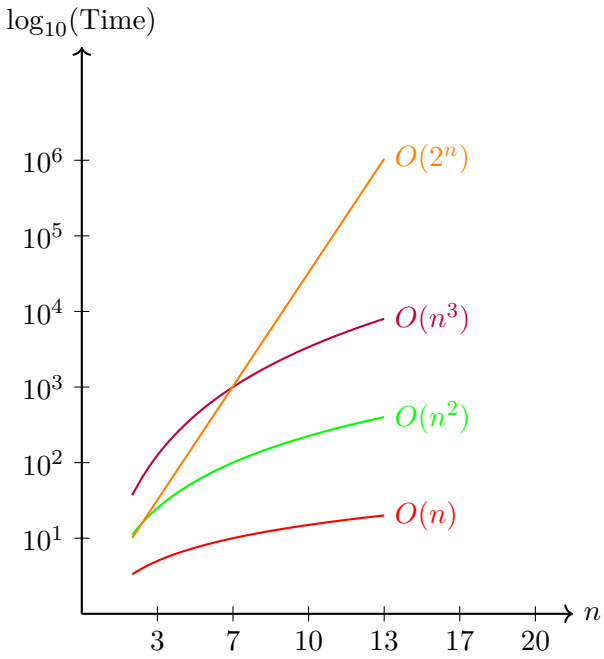


Figure 1.1: Comparison of growth rates for n up to 20: linear, quadratic, cubic, and exponential (log scale)

1.4 Analyzing Algorithm Efficiency

Efficiency analysis evaluates the time and space complexity across the best, worst, and average cases, providing a holistic view of performance. We start with the Algorithm LinearSearch.

Algorithm LinearSearch: Linear search for a key in an array

input: $A[0..n-1]$: array of int; *key*: int
output: index of *key* in A, or -1 if not found

```
1 for i = 0 to n − 1 do
2 |   if A[i] = key then return i ;
3 end
4 return −1
```

The time complexity of the algorithm is as follows.

- *Best*: $O(1)$, key at $A[0]$.
- *Worst*: $O(n)$, key absent.
- *Average*: $\frac{n+1}{2} = O(n)$, assuming the key is present and equally likely to occur at any position.

We now consider the example of a *nested loop* shown in Algorithm NestedSum. The time complexity of the algorithm is $O(n^2)$. The space complexity of the algorithm is $O(1)$. From

Algorithm NestedSum: Sum of products via nested loops

```
   input: integer n
   output: sum of products
1  total := 0 ;
2  for i = 0 to n − 1 do
3  |  for j = 0 to n − 1 do
4  |  |  total := total + i · j ;
5  |  end
6  end
7  return total
```

now on, the time and space complexity will always refer to the *worst case* unless otherwise specified.

Let us consider another example (Algorithm PairSum), in which our goal is to find the number of pairs in an array A that sum to the provided target sum. The reader should verify that the time complexity of the algorithm is $O(n^2)$.

Algorithm PairSum: Counting pairs that sum to a target

```
   input: A[0..n − 1]: array of int; target: int
   output: number of pairs summing to target
1  count := 0 ;
2  for i = 0 to n − 1 do
3  |  for j = i + 1 to n − 1 do
4  |  |  if A[i] + A[j] = target then  count := count + 1 ;
5  |  end
6  end
7  return count
```

We now consider Algorithm BinarySearch, which speeds up Algorithm LinearSearch. Note that linear search works on any array, whether sorted or not. Binary search, however, requires the array to be sorted. Given a *key*, the algorithm compares it to the middle element of the array A. If *key* is equal to the middle element, then we have found the index in the array, *mid*, such that $A[mid]$ equals *key*. If *key* is less than the middle element, then we know that it cannot be on the right side of the array A because the array is sorted. We can then restrict our attention to the left side of the array. Finally, if *key* is greater than the middle element, then we can restrict our attention to the right side of the array. If $T(n)$ equals the time complexity of searching in a sorted array of size n, then the recurrence $T(n) = T(n/2) + 1$ is satisfied. Later in this book, we show that $T(n) = O(\log n)$.

As another example, we describe the problem called Towers of Hanoi. The Towers of Hanoi problem involves three pegs and n disks of increasing size, initially stacked on one peg in decreasing size order. The goal is to move the entire stack to another peg, obeying two rules: only one disk can be moved at a time, and a larger disk cannot be placed on a smaller disk. The solution requires $2^n - 1$ moves, achieved recursively by moving $n - 1$ disks to a spare peg,

Algorithm BinarySearch: Binary search for a key in a sorted array

input: $A[low..high]$: sorted array of int; key: int
output: index of key in A, or -1 if not found

```
1 if low > high then return −1 ;
2 mid := ⌊(low + high)/2⌋ ;
3 if A[mid] = key then return mid ;
4 else if A[mid] > key then return BinarySearch(A, key, low, mid − 1) ;
5 else return BinarySearch(A, key, mid + 1, high) ;
```

transferring the largest disk, then moving the $n-1$ disks onto the largest. Algorithm Hanoi shows a recursive solution to this problem. Here, the recurrence is: $T(n) = 2T(n-1) + 1$, $T(1) = 1$. On solving this recurrence, we get $T(n) = 2^n - 1$.

Algorithm Hanoi: Towers of Hanoi

input: n: number of disks; $source, aux, target$: pegs

```
1 if n = 1 then Move disk from source to target ;
2 else
3 |   Hanoi(n − 1, source, target, aux) ;
4 |   Move disk from source to target ;
5 |   Hanoi(n − 1, aux, source, target) ;
6 end
```

Finally, we consider the problem of sorting an array A using a method called *MergeSort*. The algorithm uses recursion to sort an array using a method called divide-and-conquer. It sorts the left half of the array and the right half of the array, recursively. Now, given that the left side and the right side of the array are sorted, it merges these two sides to get a fully sorted array. The Merge procedure is described later in Section 4.4. In this example, we have the recurrence $T(n) = 2T(n/2) + n$. We later show that the recurrence can be solved to determine that $T(n) = O(n \log n)$.

Algorithm MergeSort: Merge sort

input: $A[low..high]$: array of int
output: array $A[low..high]$ sorted in non-decreasing order

```
1 if low < high then
2 |   mid := ⌊(low + high)/2⌋ ;
3 |   MergeSort(A, low, mid) ;
4 |   MergeSort(A, mid + 1, high) ;
5 |   Merge(A, low, mid, high) ;
6 end
```

1.5 Common Data Structures

We briefly review four fundamental data structures: linked lists, stacks, simple queues, and binary search trees. We assume the reader has seen these before; our goal here is to establish the time complexity of each operation for use in later chapters.

Linked Lists

A linked list is a linear sequence of nodes, where each node contains data and a reference (pointer) to the next node. A singly linked list supports traversal in one direction; a doubly linked list adds a pointer to the previous node, enabling traversal in both directions.

- **Insert at head**: Create a new node and set its next pointer to the current head. $O(1)$.
- **Insert at tail**: If a tail pointer is maintained, append a new node and update the tail pointer. $O(1)$ with a tail pointer; $O(n)$ otherwise, since we must traverse the entire list.
- **Delete**: Given a pointer to the node, redirect the predecessor's next pointer to skip it. $O(1)$ with a direct pointer (in a doubly linked list or when the predecessor is known); $O(n)$ in a singly linked list if the node must first be found by traversal.
- **Search**: Walk the list from head to tail, comparing each element. $O(n)$ in the worst case.

Stacks

A stack is a Last-In-First-Out (LIFO) structure, typically implemented with an array or a linked list. Elements are added and removed at one end, called the *top*.

- **Push**: Add an element to the top. $O(1)$.
- **Pop**: Remove and return the element at the top. $O(1)$.
- **Top** (or **Peek**): Return the top element without removing it. $O(1)$.

Simple Queues

A simple queue is a First-In-First-Out (FIFO) structure. Elements are added at the rear and removed from the front. A linked-list implementation maintains pointers to both ends.

- **Enqueue**: Add an element at the rear. $O(1)$.
- **Dequeue**: Remove and return the element at the front. $O(1)$ with a linked list or a circular array; $O(n)$ with a naive array implementation that shifts all elements.
- **Front** (or **Peek**): Return the front element without removing it. $O(1)$.

Binary Search Trees

A binary search tree (BST) is a binary tree in which every node's left subtree contains only smaller values and its right subtree contains only larger values. This ordering invariant enables efficient search by eliminating half the remaining candidates at each step.

- **Search**: Starting at the root, compare the target with the current node and recurse into the left or right subtree accordingly. $O(h)$, where h is the height of the tree.
- **Insert**: Search for the correct position (a null child pointer) and place the new node there. $O(h)$.
- **Delete**: Locate the node, then handle three cases: a leaf is simply removed; a node with one child is replaced by that child; a node with two children is replaced by its in-order successor (or predecessor), which is then deleted from its original position. $O(h)$.

In the worst case (a degenerate tree shaped like a linked list), $h = n$. A balanced BST guarantees $h = O(\log n)$, giving $O(\log n)$ time for all three operations.

1.6 Heaps

We now describe a data structure *heap* that is useful for many classical algorithms, such as Dijkstra's shortest path algorithm. A heap is a complete binary tree satisfying the heap property: In a minimum heap, the value of each parent is less than or equal to the values of its children. We focus on min-heaps here, implemented as arrays.

Structure and Representation

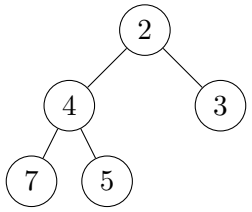


Figure 1.2: Min-heap as a binary tree

2	4	3	7	5
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$

Figure 1.3: Min-heap as an array

Figures 1.2 and 1.3 show a min-heap with values 2, 4, 3, 7, 5.

Heapify

The Heapify algorithm ensures the min-heap property at a given node by comparing it with its children and swapping with the smallest if necessary, then recursively applying the process downward. It takes an array A, an index i, and a heap size n, running in $O(\log n)$ time due to the height of the tree.

Algorithm Heapify: Restoring the min-heap property at a node

input: array A, index i, heap size n

```
1 smallest := i ;
2 left := 2i + 1 ;
3 right := 2i + 2 ;
4 if left < n and A[left] < A[smallest] then smallest := left ;
5 if right < n and A[right] < A[smallest] then smallest := right ;
6 if smallest ≠ i then
7 |   Swap A[i] and A[smallest] ;
8 |   Heapify(A, smallest, n);
9 end
```

Insert

The Insert algorithm adds a new value to the heap by placing it at the end of the array and shifting it up to its correct position, comparing it with its parent until the min-heap property is restored. It runs in $O(\log n)$ time, as it traverses up the tree's height.

Algorithm HeapInsert: Inserting a key into a min-heap

input: array A, value key, heap size n

```
1 n := n + 1 ;
2 A[n − 1] := ∞ ;
3 i := n − 1 ;
4 while i > 0 and A[⌊(i − 1)/2⌋] > key do
5 |   A[i] := A[⌊(i − 1)/2⌋] ;
6 |   i := ⌊(i − 1)/2⌋;
7 end
8 A[i] := key
```

Extract-Min

The Extract-Min algorithm removes the minimum element (root) by replacing it with the last element, reducing the heap size, and then calling Heapify to restore the min-heap property. It operates in $O(\log n)$ time due to the heapification step.

Algorithm HeapExtractMin: Extracting the minimum from a min-heap

```
   input: array A, heap size n
1  if n < 1 then return error ;
2  min := A[0] ;
3  A[0] := A[n − 1] ;
4  n := n − 1 ;
5  Heapify(A, 0, n) ;
6  return min
```

Build-Heap

The Build-Heap algorithm constructs a min-heap from an unsorted array by iteratively applying Heapify from the last non-leaf node up to the root. It runs in $O(n)$ time, as the cumulative effect of heapifying all subtrees is linear in the number of nodes.

Algorithm BuildHeap: Building a min-heap from an unsorted array

```
   input: array A, size n
1  for i := ⌊n/2⌋ − 1 downto 0 do
2  |  Heapify(A, i, n)
3  end
```

The time complexity of various operations on a heap can be summarized as follows.

- **Heapify**: $O(\log n)$
- **Insert**: $O(\log n)$
- **Extract-Min**: $O(\log n)$
- **Build-Heap**: $O(n)$

1.7 Organization of the Book

Chapter 2 presents the general Lattice Linear Predicate (LLP) algorithm. It shows how many combinatorial optimization problems can be solved by a single framework and introduces the notions of forbidden indices and advance operations.

Chapter 3 presents the stable marriage problem. It gives the classical Gale–Shapley algorithm and shows that the LLP algorithm yields a more general version that handles additional constraints.

Chapter 4 discusses various sorting algorithms, including bubble sort, insertion sort, merge sort, quicksort, bitonic sort, and radix sort.

Chapter 5 discusses basic graph algorithms, including various traversal techniques (BFS, DFS), layering of directed acyclic graphs, and connected components.

Chapter 6 covers greedy algorithms: interval scheduling, interval partitioning, scheduling to minimize lateness, and Huffman coding.

Chapter 7 discusses algorithms for computing the shortest path in a graph, including Dijkstra's algorithm, Bellman–Ford, Floyd–Warshall, and Johnson's algorithm. Both classical and LLP formulations are presented.

Chapter 8 describes algorithms for computing the minimum spanning tree in a weighted undirected graph. By considering different lattices on the set of edges, one can derive classical Kruskal's, Prim's, and Borůvka's algorithms.

Chapter 9 discusses the divide-and-conquer strategy, with applications to closest pair, counting inversions, convex hull, Karatsuba multiplication, Strassen matrix multiplication, and the FFT.

Chapter 10 discusses dynamic programming. Problems include the longest increasing subsequence, the optimal binary search tree, the knapsack problem, weighted interval scheduling, and segmented least squares. All of these are shown to be solvable using the LLP algorithm.

Chapter 11 discusses network flow, including the Ford–Fulkerson and Edmonds–Karp algorithms, the max-flow min-cut theorem, and the lattice of min-cuts.

Chapter 12 describes bipartite matching, including augmenting-path algorithms and maximum antichain computation via LLP.

Chapter 13 describes the theory of NP-completeness. This theory relates the difficulty of solving one problem to that of another and includes reductions among SAT, Clique, Vertex Cover, Hamiltonian Path, and Subset Sum.

Chapter 14 discusses approximation algorithms for NP-hard optimization problems, including a 2-approximation for vertex cover, the greedy H_n-approximation for set cover, and a fully polynomial-time approximation scheme (FPTAS) for the knapsack problem. Both classical and LLP formulations are presented, including primal-dual treatments of weighted vertex cover and weighted set cover. A standard course in algorithms may cover all the chapters up to this point. The remaining chapters may be covered depending on the interest of the instructor.

Chapter 15 discusses the housing allocation problem, a one-sided version of the stable marriage problem solved via the top-trading-cycles algorithm and its LLP formulation.

Chapter 16 describes linear programming, another general algorithm design technique. We show that many problems described using the LLP paradigm can also be solved using linear programming.

Chapter 17 discusses the assignment problem (weighted bipartite matching), with LLP algorithms for computing market-clearing prices.

Chapter 18 discusses subclasses of Boolean satisfiability — Horn formulas and 2-SAT — whose satisfiability can be evaluated efficiently, including an LLP formulation for Horn-SAT.

Chapter 19 discusses number-theoretic algorithms from the lens of LLP: GCD, extended GCD, least common multiple, the Chinese Remainder Theorem, and computing the multiplicative order.

Chapter 20 discusses the predicate detection problem in a poset. It explores special cases of LLP algorithms such as conjunctive predicates and the complexity of determining whether a given predicate is lattice-linear.

1.8 Summary

This chapter provided an introduction to algorithms and their analysis. It covered the definition of an algorithm, asymptotic notation for characterizing growth rates, and efficiency analysis of

several fundamental algorithms. The chapter also reviewed essential data structures—linked lists, stacks, queues, binary search trees, and heaps—that are used throughout the book.

Algorithm	Problem	Time Complexity
Euclid's GCD	Greatest Common Divisor	$O(\log(\min(a,b)))$
Linear Search	Search in unsorted array	$O(n)$
Binary Search	Search in sorted array	$O(\log n)$
Tower of Hanoi	Move n disks	$O(2^n)$
Build-Heap	Build a min-heap from array	$O(n)$

1.9 Problems

1. **(Asymptotic Notation Proofs)** Prove or disprove each of the following:

   (a) $3^n = O(2^n)$

   (b) If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$

   (c) If $f(n) = O(g(n))$, then $f(n)^3 = O(g(n)^3)$

2. **(Big-O Algebraic Properties)** Prove the following statements.

   (a) Prove that for any constant $d > 0$, $f(n) = O(d \cdot g(n))$ iff $f(n) = O(g(n))$.

   (b) Prove that if $f(n) = O(h_1(n))$ and $g(n) = O(h_2(n))$ then

   $$f(n) + g(n) = O(\max\{h_1(n), h_2(n)\}).$$

   (c) Prove that all logarithms are asymptotically equivalent. That is, prove that $\log_a(n) = O(\log_b(n))$ for any positive a and b.

3. **(Build Heap Linear Time)** Show that the time complexity of BuildHeap is $O(n)$.

4. **(kth Smallest Heap)** Suppose that you have a min Heap of size n. Give the pseudo-code to find and delete the k^{th} smallest element in the heap. What is the time complexity of your algorithm in terms of n and k ? Assume that you have a method available to you for a Heap that can insert any element in the heap in $O(\log n)$ time.

5. **(Two-Sum Min Max Heaps)** Suppose that we have an array A of n *distinct* integers in two forms: a min Heap H and a max heap K. We are given the following functions on the min-heap:
   $H.getMin()$ // return the least value in the heap H
   $H.deleteMin()$ // delete the least value from the heap H
   Similarly, we have analogous methods for the max-heap. Give pseudo-code of an efficient program to return a value $A[i]$ in the array A such that there exists j where $A[i] + A[j]$ equals m. If there are no two such distinct entries, then the program should return null. Give the tight asymptotic time complexity of your method.

6. **(Function Growth Ranking)** Rank the following functions by growth rate from slowest to fastest. For each consecutive pair in your ordering, state whether the relationship is O, Θ, or neither.

$$n^2, \quad 2^n, \quad n \log n, \quad n!, \quad \log n, \quad n^{1.5}, \quad n \log^2 n$$

7. **(Linear Recurrence Solution)** Solve the recurrence $T(n) = 2\,T(n/2)+n$ with $T(1) = 1$ and prove that $T(n) = \Theta(n \log n)$.

8. **(k Largest Min Heap)** Given an unsorted array A of n integers and an integer $k \leq n$, describe an algorithm that returns the k largest elements in A in $O(n \log k)$ time using a min-heap of size k. Prove the time bound.

9. **(Binary Search Correctness)** Prove the correctness of Algorithm BinarySearch using a loop invariant. State the invariant precisely and show that it holds at initialization, is maintained by each iteration, and implies correctness at termination.

10. **(Stack Merging Sums)** Let A be an array of n integers and let S be an initially empty stack. Consider the following algorithm: for $i = 1$ to n, push $A[i]$ onto S. After each push, while S is not empty and the top two elements of S are equal, pop both and push their sum. What does S contain when the algorithm finishes on input $A = [2, 2, 4, 8, 2, 2, 4]$? What is the worst-case time complexity of the algorithm? Justify your answer using an amortized argument.

11. **(Euclid GCD Implementation)** Implement both versions of Euclid's GCD algorithm in Java: (a) the subtraction-based version, which replaces the larger value by the difference of the two values, and (b) the version that uses the *mod* operator instead of repeated subtraction (Algorithm GCD from this chapter). Run both on the input pair $(a, b) = (10^9 + 7,\ 10^8 + 7)$ and compare the number of iterations each version requires. Report the ratio.

12. **(Min Heap Implementation)** Implement a min-heap in Java with the following operations:
`insert(int key)`, `extractMin()`, `buildHeap(int[] A)`.
Use a 1-based array as the underlying storage. Test your implementation by inserting the values $[15, 10, 20, 17, 8, 25, 18, 5]$ one at a time, then extracting all elements in order; verify that they come out sorted. As a second test, build a heap from the same array in $O(n)$ time using `buildHeap` and verify the output is the same.

13. **(Merge Sort Comparisons)** Implement Merge Sort in Java. Instrument your code to count the exact number of element comparisons made during the sort. Run it on random arrays of sizes $n = 10^3,\ 10^4,\ 10^5$, and 10^6. For each n, report the measured comparison count and the ratio count$/(n \log_2 n)$. Is the ratio approximately constant?

14. **(Next Greater Element)** Given an array A of n integers, the *next greater element* of $A[i]$ is the first element to the right of $A[i]$ that is strictly larger than $A[i]$, or -1 if no such element exists. Implement a Java program that computes the next-greater-element array in $O(n)$ time using a stack. Test on $A = [4, 5, 2, 25, 7, 8, 6, 1]$ and verify the output is $[5, 25, 25, -1, 8, -1, -1, -1]$.

1.10 Bibliographic Remarks

There are many excellent books on algorithms. The classic graduate text is Cormen, Leiserson, Rivest, and Stein [CLRS09], commonly known as "CLRS." Other standard references include Kleinberg and Tardos [KT06], Sedgewick and Wayne [SW11], the design-manual treatment of Skiena [Ski08], the more compact Dasgupta, Papadimitriou, and Vazirani [DPV08], and the open-access text of Erickson [Eri19a]. The foundational text of Aho, Hopcroft, and Ullman [AHU74a] remains a touchstone, as does Horowitz and Sahni [HSR01]. Moore and Mertens [MM11] give a sweeping treatment that connects algorithms with computational complexity and statistical physics. Knuth's multivolume *The Art of Computer Programming* [Knu97a, Knu98] is the authoritative reference for asymptotic notation and the analysis of fundamental data structures.

For a discussion of the asymptotic notation of the algorithm time complexity, we refer the reader to Knuth [Knu97a]. Heaps were introduced by J.W.J. Williams in 1964 as part of the heapsort algorithm, which leverages the heap structure for efficient sorting in $O(n \log n)$ time. The linear-time Build-Heap construction was later analyzed by Floyd, showing its $O(n)$ complexity. Heaps have since become foundational in priority queue implementations, with applications in scheduling, graph algorithms (e.g., Dijkstra's), and data compression (e.g., Huffman coding).

Chapter 2

Lattice Linear Predicate Detection

2.1 Introduction

Many classical combinatorial optimization problems — stable marriage, shortest paths, minimum spanning trees, market-clearing prices, housing allocation — appear at first glance to require entirely different algorithmic techniques. The Gale–Shapley algorithm proposes and rejects; Dijkstra's algorithm relaxes edges in priority order; Borůvka's algorithm merges components. Yet a closer look reveals a shared structure: in every case, the set of feasible solutions is closed under a natural "meet" operation, and the optimal solution is the *least* element of a *distributive lattice* (see Appendix 21 for background on lattices) that satisfies a Boolean predicate expressing feasibility.

In this chapter, we formalize this observation as the Lattice-Linear Predicate (LLP) algorithm. The framework has three ingredients:

1. A *distributive lattice* L that models the search space. Each element of L is a candidate solution represented as a vector of component values.

2. A Boolean *predicate* B on L that captures the goal: $B(G)$ is true when G is a feasible (or optimal) solution.

3. A *lattice-linearity* condition on B: whenever $B(G)$ is false, at least one component j of G is *forbidden* — meaning that no solution above G with the same value of $G[j]$ can satisfy B. Advancing that component brings G closer to a feasible solution.

Once these ingredients are in place, a single generic algorithm repeatedly advances forbidden components until the predicate is satisfied (or the top of the lattice is reached, signaling that no solution exists). The algorithm is *correct by construction*: lattice-linearity guarantees that every advance makes irrevocable progress, so the algorithm terminates at the unique least fixpoint.

The practical payoff is substantial. The LLP framework unifies a wide range of problems that are traditionally taught with separate algorithms:

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

Problem	Classical algorithm	LLP lattice
Stable marriage	Gale–Shapley	proposal vectors
Single-source shortest path	Dijkstra, Bellman–Ford	distance vectors
Minimum spanning tree	Kruskal, Prim, Borůvka	edge-selection vectors
Market-clearing prices	Demange–Gale–Sotomayor	price vectors
Housing allocation	Top-trading cycles	allocation vectors
Horn satisfiability	Unit propagation	truth-assignment vectors
Job scheduling	Critical-path method	completion-time vectors

Table 2.1: Problems unified by the LLP framework. Each classical algorithm is a specific schedule of the generic LLP algorithm on the corresponding lattice.

In each case, the classical algorithm emerges as a *specific schedule* of the generic LLP algorithm — a particular order in which forbidden components are advanced. Different schedules yield different classical algorithms for the same problem (for example, Dijkstra's algorithm and Bellman–Ford are two schedules of the same LLP instance).

A key advantage of the LLP viewpoint is *composability*: because the conjunction of two lattice-linear predicates is itself lattice-linear (Lemma 2.3(c)), one can add side constraints to any of the above problems — forced or forbidden pairs in stable marriage, fairness constraints in shortest paths, budget caps in market-clearing prices — and the same algorithm solves the constrained version with no additional machinery.

This chapter is organized as follows. Section 2.2 formally defines the lattice-linear predicates. These predicates are defined on a distributive lattice. The problem is framed as the search for an element in the lattice that satisfies the predicate. Generally, we are interested in the least element in the lattice that satisfies the predicate. Section 2.3 gives the notation that we use for programs based on lattice-linear predicates. It specifies the lattice that we are working on, the starting element in the lattice, the top element of the lattice, and the predicate *forbidden*, which allows us to advance in the lattice. Section 2.5 lists some desirable properties of the LLP algorithm. The algorithm is naturally *nondeterministic*. There are multiple ways in which the algorithm can advance in the lattice. The algorithm can advance on multiple components simultaneously. The algorithm is *online*. It does not inspect any element higher than the current element of the lattice that is under consideration. Finally, the algorithm returns an optimal answer for all components. Section 2.8 gives the dual of the lattice-linearity property. Many problems such as the stable marriage problem satisfy not only lattice-linearity but also its dual. For such predicates, one can also start from the top element of the lattice and traverse it downwards looking for an element that satisfies the given predicate. When applied to the stable marriage problem, one can find the woman-optimal stable marriage in this manner.

2.2 Lattice-Linear Predicates

A Warm-up Trace

Before stating the formal definitions, we walk through a tiny instance to expose the mechanics. Consider a job-scheduling problem with three jobs $1, 2, 3$ that require times $t = [2, 3, 1]$ to complete, where jobs 2 and 3 each depend on job 1. Let $G[j]$ denote the (candidate) completion

time of job j. A feasible schedule must satisfy

$$B_{jobs}(G) \equiv \forall j : G[j] \geq t_j \wedge \forall i \in pre(j) : G[j] \geq G[i] + t_j.$$

We search the lattice of completion-time vectors starting from the bottom, $G = [0, 0, 0]$. At each step we look for a single index whose constraint is violated — such an index is called *forbidden* — and *advance* (raise) its value just enough to remove the violation:

G before	violation found	G after
$[0, 0, 0]$	$G[1] < t_1 = 2$	$[2, 0, 0]$
$[2, 0, 0]$	$G[2] < G[1] + t_2 = 5$	$[2, 5, 0]$
$[2, 5, 0]$	$G[3] < G[1] + t_3 = 3$	$[2, 5, 3]$
$[2, 5, 3]$	none — B_{jobs} holds	—

The trajectory $[0, 0, 0] \to [2, 0, 0] \to [2, 5, 0] \to [2, 5, 3]$ is monotone in the componentwise order: no value ever decreases, and the algorithm terminates at a feasible vector. Two properties make this work. First, when index j is identified as forbidden, B_{jobs} cannot be satisfied without raising $G[j]$, so we never overshoot. Second, whenever B_{jobs} is false, at least one forbidden index exists — so progress is always available. The rest of this section formalizes these two properties as the *forbidden* predicate and *lattice-linearity*; a predicate with both admits the simple iterative algorithm sketched above, climbing from the bottom of the lattice straight to the minimum feasible point.

Let L be the lattice of all n-dimensional vectors of reals greater than or equal to the zero vector and less than or equal to a given vector T where the order on the set of vectors is defined by the natural componentwise $\leq$. The minimum element of this lattice is the zero vector. The lattice is used to model the search space of the combinatorial optimization problem. For simplicity, we are considering the lattice of vectors of nonnegative reals; later, we show that our results are applicable to any distributive lattice. The combinatorial optimization problem is modeled as finding the minimum element in L that satisfies a boolean *predicate* B, where B models *feasible solutions*. We are interested in algorithms to solve the combinatorial optimization problem with n processes. We will assume that the system maintains as its state the current candidate vector $G \in L$ in the search lattice. Even if we are using a sequential algorithm, we say that $G[i]$ is maintained in the process i. We call G, the global state, and $G[i]$, the state of process i.

Finding an element in the lattice that satisfies the given predicate B is called the *predicate detection problem*. Finding the *minimum* element that satisfies B (whenever it exists) is the combinatorial optimization problem. We now define *lattice-linearity*, which enables efficient computation of this minimum element. Lattice-linearity was originally introduced in the context of detecting global conditions in a distributed system, where it is simply called linearity. We use the term *lattice-linearity* to avoid confusion with the standard usage of linearity. A key concept in the development of an efficient predicate detection algorithm is that of a *forbidden* state. Given a predicate B and a vector $G \in L$, a state $G[i]$ is *forbidden* (or equivalently, the index i is forbidden) if for any vector $H \in L$, where $G \leq H$, if $H[i]$ equals $G[i]$, then B is false for H. Formally,

Definition 2.1 (Forbidden State) *Given any distributive lattice L of n-dimensional vectors of $\mathbf{R}_{\geq 0}$, and a predicate B, we define $forbidden(G, i, B) \equiv \forall H \in L : G \leq H : (G[i] = H[i]) \Rightarrow \neg B(H)$.*

We define a predicate B as *lattice-linear* with respect to a lattice L if, for any global state G, B is false in G implies that G contains a *forbidden state*. Formally,

Definition 2.2 (Lattice-Linear Predicate) *A boolean predicate B is* lattice-linear *with the polytime algorithm $\mathcal{A}$ with respect to a finite distributive lattice L generated from a poset P iff $\forall G \in L : \neg B(G) \Rightarrow (\exists i \in \mathcal{A}(G) : forbidden(G, i, B))$.*

The algorithm $\mathcal{A}$ is efficient in the size of the poset P. It tells us which indices are forbidden. From now on, the algorithm $\mathcal{A}$ is implicit. We now give some examples of lattice-linear predicates.

1. **Job Scheduling Problem**: Our first example relates to scheduling of n jobs. Each job j requires time t_j for completion and has a set of prerequisite jobs, denoted by $pre(j)$, such that it can be started only after all its prerequisite jobs have been completed. Our goal is to find the minimum completion time for each job. We let our lattice L be the set of all possible completion times. A completion vector $G \in L$ is feasible iff $B_{jobs}(G)$ holds where $B_{jobs}(G) \equiv \forall j : (G[j] \geq t_j) \wedge (\forall i \in pre(j) : G[j] \geq G[i] + t_j)$. B_{jobs} is lattice-linear because if it is false, then there exists j such that either $G[j] < t_j$ or $\exists i \in pre(j) : G[j] < G[i] + t_j$. We claim that $forbidden(G, j, B_{jobs})$. Indeed, any vector $H \geq G$ cannot be feasible with $H[j]$ equal to $G[j]$. The minimum of all vectors that satisfy feasibility corresponds to the minimum completion time.

2. **Prefix Sum Problem**: Our second example relates to (exclusive) prefix sum of an array A with non-negative reals. We are required to output an array G such that $G[j]$ equals sum of all entries in A from 0 to $j-1$. We define G to be feasible iff B_{prefix} holds where $B_{prefix} \equiv (\forall j > 0) : (G[j] \geq G[j-1] + A[j-1])$. Again, it is easy to verify that B_{prefix} is lattice-linear. The minimum vector G that satisfies B_{prefix} corresponds to the exclusive prefix sum of the array A.

3. **Continuous Optimization Problem**: We are required to find minimum nonnegative x and y such that $B \equiv (x \geq y^2/4 + 5) \wedge (y \geq x - 4)$. We view this problem as finding the minimum (x, y) pair such that B holds. It is easy to verify that B is lattice-linear. If the first conjunct is false, then x is forbidden. Unless x is increased, the predicate cannot become true, even if other variables (y for this example) increase. If the second conjunct is false, then y is forbidden. In this case, $x = 6$ and $y = 2$ is the pointwise minimum solution. For some predicates, there may not be any solution. For example, when $B \equiv (x \geq 2y^2 + 5) \wedge (y \geq x - 4)$, there is no nonnegative (x, y) pair such that B holds (verify this!). The predicate B is still lattice-linear, but by advancing along x and y we go beyond any bounded (x, y).

4. **A Non-Lattice-Linear Predicate**: As an example of a predicate that is not lattice-linear, consider the predicate $B \equiv \sum_j G[j] \geq 1$ defined on the space of two-dimensional

vectors. Consider the vector G equal to $(0,0)$. The vector G does not satisfy B. For B to be lattice-linear, either the first index or the second index should be forbidden. However, none of the indices are forbidden in $(0,0)$. The index 0 is not forbidden because the vector $H = (0,1)$ is greater than G, has $H[0]$ equal to $G[0]$ but it still satisfies B. The index 1 is also not forbidden because $H = (1,0)$ is greater than G, has $H[1]$ equal to $G[1]$ but it satisfies B.

The following lemma is useful for proving lattice-linearity of predicates.

Lemma 2.3 *Let B be any Boolean predicate defined on a lattice L of vectors.*

(a) *Let $f : L \rightarrow \mathbf{R}_{\geq 0}$ be any monotone function defined on the lattice L of vectors of $\mathbf{R}_{\geq 0}$. Consider the predicate $B \equiv G[i] \geq f(G)$ for some fixed i. Then, B is lattice-linear.*

(b) *Let L_B be the subset of the lattice L of the elements that satisfy B. Then, B is lattice-linear implies that L_B is closed under meets.*

(c) *If B_1 and B_2 are lattice-linear then $B_1 \wedge B_2$ is also lattice-linear.*

Proof:

(a) Suppose B is false for G. This implies that $G[i] < f(G)$. Consider any vector $H \geq G$ such that $H[i]$ is equal to $G[i]$. Since $G[i] < f(G)$, we get that $H[i] < f(G)$. The monotonicity of f implies that $H[i] < f(H)$ which shows that $\neg B(H)$.

(b) We show the contrapositive. Let $Y = \{H_1, H_2, \ldots, H_k\}$ be any subset of L_B such that its infimum G does not belong to L_B. Since G is the infimum of Y, for any i, there exists $j \in \{1 \ldots k\}$ such that $G[i] = H_j[i]$. Since $B(H_j)$ is true for all j, for each i, the witness H_j shows i is not forbidden in G. Thus none of the indices in G are forbidden, yet $\neg B(G)$, so lattice-linearity does not hold.

(c) Suppose that $\neg(B_1 \wedge B_2)$. This implies that one of the conjuncts is false and therefore, from the lattice-linearity of that conjunct, a forbidden state exists.

■

Thus lattice-linearity implies meet-closure of the satisfying set, but the converse need not hold: a predicate can be meet-closed without being lattice-linear.

For the job scheduling example, we define B_j as $G[j] \geq max(t_j, max\{G[i]+t_j \mid i \in pre(j)\})$. Since $f_j(G) = max(t_j, max\{G[i] + t_j \mid i \in pre(j)\})$ is a monotone function, it follows from Lemma 2.3(a) that B_j is lattice-linear. The predicate $B_{jobs} \equiv \forall j : B_j$ is lattice-linear due to Lemma 2.3(c). Also note that the problem of finding the minimum vector that satisfies B_{jobs} is well-defined due to Lemma 2.3(b).

We now discuss detection of lattice-linear predicates which requires an additional assumption called the *efficient advancement property* — there exists an efficient (polynomial time) algorithm to determine the forbidden state. This property holds for all the problems considered in this book. Once we determine j such that $forbidden(G, j, B)$, we also need to determine how to advance along the index j. To that end, we extend the definition of forbidden as follows.

Definition 2.4 (α-forbidden) *Let B be any Boolean predicate on the lattice L of all assignment vectors. For any G, j and positive real $\alpha > G[j]$, we define $forbidden(G, j, B, \alpha)$ iff*

$$\forall H \in L : H \geq G : (H[j] < \alpha) \Rightarrow \neg B(H).$$

Given any lattice-linear predicate B, suppose $\neg B(G)$. This means that G must be advanced on all indices j such that $forbidden(G, j, B)$. We use a function $\alpha(G, j, B)$ such that $forbidden(G, j, B, \alpha(G, j, B))$ holds whenever $forbidden(G, j, B)$ is true. With the notion of $\alpha(G, j, B)$, we have the algorithm LLP shown in Fig. LLP. The algorithm LLP has two inputs, the predicate B and the top element of the lattice T. It returns the least vector G which is less than or equal to T and satisfies B (if it exists). Whenever B is not true in the current vector G, the algorithm advances on all forbidden indices j in parallel. This simple algorithm can be used to solve a wide variety of combinatorial optimization problems by instantiating different $forbidden(G, j, B)$ and $\alpha(G, j, B)$.

Algorithm LLP: Algorithm LLP to find the minimum vector at most T that satisfies B

input: T: vector (top element), B: predicate
output: the minimum vector $G \leq T$ that satisfies B, or null

```
1 var G: vector of reals initially ∀i : G[i] := 0;
2 while ∃j : forbidden(G, j, B) do
3 |   forall j such that forbidden(G, j, B) in parallel do
4 |   |   if α(G, j, B) > T[j] then return null;
5 |   |   else G[j] := α(G, j, B);
6 |   end
7 end
8 return G ;                                   // the optimal solution
```

Theorem 2.5 *Suppose that there exists a fixed constant $\delta > 0$ such that $\alpha(G, j, B) - G[j] \geq \delta$ whenever $forbidden(G, j, B)$. Then, the algorithm LLP finds the least vector $G \leq T$ that satisfies B, if one exists.*

Proof: Since $G[j]$ increases by at least δ for at least one forbidden j in every iteration of the *while* loop, the algorithm terminates in at most $\sum_i \lceil T[i]/\delta \rceil$ number of steps.

We show that the algorithm maintains the invariant $(I1)$ that for all indices j, any vector V such that $V[j]$ is less than $G[j]$ cannot satisfy B. Formally, the invariant $(I1)$ is

$$\forall j : (\forall V \in L : (V[j] < G[j]) \Rightarrow \neg B(V)).$$

Initially, the invariant holds trivially because G is initialized to 0. Suppose $forbidden(G, j, B)$. Then, we increase $G[j]$ to $\alpha(G, j, B)$. We need to show that this change maintains the invariant. Pick any V such that $V[j] < \alpha(G, j, B)$. We now do a case analysis. If $V \geq G$, then $\neg B(V)$

holds from the definition of $\alpha(G, j, B)$. Otherwise, there exists some k such that $V[k] < G[k]$. In this case, $\neg B(V)$ holds due to $(I1)$.

We can now show Theorem 2.5 using the invariant. First, suppose that the algorithm LLP terminates because $\alpha(G, j, B) > T[j]$. In this case, there is no feasible vector in L due to the invariant (because the predicate B is false for all values of $G[j]$). Now suppose that the algorithm terminates because there does not exist any j such that $forbidden(G, j, B)$. This implies that G satisfies B due to lattice-linearity of B. It is also the least vector that satisfies B due to the invariant $(I1)$.

■

For the job scheduling example, we get an algorithm to find the minimum completion time using $forbidden(G, j, B_{jobs}) \equiv (G[j] < t_j) \vee (\exists i \in pre(j) : G[j] < G[i] + t_j)$, and $\alpha(G, j, B_{jobs}) = \max\{t_j, \max\{G[i] + t_j | i \in pre(j)\}\}$.

For the prefix sum example, we get an algorithm using $forbidden(G, j, B_{prefix}) \equiv (G[j] < G[j-1] + A[j-1])$ and $\alpha(G, j, B_{prefix}) = G[j-1] + A[j-1]$ for all $j > 0$.

We now show, on account of Lemma 2.3(c), that if we have an algorithm for a problem, then we also have one for the constrained version of that problem.

Lemma 2.6 *Let LLP be the algorithm to find the least vector G that satisfies B_1 if one exists. Then, LLP can be adapted to find the least vector G that satisfies $B_1 \wedge B_2$ for any lattice-linear predicate B_2.*

Proof: The algorithm LLP can be used with the following changes:
$forbidden(G, j, B_1 \wedge B_2) \equiv forbidden(G, j, B_1) \vee forbidden(G, j, B_2)$, and
$\alpha(G, j, B_1 \wedge B_2) = \max\{\alpha(G, j, B_1), \alpha(G, j, B_2)\}$.

■

For example, suppose that we want the minimum completion time of jobs with the additional lattice-linear constraint that $B_2(G) \equiv (G[1] = G[2])$. B_2 is lattice-linear with $forbidden(G, 1, B_2) \equiv (G[1] < G[2])$ and $forbidden(G, 2, B_2) \equiv (G[2] < G[1])$. Applying Lemma 2.6, we get an algorithm for the constrained version.

2.3 LLP Notation

We now describe the notation used in LLP algorithms. Fig. 2.1 shows LLP algorithms for the job scheduling problem. We have a single variable G in all the examples shown in Fig. 2.1.

All other variables are derived directly or indirectly from G. G is an array of objects such that $G[j]$ is the state of component j. There are three sections of the program.

The **var section** declares the state variables and their initial values. For example, the *var* section of the job scheduling program in Fig. 2.1 specifies that $G[j]$ is an integer initially equal to $t[j]$.

The **always section** defines additional variables that are derived from G. The actual implementation of these variables is left to the system. They can be viewed as macros.

The third section gives the desirable predicate, either by using the **forbidden** predicate or the **ensure** predicate. The *forbidden* predicate has an associated *advance clause* that specifies how $G[j]$ must be advanced whenever the forbidden predicate is true. For many problems, it is more convenient to use the complement of the forbidden predicate. The *ensure section* specifies the desirable predicates of the form $(G[j] \geq expr)$ or $(G[j] \leq expr)$. The statement *ensure* $G[j] \geq expr$ simply means that whenever component j finds $G[j]$ to be less than $expr$, it advances $G[j]$ to $expr$. Since $expr$ may refer to G, just by setting $G[j]$ equal to $expr$, there is no guarantee that $G[j]$ continues to be equal to $expr$ — the value of $expr$ may change due to changes in other components. We use the *ensure* statement whenever $expr$ is a monotonic function of G and therefore the predicate is lattice-linear.

(a) Using **forbidden**/**advance**:

Algorithm LLP-JobScheduling-Forbidden: Job scheduling via forbidden/advance

input: t: array$[0..n-1]$ of int; pre: array$[0..n-1]$ of sets of int
output: G: array$[0..n-1]$ of int, initially $t[i]$

1 **forbidden**(j): $G[j] < \max\{G[i] + t[j] \mid i \in pre(j)\}$
2 $\Rightarrow$ **advance**: $G[j] := \max\{G[i] + t[j] \mid i \in pre(j)\}$;

(b) Using **ensure** clause:

Algorithm LLP-JobScheduling-Ensure: Job scheduling via ensure

input: t: array$[0..n-1]$ of int; pre: array$[0..n-1]$ of sets of int
output: G: array$[0..n-1]$ of int, initially $t[i]$

1 **ensure**(j): $G[j] \geq \max\{G[i] + t[j] \mid i \in pre(j)\}$;

Figure 2.1: LLP program for the job scheduling problem: (a) using the *forbidden*/*advance* notation, (b) using the compact *ensure* clause. Both are equivalent; the *ensure* form is used when the advance expression is a monotone function of G.

Consider four jobs with execution times $t = (3, 2, 4, 1)$ and prerequisites $pre(2) = \{1\}$, $pre(3) = \{1\}$, $pre(4) = \{2, 3\}$ (job 1 has no prerequisites). The LLP algorithm initializes $G = (3, 2, 4, 1)$. Depending on the evaluation order, a job may need to be advanced more than once.

Example 2.7 Suppose we evaluate job 4 before job 3:

1. Job 1 not forbidden. Advance job 2: $G = (3, 5, 4, 1)$.
2. Job 4: requires $\max(G[2] + 1, G[3] + 1) = \max(6, 5) = 6$, but $G[4] = 1$. Advance: $G[4] := 6$. Now $G = (3, 5, 4, 6)$.
3. Job 3: requires $G[3] \geq G[1] + 4 = 7$, but $G[3] = 4$. Advance: $G[3] := 7$. Now $G = (3, 5, 7, 6)$.
4. Job 4 is forbidden *again*: requires $\max(6, 8) = 8$, but $G[4] = 6$. Advance: $G[4] := 8$. Now $G = (3, 5, 7, 8)$.

Job 4 was advanced *twice* because it was processed before its prerequisite job 3 was settled.

An Efficient Reformulation Using Fixed Indices

To avoid such repeated advances, we reformulate the forbidden predicate using a $fixed$ array. Note that this is an implementation refinement: the state is augmented with the $fixed$ array, so the lattice is no longer just the space of completion-time vectors G. Let $fixed[j]$ be a boolean, initially false for all j. A job j is forbidden if $\neg fixed[j] \wedge (\forall i \in pre(j) : fixed[i])$. When j is forbidden, we advance $G[j] := \max\{G[i] + t[j] \mid i \in pre(j)\}$ and set $fixed[j] := true$. Since all predecessors are already fixed (their values will not change), this advance is final — each job becomes forbidden at most once. Algorithm LLP-JobScheduling-Fixed gives the LLP program.

Algorithm LLP-JobScheduling-Fixed: Job scheduling with fixed predicates

input: t: array$[0..n-1]$ of int; pre: array$[0..n-1]$ of sets of int
output: G: array$[0..n-1]$ of int, initially $t[i]$; *fixed*: array$[0..n-1]$ of boolean, initially *false*

1 **forbidden**(j): $\neg fixed[j] \wedge \forall i \in pre(j) : fixed[i]$
2 $\Rightarrow$ **advance**: $G[j] := \max(t[j], \max\{G[i] + t[j] \mid i \in pre(j)\})$; $fixed[j] := true$;

The efficiency of the algorithm may depend on the formulation of $\alpha(G, j, B)$. We will see such optimizations for many problems later in the book, including the shortest-path algorithm.

Example 2.8 Using the same four-job instance, we trace Algorithm LLP-JobScheduling-Fixed. Initialize $G = (3, 2, 4, 1)$ and $fixed = (\text{false}, \text{false}, \text{false}, \text{false})$.

1. Job 1: $pre(1) = \emptyset$, so all predecessors are vacuously fixed. Forbidden. Advance: $G[1] := 3$ (unchanged); $fixed[1] := true$. $G = (3, 2, 4, 1)$.
2. Job 2: $pre = \{1\}$ and $fixed[1]$ is true. Forbidden. Advance: $G[2] := G[1] + t[2] = 5$; $fixed[2] := true$. $G = (3, 5, 4, 1)$.
3. Job 3: $pre = \{1\}$ and $fixed[1]$ is true. Forbidden. Advance: $G[3] := G[1] + t[3] = 7$; $fixed[3] := true$. $G = (3, 5, 7, 1)$.
4. Job 4: $pre = \{2, 3\}$ and both are fixed. Forbidden. Advance: $G[4] := \max(6, 8) = 8$; $fixed[4] := true$. $G = (3, 5, 7, 8)$. All jobs fixed. Done.

Each job is advanced exactly once; when job j is advanced, its predecessors are scanned to compute the maximum, costing $O(|pre(j)|)$ time. Summing over all jobs gives $O(n + m)$ total work (where $m = |\bigcup_j pre(j)|$).

The $fixed$ predicate ensures that a job is only processed after all its prerequisites are settled, eliminating repeated advances.

Fig. 2.2 illustrates the difference between the two formulations, projected onto the $(G[3], G[4])$ plane (since $G[1]$=3 and $G[2]$=5 are settled early in both traces). A state (g_3, g_4) is feasible when $g_3 \geq G[1] + t[3] = 7$ and $g_4 \geq \max(G[2] + t[4], g_3 + t[4]) = \max(6, g_3 + 1)$. The feasible states (shaded) are $(7, 8)$, $(7, 9)$, and $(8, 9)$; the least feasible state is $(7, 8)$.

2.4 A Sequential Implementation of the LLP Algorithm

The LLP algorithm can be implemented sequentially by maintaining a set S of forbidden indices. Algorithm LLP-Sequential gives LLP-SEQ, a general sequential implementation.

The algorithm maintains the invariant: **(I1)** S empty implies G is the optimal solution. The set S need not contain all forbidden indices at every step; it may also contain non-forbidden indices, which are simply skipped. The key implementation choices are:

- *Storage for S*: a linked list, queue, stack, or heap depending on the removal strategy. For the shortest path problem, a priority queue keyed by $G[j]$ is efficient.
- *Initialization*: insert all indices into S, or only those initially forbidden. For job scheduling with acyclic prerequisites, initialize with jobs having no predecessors.
- *Removal order*: simple FIFO, or priority-based (e.g., smallest $G[j]$ first for shortest path, ensuring indices are fixed once processed).
- *Updating S*: use a dependency list $dependList(j)$ listing indices whose forbidden status may change when $G[j]$ changes. For graph problems, these are typically the neighbors of j.

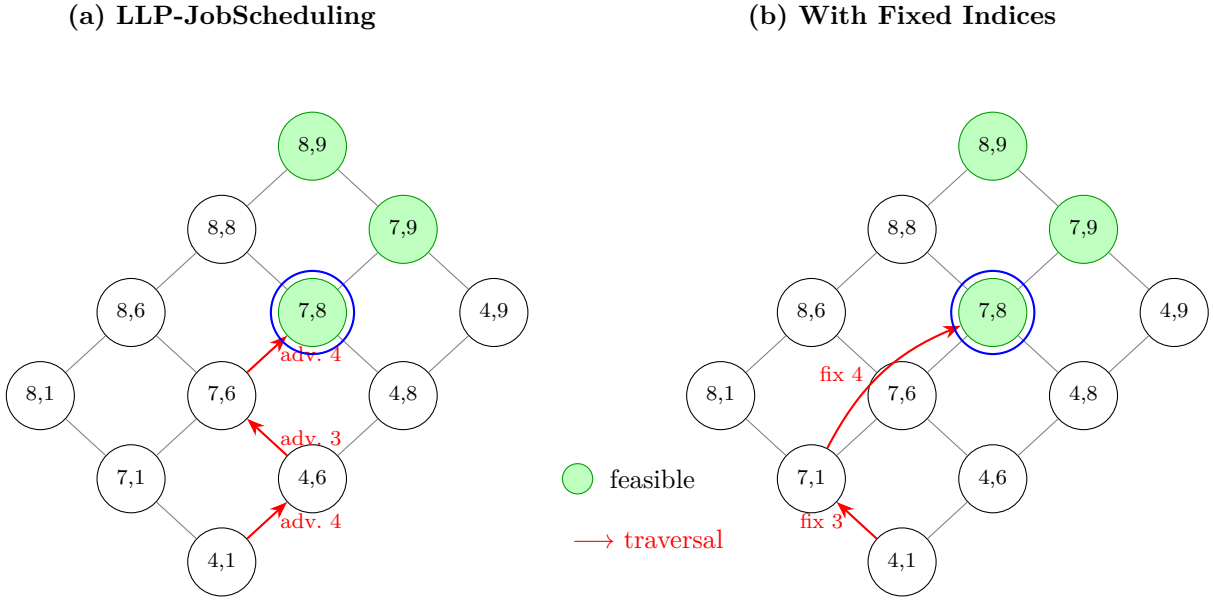


Figure 2.2: LLP traversal for the 4-job scheduling example, projected onto $(G[3], G[4])$ since $G[1] = 3$ and $G[2] = 5$ settle early. The feasible states $\{(7,8),(7,9),(8,9)\}$ are shaded; the least is $(7,8)$. (a) basic formulation (bad evaluation order): advance job 4 to $(4,6)$, then job 3 to $(7,6)$, then job 4 *again* to $(7,8)$. Job 4 is advanced twice. (b) with fixed indices: fix job 3 first (reaching $(7,1)$), then fix job 4 which advances directly to $(7,8)$. Each job advances once.

Algorithm LLP-Sequential: Sequential algorithm to find the minimum vector $\leq T$ that satisfies B

input: T: vector (top element), B: predicate
output: the minimum vector $G \leq T$ that satisfies B, or null

```
1  var G: vector initially ∀i : G[i] := 0; S: set of indices;
2  initialize(S, G) ;                    // add initial forbidden indices to S
3  while ¬S.empty() do
4  |  j := S.remove(G);
5  |  if forbidden(G, j, B) then
6  |  |  G[j] := α(G, j, B);
7  |  |  if G[j] > T[j] then return null;
8  |  |  S.updateForbidden(G, j);
9  |  end
10 end
11 return G
```

The while loop executes at most $O(mn)$ times, since each of n components advances at most m times (where m is the height of a single chain in the lattice). If each iteration costs $O(n)$ for remove, advance, and update, the total is $O(mn^2)$; for problems where these operations are $O(1)$, the total is $O(mn)$.

2.5 Properties of the LLP Algorithm

The LLP algorithm has many useful properties. We list them here so that the reader can apply them to various problems studied in this book. These properties are applicable to all the problems for which the LLP algorithm is used. We illustrate them with the job scheduling problem; in Chapter 3, we reinforce them in the context of the stable marriage problem.

1. **Nondeterminism in Evaluation of Forbidden Predicate**: Given a global state G, there may be multiple indices j for which $G[j]$ is forbidden. The LLP algorithm is correct irrespective of the order in which these indices are updated. The efficiency may differ depending upon the order, but the correctness is independent of the order. For example, in the job scheduling problem, the minimum completion time vector is the same regardless of the order in which forbidden jobs are advanced.

2. **Parallel Evaluation of Forbidden Predicate**: Suppose that G is shared among different threads, such that thread j is responsible for evaluating $forbidden(G, j)$. While this thread is evaluating the predicate, other threads may have advanced on other indices, i.e., thread j may have old information of $G[i]$ for $i \neq j$. However, this would still keep the algorithm correct. For example, in job scheduling, multiple forbidden jobs can evaluate their prerequisites and advance their completion times simultaneously, even with stale values of G.

3. **No Lookahead Required for Evaluation of Forbidden Predicate**: The LLP algorithm determines whether an index j is forbidden depending upon only the current global state G. This means that these algorithms are applicable in online settings where the future part of the lattice is revealed only when a forbidden index needs to advance. For job scheduling, the forbidden predicate for job j depends only on the current completion times of its prerequisites — not on jobs that may become forbidden later. For some problems, such as the shortest path problem, the weights on the edges are required to evaluate the forbidden predicate, so the graph must be known and static.

4. **The Optimal Feasible Global State is Optimal for Individual Components**: Suppose that in the job scheduling problem, we are interested in the minimum completion time for a single job j, ignoring all other jobs. The LLP algorithm returns a vector G_{llp} such that $G_{llp}[j]$ equals this individually optimal value. More generally, let G be the global state that is feasible and optimal with respect to index i, i.e., for all feasible H, $G[i] \leq H[i]$. Let G_{llp} be the global state computed by the LLP algorithm, then $G[i] = G_{llp}[i]$. This follows from the meet-closure property of feasible states. If $G[i] < G_{llp}[i]$, then the global state given by $G \sqcap G_{llp}$ is also feasible and strictly smaller than G_{llp}, contradicting that G_{llp} is the least global state that satisfies the feasibility predicate.

2.6 Additional Examples of LLP Algorithms

To further illustrate the generality of the LLP framework, we present an additional example: computing the greatest common divisor (GCD) of an array. Fig. 2.3 shows the LLP program for this problem.

(a) GCD of an array:

Algorithm LLP-GCD: Finding the GCD of an array via LLP

input: A: array$[0..n-1]$ of int
output: G: array$[0..n-1]$ of int, initially $A[i]$

1 **forbidden**(j): $\exists i \in [0..n-1] : G[j] > G[i]$
2 $\Rightarrow$**advance**: **if** $G[j] \bmod G[i] = 0$ **then** $G[j] := G[i]$ **else** $G[j] := G[j] \bmod G[i]$;

Figure 2.3: Finding the GCD of an array via LLP.

GCD. For computing the GCD of an array of positive integers A, the lattice is ordered by componentwise $\geq$ (decreasing order). Component j is forbidden if $G[j] > G[i]$ for some i. In this case, we replace $G[j]$ by $G[i]$ if $G[i]$ divides $G[j]$; otherwise, we replace $G[j]$ by $G[j] \bmod G[i]$. When the algorithm terminates, all components are equal to the GCD of A. The invariant is that every common divisor of A is also a common divisor of G. This holds initially because $G = A$. After each advance, if $G[i]$ divides $G[j]$, then every divisor of $G[i]$ also divides $G[j]$, so setting $G[j] := G[i]$ preserves the invariant. If $G[i]$ does not divide $G[j]$, the common divisors of $G[i]$ and $G[j]$ are exactly those of $G[i]$ and $G[j] \bmod G[i]$.

2.7 The *priority* Tag

The *forbidden* and *advance* clauses of an LLP program specify *which* indices can be advanced and *how* to advance them, but they deliberately leave the *order* in which forbidden indices are processed unspecified. Every fair schedule that keeps advancing forbidden indices reaches the same least fixed point, so correctness is independent of the order. For efficiency, however, the order often matters. The LLP framework provides an optional *priority* tag that lets the designer influence the schedule without changing the predicate.

An LLP program may carry a third clause alongside *forbidden* and *advance*:

priority(j): *expression*

where *expression* is a real-valued (or ordinal-valued) function of the current state G and the index j. The semantics is:

> *Whenever more than one index is forbidden, remove (and advance) the forbidden index j whose **priority**(j) is smallest first, breaking ties arbitrarily.*

Because the least-fixed-point solution is independent of the schedule, adding a *priority* clause never changes the final answer of a lattice-linear program; it can only change the running time and, in some cases, the number of advance steps. A typical implementation maintains a

priority queue keyed on **priority**(j) that contains all currently forbidden indices. After each advance, any indices whose forbidden status may have changed (either newly forbidden, or newly not forbidden) are inserted into or removed from the queue. The choice of data structure (binary heap, bucket queue, Fibonacci heap) depends on the range and update pattern of the priority.

We now give some examples where the tag *priority* will come in handy.

BFS and Dijkstra's algorithm. In the LLP formulation of BFS (Chapter 5), an index j is forbidden when some predecessor i has $G[j] > G[i] + 1$. Setting

$$\textbf{priority}(j) := G[j]$$

processes vertices in increasing order of their current distance. The first time a vertex j is dequeued, $G[j]$ already equals its final shortest-path distance — so j is advanced at most once. The schedule matches the classical BFS-with-queue algorithm. The same priority turns the LLP program for the single-source shortest path (Chapter 7) into Dijkstra's algorithm.

Many dynamic-programming recurrences have the form

$$G[j] = \min_{i \in pre(j)} (G[i] + w(i, j)),$$

on a DAG (e.g., shortest path in a DAG, longest increasing subsequence ending at each index, optimal binary search tree). Taking

$$\textbf{priority}(j) := \text{topological-order rank of } j$$

forces predecessors to be fixed before their successors, so every $G[j]$ is advanced exactly once — recovering the familiar "fill the DP table in topological order" schedule. Chapter 10 exploits exactly this pattern.

In the LLP program for layering (Chapter 5), j becomes forbidden when all its predecessors are fixed. Choosing

$$\textbf{priority}(j) := \max_{i \in pre(j)} G[i] + 1$$

processes vertices in order of their final layer number, giving a single $O(n + m)$ pass.

In every case, correctness is inherited from lattice-linearity; the *priority* clause only changes the order of evaluation and thus the running time. This separation of concerns — correctness from the predicate, efficiency from the schedule — is a recurring theme throughout the book.

2.8 Dual of Lattice-Linearity

We briefly discuss detection of predicates when the search starts from the top of the lattice in addition to the bottom of the lattice. Many predicates, such as $B_{stableMarriage}$ satisfy not only the lattice-linearity but also its dual. Thus, the set of elements satisfying these predicates is closed not only for the meet operation but also for the join operation. In addition, whenever the predicate is false, we can efficiently determine the forbidden indices. If we are okay with returning either of the elements as our final answer, then searching for any of the elements in parallel may speed up the algorithm by a factor of the height of the lattice. The dual of the lattice-linear property can be formally defined as follows.

Definition 2.9 (dual of Lattice-Linearity Property) *A Boolean predicate B has the dual of lattice-linearity property with the polytime algorithm $\mathcal{A}$ with respect to a finite distributive lattice L generated from a poset P if*

$$\forall G \in L : \neg B(G) \Rightarrow \exists i \in \mathcal{A}(G) : \mathit{dual\text{-}forbidden}(i, G, B)$$

where $\mathit{dual\text{-}forbidden}(i, G, B)$ is defined as

$$\forall H \leq G : (H[i] = G[i]) \Rightarrow \neg B(H).$$

The above definition states that whenever B is false in G, the algorithm $\mathcal{A}(G)$ returns an index such that any global state H less than G that matches G on that index also has B false.

We next define a predicate B as a regular predicate when it satisfies lattice-linearity as well as its dual.

Definition 2.10 (Regular Predicate) *A predicate B is regular in a lattice L iff B is lattice-linear and also dual lattice-linear.*

From the definition of regular predicates, it follows that the subset of elements that satisfy B is a sublattice of L. Since a sublattice of a distributive lattice is also distributive, we conclude that the subset of elements in L satisfying B is also distributive.

For a regular predicate, one can search for the satisfying element starting from both the bottom and the top of the lattice, as shown in Algorithm Regular.

Algorithm Regular: Algorithm for detecting predicates with efficient advancement property and its dual

```
   input: B: a regular predicate on lattice L
   output: an element of L satisfying B, or false
 1 var G: element of L initially ⊥ (the bottom element of L);
 2 Z: element of L initially ⊤ (the top element of L);
 3 while ¬B(G) and ¬B(Z) do
 4 |   forall i: forbidden(i, G) do
 5 |   |   if G cannot be advanced on i then return false;
 6 |   |   else advance G on i;
 7 |   end
 8 |   forall j: dual-forbidden(j, Z) do
 9 |   |   if Z cannot be retreated on j then return false;
10 |   |   else retreat Z on j;
11 |   end
12 end
13 if B(G) then return G;
14 else return Z ; // an optimal solution;
```

Applying the idea to the stable marriage problem, one can search for the stable marriage

starting from the top choices for all men (the $\perp$ element) of the lattice (in parallel) with the bottom choices for all men (the $\top$) element of the lattice. This algorithm will traverse the distance in a lattice given by the minimum of the distance of a stable marriage from the top or the bottom. Hence, depending upon the distance of man-optimal and man-pessimal stable marriages, it will return the stable marriage that is closer to the bottom or the top of the lattice.

In the above analysis, we ignore the 2 penalty factor that we incur because we run the algorithm from both the bottom and top of the lattice.

2.9 Summary

This chapter presented the Lattice-Linear Predicate (LLP) algorithm, a general method for finding the least element in a distributive lattice satisfying a given predicate. The LLP algorithm is nondeterministic and online: it does not inspect any element of the lattice above the current state. We showed that lattice-linear predicates are closed under conjunction, which means that constrained versions of problems are solved by the same algorithm with no additional machinery.

We also discussed the dual of lattice-linearity, which enables traversal from the top of the lattice downward. For predicates that are both lattice-linear and dual lattice-linear (called *regular* predicates), one can search from both ends simultaneously.

Table 2.2 lists the algorithms presented in this chapter.

Algorithm	**Problem/Topic**	**Time Complexity**
LLP	General LLP (least fixpoint)	$O(h \cdot n)$ iterations
LLP-JobScheduling-Fixed	Job Scheduling with fixed	$O(n + m)$
LLP-SEQ	General LLP (sequential)	problem-dependent
Regular	Regular predicate (bidirectional)	$O(h \cdot n)$ iterations

Table 2.2: Algorithms for lattice-linear predicate detection

Here n is the number of components, m is the number of precedence edges (for job scheduling), and h is the height of the lattice. The LLP and Regular algorithms give the number of iterations of the outer loop; the cost per iteration depends on the specific predicate and advance operation.

2.10 Problems

1. **(Acyclic Relation Predicate)** Let G be an n-dimensional vector of positive numbers. Let *pred* be a binary acyclic relation on $[n]$. Show that the predicate $B \equiv \forall (i,j) \in pred : G[j] \geq G[i] + 1$ is a lattice-linear predicate.

2. **(Order Ideal Predicate)** Let $(X, \leq)$ be a poset. A subset $Y \subseteq X$ is an order ideal if it satisfies

$$\forall u, v \in X : (v \in Y) \wedge (u \leq v) \Rightarrow (u \in Y)$$

Show that the predicate $B(Y) \equiv$ "Y is an order ideal of $(X, \leq)$" is lattice-linear in the boolean lattice of all subsets of X.

3. **(Disjunction Not Closed)** Show that lattice-linearity is not closed under disjunction.

4. **(Negation Not Closed)** Show that lattice-linearity is not closed under negation.

5. **(Precedence Job Scheduling Trace)** Consider the job scheduling instance with $n = 5$ jobs, processing times $t = [2, 3, 1, 4, 2]$, and precedence constraints $1 \to 3$, $1 \to 4$, $2 \to 4$, $3 \to 5$, $4 \to 5$. Trace the execution of Algorithm LLP-JobScheduling-Fixed. At each step, state which indices are forbidden, show the advance, and give the updated vector G. What is the final completion-time vector and the critical path?

6. **(Predicate Lattice Linearity Cases)** Determine whether each of the following predicates on vectors $G \in \mathbb{Z}^n_{\geq 0}$ is lattice-linear on the componentwise lattice. Prove your answer.

   (a) $B_1(G) \equiv \forall i < n :\ G[i] \leq G[i+1]$ (the vector is non-decreasing).

   (b) $B_2(G) \equiv G[1] + G[2] + \cdots + G[n] \geq k$ for a fixed constant k.

   (c) $B_3(G) \equiv \max_i G[i] - \min_i G[i] \leq d$ for a fixed constant d (bounded range).

7. **(LCM Lattice Linear)** Given an array $A[1..n]$ of positive integers, the *least common multiple* (LCM) of A is the smallest positive integer divisible by every $A[i]$. Formulate the LCM computation as an LLP problem: define the lattice, the state vector, the predicate B, the forbidden condition, and the advance operation. Prove that B is lattice-linear.

8. **(Release Time Constraint)** Consider the job scheduling problem with the additional constraint that each job i has a *release time* $r[i]$: job i cannot start before time $r[i]$, so its completion time must satisfy $G[i] \geq r[i] + t[i]$.

   (a) Show that the release-time predicate $B_r(G) \equiv \forall i :\ G[i] \geq r[i] + t[i]$ is lattice-linear.

   (b) Using Lemma 2.3(c), give the combined forbidden and advance operations for the constrained problem (precedence constraints $\wedge$ release-time constraints).

9. **(Permutation Not Lattice Linear)** Prove that the predicate $B(G) \equiv$ "G is a permutation of $(1, 2, \ldots, n)$" is *not* lattice-linear on the componentwise lattice of vectors in $\{1, \ldots, n\}^n$.

10. **(Median Stable Matching)** Let L be any finite distributive lattice of global states, and let B be any Boolean predicate defined on L that is both lattice-linear and dual lattice-linear (a *regular predicate*; see Section 2.8). Let $M = \{M_1, M_2, \ldots, M_k\}$ be any subset of L of size k all of whose elements satisfy B. For each index m, the multiset $\{M_i[m] : 1 \leq i \leq k\}$ has a sorted form; for any $j \in [1..k]$, define the generalized j-median state G^j by

$$G^j[m] \ :=\ \text{the } j\text{-th smallest value in the multiset } \{M_i[m] : 1 \leq i \leq k\}.$$

   (a) Show that G^j also satisfies B.

(b) Apply (a) to the stable marriage problem (Chapter 3) to conclude that the j-th median of any set of k stable matchings is itself a stable matching — a theorem of Teo and Sethuraman [TS98].

11. **(Incremental Job Insertion)** A least feasible completion vector G^* has been computed by LLP for the job-scheduling predicate B_{jobs} on jobs $1, \ldots, n$. A new job $n+1$ arrives with processing time t_{n+1} and a set of prerequisites $pre(n+1) \subseteq \{1, \ldots, n\}$. Give an algorithm that updates G^* to the new least feasible vector G^{**} on $n+1$ jobs without re-running LLP from $\bot$.

2.11 Bibliographic Remarks

The Lattice-Linear Predicate (LLP) algorithm is a general technique for designing algorithms for combinatorial optimization problems. The lattice-linearity property was introduced by Chase and Garg [CG98] in the context of detecting global predicates in distributed computations. In that setting, a distributed system of n processes generates a lattice of global states, and the goal is to find the least global state satisfying a given predicate. The key observation — that if the predicate is lattice-linear, a single pass through the lattice suffices — was later applied to algorithm design.

The application of LLP to the stable marriage problem is shown in Garg [Gar17], and its systematic application to algorithm design — including the shortest path problem, the assignment problem, and many others — is developed in Garg [Gar20b]. The LLP framework has since been extended to dynamic programming problems [GS24], to the housing allocation problem [Gar22], and to the minimum spanning tree problem [GG19]. Gupta and Kulkarni extend LLP algorithms for deriving self-stabilizing algorithms [GK23]. Garg [Gar23] shows that many generalizations of the stable matching problem, including the constrained stable marriage problem with ties, can also be solved using the LLP framework.

When the feasibility predicate has the form $\forall i : G[i] \geq f_i(G)$ for monotone functions f_i, the LLP algorithm is closely related to computing the least fixed point of a monotone function on a lattice via the Knaster–Tarski fixed-point theorem [Tar55, LNS82]. However, the LLP framework is more general in three respects: (1) the predicate B need not have the form $G \geq f(G)$ — we only require B to be closed under meets; (2) the Knaster–Tarski theorem requires the function to map the lattice to itself, guaranteeing a fixed point exists, whereas in LLP the algorithm may return null (e.g., cyclic job scheduling with positive weights has no solution); (3) the LLP framework supports advancement on multiple components, which the classical fixed-point iteration does not address.

The technique of *chaotic iteration* (also called *asynchronous iteration*) in abstract interpretation and dataflow analysis [CC77] also computes least fixed points by iterating on individual components in arbitrary order.

The LLP algorithm can be viewed as a generalization of chaotic iteration to predicates that are not necessarily of the fixed-point form.

The classical algorithms unified by the LLP framework include Dijkstra's algorithm [Dij59] for shortest paths, the Gale–Shapley algorithm [GS62] for stable marriage, and Kruskal's and Prim's algorithms for minimum spanning trees. For lattice theory background, see Appendix 21 and the texts by Birkhoff [Bir67], Davey and Priestley [DP90], and Garg [Gar15].

Chapter 3

The Stable Marriage Problem

3.1 Introduction

Matching markets are everywhere. The U.S. National Resident Matching Program (NRMP) places more than 40,000 medical-school graduates into hospital residency positions each year, using an algorithm that traces directly to the work of Gale and Shapley. Big-city public-school assignment systems — New York, Boston, Singapore — match hundreds of thousands of children to schools using essentially the same procedure. Kidney-exchange clearinghouses pair incompatible donor–recipient pairs into chains so that more transplants can take place. In each setting the underlying question is the same: given a set of agents on each side and a preference order from every agent, find a pairing in which no two unpaired agents would both prefer each other.

The Stable Matching Problem formalizes this question. Gale and Shapley introduced it in their 1962 paper [GS62] and showed two surprising results: a stable matching always exists, and the deferred-acceptance algorithm they introduced constructs one in $O(n^2)$ time. The economic significance of this line of work was recognized by the 2012 Nobel Prize in Economics awarded to Lloyd Shapley and Alvin Roth for the theory of stable allocations and the practice of market design.

Beyond its applications, stable matching is also the first concrete instance of the lattice-linear-predicate (LLP) framework introduced in Chapter 2. The set of *proposal vectors* forms a finite distributive lattice; the stability requirement is a lattice-linear predicate; and the Gale–Shapley algorithm is exactly the LLP main loop applied to that predicate. The chapter therefore reads as the first concrete LLP application and provides a template that later chapters — shortest paths, minimum spanning trees, network flow, and several others — will reuse.

In the standard problem there are n men numbered $1, 2, \ldots, n$ and n women numbered $1, 2, \ldots, n$, each with a totally ordered preference list. The matrices $mpref$ and $wpref$ encode the lists: $mpref[i][k] = j$ iff woman j is man i's k^{th} preference, and analogously for $wpref$. A matching has a *blocking pair* (m, w) if m and w are not married to each other yet each prefers the other to their current partner; the matching is *stable* if it has no blocking pair. Fig. 3.1 shows a small instance.

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

$mpref$

$\boldsymbol{m_1}$	w_2	w_3	w_1
$\boldsymbol{m_2}$	w_2	w_3	w_1
$\boldsymbol{m_3}$	w_3	w_1	w_2

$wpref$

$\boldsymbol{w_1}$	m_1	m_3	m_2
$\boldsymbol{w_2}$	m_2	m_1	m_3
$\boldsymbol{w_3}$	m_1	m_2	m_3

Figure 3.1: Stable matching problem with men's preference list ($mpref$) and women's preference list ($wpref$). The first column of each table (in bold) indexes by man / woman; the remaining cells give that person's ordered preference list, most preferred first.

This chapter is organized as follows. Section 3.2 introduces the proposal-vector lattice on which all algorithms in this chapter operate. Section 3.3 presents the Gale–Shapley algorithm with its $O(n^2)$ analysis. Section 3.4 generalizes Gale–Shapley to Algorithm α, which takes any starting proposal vector to the nearest stable proposal vector that dominates it; we extend this to allow $n_m \neq n_w$ men and women, assuming $n_w \geq n_m$. Section 3.5 gives the LLP formulation of stable matching together with the constrained version, in which additional lattice-linear constraints — regret bounds (e.g., Peter's regret must be less than Paul's), *forced pairs*, and *forbidden pairs* — can be imposed without changing the algorithm.

3.2 Proposal Vector Lattice

We use the notion of a *proposal vector* for our algorithms. A vector of (man) proposals, G, is of dimension n, the number of men. We view any vector G as follows: $(G[i] = k)$ if man i has proposed to his k^{th} preference, i.e., the woman given by $mpref[i][k]$. If $mpref[i][k]$ equals j, then $G[i] = k$ corresponds to man i proposing to woman j. The woman that man i proposes to in G is $mpref[i][G[i]]$. The vector $(1, 1, \ldots, 1)$ corresponds to the proposal vector in which every man has proposed his top choice. Similarly, $(n, n, \ldots, n)$ corresponds to the vector in which every man has proposed to his last choice. Our algorithms can also handle the case when the lists are incomplete, that is, a man prefers to stay alone to be matched to some women. However, for simplicity, we assume complete lists. It is clear that the set of all proposal vectors forms a distributive lattice under the natural less than order in which the meet and join are given by the component-wise minimum and the component-wise maximum, respectively. This lattice has n^n elements.

Given any proposal vector, G, there is a unique matching defined as follows: man i and $mpref[i][G[i]]$ are matched in G if the proposal of man i is the best proposal for that woman in G or any proposal before G. The man i is unmatched otherwise. A woman q is unmatched in G if she does not receive any proposal in G or earlier.

A proposal vector G represents a *man-saturating matching* iff no woman receives more than one proposal in G. Formally, G is a man-saturating matching if $\forall i, j : i \neq j : mpref[i][G[i]] \neq mpref[j][G[j]]$. When the number of men equals the number of women, a man-saturating matching is a perfect matching (all men and women are matched). When the number of men is less than the number of women, then G is a man-saturating matching if every man is matched (but some women are unmatched). We say that a matching M_1 (or a marriage) is less than another matching M_2 if the proposal vector for M_1 is less than that of M_2.

A proposal vector G may have one or more blocking pairs. A pair of man and woman (m, w) is a *blocking pair* in G iff $mpref[m][G[m]]$ is not w, man m prefers w to $mpref[m][G[m]]$, and

woman w prefers m to any proposal received in G. Observe that this definition works even when a woman w is unmatched, i.e. she has not received any proposals in G. In this case, woman w prefers m to staying alone, and m prefers w to $mpref[m][G[m]]$.

A proposal vector G is a stable marriage (or a stable proposal vector) iff it is a man-saturating matching and there are no blocking pairs in G. The man-optimal marriage is the *least* stable matching in the proposal lattice, and the woman-optimal marriage is the *greatest* stable matching.

3.3 Gale–Shapley Algorithm

In this section, we first present an algorithm due to Gale and Shapley for this problem (also known as the deferred acceptance algorithm). In this algorithm, a free man proposes to women in order of his preference list, i.e., he first proposes to his top choice. Only when a man is rejected by his top choice would he move to his next top choice. We maintain the list of all men who are not engaged in the variable $mList$. Initially, all men are free (not engaged) and are on this list. For this section, recall that we assume that the number of men is equal to the number of women.

When any woman z receives a proposal from a man i, she always accepts it if she is not engaged. In this case, they both get engaged, and the variable $partner[z]$ is set to i. If the woman z is engaged, then she compares this proposal with her existing partner. If she prefers this proposal to her existing partner, then she breaks the engagement with her existing partner and makes i her partner. The previous partner joins $mList$, the list of free men. If the woman z prefers her existing partner, then the proposal of the man i is rejected, and the man i goes back to $mList$. Algorithm Gale–Shapley shows the pseudocode for these steps.

Algorithm Gale–Shapley: Finding the man-optimal marriage

input: *mpref*: array[1..n][1..n] of int; *rank*: array[1..n][1..n] of int;
output: G: man-optimal stable marriage

```
1  var mList: list of 1..n // list of men that are free, initially includes all
       men
2  partner: array[1..n] of 0..n initially partner[i] = 0 for all i // Current fiance for
       woman i
3  G: array[1..n] of 0..n initially G[i] = 0 for all i // Number of proposals made by
       man i
4  mList := [1..n] // Initialize all men as part of this list
5  while mList is nonempty do
6  |  i := mList.removeFirst();
7  |  G[i] := G[i] + 1 // Move to the next available top choice for man i
8  |  z := mpref[i][G[i]] // Woman corresponding to that choice
9  |  if partner[z] = 0 then partner[z] := i;
10 |  else if rank[z][i] < rank[z][partner[z]] then
11 |  |  mList.add(partner[z]);
12 |  |  partner[z] := i;
13 |  end
14 |  else  mList.add(i) ;
15 end
16 return G
```

It is easy to verify the following properties of the algorithm.

Lemma 3.1 *All the following statements are valid.*

1. *As the algorithm progresses, the partner for a man can only worsen and the partner for a woman can only improve.*
2. *Once a woman is engaged, she stays engaged.*

Observe that once all men are engaged, the list *mList* is empty and the *while* loop in the algorithm terminates. Is it possible for all men to run out of all the choices and still not have a stable marriage? We claim that whenever a man i proposes to his last choice, the proposal is always accepted. The proposal is always accepted if the woman is not engaged. If the woman is engaged, we can deduce that all women are engaged because the man had previously proposed to all other women who are now engaged. However, this is a contradiction because all n women are engaged to different men and the man i is not engaged. Hence, we conclude that the last proposal of any man is always accepted.

How many times can the *while* loop iterate? Observe that in every iteration $G[i]$ increases by 1 for some i. Since $G[i]$ starts with 0 and cannot increase beyond n, the total number of iterations for the *while* loop is $O(n^2)$. Each iteration of the *while* loop takes $O(1)$ time. Hence, the time complexity of the Gale–Shapley Algorithm is $O(n^2)$.

We now show that the GS algorithm returns the man-optimal stable marriage. Suppose

that a man m is rejected by his best valid partner woman w in the GS algorithm. Let this be the first time a valid pair is rejected. Suppose w rejected m for some man m'. Thus, we know that w prefers m' to m.

Let M be a marriage in which m is married to w because (m, w) is a valid pair. In the marriage, M, let m' be married to w'. When w rejected m for m' in the GS marriage, it was the first rejection of any valid pair. This means that m' must have proposed to w before proposing to w'. Thus, we also get that m' prefers w to w'. However, this results in (m', w) forming a blocking pair for M. The man m' prefers w to w' and the woman w prefers m' to m, a contradiction. Hence, we conclude that the GS algorithm returns the man-optimal stable marriage.

Example 3.2 We trace the Gale–Shapley algorithm on the instance in Fig. 3.1. Initially, $G = (0, 0, 0)$, $partner = (0, 0, 0)$, and $mList = [m_1, m_2, m_3]$.

1. m_1 proposes to $mpref[1][1] = w_2$. $G = (1, 0, 0)$. w_2 is free; $partner[w_2] := m_1$.
2. m_2 proposes to $mpref[2][1] = w_2$. $G = (1, 1, 0)$. w_2 prefers m_2 to m_1 ($rank[w_2][m_2] = 1 < rank[w_2][m_1] = 2$). $partner[w_2] := m_2$; m_1 returns to $mList$.
3. m_3 proposes to $mpref[3][1] = w_3$. $G = (1, 1, 1)$. w_3 is free; $partner[w_3] := m_3$.
4. m_1 proposes to $mpref[1][2] = w_3$. $G = (2, 1, 1)$. w_3 prefers m_1 to m_3 ($rank[w_3][m_1] = 1 < rank[w_3][m_3] = 3$). $partner[w_3] := m_1$; m_3 returns to $mList$.
5. m_3 proposes to $mpref[3][2] = w_1$. $G = (2, 1, 2)$. w_1 is free; $partner[w_1] := m_3$.

$mList$ is empty. The algorithm terminates with $G = (2, 1, 2)$: $m_1 \leftrightarrow w_3$, $m_2 \leftrightarrow w_2$, $m_3 \leftrightarrow w_1$. This is the man-optimal stable marriage.

3.4 Algorithm α: Upward Traversal

The algorithm α generalizes the Gale–Shapley algorithm in two fundamental ways.

- *Arbitrary Initial Proposal Vector*: Gale–Shapley algorithm always finds the man-optimal stable marriage. Suppose that we are interested in finding a stable marriage such that the proposal vector is at least I. For example, if $I = (3, 1, 2, 1)$, then the first man cannot propose to his two top choices, and the third man cannot propose to his top choice. The algorithm α works even when the initial proposal vector is arbitrary instead of the top choice for each man. Observe that the standard Gale–Shapley algorithm does not work as is when the starting proposal vector is arbitrary. The simple Gale–Shapley algorithm would require men to make proposals and women to accept the best proposals they have received so far. If the starting proposal vector is a perfect matching but not stable, then each woman gets a unique proposal. All women would accept the only proposal received, but the resulting marriage may not be stable.

- *Unequal number of men and women*: We consider the case where the number of women exceeds the number of men. If the number of men is greater than the number of women, then we can simply switch the roles in our algorithm.

If we started with the top choices of all men, then the Gale–Shapley algorithm would still return a man-optimal stable matching with the excess women unmatched. However, if we start from an arbitrary proposal vector, we can end up with all women getting unique proposals, but there may exist an unmatched woman who is preferred by some man over his current match. To address this problem, we first do a simple check on the initial proposal vector given by the following lemma. Let $numw(I)$ denote the total number of unique women that have been proposed in all vectors that are less than or equal to I.

Lemma 3.3 *Given any stable marriage instance with n_m men and the initial proposal vector I there is no stable marriage for any proposal vector $G \geq I$ whenever $numw(I)$ exceeds n_m.*

Proof: Consider any proposal vector $G \geq I$. Since the total number of men is n_m, there is at least one woman q who has been proposed in the past of G who does not have any proposal in G. Suppose that the proposal was made by man p. Then, man p prefers q to $mpref[p][G[p]]$ and q prefers p to staying alone.

■

Hence, in our algorithm, we only consider I such that the total number of women proposed up to I is at most n_m. Even when the number of men equals the number of women, the proposal vector may be a perfect matching but not stable. To address this problem, we first define $forbidden(G, i)$ as the predicate that there exists another man j such that (1) both i and j have proposed to the same woman in G and that woman prefers j, or (2) $(j, mpref[i][G[i]])$ is a blocking pair in G. We first show that

Lemma 3.4 *Let G be any proposal vector such that $numw(G) \leq n_m$. There exists a man i such that $forbidden(G, i)$ iff G is not a stable marriage.*

Proof: First, suppose that there exists i such that $forbidden(G, i)$. This means that there exists a man j such that j has proposed to the same woman and that woman prefers j or $(j, mpref[i][G[i]])$ is a blocking pair in G. If both i and j have proposed to the same woman in G, then it is clearly not a matching. If $(j, mpref[i][G[i]])$ is a blocking pair, then G is not stable.

Conversely, assume that G is not a stable marriage. This means that either G is not a man-saturating matching or there is a blocking pair for G. If G is not a man-saturating matching, then there must be some woman who has been proposed by multiple men. Any man i who is not the most favored in the set of proposals satisfies $forbidden(G, i)$. If G is a man-saturating matching but not a stable marriage, then there must be a blocking pair (p, q). If q has been proposed in G by man i, then $(p, mpref[i][G[i]])$ is a blocking pair. Hence, $forbidden(G, i)$ holds. If q has not been proposed in G then we know at least $n_m + 1$ women who belong to $numw(G)$ which violates our requirement on G.

■

Algorithm UpwardTraversal exploits the $forbidden(G, i)$ function to find a stable marriage in the proposal lattice. The basic idea is that if a man i is forbidden in the current proposal vector G, then he must go down his preference list until he finds a woman who is unmatched or prefers him to her current match. The *while* loop iterates until none of the men is forbidden in G. If the while loop terminates, then G is a stable marriage due to Lemma 3.4. The man i advances on his preference list until his proposal is the most preferred proposal to the woman among all proposals that are made to her in any proposal vector less than or equal to G. If there is no such proposal, then there does not exist any $G \geq I$ such that G is stable and the algorithm returns null. Otherwise, the man makes that proposal.

Algorithm UpwardTraversal: Algorithm that returns the smallest stable marriage greater than or equal to the given proposal vector I

input: a stable marriage instance, initial proposal vector I
output: smallest stable marriage greater than or equal to I (if one exists)

1 $forbidden(G, i)$ holds if there exists another man j such that either (1) both i and j have proposed to the same woman in G and that woman prefers j, or (2) $(j, mpref[i][G[i]])$ is a blocking pair for G.;
2 **if** $numw(I) > n_m$ **then return** null // no stable matching exists ;
3 **else** $G := I$;
4 **while** *there exists a man i such that* $forbidden(G, i)$ **do**
5 find the next woman q in the list of man i s.t. i has the most preferred proposal to q until G;
6 **if** *no such choice after* $G[i]$ *or the number of women proposed including* q *exceeds* n_m **then return** null // no stable matching exists ;
7 **else** $G[i]$:= choice that corresponds to woman q;
8 **end**
9 **return** G

There are two main differences from the Gale–Shapley algorithm. The first difference is the simple check on the number of women who have been proposed up to G. We require $numw(G) \leq n_m$. Clearly, if the number of women is equal to the number of men, then $numw(G)$ cannot exceed n_m and this check can be dropped. Also, even if the number of women exceeds the number of men, but the initial proposal vector is $(1, 1, \ldots, 1)$, the number of women proposed cannot exceed n_m. However, this check is required when the number of women exceeds n_m and the initial proposal vector is arbitrary.

The second difference is in the definition of $forbidden(G, i)$. In the standard Gale–Shapley algorithm, a man advances on his preference list only when the woman he has proposed to is either matched with someone more preferable or receives a proposal from a more preferable man. Whenever the Gale–Shapley algorithm reaches a perfect matching, it is a stable matching. For any arbitrary I (for example, a perfect matching that is not stable), it is important to take blocking pairs into consideration as part of the forbidden predicate.

The time complexity of Algorithm α is $O(n_m^2)$: each man advances at most n_m times on his preference list, and each advancement takes $O(1)$ time using appropriate data structures.

When $n_m = n_w = n$, this reduces to the standard $O(n^2)$ complexity of the Gale–Shapley algorithm.

3.5 LLP Stable Matching Algorithm

We now derive the algorithm for the stable matching problem using Lattice-Linear Predicates. We assume that the number of men equals the number of women in this section. We let $G[i]$ be the choice number that man i has proposed to. Initially, $G[i]$ is 1 for all men. The woman that man i proposes to in G is $mpref[i][G[i]]$.

Definition 3.5 *An assignment G is feasible for the stable marriage problem if (1) it corresponds to a perfect matching (all men are paired with different women) and (2) it has no blocking pairs.*

We show that the predicate "G is a stable marriage" is a lattice-linear predicate.

Lemma 3.6 *The predicate that a vector G corresponds to a stable marriage is lattice-linear.*

Proof: Let $z = mpref[j][G[j]]$. Recall from Lemma 3.4 that G is not a stable marriage iff $\exists j : forbidden(G, j)$, where $forbidden(G, j)$ is expressed formally as $(\exists i : \exists k \leq G[i] : (z = mpref[i][k]) \wedge (rank[z][i] < rank[z][j]))$.

We show that if $forbidden(G, j)$ holds, then for any $H \geq G$ with $H[j] = G[j]$, H is also not a stable marriage. Since $H[j] = G[j]$, $mpref[j][H[j]] = z$. Since $H[i] \geq G[i]$, the same k that satisfies $k \leq G[i]$ also satisfies $k \leq H[i]$, so the forbidden condition carries over to H. Hence $forbidden(H, j)$ holds, so H is not a stable marriage.

■

Lemma 3.6 immediately gives us the Algorithm LLP-StableMarriage. To keep the forbidden clause readable we introduce an auxiliary variable $z = mpref[j][G[j]]$ — the woman that thread j is currently proposing to — using the **always** section. The **always** section defines variables that are derived from G, viewed as macros: whenever $G[j]$ changes, so does z.

If man j is forbidden, it is clear that any vector in which man j is matched with z and man i is matched with his current or a worse choice can never be a stable marriage. Thus, it is safe for man j to advance to the next choice.

Algorithm LLP-StableMarriage: Man-optimal stable marriage via LLP

input: $mpref$: array[1..n][1..n] of int; $rank$: array[1..n][1..n] of int; I: array[1..n] of int
output: G: array[1..n] of int, initially $I[i]$

1 **always**(j): $z = mpref[j][G[j]]$;
2 **forbidden**(j): $\exists i \in [1..n] : \exists k \in [1..G[i]] : (z = mpref[i][k]) \wedge (rank[z][i] < rank[z][j])$
3 $\Rightarrow$ **advance**: $G[j] := G[j] + 1$;

The general properties of the LLP algorithm (Section 2.5) are well illustrated by the stable marriage problem:

- *Nondeterminism*: The man-optimal stable marriage is the same regardless of the order in which forbidden men advance on their preference lists.

- *No lookahead*: A man need not reveal his full preference list in advance. Only when he is rejected (forbidden) does he reveal his next choice. This makes the algorithm applicable in online settings.

- *Individual optimality*: The LLP solution is optimal for each man individually. If man m seeks the stable marriage that gives him his best possible partner (among all stable marriages), the LLP algorithm returns exactly that partner for m.

Algorithm LLP-StableMarriage works for any initialization I of G vector. If we know that G is initialized to $[1, 1, \ldots, 1]$, then we can simplify the forbidden condition to

$$(\exists i : (z = mpref[i][G[i]]) \wedge (rank[z][i] < rank[z][j])).$$

Observe that if we assume sequential implementation and if we implement a variable $partner[z]$ for any woman z, we can further simplify the condition to

$$(partner[z] \neq null) \wedge (rank[z][partner[z]] < rank[z][j]).$$

This is precisely the condition used in the Gale–Shapley algorithm which is sequential and starts with the initial vector $[1, 1, \ldots, 1]$.

We now present an algorithm to find stable marriages that satisfy additional constraints. The following lemma proves lattice-linearity of many such constraints.

Lemma 3.7 *The following constraints are lattice-linear.*

1. *The regret of man i is at most that of man j.*
2. *Man i cannot be married to woman j.*
3. *The regret of man i is equal to that of man j.*

Proof: Let B be the predicate that G is a stable marriage and it satisfies the corresponding additional constraint.

1. Suppose that G is a stable marriage but it does not satisfy B. This means that the regret of man i is more than the regret of man j. In this case, we have $forbidden(G, j)$, because unless $G[j]$ is advanced, the predicate cannot become true.

2. If $mpref[i][G[i]] = j$, then $forbidden(G, i)$ holds.

3. This condition is a conjunction of two lattice-linear conditions of type in part (1).

■

Observe that we have not given the argument that the LLP predicate eventually becomes true. In general, this may not happen for all LLP predicates. For the standard stable marriage, however, the LLP predicate becomes true. We can define the predicate $D(k)$ on the set of assignments as "By the assignment G, at least k different women have been proposed." It is clear that before all the proposals are done $D(n)$ holds. It is easy to show that $D(k)$ implies that there are k valid pairs in the assignment. Therefore, $D(n)$ implies that B holds on G.

We now enumerate advantages of the LLP algorithm.

1. It gives us a more general algorithm. Instead of starting from the initial vector, we can start from any vector and find a stable marriage greater than or equal to that vector, if one exists.

2. We can use the LLP algorithm for finding a stable marriage that satisfies additional constraints such as the regret of man i is at most that of man j.

3. We can automatically deduce that the meet of two stable marriages is also a stable marriage.

Example 3.8 As an example of a constrained stable marriage, consider the instance in Fig. 3.1 with the additional constraint that the regret of m_1 must be at most the regret of m_2 (i.e., $G[1] \leq G[2]$). This constraint is lattice-linear: $forbidden(2) \equiv (G[1] > G[2])$ with $advance(2) : G[2] := G[1]$. By Lemma 2.3(c), the conjunction of the stability predicate and this constraint is also lattice-linear, so the LLP algorithm finds the least stable marriage satisfying the constraint. In the unconstrained man-optimal marriage $G = (2, 1, 2)$, m_1 has regret 2 while m_2 has regret 1, violating the constraint. Since $(2, 1, 2)$ is in fact the only stable marriage of this instance, no stable marriage satisfies $G[1] \leq G[2]$; the LLP algorithm with the combined predicate advances past the top of the lattice and reports that the constrained problem is infeasible (returns null).

3.6 Extending Algorithms

The Gale–Shapley algorithm and its LLP variants presented in this chapter all assume a clean, static, centralized setting: a fixed set of n men and n women, complete and truthful preference lists, and a single process that has access to all of them. Real applications rarely come packaged so neatly. Readers are encouraged to study how the algorithms of this chapter adapt, generalize, or fail under the following conditions:

1. **Incremental setting:** Men and/or women may join or leave the system dynamically after an initial matching has been computed. Does the existing matching need to be recomputed from scratch, or can it be repaired locally while preserving stability?

2. **Dynamic setting:** The preference lists change over time (for example, a participant revises their ranking after learning more about the candidates). How should the algorithm track a moving stable matching?

3. **Distributed setting:** Each participant has access only to their own preference list and can communicate only with a subset of the other side. Design a stable matching algorithm in which no single node sees the entire instance.

4. **Privacy setting:** Each participant is willing to reveal only partial information about their preferences (e.g., their top k choices, or a coarse ranking). Can a stable matching still be computed, possibly with a degradation in the quality of the result?

5. **Constrained setting:** The matching must satisfy additional side constraints—for example, forced pairs that must appear in the matching, forbidden pairs that must not, regional or diversity caps, or couples who must be matched to nearby partners. Some such constraint families are already treated in the book; others make for good exercises.

6. **Ties and indifference:** Participants are allowed ties in their preference lists. Stable matchings may no longer be unique, and several notions of stability (weakly, strongly, super-) diverge. Which of them can be computed with LLP-style algorithms?

7. **Parallel setting:** The algorithm is parallelized to reduce the wall-clock time of computing a stable matching. How large can the parallel speedup be in the worst case, and how does it compare with the critical-path analysis of Algorithms α and β?

8. **Approximation setting:** When no stable matching exists (e.g., in the stable roommate problem with odd-length preference cycles), the goal is to find a near-stable solution that minimizes the number or severity of blocking pairs.

9. **Fairness setting:** Gale–Shapley is known to be men-optimal and women-pessimal (or vice versa). The objective is to produce a stable matching that is fair across the two sides, for example by minimizing the sum or the maximum of regrets.

10. **Enumeration setting:** The task is to enumerate all stable matchings of a given instance, or the top-k stable matchings according to a given objective. The lattice of stable matchings makes this a natural LLP target.

11. **Online setting:** Participants arrive one at a time in an adversarial or random order, and the algorithm must make irrevocable decisions as each participant is revealed. How close to an offline stable matching can one stay?

12. **Byzantine setting:** Some participants behave strategically or maliciously—misreporting preferences, refusing to execute the protocol, or colluding to disrupt the matching for truthful participants. Which of the algorithms in this chapter are robust to such behavior, and what defenses are possible?

3.7 Summary

This chapter studied the stable marriage problem through the lens of lattice-linear predicate detection. We modeled the problem using the proposal vector lattice, where each vector G assigns a choice number to every man. The key insight is that the predicate "G is a

stable marriage" is lattice-linear: if man j is forbidden in G (another man is preferred by $mpref[j][G[j]]$), then j remains forbidden in every $H \geq G$ with $H[j] = G[j]$.

We presented the Gale–Shapley algorithm as a special case of the LLP framework. Algorithm α generalizes Gale–Shapley to find the man-optimal stable marriage from any initial proposal vector I, and LLP-StableMarriage expresses the same algorithm in the LLP forbidden/advance notation.

Table 3.1 lists the algorithms discussed in this chapter.

Algorithm	**Problem**	**Time Complexity**
Gale–Shapley	Stable Marriage	$O(n^2)$
α (Upward Traversal)	Stable Marriage $\geq I$	$O(n^2)$
LLP-StableMarriage	Stable Marriage	$O(n^2)$

Table 3.1: Algorithms for the stable marriage problem

3.8 Problems

1. **(GS Invariant And No Blocking Pair)** Show that Gale–Shapley algorithm maintains the invariant that every woman is partnered with the most preferred man of all the proposals made so far to her. Use this claim to show that there is no blocking pair for all the engaged couples during the execution of Gale–Shapley algorithm.

2. **(Woman Pessimal Stable Marriage)** Show that the man-oriented Gale–Shapley algorithm returns the stable marriage, which is the worst stable marriage from the perspective of women.

3. **(Stable Marriage Sublattice Closure)** Show that the set of stable marriages is closed under the meet operation and the join operation of the proposal lattice.

4. **(Check Stable Marriage Assignment)** Given an assignment vector, how will you check whether it forms a stable marriage?

5. **(Trace Gale Shapley n4)** Consider $n = 4$ with the following preferences. Men: m_1: w_1, w_2, w_3, w_4; m_2: w_2, w_1, w_4, w_3; m_3: w_1, w_4, w_3, w_2; m_4: w_4, w_1, w_2, w_3. Women: w_1: m_2, m_1, m_4, m_3; w_2: m_4, m_3, m_1, m_2; w_3: m_1, m_3, m_2, m_4; w_4: m_3, m_4, m_2, m_1. Trace the man-oriented Gale–Shapley algorithm. Show the proposal vector G after each round and give the final stable matching.

6. **(Find Blocking Pairs n3)** Consider $n = 3$ with men's preferences: m_1: w_1, w_2, w_3; m_2: w_2, w_1, w_3; m_3: w_1, w_3, w_2. Women's preferences: w_1: m_2, m_1, m_3; w_2: m_1, m_3, m_2; w_3: m_3, m_2, m_1. The matching $M = \{(m_1, w_2), (m_2, w_3), (m_3, w_1)\}$ is given. Find all blocking pairs, or prove that M is stable.

7. **(Constrained Stable Matching)** Using the instance from the Gale–Shapley trace problem above (the $n = 4$ instance), find the man-optimal stable matching subject to the constraint that m_1 must be matched to either w_1 or w_2 (i.e., $G[1] \leq 2$). Does a constrained stable matching exist?

8. **(Stable Roommate Nonexistence)** The *stable roommate problem* is the non-bipartite version of stable matching: given $2n$ people (no men/women distinction), each with a preference ranking over all others, find a matching into n pairs with no blocking pair. Show that stable matchings need not exist in the roommate setting by constructing a 4-person instance with no stable matching. Explain why the bipartite structure of the stable marriage problem guarantees existence.

9. **(Opposing Preferences Across Matchings)** Let S_1 and S_2 be two distinct stable matchings for the same instance. A man m is said to *prefer* S_1 if he prefers his partner in S_1 to his partner in S_2, and similarly for women. Prove: if man m prefers S_1, then his partner in S_1 prefers S_2.

10. **(Quadratic Proposals Worst Case)** Construct an instance with n men and n women in which the man-oriented Gale–Shapley algorithm makes $\Omega(n^2)$ proposals.

11. **(Quadratic Upward Traversal Implementation)** Give an $O(n^2)$ implementation of Algorithm α (Upward Traversal). Your implementation should use:

    - An array $curBest[1..n]$ where $curBest[q]$ is the most preferred man among all men who have proposed to woman q in any proposal vector less than or equal to G.
    - A list $mList$ of men i such that $forbidden(G, i)$ holds.

    Show that $forbidden(G, i)$ holds iff $curBest[mpref[i][G[i]]] \neq i$. Use this to argue that the while loop terminates with a stable marriage or correctly reports that none exists.

3.9 Bibliographic Remarks

The stable matching problem was introduced by Gale and Shapley [GS62] and has been studied extensively, with several books and survey articles devoted to the topic [GI89, Knu97c, RS92, IM08, Man13]. The notion of *regret* in a matching is discussed in [GI89]. Stable marriage with restricted pairs (forced and forbidden pairs) is studied by Dias et al. [DdFdFS03]. The lattice theory background on order ideals and distributive lattices used in this chapter can be found in Davey and Priestley [DP90]. The existence of a stable matching via Tarski's fixed-point theorem is shown by Adachi [Ada00].

The lattice-linear predicate (LLP) approach to stable marriage presented in this chapter was introduced by Garg [Gar17]. The application of predicate detection to combinatorial optimization problems, including stable matching, is discussed in [Gar20b]. Hu and Garg [HG21] give a characterization of super-stable matchings. Garg and Hu [GH20] present improved paths to stability for the stable marriage problem. Extensions to the constrained stable marriage problem with ties are discussed by Garg [Gar23]. A generalization of Teo and Sethuraman's median stable marriage theorem is given in [Gar20a].

Chapter 4

Sorting Algorithms

4.1 Introduction

In this chapter, we consider some basic ideas in the design of algorithms that are crucially based on the notion of *order* in a set of elements of size n. We will assume that the order is total, that is, for every two elements x and y, either x is less than or equal to y, or y is less than or equal to x in that order.

Sorting is among the most fundamental problems in computer science and one of the most studied. A large fraction of the running time of real systems is spent arranging data into some order, and many algorithms that appear unrelated to sorting reduce to it as a subroutine. Once the data is sorted, many tasks become dramatically cheaper: searching for a value takes $O(\log n)$ time by binary search instead of $O(n)$; computing the median, quantiles, or any order statistic becomes $O(1)$ after an $O(n \log n)$ preprocessing; and detecting duplicates or taking the union, intersection, or difference of two sets takes linear time. Sorting also enables efficient algorithms for geometric problems such as finding the closest pair of points and constructing convex hulls, and it is a building block for data compression, indexing, and scheduling.

Before we discuss sorting algorithms, let us elaborate on the binary search mentioned earlier. Consider the problem of searching for an element in a sorted array. Suppose that we are given a sorted array A of size n and an element x. We are interested in finding out if there exists i such that $A[i]$ equals x. On a single processor, we can accomplish this using binary search in $O(\log n)$ time. The idea is to compare x with the middle element of the array. If the middle element is equal to x, we are done; otherwise, the range of indices of A that can possibly have x is divided by a factor of 2.

This chapter is organized as follows. Section 4.2 discusses sorting algorithms based on the swapping of consecutive entries that are out of order. This is a general class of algorithms that are very natural. However, these algorithms take $O(n^2)$ sequential time in the worst case. Section 4.3 discusses *QuickSort*. Section 4.4 discusses *MergeSort*. Since MergeSort and QuickSort are based on the divide-and-conquer paradigm, they are revisited in Chapter 9. All these sorting algorithms are based on the comparison of two entries in the array. Finally, Section 4.6 describes sorting algorithms that are not based on the comparison of two entries.

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

4.2 Algorithms Based on Swapping Consecutive Entries

Let A be an array of distinct integers. Our goal is to sort the array.

Algorithm BubbleSort: Bubble Sort

input: $A[1..n]$: array of int

```
1 repeat
2 |  found := false;
3 |  for j = 1 to n − 1 do
4 |  |  if A[j] > A[j + 1] then
5 |  |  |  found := true;
6 |  |  |  swap(A[j], A[j + 1]);
7 |  |  end
8 |  end
9 until ¬found;
```

Algorithm BubbleSort swaps entries that are out of order in one iteration of the *for* loop.

Example 4.1 Consider sorting the array $A = [5, 3, 8, 1]$ using Bubble Sort.

Pass 1 ($j = 1, 2, 3$): Compare $(5, 3)$, swap $\rightarrow [3, 5, 8, 1]$. Compare $(5, 8)$, no swap. Compare $(8, 1)$, swap $\rightarrow [3, 5, 1, 8]$. The largest element 8 has "bubbled" to the end.

Pass 2: Compare $(3, 5)$, no swap. Compare $(5, 1)$, swap $\rightarrow [3, 1, 5, 8]$. Compare $(5, 8)$, no swap.

Pass 3: Compare $(3, 1)$, swap $\rightarrow [1, 3, 5, 8]$. No more swaps needed; $found$ remains false in the next pass, and the algorithm terminates.

Each pass moves at least one element to its correct position, so at most $n - 1$ passes do useful work; one additional pass with no swaps is needed to detect termination. Hence, the repeat loop is executed at most n times, giving a sequential time complexity of $O(n^2)$.

Another schedule is to increase the size of the sorted array one at a time.

Algorithm InsertionSort: Insertion Sort

input: $A[1..n]$: array of int

```
1 for i = 2 to n do
2 |  for j = i − 1 downto 1 do
3 |  |  if A[j] ≤ A[j + 1] then break;
4 |  |  swap(A[j], A[j + 1]);
5 |  end
6 end
```

The time complexity is clearly $O(n^2)$ due to nested *for* loops.

Example 4.2 Consider sorting $A = [5, 3, 8, 1]$ using Insertion Sort.
$i = 2$: Insert $A[2] = 3$ into the sorted subarray $[5]$. Compare $(5, 3)$, swap $\rightarrow [3, 5, 8, 1]$.
$i = 3$: Insert $A[3] = 8$ into $[3, 5]$. Compare $(5, 8)$, no swap (break). Array: $[3, 5, 8, 1]$.
$i = 4$: Insert $A[4] = 1$ into $[3, 5, 8]$. Compare $(8, 1)$, swap $\rightarrow [3, 5, 1, 8]$. Compare $(5, 1)$, swap $\rightarrow [3, 1, 5, 8]$. Compare $(3, 1)$, swap $\rightarrow [1, 3, 5, 8]$. Sorted.

We now prove that any comparison-based sorting algorithm that restricts itself to comparing consecutive entries (adjacent elements) in an array of size n requires at least $\Omega(n^2)$ comparisons in the worst case. This applies to algorithms such as Bubble Sort or Insertion Sort, where comparisons are limited to pairs of elements at positions i and $i + 1$.

Theorem 4.3 *Consider an array $A = [a_1, a_2, \ldots, a_n]$ of n distinct elements. A sorting algorithm that uses only consecutive comparisons requires at least $\Omega(n^2)$ comparisons in the worst case.*

Proof:
The maximum number of inversions in a permutation of n elements is achieved by the reverse order $[n, n-1, \ldots, 2, 1]$, with:

$$\text{number of inversions} = \binom{n}{2} = \frac{n(n-1)}{2}.$$

Each adjacent comparison can resolve at most one inversion (e.g., swapping a_i and a_{i+1}), so a permutation with $\frac{n(n-1)}{2}$ inversions requires at least that many comparisons to sort fully.

■

4.3 Quicksort

Quicksort (due to C.A.R. Hoare) is one of the fastest sorting algorithms on sequential computers. It has $O(n^2)$ worst case time complexity but requires $O(n \log n)$ time on average. The algorithm is based on partitioning the array into two parts using a *pivot*. Once we have the property that the lower side of the array is less than or equal to the pivot, and the upper side of the array has elements greater than the pivot, then we can recurse on each side. Since sorting each of the sides is independent, they can be sorted in parallel.

There are multiple methods to choose a pivot to partition the array A. Choosing an element at random to be the pivot is an easy method that will result in approximately equal sized partitions on average. This gives us the average case sequential time complexity of $O(n \log n)$. In the worst case, the recursion may reduce the range by only 1, resulting in the sequential time complexity of $O(n^2)$. For example, if the array is already sorted and the pivot is always the first element, then each partition produces one part of size 1 and one part of size $n - 1$, giving the recurrence $T(n) = T(n - 1) + O(n)$, which yields $T(n) = O(n^2)$.

We describe a slight variant of the Quicksort algorithm in which we partition the array into three parts. Algorithm ThreeWayPartition takes array A, *low*, and *high* as input parameters.

Algorithm QuickSort: Sequential QuickSort

```
   input: A[low..high]: array of int
1  if low < high then
2  |  pivot := choosePivot();
3  |  (p, q) := Partition(A, pivot, low, high);
4  |  QuickSort(A, low, p);
5  |  QuickSort(A, q, high);
6  end
```

The part of the array this method partitions is given by

$$\{A[i] \mid low \leq i < high\}$$

Note that when *low* equals *high*, the range is empty.

The method *partition* returns two indices p and q. All elements in the range $[low \ldots p)$ are strictly less than the pivot, in the range $[p \ldots q)$ are equal to the pivot, and in the range $[q \ldots high)$ are greater than the pivot. We only need to recurse on the first and the third parts. Depending upon the pivot, any (or both!) of the first and third parts may be empty.

Once we have a pivot, how do we partition the array into three parts: the first partition with all elements less than the pivot, the second one with all elements equal to the pivot, and the third partition with elements strictly greater than the pivot? Algorithm ThreeWayPartition is due to Dijkstra, who called this problem the Dutch National Flag problem. Recall that we maintain the invariant that the entries $[low \ldots p)$ are less than the pivot, $[p \ldots q)$ are equal to the pivot and $[k \ldots high)$ are greater than the pivot. The algorithm initializes p and q to *low*, therefore the first two ranges are empty initially and trivially satisfy the invariants. It also initializes k to *high*, making the range $[k \ldots high)$ empty and thereby ensuring that the invariant holds initially. The range $[q..k)$ corresponds to the initial input and initially contains entries that may be less than, equal to, or greater than the pivot. In each iteration of the *while* loop, this range is reduced by one by increasing q or decreasing k. Depending on the comparison between $A[q]$ and the pivot, the entry $A[q]$ is placed in the appropriate range keeping the invariants.

It is easy to verify that the algorithm takes $O(n)$ time when the range has n elements.

Algorithm ThreeWayPartition: Three-Way partition (Dutch national flag problem)

input: $A[low..high]$: array of int; $pivot$: int
output: (p, q): boundary indices

```
p, q := low;
k := high;
while q < k do
    if A[q] < pivot then
        swap(A[p], A[q]);
        p := p + 1;
        q := q + 1;
    end
    else if A[q] > pivot then
        k := k − 1;
        swap(A[q], A[k]);
    end
    else q := q + 1 ;
end
return (p, q)
```

Example 4.4 Consider partitioning $A = [3, 7, 2, 5, 1, 8, 4]$ with $pivot = 4$, $low = 1$, $high = 8$. Initially $p = q = 1$, $k = 8$.

p	q	k	Array	Action
1	1	8	$[3, 7, 2, 5, 1, 8, 4]$	$A[1]$=3 $<$ 4: swap $A[1], A[1]$; p=2, q=2
2	2	8	$[3, 7, 2, 5, 1, 8, 4]$	$A[2]$=7 $>$ 4: k=7; swap $A[2], A[7]$
2	2	7	$[3, 4, 2, 5, 1, 8, 7]$	$A[2]$=4 = 4: q=3
2	3	7	$[3, 4, 2, 5, 1, 8, 7]$	$A[3]$=2 $<$ 4: swap $A[2], A[3]$; p=3, q=4
3	4	7	$[3, 2, 4, 5, 1, 8, 7]$	$A[4]$=5 $>$ 4: k=6; swap $A[4], A[6]$
3	4	6	$[3, 2, 4, 8, 1, 5, 7]$	$A[4]$=8 $>$ 4: k=5; swap $A[4], A[5]$
3	4	5	$[3, 2, 4, 1, 8, 5, 7]$	$A[4]$=1 $<$ 4: swap $A[3], A[4]$; p=4, q=5

Now $q = k = 5$, so the loop ends. Return $(p, q) = (4, 5)$. The array is $[3, 2, 1, 4, 8, 5, 7]$: positions [1..3] are less than 4, position [4] equals 4, and positions [5..7] are greater than 4.

The same partition routine yields a linear-expected-time algorithm for finding the k-th smallest element of an array, called QuickSelect: instead of recursing on both sides of the pivot, recurse only on the side that contains rank k. This is explored in the chapter exercises.

4.4 Merge Sort

Merge Sort is the classic example of the divide-and-conquer method in algorithm design. To sort an array A, we divide it into two halves, recursively sort each half, and then merge the two sorted halves. The base case is when a half has only a single element and is already sorted. The key step is the merge: given two sorted arrays B and C, we would like to merge them into another array D so that D is sorted.

Example 4.5 Sort $A = [5, 3, 8, 1]$ using Merge Sort.

Split: $[5, 3]$ and $[8, 1]$
Recursively sort left: split $[5]$ and $[3]$, merge $\rightarrow [3, 5]$
Recursively sort right: split $[8]$ and $[1]$, merge $\rightarrow [1, 8]$
Merge $[3, 5]$ and $[1, 8]$: compare 3 and 1 $\rightarrow$ 1; compare 3 and 8 $\rightarrow$ 3;
compare 5 and 8 $\rightarrow$ 5; copy 8. Result: $[1, 3, 5, 8]$.

So, it is sufficient to consider the following problem: we have two sorted arrays B and C, each of size n. We would like to merge them into another array D so that D is sorted.

Algorithm MergeSort: Sequential merge sort

```
   input: A[low..high]: array of int
1  if low < high then
2  |  mid := ⌊(low + high)/2⌋;
3  |  B := MergeSort(A, low, mid);
4  |  C := MergeSort(A, mid + 1, high);
5  |  return MergeTwo(B, C);
6  end
7  else return [A[low]] ;
```

It is easy to design a sequential algorithm that merges two arrays B and C into D. We simply keep two indices i and j in the arrays B and C, respectively. In any step of the algorithm, we compare $B[i]$ and $C[j]$. If $B[i]$ is smaller than $C[j]$, then we copy $B[i]$ into the next available slot in D and advance the index i. If $C[j]$ is smaller than $B[i]$, then we copy $C[j]$ into the next available slot in D and advance the index j. This algorithm takes $O(n)$ time.

Sequentially, $MergeTwo$ takes $O(m + n)$ time.

Algorithm MergeTwo: Merging two sorted arrays

input: $B[1..m], C[1..n]$: sorted arrays of int
output: $D[1..m+n]$: merged sorted array

```
1  i, j, k := 1, 1, 1;
2  while i ≤ m and j ≤ n do
3  |  if B[i] < C[j] then
4  |  |  D[k] := B[i];
5  |  |  i := i + 1;
6  |  end
7  |  else
8  |  |  D[k] := C[j];
9  |  |  j := j + 1;
10 |  end
11 |  k := k + 1;
12 end
13 while i ≤ m do
14 |  D[k] := B[i]; i := i + 1; k := k + 1
15 end
16 while j ≤ n do
17 |  D[k] := C[j]; j := j + 1; k := k + 1
18 end
```

Example 4.6 Merge $B = [2, 5, 8]$ and $C = [1, 4, 9]$.
Step 1: Compare $B[1]$=2 and $C[1]$=1. Since $1 < 2$, copy 1 to D. $D = [1]$.
Step 2: Compare $B[1]$=2 and $C[2]$=4. Since $2 < 4$, copy 2. $D = [1, 2]$.
Step 3: Compare $B[2]$=5 and $C[2]$=4. Copy 4. $D = [1, 2, 4]$.
Step 4: Compare $B[2]$=5 and $C[3]$=9. Copy 5. $D = [1, 2, 4, 5]$.
Step 5: Compare $B[3]$=8 and $C[3]$=9. Copy 8. $D = [1, 2, 4, 5, 8]$.
B is exhausted; copy remaining $C[3]$=9. $D = [1, 2, 4, 5, 8, 9]$.

For n total elements, the recurrence of the merge sort is $T(n) = 2T(n/2) + O(n)$, resulting in $O(n \log n)$ time.

4.5 Lower Bound for Comparison-Based Sorting

We show that any comparison-based sorting algorithm for an array of n distinct elements requires at least $\Omega(n \log n)$ comparisons in the worst case. This result applies to algorithms using unrestricted pairwise comparisons (e.g., Merge Sort), contrasting with the $\Omega(n^2)$ bound for consecutive-only comparisons. We use a decision tree model.

Theorem 4.7 *Given an array $A = [a_1, a_2, \ldots, a_n]$ of n distinct elements, a comparison-based sorting algorithm uses binary comparisons ($a_i < a_j$) to sort A into $a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}$, where π is a permutation. The minimum number of comparisons needed to sort any input permutation in the worst case is $\Omega(n \log n)$.*

Proof: We model the sorting algorithm as a decision tree, where internal nodes represent comparisons, and leaves correspond to permutations. The worst-case number of comparisons is the tree's height.

For n distinct elements, there are $n!$ possible permutations. The algorithm must distinguish all $n!$ permutations, so the decision tree requires at least $n!$ leaves.

In a comparison-based algorithm, each node compares two elements (a_i vs. a_j), branching on the less-than operator ($<$) or the greater-than operator ($>$). With distinct elements, each node has two children (binary tree). After k comparisons, the maximum number of leaves is 2^k.

To cover all permutations:

$$2^k \geq n!.$$

Taking the base-2 logarithm:

$$k \geq \log_2(n!).$$

To show that $\log_2(n!) = \Omega(n \log n)$, observe that

$$n! \;\geq\; \left(\frac{n}{2}\right)^{n/2},$$

since the product $n!$ contains at least $n/2$ factors each at least $n/2$. Taking logarithms:

$$\log_2(n!) \;\geq\; \frac{n}{2} \log_2\left(\frac{n}{2}\right) \;=\; \Omega(n \log n).$$

For $n = 3$, there are $3! = 6$ permutations, so the decision tree needs at least $\lceil \log_2 6 \rceil = 3$ levels. Fig. 4.1 shows a decision tree that sorts three elements using at most 3 comparisons. At each internal node, the label $i{:}j$ denotes the comparison $a_i < a_j$; the left child corresponds to "yes" and the right child to "no" (i.e., $a_i > a_j$). Each leaf is labeled with the sorted order of the indices.

Since $k \geq \log_2(n!)$ and $\log_2(n!) = \Omega(n \log n)$, we conclude:

$$k = \Omega(n \log n).$$

■

4.6 Non-comparison Based Sorting

All our sorting algorithms, so far, have been based on comparison. Any comparison-based sorting algorithm must do at least $\Omega(n \log n)$ work. We now show two non-comparison-based sorting algorithms that escape this lower bound by exploiting structure in the keys: *counting sort*, when the keys come from a small integer range $[0..r-1]$, and *radix sort*, when the keys have a fixed number of digits in some radix r (generally a power of 2). Radix sort uses counting sort as its inner stable pass, so we present counting sort first.

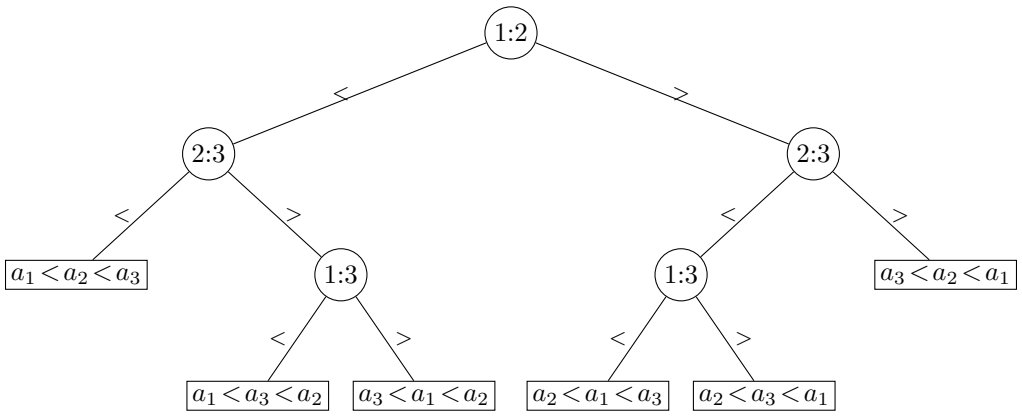


Figure 4.1: A decision tree for sorting three elements. Internal nodes are comparisons $i{:}j$ (is $a_i < a_j$?); left branches mean "yes," right branches mean "no." The six leaves correspond to the six permutations.

4.6.1 Counting Sort

A single linear pass suffices to sort stably when values lie in $[0..r-1]$: count the occurrences of each value, convert the counts to cumulative offsets, then scatter each element to its target slot.

Algorithm CountSort: Counting sort, sequential form.

input: $A[1..n]$ with $A[i] \in \{0, \ldots, r-1\}$
output: $B[1..n]$, stably sorted

```
1  C[v] := 0 for every v ∈ {0, ..., r − 1};
2  for i := 1 to n do
3  |  C[A[i]] := C[A[i]] + 1 ;                          // frequencies
4  end
5  for v := 1 to r − 1 do
6  |  C[v] := C[v] + C[v − 1] ;                        // cumulative offsets
7  end
8  for i := n to 1 by −1 do
9  |  B[C[A[i]]] := A[i];   C[A[i]] := C[A[i]] − 1;
10 end
```

Time and space are both $O(n+r)$. Scanning the input from right to left preserves stability: equal values are placed in reverse input order at decreasing positions of their bucket, so the leftmost equal element ends up first.

4.6.2 Radix Sort

Radix sort processes the keys one digit at a time. In our examples we use base 10, as it is easy for humans to read. Suppose that we need to sort $[85, 72, 94, 45, 13, 12, 61, 81]$. Each key has

two digits, so $k = 2$. When sorting on a single digit, we exploit the fact that there are only r possibilities and use counting sort as the inner pass. The number of passes equals the number of digits.

It may appear that it is easier to sort starting from the most significant digit (msd) first. If we used this strategy, we would get $[13, 12, 45, 61, 72, 85, 81, 94]$ after the first pass. The second pass would consist of sorting all subarrays with the same msd, and we would get the array $[12, 13, 45, 61, 72, 81, 85, 94]$. The problem with this approach is that we are forced to maintain different subarrays, one for each digit after the first pass. With every subsequent pass, it becomes even more cumbersome. Hence, somewhat counterintuitively, we will employ the least significant digit first strategy. After the first pass, we get $[61, 81, 72, 12, 13, 94, 85, 45]$. We do not need to remember any sublists that are created during the first pass. Now, we sort by the second least significant digit. We need to ensure that if two numbers have the same digit, then their relative order from the previous pass is preserved. In other words, we require our sorting algorithm at each pass to be *stable*: a sorting algorithm is stable if elements with equal keys appear in the output in the same order as they appear in the input. After the second pass, we get the sorted array $[12, 13, 45, 61, 72, 81, 85, 94]$. Since 12 appeared before 13 after the first pass, the order is preserved after the second pass.

Example 4.8 Sort $[85, 72, 94, 45, 13, 12, 61, 81]$ by LSD radix sort (base 10).

Pass 1 (sort by ones digit): distribute into buckets:

Digit	Numbers
1	61, 81
2	72, 12
3	13
4	94
5	85, 45

Concatenating: $[61, 81, 72, 12, 13, 94, 85, 45]$.

Pass 2 (sort by tens digit, stably):

Digit	Numbers
1	12, 13
4	45
6	61
7	72
8	81, 85
9	94

Concatenating: $[12, 13, 45, 61, 72, 81, 85, 94]$. Sorted!

Note that stability is essential: in Pass 2, both 81 and 85 have tens digit 8, and their relative order from Pass 1 (81 before 85) is preserved, ensuring correctness.

Thus, the sequential algorithm is simply stated as Algorithm RadixSort, using Counting Sort (Section 4.6.1) for the inner stable pass.

Each CountSort pass runs in $O(n + r)$ time, so the overall time complexity is $O(k(n + r))$,

Algorithm RadixSort: Radix sort for integers with k digits

```
   input: A[1..n]: array of k-digit integers in radix r
1  for i = 1 to k do
2  |  CountSort(A, digit i);                    // stable inner pass
3  end
```

which is $O(kn)$ for constant radix. For $r = 10$, sorting $[85, 72, 94, 45, 13, 12, 61, 81]$ yields $O(kn)$ time (here, $k = 2$, $O(n)$ per pass).

4.7 LLP Perspective

At face value, the sorting problem does not appear to be searching for a satisfying element in a lattice. However, it can be viewed from that angle. Sorting an array A is the same as finding a permutation π such that applying π to A yields the sorted array. For example, if $A = [45, 12, 15]$, then the permutation $[3, 1, 2]$ sorts A once the entry at position i moves to position $\pi(i)$.

There are many ways to represent a permutation. We represent it by its *inversion vector*. The *inversion number* of entry i is the number of entries less than i that appear to the right of i in the permutation. Thus, the permutation $[3, 1, 2]$ is equivalently written as the inversion vector $[2, 0, 0]$: there are two numbers less than 3 that appear to the right of 3, zero numbers less than 1 to the right of 1, and zero to the right of 2. There is a one-to-one correspondence between permutations and inversion vectors.

Consider the set of all inversion vectors. The last entry is always zero (nothing can appear to its right). The first entry can have $0 \ldots n-1$ inversions, the second entry $0 \ldots n-2$, and so on; the total number of inversion vectors is $n!$. We order inversion vectors by componentwise comparison. Under this order, the set of inversion vectors forms a finite distributive lattice. The bottom element is the zero vector, corresponding to the identity permutation. Applying the zero inversion vector to any array leaves it unchanged. *Our goal is to find the inversion vector whose application produces the sorted array.* Figure 4.2 illustrates this lattice for arrays of size three: each node is labeled with an inversion vector $(a, b, 0)$ and the array obtained by applying the corresponding permutation to $A = [45, 12, 15]$. The sorted array $[12, 15, 45]$ sits at $(2, 0, 0)$, so sorting A is equivalent to advancing from the bottom of the lattice to that node.

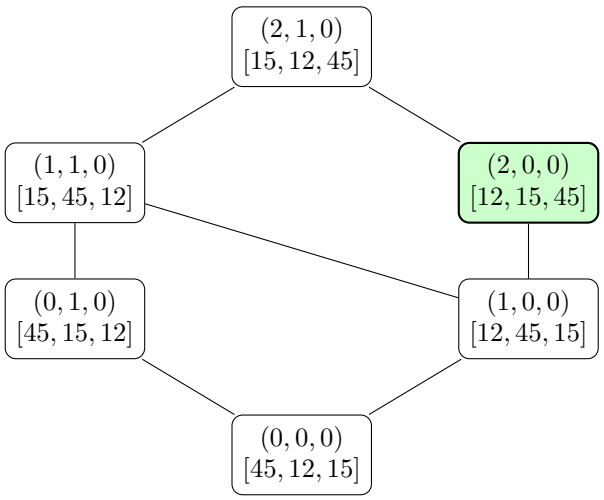


Figure 4.2: Lattice of inversion vectors for arrays of size three, applied to $A = [45, 12, 15]$. Each node shows an inversion vector $(a, b, 0)$ and the array obtained by applying the corresponding permutation to A. The sorted array $[12, 15, 45]$ sits at $(2, 0, 0)$ (shaded); sorting reduces to advancing upward in the lattice to that node.

We formalize when an inversion vector is *feasible*. Suppose G is less than the unique inversion vector that sorts A. Then, when G is applied to A, the array is not sorted, so there exists an index i with a missing inversion: some $j > i$ with $A[j] < A[i]$. Any inversion vector in which the count for index i is not increased cannot reach the sorted array. Rather than maintain G explicitly, we keep only the effect of applying G to A — that is, we mutate A in place.

The simplest choice of forbidden predicate checks only *immediate* inversions: $A[i] > A[i+1]$. Swapping $A[i]$ and $A[i+1]$ advances $G[i]$ by one. This yields Algorithm LLP-Sort-Adjacent.

Algorithm LLP-Sort-Adjacent: Sorting via adjacent swaps

input: $A[1..n]$: array of int
output: $A[1..n]$: sorted array

1 **forbidden**(j): $A[j] > A[j+1]$
2 $\Rightarrow$ **advance**: $swap(A[j], A[j+1])$; `// Increment` $G[j]$

Both Bubble Sort and Insertion Sort are deterministic schedules of LLP-Sort-Adjacent: they fix an order in which the forbidden indices are evaluated (round-robin left-to-right for Bubble Sort; incremental for Insertion Sort). The generic LLP framework, by contrast, is nondeterministic — multiple indices may be forbidden at any point, and any of them may advance.

A slight generalization widens the forbidden predicate to witness *any* right-hand index, not just the immediate neighbor. Algorithm LLP-Sort-Pairwise declares j forbidden whenever any $k > j$ has $A[j] > A[k]$, and advances by swapping $A[j]$ with $A[k]$.

Algorithm LLP-Sort-Pairwise: Sorting via non-adjacent swaps

input: $A[1..n]$: array of int
output: $A[1..n]$: sorted array
1 **forbidden**(j): $\exists\, k > j : A[j] > A[k]$
2 $\Rightarrow$ **advance**: $swap(A[j], A[k])$; //

The forbidden predicate of LLP-Sort-Pairwise implies that of LLP-Sort-Adjacent, so any execution of LLP-Sort-Adjacent is also a valid execution of LLP-Sort-Pairwise. The extra freedom in choosing k lets a single advance step resolve multiple inversions at once; Quicksort can be viewed as an implementation of LLP-Sort-Pairwise where the pivot induces a batched advance: every j on the less-than-pivot side and every k on the greater-than-pivot side are resolved in a single partition pass.

4.8 Summary

Table 4.1 lists the sorting algorithms discussed in this chapter, with their sequential time complexities.

Algorithm	**Time**
Bubble Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Merge Sort	$O(n \log n)$
QuickSort	$O(n^2)$ worst, $O(n \log n)$ average
Counting Sort	$O(n + r)$
Radix Sort	$O(kn)$

Table 4.1: Sequential sorting algorithms. For counting sort, r is the range of input values. For radix sort, k is the number of digits and the radix is treated as a constant.

This chapter has surveyed a wide range of sorting algorithms. The simple comparison-based methods, Bubble Sort and Insertion Sort, run in $\Theta(n^2)$ time. The divide-and-conquer methods are far more efficient: Merge Sort runs in $O(n \log n)$ worst-case time, and QuickSort in $O(n \log n)$ expected time. The non-comparison-based Counting Sort and Radix Sort bypass the comparison lower bound by exploiting structure in the keys, and serve as a reminder that lower bounds are model-specific.

4.9 Problems

1. **(Binary Search Implementation)** Implement the binary search algorithm discussed in Section 4.1.

2. **(Radix Sort Stability)** Radix Sort requires the per-digit pass to be a *stable* sort. Give a small counterexample showing that an unstable per-digit sort can produce an incorrect final output, even when each individual pass correctly groups elements by their current digit.

3. **(k-Way Merge)** Give an algorithm to merge k sorted arrays of size n into a single sorted array.

4. **(Two-Part QuickSort)** We have partitioned the array into three parts in the QuickSort algorithm. Give a version of the QuickSort algorithm in which the array is partitioned only in two parts: entries that are less than *pivot* and the entries that are greater than or equal to the pivot.

5. **(QuickSelect Expected Linear)** Using the partition routine from QuickSort, give an algorithm that returns the k-th smallest element of $A[1..n]$ in expected $O(n)$ time. Justify the expected complexity.

6. **(Sorting Algorithm Stability)** A sorting algorithm is *stable* if it preserves the relative order of records with equal keys. State which of the following are stable, and justify each: Bubble Sort, Insertion Sort, Merge Sort, QuickSort (Lomuto and Hoare partitions), and Radix Sort.

7. **(Counting Sort)** Given an array $A[1..n]$ of integers in the range $[0, k]$, give an algorithm that sorts A in $O(n + k)$ time. Discuss when Counting Sort is preferable to Radix Sort.

8. **(External Memory Merge Sort)** Suppose the array $A[1..n]$ does not fit in memory. RAM holds $M \ll n$ elements, and reads and writes to disk occur in blocks of B elements. Describe an external-memory Merge Sort and count the number of block I/Os it performs.

9. **(k-Near-Sorted Sorting)** An array $A[1..n]$ is said to be *k-near-sorted* if every element is at most k positions away from its position in the sorted output. Show that such an array can be sorted in $O(n \log k)$ time. (Hint: maintain a min-heap of size $k + 1$.)

10. **(Incremental Sort)** A sorted array $A[1..n]$ has a new element a_{n+1} inserted. Update the sort in $O(\log n + k)$ time, where k is the number of elements displaced.

11. **(Stable Sort with Ties)** The comparison sort must be *stable*: equal-key elements must keep their input order. Show that LLP-Sort-Adjacent is automatically stable, but LLP-Sort-Pairwise is not.

4.10 Bibliographic Remarks

Quicksort was invented by Hoare in 1961 [Hoa61]; the original paper already contained a two-way in-place partition. The three-way "Dutch National Flag" partition is due to Dijkstra and appears in *A Discipline of Programming* [Dij76]. Merge sort is usually attributed to von Neumann in 1945, and its recurrence-based analysis is given in Knuth's *The Art of Computer Programming, Vol. 3: Sorting and Searching* [Knu98], which also catalogs Bubble Sort, Insertion Sort, and numerous refinements. Radix sort predates digital computers: it was the basis of Herman Hollerith's electromechanical tabulator used for the 1890 U.S. Census, and a modern analysis (including stability via Counting Sort as the inner pass) appears in [Knu98]. The $\Omega(n \log n)$ lower bound for comparison-based sorting via the decision-tree argument is also treated in [Knu98].

Chapter 5

Graphs

5.1 Introduction

Graph theory, a cornerstone of algorithmic design, is the foundation of numerous modern applications in diverse domains. In *social network analysis*, graphs model relationships where vertices represent individuals and edges denote interactions, enabling the detection of communities using algorithms such as clustering. In *transportation and logistics*, directed graphs optimize routing: companies like Amazon use shortest-path algorithms (for example, Dijkstra's algorithm) in road networks to minimize delivery times, while airlines use bipartite matching to schedule crews efficiently. *Computer networks* rely on graphs to represent topologies, with spanning tree protocols ensuring loop-free communication and centrality measures identifying critical nodes for cybersecurity. In *bioinformatics*, graphs map protein interactions or gene regulatory networks, where traversal algorithms identify functional pathways, helping drug discovery.

Graphs come in numerous varieties. They may be *directed* or *undirected*. They may be *simple* or with loops and parallel edges. They may be *weighted* or *unweighted*. They may be *acyclic* or not. For example, when modeling transportation networks, the nodes may represent important milestones in a city. A directed edge may represent a one-way street, and an undirected edge may represent a street that can be traversed in either direction. The weights may represent the distance between the nodes.

In this chapter, we cover some basic traversal algorithms in a simple directed graph. This chapter is organized as follows. Section 5.2 describes various commonly used representations of graphs. Section 5.3 gives the breadth-first-search (BFS) method of traversing a graph. Section 5.4 gives the depth-first-search (DFS) method of traversing a graph. Section 5.5 gives an algorithm to "layer" a directed acyclic graph. Section 5.6 shows how to topologically sort a directed acyclic graph. Section 5.7 gives an algorithm to compute the strongly connected components of a directed graph. Section 5.8 revisits these problems from the Lattice-Linear Predicate (LLP) perspective. Many other algorithms on graphs, such as the shortest path algorithms and minimum spanning tree algorithms, are covered in later chapters.

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

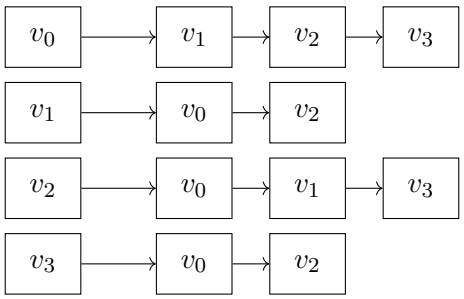


Figure 5.2: Adjacency list representation of an undirected graph

5.2 Graph Representations

We present an undirected graph and its directed counterpart, each with vertices v_0, v_1, v_2, v_3, shown with adjacency matrix and adjacency list representations (as linked lists).

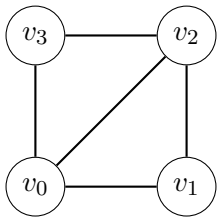


Figure 5.1: An undirected graph

The graph in Fig. 5.1 can be represented using an adjacency matrix as follows.

$$\begin{bmatrix} & v_0 & v_1 & v_2 & v_3 \\ v_0 & 0 & 1 & 1 & 1 \\ v_1 & 1 & 0 & 1 & 0 \\ v_2 & 1 & 1 & 0 & 1 \\ v_3 & 1 & 0 & 1 & 0 \end{bmatrix}$$

An entry of 1 indicates an edge; the matrix is symmetric. Such a representation allows one to answer any question of the form: is there an edge between v_i and v_j in constant time. However, it takes $O(n^2)$ space irrespective of the number of edges in the graph. When the graph is sparse, the adjacency list representation is less wasteful. Fig. 5.2 shows the adjacency list representation of the same graph.

We now consider the directed version of the graph in Fig. 5.3. As before, we can have the adjacency matrix and adjacency list representations shown below.

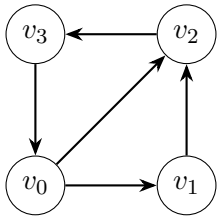


Figure 5.3: A directed graph.

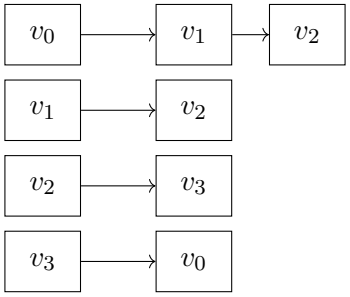


Figure 5.4: Adjacency representation of a directed graph

$$\begin{bmatrix} & v_0 & v_1 & v_2 & v_3 \\ v_0 & 0 & 1 & 1 & 0 \\ v_1 & 0 & 0 & 1 & 0 \\ v_2 & 0 & 0 & 0 & 1 \\ v_3 & 1 & 0 & 0 & 0 \end{bmatrix}$$

5.3 Breadth-First Search (BFS)

In this section, we describe the breadth-first-search (BFS), one of the two most common ways of traversing a directed graph (the other being depth-first search, covered in the next section). Traversal is required for many reasons. We may be interested in searching for a node, finding the number of components in a graph, or checking for some user-specified property. We first present a general (nondeterministic) traversal algorithm; different strategies for choosing which vertex to explore next yield BFS or DFS as special cases.

General Traversal of a Directed Graph

In Algorithm Traversal, we keep *visited* as the Boolean vector marking the set of vertices that can be reached from the vertex v_0 ($visited[i] = 1$ iff v_i has been reached). We also keep S as a set of vertices that have been visited but not explored. Using different strategies to remove an index from S, we can traverse the graph in a different order. If S is maintained as a queue, the removal corresponds to removing from the head of the queue, and the addition corresponds to

appending at the end of the queue, then we traverse the graph in a *breadth-first* manner. If S is maintained as a stack such that removal corresponds to the *pop*() and addition corresponds to the *push*() method on the stack, then we traverse the graph in a depth-first search manner.

Algorithm Traversal: Reachable vertices from a source vertex

input: $dep(j)$: adjacency list for each vertex j
output: $visited[0..n-1]$: int, initially 0

```
1  S: set of indices;
2  S.add(0) // add 0 as the initial vertex to start
3  visited[0] := 1;
4  while ¬S.empty() do
5  |  j := S.removeAny(visited) // remove any index from S
6  |  forall (k ∈ dep(j)) do
7  |  |  if (visited[k] = 0) then
8  |  |  |  visited[k] := 1 // advance on k
9  |  |  |  S.add(k) // update frontier
10 |  |  end
11 |  end
12 end
13 return visited ;                                  // the reachable set
```

Each vertex is added to and removed from S at most once, and each edge (j, k) is examined exactly once (when j is being explored). Hence Algorithm Traversal runs in $O(n + m)$ time.

Suppose that we are given one of the nodes v_0 in the graph. Our task is to find all the nodes that are reachable from v_0. Algorithm BFS-Traversal can be used for this purpose. It will generate a tree rooted at v_0 such that every node is at the minimum distance from v_0. We maintain a variable *parent* that gives the parent of any node x in the graph. If x is reachable from v_0, then following the *parent* pointer from x to v_0, we will get a shortest path from v_0 to x. A node v_i has its parent set to v_j if v_j is along a path from v_0 to v_i and has its distance one less than that of v_i.

Each vertex is enqueued at most once and dequeued at most once; each edge (j, k) is examined exactly once when j is dequeued. Hence Algorithm BFS-Traversal runs in $O(n+m)$ time.

Algorithm BFS-Traversal: Breadth-first search tree in a directed graph

input: $dep(j)$: adjacency list for each vertex j
s: source vertex
output: $dist[0..n-1]$: int, initially ∞
$parent[0..n-1]$: int, initially -1

```
1  S: queue of indices;
2  S.add(s);
3  dist[s] := 0;
4  while ¬S.empty() do
5  |  j := S.removeFirst();
6  |  forall (k ∈ dep(j)) do
7  |  |  if (dist[k] > dist[j] + 1) then
8  |  |  |  dist[k] := dist[j] + 1;
9  |  |  |  parent[k] := j;
10 |  |  |  append k to S;
11 |  |  end
12 |  end
13 end
14 return dist ;                    // distances from s; BFS tree given by parent
```

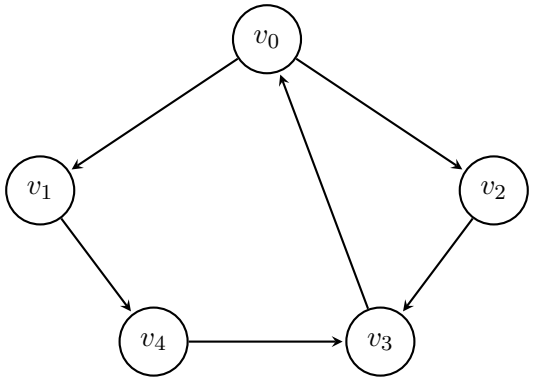


Figure 5.5: A directed graph used to illustrate BFS and DFS traversals.

Example 5.1 [BFS on a directed graph] Run BFS from source v_0 on the graph in Figure 5.5. Initially $dist[v_0] = 0$ and the queue is $[v_0]$.

- Dequeue v_0. Neighbors v_1, v_2 have $dist = \infty$; set $dist[v_1] = 1, dist[v_2] = 1, parent[v_1] = parent[v_2] = v_0$, enqueue both.
- Dequeue v_1. Neighbor v_4: set $dist[v_4] = 2$, $parent[v_4] = v_1$, enqueue v_4.
- Dequeue v_2. Neighbor v_3: set $dist[v_3] = 2$, $parent[v_3] = v_2$, enqueue v_3.
- Dequeue v_4. Neighbor v_3 already has $dist[v_3] = 2$ (not improved).
- Dequeue v_3. Neighbor v_0 already has $dist[v_0] = 0$ (not improved).

Final distances from v_0:

Node	$dist[\cdot]$ (distance)	$parent[\cdot]$
v_0	0	–
v_1	1	v_0
v_2	1	v_0
v_4	2	v_1
v_3	2	v_2

The BFS tree is shown in Figure 5.6.

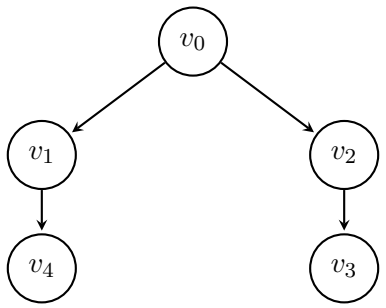


Figure 5.6: BFS tree rooted at v_0 for example 5.1. Every vertex appears at its shortest distance from v_0.

5.4 Depth-First Search (DFS)

Another way to traverse a graph is the depth-first search method. While the breadth-first search traverses the graph one layer at a time, the depth-first search continues to take a path as far as it can go, and then it backtracks to traverse the remaining graph (in the same fashion!).

Algorithm DFS-Traversal: Depth-first search tree in a directed graph

input: $dep(j)$: adjacency list for each vertex j
output: $visited[0..n-1]$: int, initially 0
$parent[0..n-1]$: int, initially -1
$discovered[0..n-1]$: int
$finished[0..n-1]$: int

```
1  tick: int, initially 1;
2  Function DFS(j):
3      visited[j] := 1;
4      discovered[j] := tick;
5      tick := tick + 1;
6      for k ∈ dep(j) do
7          if visited[k] = 0 then
8              parent[k] := j;
9              DFS(k);
10         end
11     end
12     finished[j] := tick;
13     tick := tick + 1;
14 end
15 DFS(0);
```

The recursive call $DFS(k)$ is made for each vertex k at most once, and each edge (j, k) is examined exactly once when j's adjacency list is scanned. Hence Algorithm DFS-Traversal runs in $O(n + m)$ time.

Algorithm DFS-Traversal visits all vertices reachable from vertex v_0 in a depth-first search manner. The variable *visited* is used to mark all the vertices that can be reached from v_0. The variable *discovered* is used to store the time (tick) when each vertex in the graph is explored. The variable $finished$ is used to store the tick when the algorithm is finished exploring a vertex. The variable *parent* is used to store the depth-first tree. Anytime a vertex k is visited for the first time from a vertex j, the $parent[k]$ is recorded as j.

The variables *discovered* and $finished$ are used only to record the *tick* when a vertex is starting to be explored and finished for exploration.

Example 5.2 [DFS on a directed graph] Run DFS from source v_0 on the graph in Figure 5.5. From v_0 we visit v_1; from v_1 we visit v_4; from v_4 we visit v_3. When v_3 is explored, its only outgoing edge points to v_0, which is already visited, so v_3 finishes. We backtrack to v_4 (no other outgoing edges), then to v_1, and finally to v_0. From v_0 we now visit v_2, whose outgoing edge leads to v_3 (already visited); v_2 finishes, and DFS terminates.

The resulting *discovered* and *finished* ticks are:

Node	*discovered*	*finished*
v_0	1	10
v_1	2	7
v_4	3	6
v_3	4	5
v_2	8	9

The DFS tree is shown in Figure 5.7.

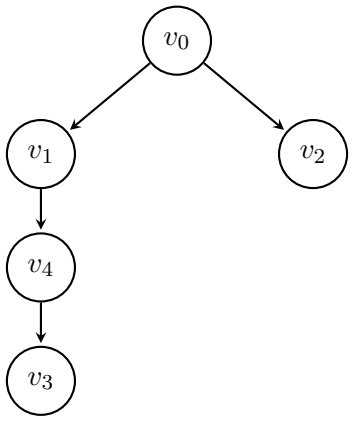


Figure 5.7: DFS tree rooted at v_0 for example 5.2. The edge $v_2 \to v_3$ of the original graph is a *cross edge* (not part of the DFS tree, since v_3 is discovered earlier via v_4).

5.5 Layering of a Directed Acyclic Graph

Suppose that we are given a directed graph with no cycles. This graph can be "layered" as follows . Every vertex i in the graph is assigned a number $G[i]$ such that if there is a directed edge from i to j, then the number assigned to i is strictly less than that of j, i.e., for all edges $(i, j) \in E$, $G[i]$ is less than $G[j]$. An application of this concept is the prerequisite structure of courses in a university, and the integer assigned to the course corresponds to the earliest semester in which the course can be taken by a student.

Our goal is to compute, for each vertex j, the integer $G[j]$ equal to the length of the longest path from any source vertex (a vertex with no incoming edges) to j. We present a classical algorithm that maintains the in-degree of every vertex and processes vertices in the order in which all of their predecessors have been assigned layers.

The idea is simple. Initially, every vertex with in-degree 0 lies at layer 0. We place all such vertices in a queue. We then repeatedly remove a vertex j from the queue, and for every successor k of j (i.e., $(j,k) \in E$), we decrement k's in-degree. Whenever k's in-degree becomes 0, we have seen all of k's predecessors and know their layers; we then set $G[k]$ to one more than the maximum of $G[i]$ over all predecessors i of k, and enqueue k. Algorithm Layering gives the details.

Algorithm Layering: Layering a directed acyclic graph

input: directed acyclic graph (V,E) with $|V| = n$, $|E| = m$
$pre(j)$: list of predecessors of j
$succ(j)$: list of successors of j
output: $G[0..n-1]$: layer of each vertex

```
1  indeg[j] := |pre(j)| for every j;
2  Q: queue of indices initially empty;
3  forall j ∈ V do
4  |  if indeg[j] = 0 then
5  |  |  G[j] := 0;
6  |  |  enqueue j into Q;
7  |  end
8  end
9  while ¬Q.empty() do
10 |  j := Q.removeFirst();
11 |  forall k ∈ succ(j) do
12 |  |  indeg[k] := indeg[k] − 1;
13 |  |  if indeg[k] = 0 then
14 |  |  |  G[k] := 1 + max_{i∈pre(k)} G[i];
15 |  |  |  enqueue k into Q;
16 |  |  end
17 |  end
18 end
19 return G;
```

Each vertex is enqueued and dequeued at most once, and each edge (j,k) is processed exactly once when j is dequeued. Hence the time complexity of the algorithm is $O(n+m)$.

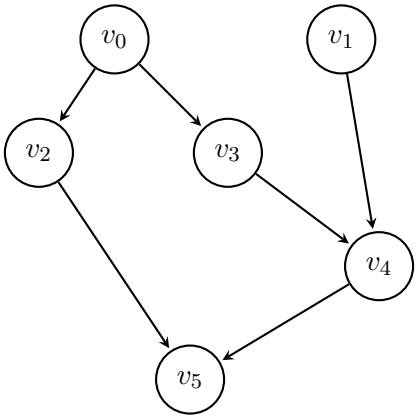


Figure 5.8: A directed acyclic graph used to illustrate layering.

Example 5.3 [Layering a DAG] Run the layering algorithm on the DAG of Figure 5.8. In-degrees are $indeg[v_0] = indeg[v_1] = 0$, $indeg[v_2] = indeg[v_3] = 1$, $indeg[v_4] = indeg[v_5] = 2$.

- Initially v_0 and v_1 have in-degree 0; set $G[v_0] = G[v_1] = 0$ and enqueue both. Queue: $[v_0, v_1]$.
- Dequeue v_0. Successors v_2, v_3: their in-degrees drop to 0; set $G[v_2] = G[v_3] = 1$ and enqueue them.
- Dequeue v_1. Successor v_4: in-degree drops to 1 (not yet ready).
- Dequeue v_2. Successor v_5: in-degree drops to 1 (not yet ready).
- Dequeue v_3. Successor v_4: in-degree drops to 0; set $G[v_4] = 1 + \max(G[v_1], G[v_3]) = 1 + \max(0, 1) = 2$ and enqueue v_4.
- Dequeue v_4. Successor v_5: in-degree drops to 0; set $G[v_5] = 1 + \max(G[v_2], G[v_4]) = 1 + \max(1, 2) = 3$ and enqueue v_5.
- Dequeue v_5 (no successors).

The computed layers are

Vertex	$G[\cdot]$ (layer)
$v_0,\ v_1$	0
$v_2,\ v_3$	1
v_4	2
v_5	3

The layered view of the DAG is shown in Figure 5.9. Each $G[j]$ equals the length of the longest path from any source to j; the longest path to v_5 is $v_0 \to v_3 \to v_4 \to v_5$ of length 3.

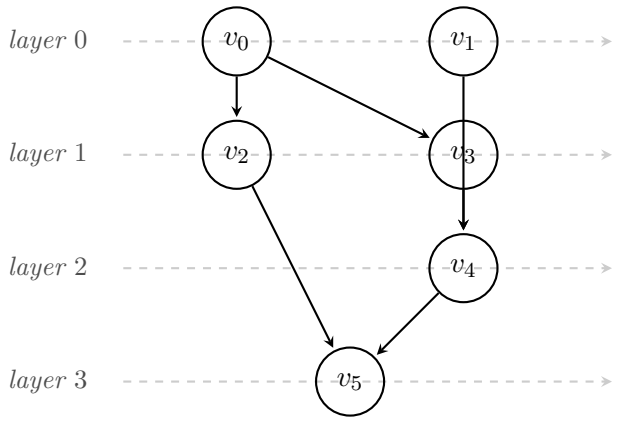


Figure 5.9: The DAG of figure 5.8 redrawn with each vertex placed on its computed layer. Every directed edge goes from a strictly smaller layer to a strictly greater one.

5.6 Topological Sort

Layering assigns a number $G[j]$ to each vertex such that $G[i] < G[j]$ along every edge (i, j), but it does not break ties between vertices on the same layer. In many applications — evaluating arithmetic expressions, ordering recipe steps, scheduling courses one per term, processing dependencies in a build system — we want a *total* order on the vertices that respects every edge. This is called a topological sort of the DAG: a linear arrangement $v_{\sigma(1)}, v_{\sigma(2)}, \ldots, v_{\sigma(n)}$ such that for every edge $(v_i, v_j) \in E$, v_i precedes v_j in the arrangement.

A simple modification of Layering produces a topological sort. Instead of recording the layer of each vertex, we record the order in which it is dequeued: the first vertex dequeued gets position 1, the second gets position 2, and so on. The resulting permutation respects all edges because a vertex can be dequeued only after all of its predecessors have been dequeued. Because the change to Layering is so small, we leave the algorithm as an exercise (Problem 9).

The time complexity is $O(n+m)$ for the same reason as Layering: each vertex is enqueued and dequeued at most once and each edge is examined exactly once. If the input graph has a cycle, fewer than n vertices are ever dequeued — a cheap cycle-detection byproduct.

Example 5.4 Consider the DAG of Figure 5.8 (in-degrees $indeg[v_0] = indeg[v_1] = 0$, $indeg[v_2] = indeg[v_3] = 1$, $indeg[v_4] = indeg[v_5] = 2$). Initial queue $[v_0, v_1]$. Dequeueing in order produces $order = [v_0, v_1, v_2, v_3, v_4, v_5]$. The reader can verify that every edge of Figure 5.8 runs left-to-right in this list. The arrangement is not unique: starting from initial queue $[v_1, v_0]$ would yield $[v_1, v_0, v_2, v_3, v_4, v_5]$, also a valid topological order.

Topological sort underlies many later algorithms in this book: dynamic programming on DAGs (e.g., longest path, shortest path on a DAG), the 2-SAT decision algorithm (Chapter 18), and dependency-driven scheduling.

5.7 Strongly Connected Components

Given a directed graph $\mathcal{G} = (V, E)$, two vertices u and v are said to be in the same *strongly connected component* (SCC) if there is a directed path from u to v and a directed path from v to u. The relation "u and v are mutually reachable" is an equivalence relation, so the vertex set V partitions into maximal strongly connected components. Collapsing each SCC into a single super-vertex yields the *component graph* $\mathcal{G}^{\text{scc}}$, which is always a directed acyclic graph.

Layering handles the acyclic case; SCC computation is the natural counterpart for graphs that may contain cycles. SCC decomposition is used, for example, in compiler dataflow analysis (to identify strongly connected regions of a control-flow graph), in dependency resolution (to detect cyclic package or module dependencies), and in 2-SAT (where satisfiability reduces to computing SCCs of the implication graph).

We present Kosaraju's algorithm, a simple and efficient $O(n+m)$-time algorithm that uses two depth-first searches (DFS). Its correctness rests on the following lemma.

Lemma 5.5 *Let u and v be vertices in distinct strongly connected components S_u and S_v of $\mathcal{G}$ (each S_x is the vertex set of the SCC containing x). Let $f(S) = \max_{w \in S} f[w]$ denote the maximum DFS finishing time among vertices of component S. If there is an edge in $\mathcal{G}^{\text{scc}}$ from S_u to S_v, then $f(S_u) > f(S_v)$.*

Proof: Consider the DFS executed on $\mathcal{G}$ and the vertex that is visited first among $S_u \cup S_v$. If it is in S_u, then since S_u is strongly connected, DFS explores all of S_u and, via the edge to S_v, all of S_v before finishing the start vertex, so $f(S_u) > f(S_v)$. If instead the first visited vertex lies in S_v, then DFS finishes S_v first (since no edge returns from S_v to S_u in $\mathcal{G}^{\text{scc}}$); the earliest vertex visited in S_u is explored later, and its finishing time — hence $f(S_u)$ — is strictly greater than $f(S_v)$.

■

The SCC-id vector $C[\,]$ assigned by Algorithm Kosaraju-SCC satisfies $C[u] = C[v]$ iff the vertex sets $S_u = S_v$, i.e., iff u and v lie in the same strongly connected component.

As a consequence, if we process vertices in decreasing order of finishing time from the first DFS but on the *transpose* graph $\mathcal{G}^{\text{T}}$ (the graph obtained by reversing every edge), then each DFS tree from a new root is exactly one SCC. Edges in $\mathcal{G}^{\text{T}}$ go from lower-f components to higher-f components, so a DFS from the vertex of maximum f in $\mathcal{G}^{\text{T}}$ cannot leave its SCC; after that SCC is exhausted, we pick the next unvisited vertex with the largest remaining f, and so on.

Theorem 5.6 *Algorithm Kosaraju-SCC computes the strongly connected components of $\mathcal{G}$ in $O(n+m)$ time.*

Proof: *Correctness.* By Lemma 5.5, for every edge $(S_u, S_v) \in \mathcal{G}^{\text{scc}}$ we have $f(S_u) > f(S_v)$. Reversing all edges gives $\mathcal{G}^{\text{T}}$ with $(S_v, S_u) \in \mathcal{G}^{\text{scc,T}}$; consequently, when we launch a DFS in $\mathcal{G}^{\text{T}}$ from the vertex with the largest f-value in its component, no transpose edge can leave the component, so the reached set is exactly that SCC. Subsequent iterations remove fully visited components and repeat on the remaining sub-DAG.

Algorithm Kosaraju-SCC: Strongly connected components

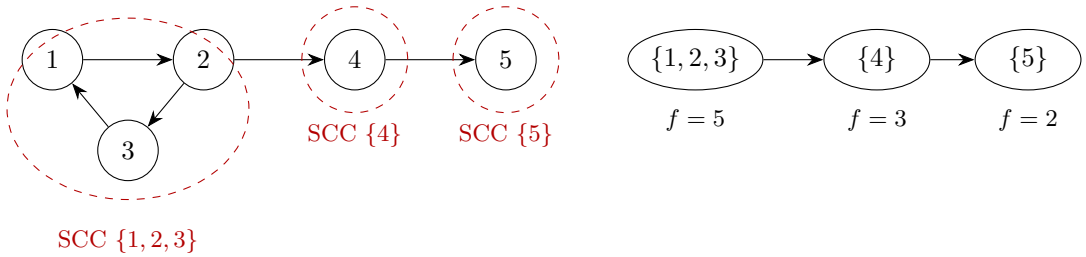
input: directed graph $\mathcal{G} = (V, E)$ with $|V| = n$, $|E| = m$
output: $C[1..n]$: $C[v]$ is the identifier of the SCC containing v

```
// Pass 1:  finishing-time DFS on G
```
Perform DFS on $\mathcal{G}$ and record the finishing time $f[v]$ for every vertex v;
Let $S = [v_1, v_2, \ldots, v_n]$ be the list of vertices sorted in decreasing order of f;

```
// Pass 2:  DFS on the transpose in reverse finishing order
```
Construct $\mathcal{G}^{\mathrm{T}} = (V, E^{\mathrm{T}})$ where $E^{\mathrm{T}} = \{(v, u) \mid (u, v) \in E\}$;
Mark all vertices in $\mathcal{G}^{\mathrm{T}}$ as unvisited;
$k := 0$;
for $i = 1$ *to* n **do**
  if v_i *is unvisited in* $\mathcal{G}^{\mathrm{T}}$ **then**
    $k := k + 1$;
    Run DFS in $\mathcal{G}^{\mathrm{T}}$ starting at v_i; for every vertex w reached, set $C[w] := k$ and mark w visited;
  end
end
return C

Complexity. Each of the two DFS passes runs in $O(n + m)$ time. Constructing $\mathcal{G}^{\mathrm{T}}$ also takes $O(n + m)$ time using adjacency lists. Sorting vertices by finishing time can be done implicitly by pushing each vertex onto a stack when it finishes during the first DFS, so no additional sorting cost is incurred.

■



(a) Original graph $\mathcal{G}$ (b) Condensation $\mathcal{G}^{\mathrm{scc}}$

Figure 5.10: (a) example directed graph $\mathcal{G}$ with three strongly connected components $\{1, 2, 3\}$, $\{4\}$, and $\{5\}$. (b) the condensation $\mathcal{G}^{\mathrm{scc}}$, with each node labeled by the maximum DFS finishing time $f(S)$ of its component. Edges in the condensation point from larger-f components to smaller, illustrating lemma 5.5.

Example 5.7 Consider the directed graph on $\{1, \ldots, 5\}$ with edges E shown in Figure 5.10, where

$$E = \{(1,2), (2,3), (3,1), (2,4), (4,5)\}.$$

DFS starting at vertex 1 finishes the vertices in order $3, 5, 4, 2, 1$, giving finishing times $f[3]$=1, $f[5]$=2, $f[4]$=3, $f[2]$=4, $f[1]$=5. The transpose graph has edges

$$E^{\mathrm{T}} = \{(2,1), (3,2), (1,3), (4,2), (5,4)\}.$$

Processing vertices in decreasing order of f on the transpose yields three DFS trees: $\{1,2,3\}$ from vertex 1, $\{4\}$ from vertex 4, and $\{5\}$ from vertex 5, matching the three SCCs circled in Figure 5.10.

5.8 LLP Perspective

The classical graph algorithms of the preceding sections — traversal, BFS, layering — each have a natural LLP formulation. The value of recasting them is not new algorithms but a uniform correctness argument: once the predicate is shown to be lattice-linear, correctness follows from the LLP framework (Chapter 2) rather than from a case-specific proof. The LLP formulation also reveals which algorithms admit free parallelism — any set of independently forbidden indices can advance simultaneously.

Graph traversal as an LLP algorithm

Before considering BFS, which tracks distances, we revisit the plain reachability problem of Section 5.3: given a source vertex v_0, compute the set of vertices reachable from v_0. We maintain a Boolean vector G with $G[0] = 1$ and $G[j] = 0$ for $j \neq 0$ and search for the least vector satisfying

$$B_{reach}(G) \equiv G[0] = 1 \ \wedge \ \forall (i,j) \in E : (G[i] = 1 \Rightarrow G[j] = 1).$$

An index j is forbidden when some predecessor i has $G[i] = 1$ but $G[j] = 0$; the advance rule sets $G[j] := 1$. Algorithm LLP-Traversal captures this.

Algorithm LLP-Traversal: Reachable vertices via LLP

input: $edges(j)$: list of adjacent vertices of j
output: $G[j]$: int, initially 1 if $j = 0$, else 0
1 **forbidden**(j): $\exists i : j \in edges(i) \wedge G[i] = 1 \wedge G[j] = 0$
2 $\Rightarrow$ **advance**: $G[j] := 1$;

Each $G[j]$ advances at most once (from 0 to 1), and evaluating the forbidden predicate amortises to $O(n+m)$ work across all indices. The sequential running time is therefore $O(n+m)$.

BFS as an LLP algorithm

For breadth-first search from a source vertex v_0 we maintain a distance vector G with $G[0] = 0$ and $G[j] = \infty$ for $j \neq 0$. We want the least vector satisfying

$$B_{bfs}(G) \equiv \forall j \neq 0 : \forall (i,j) \in E : G[j] \leq G[i] + 1.$$

An index j is forbidden if it has a predecessor i with $G[j] > G[i] + 1$; the advance rule sets $G[j] := G[i] + 1$. Algorithm LLP-BFS gives the resulting LLP algorithm.

Algorithm LLP-BFS: BFS distances via LLP

input: $pre(j)$: list of predecessors of vertex j
output: $G[j]$: int, initially 0 if $j = 0$, else ∞
1 **forbidden**(j): $G[j] > \min\{G[i] + 1 \mid i \in pre(j)\}$
2 $\Rightarrow$ **advance**: $G[j] := \min\{G[i] + 1 \mid i \in pre(j)\}$;

A sequential implementation that evaluates forbidden indices in breadth-first order runs in $O(n + m)$ time and matches the classical BFS bound.

The LLP specification above does not fix an order in which forbidden indices are evaluated. In sequential or resource-constrained settings, it can be advantageous to prioritize which forbidden index is processed first. We write the prioritization with a **priority** clause alongside **forbidden** and **advance**. In Algorithm LLP-Traversal-BFS-priority, priority is given by the current value of $G[j]$, so the search explores vertices in order of distance from v_0 — exactly as in classical BFS with a queue.

Algorithm LLP-Traversal-BFS-priority: Prioritized BFS traversal via LLP

input: $pre(j)$: list of predecessors of vertex j
output: $G[j]$: int, initially 0 if $j = 0$, else ∞
1 **ensure**(j): $G[j] \leq \min\{G[i] + 1 \mid i \in pre(j)\}$;
2 **priority**: $G[j]$

Processing indices in increasing $G[j]$ order using a priority queue keyed on $G[j]$ matches the classical BFS schedule and gives a sequential running time of $O(n + m)$.

Layering as an LLP algorithm

For layering a directed acyclic graph we want the least vector G of natural numbers satisfying

$$B_{layer}(G) \equiv \forall (i,j) \in E : G[i] < G[j].$$

Define $fixed[j]$ to be true once $G[j]$ has been finalized. An index j is forbidden when it is not fixed but all of its predecessors are fixed; the advance rule then sets $G[j]$ to one more than the maximum layer of its predecessors. Algorithm LLP-Layering captures this.

Algorithm LLP-Layering: DAG layering via LLP

input: $pre(j)$: list of predecessors of vertex j
output: $G[j]$: int, initially 0
$fixed[j]$: boolean, initially *true* if $pre(j) = \{\}$, else *false*
1 **forbidden**(j): $\neg fixed[j] \wedge (\forall i \in pre(j) : fixed[i])$
2 $\Rightarrow$ **advance**: $G[j] := \max_{i \in pre(j)} G[i] + 1$; $fixed[j] := true$;

A sequential implementation that evaluates forbidden indices in any topological order runs in $O(n+m)$ time.

5.9 Summary

This chapter covered fundamental graph algorithms: breadth-first search (BFS), depth-first search (DFS), layering of directed acyclic graphs, topological sorting, strongly connected components, and connected components.

Table 5.1 summarizes the time complexities of the sequential algorithms discussed in this chapter.

Problem	Algorithm	Time Complexity
Breadth-First Search	BFS-Traversal	$O(m+n)$
Depth-First Search	DFS-Traversal	$O(m+n)$
Breadth-First Search	LLP-BFS	$O(m+n)$
Layering	Layering	$O(m+n)$
Layering	LLP-Layering	$O(m+n)$
Strongly Connected Components	Kosaraju-SCC	$O(m+n)$

Table 5.1: Algorithms discussed in this chapter.

5.10 Problems

1. **(BFS Component Sizes)** Given an undirected graph with n vertices and m edges, find the size (number of vertices) of each connected component using BFS. The graph is represented as an adjacency list. Return a list of sizes for all components.

2. **(Undirected Cycle Detection DFS)** Given an undirected graph with n vertices and m edges, determine if the graph contains a cycle using DFS traversal. The graph is represented as an adjacency list.

3. **(Bipartite Check By Coloring)** Give an efficient algorithm to check if the given undirected graph $G = (V, E)$ is bipartite.

4. **(DFS Component Count)** Give an algorithm to find the number of connected components in an undirected graph using DFS.

5. **(Find All Bridges)** Give an efficient algorithm to find all bridges in an undirected graph. A bridge is an edge whose removal increases the number of connected components.

6. **(Find Articulation Points)** Given an undirected graph with n vertices and m edges, find all articulation points using DFS traversal. An articulation point is a vertex whose removal increases the number of connected components in the graph.

7. **(Cycle Error In Layering)** Algorithm Layering gives us the least integral labels satisfying that, for all edges (i, j), $G[i]$ is strictly less than $G[j]$. Such a property is

possible only for acyclic graphs. Modify the algorithm to output "error" whenever there is a cycle in the input graph.

8. **(Dual Layering From Top)** Algorithm Layering works by assigning level numbers to the minimal vertices in the graph and then to the next layer of vertices. Give the dual algorithm that starts with assigning level number to the maximal vertices and then to the layer of vertices that are below the maximal level.

9. **(Topological Sort by Modifying Layering)** Modify the Layering algorithm (Algorithm Layering) so that, instead of assigning a layer number to each vertex of a DAG, it produces a topological sort: a linear arrangement $v_{\sigma(1)}, v_{\sigma(2)}, \ldots, v_{\sigma(n)}$ of the vertices such that for every edge $(v_i, v_j) \in E$, v_i precedes v_j. Your algorithm should run in $O(n + m)$ time and report "error" when the input graph contains a cycle.

10. **(Directed Cycle Three Color DFS)** Problem 2 detects cycles in an *undirected* graph by treating any non-parent visited neighbor as a back edge. This rule fails on a directed graph because a directed edge to an already-visited vertex may simply be a forward or cross edge in the DFS tree, not a cycle. Give an algorithm that decides whether a directed graph contains a cycle, using DFS with three-color marking: *white* (unvisited), *gray* (on the recursion stack), *black* (fully explored).

5.11 Bibliographic Remarks

Breadth-first and depth-first search are classical graph-traversal techniques; both are treated in detail in Cormen, Leiserson, Rivest, and Stein [CLRS01]. The layering of a directed acyclic graph (equivalently, topological sort by longest-path length) is covered in most algorithms texts, including [CLRS01]. The in-degree-based approach used in Algorithm Layering is due to A. B. Kahn (1962). Algorithm Kosaraju-SCC for strongly connected components (two DFS passes on $\mathcal{G}$ and $\mathcal{G}^{\mathrm{T}}$) is due to S. Rao Kosaraju (unpublished, 1978) and was independently described by M. Sharir in 1981. An alternative $O(n + m)$-time one-pass algorithm based on DFS low-link values is due to R. E. Tarjan (1972). A full treatment of both algorithms is given in [CLRS01]. 2-SAT, which reduces in linear time to SCC of the implication graph, is due to Aspvall, Plass, and Tarjan (1979) and is discussed in Chapter 18.

Chapter 6

Greedy Algorithms

6.1 Introduction

Many problems can be solved by making multiple choices such that at the end of these choices, we have the solution to the problem. The greedy algorithm proceeds as follows: at every step, commit irrevocably to the locally-best choice under a fixed selection rule, and never revisit past choices. The algorithm typically starts with the trivial solution of size 0 or 1 and increases the size of the partial solution one step at a time until the original problem of size n is solved. For many applications, this strategy results in a global optimum or a provably near-optimal solution.

A greedy algorithm is cheaper than exhaustive search or dynamic programming precisely because it never backtracks, but this restriction is also its weakness: committing too early can miss the global optimum. Part of the task of this chapter is to identify problems and greedy rules that *provably* yield an optimum, and to prove correctness using the **exchange argument**: take any optimal solution O that disagrees with the greedy solution S_G, identify the earliest point of disagreement, and show that swapping S_G's choice into O produces a solution at least as good; iterating transforms O into S_G without loss.

This chapter is organized as follows. Section 6.2 describes a greedy algorithm for the interval scheduling problem. In this problem, we schedule a maximum number of jobs on a processor. Section 6.3 describes a greedy algorithm for the problem of scheduling intervals using as few rooms as possible so that no two overlapping intervals are assigned to the same room. Section 6.4 discusses the problem of scheduling n jobs so that the maximum lateness of jobs is minimized. Section 6.5 discusses the construction of a Huffman tree that minimizes the expected length of the binary code for a symbol. Section 6.6 discusses the fractional knapsack problem, where a greedy strategy based on value-to-weight ratio yields an optimal solution. Section 6.7 describes the LLP versions of these algorithms.

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

6.2 Interval Scheduling Problem

We start with a simple greedy algorithm. Suppose that we have n intervals with start times s_i and finish times f_i for jobs $i = 1..n$, where $s_i < f_i$. We assume that activity i occurs during the half-open interval $[s_i, f_i)$. Two intervals i and j are *compatible* if $f_i \leq s_j$ or $f_j \leq s_i$. Our goal is to select a maximum-sized subset of mutually compatible intervals. We assume that the intervals are sorted by their finish times. We assume that jobs with the same finish times are sorted according to their start times. Furthermore, no two intervals have identical start and finish times.

The key idea is to leave as much room as possible for future activities. By always selecting the activity that finishes earliest, we ensure that the remaining time interval is as large as possible for accommodating other compatible activities. Any activity that finishes later would only reduce the available space for subsequent choices. Thus, choosing the earliest finishing activity is the safest local decision that preserves maximum flexibility for the rest of the schedule.

Let us begin with a sequential algorithm for this problem, shown in Algorithm Interval. Let G be the set of mutually compatible intervals represented as a Boolean array such that $G[i]$ is 1 iff the activity i is in the subset of mutually compatible intervals that can be selected. It is clear that the first activity (with the least finishing time) is always in G. We now traverse over all intervals as follows. We use the variable i to denote the current activity under consideration and the variable *last* as the last activity that was included in G. We initialize *last* to 1. The current activity is included in G if and only if its start time is greater than or equal to the finish time of the last activity. If it is, our current activity becomes the last activity, and we explore the next activity.

Algorithm Interval: Sequential interval scheduling

input: s: array[1..n] of int // start times
f: array[1..n] of int // finish times
output: G: array[1..n] of 0..1, initially 0 // selected intervals

```
1 G[1] := 1;                                  // first interval always selected
2 int last := 1;                              // index of last selected interval
3 for int i := 2 to n do
4 |   if s[i] ≥ f[last] then
5 |   |   G[i] := 1;                                  // interval i is compatible
6 |   |   last := i
7 |   end
8 end
```

The main loop visits each interval once and performs $O(1)$ work per visit, so the algorithm runs in $O(n)$ time when the intervals are already sorted by finish time. If the intervals are unsorted, the initial sort dominates and the total cost is $O(n \log n)$. No extra space beyond the output array is needed, giving $O(n)$ space.

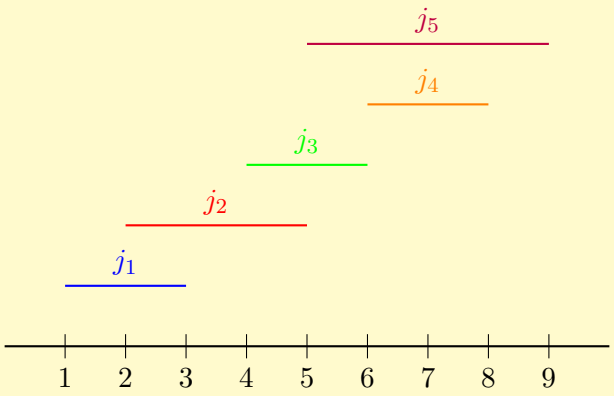
Theorem 6.1 (Interval Scheduling optimality) *Selecting activities in increasing order of finish times yields a maximum-sized set of mutually compatible activities.*

Proof: We prove the theorem using the **exchange argument**. Let S_G be the set of activities chosen by the greedy algorithm, which selects the activity with the earliest finish time first. Let O be an optimal solution with the maximum number of activities. The greedy algorithm selects the activity with the earliest finish time, say A_1. Suppose O selects a different first activity A_k. Since activities are sorted by finish time, $f_1 \leq f_k$. Replacing A_k with A_1 in O still allows the remaining selections in O to proceed without reducing the total count. By repeatedly applying this argument for the next selected activity, we replace each activity in O with the corresponding activity of S_G without decreasing the number of selected activities. After at most n exchanges, O is transformed into S_G. Since O was an optimal solution and S_G has the same size, S_G is also optimal.

■

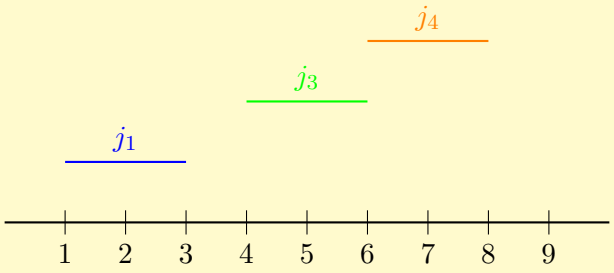
Example 6.2 Suppose that we are given five jobs, each with a start time and an end time. The goal is to select the maximum number of non-overlapping jobs.

Job	Start Time	End Time
j_1	1	3
j_2	2	5
j_3	4	6
j_4	6	8
j_5	5	9



An optimal schedule selects the maximum number of non-overlapping jobs. The optimal solution is:

$$\{j_1, j_3, j_4\}$$



6.3 Interval Partition Problem

Consider a university that offers multiple courses for students. Every course i has a fixed slot $[s_i, f_i)$ where s_i indicates the start time of the lecture in course i and f_i indicates the finish time for that lecture. There are n courses in total. The university wants to use as few classrooms as possible for lectures. Our task is to assign a room to each course so that if two lectures overlap, then they must be assigned different rooms. Since each lecture is modeled as an interval, this problem is called the *interval partition problem*.

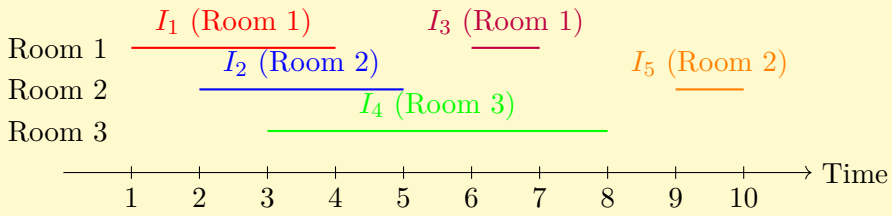
For this problem, we first look at the lower bound on the number of rooms required. Suppose that there are m overlapping lectures. Then it is clear that any room assignment

must use at least m rooms.

At any moment, the number of rooms required is determined by how many intervals overlap at that time. Therefore, the goal is simply to reuse rooms whenever possible. When a new interval begins, we assign it to the room that becomes free the earliest. If no room is free, only then do we open a new one. This greedy rule ensures that we never use more rooms than necessary, since a new room is created only when all existing rooms are simultaneously occupied.

We now show a simple greedy algorithm that achieves the lower bound on the number of rooms. We sort the courses based on the start times of the lectures. We assign courses to rooms one at a time. The first course is assigned the room 1. Now suppose that k courses have been assigned their rooms. We assign the course $k+1$ as follows: assign it to the least numbered room that is available at the start time of the lecture. It is clear that a course is assigned a *new* room only if there are overlapping lectures for all smaller-numbered rooms. Alternatively, the intuition is as follows. When an interval begins, we already know every interval that could possibly conflict with it. We get a new room only if it is necessary. Thus, our greedy algorithm achieves the minimum number of rooms assigned.

Example 6.3 Given the intervals $I_1 = [1, 4)$, $I_2 = [2, 5)$, $I_3 = [6, 7)$, $I_4 = [3, 8)$, $I_5 = [9, 10)$, sorted by start time they become $[1, 4), [2, 5), [3, 8), [6, 7), [9, 10)$. The greedy assignment is:



The minimum number of rooms required is equal to the maximum number of overlapping intervals, which is 3.

Theorem 6.4 (Interval Partitioning optimality) *For the Interval Partitioning Problem, assigning intervals to the first available room in order of their start times produces an optimal solution, that is, it minimizes the number of rooms required.*

Proof: The greedy algorithm sorts intervals by start time and assigns each to the first available room, opening a new room only when all existing rooms are busy.

Let d be the maximum number of intervals overlapping at any single time. Any valid assignment needs at least d rooms, since d mutually overlapping intervals must go in d distinct rooms.

Conversely, the greedy algorithm never opens more than d rooms: it opens a new room only when every existing room holds an interval overlapping the current one, so d intervals overlap at that moment, and the new room is at most the d-th.

Therefore the greedy algorithm uses exactly d rooms, which is optimal.

■

Algorithm IntervalPartition is a greedy algorithm for the Interval Partition problem.

Algorithm IntervalPartition: Sequential interval partition

input: s: array[1..n] of int // start times (sorted: $s[1] \leq s[2] \leq \cdots \leq s[n]$)
f: array[1..n] of int // finish times
output: *room*: array[1..n] of int // room assigned to each interval

var: Q: min-priority queue of $(finishTime, roomNumber)$, initially empty
numRooms: int, initially 0

```
1  for int i := 1 to n do
2  |  if Q is not empty and Q.min().finishTime ≤ s[i] then
3  |  |  r := Q.extract-min().roomNumber;          // reuse earliest-free room
4  |  else
5  |  |  numRooms := numRooms + 1;                  // allocate a new room
6  |  |  r := numRooms
7  |  end
8  |  room[i] := r;
9  |  Q.insert(f[i], r);                          // room r busy until f[i]
10 end
11 return room, numRooms
```

Now, let us determine the time complexity of our algorithm. The sorting of intervals based on the start times can be done in $O(n \log n)$ time. Suppose that we keep *available* rooms in a heap ordered by the $(finishTime, roomNumber)$. Whenever we need a room for a lecture, we simply remove the minimum available in the heap. This heap operation takes $O(\log n)$ time. Thus, we take $O(n \log n)$ time for the entire algorithm.

6.4 Minimizing Maximum Lateness of Jobs

Suppose that we have n jobs such that each job j takes time t_j to execute and should ideally be completed before its deadline d_j. Our task is to schedule these jobs on a single processor. Suppose that the job j is started at time s_j. Then, the job ends at time $s_j + t_j$. If $s_j + t_j$ is less than or equal to the deadline d_j, then this job is not late. Otherwise, this job has *lateness* equal to $s_j + t_j - d_j$. Our task is to schedule these jobs so that the maximum lateness is minimized.

Jobs with earlier deadlines are more urgent because delaying them risks incurring large lateness that cannot be corrected later. By scheduling jobs in increasing order of deadlines, we ensure that the most time-sensitive jobs are completed as early as possible. Delaying a job with a later deadline is less risky, since it has more slack. Thus, prioritizing earliest deadlines balances the schedule to minimize the worst-case lateness.

Let us start with a sequential algorithm. Should we consider these jobs in the order of their processing times, their *slack* times $(d_j - t_j)$, or their deadlines d_j? If a job has an earlier

deadline, delaying it is always more dangerous than delaying a job with a later deadline. Thus, for this problem, we should consider jobs in the order of their deadlines.

Algorithm MinMaxLateness: Sequential minimum-lateness scheduling

input: t: array$[1 \ldots n]$ of int // processing times, sorted by deadline
d: array$[1 \ldots n]$ of int // deadlines in non-decreasing order
output: G: array$[1 \ldots n]$ of int, initially 0 // start time of each job

```
1 int last := 0;
2 int lateness := 0;
3 for int i := 1 to n do
4 |   int finish := last + t[i];
5 |   G[i] := last;
6 |   last := finish;
7 |   if (finish > d[i]) then lateness := max(lateness, finish − d[i]);
8 end
9 return G, lateness;
```

Algorithm MinMaxLateness processes jobs in the order of increasing deadlines. At each step, $last$ holds the current end-of-schedule time, $G[i]$ records the starting time of job i, and $finish$ is the moment it completes. The loop performs $O(1)$ work per job, so the algorithm runs in $O(n)$ time once the jobs are sorted by deadline. If the jobs are not pre-sorted, the initial sort dominates and the total time is $O(n \log n)$. The algorithm uses $O(n)$ space for the output array G.

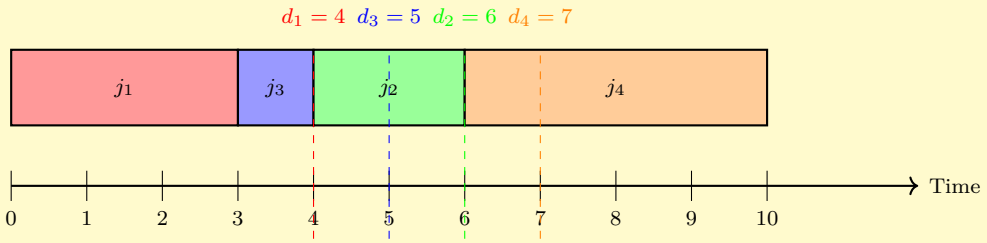
Example 6.5 Consider the following example.

Job j_i	**Processing Time** t_i	**Deadline** d_i
j_1	3	4
j_2	2	6
j_3	1	5
j_4	4	7

Sorting by deadline gives the order j_1, j_3, j_2, j_4. Tracing the algorithm row by row:

i	job	$t[i]$	$d[i]$	$G[i]$ (= *last* on entry)	*finish*	*lateness* after
1	j_1	3	4	0	3	0
2	j_3	1	5	3	4	0
3	j_2	2	6	4	6	0
4	j_4	4	7	6	10	3

Only job j_4 runs past its deadline, contributing lateness $10 - 7 = 3$. The resulting schedule using Earliest Deadline First (EDF) is shown next.



The job j_2 finishes at time 6 and has a deadline of 6. The job j_4 finishes at time 10 but has a deadline of 7. Using earliest deadline first, this schedule minimizes *maximum lateness*, which is 3 units.

Theorem 6.6 (EDF optimality) *For the problem of scheduling n jobs with known processing times and deadlines on a single machine to minimize the maximum lateness, the Earliest Deadline First (EDF) algorithm produces an optimal solution.*

Proof:

The EDF algorithm schedules jobs in increasing order of their deadlines. Let S_G be the schedule produced by the EDF algorithm, and let O be an optimal schedule that minimizes maximum lateness. Suppose for contradiction that O is different from S_G and has a different order of jobs while achieving a smaller maximum lateness.

Consider the first position in which O and S_G differ. Let job j be scheduled earlier in O than job i, while S_G places job i before job j. Since EDF sorts jobs by increasing deadlines, we have $d_i \leq d_j$. Since both schedules are contiguous, swapping i and j in O does not delay the completion time of any job before i, nor does it improve the lateness of any job. Instead, it ensures that the job with an earlier deadline finishes sooner, which prevents an increase

in lateness. By iterative application of this argument, we can transform O into S_G without increasing the maximum lateness. Since O was assumed to be optimal, it follows that S_G is also optimal. Therefore, EDF produces an optimal schedule.

■

6.5 Huffman Tree

The goal of the Huffman tree problem is to find an efficient encoding of a set of symbols. We are given n symbols. For each symbol i, the value $p[i]$ gives the probability that it appears in the given text, with $\sum_i p[i] = 1$. Our goal is to design a *prefix* code such that the given text can be translated to a binary string from which the original text can be recovered. We will create a binary tree so that the leaves correspond to the symbols and the path from the root to the leaf corresponds to the *code* for the symbol. Note that only the leaves of the tree correspond to symbols. Also note that leaves may be at different levels, and therefore this method returns a variable-length code for symbols rather than a fixed-length code.

In an optimal prefix code, symbols with higher probability should have shorter codewords, while rare symbols can tolerate longer ones. The greedy strategy builds the code tree from the bottom up by repeatedly merging the two least probable symbols. This ensures that the least important symbols are pushed deeper in the tree, where longer codewords are assigned. By making the locally optimal choice of combining the least probable symbols at each step, we construct a globally optimal encoding.

Given n symbols with their probabilities of occurrence, the Huffman coding algorithm constructs an optimal prefix code using the following greedy approach. We first insert all symbols into a min-priority queue keyed by probability. As long as the queue contains more than one node, we remove the two nodes with the smallest probabilities, merge them into a new node whose probability is the sum of the two, and insert this merged node back into the queue. When only one node remains, it is the root of the Huffman tree. Finally, we obtain the code for each symbol by traversing the tree from the root, labeling the left branch with 0 and the right branch with 1.

Algorithm HuffmanCoding: Huffman coding

input: set of symbols $S = \{s_1, s_2, \ldots, s_n\}$ and probabilities $p(s_1), p(s_2), \ldots, p(s_n)$ with $\sum_i p(s_i) = 1$

output: Huffman tree T

```
1  Q := priority queue (min-heap) of nodes initially null ;
2  for each s_i ∈ S do
3  |  Create a leaf node n_i with probability p(s_i) and symbol s_i;
4  |  insert n_i into Q;
5  end
6  for i = 1 to n − 1 do
7  |  x := extract-min(Q) ;              // remove node with smallest probability
8  |  y := extract-min(Q) ;                                  // remove next smallest
9  |  z := new internal node with children x and y;
10 |  p(z) := p(x) + p(y) ;                                    // sum probabilities
11 |  insert z into Q;
12 end
13 T := extract-min(Q) ;                                      // final tree (root)
14 return T
```

Example 6.7 Consider six symbols with probabilities that exercise merges of internal nodes with each other:

Symbol	A	B	C	D	E	F
Probability	0.45	0.13	0.12	0.16	0.09	0.05

The priority queue evolves as follows, with probabilities shown in non-decreasing order at each step and internal nodes denoted by the sum of their children:

Step	Extracted (smallest two)	Queue after insertion
0	—	.05, .09, .12, .13, .16, .45
1	.05, .09 → .14	.12, .13, .14, .16, .45
2	.12, .13 → .25	.14, .16, .25, .45
3	.14, .16 → .30	.25, .30, .45
4	.25, .30 → .55	.45, .55
5	.45, .55 → 1.0	1.0

The resulting Huffman codes are:

Symbol	Probability	Huffman Code
A	0.45	0
C	0.12	100
B	0.13	101
F	0.05	1100
E	0.09	1101
D	0.16	111

The expected code length is $0.45 \cdot 1 + (0.12 + 0.13) \cdot 3 + (0.05 + 0.09) \cdot 4 + 0.16 \cdot 3 = 2.24$ bits per symbol, compared with 3 bits per symbol for a fixed-length code, a saving of about 25%.

The time and space complexity of the Huffman coding algorithm depend on the use of a min-heap for the priority queue Q. For the time complexity, building Q from the n leaf nodes costs $O(n)$. The main loop runs $n-1$ times, and each iteration performs two extract-min operations and one insertion, all at cost $O(\log n)$; the loop therefore contributes $O(n \log n)$. The final extract-min is $O(\log n)$, which is absorbed, giving an overall running time of $O(n \log n)$. For the space complexity, the priority queue holds at most n nodes at any time, and the constructed tree has n leaves and $n-1$ internal nodes for a total of $2n-1$ nodes; both contribute $O(n)$, so the algorithm uses $O(n)$ space, where n is the number of symbols.

Theorem 6.8 (Huffman optimality) *Algorithm HuffmanCoding constructs a prefix code of minimum expected length $\sum_i p_i \ell_i$, where ℓ_i is the length of the codeword for symbol i.*

Proof: By induction on n. For $n = 2$, both symbols get a one-bit code, which is trivially optimal.

For $n > 2$, let u and v be the two symbols with the smallest probabilities.

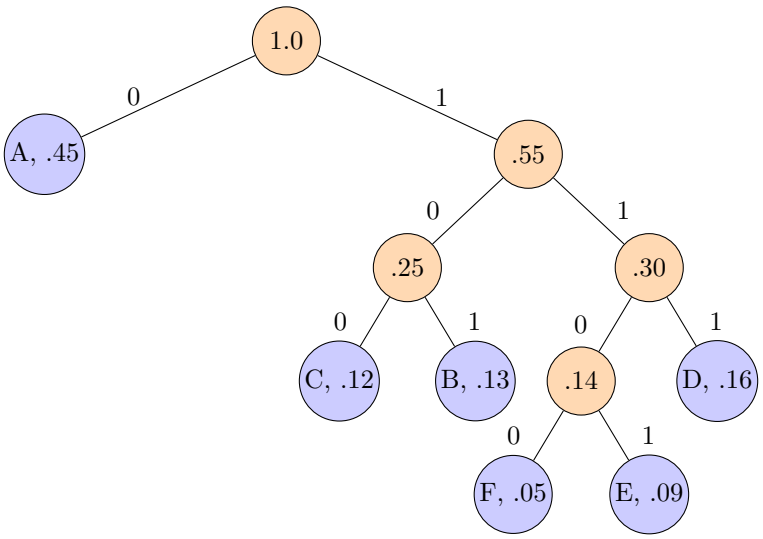


Figure 6.1: Huffman tree for symbols A, B, C, D, E, F with probabilities $0.45, 0.13, 0.12, 0.16, 0.09, 0.05$. Leaves (blue) carry the source symbols; internal nodes (orange) carry the summed probabilities created by merges.

Greedy-choice property. In any optimal tree, there exist two sibling leaves at the deepest level (since the tree is full). Because u and v have the smallest probabilities, we can swap them into those deepest positions without increasing cost — moving a less probable symbol deeper and a more probable symbol shallower can only decrease or maintain the expected length. So there is an optimal tree in which u and v are siblings at the deepest level.

Optimal-substructure property. In such a tree, replace u and v by a single leaf z with probability $p_u + p_v$. This reduces the problem to $n - 1$ symbols, and the expected code length changes by exactly $p_u + p_v$ — a constant that does not depend on the tree structure. So an optimal tree for the reduced problem yields an optimal tree for the original problem when we expand z back into u and v.

By the inductive hypothesis, Huffman's algorithm produces an optimal tree for the $n - 1$ symbol problem. Expanding z back gives an optimal tree for all n symbols.

■

6.6 Fractional Knapsack

The knapsack problem is a natural test case for the reach of greedy methods. We are given n items with values $v_1, \ldots, v_n$ and weights $w_1, \ldots, w_n$, and a knapsack capacity W. In the 0/1 *knapsack problem*, we must choose a subset $S \subseteq [n]$ with $\sum_{i \in S} w_i \leq W$ that maximizes $\sum_{i \in S} v_i$; here no fixed greedy rule is optimal. For capacity $W = 5$ and items $\{(v{=}10, w{=}5), (v{=}7, w{=}3), (v{=}6, w{=}2)\}$, the rule "take the largest value first" picks item 1 alone, for value 10, while items 2 and 3 together give 13. The 0/1 knapsack problem needs dynamic programming (Chapter 10) and in general is NP-hard.

In the *fractional knapsack problem*, items are divisible: we may take a fraction $x_i \in [0, 1]$ of each item, paying weight $x_i w_i$ and earning value $x_i v_i$. The capacity constraint becomes $\sum_i x_i w_i \leq W$. Divisibility restores the power of greed.

To maximize value within a limited capacity, we should prioritize items that provide the greatest value per unit weight. Sorting items by their value-to-weight ratio ensures that each unit of capacity is used as efficiently as possible. We take as much as possible of the highest-ratio item before moving to the next. Since items are divisible, any deviation from this rule would replace higher-value density with lower-value density, leading to a worse total value.

Algorithm FractionalKnapsack: Greedy algorithm for the fractional knapsack problem

input: $v[1..n]$, $w[1..n]$: reals; W: real capacity
output: $x[1..n]$: fractional selections, initially 0

```
1  Sort items in non-increasing order of the ratio v[i]/w[i];
2  real rem := W;
3  for i := 1 to n do
4  |  if w[i] ≤ rem then
5  |  |  x[i] := 1;
6  |  |  rem := rem − w[i];
7  |  else
8  |  |  x[i] := rem/w[i];                    // take the remaining fraction
9  |  |  break;
10 |  end
11 end
```

Theorem 6.9 (Fractional Knapsack optimality) *Algorithm FractionalKnapsack returns a selection of maximum total value for the fractional knapsack problem.*

Proof: Assume items are indexed so that $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$. The greedy algorithm fills items in this order, taking as much of each item as the remaining capacity allows.

Let O be any optimal solution. If O differs from the greedy solution, let j be the first item where they disagree — the greedy solution takes more of item j than O does. Since both solutions use the same total capacity, O must compensate by taking more of some later item $j' > j$. We can transfer weight from item j' to item j in O without losing value, because $v_j/w_j \geq v_{j'}/w_{j'}$. Repeating this process eliminates all disagreements without reducing the total value, transforming O into the greedy solution. Hence the greedy solution is optimal.

■

Example 6.10 Consider four items and a knapsack of capacity $W = 10$:

Item i	Value v_i	Weight w_i	Ratio v_i/w_i
1	10	2	5
2	12	3	4
3	6	2	3
4	8	4	2

The items are already sorted by non-increasing value-per-weight ratio. Tracing the algorithm:

i	w_i	rem (before)	x_i	Value added
1	2	10	1	10
2	3	8	1	12
3	2	5	1	6
4	4	3	3/4	6

The greedy algorithm takes items 1, 2, and 3 in full and three-quarters of item 4. The total weight used is $2 + 3 + 2 + 3 = 10 = W$, and the total value is $10 + 12 + 6 + 6 = 34$.

For contrast, refusing to break item 4 and taking items $1, 2, 3$ alone gives value 28; taking item 4 whole instead of item 3 yields weight $2+3+4 = 9$ and value $10+12+8 = 30$. Both alternatives are worse, confirming the optimality of the greedy fractional selection on this instance.

The sort dominates the running time: $O(n \log n)$ sequentially, with $O(n)$ additional work for the loop. In parbook, observe that the prefix sums of w_i along the sorted order determine, in parallel, the cut-off index k where the knapsack becomes full; a parallel prefix-sum computes them in $O(\log n)$ time and $O(n)$ work, and the fractional coefficient x_k is then read off in $O(1)$ time. The total parallel time is $O(\log n)$ after the sort, matching the bound for Minimizing Maximum Lateness.

6.7 LLP Perspective

In this section, we provide LLP versions of greedy algorithms.

LLP: Interval Scheduling Algorithm

In this subsection, $G[j]$ is a Boolean indicating whether interval j is selected.

We are given n jobs sorted by their finish times. Jobs with the same finish times are sorted according to their start times. We are interested in finding a maximum-sized subset of $[n]$ such that each pair of jobs is mutually compatible. We define B on a subset of jobs to be true if the subset is a maximum-sized subset of mutually compatible jobs. For example, consider the set of jobs as $\{[1, 3), [2, 4), [5, 8), [6, 9)\}$. It can be verified that there is no subset of size three that has mutually compatible jobs. However, there are multiple subsets of size two that are mutually compatible. For example, $\{[1, 3), [5, 8)\}$ is one such subset. Another example is

$\{[2,4),[5,8)\}$. The predicate B as defined is not lattice-linear in the lattice of all subsets. It is not closed under the meet operation. The meet of the two examples is the subset $\{[5,8)\}$, which is not a maximum-sized subset that has mutually compatible jobs. We redefine B to be the lexicographically least maximum-sized subset of mutually compatible jobs. Now, simply by the definition, B is true exactly on one subset. Hence, B is lattice-linear: for any state G that does not satisfy B, there exists an index j such that every state above G that satisfies B has a different value of $G[j]$. Because the unique satisfying state G^* differs from G in at least one coordinate j, that coordinate is a forbidden index — advancing j toward $G^*[j]$ makes progress and cannot overshoot the unique target.

We determine the lexicographically least maximum-sized subset of mutually compatible jobs as follows. We maintain a linked list of all the jobs under consideration. These jobs are in the order of their finish times. The algorithm will maintain a Boolean $G[j]$ for each job j. The variable $G[j]$ is initially false and is set to true only if it is guaranteed to be in the maximal set of jobs that the algorithm chooses. We call a job j *fixed* if $G[j]$ is true. The *init* command always fixes the first job. We now make the following observations.

1. If any job j has its start time greater than or equal to the finish time of the previous job, then that job is always included in the final set. Thus, the job j can be fixed.

2. For any job, if the previous job in the linked list is fixed and the start time of the current job is less than the finish time of the previous job, then this job can be deleted.

It follows from the above two rules that eventually all jobs will be fixed or deleted. The algorithm terminates when all jobs are fixed. The set of all fixed jobs is a maximum-sized set of jobs that can be completed. Furthermore, it is the lexicographically least such set.

The LLP algorithm for the activity selection problem is shown in Algorithm LLP-IntervalScheduling.

Algorithm LLP-IntervalScheduling: LLP program for the interval scheduling problem

---

```
// Assume that all jobs are in a doubly linked list
// The variable prev denotes the previous node in the linked list
```
input: $s[j]$: int, $f[j]$: int
output: $G[j]$: boolean, initially if $(j = 1)$ then $G[j] := true$ else $G[j] := false$

1 **forbidden**(j): $\neg G[j] \wedge f[prev] \leq s[j]$
2 $\quad\Rightarrow$ **advance**: $G[j] := true$;
3 **forbidden**(j): $(j > 1) \wedge G[prev] \wedge s[j] < f[prev]$
4 $\quad\Rightarrow$ **advance**: $next[prev[j]] := next[j]$; $prev[next[j]] := prev[j]$; `// delete from linked list`

For example, consider the following four jobs: $job_1 = [1,4), job_2 = [2,5), job_3 = [4,5), job_4 = [5,7)$. The first job gets fixed because the first job is always fixed. The fourth job also gets fixed because its starting point is at least as large as the finishing time of the previous job. Since the first job is fixed and the second job starts before the first job is finished, it is deleted from the list. Now, the third job is also fixed because it starts after the first job is finished.

Each job in the linked list is either fixed by the first advance rule or deleted by the second, and both actions are $O(1)$ because the list pointers let us reach *prev* directly. Every job is touched at most twice (once to be fixed, possibly once to be deleted), so the main loop runs in $O(n)$ time once the jobs are sorted by finish time; the overall cost is therefore $O(n \log n)$ including the sort, with $O(n)$ space for the list.

LLP: Interval Partition Algorithm

In this subsection, $G[j]$ is the room number assigned to course j.

We now explore an LLP version of the algorithm for Interval Partition. Let $G[i]$ denote the room assigned to course i (where courses are sorted based on start times). We initialize $G[i]$ to 1 for all i. This room assignment is feasible iff there are no overlapping intervals. We now define the feasibility of the room assignment for any set of intervals. A room assignment is feasible if for any pair of intervals i, j whenever they overlap, they are in different rooms. Formally, let B be defined as

$$\forall i, j \in [n] : (i \neq j) \wedge (s_j < f_i) \wedge (s_i < f_j) \Rightarrow G[i] \neq G[j]$$

It can again be observed that B is not lattice-linear. For example, if there are only two overlapping activities, then both G vectors, $[1, 2]$ and $[2, 1]$, satisfy B. However, their minimum $[1, 1]$ does not satisfy B. We apply the same trick as in the previous problem. Instead of finding any feasible room assignment, we find the lexicographically least room assignment. Thus, B is redefined as

$$\forall j : G[j] = \min\{r \mid \forall i < j : (s[j] < f[i]) \Rightarrow (G[i] \neq r)\}$$

Thus, $G[j]$ is assigned the least numbered free room available when the course j starts. With this new definition, it is clear that B holds only for one possible G.

The predicate B may not be true for the initial assignment when there are overlapping intervals. Our interest is in finding the least assignment to the rooms that makes the predicate true. We consider the lattice of n-dimensional vectors of natural numbers. The bottom element of this lattice is the vector with all the ones. We use $fixed[i]$ to denote that the course i has been assigned a room that will not be changed. Let $pre(i)$ denote all courses numbered less than i that start prior to course i or at the same time as course i and finish later than s_i.

We say that index j is forbidden if all courses in $pre(j)$ are $fixed$ and $\neg fixed[j]$. Whenever index j is forbidden, it can be advanced as follows. Let r be the least room that has not been assigned to any overlapping course prior to j. Then $G[j]$ is set to r and $fixed[j]$ is set to true.

Algorithm LLP-IntervalPartition: LLP program for the interval partition problem

---

input: $s[j]$: int, $f[j]$: int; $pre[j]$: set of $1..n$ initially
$pre[j] = \{i \mid (i < j) \wedge (s[i] \leq s[j]) \wedge (s[j] < f[i])\}$
output: $G[j]$: int initially 1; $fixed[j]$: boolean initially false

1 **forbidden**(j): $(\forall i \in pre(j) : fixed(i)) \wedge \neg fixed(j)$
2 $\Rightarrow$ **advance**: $G[j] := \min\{r \mid \forall k \in pre(j) : r \neq G[k]\}$; $fixed[j] := true$;

As before, we assume that the intervals are provided to us, sorted by their start times. Each index j keeps track of the number of intervals in $pre(j)$ that have been fixed, so checking whether j is forbidden is $O(1)$. Finding the least free room in the advance step uses a heap keyed by the occupied rooms at time $s[j]$; each heap operation costs $O(\log n)$. There are exactly n advances, one per course, so the main loop contributes $O(n \log n)$ time, which dominates the initial sort and gives an overall sequential running time of $O(n \log n)$. The space complexity is $O(n)$: the arrays G and $fixed$ use $O(n)$ each, and the heap stores at most n rooms.

LLP: Minimizing Maximum Lateness of Jobs

In this subsection, $G[j]$ is the start time of job j.

We now consider the LLP version of computing a job schedule that minimizes the maximum lateness of jobs. The greedy approach is to schedule jobs in order of increasing deadline. For simplicity, assume that all deadlines are distinct and that the jobs are indexed so that $d[1] < d[2] < \cdots < d[n]$.

Let $G[j]$ denote the starting time of job j, initially 0. The lattice is $\mathbb{N}^n$ with componentwise order, and the bottom element is the all-zeros vector. The target vector $G^\star$ is characterized by

$$B \equiv \forall j : G[j] = \sum_{i<j} t[i].$$

Because $G^\star$ is uniquely determined, B is lattice-linear on its own.

We take the forbidden condition to be the direct value comparison

$$\text{Forbidden}(j) \equiv G[j] < \sum_{i<j} t[i],$$

and the advance step simply raises $G[j]$ to that target value. Initially every $j \geq 2$ is forbidden (since the right-hand side is positive while $G[j] = 0$), so all $n - 1$ indices can advance in the same parallel round.

Algorithm LLP-MinMaxLateness: LLP program for the minimizing maximum lateness problem

---

input: $t[j]$: int, processing time of job j; $d[j]$: int, deadline of job j (jobs indexed in non-decreasing deadline order)

output: $G[j]$: int, initially 0, the starting time of job j; *lateness*: int, initially 0

1 **forbidden**(j): $G[j] < \sum_{i<j} t[i]$

2 $\quad \Rightarrow$ **advance**: $G[j] := \sum_{i<j} t[i]$;

3 $lateness := \max_j \max(0,\ G[j] + t[j] - d[j])$;

Example 6.11 Continuing Example 6.5, we trace MinMaxLateLLP on the same four jobs with processing times $t = (3, 2, 1, 4)$ and deadlines $d = (4, 6, 5, 7)$. Reindex them in deadline order: $j_1 \rightsquigarrow 1$, $j_3 \rightsquigarrow 2$, $j_2 \rightsquigarrow 3$, $j_4 \rightsquigarrow 4$, so the deadline-sorted processing times are $(3, 1, 2, 4)$ and deadlines are $(4, 5, 6, 7)$. The targets $\sum_{i<j} t[i]$ are $(0, 3, 4, 6)$. Initially $G = (0, 0, 0, 0)$ and indices $2, 3, 4$ are all forbidden. One parallel round advances $G[2] := 3$, $G[3] := 4$, $G[4] := 6$ simultaneously, after which no index is forbidden. The final *lateness* reduction reads $G + t - d = (-1, -1, 0, 3)$, giving $lateness = 3$, attained by j_4.

round	forbidden set	advances applied	G after
0	$\{2, 3, 4\}$	—	$(0, 0, 0, 0)$
1	$\emptyset$	$G[2] := 3$, $G[3] := 4$, $G[4] := 6$ (parallel)	$(0, 3, 4, 6)$

This matches the schedule drawn in Example 6.5: the jobs start at times $0, 3, 4, 6$ in deadline order, and the maximum lateness is 3 units.

Sequentially, computing each target $\sum_{i<j} t[i]$ takes $O(n)$ via a left-to-right scan, and the lateness reduction is another $O(n)$, so the total time is $O(n)$ once the deadlines are sorted and $O(n \log n)$ otherwise. The arrays G and the scalar *lateness* use $O(n)$ space.

LLP: Fractional Knapsack

In this subsection, $G[j] \in [0, 1]$ is the fraction of item j taken.

We now give an LLP version of the fractional knapsack algorithm. Assume the items are indexed in non-increasing order of v_j/w_j. Let $G[j] \in [0, 1]$ denote the fraction of item j included in the knapsack, initially 0. The lattice is $[0, 1]^n$ with componentwise order and bottom the all-zeros vector.

Let $S_j = \sum_{k \le j} w[k]$ denote the cumulative weight up through item j, and define the target fraction

$$G^\star[j] \;=\; \begin{cases} 1 & \text{if } S_j \le W, \\ (W - S_{j-1})/w[j] & \text{if } S_{j-1} < W < S_j, \\ 0 & \text{if } S_{j-1} \ge W. \end{cases}$$

The predicate $B \equiv (\forall j : G[j] = G^\star[j])$ is satisfied by a unique vector $G^\star$ and is therefore lattice-linear on its own. We take the forbidden condition to be the direct value comparison

$$\text{forbidden}(j) \equiv G[j] < G^\star[j],$$

and the advance step raises $G[j]$ to $G^\star[j]$. Initially every j with $G^\star[j] > 0$ is forbidden, so all such indices can advance in a single parallel round.

Sequentially, computing the prefix sums S_j takes $O(n)$ time once the items are sorted, and each target $G^\star[j]$ is obtained in $O(1)$ from S_{j-1}, S_j. The total running time is therefore $O(n \log n)$ (dominated by the sort), with $O(n)$ space.

6.8 Constrained Greedy Algorithms

The greedy algorithms above made each local choice (earliest finish time, earliest available room, earliest deadline, ...) without external constraints. Real applications often add side

Algorithm LLP-FractionalKnapsack: LLP program for fractional knapsack

input: $v[1..n]$, $w[1..n]$: reals (sorted so that $v[j]/w[j]$ is non-increasing); W: capacity
output: $G[j]$: real in $[0, 1]$, initially 0

`// Let` $S_j = \sum_{k \le j} w[k]$ `and` $G^\star[j]$ `be the target value defined above.`
1 **forbidden**(j): $G[j] < G^\star[j]$
2 $\Rightarrow$ **advance**: $G[j] := G^\star[j]$;

constraints — "this booking must be honored", "this room cannot be used for this lecture", "this job cannot start before its parts arrive". Many such constraints are lattice-linear in the same coordinate G used by the LLP formulation, so they slot in cleanly as new conjuncts of the forbidden predicate. We illustrate with three of the chapter's problems; the other two are deferred to the exercises.

Each constrained variant below pairs an existing chapter algorithm with a small constraint program on the same lattice G. The LLP loop is then run on the conjunction of the two forbidden predicates: an index j is forbidden when *either* predicate forbids it, and the advance rule is the one provided by whichever predicate raised the flag.

Interval Scheduling with a Mandatory Subset

A subset $S \subseteq [n]$ of intervals must appear in the selected set (e.g., VIP bookings). Infeasibility occurs precisely when two mandatory intervals overlap. The constraint program is

Algorithm Mandatory: Mandatory-subset constraint program

input: $S \subseteq [n]$: mandatory subset; $G[1..n]$: Boolean

1 **forbidden**(j): $j \in S \wedge \neg G[j]$
2 $\Rightarrow$ **advance**: **if** j overlaps some k with $G[k]$ **then return** "infeasible" **else** $G[j] := \mathit{true}$;

The constrained scheduler runs the LLP loop with the conjunction of the forbidden predicates of Algorithm LLP-IntervalScheduling and Algorithm MANDATORY: j is forbidden when *either* the interval-scheduling rule or the mandatory-subset rule forbids it, and the advance step is the one contributed by the rule that raised the flag.

Interval Partitioning with Excluded Rooms

For each interval j, let R_j be the set of rooms unsuitable for j (e.g., a chemistry lecture cannot use a room without a fume hood). The algorithm always succeeds; the room count exceeds the unconstrained optimum by at most $\max_j |R_j|$. The constraint program is

Algorithm ExcludedRooms: Excluded-rooms constraint program

input: $R[1..n]$: array of room sets; $G[1..n]$: room assignments

1 **forbidden**(j): $G[j] \in R_j$
2 $\Rightarrow$ **advance**: $G[j] := \min\{r \notin R_j : r \neq G[k]\ \forall k \in \mathit{pre}(j)\}$;

The constrained partitioner runs the LLP loop with the conjunction of the forbidden predicates of Algorithm LLP-IntervalPartition and Algorithm EXCLUDEDROOMS.

Min-Max Lateness with Release Times and Precedence

Each job j has a release time r_j (cannot start before r_j) and may depend on a set $pre(j)$ of jobs (cannot start before all of them finish). Two constraint programs capture these:

Algorithm ReleaseTimes: Release-times constraint program

input: $r[1..n]$: release times; $G[1..n]$: start times

1 **forbidden**(j): $G[j] < r_j$
2 $\Rightarrow$ **advance**: $G[j] := r_j$;

Algorithm Precedences: Precedence-respecting constraint program

input: $pre(j)$ for each job j; processing times $t[1..n]$; $G[1..n]$: start times

1 **forbidden**(j): $\exists\, i \in pre(j) : G[j] < G[i] + t_i$
2 $\Rightarrow$ **advance**: $G[j] := \max_{i \in pre(j)}(G[i] + t_i)$;

The deadline-sorted lex-min schedule of Section 6.4 is recovered by running the LLP loop on the conjunction of three forbidden predicates: the chapter's lateness rule plus the two constraint rules above. Infeasibility arises iff the precedence relation contains a cycle.

Constrained variants of fractional knapsack (per-item bounds) and Huffman coding (length-limited codewords) admit similar compositions; both are explored in the chapter exercises.

6.9 When Greedy Fails

The problems we have studied share a feature that makes the greedy approach succeed: each one has an *exchange property* strong enough to carry the correctness proof. Once we identify the "right" local rule (earliest finish time, earliest available room, earliest deadline, merge the two least probable symbols, highest value-per-weight), any solution that deviates from it can be transformed into the greedy solution without losing value — that is the content of the exchange argument.

Many closely related problems lack this property, and for them greedy fails. Replacing divisibility of items with indivisibility turns Fractional Knapsack into the 0/1 Knapsack problem, which is NP-hard and has no known greedy rule that guarantees an optimum. The counterexample from the chapter introduction makes the failure concrete.

Item i	Value v_i	Weight w_i	Density v_i/w_i
1	10	5	2.00
2	7	3	2.33
3	6	2	3.00

With capacity $W = 5$, the highest-value-first rule picks item 1 (the only single item with value 10) and stops, since the remaining capacity is 0. The optimal solution instead leaves item 1 on the shelf and takes items 2 and 3 together (weight $3 + 2 = 5$, value $7 + 6 = 13$).

Replacing compatibility with a weight on activities similarly turns Interval Scheduling into Weighted Interval Scheduling, for which no fixed greedy rule is optimal and dynamic programming is required. Making the tasks non-preemptive on multiple machines changes the scheduling landscape in a way that local-choice heuristics can only approximate.

The cautionary tale, then, is that greedy is a consequence of structure, not a method of last resort: when the problem admits a sound exchange argument, greedy produces an exact optimum in low polynomial time; when it does not, even the most intuitive local rule can be arbitrarily far from optimal, and a richer technique such as dynamic programming (Chapter 10) or approximation (Chapter 14) is needed.

6.10 Summary

The greedy strategy for *Interval Scheduling* prioritizes activities with the earliest finish time to maximize the number of non-overlapping intervals, ensuring that more activities can be accommodated. In *Interval Partitioning*, assigning each interval to the earliest available room ensures minimal room usage by efficiently tracking overlapping intervals. For *Minimizing Maximum Lateness*, scheduling jobs by their earliest deadline prevents unnecessary delays, minimizing the worst-case lateness across all jobs. For *Huffman Tree Construction*, merging the least probable symbols first ensures that more probable symbols receive shorter codes, leading to an optimal encoding that minimizes the total cost of transmission. Finally, for the *Fractional Knapsack Problem*, selecting items by highest value-to-weight ratio fills the knapsack with the most valuable mix per unit weight, yielding the maximum achievable total value.

Table 6.1 summarizes all five problems considered in this chapter. In the table, n denotes the number of intervals (for interval scheduling and interval partitioning), the number of jobs (for minimizing lateness), the number of symbols (for Huffman coding), or the number of items (for fractional knapsack). The time complexity includes the time to sort the input as required by the strategy.

Problem	**Strategy**	**Time Complexity**
Interval Scheduling	Earliest finish time	$O(n \log n)$
Interval Partitioning	Earliest available room	$O(n \log n)$
Minimizing Lateness	Earliest deadline	$O(n \log n)$
Huffman Tree	Merge smallest probabilities	$O(n \log n)$
Fractional Knapsack	Highest value/weight ratio	$O(n \log n)$

Table 6.1: Summary of greedy algorithms.

6.11 Problems

1. **(Train Maintenance Scheduling)** A railway station needs to schedule maintenance for the incoming trains before their next departure. Each train has a maintenance duration and a deadline. Find the optimal order of maintenance to minimize the worst delay.

2. **(K Overlap Interval Selection)** Given n intervals and an integer k, find the maximum subset of intervals such that at most k intervals overlap at any time.

3. **(Gap Constrained Room Scheduling)** Each task has a required start time and finish time, but there must be a mandatory idle gap of at least g units between consecutive

tasks in the same room. Find the minimum number of rooms required.

4. **(Typed Machine Scheduling)** Each task has a start and finish time, but also requires a *specific type of machine*. Different machine types may be limited in number. Find the minimum number of machines needed.

5. **(Huffman Code Optimality)** Prove that the Huffman coding algorithm produces a prefix code of minimum expected length. State the key greedy-choice and optimal-substructure properties used in the proof.

6. **(Interval Point Cover)** Given n closed intervals on the real line, find the minimum number of points required so that every interval contains at least one chosen point. Give a greedy algorithm, prove its correctness, and state its time complexity.

7. **(Greedy Coin Change)** Given coin denominations $\{1, 5, 10, 25\}$ and an amount A, give a greedy algorithm that returns the minimum number of coins summing to A. Then exhibit a denomination set on which the same "largest first" greedy rule is *not* optimal.

8. **(Profitable Job Sequencing)** Given n jobs, each requiring one unit of processing time, with profit $p_i \geq 0$ and deadline $d_i \geq 1$, schedule a subset on a single machine (one job per integer time slot $1, 2, 3, \ldots$) so that every scheduled job finishes by its deadline. Maximize the total profit. Give a greedy algorithm and state its time complexity.

9. **(Optimal Merge Pattern)** Given n sorted files of sizes $s_1, s_2, \ldots, s_n$, we wish to merge them pairwise (using two-way merges) into one sorted file. The cost of merging two files of sizes a, b is $a + b$. Find a merge sequence that minimizes the total merge cost. Give a greedy algorithm and explain its connection to Huffman coding.

10. **(Activity Greedy Counterexamples)** For the standard activity selection problem (Section 6.2), the rule "earliest finish time" is optimal. Show by counterexample that each of the following rules can produce a strictly suboptimal schedule: (a) earliest start time first; (b) shortest duration first.

11. **(Bounded Fractional Knapsack)** For each item j we are given lower and upper bounds $0 \leq \alpha_j \leq \beta_j \leq 1$ on the fraction taken, in addition to the usual values v_j, weights w_j, and capacity W. The constrained predicate adds $\forall j : G[j] \in [\alpha_j, \beta_j]$ to the fractional-knapsack predicate. Give an LLP-style algorithm and state when the predicate is infeasible.

12. **(Length Limited Huffman Codes)** Given symbol probabilities $p_1, \ldots, p_n$ and a maximum codeword length L, find a prefix code that minimizes expected length subject to the constraint that no codeword exceeds L bits. Show that the constrained predicate, with $G[i] =$ depth of leaf i in the code tree, gains the conjunct $\forall i : G[i] \leq L$. Identify when the predicate is infeasible and outline an LLP-style algorithm.

13. **(Incremental Interval Scheduling)** A maximum-compatible set G^* of intervals has been computed by INTERVAL (Algorithm Interval). A new interval (s_{n+1}, f_{n+1}) arrives. Update G^* in $O(\log n)$ time, assuming intervals are stored by finish time in a balanced BST.

14. **(Constrained Interval Partition with Forbidden Rooms)** The interval-partition problem (IntervalPartition, Algorithm IntervalPartition) is extended: each interval j has a set $\text{forbid}_j \subseteq \{1, \ldots, m\}$ of rooms it cannot use. Express this as the conjunction [IntervalPartition && ForbiddenRooms], give the forbidden predicate of the second operand, and analyze the running time.

15. **(Ties in Interval Scheduling)** Several intervals may share the same finish time f_j. Algorithm Interval assumes finish times are totally ordered; what happens if they are not? Modify the algorithm to remain optimal under ties.

16. **(Approximate 0/1 Knapsack)** Recall (Section 6.6) that 0/1 knapsack is NP-hard while fractional knapsack admits a simple greedy. Construct an $(1 - \varepsilon)$-approximation for 0/1 knapsack using the fractional greedy as a building block.

17. **(Enumerate All Maximum-Compatible Sets)** Output all distinct maximum-compatible subsets of intervals.

18. **(Online Interval Scheduling)** Intervals are revealed one at a time in adversarial order. Each interval, on arrival, must be irrevocably accepted or rejected. Show that no deterministic online algorithm can achieve better than $1/n$ of the offline optimum.

6.12 Bibliographic Remarks

This chapter on greedy algorithms covers several classic problems: Interval Scheduling, Interval Partitioning, Minimizing Lateness, Huffman Tree construction, and Fractional Knapsack. The Interval Scheduling Problem and Interval Partitioning Problem are covered in [CLRS09] and [KT06]. For the Minimizing Lateness problem, scheduling jobs by earliest deadline (EDF) to minimize maximum lateness is a well-known greedy solution. The optimality of this algorithm, proven via an exchange argument, is a standard result in scheduling theory. [LRK76] provides one of the first rigorous treatments of this problem, establishing the effectiveness of the EDF rule for scheduling a single machine with deadlines. The Huffman tree construction, which builds an optimal prefix code by greedily merging the least probable symbols, is a cornerstone of data compression. Introduced by [Huf52], this algorithm's greedy nature and optimality for variable-length coding are foundational to information theory. Our sequential description mirrors the classic priority-queue implementation in $O(n \log n)$ time, as detailed in [CLRS09]. The fractional knapsack problem and the contrast with the NP-hard 0/1 variant are treated in [CLRS09, KT06].

Chapter 7

The Shortest Path Problem

7.1 Introduction

The single source shortest path (SSSP) problem has wide applications in transportation, networking, route-planning, and many other fields. It asks, given a weighted directed graph with n vertices and m edges and a distinguished *source* vertex v_0, for $cost[x]$: the minimum total weight of any directed path from v_0 to x, where the cost of a path is the sum of edge weights along it. A related and equally fundamental problem is the *all-pairs shortest-path (APSP) problem*, which asks for the matrix of shortest-path costs between every pair of vertices.

Table 7.1 summarizes the four sequential algorithms presented in this chapter.

Algorithm	**Problem**	**Neg. edges?**	**Time**
Dijkstra	SSSP	No	$O(m \log n)$
BellmanFord	SSSP	Yes	$O(nm)$
FloydWarshall	APSP	Yes	$O(n^3)$
Johnson	APSP	Yes	$O(nm \log n)$

Table 7.1: Overview of shortest-path algorithms in this chapter.

All of these sequential algorithms admit reformulations in the *lattice-linear predicate (LLP)* framework introduced in Chapter 2. In an LLP formulation, a single generic algorithm — advance every forbidden index until the predicate B holds — unifies Dijkstra's single-fix rule, BellmanFord's edge-relaxation loop, and FloydWarshall's triple-nested update, each arising as a special case of the generic advance. The LLP reformulations expose parallelism (many forbidden indices can advance concurrently).

This chapter is organized as follows. Section 7.2 describes Dijkstra's algorithm for SSSP on graphs with non-negative edge weights. Section 7.3 describes the BellmanFord algorithm, which lifts the non-negativity restriction and also detects negative-weight cycles. Section 7.4 presents the FloydWarshall algorithm for APSP in $O(n^3)$ time via the intermediate-vertex recurrence. Section 7.5 develops Johnson's algorithm, which uses the BellmanFord result to reweight edges and then applies Dijkstra from every source, yielding APSP in $O(nm \log n)$

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

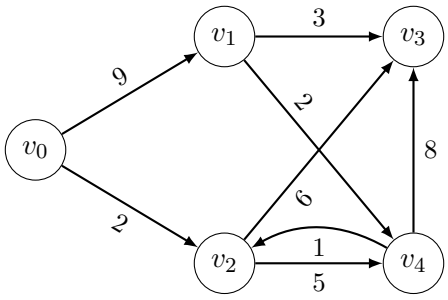


Figure 7.1: A weighted directed graph

time — faster than FloydWarshall on sparse graphs.

The remaining sections reformulate these algorithms in the LLP framework, presenting LLP versions of BellmanFord, FloydWarshall, and Johnson's algorithm.

7.2 Dijkstra's Algorithm

We consider a directed weighted graph (V, E, w) where V is the set of vertices, E is the set of directed edges, and w is a map from the set of edges to positive reals (see Fig. 7.1 for a running example). We assume that the graph is loop-free.

Algorithm Dijkstra: Finding the shortest costs from v_0 .

input: Graph $\mathcal{G} = (V, E)$ with edge weights $w[j, k]$; source vertex v_0
output: $d[0..n-1]$: array of integer initially $\forall i : d[i] = \infty$

```
1  fixed: array[0...n-1] of boolean initially ∀i : fixed[i] = false;
2  H: binary heap of (j, c) initially empty ;    // j is the vertex and c is the cost
3  d[0] := 0;
4  H.add(0, d[0]);
5  while ¬H.empty() do
6  |  (j, c) := H.removeMin();
7  |  if fixed[j] then continue;
8  |  fixed[j] := true;
9  |  forall k: ¬fixed[k] ∧ (j, k) ∈ E do
10 |  |  if d[k] > d[j] + w[j, k] then
11 |  |  |  d[k] := d[j] + w[j, k];
12 |  |  |  H.add(k, d[k]);
13 |  |  end
14 |  end
15 end
```

Dijkstra's algorithm maintains $d[i]$, which is a cost to reach v_i from v_0. Each vertex x in the graph has initially $d[x]$ equal to ∞. Whenever a vertex is discovered for the first time, its $d[x]$ becomes less than ∞. We use the predicate $discovered(x) \equiv d[x] < \infty$. The variable d

decreases for a vertex whenever a shorter path is found due to edge relaxation.

In addition to the variable d, a Boolean array *fixed* is maintained. Thus, every discovered vertex is either *fixed* or *non-fixed*. The invariant maintained by the algorithm is that if a vertex x is *fixed* then $d[x]$ gives the final shortest cost from vertex v_0 to x. If x is *non-fixed*, then $d[x]$ is the cost of the shortest path to x that only goes through fixed vertices.

When all edge weights are non-negative, distances can only increase as we extend a path. Thus, among all currently discovered vertices, the one with the smallest tentative distance must already have its final shortest-path value. Any alternative path to this vertex would have to go through another vertex whose distance is at least as large, and then incur an additional non-negative cost, making it no better. Therefore, we can safely fix the closest unfixed vertex at each step and propagate its distance to its neighbors, growing the set of vertices with known optimal distances in a greedy manner.

The algorithm uses a heap H that contains all vertices that have been discovered but are not fixed along with their distance estimates d. We view the heap as consisting of tuples of the form $(j, d[j])$ where the heap property is with respect to the d values. The algorithm has one main *while* loop that removes the vertex with the minimum distance from the heap with the method H.removeMin(), say v_j, and marks it as fixed. It then *explores* the vertex v_j by relaxing all its adjacent edges going to non-fixed vertices v_k. The value of $d[k]$ is updated to the minimum of $d[k]$ and $d[j] + w[j, k]$. This step is called *edge relaxation*. If v_k is not on the heap, then it is inserted; else, if $d[k]$ has decreased, then the label associated with the vertex k is inserted in the heap. We abstract this step as the method H.add$(k, d[k])$. Since the vertex may already be on the heap, the insertion may cause a vertex to be present in a heap with different distances. Hence, when we remove a vertex from the heap, if it is already fixed, we simply go to the next vertex in the heap. The algorithm terminates when the heap is empty. At this point there are no discovered non-fixed vertices and d reflects the cost of the shortest path to all discovered vertices. If a vertex j is not discovered then $d[j]$ is infinity reflecting that v_j is unreachable from v_0.

Observe that every vertex goes through the following states. Every vertex x is initially *undiscovered* (that is, $d[x] = \infty$). If x is reachable from the source vertex, then it is eventually *discovered* (i.e., $d[x] < \infty$). A discovered vertex is initially *non-fixed*, and is therefore in the heap H. Whenever a vertex is removed from the heap it is a *fixed* vertex. A fixed vertex may either be *unexplored* or *explored*. Initially, a fixed vertex is unexplored. It is considered explored when all its outgoing edges have been relaxed.

The following lemma simply summarizes the well-known properties of Dijkstra's algorithm. We leave them as an exercise.

Lemma 7.1 *The outer loop in Dijkstra's algorithm satisfies the following invariants.*
(a) For all vertices x: $fixed[x] \Rightarrow (d[x] = cost[x])$.
(b) For all vertices x: $d[x]$ is equal to the cost of the shortest path from v_0 to x such that all vertices in the path before x are fixed.

Theorem 7.2 (Dijkstra correctness) *On a directed graph with non-negative edge weights, Algorithm Dijkstra terminates with $d[x] = cost[x]$ for every vertex x, where $cost[x]$ is the length of the shortest path from v_0 to x (or ∞ if x is unreachable).*

Proof: By Lemma 7.1(a), every vertex marked $fixed$ receives its correct shortest-path cost. It remains to show that every reachable vertex eventually becomes $fixed$. An extracted vertex is inserted into the heap when a relaxed edge discovers it for the first time; since the heap is not drained until it is empty and every relaxation either inserts a new vertex or replaces an old key with a smaller one, every discovered vertex is eventually extracted and fixed. When the heap empties, by Lemma 7.1(b) every unreached vertex has $d[x] = \infty$, which matches $cost[x]$.

The greedy invariant — that the minimum-key vertex on the heap is ready to be fixed — depends on non-negativity: if v has the smallest heap key d, no path through a not-yet-fixed vertex u can improve on d, because the sub-path through u already costs at least the heap key of u, which is $\geq d$, and all remaining edge weights are non-negative.

■

For complexity analysis, let n be the number of vertices and m be the number of edges. We analyze the version of Dijkstra's algorithm in which, instead of adjusting the key in the heap for a vertex, we simply insert the vertex in the heap. As a result, the heap may have a vertex multiple times with different keys. When a vertex is removed, we check if it has already been fixed. If it is fixed, then we do not need to explore it, and we can remove the next vertex in the heap. Since a vertex can be inserted in the heap only when an edge is relaxed, we find that there are at most m insertions in the heap. We can also conclude that there are at most m deletions from the heap. Since an insertion or deletion of a heap takes $O(\log m) = O(\log n)$ time, we get the overall time complexity of $O(m \log n)$.

Example 7.3 Here is a walk-through of Dijkstra's algorithm on the graph of Figure 7.1, starting from the vertex v_0.

1. Set v_0 as the source vertex and assign it a distance of 0. Assign all other vertices a tentative distance of infinity. We insert the source vertex v_0 in the heap H.

2. Since the heap is not empty, we remove the vertex at the top of the heap. It is v_0 with a distance of 0. We mark that vertex as *fixed*. We examine its neighbors v_1 and v_2. For $v_0 \to v_1$, the existing distance to v_1 is infinity. The distance 0 (distance to v_0) + 9 (edge weight from v_0 to v_1) is less than infinity, so we update the distance to v_1 to 9 and add it to the heap. For $v_0 \to v_2$, the existing distance to v_2 is infinity. The distance 0 + 2 is less than infinity, so update the distance to v_2 to 2 and add v_2 to the heap.

3. We remove the vertex at the top of the heap. The smallest distance among the non-fixed vertices is 2 (for v_2), so set v_2 as the current vertex. We examine its neighbors, v_3 and v_4. For $v_2 \to v_3$, the existing distance to v_3 is infinity. Since 2 (distance to v_2) + 6 (edge weight from v_2 to v_3) is less than infinity, we update the distance to v_3 to 8. For $v_2 \to v_4$, the existing distance to v_4 is infinity. Since $2 + 5$ is less than infinity, we update the distance to v_4 to 7.

4. Continuing in this manner until the heap becomes empty, we get the final distances as: v_0: 0, v_1: 9, v_2: 2, v_3: 8, v_4: 7.

7.3 BellmanFord Algorithm

In this section, we give an algorithm that computes the shortest paths from any given vertex even when the edge weights are negative. Why could we not apply Dijkstra's algorithm for this problem? Consider the case where the node that can be reached from the source node on one edge with a minimum cost w has a cost of x. Let this node be v. Dijkstra's algorithm assumes that this is the shortest path to node v, because any other path would already have incurred the cost of w by going to some other vertex. However, in the presence of negative weights, there may be a longer path to v by incurring more cost initially but then traversing an edge with negative cost.

When there are negative cost edges, we have to be careful that there are no negative cost cycles. In the presence of negative cost cycles, the cost of going from s to some vertex v may be decreased in an unbounded manner by going through the cycle. For now, we will assume that the graph may have negative cost edges but no negative cost cycles.

In the 1950s, Bellman and (independently) Ford gave an algorithm for graphs with negative weights. The algorithm uses dynamic programming. To establish a recurrence relation, we need to come up with an appropriate variable with an index k such that its base case when k is small is easy and when k is large, we reach our goal. We let $d^k[v]$ be the cost of reaching v from the source vertex s with a path of at most k edges. It is clear that the base case of k equal to 0 is quite simple. The value $d^0[v]$ equals 0 when v equals s and ∞ otherwise. When k equals $n-1$, we claim that $d^k[v]$ is exactly equal to the shortest path from s to v. Why could

there not be a path with n edges or more, with a lower cost? If there is a path with n edges, then at least some vertex w appears more than once. The path from w to itself forms a cycle. From our assumption, there are no negative cost cycles. If the cycle is not negative, then we get a smaller path with equal or lower cost by removing the cycle.

The recurrence relation is as follows: $d^0[v] = 0$ if v equals s and ∞, otherwise.

$$d^k[v] = \min\{d^{k-1}[v], \min_{(u,v)\in E} d^{k-1}[u] + w(u,v)\}$$

We let $p[v]$ denote the predecessor of node v in the shortest path. Initially, $p[v]$ is null for all vertices.

Algorithm BellmanFord shows the procedure used when the edge weights in a directed graph may be negative.

Algorithm BellmanFord: Finding the shortest costs from v_0 when edge weights may be negative .

```
   input: Graph G = (V, E), source vertex s, edge weights w(u, v)
   output: Distance array d[], Predecessor array p[]
 1 forall vertex v ∈ V do
 2 |   d[v] := ∞;
 3 |   p[v] := null;
 4 end
 5 d[s] := 0;
 6 for k = 1 to n − 1 do
 7 |   forall edge (u, v) ∈ E do
 8 |   |   if d[u] + w(u, v) < d[v] then
 9 |   |   |   d[v] := d[u] + w(u, v);
10 |   |   |   p[v] := u;
11 |   |   end
12 |   end
13 end
14 forall edge (u, v) ∈ E do
15 |   if d[u] + w(u, v) < d[v] then return "Negative cycle detected" ;
16 end
17 return d[], p[];
```

The first *for* loop initializes the value of d and p for the base case corresponding to k equal to zero. We now compute d^k for k equal to $1 \ldots n-1$. Instead of computing the recurrence explicitly for each vertex, we *relax* each edge to check if $d^k[v]$ can be reduced by some edge (u, v). Finally, we check if there is a negative cost cycle in the graph. If there is a negative cost cycle that can be reached from the source vertex, then there would be an edge (u, v) in the graph such that $d[v]$ can be reduced further by using that edge even after k has reached the value $n-1$.

The time complexity of BellmanFord is $O(nm)$, making it less efficient than Dijkstra's algorithm for graphs with non-negative weights but more versatile due to its handling of negative weights. The space complexity is $O(n)$ for the arrays d and p.

In our algorithm, we have iterated k from 1 to $n-1$. What if the values of d do not change

in some iteration of k? The reader is invited to change the algorithm so that it checks for this condition and terminates sooner.

Theorem 7.4 (BellmanFord correctness) *Let $\mathcal{G} = (V, E)$ be a directed graph with edge weights w and source s, containing no negative-weight cycle reachable from s. After the main loop of Algorithm BellmanFord completes, $d[v]$ equals the cost of a shortest path from s to v for every $v \in V$. The final scan correctly reports a negative-weight cycle if and only if one is reachable from s.*

Proof: By induction on k. Let $d^k[v]$ denote the cost of a shortest path from s to v with at most k edges (or ∞ if no such path exists). The initialization gives $d[v] = d^0[v]$. Assume that after $k-1$ iterations, $d[v] \le d^{k-1}[v]$ for all v. Relaxing every edge once in iteration k ensures that for any edge (u, v), $d[v] \le d[u] + w(u, v) \le d^{k-1}[u] + w(u, v)$, which is at most $d^k[v]$ taken over all such u. Hence $d[v] \le d^k[v]$ after iteration k.

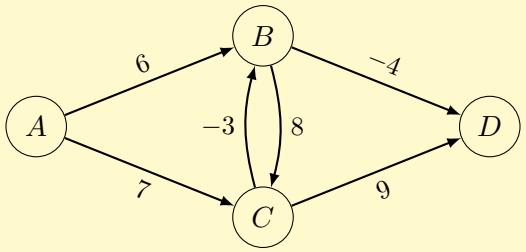
Conversely, $d[v]$ is never smaller than the true shortest-path cost $cost[v]$, because $d[v]$ is only updated via $d[v] := d[u] + w(u, v)$, so $d[v]$ always equals the weight of some path from s to v (or ∞), and no path can be cheaper than the shortest one.

Combining: after $n-1$ iterations, $cost[v] \le d[v] \le d^{n-1}[v] = cost[v]$, since any shortest path without a negative cycle has at most $n-1$ edges.

For the negative-cycle test: if the algorithm can still relax some edge after $n-1$ iterations, then there is a path from s to some vertex with cost less than d^{n-1}, which is only possible if a negative-weight cycle is reachable from s. Conversely, in the presence of such a cycle, edges along it can always be relaxed.

■

Example 7.5 Consider the directed graph below on vertices A, B, C, D with source A. The cycle $B \to C \to B$ has weight $8 + (-3) = 5$, and there is no other directed cycle, so the graph has no negative-weight cycle.



Starting from $d[A] = 0$ and $d[B] = d[C] = d[D] = \infty$, the three iterations of the main loop produce:

After iteration	$d[A]$	$d[B]$	$d[C]$	$d[D]$
0 (init)	0	∞	∞	∞
1	0	4	7	2
2	0	4	7	0
3	0	4	7	0

The values stabilize at iteration 2; the subsequent iteration leaves d unchanged, and the negative-cycle check passes. The shortest-path costs from A are therefore $d[A] = 0$, $d[B] = 4$, $d[C] = 7$, $d[D] = 0$.

7.4 FloydWarshall Algorithm

Suppose that $A[i, j]$ represents the weight of the direct edge from i to j denoting the cost of going from i to j using at most one edge. We assume that all diagonal entries are zero and that there is no negative weight cycle. Our goal is to find a matrix $G[i, j]$ such that $G[i, j]$ is the least cost to go from i to j using any number of edges.

Rather than focusing on paths from a single source, we build shortest paths between all pairs of vertices by progressively allowing more intermediate vertices. At stage k, we assume that only vertices $\{1, \ldots, k\}$ may appear as intermediates. For each pair (i, j), the shortest path either avoids vertex k entirely or passes through it, in which case it splits into two shorter subpaths $(i \to k)$ and $(k \to j)$. By systematically considering each vertex as a possible intermediate, we explore all possible path decompositions and obtain the shortest distances between all pairs. We define $d^k[i, j]$ as the cost of going from vertex i to vertex j using intermediate vertices from $1..k$. When k is zero, it is clear that we cannot use any intermediate vertex. Thus,

$$d^0[i, j] = 0 \; if \; (i = j), w(i, j) \; otherwise.$$

Observe that if there is no edge from i to j when $i \neq j$, $w(i, j)$ is ∞. Now, consider the case when $k \geq 1$. Then, we get the following recurrence:

$$d^k[i, j] = \min\{d^{k-1}[i, j], d^{k-1}[i, k] + d^{k-1}[k, j]\}$$

Thus, we get the FloydWarshall algorithm (Algorithm FloydWarshall).

Algorithm FloydWarshall: All pairs shortest path algorithm

input: A weighted graph represented by an $n \times n$ cost matrix $G[i, j]$, where $G[i, j]$ is the cost from vertex i to j (possibly ∞ if no edge exists).

output: Updated matrix $G[i, j]$ containing shortest path costs between all pairs of vertices.

```
1 for k = 1 to n do
2 |  for i = 1 to n do
3 |  |  for j = 1 to n do
4 |  |  |  if G[i,k] + G[k,j] < G[i,j] then
5 |  |  |  |  G[i,j] := G[i,k] + G[k,j];
6 |  |  |  end
7 |  |  end
8 |  end
9 end
```

The reader is invited to make two changes to the algorithm. First, modify the algorithm to return an error message when there is a negative cost cycle in the graph. Second, add a variable $p[u][v]$ to the algorithm to return the predecessor of v on the path from u to v.

The time complexity of the FloydWarshall algorithm is $O(n^3)$: the three nested loops iterate n times each and the body is $O(1)$. The space complexity is $O(n^2)$ for the matrix G, with updates done in place.

Theorem 7.6 (FloydWarshall correctness) *On a directed graph without negative-weight cycles, Algorithm FloydWarshall terminates with $G[i, j]$ equal to the cost of a shortest path from vertex i to vertex j, for every pair (i, j).*

Proof: Let $d^k[i, j]$ denote the cost of a shortest path from i to j whose internal vertices all lie in $\{1, 2, \ldots, k\}$, or ∞ if no such path exists. By induction on k: the base case $k = 0$ matches the input matrix (only direct edges). For the inductive step, any shortest (i, j)-path using internal vertices from $\{1, \ldots, k\}$ either avoids k — in which case its cost is $d^{k-1}[i, j]$ — or uses k, in which case its two halves are shortest paths from i to k and from k to j using internal vertices from $\{1, \ldots, k-1\}$, of cost $d^{k-1}[i, k] + d^{k-1}[k, j]$. The recurrence taking the minimum of the two is exactly what the algorithm computes in iteration k. After n iterations, $G[i, j] = d^n[i, j]$, which is the unrestricted shortest-path cost since all vertices are now permissible as internal.

■

Example 7.7 Consider a graph on vertices $\{1, 2, 3\}$ with direct-edge costs given by

$$G^{(0)} = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix}.$$

After iteration $k = 1$ (allowing vertex 1 as an intermediate), only the $(3, 2)$ entry could potentially change: $G[3, 2] = \min(\infty, G[3, 1]+G[1, 2]) = \min(\infty, 3+4) = 7$. After iteration $k = 2$, the $(1, 3)$ entry becomes $\min(11, 4 + 2) = 6$ and the $(3, 3)$ entry is checked to be 0. Iteration $k = 3$ updates $(2, 1)$ via $(2, 3, 1)$ to $\min(6, 2 + 3) = 5$. The final all-pairs shortest-path matrix is

$$G = \begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}.$$

7.5 Johnson's Algorithm

Johnson's algorithm solves all-pairs shortest paths on directed graphs whose edge weights may be negative but contain no negative-weight cycle. It combines BellmanFord and Dijkstra: a single BellmanFord run computes a non-negative *price* vector with which the edges are reweighted, and Dijkstra is then run from every source on the reweighted graph. On sparse graphs this beats the $O(n^3)$ FloydWarshall bound: using Dijkstra yields $O(nm \log n)$ total time.

Negative edge weights prevent the direct use of Dijkstra's algorithm, but we can transform the graph so that all edge weights become non-negative while preserving shortest paths. This is done by assigning a potential (or price) to each vertex and adjusting edge weights accordingly. The reweighting shifts all path costs by a constant depending only on endpoints, so relative ordering of paths is unchanged. Once all edges are non-negative, we can efficiently run Dijkstra from each source. Thus, Johnson's algorithm combines the flexibility of BellmanFord (to handle negative edges) with the efficiency of Dijkstra (for fast shortest-path computation).

Let $\mathcal{G}$ be the input graph. The *reweighted* graph $\mathcal{G}'$ has the same vertices and edges, with edge weights

$$w'[i, j] = w[i, j] + price[j] - price[i], \tag{7.1}$$

where *price* is a non-negative *price* vector such that every $w'[i, j] \geq 0$. This reweighting preserves shortest paths:

Lemma 7.8 *Let s and t be any two vertices in the graph. The weight of any path on the graph $\mathcal{G}'$ from s to t equals the weight in the graph $\mathcal{G}$ plus $(price[t] - price[s])$.*

Proof: Let the path be $v_0, v_1, v_2, \ldots, v_k$ where $v_0 = s$ and $v_k = t$. As we compute the cost of the path in $\mathcal{G}'$, we add the price of every intermediate vertex once and subtract it once. Only the price of v_0 is subtracted exactly once, and the price of v_k is added exactly once, giving us the lemma.

■

Since path costs shift by a constant $price[t] - price[s]$, shortest paths are unchanged. It remains to find a feasible *price*, which exists exactly when $\mathcal{G}$ has no negative-weight cycle:

Lemma 7.9 *A feasible non-negative price vector exists iff the graph $\mathcal{G}$ has no negative-weight cycle. When it does, $price[v] := -dist(s^\star, v)$ is feasible, where $dist(s^\star, v)$ is the BellmanFord distance to v from a fresh auxiliary vertex $s^\star$ joined to every $x \in V$ by a 0-weight edge.*

Proof: ($\Rightarrow$) By Lemma 7.8 with $s = t$, any cycle has the same weight in $\mathcal{G}$ and $\mathcal{G}'$, so a negative-weight cycle in $\mathcal{G}$ leaves some reweighted edge negative.

($\Leftarrow$) Augmenting $\mathcal{G}$ with $s^\star$ cannot create a new negative cycle since $s^\star$ has no incoming edges; hence $dist(s^\star, v)$ is well-defined and ≤ 0. Set $price[v] := -dist(s^\star, v) \geq 0$. The Bellman-Ford triangle inequality $dist(s^\star, j) \leq dist(s^\star, i) + w[i, j]$ rearranges to $w[i, j] + price[j] - price[i] \geq 0$, so *price* is feasible.

■

BellmanFord from $s^\star$ finds such a *price* in at most n edge-relaxation passes (matching the longest simple path), and reports a negative cycle otherwise.

Algorithm Johnson: Johnson's algorithm for all-pairs shortest paths on graphs with negative edges

input: directed graph $\mathcal{G} = (V, E)$ with edge weights w (possibly negative)
output: matrix D with $D[u, v]$ the shortest-path cost from u to v; or a report of a negative cycle

```
1  Add vertex s* and an edge (s*, v) of weight 0 for every v ∈ V;
2  Run BellmanFord from s*; if it reports a negative cycle, return "negative-weight
     cycle";
3  price[v] := −dist(s*, v) for every v ∈ V;
4  forall edges (i, j) ∈ E do
5  |  w'[i, j] := w[i, j] + price[j] − price[i];
6  end
7  forall u ∈ V do
8  |  Run Dijkstra from u on the graph with weights w' to get dist'(u, v) for every v;
9  |  forall v ∈ V do
10 |  |  D[u, v] := dist'(u, v) + price[u] − price[v];          // undo the reweighting
11 |  end
12 end
13 return D;
```

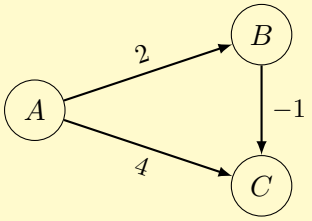
Theorem 7.10 (Johnson correctness) *On a directed graph without a negative-weight cycle, Algorithm Johnson returns $D[u,v] = dist(u,v)$ for every pair of vertices. If the graph has a negative-weight cycle, the algorithm reports it.*

Proof: The BellmanFord step correctly finds $dist(s^\star, v)$ or a negative cycle; by Lemma 7.9, $price[v] := -dist(s^\star, v)$ makes every reduced weight non-negative, so each Dijkstra call is valid. Lemma 7.8 gives $dist'(u,v) = dist(u,v) + price[v] - price[u]$, and the final line of the algorithm inverts this shift.

■

Complexity is dominated by the single BellmanFord call ($O(nm)$) and the n Dijkstra calls. This gives $O(nm \log n)$, beating FloydWarshall's $O(n^3)$ on sparse graphs.

Example 7.11 Consider the directed graph on three vertices A, B, C with edges $A \to B$ of weight 2, $B \to C$ of weight -1, and $A \to C$ of weight 4:



The graph has no negative-weight cycle (it has no directed cycle at all).

Step 1 (BellmanFord from $s^\star$). Add an auxiliary vertex $s^\star$ with a 0-weight edge to A, B, and C. BellmanFord returns $dist(s^\star, A) = 0$, $dist(s^\star, B) = 0$, $dist(s^\star, C) = -1$ (realized by $s^\star \to B \to C$).

Step 2 (price vector). Set $price := -dist(s^\star, \cdot)$, giving $price(A) = 0$, $price(B) = 0$, $price(C) = 1$.

Step 3 (reweighting). The reduced weights $w'[i,j] = w[i,j] + price[j] - price[i]$ are $w'(A,B) = 2 + 0 - 0 = 2$, $w'(B,C) = -1 + 1 - 0 = 0$, $w'(A,C) = 4 + 1 - 0 = 5$, all non-negative.

Step 4 (Dijkstra from every source on reweighted graph). The reweighted distances are

$$dist'(A,A) = 0, \quad dist'(A,B) = 2, \quad dist'(A,C) = 2 \text{ (via } A \to B \to C);$$

$$dist'(B,B) = 0, \quad dist'(B,C) = 0; \qquad dist'(C,C) = 0.$$

(Entries from B or C back to A are ∞.)

Step 5 (undo the shift). Recover $D[u,v] = dist'(u,v) + price[u] - price[v]$:

$$D = \begin{pmatrix} 0 & 2 & 1 \\ \infty & 0 & -1 \\ \infty & \infty & 0 \end{pmatrix},$$

where rows are indexed by u and columns by v. Verification: the $A \to C$ entry picks the two-edge path $A \to B \to C$ of cost $2 + (-1) = 1$, cheaper than the direct $A \to C$ edge of cost 4.

7.6 LLP Perspective

In this section we collect the LLP reformulations of the algorithms introduced earlier in the chapter.

LLP-Dijkstra Algorithm

We now cast Dijkstra's algorithm (Section 7.2) in the LLP framework. The objects being computed are the distances $G[0..n-1]$ from the source vertex v_0. Using the same lattice as in the LLP-BellmanFord formulation, we regard distances as elements of $(\mathbb{R}_{\geq 0} \cup \{\infty\})^n$ and place the "small" distances *higher* in the lattice: $G \leq_L G' \iff \forall i : G[i] \geq G'[i]$. The feasibility predicate is the shortest-path recurrence

$$B \equiv \forall (i,j) \in E : \; G[j] \leq G[i] + w[i,j],$$

which is lattice-linear by the same argument used for LLP-BellmanFord.

Dijkstra's algorithm is a particular LLP schedule that exploits non-negative edge weights to fix one vertex at a time. Let $fixed[i]$ be a Boolean flag (initially $false$ for all i), and initialize $G[0] := 0$ and $G[i] := \infty$ for $i \neq 0$. An index j is *forbidden* when it is the unique non-fixed vertex whose current $G[j]$ is the minimum among non-fixed vertices; the advance rule fixes j and relaxes its outgoing edges.

The efficient implementation maintains the set of non-fixed discovered vertices in a min-heap H keyed by G. Extracting the minimum of the heap yields the forbidden vertex at each round, and edge relaxation inserts updated keys back into the heap. The algorithm below is literally Algorithm Dijkstra rewritten in LLP notation.

Algorithm LLP-Dijkstra: Dijkstra's algorithm as an LLP schedule

input: graph (V, E) with non-negative edge weights $w[i,j]$; source v_0
output: $G[0..n-1]$: real, initially $G[0] = 0$ and $G[i] = \infty$ for $i \neq 0$

1 $fixed$: array$[0..n-1]$ of boolean, initially $\forall i : fixed[i] = false$;
2 **priority**(j): $G[j]$;
3 **forbidden**(j): $\neg fixed[j] \wedge G[j] < \infty$
4 $\Rightarrow$ **advance**: $fixed[j] := true$;
5 **forall** $(j,k) \in E$ with $\neg fixed[k]$: $G[k] := \min(G[k],\ G[j] + w[j,k])$;

Correctness follows from the same invariant as in Section 7.2: whenever a vertex is extracted from the heap with key G_j, non-negativity of the edge weights guarantees that no still-undiscovered vertex can lead to a shorter path to j, so j can safely be fixed. Lemma 7.1 captures both parts of this invariant.

The LLP formulation makes clear that Dijkstra's algorithm differs from LLP-BellmanFord only in its schedule: BellmanFord advances every forbidden index concurrently, while Dijkstra advances exactly one index per round (the non-fixed vertex of minimum G). The min-heap makes this schedule efficient, yielding the same $O(m \log n)$ sequential running time as the heap-based Dijkstra of Section 7.2.

LLP-BellmanFord Algorithm

Consider the distance of other vertices from s. To apply LLP algorithm, we view finding the optimal distance vector as finding the largest $G[x]$ for every vertex $x \neq v_0$ which satisfies the following feasibility predicate:

$$\text{for all edges } (i,j) \in E : G[j] \leq G[i] + w[i,j]$$

We define B_e for any edge $e = (i,j)$ as $G[j] \leq G[i] + w[i,j]$. The feasibility predicate B can be written as

$$B \equiv \wedge_{e \in E} B_e$$

We will view the search for optimal distance vector as searching for G vector in a lattice defined as follows. In this lattice $G \leq_L G'$ iff $\forall i : G[i] \geq G'[i]$. Thus, finding the least vector in this lattice corresponds to finding the largest vector G that satisfies B. The top element is the zero vector and the bottom element is $(\infty, \infty, \ldots, \infty)$.

We now claim that

Theorem 7.12 *B is a lattice linear predicate.*

Proof: It is sufficient to show that B_e is lattice linear because any conjunction of linear predicates is also lattice linear. B_e for $e = (i,j)$ is equivalent to $G[j] \leq G[i] + w[i,j]$. This is equivalent to $G[j] \geq_L G[i] + w[i,j]$. Since the right hand side is a monotonic function of G, we get that B_e is lattice linear.

■

Algorithm LLP-Edge-Relaxation: Finding the minimum distance vector satisfying B

input: $pre(j)$: list of $1..n$; $w[i,j]$: int for all $i \in pre(j)$
output: $G[0..n-1]$: array of int, initially if $(j = s)$ then 0 else $maxint$
1 **ensure**: $G[j] \leq \min\{G[i] + w[i,j] \mid i \in pre(j)\}$;

Algorithm LLP-Edge-Relaxation gives the correct answer; however, it may result in large computation complexity if G's are lowered in a non-disciplined manner. We now optimize

this algorithm. Algorithm LLP-BellmanFord chooses forbidden indices more carefully. The algorithm checks across all vertices whether any vertex is forbidden; if there is any forbidden vertex, then all edges are relaxed.

Algorithm LLP-BellmanFord: Finding the minimum distance vector satisfying B

---

input: $pre(j)$: adjacency list; $w[i,j]$: edge weights
output: $G[0..n-1]$: array of real initially $(G[0]=0) \wedge (\forall i \neq 0 \colon G[i]=\infty)$

1 **forbidden**(j): $\exists i \in pre(j) : G[j] > G[i] + w[i,j]$
2 $\quad \Rightarrow$ **advance**: $\forall k : G[k] := \min\{G[i] + w[i,k] \mid i \in pre(k)\}$;

Let us analyze the sequential time complexity of Algorithm LLP-BellmanFord. At most n rounds of edge relaxation are needed; each round can be implemented in $O(m)$ time, giving an overall sequential time complexity of $O(mn)$.

Earlier, we had assumed that the graph does not have any negative cost cycle. What if the user input is such that it has a negative cost cycle? In that case, Algorithm LLP-BellmanFord will never terminate. The standard fix, as in the classical BellmanFord algorithm of Section 7.3, is to cap the number of *while*-loop iterations at $|V|-1$ and report "negative cost cycle" if any index is still forbidden after that.

LLP-FloydWarshall

We now give an LLP formulation of the FloydWarshall algorithm. Suppose that $A[i,j]$ represents the weight of the direct edge from i to j, denoting the cost of going from i to j using at most one edge. We assume that all diagonal entries are zero and that there is no negative weight cycle. Our goal is to find a matrix $G[i,j]$ such that $G[i,j]$ is the least cost of going from i to j by using any number of edges.

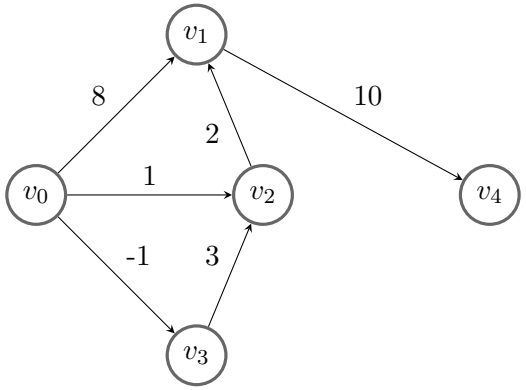


Figure 7.2: A directed graph used as a running example for the LLP-FloydWarshall algorithm.

Let us formulate this problem as finding the least vector of size n^2 in a distributive lattice. We initialize G as A. This is the bottom element of our distributive lattice. Our goal is to find

the least vector that satisfies the following constraint:

$$B_{FloydWarshall} \equiv \forall i, j : i \neq j : G[i,j] \leq \min_k \{G[i,k] + G[k,j]\}.$$

When i equals j, $G[i,j]$ equals 0 and will stay fixed through the execution of the algorithm. The cost of going from i to j must be smaller than the cost of going from i to k and k to j. Algorithm LLP-FloydWarshall gives a high-level LLP algorithm for all pairs shortest path.

Algorithm LLP-FloydWarshall: Computing the shortest cost matrix for a directed graph

input: A: matrix of real ($A[i,j]$ is the weight of the edge from i to j)
output: G: matrix of real, initially $\forall i, j : G[i,j] = A[i,j]$
1 **forbidden**(i,j): $(\exists k : G[i,j] > G[i,k] + G[k,j])$
2 $\Rightarrow$ **advance**: $G[i,j] := \min_k \{G[i,k] + G[k,j]\}$;

LLP-Johnson

We now revisit Johnson's reweighting framework of Section 7.5. Recall the reweighting identity $w'[i,j] = w[i,j] + G[j] - G[i]$ from Equation 7.1 and Lemma 7.8, which says that every path from s to t in the reweighted graph Y has weight $w_X(\text{path}) + G[t] - G[s]$. The sequential algorithm of the previous section uses BellmanFord from an auxiliary source $s^\star$ to obtain a feasible price vector. Here we give an LLP algorithm that finds such a price vector directly. Our feasibility predicate B for the pricing vector is

$$\forall (i,j) \in E : w[i,j] + G[j] - G[i] \geq 0$$

Furthermore, we require $G[i] \geq 0$ for all i. We first show that B is lattice linear.

Lemma 7.13 *Let X be any graph such that the edge (i,j) has weight $w[i,j]$ and every vertex i has price $G[i]$. Consider the lattice of all non-negative price vectors. Then, the predicate*

$$B \equiv \forall (i,j) \in E : w[i,j] + G[j] - G[i] \geq 0$$

is lattice linear.

Proof: Since the lattice linearity is closed under conjunction, it suffices to show that $B_e \equiv w[i,j] + G[j] - G[i] \geq 0$ is lattice linear for an arbitrary edge $e = (i,j)$. B_e can be rewritten as $G[j] \geq G[i] - w[i,j]$. The right-hand side of this inequality is a monotone function on G and hence, from the Key Lemma of lattice-linearity, we get that B_e is lattice linear.

■

By applying the LLP algorithm, we get the following method to find the price vector.

Termination and iteration bounds for Algorithm LLP-Johnson match those of the sequential construction in Section 7.5: by Lemma 7.9, the advance rule converges to a feasible price vector

Algorithm LLP-Johnson: Finding the minimum price vector

input: $pre(j)$: list of $1..n$; $w[i,j]$: int for all $i \in pre(j)$
output: $G[0..n-1]$: array of int, initially 0
1 **forbidden**(j): $\exists i \in pre(j) : G[j] < G[i] - w[i,j]$
2 $\Rightarrow$ **advance**: $\forall k : G[k] := \max\{G[i] - w[i,k] \mid i \in pre(k)\}$;

iff the graph has no negative-weight cycle, and at most $|V|$ rounds of edge relaxation suffice. From the Key Lattice-Linearity Lemma, the set of all feasible price vectors is closed under meets, so the LLP algorithm converges to the componentwise-minimum feasible price vector; since adding a positive constant to all prices preserves feasibility and leaves reduced weights unchanged, this minimum vector has at least one zero component.

Each LLP iteration can be computed in $O(m)$ time, so the graph is "reweighted" in $O(mn)$ time — the same asymptotic bound that BellmanFord achieves in the sequential algorithm. Plugging the resulting price vector into Dijkstra from every source yields an all-pairs shortest-path algorithm with running time $O(nm \log n)$, which is faster than the $O(n^3)$ FloydWarshall algorithm whenever the graph is not dense.

7.7 Summary

This chapter covered four algorithms for shortest-path problems. Dijkstra's algorithm solves single-source shortest paths on graphs with non-negative edge weights in $O(m \log n)$ time. BellmanFord handles negative edge weights by relaxing all edges over $n-1$ iterations in $O(nm)$ time, and also detects negative-weight cycles. FloydWarshall computes all-pairs shortest paths in $O(n^3)$ time using an intermediate-vertex recurrence. Johnson's algorithm combines BellmanFord and Dijkstra to solve all-pairs shortest paths in $O(nm \log n)$ time, which is faster than FloydWarshall on sparse graphs. The chapter also presented LLP reformulations of these algorithms that expose parallelism by allowing multiple forbidden indices to advance concurrently.

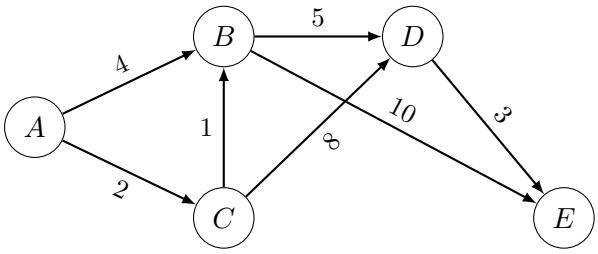
Table 7.2 lists all the algorithms discussed in this chapter.

Problem	**Algorithm**	**Time Complexity**
Shortest Path	Dijkstra	$O(m \log n)$
Shortest Path	BellmanFord	$O(mn)$
All Pairs Shortest Path	FloydWarshall	$O(n^3)$
All Pairs Shortest Path	Johnson	$O(nm \log n)$
Shortest Path	LLP-Dijkstra	$O(m \log n)$
Shortest Path	LLP-BellmanFord	$O(mn)$
All Pairs Shortest Path	LLP-FloydWarshall	$O(n^3)$
All Pairs Shortest Path	LLP-Johnson	$O(nm \log n)$

Table 7.2: Algorithms discussed in this chapter.

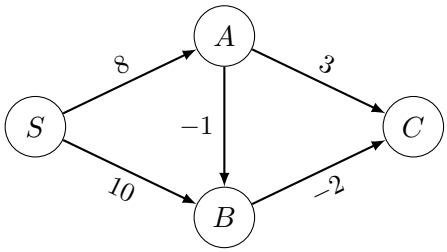
7.8 Problems

1. **(Dijkstra Example)** Consider the directed graph below:



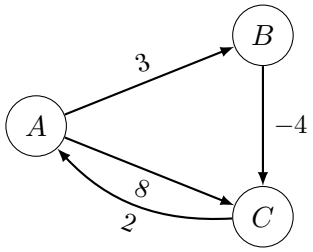
   Find the shortest path from A to E using Dijkstra's algorithm.

2. **(Bellman Ford Negative Weights)** Consider the directed graph below:



   Find the shortest path from S to C using the BellmanFord algorithm.

3. **(Floyd Warshall All Pairs)**
   Consider the directed graph below:



   Find the shortest path for all pairs using the FloydWarshall algorithm.

4. **(Dijkstra Correctness Lemma)** Prove Lemma 7.1.

5. **(Sensitive Edge Detection)** Give an algorithm to determine which of the edges in a directed graph are sensitive to the shortest path between s and t. An edge is sensitive, if its weight is reduced, then the shortest path from s to t is also reduced.

6. **(Bottleneck Shortest Path)** Let $G = (V, E, w)$ be an undirected graph with non-negative edge weights, and let $s, t \in V$. Define the *bottleneck* of a path P as $\max_{e \in P} w(e)$. Describe an algorithm that finds an s-t path of minimum bottleneck, and state its time complexity.

7.9 Bibliographic Remarks

The single source shortest path problem has a rich history. For the history of Dijkstra's algorithm, the reader is referred to the book by [Eri19b]. One popular research direction is to improve the worst case complexity of Dijkstra's algorithm by using different data structures. For example, by using Fibonacci heaps for the min-priority queue, Fredman and Tarjan [FT87] gave an algorithm that takes $O(m + n \log n)$. There are many algorithms that run faster when weights are small integers bounded by some constant γ. For example, Ahuja et al. [AMOT90] gave an algorithm that uses Van Emde Boas tree as the priority queue to give an algorithm that takes $O(m \log \log \gamma)$ time. Thorup [Tho00] gave an implementation that takes $O(n+m \log \log n)$ for graphs with integer weights on the RAM model. Raman [Ram97] gave an algorithm with $O(m+n\sqrt{\log n \log \log n})$ time. The LLP algorithm for the shortest path is taken from [AKG20]. Bellman and Ford's algorithm is from [Bel58] and [For56].

Chapter 8

The Minimum Spanning Tree Problem

8.1 Introduction

In this chapter, we discuss the problem of finding the minimum spanning tree (MST) of an undirected weighted graph. Let $\mathcal{G} = (V, E)$ be a connected undirected graph with $n = |V|$ vertices and $m = |E|$ edges, where each edge $e \in E$ has a weight $w(e) \in \mathbb{R}$. A *minimum spanning tree* (MST) of the graph is a subset $T \subseteq E$ such that:

1. T connects all vertices in V (i.e., (V, T) is connected),
2. The total weight $\sum_{e \in T} w(e)$ is minimized among all such subsets.

Equivalently, T is a tree that spans all vertices with the minimum possible total edge weight. A maximum spanning tree is defined similarly, but maximizes the total weight instead.

We assume that all edge weights are distinct. It is known that when all edge weights are distinct, a connected graph has a unique minimum spanning tree, and, more generally, a graph that is not necessarily connected has a unique minimum spanning forest. If edge weights may repeat, a minimum spanning tree (or forest) still exists but need not be unique. For simplicity of exposition, we will assume that the underlying graph is connected.

The primary motivation for finding an MST is to connect all nodes in a network with the minimum total cost while ensuring connectivity and avoiding cycles. This has numerous real-world applications:

- *Network Design*: In telecommunications or electrical grids, an MST minimizes the cost of laying cables or wires to connect all points (e.g., cities or computers) without redundant loops.
- *Transportation Networks*: Building roads or pipelines between locations with minimal total length or cost.

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

- *Clustering and Approximation Algorithms*: MSTs are used in algorithms for clustering data points or approximating solutions to NP-hard problems like the Traveling Salesman Problem (TSP), where an MST provides a lower bound for tours.
- *Computer Networks*: Efficient broadcasting and routing in networks, reducing latency and resource usage.

This chapter is organized as follows. Section 8.2 describes the key properties of a minimum spanning tree, including the notion of a *fragment*. Section 8.3 describes the Find-Union data structure, which is used by several MST algorithms to detect cycles efficiently. Section 8.4 presents Kruskal's algorithm, a greedy algorithm that chooses the least-cost feasible edge. Section 8.5 presents Prim's algorithm, another greedy algorithm that grows a single fragment. Section 8.6 describes Boruvka's algorithm, which extends multiple fragments in every iteration. Section 8.7 compares the three algorithms in terms of data structures, graph density, and parallelism. Section 8.8 reformulates these algorithms in the LLP framework. Section 8.9 discusses constrained minimum spanning trees with mandatory edges.

8.2 Key Properties of a Minimum Spanning Tree

The notion of a *fragment* is crucial in understanding MST algorithms. A fragment is simply a subtree of the MST. Consider the graph in Fig. 8.1. The minimum spanning tree in this graph corresponds to the edges $\{2, 3, 4, 7\}$. The subtree formed by the edges 3 and 4 is a fragment with three vertices $\{a, b, c\}$ and two edges $\{(a, c), (b, c)\}$. A crucial property of the MST is as follows.

Lemma 8.1 *Let F be a fragment. Let e be the edge with the minimum weight that is outgoing from F. Then $F \cup \{e\}$ is also a fragment.*

Proof: Let T be the minimum spanning tree. If $e \in T$, then $F \cup \{e\}$ is trivially a fragment. If $e \notin T$, adding e to T creates a unique cycle C. This cycle must contain another edge f that is outgoing from F (since the cycle enters and exits F). Since $w(e) \leq w(f)$ (as e has minimum outgoing weight), $T' = T \setminus \{f\} \cup \{e\}$ is a spanning tree with $w(T') \leq w(T)$. Since T is a minimum spanning tree, T' is also a minimum spanning tree and it contains e. Hence $F \cup \{e\}$ is a fragment of T'.

■

In the fragment formed by edges with weights 3 and 4, there are three outgoing edges — edges with weights 7, 9, and 11. The edge 5 is not outgoing, as it connects vertices that are part of the fragment. According to Lemma 8.1, the edge with weight 7 can be added to the edges with weights 3 and 4 to grow the fragment.

The following two properties, phrased in terms of edge-weight extremes across a cut and around a cycle, are the workhorses behind every correctness proof in this chapter.

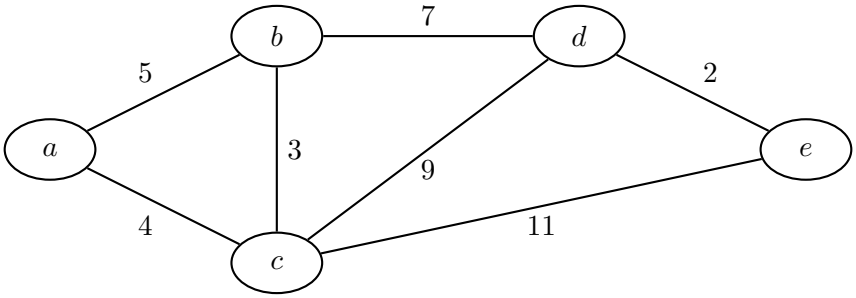


Figure 8.1: An undirected weighted graph

Lemma 8.2 (Cut Property) *Let $S \subsetneq V$ and let e be the minimum weight edge crossing the cut $(S, V \setminus S)$. Then e belongs to some MST; under distinct weights, e belongs to every MST.*

Proof: Let T be any MST and suppose $e \notin T$. Adding e to T creates a unique cycle, and since the cycle starts and ends in S it must contain another edge $f \neq e$ that also crosses the cut. By hypothesis $w(e) \leq w(f)$, so $T' = T \cup \{e\} \setminus \{f\}$ is a spanning tree with $w(T') \leq w(T)$; hence T' is an MST containing e. Under distinct weights, $w(e) < w(f)$ and the inequality is strict, forcing e to be in every MST.

■

Lemma 8.3 (Cycle Property) *Let C be a cycle in the graph and let e be the strictly maximum weight edge in C. Then e does not belong to any MST.*

Proof: Suppose, for contradiction, that $e \in T$ for some MST T. Removing e partitions V into two components S and $V \setminus S$. Because C starts and ends on the same side, some other edge $f \in C \setminus \{e\}$ also crosses the cut $(S, V \setminus S)$. By hypothesis $w(f) < w(e)$, so $T' = T \setminus \{e\} \cup \{f\}$ is a spanning tree with $w(T') < w(T)$, contradicting the optimality of T.

■

8.3 The Find-Union Data Structure

A central operation in several MST algorithms — and in Kruskal's algorithm in particular — is to test whether the next edge under consideration would close a cycle in the partially built forest. The Find-Union data structure (also known as Disjoint Set Union) supports this test efficiently. It maintains a collection of disjoint sets and supports two primary operations:

- **Find**: Determines which set a particular element belongs to by returning its representative (or "root"). With path compression, this operation is nearly constant time, $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function.

- **Union**: Merges two sets into one by linking their roots, typically using rank or size heuristics to keep trees balanced, also achieving $O(\alpha(n))$ amortized time.

Initially, each element is in its own set. When this structure is used to build a spanning forest, the *Union* operation merges sets of vertices connected by selected edges, while the *Find* operation checks if adding an edge would form a cycle (i.e., if its endpoints are already in the same set).

Each set is maintained as a rooted tree in which every node has a *parent* pointer. The root of the tree serves as the identity of the set. The *Find* operation finds the root of the tree. Observe that finding the root of the tree by traversing the parent pointer takes time proportional to the distance of the node from the root. One method of reducing this distance in future traversals is *path compression*: we make all the nodes on the find path point directly to the root, so subsequent traversals from those nodes are essentially constant time.

The *Union* operation merges two trees by making the root of one of them point to the other root. For correctness, it does not matter which root points to the other root. However, to keep the lengths of the longest paths small, we use the notion of *rank* to decide. Our rule is as follows: the root of the tree with the lower rank points to the root with the higher rank. The rank of the new tree is the same as that of the bigger rank. If both roots have the same rank, then it does not matter which one becomes the new root, and the rank of the new tree is one more than the rank of the old trees. It is a simple exercise to show that the rank of any tree with n nodes is at most $O(\log n)$ and the height of the tree is at most its rank. With these optimizations, any sequence of n Find and Union operations takes $O(n\alpha(n))$ amortized time.

Both algorithms share the following data structures:
parent: array[1..n] of int, initially $\forall v : parent[v] = v$
rank: array[1..n] of int, initially $\forall v : rank[v] = 0$

Algorithm Find: Finding the root of x's set with path compression

input: vertex x
output: root of the set containing x

```
1 if parent[x] ≠ x then parent[x] := Find(parent[x]) //path compression ;
2 return parent[x]
```

Algorithm Union: Computing union by rank of sets containing x and y

input: vertices x, y

```
1 rootX := Find(x);
2 rootY := Find(y);
3 if rootX = rootY then return// already in same set, no union needed ;
4 if rank[rootX] < rank[rootY] then parent[rootX] := rootY ;
5 else if rank[rootX] > rank[rootY] then parent[rootY] := rootX ;
6 else parent[rootY] := rootX;
7 rank[rootX] := rank[rootX] + 1 // increment rank if equal ;
```

Example 8.4 [Find-Union trace] We trace the data structure on five elements $\{1, 2, 3, 4, 5\}$ under the sequence UNION$(1, 2)$, UNION$(3, 4)$, UNION$(1, 3)$, UNION$(2, 5)$, FIND(4). Figure 8.2 shows the forest after each step.

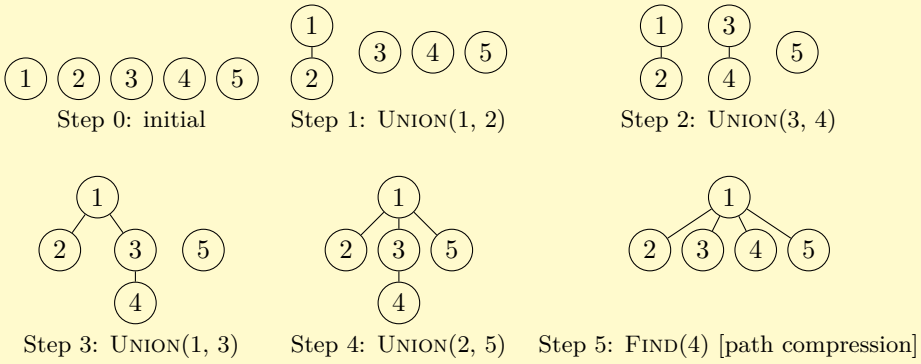


Figure 8.2: Forest after each step of the Find-Union trace. The rank of root 1 is $0, 1, 1, 2, 2, 2$ across the six panels; ranks of all other roots stay at their initial values. Step 5 shows the effect of path compression: FIND(4) walks $4 \to 3 \to 1$ and then rewires every node on the find path directly to the root.

A few details are worth noting. At Step 3, both roots have rank 1, so the tie-breaking rule attaches root 3 under root 1 and bumps $rank[1]$ to 2. At Step 4, FIND(2) traverses one pointer $(2 \to 1)$ and returns 1; FIND(5) returns 5. Because $rank[1] = 2 > rank[5] = 0$, vertex 5 is attached under root 1 without changing any rank.

At Step 5, FIND(4) walks $4 \to 3 \to 1$. Without path compression, the forest would still look like Step 4 and the next FIND(4) would re-traverse the same path. With path compression (Algorithm Find), each node on the find path is rewired directly to the root, giving the flat star-shaped tree in the final panel: subsequent FIND calls on 3 or 4 now take a single step. The *rank* values bound tree height but are themselves frozen during FIND — compression flattens the tree without invalidating the ranks already assigned. This separation is what gives the combined heuristics their $O(\alpha(n))$ amortized cost.

8.4 Kruskal's Algorithm

Kruskal's algorithm is a canonical greedy algorithm for the minimum-spanning tree problem. It chooses edges one at a time in a greedy fashion. Let T be the set of edges chosen by the algorithm at any stage. The algorithm maintains the invariant that T does not have any cycle. Initially T is empty and therefore trivially satisfies the invariant. The algorithm considers the edges in increasing order of weights. For this reason, it is sometimes also known as the *shortest-edge-next* algorithm. Suppose that the algorithm has chosen the edges in T so far. To grow T, it finds the edge with the least weight that does not form a cycle with existing edges in T. If any edge e forms a cycle with T, it is rejected, and the algorithm considers the least weight edge of the remaining edges. We use the Find-Union data structure of Section 8.3 to detect cycles efficiently: an edge (u, v) closes a cycle in T iff $Find(u) = Find(v)$.

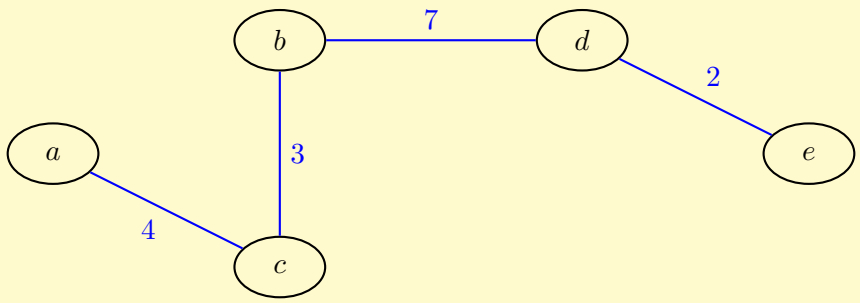
Algorithm Kruskal: Computing minimum spanning tree

input: Undirected Weighted Graph $\mathcal{G} = (V, E, w)$ with $n = |V|$
output: Minimum Weight Spanning Tree T

```
1  T: set of edges, initially {};
2  forall v ∈ V do
3  |  parent[v] := v;   rank[v] := 0 ;          // MakeSet for each vertex
4  end
5  sort the edges of E in non-decreasing order of weight;
6  foreach edge (u, v) ∈ E in sorted order do
7  |  if Find(u) ≠ Find(v) then
8  |  |  T := T ∪ {(u, v)};
9  |  |  Union(u, v);
10 |  |  if |T| = n − 1 then return T;
11 |  end
12 end
13 return T ;                          // graph is disconnected if |T| < n − 1
```

In Algorithm Kruskal, the variable T keeps the set of all edges chosen. At every step, (V, T) is an acyclic subgraph — that is, a spanning forest. The Find-Union structure represents the connected components of this forest: $Find(u)$ returns the representative of u's component, and $Union(u, v)$ merges two components when an edge is added. Adding the next lightest edge either joins two distinct components (reducing the number of components by one) or closes a cycle within a single component (in which case the edge is rejected). The algorithm terminates when $|T| = n - 1$, at which point the forest has coalesced into a spanning tree, or when the edges have all been processed and $|T| < n-1$, indicating that the input graph is disconnected.

Example 8.5 Consider the graph shown in Fig. 8.1. Kruskal's algorithm first chooses the edge with weight 2 since it has the least weight. Then, it chooses the edges with weight 3 and 4 because these edges do not form any cycle. The next edge has weight 5; this edge is rejected because it forms a cycle with the already chosen edges with weights 3 and 4. The algorithm then chooses the edge with weight 7 and terminates with $T = \{2, 3, 4, 7\}$. The resulting MST is shown below.

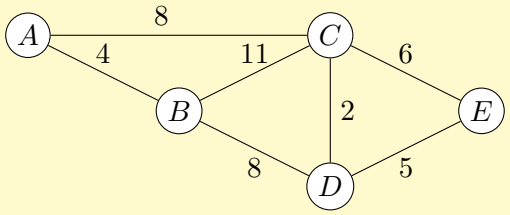


Correctness. We show that every edge added by the algorithm belongs to some MST. When

the algorithm adds $e = (u, v)$, the vertices u and v are in different components of the forest (V, T). Let S be the vertex set of u's component. Then e crosses the cut $(S, V \setminus S)$, and e is the lightest edge crossing this cut — any lighter edge would have been considered earlier and, since it also crosses the cut, would have been accepted (merging the two sides), contradicting u and v being in different components. By the cut property (Lemma 8.2), e belongs to some MST. Hence every edge in the final T belongs to an MST, so T is an MST.

Complexity. The algorithm is dominated by the cost of sorting the edges: $O(m \log n)$. Each *Find* or *Union* takes $O(\alpha(n))$ amortized time, so the total cost of the $O(m)$ Find-Union operations is $O(m\alpha(n))$, which is dominated by sorting. The overall complexity of Kruskal's algorithm is therefore $O(m \log n)$.

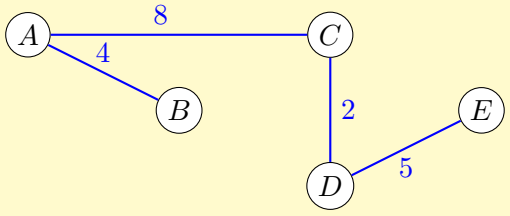
Example 8.6 Consider a graph with five vertices A, B, C, D, E and edges A-B: 4, A-C: 8, B-C: 11, B-D: 8, C-D: 2, C-E: 6, D-E: 5. We apply Kruskal's algorithm.



Edges in sorted order: C-D (2), A-B (4), D-E (5), C-E (6), A-C (8), B-D (8), B-C (11). Initially each vertex is its own set. Process each edge with *Find* and *Union*:

1. C-D (2): $Find(C) \neq Find(D)$, so $Union(C, D) \to \{A\}, \{B\}, \{C, D\}, \{E\}$.
2. A-B (4): $Find(A) \neq Find(B)$, so $Union(A, B) \to \{A, B\}, \{C, D\}, \{E\}$.
3. D-E (5): $Find(D) \neq Find(E)$, so $Union(D, E) \to \{A, B\}, \{C, D, E\}$.
4. C-E (6): $Find(C) = Find(E)$; skip.
5. A-C (8): $Find(A) \neq Find(C)$, so $Union(A, C) \to \{A, B, C, D, E\}$. MST complete; the remaining edges B-D (8) and B-C (11) would close cycles.

The MST consists of C-D (2), A-B (4), D-E (5), A-C (8), with total weight 19.



8.5 Prim's Algorithm

Prim's algorithm is also a greedy algorithm. It builds the minimum spanning tree by increasing the size of a single fragment by adding the minimum weight outgoing edge of the fragment.

It simply exploits Lemma 8.1 to increase the size of the fragment until it becomes the MST. At any stage, Prim's algorithm has a fragment F. It finds the minimum outgoing edge of that fragment e. This edge can be viewed as the edge from the fragment to its nearest neighbor. Therefore, this algorithm is sometimes also known as the *nearest-neighbor-next* algorithm. To find the nearest neighbor, every vertex v maintains a label d that corresponds to the cost of adding v to the fragment. At each iteration, the algorithm chooses the vertex v with the minimum value d and adds it to the fragment. The array $fixed$ keeps track of the vertices in the fragment. Whenever a new vertex v is *fixed* and added to the fragment, the d values for any adjacent vertex v' are updated as follows. We check whether the weight of the edge (v, v') is lower than the previous value of $d[v']$. If this is true, then $d[v']$ is updated to $w[v, v']$. We also use a *parent* pointer with each node, which keeps track of the node v that is responsible for the d value of v'.

Algorithm Prim: Finding a minimum spanning tree (MST)

input: Undirected Weighted Graph: $\mathcal{G} = (V, E, w)$
output: Minimum Weight Spanning Tree T

```
1  d: array[0..n − 1] of real initially ∞; //d[v] is the cost to add vertex v;
2  parent: array[0..n − 1] of int initially −1; // parent[v] is the node that corresponds to
     d[v] cost;
3  fixed : array[0..n − 1] of boolean initially false; //vertices whose d value is fixed;
4  T: set of edges initially {};
5  d[0] := 0;
6  while |T| < n − 1 do
7  |  v := arg min_i{d[i] | ¬fixed[i]};
8  |  if d[v] = ∞ then return null // no spanning tree in the graph;
9  |  if parent[v] ≠ −1 then add (v, parent[v]) to T ;
10 |  fixed[v] := true;
11 |  forall (v, v') ∈ E do
12 |  |  if ¬fixed[v'] ∧ w[v, v'] < d[v'] then
13 |  |  |  d[v'] := w[v, v'];
14 |  |  |  parent[v'] := v;
15 |  |  end
16 |  end
17 end
18 return T;
```

Example 8.7 Consider again the graph shown in Fig. 8.1. For Prim's algorithm, we start from a fixed node. Suppose that we start from the vertex a. Then the nearest neighbor is c with a cost of 4. The next nearest neighbor to the fragment with vertices $\{a, c\}$ is the vertex b. The cost of adding b is 3. At this point, we have the vertices $\{a, b, c\}$ in the fragment. The cost of adding vertex d is 7 and the cost of adding vertex e is 11. We add the vertex d to our fragment with the cost 7. Finally, e is added with the cost 2. Hence, the edges are added to the tree in the order $4, 3, 7, 2$. The set of edges chosen is identical to that of Kruskal's algorithm, even though the order in which they are chosen is different. This is not surprising because there is a unique minimum spanning tree when all edge weights are distinct (Problem 3).

Correctness. Prim's algorithm maintains the invariant that the fragment F (the set of fixed vertices together with T) is contained in some MST. Initially $F = (\{v_0\}, \emptyset)$, which is trivially contained in every MST. Suppose the invariant holds just before an iteration that fixes vertex v. By construction, the edge $(parent[v], v)$ is the lightest edge crossing the cut $(F, V \setminus F)$: its weight is $d[v]$, and for any other edge (u', v') with $u' \in F$, $v' \notin F$ we have $d[v'] \geq w(u', v')$ and $d[v] \leq d[v']$. By the cut property (Lemma 8.2), this edge belongs to some MST that extends F. After $n - 1$ iterations, F is a spanning tree, and hence an MST.

Complexity. The step of finding the vertex v with a minimum value d can be performed by simply traversing the array d or maintaining d in a heap. If we simply traverse the array d to find the minimum, the work complexity of the above algorithm is $O(n^2+m)$. In every iteration of the while loop, we perform $O(n)$ work to find the minimum, and there are $O(n)$ iterations of the loop. The work for processing edges over all iterations is $O(m)$ because every edge is processed at most once. Using a binary heap, we perform n extract-min operations and at most m decrease-key operations, yielding $O(m \log n)$ work complexity.

Comparison with Dijkstra's algorithm. Prim's and Dijkstra's algorithms (Chapter 7) share the same control structure: both maintain a priority-keyed set of frontier vertices, repeatedly extract the one with the smallest key, fix it, and relax the keys of its neighbors. The difference lies entirely in what the key means. In Prim, $d[v']$ is the weight of the *single* lightest edge from the current fragment to v', so the relaxation rule reads $d[v'] := \min(d[v'], w(v, v'))$. In Dijkstra, $d[v']$ is the length of the shortest path from the source to v' through already-fixed vertices, so the relaxation rule becomes $d[v'] := \min(d[v'], d[v] + w(v, v'))$. As a consequence, Prim's correctness rests on the cut property and holds for arbitrary real edge weights (including negative), whereas Dijkstra's correctness additionally requires that extending a path cannot shorten it, which is true only for non-negative weights.

8.6 Boruvka's Algorithm

In Prim's algorithm, we started with a trivial fragment that included only the vertex v_0. We kept increasing the size of the fragment until it became a spanning tree. In Boruvka's algorithm, we may have more than one fragment. We increase the size of all fragments by adding the minimum outgoing edge for each fragment.

Algorithm Boruvka presents the sequential Boruvka's algorithm to find the MST. We use T to denote the set of tree edges. Initially, T is empty. When we determine the components in

Algorithm Boruvka: Finding minimum spanning tree using Boruvka's algorithm

```
   input: undirected connected weighted graph: G = (V, E, w)
   output: minimum weight spanning tree T
 1 T: { set of edges } initially {};
 2 cid: array[1..n] of 0..n initially all 0;
 3 while |T| < n − 1 do
 4 |   visited: array[1..n] of boolean initially all false;
 5 |   mwe: array[1..n] of edge initially all null;
 6 |   dist: array[1..n] of real initially all ∞;
 7 |   for i := 1 to n do
 8 |   |   if (¬visited[i]) then // do a BFS in the graph (V, T) from vertex i setting cid
   |   |    of every visited vertex to i;
 9 |   |   BFS(i); ;
10 |   end
11 |   for (i, j) ∈ E such that (cid[i] ≠ cid[j]) do
12 |   |   if w[i, j] < dist[cid[i]] then
13 |   |   |   dist[cid[i]] = w[i, j];
14 |   |   |   mwe[cid[i]] = (i, j);
15 |   |   end
16 |   |   if w[i, j] < dist[cid[j]] then
17 |   |   |   dist[cid[j]] = w[i, j];
18 |   |   |   mwe[cid[j]] = (i, j);
19 |   |   end
20 |   end
21 |   forall i such that cid[i] = i do
22 |   |   T := T ∪ {mwe[i]}
23 |   end
24 end
25 return T;
```

(V, T), we find that there are n components, since each vertex is a component by itself when T is empty. The algorithm finds the minimum-weight outgoing edge for each component as follows. In any iteration, we use BFS to find the least numbered vertex to which any vertex is connected in the graph (V, T). This vertex serves as the identifier for the component of the node i, and we use the variable $cid[i]$ to store it. Once we have determined the component identity of all nodes, we move to the next step of determining the minimum weight outgoing edge for each component. We traverse all edges, and for each edge that connects two different components, we check whether it is cheaper than the previously known outgoing edge for the component on either side. Once we have determined all minimum weight edges for each component, we add these to T and start the next iteration.

Example 8.8 Consider the graph in Fig. 8.1. Initially, T is empty and there are 5 components. We compute mwe for each component and get the edges $4, 3, 3, 2, 2$ as the minimum weight edges of a, b, c, d, e, respectively. Once these edges are added, we have two components: $\{a, b, c\}$ and $\{d, e\}$. We then find mwe of these two components as the edge 7. On adding this edge, we have chosen $(n - 1)$ edges, and the algorithm terminates with the edges $\{2, 3, 4, 7\}$.

After every iteration, the number of connected components in (V, T) is reduced by at least a factor of two. This is because every component gets attached to some other component. The worst case is when every least weight edge chosen by any component is also chosen as the least weight edge by the component on the other side of the edge. Hence, the algorithm takes at most $O(\log n)$ iterations of the while loop. It is easy to see that the least weight edge outgoing from each component is found in $O(m)$ work. Thus, the algorithm takes $O(m \log n)$ work.

8.7 Comparison of Algorithms

All three algorithms of this chapter run in $O(m \log n)$ time, but they differ substantially in the data structures they rely on, the input formats they prefer, and the kind of parallelism they expose. This section summarizes those practical trade-offs so that a reader faced with a concrete instance can pick the right algorithm.

Input representation. Kruskal's algorithm is most natural when the edges are already available as a sorted list — the only operations it performs on the graph are "scan the next edge" and "are these two endpoints already connected". In particular, Kruskal's is well suited to streaming settings, where edges arrive one at a time in weight order and the graph need not be kept in memory. Prim's algorithm, on the other hand, needs random access to each vertex's neighbors and is the natural choice when the graph is represented as an adjacency list. Boruvka's algorithm operates on both edge and vertex sets and can be implemented with either representation.

Graph density. When the graph is very dense ($m = \Theta(n^2)$), the $O(m \log n)$ bound of all three algorithms is dominated by the edges themselves. Prim's algorithm implemented with an array-based scan (rather than a heap) achieves $O(n^2)$ work, which beats $O(m \log n) =$

$O(n^2 \log n)$ in this regime. For sparse graphs ($m = O(n)$), all three algorithms are within a $\log n$ factor of linear, and the $O(n^2)$ variant of Prim's becomes strictly worse than Kruskal's.

Required auxiliary structures. Kruskal's algorithm requires a union-find structure to test cycle-closing in $O(\alpha(n))$ time per query, plus a sort on the edge weights. Prim's algorithm requires a priority queue keyed on the cost-to-fragment of each fringe vertex. Boruvka's algorithm, by contrast, needs neither a global sort nor a priority queue: each round is a parallel scan of the edges followed by a connected-component relabeling.

Parallelism. This is where the three algorithms diverge the most. Kruskal's algorithm is inherently sequential: edges must be committed in weight order so that the cut property can be applied, and each commit may reject its predecessors in the sort. Prim's algorithm is also essentially sequential because it grows one fragment at a time from a single root, though modest parallelism is available in the edge-relaxation step. Boruvka's algorithm, finally, is the parallelism-friendly one: every component picks its minimum outgoing edge independently in each round, and the whole round can be carried out in $O(\log n)$ time with $O(m)$ processors. This is why the parallel MST algorithms in the literature — and the LLP-Boruvka algorithm of the next section — are all Boruvka variants.

Unifying principle. Despite their different strategies, all three algorithms construct an MST by repeatedly selecting edges that are certified safe by the Cut Property (Lemma 8.2). Kruskal's algorithm applies the cut property to the cut defined by the two components that the next lightest edge would join. Prim's algorithm applies it to the cut $(F, V \setminus F)$ separating the current fragment from the rest. Boruvka's algorithm applies it simultaneously to every current component's cut. The cycle property (Lemma 8.3) provides the complementary guarantee: any edge that is the heaviest in some cycle is safely excluded. Together, these two properties explain why all three algorithms are correct.

8.8 LLP Perspective

In this section we collect the LLP reformulations of the algorithms introduced earlier in the chapter.

LLP-Kruskal Algorithm

In this subsection, $G[j]$ is a Boolean indicating whether edge e_j is in the spanning tree.

We now give an LLP version for Kruskal's algorithm based on the simple Boolean lattice of all edges. Our distributive lattice is the Boolean lattice formed from all edges. We assume that the edges are given to us in increasing order. Then, an edge e_j between vertices v_k and v_l is in the minimum spanning tree iff there is no path between vertices v_k and v_l using edges $e_1 \ldots e_{j-1}$. Note that when j is 1, e_j corresponds to the lightest edge and is always in the minimum spanning tree. An LLP version is presented as Algorithm LLP-Kruskal.

Algorithm LLP-Kruskal: Finding the minimum spanning tree in a graph

input: undirected weighted graph: $\mathcal{G} = (V, E, w)$, edges in increasing order of weights
output: G: array$[1..m]$ of $\{0, 1\}$, indicating MST edges

```
1 var G: array[1..m] of {0,1};
2 init ∀j : G[j] := 0;;
3 forbidden(j): (G[j] = 0) and there exists no path from v_k to v_l with edges 1..j − 1
    for the edge e_j = (v_k, v_l)
4    ⇒ advance: G[j] := 1;
```

Example 8.9 Consider the graph in Fig. 8.1. Sorting the edges by weight gives

$$e_1 = (d, e, 2),\ e_2 = (b, c, 3),\ e_3 = (a, c, 4),\ e_4 = (a, b, 5),$$
$$e_5 = (b, d, 7),\ e_6 = (c, d, 9),\ e_7 = (c, e, 11).$$

Starting from $G = (0, 0, 0, 0, 0, 0, 0)$, an index j is forbidden when $G[j] = 0$ and e_j's endpoints are not yet connected by an edge e_i with $i < j$ and $G[i] = 1$.

- $j = 1$: no prior edges, so d and e are not connected. Advance $G[1] := 1$.
- $j = 2$: b and c are not connected by $\{e_1\}$. Advance $G[2] := 1$.
- $j = 3$: a and c are not connected by $\{e_1, e_2\}$. Advance $G[3] := 1$.
- $j = 4$: a and b are already connected via a–c–b using $\{e_2, e_3\}$. Not forbidden; $G[4]$ stays 0.
- $j = 5$: b and d are not connected by $\{e_1, e_2, e_3\}$. Advance $G[5] := 1$.
- $j = 6$: c and d are connected via c–b–d. Not forbidden.
- $j = 7$: c and e are connected via c–b–d–e. Not forbidden.

The algorithm terminates with $G = (1, 1, 1, 0, 1, 0, 0)$. The MST consists of the edges e_1, e_2, e_3, e_5, that is, $(d, e, 2)$, $(b, c, 3)$, $(a, c, 4)$, $(b, d, 7)$, with total weight 16.

LLP-Prim Algorithm

In this subsection, $G[v]$ stores the weight of the cheapest edge attaching v to the tree-so-far rooted at v_0. Because all edge weights are assumed distinct, the weight uniquely identifies the chosen edge.

We now show an LLP algorithm to find a minimum spanning tree rooted at a fixed vertex v_0. As before, we assume that all edge weights are unique. Every node other than v_0 has to choose an edge that we call the G edge. A node keeps all its edges in the sorted order and begins by choosing its least edge as its G edge. Initially the chosen edge of v may lead to a non-fixed neighbor, so following G-edges from v need not reach v_0; once v becomes *fixed* (defined below), the chain $v \to G\text{-edge}(v) \to \cdots$ reaches v_0 and $G[v]$ is the weight of the edge

by which v is attached to the tree-so-far. For the graph in Fig. 8.1, with root as the vertex a, we get the sorted adjacency lists shown in Fig. 8.3 for the other vertices.

Vertex	**Sorted adjacent edge weights**
b	3, 5, 7
c	3, 4, 9, 11
d	2, 7, 9
e	2, 11

Figure 8.3: Sorted adjacency lists for each non-root vertex of the graph in fig. 8.1, with root a.

We let $G[i]$ be the edge chosen by node i. Thus, initially, the vector G is $\{3, 3, 2, 2\}$. Observe that the set of all possible combinations of edges forms a distributive lattice. Since every node except v_0 has exactly one outgoing edge, following the chosen edge from every node, we either reach v_0, or end up in a cycle. We define a vertex as *fixed* if traversing the path starting from the edge proposed by that vertex leads to v_0. The vertex v_0 is trivially fixed. It is clear that the edge proposed by a non-fixed vertex can only lead to a non-fixed vertex, and the edge proposed by a fixed vertex can only lead to a fixed vertex. Thus, any G partitions the set of vertices into *fixed* and *non-fixed* vertices. In our example, the initial vector $\{3, 3, 2, 2\}$ makes the vertex a fixed and all other vertices $\{b, c, d, e\}$ non-fixed.

We define the set of edges between these two partitions as follows.

$$E'(G) := \{(i, k) \in E \mid fixed(i, G) \wedge \neg fixed(k, G)\}$$

In our example, we get the following edges as $E'(G) = \{(a, b), (a, c)\}$.

Let $x = (i, j)$ be the edge in E' with minimum $w[i, j]$. For our example, it is (a, c) with an edge weight of 4. We claim that the process j is forbidden in G. Consider any H such that $H[j] < w[i, j]$. Suppose, if possible, that H forms the minimum spanning tree. Any spanning tree must have an edge y between the set $fixed(G)$ and *non-fixed*(G). We claim that H cannot be the minimum spanning tree. This claim holds because the edge set $H - \{y\} + \{x\}$ is also a spanning tree and has a lower weight than H. Therefore, unless the process j advances to the edge (i, j), G cannot be the minimum spanning tree.

When we advance $G[j]$ to that edge, v_j becomes fixed. Observe that additional nodes may become fixed if their proposed edge leads to v_j directly or indirectly. The algorithm continues to advance G until all vertices become fixed or the edge set $E'(G)$ is empty. In the latter case, the graph is not connected and there is no minimum spanning tree.

Algorithm LLP-Prim shows the forbidden condition and the advance statement.

Algorithm LLP-Prim: Finding the minimum spanning tree in a graph

input: undirected weighted graph: $\mathcal{G} = (V, E, w)$
output: G: array$[1..n-1]$ of real, encoding MST edges
1 **var** G: array$[1..n-1]$ of real initially $\forall i : G[i] =$ minimum edge adjacent to i;
2 **always**;
3 $\quad fixed(j, G) \equiv$ there exists a directed path from v_j to v_0 using edges in G;
4 $\quad E' := \{(i,k) \in E \mid fixed(i, G) \wedge \neg fixed(k, G)\}$;
5 **forbidden**(j): $\exists i : (i,j) \in E'$ such that it has minimum weight $w[i,j]$ of all edges in E'
6 $\quad \Rightarrow$ **advance**: $G[j] := min\{w[i,j] \mid (i,j) \in E'\}$;

Example 8.10 Take the graph of Fig. 8.1 with root $v_0 = a$. Each non-root vertex j initializes $G[j]$ to the weight of its lightest adjacent edge, giving $G = (G[b], G[c], G[d], G[e]) = (3, 3, 2, 2)$, i.e., $b \to c$, $c \to b$, $d \to e$, $e \to d$. Only a is fixed; the other four vertices form two disconnected cycles, so none is fixed.
Iteration 1. The cross-cut set is $E' = \{(a, b, 5), (a, c, 4)\}$, with minimum $(a, c, 4)$, so $j = c$ is forbidden. Advance $G[c] := 4$. Now c is fixed, and b becomes fixed transitively via $b \to c \to a$. Fixed $= \{a, b, c\}$; $G = (3, 4, 2, 2)$.
Iteration 2. Now $E' = \{(b, d, 7), (c, d, 9), (c, e, 11)\}$, with minimum $(b, d, 7)$, so $j = d$ is forbidden. Advance $G[d] := 7$. Now d is fixed, and e becomes fixed transitively via $e \to d \to b \to c \to a$. Fixed $= \{a, b, c, d, e\}$; $G = (3, 4, 7, 2)$.
All vertices are fixed; the algorithm terminates with the MST edges

$$\{(b, c, 3),\ (a, c, 4),\ (b, d, 7),\ (d, e, 2)\},$$

total weight 16.

LLP-Boruvka Algorithm

In this subsection, $G[v]$ stores v's parent in the rooted tree: initially v's chosen lightest-edge endpoint, and ultimately — after pointer-jumping — the leader (root) of v's component.

In this section, we give a recursive version of the algorithm and implement each recursive instance of Boruvka's algorithm with the LLP algorithm. As before, we assume that the input to our algorithm is a connected graph without any self-loops. The LLP algorithm uses the single variable G. For each instance, we let the minimum weight edge (mwe) adjacent to each vertex v be denoted by $mwe[v]$. We denote this directed graph by M: the set of vertices is V and the set of edges is $\{(v, w) | (v, w) = mwe[v]\}$. We initialize $G[v]$ with the node w when $mwe[v] = (v, w)$ except when $mwe[w] = (w, v)$ and $v < w$. For the latter case, we make $G[v]$ equal to v. As a result of this initialization, the structure $G[v]$ forms a rooted tree. We say that the edge (v, w) is *incident* to v when $G[v]$ equals w for w different from v.

Once we have rooted trees for the graph, we convert the rooted trees to rooted stars using pointer-jumping. A node j is considered forbidden if $G[j] \neq G[G[j]]$. It is advanced by setting $G[j]$ to $G[G[j]]$. The algorithm stops this iteration when no node is forbidden. We show that each instance is indeed an LLP algorithm. First note that $G[v]$ is always reachable from v in

Algorithm LLP-Boruvka: Finding MST using Boruvka's algorithm via LLP

input: undirected *connected* weighted graph $\mathcal{G} = (V, E, w)$
output: minimum weight spanning tree T

```
1  function Boruvka(V, E):
2    if |V| = 1 then return {};
     var: G: array[1..n] of 1..n, initially G[v] = w such that mwe[v] = (v, w), except
          when mwe[w] = (w, v) and v < w; in that case G[v] := v
3    forbidden(j): G[j] ≠ G[G[j]]
4      ⇒ advance: G[j] := G[G[j]];
5    return T ∪ Boruvka(V',E') where;
6    T := {(v, w) | (v, w) is the minimum weight edge of v};
7    V' := {v ∈ V | G[v] = v};
8    E' := {(G[v], G[w]) | (v, w) ∈ E ∧ G[v] ≠ G[w]};
9      with parallel edges collapsed to the lightest one: for distinct components p, q,;
10     w'(p, q) := min{w(v, w) | (v, w) ∈ E, {G[v], G[w]} = {p, q}};
```

the directed graph M. Initially, this claim holds because $G[v]$ is set to w where (v, w) is an edge in M. The only statement executed in the program is $G[v] := G[G[v]]$ which preserves the claim.

We also observe that M is a collection of rooted trees and if w is reachable from v, then there is a unique path from v to w. Furthermore, because all edge weights are assumed distinct, the minimum outgoing edge selected at each vertex is unique, and therefore the weight of edges along any directed path is strictly decreasing. To see this, if v points to w and w points to u then the weight of the edge (w, u) must be strictly smaller than the weight of the edge (v, w) by the property that each vertex points to its minimum weight edge. We are now ready to show the following lemma.

Lemma 8.11 *The predicate $B \equiv \forall j : G[j] = G[G[j]]$ is lattice-linear.*

Proof: Suppose B is false, so there exists j with $G[j] \neq G[G[j]]$. Let $k = G[j]$. Since k is not the root (roots satisfy $G[r] = r$, which would give $G[j] = G[G[j]]$), and M is a rooted tree, the path from k to the root does not pass through k again. Since $G[k]$ always lies on this path (by the reachability invariant), we have $G[k] \neq k$ permanently. Now, advancing any other index i only changes $G[i]$, not $G[j]$, so $G[j]$ remains k. And $G[G[j]] = G[k] \neq k = G[j]$. Therefore j remains forbidden.

■

Lemma 8.12 *Each instance of finding connected components terminates.*

Proof: Each advance sets $G[j] := G[G[j]]$, moving j's pointer strictly closer to the root along the tree path. The distance from j to the root decreases by at least one with each advance

of j. Since the total distance summed over all vertices is finite and strictly decreasing, the algorithm terminates.

■

Algorithm LLP-Boruvka shows the entire algorithm. Observe that there are no synchronization requirements for any instance of the LLP algorithm. Any thread j that finds $G[j]$ different from $G[G[j]]$ can set it to $G[G[j]]$. In particular, $G[G[j]]$ may have changed between the observation of $(G[j] \neq G[G[j]])$ and setting of $G[j]$ to $G[G[j]]$, but the algorithm still works correctly.

Example 8.13 Consider the graph in Fig. 8.1. The minimum-weight adjacent edge of each vertex is

$$mwe[a] = (a, c, 4), \quad mwe[b] = (b, c, 3), \quad mwe[c] = (b, c, 3),$$
$$mwe[d] = (d, e, 2), \quad mwe[e] = (d, e, 2).$$

Vertices b and c share the same minimum edge (b, c), as do d and e via (d, e). Breaking each 2-cycle in favor of the smaller-numbered vertex, the initialization gives

$$G[a] = c, \quad G[b] = b, \quad G[c] = b, \quad G[d] = d, \quad G[e] = d,$$

which corresponds to two rooted trees: $\{a \to c \to b\}$ rooted at b, and $\{e \to d\}$ rooted at d. The contributed MST edges so far are $\{(a, c, 4), (b, c, 3), (d, e, 2)\}$.
Pointer jumping. Only a is forbidden, since $G[a] = c \neq b = G[G[a]]$. Advancing sets $G[a] := b$. Now every vertex points directly to its root: $G = (b, b, b, d, d)$, two rooted stars $\{a, b, c\} \to b$ and $\{d, e\} \to d$.
Recursive call. Contract each star to a single vertex, leaving the two-vertex graph $V' = \{b, d\}$ with the cross-star edges $(b, d, 7), (c, d, 9), (c, e, 11)$ reprojected onto (b, d) as parallel edges of weights $7, 9, 11$. Collapsing these parallel edges to the lightest one leaves the single edge $(b, d, 7)$. Both components' minimum outgoing edge is thus $(b, d, 7)$, yielding initialization $G[b] = b$, $G[d] = b$ — already a rooted star. The edge $(b, d, 7)$ is added, and the recursion terminates with a single vertex.

The final MST is $\{(d, e, 2), (b, c, 3), (a, c, 4), (b, d, 7)\}$, total weight 16.

8.9 Constrained Minimum Spanning Tree

In some applications a minimum spanning tree must be chosen subject to a side condition that the unconstrained algorithms of this chapter ignore. A network designer building a backbone may insist that certain already-installed links be reused; a road planner extending an existing network may be required to keep all current segments in service.

Mandatory edges

Let $M \subseteq E$ be a set of edges that *must* appear in the chosen spanning tree. The *mandatory-edge MST* problem asks for a minimum-weight spanning tree T with $M \subseteq T$. The problem is

feasible iff M is acyclic.

The key observation is that the mandatory edges cannot simply be composed in conjunction with LLP-Kruskal. The forbidden predicate of LLP-Kruskal for edge $e_j = (v_k, v_l)$ checks connectivity only through edges $e_1, \ldots, e_{j-1}$ (those with smaller index, i.e., lighter weight). If a mandatory edge e_i has a larger index than e_j, committing e_i establishes connectivity that e_j's forbidden predicate cannot see. As a result, a conjunction could commit an edge e_j that forms a cycle with a mandatory edge of higher index.

The correct approach is a two-phase algorithm: first commit all mandatory edges, then run LLP-Kruskal on the residual graph.

Algorithm Mandatory: Mandatory-edge commitment for MST

input: $M \subseteq E$: mandatory edges; $G[1..m]$: Boolean, initially all false

```
1 forall e_j ∈ M do
2 |   if e_j closes a cycle with {e_k : G[k]} then return "infeasible";
3 |   G[j] := true;
4 end
```

After Algorithm Mandatory commits all mandatory edges and updates the Find-Union structure accordingly, LLP-Kruskal is run on the full edge set. Its forbidden predicate now checks connectivity through *all* committed edges — both mandatory and non-mandatory — so no redundant edge is added. That is, the constrained algorithm is:

1. Run Mandatory(M, G). If infeasible, stop.
2. Run LLP-Kruskal(E, w, G) starting from the G produced by step 1.

This is equivalent to contracting the edges of M, computing an MST of the contracted graph, and uncontracting. The complexity is $O(m \log n)$: the mandatory phase commits at most $|M| \leq n - 1$ edges with $O(\alpha(n))$ per cycle test, and the Kruskal phase is $O(m \log n)$.

Theorem 8.14 *The mandatory-edge MST problem is feasible if and only if the mandatory-edge set M is acyclic. When feasible, running Mandatory followed by LLP-Kruskal returns the minimum-weight spanning tree containing all edges in M, in $O(m \log n)$ time.*

Proof: If M contains a cycle, any spanning tree including M would also contain a cycle, which is impossible. Conversely, if M is acyclic, committing all edges of M produces a forest. Running LLP-Kruskal on the residual completes this forest to a spanning tree by adding the lightest non-cycle-forming edges, which is exactly the MST of the graph contracted by M.

■

8.10 Summary

Three classical algorithms compute a minimum spanning tree, each exploiting the cut and cycle properties in a different way. *Kruskal's* algorithm scans edges in non-decreasing weight order and admits each edge that does not close a cycle; the Find-Union data structure reduces the cycle check to near-constant amortized time. *Prim's* algorithm grows a single fragment

from an arbitrary root, repeatedly adding the lightest edge that leaves the fragment — an instance of the nearest-neighbor-next strategy. *Boruvka's* algorithm extends all fragments in parallel: in each iteration, every current component contributes its minimum outgoing edge, so the number of components is at least halved, and the algorithm terminates in $O(\log n)$ iterations.

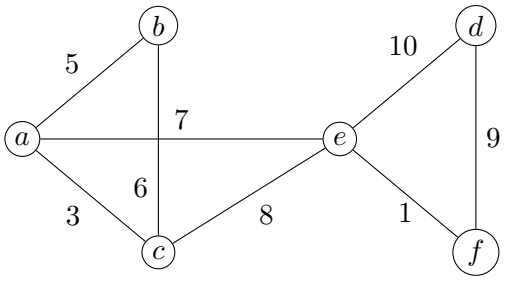
Table 8.1 summarizes the time complexity of the sequential algorithms discussed in this chapter.

Algorithm	**Time Complexity**
Kruskal	$O(m \log n)$
Prim	$O(m \log n)$
Boruvka	$O(m \log n)$
LLP-Kruskal	$O(m \log n)$
LLP-Prim	$O(m \log n)$
LLP-Boruvka	$O(m \log n)$

Table 8.1: Minimum spanning tree algorithms discussed in this chapter.

8.11 Problems

1. **(MST Algorithm Comparison)** Consider the undirected weighted graph on six vertices $\{a, b, c, d, e, f\}$ shown below.



   Find the minimum spanning tree using Kruskal's algorithm, Prim's algorithm, and Boruvka's algorithm.

2. **(Iterative Find with Path Compression)** Algorithm Find implements FIND recursively. Give an iterative version that performs path compression without using recursion or a stack.

3. **(Unique MST Distinct Weights)** Show that there is a unique minimum spanning tree in any weighted undirected graph when all edge weights are distinct.

4. **(Maximum Spanning Tree Negation)** Show that replacing every edge weight $w(e)$ by $-w(e)$ and then running Kruskal's (or Prim's) algorithm yields a *maximum* weight spanning tree of the original graph.

5. **(Bottleneck Spanning Tree)** A *bottleneck spanning tree* of a weighted graph is a spanning tree that minimizes the weight of its heaviest edge. Show that every minimum spanning tree is also a bottleneck spanning tree.

6. **(Edge Weight Decrease Update)** Let T be the MST of an undirected weighted graph $G = (V, E, w)$. Suppose the weight of exactly one edge $e = (u, v) \in E$ is decreased. Describe an $O(n)$-time algorithm to update T (whether or not e was in the original MST), and justify its correctness.

7. **(Excluded Edges MST)** An undirected weighted graph G has a set $F \subseteq E$ of *excluded* edges that must not appear in the spanning tree. Adapt Kruskal's algorithm to compute a minimum spanning tree of $(V, E \setminus F)$ (if one exists) and state a necessary and sufficient condition on F for a feasible spanning tree to exist.

8. **(Prim vs Dijkstra Weights)** Both Prim's algorithm and Dijkstra's algorithm repeatedly pick a vertex outside the current frontier whose "cost" is minimum and add it to the frontier. State precisely the *cost* each algorithm minimizes, and use this difference to explain why non-negative edge weights are irrelevant to Prim's correctness (it works for any real weights), whereas Dijkstra's requires non-negative edge weights.

9. **(Boruvka Halving Bound)** Show that in Boruvka's algorithm, regardless of implementation, the number of connected components strictly decreases by at least a factor of two after each iteration. Deduce that the algorithm terminates in at most $\lceil \log_2 n \rceil$ iterations.

10. **(Heaviest MST Edge Cut)** Assume all edge weights are distinct. Let T be the MST of $G = (V, E, w)$ and let $e \in T$ be the *heaviest* edge in T. Removing e from T partitions V into two components S and $V \setminus S$. Prove that e is the *lightest* edge crossing the cut $(S, V \setminus S)$ in G.

11. **(Incremental MST: Edge Insertion)** An MST T of $\mathcal{G} = (V, E, w)$ has been computed. A new edge $e = (u, v)$ of weight w_e is then inserted. Give an algorithm that updates T in $O(|V|)$ time, exploiting the cycle lemma.

12. **(Incremental MST: Edge Deletion)** An MST T of $\mathcal{G}$ has been computed and an MST edge $e = (u, v) \in T$ is then deleted. Give an algorithm that updates T in $O(|E|)$ time, exploiting the cut lemma.

13. **(Dynamic MST Under Weight Changes)** The weight of an edge e changes from w to w'. Describe an MST update that handles both increases and decreases, and discuss when a full rebuild is unavoidable.

14. **(Constrained MST With Mandatory Edges)** A subset $M \subseteq E$ of edges is mandatory: every output spanning tree must contain M. Reduce the problem to ordinary MST on a smaller graph.

15. **(Fair MST: Bottleneck and Egalitarian Trees)** Define the *bottleneck spanning tree* as a spanning tree whose maximum-weight edge is minimised. Show that every MST is a bottleneck spanning tree, but the converse need not hold. Sketch an $O(|V| + |E|)$ algorithm for the bottleneck problem.

8.12 Bibliographic Remarks

The minimum spanning tree problem has an unusually long and well-documented history. Otakar Borůvka published the first MST algorithm in 1926 in connection with an electrical-grid layout problem in Moravia [Bor26]; the paper also contains the first proof that the MST is unique when edge weights are distinct. A detailed historical survey and an English translation of Borůvka's original papers is given by Nešetřil, Milková, and Nešetřilová [NMN01]. The classical sequential algorithms bearing the names of Kruskal [Kru56] and Prim [Pri57] were developed independently in the 1950s. Prim's algorithm was in fact discovered earlier by Vojtěch Jarník in 1930, and later independently by Dijkstra [Dij59] as a specialization of his shortest-path algorithm — a coincidence that is not accidental: the two algorithms have the same control structure and differ only in the relaxation rule (see Section 8.5).

The efficient implementation of Kruskal's algorithm hinges on the union-find data structure. Tarjan [Tar75] proved the near-constant $O(\alpha(n))$ amortized bound for union-find with path compression and union by rank used in Algorithms Find and Union. An $O(m+n\log n)$ variant of Prim's algorithm, based on Fibonacci heaps, is due to Fredman and Tarjan [FT87]. For a textbook treatment of the three sequential algorithms and their complexity, see Cormen et al. [CLRS01] or Kleinberg and Tardos [KT06].

A line of subsequent research has pushed the sequential complexity below $O(m\log n)$. Karger, Klein, and Tarjan [KKT95] gave a randomized algorithm running in expected $O(m)$ time — the first linear-time MST algorithm in any model. On the deterministic side, Chazelle [Cha00] obtained $O(m\,\alpha(m,n))$, inverse-Ackermann in the edge count, using an intricate soft-heap data structure. Pettie and Ramachandran [PR02] then gave an optimal algorithm whose running time matches the minimum number of comparisons needed to decide any MST instance, though its exact asymptotic complexity remains open.

In the parallel and distributed setting, the textbook of Sanders, Mehlhorn, Dietzfelbinger, and Dementiev [SMDD19] is our reference for the parallel Borůvka algorithm of Section 8.6. The pointer-jumping contraction technique that lies at its heart was popularized for parallel graph algorithms by Tarjan and Vishkin [TV85]. Practical shared-memory implementations for sparse graphs are described by Bader and Cong [BC06]. The classical distributed MST algorithm of Gallager, Humblet, and Spira [GHS83] computes an MST in a message-passing model with $O(n\log n)$ message complexity.

The LLP formulations presented in this chapter recast the three classical algorithms as instances of a single lattice-linear predicate framework. The LLP-Prim algorithm in particular is due to Alves and Garg [AG22].

Chapter 9

Divide and Conquer

9.1 Introduction

Divide-and-conquer is one of the oldest and most widely used algorithm-design strategies. The idea is to break a problem into a small number of independent sub-problems of the same kind, solve the sub-problems recursively, and then combine their solutions to produce the solution of the original problem. The strategy predates modern computer science: von Neumann's merge sort (1945) is already a textbook example, and the approach underlies algorithms in virtually every area of computing — sorting, searching, numerical computation, computational geometry, and signal processing.

Three features make divide-and-conquer particularly attractive. First, the recursive structure often yields surprisingly tight complexity analyses through the Master Theorem (Section 9.2). Second, because the sub-problems are independent of each other, every divide-and-conquer algorithm is inherently parallel: the recursive calls can be carried out simultaneously on independent processors. Third, and more subtly, decomposing a problem into smaller sub-problems sometimes exposes algebraic identities, as in Karatsuba's multiplication and Strassen's matrix multiplication, that lower the asymptotic complexity below what naive approaches can achieve.

We have already seen two instances of this pattern in Chapter 4: Quicksort, which partitions an array about a pivot and recurses on each part, and Mergesort, which splits the array in half and merges the sorted halves. In both cases the two sub-problems are independent of each other, which makes the approach naturally parallel — a theme that recurs throughout this chapter.

Viewing a problem as finding an appropriate element in a distributive lattice, the divide-and-conquer approach corresponds to finding two appropriate elements in two sub-lattices independently (and in parallel), then combining them to find the required element in the original lattice. Section 9.9 formalizes this idea via the notion of a *splittable predicate*.

This chapter is organized as follows. Section 9.2 gives the Master Theorem for analyzing the time complexity of divide-and-conquer algorithms. Section 9.3 discusses the problem of determining nearest neighbors in the Euclidean plane. Section 9.4 discusses the problem of

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

counting inversions in an array. Section 9.5 describes a solution to the problem of computing the planar convex hull of a set of points. Section 9.6 describes Karatsuba's method for multiplying two natural numbers using the divide-and-conquer approach. Section 9.7 gives Strassen's method for matrix multiplication. Section 9.8 presents the Fast Fourier Transform and its application to polynomial multiplication. Section 9.9 introduces splittable predicates, the LLP-flavored framework that captures the divide-and-conquer pattern.

9.2 The Master Theorem

When we apply the divide-and-conquer paradigm, the time complexity of the algorithm is usually analyzed through a recurrence relation on the input size. The Master Theorem solves a broad family of such recurrences in closed form.

Theorem 9.1 (Master Theorem) *Let $a \geq 1$, $b > 1$, and $c \geq 0$, and suppose that $T(n) = a\,T(n/b) + O(n^c)$ with $T(1) = O(1)$. Then*

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } c < \log_b a, \\ O(n^c \log n) & \text{if } c = \log_b a, \\ O(n^c) & \text{if } c > \log_b a. \end{cases}$$

Here a is the number of sub-problems, b is the factor by which the sub-problem size shrinks, and $O(n^c)$ is the cost of work done outside the recursive calls.

Proof: Expanding the recurrence k times gives

$$T(n) = a^k\,T(n/b^k) + \sum_{i=0}^{k-1} O(a^i (n/b^i)^c) = a^k\,T(n/b^k) + n^c \sum_{i=0}^{k-1} r^i,$$

where $r = a/b^c$. The recursion terminates at $k = \log_b n$, where $a^k = n^{\log_b a}$ and $T(1) = O(1)$, so the leaf contribution is $O(n^{\log_b a})$. The remaining geometric sum in r is handled by three cases.

- If $c < \log_b a$ then $r > 1$; the sum is $\Theta(r^k)$, and $n^c r^k = n^{\log_b a}$, so $T(n) = O(n^{\log_b a})$ (work is concentrated at the leaves).
- If $c = \log_b a$ then $r = 1$; the sum is $\Theta(k) = \Theta(\log n)$, so $T(n) = O(n^c \log n)$ (work is balanced across levels).
- If $c > \log_b a$ then $r < 1$; the sum is $\Theta(1)$, so the root cost n^c dominates and $T(n) = O(n^c)$.

Substituting these bounds into the expansion gives the three cases of the theorem.

■

Fig. 9.1 illustrates the proof for Merge Sort, where $a = b = 2$ and $c = 1$. Every level of the recursion tree costs n (four levels for $n = 8$), and the $1 + \log_2 n$ levels together give $T(n) = O(n \log n)$. This is the balanced case $c = \log_b a$ of the theorem; in the other two cases the level costs form a geometric sequence rather than being constant, and either the leaves or the root dominate.

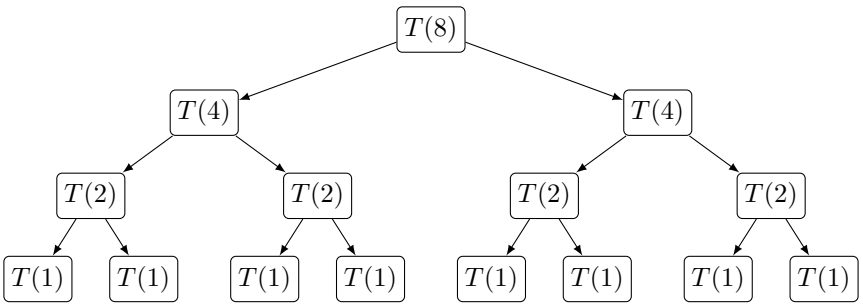


Figure 9.1: Recursion tree for $T(n) = 2T(n/2) + O(n)$ with $n = 8$. Every level costs $n = 8$, and there are $1 + \log_2 n = 4$ levels, giving $T(n) = O(n \log n)$.

Applications

We apply the Master Theorem to several recurrences that arise elsewhere in the book.

1. *Merge Sort.* $T(n) = 2T(n/2) + O(n)$: $a = 2$, $b = 2$, $c = 1 = \log_2 2$; case 2 gives $T(n) = O(n \log n)$.
2. *Binary Search.* $T(n) = T(n/2) + O(1)$: $a = 1$, $b = 2$, $c = 0 = \log_2 1$; case 2 gives $T(n) = O(\log n)$.
3. *Strassen's matrix multiplication.* $T(n) = 7T(n/2)+O(n^2)$: $a = 7$, $b = 2$, $c = 2 < \log_2 7 \approx 2.81$; case 1 gives $T(n) = O(n^{2.81})$.
4. *Karatsuba's algorithm.* $T(n) = 3T(n/2) + O(n)$: $a = 3$, $b = 2$, $c = 1 < \log_2 3 \approx 1.58$; case 1 gives $T(n) = O(n^{1.58})$.
5. *Case 3 example.* $T(n) = 2T(n/2) + O(n^2)$: $a = 2$, $b = 2$, $c = 2 > \log_2 2 = 1$; case 3 gives $T(n) = O(n^2)$.
6. *Case 1 example.* $T(n) = 9T(n/3) + O(n)$: $a = 9$, $b = 3$, $c = 1 < \log_3 9 = 2$; case 1 gives $T(n) = O(n^2)$.
7. *Case 2 example.* $T(n) = T(2n/3) + O(1)$: $a = 1$, $b = 3/2$, $c = 0 = \log_{3/2} 1$; case 2 gives $T(n) = O(\log n)$.

9.3 Nearest Neighbors in the Euclidean Space

We are given n points in the two-dimensional Euclidean space. Our goal is to find a pair of points with the smallest distance between them. That is, determine two points (p_i, p_j) such that the Euclidean distance between them is minimized:

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

The easiest algorithm would be to compute the distance between every pair of points and choose a pair with the least distance. This approach requires $O(n^2)$ sequential time. Our goal is to reduce the sequential time complexity for this problem. We present a divide-and-conquer algorithm that achieves $O(n \log^2 n)$ time; a classical refinement that maintains y-sorted order through the recursion reduces this to $O(n \log n)$, as noted at the end of the analysis.

The divide-and-conquer approach for this problem is as follows. We divide the set of points into two sets S_1 and S_2. Suppose that we have the nearest neighbors in S_1 and S_2. Now, if we had nearest neighbors such that one point is in S_1 and the second point is in S_2, then we simply have to choose the smallest of these three distances: nearest neighbors in S_1, nearest neighbors in S_2, and nearest neighbors across the partition. Suppose that the distance of the nearest neighbors in S_1, or in S_2 is δ. Then, we only need to consider the nearest neighbors across the partition if they are closer than δ. Let S_y be the set of points within δ of the dividing line sorted in the y-coordinate. We claim that if $s, s' \in S$ are such that $d(s, s') < \delta$, then s and s' are within 15 positions of each other in the sorted list S_y. Let L be the vertical line that divides the set of points according to the coordinate x. We consider boxes of size $\delta/2$ around L. We first show that there cannot be two points from the same side within a box. The largest distance within a box is $\sqrt{2}(\delta/2)$. This is equal to $\delta/\sqrt{2}$, which is less than δ.

We next claim that there cannot be two points that are sixteen positions apart in S_y and still have distance less than δ. Since the points are sixteen positions apart and at most one point in each box, they span at least four rows of boxes. There are at least two complete rows of boxes between them. This means that the vertical distance between them is at least $2 \cdot \delta/2 = \delta$, so the Euclidean distance is at least δ.

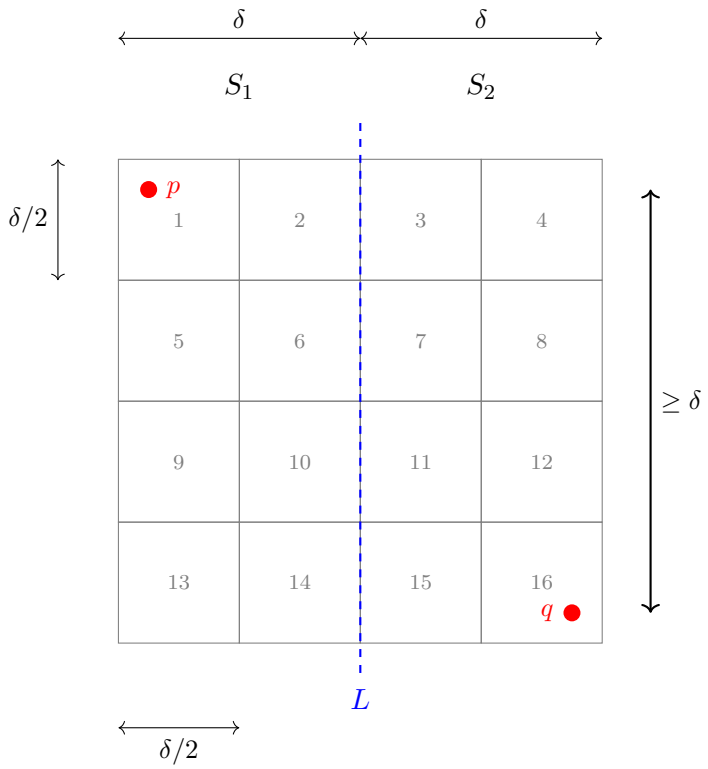


Figure 9.2: The δ-strip around the dividing line L, partitioned into $\delta/2 \times \delta/2$ boxes. Each box contains at most one point from its side. Numbers show the sorted-by-y positions. Points p (position 1) and q (position 16) span four rows with two complete rows between them, giving a vertical distance of at least δ.

Hence, it is sufficient to consider points that are at most 15 positions apart in S_y. (A tighter packing argument, using a $2\delta \times \delta$ rectangle that contains at most 8 points, reduces this bound from 15 to 7; we use the looser bound for simplicity.)

Algorithm ClosestPair: Closest pair of points

```
input: set of n points P in 2D space
output: the closest pair of points
Sort P by x-coordinate // presort once
Function ClosestPair(P):
    if n ≤ 3 then
        return BruteForce(P) // Base case: use brute-force for small
            input
    end
    Split P into left (P_L) and right (P_R) halves;
    (p_1, p_2) := ClosestPair(P_L);
    (p_3, p_4) := ClosestPair(P_R);
    best := min(d(p_1, p_2), d(p_3, p_4));
    S := points within best of the dividing line;
    Sort S by y-coordinate;
    for each point p in S do
        for each of the next 15 points q do
            if d(p, q) < best then
                Update best and closest pair;
            end
        end
    end
    return closest pair;
```

Example 9.2 Consider the set of points

$$P = \{(1,2), (3,7), (5,4), (8,9), (9,6), (10,3)\}.$$

Sorting. Arrange by x-coordinate: $[(1,2), (3,7), (5,4), (8,9), (9,6), (10,3)]$. *Divide.* Split into $P_L = [(1,2), (3,7), (5,4)]$ and $P_R = [(8,9), (9,6), (10,3)]$. *Recurse.* The closest pair in P_L is $\{(3,7), (5,4)\}$ with $d = \sqrt{13} \approx 3.61$; in P_R it is $\{(8,9), (9,6)\}$ with $d = \sqrt{10} \approx 3.16$. *Combine.* Let $\delta = \min(3.61, 3.16) = 3.16$ and place the dividing line at $x = 5$. Only $(3,7), (5,4), (8,9)$ lie within δ of it; the pair $(9,6)$ is outside the strip since $|9-5| = 4 > 3.16$. Checking strip pairs (sorted by y-coordinate) yields distances $\sqrt{13}, \sqrt{34}, \sqrt{29}$, none improving on δ. *Output.* The closest pair is $\{(8,9), (9,6)\}$ at distance $\sqrt{10} \approx 3.16$.

The time complexity is as follows. Presorting the points by x-coordinate takes $O(n \log n)$ time and is done once, outside the recursion. At each recursive level, the strip is sorted by y-coordinate in $O(n \log n)$ time and the strip scan takes $O(n)$ time, giving the recurrence $T(n) = 2T(n/2) + O(n \log n)$, which solves to $T(n) = O(n \log^2 n)$. The classical $O(n \log n)$ algorithm avoids re-sorting by maintaining y-sorted order through the recursion (as in Merge-

sort), reducing the combine step to $O(n)$.

9.4 Counting Inversions in an Array Using Divide and Conquer

Given an array $A[1 \ldots n]$, an **inversion** is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The goal is to count the total number of inversions in the array efficiently. A brute-force approach iterates through all pairs and checks if $A[i] > A[j]$, requiring $O(n^2)$ time. Using Merge Sort with an additional counting step, we can solve this problem in $O(n \log n)$ time.

We modify the *Merge Sort* algorithm to count inversions efficiently: (1) Recursively count inversions in the left and right halves. (2) Count the number of inversions across the two halves while merging. (3) Sum up all these counts.

Algorithm CountingInversions: Finding the number of inversions in an array

input: array A of size n
output: total number of inversions

```
1  Function CountInversions(A, left, right):
2      if left ≥ right then
3          return 0 // Base case: single element has no inversions
4      end
5      mid := (left + right)/2;
       // Sort the left half and count the inversions
6      invLeft := CountInversions(A, left, mid);
       // Sort the right half and count the inversions
7      invRight := CountInversions(A, mid + 1, right);
8      invMerge := MergeAndCount(A, left, mid, right);
9      return invLeft + invRight + invMerge;
10 Function MergeAndCount(A, left, mid, right):
11     i := left, j := mid + 1, k := 0, count := 0;
12     Create temp array temp of size (right − left + 1);
13     while i ≤ mid and j ≤ right do
14         if A[i] ≤ A[j] then
15             temp[k++] := A[i++];
16         end
17         else
18             temp[k++] := A[j++];
19             count := count + (mid − i + 1) // Count inversions
20         end
21     end
22     while i ≤ mid do
23         temp[k++] := A[i++];
24     end
25     while j ≤ right do
26         temp[k++] := A[j++];
27     end
28     Copy temp back to A[left, ..., right];
29     return count;
```

Example 9.3 Consider the array $A = [5, 3, 7, 1, 8, 2]$ of size 6. The recursion proceeds as follows.

- Left half $[5, 3, 7]$. Recursing further: $[5, 3]$ contributes the single inversion $(5, 3)$; $[7]$ contributes none. Merging the sorted halves $[3, 5]$ and $[7]$ adds no inversion since every left element is smaller than 7. Left-half total: 1.
- Right half $[1, 8, 2]$. Recursing further: $[1, 8]$ contributes none; $[2]$ contributes none. Merging $[1, 8]$ and $[2]$: $1 \leq 2$ is emitted, then $8 > 2$ contributes the single inversion $(8, 2)$. Right-half total: 1.
- Merging the sorted halves $[3, 5, 7]$ and $[1, 2, 8]$ while counting cross-half inversions: whenever an element from the right half is emitted with left elements still pending, the number of pending left elements counts as inversions. Emitting 1 leaves 3 left elements pending; emitting 2 leaves 3 left elements pending; the remaining comparisons emit from the left. Cross-half total: $3 + 3 = 6$.

The grand total is $1 + 1 + 6 = 8$ inversions. A direct enumeration of pairs (i, j) with $i < j$ and $A[i] > A[j]$ confirms 8: $(5, 3), (5, 1), (5, 2), (3, 1), (3, 2), (7, 1), (7, 2), (8, 2)$.

It is easy to see that the merge sort recursion takes $O(n \log n)$ time. The Merge step inversion counting takes $O(n)$ time. Thus, the overall time complexity is $O(n \log n)$.

9.5 Planar Convex Hull

Suppose that we are given S, a set of points in a plane. The convex hull of S is the smallest convex polygon that contains all points of S. The convex polygon can be described by the ordered list of points (in the clockwise direction) that defines the boundary of the polygon. Our task is to compute the convex hull of S.

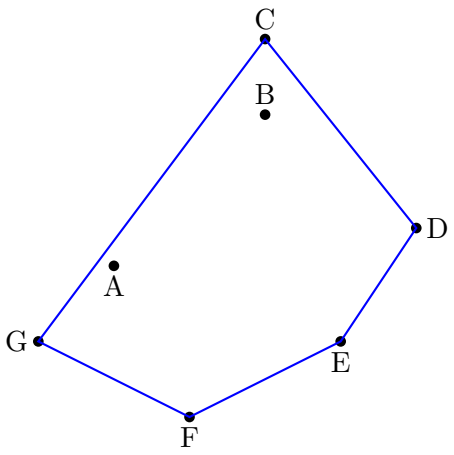


Figure 9.3: An example of planar convex hull

Let points $\{p_1, ..., p_n\}$ be sorted in their x-coordinate. For simplicity, we assume that x coordinates are unique. It is easy to verify that the point with the smallest x-coordinate, p_1, and the largest x-coordinate, p_n, are always in the convex hull. These two points also divide up the convex hull into two sets: the upper hull and the lower hull. The upper hull consists of all points in the convex hull starting from p_1 and ending at p_n (but not including p_n) in the clockwise direction. The lower hull consists of all points starting from p_n and ending at p_1 (but not including p_1).

To compute the convex hull of S, we divide the set of points into two sets S_1 and S_2 such that S_1 has first $n/2$ points with smaller x coordinate and S_2 has $n/2$ points with the larger x coordinate. Similar to mergesort, we can compute convex hulls of S_1 and S_2. The only task left is to merge the convex hull of S_1 and S_2 to get the convex hull of S. We first get the upper hulls of S_1 and S_2 and merge them to get the upper hull of S. This procedure can be repeated for lower hulls. To merge upper hulls, we need to determine the upper common tangent of $UH(S_1)$ and $UH(S_2)$. It is known that the upper common tangent can be determined sequentially in $O(n)$ time. Hence, we get the following recurrence:

$$T(n) \leq 2T(n/2) + O(n)$$

Solving this recurrence, we get $T(n) = O(n \log n)$.

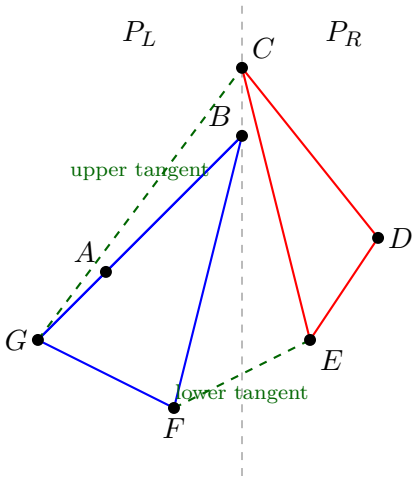


Figure 9.4: Divide-and-conquer for convex hull on the points of Fig. 9.3. The left subset $P_L = \{G, A, F, B\}$ has its hull shown in blue, and $P_R = \{C, E, D\}$ in red. The dashed green lines show the upper tangent (G–C) and lower tangent (F–E) used to merge the two hulls.

Example 9.4 Consider the seven points shown in Fig. 9.3: $A(0,0)$, $B(2,2)$, $C(2,3)$, $D(4,0.5)$, $E(3,-1)$, $F(1,-2)$, $G(-1,-1)$. Sorting by x-coordinate gives G, A, F, B, C, E, D, which splits into $P_L = \{G, A, F, B\}$ and $P_R = \{C, E, D\}$, as shown in Fig. 9.4. Recursing on P_L yields the hull G, F, B (the point A lies on edge G–B and is therefore not a hull vertex), and on P_R the hull is the triangle C, E, D. The upper common tangent of the two hulls connects G to C, and the lower common tangent connects F to E. Merging along the tangents gives the final hull C, D, E, F, G of five points, exactly the blue polygon of Fig. 9.3. The interior points A and B are discarded at the merge step.

9.6 Karatsuba's Multiplication Algorithm

A notable application of the divide-and-conquer approach is Karatsuba's multiplication algorithm, which computes the product of two large integers faster than the classical grade-school method. Introduced by Anatoly Karatsuba in 1962 [KO62], it reduces the complexity from $O(n^2)$ to $O(n^{\log_2 3})$, approximately $O(n^{1.585})$, where n is the number of digits.

Consider multiplying two polynomials $A(x) = a_0 + a_1 x$ and $B(x) = b_0 + b_1 x$, where coefficients are single-digit numbers (e.g., base-10 digits). Their product is:

$$C(x) = A(x) \cdot B(x) = a_0 b_0 + (a_0 b_1 + a_1 b_0)x + a_1 b_1 x^2.$$

Naive multiplication computes each term $(a_0 b_0, a_0 b_1, a_1 b_0, a_1 b_1)$, requiring 4 multiplications and 3 additions, yielding $O(n^2)$ for degree-n polynomials.

Now, represent two n-digit numbers $X = x_1 \cdot 10^{n/2} + x_0$ and $Y = y_1 \cdot 10^{n/2} + y_0$ (assuming n is even), where x_1, x_0, y_1, y_0 are $n/2$-digit numbers. Their product is:

$$X \cdot Y = (x_1 \cdot 10^{n/2} + x_0)(y_1 \cdot 10^{n/2} + y_0) = x_1 y_1 \cdot 10^n + (x_1 y_0 + x_0 y_1) \cdot 10^{n/2} + x_0 y_0,$$

akin to polynomial multiplication with $x = 10^{n/2}$. Naive computation requires 4 multiplications of $n/2$-digit numbers, suggesting $O(n^2)$ overall. Karatsuba's algorithm optimizes this to 3 multiplications.

Karatsuba's insight reduces the number of multiplications by computing: 1. $p_1 = x_0 y_0$, 2. $p_2 = x_1 y_1$, 3. $p_3 = (x_0 + x_1)(y_0 + y_1)$.

Then:

$$X \cdot Y = p_2 \cdot 10^n + [p_3 - (p_1 + p_2)] \cdot 10^{n/2} + p_1.$$

$p_3 = x_0 y_0 + x_0 y_1 + x_1 y_0 + x_1 y_1 = p_1 + p_2 + (x_1 y_0 + x_0 y_1)$, Thus, $p_3 - (p_1 + p_2) = x_1 y_0 + x_0 y_1$.

This requires only 3 multiplications (p_1, p_2, p_3) instead of 4, plus additional additions and subtractions, which are $O(n)$.

For n-digit numbers:

- Split: $X = x_1 \cdot 10^{n/2} + x_0$, $Y = y_1 \cdot 10^{n/2} + y_0$.
- Recursively compute p_1, p_2, p_3 on $n/2$-digit numbers.
- Combine using the formula above.

The recurrence is:

$$T(n) = 3T(n/2) + O(n),$$

solving to $T(n) = O(n^{\log_2 3})$ via the Master Theorem (Case 1: $a = 3, b = 2, \log_b a \approx 1.585$).

Algorithm Karatsuba: Karatsuba's multiplication algorithm

input: numbers X, Y, digit length n (power of 2)
output: product $X \cdot Y$

if $n = 1$ **then**
  return $X \cdot Y$ `// base case: single-digit multiplication`
end
$x_1 := \lfloor X/10^{n/2} \rfloor$, $x_0 := X \mod 10^{n/2}$ `// high and low halves of` X
$y_1 := \lfloor Y/10^{n/2} \rfloor$, $y_0 := Y \mod 10^{n/2}$ `// high and low halves of` Y
$p_1 := \text{Karatsuba}(x_0, y_0, n/2)$ `// low product`
$p_2 := \text{Karatsuba}(x_1, y_1, n/2)$ `// high product`
$p_3 := \text{Karatsuba}(x_0 + x_1, y_0 + y_1, \lceil n/2 \rceil + 1)$ `// middle term helper`
return $p_2 \cdot 10^n + [p_3 - (p_1 + p_2)] \cdot 10^{n/2} + p_1$

Example 9.5 Let $X = 1234$, $Y = 5678$, $n = 4$.
Divide. With $10^{n/2} = 100$: $x_1 = 12$, $x_0 = 34$, $y_1 = 56$, $y_0 = 78$.
Recurse. $p_1 = 34 \cdot 78 = 2652$, $p_2 = 12 \cdot 56 = 672$, $p_3 = (34+12)(78+56) = 46 \cdot 134 = 6164$.
Combine. $p_3 - (p_1 + p_2) = 6164 - 3324 = 2840$, so

$$X \cdot Y = 672 \cdot 10^4 + 2840 \cdot 10^2 + 2652 = 7006652,$$

matching the direct product $1234 \cdot 5678$.

Karatsuba's algorithm reduces multiplications from 4 to 3 per level, trading them for additions/subtractions. For $n = 2^k$:
- Levels: $k = \log_2 n$,
- Multiplications: $3^k = 3^{\log_2 n} = n^{\log_2 3}$,
- Additions: $O(n \log n)$.

Compared to FFT ($O(n \log n)$), Karatsuba is slower but avoids complex numbers, making it simpler for integer arithmetic. The example shows its practicality for small numbers, scaling efficiently for larger n.

9.7 Strassen's Matrix Multiplication

Strassen's algorithm is a divide-and-conquer approach to matrix multiplication that reduces the time complexity from the naive $O(n^3)$ to $O(n^{\log_2 7}) \approx O(n^{2.807})$ for multiplying two $n \times n$ matrices. Unlike the standard method, which performs 8 recursive multiplications for 2×2 submatrices, Strassen's method uses 7 multiplications by cleverly combining submatrix operations, trading some multiplications for additional additions.

Given two $n \times n$ matrices A and B, where n is a power of 2 (padded with zeros if necessary),

Algorithm Strassen: Strassen's matrix multiplication

input: A, B: $n \times n$ matrices, n a power of 2
output: C: $n \times n$ matrix, $C = A \cdot B$

if $n = 1$ **then**
  $C[1,1] := A[1,1] \cdot B[1,1]$;
  return C ;
end
Divide A into $A_{11}, A_{12}, A_{21}, A_{22}$; // Each $n/2 \times n/2$
Divide B into $B_{11}, B_{12}, B_{21}, B_{22}$; // Each $n/2 \times n/2$

$M_1 := \text{Strassen}(A_{11} + A_{22}, B_{11} + B_{22})$;
$M_2 := \text{Strassen}(A_{21} + A_{22}, B_{11})$;
$M_3 := \text{Strassen}(A_{11}, B_{12} - B_{22})$;
$M_4 := \text{Strassen}(A_{22}, B_{21} - B_{11})$;
$M_5 := \text{Strassen}(A_{11} + A_{12}, B_{22})$;
$M_6 := \text{Strassen}(A_{21} - A_{11}, B_{11} + B_{12})$;
$M_7 := \text{Strassen}(A_{12} - A_{22}, B_{21} + B_{22})$;

$C_{11} := M_1 + M_4 - M_5 + M_7$;
$C_{12} := M_3 + M_5$;
$C_{21} := M_2 + M_4$;
$C_{22} := M_1 - M_2 + M_3 + M_6$;

return $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$;

Strassen's algorithm divides each matrix into four $n/2 \times n/2$ submatrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}.$$

The product $C = A \cdot B$ is computed as:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},$$

where traditionally:

- $C_{11} = A_{11}B_{11} + A_{12}B_{21}$,
- $C_{12} = A_{11}B_{12} + A_{12}B_{22}$,
- $C_{21} = A_{21}B_{11} + A_{22}B_{21}$,
- $C_{22} = A_{21}B_{12} + A_{22}B_{22}$.

This requires 8 multiplications and 4 additions of $n/2 \times n/2$ matrices. Strassen's insight is to define 7 intermediate products (M_1 to M_7) that allow the computation of C with fewer multiplications:

The recurrence relation for Strassen's algorithm is:

$$T(n) = 7T(n/2) + O(n^2),$$

where 7 is the number of recursive multiplications, and $O(n^2)$ accounts for the additions and subtractions of $n/2 \times n/2$ matrices. Using the Master Theorem ($a = 7$, $b = 2$, $f(n) = O(n^2)$, $\log_b a = \log_2 7 \approx 2.807 > 2$):

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.807}).$$

This is an improvement over the standard $O(n^3)$.

Example 9.6 Consider the 2×2 matrices

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \qquad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}.$$

The seven Strassen products evaluate to

$$\begin{aligned} &M_1 = (1+4)(5+8) = 65, && M_2 = (3+4)\cdot 5 = 35, && M_3 = 1\cdot(6-8) = -2, \\ &M_4 = 4\cdot(7-5) = 8, && M_5 = (1+2)\cdot 8 = 24, && M_6 = (3-1)(5+6) = 22, \\ &M_7 = (2-4)(7+8) = -30. \end{aligned}$$

Combining them:

$$\begin{aligned} &C_{11} = M_1 + M_4 - M_5 + M_7 = 19, && C_{12} = M_3 + M_5 = 22, \\ &C_{21} = M_2 + M_4 = 43, && C_{22} = M_1 - M_2 + M_3 + M_6 = 50. \end{aligned}$$

Hence $C = A \cdot B = \left[\begin{smallmatrix} 19 & 22 \\ 43 & 50 \end{smallmatrix}\right]$, matching the direct product. The example uses 7 scalar multiplications rather than the textbook 8.

9.8 Fast Fourier Transform

Motivation: polynomial multiplication

Given two polynomials

$$A(x) = \sum_{i=0}^{n-1} a_i x^i \quad \text{and} \quad B(x) = \sum_{j=0}^{n-1} b_j x^j,$$

their product $C(x) = A(x) \cdot B(x) = \sum_{k=0}^{2n-2} c_k x^k$ has coefficients

$$c_k = \sum_{i+j=k} a_i \, b_j, \qquad k = 0, 1, \ldots, 2n-2.$$

Computing every c_k directly from the coefficient vectors $(a_0, \ldots, a_{n-1})$ and $(b_0, \ldots, b_{n-1})$ takes $\Theta(n^2)$ scalar multiplications. Polynomial multiplication is also the building block of integer multiplication (each long integer is a polynomial in the base, evaluated at the base) and of convolution (so the same algorithm computes the discrete convolution of two signals). The Fast Fourier Transform, due to Cooley and Tukey [CT65], reduces the cost to $O(n \log n)$ by exploiting an alternative representation of polynomials.

Two representations of a polynomial

A polynomial of degree at most $n-1$ can be specified in two equivalent ways:

- the *coefficient representation* $(a_0, a_1, \ldots, a_{n-1})$, and
- the *point-value representation* $\big((x_0, A(x_0)), (x_1, A(x_1)), \ldots, (x_{n-1}, A(x_{n-1}))\big)$ at any n distinct points $x_0, \ldots, x_{n-1}$.

Two distinct points determine a unique line, three distinct points determine a unique parabola, and in general n distinct points determine a unique polynomial of degree at most $n-1$ (this is the Lagrange interpolation theorem).

The coefficient representation makes it cheap to *evaluate* the polynomial at one point ($O(n)$ via Horner's rule) but expensive to multiply ($\Theta(n^2)$). The point-value representation reverses this: evaluation at the chosen sample points is free, multiplication is $O(n)$ (multiply pointwise), but recovering the coefficients of C at the end — the *interpolation* step — looks expensive at first sight.

The FFT's contribution is to choose a special set of evaluation points — the complex roots of unity — so that *both* the coefficient-to-point-value transform and its inverse can be computed in $O(n \log n)$ time.

Roots of unity

Let N be a power of two with $N \geq 2n-1$ (so that the product polynomial C of degree at most $2n-2$ is uniquely determined by N point values). The *N-th roots of unity* are the N complex numbers

$$\omega^0, \omega^1, \omega^2, \ldots, \omega^{N-1}, \qquad \omega = e^{2\pi i/N}.$$

They have three properties that make them the perfect evaluation set:

1. *Symmetry.* $\omega^{k+N/2} = -\omega^k$ for every k, since $\omega^{N/2} = e^{\pi i} = -1$.
2. *Recursive structure.* $(\omega^k)^2 = \omega^{2k}$, and the squares $\{\omega^{2k} : k = 0, \ldots, N-1\}$ are exactly the $(N/2)$-th roots of unity, each repeated twice.
3. *Orthogonality.* For $0 \leq j, k < N$, $\sum_{\ell=0}^{N-1} \omega^{\ell(j-k)} = N$ if $j = k$ and 0 otherwise.

Properties (1)–(2) drive the divide-and-conquer recursion below; property (3) provides a closed-form inversion formula.

The DFT and its inverse

The *discrete Fourier transform* of a coefficient vector $a = (a_0, \ldots, a_{N-1})$ is the vector $\hat{a} = (\hat{a}_0, \ldots, \hat{a}_{N-1})$ of values of A at the N roots of unity:

$$\hat{a}_k = A(\omega^k) = \sum_{j=0}^{N-1} a_j \, \omega^{jk}, \qquad k = 0, 1, \ldots, N-1.$$

By the orthogonality property, the *inverse* DFT is given by the same formula with $\omega^{-1} = \overline{\omega}$ in place of ω and a scaling by $1/N$:

$$a_j = \frac{1}{N} \sum_{k=0}^{N-1} \hat{a}_k \, \omega^{-jk}.$$

Thus, polynomial multiplication can be done as in Algorithm FFTMultiply: forward DFT both inputs, pointwise multiply, inverse DFT.

Algorithm FFTMultiply: Polynomial multiplication via FFT

input: coefficient vectors $a = (a_0, \ldots, a_{n-1})$ and $b = (b_0, \ldots, b_{n-1})$
output: coefficient vector $c = (c_0, \ldots, c_{2n-2})$ with $C(x) = A(x)B(x)$

$N \leftarrow$ smallest power of 2 with $N \geq 2n - 1$;
Pad a and b with zeros to length N;
$\hat{a} := \mathrm{FFT}(a, \omega)$; `// forward DFT,` $\omega = e^{2\pi i/N}$
$\hat{b} := \mathrm{FFT}(b, \omega)$;
for $k = 0$ **to** $N - 1$ **do**
  $\hat{c}_k := \hat{a}_k \cdot \hat{b}_k$; `// pointwise multiplication`
end
$c := \mathrm{FFT}(\hat{c}, \omega^{-1})$, then divide every entry by N; `// inverse DFT`
return $(c_0, c_1, \ldots, c_{2n-2})$;

The recursive FFT

To compute a DFT of size N in $O(N \log N)$ time, split the input by the parity of the index. Write

$$A(x) = A_e(x^2) + x \cdot A_o(x^2),$$

where

$$A_e(y) = \sum_{j=0}^{N/2-1} a_{2j}\, y^j, \qquad A_o(y) = \sum_{j=0}^{N/2-1} a_{2j+1}\, y^j.$$

Both A_e and A_o have size $N/2$. Evaluating A at the N points ω^k ($k = 0, \ldots, N-1$) reduces to evaluating A_e and A_o at the $N/2$ points ω^{2k}, which are the $(N/2)$-th roots of unity. The *butterfly identity* below uses the symmetry $\omega^{k+N/2} = -\omega^k$ to fold the answer back together:

$$\hat{a}_k = A_e(\omega^{2k}) + \omega^k\, A_o(\omega^{2k}), \qquad \hat{a}_{k+N/2} = A_e(\omega^{2k}) - \omega^k\, A_o(\omega^{2k}).$$

Algorithm FFT: Recursive fast Fourier transform

input: vector $a = (a_0, a_1, \ldots, a_{N-1})$ with N a power of 2; primitive root ω
output: vector $\hat{a} = (\hat{a}_0, \ldots, \hat{a}_{N-1})$ with $\hat{a}_k = \sum_j a_j \omega^{jk}$

if $N = 1$ **then**
  return (a_0)
end
$a^{(e)} := (a_0, a_2, \ldots, a_{N-2})$; $\quad a^{(o)} := (a_1, a_3, \ldots, a_{N-1})$;
$\hat{a}^{(e)} := \mathrm{FFT}(a^{(e)}, \omega^2)$; `// recurse on even-indexed`
$\hat{a}^{(o)} := \mathrm{FFT}(a^{(o)}, \omega^2)$; `// recurse on odd-indexed`
$z := 1$; `// the running power` ω^k
for $k = 0$ **to** $N/2 - 1$ **do**
  $\hat{a}_k := \hat{a}_k^{(e)} + z \cdot \hat{a}_k^{(o)}$;
  $\hat{a}_{k+N/2} := \hat{a}_k^{(e)} - z \cdot \hat{a}_k^{(o)}$;
  $z := z \cdot \omega$;
end
return $(\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{N-1})$;

The recurrence is $T(N) = 2\,T(N/2) + O(N)$, which by the Master Theorem (Theorem 9.1, balanced case) yields $T(N) = O(N \log N)$. The same algorithm computes the inverse DFT when invoked with ω^{-1} in place of ω, after which every output entry is divided by N.

Why this beats $\Theta(n^2)$

The naive coefficient-multiplication algorithm uses $\Theta(n^2)$ scalar multiplications and additions because each output coefficient depends on a different subset of input pairs. By moving to point-value representation, multiplication becomes pointwise — only N scalar multiplications — and *all* the work is funnelled into the two DFTs and one inverse DFT. The roots of unity are a particularly useful evaluation set whose recursive structure makes those DFTs themselves cheap.

Example 9.7 Let $A(x) = 1 + 2x$ and $B(x) = 1 + 3x$. The direct product is $C(x) = 1 + 5x + 6x^2$. We use $N = 4$, $\omega = i$, so the four roots of unity are $\{1, i, -1, -i\}$. Forward DFTs:

k	0	1	2	3
ω^k	1	i	-1	$-i$
$A(\omega^k)$	3	$1+2i$	-1	$1-2i$
$B(\omega^k)$	4	$1+3i$	-2	$1-3i$
$C(\omega^k)$	12	$-5+5i$	2	$-5-5i$

Inverse DFT: $c_k = \frac{1}{4}\sum_{\ell=0}^{3} C(\omega^\ell)\,\omega^{-k\ell}$. For instance, $c_0 = \frac{1}{4}(12+(-5+5i)+2+(-5-5i)) = \frac{4}{4} = 1$, and similarly $c_1 = 5$, $c_2 = 6$, $c_3 = 0$. We recover $C(x) = 1 + 5x + 6x^2$, agreeing with the direct computation.

To see Algorithm FFT concretely, consider the forward DFT of $A = (1, 2, 0, 0)$. The split is $a^{(e)} = (1, 0)$ and $a^{(o)} = (2, 0)$. Recursing on each (with $\omega^2 = -1$): the size-2 DFT of $(1, 0)$ is $(1, 1)$ and of $(2, 0)$ is $(2, 2)$. The butterfly with $z = 1, i$ then gives

$$\hat{a}_0 = 1 + 1 \cdot 2 = 3, \quad \hat{a}_1 = 1 + i \cdot 2 = 1 + 2i, \quad \hat{a}_2 = 1 - 1 \cdot 2 = -1, \quad \hat{a}_3 = 1 - i \cdot 2 = 1 - 2i,$$

matching the table above.

The FFT is one of the most widely used algorithms in scientific computing: it underlies fast integer multiplication (Schönhage–Strassen), large-scale signal processing (where the DFT converts time-domain samples to frequency-domain amplitudes), and numerical solution of partial differential equations.

9.9 Splittable Predicates

Divide-and-conquer admits a uniform LLP-flavored formulation: a problem on a lattice L is reduced to two independent subproblems on lattices L_1 and L_2, whose solutions combine to give the solution on L. We call a lattice-linear predicate *splittable* if such a decomposition exists.

Let L be a finite distributive lattice and B be any lattice-linear predicate. We call B splittable if there exist two lattices L_1 and L_2 and two lattice-linear predicates B_1 and B_2

such that given the least solution for B_1 and B_2 in L_1 and L_2, one can determine the least solution for B in L. The effort to determine the least solution for B in L is reduced if we have the solutions for B_1 and B_2. If B_1 and B_2 are also splittable, then we can apply this strategy recursively. Once the lattice is trivial, i.e., it is totally ordered, then finding the least solution is simple. Formally,

Definition 9.8 (Splittable Lattice-Linear Predicate) *A nonempty Lattice-Linear Predicate B is splittable for a finite distributive lattice L, if there exist two nonempty lattices L_1 and L_2 and two lattice-linear predicates B_1 and B_2 such that there exists a map $f : L_1 \times L_2 \to L$, where $f(G_1, G_2)$ is the least element satisfying B in L such that G_1 and G_2 are the least elements in L_1 and L_2 satisfying B_1 and B_2.*

Note that this definition models the subset of divide-and-conquer algorithms in which the combine phase depends only on the least solutions of the sub-problems. Some divide-and-conquer algorithms (e.g., closest pair, convex hull) require additional information from the sub-problems beyond their final solutions; these are not directly captured by this definition but can be accommodated by enriching the state in L_1 and L_2.

As an example of a predicate that is not splittable, consider the stable marriage problem. Given the distributive lattice of all assignments, the predicate B that the assignment is a stable marriage is not splittable. The predicate B that is true on all elements is trivially splittable into B_1 and B_2 also as true elements. The lattice L can be split into arbitrary $L_1 \times L_2$. Nontrivial splittable predicates underlie the divide-and-conquer approach to algorithm design.

9.10 Summary

In this chapter, we have shown that many problems can be solved using the divide-and-conquer approach. The following table summarizes some of the problems discussed in this chapter.

Problem	Algorithm	Time Complexity
Sorting	MergeSort	$O(n \log n)$
Nearest Neighbors	ClosestPair	$O(n \log^2 n)$
Counting Inversions	CountingInversions	$O(n \log n)$
Convex Hull	Divide-Conquer	$O(n \log n)$
Integer Multiplication	Karatsuba	$O(n^{\log_2 3})$
Matrix Multiplication	Strassen's Algorithm	$O(n^{\log_2 7})$
Polynomial Multiplication	FFT	$O(n \log n)$

Table 9.1: Divide-and-Conquer algorithms

9.11 Problems

1. **(Master Theorem Recurrences)** Use the Master Theorem to solve the following recurrences. In each case, state which case of the Master Theorem applies.

(a) $T(n) = 9\,T(n/3) + n$
(b) $T(n) = 2\,T(n/2) + n \log n$
(c) $T(n) = 3\,T(n/4) + n \log n$
(d) $T(n) = 7\,T(n/2) + n^2$

2. **(Randomized Quickselect)** Give a divide-and-conquer algorithm to find the k-th smallest element of an unsorted array $A[1..n]$ in expected $O(n)$ time. Prove the expected time bound.

3. **(Skyline Problem)** Consider the *skyline problem*: given n rectangular buildings defined by triples (l_i, h_i, r_i) (left, height, right), compute the silhouette they form. Give a divide-and-conquer algorithm and its complexity.

4. **(Mergesort Stability)** Prove or disprove: the standard top-down mergesort algorithm (recursive merge-sort that splits in the middle and merges two sorted halves) is *stable*, i.e., it preserves the relative order of equal elements.

9.12 Bibliographic Remarks

The general divide-and-conquer strategy and the Master Theorem are thoroughly treated in [CLRS09]. Strassen's matrix multiplication (Section 9.7), first proposed by [Str69], reduces the complexity of a matrix multiplication of size $n \times n$ from $O(n^3)$ to $O(n^{2.807})$. Karatsuba's multiplication introduced by [KO62] reduces the complexity of multiplying two n digit numbers from $O(n^2)$ to $O(n^{1.585})$. The Closest Pair problem uses a divide-and-conquer approach from [SH75]. The version presented in this chapter runs in $O(n \log^2 n)$; the classical refinement, which maintains y-sorted order through the recursion, achieves $O(n \log n)$. Counting Inversions via Mergesort, achieving $O(n \log n)$, is a standard application of the divide-and-conquer approach. The Planar Convex Hull divide-and-conquer algorithm is based on [PH77], with its $O(n \log n)$ time complexity. Graham's scan [Gra79] is an alternative $O(n \log n)$ algorithm that is not based on divide and conquer. For further reading, we refer the readers to [Hoa62], [Str69], [KO62], [SH75], [PH77], and [Gra79].

Chapter 10

Dynamic Programming

10.1 Introduction

Dynamic programming, introduced by Bellman in 1952 [Bel52], is a method for solving problems whose recursive solution involves overlapping sub-problems. Naive recursion recomputes the same sub-problems exponentially many times; dynamic programming records each sub-problem's answer in a table the first time it is computed and reuses it thereafter, replacing exponential redundancy with polynomial work. The name predates computer science: Bellman coined "dynamic programming" to describe a method for time-staged decision processes; today the term covers any algorithm built around a recurrence over a memoized table.

Three ingredients distinguish dynamic-programming problems from generic recursive ones. First, the optimal solution must satisfy a *principle of optimality*: an optimal solution to the whole problem extends an optimal solution to a smaller sub-problem. Second, the sub-problems must *overlap* so that memoization actually saves work; if every recursive call is on a fresh sub-problem, divide-and-conquer suffices and the memo table is wasted. Third, the recursion must terminate, so the sub-problem space is finite and admits an order in which the table can be filled bottom-up.

The lattice-linear predicate (LLP) framework introduced in Chapter 2 reformulates these problems as searches for the least vector G in a finite distributive lattice that satisfies a lattice-linear predicate B. Dynamic-programming recurrences supply the recurrence; LLP supplies a parallel scheduler. Where the dynamic-programming algorithm fills its table in a fixed bottom-up order, the LLP algorithm advances any forbidden index in parallel, and reaches the same fixed point. The LLP view also makes it easy to add lattice-linear constraints to a problem (a feature exercised in the constrained scheduling and constrained binary-search-tree problems below) without changing the algorithm.

This chapter is organized as follows. Section 10.2 contrasts recursion with dynamic programming on the Fibonacci recurrence, motivating memoization. Section 10.3 develops weighted interval scheduling. Section 10.4 presents the longest increasing subsequence problem. Section 10.5 gives the optimal binary search tree problem. Section 10.6 addresses chain matrix multiplication. Section 10.7 treats the knapsack problem. Section 10.8 collects the LLP

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

formulations of the algorithms in this chapter.

10.2 Recursion vs Dynamic Programming

Dynamic programming is applicable to problems in which it is easy to set up a recurrence relation so that the solution of the problem at hand can be derived from the solutions to problems with smaller sizes. The problem can be solved using recursion; however, recursion can result in many duplicate computations.

As a simple example, suppose that our goal is to compute the Fibonacci number F_i such that it satisfies the following conditions:

- $F_0 = 1$
- $F_1 = 1$
- $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$

A recursive algorithm is as follows:

Algorithm RecFib: Recursive Fibonacci function

```
   input: n: integer
   output: F(n)
1  function F(n):
2  |  if n < 2 then return 1 ;
3  |  else return F(n − 1) + F(n − 2) ;
```

Example 10.1 To compute $F(4)$, the recursive algorithm calls $F(3)$ and $F(2)$. Now $F(3)$ in turn calls $F(2)$ and $F(1)$, so $F(2)$ is computed twice. As n grows the redundancy explodes: the recursion tree of $F(n)$ has $\Theta(\phi^n)$ nodes (where ϕ is the golden ratio), although only $n + 1$ distinct sub-problems exist.

Using *memoization* for dynamic programming, we avoid such duplicate computations. Memoization is used to recall previously computed values.

An implementation using memoization is as follows:

Algorithm RecFibMemo: Recursive Fibonacci with memoization

```
   input: n: integer
   output: Fibonacci(n)
1  var A: array[0..N] of int initially ∀i : A[i] = −1
2  function F(n):
3  |  if A[n] ≠ −1 then return A[n] ;
4  |  else if n < 2 then A[n] ← 1 ;
5  |  else A[n] ← F(n − 1) + F(n − 2) ;
6  |  return A[n] ;
```

Alternatively, instead of computing $F(n)$ in a top-down manner, one can compute it in a bottom-up fashion. Assume that we keep an array A such that $A[i]$ will eventually store

$F(i)$. We start by filling in the array A from the lower to higher indices. The recurrence relation guarantees that if we have two previous entries, we can fill any entry $A[i]$. Thus, we can compute $F(n)$ in $O(n)$ time.

In this example, we had a simple array. For more nontrivial examples, we may have two- or three-dimensional tables.

10.3 Weighted Interval Scheduling

Recall the problem of Interval Scheduling from Section 6.2. Suppose that we have n intervals with start times s_i and finish times f_i for jobs $i = 1..n$, where $s_i < f_i$. We assume that activity i occurs during the open interval $[s_i, f_i)$. Two intervals i and j are *compatible* if $f_i \leq s_j$ or $f_j \leq s_i$. Our goal is to select a maximum-sized subset of mutually compatible intervals. We assume that the intervals are sorted by their finish times. We assume that jobs with the same finish times are sorted according to their start times. Furthermore, no two intervals have identical start and finish times. For this problem, we know that a greedy algorithm works. The interval that finishes earliest and starts after the last chosen interval is always selected.

Now consider a generalization of the problem in which each interval is assigned a *weight* denoting its priority. We would like to compute a subset of intervals such that they are non-overlapping and have the maximum weight. In this generalization, the greedy algorithm does not work.

As before, we assume that the intervals are sorted according to their *finish* times. Let $opt[j]$ denote the optimal value that can be obtained from the intervals $1 \ldots j$. When j is zero, the set of intervals selected is empty and $opt[0]$ is zero. Let $p[j]$ be the largest interval before the interval j which is disjoint from the interval j. Then we can set up a recurrence relation as follows. The maximum value we can obtain from the intervals $1 \ldots j$ is equal to the maximum of two values obtained by including the interval j or not including the interval j. If we include the interval j, then the value obtained is $w_j + opt[p[j]]$. If we do not include the interval j, then the value obtained is $opt[j-1]$. Thus, we have

$$opt[j] = \max\{w_j + opt[p[j]], opt[j-1]\}$$

This recurrence relation allows us to obtain the algorithm for weighted interval scheduling as shown in Fig. WeightedIntervalScheduling.

The sequential algorithm takes $O(n)$ time when the p array is available. This time complexity is optimal. Computing the p array takes $O(n \log n)$ time.

Algorithm WeightedIntervalScheduling: Sequential weighted interval scheduling

```
1  // jobs are sorted by their finish times;
   input: s: array[1...n] of int (start times); f: array[1...n] of int (finish times); w:
          array[1...n] of int (weight of the interval)
   output: G: array[1...n] of 0..1, indicating selected intervals; opt: array of optimal
           values
2  var G: array[1...n] of 0..1 initially 0;
3  p: array[1...n] of int ;                          // previous non-overlapping interval
4  opt: array[0...n] of int ;                                          // optimal value
5  // p[j] is the largest interval i such that f[i] ≤ s[j];
6  // Compute all p[j] in O(n log n) time using binary search on the sorted finish times
7  // Phase 1: compute optimal values;
8  opt[0] := 0;
9  for int cur := 1 to n do
10 |  opt[cur] := max(w[cur] + opt[p[cur]], opt[cur − 1]);
11 end
12 // Phase 2: backtrack to find selected intervals;
13 cur := n;
14 while cur > 0 do
15 |  if w[cur] + opt[p[cur]] ≥ opt[cur − 1] then
16 |  |  G[cur] := 1;
17 |  |  cur := p[cur];
18 |  else
19 |  |  cur := cur − 1;
20 |  end
21 end
```

Example 10.2 Consider the following 5 intervals, sorted by finish time:

Interval	s_i	f_i	w_i
1	1	3	4
2	2	5	6
3	4	6	5
4	6	8	3
5	5	9	7

The intervals are shown in Fig. 10.1. We first compute $p[i]$ (the last nonoverlapping interval before i): $p[1] = 0$, $p[2] = 0$ ($f_1 = 3 > s_2 = 2$), $p[3] = 1$, $p[4] = 3$, $p[5] = 2$. Then we compute $opt[i]$:

- $opt[0] = 0$.
- $opt[1] = \max(opt[0], w_1 + opt[p[1]]) = \max(0, 4) = 4$ (select 1).
- $opt[2] = \max(opt[1], w_2 + opt[p[2]]) = \max(4, 6) = 6$ (select 2).
- $opt[3] = \max(opt[2], w_3 + opt[p[3]]) = \max(6, 5 + 4) = 9$ (select 1, 3).
- $opt[4] = \max(opt[3], w_4 + opt[p[4]]) = \max(9, 3 + 9) = 12$ (select 1, 3, 4).
- $opt[5] = \max(opt[4], w_5 + opt[p[5]]) = \max(12, 7 + 6) = 13$ (select 2, 5).

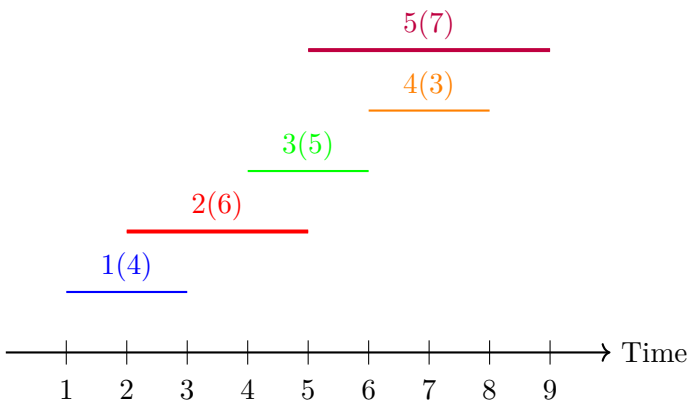


Figure 10.1: Weighted intervals (weights in parentheses) with optimal selection (thick lines).

10.4 Longest Increasing Subsequences

The Longest Increasing Subsequence (LIS) problem involves finding a subsequence of a given array of integers, where the elements are in strictly increasing order and the subsequence is as long as possible. A subsequence is derived by deleting zero or more elements from the original array without changing the order of the remaining elements.

For example, in the array $[10, 9, 2, 5, 3, 7, 101, 18]$, one possible LIS is $[2, 5, 7, 101]$ with length 4. The problem has applications in data analysis, sequence alignment, and more. We present an iterative dynamic programming solution to compute the LIS length efficiently.

Our problem can be stated as follows. Given an array $A = [a_1, a_2, \ldots, a_n]$ of n integers, find the length of the longest subsequence $[a_{i_1}, a_{i_2}, \ldots, a_{i_k}]$ such that:

$$i_1 < i_2 < \cdots < i_k \quad \text{and} \quad a_{i_1} < a_{i_2} < \cdots < a_{i_k}.$$

We use an iterative dynamic programming approach to solve the LIS problem. Define $dp[i]$ as the length of the longest increasing subsequence ending at index i (including a_i).

The idea is to build the dp array iteratively by comparing each element with all the previous elements.

- Initialize $dp[i] = 1$ for all i (each element is an LIS of length 1 itself).
- For each i, check all $j < i$. If $a[j] < a[i]$, update $dp[i]$ as $\max(dp[i], dp[j] + 1)$.
- The length of the LIS is the maximum value in the dp array.

Algorithm LIS: Longest increasing subsequence (iterative)

input: array A of n integers
output: length of the LIS

```
1 dp: array [0...n − 1] of int initially ∀i : dp[i] = 1;
2 for i = 1 to n − 1 do
3 |   for j = 0 to i − 1 do
4 |   |   if A[j] < A[i] then  dp[i] = max(dp[i], dp[j] + 1) ;
5 |   end
6 end
7 return max(dp[0], dp[1], ..., dp[n − 1]);
```

$dp[i]$ represents the longest increasing subsequence ending in $A[i]$, considering all prior elements. By checking all $j < i$, we ensure that every possible subsequence ending at i is evaluated. The maximum $dp[i]$ gives the overall length of the LIS, as it captures the longest chain possible.

Example 10.3 Consider the array $A = [3, 10, 2, 1, 20]$. We compute $dp[i]$ for each index:

- $A[0] = 3$, $dp[0] = 1$.
- $A[1] = 10$, $dp[1] = 2$.
- $A[2] = 2$, $dp[2] = 1$.
- $A[3] = 1$, $dp[3] = 1$.
- $A[4] = 20$, $dp[4] = 3$.

The final $dp = [1, 2, 1, 1, 3]$, the LIS length is $\max(dp) = 3$, and an LIS is $[3, 10, 20]$, as shown in Fig. 10.2.

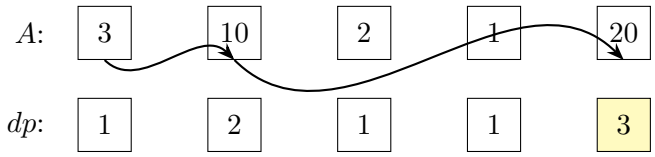


Figure 10.2: Longest increasing subsequence for $A = [3, 10, 2, 1, 20]$. The highlighted $dp[4] = 3$ indicates the LIS length. Arrows show one possible LIS: $[3, 10, 20]$.

The time complexity is $O(n^2)$ and the space complexity is $O(n)$ for the dp array.

10.5 Optimal Binary Search Tree

The Optimal Binary Search Tree problem involves constructing a binary search tree (BST) for n symbols that minimizes the expected search cost, given the probability of each symbol's occurrence. Each symbol i (ranging from 0 to $n-1$) has a probability $p[i]$ of being searched, and these probabilities sum to 1, i.e., $\sum_{i=0}^{n-1} p[i] = 1$. The objective is to arrange the symbols in a BST to minimize the total cost, defined as the sum of each symbol's probability multiplied by its depth, where the root has depth 1. For example, given three symbols A, B, and C with probabilities $p = [0.4, 0.3, 0.3]$, an optimal BST might place B as the root to reduce the average search time. This problem finds applications in areas like database indexing and compiler design, where efficient search structures are crucial. (For simplicity, we consider only successful searches; the classical formulation due to Knuth also includes probabilities for unsuccessful searches between consecutive keys.) We present an iterative dynamic programming solution to compute the minimum cost.

Formally, given n symbols indexed from 0 to $n-1$, each associated with probability $p[i]$, and defining the sum of probabilities from i to j as $s[i][j] = \sum_{k=i}^{j} p[k]$, the task is to construct a BST that minimizes the expected search cost, expressed as $\text{Cost} = \sum_{i=0}^{n-1} p[i] \cdot \text{depth}(i)$, where $\text{depth}(i)$ represents the depth of the symbol i in the tree.

Algorithm OptimalBinarySearchTree solves this by iterative dynamic programming. Two key arrays are defined: $dp[i][j]$ represents the minimum cost of a BST for symbols from i to j (inclusive), and $s[i][j]$ denotes the sum of probabilities from i to j, which serves as the cost increase at each level. The solution builds the dp table iteratively over increasing lengths of subarrays. When considering a single symbol ($j = i$), the cost is simply $dp[i][i] = p[i]$, reflecting the contribution of a leaf node. For larger ranges, each symbol r from i to j is tested as the root, combining the left subtree $[i, r-1]$ and the right subtree $[r+1, j]$. The total cost for a root r is computed as $dp[i][r-1] + dp[r+1][j] + s[i][j]$, where $s[i][j]$ accounts for the increased depth of all symbols in the subtree. We adopt the convention that an empty interval has cost 0: when $r = i$, the left subtree $[i, r-1]$ is empty and $dp[i][r-1] = 0$, and when $r = j$, the right subtree $[r+1, j]$ is empty and $dp[r+1][j] = 0$. To optimize, $s[i][j]$ is precomputed iteratively.

Algorithm OptimalBinarySearchTree: Optimal binary search tree (iterative)

```
   input: array p[0...n-1] of probabilities
   output: minimum cost of optimal BST
 1 Initialize dp[0...n-1][0...n-1] and s[0...n-1][0...n-1] with 0;
 2 for i = 0 to n-1 do
 3 |   dp[i][i] = p[i];
 4 |   s[i][i] = p[i];
 5 end
 6 for l = 1 to n-1 do
 7 |   for i = 0 to n-1-l do
 8 |   |   j = i + l;
 9 |   |   s[i][j] = s[i][j-1] + p[j];
10 |   |   dp[i][j] = ∞;
11 |   |   for r = i to j do
12 |   |   |   left_cost = 0;
13 |   |   |   right_cost = 0;
14 |   |   |   if r > i then left_cost = dp[i][r-1] ;
15 |   |   |   if r < j then right_cost = dp[r+1][j] ;
16 |   |   |   cost = s[i][j] + left_cost + right_cost;
17 |   |   |   dp[i][j] = min(dp[i][j], cost);
18 |   |   end
19 |   end
20 end
21 return dp[0][n-1];
```

The correctness of this approach comes from the fact that $dp[i][j]$ computes the minimum cost by exhaustively trying each possible root r and combining the optimal costs of the resulting subproblems. The term $s[i][j]$ ensures that the cost reflects the increase in depth for all symbols in the subtree. By filling the table bottom-up, from single symbols to the full range, the iterative method guarantees that $dp[0][n-1]$ yields the optimal cost.

To analyze the time complexity, consider the three nested loops in the algorithm. The outer loop iterates over lengths l from 1 to $n-1$, which takes $O(n)$ time. For each l, the middle loop

iterates over the starting indices i from 0 to $n-1-l$, also $O(n)$ per length. Within these, the inner loop tests each root r from i to j, which can be up to $O(n)$ iterations. This results in a total of $O(n) \cdot O(n) \cdot O(n) = O(n^3)$ operations. Precomputing $s[i][j]$ requires $O(n^2)$ time, as each entry builds on the previous, but this is dominated by the main computation. The total time complexity is therefore $O(n^3)$, with a space complexity of $O(n^2)$ for the tables dp and s.

10.6 Chain Matrix Multiplication

A problem very similar to the Optimal Binary Search Tree is that of constructing an optimal way of multiplying a chain of matrices. Since matrix multiplication is associative, the product of matrices $(M_1 * M_2) * M_3$ is equal to $M_1 * (M_2 * M_3)$. However, depending on the dimensions of the matrices, the computational effort may be different.

Example 10.4 Consider three matrices M_1, M_2, M_3 with dimensions 30×10, 10×30, and 30×2, respectively. The product $M_1 \cdot M_2 \cdot M_3$ has dimension 30×2 regardless of the order of multiplication, but the cost differs:

- $(M_1 \cdot M_2) \cdot M_3$: computing $M_1 \cdot M_2$ takes $30 \cdot 10 \cdot 30 = 9000$ scalar multiplications and yields a 30×30 matrix; multiplying by M_3 adds $30 \cdot 30 \cdot 2 = 1800$, for a total of 10800.
- $M_1 \cdot (M_2 \cdot M_3)$: computing $M_2 \cdot M_3$ takes $10 \cdot 30 \cdot 2 = 600$ and yields a 10×2 matrix; multiplying M_1 by this adds $30 \cdot 10 \cdot 2 = 600$, for a total of 1200.

The right-associative grouping is nine times cheaper.

We let the dimension of matrix M_i be $m_{i-1} \times m_i$. Note that this keeps the matrix product well-defined because the dimension of the matrix M_{i+1} is $m_i \times m_{i+1}$ and the product $M_i \times M_{i+1}$ is well-defined. We can view any evaluation of a chain as a binary tree where the intermediate nodes are the multiplication operation, and the leaves are the matrices themselves. Suppose that our goal is to compute the optimal binary tree for multiplying matrices in the range $M_i \dots M_j$. Taking ideas from the previous section, we let $G[i, j]$ denote the optimal cost of computing the product of matrices in the range $M_i \dots M_j$. Suppose that this product is broken into products of $M_i \dots M_k$ and $M_{k+1} \dots M_j$ and then the multiplication of these two matrices. We can compute the cost of this tree as

$$G[i, k] + G[k + 1, j] + m_{i-1} m_k m_j$$

Then, we have the following predicate on G.

$$G[i, j] \geq \min_{i \leq k < j} (G[i, k] + m_{i-1} m_k m_j + G[k + 1, j])$$

The reader will notice the similarity to the optimal binary search tree problem, and the same algorithm can be adapted to solve this problem.

10.7 Knapsack Problem

We are given n items with weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$. We are also given a knapsack that has a capacity of W. Our goal is to determine the subset of items that can be carried in the knapsack and that maximizes the total value. The standard dynamic programming solution is based on the memoization of the following dynamic programming formulation [Vaz01, DPW10]. Let $G[i, w]$ be the maximum value that can be obtained by picking items from $1..i$ with the capacity constraint of w. Then,

$$G[i,w] = \begin{cases} G[i-1,w] & \text{if } w_i > w, \\ \max(G[i-1,w],\ G[i-1,w-w_i]+v_i) & \text{if } w_i \le w. \end{cases}$$

If $w_i > w$, item i cannot fit and is skipped. Otherwise, the first argument of the max corresponds to excluding item i, and the second to including it. The base cases are simple. The values of $G[0, w]$ and $G[i, 0]$ are zero for all w and i. Our goal is to find $G[n, W]$. By filling in the two-dimensional array G for all values of $0 \le i \le n$ and $0 \le w \le W$, we get an algorithm with time complexity $O(nW)$.

The following algorithm computes $G[n, W]$ and tracks the selected items:

Algorithm Knapsack: 0/1 Knapsack dynamic programming

```
   input: weights w[1..n], values v[1..n], capacity W
   output: maximum value and selected items
 1 var G: array[0...n, 0...W] of int;
 2 selected: set of int;
 3 for w = 0 to W do G[0, w] := 0 ;
 4 for i = 0 to n do G[i, 0] := 0 ;
 5 for i = 1 to n do
 6 |   for w = 1 to W do
 7 |   |   if w_i > w then G[i, w] := G[i − 1, w] ;
 8 |   |   else G[i, w] := max(G[i − 1, w], G[i − 1, w − w_i] + v_i) ;
 9 |   end
10 end
11 selected := ∅;
12 i := n;
13 w := W;
14 while (i > 0) ∧ (w > 0) do
15 |   if G[i, w] ≠ G[i − 1, w] then
16 |   |   selected := selected ∪ {i};
17 |   |   w := w − w_i;
18 |   end
19 |   i := i − 1;
20 end
21 return G[n, W], selected;
```

10.8 LLP Perspective

In this section we collect the LLP reformulations of the dynamic-programming algorithms introduced earlier in the chapter. Each formulation views the problem as the search for the least vector in a finite distributive lattice satisfying a lattice-linear predicate B; the bottom-up DP table-fill is one valid schedule, but the LLP formulation also admits parallel schedules in which several forbidden indices advance concurrently.

LLP-WeightedIntervalScheduling

For weighted interval scheduling (Section 10.3), let $G[j]$ denote $opt[j]$, the maximum weight achievable using intervals $1, \ldots, j$. The lattice is $\mathbb{Z}_{\geq 0}^{n+1}$ with the usual order; j is forbidden when $j-1$ and $p[j]$ are both fixed but j is not yet, and the advance sets $G[j]$ to the recurrence value.

input: p: array$[1 \ldots n]$ of int; w: array$[1 \ldots n]$ of int
output: G: array$[0 \ldots n]$ of int, with $G[n] = opt[n]$
init: $G[j] := 0$, $fixed[j] := false$ for $j = 0 \ldots n$; $fixed[0] := true$;
forbidden(j): $\neg fixed[j] \wedge fixed[j-1] \wedge fixed[p[j]]$
  $\Rightarrow$ **advance**: $G[j] := \max(G[j-1],\ w[j] + G[p[j]])$; $fixed[j] := true$;

LLP-LIS

For the longest increasing subsequence problem (Section 10.4), let $G[j]$ denote the length of the longest strictly increasing subsequence ending at index j. With $pre(j) = \{i < j : A[i] < A[j]\}$, the index j is forbidden when its predecessors are all fixed but j itself is not.

input: A: array$[1..n]$ of real
output: G: array$[1..n]$ of int; $\max_j G[j]$ is the LIS length
init: $G[j] := 1$, $fixed[j] := false$ for $j = 1 \ldots n$;
forbidden(j): $\neg fixed[j] \wedge \forall i \in pre(j) : fixed[i]$
  $\Rightarrow$ **advance**: $G[j] := \max(\{1\} \cup \{G[i] + 1 : i \in pre(j)\})$; $fixed[j] := true$;

LLP-OptimalBinarySearchTree

For the optimal binary search tree problem (Section 10.5), let $G[i, j]$ denote the least cost of any binary search tree built from the keys in the range $i..j$, and let $s[i][j] = \sum_{k=i}^{j} \pi[k]$ denote the sum of access frequencies in that range (with $s[i][j] = 0$ when $i > j$). If symbol k is the root of the optimal tree on $i..j$, the optimal-substructure property gives

$$G[i, j] = \min_{i \leq k \leq j} (G[i, k-1] + s[i][j] + G[k+1, j]).$$

Lemma 10.5 *The predicate* $B \equiv \forall i, j :\ G[i, j] \geq \min_{i \leq k \leq j}(G[i, k-1] + s[i][j] + G[k+1, j])$ *is lattice-linear.*

Proof: If B is false then some index (i, j) satisfies $G[i, j] < \min_{i \le k \le j}(G[i, k-1]+s[i][j]+G[k+1, j])$. The right-hand side is monotone non-decreasing in every component of G: increasing any $G[i, k-1]$ or $G[k+1, j]$ can only increase each term in the minimum, and therefore cannot decrease the minimum itself. Thus the violation cannot disappear unless $G[i, j]$ itself increases. Hence (i, j) is forbidden, and the predicate is lattice-linear.

■

The LLP algorithm uses a single variable G initialized to zero everywhere; it advances $G[i, j]$ whenever it is smaller than the right-hand side above. Algorithm LLP-OptimalBinarySearchTree captures this.

Algorithm LLP-OptimalBinarySearchTree: Finding an optimal binary search tree

```
1 P_{i,j}: Code for thread (i, j);
  input: p: array of real (frequency of each symbol)
  output: G: array[1..n, 1..n] of real, optimal BST costs
2 init: ∀i, j : G[i, j] = 0;
3 ensure: G[i, j] ≥ min_{i≤k≤j} G[i, k − 1] + Σ_{ℓ=i}^{j} p[ℓ] + G[k + 1, j];
4 priority: (j − i);
```

The **priority** statement schedules indices in non-decreasing order of $(j - i)$, processing the singleton ranges first, then ranges of width 2, and so on; this ensures that each $G[i, j]$ is updated at most once. With $O(n^2)$ index pairs and $O(n)$ work per advance, the work complexity is $O(n^3)$.

Constrained variants. The same algorithm extends to constrained versions of the problem with no change of structure:

Lemma 10.6 *The following predicates are lattice-linear:*

1. *The key x is not a parent for any key. The ensure predicate becomes $G[i, j] \ge \min_{i \le k \le j,\ k \ne x} G[i, k-1]+s[i][j]+G[k+1, j]$, whose right-hand side remains monotone in G.*
2. *The left and right subtrees of any internal node differ in size by at most 1. The ensure predicate restricts k to indices satisfying $\big||k - 1 - i| - |j - k - 1|\big| \le 1$; the right-hand side remains monotone in G.*

LLP-Knapsack

For the 0/1 knapsack problem (Section 10.7) with weights w_i, values v_i, and capacity W, let $G[i, c]$ be the maximum value using items $1, \ldots, i$ within capacity c. The lattice is $\mathbb{Z}_{\ge 0}^{n \times (W+1)}$ with the usual order.

input: w, v: array$[1..n]$ of int; W: int
output: G: array$[0..n,\ 0..W]$ of int, with $G[n, W]$ = optimal value
init: $G[0, c] := 0$ for all c; $G[i, c] := 0$ otherwise;
forbidden(i, c): $G[i, c] < G[i-1, c]$
  $\Rightarrow$ **advance**: $G[i, c] := G[i-1, c]$;
forbidden(i, c): $(w_i \leq c) \wedge (G[i, c] < G[i-1, c-w_i] + v_i)$
  $\Rightarrow$ **advance**: $G[i, c] := G[i-1, c-w_i] + v_i$;

10.9 Summary

Dynamic programming captures problems whose optimal solutions can be assembled from optimal solutions to overlapping sub-problems. The trio of recurrence relation, memoization, and bottom-up table fill are sufficient to convert exponential recursion into polynomial computation. The algorithms presented in this chapter share the following high-level recipe: (i) identify the sub-problem structure — typically indexed by intervals $[i, j]$, prefixes $1..j$, or item-and-capacity pairs (i, c); (ii) write a recurrence that expresses the optimal answer for each sub-problem in terms of strictly smaller sub-problems; (iii) fill the resulting table in any topological order respecting the recurrence. Lattice-linear predicate algorithms (LLP) provide an alternative parallel scheduler for the same computation.

Table 10.1 summarizes the time complexity of the sequential algorithms discussed in this chapter.

Problem	**Algorithm**	**Time Complexity**
Weighted interval scheduling	WeightedIntervalScheduling	$O(n \log n)$
Increasing subsequence	Longest-Increasing-Subsequence	$O(n^2)$
Optimal binary search tree	OptimalBinarySearchTree	$O(n^3)$
Chain matrix multiplication	Recurrence-based DP	$O(n^3)$
Knapsack	Knapsack	$O(nW)$

Table 10.1: Algorithms discussed in this chapter.

10.10 Problems

1. **(Optimal BST Reconstruction)** Modify the algorithm for Optimal Binary Search Tree to return not only the optimal cost but also the tree itself.

2. **(Longest DAG Path) (Longest Path in Directed Acyclic Graphs)** We are given a directed acyclic graph (V, E) with n nodes and m edges such that each edge has a positive real cost. We are also given a distinguished vertex v_0. Give an LLP-based algorithm to find the longest path from v_0 to all vertices.

3. **(Polygon Triangulation)** We are given a polygon and we are required to triangulate it optimally. We are given a weight function that takes a triangle on the vertices v_i, v_j and v_k and returns a real-valued function $w(v_i, v_j, v_k)$. Our goal is to triangulate the

given polygon to minimize the sum of weights of all triangles. *(Hint: Devise a recurrence relation similar to the matrix chain product problem.)*

4. **(Longest Common Subsequence)** Give a dynamic programming algorithm for the *Longest Common Subsequence* (LCS) problem: given two sequences $X = x_1 \dots x_m$ and $Y = y_1 \dots y_n$, compute the length of the longest sequence that appears as a subsequence of both. State the recurrence, give the time and space complexity, and describe how to recover an optimal LCS from the DP table.

5. **(Edit Distance)** Give a dynamic programming algorithm for the *Edit Distance* problem: given strings X of length m and Y of length n, compute the minimum number of insertions, deletions, and substitutions to transform X into Y. Prove optimal substructure.

6. **(Coin Change Minimum Coins)** Give a dynamic programming algorithm for the *Coin Change* problem: given denominations $c_1, c_2, \dots, c_k$ (all positive integers) and a target amount N, find the minimum number of coins that sum to N (or report that N cannot be made).

10.11 Bibliographic Remarks

This chapter explores the paradigm of solving optimization problems by breaking them into overlapping subproblems, with applications to the Longest Increasing Subsequence (LIS), Optimal Binary Search Tree (OBST), Knapsack, Weighted Interval Scheduling, and Chain Matrix Multiplication. The chapter introduces dynamic programming (DP) as a method to establish recurrence relations, avoiding redundant computations through memoization or bottom-up computation, as originally formalized by Bellman [Bel52]. Cormen et al. [CLRS01] provide a comprehensive treatment of sequential DP algorithms for these problems.

The LLP-based approach to dynamic programming problems is detailed in Garg [Gar22]. The OBST problem, introduced by Knuth [Knu71], minimizes expected search cost in a binary search tree. The Knapsack problem, formulated in Horowitz and Sahni [HS74] and Ibarra and Kim [IK75], is solved in $O(nW)$ time via DP. Vazirani [Vaz01] and Williamson [DPW10] provide modern DP formulations for Knapsack, including approximation techniques.

Weighted Interval Scheduling, covered in Kleinberg and Tardos [KT06], extends greedy and DP techniques to optimization problems with weighted objectives. The Chain Matrix Multiplication problem, structurally similar to OBST, is addressed using a recurrence akin to Knuth's formulation, as noted in Aho, Hopcroft, and Ullman [AHU74b]. Papadimitriou and Steiglitz [PS82] offer additional context for combinatorial optimization problems like Knapsack and matrix multiplication.

Chapter 11

Network Flow

11.1 Introduction

How much water can flow through a network of pipes from a reservoir to a city? How many vehicles per hour can travel from a suburb to downtown through a road network? How many packets per second can be routed from a server to a client through the internet? These seemingly different questions share a common mathematical structure: they are all instances of the maximum flow problem.

The maximum flow problem is one of the most fundamental problems in combinatorial optimization and network theory. Given a directed graph where each edge has a limited capacity, the goal is to ship as much "flow" as possible from a designated source vertex s to a sink vertex t without exceeding any edge's capacity. The problem was first studied in the 1950s in the context of Soviet railway networks, when the RAND Corporation sought to determine the maximum rate at which supplies could be transported through the Soviet rail system [JF56]. Harris and Ross formulated the problem in a secret 1955 report, and shortly thereafter Ford and Fulkerson developed both a solution algorithm and the celebrated Max-Flow Min-Cut Theorem.

Beyond its historical origins, the maximum flow problem has remarkably diverse applications:

- *Transportation and logistics.* Determining the maximum throughput of a road, rail, or shipping network between two locations.
- *Telecommunication networks.* Computing the maximum bandwidth between two nodes in a data network, or the maximum number of non-interfering calls that can be simultaneously routed.
- *Bipartite matching.* Finding a maximum matching in a bipartite graph reduces to a max-flow problem.
- *Image segmentation.* In computer vision, separating foreground from background in an image can be formulated as a minimum cut problem, which is equivalent to maximum flow.

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

- *Scheduling.* Determining whether a set of tasks can be assigned to machines subject to capacity constraints.

The power of maximum flow also stems from its deep connection to *minimum cuts*. The Max-Flow Min-Cut Theorem—one of the most beautiful results in combinatorics—states that the maximum amount of flow from s to t equals the minimum capacity of any cut separating s from t. This duality has profound algorithmic consequences and connects flow theory to linear programming duality.

This chapter is organized as follows. Section 11.2 provides the definitions of flow networks and flows. Section 11.3 presents the FordFulkerson algorithm and proves its correctness with a detailed worked example. Section 11.4 establishes the Max-Flow Min-Cut Theorem. Section 11.5 presents the EdmondsKarp algorithm, which achieves a strongly polynomial running time by choosing augmenting paths carefully. Section 11.6 demonstrates two classical reductions — bipartite matching and edge-disjoint paths — to maximum flow. Section 11.7 casts the chapter through the lattice-linear predicate framework: the lattice of minimum cuts and an algorithm for finding mincuts satisfying a lattice-linear side predicate.

11.2 Definitions

Definition 11.1 (Flow Network) *A flow network is a directed graph $\mathcal{N} = (V, E)$ with a source vertex $s \in V$, a sink vertex $t \in V$, and a capacity function $c : E \to \mathbb{R}^+$ that assigns a non-negative capacity $c(u, v)$ to each edge $(u, v) \in E$.*

We write $n = |V|$ and $m = |E|$ for the number of vertices and edges, respectively. We assume that there are no edges entering s and no edges leaving t. If an edge $(u, v) \notin E$, we define $c(u, v) = 0$.

Definition 11.2 (Flow) *A flow in a flow network $\mathcal{N}$ is a function $f : E \to \mathbb{R}^+$ that satisfies:*

1. ***Capacity constraint:*** $\forall (u, v) \in E,\ 0 \le f(u, v) \le c(u, v)$.
2. ***Flow conservation:*** $\forall v \in V \setminus \{s, t\},\ \sum_{u:(u,v)\in E} f(u, v) = \sum_{w:(v,w)\in E} f(v, w)$.

For convenience, we define $f(u, v) = 0$ whenever $(u, v) \notin E$. The capacity constraint says that the flow on each edge cannot exceed its capacity. The conservation constraint says that for every vertex other than s and t, the total flow entering the vertex equals the total flow leaving it—flow is neither created nor destroyed at intermediate vertices.

Definition 11.3 (Value of Flow) *The value of a flow f is defined as the total flow out of the source:*

$$|f| = \sum_{v:(s,v)\in E} f(s, v)$$

The *maximum flow problem* asks: given a flow network $\mathcal{N}$ with source s and sink t, find a flow f of maximum value $|f|$.

Definition 11.4 (Residual Network) *Given a flow network $\mathcal{N} = (V, E)$ and a flow f, the residual network $\mathcal{N}_f = (V, E_f)$ consists of edges with remaining capacity, defined as:*

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$$

where $c_f(u, v)$ is the residual capacity defined as:

$$c_f(u, v) = c(u, v) - f(u, v) + f(v, u).$$

The residual network captures the remaining capacity to push additional flow. For a forward edge $(u, v) \in E$ with no anti-parallel edge, $c_f(u, v) = c(u, v) - f(u, v)$ is the unused capacity. For the reverse direction, $c_f(v, u) = f(u, v)$, representing the ability to "cancel" previously sent flow. When both (u, v) and (v, u) are edges in E, the formula correctly combines the unused forward capacity with the cancellable reverse flow.

Definition 11.5 (Augmenting Path) *An augmenting path is a simple path from the source s to the sink t in the residual network $\mathcal{N}_f$.*

The *bottleneck capacity* of an augmenting path P is $c_f(P) = \min_{(u,v)\in P} c_f(u, v)$, the minimum residual capacity along the path.

Definition 11.6 (Cut) *An s-t cut (S, T) is a partition of V into two sets S and $T = V \setminus S$ such that $s \in S$ and $t \in T$. The* capacity *of the cut is:*

$$c(S, T) = \sum_{\substack{u\in S, v\in T \\ (u,v)\in E}} c(u, v)$$

Note that the capacity of a cut only counts edges going from S to T, not edges from T to S.

11.3 The FordFulkerson Algorithm

The FordFulkerson algorithm is a greedy method that computes the maximum flow in a flow network. It was published in 1956 by L. R. Ford, Jr. and D. R. Fulkerson. The algorithm works by repeatedly finding augmenting paths in the residual network and increasing the flow along these paths until no more augmenting paths exist.

Algorithm FordFulkerson: FordFulkerson algorithm

input: graph $\mathcal{N}$, source s, sink t
output: maximum flow f

```
Initialize flow f(u, v) = 0 for all edges (u, v) in N;
while there exists an augmenting path P from s to t in the residual network N_f do
    Let c_f(P) := min{c_f(u, v) : (u, v) ∈ P};
    // residual capacity of path P
    for each edge (u, v) in P do
        if (u, v) ∈ E then f(u, v) = f(u, v) + c_f(P) // Increase flow ;
        else f(v, u) = f(v, u) − c_f(P) // Decrease flow in the reverse edge ;
    end
end
return f
```

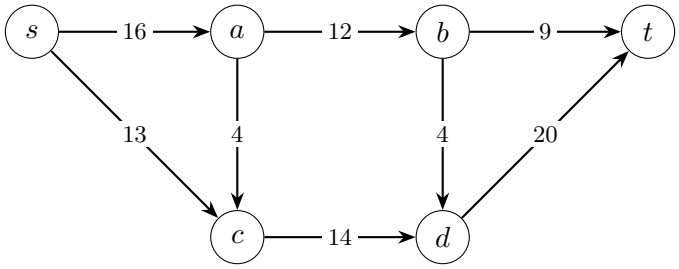
Theorem 11.7 *If all capacities are integers, the FordFulkerson algorithm terminates and returns a maximum flow.*

Proof: *Termination.* Each augmentation increases the total flow by the bottleneck capacity of the augmenting path, which is a positive integer (since all capacities are integers and the flow remains integer-valued throughout). The total flow is bounded above by the sum of capacities leaving s, so the algorithm terminates after finitely many iterations.

Correctness. When the algorithm terminates, there are no augmenting paths in the residual graph. That the resulting flow is maximum follows from the Max-Flow Min-Cut Theorem (Theorem 11.9), proved in Section 11.4.

■

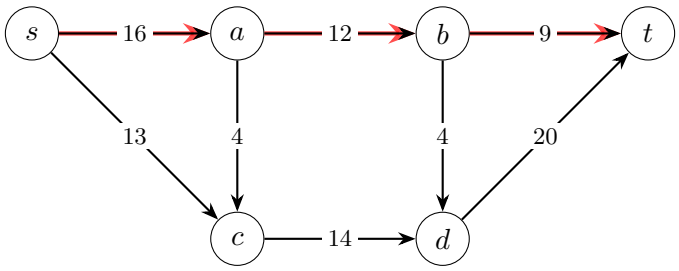
Let us demonstrate the FordFulkerson algorithm step by step on a concrete flow network. Consider a flow network with six vertices $\{s, a, b, c, d, t\}$ and the following capacities:



Initially, all flow values are 0. The edge labels show capacity.

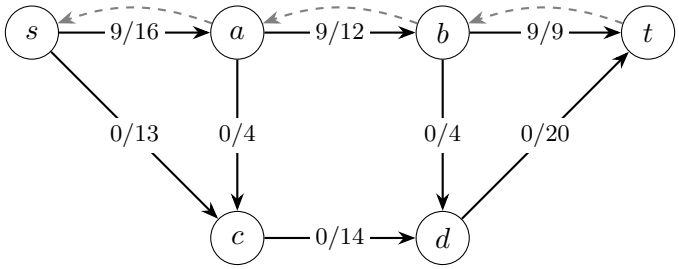
First Iteration

We look for a path from s to t in the residual network. Let us choose the path $s \to a \to b \to t$.



The minimum residual capacity on this path is $\min\{16, 12, 9\} = 9$. We augment the flow by 9 units along this path.

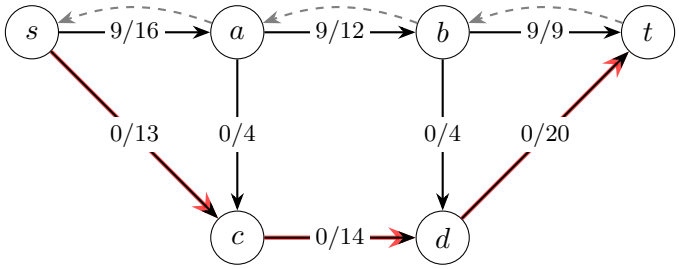
Updated Flow After First Iteration



The flow value is now 9. The edge labels show flow/capacity. The residual network now includes reverse edges where the flow was pushed.

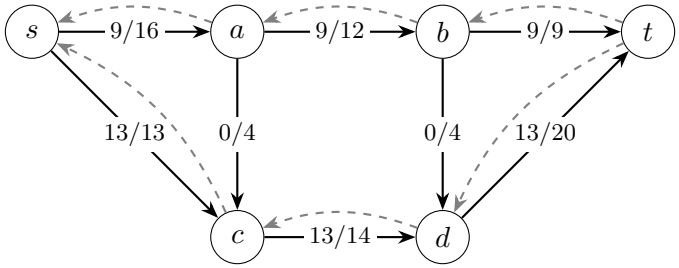
Second Iteration

We find another path in the residual network: $s \to c \to d \to t$.



The minimum residual capacity on this path is $\min\{13, 14, 20\} = 13$. We augment the flow by 13 units.

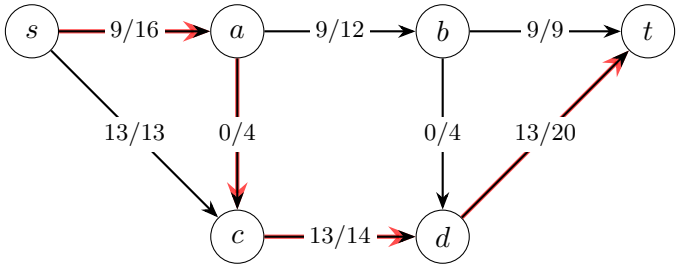
Updated Flow After Second Iteration



The flow value is now $9 + 13 = 22$.

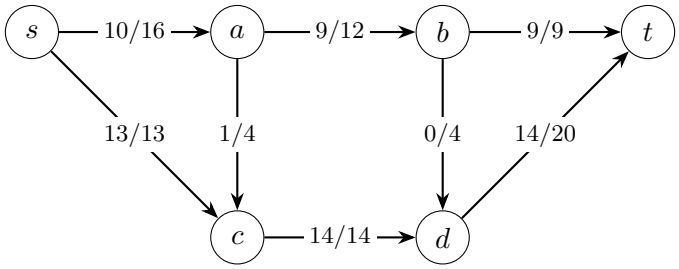
Third Iteration

We find another path: $s \to a \to c \to d \to t$.



The minimum residual capacity on this path is $\min\{7, 4, 1, 7\} = 1$. We augment the flow by 1 unit.

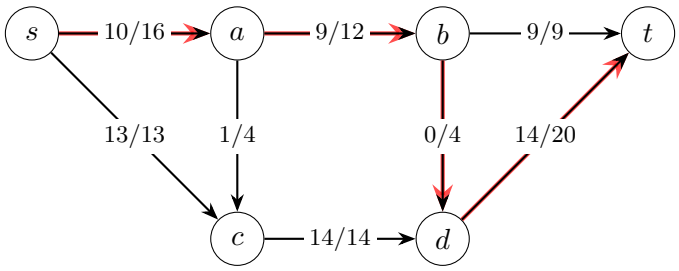
Updated Flow After Third Iteration



The flow value is now $10 + 13 = 23$.

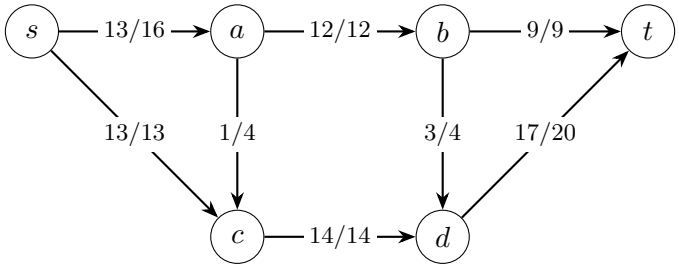
Fourth Iteration

We find one more path: $s \to a \to b \to d \to t$.



The minimum residual capacity on this path is $\min\{6, 3, 4, 6\} = 3$. We augment the flow by 3 units.

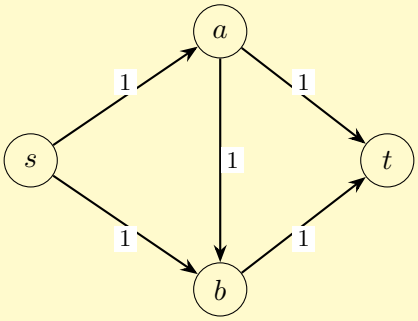
Final Flow



The flow value is now $13 + 13 = 26$. There are no more augmenting paths in the residual network, so the algorithm terminates. The maximum flow is 26.

A natural question is: why do we need reverse edges in the residual network? Could we not simply send flow greedily along forward edges? The following small example shows that without the ability to "undo" flow via reverse edges, a greedy approach can get stuck at a suboptimal solution.

Example 11.8 Consider the following network:



The maximum flow is 2 (one unit along $s \to a \to t$ and one along $s \to b \to t$). However, suppose the algorithm first chooses the path $s \to a \to b \to t$, sending 1 unit. Now all forward edges on this path are saturated. Without reverse edges, we are stuck with flow value 1.

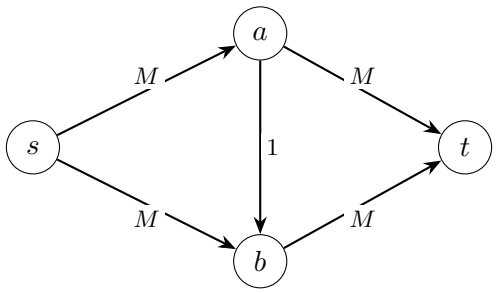


Figure 11.1: An example that shows that the FordFulkerson algorithm may take time proportional to the maxflow in the graph.

With the residual network, however, the reverse edge from b to a (with residual capacity 1) allows the algorithm to find the augmenting path $s \to b \to a \to t$ in the residual graph, pushing one more unit. The result: 2 units reach t, which is optimal. The reverse edge effectively "reroutes" the unit that was going through $a \to b$, redirecting the flow from a directly to t and the flow from b directly to t.

Let us determine the running time of the FordFulkerson algorithm. Suppose that all capacities in the graph are integral. Let f^* be the maximum flow in the graph. Every iteration of FordFulkerson's algorithm finds an augmenting path in the residual network of at least one unit. This step takes $O(m)$ time. Hence, the total running time is $O(f^*m)$. This algorithm is pseudo-polynomial because the running time depends on the numeric value of the maximum flow f^*, which can be exponential in the input size. It is desirable to get an algorithm whose time complexity depends only on the number of vertices and edges, not on the capacities. For large capacities, this can lead to inefficiencies, motivating strongly polynomial algorithms like the EdmondsKarp and Dinitz algorithms.

To see how bad FordFulkerson can be with poor augmenting path choices, consider the network shown in Figure 11.1.

Here the four outer edges have capacity M, a large integer, and the middle edge (a, b) has capacity 1. The maximum flow is $2M$. If the algorithm alternately chooses paths $s \to a \to b \to t$ and $s \to b \to a \to t$ (using the reverse edge each time), each augmentation sends only 1 unit. The algorithm would require $2M$ iterations instead of the 2 iterations needed by choosing $s \to a \to t$ and $s \to b \to t$.

11.4 Min-Cut and the Max-Flow Min-Cut Theorem

There is a beautiful duality between flows and cuts. Intuitively, a cut represents a "bottleneck" in the network—removing the cut edges disconnects s from t, so no flow can pass. The capacity of the minimum cut therefore provides an upper bound on the maximum flow. Remarkably, this bound is always tight.

Theorem 11.9 (Max-Flow Min-Cut Theorem) *In any flow network, the maximum value of a flow is equal to the minimum capacity of an s-t cut.*

Proof: We prove three facts: (1) every flow value is at most every cut capacity; (2) a flow with no augmenting path achieves equality with some cut; (3) such a flow exists.

(1) Weak duality. Let f be any flow and (S, T) any s-t cut. By flow conservation summed over all vertices in S, the flow value equals the net flow across the cut: $|f| = \sum_{\substack{u \in S, v \in T \\ (u,v) \in E}} f(u, v) - \sum_{\substack{u \in T, v \in S \\ (u,v) \in E}} f(u, v) \leq c(S, T)$.

(2) No augmenting path implies equality. Suppose f is a flow with no s-t path in the residual graph. Define S as the set of vertices reachable from s in the residual graph, and $T = V \setminus S$. Since $t \notin S$, the pair (S, T) is a valid cut. For every edge (u, v) with $u \in S, v \in T$: since v is not reachable, the residual capacity $c_f(u, v) = 0$, so $f(u, v) = c(u, v)$. For every edge (u, v) with $u \in T, v \in S$: since u is not reachable but v is, $f(u, v) = 0$. Therefore $|f| = c(S, T)$.

(3) Existence. The FordFulkerson algorithm terminates (as shown above) with a flow that has no augmenting path; by (2) this flow achieves equality with the cut (S, T) constructed above. Combined with (1), the flow is maximum and the cut is minimum.

■

According to the Max-Flow Min-Cut theorem, the value of the maximum flow equals the capacity of the minimum cut. Let us identify the minimum cut in our example by finding the set of vertices reachable from s in the final residual network.

Let us trace the vertices reachable from s in the final residual network. From s: we can reach a because $s \to a$ has residual capacity $16 - 13 = 3 > 0$. From a: we can reach c because $a \to c$ has residual capacity $4 - 1 = 3 > 0$. From c: the edge $c \to d$ is saturated (residual 0), so d is not reachable from c. No further vertices are reachable. So $S = \{s, a, c\}$ and $T = \{b, d, t\}$.

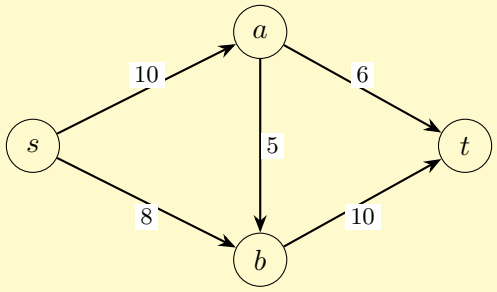
The cut edges (from S to T) are:

- (a, b) with capacity 12
- (c, d) with capacity 14

The total capacity is $12 + 14 = 26$, which is exactly equal to the maximum flow, confirming the Max-Flow Min-Cut theorem.

We give another example.

Example 11.10 Consider a simpler network with four vertices:



Iteration 1: Path $s \to a \to t$, bottleneck $\min\{10, 6\} = 6$. Flow $= 6$.
Iteration 2: Path $s \to a \to b \to t$, bottleneck $\min\{4, 5, 10\} = 4$. Flow $= 10$.
Iteration 3: Path $s \to b \to t$, bottleneck $\min\{8, 6\} = 6$. Flow $= 16$.
No more augmenting paths exist. Maximum flow $= 16$.

To find the minimum cut, we check which vertices are reachable from s in the final residual network. After the three augmentations, $s \to a$ has residual $10 - 10 = 0$ and $s \to b$ has residual $8 - 6 = 2$, so b is reachable. From b, $b \to t$ has residual $10 - 10 = 0$ and the reverse edge of $a \to b$ has residual 4, so a is reachable. From a, $a \to t$ has residual $6 - 6 = 0$. Thus $S = \{s, a, b\}$, $T = \{t\}$, and the minimum cut has capacity $c(a, t) + c(b, t) = 6 + 10 = 16$, matching the max flow.

11.5 EdmondsKarp Algorithm

Figure 11.1 showed that poor choices of augmenting paths can cause the FordFulkerson algorithm to run for $\Theta(f^*)$ iterations, where f^* can be exponential in the input size. The EdmondsKarp modification fixes this by always choosing a *shortest* augmenting path (fewest edges), found via breadth-first search (BFS) in the residual graph.

Algorithm EdmondsKarp: EdmondsKarp algorithm

input: graph $\mathcal{N} = (V, E)$, capacities c, source s, sink t
output: maximum flow f

```
Initialize f(u,v) ← 0 for all (u,v) ∈ E;
while there exists an augmenting path in N_f do
    Use BFS to find shortest path P from s to t in N_f;
    Compute c_f(P) = min_{(u,v)∈P} c_f(u,v);
    for each edge (u,v) ∈ P do
        if (u,v) ∈ E then f(u,v) ← f(u,v) + c_f(P) // Increase flow ;
        else f(v,u) ← f(v,u) − c_f(P) // Decrease flow in the reverse edge ;
    end
end
return f;
```

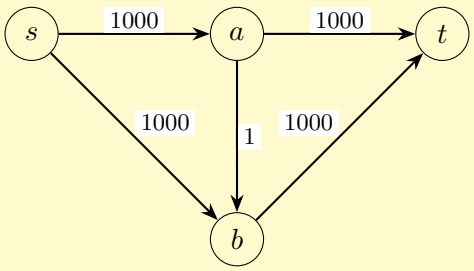
The key insight is that using BFS guarantees that the distance (in number of edges) from s to any vertex in the residual graph never decreases across augmentations. Moreover, after at

most $O(n \cdot m)$ augmentations, the algorithm terminates.

Theorem 11.11 *The EdmondsKarp algorithm runs in $O(nm^2)$ time.*

This is a *strongly polynomial* bound—it depends only on n and m, not on the magnitudes of capacities.

Example 11.12 Consider the network from Figure 11.1 with $M = 1000$:



With arbitrary path selection, FordFulkerson might take 2000 iterations. With BFS (EdmondsKarp), the shortest paths from s to t are $s \to a \to t$ and $s \to b \to t$, each with 2 edges. The algorithm chooses one of these, sending 1000 units, then the other, sending 1000 units. Total: 2 iterations, giving maximum flow 2000.

When all capacities in a flow network are integers, the FordFulkerson algorithm (and hence the EdmondsKarp algorithm) always produces an integer-valued maximum flow.

Theorem 11.13 (Integrality Theorem) *If all capacities in a flow network are integers, then there exists a maximum flow in which the flow on every edge is an integer.*

Proof: The FordFulkerson algorithm starts with the zero flow (integer-valued). In each iteration, the bottleneck capacity of the augmenting path is the minimum of integer residual capacities, hence an integer. Augmenting by an integer amount preserves the integrality of all flow values. Since the algorithm terminates with a maximum flow, this maximum flow is integer-valued.

■

This theorem has important combinatorial consequences. For instance, it implies that the maximum number of edge-disjoint s-t paths equals the minimum number of edges whose removal disconnects s from t (Menger's theorem), since we can model edge-disjoint paths as a max-flow problem with unit capacities.

11.6 Applications of Max-Flow

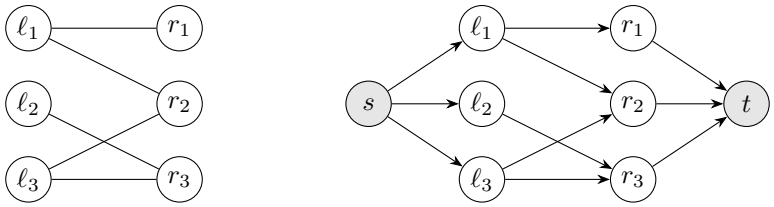
The max-flow problem reduces several classical combinatorial problems to a single algorithmic primitive. We illustrate two such reductions; both produce a flow network with unit capacities,

so by the integrality theorem (Theorem 11.13) the resulting maximum flow is integer-valued and corresponds directly to a combinatorial answer.

Bipartite Matching

Given a bipartite graph $\mathcal{G} = (L \cup R, E)$ with edges only between L and R, a *matching* is a set of edges no two of which share an endpoint. The *maximum bipartite matching* problem asks for a matching of largest size.

To reduce to max-flow, build a directed network $\mathcal{N}' = (V', E')$ with $V' = L \cup R \cup \{s, t\}$. For every $\ell \in L$ add an edge (s, ℓ), for every $r \in R$ add an edge (r, t), and orient every edge $(\ell, r) \in E$ from L to R. Give every edge unit capacity (Fig. 11.2). A maximum integer-valued flow in $\mathcal{N}'$ corresponds exactly to a maximum matching in $\mathcal{G}$: an edge (ℓ, r) is in the matching iff its flow is 1, since flow conservation and unit capacities at ℓ and r ensure that each vertex is matched at most once.



(a) Bipartite graph $\mathcal{G}$ (b) Flow network $\mathcal{N}'$ with unit capacities

Figure 11.2: Reduction from bipartite matching to max-flow. Every edge in (b) has capacity 1. A maximum flow corresponds to a maximum matching in (a).

If the maximum flow has value $|f^*|$, then $\mathcal{G}$ has a matching of size $|f^*|$, and conversely. Running EdmondsKarp on $\mathcal{N}'$ thus solves bipartite matching in $O(nm^2)$ time, with sharper bounds available via Hopcroft-Karp.

Edge-Disjoint Paths from s to t

Given a directed graph $\mathcal{G} = (V, E)$ and two vertices $s, t \in V$, an *edge-disjoint s-t path collection* is a set of paths from s to t no two of which share an edge. The problem asks for the largest such collection.

To reduce to max-flow, assign capacity 1 to every edge of $\mathcal{G}$ (Fig. 11.3). The maximum number of edge-disjoint s-t paths equals the value of a maximum integer flow: any flow of value k decomposes into k edge-disjoint paths (because each edge carries flow 0 or 1 and flow conservation routes exactly k units from s to t); conversely, k edge-disjoint paths give a flow of value k. Combined with the Max-Flow Min-Cut Theorem this is the edge form of *Menger's Theorem*: the maximum number of edge-disjoint s-t paths equals the minimum number of edges whose removal disconnects s from t.

The same reduction yields the *vertex-disjoint* version with a standard vertex-splitting trick: replace every internal vertex v by two copies $v_{\text{in}}, v_{\text{out}}$ joined by a single capacity-1 edge, and route all incoming edges into v_{in} and all outgoing edges out of v_{out}. A unit flow that uses two

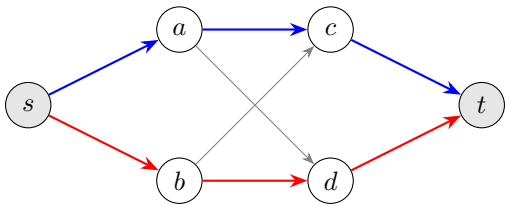


Figure 11.3: Edge-disjoint paths from s to t in a graph with all unit-capacity edges. The blue and red paths are edge-disjoint; gray edges show two further unit-capacity edges that are not used. The minimum s-t edge cut has size 2 (e.g., the two edges out of s), matching the maximum number of edge-disjoint paths.

paths through the same vertex would saturate that internal edge, so flow conservation again limits each vertex to one passing path.

11.7 LLP Perspective

In this section we recast the maximum-flow / minimum-cut machinery in the lattice-linear predicate framework of Chapter 2. We first show that the property of being a minimum s-t cut is itself a lattice-linear predicate, so the set of minimum cuts forms a finite distributive lattice. This places the *constrained-mincut problem* — finding a mincut that also satisfies a side predicate B — within reach of the LLP main loop. The resulting Algorithm LLP-MinCut runs in $O(MF(n,m)+km)$ time, where $MF(n,m)$ is the time to compute one max-flow and k is a complexity parameter of B (the number of times B flips along any chain in the cut lattice).

Lattice of Mincuts

We first show that the property of a cut (S,T) being a mincut is lattice-linear. A set $S \subseteq V$ is a mincut if:

- $s \in S$, $t \notin S$ (i.e., S is a cut).
- $\text{val}(S) = \sum_{(u,v)\in E: u\in S, v\in V\setminus S} c(u,v) \leq \text{val}(S')$ for all cuts S'.

We can now show:

Lemma 11.14 *The predicate $M(S) = (s \in S \text{ and } t \notin S)$ and ($val(S)$ is minimum) is lattice-linear with efficient advancement.*

Proof: We use submodularity of the cut function. For min cuts S_1, S_2, the submodularity inequality is:

$$\text{val}(S_1) + \text{val}(S_2) \geq \text{val}(S_1 \cup S_2) + \text{val}(S_1 \cap S_2).$$

If $\text{val}(S_1) = \text{val}(S_2) = \mu$ (the min cut capacity), then $\text{val}(S_1 \cup S_2) + \text{val}(S_1 \cap S_2) \leq 2\mu$. Since $\text{val}(S) \geq \mu$ for any cut, each term must equal μ. Thus, $S_1 \cap S_2$ and $S_1 \cup S_2$ are min cuts (noting $s \in S_1 \cap S_2$, $t \notin S_1 \cup S_2$).

For the advancement property, we need to show that for any non-mincut S, we can identify a vertex whose inclusion in S is necessary for reaching a mincut. Let f^* be any maximum flow and G_f the corresponding residual graph. By Theorem 11.9, a cut S is a mincut if and only if every forward crossing edge (u, v) with $u \in S$, $v \notin S$ is saturated ($f(u, v) = c(u, v)$) and every reverse crossing edge (v, u) carries zero flow ($f(v, u) = 0$).

Suppose S is not a mincut and $S \neq V \setminus \{t\}$. Then there exists a crossing edge (u, v) with $u \in S$, $v \notin S$, that is not saturated or a reverse crossing edge that carries positive flow. In either case, the residual capacity $c_f(u, v) > 0$. If $v = t$, then no superset of S can be a mincut (since t cannot be added), so all indices are forbidden. Otherwise, v must be included in S to eliminate this unsaturated crossing edge—thus v is the advancing index. Since the residual graph can be computed in $O(m)$ time, advancement is efficient.

■

LLP-MinCut Algorithm

Since finding a mincut satisfying an arbitrary predicate B is NP-complete in general (an arbitrary B can encode an NP-hard constraint on the cut, such as requiring the source side to be an independent set), we focus on special cases of B. We first assume that the given predicate B is lattice-linear. This means that the set of cuts that satisfy B form an inf-semilattice and we have the efficient advancement property. Since the set of mincuts form a sublattice, we get that the set of mincuts that satisfy B also forms an inf-semilattice.

We now have a simple algorithm to find a mincut that satisfies B.

To run the LLP algorithm, we encode the cut S by its indicator vector G, where $G[j] = 1$ iff $v_j \in S$. Lattice operations on G correspond to set operations on S: the meet of two indicators is the indicator of their intersection, and the join is the indicator of their union.

Algorithm LLP-MinCut: To find the minimum vector that satisfies B

```
   input: B: lattice-linear predicate on cuts
   output: G: least mincut satisfying B, or null
 1 G := LeastMincut;
 2 while ¬B(G) do
 3 |   while ∃j : forbidden(G, j, B) do
 4 |   |   if G[j] = 1 then return null;
 5 |   |   else G[j] := 1;
 6 |   end
 7 |   if there does not exist LeastMincut ≥ G then
 8 |   |   return null ;                                  // no such cut
 9 |   else
10 |   |   G := LeastMincut ≥ G;
11 |   end
12 end
13 return G ;                                          // the optimal solution
```

The algorithm alternates between finding the mincut after G and finding G that satisfies B. If it cannot find a mincut or a cut satisfying B, it returns null. Assuming that the complexity

of computing $\exists j : forbidden(G, j, B)$ is $O(m)$, the above algorithm takes $O(MF(n, m) + mn)$ given the join-irreducible elements of the mincut lattice. The outer while loop executes at most n times because at least one bit in the vector G becomes 1 after every iteration.

We now show that for many lattice-linear predicates, the complexity of the above algorithm is $O(MF(n, m) + km)$ where k is a parameter dependent upon the predicate B.

To concretize our discussion, consider the predicate $B(S)$ that holds whenever either both vertices u and v belong to S or neither belong to S. We are interested in finding S such that S is a mincut and satisfies $B(S)$.

We note here that the predicate, when evaluated over increasing sets of S, can transition at most twice. Either it is initially false and then turns true. This happens when S initially has u or v, and then later it gets both. Alternatively, it may be initially true, then turn false and then finally turn true. This happens when initially, neither u nor v is in S. Then, one of them gets added to S and finally both of them get added.

The predicate B that can transition at most k times is termed a *k-Transition predicate*. Formally, let $J(P)$ denote the lattice of ideals of a poset P.

Definition 11.15 *A predicate $B : J(P) \to \{true, false\}$ on the ideals of a poset P is a* k-Transition Predicate *if, for any chain of ideals $I_1 \subseteq I_2 \subseteq \cdots \subseteq I_n \in J(P)$, the sequence $B(I_1), B(I_2), \ldots, B(I_n)$ has at most k transitions, where a transition occurs at index i if $B(I_i) \neq B(I_{i+1})$.*

For example, the *stable* predicates [CL85] are 1-Transition predicates. Similarly, predicates that are true on a single mincut [GS24] are also 1-Transition predicates. We will assume that the algorithm for detecting whether B is true on a cut S is $O(m)$.

Thus, if the lattice-linear predicate is a k-transition predicate, then the time complexity is bounded by $O(MF(n, m) + km)$.

11.8 Summary

This chapter presented algorithms for the maximum flow problem in directed networks and established the duality between maximum flows and minimum cuts. Two unit-capacity reductions — bipartite matching and edge-disjoint s-t paths — show that max-flow is the algorithmic engine behind several classical combinatorial problems. The lattice of minimum cuts is itself a finite distributive lattice, allowing the LLP framework to extend max-flow with side constraints expressed as lattice-linear predicates.

Table 11.1 summarizes the algorithms discussed in this chapter.

Here f^* is the value of the maximum flow, $MF(n, m)$ denotes the time to compute a maximum flow, and k is a parameter dependent on the predicate.

11.9 Problems

1. **(Ford Fulkerson Trace)** Consider the flow network below, with capacities on edges. Use the FordFulkerson algorithm to compute the maximum flow from s to t. Show one possible sequence of augmenting paths and the final flow value.

Problem	Algorithm	Time Complexity
Maximum Flow	FordFulkerson	$O(f^*m)$
Maximum Flow	EdmondsKarp	$O(nm^2)$
Bipartite Matching	via EdmondsKarp	$O(nm^2)$
Edge-disjoint s-t paths	via EdmondsKarp	$O(nm^2)$
Constrained Mincut	LLP Mincut	$O(MF(n,m)+km)$

Table 11.1: Algorithms discussed in this chapter.

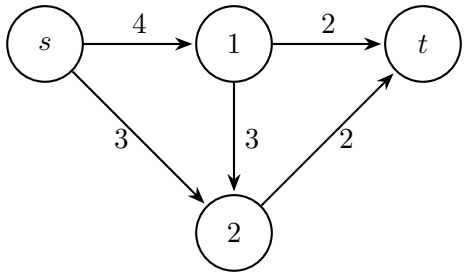


Figure 11.4: Flow network with capacities.

2. **(Edge Disjoint Path Count)** Let $G = (V, E)$ be a directed graph and let s and t be two nodes in V. Describe an efficient algorithm to find all the edge-disjoint paths from s to t in G. (Edge-disjoint means that none of the paths have any edges in common; they are allowed to have vertices in common.)

3. **(Capacity Change Update)** Suppose you are given a flow network $G = (V, E)$ with source s, sink t, integer capacities, and a maximum flow f of G along with the residual graph G_f. There are n vertices and m edges in the flow network.

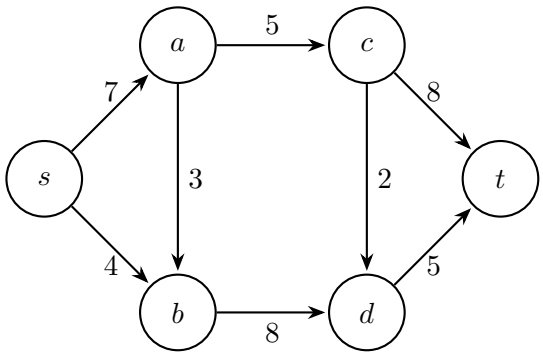
   (a) We increase the capacity of a single edge $(u, v) \in E$ by one. Give an $O(m+n)$ time algorithm to update the maximum flow.

   (b) We decrease the capacity of a single edge $(u, v) \in E$ by one. Give an $O(m+n)$ time algorithm to update the maximum flow.

4. (**Vertex capacities.**) In some applications, vertices also have capacity constraints. Given a flow network $G = (V, E)$ where each vertex $v \in V \setminus \{s, t\}$ has a capacity $c(v)$ limiting the total flow through v, show how to reduce this problem to a standard maximum flow problem (where only edges have capacities).

5. (**Multiple sources and sinks.**) Suppose a flow network has multiple sources $s_1, s_2, \ldots, s_k$ and multiple sinks $t_1, t_2, \ldots, t_l$. Show how to reduce this to a single-source, single-sink max-flow problem.

6. (**Minimum cut.**) Find the minimum s-t cut in the following network. Verify your answer by finding a flow of the same value.



7. (**Unique minimum cut.**) Prove or disprove: a flow network always has a unique minimum cut.

8. **(Min Cut Lattice Realization)** Let $(L, \leq)$ be a finite distributive lattice. Show that there exists a directed graph G with distinguished vertices s and t, and a capacity function c on the edges, such that the lattice of minimum s-t cuts of G (ordered by the set of vertices reachable from s) is isomorphic to L.

9. **(Incremental Max-Flow under Edge Insertion)** A max-flow f of value $|f|$ has been computed by EdmondsKarp on $\mathcal{N} = (V, E)$. A new edge (u, v) of capacity $c(u, v)$ is inserted. Give an algorithm that updates f to a max-flow on the augmented network without recomputing from scratch, and bound its running time in terms of the new max-flow value.

10. **(Dynamic Capacity Decrease)** A single edge $(u, v) \in E$ has its capacity reduced from $c(u, v)$ to $c'(u, v) < c(u, v)$. The previously computed max-flow f may now be infeasible if $f(u, v) > c'(u, v)$. Describe a repair procedure and bound the number of augmenting-path steps.

11. **(Constrained Max-Flow with Lower Bounds)** Each edge (u, v) has both a capacity $c(u, v)$ and a lower bound $\ell(u, v) \geq 0$. A feasible flow satisfies $\ell(u, v) \leq f(u, v) \leq c(u, v)$ in addition to conservation. Reduce this constrained-flow problem to ordinary max-flow on an auxiliary network in polynomial time.

12. **(Min-Cut with Ties)** A network may admit several distinct minimum s-t cuts of the same capacity. Show that the family of minimum cuts forms a finite distributive lattice under set inclusion of the source side S, and give an algorithm that returns both the *minimum* and the *maximum* mincut in the same time as one max-flow computation.

13. **(Approximate Max-Flow via Capacity Scaling)** Capacities are huge integers; running FordFulkerson directly takes $O(f^*|E|)$, which may be exponential in the input size. Design an algorithm that returns a flow of value at least $(1 - \varepsilon)f^*$ in time polynomial in $|V|$, $|E|$, $\log U$, and $1/\varepsilon$, where $U = \max_e c(e)$.

11.10 Bibliographic Remarks

The maximum flow problem and its solutions have been extensively studied in combinatorial optimization and network theory. The original FordFulkerson algorithm was introduced in 1956 [JF56], together with the foundational Max-Flow Min-Cut Theorem that proves its correctness. The Dinitz algorithm [Din70] and the algorithm by Edmonds and Karp [EK72] offer strongly polynomial time guarantees. More advanced algorithms such as Goldberg and Tarjan's push-relabel method [GT88] provide even better performance in practice and theory, particularly for dense networks. For comprehensive treatments of flow algorithms, see Ahuja, Magnanti, and Orlin [AMO93], Schrijver [Sch03], and Korte and Vygen [KV18].

The applications discussed in Section 11.6 have rich histories of their own. The reduction of bipartite matching to max-flow goes back to Kőnig's theorem (1931). Hopcroft and Karp [HK73] obtained an $O(m\sqrt{n})$ algorithm for bipartite matching that exploits the unit-capacity structure to find blocking flows. The connection between max-flow and edge-disjoint paths predates max-flow itself: it is the algorithmic content of Menger's theorem [Men27]. The integrality theorem and its consequences are surveyed in detail in Schrijver [Sch03].

The lattice structure of minimum cuts of Section 11.7 was characterized by Picard and Queyranne [PQ80]; the LLP framework's treatment in this chapter views their structure through the lens of lattice-linear predicates and reduces constrained-mincut problems to a single max-flow plus lattice traversal. Streit and Garg [SG25] give a detailed treatment of constrained cuts, flows, and lattice-linearity.

Chapter 12

Bipartite Matching

12.1 Introduction

In this chapter, we discuss algorithms for matching problems in bipartite graphs and related combinatorial optimization problems. Figure 12.1 shows the relationships between various structures studied in this chapter. The three rows correspond to three classical min-max theorems: König's theorem relates matching and vertex cover, Dilworth's theorem relates chain decomposition and antichain, and Menger's theorem relates vertex-disjoint paths and vertex cuts. The vertical arrows show reductions between these problems.

The matching problem is one of the most studied problems in combinatorial optimization, with applications ranging from job assignment and resource allocation to network design and computational biology. Bipartite matching, in particular, is a building block: many problems that appear unrelated — finding a minimum vertex cover, partitioning a poset into the fewest chains, exhibiting a maximum antichain, computing vertex-disjoint paths between two terminals — all reduce to bipartite matching by reductions covered in this chapter. The classical augmenting-path algorithm therefore deserves a careful treatment, both because it solves matching directly in $O(nm)$ time and because the same algorithm, run on a problem-specific transformation, settles each of those four neighboring problems. The chapter develops the algorithm first, then makes the reductions explicit so that readers see why a single technique unlocks all of Figure 12.1.

This chapter is organized as follows. Section 12.2 describes the classical sequential augmenting path algorithm for maximum cardinality matching in bipartite graphs. Section 12.3 shows how to reduce chain partition of a poset to bipartite matching via the strict split construction, and gives an LLP algorithm for finding the maximum antichain. Section 12.4 shows how to convert a maximum matching into a minimum vertex cover, establishing König's theorem algorithmically. Section 12.5 proves Dilworth's theorem, connecting the minimum chain cover to the maximum antichain. Section 12.6 summarizes the reductions between all the structures in Figure 12.1. Section 12.7 presents Menger's theorem on vertex-disjoint paths and vertex cuts in directed graphs, and shows how it relates to bipartite matching via poset reductions.

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

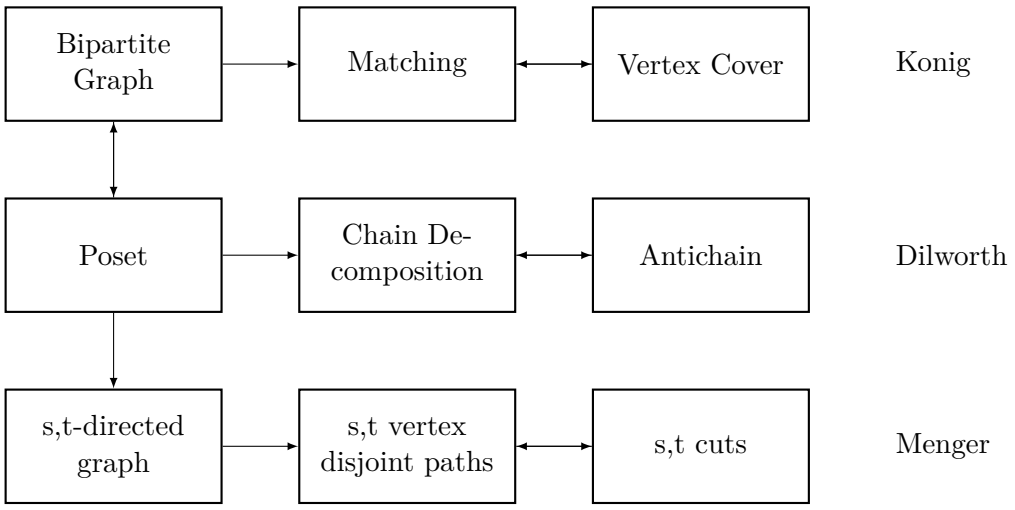


Figure 12.1: Various structures and transformations between them

12.2 A Bipartite Matching Algorithm

Given an undirected graph (V, E), a *matching* M is a subset of edges E such that no two edges are incident on the same vertex. As an example, consider a bipartite graph (L, R, E) such that the vertex set L denotes a set of men, R denotes a set of women and E denotes a symmetric "likes" relation. Then, a matching M is simply a set of edges such that no man or woman is adjacent to two edges in M. If the number of men and the number of women are equal to n and every man is matched, then we call that matching a *perfect matching*. The perfect matching may not always exist. The following famous Theorem gives a necessary and sufficient condition for a perfect matching to exist in a bipartite graph. It is based on the notion of *overdemanded* subset. We say that $L' \subseteq L$ is overdemanded if there exists $R' \subseteq R$ such that the set of neighbors of R' in the graph is L' and the size of R' is strictly bigger than $|L'|$. Intuitively, L' is overdemanded because R' can only match with vertices in L' but the size of L' is smaller than the size of R'. Equivalently, the absence of an overdemanded subset is Hall's neighborhood condition: $|N(R')| \geq |R'|$ for every $R' \subseteq R$, where $N(R')$ denotes the set of neighbors of R'. We can now state the Theorem.

Theorem 12.1 (Hall's Theorem) *The necessary and sufficient condition for a perfect matching to exist in a bipartite graph (L, R, E) where $|L| = |R|$, is that L does not have any overdemanded subset L'.*

Proof: *Necessity.* If a perfect matching exists, each $v \in R'$ is matched to a distinct neighbor, so $|N(R')| \geq |R'|$.

Sufficiency, by induction on $n = |R|$. The base $n = 0$ is trivial.

Case 1: Every proper non-empty subset $R' \subsetneq R$ satisfies $|N(R')| > |R'|$. Pick any $v \in R$ and match it to any neighbor u. Removing u and v reduces each neighborhood size by at most

one, so Hall's condition holds in the smaller graph. By induction, the remaining vertices have a perfect matching.

Case 2: Some proper non-empty $R^* \subsetneq R$ satisfies $|N(R^*)| = |R^*|$. By induction, R^* and $N(R^*)$ have a perfect matching. Remove these vertices. For any $R' \subseteq R \setminus R^*$, apply Hall's condition to $R^* \cup R'$ in the original graph: $|N(R^* \cup R')| \geq |R^*| + |R'|$. Since $N(R^*)$ accounts for exactly $|R^*|$ of these neighbors, R' has at least $|R'|$ remaining neighbors. By induction, the remaining graph also has a perfect matching.

■

Observe that even though Hall's theorem gives us insight about the perfect matching, it cannot be directly applied to efficiently check for perfect matching because there are 2^n possible subsets of L.

We now give an efficient algorithm for a more general problem called the *maximum cardinality matching problem*. This problem requires us to find a matching of the largest size. When the number of men and the number of women are equal to n, then a perfect matching exists iff the size of the maximum cardinality matching is n. The algorithm is based on the idea of increasing the size of a matching by using an *augmenting path*. Suppose that the algorithm has a matching M_t in iteration t. The algorithm is then guaranteed to find a matching M_{t+1} of size one bigger than M_t whenever a matching with a larger size exists. If we can make this idea work, then we can start with M_0 as the empty matching and then keep applying the idea to reach a maximum cardinality matching.

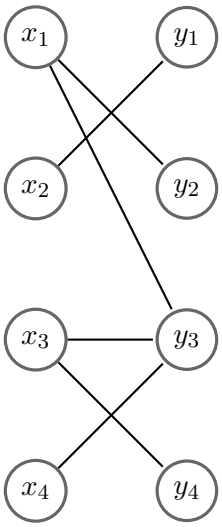


Figure 12.2: A bipartite graph

To understand the idea of an augmenting path for a matching M, we define a vertex as *exposed* if it is not incident on any edge in M. Now suppose that there exists a path from an exposed vertex to another exposed vertex that alternates between unmatched and matched edges. Such a path must start with an unmatched edge and end with an unmatched edge (by the definition of exposed vertices). Furthermore, such a path has an odd number of edges, with the unmatched edges exactly one more than the matched edges. Then, by simply switching

the matched with unmatched edges along that path, we can increase the size of matching by one.

Algorithm Seq-AugmentingPath: Finding a maximum cardinality matching

input: (L, R, E): bipartite graph
output: M: maximum cardinality matching
1 $M := \{\}$;
2 **while** *there exists an alternating path* P *in* (L, R, E) *for* M **do**
3 $\quad M :=$ matching M with edges switched in the path P;
4 **end**
5 **return** M;

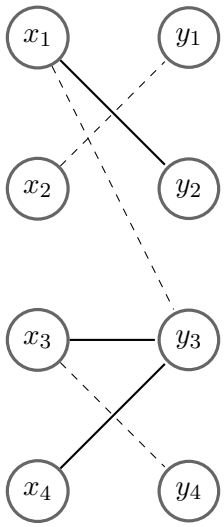


Figure 12.3: A matching M shown with dashed edges in the bipartite graph

How do we find an alternating path? One can start with any exposed vertex and do a breadth-first-search such that the BFS tree alternates between unmatched and matched edges. Such a search will result in either finding an alternating path or an overdemanded set. In our example of the matching M in Fig. 12.3, if we start from x_4, we get the alternating path x_4, y_3, x_1, y_2. If there are $2n$ vertices and m edges, every iteration of the *while* loop takes $O(m)$ time for breadth-first-search and there can be at most n iterations, giving us the time complexity of $O(nm)$. By augmenting along multiple paths in every iteration, the complexity can be reduced to $O(m\sqrt{n})$ [HK71].

We now give a sequential algorithm for a slight variant of the matching problem in a bipartite graph (L, R, E). Let the vertices of L be numbered $1..n$. Let L_i denote the set of vertices $\{v_1, v_2, \ldots, v_i\}$, for all $1 \leq i \leq n$. The output of our algorithm is a vector G such that $\sum_i G[i]$ is the size of the maximum matching in the graph (L_i, R, E_i) where E_i are the edges restricted to the set $L_i \cup R$. For simplicity, we set L_0 to $\emptyset$. We let S be the matched vertices in R. It is sufficient to describe the sequential algorithm when a new vertex v_i is added to the left side. We simply look for an alternating path from v_i to any unmatched vertex in R. If we

find any such path, then the newly matched vertex in R is added to S. If there is no path, then L_i is a constricted set, and the set of vertices in R matched to L_{i-1} is identical to the set of vertices matched to L_i. In either case, we continue the procedure to the next vertex in L, if any.

Algorithm BipartiteMatching: Sequential algorithm for Bipartite Matching

```
   input: (L, R, E): bipartite graph
   output: G[1..n]: matching indicator; S[1..n]: partner array
1  G: array[1..n] of int initially ∀j : G[j] := 0;
2  S: array[1..n] of 0..n initially ∀j : S[j] := 0;
3  for j := 1 to n do
4  |  if there exists an alternating path from v_j to any vertex k in R with (S[k] = 0)
   |   then
5  |  |  Assign S[] in the alternating path;
6  |  |  G[j] := 1;
7  |  end
8  end
9  return G;
```

Algorithm BipartiteMatching searches the element in the boolean lattice B_n such that $G[i]$ equals 1 if the size of the bipartite matching for (L_i, R, E_i) is one more than for (L_{i-1}, R, E_{i-1}), and 0 otherwise. The time complexity of the algorithm is $O(nm)$ because the *for* loop has n iterations and each iteration takes $O(m)$ to search for a path. The matching itself is stored in all nonzero S entries.

A key advantage of Algorithm BipartiteMatching is that it is a *deterministic* algorithm. Given a labeling of indices in L, it produces a single fixed G.

12.3 Chain Partition of a Poset

In this section, we show that maximum matching in a bipartite graph also allows us to partition a poset into the minimum number of chains.

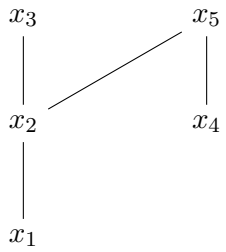


Figure 12.4: A poset

Assume that we have an algorithm for bipartite matching. How do we use it for chain decomposition? This relationship between chain decomposition and bipartite matching is based on the idea of a strict split of a poset.

Definition 12.2 (Strict Split of a Poset) *Given a poset P, the* **strict split** *of the poset P, denoted by $S(P)$, is a bipartite graph (L, R, E) where $L = \{x^- | x \in P\}$, $R = \{x^+ | x \in P\}$, and $E = \{(x^-, y^+) | x < y \text{ in } P\}$.*

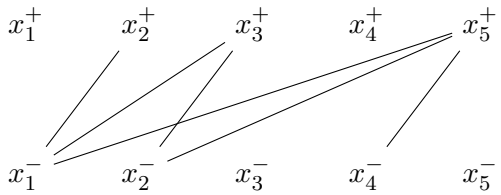


Figure 12.5: A strict split of the poset in fig. 12.4

We state the following theorem due to Fulkerson [Ful56].

Theorem 12.3 *[Ful56] Any matching in $S(P)$ can be reduced to a chain cover of P.*

Proof: We start with the trivial chain partition of P with each element in P being a chain. Thus, initially, we have n chains for n elements. We now consider each edge (x^-, y^+) in the matching. Each such edge implies that $x < y$ in P and we combine the chains corresponding to x and y. Since there can be at most one edge incident on x^- in a matching, x is the top element of its chain and similarly y is the lowest element of the chain. Therefore, these chains can be combined into one chain. If there are e edges in the matching, after this procedure we have a chain cover with $n - e$ chains.

■

Thus, any algorithm for bipartite matching can be used to determine a chain cover.

We now show an LLP algorithm that exhibits an antichain of size $n - e$, where e is the maximum matching size. We assume that we are given a minimum chain cover of size $m = n-e$. The algorithm maintains an index $G[j]$ into each chain C_j, starting at the bottom. It advances $G[j]$ whenever the current element $C_j[G[j]]$ is dominated by some element $C_k[G[k]]$ in another chain. At termination, the elements $\{C_1[G[1]], \ldots, C_m[G[m]]\}$ form a maximum antichain.

Algorithm LLP-Antichain: LLP algorithm for finding maximum antichain

input: a poset $(P, \leq)$ decomposed in m chains
output: $\{C_1[G[1]], C_2[G[2]], \ldots, C_m[G[m]]\}$: maximum antichain
1 G: array$[1..m]$ of int initially $\forall j : G[j] := 1$;
2 **forbidden**(j): $\exists k \neq j : C_j[G[j]] \leq C_k[G[k]]$ in P
3 $\Rightarrow$ **advance**: $G[j] := G[j] + 1$; `// move to successor on chain` j

The sequential time complexity of Algorithm LLP-Antichain is $O(nm)$. The total number of advances across all chains is at most n (the sum of all chain lengths). Each advance requires checking the forbidden predicate, which compares the current element against the current

element of each of the other $m-1$ chains using the poset relation; this takes $O(m)$ time. Hence the total work is $O(nm)$.

Example 12.4 [Running example] Consider the 5-element poset of Fig. 12.4, whose relations are $x_1 < x_2 < x_3$, $x_4 < x_5$, and $x_2 < x_5$. Its strict split $S(P)$ has $L = \{x_1^-, \ldots, x_5^-\}$, $R = \{x_1^+, \ldots, x_5^+\}$, and the six edges

$$(x_1^-, x_2^+),\ (x_1^-, x_3^+),\ (x_1^-, x_5^+),\ (x_2^-, x_3^+),\ (x_2^-, x_5^+),\ (x_4^-, x_5^+).$$

A maximum matching has size $e = 3$, e.g.

$$M^* \ = \ \{(x_1^-, x_2^+),\ (x_2^-, x_3^+),\ (x_4^-, x_5^+)\}.$$

By Theorem 12.3, the corresponding minimum chain cover of P has $n - e = 2$ chains: $C_1 = (x_1, x_2, x_3)$ and $C_2 = (x_4, x_5)$. We will revisit this instance in Sections 12.4, 12.5, and 12.7 to see how the same matching M^* yields a minimum vertex cover, a maximum antichain, and a maximum set of vertex-disjoint paths.

12.4 Vertex Cover in a Bipartite Graph

A vertex cover of a graph (V, E) is a subset of vertices $V' \subseteq V$ such that every edge is incident on some vertex in V'. Computing the vertex cover of a general graph is NP-complete, but the vertex cover of a bipartite graph can be done efficiently. In fact, we have the following Theorem.

Theorem 12.5 (König-Egerváry Theorem [Kon31, Ege31]) *In a* bipartite *graph, the maximum size of a matching equals the minimum size of a vertex cover.*

We give an algorithmic proof of the above theorem. Given a matching M for any bipartite graph, we construct its vertex cover as follows. For every edge $(u, v) \in M$ where $u \in L$ and $v \in R$, we are going to choose one of the vertices $\{u, v\}$ in the vertex cover. This vertex cover will have size equal to $|M|$. Furthermore, there cannot be a vertex cover of strictly smaller size because any vertex cover must include at least one of the vertices in $\{u, v\}$ to cover all the edges in M. Now, the only question is which of the vertices $\{u, v\}$ do we choose to be part of the vertex cover. The answer is quite simple. If there is any vertex x adjacent to u such that x is not part of the matching M, then u would have to be part of the vertex cover. Otherwise, the edge (x, u) would not be covered. Can we have a vertex x that is adjacent to both u and v? No, because this would form an odd size cycle that cannot be present in a bipartite graph. Finally, could we have two vertices x and y that are not part of the matching M such that x is adjacent to u and y is adjacent to v? In this case, M is not a matching of maximum size because we can include (x, u) and (v, y) instead of just the edge (u, v).

Algorithm VertexCoverFromMatching gives an algorithm to convert a bipartite matching into a vertex cover. It takes as input a bipartite graph and a matching M. The boolean array C gives the vertex cover. The variable array *partner* is used to mark partner of any vertex in

Algorithm VertexCoverFromMatching: Parallel algorithm to convert a matching to a vertex cover in a bipartite graph

input: (L, R, E): bipartite graph; M: matching in the bipartite graph (subset of E)
output: $C[1..n]$: vertex cover set

```
1  C: array[1..n] of boolean initially ∀j : C[j] := false ;        // vertex cover set
2  partner: array[1..n] of int initially ∀j : partner[j] := 0;
3  for (u, v) ∈ M in par (assume u < v) do
4  |  C[u] := true;
5  |  partner[u] := v;
6  |  partner[v] := u;
7  end
8  for (u, v) ∈ E − M in par (assume u < v) do
9  |  if ¬C[u] ∧ ¬C[v] then
10 |  |  if partner[u] ≠ 0 then
11 |  |  |  C[partner[u]] := false;
12 |  |  |  C[u] := true;
13 |  |  else
   |  |  |  // partner[u] = 0 implies partner[v] ≠ 0
14 |  |  |  C[partner[v]] := false;
15 |  |  |  C[v] := true;
16 |  |  end
17 |  end
18 end
19 return C;
```

matching M. If a vertex u has no partner then $partner[u]$ is zero. We initialize the *partner* array to zero. The algorithm first sets the partner for each vertex by going over all the edges in M. It also adds the lower numbered vertex u in the vertex cover for every edge (u, v) in M. This can be done in parallel in $O(1)$ time. At this point, we have exactly one vertex in C for every edge in M. The set C covers all edges in M, but may not cover all other edges.

In the second step, we ensure that all edges are covered. We maintain the invariant that there is exactly one vertex in C for every edge in M. We go over all edges (u, v) in parallel where u is less than v. If neither of the endpoints of the edge are in C, then we consider two cases. From the invariant, we know that the edge (u, v) is not in matching M. If u is matched to some vertex w in M, i.e., $partner[u]$ equals w, then we remove w from C and add u to C. Otherwise, v is matched to some vertex w' in M. In this case, we remove w' from C and add v to C. If neither u nor v is matched, then M is clearly not a maximum matching since it can be increased in size by adding the edge (u, v). Observe that we maintain the invariant that there is exactly one vertex in C for every edge in M.

We now show that in the second step it cannot happen that for a matched edge $(u, w) \in M$, some edge requires $C[u]$ be set to 1 and some other edge requires $C[w]$ to be set to 1. An edge forces $C[u]$ to be set to 1 only when that edge is uncovered, which requires $u \notin C$; by the invariant that exactly one endpoint of each matched edge lies in C, this forces u's partner $w \in C$. But then every edge incident to w is already covered, so no edge can demand that $C[w]$ be set to 1. Hence the two demands cannot occur together for (u, w), and the invariant—exactly one vertex of C for every edge of M—is preserved.

Example 12.6 [Running example, continued] For the strict-split graph of Example 12.4 and the matching M^*, Algorithm VertexCoverFromMatching produces a minimum vertex cover of size 3, e.g. $\{x_1^-,\ x_2^-,\ x_5^+\}$: x_1^- covers the three edges out of x_1^-, x_2^- covers the two remaining edges out of x_2^-, and x_5^+ covers (x_4^-, x_5^+). By König's theorem, no smaller vertex cover exists, since the maximum matching has size 3.

12.5 Dilworth's Theorem and Maximum Antichain

We now establish the relationship between the chain decomposition of a poset and the maximum antichain. Recall from Section 12.3 that the minimum chain cover of a poset P with n elements can be obtained from the maximum matching in the strict split $S(P)$. If the maximum matching has e edges, then the minimum chain cover has $n - e$ chains.

Definition 12.7 (Antichain) *An antichain in a poset $(P, \leq)$ is a set of elements $A \subseteq P$ such that no two elements in A are comparable, i.e., for all $x, y \in A$ with $x \neq y$, neither $x \leq y$ nor $y \leq x$.*

Theorem 12.8 (Dilworth's Theorem [Dil50]) *In any finite poset, the minimum number of chains needed to cover all elements equals the maximum size of an antichain.*

Proof: Let m be the minimum number of chains in a chain cover and let a be the maximum size of an antichain.

First, we show $m \geq a$. Any antichain has at most one element from each chain (since elements in the same chain are comparable). Hence, $a \leq m$.

Second, we show $m \leq a$ by constructing an antichain of size m. Given a minimum chain cover with m chains $C_1, C_2, \ldots, C_m$, we use the LLP algorithm (Algorithm LLP-Antichain) to find an antichain. The algorithm maintains a vector G where $G[j]$ is an index into chain C_j. Let $C_j[G[j]]$ denote the $G[j]$-th element in chain C_j from the bottom. The predicate *forbidden*(j) is true if there exists another chain k such that $C_j[G[j]] \leq C_k[G[k]]$ in the poset, i.e., the current element of chain j is dominated by the current element of some other chain. The advance operation moves $G[j]$ to the next element in the chain. When no index is forbidden, the set $\{C_1[G[1]], C_2[G[2]], \ldots, C_m[G[m]]\}$ forms an antichain of size m.

We now verify that the algorithm terminates with a valid antichain. The key invariant is that an antichain of size m always exists at or above the current state G. Initially, since the chain cover has m chains, each chain contributes exactly one element to any antichain, and such an antichain exists (we will show this by the end of the proof). Whenever the algorithm advances $G[j]$, the set of vectors $\geq G$ shrinks, but the invariant is maintained: if $C_j[G[j]] \leq C_k[G[k]]$ for some k, then no antichain picking $C_j[G[j]]$ from chain j and $C_k[G[k]]$ from chain k is valid, so any antichain above G must pick a higher element from chain j. Thus advancing $G[j]$ does not eliminate any valid antichain. Since the set of valid antichains above G is nonempty and each chain is finite, the algorithm never advances past the end of any chain. At termination, for every pair (j, k), neither $C_j[G[j]] \leq C_k[G[k]]$ nor $C_k[G[k]] \leq C_j[G[j]]$ holds (otherwise one of them would be forbidden). Hence the selected elements are pairwise incomparable, giving us an antichain of size m.

■

Example 12.9 [Running example, continued] The two-chain cover from Example 12.4 is $C_1 = (x_1, x_2, x_3)$, $C_2 = (x_4, x_5)$. Algorithm LLP-Antichain starts with $G = [1, 1]$, so the representatives are x_1 and x_4. These two elements are incomparable in P, so neither index is forbidden, and the algorithm terminates immediately with the maximum antichain $\{x_1, x_4\}$ of size $2 = n - e$. This confirms Dilworth's theorem on the running example.

Combining Dilworth's theorem with Fulkerson's theorem (Theorem 12.3), we get that for a poset with n elements whose strict split has maximum matching of size e:

- Minimum chain cover size $= n - e$
- Maximum antichain size $= n - e$
- Minimum vertex cover in $S(P) = e$ (by König's theorem)

This establishes the correspondence shown in the second row of Figure 12.1: the chain decomposition and antichain of a poset are dual to each other via Dilworth's theorem, just as matching and vertex cover are dual via König's theorem.

12.6 Reductions Between Structures

Figure 12.1 shows multiple reductions between combinatorial structures. We now justify each arrow in the figure.

Bipartite Graph ↔ Poset

Given a bipartite graph (L, R, E), we can construct a poset as follows. Let the elements of the poset be $L \cup R$ and define the partial order such that $x < y$ iff $x \in L$, $y \in R$, and $(x, y) \in E$. Conversely, given a poset P, its strict split $S(P)$ gives a bipartite graph.

More precisely, the reduction from posets to bipartite graphs uses the strict split (Definition in Section 12.3): given a poset $(P, \leq)$, we construct the bipartite graph $S(P) = (L, R, E)$ where each element x appears as x^- in L and x^+ in R, with edge (x^-, y^+) whenever $x < y$ in P.

The key property is that a matching of size e in $S(P)$ corresponds to combining e pairs of consecutive elements in chains, yielding a chain cover of size $n - e$ (Theorem 12.3).

Matching ↔ Vertex Cover (König's Theorem)

The arrow from matching to vertex cover is justified by Algorithm VertexCoverFromMatching in Section 12.4: given any maximum matching, we can construct a minimum vertex cover of the same size in $O(1)$ parallel time.

The arrow from vertex cover to matching follows because any vertex cover of size k must include at least one endpoint of each matched edge. Hence, the vertex cover size is at least the matching size. Combined with König's theorem, a minimum vertex cover immediately certifies the optimality of a maximum matching.

Chain Decomposition ↔ Antichain (Dilworth's Theorem)

Given a minimum chain cover of size m, Algorithm LLP-Antichain produces a maximum antichain of size m using an LLP computation. Conversely, an antichain of size a certifies that any chain cover requires at least a chains (since each chain contains at most one antichain element).

Poset → s, t-Directed Graph

Given a poset $(P, \leq)$, we construct a directed graph as follows. For each element $x \in P$, create two vertices x_{in} and x_{out} connected by a directed edge (x_{in}, x_{out}) with capacity 1. For each relation $x < y$ in P, add an edge (x_{out}, y_{in}) with capacity 1. Add a source vertex s with edges to all x_{in} where x is a minimal element, and a sink vertex t with edges from all x_{out} where x is a maximal element.

In this construction, vertex-disjoint s-t paths correspond to chains in the poset, and a minimum s-t vertex cut corresponds to a maximum antichain. This gives an alternative proof of Dilworth's theorem via Menger's theorem.

12.7 Menger's Theorem

We now discuss the third row of Figure 12.1: the relationship between vertex-disjoint paths and vertex cuts in directed graphs.

Definition 12.10 (Vertex-Disjoint Paths) *Given a directed graph $\mathcal{G} = (V, E)$ with two distinguished vertices s and t, a set of s-t paths is* vertex-disjoint *if no two paths share any internal vertex (i.e., any vertex other than s and t).*

Definition 12.11 (s-t Vertex Cut) *An s-t vertex cut is a set of vertices $C \subseteq V \setminus \{s, t\}$ whose removal disconnects t from s, i.e., there is no directed path from s to t in $\mathcal{G} - C$.*

Theorem 12.12 (Menger's Theorem [Men27]) *In a directed graph $\mathcal{G} = (V, E)$, the maximum number of vertex-disjoint s-t paths equals the minimum size of an s-t vertex cut.*

Proof: Let k be the maximum number of vertex-disjoint paths and c be the minimum vertex cut size.

First, $k \leq c$: any s-t vertex cut must contain at least one internal vertex from each vertex-disjoint path (otherwise that path would still connect s to t). Hence $c \geq k$.

Second, $k \geq c$: we reduce the problem to max-flow. Replace each internal vertex v by two vertices v_{in} and v_{out} with a directed edge (v_{in}, v_{out}) of capacity 1. Replace each original edge (u, v) by (u_{out}, v_{in}) with capacity ∞ (or n). The maximum flow from s to t in this network equals the maximum number of vertex-disjoint s-t paths (since each internal vertex can be used at most once due to capacity 1). By the max-flow min-cut theorem, this equals the minimum cut, which corresponds to removing vertices (those with saturated vertex-edges) that disconnect s from t.

■

Given the reduction to max-flow described in the proof, we can compute the maximum number of vertex-disjoint s-t paths using any max-flow algorithm on the transformed network. Since all capacities are 0 or 1 on vertex edges, the max-flow is integral and equals the number of vertex-disjoint paths.

The sequential time complexity is $O(km)$ where k is the number of vertex-disjoint paths (at most n), since each augmenting path takes $O(m)$ time in the transformed network (which has $O(m)$ edges from the vertex splitting) and there are k such paths.

As with König's and Dilworth's theorems, Menger's theorem provides a bidirectional certification. A set of k vertex-disjoint s-t paths certifies that any vertex cut requires at least k vertices (since each path must pass through at least one cut vertex). Conversely, a vertex cut of size c certifies that no more than c vertex-disjoint paths exist (since removing the c cut vertices destroys all s-t paths). When $k = c$, each structure certifies the optimality of the other.

Menger's theorem generalizes König's theorem. Given a bipartite graph (L, R, E), construct a directed graph as follows: add a source s with edges to all vertices in L, direct all edges from L to R, and add a sink t with edges from all vertices in R. Split each vertex in $L \cup R$ into in/out pairs with unit capacity. Then:

- Vertex-disjoint s-t paths correspond to matched edges.
- A minimum s-t vertex cut corresponds to a minimum vertex cover.

Thus, König's theorem (max matching = min vertex cover) is a special case of Menger's theorem (max vertex-disjoint paths = min vertex cut) applied to the bipartite construction.

As described in Section 12.6, given a poset $(P, \leq)$, we construct a directed graph with vertex splitting. In this construction:

- Each s-t path passes through a sequence of elements $x_1 < x_2 < \cdots < x_k$, corresponding to a chain.
- The maximum number of vertex-disjoint s-t paths equals the minimum chain cover size.
- A minimum s-t vertex cut corresponds to a maximum antichain: the elements whose removal disconnects all chains from source to sink.

By Menger's theorem, min chain cover = max antichain, which is exactly Dilworth's theorem. This provides a unified view: all three min-max results in Figure 12.1 (König, Dilworth, Menger) are instances of the max-flow min-cut theorem applied to appropriately constructed networks.

Edge-disjoint paths

There is a parallel statement of Menger's theorem for edges, which is in many ways the more natural form for the max-flow / min-cut framework.

Theorem 12.13 (Edge-Menger) *In a directed graph $\mathcal{G} = (V, E)$, the maximum number of edge-disjoint s-t paths equals the minimum number of edges in an s-t edge cut.*

Proof: Assign every edge unit capacity. By integrality of the max-flow on a unit-capacity network, the maximum flow value equals the maximum number of edge-disjoint s-t paths (each edge can carry flow 0 or 1, and conservation routes integer paths). By the max-flow min-cut theorem, this also equals the minimum capacity of any s-t cut, which on a unit-capacity graph is exactly the size of the smallest edge set whose removal disconnects s from t.

■

The vertex version (Theorem 12.12) reduces to the edge version by the standard vertex-splitting trick of the proof above; conversely, the edge version reduces to vertex-Menger on the line graph. Both forms thus express the max-flow / min-cut duality, specialized to unit-capacity networks of two flavors.

Example 12.14 [Running example, continued] Build the s-t graph for the running poset of Example 12.4 as follows. Split each x_i into x_i^-, x_i^+ as in the strict split, add a source s with unit-capacity edges to every x_i^- and a sink t receiving unit-capacity edges from every x_i^+, and orient the strict-split edges from L to R. The maximum vertex-disjoint s-t paths in this graph have size 3, matching the maximum-matching size of $S(P)$:

$$s \to x_1^- \to x_2^+ \to t, \quad s \to x_2^- \to x_3^+ \to t, \quad s \to x_4^- \to x_5^+ \to t.$$

Note that the first two paths are vertex-disjoint because x_2^+ and x_2^- are distinct vertices in the split graph, even though they both originate from x_2. A minimum vertex cut of size 3 — the same $\{x_1^-, x_2^-, x_5^+\}$ from Example 12.4 continued — disconnects s from t, confirming Menger's theorem on the running example.

12.8 Summary

This chapter presented algorithms for bipartite matching and related combinatorial optimization problems, and established the connections between matching, vertex cover, chain decomposition, antichain, and vertex-disjoint paths through König's, Dilworth's, and Menger's theorems.

Problem	Algorithm	Time Complexity
Maximum Matching	Augmenting Path	$O(nm)$
Chain Partition	Reduction to Matching	$O(nm)$
Maximum Antichain	LLP-Antichain	$O(nm)$
Minimum Vertex Cover	Matching to Vertex Cover	$O(n)$
Vertex-Disjoint Paths	Max-Flow Reduction	$O(km)$

Here n is the number of vertices, m is the number of edges, and k is the number of vertex-disjoint paths.

12.9 Problems

1. **(Konig Egervary Theorem)** Show that the minimum size of a vertex cover is equal to the maximum size of a matching in a bipartite graph.

2. **(Matching LP Dual)** Give a linear program formulation for the maximum matching problem and show that its dual corresponds to the minimum vertex cover problem.

3. **(Hall's Theorem)** Prove Hall's Theorem.

4. **(Edmonds Matrix Determinant)** Edmonds' matrix is defined for a bipartite graph (U, V, E) with $|U| = |V| = n$ as follows. Let the vertices in U be $u_1, u_2, \ldots, u_n$ and the vertices in V be $v_1, v_2, \ldots, v_n$. We set $A[i, j] = x_{i,j}$ if $(u_i, v_j) \in E$ and 0 otherwise. Show that Edmonds' matrix of a balanced bipartite graph has a perfect matching iff the polynomial corresponding to the determinant of Edmonds' matrix is not identically zero.

5. **(Edge Disjoint Paths Flow)** We are given an unweighted graph with two distinguished vertices s and t. We are required to find the total number of paths from s to t such that these paths do not share any edge. Give an algorithm to find this number, and justify its correctness. Suppose that the graph has n vertices and m edges, give the time complexity of your algorithm.

6. **(Bipartite Vertex Cover Output)** Modify Algorithm BipartiteMatching to output the vertex cover in addition to bipartite matching.

7. **(Berge's Augmenting Path Theorem)** Prove that a matching M in a bipartite graph is maximum if and only if there is no augmenting path with respect to M.

8. **(K-Regular Bipartite Decomposition)** Prove that every k-regular bipartite graph (with $k \geq 1$) has a perfect matching. Conclude that the edge set can be partitioned into k perfect matchings.

9. **(Min Cost Bipartite Matching) (Assignment / Min-Cost Bipartite Matching.)** Given a bipartite graph $G = (L, R, E)$ with $|L| = |R| = n$ and non-negative edge costs c_{uv}, formulate the minimum-cost perfect matching problem as a minimum-cost flow problem. Briefly describe the resulting algorithm and its running time.

10. **(Job Assignment Feasibility) (Job-Assignment Variant.)** A company has n workers and $m \geq n$ jobs, where each worker can perform only a subset of the jobs. Each worker must be assigned to exactly one job; each job at most one worker. Give an efficient algorithm to decide whether such an assignment exists and compute it when it does.

11. **(Dynamic Edge Removal)** An edge $(u, v) \in E$ is deleted. If $(u, v) \notin M$ the matching is unaffected; if $(u, v) \in M$, the matching $M \setminus \{(u, v)\}$ has one fewer edge. Decide whether $|M|$ can be restored to its previous size, and give an algorithm running in $O(|V| + |E|)$ time.

12. **(Constrained Matching with Forbidden Pairs)** A set $F \subseteq E$ of forbidden pairs must *not* appear in the output matching. Decide whether a perfect matching avoiding F exists, and give an algorithm that returns one in $O(\sqrt{n}\, m)$ time.

13. **(Fair Two-sided Matching)** Each $u \in L$ and $r \in R$ has a strict preference list. Find a matching M that minimises $\max_{e \in M}(\text{rank}_L(e) + \text{rank}_R(e))$, i.e. the worst-pair regret. Compare to the egalitarian objective (sum of ranks).

14. **(König's Theorem via Max-Flow Min-Cut)** Given a bipartite graph (L, R, E), construct a flow network by adding a source s with unit-capacity edges to every vertex in L, and a sink t with unit-capacity edges from every vertex in R. Direct all edges from L to R with capacity 1.

    (a) Show that every integral s-t flow of value k corresponds to a matching of size k, and vice versa.

    (b) Show that every minimum s-t cut of capacity c corresponds to a vertex cover of size c, and vice versa.

(c) Conclude König's theorem (max matching = min vertex cover) from the max-flow min-cut theorem.

15. **(Dilworth's Theorem via König's Theorem)** Let $(P, \leq)$ be a finite poset on n elements. Recall the strict split bipartite graph $S(P) = (L, R, E)$ where $L = \{x^- : x \in P\}$, $R = \{x^+ : x \in P\}$, and $(x^-, y^+) \in E$ iff $x < y$ in P.

    (a) Show that a matching of size m in $S(P)$ yields a chain partition of P into $n - m$ chains.

    (b) Show that a chain partition of size k yields a matching of size $n - k$.

    (c) Let C be a minimum vertex cover of $S(P)$. Define $A = \{x \in P : x^- \notin C \text{ and } x^+ \notin C\}$. Prove that A is an antichain.

    (d) Using König's theorem and parts (a)–(c), prove Dilworth's theorem: *the minimum number of chains needed to cover P equals the maximum size of an antichain.*

16. **(Menger's Theorem via Max-Flow Min-Cut)** Let $G = (V, E)$ be a directed graph with distinguished vertices s and t.

    (a) Construct a flow network from G by replacing each internal vertex v with two vertices v_{in} and v_{out} connected by a capacity-1 edge, and replacing each original edge (u, v) with edge (u_{out}, v_{in}) of capacity n. Show that the maximum integral flow equals the maximum number of vertex-disjoint s-t paths.

    (b) Show that the minimum cut in this network corresponds to a minimum s-t vertex cut in G.

    (c) Conclude Menger's theorem from the max-flow min-cut theorem.

17. **(Birkhoff's Theorem on Doubly Stochastic Matrices)** An $n \times n$ matrix A is *doubly stochastic* if all entries are nonnegative and every row and column sums to 1. A *permutation matrix* is a $\{0, 1\}$-matrix with exactly one 1 in each row and column. Birkhoff's theorem states that every doubly stochastic matrix is a convex combination of permutation matrices.

    (a) Show that the support of a doubly stochastic matrix (the bipartite graph of its nonzero entries) contains a perfect matching. (*Hint:* use Hall's theorem or the k-regular decomposition result.)

    (b) Using part (a), prove Birkhoff's theorem by induction on the number of nonzero entries. Specifically, given a doubly stochastic matrix A, find a permutation matrix P whose support is contained in the support of A, and show that $A' = \frac{1}{1-\lambda}(A - \lambda P)$ is doubly stochastic with fewer nonzero entries for a suitable $\lambda > 0$.

    (c) Give an upper bound on the number of permutation matrices needed in the decomposition.

18. **(König's Matrix Theorem)** Let A be an $m \times n$ matrix with entries in $\{0, 1\}$. A set of 1s in A is *independent* if no two are in the same row or column. A *line cover* is a collection of rows and columns such that every 1 in A lies in at least one selected row or column.

(a) Show that the maximum number of independent 1s equals the maximum matching in the bipartite graph $G_A = (R, C, E)$ where R is the set of rows, C is the set of columns, and $(r_i, c_j) \in E$ iff $A[i, j] = 1$.

(b) Show that a line cover of size k corresponds to a vertex cover of size k in G_A, and vice versa.

(c) Conclude König's matrix theorem: *the maximum number of independent* 1*s equals the minimum number of lines needed to cover all* 1*s.*

(d) Apply the result to the following matrix and exhibit both a maximum set of independent 1s and a minimum line cover:

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

12.10 Bibliographic Remarks

The sequential augmenting path algorithm, detailed in Section 12.2, is a cornerstone of matching theory, achieving a time complexity of $O(nm)$ for a graph with n vertices and m edges. This algorithm is enhanced by the seminal work of Hopcroft and Karp [HK71], who introduced a method to augment multiple paths per iteration, reducing the complexity to $O(m\sqrt{n})$. Their approach remains a standard for efficient bipartite matching. Hall's Theorem, presented in the chapter, is a fundamental result characterizing the existence of a perfect matching, with roots in early combinatorial optimization. For a deeper exploration of matching theory, including Hall's Theorem and its extensions, Lovász and Plummer [LP86] offer a classic and authoritative reference. Section 12.3 connects bipartite matching to poset chain partitions through strict split construction, leveraging Fulkerson's theorem [Ful56]. Fulkerson's result elegantly reduces matchings in the bipartite graph $S(P)$ to chain covers in the poset P.

The König-Egervary theorem [Kon31, Ege31] establishing the equality of maximum matching and minimum vertex cover in bipartite graphs is one of the earliest results in combinatorial optimization.

Dilworth's theorem [Dil50] on the duality between chain covers and antichains in posets has deep connections to matching theory via the strict split construction. Menger's theorem [Men27] on vertex-disjoint paths predates both König's and Dilworth's results and provides a unified framework through which both can be derived as special cases of the max-flow min-cut theorem [JF56].

Chapter 13

Intractability

13.1 Introduction

This chapter discusses NP-completeness, a foundational concept in computational complexity that identifies the hardest problems in NP. We define P and NP, explore polynomial-time reductions, and prove NP-completeness for a broad set of problems: 3-SAT, Clique, Vertex Cover, Independent Set, Subset Sum, Hamiltonian Cycle, and TSP. We also discuss methods to deal with NP-complete problems.

The classes P and NP classify computational problems by their solvability and verifiability. Pincludes problems with polynomial-time deterministic algorithms, while NPincludes those verifiable in nondeterministic polynomial time. The unresolved question P = NP drives much of complexity theory.

The problems are specified as requiring an *output* for the given *input*. We will use n for the size of the input. Our interest would be in determining the time required to compute the output. The output we require would be *binary*. For example, our input may be a Boolean expression B on Boolean variables. We would require the output to be 1 if the Boolean expression is satisfiable and 0 otherwise. At first, it may seem that in practical applications, the user may be interested in finding a satisfying assignment rather than simply that B is satisfiable. However, if someone gave us a method f to efficiently check whether B is satisfiable, then we can use f to determine a satisfying assignment. If B is satisfiable, then we can easily compute another Boolean expression B' in which the Boolean variable x_1 is set to true. We now ask whether B' is satisfiable. If B' is satisfiable, we continue with the other variables. If B' is not satisfiable, then we know that x_1 must be false. We compute a Boolean expression B'' from B by setting x_1 to false. Thus, in n applications of f, we would have found a satisfying assignment. This idea is applicable to all problems considered in this chapter, and we will restrict ourselves to such problems. This phenomenon is known as *self-reducibility*: a decision oracle for a problem can be used to solve the corresponding search problem with only a polynomial number of oracle calls.

This chapter is organized as follows. Section 13.2 defines the complexity class P. Section 13.3 introduces polynomial-time reductions, defines NP and NP-completeness, and proves

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

NP-completeness for 3-SAT, Independent Set, and Vertex Cover. Section 13.4 extends the catalog by proving NP-completeness for Clique, Subset Sum, Hamiltonian Cycle, and TSP, plus further problems summarized in Figure 13.8. Section 13.5 discusses the co-NP class. Section 13.6 surveys five strategies for coping with NP-hardness in practice; the approximation strategy is developed in depth in Chapter 14.

13.2 Class P

Definition 13.1 (Class P) *The complexity class* P *is the set of all decision problems* $L \subseteq \{0,1\}^*$ *for which there exists a deterministic Turing machine* M *and a polynomial function* $p : \mathbb{N} \to \mathbb{N}$ *such that:*

1. *M halts on every input $x \in \{0,1\}^*$ in at most $p(|x|)$ steps, where $|x|$ denotes the length of x,*
2. *M accepts x if and only if $x \in L$.*

Formally,

$$\mathrm{P} = \{L \mid \text{there exists a deterministic Turing machine } M \text{ and a polynomial } p \text{ such that } M \text{ decides } L \text{ in time } O(p(n))\},$$

where n is the input size.

The notation used in the definition is as follows:

- P: The complexity class of decision problems solvable in polynomial time by a deterministic Turing machine. The boldface emphasizes it as a formal class name.
- L: A language, representing a decision problem, which is a set of strings over the binary alphabet $\{0,1\}$.
- M: A deterministic Turing machine, a theoretical model of computation with a single, fixed sequence of steps for any input. For our purposes, we can use any practical programming language such as Java or C++.
- p: A polynomial function $p : \mathbb{N} \to \mathbb{N}$, mapping natural numbers (input sizes) to natural numbers (time bounds).

Informally, a problem L is in P if we can write a program that determines if the given instance x is in L in time that is polynomial in the size of x. Most of the problems that we have seen in this book so far belong to P.

Example 13.2 The problem PATH of deciding whether vertex t is reachable from s in a graph is in P; BFS decides it in $O(n+m)$ time (Section 5.3).

13.3 Polytime Reductions

We define a relation between various problems as follows. We say that a problem π_1 is at least as hard as another problem π_2 if by using solutions to the problem π_1 we can solve the problem π_2 in time polynomial in the size of the problem.

Definition 13.3 (Polynomial-Time Reduction (Karp Reduction)) π_2 *is* polynomially reducible *to* π_1*, denoted* $\pi_2 \leq_P \pi_1$*, if there exists a polynomial-time computable function* f *such that for every instance* x *of* π_2*,* $x \in \pi_2$ *iff* $f(x) \in \pi_1$*.*

It is easy to verify that $\leq_P$ is a reflexive and transitive relation.

A consequence of the above definition is that if there exists an efficient algorithm to solve π_1, then we can use that solution to solve π_2. Another consequence, important to this chapter, is that if π_2 is *hard* to solve, then π_1 is also hard to solve.

A template for NP-completeness proofs. Every NP-completeness proof in this chapter follows the same four-step pattern:

1. **Membership in** NP**.** Exhibit a short certificate for yes instances of the target problem Π that can be verified in polynomial time.
2. **Choose a source.** Pick a problem Π' already known to be NP-complete.
3. **Reduction.** Describe a polynomial-time construction f that maps every instance x of Π' to an instance $f(x)$ of Π.
4. **Correctness.** Prove both directions: $x \in \Pi' \Rightarrow f(x) \in \Pi$, and $f(x) \in \Pi \Rightarrow x \in \Pi'$.

Readers are encouraged to identify these four steps in each proof that follows.

Let us show that the problem of vertex cover is harder than the problem of independent set in an undirected graph. Let $G = (V, E)$ be an undirected graph with the vertex set V and the edge set E.

Definition 13.4 (Independent Set) *The* Independent Set *problem asks: Given a graph* G *and an integer* k*, does there exist a subset* $S \subseteq V$ *of at least* k *vertices such that no two vertices in* S *are adjacent (i.e.,* $\forall u, v \in S, \{u, v\} \notin E$*)? Such a set* S *is called an* independent set*.*

Definition 13.5 (Vertex Cover) *The* Vertex Cover *problem asks: Given a graph G and an integer k, does there exist a subset $C \subseteq V$ of at most k vertices such that every edge in E is incident to at least one vertex in C (i.e., $\forall \{u, v\} \in E, u \in C$ or $v \in C$)? Such a set C is called a* vertex cover.

The following theorem captures the complementary relationship between Independent Set and Vertex Cover.

Theorem 13.6 (Independent Set – Vertex Cover duality) *In any graph $G = (V, E)$ with $|V| = n$, a set $S \subseteq V$ is an independent set iff $V \setminus S$ is a vertex cover. Consequently, G has an independent set of size $\geq k$ iff it has a vertex cover of size $\leq n - k$.*

Proof:

- ($\Rightarrow$): Suppose $S \subseteq V$ is an independent set with $|S| \geq k$. Define $C = V \setminus S$. Since S is independent, no edge $\{u, v\} \in E$ has both $u, v \in S$. Thus, for every edge $\{u, v\} \in E$, at least one of u or v is in $C = V \setminus S$. Hence, C is a vertex cover. Moreover, $|C| = |V| - |S| \leq |V| - k = n - k$.
- ($\Leftarrow$): Suppose $C \subseteq V$ is a vertex cover with $|C| \leq n - k$. Define $S = V \setminus C$. For any edge $\{u, v\} \in E$, since C is a vertex cover, at least one of u or v is in C, so at most one is in S. But if both $u, v \in S$, then neither is in C, contradicting that C covers $\{u, v\}$. Thus, no edge connects vertices in S, so S is an independent set. Moreover, $|S| = |V| - |C| \geq |V| - (n - k) = k$.

■

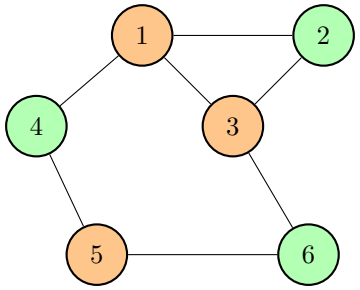


Figure 13.1: A graph $G = (V, E)$ with $V = \{1, \ldots, 6\}$ and edges $\{\{1,2\},\{1,3\},\{2,3\},\{1,4\},\{4,5\},\{5,6\},\{3,6\}\}$. The set $S = \{2, 4, 6\}$ is a maximum independent set (shaded green); its complement $C = \{1, 3, 5\}$ is the corresponding minimum vertex cover (shaded orange).

Example 13.7 In Figure 13.1, G has vertex set $V = \{1, \ldots, 6\}$. The set $S = \{2, 4, 6\}$ is an independent set of size 3: no two of its vertices are joined by an edge. It is in fact a maximum independent set, since the triangle $\{1, 2, 3\}$ forces any larger independent set to pick at least two pairwise-nonadjacent vertices from $\{4, 5, 6\}$, but then one of $\{1, 2, 3\}$ becomes adjacent to a chosen pendant. Its complement $C = V \setminus S = \{1, 3, 5\}$ is a vertex cover: every edge has at least one endpoint in $\{1, 3, 5\}$. Consistent with Theorem 13.6, $|S| = 3 = n - |C|$.

Theorem 13.6 immediately yields a polynomial-time reduction Independent Set $\leq_P$ Vertex Cover: given an instance (G, k) of Independent Set, output the Vertex Cover instance $(G, |V| - k)$. The reduction runs in $O(|V|)$ time, and by the theorem, (G, k) is a *yes* instance of Independent Set iff $(G, |V| - k)$ is a *yes* instance of Vertex Cover.

Conversely, we argue that the Independent Set problem is at least as hard as the Vertex Cover problem by providing a polynomial-time reduction from Vertex Cover to Independent Set.

Given an instance $(G = (V, E), k)$ of Vertex Cover, construct an instance of Independent Set as follows:

- Use the same graph $G = (V, E)$.
- Set the parameter $k' = |V| - k$.

The reduction is polynomial-time, identical to the previous reduction.

Since we have reduced Vertex Cover to Independent Set in polynomial time, Independent Set is at least as hard as Vertex Cover. Formally, Vertex Cover $\leq_P$ Independent Set.

The bidirectional reductions show that Independent Set and Vertex Cover are computationally equivalent in terms of polynomial-time solvability: Independent Set $\leq_P$ Vertex Cover and Vertex Cover $\leq_P$ Independent Set. The reductions exploit the complementary relationship: S is an independent set if and only if $V \setminus S$ is a vertex cover.

We now show that the problem of *Set Cover* is harder than the vertex cover.

Definition 13.8 (Set Cover) *The* Set Cover problem *is defined as follows: Given a universe U of n elements $\{e_1, e_2, \ldots, e_n\}$, a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of subsets of U such that $\bigcup_{i=1}^{m} S_i = U$, and an integer k, does there exist a subcollection $\mathcal{C} \subseteq \mathcal{S}$ of at most k subsets such that $\bigcup_{S_i \in \mathcal{C}} S_i = U$? In other words, we seek a cover of U using at most k subsets from $\mathcal{S}$.*

It is easy to see that the vertex cover is a special case of set cover. Formally, given a Vertex Cover instance $(G = (V, E), k)$, build a Set Cover instance $(U, \mathcal{S}, k')$ by taking $U = E$ (one universe element per edge), $\mathcal{S} = \{S_v : v \in V\}$ where $S_v = \{e \in E : v \in e\}$ is the set of edges incident to v, and $k' = k$. The construction runs in $O(|V| + |E|)$ time. Choosing $\mathcal{C} \subseteq \mathcal{S}$ of size at most k that covers U corresponds exactly to choosing $\leq k$ vertices that cover every edge. Hence Vertex Cover $\leq_P$ Set Cover.

So far, our problems were graph-theoretic. Let us now investigate problems that relate to satisfiability of boolean expression. A Boolean expression is said to be in Conjunctive Normal

Form (CNF) if it is a conjunction of *clauses* where each clause is a disjunction of *literals*. A literal is any boolean variable or its negation. We call the problem of checking satisfiability of a boolean expression in CNF the problem SAT. We define 3-SAT as the special case of SAT where each clause has exactly three literals.

Having seen several reductions between problems, we now formalize what it means for a problem to be "as hard as any problem in NP".

Definition 13.9 (Class NP) *A decision problem $L \subseteq \{0,1\}^*$ is in the class NP if there exists a polynomial-time deterministic verifier V and a polynomial p such that for every $x \in \{0,1\}^*$:*

$$x \in L \iff \exists\, c \in \{0,1\}^{\leq p(|x|)} \text{ with } V(x,c) = 1.$$

The string c is called a certificate *(or* witness*) for x.*

Informally, the class NP includes problems where a *yes* certificate is verifiable in polynomial time. A *yes* certificate is an input, of size polynomial in $|x|$, that certifies the given instance is a *yes* instance.

Example 13.10 SAT is in NP: given a CNF formula ϕ over n variables, a satisfying assignment $\tau \in \{0,1\}^n$ serves as a certificate of size n. Substituting τ into ϕ and evaluating each clause takes polynomial time, so the verifier accepts iff $\phi(\tau) = 1$.

As another example, consider the Subset Sum problem: given a set S of integers and a target t, is there a subset of S summing to t? A *yes* certificate is simply a candidate subset $S' \subseteq S$; verifying that $\sum_{a \in S'} a = t$ takes linear time.

Note that we require certificates only for *yes* instances. We claim

$$\mathrm{P} \subseteq \mathrm{NP}.$$

Given any problem $\pi \in \mathrm{P}$, there is a polynomial-time algorithm for π, so certification is trivial: the verifier simply ignores the certificate and runs the algorithm. Whether the containment is strict — whether there is a problem in NP but not in P — is the most famous open problem in theoretical computer science.

Definition 13.11 (NP-hard) *A decision problem L is* NP-hard *if for every $L' \in \mathrm{NP}$, $L' \leq_P L$.*

Definition 13.12 (NP-complete) *A decision problem L is* NP-complete *if $L \in \mathrm{NP}$ and L is NP-hard.*

The first problem to be shown NP-complete was SAT, by Cook in 1971 and independently by Levin in 1973.

Theorem 13.13 (Cook-Levin Theorem) *SAT is NP-complete.*

Proof: *Sketch.* SAT is in NP: a satisfying assignment is the certificate. For NP-hardness, fix any $L \in \text{NP}$ decided by a nondeterministic polynomial-time Turing machine M running for at most $p(n)$ steps on inputs of length n. Introduce Boolean variables that describe an accepting computation *tableau* of M on input x: one variable per (tape cell, time step, symbol), per (time step, state), and per (time step, head position). Clauses encode that (i) the initial row records x and the start state, (ii) every later row follows from the previous one by one of M's transitions, and (iii) some row enters an accepting state. The resulting CNF has $O(p(n)^2)$ variables and clauses, is constructible in polynomial time, and is satisfiable iff M accepts x. Hence $L \leq_P \text{SAT}$. A fully detailed tableau encoding is given in Sipser [Sip13] and Arora–Barak [AB09].

■

Corollary 13.14 *If any NP-complete problem is in* P*, then* $\text{P} = \text{NP}$.

Proof: Let L be an NP-complete problem with $L \in \text{P}$. For any $L' \in \text{NP}$, we have $L' \leq_P L$ by definition of NP-hardness; composing the polynomial-time reduction with the polynomial-time algorithm for L yields a polynomial-time algorithm for L', so $L' \in \text{P}$. Hence $\text{NP} \subseteq \text{P}$. Combined with the trivial inclusion $\text{P} \subseteq \text{NP}$, we get $\text{P} = \text{NP}$.

■

Figure 13.2 illustrates the relationship between P, NP-complete, and NP. P is a subset of NP, as polynomial-time solvable problems are verifiable in polynomial time. NP-complete problems are the hardest in NP, with every NP problem reducing to them in polynomial time. If $\text{P} \neq \text{NP}$, NP-complete problems are disjoint from P.

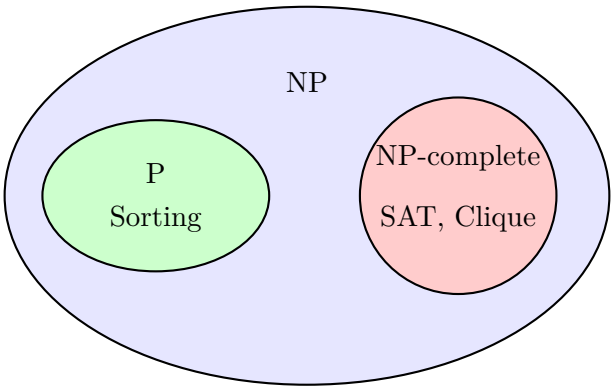


Figure 13.2: Relationship between P, NP-complete, and NP, assuming $\text{P} \neq \text{NP}$.

We now investigate the complexity of 3-SAT compared to SAT.

It is clear that SAT is at least as hard as 3-SAT because 3-SAT is a special case of SAT. Somewhat surprisingly, 3-SAT is as hard as SAT.

Theorem 13.15 *3-SAT is NP-complete.*

Proof: 3-SAT is in NP: a satisfying truth assignment is a certificate, and evaluating a 3-CNF formula under a given assignment takes polynomial time.

To establish NP-hardness we give a polynomial-time reduction SAT $\leq_P$ 3-SAT; NP-hardness of 3-SAT then follows from NP-hardness of SAT (Cook–Levin, Theorem 13.13).

Given a CNF formula $\phi = C_1 \wedge \cdots \wedge C_m$ with $C_i = l_{i1} \vee \cdots \vee l_{ik_i}$, we construct a 3-CNF formula ψ clause-by-clause.

Construction. For each clause C_i:

- **Case 1:** $k_i = 1$ (clause (l_1)): introduce fresh variables y_i, z_i and replace C_i by
$$(l_1 \vee y_i \vee z_i) \wedge (l_1 \vee y_i \vee \neg z_i) \wedge (l_1 \vee \neg y_i \vee z_i) \wedge (l_1 \vee \neg y_i \vee \neg z_i).$$
- **Case 2:** $k_i = 2$ (clause $(l_1 \vee l_2)$): introduce y_i and replace by
$$(l_1 \vee l_2 \vee y_i) \wedge (l_1 \vee l_2 \vee \neg y_i).$$
- **Case 3:** $k_i = 3$: keep $(l_1 \vee l_2 \vee l_3)$ unchanged.
- **Case 4:** $k_i > 3$ (clause $(l_1 \vee \cdots \vee l_k)$): introduce $y_{i,1}, \ldots, y_{i,k-3}$ and replace by
$$(l_1 \vee l_2 \vee y_{i,1}) \wedge (\neg y_{i,1} \vee l_3 \vee y_{i,2}) \wedge \cdots \wedge (\neg y_{i,k-3} \vee l_{k-1} \vee l_k).$$

Each clause of length k_i yields $O(k_i)$ clauses of length exactly 3, and no more than $O(k_i)$ fresh variables, so the overall construction is polynomial in $|\phi|$.

Correctness. Each replacement preserves satisfiability clause-by-clause. In Case 1, if l_1 is true then every one of the four clauses is already satisfied by l_1. Conversely, if l_1 is false, the four clauses simplify to
$$(y_i \vee z_i) \wedge (y_i \vee \neg z_i) \wedge (\neg y_i \vee z_i) \wedge (\neg y_i \vee \neg z_i),$$
which is unsatisfiable: a quick truth-table check shows that every assignment of (y_i, z_i) falsifies at least one of the four clauses. Hence the conjunction is logically equivalent to l_1. Case 2 is similar for $(l_1 \vee l_2)$. Case 4 is verified by induction: if some l_j is true in C_i, set $y_{i,1} = \cdots = y_{i,j-2} = T$ and $y_{i,j-1} = \cdots = y_{i,k-3} = F$ to satisfy every new clause; conversely, if every new clause is satisfied but all l_j were false, the chain of implications $y_{i,1} \Rightarrow y_{i,2} \Rightarrow \cdots \Rightarrow y_{i,k-3}$ together with the last clause $\neg y_{i,k-3} \vee l_{k-1} \vee l_k$ forces a contradiction. Hence ϕ is satisfiable iff ψ is, and since ψ is a 3-CNF, SAT $\leq_P$ 3-SAT.

■

Example 13.16 [SAT to 3-SAT] Consider the SAT instance

$$\phi \;=\; (x_1) \;\wedge\; (x_2 \vee x_3) \;\wedge\; (x_1 \vee \neg x_2 \vee x_3 \vee x_4),$$

which has one 1-clause, one 2-clause, and one 4-clause. Applying the reduction:

- The 1-clause (x_1) uses dummy variables y_1, z_1 and becomes

$$(x_1 \vee y_1 \vee z_1) \wedge (x_1 \vee y_1 \vee \neg z_1) \wedge (x_1 \vee \neg y_1 \vee z_1) \wedge (x_1 \vee \neg y_1 \vee \neg z_1).$$

- The 2-clause $(x_2 \vee x_3)$ uses dummy y_2 and becomes

$$(x_2 \vee x_3 \vee y_2) \wedge (x_2 \vee x_3 \vee \neg y_2).$$

- The 4-clause $(x_1 \vee \neg x_2 \vee x_3 \vee x_4)$ uses dummy y_3 and becomes

$$(x_1 \vee \neg x_2 \vee y_3) \wedge (\neg y_3 \vee x_3 \vee x_4).$$

The resulting 3-CNF formula ψ has $4 + 2 + 2 = 8$ clauses, each of size exactly 3. The assignment $x_1 =$ true, $x_2 =$ true, $x_3 =$ true, $x_4 =$ false satisfies ϕ; extending it with $y_3 =$ false (any choice of y_1, z_1, y_2) satisfies ψ.

Theorem 13.17 *Independent Set is NP-complete.*

Proof: We follow the four-step template announced earlier in this section.

Step 1 (Membership in NP). A candidate subset $S \subseteq V$ is a certificate, and checking that no edge of G has both endpoints in S takes $O(|E|)$ time.

Step 2 (Source). We reduce from 3-SAT, known to be NP-complete by Theorem 13.15.

Step 3 (Reduction). Given a 3-SAT formula ϕ with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$, where each clause $C_j = (l_{j1} \vee l_{j2} \vee l_{j3})$, we construct an instance $(G = (V, E), k)$:

- **Vertices.** For each clause C_j create three vertices v_{j1}, v_{j2}, v_{j3}, one per literal. Thus $V = \{v_{jk} \mid 1 \le j \le m,\ 1 \le k \le 3\}$, and $|V| = 3m$.
- **Intra-clause edges.** For each clause C_j, form a triangle on $\{v_{j1}, v_{j2}, v_{j3}\}$. This ensures at most one vertex per clause is picked by any independent set.
- **Inter-clause edges.** For any two vertices v_{jk} and $v_{j'k'}$ with $j \neq j'$, add the edge $\{v_{jk}, v_{j'k'}\}$ iff the literals l_{jk} and $l_{j'k'}$ are *inconsistent* (one is x_i and the other $\neg x_i$).
- **Parameter.** $k = m$.

The construction runs in $O(m^2)$ time.

Step 4 (Correctness). We claim that ϕ is satisfiable iff G has an independent set of size at least m.

($\Rightarrow$) Suppose ϕ has a satisfying assignment τ. In each clause C_j choose one literal l_{jk} that is true under τ, and let $S = \{v_{jk} : \text{the picked vertex of } C_j\}$. Then $|S| = m$. No intra-clause edge lies inside S (one vertex per clause), and no inter-clause edge lies inside S either: if $v_{jk}, v_{j'k'} \in S$ were joined by an inconsistency edge then l_{jk} and $l_{j'k'}$ could not both be true under τ. Hence S is independent of size m.

($\Leftarrow$) Suppose G has an independent set S with $|S| \geq m$. Each clause gadget is a triangle, so at most one vertex per clause is in S; with m clauses and $|S| \geq m$, S has exactly one vertex per clause. Set $x_i = \text{true}$ if some $v_{jk} \in S$ corresponds to $l_{jk} = x_i$, and $x_i = \text{false}$ if some $v_{jk} \in S$ corresponds to $l_{jk} = \neg x_i$. This assignment is consistent: a conflict would require both x_i and $\neg x_i$ to appear in S, but those two vertices are joined by an inconsistency edge, contradicting independence. The assignment makes some literal true in every clause, so ϕ is satisfied.

Hence $\phi \in$ 3-SAT $\Leftrightarrow (G, m) \in$ Independent Set, completing the reduction. Subsequent NP-completeness proofs in this chapter follow the same four-step template; we abbreviate the step labels for brevity.

■

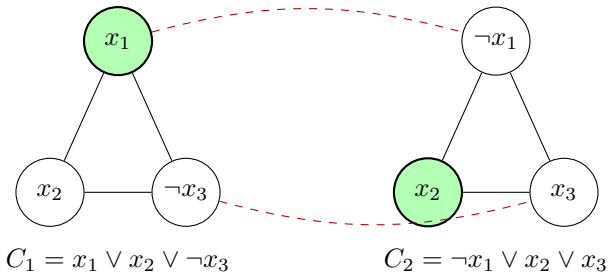


Figure 13.3: Reduction of the 3-SAT instance $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$ to independent set. Each clause becomes a triangle (clause gadget). Dashed red edges connect inconsistent literals across clauses: $x_1 \leftrightarrow \neg x_1$ and $\neg x_3 \leftrightarrow x_3$. The shaded vertices $\{x_1$ from C_1, x_2 from $C_2\}$ form an independent set of size $2 = m$, corresponding to the satisfying assignment $x_1 = \text{true}, x_2 = \text{true}, x_3 = \text{anything}$.

Example 13.18 [3-SAT to Independent Set] Consider the 3-SAT instance $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$ with $m = 2$ clauses. The reduction produces the graph in Figure 13.3 with $3m = 6$ vertices and $k = m = 2$. Each clause triangle enforces that at most one literal per clause is picked; the dashed inconsistency edges forbid picking x_1 from C_1 together with $\neg x_1$ from C_2, or $\neg x_3$ from C_1 together with x_3 from C_2. Picking x_1 from C_1 and x_2 from C_2 (shaded) yields an independent set of size 2, and reading off the corresponding literals gives the satisfying assignment $x_1 = \text{true}$, $x_2 = \text{true}$ (x_3 is unconstrained).

With Independent Set now established as NP-complete, the duality between Independent

Set and Vertex Cover (Theorem 13.6) immediately gives the NP-completeness of Vertex Cover as well.

Corollary 13.19 *Vertex Cover is NP-complete.*

Proof: Vertex Cover is in NP: a candidate cover $C \subseteq V$ with $|C| \leq k$ is a certificate, verified in $O(|E|)$ time by checking that every edge is incident to some vertex in C. For NP-hardness, reduce Independent Set to Vertex Cover via Theorem 13.6: given an Independent Set instance (G, k) with $|V| = n$, output the Vertex Cover instance $(G, n - k)$. By the theorem, G has an independent set of size $\geq k$ iff G has a vertex cover of size $\leq n - k$. Combined with Theorem 13.17 (Independent Set is NP-complete), Vertex Cover is NP-hard.

■

13.4 More NP-Complete Problems

In this section we continue building the catalog of NP-complete problems by proving NP-completeness of Clique and Subset Sum. For each problem we start from a problem already known to be NP-complete and give a polynomial-time reduction to the new problem.

Definition 13.20 (Clique) *The* Clique *problem asks: Given a graph $G = (V, E)$ and an integer k, does there exist a subset $Q \subseteq V$ of at least k vertices such that every pair of distinct vertices in Q is adjacent (i.e., $\forall u, v \in Q$ with $u \neq v$, $\{u, v\} \in E$)? Such a set Q is called a* clique.

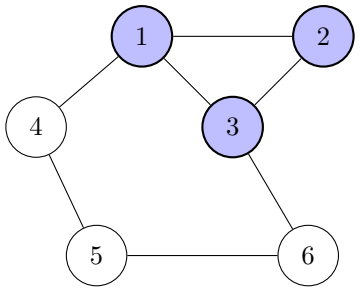


Figure 13.4: The same graph as in Figure 13.1, with the clique $Q = \{1, 2, 3\}$ shaded blue.

Example 13.21 In Figure 13.4, the set $Q = \{1, 2, 3\}$ is a clique of size 3: all three pairs $\{1, 2\}, \{1, 3\}, \{2, 3\}$ are edges. No clique of size 4 exists: any vertex outside the triangle is 4, 5, or 6, each of which has at most two neighbors, so no four vertices can be pairwise adjacent. Note that this clique is distinct from the vertex cover $C = \{1, 3, 5\}$ identified in Example 13.7; the two sets share only the vertices 1 and 3.

We now prove that Clique is NP-complete, using a polynomial-time reduction 3-SAT $\leq_P$ Clique.

Theorem 13.22 *Clique is NP-complete.*

Proof: Clique is in NP: a candidate clique $Q \subseteq V$ with $|Q| \geq k$ is a certificate, and verifying that every pair of vertices in Q is joined by an edge takes $O(|Q|^2)$ time.

To establish NP-hardness we reduce 3-SAT to Clique. Given a 3-SAT formula ϕ with n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$ — where each clause $C_j = (l_{j1} \vee l_{j2} \vee l_{j3})$ and l_{jk} is a literal — we construct an instance $(G = (V, E), k)$ of Clique:

1. **Vertices**: For each literal l_{jk} in clause C_j, create a vertex v_{jk}. Thus $V = \{v_{jk} \mid j = 1, \ldots, m,\ k = 1, 2, 3\}$, and $|V| = 3m$.

2. **Edges**: Add an edge $\{v_{jk}, v_{j'k'}\}$ (for $j \neq j'$) iff the literals l_{jk} and $l_{j'k'}$ are *consistent*, i.e., they are not of the form x_i and $\neg x_i$. No edges are placed between vertices of the same clause.

3. **Parameter**: $k = m$.

The construction runs in $O(m^2)$ time. We claim that ϕ is satisfiable iff G has a clique of size at least m.

($\Rightarrow$) Suppose ϕ has a satisfying assignment τ. In each clause C_j pick one literal l_{jk} that is true under τ, and let $Q = \{v_{jk} : \text{the picked vertex of } C_j\}$. Then $|Q| = m$, vertices of Q come from distinct clauses, and every pair of picked literals is consistent under τ (both evaluate to true), so every pair of vertices in Q is connected by an edge. Hence Q is a clique of size m.

($\Leftarrow$) Suppose G has a clique Q with $|Q| \geq m$. Since vertices of the same clause are never adjacent, Q contains at most one vertex per clause; with $|Q| \geq m$ we have $|Q| = m$ and exactly one vertex per clause. Read off the literals: if $v_{jk} \in Q$ corresponds to $l_{jk} = x_i$, set $x_i = \text{true}$; if $l_{jk} = \neg x_i$, set $x_i = \text{false}$. No conflict arises: a conflict would require two vertices $v_{jk}, v_{j'k'} \in Q$ labeled by x_i and $\neg x_i$, but then they would be non-adjacent in G, contradicting that Q is a clique. The resulting assignment makes some literal true in each clause, so ϕ is satisfied.

Hence $\phi \in$ 3-SAT $\Leftrightarrow (G, m) \in$ Clique, and Clique is NP-hard. Combined with Clique $\in$ NP, the claim follows.

■

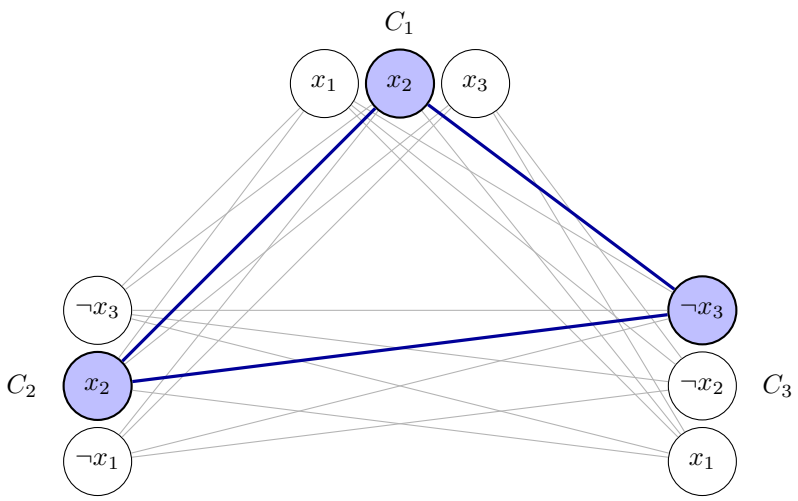


Figure 13.5: Reduction of the 3-SAT instance $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$ to clique. No edges exist within the same clause gadget; an edge joins two vertices from different clauses iff their literals are non-contradictory (gray). The shaded vertices $\{x_2$ in $C_1,\ x_2$ in $C_2,\ \neg x_3$ in $C_3\}$ together with the bold blue edges form a clique of size $m = 3$, witnessing that ϕ is satisfiable.

Example 13.23 [3-SAT to Clique] Consider the 3-SAT instance

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3),$$

with $m = 3$ clauses. The reduction produces the graph in Figure 13.5 on $3m = 9$ vertices, with clique threshold $k = m = 3$. Note the complement structure compared with the Independent Set reduction (Figure 13.3): here a clause gadget has *no* internal edges, while inter-clause edges join non-contradictory literals. The shaded vertices $\{x_2$ in $C_1,\ x_2$ in $C_2,\ \neg x_3$ in $C_3\}$ form a clique of size $k = 3$ (bold blue edges). Reading off the corresponding literals gives the satisfying assignment $x_2 = \text{true},\ x_3 = \text{false}$ (with x_1 unconstrained).

Definition 13.24 (Subset Sum) *The Subset Sum problem asks whether, given a set of positive integers $S = \{a_1, \ldots, a_n\}$ and a target t, there exists a subset $S' \subseteq S$ summing to t.*

Definition 13.25 (3D-Matching) *The 3D-Matching (3DM) problem is defined as follows. Given three disjoint sets $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_n\}$, $Z = \{z_1, \ldots, z_n\}$ and a set of triples $T \subseteq X \times Y \times Z$, does there exist a subset $M \subseteq T$ of size n such that every element of $X \cup Y \cup Z$ appears in exactly one triple of M?*

3D-Matching is one of Karp's classical NP-complete problems [Kar72]; the reduction 3-SAT $\leq_P$ 3DM uses "variable gadgets" (one for each x_i, with $2 \cdot \text{occ}(x_i)$ matched triples acting as a

truth flag) and "clause gadgets" (one triple per literal, sharing core elements that force exactly one literal per clause to be selected), plus padding triples to balance the universe sizes. We take 3DM's NP-completeness for granted in what follows and use it to prove NP-completeness of Subset Sum. (An alternative direct route, Vertex Cover $\leq_P$ Subset Sum, is left as an exercise.)

Theorem 13.26 *Subset Sum is NP-complete.*

Proof: Subset Sum is in NP: a subset $S' \subseteq S$ is a certificate, and checking $\sum_{a \in S'} a = t$ takes time polynomial in the size of the input.

To establish NP-hardness we give a polynomial-time reduction 3D-Matching $\leq_P$ Subset Sum. Given a 3DM instance with sets $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_n\}$, $Z = \{z_1, \ldots, z_n\}$ and triples $T = \{t_1, \ldots, t_m\} \subseteq X \times Y \times Z$, we construct a Subset Sum instance (S, t) with $3n$ digits in base $b = m + 1$ to ensure carry-free addition:

- **Elements**: the $3n$ elements $x_1, \ldots, x_n,\ y_1, \ldots, y_n,\ z_1, \ldots, z_n$ are assigned to digit positions $3n-1, 3n-2, \ldots, 0$ respectively.
- **Base**: use $b = m + 1$. Since each element appears in at most m triples, the sum in each digit position is at most $m < b$, so additions produce no carries.
- **Numbers**: for each triple $t_j = (x_{j_1}, y_{j_2}, z_{j_3})$, create

$$a_j \ = \ b^{3n-j_1} + b^{2n-j_2} + b^{n-j_3},$$

  so a_j is 1 in the three digit positions of its three elements and 0 elsewhere.
- **Target**:

$$t \ = \ \sum_{k=1}^{n} (b^{3n-k} + b^{2n-k} + b^{n-k}),$$

  which has a 1 in each of the $3n$ digit positions.
- **Set** S: $S = \{a_1, \ldots, a_m\}$.

The construction is clearly polynomial time: each a_j and t have $O(n \log m)$ bits.

($\Rightarrow$) If T contains a perfect matching $M \subseteq T$ of n triples covering each element exactly once, then $S' = \{a_j : t_j \in M\}$ sums to t: each digit position receives exactly one 1.

($\Leftarrow$) Conversely, if $S' \subseteq S$ sums to t, then because addition is carry-free in base b and t has a 1 in each of the $3n$ positions, each digit position of S' must sum to exactly 1. Each a_j contributes three 1s, so $|S'| = n$, and each element is covered exactly once: the corresponding triples form a perfect matching.

Hence $(X, Y, Z, T) \in$ 3DM $\Leftrightarrow$ $(S, t) \in$ Subset Sum, so Subset Sum is NP-hard. Combined with Subset Sum $\in$ NP, the claim follows.

■

Example 13.27 [3D-Matching to Subset Sum] Consider the 3DM instance with $n = 2$, so $X = \{x_1, x_2\}$, $Y = \{y_1, y_2\}$, $Z = \{z_1, z_2\}$, and $m = 3$ triples:

$$t_1 = (x_1, y_1, z_1), \qquad t_2 = (x_2, y_2, z_2), \qquad t_3 = (x_1, y_2, z_1).$$

The unique perfect matching is $M = \{t_1, t_2\}$: t_1 covers x_1, y_1, z_1 and t_2 covers x_2, y_2, z_2, so every element appears in exactly one chosen triple. (The pair $\{t_1, t_3\}$ fails because x_1 is covered twice; similarly for $\{t_2, t_3\}$ with y_2.)

The reduction uses base $b = m + 1 = 4$ and $3n = 6$ digit positions, one per element. Writing numbers in base 4 with six digit positions (most significant digit leftmost corresponds to x_1, least significant to z_2):

triple	a_j	base-4 digits $(x_1\, x_2\, y_1\, y_2\, z_1\, z_2)$
$t_1 = (x_1, y_1, z_1)$	$4^5 + 4^3 + 4^1 = 1092$	1 0 1 0 1 0
$t_2 = (x_2, y_2, z_2)$	$4^4 + 4^2 + 4^0 = 273$	0 1 0 1 0 1
$t_3 = (x_1, y_2, z_1)$	$4^5 + 4^2 + 4^1 = 1044$	1 0 0 1 1 0

The target $t = 4^5 + 4^4 + 4^3 + 4^2 + 4^1 + 4^0 = 1365$ has the base-4 digit string 1 1 1 1 1 1, one 1 in every column. The Subset Sum instance is $(S, t) = (\{1092, 273, 1044\},\ 1365)$.

Observe that $a_1 + a_2 = 1092 + 273 = 1365 = t$: the corresponding base-4 addition 1 0 1 0 1 0 + 0 1 0 1 0 1 = 1 1 1 1 1 1 is carry-free and each column sums to exactly 1, matching the fact that $\{t_1, t_2\}$ covers every element exactly once. No other subset of S sums to t: for example, $a_1 + a_3$ produces a 2 in the x_1 column (since t_1 and t_3 both use x_1), so the resulting base-4 string is not 111111.

Two further reductions in the chain Cook–Levin → SAT → 3-SAT → Hamiltonian Path → Hamiltonian Cycle → TSP advertised by Figure 13.8 are short enough to give in full here. We take Hamiltonian Path's NP-completeness as established (the 3-SAT → HP reduction with variable-and-clause gadgets is given in, e.g., Sipser [Sip13]).

Theorem 13.28 *Hamiltonian Cycle is NP-complete.*

Proof: HC is in NP: a permutation of the vertices is a certificate, verified in $O(n)$ time. For NP-hardness we reduce HP $\leq_P$ HC. Given an HP instance $G = (V, E)$, build $G' = (V \cup \{v^\star\}, E \cup \{\{v^\star, u\} : u \in V\})$: a fresh vertex $v^\star$ adjacent to every original vertex. The construction is $O(n)$. If G has a Hamiltonian path $u_1, u_2, \ldots, u_n$, then $u_1, u_2, \ldots, u_n, v^\star, u_1$ is a Hamiltonian cycle in G'. Conversely, removing $v^\star$ from any Hamiltonian cycle of G' leaves a Hamiltonian path of G between the two original neighbors of $v^\star$ in the cycle.

■

Theorem 13.29 *The decision version of the Traveling Salesman Problem is NP-complete.*

Proof: TSP is in NP: a permutation of the vertices certifies a tour, verified in $O(n)$ time. For NP-hardness we reduce HC $\leq_P$ TSP. Given an HC instance $G = (V, E)$ with $n = |V|$,

Problem	Definition	Reduction From
3-Colorability	Graph G 3-colorable?	3-SAT
3D-Matching	Perfect matching in $X \times Y \times Z$?	3-SAT
Knapsack	Items fit W, value $\geq V$?	Subset Sum
Hamiltonian Path	Path visits all vertices once?	3-SAT
Hamiltonian Cycle	Cycle visits all vertices once?	Hamiltonian Path
Traveling Salesman	Tour with weight $\leq B$?	Hamiltonian Cycle

Figure 13.6: NP-complete problems and reductions

build the complete graph $G' = (V, V \times V)$ with edge weights $w(u, v) = 1$ if $\{u, v\} \in E$ and $w(u, v) = 2$ otherwise; set the threshold $B = n$. The construction is $O(n^2)$. G' admits a TSP tour of weight $\leq n$ iff every edge of the tour has weight 1, which is equivalent to the tour using only edges of G, i.e., a Hamiltonian cycle of G.

■

We now list further problems which have been shown to be NP-complete. Table 13.6 summarizes six NP-complete decision problems: 3-Colorability, 3D-Matching, Knapsack, Hamiltonian Path, Hamiltonian Cycle, and Traveling Salesman. For each problem, we provide its definition and identify a known NP-complete problem that can be polynomially reduced to it. The Hamiltonian Cycle and TSP reductions are made explicit in Theorems 13.28 and 13.29 above; the others are sketched in the references.

1. **3-Colorability**:
   - *Definition*: Given a graph $G = (V, E)$, can the vertices be colored with three colors such that no adjacent vertices share the same color?
   - *Reduction From*: 3-SAT. Construct a graph with gadgets for variables (true/false colors) and clauses (ensuring at least one true literal), where a 3-coloring encodes a satisfying assignment.
2. **3D-Matching**:
   - *Definition*: Given sets X, Y, Z, each of size n, and triples $T \subseteq X \times Y \times Z$, is there a subset of n triples covering each element exactly once?
   - *Reduction From*: 3-SAT. Create variable gadgets (true/false triples) and clause gadgets (triples for true literals), where a matching corresponds to a satisfying assignment.
3. **Knapsack**:
   - *Definition*: Given items with weights w_i, values v_i, capacity W, and target V, is there a subset with weight $\leq W$, value $\geq V$?
   - *Reduction From*: Subset Sum. Set $v_i = w_i$, $W = t$, $V = t$, making Knapsack equivalent to Subset Sum.

4. **Hamiltonian Path**:
    - *Definition*: Given a graph $G = (V, E)$, is there a path visiting each vertex exactly once?
    - *Reduction From*: 3-SAT. Build a graph with variable paths (true/false) and clause nodes, where a Hamiltonian path encodes a satisfying assignment.
5. **Hamiltonian Cycle**:
    - *Definition*: Given a graph $G = (V, E)$, is there a cycle visiting each vertex exactly once?
    - *Reduction From*: Hamiltonian Path. Add a vertex connected to all others, where a Hamiltonian cycle implies a Hamiltonian path in the original graph.
6. **Traveling Salesman**:
    - *Definition*: Given a complete graph with edge weights and bound B, is there a tour of weight $\leq B$?
    - *Reduction From*: Hamiltonian Cycle. Set weights: 1 for original edges, 2 otherwise, $B = n$. A tour of weight n is a Hamiltonian cycle.

13.5 co-NP Class of Problems

In computational complexity theory, the class co-NP plays a crucial role alongside NP in understanding the structure of decision problems.

Definition 13.30 (co-NP) *The complexity class co-NP consists of all decision problems whose* complements *are in* NP*. That is,* $L \in$ *co-*NP *iff* $\overline{L} \in$ NP*.*

Equivalently, a problem L is in co-NP if there exists a polynomial-time verifiable certificate for *no* instances (i.e., instances $x \notin L$). This means there is a polynomial-time Turing machine M and a polynomial p such that for every input $x \notin L$, there exists a certificate c of length at most $p(|x|)$ where $M(x, c) = 1$, and for every $x \in L$, no such certificate exists.

In contrast, a problem L is in NP if there exists a polynomial-time verifiable certificate for *yes* instances ($x \in L$). Thus, co-NP is the class of problems where the evidence for rejection is efficiently verifiable, while NP focuses on evidence for acceptance.

The relationship between co-NP and NP is symmetric due to their complementary definitions:

- If $L \in$ NP, then $\overline{L} \in$ co-NP.
- If $L \in$ co-NP, then $\overline{L} \in$ NP.

This symmetry implies that NP and co-NP are *dual* classes. Notably:

- P $\subseteq$ NP $\cap$ co-NP, since if a problem is in P, both its yes and no instances can be decided in polynomial time, providing trivial certificates.

- NP ∩ co-NP contains problems like the decision version of Integer Factorization (given n and a bound k, does n have a non-trivial factor $\leq k$?), where both a *yes* answer (exhibit such a factor) and a *no* answer (exhibit the full prime factorization together with a primality certificate for each prime factor) are verifiable in polynomial time. This problem is not known to be in P.

A central question is whether NP = co-NP. If NP = co-NP, then for every NP problem, its complement would also have efficiently verifiable yes certificates, implying a symmetry in verification. However, it is widely believed that NP ≠ co-NP, as their equality would collapse the polynomial hierarchy and have profound implications for problems like SAT and UNSAT.

The following figure illustrates the relationship between NP, co-NP, and P. P is explicitly drawn as a distinct region within the intersection of NP and co-NP, representing problems solvable in polynomial time. NP and co-NP are dual classes, where a problem is in co-NP if its complement is in NP. Their intersection, NP ∩ co-NP, contains P and problems like Integer Factorization, not known to be in P.

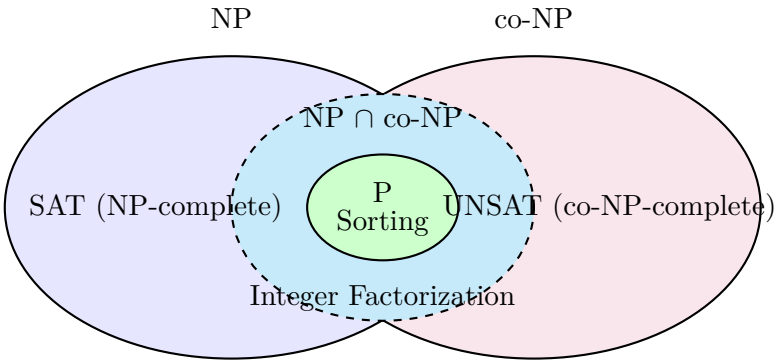


Figure 13.7: Relationship between NP, co-NP, and P. P is drawn within NP ∩ co-NP, which also includes problems like integer factorization. SAT is the canonical NP-complete problem; its complement UNSAT is co-NP-complete. It is unknown whether NP = co-NP.

We present several examples of problems in co-NP, focusing on their complements in NP and their verification mechanisms.

Definition 13.31 (Tautology) *The* Tautology problem *asks: Given a Boolean formula ϕ in disjunctive normal form (DNF), is ϕ true for all possible truth assignments (i.e., is ϕ a tautology)?*

Definition 13.32 (UNSAT) *The* Unsatisfiability (UNSAT) problem *asks: Given a Boolean formula ϕ in conjunctive normal form (CNF), is ϕ unsatisfiable (i.e., no truth assignment makes ϕ true)?*

Definition 13.33 (Graph Non-Isomorphism) *The* Non-Isomorphism problem *for graphs asks: Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, are G_1 and G_2 not isomorphic (i.e., there exists no bijection $f : V_1 \to V_2$ preserving edges)?*

Theorem 13.34 *Tautology, UNSAT, and Graph Non-Isomorphism are all in co-*NP*.*

Proof: For each problem, the complement is in NP:

- The complement of Tautology is: Is ϕ not a tautology? A no instance (non-tautology) has a certificate: a truth assignment making ϕ false, verifiable in polynomial time by evaluating ϕ. Thus, Tautology is in co-NP.

- The complement of UNSAT is SAT, which is in NP: a yes instance of SAT has a certificate (a satisfying assignment) verifiable in polynomial time. Thus, UNSAT is in co-NP.

- The complement of Non-Isomorphism is Graph Isomorphism, which is in NP: a yes instance has a certificate (a bijection f) verifiable in polynomial time by checking edge preservation. Thus, Non-Isomorphism is in co-NP.

■

A decision problem L is *co-NP-complete* if $L \in$ co-NP and every $L' \in$ co-NP reduces to L in polynomial time. UNSAT and Tautology are both co-NP-complete: their complements (SAT and DNF non-tautology) are NP-complete, and a polynomial-time reduction from any NP-complete problem L' to SAT extends, after complementation, to a polynomial-time reduction from $\overline{L'}$ to UNSAT. It is not known if Non-Isomorphism is co-NP-complete.

Several open problems in complexity theory involve co-NP, reflecting its central role in understanding computational limits. We highlight key questions:

- Does NP = co-NP? The most significant open problem is whether NP = co-NP. If NP = co-NP, then for every NP-complete problem like SAT, its complement (UNSAT) would also be in NP, implying efficient certificates for unsatisfiability. This would collapse the polynomial hierarchy, as the hierarchy relies on the distinction between NP and co-NP. Most researchers believe NP $\neq$ co-NP due to the intuitive asymmetry: verifying satisfiability (NP) seems easier than verifying unsatisfiability (co-NP).

- Which problems lie in NP$\cap$co-NP without being known to lie in P? The decision version of Integer Factorization is the classic example; Primality was once a candidate but is now known to be in P via the AKS algorithm. Identifying further natural candidates (e.g., Graph Isomorphism) remains open. It is conjectured that NP $\cap$ co-NP $\neq$ NP and NP $\cap$ co-NP $\neq$ co-NP, but the boundaries are unclear.

13.6 Coping with NP-hardness

A proof that a problem is NP-complete is rarely the end of the engineering conversation; it is the start of one. Practitioners use a five-strategy menu, all of which are explored elsewhere in the literature and several of which we develop in the rest of this book:

- **Approximation algorithms.** Trade optimality for polynomial running time. Chapter 14 develops this strategy in detail.

- **Parameterized / fixed-parameter tractable algorithms.** Identify a problem-specific parameter k (cover size, treewidth, solution depth) and design algorithms whose exponential cost is confined to k rather than the input size. The canonical example is Vertex Cover in $O(2^k \cdot (n+m))$ time, which is practical whenever the optimal cover is small.

- **Industrial SAT and ILP solvers.** Modern conflict-driven SAT solvers and branch-and-cut ILP solvers routinely dispatch instances with millions of variables, even though the worst-case problems are NP-hard. Encoding a target NP-complete problem as SAT or ILP and handing the result to such a solver is often the most cost-effective practical answer.

- **Special-case polynomial algorithms.** Many NP-complete problems become polynomial when the input is restricted: 2-SAT is in P, planar graph 3-coloring admits a $2^{O(\sqrt{n})}$ algorithm, and Vertex Cover on bipartite graphs is in P via König's theorem (Chapter 12). Recognizing the special case can save an exponential factor.

- **Heuristics and randomization.** When all else fails, local-search heuristics (simulated annealing, genetic algorithms, tabu search) and randomized rounding give solutions with no worst-case guarantees but often excellent empirical behavior.

Chapter 14 treats the first strategy in depth; the others are pursued in the references at the end of the chapter.

13.7 Summary

This chapter introduced NP-completeness and the theory of computational intractability, covering the complexity classes P, NP, and co-NP, and polynomial-time reductions. Approximation algorithms, the most prominent constructive response to NP-hardness, are developed in Chapter 14.

Figure 13.8 summarizes the chain of reductions established in this chapter and the catalog. Each arrow $A \to B$ denotes a polynomial-time reduction $A \leq_P B$ used to transfer NP-hardness from A to B.

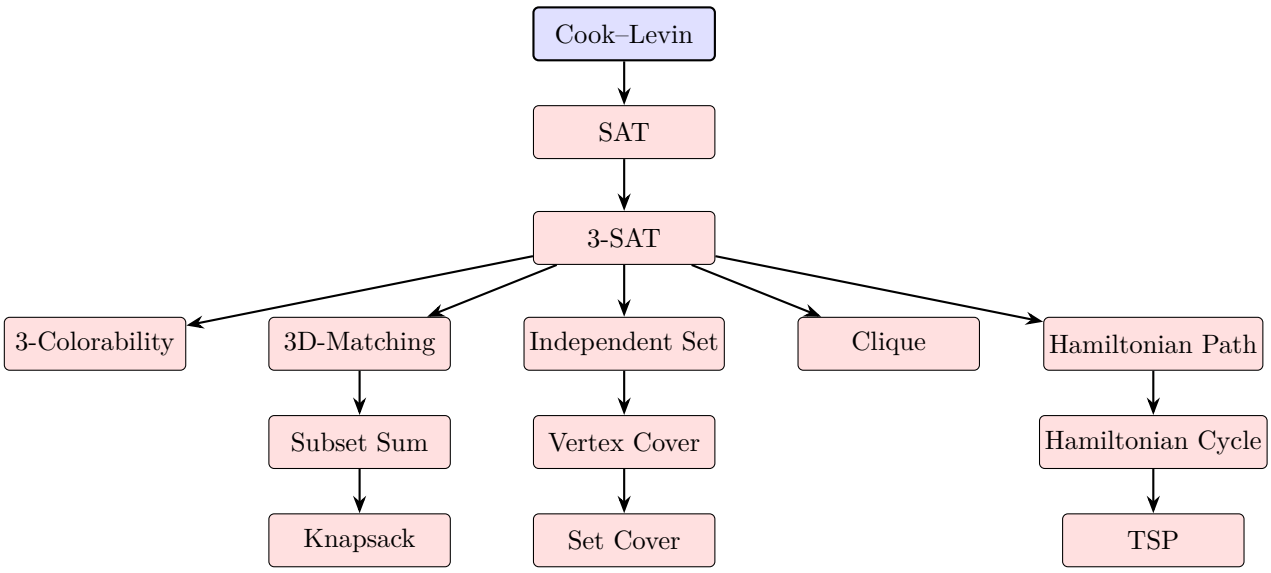


Figure 13.8: Reductions established or cited in this chapter. An arrow $A \to B$ means $A \leq_P B$, transferring NP-hardness from A to B. Cook–Levin (blue) seeds the chain by giving SAT's NP-hardness from first principles; every other problem inherits NP-hardness through one of these reductions.

All of the following problems are NP-complete; the table records the key technique used to establish each result.

Problem/Topic	**Key Technique**
SAT	Cook–Levin Theorem
3-SAT	Reduction from SAT
Clique	Reduction from 3-SAT
Independent Set	Reduction from 3-SAT
Vertex Cover	Reduction from Independent Set (duality)
Subset Sum	Reduction from 3D-Matching

13.8 Problems

1. **(Exact Length Path NPC)** You are given a graph $G = (V, E)$ with non-negative lengths $w_e \geq 0$ for all edges $e \in E$. You are also given a source node s and destination node t and an integer K. Show that deciding whether there is an s-t path of length exactly K is NP-complete.

2. **(Job Scheduling Deadlines NPC)** Consider the *Job Scheduling with Deadlines* problem: Given n jobs with processing times p_i, deadlines d_i, and profits r_i, and a single processor, select a subset of jobs to maximize total profit such that each selected job completes by its deadline (job i takes p_i time units, must finish by d_i, and jobs are non-preemptive). Prove this problem is NP-complete.

3. **(Partition NPC)** (*Partition.*) Given a multiset of positive integers $A = \{a_1, \ldots, a_n\}$, decide whether A can be partitioned into two subsets A_1, A_2 with $\sum_{a \in A_1} a = \sum_{a \in A_2} a$. Show that Partition is NP-complete.
   Hint: Partition is in NP (a candidate A_1 is a polynomial-time-verifiable certificate). For NP-hardness, reduce from Subset Sum: given (S, t) with $\sigma = \sum_{a \in S} a$, add one new element $|2t - \sigma|$ (or two padding elements) so that the augmented multiset has a balanced partition iff the original Subset Sum has a solution.

4. **(Knapsack Decision NPC)** (*Knapsack* (decision version).) Given items with weights $w_1, \ldots, w_n$, values $v_1, \ldots, v_n$, capacity W, and threshold V, decide whether there is a subset $I \subseteq \{1, \ldots, n\}$ with $\sum_{i \in I} w_i \leq W$ and $\sum_{i \in I} v_i \geq V$. Show that Knapsack is NP-complete.
   Hint: Reduce from Subset Sum: given (S, t), set $w_i = v_i = a_i$, $W = V = t$. A subset realizes weight $\leq t$ *and* value $\geq t$ iff it sums to exactly t.

5. **(Hamiltonian Cycle NPC)** (*Hamiltonian Cycle.*) Given an undirected graph G, decide whether G has a Hamiltonian cycle (a cycle visiting every vertex exactly once). Show that Hamiltonian Cycle is NP-complete.
   Hint: In NP — the cycle itself is the certificate. For NP-hardness, reduce from Hamiltonian Path: attach a new vertex u adjacent to every vertex of G; G has a Hamiltonian path iff the new graph has a Hamiltonian cycle through u.

6. **(Traveling Salesman NPC)** (*Traveling Salesman* (decision).) Given a complete weighted graph G and a bound B, decide whether G has a tour visiting every vertex exactly once with total weight at most B. Show that TSP is NP-complete.
   Hint: Reduce from Hamiltonian Cycle. Given $G = (V, E)$, build a complete graph G' on V with edge weight $w(e) = 0$ if $e \in E$ and $w(e) = 1$ if $e \notin E$; set $B = 0$. A Hamiltonian cycle in G corresponds to a tour in G' of weight 0.

7. **(Three Colorability NPC)** (*3-Colorability.*) Given an undirected graph G, decide whether the vertices of G can be colored with three colors so that no edge has both endpoints of the same color. Show that 3-Colorability is NP-complete.
   Hint: Reduce from 3-SAT. Introduce three anchor vertices T, F, N forming a triangle (forcing them to take the three color classes). For each variable x_i, add two vertices $x_i, \neg x_i$ joined by an edge and each joined to N (forcing one to color T, the other to color F). For each clause, add a six-vertex *OR-gadget* that is 3-colorable iff at least one of its three input literals is colored T.

8. **(Dominating Set NPC)** (*Dominating Set.*) Given a graph $G = (V, E)$ and an integer k, decide whether there exists $D \subseteq V$ with $|D| \leq k$ such that every vertex not in D has at least one neighbor in D. Show that Dominating Set is NP-complete.
   Hint: Reduce from Vertex Cover, assuming WLOG that G has no isolated vertices. Given (G, k), construct G' from G by subdividing each edge $\{u, v\}$ with a new vertex w_{uv} and adding the triangle $\{u, v, w_{uv}\}$; then G has a vertex cover of size $\leq k$ iff G' has a dominating set of size $\leq k$.

9. **(Max Cut NPC)** (*Max-Cut.*) Given an undirected graph G and an integer k, decide whether V can be partitioned into two sets A, B so that the number of edges with one

endpoint in A and the other in B is at least k. Show that Max-Cut is NP-complete.
Hint: Max-Cut is in NP (a candidate bipartition is a certificate). For NP-hardness, reduce from Not-All-Equal 3-SAT (NAE-3-SAT), which is itself NP-complete by a reduction from 3-SAT. Each NAE clause produces a small graph gadget whose edges are cut precisely when the clause is "not-all-equal" satisfied.

10. **(Bin Packing NPC)** (*Bin Packing* (decision)). Given n items of sizes $s_1, \ldots, s_n \in (0, 1]$ and an integer k, decide whether the items can be packed into k unit-capacity bins. Show that Bin Packing is NP-complete (even for $k = 2$).
Hint: Reduce from Partition. Given a multiset A of positive integers with sum $2t$, scale each a_i to $a_i/t \in (0, 1]$ (so the scaled sizes total 2); the scaled items fit into $k = 2$ unit bins iff each bin is filled to exactly 1, i.e. iff A admits a balanced partition.

11. **(Vertex Cover Subset Sum)** (*Vertex Cover* $\leq_P$ *Subset Sum.*) Give a polynomial-time reduction from Vertex Cover to Subset Sum. *Hint*: encode each vertex as a digit position and each edge as a number whose digits flag its two endpoints; choose a base larger than the maximum vertex degree to keep additions carry-free. This closes the alternative route to Subset Sum's NP-hardness mentioned in the chapter.

12. **(Vertex Cover FPT Algorithm)** (*Vertex Cover is FPT.*) Give a $O(2^k \cdot (n+m))$-time algorithm for deciding whether a graph $G = (V, E)$ has a vertex cover of size $\leq k$. *Hint*: pick any uncovered edge $\{u, v\}$ and branch on which of u, v is in the cover, decreasing k by 1 in each branch.

13. **(2-SAT Polynomial Time)** (*2-SAT is in P.*) Show that 2-SAT can be decided in $O(n+m)$ time by building the implication graph (literals as vertices, each clause $(\ell_1 \vee \ell_2)$ contributing edges $\overline{\ell_1} \to \ell_2$ and $\overline{\ell_2} \to \ell_1$) and computing strongly connected components: the formula is satisfiable iff no variable lies in the same SCC as its negation.

14. **(Incremental SAT)** A 3-CNF formula ϕ on n variables has been determined to be satisfiable, together with a witness assignment τ. A new clause C_{m+1} is appended to form ϕ'. Show that deciding satisfiability of ϕ' is no easier than deciding SAT in general, by giving a reduction.

15. **(Dynamic Vertex Cover under Edge Insertions)** An NP-hard instance of decision Vertex Cover (G, k) is given; edges are inserted one by one and after each insertion the algorithm must decide whether a cover of size $\leq k$ still exists. Show that no polynomial-time fully dynamic algorithm exists unless P = NP, and discuss the parameterised view.

16. **(Constrained Independent Set)** Given a graph $G = (V, E)$, an integer k, and two disjoint subsets $V_{\text{in}}, V_{\text{out}} \subseteq V$ that must respectively be *included in* and *excluded from* the output independent set, decide whether a constrained independent set of size $\geq k$ exists. Reduce to plain Independent Set.

17. **(Approximate Vertex Cover via LP Rounding)** Round the LP relaxation of Vertex Cover to obtain a 2-approximation. State the LP, give the rounding rule, and prove the guarantee.

13.9 Bibliographic Remarks

The Cook-Levin Theorem (Theorem 13.13), independently established by Cook [Coo71] and Levin [Lev73], marks SAT as the first NP-complete problem, showing that every NP problem can be reduced to SAT in polynomial time. This result laid the groundwork for identifying other NP-complete problems, as detailed in the chapter's reductions (e.g., SAT $\leq_P$ 3-SAT, 3-SAT $\leq_P$ Independent Set, 3-SAT $\leq_P$ Clique). Karp [Kar72] significantly expanded this landscape by proving NP-completeness for 21 fundamental problems, including Clique, Vertex Cover, and Hamiltonian Path, many of which are covered in Figure 13.8. For a comprehensive treatment of NP-completeness and its reductions, Garey and Johnson [GJ79] remains a definitive reference.

Polynomial-time reductions, as illustrated in Sections 13.3 and 13.4, are central to establishing the hardness of problems like Independent Set, Vertex Cover, and 3-SAT. These reductions exploit structural relationships, such as the complementary nature of Independent Set and Vertex Cover, to demonstrate computational equivalence. The chapter's treatment of co-NP (Section 13.5) introduces the dual class of problems with efficiently verifiable no certificates, with examples like UNSAT and Tautology. Papadimitriou [Pap94] offers an authoritative resource on complexity classes, including NP, co-NP, and their intersections.

Chapter 14

Approximation Algorithms

14.1 Introduction

A proof that an optimization problem is NP-hard is rarely the end of the engineering conversation: it is the start of one. *Approximation algorithms* respond to NP-hardness by trading optimality for polynomial running time. Instead of insisting on the exact optimum, an approximation algorithm computes, in polynomial time, a feasible solution whose cost is provably within a guaranteed factor of the optimum.

This chapter develops the basic theory of approximation algorithms and works through several canonical examples lifted from Chapter 13. We assume the reader is familiar with the NP-completeness of Vertex Cover, Set Cover, and Knapsack; for these problems the goal of this chapter is constructive: we do not just prove they are hard, we present algorithms that return a solution with a guarantee.

The chapter is organized as follows. Section 14.2 defines the approximation ratio formally and distinguishes minimization from maximization variants. Section 14.3 gives a 2-approximation for minimum Vertex Cover via maximal matching, extends it to *weighted* Vertex Cover, and presents the LLP variants of both. Section 14.4 gives the greedy H_n-approximation for Set Cover with its parallel LLP variant, and a primal-dual extension to weighted Set Cover. Section 14.5 introduces approximation schemes (PTAS / FPTAS), sketches the FPTAS for Knapsack, and presents its parallel LLP form via lattice scaling. Section 14.6 shows how side constraints compose with LLP approximation algorithms. Section 14.7 surveys hardness-of-approximation results that bound the best achievable ratio for several problems.

14.2 The Approximation Ratio

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

Definition 14.1 (Approximation Ratio) *Let Π be a minimization optimization problem and A a polynomial-time algorithm that, on every instance I, outputs a feasible solution of cost $A(I)$. Let $OPT(I)$ denote the cost of an optimal solution to I. We say A is an* α-approximation algorithm *for Π (with $\alpha \geq 1$) if for every instance I,*

$$A(I) \leq \alpha \cdot OPT(I).$$

The constant α is called the approximation ratio *of A. For a maximization problem, the condition is $A(I) \geq \frac{1}{\alpha} \cdot OPT(I)$.*

The smaller α is, the stronger the guarantee. An exact polynomial-time algorithm is a 1-approximation; for an NP-hard problem we expect $\alpha > 1$. Some problems admit constant-factor approximations independent of the input size; others admit only $O(\log n)$-approximations; others admit no constant-factor approximation at all unless P = NP.

14.3 Vertex Cover

Recall (Chapter 13) that the minimum Vertex Cover problem asks, given an undirected graph $\mathcal{G} = (V, E)$, for the smallest subset $C \subseteq V$ such that every edge of $\mathcal{G}$ has at least one endpoint in C. The decision version is NP-complete, hence the optimization version is NP-hard. We give a clean 2-approximation.

Algorithm ApproxVertexCover: ApproxVertexCover($\mathcal{G}$)

input: graph $\mathcal{G} = (V, E)$
output: vertex cover C

```
C := ∅ ;
E' := E ;
while E' ≠ ∅ do
    Pick any edge (u, v) ∈ E' ;
    C := C ∪ {u, v} ;
    Remove all edges incident to u or v from E' ;
end
return C
```

Time: $O(m)$.

Theorem 14.2 *ApproxVertexCover is a 2-approximation algorithm for the minimum Vertex Cover problem.*

Proof: Let F be the set of edges picked by the algorithm in successive iterations. Because the algorithm removes every edge incident to the endpoints of a picked edge before the next iteration, no two edges of F share an endpoint; i.e., F is a matching. The algorithm outputs C consisting of both endpoints of each edge in F, so $|C| = 2|F|$.

Every vertex cover C^* must contain at least one endpoint of each edge in F, and since the edges of F are pairwise disjoint, $|C^*| \geq |F|$. In particular, $OPT \geq |F|$. Therefore

$|C| = 2|F| \leq 2 \cdot OPT$, proving the 2-approximation guarantee.

■

Tightness. The factor 2 is tight for this algorithm: on a complete bipartite graph $K_{n,n}$, the algorithm picks all $2n$ vertices, while the optimum is n. More refined approximations (LP-rounding, semi-definite programming) shave the constant slightly, but no polynomial algorithm is known to achieve a $(2-\epsilon)$-approximation under the Unique Games Conjecture.

Example 14.3 Consider the graph (V, E) shown in Figure 14.1. The minimum vertex cover has size 2: $\{1, 4\}$ covers every edge (vertex 1 covers $(1, 2)$ and $(1, 3)$; vertex 4 covers $(2, 4)$, $(3, 4)$, and $(4, 5)$).

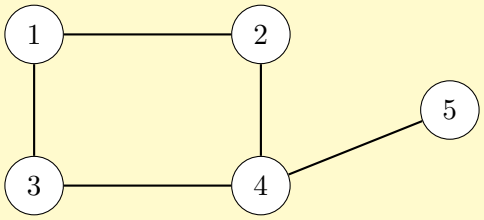


Figure 14.1: Graph for examples 14.3 and 14.6: $V = \{1, \ldots, 5\}$ with $E = \{(1, 2), (1, 3), (2, 4), (3, 4), (4, 5)\}$.

Now consider the execution of the algorithm. **Iter 1.** $E' = E$. Pick the first edge $(1, 2)$; add $\{1, 2\}$ to C. Remove every edge incident to 1 or 2: $(1, 2), (1, 3), (2, 4)$ are removed. $E' = \{(3, 4),\ (4, 5)\}$.

Iter 2. Pick $(3, 4)$; add $\{3, 4\}$ to C. Remove $(3, 4)$ and $(4, 5)$ (incident to 4). $E' = \emptyset$.

The output $C = \{1, 2, 3, 4\}$ has size 4. The matching $F = \{(1, 2), (3, 4)\}$ has size 2, so $|C| = 2|F| = 4 = 2 \cdot OPT$. The algorithm hits the 2-approximation ratio on this instance.

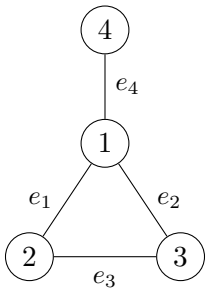
From vertex lattice to edge (matching) lattice

Recall that an LLP algorithm computes the minimum element of a meet-semilattice of feasible states. A natural first attempt is to take the state space to be vectors $G \in \{0, 1\}^n$ where $G[v] = 1$ iff v is in the cover, and the predicate $B(G) =$ "every edge has at least one endpoint with $G[v] = 1$". The set of vertex covers is closed under join (componentwise OR) but *not* under meet: $G_1 = (1, 0, 1, 0)$ and $G_2 = (0, 1, 0, 1)$ may both be vertex covers of a 4-cycle, yet their meet $(0, 0, 0, 0)$ is not. So Vertex Cover is not directly an LLP problem on the vertex lattice.

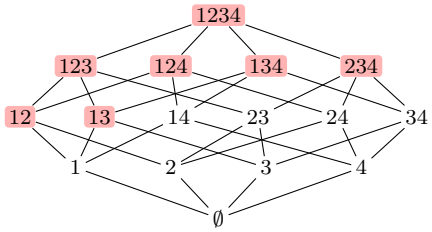
The fix is to switch to an *edge-indexed* lattice. Take $G \in \{0, 1\}^E$, where $G[e] = 1$ iff edge e is in a selected matching F. The intersection of two matchings is again a matching — any subset of pairwise non-adjacent edges is pairwise non-adjacent — so the lattice $\{0, 1\}^E$ admits a clean inf-semilattice structure on edge sets that are matchings. The cover C is then derived

as the union of endpoints of the selected edges: $C = \bigcup_{e \in E,\, G[e]=1} e$. This is the form we adopt below.

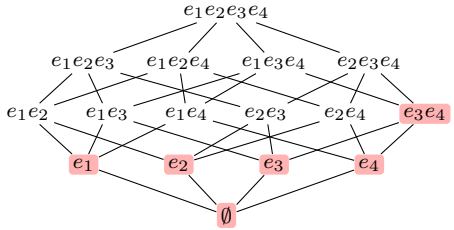
Figure 14.2 illustrates the contrast on a small example. Panel (a) shows a 4-vertex graph: a triangle on $\{1,2,3\}$ with a pendant edge $\{1,4\}$. Panel (b) shows the Boolean lattice of vertex subsets, with vertex covers shaded. The two minimum vertex covers $\{1,2\}$ and $\{1,3\}$ have meet $\{1\}$, which is *not* a vertex cover — so the shaded region is not closed under meet. Panel (c) shows the same Boolean lattice viewed as subsets of edges, with matchings shaded. Every subset of a matching is a matching, so the shaded region is downward-closed and, in particular, closed under meet. This is the structural reason for moving from the vertex lattice to the edge lattice.



(a) Graph G: triangle $\{1,2,3\}$ with pendant edge $\{1,4\}$. Edges: e_1=$\{1,2\}$, e_2=$\{1,3\}$, e_3=$\{2,3\}$, e_4=$\{1,4\}$.



(b) Vertex lattice 2^V. Vertex covers are shaded red. Note: $\{1,2\} \wedge \{1,3\} = \{1\}$ is unshaded — not closed under meet.



(c) Edge lattice 2^E. Matchings are shaded red. Downward-closed: subsets of matchings are matchings, so closed under meet.

Figure 14.2: From the vertex lattice to the edge lattice. (a) the input graph. (b) the lattice of vertex subsets; vertex covers (red) form an up-set but are *not* meet-closed — the meet of the minimum covers $\{1,2\}$ and $\{1,3\}$ is $\{1\}$, which is not a cover. (c) the lattice of edge subsets; matchings (red) form a downward-closed family and *are* meet-closed.

Lex-Match: an edge-indexed LLP

Order the edges of E lexicographically by the canonical sorted endpoint pair — $(1,3) < (1,5) < (2,3) < \ldots$. Call an edge e *uncovered at* G if no currently-selected edge shares an endpoint with e, and *lex-minimal at* G if it is uncovered and lex-smaller than every other uncovered edge sharing an endpoint with it. The algorithm is: in parallel, select every lex-minimal edge; repeat until no edge is uncovered.

Algorithm LLP-LexMatchVertexCover: Parallel 2-approximation for vertex cover via lex-minimal edges

input: graph (V, E) with vertices labeled $1, \ldots, n$
output: $C \subseteq V$: a vertex cover

1 G: array$[E]$ of boolean, initially $\forall e \in E :\ G[e] :=$ false ; `// matching edges`
2 *uncovered*$(e) := \forall\, e' \in E :\ G[e'] \Rightarrow e \cap e' = \emptyset$; `// no selected edge shares an endpoint with` e
3 *lexmin*$(e) :=$ *uncovered*$(e) \wedge \forall\, e' \in E :\ (\textit{uncovered}(e') \wedge e \cap e' \neq \emptyset) \Rightarrow e \leq_{\text{lex}} e'$;
4 **forbidden**(e): *lexmin*(e)
5 $\Rightarrow$ **advance**: $G[e] :=$ true;
6 **return** $C := \{v \in V : \exists\, e \in E \text{ with } G[e] \wedge v \in e\}$

The selected edges form a matching. The key structural lemma is that, in any single round, the set of lex-minimal edges is pairwise non-adjacent.

Lemma 14.4 *At any state G, no two lex-minimal edges share an endpoint.*

Proof: Suppose (u, v) and (v, w) are both lex-minimal at G. Each is uncovered; each shares the endpoint v with the other; hence each is among the neighbors of the other. Lex-minimality of (u, v) gives $(u, v) \leq_{\text{lex}} (v, w)$, and lex-minimality of (v, w) gives $(v, w) \leq_{\text{lex}} (u, v)$. So $(u, v) = (v, w)$, contradicting their distinctness.

■

Lemma 14.4 reduces the LLP-LexMatchVertexCover algorithm, in a single round, to picking a matching F of lex-minimal edges and adding all endpoints to the cover — exactly the structure used by the sequential ApproxVertexCover of Section 14.3. The 2-approximation argument carries over verbatim; LLP-LexMatchVertexCover is a special instance of Algorithm ApproxVertexCover. Hence,

Theorem 14.5 *LLP-LexMatchVertexCover is a 2-approximation algorithm for the minimum Vertex Cover problem.*

Termination and worst-case parallel depth. At least one edge is selected per round: the globally lex-smallest uncovered edge has no lex-smaller incident edge and so is trivially lex-minimal. Each selected edge contributes two new endpoints to the derived cover C, and a vertex never leaves C once it enters. Hence $|C|$ grows by at least 2 per round, and the algorithm terminates in at most $\lceil n/2 \rceil$ rounds.

This $\Theta(n)$ bound is tight in the worst case. Take the path $v_1 - v_2 - \cdots - v_n$ with the natural labeling. In round 1 only the edge (v_1, v_2) fires: every other edge (v_i, v_{i+1}) with $i \geq 2$ has the incident edge (v_{i-1}, v_i) which is lex-smaller, so (v_i, v_{i+1}) is not lex-minimal. Hence, the algorithm takes $\Theta(n)$ rounds.

Example 14.6 $V = \{1, 2, \ldots, 8\}$ with edges

$$E = \{(1,2),\ (1,5),\ (3,4),\ (3,7),\ (4,5),\ (5,6),\ (7,8)\}.$$

Vertex 5 is a hub of degree 3 connecting the path 2−1−5−6 on top to the branch 5−4−3−7−8 below (Figure 14.3). Sorted lexicographically: $(1,2) < (1,5) < (3,4) < (3,7) < (4,5) < (5,6) < (7,8)$.

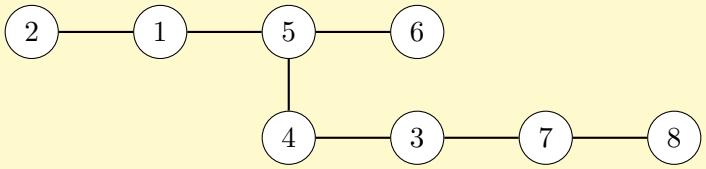


Figure 14.3: Connected graph for Example 14.6. Vertex 5 is the hub linking the two branches.

Round 1. Two edges fire in parallel:

- $(1,2)$: incident uncovered edges $\{(1,2),(1,5)\}$; lex-min is $(1,2)$, so $(1,2)$ is lex-minimal.
- $(3,4)$: incident uncovered edges $\{(3,4),(3,7),(4,5)\}$; lex-min is $(3,4)$, so $(3,4)$ is lex-minimal.

Advance both selected edges in parallel: $G[(1,2)] :=$ true and $G[(3,4)] :=$ true. Edges $(1,5)$, $(3,7)$, and $(4,5)$ now share a cover endpoint with a selected edge and so are no longer uncovered, leaving $(5,6)$ and $(7,8)$ as the only uncovered edges.

Round 2. Two more edges fire in parallel: $(5,6)$ and $(7,8)$. Advance: $G[(5,6)] :=$ true and $G[(7,8)] :=$ true.

Termination. Every edge has an endpoint in the cover, so no edge is uncovered and the algorithm halts. Output $C = \{1,2,3,4,5,6,7,8\}$, $|C| = 8$. A minimum vertex cover is $\{1,4,5,7\}$ of size 4, so $OPT = 4$ and the algorithm hits the tight ratio $|C|/OPT = 2$.

Weighted Vertex Cover

A natural extension is the *minimum weight Vertex Cover* problem: each vertex v has a weight $w(v) \geq 0$, and the goal is a cover $C \subseteq V$ minimizing $w(C) = \sum_{v \in C} w(v)$. The decision version is NP-complete, since it contains unweighted Vertex Cover as the case $w \equiv 1$.

The matching argument fails. The lex-first algorithm of Section 14.3 bounded $|C| = 2|F|$ for a matching F and used $|C^*| \geq |F|$ to conclude $|C| \leq 2 \cdot OPT$. With weights, the same construction gives $w(C) = \sum_{(u,v) \in F} (w(u) + w(v))$, while the optimum need pay only $\min(w(u), w(v))$ on each matching edge — so the ratio can grow with the weight imbalance, and is unbounded in general. (A single edge (a,b) with $w(a) = 1, w(b) = K$ gives ratio $1 + K$.)

A matching alone does not witness enough of the weighted optimum.

Edge prices and tightness. The standard repair, due to Bar-Yehuda and Even [BYE85], replaces the discrete matching with a continuous *pricing* process. Maintain a price $G[\{u,v\}] \geq 0$ on each edge $\{u,v\} \in E$. Vertex v is *tight* when its incident prices saturate its weight,

$$\mathit{tight}(v) \;\equiv\; \sum_{\{u,v\}\in E} G[\{u,v\}] \;\geq\; w(v),$$

and an edge $\{u,v\}$ is *slack* when neither endpoint is tight. To keep the process a well-formed LLP computation, we raise the price of only the *lex-minimal* slack edges — those lex-smaller than every adjacent slack edge — reusing the *lexmin* rule of LLP-LexMatchVertexCover. By the same argument as Lemma 14.4, distinct lex-minimal slack edges are pairwise non-adjacent, so raising them in parallel never touches a shared vertex. Each selected edge $\{u,v\}$ is raised by the amount

$$r_{\{u,v\}} \;:=\; \min\Big(\, w(u) - \sum_{\{u,x\}\in E} G[\{u,x\}],\quad w(v) - \sum_{\{v,y\}\in E} G[\{v,y\}] \,\Big),$$

which makes at least one of its endpoints tight while preserving the per-vertex budget $\sum_{\{u,v\}\in E} G[\{u,v\}] \leq w(v)$. The output cover is $C(G) := \{v : \mathit{tight}(v)\}$.

Algorithm LLP-WeightedVertexCover: Parallel 2-approximation for weighted vertex cover via lex-minimal slack edges

input: n: int; E: set of edges (each $\{u,v\}$ with $u,v \in [1..n]$); w: array$[1..n]$ of real
output: C: array$[1..n]$ of boolean

1 G: array$[E]$ of real, initially $\forall \{u,v\} \in E :\; G[\{u,v\}] := 0$; `// lattice of edge prices`
2 $\mathit{price}(v) := \sum_{\{u,v\}\in E} G[\{u,v\}]$;
3 $\mathit{tight}(v) := (\mathit{price}(v) \geq w[v])$;
4 $\mathit{slack}(e) := \neg\, \mathit{tight}(u) \wedge \neg\, \mathit{tight}(v)$; `// for` $e = \{u,v\}$
5 $\mathit{slackmin}(e) := \mathit{slack}(e) \wedge \forall\, e' \in E :\; (\mathit{slack}(e') \wedge e \cap e' \neq \emptyset) \Rightarrow e \leq_{\text{lex}} e'$;
6 **forbidden**(e): $\mathit{slackmin}(e)$
7 $\Rightarrow$ **advance**: $G[e] := G[e] + \min\,(w[u] - \mathit{price}(u),\; w[v] - \mathit{price}(v))$, where $e = \{u,v\}$;
8 **return** $C[v] := \mathit{tight}(v)$ *for all* $v \in [1..n]$

At least one vertex becomes tight per round, so the loop terminates in at most $|V|$ rounds.

Approximation guarantee. The total price summed over all edges lower-bounds the optimum cover weight.

Lemma 14.7 *At every step the prices satisfy the per-vertex budget* $\sum_{\{u,v\}\in E} G[\{u,v\}] \leq w(v)$ *for every* $v \in V$. *Consequently* $\sum_{e\in E} G[e] \leq OPT$.

Proof: *Budget feasibility.* The bound $\sum_{\{u,v\}\in E} G[\{u,v\}] \leq w(v)$ holds initially (when $G \equiv 0$) and is preserved by every advance: a selected edge $\{u,v\}$ is raised by $r_{\{u,v\}}$, which is at most the

remaining budget of each endpoint, so neither budget is exceeded; and since the lex-minimal slack edges of a round are pairwise non-adjacent, no vertex is raised by two edges in the same round.

Price bound. Let C^* be any vertex cover. Assign each edge to one of its endpoints in C^*. The edges assigned to any $v \in C^*$ are a subset of v's incident edges, so their total price is at most $\sum_{\{u,v\}\in E} G[\{u,v\}] \le w(v)$. Summing over C^* gives $\sum_{e\in E} G[e] \le w(C^*)$. Taking $C^* = OPT$ yields the claim.

■

Theorem 14.8 *LLP-WeightedVertexCover returns a vertex cover C with $w(C) \le 2{\cdot}OPT$.*

Proof: The loop terminates only when no edge is slack, so every edge has a tight endpoint and C is a valid cover. Every $v \in C$ is tight, so $w(v) = \sum_{\{u,v\}\in E} G[\{u,v\}]$. Summing over C counts each $G[e]$ at most twice (once per endpoint in C), so $w(C) \le 2\sum_{e\in E} G[e] \le 2 \cdot OPT$ by Lemma 14.7.

■

LLP perspective. The lattice variable is now real-valued: $G \in \mathbb{R}^E_{\ge 0}$ ordered componentwise. The forbidden predicate "$\{u,v\}$ is a lex-minimal slack edge" is lattice-linear: because distinct lex-minimal slack edges are non-adjacent, advancing one never affects another's endpoints, so a forbidden edge stays forbidden until it is itself advanced. Each advance raises $G[\{u,v\}]$ monotonically, so the lattice traversal is bottom-up, exactly as in every LLP algorithm in this chapter. Two features distinguish this case from unweighted Vertex Cover: the lattice is continuous rather than $\{0,1\}^V$, and each advance raises an edge's price by the amount that makes one of its endpoints tight. This pattern — continuous LLP variables coordinating through a shared resource — recurs in primal-dual approximations and reappears below for weighted Set Cover.

Specialization to unit weights. With $w \equiv 1$, every vertex starts with price 0, so every edge is slack and *slackmin* reduces to the lex-minimal uncovered edge — exactly the *lexmin* rule of LLP-LexMatchVertexCover. Advancing a lex-minimal edge $\{u,v\}$ raises $G[\{u,v\}]$ by $\min(1,1) = 1$, making both endpoints tight at once, so both enter the cover. Thus for unit weights LLP-WeightedVertexCover is *identical* to LLP-LexMatchVertexCover, and their 2-approximation guarantees coincide.

Example 14.9 [LLP-WeightedVertexCover on a weighted star] Take the star $K_{1,3}$ in Figure 14.4: center c of weight $w(c) = 10$ and leaves ℓ_1, ℓ_2, ℓ_3 of weights $1, 4, 4$ respectively, with edges $e_i = (c, \ell_i)$ for $i \in \{1, 2, 3\}$. The optimum cover is $\{\ell_1, \ell_2, \ell_3\}$ of weight 9 (vs. $\{c\}$ of weight 10), so $OPT = 9$.

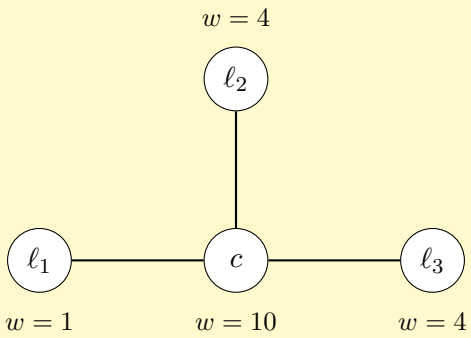


Figure 14.4: Weighted star $K_{1,3}$ for example 14.9.

Round 1. All three edges are slack. They all meet the center c, so they are pairwise adjacent, and the lex-minimal one, e_1, is the only slack-minimal edge. Advance e_1 by $\min\,(w(c)-0,\ w(\ell_1)-0) = \min(10, 1) = 1$, so $G[e_1] = 1$. Now ℓ_1 is tight; $price(c) = 1 < 10$, and ℓ_2, ℓ_3 still have price 0.

Round 2. The slack edges are e_2 and e_3 (e_1 is covered via tight ℓ_1); both meet c, so the lex-minimal one is e_2. Advance e_2 by $\min\,(w(c)-price(c),\ w(\ell_2)-0) = \min(10-1, 4) = 4$, so $G[e_2] = 4$. Now ℓ_2 is tight; $price(c) = 5 < 10$.

Round 3. The only slack edge is e_3. Advance it by $\min\,(w(c) - price(c),\ w(\ell_3) - 0) = \min(10 - 5, 4) = 4$, so $G[e_3] = 4$. Now ℓ_3 is tight; $price(c) = 9 < 10$.

Termination. Every edge has a tight endpoint, so no edge is slack. Output $C = \{\ell_1, \ell_2, \ell_3\}$ of weight $9 = OPT$. The bound $w(C) \le 2 \cdot OPT = 18$ is comfortably satisfied; in fact dual feasibility is tight, with $\sum_e G[e] = 1 + 4 + 4 = 9 = OPT$. (Because every edge of a star meets the center, the lex-minimal rule advances one edge per round here; parallel advances occur when the slack edges form a larger matching.)

14.4 Set Cover

Set Cover generalizes Vertex Cover: given a universe U of n elements and a collection $\mathcal{S}$ of subsets of U, find the smallest sub-collection $\mathcal{C} \subseteq \mathcal{S}$ whose union is U. (Vertex Cover is the special case in which each element of U corresponds to an edge and each set in $\mathcal{S}$ to the incident edges of a vertex.) Set Cover admits a clean greedy approximation: repeatedly pick the set in $\mathcal{S}$ that covers the most uncovered elements, until every element is covered.

Algorithm ApproxSetCover: ApproxSetCover(U, $\mathcal{S}$)

input: universe U, collection $\mathcal{S}$ of subsets of U
output: a sub-collection $\mathcal{C} \subseteq \mathcal{S}$ covering U

$\mathcal{C} := \emptyset$; $R := U$;
while $R \neq \emptyset$ **do**
  Pick $S \in \mathcal{S}$ maximizing $|S \cap R|$;
  $\mathcal{C} := \mathcal{C} \cup \{S\}$; $R := R \setminus S$;
end
return $\mathcal{C}$

Theorem 14.10 *ApproxSetCover returns a cover of size at most $H_n \cdot OPT$, where $H_n = \sum_{k=1}^{n} 1/k = O(\ln n)$ is the n-th harmonic number and $n = |U|$.*

The proof, due to Chvátal [Chv79], charges each newly covered element a fee of $1/|S \cap R|$ at the moment S is picked; the total fee equals $|\mathcal{C}|$, while a careful averaging argument shows the fee on any optimal set is at most H_n, yielding $|\mathcal{C}| \leq H_n \cdot OPT$. The full charging argument is left as a problem; see Problem 10.

Comparison with Vertex Cover. Specializing ApproxSetCover to a Vertex Cover instance gives an $O(\log n)$-approximation, weaker than the 2-approximation of Section 14.3. The 2-approximation exploits the special structure of Vertex Cover (each element appears in exactly two sets) that the general greedy ignores. This pattern recurs: domain-specific structure often beats the generic greedy.

Example 14.11 [ApproxSetCover] Take $U = \{1, 2, 3, 4, 5, 6\}$ and $\mathcal{S} = \{S_1, S_2, S_3\}$ where $S_1 = \{1, 2, 3, 4\}$, $S_2 = \{1, 2, 5\}$, $S_3 = \{3, 4, 6\}$. The optimal cover is $\{S_2, S_3\}$ since $S_2 \cup S_3 = U$; no single set covers U, so $OPT = 2$.

Iter 1. $R = U$. Coverage: $|S_1 \cap R| = 4$, $|S_2 \cap R| = 3$, $|S_3 \cap R| = 3$. Pick the unique maximum $S_1 = \{1, 2, 3, 4\}$. $\mathcal{C} = \{S_1\}$, $R = \{5, 6\}$.

Iter 2. Coverage on R: $|S_2 \cap R| = 1$ (element 5), $|S_3 \cap R| = 1$ (element 6). Tied at 1; pick S_2 (lex-min). $\mathcal{C} = \{S_1, S_2\}$, $R = \{6\}$.

Iter 3. Only S_3 covers element 6. Pick S_3. $R = \emptyset$.

The output is $\mathcal{C} = \{S_1, S_2, S_3\}$ of size 3, while $OPT = 2$. The greedy is fooled by the large attractor S_1: it covers four elements at once, leaving two stranded elements $\{5, 6\}$ that no single remaining set can cover together. The ratio is $3/2$, well within the bound $|\mathcal{C}| \leq H_6 \cdot OPT \approx 2.45 \cdot 2 \approx 4.9$.

Lexically-First Set Cover

The lex-min idea generalizes directly to Set Cover. The sequential greedy of Section 14.4 picks one set at a time; the parallel LLP analogue picks every set that is locally maximal in coverage

and lex-minimal among ties. Two such sets can never share an uncovered element, so they can be added to the cover simultaneously.

Let R denote the set of currently uncovered elements. Two sets $s, s' \in \mathcal{S}$ are *neighbors at* G if $(s \cap R) \cap (s' \cap R) \neq \emptyset$, i.e., they compete for at least one uncovered element. Define

$$\mathit{lexmaxcov}(s) \equiv (s \cap R \neq \emptyset) \wedge \forall s' \text{ neighbor of } s\colon \big(|s \cap R| > |s' \cap R|\big) \vee \big(|s \cap R| = |s' \cap R| \wedge s \leq_{\text{lex}} s'\big).$$

A set is selected in a round iff it is *lexmaxcov*.

Algorithm LLP-LexGreedySetCover: Parallel H_n-approximation for set cover via lex-first locally-maximal sets

input: universe U; collection $\mathcal{S} = \{s_1, \ldots, s_m\}$ with each $s_i \subseteq U$
output: $\mathcal{C} \subseteq \mathcal{S}$

G: array[$\mathcal{S}$] of boolean, initially $\forall s \in \mathcal{S}:\ G[s] :=$ false ; `// set indicators -- the lattice`
$\mathit{covered}(e) := \exists\, s \in \mathcal{S}:\ G[s] \wedge e \in s$; `// e is in some selected set`
$\mathit{cov}(s) := |\{e \in s : \neg\, \mathit{covered}(e)\}|$; `// coverage of s on uncovered elements`
$\mathit{nbr}(s, s') := \exists\, e \in s \cap s':\ \neg\, \mathit{covered}(e)$; `// s, s' share an uncovered element`
$\mathit{lexmaxcov}(s) := \mathit{cov}(s) > 0 \wedge \forall s' \in \mathcal{S}:\ \mathit{nbr}(s, s') \Rightarrow (\mathit{cov}(s) > \mathit{cov}(s') \vee (\mathit{cov}(s) = \mathit{cov}(s') \wedge s \leq_{\text{lex}} s'))$;
forbidden(s): $\mathit{lexmaxcov}(s)$
  $\Rightarrow$ **advance**: $G[s] :=$ true;
return $\mathcal{C} := \{s \in \mathcal{S} : G[s]\}$

Selected sets are pairwise disjoint on R. The matching argument from Lemma 14.4 carries over.

Lemma 14.12 *At any state G, no two lexmaxcov sets share an uncovered element.*

Proof: Suppose s and s' are both *lexmaxcov* and $(s \cap R) \cap (s' \cap R) \neq \emptyset$. Then they are neighbors. From s's local optimality, $|s \cap R| > |s' \cap R|$ or ties with $s \leq_{\text{lex}} s'$. From s''s local optimality, the symmetric inequality. The two strict cases contradict each other, so $|s \cap R| = |s' \cap R|$ and $s \leq_{\text{lex}} s' \leq_{\text{lex}} s$, hence $s = s'$.

■

Approximation guarantee. The parallel version retains the H_n bound of the sequential greedy.

Theorem 14.13 *LLP-LexGreedySetCover returns a cover of size at most $H_n \cdot OPT$, where H_n is the n-th harmonic number and $n = |U|$.*

Proof: We adapt Chvátal's charging argument. When set s is selected in round t with current uncovered set R_t, charge each newly-covered element $x \in s \cap R_t$ a fee of $1/|s \cap R_t|$. The total fee equals the number of selected sets, i.e., $|\mathcal{C}|$.

Now bound the total fee on any single optimal set S^*. Order the elements of S^* as $x_1, x_2, \ldots, x_p$ by the round in which they are first covered: $t_1 \le t_2 \le \cdots \le t_p$. At round t_i, x_i is covered by some selected set s containing x_i. Since x_i is also in S^* and uncovered at round t_i, the sets s and S^* share an uncovered element x_i, so S^* is a neighbor of s at round t_i. Local maximality of s then gives $|s \cap R_{t_i}| \ge |S^* \cap R_{t_i}| \ge p - i + 1$ (since $x_i, x_{i+1}, \ldots, x_p$ are still uncovered). Hence the fee on x_i is at most $1/(p-i+1)$, and the total fee on S^* is at most $\sum_{i=1}^{p} 1/(p-i+1) = H_p \le H_n$.

Summing over the OPT optimal sets covers every element of U at least once, so $|\mathcal{C}|$ = total fee $\le H_n \cdot OPT$.

■

Remark. A single round may select several sets at once, but this does not weaken the bound. By Lemma 14.12 the sets chosen in a round are pairwise disjoint on the uncovered universe, so each newly covered element is charged exactly once; and local maximality bounds each selected set's fee independently, since any optimal set S^* sharing an uncovered element with a selected set s is a neighbor of s and hence satisfies $|S^* \cap R| \le |s \cap R|$. In particular, a set enters the cover only when no strictly larger set overlaps its uncovered elements, so parallel selection never adds "too many" small sets: the parallel greedy is never worse than the sequential H_n bound.

Vertex Cover as a special case. Vertex Cover is the special case in which the universe is the edge set E, each set $s_v = \{e : v \in e\}$ corresponds to a vertex v, and every element is in exactly two sets. Specializing LLP-LexGreedySetCover to this case selects, in each round, every *vertex* that is locally maximal in degree-among-uncovered-edges and lex-minimal among ties — a strictly different selection rule from LLP-LexMatchVertexCover, which selects *edges*. The two rules do *not* carry the same guarantee, however. The edge-selection rule (LLP-LexMatchVertexCover) picks a matching and is a 2-approximation (Section 14.3). The vertex-selection rule inherits only the harmonic bound of Theorem 14.13: greedy-by-degree applied to Vertex Cover can be $\Theta(\log n)$ times optimal in the worst case, so it is *not* a 2-approximation. Their parallel structures also differ: the edge-selection rule produces a matching of edges, while the vertex-selection rule produces, in each round, a set of vertices no two of which share an uncovered edge (Lemma 14.12).

Example 14.14 [LLP-LexGreedySetCover on the running instance] Take the same instance as Example 14.11: $U = \{1, \ldots, 6\}$ and

$$\mathcal{S} = \{S_1{=}\{1,2,3,4\},\ S_2{=}\{1,2,5\},\ S_3{=}\{3,4,6\}\}$$

(Figure 14.5). Order the sets by index: $S_1 <_{\text{lex}} S_2 <_{\text{lex}} S_3$.

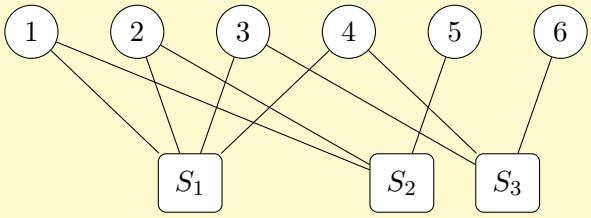


Figure 14.5: Element-set incidence for examples 14.11 and 14.14: $U = \{1, \ldots, 6\}$, $\mathcal{S} = \{S_1, S_2, S_3\}$.

Round 1. $R = U$. Coverage: $|S_1 \cap R| = 4$, $|S_2 \cap R| = 3$, $|S_3 \cap R| = 3$.

- S_1 neighbors S_2 (share $1, 2$) and S_3 (share $3, 4$). $cov(S_1) = 4 > cov(S_2) = cov(S_3) = 3$, so S_1 dominates both. S_1 is *lexmaxcov*.
- S_2 neighbors S_1 only ($S_2 \cap S_3 = \emptyset$). Loses to S_1 on coverage. Not *lexmaxcov*.
- S_3 similarly loses to S_1 on coverage. Not *lexmaxcov*.

Only S_1 is selected. Set $G[S_1] :=$ true. $R = \{5, 6\}$.

Round 2. Coverage on $R = \{5, 6\}$: $|S_2 \cap R| = 1$ (only 5), $|S_3 \cap R| = 1$ (only 6). S_2 and S_3 share no uncovered element, so they are *not* neighbors — each is trivially *lexmaxcov* on its own. Both are selected in parallel: $G[S_2] :=$ true, $G[S_3] :=$ true. $R = \emptyset$.

The output is $\mathcal{C} = \{S_1, S_2, S_3\}$ of size 3 in 2 rounds, while $OPT = 2$. The greedy ratio is $3/2$, well within $H_6 \cdot OPT \approx 4.9$. This instance also illustrates parallelism: Round 2 selects two sets simultaneously because they no longer compete for any uncovered element.

Weighted Set Cover

The pricing technique of LLP-WeightedVertexCover generalizes directly to weighted Set Cover, where each set $s_i \in \mathcal{S}$ carries a weight $w(s_i) \geq 0$ and the goal is a sub-collection $\mathcal{C} \subseteq \mathcal{S}$ of minimum total weight covering U. The lattice variable is now an *element price* $G[e] \geq 0$ for each element $e \in U$ (in place of the edge price of Vertex Cover), and a set s_i is *tight* when $\sum_{e \in s_i} G[e] \geq w(s_i)$. As in LLP-WeightedVertexCover, to keep this a well-formed LLP computation we raise the price of only the *lex-minimal* slack elements — an uncovered element lex-smaller than every uncovered element sharing a set with it. Distinct lex-minimal slack elements never share a set, so raising them in parallel charges each set's budget through at most one element. Each selected element e is raised by $\min\{\, w(s) - \sum_{e' \in s} G[e'] : s \in \mathcal{S},\ e \in s \,\}$, which makes at least one set containing e tight (thereby covering e) while preserving every per-

set budget $\sum_{e \in s_i} G[e] \le w(s_i)$.

Algorithm LLP-WeightedSetCover: Parallel f-approximation for weighted set cover via lex-minimal element pricing.

input: Universe U; collection $\mathcal{S} = \{s_1, \dots, s_m\}$ with each $s_i \subseteq U$; weight $w : \mathcal{S} \to \mathbb{R}_{\ge 0}$
output: $\mathcal{C} \subseteq \mathcal{S}$

G: array$[U]$ of real, initially $\forall e \in U :\ G[e] := 0$; `// element prices -- the lattice`
$price(s) := \sum_{e \in s} G[e]$;
$tight(s) := (price(s) \ge w(s))$;
$slack(e) := \neg\, \exists\, s \in \mathcal{S} :\ tight(s) \wedge e \in s$; `// e lies in no tight set`
$slackmin(e) := slack(e) \wedge \forall\, e' \in U :\ \big(slack(e') \wedge (\exists\, s \in \mathcal{S} :\ e \in s \wedge e' \in s)\big) \Rightarrow e \le_{\text{lex}} e'$;
forbidden(e): $slackmin(e)$
  $\Rightarrow$ **advance**: $G[e] := G[e] + \min\{\, w(s) - price(s) \ :\ s \in \mathcal{S},\ e \in s \,\}$;
return $\mathcal{C} := \{s \in \mathcal{S} : tight(s)\}$

Approximation guarantee. Let $f := \max_{e \in U} |\{i : e \in s_i\}|$ be the maximum element frequency in $\mathcal{S}$. Budget feasibility ($\sum_{e \in s_i} G[e] \le w(s_i)$ at every step) holds because each advance raises an element's price by at most the remaining slack of every set containing it, and distinct lex-minimal slack elements share no set, so no set's budget is charged twice in a round. It gives $\sum_e G[e] \le OPT$ by the same double-counting argument as Lemma 14.7: for any cover $\mathcal{C}^*$, $\sum_e G[e] \le \sum_e G[e] \cdot |\{s \in \mathcal{C}^* : e \in s\}| = \sum_{s \in \mathcal{C}^*} \sum_{e \in s} G[e] \le \sum_{s \in \mathcal{C}^*} w(s) = w(\mathcal{C}^*)$. Tightness on every selected set then yields

$$w(\mathcal{C}) \;=\; \sum_{s \in \mathcal{C}} \sum_{e \in s} G[e] \;=\; \sum_{e \in U} G[e] \cdot |\{s \in \mathcal{C} : e \in s\}| \;\le\; f \sum_{e \in U} G[e] \;\le\; f \cdot OPT.$$

Vertex Cover is the special case $f = 2$, recovering Theorem 14.8. For instances with large f the greedy H_n-approximation of Section 14.4 is tighter, but the pricing variant has the complementary advantage of being inherently parallel and producing a primal-dual certificate.

Example 14.15 [LLP-WeightedSetCover on a small instance] Let $U = \{1, 2, 3\}$ with three weighted sets

$$s_1 = \{1\}\ (w = 3), \qquad s_2 = \{1, 2\}\ (w = 4), \qquad s_3 = \{1, 3\}\ (w = 5).$$

Element 1 lies in all three sets, so the maximum frequency is $f = 3$. Elements 2 and 3 can be covered only by s_2 and s_3 respectively, so the optimum is $\mathcal{C}^\star = \{s_2, s_3\}$ of weight $OPT = 9$.

Round 1. Every element is slack. Elements 2 and 3 each share a set with 1 (namely s_2 and s_3), and 1 is lex-smallest, so the only lex-minimal slack element is 1. Advance 1 by $\min\big(w(s_1), w(s_2), w(s_3)\big) = \min(3, 4, 5) = 3$, so $G[1] = 3$. Now $\mathit{price}(s_1) = 3 = w(s_1)$, so s_1 is tight and element 1 is covered; $\mathit{price}(s_2) = 3 < 4$ and $\mathit{price}(s_3) = 3 < 5$.

Round 2. The slack elements are 2 and 3. They share no set (their only common neighbor, element 1, is already covered), so both are lex-minimal and advance *in parallel*. Advancing 2 raises $G[2]$ by $w(s_2) - \mathit{price}(s_2) = 4 - 3 = 1$; advancing 3 raises $G[3]$ by $w(s_3) - \mathit{price}(s_3) = 5 - 3 = 2$. Now $\mathit{price}(s_2) = 4$ and $\mathit{price}(s_3) = 5$, so s_2 and s_3 are tight and elements $2, 3$ are covered.

Termination. Every element lies in a tight set. The output is $\mathcal{C} = \{s_1, s_2, s_3\}$ of weight 12. The dual prices $G = (3, 1, 2)$ certify a lower bound $\sum_e G[e] = 6 \le OPT = 9$, and the guarantee holds: $w(\mathcal{C}) = 12 \le f \sum_e G[e] = 18 \le f \cdot OPT = 27$. Note that the cover admits the redundant set s_1 — element 1 is already covered by s_2 and s_3 — because every tight set is taken; this is the source of the factor-f looseness.

14.5 Approximation Schemes: PTAS and FPTAS

A constant-factor approximation is sometimes too coarse. Two finer-grained notions make the ratio a tunable parameter.

Definition 14.16 (PTAS) *A* polynomial-time approximation scheme (PTAS) *for a minimization problem Π is a family of algorithms $\{A_\epsilon\}_{\epsilon > 0}$ such that, for every fixed $\epsilon > 0$, A_ϵ runs in time polynomial in the input size and outputs a $(1 + \epsilon)$-approximate solution.*

Definition 14.17 (FPTAS) *A* fully polynomial-time approximation scheme (FPTAS) *is a PTAS whose running time is polynomial in both the input size and $1/\epsilon$.*

The distinction matters in practice: a PTAS may have a running time like $O(n^{1/\epsilon})$, blowing up rapidly as $\epsilon \to 0$, whereas an FPTAS keeps the dependence on $1/\epsilon$ polynomial.

An FPTAS for Knapsack via value scaling

Recall the Knapsack problem from Chapter 13: given n items with weights $w_1, \ldots, w_n$, profits $v_1, \ldots, v_n$, and a knapsack capacity W, find a subset $S \subseteq [n]$ with $\sum_{i \in S} w_i \le W$ maximizing

$\sum_{i \in S} v_i$. Knapsack admits a standard pseudo-polynomial dynamic program in $O(nV)$ time, where $V = \sum_i v_i$. The FPTAS scales the profit values down so that the DP table becomes polynomially-bounded, while losing only a $(1-\epsilon)$ factor in the objective.

We first discard every item with $w_i > W$: such an item can never belong to a feasible solution, and if none remains the optimum is the empty set. Let $M = \max_i v_i$ be the largest profit among the remaining (feasible) items. Since that item fits by itself, $M \le OPT \le nM$. Define the scaling factor

$$scale = \frac{\epsilon M}{n}$$

and replace each profit by its scaled-and-rounded counterpart

$$v'_i = \left\lfloor \frac{v_i}{scale} \right\rfloor .$$

Now run the standard $O(nV')$ DP on the scaled instance, where $V' = \sum_i v'_i$.

Algorithm FPTAS-Knapsack: Fully polynomial-time approximation scheme for knapsack via value scaling

input: items (w_i, v_i) for $i \in [n]$, capacity W, accuracy parameter $\epsilon > 0$
output: subset $S \subseteq [n]$ with profit at least $(1-\epsilon) \cdot OPT$

$F := \{i \in [n] : w_i \le W\}$; `// discard items that cannot fit`
if $F = \emptyset$ **then return** $\emptyset$;
$M := \max_{i \in F} v_i$; $scale := \epsilon M / n$;
forall $i \in F$ **do**
  $v'_i := \lfloor v_i / scale \rfloor$
end
$S :=$ optimal Knapsack solution for $\{(w_i, v'_i) : i \in F\}$ with capacity W via the standard $O(nV')$ DP;
return S

Theorem 14.18 *FPTAS-Knapsack returns a feasible solution of profit at least* $(1-\epsilon) \cdot OPT$ *in* $O(n^3/\epsilon)$ *time.*

Proof: Rounding loses less than *scale* per item, so for any subset T, $\sum_{i \in T} v_i - n \cdot scale < \sum_{i \in T} scale\, v'_i \le \sum_{i \in T} v_i$. Let S^* be an optimal solution and S the solution returned by the scaled DP. Since S is optimal under the scaled values, $\sum_{i \in S} v_i \ge \sum_{i \in S} scale\, v'_i \ge \sum_{i \in S^*} scale\, v'_i > OPT - n \cdot scale = OPT - \epsilon M \ge (1-\epsilon) \cdot OPT$, where the last step uses $OPT \ge M$. The scaled DP has $V' \le nM/scale = n^2/\epsilon$, so it runs in $O(nV') = O(n^3/\epsilon)$.

■

Example 14.19 [FPTAS-Knapsack on a 3-item instance] Take $n = 3$ items with $(w_i, v_i) = (2, 30),\ (3, 40),\ (4, 50)$ and capacity $W = 6$. The original DP $O(nV)$ has $V = 120$. The optimum on this instance is $S^* = \{1, 3\}$ with profit 80 (weight $2 + 4 = 6 \le W$).

Choose accuracy $\epsilon = 0.2$. Then $M = 50$ and $scale = \epsilon M/n = 0.2 \cdot 50/3 \approx 3.33$. The scaled values are

$$v'_1 = \lfloor 30/3.33 \rfloor = 9, \quad v'_2 = \lfloor 40/3.33 \rfloor = 12, \quad v'_3 = \lfloor 50/3.33 \rfloor = 15.$$

The scaled DP picks the same set $S = \{1, 3\}$ (since $9{+}15 = 24 > 21 = 9{+}12$) for capacity 6. Recovering original profits: $\sum_{i \in S} v_i = 30{+}50 = 80$. The bound $(1{-}\epsilon){\cdot}OPT = 0.8{\cdot}80 = 64$ is comfortably satisfied; in fact this scaling preserved optimality exactly.

The benefit of the scaling: instead of running the DP up to $V = 120$, the scaled DP runs only up to $V' = 9 + 12 + 15 = 36$, a 3.3× reduction. For larger instances the gap grows: at $V = 10^6$ and $\epsilon = 0.2$, $V' \le n^2/\epsilon$ stays polynomial, while the unscaled DP is pseudo-polynomial.

Many geometric and scheduling problems admit a PTAS but not an FPTAS unless P = NP; see [DPW10] for a comprehensive treatment of the design space.

LLP-ApproxKnapsack via lattice scaling

The FPTAS-Knapsack of Section 14.5 fits cleanly into the LLP framework, provided the table is indexed the way the FPTAS DP is — by *profit*, not by capacity. (A capacity-indexed table $G[i, w]$, $w \in [0..W]$, has $\Theta(nW)$ cells; scaling the profits leaves W untouched, so it would stay pseudo-polynomial.) After the same preprocessing as FPTAS-Knapsack — discard every item with $w_i > W$ (returning the empty set if none remains), so that $M = \max_i v_i \le OPT$ — and scaling each profit to $v'_i = \lfloor v_i / scale \rfloor$, let $V' = \sum_i v'_i = O(n^2/\epsilon)$ and define

$$D[i, p] = \text{the minimum total weight of a subset of } \{1, \ldots, i\} \text{ with scaled profit at least } p,$$

for $i \in [0..n]$ and $p \in [0..V']$ (with $D[i, p] = \infty$ when no such subset exists). The lattice-linear feasibility predicate is the upper bound

$$D[i, p] \le \min\left(D[i-1, p],\ w_i + D[i-1,\ \max(0, p - v'_i)]\right),$$

with boundary conditions $D[0, 0] = 0$, $D[0, p] = \infty$ for $p \ge 1$, and $D[i, 0] = 0$. We order table entries by *reverse* numerical order — $x \preceq y$ iff $x \ge y$ as ordinary numbers — so a smaller weight sits higher in the lattice and *lowering* an entry is an advance. The table reached when no cell is forbidden (starting from ∞ at the bottom and lowering entries) is the table of minimum weights; the scaled optimum is $p^\star = \max\{\, p : D[n, p] \le W \,\}$, giving profit estimate $scale \cdot p^\star \ge (1 - \epsilon)\, OPT$.

Because the profit axis has height $V' = O(n^2/\epsilon)$, the lattice has $(n+1)(V'+1) = O(n^3/\epsilon)$ cells — polynomial in n and $1/\epsilon$, and independent of W. This is what makes the scheme fully polynomial; a capacity-indexed lattice would instead grow with W and remain merely pseudo-polynomial.

Algorithm LLP-ApproxKnapsack: Parallel LLP FPTAS for knapsack via profit scaling

input: items (w_i, v_i) for $i \in [n]$, capacity W, accuracy $\epsilon > 0$
output: scaled optimum $p^\star = \max\{p : D[n,p] \le W\}$ with $scale \cdot p^\star \ge (1-\epsilon)\,OPT$

$F := \{i \in [n] : w_i \le W\}$; `// discard items that cannot fit`
if $F = \emptyset$ **then return** $\emptyset$;
reindex the items of F as $1, \ldots, n$;
$M := \max_i v_i$; $\quad scale := \epsilon M/n$;
forall $i \in [n]$ **do**
  $v'_i := \lfloor v_i / scale \rfloor$
end
$V' := \sum_i v'_i$;
$D[0,0] := 0$; $D[0,p] := \infty$ for $p \in [1..V']$; $D[i,0] := 0$ and $D[i,p] := \infty$ for $i \ge 1$, $p \ge 1$;
forbidden(i,p) with $i \ge 1$, $p \ge 1$:
  $D[i,p] > \min\left(D[i-1,p],\ w_i + D[i-1, \max(0, p - v'_i)]\right)$
  $\Rightarrow$ **advance**: $D[i,p] := \min\left(D[i-1,p],\ w_i + D[i-1, \max(0, p - v'_i)]\right)$;
return $\max\{\, p : D[n,p] \le W \,\}$

The approximation guarantee carries over directly from Theorem 14.18: the LLP advance loop solves the same scaled instance optimally.

Example 14.20 [LLP-ApproxKnapsack on the running instance] Take the same instance as Example 14.19: items $(w_i, v_i) = (2,30),\ (3,40),\ (4,50)$ with capacity $W = 6$ and accuracy $\epsilon = 0.2$. Pre-processing gives $M = 50$, $scale = 0.2 \cdot 50/3 \approx 3.33$, and scaled profits $v'_1 = 9,\ v'_2 = 12,\ v'_3 = 15$, so $V' = 36$.

Subsets. Each subset, with its (total weight, scaled profit), is

$$\begin{array}{lll} \emptyset\,(0,0), & \{1\}\,(2,9), & \{2\}\,(3,12), \\ \{3\}\,(4,15), & \{1,2\}\,(5,21), & \{1,3\}\,(6,24), \\ \{2,3\}\,(7,27), & \{1,2,3\}\,(9,36). & \end{array}$$

Forbidden / Advance trace. Every cell (i,p) with $i \geq 1$ starts at ∞ and is forbidden until $D[i,p]$ falls to $\min\left(D[i-1,p],\, w_i + D[i-1,\max(0,p-v_i')]\right)$. At convergence, the last row $D[3,\cdot]$ — the minimum weight to reach scaled profit at least p, with no capacity constraint yet imposed — is the step function

$$D[3,p] = \begin{cases} 0 & p = 0, \\ 2 & 1 \leq p \leq 9, \\ 3 & 10 \leq p \leq 12, \\ 4 & 13 \leq p \leq 15, \\ 5 & 16 \leq p \leq 21, \\ 6 & 22 \leq p \leq 24, \\ 7 & 25 \leq p \leq 27, \\ 9 & 28 \leq p \leq 36. \end{cases}$$

Applying the capacity at readout, $p^\star = \max\{p : D[3,p] \leq W\} = 24$ (since $D[3,24] = 6 \leq 6$ but $D[3,25] = 7 > 6$), realized by $S = \{1,3\}$. The corresponding original profit is $scale \cdot p^\star \approx 3.33 \cdot 24 = 80$, exactly OPT on this instance, comfortably above the $(1-\epsilon)\cdot OPT = 64$ guarantee.

The profit-indexed table has $V' + 1 = 37$ columns here — wider than the $W + 1 = 7$ capacity columns only because the instance is tiny. Its size is $O(n^3/\epsilon)$, independent of W, which is what keeps the scheme fully polynomial as W grows.

14.6 Constrained Approximation Algorithms

The approximation algorithms above make each LLP decision (add edge endpoints, pick a max-coverage set, raise edge or element prices) without external constraints. Real applications often add side constraints — "vertex v in the cover forces vertex w in the cover", "these vertex pairs must be selected together" — and the LLP formulations slot them in cleanly as new conjuncts of the forbidden predicate.

The pattern is to pair an existing chapter algorithm with a small constraint program on the same lattice G. The LLP loop is run on the conjunction of the two forbidden predicates: an index j is forbidden when *either* predicate forbids it, and the advance rule is the one provided by whichever predicate raised the flag. This composition is a clean way to enforce side constraints, but it should be emphasised that the approximation factor of the original algorithm does *not* automatically carry over to the constrained version. The reason is subtle: the chapter algorithm's local rule may pick a vertex (or set) whose implication closure under the constraints is much larger than the optimal constrained cover, even when an alternative choice would have triggered no implications at all. We make this explicit in the discussion

below and in a problem that asks the reader to construct a family of inputs on which the constrained algorithm's ratio diverges (see the chapter problems).

Vertex Cover with Vertex Implications

A set $I \subseteq V \times V$ of *implications* is given: each pair $(v, w) \in I$ requires that if v is in the cover, then so is w. We compose this constraint with the matching-based LLP-LexMatchVertexCover of Section 14.3. The two naturally live on different lattices — LLP-LexMatchVertexCover on the edge lattice $\{0,1\}^E$, and the implication constraint on the cover lattice $\{0,1\}^V$, where a violation of (v, w) is repaired by adding the consequent w, never by removing the antecedent. To run them together we take the *product* lattice

$$L \;=\; \{0,1\}^E \times \{0,1\}^V,$$

ordered componentwise, with state (G_E, G_V): $G_E[e]$ records whether edge e is selected into the matching, and $G_V[w]$ whether vertex w is in the cover. Three forbidden/advance rules act on L; the LLP loop advances any index that any rule forbids.

Algorithm VC-with-Implications: Constrained vertex cover on the product lattice $\{0,1\}^E \times \{0,1\}^V$

input: graph (V, E) with vertices $1, \ldots, n$; $I \subseteq V \times V$: implications
output: $C \subseteq V$: a vertex cover satisfying I

G_E: array[E] of boolean, initially 0; G_V: array[V] of boolean, initially 0 ;
`// product lattice`
$\mathit{matched}(w) := \exists\, e \in E :\; G_E[e] \wedge w \in e$;
$\mathit{uncov}(e) := \forall\, w \in e :\; \neg\, \mathit{matched}(w)$; `// no endpoint of` e `is matched`
$\mathit{lexmin}(e) := \mathit{uncov}(e) \wedge \forall\, e' \in E :\; \big(\mathit{uncov}(e') \wedge e \cap e' \neq \emptyset\big) \Rightarrow e \leq_{\text{lex}} e'$;
Matching (advances G_E, from LLP-LexMatchVertexCover):;
forbidden(e): $\mathit{lexmin}(e)$
  $\Rightarrow$ **advance**: $G_E[e] := 1$;
Coupling (advances G_V):;
forbidden(w): $\exists\, e \in E :\; G_E[e] \wedge w \in e \wedge \neg\, G_V[w]$
  $\Rightarrow$ **advance**: $G_V[w] := 1$;
Implications (advances G_V; the VertexImplications constraint):;
forbidden(w): $\exists\, v :\; (v, w) \in I \wedge G_V[v] \wedge \neg\, G_V[w]$
  $\Rightarrow$ **advance**: $G_V[w] := 1$;
return $C := \{w \in V : G_V[w]\}$

An index of L is forbidden when any of the three rules forbids it, and the advance is the one supplied by that rule. The *matching* rule drives G_E exactly as in standalone LLP-LexMatchVertexCover — it reads only G_E (through *matched*) and never inspects G_V, so the implications cannot disturb it. The *coupling* rule lifts each matched endpoint into the cover G_V, and the *implications* rule (the VertexImplications constraint) closes G_V under I. All three predicates are lattice-linear on L: coupling and implications are Horn implications with a unique repair ($G_V[w] := 1$), and matching is the edge-indexed lattice-linear rule of Sec-

tion 14.3. At the least fixpoint every edge has a matched — hence covered — endpoint and every implication is honoured, so $C = \{w : G_V[w]\}$ is a vertex cover satisfying I.

Approximation factor. The output cover satisfies the implications, but its size need not be within a factor of 2 of the constrained optimum. The lex-min rule of LLP-LexMatchVertexCover may force a vertex v into the cover whose implication closure is far larger than the optimal cover that selects the *other* endpoint of v's incident edge and triggers no implications. Problem 11 asks the reader to construct a family of instances on which the ratio $|C|/OPT_{\text{constrained}}$ is unbounded.

Example 14.21 [VC-with-Implications on a two-edge graph] Let $V = \{1, 2, 3, 4, 5\}$ with edges $E = \{(1, 2), (3, 4)\}$ and the single implication $I = \{(2, 5)\}$ ("if 2 is in the cover, so is 5"). Edges are ordered $(1, 2) <_{\text{lex}} (3, 4)$, and the state is the pair (G_E, G_V), listed below as the sets of indices set to 1.

Round 1 (Matching). No vertex is matched, so both edges are uncovered. They share no endpoint, hence are non-adjacent, and each is lex-minimal in its own neighbourhood — so both are forbidden and advance in parallel: $G_E = \{(1, 2), (3, 4)\}$, $G_V = \emptyset$.

Round 2 (Coupling). Now $1, 2, 3, 4$ are matched but not yet in the cover, so the coupling rule forbids all four and lifts them into G_V: $G_V = \{1, 2, 3, 4\}$.

Round 3 (Implications). The implication $(2, 5)$ is now violated ($2 \in G_V$, $5 \notin G_V$), so 5 is forbidden and added: $G_V = \{1, 2, 3, 4, 5\}$.

Termination. Every edge has a matched endpoint and the implication is honoured. The output is $C = \{1, 2, 3, 4, 5\}$ of size 5.

The constrained optimum is $\{1, 3\}$ of size 2: covering $(1, 2)$ with vertex 1 (rather than 2) leaves 2 out of the cover, so the implication $(2, 5)$ never fires. The algorithm's ratio here is $5/2$, and the cause is exactly the blow-up analysed above: the matching rule commits *both* endpoints of each selected edge, so vertex 2 enters the cover and drags in 5 — even though covering $(1, 2)$ with its other endpoint would have triggered nothing.

14.7 Hardness of Approximation

For some optimization problems, even getting close to the optimum is NP-hard:

- **General (non-metric) TSP.** With *arbitrary* edge weights, no polynomial-time algorithm can approximate the minimum-weight tour to within *any finite* factor unless P = NP, by reducing Hamiltonian Cycle to TSP with edge weights 1 for present edges and K for absent edges; for any target ratio α, choosing $K > \alpha n$ forces every α-approximation to use only weight-1 edges, thereby solving HC. The large weights K violate the triangle inequality, so this result concerns *non-metric* inputs only; *metric* TSP, whose weights satisfy the triangle inequality, admits a constant-factor approximation (a 2-approximation via a doubled MST, improved to $\frac{3}{2}$ by Christofides).
- **Max-3-SAT.** Approximating the maximum number of simultaneously satisfiable clauses within a factor better than $\frac{7}{8}$ is NP-hard (Håstad's theorem). This bound is tight: a

uniformly random assignment satisfies each 3-literal clause with probability $\frac{7}{8}$, so by linearity of expectation the expected number of satisfied clauses is $\frac{7}{8}$ of all clauses — and hence at least $\frac{7}{8} \cdot OPT$. This expected guarantee can be made deterministic by the method of conditional expectations, yielding a $\frac{7}{8}$-approximation algorithm.

- **Set Cover.** The H_n ratio above is essentially optimal: approximating Set Cover within $(1-\epsilon)\ln n$ for any $\epsilon > 0$ is NP-hard.
- **Vertex Cover.** Approximating within a factor better than 1.36 is known to be NP-hard (Dinur–Safra), and the Unique Games Conjecture would push this to $2-\epsilon$, matching Section 14.3.

These lower bounds rely on the PCP theorem (probabilistically checkable proofs); the modern theory is treated in Arora–Barak [AB09].

14.8 Summary

This chapter introduced the theory and practice of approximation algorithms.

Problem	Algorithm	Approximation Ratio
Vertex Cover	ApproxVertexCover (matching)	2
Set Cover	ApproxSetCover (greedy)	H_n ($\approx \ln n$)
Knapsack	FPTAS via rounding + DP	$\frac{1}{1-\epsilon}$

The Knapsack entry deserves a word on convention. Knapsack is a *maximization* problem, and the FPTAS returns profit at least $(1-\epsilon)\,OPT$. By Definition 14.1, this is an approximation ratio of $\alpha = \frac{1}{1-\epsilon}$ (since $A(I) \geq \frac{1}{\alpha}\,OPT$); for small ϵ the ratio $\frac{1}{1-\epsilon}$ is commonly reparameterized as $1+\epsilon$. We list $\frac{1}{1-\epsilon}$ so that the table is consistent with the $(1-\epsilon)\,OPT$ guarantee proved in Theorem 14.18.

14.9 Problems

1. **(Tight Vertex Cover Approximation)** Show that the 2-approximation factor of ApproxVertexCover is tight by constructing a family of graphs on which the algorithm returns a cover of size exactly $2 \cdot OPT$.

2. **(F-Frequency Set Cover)** Suppose the input to Set Cover guarantees that every element of U appears in at most f sets. Show that an ApproxVertexCover-style argument gives an f-approximation. In particular, when $f = 2$ the problem reduces to Vertex Cover and we recover the 2-approximation.

3. **(Tight Set Cover Hn Bound)** Prove that the H_n analysis of ApproxSetCover is tight: exhibit a family of instances on which the greedy returns a cover of size approximately $H_n \cdot OPT$.

4. **(PTAS Versus FPTAS)** The PTAS for the Euclidean TSP runs in time roughly $n^{O(1/\epsilon)}$. Why is this only a PTAS and not an FPTAS? What does Garey–Johnson tell us about FPTAS-existence for strongly NP-hard problems?

5. **(Steiner Tree Two-Approximation)** The Steiner Tree problem asks for the minimum-weight tree spanning a designated subset R of the vertices in an edge-weighted graph. Give a 2-approximation by reducing to MST on the metric closure restricted to R.

6. **(Incremental Vertex Cover)** A 2-approximate vertex cover C has been computed on a graph $G = (V, E)$ by ApproxVertexCover. A new edge (u, v) is inserted. Update C to a 2-approximate cover on the augmented graph in $O(1)$ amortised time per insertion.

7. **(Dynamic Set Cover under Element Removal)** A greedy H_n-approximate cover $\mathcal{C}$ has been computed for $(U, \mathcal{S})$. An element e is removed from U. If e was covered only by $S \in \mathcal{C}$ and S has no other remaining covered elements, can S be removed from $\mathcal{C}$? Discuss the approximation factor of the maintained cover.

8. **(Constrained Knapsack with Mandatory Items)** A subset $F \subseteq [n]$ of items must be included in the knapsack. Adapt the FPTAS-Knapsack algorithm to respect F and return a $(1 - \epsilon)$-approximate solution among F-respecting packings.

9. **(Approximation with Ties: Tied Greedy in Set Cover)** The greedy Set Cover algorithm picks an arbitrary maximum-coverage set at each step. Show that the H_n bound holds for any tie-breaking rule, and exhibit an instance where two different tie-breakers produce covers of different sizes.

10. **(ApproxSetCover H_n Bound (Chvátal))** Prove Theorem 14.10: the greedy ApproxSetCover algorithm returns a cover of size at most $H_n \cdot OPT$, where $H_n = \sum_{k=1}^{n} 1/k = O(\ln n)$ is the n-th harmonic number and $n = |U|$. Use the charging argument of Chvátal [Chv79]: at each iteration, distribute a unit of "fee" among the elements that the greedy just covered, and bound the total fee charged to any single set $S^* \in \mathcal{S}$.

11. **(Constrained Vertex Cover Ratio Blow-up)** The constrained vertex-cover algorithm of Section 14.6 composes LLP-LexMatchVertexCover with the VertexImplications constraint program. Construct a family of instances (V_n, E_n, I_n) on which the algorithm returns a cover C with $|C|/OPT_{\text{constrained}} \to \infty$ as $n \to \infty$, where $OPT_{\text{constrained}}$ is the minimum cover that satisfies the implications. Conclude that the 2-approximation of LLP-LexMatchVertexCover does *not* carry over to the constrained version.

12. **(Enumerate Approximate Vertex Covers)** List all vertex covers whose size is at most $(1 + \epsilon) \cdot OPT$. Bound the count and the per-cover time.

14.10 Bibliographic Remarks

The ApproxVertexCover algorithm and its 2-approximation guarantee are folklore; the matching-based proof appears already in Gavril (unpublished, ca. 1974) and Bar-Yehuda–Even [BYE85]. The greedy Set Cover analysis is due to Chvátal [Chv79] (and independently Johnson [Joh74] and Lovász [Lov75]). The FPTAS for Knapsack was given by Ibarra and Kim [IK75].

The hardness-of-approximation results rely on the PCP theorem of Arora, Lund, Motwani, Sudan, and Szegedy [ALM+98]; Håstad's tight $\frac{7}{8}$ bound for Max-3-SAT appears in [Hås01].

Vazirani's monograph [Vaz03] is the standard reference for the design and analysis of approximation algorithms; Williamson and Shmoys [WS11] give a complementary modern treatment with strong emphasis on LP-based methods.

Chapter 15

The Housing Allocation Problem

15.1 Introduction

Imagine a residential complex of n tenants, each currently living in one of n apartments and each holding strong opinions about which of the other apartments they would prefer. Or imagine n research interns assigned to lab benches, n teams sharing n time-slots on a piece of expensive equipment, or n students rotating through n projects — in each setting, every participant arrives with a property they consider their own and a private ranking of every other participant's property. The question is whether and how the participants can *swap among themselves* to reach an allocation that no group can collectively beat by trading inside the group. This is the *housing allocation problem*. Its appeal is that it is purely cooperative: there is no money, no central planner choosing winners and losers, and no participant ends up worse off than when they started. It captures the structure of one-sided matching with initial endowments cleanly enough that the same algorithm reappears in kidney-exchange markets, on-campus housing assignment, and the allocation of compute slots in shared high-performance computing (HPC) clusters.

The housing market problem proposed by Shapley and Scarf [SS74] is a matching problem with one-sided preferences. There are n agents and n houses. Each agent a_i initially owns a house h_i for $i \in \{1, \ldots, n\}$ and has a completely ranked list of houses. There are variations of this problem when the agents do not own any house initially. In this book, we focus on the version with the initial endowment of houses for agents. The list of agent preferences is given by $pref[i][k]$ which specifies the k^{th} preference of the agent i. Thus, $pref[i][1] = j$ means that a_i prefers h_j as his top choice. The goal is to come up with an optimal house allocation such that each agent has a house, and no subset of agents can improve the satisfaction of agents in this subset by exchanging houses within the subset. It can be shown that there is a unique such matching called the *core* for any housing market. The standard algorithm for this problem is Gale's Top Trading Cycle Algorithm (TTC), which takes $O(n^2)$ time. This algorithm is optimal in terms of time complexity, since the input size is $O(n^2)$.

This chapter is organized as follows. Section 15.2 describes Gale's Top Trading Cycle Algorithm. Section 15.3 applies the LLP method to the housing market problem.

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

15.2 Gale's Top Trading Cycle Algorithm

Consider the housing market instance shown in Fig. 15.1. There are four agents a_1, a_2, a_3 and a_4. Initially, the agent a_i holds the house h_i. The preferences of the agents are shown in Fig. 15.1.

$a_1 : h_2, h_3, h_1, h_4$	$a_1 : h_1$	$a_1 : h_2$
$a_2 : h_1, h_4, h_2, h_3$	$a_2 : h_2$	$a_2 : h_1$
$a_3 : h_1, h_2, h_4, h_3$	$a_3 : h_3$	$a_3 : h_4$
$a_4 : h_2, h_1, h_3, h_4$	$a_4 : h_4$	$a_4 : h_3$
(i) Agents' Preferences	(ii) Initial Allocation	(iii) Matching returned by TTC

Figure 15.1: Housing market and the matching returned by the top trading cycle algorithm

The Top Trading Cycle (TTC) algorithm attributed to Gale by Shapley and Scarf [SS74] works in stages. At each stage, it has the following steps:

Step 1. We construct the *top choice* directed graph $\Gamma_t = (A, E)$ on the set of agents A as follows. We add a directed edge from agent $a_i \in A$ to agent $a_j \in A$ if a_j holds the current top house of a_i. Fig. 15.2 shows the directed graph in the first stage.

Step 2. Since each node has exactly one outgoing edge in Γ_t, there is at least one cycle in the graph (possibly a self-loop). All cycles are node-disjoint. We find all the cycles in the top trading graph and implement the trade indicated by the cycles, that is, each agent that is in any cycle gets its current top house.

Step 3. Remove all agents who get their current top houses, and remove all houses which are assigned to some agent from the preference list of remaining agents.

The above steps are repeated until each agent is assigned a house. At each stage, at least one agent is assigned a final house, so there are at most n stages. Within a stage, building Γ_t, finding all cycles in it, and updating preference lists all take $O(n)$ time on the unassigned agents (each $wish[i]$ pointer only moves forward across stages, so its total movement across the run of the algorithm is $O(n)$ per agent). The overall running time is therefore $O(n^2)$, which is optimal in the input size.

Beyond its running time, the TTC mechanism has three properties that make it the algorithm of choice for housing-market problems in practice. (i) *Individual rationality*: every agent ends up with a house at least as preferred as their endowment h_i (in the worst case, an agent's own house remains its top choice and forms a self-loop in Γ_t). (ii) *Pareto-optimality*: no allocation makes some agent strictly better off without making another strictly worse off. (iii) *Strategy-proofness*: a self-interested agent has no incentive to misreport preferences — truthful reporting is a dominant strategy [Rot82]. The combination of (i)–(iii) is rare among matching mechanisms; it is the reason TTC reappears in kidney exchange, school choice, and dormitory-allocation systems (see Section 15.6).

Algorithm TTC states the procedure formally.

Algorithm TTC: Gale's top trading cycle algorithm

input: $pref[i][k]$: the k^{th} choice of agent a_i, for $i, k \in [1..n]$. By renumbering houses, agent a_i initially owns house h_i.

output: $house[1..n]$: $house[i]$ is the house finally assigned to agent a_i.

```
1  fixed[1..n]: array of boolean, initially ∀i : fixed[i] := false;
2  wish[1..n]: array of int, initially ∀i : wish[i] := pref[i][1] ;        // current top
   remaining choice
3  while ∃i : ¬fixed[i] do
       // Step 1: build the top-choice graph Γ_t on unassigned agents
4      forall i : ¬fixed[i] do
5          advance wish[i] down pref[i] until wish[i] is held by some unassigned agent a_j;
6          add edge a_i → a_j to Γ_t;
7      end
       // Step 2: every node has out-degree 1, so Γ_t contains at least one
          cycle; cycles are node-disjoint
8      C := set of agents lying on some cycle of Γ_t;
       // Step 3: implement the trades indicated by the cycles and remove
          the participants
9      forall i ∈ C do
10         house[i] := wish[i];
11         fixed[i] := true;
12     end
13 end
14 return house;
```

Example 15.1 [TTC on the running instance] Consider the four-agent instance of Fig. 15.1. Each a_i initially owns h_i and the preference lists are $a_1 : h_2, h_3, h_1, h_4$; $a_2 : h_1, h_4, h_2, h_3$; $a_3 : h_1, h_2, h_4, h_3$; $a_4 : h_2, h_1, h_3, h_4$.

Stage 1. Top-choice graph (Fig. 15.2): $a_1 \to a_2$, $a_2 \to a_1$, $a_3 \to a_1$, $a_4 \to a_2$. The unique cycle is $\{a_1, a_2\}$, so a_1 takes h_2 and a_2 takes h_1. Both are fixed and removed.

Stage 2. Only a_3 and a_4 remain, with houses h_3 and h_4 unallocated. Top-choice now: $a_3 \to a_4$ (top remaining is h_4), $a_4 \to a_3$ (top remaining is h_3). The cycle $\{a_3, a_4\}$ assigns $a_3 \mapsto h_4$ and $a_4 \mapsto h_3$.

The final core allocation is a_1:h_2, a_2:h_1, a_3:h_4, a_4:h_3.

15.3 An LLP Algorithm for the Housing Market Problem

There are n agents and n houses. Each agent proposes houses in the decreasing order of preferences. These proposals are considered events executed by n agents. Thus, we have n events per agent. Each event is labeled as (i, h, k), which corresponds to the agent i proposing to the house h as his choice number k.

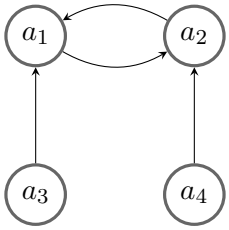


Figure 15.2: The top choice graph at the first stage.

The global state corresponds to the number of proposals made by each of the agents. Let $G[i]$ be the number of proposals made by the agent i. We will assume that in the initial state, every agent has made his first proposal. Thus, the initial global state $G = [1, 1, \ldots, 1]$. We extend the notation of indexing to subsets $J \subseteq [n]$ such that $G[J]$ corresponds to the subvector given by the indices in J.

We now model the possibility of reallocation of houses based on any global state. Recall that $pref[i][k]$ specifies the k^{th} preference of the agent a_i. Let $wish(G, i)$ denote the house proposed by a_i in the global state G, that is,

$$wish(G, i) = pref[i][G[i]]$$

A global state G satisfies *matching* if every agent proposes a different house, that is,

$$matching(G) \equiv \forall i, j : i \neq j : wish(G, i) \neq wish(G, j).$$

We generalize *matching* to refer to a subset of agents rather than the entire set.

Definition 15.2 (submatching) *Let $J \subseteq [n]$. Then $submatching(G, J)$ iff $wish(G, J)$ is a permutation of the indices in J.*

Intuitively, if $submatching(G, J)$ holds, then all agents in J can exchange houses within the subset J.

For any G, it is easy to show that

Lemma 15.3 *For all G, there always exists a nonempty J such that $submatching(G, J)$.*

Proof: Given any G, we can create a directed graph as follows. The set of vertices is agents, and there is an edge from i to j if $wish(G, i) = j$. There is exactly one outgoing edge from any vertex in $[n]$ to $[n]$ in this graph. This implies that there is at least one cycle in this graph (possibly a self-loop). The indices of agents in the cycle give us such a subset J.

■

We now show that

Lemma 15.4 *submatching*(G, J_1) *and* *submatching*(G, J_2) *implies* *submatching*$(G, J_1 \cup J_2)$.

Proof: The directed graph defined by $wish(G, \cdot)$ is a functional graph (every vertex has out-degree exactly one), so its vertex set decomposes into vertex-disjoint directed cycles plus tails leading into them. A set J satisfies $submatching(G, J)$ if and only if J is a union of some of these cycles. Since cycles are vertex-disjoint, $J_1 \cup J_2$ is again a union of cycles and therefore satisfies $submatching(G, J_1 \cup J_2)$.

■

Hence, there exists the biggest submatching in G. Note that $matching(G)$ is equivalent to $submatching(G, [n])$.

Definition 15.5 (Feasible Global State) *A global state G is feasible for the housing market problem iff it is a matching and for all global states $F < G$, there does not exist any submatching which is better in F than in G. Informally, no coalition of agents can obtain a strictly better allocation by collectively reverting to an earlier proposal vector. Note that if there exists a submatching J that is better in F than in G, then agents in J can improve their allocation by just exchanging houses within the subset J. Formally, let*

$$B_{housing}(G) \equiv matching(G) \wedge (\forall F < G : \forall J \subseteq [n] : submatching(F, J) \Rightarrow F[J] = G[J]).$$

We show that $B_{housing}(G)$ is a lattice-linear predicate. This result will let us use the lattice-linear predicate detection algorithm for the housing market problem.

Theorem 15.6 *The predicate $B_{housing}(G)$ is lattice-linear.*

Proof: Suppose that $\neg B_{housing}(G)$. This implies that either G is not a matching or it is a matching, but there exists a smaller global state F that has a submatching better than G.

First, consider the case when G is not a matching. Let J be the largest set such that $submatching(G, J)$. Consider any index $i \notin J$ such that $wish(G, i) \in J$. We claim $forbidden(G, i, B_{housing})$. Let H be any global state greater than G such that $G[i] = H[i]$.

We consider two cases.

Case 1: $H[J] > G[J]$. Then, from the second conjunct of $B_{housing}$, we know that $\neg B_{housing}(H)$ because $submatching(G, J)$ and $H[J] \neq G[J]$.

Case 2: $H[J] = G[J]$.
Since $wish(H, i) = wish(G, i)$, $wish(G, i) \in J$, and $G[J] = H[J]$, we get that H is not a matching because the house given by $wish(G, i)$ is also in the wish list of some agent in J.

Now consider the case where G is a matching, but $\neg B_{housing}(G)$. This implies,

$$\exists F < G : \exists J \subseteq [n] : submatching(F, J) \wedge F[J] < G[J].$$

However, the same F will also result in guaranteeing $\neg B_{housing}(H)$ for any $H \geq G$.

■

It is also easy to see from the proof that if an index is part of a submatching, then it will never become forbidden.

This theorem gives us the algorithm shown in Algorithm LLP-Housing-Market-Algorithm. Let G be the initial global state. Let $S(G)$ be the biggest submatching in G; since $S(G)$ is the union of all cycles in the functional graph defined by $wish(G, \cdot)$, it can be computed in $O(n)$ time. All agents such that they are not in $S(G)$ and wish a house which is part of $S(G)$ are forbidden and can move on to their next proposal. The algorithm terminates when no agent is forbidden. This algorithm is a parallel version of the top trading cycle (TTC) mechanism attributed to Gale in [SS74].

Algorithm LLP-Housing-Market-Algorithm: LLP algorithm to find the core of the housing market

input: $pref[i][k]$: the k^{th} preference of agent i
output: $G[1..n]$: proposal vector giving the core allocation

```
1 G: array[1..n] of int initially 1 ;      // every agent starts with the top choice
2 wish(G, i) = pref[i][G[i]];
3 S(G) = largest J such that submatching(G, J);
4 forbidden(j): (j ∉ S(G)) ∧ (wish(G, j) ∈ S(G))
5    ⇒ advance: G[j] := G[j] + 1;
```

Example 15.7 [LLP-Housing-Market on the running instance] On the four-agent instance of Example 15.1 we trace the proposal vector G:
$G = [1, 1, 1, 1]$. $wish$: $a_1 \to h_2$, $a_2 \to h_1$, $a_3 \to h_1$, $a_4 \to h_2$. The largest submatching is $S(G) = \{1, 2\}$ (the cycle $a_1 \leftrightarrow a_2$). Agents 3 and 4 are not in $S(G)$ and their wishes lie in $S(G)$, so both are forbidden. Advance: $G[3], G[4] := 2$.
$G = [1, 1, 2, 2]$. $wish$: $a_3 \to h_2$, $a_4 \to h_1$. $S(G) = \{1, 2\}$ still; agents $3, 4$ remain forbidden. Advance again.
$G = [1, 1, 3, 3]$. $wish$: $a_3 \to h_4$, $a_4 \to h_3$. Now $S(G) = \{1, 2, 3, 4\}$ (two cycles: $a_1 \leftrightarrow a_2$ and $a_3 \leftrightarrow a_4$). No agent is forbidden; the algorithm terminates. Reading off $pref[i][G[i]]$ gives the same core allocation a_1:h_2, a_2:h_1, a_3:h_4, a_4:h_3 as Example 15.1.

We now show that

Theorem 15.8 *There exists at least one feasible global state G such that $B_{housing}(G)$.*

Proof: Every agent has his own house on the list of preferences. If he ever makes a proposal to his own house, he forms a submatching. That particular event is never forbidden because it is part of a submatching. Hence, the lattice-linear predicate detection algorithm will never mark that event as forbidden. Since such an event exists for all processes, we are guaranteed to never go beyond this global state.

■

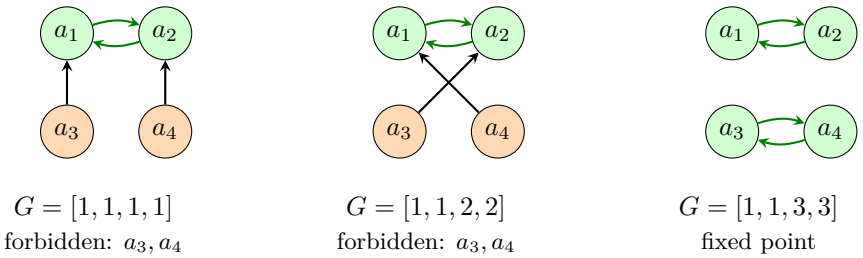


Figure 15.3: LLP-Housing-Market on the running instance of Example 15.7. Each panel shows the wish-graph at a value of G: a directed edge $a_i \to a_j$ means $wish(G, i) = h_j$. Green nodes lie in the largest submatching $S(G)$; orange nodes are forbidden and will advance. Two cycles in panel 3 exhibit the fixed point, where no agent is forbidden and G encodes the core allocation.

The above proof also shows that agents can never be worse off participating in the algorithm. Each agent will either get his own house back or get a house that he prefers to his own house.

We now show that the feasible state is in fact unique — justifying the claim of *the* core in Section 15.2.

Corollary 15.9 *The feasible global state G satisfying $B_{housing}(G)$ is unique.*

Proof: Suppose, for contradiction, that two distinct global states U and V both satisfy $B_{housing}$. Let $W = U \sqcap V$ be their componentwise meet, so $W \leq U$ and $W \leq V$ with $W \neq U$ or $W \neq V$. Because $B_{housing}$ is lattice-linear (Theorem on $B_{housing}$ in this section), it is closed under meet, and hence W also satisfies $B_{housing}$ — in particular $matching(W)$ holds. But then $W < U$ is a global state strictly smaller than U in which every component still proposes a matched house; the second conjunct of $B_{housing}(U)$ requires $U[J] = W[J]$ for every $J \subseteq [n]$ with $submatching(W, J)$, taking $J = [n]$ this forces $U = W$. By the same argument $V = W$, contradicting $U \neq V$.

■

Mechanism-design properties from the LLP framework

The three mechanism-design properties of TTC mentioned in Section 15.2 — individual rationality, Pareto-optimality, and strategy-proofness — become almost immediate consequences of the LLP formulation.

Lemma 15.10 (Individual rationality) *Let G^* be the unique state satisfying $B_{housing}$. For every agent a_i, the assigned house $\mu(i) = pref[i][G^*[i]]$ is weakly preferred to her endowment h_i.*

Proof: Let k_i be the rank of h_i on a_i's preference list, so $pref[i][k_i] = h_i$. We show $G^*[i] \leq k_i$. Consider the state G in which $G[i] = k_i$ and $G[j] = G^*[j]$ for $j \neq i$. The singleton $J = \{i\}$ satisfies $submatching(G, J)$, since $wish(G, i) = h_i$ is owned by a_i herself. Hence a_i is part of a

submatching at $G[i] = k_i$ and is never forbidden once it reaches that value (the second conjunct of $B_{housing}$ pins it there). By monotonicity of the LLP loop, $G^*[i] \leq k_i$, so the assigned house is at least as preferred as h_i.

■

Lemma 15.11 (Pareto-optimality) *The allocation μ derived from G^* is Pareto-optimal.*

Proof: Suppose some allocation μ' Pareto-dominates μ. For each agent a_i, let $G'[i]$ be the rank of $\mu'(i)$ on a_i's preference list, giving a state $G' \leq G^*$ componentwise (with strict inequality at the agent who strictly prefers μ'). Since μ' is a matching, $matching(G')$ holds, so the full set $J = [n]$ is a submatching at G'. By the second conjunct of $B_{housing}(G^*)$, $G'[J] = G^*[J]$ for $J = [n]$ — i.e., $G' = G^*$. This contradicts $G' < G^*$.

■

Lemma 15.12 (Strategy-proofness) *No agent can obtain a strictly better house by misreporting her preferences.*

The proof is left as Problem 6; see also Roth [Rot82] for the original argument.

15.4 Summary

This chapter presented algorithms for the housing market problem, which finds the unique core allocation where no subset of agents can improve by trading among themselves.

Problem	Algorithm	Time Complexity
Housing Market	Gale's Top Trading Cycle	$O(n^2)$
Housing Market	LLP Top Trading Cycle	$O(n^2)$

15.5 Problems

1. **(TTC Four Agent Example)** Consider a housing market with four agents, $\{A_1, A_2, A_3, A_4\}$, who own houses $\{h_1, h_2, h_3, h_4\}$, where A_i owns h_i. Preferences are:

Agent	Preference List
A_1	$h_2 \succ h_4 \succ h_1 \succ h_3$
A_2	$h_1 \succ h_3 \succ h_2 \succ h_4$
A_3	$h_3 \succ h_1 \succ h_4 \succ h_2$
A_4	$h_3 \succ h_2 \succ h_4 \succ h_1$

Table 15.1: Preferences for the agents.

Apply the TTC algorithm to find the final allocation.

2. **(TTC Outcome Uniqueness)** Is the outcome of the TTC algorithm unique for a given set of truthful preferences and initial endowments? Justify your answer.

3. **(NC Core Verification)** Give an algorithm to determine whether the given matching is the core of the housing market.

4. **(TTC Pareto Optimality)** Prove that the allocation returned by the TTC algorithm is Pareto optimal.

5. **(TTC Strategyproofness)** Prove or disprove: The Top Trading Cycle algorithm is strategyproof, i.e., no agent can benefit by misreporting their preferences.

6. **(LLP Strategy-Proofness)** Prove Lemma 15.12: no agent can obtain a strictly better house by misreporting her preferences. Use the LLP formulation of Section 15.3.

7. **(Serial Dictatorship Properties) (Serial Dictatorship.)** In the *Serial Dictatorship* mechanism with a fixed priority order $\pi = (a_{\pi(1)}, \ldots, a_{\pi(n)})$, agents are processed one at a time; each agent chooses her most preferred house among the houses still available. Prove that the resulting allocation is Pareto efficient and strategyproof.

8. **(YRMH-IGYT Individual Rationality) (YRMH-IGYT with Existing Tenants.)** Consider a house-allocation problem with a mix of existing tenants (who own their current house) and newcomers (who do not own any house), plus some vacant houses. The *You Request My House, I Get Your Turn* (YRMH-IGYT) mechanism, given a priority order, works as follows: for each agent in order, either she is assigned her top available house (if that house is vacant or unoccupied by a still-unassigned tenant), or a *request* is made to the current owner who then jumps to the front of the queue. Show that YRMH-IGYT respects the individual rationality of existing tenants (no tenant gets a house she likes strictly less than her endowment).

9. **(Serial Dictatorship Priority Sensitivity) (Sensitivity to Priority Order in TTC.)** In the TTC mechanism for housing markets, the initial endowment is fixed, so the outcome is independent of how we pick among multiple disjoint cycles in one step. Show, however, that for *Serial Dictatorship* (no initial endowments), different priority orders can give Pareto-incomparable allocations. Provide a small example.

10. **(TTC Individual Rationality) (Individual Rationality of TTC.)** Prove that the allocation μ returned by the TTC algorithm is *individually rational*, i.e., every agent weakly prefers $\mu(i)$ to her endowed house h_i.

11. **(Unique Core Allocation) (Uniqueness of the Core.)** Prove that in any housing market (with strict preferences), the TTC allocation is the *unique* core allocation. That is, no matching other than the TTC outcome is in the core.

12. **(Weak Preferences with Ties) (Weak Preferences with Ties.)** Suppose agents are allowed to be *indifferent* between several houses, so that each agent's preference is a weak total order rather than a strict total order. Two natural questions arise.

(a) Show by an example with three agents that the core can contain *more than one* allocation when ties are allowed.

(b) A common workaround is *deterministic tie-breaking*: extend each agent's weak preference to a strict order by breaking ties in some fixed way (for example, by ascending house index), then run ordinary TTC. Prove that the resulting allocation is always Pareto-efficient with respect to the original *weak* preferences, even though it may depend on the tie-breaking rule.

13. **(Multi-unit Housing Types) (Multi-unit Housing / House Types.)** Generalize the housing market by replacing the n distinct houses with k *house types*, where type h has c_h available copies and $\sum_{h=1}^{k} c_h = n$. Each agent owns one specific copy and ranks the *types* (treating all copies of a type as interchangeable). Design an algorithm that returns a core allocation in $O(n^2)$ time and prove its correctness by reduction to ordinary TTC.

14. **(Eligibility Constrained TTC) (Eligibility-constrained TTC.)** Each agent a_i has a private *eligibility set* $E_i \subseteq \{h_1, \ldots, h_n\}$; agent a_i may only be assigned a house in E_i. Eligibility might encode accessibility requirements, class-year restrictions, or program membership. Assume $h_i \in E_i$ for every i (an agent is always eligible for her own house).

(a) Modify the LLP forbidden predicate $B_{housing}$ from Section 15.3 to respect eligibility. Specifically, define $wish_E(G, i)$ that returns the next eligible house for a_i, and write the new forbidden predicate.

(b) Show that the modified predicate is still lattice-linear, so the LLP framework applies unchanged.

(c) Give a small example (three agents, three houses) where eligibility makes the core *strictly worse* for at least one agent than the unconstrained core.

15. **(YRMH-IGYT Mixed Market) (YRMH-IGYT vs. TTC on a Mixed Market.)** Recall the YRMH-IGYT mechanism of Problem 7, applied to a mixed market with both endowed tenants and unendowed newcomers plus some vacant houses. Consider the instance

$$\text{Tenants: } t_1(\text{owns } h_1),\ t_2(\text{owns } h_2); \quad \text{Newcomers: } n_1, n_2; \quad \text{Vacant: } v_1, v_2.$$

Priority order $\pi = (n_1, t_1, n_2, t_2)$. Preferences:

$$n_1 : v_1 \succ h_2 \succ h_1 \succ v_2; \quad t_1 : h_2 \succ v_1 \succ h_1 \succ v_2;$$
$$n_2 : h_1 \succ v_2 \succ h_2 \succ v_1; \quad t_2 : v_1 \succ h_1 \succ h_2 \succ v_2.$$

Compute the YRMH-IGYT allocation step by step and verify that every tenant ends up with a house she weakly prefers to her endowment.

16. **(Incremental Housing Allocation)** A core allocation μ has been computed by TTC on n agents and n houses. A new agent a_{n+1} arrives with her own endowment h_{n+1} and a complete preference list over $h_1, \ldots, h_{n+1}$; every existing agent appends h_{n+1} at some position in their list. Give an algorithm that updates μ to the new core μ' on $n+1$ agents in $O(n^2)$ time, reusing the old proposal vector G.

17. **(Dynamic Housing Allocation)** Agent a_k revises her preference list by swapping two adjacent houses h_p, h_q at positions $G[k]$ and $G[k]+1$. The current core μ (encoded by G) may no longer be a core under the new preferences. Describe a repair procedure and bound its work in terms of the number of agents whose wish actually changes.

18. **(Constrained Housing With Forbidden Trades)** A set F of *forbidden* pairs $\{(a_i, h_j)\}$ is given: agent a_i may never end up with house h_j (perhaps due to allergy or accessibility constraints). Modify TTC to respect F, and show that a core consistent with F may not exist.

19. **(Housing With Ties)** Allow each preference list to contain ties: agent a_i may be indifferent between several houses at the same level. The original TTC requires a strict order to define $wish(G, i)$. Define what "the core" means under ties and give an algorithm that finds one.

20. **(Enumerate All Cores Under Ties)** With ties in preferences the core is no longer unique. Show that the set of cores forms a finite distributive lattice under the componentwise rank order, and give an algorithm to enumerate all cores with $O(n)$ amortised time per core.

15.6 Bibliographic Remarks

The housing market problem has been studied by many researchers [SS74, HZ79, Zho90, AS98, AS99, RP77, Rot82, Dav13]. Possible applications of the housing market problem include assigning virtual machines to servers in cloud computers, allocating graduates to trainee positions, professors to offices, and students to dormitory rooms. This problem has also been studied by Zheng and Garg [ZG19] where it is shown that the problem of verifying that a matching is a core is in NC, but the problem of computing the core is CC-hard. The class CC (Comparator Circuits) is the complexity class containing decision problems that can be solved by comparator circuits of polynomial size. The paper [ZG19] also provides a *distributed* message passing algorithm to find the core with $O(n^2)$ messages. A parallel algorithm is described in [Gar21].

The mechanism-design properties of TTC — individual rationality, Pareto-optimality, and strategy-proofness (Roth [Rot82]) — have made it the basis of several deployed allocation systems. Roth, Sönmez, and Ünver [RSU04] adapted TTC to *kidney-paired donation*, where incompatible donor–recipient pairs swap kidneys to enable transplants that would otherwise be impossible; the mechanism is now used by national kidney-exchange registries. Abdulkadirouğlu and Sönmez [AS99] extended TTC to *college dormitory allocation*, where some students hold endowments (returning residents) while others do not (incoming first-years). Pathak and Sönmez [PS08] analyze the role of TTC in the *school-choice* debate around the Boston mechanism, the version of which (TTC vs. Deferred Acceptance) shaped the redesign of public school assignment in Boston, New Orleans, and elsewhere.

Chapter 16

Linear Programming

16.1 Introduction

Linear programming (LP) is a foundational tool in combinatorial optimization: many problems studied earlier in this book — maximum bipartite matching, shortest paths, market clearing prices, the assignment problem — admit clean LP formulations whose optima are integral and coincide with the combinatorial answer. LP is solvable in polynomial time (Khachiyan [Kha79], Karmarkar [Kar84]) but is not known to be strongly polynomial, whereas LLP yields strongly polynomial algorithms whenever the problem's feasibility predicate is lattice-linear.

The two frameworks differ in what they search and how they search:

- **Underlying space.** LP searches a real vector space; LLP searches a finite distributive lattice.
- **Feasible region.** LP allows any polyhedron $\{x : Ax \geq b, x \geq 0\}$; LLP requires the feasibility predicate to be *lattice-linear*, i.e., closed under componentwise meets and an efficient *forbidden* function. The two notions are incomparable: there are polyhedra that are not lattice-linear and lattice-linear predicates that are not polyhedra.
- **Search direction.** LP algorithms (simplex, interior point) typically maintain feasibility and improve the objective; LLP starts from the least element of the lattice and advances toward feasibility.

Linear programming is fundamentally a geometric optimization problem. Every linear inequality

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \ \leq \ b$$

defines a half-space in $\mathbb{R}^n$. The feasible region of a linear program is the intersection of finitely many half-spaces and is therefore a convex polyhedron.

The objective c^Tx defines a family of parallel hyperplanes; solving the LP amounts to sliding the objective hyperplane until it reaches the furthest feasible point in the optimization direction.

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

A fundamental theorem of linear programming states that if an LP has a finite optimum, then some optimum occurs at a vertex (extreme point) of the feasible polyhedron. This geometric fact underlies both the simplex method, which moves from vertex to adjacent vertex along edges of the polyhedron, and interior-point methods, which approach the optimal vertex through the interior.

This chapter is organized as follows. Section 16.2 motivates LP duality through a fully worked numerical example. Section 16.3 states and proves the three fundamental duality theorems (weak duality, strong duality, and complementary slackness). Section 16.4 discusses LP relaxations of integer programs and total unimodularity, the key property that makes LP relaxation tight for several combinatorial problems. Section 16.5 formulates five combinatorial problems — maximum bipartite matching, transportation, stable marriage, shortest path, and maximum flow / minimum cut — as LPs. Section 16.6 compares LP and LLP as paradigms.

16.2 A Worked Example of Linear Programming

We motivate LP duality through a small numerical example. Consider the primal LP

$$\begin{aligned} \text{minimize} \quad & 3x_1 + 5x_2 \\ \text{subject to} \quad & x_1 + x_2 \geq 4 \\ & x_1 + 3x_2 \geq 6 \\ & x_1,\, x_2 \geq 0. \end{aligned}$$

The decision variables are x_1, x_2. The primal feasible region is two-dimensional and is drawn in Fig. 16.1; the dual feasible region (also two-dimensional) is shown later, in Fig. 16.2.

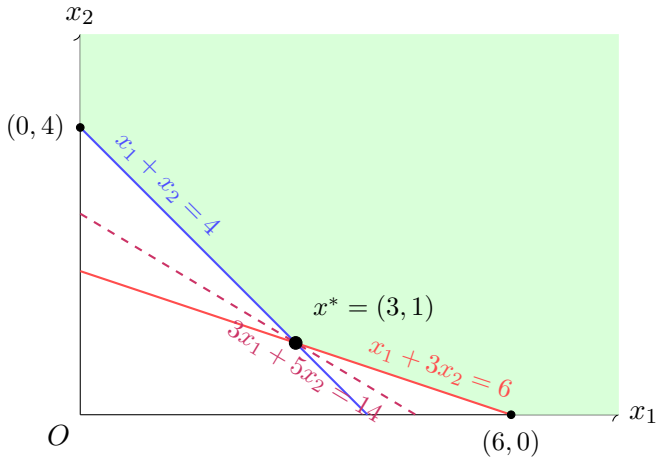


Figure 16.1: Feasible region of the primal LP, with constraints $x_1 + x_2 \geq 4$ and $x_1 + 3x_2 \geq 6$ (and $x_1, x_2 \geq 0$). The shaded region extends unboundedly to the upper right. The objective $3x_1 + 5x_2$ is minimized at the vertex $x^* = (3, 1)$, where both constraints are tight; the dashed line is the corresponding objective level set $3x_1 + 5x_2 = 14$.

The point $x = (4, 2)$ is feasible (substitute: $4 + 2 = 6 \geq 4$, $4 + 6 = 10 \geq 6$) and gives objective $12 + 10 = 22$. We seek an optimal value.

Bounding the primal. We already have an *upper* bound on the primal optimum, namely 22, from the feasible point $x = (4, 2)$. We would also like a *lower* bound, so we know whether the optimum is anywhere near 22 or much smaller.

Here is a simple idea. The two primal constraints

$$\begin{aligned} x_1 + x_2 &\geq 4, \\ x_1 + 3x_2 &\geq 6 \end{aligned}$$

hold for every feasible x. Pick any non-negative weights $y_1, y_2 \geq 0$, multiply the first constraint by y_1 and the second by y_2 (multiplying both sides of an inequality by a non-negative number preserves the direction), and add:

$$(y_1 + y_2)\, x_1 \;+\; (y_1 + 3y_2)\, x_2 \;\geq\; 4\, y_1 + 6\, y_2.$$

This combined inequality holds for every $(y_1, y_2) \geq 0$ and every primal-feasible x.

Now suppose we are lucky enough to pick (y_1, y_2) so that each coefficient on the left side is at most the corresponding primal objective coefficient $(3, 5)$:

$$y_1 + y_2 \;\leq\; 3, \qquad y_1 + 3y_2 \;\leq\; 5.$$

Then, because $x_1, x_2 \geq 0$, each term on the left of the combined inequality is dominated by the corresponding term of the primal objective, so

$$3x_1 + 5x_2 \;\geq\; (y_1 + y_2)\, x_1 + (y_1 + 3y_2)\, x_2 \;\geq\; 4\, y_1 + 6\, y_2.$$

The quantity $4y_1 + 6y_2$ is therefore a *lower bound* on the primal objective, valid for *every* primal-feasible x.

For example, $(y_1, y_2) = (1, 1)$ satisfies both coefficient constraints ($1{+}1 = 2 \leq 3$, $1{+}3 = 4 \leq 5$) and gives the lower bound $4 + 6 = 10$. The choice $(y_1, y_2) = (2, 1)$ also satisfies them ($2{+}1 = 3$, $2{+}3 = 5$, with both constraints now *tight*) and yields the stronger lower bound $8 + 6 = 14$.

The *tightest* lower bound obtainable this way is found by maximizing $4y_1 + 6y_2$ over all $(y_1, y_2) \geq 0$ satisfying the two coefficient constraints — and this maximization is precisely the *dual* LP.

The dual LP. The bounding logic above gives the dual program

$$\begin{aligned} \text{maximize} \quad & 4y_1 + 6y_2 \\ \text{subject to} \quad & y_1 + y_2 \;\leq\; 3 \\ & y_1 + 3y_2 \;\leq\; 5 \\ & y_1,\, y_2 \;\geq\; 0. \end{aligned}$$

The dual's feasible region is shown in Fig. 16.2.

The optimum is at the vertex $(y_1, y_2) = (2, 1)$, where both coefficient constraints are tight. The dual objective at $(2, 1)$ is $8 + 6 = 14$.

By the bounding argument, the primal optimum is at least 14. We can verify the primal optimum is exactly 14 at $x^* = (3, 1)$: substituting, $3 + 1 = 4$ (constraint 1 tight), $3 + 3 = 6$ (constraint 2 tight), and the objective is $3 \cdot 3 + 5 \cdot 1 = 9 + 5 = 14$.

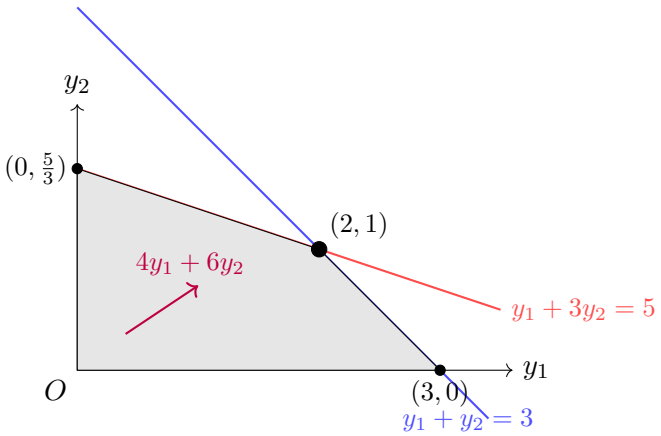


Figure 16.2: Dual feasible region for the worked example. The vertex $(2, 1)$ maximizes $4y_1+6y_2$ at value 14, attained where both constraints $y_1 + y_2 = 3$ and $y_1 + 3y_2 = 5$ are tight.

The primal optimum equals the dual optimum: both are 14. The dual variable $y_1 = 2 > 0$ corresponds to a tight primal constraint (constraint 1), and $y_2 = 1 > 0$ corresponds to a tight primal constraint (constraint 2); reciprocally, both primal variables $x_1 = 3 > 0$ and $x_2 = 1 > 0$ at the optimum correspond to tight dual constraints. These pairings illustrate *complementary slackness* (Theorem 16.5 below).

This example is no accident: every linear program in the form $\min c^T x$ subject to $Ax \geq b$, $x \geq 0$ has a dual $\max b^T y$ subject to $A^T y \leq c$, $y \geq 0$, and the two share the same optimum value. The next section makes this formal.

16.3 Duality in Linear Programming

We work with the primal-dual pair in standard form. The primal (P) is

$$\begin{aligned} \text{(P)} \quad & \text{minimize} && c^T x \\ & \text{subject to} && Ax \geq b \\ & && x \geq 0, \end{aligned}$$

and its dual (D) is

$$\begin{aligned} \text{(D)} \quad & \text{maximize} && b^T y \\ & \text{subject to} && A^T y \leq c \\ & && y \geq 0, \end{aligned}$$

where A is an $m \times n$ matrix, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$. The primal has n variables and m constraints; the dual has m variables and n constraints. Each primal constraint has a dual variable, and each primal variable has a dual constraint.

The correspondence between (P) and (D) is summarized in Table 16.1.

Primal (P)	**Dual (D)**
Minimization	Maximization
n variables $x_1, \dots, x_n$	m variables $y_1, \dots, y_m$
m inequality constraints $Ax \geq b$	n inequality constraints $A^T y \leq c$
Each variable x_j	Each constraint of (D)
Each constraint of (P)	Each variable y_i
Coefficient matrix A	Coefficient matrix A^T
Cost vector c	Right-hand side vector c
Right-hand side vector b	Cost vector b

Table 16.1: Primal–dual correspondence in standard form.

Weak Duality

Theorem 16.1 *If x is feasible for (P) and y is feasible for (D), then*

$$c^T x \;\geq\; b^T y.$$

Proof: We chain two inequalities connecting $c^T x$ to $b^T y$.

Step 1 (use dual feasibility and $x \geq 0$). Dual feasibility says $A^T y \leq c$ componentwise; that is, $(A^T y)_j \leq c_j$ for each j. Multiplying by $x_j \geq 0$ preserves the inequality, and summing over j gives

$$c^T x \;=\; \sum_j c_j\, x_j \;\geq\; \sum_j (A^T y)_j\, x_j \;=\; (A^T y)^T x \;=\; y^T A x.$$

Step 2 (use primal feasibility and $y \geq 0$). Primal feasibility says $Ax \geq b$ componentwise; that is, $(Ax)_i \geq b_i$ for each i. Multiplying by $y_i \geq 0$ preserves the inequality, and summing over i gives

$$y^T A x \;=\; \sum_i y_i\, (Ax)_i \;\geq\; \sum_i y_i\, b_i \;=\; y^T b \;=\; b^T y.$$

Chaining the two inequalities, $c^T x \geq y^T A x \geq b^T y$, which is the claim.

■

Example 16.2 For the primal LP from Section 16.2, take the primal-feasible point $x = (4, 2)$ with $c^T x = 3 \cdot 4 + 5 \cdot 2 = 22$. Take the dual-feasible point $y = (1, 1)$ (it satisfies $1 + 1 \leq 3$ and $1 + 3 \leq 5$) with $b^T y = 4 + 6 = 10$. Theorem 16.1 predicts $22 \geq 10$, a gap of 12. Tightening the dual to $y = (2, 1)$ raises the lower bound to $b^T y = 14$, so the primal optimum lies in the interval $[14, 22]$. Strong duality, treated next, says that the best primal value and the best dual value actually coincide — both are 14 in this example.

Weak duality yields three immediate consequences. First, every dual feasible y is a lower bound on the primal optimum, and every primal feasible x is an upper bound on the dual optimum. Second, if both LPs are feasible, both have finite optima. Third, if $c^T x = b^T y$ for a primal-feasible x and a dual-feasible y, then both are optimal (the gap closes).

Strong Duality

Weak duality tells us that every dual feasible y gives a lower bound on the primal optimum. Strong duality says something deeper: the *best* such lower bound is exactly equal to the primal optimum.

Geometrically, strong duality says that the supporting hyperplane defined by the optimal dual touches the primal feasible polyhedron precisely at the primal optimum, with no gap between the two values.

Theorem 16.3 *The primal (P) has a finite optimum if and only if the dual (D) has a finite optimum, and the two optima are equal:*

$$\min\{c^T x : Ax \geq b, x \geq 0\} \;=\; \max\{b^T y : A^T y \leq c, y \geq 0\}.$$

Example 16.4 For the LP from Section 16.2, the primal optimum is $x^* = (3, 1)$ with $c^T x^* = 3 \cdot 3 + 5 \cdot 1 = 14$, and the dual optimum is $y^* = (2, 1)$ with $b^T y^* = 4 \cdot 2 + 6 \cdot 1 = 14$. Theorem 16.3 asserts that these two values must agree — they do, and the weak-duality gap of 8 that we saw with the suboptimal pair $x = (4, 2)$, $y = (2, 1)$ in the previous example collapses to 0 at the primal–dual optimum.

Strong duality is the source of many classical min–max identities: max flow = min cut, max bipartite matching = min vertex cover, max antichain = min chain cover (Dilworth), among others.

Complementary Slackness Conditions

Theorem 16.5 *Let x be primal-feasible and y be dual-feasible. Then x and y are both optimal if and only if both complementary slackness conditions hold:*

1. *For each j: either $x_j = 0$ or $(A^T y)_j = c_j$ (i.e., the j-th dual constraint is tight).*
2. *For each i: either $y_i = 0$ or $(Ax)_i = b_i$ (i.e., the i-th primal constraint is tight).*

Proof: By weak duality, $c^T x \geq y^T A x \geq b^T y$, where the first inequality uses $A^T y \leq c$ and $x \geq 0$, and the second uses $Ax \geq b$ and $y \geq 0$. Optimality of both x and y is equivalent (by strong duality) to $c^T x = b^T y$, which forces equality in both inequalities. Equality in the first means $\sum_j (c_j - (A^T y)_j) x_j = 0$; with $c_j - (A^T y)_j \geq 0$ and $x_j \geq 0$, each summand is zero, giving condition 1. Equality in the second analogously gives condition 2.

■

Example 16.6 On the LP from Section 16.2 with primal–dual optimum $x^* = (3, 1)$ and $y^* = (2, 1)$, we verify all four complementary-slackness pairings.

Condition 1 (primal variables vs. dual constraints).

- $j = 1$: $x_1^* = 3 \neq 0$, so the first dual constraint must be tight. Indeed $(A^T y^*)_1 = 1 \cdot 2 + 1 \cdot 1 = 3 = c_1$.
- $j = 2$: $x_2^* = 1 \neq 0$, so the second dual constraint must be tight. Indeed $(A^T y^*)_2 = 1 \cdot 2 + 3 \cdot 1 = 5 = c_2$.

Condition 2 (dual variables vs. primal constraints).

- $i = 1$: $y_1^* = 2 \neq 0$, so the first primal constraint must be tight. Indeed $(Ax^*)_1 = 3 + 1 = 4 = b_1$.
- $i = 2$: $y_2^* = 1 \neq 0$, so the second primal constraint must be tight. Indeed $(Ax^*)_2 = 3 + 3 \cdot 1 = 6 = b_2$.

All four pairings hold, certifying joint optimality of (x^*, y^*) without re-solving either LP.

Recovering the dual from the primal. The previous example *verified* CS on a precomputed primal–dual pair. The more common use of CS is the converse: given an optimal primal x^*, recover an optimal dual y^* without re-solving the dual. On the same LP, both $x_1^*, x_2^* > 0$, so condition 1 forces both dual constraints to be tight: $y_1 + y_2 = 3$ and $y_1 + 3y_2 = 5$. This 2×2 system has the unique solution $y_1 = 2$, $y_2 = 1$, which is dual-feasible and matches $b^T y = 4 \cdot 2 + 6 \cdot 1 = 14 = c^T x^*$. In general, complementary slackness reduces dual recovery to solving a linear system whose size is the number of *strictly positive* primal variables — often much smaller than the dual itself. This is the most common practical use of duality.

16.4 LP Relaxation and Total Unimodularity

Many combinatorial problems are naturally integer linear programs (ILPs): the variables x_{ij} encode discrete choices ($\{0, 1\}$ for "include this edge" or "assign this item to that bidder") and the constraints are linear. ILPs are NP-hard in general. The standard relaxation drops integrality, replacing $x_{ij} \in \{0, 1\}$ with the inequality $x_{ij} \geq 0$. The relaxed LP can be solved in polynomial time, but its optimum may not be integral, in which case the LP relaxation gives only a bound on the ILP optimum. The remarkable fact, exploited throughout this chapter, is that for several combinatorial problems the LP optimum *is* integral and matches the ILP optimum exactly.

Definition 16.7 (Total unimodularity) *An integer matrix A is* totally unimodular *(TU) if every square submatrix has determinant in $\{-1, 0, +1\}$.*

Theorem 16.8 (Hoffman–Kruskal) *If A is TU and b is an integer vector, then every vertex of the polyhedron $\{x : Ax \geq b, x \geq 0\}$ has integer coordinates. Hence the LP optimum, attained at a vertex, is integral.*

Proof: Let v be a vertex of $P = \{x : Ax \geq b,\, x \geq 0\}$. Vertices are *basic feasible solutions*: at v, some n linearly independent constraints from $\begin{pmatrix} A \\ -I \end{pmatrix} x \geq \begin{pmatrix} b \\ 0 \end{pmatrix}$ are tight. Let $\hat{A}$ be the corresponding $n \times n$ submatrix and $\hat{b}$ the corresponding right-hand-side vector (its entries are drawn from $\{b_i\} \cup \{0\}$, all integers). Then $\hat{A}\, v = \hat{b}$.

Adjoining the identity block $-I$ to A preserves total unimodularity, so $\hat{A}$ is a square submatrix of a TU matrix. Since $\hat{A}$ is non-singular, $\det \hat{A} \in \{-1, +1\}$. The inverse of an integer matrix with determinant ± 1 is itself an integer matrix, so $\hat{A}^{-1}$ has integer entries. Hence $v = \hat{A}^{-1}\, \hat{b}$ is integer.

■

Example 16.9 Let G_1 be the path $u - v - w$ with edges $e_1 = \{u, v\}$ and $e_2 = \{v, w\}$. Its vertex–edge incidence matrix (rows indexed by u, v, w; columns by e_1, e_2) is

$$A_1 \;=\; \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

Every 1×1 submatrix is 0 or 1, and the three 2×2 submatrices have determinants

$$\det \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} = 1, \quad \det \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = 1, \quad \det \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = 1,$$

all in $\{-1, 0, +1\}$, so A_1 is totally unimodular. Hoffman–Kruskal then guarantees that the polyhedron $\{x : A_1 x \geq b,\, x \geq 0\}$ has integer vertices for any integer b — and the LP optimum is automatically integral.

In contrast, let G_2 be the triangle on vertices $\{1, 2, 3\}$ with edges e_{12}, e_{13}, e_{23}. Its incidence matrix is

$$A_2 \;=\; \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix},$$

and $\det A_2 = -2 \notin \{-1, 0, +1\}$, so A_2 is *not* TU. The difference is that G_1 is bipartite while G_2 contains an odd cycle: the vertex–edge incidence matrix of an undirected graph is TU if and only if the graph is bipartite. This is precisely why bipartite matching admits a tight LP relaxation while non-bipartite matching does not.

The standard examples of TU matrices include: (i) the incidence matrix of a bipartite graph (rows indexed by vertices, columns by edges, entry 1 if the edge is incident to the vertex), (ii) the node-arc incidence matrix of a directed graph, (iii) interval matrices (rows are

characteristic vectors of intervals on a line), and (iv) network matrices in general. Each yields a class of combinatorial problems whose LP relaxations are tight: bipartite matching, network flow, shortest paths, transportation, and assignment.

LP rounding for approximation

When the LP relaxation is *not* tight, the fractional optimum is only a bound, and an integer solution must be recovered by *rounding*. The classical example is *vertex cover*: given an undirected graph $G = (V, E)$ with non-negative vertex weights w_v, find a minimum-weight vertex set $C \subseteq V$ that covers every edge. The natural ILP is

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} w_v \, x_v \\ \text{subject to} \quad & x_u + x_v \;\geq\; 1 \quad \text{for each } (u, v) \in E, \\ & x_v \in \{0, 1\}. \end{aligned}$$

The constraint matrix here is the edge–vertex incidence matrix. In bipartite graphs it is TU and the LP relaxation is integral, recovering König's theorem. In general graphs the matrix is not TU — a triangle yields the half-integral optimum $x_v = 1/2$ for all three vertices — but the half-integrality of vertex covers is itself a structural fact: every vertex of the LP relaxation polytope has $x_v \in \{0, 1/2, 1\}$.

The *LP-rounding* approximation algorithm exploits this: solve the LP relaxation in polynomial time, then round any $x_v \geq 1/2$ up to 1 and any $x_v < 1/2$ down to 0. The rounded set C is a feasible vertex cover (every edge has at least one endpoint with $x_v \geq 1/2$, by the LP constraint $x_u + x_v \geq 1$), and its cost is at most *twice* the LP optimum — hence at most twice the integer optimum. This is the canonical 2-approximation for vertex cover.

The same template — solve the relaxation, round the fractional solution, bound the rounding loss — gives polynomial-time approximation algorithms for many NP-hard problems. A comprehensive treatment is given by Vazirani [Vaz01] and by Williamson and Shmoys [WS11].

16.5 LP Formulations of Combinatorial Problems

Many combinatorial optimization problems can be expressed naturally as linear programs by introducing variables that represent discrete choices, encoding the combinatorial constraints as linear inequalities, and optimizing a linear objective. The five formulations below illustrate the recurring patterns: matching-type (bipartite matching), flow-type (transportation), assignment-type (stable marriage), potential-based (shortest path), and capacitated-flow type (maximum flow / minimum cut).

16.5.1 Maximum Bipartite Matching

Let $G = (A \cup B, E)$ be a bipartite graph with edge weights w_{ij} for $(i,j) \in E$, $i \in A$, $j \in B$. Let $x_{ij} \in \{0,1\}$ indicate whether edge (i,j) is in the matching. The integer program is

$$\begin{aligned} \text{maximize} \quad & \sum_{(i,j)\in E} w_{ij}\, x_{ij} \\ \text{subject to} \quad & \textstyle\sum_j x_{ij} \;\le\; 1 \quad \text{for each } i \in A, \\ & \textstyle\sum_i x_{ij} \;\le\; 1 \quad \text{for each } j \in B, \\ & x_{ij} \in \{0,1\}. \end{aligned}$$

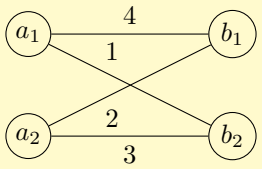
The LP relaxation drops $x_{ij} \in \{0,1\}$ to $x_{ij} \ge 0$. The constraint matrix is the bipartite vertex-edge incidence matrix, which is TU (Section 16.4); by Theorem 16.8 the LP relaxation has an integral optimum, and the LP optimum equals the integer-program optimum. With $|A| = |B| = n$ the LP has n^2 variables and $2n$ constraints.

The LP dual introduces variables u_i for each $i \in A$ and v_j for each $j \in B$:

$$\begin{aligned} \text{minimize} \quad & \sum_i u_i \;+\; \sum_j v_j \\ \text{subject to} \quad & u_i + v_j \;\ge\; w_{ij} \quad \text{for all } (i,j) \in E, \\ & u_i,\, v_j \;\ge\; 0. \end{aligned}$$

The dual is exactly the LP relaxation of the minimum-weighted-vertex-cover problem on G when weights are non-negative; strong duality recovers König's theorem (max bipartite matching = min vertex cover) for unweighted graphs.

Example 16.10 Consider the bipartite graph with $A = \{a_1, a_2\}$, $B = \{b_1, b_2\}$, complete bipartite, with edge weights as shown:



The matching LP (with x_{ij} indexed by worker i and job j) is

$$\begin{aligned} \text{maximize} \quad & 4x_{11} + x_{12} + 2x_{21} + 3x_{22} \\ \text{subject to} \quad & x_{11} + x_{12} \le 1 \quad (a_1) \\ & x_{21} + x_{22} \le 1 \quad (a_2) \\ & x_{11} + x_{21} \le 1 \quad (b_1) \\ & x_{12} + x_{22} \le 1 \quad (b_2) \\ & x_{ij} \ge 0. \end{aligned}$$

There are two perfect matchings: $\{(a_1, b_1), (a_2, b_2)\}$ with weight $4+3 = 7$, and $\{(a_1, b_2), (a_2, b_1)\}$ with weight $1 + 2 = 3$. The LP optimum, attained at the vertex $(x_{11}, x_{12}, x_{21}, x_{22}) = (1, 0, 0, 1)$, has value 7 — equal to the integer optimum. The constraint matrix is TU, so by Theorem 16.8 integrality is automatic.

The dual cover LP has variables $u_1, u_2, v_1, v_2 \ge 0$, minimizes $u_1 + u_2 + v_1 + v_2$, and requires $u_i + v_j \ge w_{ij}$ on each edge. The choice $u_1 = 4$, $u_2 = 3$, $v_1 = v_2 = 0$ is feasible (all four edge inequalities hold: $4 \ge 4$, $4 \ge 1$, $3 \ge 2$, $3 \ge 3$) and has objective $4 + 3 = 7$, matching the primal optimum and certifying joint optimality via strong duality.

16.5.2 The Transportation Problem

We have n supply nodes and n demand nodes. The supply at node i is given by a_i and the demand at node j is given by b_j. We assume that the total supply is equal to the total demand. If the supply and demand do not match, we can create fake nodes to ensure the equality.

Let w_{ij} be the cost function when an item is shipped from i to j, and let x_{ij} be the number of items shipped from supply node i to demand node j. The transportation LP is

$$\begin{aligned} \text{minimize} \quad & \sum_{i,j} w_{ij}\, x_{ij} \\ \text{subject to} \quad & \textstyle\sum_j x_{ij} = a_i \quad \text{for all } i, \\ & \textstyle\sum_i x_{ij} = b_j \quad \text{for all } j, \\ & x_{ij} \ge 0. \end{aligned}$$

The dual is

$$\begin{aligned} \text{maximize} \quad & \sum_i a_i u_i \;+\; \sum_j b_j v_j \\ \text{subject to} \quad & u_i + v_j \;\le\; w_{ij} \quad \text{for all } i, j, \\ & u_i,\, v_j \text{ unrestricted.} \end{aligned}$$

16.5.3 The Stable Marriage Problem

Let $x_{ij} \in \{0, 1\}$ indicate whether man i is matched to woman j. The matching must satisfy the assignment constraints (each person matched to exactly one partner) and the no-blocking-pair constraint: for any pair (i, j), if j is matched to a man whom she ranks below i, then i must be matched to a woman whom he ranks above j. Let $Q(j, i)$ be the set of men less preferred than i in j's preference list, and let $R(i, j)$ be the set of women more preferred than j in i's list. The integer program is

$$\begin{aligned} \text{minimize} \quad & \sum_{ij} \mathit{rank}_i(j)\, x_{ij} \\ \text{subject to} \quad & \textstyle\sum_j x_{ij} \;=\; 1 \quad \text{for all } i \text{ (assignment)} \\ & \textstyle\sum_i x_{ij} \;=\; 1 \quad \text{for all } j \text{ (assignment)} \\ & \textstyle\sum_{i' \in Q(j,i)} x_{i',j} \;-\; \sum_{j' \in R(i,j)} x_{i,j'} \;\le\; 0 \quad \text{for all } (i, j) \\ & x_{ij} \in \{0, 1\}. \end{aligned}$$

The expression $\sum_{i' \in Q(j,i)} x_{i',j}$ equals 1 iff j is currently matched below i, and $\sum_{j' \in R(i,j)} x_{i,j'}$ equals 1 iff i is matched above j, so the third constraint encodes the absence of a blocking pair at (i, j). The no-blocking-pair group adds $O(n^2)$ constraints to the $2n$ assignment constraints, so the LP has n^2 variables and $O(n^2)$ constraints.

The constraint matrix is *not* totally unimodular, so Theorem 16.8 does not apply directly. Nonetheless, Vande Vate [VV89] proved that the LP relaxation has integral optima: the convex hull of stable matchings is exactly the polytope cut out by these inequalities. Choosing different objective functions selects different stable matchings; for example, minimizing $\sum_{ij} \mathit{rank}_i(j)\, x_{ij}$ (the sum of each man's rank for his partner) yields the stable matching that minimizes total men's rank.

Example 16.11 Consider two men m_1, m_2 and two women w_1, w_2 with preference lists

$$m_1 : w_1 \succ w_2, \qquad m_2 : w_1 \succ w_2,$$

$$w_1 : m_2 \succ m_1, \qquad w_2 : m_1 \succ m_2.$$

The matching $\{(m_1, w_1), (m_2, w_2)\}$ is unstable because (m_2, w_1) is a blocking pair: m_2 prefers w_1 over his current partner w_2, and w_1 prefers m_2 over her current partner m_1. The LP inequality at $(i, j) = (m_2, w_1)$ encodes precisely this obstruction — the sum $\sum_{i' \in Q(w_1, m_2)} x_{i', w_1}$ is 1 (since m_1 is matched to w_1 and $m_1 \in Q(w_1, m_2)$), while $\sum_{j' \in R(m_2, w_1)} x_{m_2, j'}$ is 0 (since m_2 has no partner ranked above w_1), so the constraint $1 - 0 \leq 0$ is violated.

16.5.4 The Shortest Path Problem

Shortest path admits two natural LP formulations that are LP duals of each other: a *potential* formulation whose variables are vertex labels, and a *flow* formulation whose variables are edge weights. We present them as a primal-dual pair.

Let $G = (V, E)$ be a directed graph with edge weights w_{ij} (assumed such that no negative-weight cycle is reachable from the source s). Let t be the target. Assume no incoming edge to s and no outgoing edge from t.

Primal (potentials). Let x_i be an unrestricted real variable representing a tentative distance from s to i. The shortest s-t distance is the maximum value of $x_t - x_s$ subject to the relaxation inequalities $x_j \leq x_i + w_{ij}$ for every edge.

$$\begin{aligned} \text{maximize} \quad & x_t - x_s \\ \text{subject to} \quad & x_j - x_i \;\leq\; w_{ij} \quad \text{for all } (i, j) \in E, \\ & x_i \text{ unrestricted for all } i. \end{aligned}$$

The variables are unrestricted because distance potentials are defined only up to an additive constant; one may set $x_s = 0$ without loss of generality. This LP has n variables (one per vertex) and m constraints (one per edge). The optimal x_i^* is the shortest distance from s to i. This is exactly the predicate that LLP-BellmanFord searches.

Dual (unit flow). Each primal edge constraint introduces a dual variable $y_{ij} \geq 0$, interpretable as the flow on edge (i, j). The dual is

$$\begin{aligned} \text{minimize} \quad & \sum_{(i,j) \in E} w_{ij}\, y_{ij} \\ \text{subject to} \quad & \textstyle\sum_{j : (s,j) \in E} y_{sj} \;=\; 1 \quad \text{(unit flow out of } s\text{)} \\ & \textstyle\sum_{i : (i,t) \in E} y_{it} \;=\; 1 \quad \text{(unit flow into } t\text{)} \\ & \textstyle\sum_{i : (i,k) \in E} y_{ik} - \sum_{j : (k,j) \in E} y_{kj} \;=\; 0 \quad \text{for } k \in V \setminus \{s, t\} \\ & y_{ij} \;\geq\; 0 \quad \text{for all } (i, j) \in E. \end{aligned}$$

The dual minimizes the total cost of routing one unit of flow from s to t subject to flow conservation at every other vertex. Its optimum is achieved at a vertex of the polytope, which (because the constraint matrix is the node-arc incidence matrix of the directed graph and is therefore TU) is integral: a single s-t path supplied with one unit of flow. The cost of that path is the shortest s-t distance.

By strong duality, the two optima coincide. Reading off the result: the shortest s-t distance can be computed either via vertex potentials (which is what LLP-BellmanFord does) or via min-cost unit flow (a special case of the network-flow algorithms developed in the next chapter).

Complementary slackness ties the two viewpoints together. If a dual variable $y_{ij} > 0$, then the corresponding primal constraint must be tight: $x_j = x_i + w_{ij}$. In words, every edge that carries flow lies on a shortest path.

16.5.5 Maximum Flow and Minimum Cut

Let $G = (V, E)$ be a directed graph with edge capacities $c_{ij} \geq 0$, source s, and sink t. We assume no incoming edge to s and no outgoing edge from t; this is without loss of generality, since any such edges can be removed without affecting the maximum flow value. The maximum-s-t-flow problem admits a clean LP formulation: let x_{ij} be the flow on edge (i, j), and maximize the total flow out of s subject to conservation at every internal vertex and capacity at every edge.

Primal (max flow).

$$\begin{aligned} \text{maximize} \quad & \textstyle\sum_{j:(s,j)\in E} x_{sj} \\ \text{subject to} \quad & \textstyle\sum_{i:(i,k)\in E} x_{ik} \;-\; \sum_{j:(k,j)\in E} x_{kj} \;=\; 0 \quad \text{for } k \in V \setminus \{s, t\}, \\ & 0 \;\leq\; x_{ij} \;\leq\; c_{ij} \quad \text{for all } (i, j) \in E. \end{aligned}$$

Dual (min cut). The capacity constraint $x_{ij} \leq c_{ij}$ contributes a dual variable $z_{ij} \geq 0$ for each edge, and the conservation constraint contributes a free dual variable u_k for each internal vertex (set $u_s = 1, u_t = 0$). After eliminating the free variables, the dual reduces to

$$\begin{aligned} \text{minimize} \quad & \sum_{(i,j)\in E} c_{ij}\, z_{ij} \\ \text{subject to} \quad & u_i - u_j \;\leq\; z_{ij} \quad \text{for all } (i, j) \in E, \\ & u_s = 1,\; u_t = 0, \\ & z_{ij} \;\geq\; 0. \end{aligned}$$

The constraint matrix is again the node-arc incidence matrix of G and is therefore TU, so by Theorem 16.8 the dual optimum is attained at a 0/1 vertex. An integral dual optimum partitions V into $S = \{v : u_v = 1\}$ and $\bar{S} = V \setminus S$ with $s \in S, t \in \bar{S}$, and $z_{ij} = 1$ exactly on the *forward edges* of the cut $(S, \bar{S})$. The dual objective is $\sum_{(i,j)\in E,\, i\in S,\, j\in\bar{S}} c_{ij}$, i.e., the capacity of the cut.

By strong duality, the maximum-flow value equals the minimum-cut capacity — the celebrated max-flow / min-cut theorem. The combinatorial structure of flows and cuts (augmenting paths, residual graphs) is developed in Chapter 11; the LP perspective above gives the cleanest proof that the two quantities coincide.

16.6 Comparison with LLP

Linear programming and lattice-linear predicate detection are both frameworks for searching for an optimal feasible solution, with complementary strengths.

LP's strengths. LP is the universal hammer: any objective minimized over any polyhedron, including with mixed-integer or 0–1 variables, equality constraints, free variables (no non-negativity), and variable bounds, fits into the LP/ILP framework. Standard LP solvers (Simplex Method, Interior Point Methods, Ellipsoid Method, Cutting Plane Method, Branch and Bound Method; see [Dan63, Kar84, Kha79, Sch86, Van20]) handle any of these. LP duality is itself a powerful tool: it provides certificates of optimality, reduces min–max identities to algebraic transposition, and underlies sensitivity analysis. The chapter's bipartite-matching, transportation, stable-marriage, and shortest-path examples are all instances of *combinatorial* problems whose LP relaxations happen to have integral optima — but the LP framework also accommodates problems with no special integrality structure, where one settles for the LP bound or applies branch-and-bound.

LLP's strengths. The LLP framework requires the feasibility predicate to be *lattice-linear*: closed under componentwise meets in a distributive lattice. This is a real restriction — generic polyhedra do not have this property — but in return, LLP yields strongly polynomial algorithms for the problems it covers. The advance operation simply lifts components of the state vector toward the least fixpoint, with no need for a feasibility-preserving search through a polyhedron's vertices.

Where they meet, where they diverge. The feasibility predicates of the LP-relaxation-tight problems treated in this chapter are all lattice-linear (or close to it after a coordinate transform), so LLP and LP both apply. On these problems the LLP algorithm is at least as fast as the strongly polynomial LP-based algorithm; on bipartite matching, shortest paths, and the assignment problem the LLP algorithm matches the best known bounds. On the other hand, LLP cannot handle, e.g., a generic LP $\min c^T x$ subject to $Ax \geq b$ with no lattice structure on the feasible space. The right reading is: LP for generality, LLP for efficiency on structured problems.

A summary of the differences:

- **Underlying space.** LP: real vector space. LLP: finite distributive lattice.
- **Feasible region.** LP: any polyhedron. LLP: meet-closed (lattice-linear) predicates.
- **Objective.** LP: linear cost function $c^T x$. LLP: the lattice's own order (find the infimum or supremum of the feasible set).
- **Algorithms.** LP: simplex (worst-case exponential, fast in practice; [Dan63]), interior point (polynomial; [Kar84]), ellipsoid (polynomial; [Kha79]). LLP: forbidden-index detection plus parallel advance, strongly polynomial whenever applicable.
- **Strong polynomiality.** LP: open in general. LLP: yes for lattice-linear predicates.

- **Parallelism.** LP: limited (interior-point steps parallelize; simplex pivots are inherently sequential). LLP: every forbidden index can advance concurrently, with worst-case parallel time bounded by the lattice height.

16.7 Summary

This chapter introduced linear programming, developed LP duality through a worked example and three formal theorems (weak duality, strong duality, complementary slackness), discussed total unimodularity as the bridge between integer and linear programming, and formulated five combinatorial optimization problems as LPs (maximum bipartite matching, transportation, stable marriage, shortest path, and maximum flow / minimum cut). The chapter closed by comparing LP and the lattice-linear predicate approach as paradigms.

Method	**Worst-case complexity**	**Reference**
Simplex	Exponential	[Dan63, KM72]
Ellipsoid	Polynomial	[Kha79]
Interior point	Polynomial	[Kar84]
LLP detection	Strongly polynomial (when applicable)	Earlier chapters

16.8 Problems

1. **(Graphical LP Method)** A company produces two products, X and Y, with profits of \$3 and \$4 per unit, respectively. Production requires labor and materials, with constraints:

   - Labor: $2X + Y \leq 8$ hours.
   - Materials: $X + 2Y \leq 8$ units.
   - Non-negativity: $X, Y \geq 0$.

   Formulate and solve the linear program to maximize profit, using the graphical method.

2. **(Three Product Simplex)** A factory produces three items, A, B, and C, with profits \$5, \$3, and \$4 per unit. Constraints are:

   - Machine hours: $3A + B + 2C \leq 12$.
   - Labor hours: $A + 2B + C \leq 10$.
   - Non-negativity: $A, B, C \geq 0$.

   Formulate and solve using the simplex method to maximize profit.

3. **(Max Flow Linear Program)** Formulate the max-flow problem for the network below as a linear program and solve it.

4. **(Dual Derivation Bound)** *Dual derivation and verification.* Consider the primal LP

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

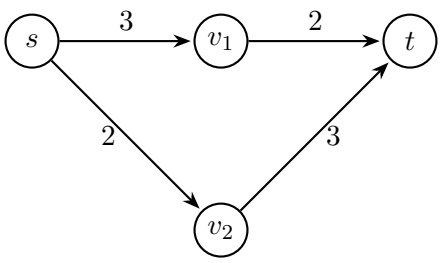


Figure 16.3: Flow network for problem 3.

Write the dual program. Find the optimal dual solution, and use complementary slackness to recover the optimal primal solution. Verify that both objective values agree, confirming strong duality.

5. *Total unimodularity.* Consider the bipartite graph $K_{2,3}$ with vertex partition $A = \{a_1, a_2\}$, $B = \{b_1, b_2, b_3\}$ and all six edges. Write the 5×6 vertex-edge incidence matrix M (rows: vertices; columns: edges; entry 1 if the edge is incident to the vertex). Verify directly that every 1×1, 2×2, and 3×3 submatrix has determinant in $\{-1, 0, +1\}$. Conclude that the LP relaxation of maximum bipartite matching on $K_{2,3}$ has an integer optimum, by Theorem 16.8.

6. **(LP Complementary Slackness)** *Complementary slackness.* For the LP

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

a colleague claims that $x^* = (3, 1)$ is optimal, with witness dual $y^* = (3/2, 1/2)$. Verify primal feasibility, dual feasibility, and the two complementary slackness conditions. State the optimum value of the LP.

7. **(Vertex Cover LP Duality)** *Vertex cover by LP duality.* Let $G = (V, E)$ be a graph with non-negative vertex weights w_v. The minimum-weight vertex cover problem asks for a subset $S \subseteq V$ minimizing $\sum_{v \in S} w_v$ such that every edge has at least one endpoint in S. Write the LP relaxation of this 0/1 program. Show that the dual of this LP is a fractional matching LP. State the strong duality consequence: when G is bipartite, this recovers König's theorem (max matching = min vertex cover for unweighted graphs).

8. **(Lattice-Linear vs Polyhedral)** *Lattice-linearity versus polyhedrality.* For each of the following predicates on $G \in \mathbb{Z}^n_{\geq 0}$, decide whether it is (a) lattice-linear, (b) describable as a conjunction of half-spaces (i.e., a polyhedron), (c) both, or (d) neither. Justify each answer.

   (P1) $\forall i :\ G[i] \geq 0$.

   (P2) $\forall i :\ G[i] \leq u_i$ for some fixed vector $u \in \mathbb{Z}^n_{\geq 0}$.

   (P3) $\forall (i, j) \in E :\ G[j] \leq G[i] + w_{ij}$ on a directed graph (V, E) with non-negative weights.

   (P4) $G[1] + G[2] \geq 1$.

   (P5) $\max_i G[i] \leq k$ for some fixed k.

9. **(Shortest Path LP and LLP)** *Shortest path: LP and LLP on the same instance.* Consider the directed graph $V = \{s, a, b, t\}$ with edges (s, a) weight 3, (s, b) weight 5, (a, b) weight 1, (a, t) weight 7, (b, t) weight 2.

   (a) Write the LP primal of Section 16.5.4 (potentials x_v, $\max x_t - x_s$ subject to $x_v - x_u \le w_{uv}$). Solve it.

   (b) Write the LLP-BellmanFord predicate and trace the LLP loop from $G = (0, \infty, \infty, \infty)$ to its least fixpoint.

   (c) Verify the LP optimum and the LLP fixpoint give the same shortest distance.

10. **(LP Beyond LLP Reach)** *A constraint LP handles but LLP does not directly.* Consider the LP

$$\min \; -x_1 - x_2 \quad \text{s.t.} \quad x_1 + x_2 \le 5, \quad x_1, x_2 \ge 0.$$

    (a) Solve the LP. What is the optimal vertex?

    (b) The constraint $x_1 + x_2 \le 5$ is a half-space. Show that as a predicate on $\mathbb{Z}^2_{\ge 0}$ it is *not* lattice-linear in the LLP sense (no useful least-fixpoint structure).

    (c) Why does the LLP framework not apply directly to this LP? What modification would be needed?

11. **(LP Duality LLP Certificate)** *LP duality as a certificate for LLP output.* Take the LLP-Assignment output on the 3×3 instance from the assignment chapter (valuation matrix with rows $b_1 = (10, 8, 6)$, $b_2 = (9, 7, 4)$, $b_3 = (8, 6, 5)$). The LLP algorithm returns the minimum market clearing price $G^* = (3, 1, 0)$ and matching $b_1 \to h_1$, $b_2 \to h_2$, $b_3 \to h_3$. Construct LP primal and dual variables that certify optimality of this matching, and verify both feasibility conditions and complementary slackness.

12. **(Triangle Integrality Gap)** *Integrality gap on a triangle.* Consider the unweighted maximum-matching LP on the triangle K_3 with vertices $\{1, 2, 3\}$ and three edges. Write the LP relaxation. Show that $x_e = 1/2$ for every edge is feasible with objective $3/2$. Show that the maximum $0/1$ matching has size 1. Hence the integrality gap on K_3 is at least $3/2$. Why does this not contradict Theorem 16.8?

13. **(Incremental LP)** The polynomial-time LP $\min c^T x$ s.t. $Ax \ge b$, $x \ge 0$ has been solved with optimum x^*. A new constraint $a_{m+1}^T x \ge b_{m+1}$ is appended. Give an algorithm to update x^* to the new optimum x^{**} in time substantially less than re-solving from scratch.

14. **(Dynamic Objective)** The constraint set $\{Ax \ge b, x \ge 0\}$ is fixed; the objective c changes adversarially over time. Maintain the optimum vertex $x^*(c)$ as c changes, with amortised work per change much smaller than re-solving.

15. **(Constrained LP With Forced Variables)** A set F of variables must be at their integer bounds $x_j = u_j$ (in $\{0, 1\}$ for binary problems). The relaxation may have a non-integer optimum; you must enforce the side constraints. Give a polynomial-time algorithm when the constraint matrix is totally unimodular, and discuss the case when it is not.

16. **(LP With Ties / Degeneracy)** At an optimal vertex of the LP, several constraints may be tight — the basis is *degenerate*. Simplex pivots may cycle without progress. Describe a tie-breaking rule that guarantees finite termination.

16.9 Bibliographic Remarks

The origins of linear programming go back to Kantorovich [Kan39] on the Soviet side and to von Neumann's duality work in game theory [vN47] on the American side. Dantzig [Dan63] introduced the simplex method, the dominant LP algorithm in practice. Khachiyan [Kha79] gave the first polynomial-time algorithm for LP via the ellipsoid method, settling LP's complexity classification; Karmarkar [Kar84] introduced the interior-point method, which is faster in practice on large instances. Bland [Bla77] gave anti-cycling rules that make simplex terminate; Klee and Minty [KM72] constructed the cube example showing simplex can take exponentially many pivots in the worst case.

The integer-programming side rests on Hoffman and Kruskal's theorem [HK56] that total unimodularity implies integral LP optima, Birkhoff's theorem on doubly-stochastic matrices [Bir46] which yields the bipartite-matching polytope as a special case, and Edmonds's matching polytope [Edm65] (with blossom inequalities) for the non-bipartite case. Vande Vate [VV89] characterized the stable-marriage polytope. Schrijver's two-volume monograph [Sch86, Sch03] is the canonical reference for both LP theory and its combinatorial-optimization applications. Vanderbei [Van20] is a modern textbook treatment.

Chapter 17

The Assignment Problem

17.1 Introduction

The assignment problem arises whenever a set of indivisible resources must be matched one-to-one with a set of agents who value them differently. Sponsored-search ad slots are auctioned to advertisers; donor kidneys must be matched with compatible recipients; ride-hailing platforms dispatch drivers to riders; faculty are assigned to courses; and GPU jobs are mapped onto cluster nodes. In each case, every (resource, agent) pair carries a numeric value, and the goal is to choose a one-to-one assignment that maximizes the total.

In this chapter, we consider the assignment problem in which we seek an assignment of items to bidders such that the total payoff is maximized. We view items and bidders as nodes in a bipartite graph. The weight $v_{b,i}$ of an edge between the bidder b and the item i is the utility of assigning the item i to the bidder b. The assignment problem then corresponds to finding the maximum weight bipartite matching.

The assignment problem has a rich history. The most famous algorithm for this problem is due to the American scientist Kuhn, who called it the Hungarian algorithm because it is inspired by the earlier works of two Hungarian mathematicians, König and Egerváry. Another American mathematician, Munkres, observed that the algorithm is strongly polynomial, and the algorithm is also known as the Kuhn-Munkres algorithm. It was later discovered that the problem had previously been solved by Jacobi in the 19th century and was published in Latin in the year 1890.

We approach the assignment problem through *prices* rather than through augmenting paths. The Hungarian and Kuhn–Munkres algorithms maintain a partial matching and grow it via alternating-tree searches. Ascending-price (auction) algorithms, introduced by Demange, Gale, and Sotomayor [DGS86], instead maintain a price vector and increase it monotonically until the demand graph admits a perfect matching. The price-ascending view fits the LLP framework directly. The LLP algorithm generalizes naturally to lattice-linear side constraints, which is what the chapter exploits in Section 17.4. The augmenting-path Hungarian algorithm achieves the same $O(n^3)$ strongly-polynomial bound.

This chapter is organized as follows. Section 17.2 formulates the assignment problem as

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

a maximum weight bipartite matching. Section 17.3 introduces the market clearing price predicate, proves its lattice-linearity, establishes existence of a clearing price, and develops the LLP-Assignment algorithm, which advances each forbidden item's price by the slack-jump — the largest LLP-correct advance — achieving a strongly polynomial $O(n^3)$ bound. A worked 3×3 example traces the algorithm explicitly. Section 17.4 generalizes the predicate to incorporate lattice-linear constraints on the prices, and traces the constrained algorithm on the running example. Section 17.5 discusses certificates of optimality for both the unconstrained and constrained variants. Section 17.6 concludes with a complexity table.

17.2 Problem Formulation

Let I be a set of n indivisible items and U, a set of n bidders. Each item $i \in I$ receives a valuation $v_{b,i}$ from each bidder $b \in U$. The valuation of any item i is an integer between 0 and T. Each item i has a price $G[i]$ which is also an integer between 0 and T.

We will assume that the number of items is equal to the number of bidders. If one of the sets is smaller, say there are fewer items than bidders, then we can add virtual items such that all bidders give a valuation of zero to those items. We also note that we are considering the maximum weight bipartite matching. By subtracting the valuation from the maximum valuation for each edge and calling it the cost of that edge, we can transform this problem to minimum-cost bipartite matching.

It is important to note that it is the relative valuation of the items that is important, not the absolute values. Suppose that there are three items and a bidder provides a valuation of $(10, 8, 4)$ for these items, then the assigned matching will not change if his valuation is $(6, 4, 0)$ instead. The weight of the matching found for the valuation $(6, 4, 0)$ would be exactly less by 4 because in each assignment a bidder is assigned exactly one item. Hence, one can assume that there is at least one item for each bidder that is valued as 0. Similarly and dually, suppose that an item is valued by three bidders as $(10, 8, 4)$. Then, the assignment will not change if this item is valued as $(6, 4, 0)$ instead. If we view V as a square matrix such that $V[b, i]$ equals $v_{b,i}$, then we can assume that there is a zero in every row by subtracting the minimum value in each row from every entry in the row. Similarly, we can assume that there is at least one zero in every column.

17.3 Market Clearing Price

In this section, we apply the LLP technique to the problem of finding a market clearing price. The set of feasible price vectors is given by

$$\{G \mid 0 \leq G[i] \leq T, 1 \leq i \leq n\}$$

This set of price vectors forms a distributive lattice under the component-wise comparison of the vectors. The minimum is given by the zero vector and the maximum is given by the vector T.

Given a price vector G, we define the bipartite *demand graph* with edge set E as follows. One side of the bipartite graph is the set of items I. The other side is the set of bidders U.

We now add edges between items and bidders as follows.

$$(j, b) \in E \equiv \forall i : (v_{b,j} - G[j]) \geq (v_{b,i} - G[i]).$$

Informally, there is an edge between item j and bidder b if the payoff for the bidder (the bid minus the price) is maximized with that item. Given any set $U' \subseteq U$, let $N(U', G)$ denote all the items that are adjacent to the vertices in U' in the demand graph (I, U, E). A price vector G is a *market clearing price* if the demand graph (I, U, E) has a perfect matching.

We construct a computation graph $(E, \rightarrow)$ for this problem as follows. The graph has one chain per item: item i has T events, where the k-th event represents setting $G[i] = k$. The global state G records, for each item i, the number of events item i has executed, which is exactly its current price. A price vector G is a *market clearing price*, denoted by $B_{clearingPrice}(G)$, if the demand graph (I, U, E) has a perfect matching. The unconstrained instance has no cross-item edges in the computation graph; we will see the more interesting computation graph in Fig. 17.1 once constraints are introduced in Section 17.4. We now claim that

Lemma 17.1 *The predicate $B_{clearingPrice}(G)$ is a lattice-linear predicate on the lattice of price vectors.*

Proof: Suppose $B_{clearingPrice}(G)$ is false, so the demand graph at G has no perfect matching. By Hall's theorem, there exists a minimal over-demanded set of items J. We show that every item $j \in J$ is forbidden: for any $H \geq G$ with $H[j] = G[j]$, $B_{clearingPrice}(H)$ is also false.

Since $H[j] = G[j]$, item j's price is unchanged. For every other item i, $H[i] \geq G[i]$, so the payoff $v_{b,i} - H[i] \leq v_{b,i} - G[i]$ for every bidder b. Therefore any bidder who demanded an item in J at price G still demands an item in J at price H (the alternatives outside J have not become more attractive). Hence J remains over-demanded at H, and $B_{clearingPrice}(H)$ is false.

■

Lemma 17.2 *A market clearing price exists for any non-negative valuation matrix V.*

Proof: By Lemma 17.1, whenever $B_{clearingPrice}(G)$ is false, at least one item is forbidden. The LLP algorithm starting at $G = \mathbf{0}$ monotonically increases prices. No item j with $G[j] > \max_b v_{b,j}$ can belong to any overdemanded set, since no bidder demands an item whose price exceeds every bidder's valuation for it. Hence the algorithm operates within the finite lattice $\prod_j [0, \max_b v_{b,j}]$, terminates, and by Lemma 17.1 the final state satisfies $B_{clearingPrice}$.

■

We are now ready to design an algorithm.

The LLP format of Algorithm LLP-Assignment specifies the forbidden predicate and the advance step. The advance uses the *slack-jump*: for any forbidden item j, the smallest price increase that creates a new tight edge incident to j is the slack

$$step[j] \ := \ \min_{(j,b) \in E} \Big[(v_{b,j} - G[j]) - \max_{i \neq j} (v_{b,i} - G[i]) \Big],$$

Algorithm LLP-Assignment: finding the minimum cost assignment vector

input: $v[b, i]$: int for all b, i
output: $G[1..n]$: minimum clearing price vector
var: G: array$[1..n]$ of $0..maxint$ initially $G[j] := 0$
$E = \{(k, b) \mid \forall i : (v[b, k] - G[k]) \geq (v[b, i] - G[i])\}$;
$demand(U') = \{k \mid \exists b \in U' : (k, b) \in E\}$;
$overDemanded(J) \equiv \exists U' \subseteq U : (demand(U') = J) \wedge (|J| < |U'|)$;
forbidden(j): $\exists$ minimal J: $OverDemanded(J) \wedge (j \in J)$
  $\Rightarrow$ **advance**: $step[j] = \min\{(v[b, j] - G[j]) - \max_{i \neq j}(v[b, i] - G[i]) \mid (j, b) \in E\}$;;
    $G[j] := G[j] + step[j]$;

i.e., the gap between bidder b's payoff for j and bidder b's payoff for the second-best item. This is the largest advance that does not overshoot the least fixpoint: j remains forbidden until $step[j]$ is added, and any larger advance would skip past a critical price. Algorithm Assignment gives the procedural implementation:

Algorithm Assignment: Finding the minimum clearing price vector

input: $v[b, i]$: valuations for all bidders b and items i
output: $G[1..n]$: minimum clearing price vector; M: perfect matching
G: real initially $\forall i : G[i] = 0$;
while *true* **do**
  $E := \{(i, b) \mid \forall j : (v_{b,i} - G[i]) \geq (v_{b,j} - G[j])\}$;
  if *there exists a perfect matching* M *in* E **then return** M;
  else
    $J :=$ minimal OverDemanded set in the bipartite graph with edges E;
    forall $j \in J$ **do**
      $step[j] = \min\{(v_{b,j} - G[j]) - \max_{i \neq j}(v_{b,i} - G[i]) \mid (j, b) \in E\}$;
      $G[j] := G[j] + step[j]$;
    end
  end
end

Theorem 17.3 *LLP-Assignment terminates in $O(n^3)$ time, and the price vector it returns is the unique* minimum *market clearing price.*

Proof: *Minimality.* By Lemma 17.1 the clearing predicate $B_{clearingPrice}$ is lattice-linear, so by the LLP main theorem (Chapter 2) every advance schedule that advances each forbidden index by an increment no larger than the least-fixpoint distance terminates at the unique minimum element of the price lattice satisfying $B_{clearingPrice}$. The slack $step[j]$ is exactly the smallest increase at which a new tight edge appears at j; any larger advance would overshoot the least fixpoint, and any smaller advance leaves the demand graph unchanged. So the advance is correctly sized.

Strongly polynomial bound. Define a *phase* as the iterations between two successive increases in the cardinality of a maximum matching in the demand graph E; there are at most n phases. Within a phase, each advance grows the alternating tree rooted at the unmatched bidders by at least one new vertex, so a phase has at most n advances. Each advance computes $step[j]$ in $O(n^2)$ work and updates E incrementally. The total cost is $O(n^3)$ with the standard incremental implementation, matching the Kuhn–Munkres bound.

■

Example 17.4 [LLP-Assignment on a 3×3 instance] We trace LLP-Assignment on three buyers $\{b_1, b_2, b_3\}$ and three items $\{h_1, h_2, h_3\}$ with valuation matrix

$$V = \begin{array}{c|ccc} & h_1 & h_2 & h_3 \\ \hline b_1 & 10 & 8 & 6 \\ b_2 & 9 & 7 & 4 \\ b_3 & 8 & 6 & 5 \end{array}$$

At prices $G = (p_1, p_2, p_3)$, buyer b_i demands $D(b_i) = \arg\max_j (v_{ij} - p_j)$; the algorithm raises prices on a minimal over-demanded set by the slack jump $step[j]$ until the demand graph admits a perfect matching.

Step 0. $G = (0, 0, 0)$. Payoffs equal valuations, so $D(b_1) = D(b_2) = D(b_3) = \{h_1\}$. The set $J = \{h_1\}$ is over-demanded.

Step 1. Compute the slack at h_1: for each buyer b with $(h_1, b) \in E$ we need $v_{b,1} - p_1$ minus b's second-best payoff. Currently $step[h_1] = \min\{10-8,\ 9-7,\ 8-6\} = 2$. Advance p_1 by 2, giving $G = (2, 0, 0)$ in a single advance. New payoffs:

	h_1	h_2	h_3
b_1	8	8	6
b_2	7	7	4
b_3	6	6	5

Now $D(b_1) = D(b_2) = D(b_3) = \{h_1, h_2\}$, so the minimal over-demanded set is $J = \{h_1, h_2\}$.

Step 2. Compute slacks at h_1 and h_2. For h_1: tight edges go to all three buyers, and each buyer's second-best (excluding h_1) is h_2 at payoff $8, 7, 6$ respectively, so $step[h_1] = \min\{8-8,\ 7-7,\ 6-6\} = 0$. Wait — a slack of zero means h_2 is already tied; the demand-graph update for h_1 alone merges the duplicate tie without raising the price. The over-demanded condition forces us to advance *both* indices in J together. Recomputing for h_2: $step[h_2] = \min\{8-6,\ 7-4,\ 6-5\} = 1$, the gap to each buyer's second-best item (now h_3). With $J = \{h_1, h_2\}$ advanced jointly, the LLP loop raises p_1 and p_2 each by 1 in a single advance, yielding $G = (3, 1, 0)$. New payoffs:

	h_1	h_2	h_3
b_1	7	7	6
b_2	6	6	4
b_3	5	5	5

Now $D(b_1) = D(b_2) = \{h_1, h_2\}$ and $D(b_3) = \{h_1, h_2, h_3\}$. The demand graph admits the perfect matching $b_1 \to h_1,\ b_2 \to h_2,\ b_3 \to h_3$, and the algorithm returns $G = (3, 1, 0)$ as the minimum market clearing price after just two advances.

17.4 Constrained Market Clearing Price Problem

We now generalize the problem of finding a market clearing price to that of finding a constrained market clearing price. For example, constraints on the clearing prices of the form $(G[j] \geq k) \Rightarrow (G[i] \geq k')$ for $1 \leq k, k' \leq C$ are lattice-linear. The constraint says that if item j is priced at least k, then item i must be priced at least k'. The constraint $G[i] \geq G[j]$ is also lattice-linear. Observe that the constraint $G[i] \geq G[j]$ is equivalent to

$$(G[j] \geq 1 \Rightarrow G[i] \geq 1) \wedge (G[j] \geq 2 \Rightarrow G[i] \geq 2) \ldots (G[j] \geq C \Rightarrow G[i] \geq C)$$

Similarly, the constraint $(G[i] = G[j])$ can be modeled as $(G[i] \geq G[j]) \wedge (G[j] \geq G[i])$. Also, a constraint of the form $G[j] \geq f(G)$ for monotone f is lattice-linear. Given any set of valuations, and a boolean predicate B that is a conjunction of lattice-linear constraints, a price vector G is a *constrained market clearing price*, denoted by $constrainedClearing(G)$ iff $clearing(G) \wedge B(G)$. Since $B(G)$ is lattice-linear, it is sufficient to give an algorithm for $clearing(G)$. It follows that the set of constrained market clearing price vectors is closed under meets. By applying the lattice-linear predicate detection, we get an algorithm to compute the least constrained market clearing price shown in Fig. LLP-Assignment. We get a generalization of Demange, Gale and Sotomayor's exact auction mechanism [DGS86] to incorporate lattice-linear constraints on the market clearing price.

We construct a computation graph $(E, \to)$ for this problem as follows (see Fig. 17.1). For any constraint $(G[j] \geq k) \Rightarrow (G[i] \geq k')$, we put an edge from the event k' on the chain for item i to the event k on the chain for item j. By putting such edges, we get a computation graph and any price vector that does not satisfy constraints is not a consistent global state.

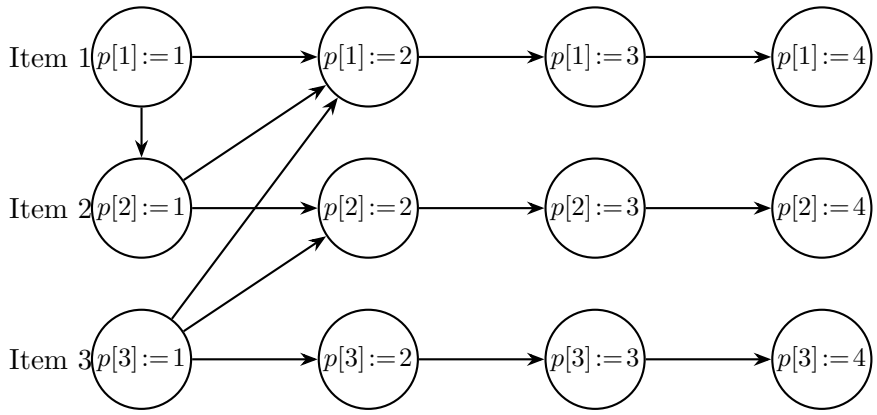


Figure 17.1: The computation graph for a market with three items and three bidders. The valuation and price of any item is a number between 0 and 4. The computation graph models the constraint $B \equiv (p[2] \geq 2 \Rightarrow p[1] \geq 3) \wedge (p[1] \geq 2 \Rightarrow p[3] \geq 1)$

Example 17.5 [Constrained MCP on the computation graph of Fig. 17.1] Take three bidders b_1, b_2, b_3 with valuation vectors over items $(1, 2, 3)$:

$$V \;=\; \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline b_1 & 4 & 1 & 0 \\ b_2 & 4 & 1 & 2 \\ b_3 & 4 & 2 & 0 \end{array}$$

The two implication constraints from Fig. 17.1 are

$$B \;\equiv\; \big(\, p[2] \geq 2 \;\Rightarrow\; p[1] \geq 3 \,\big) \;\wedge\; \big(\, p[1] \geq 2 \;\Rightarrow\; p[3] \geq 1 \,\big).$$

We trace the constrained LLP loop, raising prices until the demand graph admits a perfect matching *and* B is satisfied.

Step 0. $G = [0, 0, 0]$. Every bidder's payoff is the valuation itself, so all three demand item 1. The set $\{1\}$ is over-demanded; advance $p[1]$.

Step 1. $G = [1, 0, 0]$. Payoffs:

	1	2	3
b_1	3	1	0
b_2	3	1	2
b_3	3	2	0

All three bidders still demand item 1; $\{1\}$ remains over-demanded. Advance $p[1]$.

Step 2. $G = [2, 0, 0]$. Payoffs:

	1	2	3
b_1	2	1	0
b_2	2	1	2
b_3	2	2	0

Now $D(b_1) = \{1\}$, $D(b_2) = \{1, 3\}$, $D(b_3) = \{1, 2\}$. The demand graph admits the perfect matching $1 \to b_1, 2 \to b_3, 3 \to b_2$ — so the price clears the market. *However*, the constraint $(p[1] \geq 2 \Rightarrow p[3] \geq 1)$ is violated because $p[1] = 2$ but $p[3] = 0$. The constraint forbids item 3; advance $p[3]$.

Step 3. $G = [2, 0, 1]$. Payoffs:

	1	2	3
b_1	2	1	-1
b_2	2	1	1
b_3	2	2	-1

B is now satisfied. But $D(b_1) = D(b_2) = \{1\}$ and $D(b_3) = \{1, 2\}$, so $\{1\}$ is over-demanded again. Advance $p[1]$.

Step 4. $G = [3, 0, 1]$. Payoffs:

	1	2	3
b_1	1	1	-1
b_2	1	1	1
b_3	1	2	-1

Now $D(b_1) = \{1, 2\}$, $D(b_2) = \{1, 2, 3\}$, $D(b_3) = \{2\}$. The demand graph admits $1 \to b_1, 2 \to b_3, 3 \to b_2$ and B holds. The algorithm returns $G = [3, 0, 1]$ as the least constrained market clearing price.

17.5 Certificates of Optimality

Suppose that a colleague gives you an assignment M and claims it is optimal. Checking that M is proper (every item is matched to exactly one bidder) is easy. The harder question is: how can the colleague convince you of optimality *without your having to recompute the optimal assignment from scratch*?

The answer is to provide an LP-duality certificate: the colleague hands you both the matching M and a price vector G. For the item i matched to bidder b, set $p_i := G[i]$ and $q_b := v_{b,i} - p_i$. It then suffices to verify the dual feasibility condition

$$\forall i, j : \quad p_i + q_j \geq v_{i,j}.$$

This check is $O(n^2)$ sequentially and $O(1)$ time in parallel with n^2 processors. By construction $p_i + q_b = v_{b,i}$ for every matched pair (i, b), so primal complementary slackness holds; combined with the inequality above, this exhibits primal and dual feasible solutions with equal objective value, certifying optimality of M via LP duality.

Conversely, suppose the assignment is *not* optimal. The colleague can be shown a witness of non-optimality without computing the actual optimum.

Theorem 17.6 *An assignment M is optimal if and only if there is no positive weight cycle with respect to M.*

Proof: A positive weight cycle is an even-length cycle of alternating matched and unmatched edges in which the total weight of the unmatched edges exceeds that of the matched edges. If such a cycle exists, switching matched and unmatched edges along the cycle yields a matching of strictly higher weight, so M is not optimal.

Conversely, suppose M is not optimal and let M^* be an optimal matching. The symmetric difference $M \triangle M^*$ decomposes into vertex-disjoint alternating paths and even-length cycles. Since M has lower weight than M^*, some component of $M \triangle M^*$ contributes positively to the weight gain when we replace M's edges in that component with M^*'s; on a cycle component this is exactly a positive weight cycle.

■

Certificates for the constrained variant. The same dual-feasibility certificate works for an instance of the Constrained Market Clearing Price problem (Section 17.4), with one extra step: in addition to checking $p_i + q_b \geq v_{b,i}$, the verifier checks that the supplied price vector G satisfies the side constraint $B(G)$. Since B is by hypothesis a conjunction of lattice-linear constraints (e.g., $G[i] \geq G[j]$ or $G[j] \geq f(G)$ for monotone f), each conjunct is a constant-time test on G, so the constrained certificate is checkable in $O(n^2 + |B|)$ time. The matching M produced by LLP-Assignment together with the constrained G is then certified optimal among assignments whose induced price vector lies in B.

17.6 Summary

This chapter presented LLP-Assignment, an LLP-based ascending-price algorithm for the assignment problem. The algorithm advances each forbidden item's price by the slack-jump — the largest LLP-correct advance, which lifts the price exactly to the next critical price — yielding a strongly polynomial $O(n^3)$ bound, where n is the number of items (or bidders). This matches the best known strongly polynomial bound for the assignment problem, achieved by the Kuhn–Munkres (Hungarian) and Edmonds–Karp algorithms.

Algorithm	Problem	Complexity
LLP-Assignment	Assignment (min clearing price)	$O(n^3)$
ConstrainedMarketClearingPrice	Constrained assignment	$O(n^3)$

Table 17.1: Algorithms and their complexities (n = number of items/bidders).

17.7 Problems

1. **(Clearing Prices Join Closure)** Show that the set of market clearing price vectors is also closed under the join operation.

2. **(Assignment LP and Dual)** Give the linear programming formulation and its dual for solving the assignment problem.

3. **(Verifying Optimal Assignment)** Suppose that your friend gives you an assignment and claims that the assignment is optimal. It is easy to check that the assignment is proper, i.e., every item is assigned to exactly one bidder. But, how can she quickly convince you of the optimality?

4. **(LLP Assignment Trace)** Run Algorithm Assignment by hand on the 3×3 instance with valuation matrix
$$V = \begin{pmatrix} 5 & 3 & 1 \\ 3 & 5 & 1 \\ 2 & 3 & 4 \end{pmatrix}.$$
Report the sequence of price vectors visited, the final minimum market clearing price, and a supporting perfect matching.

5. **(Unique Minimum Clearing Price)** Show that there is a unique minimum market clearing price vector. (Hint: use Lemma 17.1.)

6. **(Reserve Price Constraint)** Suppose the auctioneer imposes a reserve price $r_i \geq 0$ on each item i, meaning that no item may be sold below its reserve. Show how to model this as a constrained market clearing price problem and how Algorithm LLP-Assignment should be modified.

7. **(Uniform Valuation Shift)** Suppose all valuations $v_{b,i}$ are increased by the same constant c, giving new valuations $v'_{b,i} = v_{b,i} + c$. How does the optimal assignment change? How does the minimum market clearing price change?

8. **(Incremental Assignment)** A maximum-weight assignment has been computed by LLP-Assignment on n bidders and n items, producing price vector G and matching M. A new (bidder, item) pair arrives, together with the new bidder's valuations for all items (including the new one) and existing bidders' valuations for the new item. Update (G, M) to the new (G', M') on $n+1$ pairs without restarting from scratch.

9. **(Dynamic Valuations)** Bidder b_k revises her valuation for item i from $v_{b_k,i}$ to $v'_{b_k,i}$. The current price vector G may no longer be market-clearing. Describe a repair procedure with running time bounded by the work of one slack-jump iteration.

10. **(Constrained Assignment With Caps)** Each bidder b has a cap $c_b \geq 1$ on the number of items she may receive; total caps equal the number of items. This is the *transportation problem* (Section 16.5 of the LP chapter). Adapt LLP-Assignment to caps.

11. **(Assignment With Ties)** The valuation matrix may contain ties: $v_{b,i} = v_{b,j}$ for some i, j. Then b is indifferent between i and j at the same price, and $D(b) = \{i, j\}$. Show that LLP-Assignment still terminates and produces the minimum clearing price; the matching itself may not be unique.

12. **(Fair Assignment)** Among all maximum-weight assignments, find one that minimises the maximum bidder regret $\max_b[\max_i v_{b,i} - v_{b,\mu(b)}]$.

17.8 Bibliographic Remarks

Kuhn's Hungarian method for solving the assignment problem is from [Mun57]. Shapley and Shubik [SS71] introduced the assignment game and connected market-clearing prices to the core of a cooperative game; Demange, Gale, and Sotomayor [DGS86] gave the ascending-price (exact) auction mechanism that the LLP algorithm of this chapter generalizes. Bertsekas's auction algorithm [Ber92] is a closely related ascending-bid scheme with strong empirical performance. The classical Edmonds–Karp paper [EK72] embeds the assignment problem inside minimum-cost flow, which is the natural generalization to non-bipartite supply-and-demand instances and to the transportation problem. The notion of a constrained market clearing price is from [Gar20b]. The monograph by Burkard, Dell'Amico, and Martello [BDM12] is a comprehensive reference for variants of the assignment problem and their algorithms.

Chapter 18

Horn and 2-SAT Satisfiability

18.1 Introduction

The satisfiability problem—given a Boolean formula, determine whether there exists an assignment of truth values to its variables that makes the formula true—is the canonical NP-complete problem. Yet within this vast landscape of intractability lie islands of tractability: special classes of Boolean formulas for which satisfiability can be decided efficiently. Understanding these tractable classes is important for both theory and practice, as many real-world constraint satisfaction problems fall naturally into these classes.

In this chapter, we study two of the most important such classes: *Horn formulas* and *2-SAT formulas*. Horn formulas arise naturally in logic programming, database theory, and artificial intelligence. The Prolog programming language, for instance, is built entirely on Horn clauses: every Prolog rule is a Horn implication, and every Prolog query is answered by finding a satisfying assignment for a conjunction of Horn clauses. In database theory, Horn clauses model integrity constraints and deductive rules.

The 2-SAT problem, where every clause has at most two literals, appears in numerous settings: type checking in programming languages, constraint propagation in planning, and as a subroutine in approximation algorithms for harder optimization problems.

What makes these two classes tractable is fundamentally different. Horn formulas are lattice-linear predicates: the set of satisfying assignments is closed under meet (componentwise AND), and the Lattice Linear Predicate (LLP) framework yields the *least* satisfying assignment. In contrast, 2-SAT formulas are *not* lattice-linear in general, but their structure can be captured by an *implication graph* whose strongly connected components reveal satisfiability in linear time.

Throughout this chapter, we assume that the predicate is given in the *conjunctive normal form* (CNF). A Boolean formula B in conjunctive normal form is simply a conjunction of *clauses*, where each clause is a pure disjunction of *literals*. A literal is a variable in its simple form or in a complemented form. For example, suppose that we have n Boolean variables $\{x_1, x_2, \ldots, x_n\}$. Then, the formula $(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2)$ is in CNF with two clauses. The first clause has three literals $\{\neg x_1, \neg x_2, x_3\}$, and the second clause has two literals $\{x_1, x_2\}$.

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

The chapter is organized as follows. Section 18.2 defines Horn clauses and presents the LLP-based algorithm for Horn satisfiability, proving lattice-linearity along the way. Section 18.3 gives an alternative proof via arithmetization. Section 18.4 develops the implication-graph approach for 2-SAT. Section 18.5 summarizes the results.

18.2 Horn Satisfiability

A clause is a *Horn clause* if it has at most one positive literal. An example of a Horn clause is $(\neg x_1 \vee \neg x_2 \vee x_3)$. Note that this formula is also equivalent to $(x_1 \wedge x_2) \Rightarrow x_3$. A Horn clause written in this form is also called a *Horn implication* or a *definite clause*. Notice that it has only positive literals on the left-hand side and a single positive literal on the right-hand side. The left-hand side may be empty. Thus, x_3 is also a Horn implication equivalent to $\mathit{true} \Rightarrow x_3$.

Another example of a Horn clause is $(\neg x_1 \vee \neg x_2 \vee \neg x_3)$. This clause does not have any positive literal. It is a *pure negative* clause.

A Horn formula is just a conjunction of Horn clauses. The problem of HornSAT is to determine if the given Horn formula is satisfiable.

Example 18.1 Consider the following Horn formula B with three variables x_1, x_2, and x_3:

$$x_1 \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3),$$

or, written as implications, $\top \Rightarrow x_1$, $x_1 \Rightarrow x_2$, and $x_1 \wedge x_2 \Rightarrow x_3$. The unit fact forces $x_1 = \text{true}$; the second clause then forces $x_2 = \text{true}$; the third clause forces $x_3 = \text{true}$. Hence $(1, 1, 1)$ satisfies B, and any satisfying assignment must agree with it on every variable. So $(1, 1, 1)$ is the *least* satisfying assignment in the Boolean lattice of three variables. We will use B as the running example throughout this section.

Example 18.2 The Boolean expression $B = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$ is a Horn formula with two implications: $x_1 \Rightarrow x_2$ and $x_2 \Rightarrow x_3$. Its satisfying assignments are shown in the Boolean lattice of Fig. 18.1. Note that the four satisfying assignments $\{000, 001, 011, 111\}$ form a meet-closed (i.e., closed under componentwise AND) subset of the lattice. The least satisfying assignment is 000.

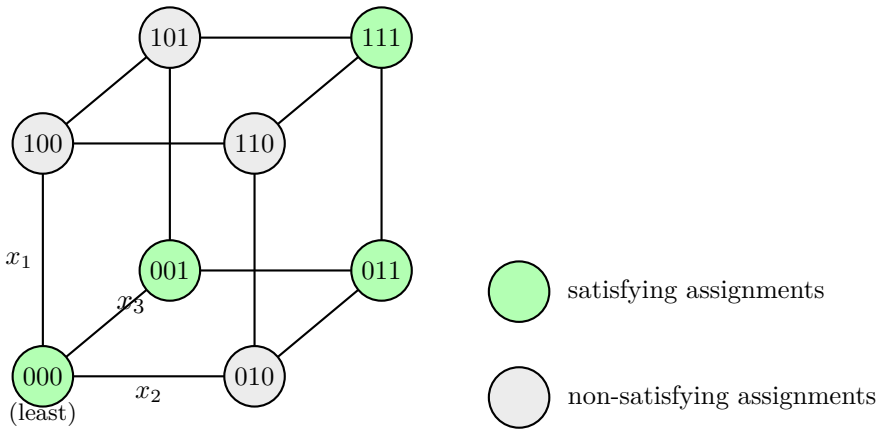


Figure 18.1: Lattice of Boolean assignments on (x_1, x_2, x_3) for the Horn formula $B = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$. The four satisfying assignments form a meet-closed subset $\{000, 001, 011, 111\}$.

Example 18.3 Consider the following set of Horn clauses with two variables x_1 and x_2:

$$(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge x_1$$

This set of Horn clauses is unsatisfiable because there is no assignment of truth values to x_1 and x_2 that can satisfy all the clauses simultaneously. To see why: the unit clause x_1 forces $x_1 = \text{true}$. The first implication $x_1 \Rightarrow x_2$ then forces $x_2 = \text{true}$. But now the pure negative clause $(\neg x_1 \vee \neg x_2)$ is violated.

Example 18.4 Horn clauses arise naturally in logic programming. Consider a small knowledge base:

$\text{parent}(alice, bob)$	(fact: Alice is Bob's parent)
$\text{parent}(bob, carol)$	(fact: Bob is Carol's parent)
$\text{parent}(X, Y) \wedge \text{parent}(Y, Z) \Rightarrow \text{grandparent}(X, Z)$	(rule)

The query "Is Alice a grandparent of Carol?" amounts to checking whether

$$\text{grandparent}(alice, carol)$$

can be derived. Horn satisfiability and Horn entailment are closely related: the facts and rules are Horn clauses, and Prolog's inference mechanism corresponds to repeated application of Horn implications (forward chaining), which is precisely the LLP advance rule of Algorithm LLP-HornSAT.

We now show that Horn formulas are lattice-linear predicates, which enables us to apply

the LLP framework to find the least satisfying assignment efficiently.

Lemma 18.5 *Any Horn formula is a lattice-linear predicate.*

Proof: Let $B = B_1 \wedge B_2 \wedge \ldots \wedge B_m$ be the Horn formula, where each B_i for $i \in [m]$ is a Horn clause. Since lattice-linear predicates are closed under conjunction, it suffices to show that every Horn clause B_i is lattice-linear. We consider two types of Horn clauses.

First, suppose that B_i is a Horn implication, i.e., $B_i \equiv (x_{i_1} \wedge x_{i_2} \wedge \ldots \wedge x_{i_{k-1}} \Rightarrow x_{i_k})$. Consider any Boolean vector G in which B_i is false. This implies that $x_{i_1}, x_{i_2}, \ldots, x_{i_{k-1}}$ are true in G but x_{i_k} is false. Consider any H such that $H \geq G$ and $H[i_k] = G[i_k]$. Since $H \geq G$, we have that $(x_{i_1} \wedge x_{i_2} \wedge \ldots \wedge x_{i_{k-1}})$ holds in H. Furthermore, since $H[i_k]$ is equal to $G[i_k]$, x_{i_k} is false in H. Hence, B_i is also false in H.

Now suppose that B_i is a pure negative clause, i.e., $B_i \equiv (\neg x_{i_1} \vee \neg x_{i_2} \vee \ldots \vee \neg x_{i_k})$. Suppose that B_i is false in G. This implies that all the variables $x_{i_1}, \ldots, x_{i_k}$ are true. Consider any Boolean vector $H \geq G$. H must also have all these variables true and, therefore, B_i is false in H (and any index is trivially forbidden).

■

The proof of Lemma 18.5 also shows us the appropriate advancement function. For Horn implications, we must advance on the right-hand-side literal. For pure negative clauses, all Boolean vectors greater than G do not satisfy B. Hence, the algorithm can simply return that "no satisfying vector exists."

We can now apply the LLP algorithm to find a satisfying assignment for a Horn formula. We assume that the Horn formula uses the variables $x_1, \ldots, x_n$. These n Boolean variables define a Boolean lattice L of size 2^n that consists of Boolean vectors of size n.

Algorithm LLP-HornSAT: Algorithm to find the least assignment that satisfies B

input: Horn CNF B with variables $x_1, \ldots, x_n$
output: G: least satisfying assignment, or null

```
1 G: vector of boolean initially ∀i : G[i] = false;
2 forbidden(j): ∃h: all antecedents of B_h are true, they imply x_j, and x_j is false in G
3    ⇒ advance: G[j] := true;
4 forbidden(j): ∃h: a negative clause B_h is false, and x_j is a part of B_h
5    ⇒ advance return null ;                  // no satisfying assignment exists
6 return G ;                                  // the least assignment that satisfies B
```

We leave an efficient implementation of Algorithm LLP-HornSAT as an exercise.

We also get the following consequence of Lemma 18.5, which follows from its lattice-linearity.

Corollary 18.6 *If G and H satisfy the Horn formula, then $G \sqcap H$ also satisfies that formula.*

A direct proof of this fact is left as an exercise. Although the set of assignments satisfying a Horn formula is closed under meets, it is not closed under joins. For example, consider the Horn formula $B \equiv (\neg x_1 \vee \neg x_2)$. The bit vectors $G = [1, 0]$ and $H = [0, 1]$ satisfy B but their join $[1, 1]$ does not satisfy B.

Example 18.7 Let us trace LLP-HornSAT on the Horn formula B from Example 18.1, written as the three implications

$$C_1 : \ \top \Rightarrow x_1, \qquad C_2 : \ x_1 \Rightarrow x_2, \qquad C_3 : \ x_1 \wedge x_2 \Rightarrow x_3.$$

Start with $G = (0, 0, 0)$.

Step 0. C_1 has empty antecedent and consequent x_1, which is false. So C_1 is violated and index 1 is forbidden. Advance: $G[1] :=$ true, giving $G = (1, 0, 0)$.

Step 1. C_1 now holds. C_2's antecedent x_1 is true but the consequent x_2 is false, so C_2 is violated and index 2 is forbidden. Advance: $G[2] :=$ true, giving $G = (1, 1, 0)$.

Step 2. C_1, C_2 hold. C_3's antecedents $x_1 \wedge x_2$ are both true but the consequent x_3 is false, so C_3 is violated and index 3 is forbidden. Advance: $G[3] :=$ true, giving $G = (1, 1, 1)$.

Step 3. All three clauses are satisfied; no index is forbidden. The algorithm returns $G = (1, 1, 1)$ as the least satisfying assignment, agreeing with Example 18.1. The trace illustrates how the LLP advance rule realizes the standard *forward chaining* of Horn solvers: each violated implication fires its consequent, possibly enabling further implications in the next iteration.

Example 18.8 Now augment B with the pure negative clause $C_4 \equiv (\neg x_2 \vee \neg x_3)$:

$$B' = B \wedge (\neg x_2 \vee \neg x_3).$$

The trace proceeds through Steps 0–2 of Example 18.7 unchanged, reaching $G = (1, 1, 1)$. At this state the pure negative clause C_4 becomes false because both x_2 and x_3 are true. The advance rule for negative clauses fires the unsatisfiability return, so LLP-HornSAT returns *null*: B' has no satisfying assignment. This matches the lemma that pure negative clauses, once violated, can never be re-satisfied by further advances.

The LLP-HornSAT algorithm with m clauses and n variables can be implemented in $O(m+n)$ time.

Dual-Horn formulas. A CNF formula is *dual-Horn* if every clause has at most one *negative* literal. Dual-Horn formulas are the bit-complement counterpart to Horn: a formula F is dual-Horn iff $F[x \mapsto \neg x]$ (with every literal flipped) is Horn. Consequently, the set of satisfying assignments of a dual-Horn formula is closed under componentwise OR (join) rather than meet, and forms a join-semilattice. The LLP framework applied to the dual Boolean lattice (with $\bot = (1, 1, \ldots, 1)$ and the order reversed) yields the *greatest* satisfying assignment in $O(n + m)$ time. The forbidden/advance rules dualize: a clause with one negative literal $\neg x_j$

and antecedents $x_{i_1}, \ldots, x_{i_{k-1}}$ all set false forbids index j, advancing $G[j] := \mathit{false}$. Dual-Horn satisfiability is therefore solved by the same algorithm with the lattice flipped, and the Renamable Horn problem (Exercise) asks when this flip can be applied to selected variables to bring an arbitrary formula into one of these two classes.

18.3 Arithmetization of Horn Clauses

In this section, we give another proof that Horn clauses are lattice-linear by considering arithmetic expressions instead of Boolean expressions. We use the natural assignment of Boolean variables x_i to integer binary variables y_i. If x_i is false, then y_i is assigned a value 0, otherwise it is assigned 1. Given any clause, we translate it into an arithmetic predicate as follows. Every positive literal x_i in any clause is changed to y_i and every negative literal $\neg x_i$ is changed to $(1 - y_i)$. The clause is true iff the sum of all values so replaced in any clause is at least 1. For example, clause $(\neg x_1 \vee \neg x_2 \vee x_3)$ is equivalent to $(1 - y_1) + (1 - y_2) + y_3 \geq 1$. If all y_i take values in the set $\{0, 1\}$, then it is easy to verify that the clause is true for $x's$ iff the corresponding arithmetic predicate is true for $y's$. Let us see how the implication Horn clauses get translated. An implication Horn clause $(x_{i_1} \wedge x_{i_2} \wedge \ldots \wedge x_{i_{k-1}} \Rightarrow x_{i_k})$ gets translated into $(1 - y_{i_1}) + (1 - y_{i_2}) + \ldots + (1 - y_{i_{k-1}}) + y_{i_k} \geq 1$ which is equivalent to $y_{i_k} \geq y_{i_1} + y_{i_2} + \ldots + y_{i_{k-1}} - (k - 2)$. The right-hand side is a monotone function of y; hence, the predicate is lattice-linear. A pure negative clause $(\neg x_{i_1} \vee \neg x_{i_2} \vee \ldots \vee \neg x_{i_k})$ is translated into $(1 - y_{i_1}) + (1 - y_{i_2}) + \ldots + (1 - y_{i_k}) \geq 1$. This predicate is equivalent to $k - 1 \geq y_{i_1} + y_{i_2} + \ldots + y_{i_k}$ which, once it becomes false, stays false. Hence, predicates for negative clauses are also lattice-linear.

Example 18.9 Let us arithmetize the running Horn formula from Example 18.1,

$$B = x_1 \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3).$$

Mapping each Boolean x_i to $y_i \in \{0, 1\}$:

- The unit clause x_1 becomes $y_1 \geq 1$.
- The implication clause $(\neg x_1 \vee x_2)$ becomes $(1 - y_1) + y_2 \geq 1$, i.e., $y_2 \geq y_1$.
- The implication clause $(\neg x_1 \vee \neg x_2 \vee x_3)$ becomes $(1 - y_1) + (1 - y_2) + y_3 \geq 1$, i.e., $y_3 \geq y_1 + y_2 - 1$.

Each lower bound on the consequent is a *monotone* function of the antecedents, so each clause is lattice-linear and the conjunction B is too. The least satisfying integer point is $y = (1, 1, 1)$, matching the trace in Example 18.7.

If we further add the pure negative clause $(\neg x_2 \vee \neg x_3)$ from Example 18.8, it arithmetizes to $(1 - y_2) + (1 - y_3) \geq 1$, equivalent to $y_2 + y_3 \leq 1$. This constraint becomes false at $y = (1, 1, 1)$ and cannot be repaired by raising any y_i further — making the unsatisfiability in Example 18.8 visible as an upper-bound cut on the integer lattice.

18.4 2-SAT

A conjunctive normal form formula is a 2-SAT formula iff every clause has at most two literals. For example, formula $B_1 \equiv x_1 \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3)$ is a 2-SAT formula.

If we consider the lattice of Boolean vectors, a 2-SAT formula may not be closed under meet. For example, consider the 2-SAT formula $(x_1 \vee x_2)$. The Boolean vectors 10 and 01 satisfy the formula, but their meet 00 does not. Hence, there is no *minimum* satisfying assignment and the simple LLP algorithm is not applicable.

This means we need a fundamentally different approach for 2-SAT. The key insight is that every 2-literal clause $(\ell_1 \vee \ell_2)$ can be read as two implications: $\neg\ell_1 \Rightarrow \ell_2$ and $\neg\ell_2 \Rightarrow \ell_1$. These implications define a directed graph—the *implication graph*—whose structure completely characterizes satisfiability.

We now outline an approach to detect if there is any element in the lattice that satisfies B, a 2-SAT formula. The key idea for the 2-SAT algorithm is to convert the predicate to an implication graph. We assume that every clause has two literals. A unit clause x_i or $\neg x_i$ forces the value of the variable. It can be replaced in B, resulting in a 2-SAT formula with fewer variables. Now, we construct an implication graph $I(B)$ for B as follows. The graph has $2n$ vertices, one for each literal. Every clause $(l_i \vee l_j)$ in B is true iff $(\neg l_i \Rightarrow l_j) \wedge (\neg l_j \Rightarrow l_i)$. Therefore, we add two directed edges to the graph — from $\neg l_i$ to l_j and from $\neg l_j$ to l_i.

Example 18.10 Consider the following 2-SAT formula with three variables x_1, x_2, and x_3:

$$(\neg x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee x_3)$$

The implication graph of B is shown in Fig. 18.2.
Each clause produces two directed edges:

- $(\neg x_1 \vee x_2)$ gives $x_1 \to x_2$ and $\neg x_2 \to \neg x_1$.
- $(x_1 \vee x_3)$ gives $\neg x_1 \to x_3$ and $\neg x_3 \to x_1$.
- $(\neg x_2 \vee x_3)$ gives $x_2 \to x_3$ and $\neg x_3 \to \neg x_2$.

This formula is satisfiable. For instance, $x_1 = x_2 = x_3 =$ true works. Note that no variable and its negation are in the same strongly connected component.

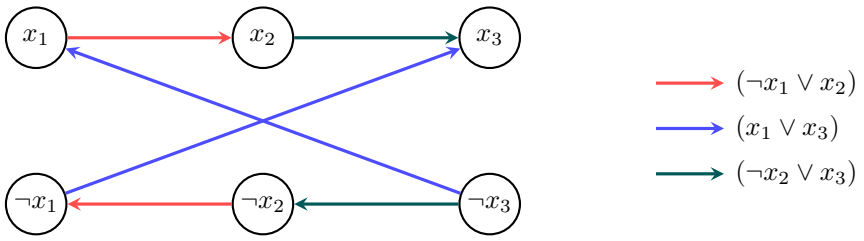


Figure 18.2: Implication graph for $(\neg x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee x_3)$.

An important structural property of the implication graph is its *skew symmetry*: if there is an edge from ℓ to ℓ', then there is also an edge from $\neg\ell'$ to $\neg\ell$. This follows directly from the construction: both edges come from the same clause. As a consequence, if there is a path from ℓ to ℓ', there is also a path from $\neg\ell'$ to $\neg\ell$.

The following theorem captures the soundness and completeness of the SCC-based approach. It is the structural fact behind Step 3 of Algorithm 2-SAT.

Theorem 18.11 *A 2-SAT formula B is satisfiable iff there is no variable x_i such that $\neg x_i$ is reachable from x_i* and *x_i is reachable from $\neg x_i$ in the implication graph $I(B)$ — equivalently, iff no SCC of $I(B)$ contains both x_i and $\neg x_i$.*

Proof: ($\Rightarrow$) Suppose some variable x_i has both paths in $I(B)$. Pick any assignment σ. Whichever value σ gives x_i, the path from that literal to its negation forces a contradiction along an implication chain that σ must respect (because every edge of $I(B)$ encodes a clause-level implication). Hence no σ satisfies B.

($\Leftarrow$) Suppose no SCC contains both x_i and $\neg x_i$. Step 4 of Algorithm 2-SAT constructs a satisfying assignment: process the SCCs in reverse topological order and set every unassigned literal in the current SCC to true (and its negation to false). Consider any clause $(a \vee b)$, which contributes edges $\neg a \to b$ and $\neg b \to a$. Suppose for contradiction that both a and b are false, so $\neg a$ and $\neg b$ were set to true. Since $\neg a \to b$ is an edge, SCC($\neg a$) precedes or equals SCC(b) in topological order, so SCC(b) was processed no later than SCC($\neg a$) in reverse topological order. When SCC(b) was processed, b was unassigned (otherwise b would already be true or false; if true, we have a contradiction with b false), so b was set to true — contradicting $b =$ false. Hence every clause is satisfied.

■

We leave the following companion claims as exercises.

Exercise 18.1 *1. Suppose B is satisfiable. Then, all satisfying assignments set x_j to true iff there exists a path from $\neg x_j$ to x_j in the implication graph.*

2. Suppose B is satisfiable. Then, all satisfying assignments set x_j to false iff there exists a path from x_j to $\neg x_j$ in the implication graph.

Algorithm 2-SAT: 2-SAT satisfiability algorithm via SCCs

input: a 2-SAT formula F with variables $x_1, \ldots, x_n$ and m clauses.
output: UNSATISFIABLE, or SATISFIABLE together with a satisfying assignment.

```
// Step 1: construct the implication graph
```
foreach *clause* $(a \vee b)$ *in* F **do**
  Add directed edges $(\neg a \to b)$ and $(\neg b \to a)$;
end
```
// Step 2: compute strongly connected components (SCCs)
```
Use Kosaraju's or Tarjan's algorithm to compute SCCs of the graph;
```
// Step 3: check for contradictions
```
foreach *variable* x_i **do**
  if x_i *and* $\neg x_i$ *belong to the same SCC* **then**
    return UNSATISFIABLE;
  end
end
```
// Step 4: construct an assignment
```
Topologically sort the SCCs of the implication graph (treating each SCC as a single vertex);
Mark every literal as unassigned;
foreach *SCC* S *in* reverse *topological order* **do**
  foreach *literal* $\ell \in S$ *that is unassigned* **do**
    Assign $\ell :=$ true and $\neg\ell :=$ false;
  end
end
return SATISFIABLE and the constructed assignment;

Algorithm 2-SAT works as follows.

In the algorithm, we need to check if there exists a path from x_j to $\neg x_j$ and vice versa. One way to accomplish this is to find strongly connected components (SCCs) in the graph. An SCC is a subgraph in which every vertex is reachable from every other vertex in the same SCC. (One can use Tarjan's algorithm or Kosaraju's algorithm to find SCCs.) The expression is satisfiable if and only if for every variable x_j, x_j and $\neg x_j$ are not in the same SCC. If the expression is satisfiable, an assignment is constructed in Step 4 by processing the SCCs in *reverse* topological order: every unassigned literal in the current SCC is set to true (and its negation to false). To see why this is sound, recall that when each SCC is contracted to a single vertex, the resulting directed acyclic graph admits a topological order in which all edges go from earlier to later SCCs. So when we set a literal ℓ to true, every implication edge $\ell \to \ell'$ leads to an SCC that has already been processed and therefore ℓ' has already been set to true. The check in Step 3 guarantees that no SCC contains both x_j and $\neg x_j$, so within a single SCC we never set both a literal and its negation to true. The sequential time complexity of the

2-SAT problem is linear in the number of clauses.

Example 18.12 Consider the 2-SAT formula:

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$$

The implication graph has edges: $\neg x_1 \to x_2$, $\neg x_2 \to x_1$ (from clause 1); $x_1 \to \neg x_2$, $x_2 \to \neg x_1$ (from clause 2); $x_1 \to x_2$, $\neg x_2 \to \neg x_1$ (from clause 3); $\neg x_1 \to \neg x_2$, $x_2 \to x_1$ (from clause 4).

From x_1 we can reach $\neg x_1$: $x_1 \to x_2 \to \neg x_1$. From $\neg x_1$ we can reach x_1: $\neg x_1 \to x_2 \to x_1$. So x_1 and $\neg x_1$ are in the same SCC. The formula is unsatisfiable.

Example 18.13 Consider the 2-SAT formula:

$$(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3)$$

The implication graph has edges: $x_1 \to x_2$, $\neg x_2 \to \neg x_1$ (from clause 1); $x_2 \to x_3$, $\neg x_3 \to \neg x_2$ (from clause 2); $\neg x_1 \to \neg x_3$, $x_3 \to x_1$ (from clause 3).

The SCCs are $S_+ = \{x_1, x_2, x_3\}$ and $S_- = \{\neg x_1, \neg x_2, \neg x_3\}$, as shown in Fig. 18.3. No variable shares an SCC with its negation, so the formula is satisfiable. There are no edges between S_+ and S_- in the SCC-contracted graph, so either order is a valid topological sort. Pick the topological order (S_-, S_+). Processing in *reverse* topological order, we first visit S_+: all three literals x_1, x_2, x_3 are unassigned, so we set them to true (and $\neg x_1, \neg x_2, \neg x_3$ to false). When we next visit S_-, every literal is already assigned, so nothing changes. The constructed assignment is $x_1 = x_2 = x_3 = \text{true}$. Verification: clause 1 is $(\neg x_1 \vee x_2) = (F \vee T) = T$; clause 2 is $(\neg x_2 \vee x_3) = (F \vee T) = T$; clause 3 is $(x_1 \vee \neg x_3) = (T \vee F) = T$. All clauses satisfied.

18.5 Summary

In this chapter, we explored two important subclasses of Boolean formulas, Horn and 2-SAT, that form the boundary between tractable and intractable satisfiability problems.

Class	Restriction	Sequential Time	Approach
Horn SAT	≤ 1 positive literal/clause	$O(n+m)$	LLP (lattice-linear)
2-SAT	≤ 2 literals/clause	$O(n+m)$	Implication graph + SCC
3-SAT	≤ 3 literals/clause	NP-complete	—

We showed that Horn formulas are *lattice-linear predicates*, and therefore admit a solution through the LLP framework that yields the least satisfying assignment. The meet-closure property (Corollary 18.6) is a direct consequence of lattice-linearity and has important implications: the intersection of any two models of a Horn formula is again a model.

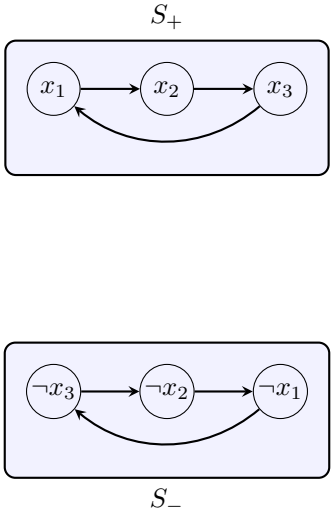


Figure 18.3: The two SCCs of the implication graph for the formula in Example 18.13. There are no edges between S_+ and S_- in the SCC-contracted graph, so either is a valid topological order. The reverse-topological-order rule of Algorithm 2-SAT sets every literal in the first-visited SCC to true.

For 2-SAT formulas, we presented the implication-graph construction and demonstrated that satisfiability can be checked through the detection of strongly connected components. This approach yields a linear-time sequential algorithm.

It is instructive to compare the two approaches. Horn satisfiability exploits the *algebraic* structure of the solution space (it forms a meet-semilattice), while 2-SAT exploits the *graph-theoretic* structure of the implication relationships. The general SAT problem becomes NP-complete precisely when neither structure is present: with 3 or more literals per clause and no restriction on positive literals, neither the lattice-linear approach nor the implication-graph approach applies.

18.6 Problems

1. **(Horn SAT Satisfiability Example)** Is the following Horn SAT formula satisfiable? If so, give the least satisfying assignment.
$(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_1) \wedge (\neg x_3 \vee \neg x_1) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2)$

2. **(Efficient Horn SAT Algorithm)** Give an efficient implementation of Algorithm LLP-HornSAT. In particular, give efficient data structures to represent Horn implications and pure negative clauses. Your algorithm should be linear in the length of the Horn formula.

3. **(Renamable Horn Formula)** A formula is a Renamable Horn formula if, by flipping the polarity of certain variables, the formula can be transformed to a Horn formula. For example, $(\neg x_1 \vee \neg x_2 \vee \neg x_4) \wedge (x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee \neg x_4) \wedge (x_1 \vee \neg x_3)$ is not a Horn formula. However, renaming x_1 as $\neg x_1$ and x_3 as $\neg x_3$, we get a Horn formula. Give an

algorithm to determine whether the given formula is a Renamable Horn formula. (Hint: Can you come up with a 2-SAT formula that is true iff the given formula is a renamable Horn formula).

4. **(Direct Proof Of Horn Corollary)** Prove Corollary 18.6 without invoking the lattice-linearity of Horn clauses.

5. **(Reachability As 2-SAT)** Consider a directed graph with n nodes that includes two distinguished nodes s and t. Show that there exists a 2-SAT formula B on n variables such that the formula is unsatisfiable iff t is reachable from s.

6. **(Forced Variables In 2-SAT)** Suppose a 2-SAT formula B is satisfiable. Then, all satisfying assignments of B must have x_j as true iff there exists a path from $\neg x_j$ to x_j in the implication graph.

7. **(2-SAT SCC Characterization)** (a) Show that there exists a satisfying assignment to a 2-SAT formula B iff there is no variable x_i such that $\neg x_i$ is reachable from x_i and x_i is reachable from $\neg x_i$ in the implication graph I constructed from B. (b) Show that every literal in any strongly connected component of the implication graph must be assigned the same value, *true* or *false*.

8. (**Counting satisfying assignments.**) Show that counting the number of satisfying assignments of a 2-SAT formula is #P-complete, even though *deciding* satisfiability is polynomial. (Hint: reduce from #MONOTONE-2-SAT or use a reduction from counting independent sets in a graph.)

9. (**Horn clause and databases.**) A relational database stores facts (e.g., "Alice is Bob's parent") and rules (e.g., "if X is a parent of Y and Y is a parent of Z, then X is a grandparent of Z"). Show that the problem of determining whether a given fact can be derived from a set of ground facts and rules can be modeled as a Horn satisfiability problem.

10. (**Maximum satisfiable Horn clauses.**) Given a Horn formula, the MAX-HORN-SAT problem asks for an assignment that satisfies the maximum number of clauses. Show that this problem is NP-hard. (Hint: reduce from the vertex cover problem.)

11. (**XOR-SAT.**) An XOR-SAT formula is a conjunction of XOR-clauses, where each clause is the XOR (exclusive-or) of literals. For example, $(x_1 \oplus x_2) \wedge (x_2 \oplus \neg x_3)$. Show that XOR-SAT can be solved in polynomial time. (Hint: model each clause as a linear equation over $GF(2)$ and use Gaussian elimination.)

12. **(Incremental HornSAT)** A Horn formula B has been solved by LLP-HornSAT, yielding the least satisfying assignment G^*. A new Horn clause C_{m+1} is appended. Update G^* in time $O(|C_{m+1}|)$ when possible, or identify $B \wedge C_{m+1}$ as unsatisfiable.

13. **(Constrained HornSAT With Forced Literals)** A subset F of literals must be true in the output (e.g., user requirements that override the formula). Compute the least satisfying assignment subject to F, or declare infeasibility.

14. **(Fair Horn / Balanced Truth Counts)** Among all satisfying assignments of a Horn formula, find one with the most balanced number of true literals (minimising $|\#\text{true} - n/2|$).

15. **(Enumerate All Satisfying Assignments of a Horn Formula)** Given a satisfiable Horn formula B, enumerate all satisfying assignments with $O(n)$ amortised time per assignment.

16. **(Online HornSAT)** Horn clauses arrive in adversarial order. Maintain the least satisfying assignment (or declare unsatisfiability) incrementally. Show that, after $O(n)$ one-time initialization, the per-clause cost is $O(1)$ amortised.

18.7 Bibliographic Remarks

Horn formulas, a subclass of propositional formulas in which each clause has at most one positive literal, are central to efficient satisfiability algorithms. [DG84] present a linear-time algorithm to test the satisfiability of a Horn formula, using unit propagation. The 2-SAT problem, which determines the satisfiability of a CNF formula with at most two literals per clause, is solvable in linear time. [APT79] introduce a linear-time algorithm using implication graphs and strongly connected components.

Horn clauses form the logical foundation of the Prolog programming language and Datalog, a query language for deductive databases [CGT89].

The q-Horn class, which generalizes both Horn and 2-SAT, was introduced by Boros, Crama, and Hammer [BCH90] and is also solvable in polynomial time. Renamable Horn formulas were studied by Lewis [Lew78]. For a comprehensive treatment of the complexity landscape of satisfiability, the reader is referred to the Handbook of Satisfiability [BHvMW09].

Chapter 19

Algorithms in Number Theory

19.1 Introduction

Number theory is one of the oldest branches of mathematics and a rich source of algorithmic problems. Classical questions — computing the greatest common divisor, solving systems of congruences, counting integers coprime to a given number — date back more than two thousand years, yet they remain central in modern computer science. They underlie public-key cryptography (RSA, Diffie–Hellman, elliptic curve cryptography), error-correcting codes, hashing, random number generation, and secret sharing schemes.

In this chapter, we revisit several standard number-theoretic computations through the lens of Lattice-Linear Predicates (LLP). The key observation is that the set of all natural numbers forms a distributive lattice under the *divides* relation: we define $x \leq y$ for two natural numbers x and y if x divides y. In this (infinite) lattice, the bottom element is 1 since it divides all numbers. The join of any two elements is given by their *least common multiple*, and the meet of any two elements is given by their *greatest common divisor*. This lattice-theoretic viewpoint lets us re-derive classical algorithms as instances of the generic LLP framework, and it also suggests natural parallel and distributed implementations.

We proceed as follows. We first present two lattice-linear formulations of the GCD problem, one that reduces to Euclid's classical algorithm and one that searches in the opposite direction, followed by the extended Euclidean algorithm for computing Bézout coefficients. We then apply LLP to the Chinese Remainder Theorem, yielding a polynomial-time CRT solver built on the extended Euclidean subroutine. Finally, we study several multiplicative arithmetical functions — the Möbius function μ, Euler's totient ϕ, and a few related functions — and prove some classical summation identities by viewing numbers as elements of a poset of prime powers.

Section 19.2 introduces the first GCD algorithm, LLP-GCD-Descend, which is a descending LLP algorithm that starts at $G = [a, b]$ and repeatedly reduces entries using Euclidean mod operations; we analyze its sequential complexity and give a worked example. Section 19.3 presents the complementary ascending algorithm LLP-GCD-Ascend, which starts at $G = [1, 1]$ and grows each entry using ceiling operations. Section 19.4 extends LLP-GCD-Descend into the extended Euclidean algorithm, which additionally produces Bézout coefficients (x, y) such

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

that $ax + by = \gcd(a, b)$. Section 19.5 applies the LLP framework to the Chinese Remainder Theorem, giving a polynomial-time algorithm based on LLP-ExtGCD together with its complexity analysis. Section 19.6 views numbers as a distributive lattice of prime powers and uses this view to prove classical summation identities for the Möbius function μ, Euler's totient ϕ, and several other multiplicative arithmetical functions.

19.2 Greatest Common Divisor (GCD) Algorithm

Let a and b be two natural numbers, both at least 1. The set of common divisors of a and b forms a distributive lattice under divisibility, with 1 at the bottom and $\gcd(a, b)$ at the top. We compute $\gcd(a, b)$ by maintaining an array G of size 2 of natural numbers that is reduced toward the GCD.

We start with $G[1] := a$ and $G[2] := b$. The LLP algorithm reduces the values in G until both entries equal $\gcd(a, b)$. The predicate is

$$B \equiv (G[1] \text{ divides } a) \wedge (G[2] \text{ divides } b) \wedge (G[1] = G[2]).$$

When B holds, $G[1] = G[2]$ is a common divisor of a and b. The LLP algorithm finds the greatest such value, which is $\gcd(a, b)$.

We show that the algorithm preserves every common divisor of a and b.

Lemma 19.1 *Let G be the current state of the LLP loop. Every common divisor of a and b is also a common divisor of $G[1]$ and $G[2]$. In particular, $\gcd(a, b)$ divides both $G[1]$ and $G[2]$.*

Proof: We argue by induction on the number of advance steps. Initially $G = [a, b]$, so every common divisor of a and b trivially divides both entries.

For the inductive step, suppose the invariant holds at G and the algorithm advances $G[j]$ via the comparator $G[i]$ with $G[j] > G[i]$ (where $\{i, j\} = \{1, 2\}$). The advance has two cases.

Case 1: $G[i]$ divides $G[j]$. The advance sets $G[j] := G[i]$. Let d be any common divisor of a and b. By the inductive hypothesis d divides $G[i]$. The new $G[j]$ equals $G[i]$, which d divides.

Case 2: $G[i]$ does not divide $G[j]$. The advance sets $G[j] := G[j] \bmod G[i]$. Let d be any common divisor of a and b. By the inductive hypothesis d divides both $G[i]$ and $G[j]$, hence d divides any integer linear combination of $G[i]$ and $G[j]$, including $G[j] - \lfloor G[j]/G[i] \rfloor \cdot G[i] = G[j] \bmod G[i]$.

By induction, the invariant holds throughout the execution.

■

If $G[1] \neq G[2]$, the forbidden index is the larger of the two; the advance reduces that entry while preserving every common divisor of a and b (by Lemma 19.1). When the algorithm terminates with $G[1] = G[2] = d$, the invariant gives $\gcd(a, b) \mid d$. Since each advance preserves $\gcd(G[1], G[2])$, we have $d = \gcd(d, d) = \gcd(G[1], G[2]) = \gcd(a, b)$.

Algorithm LLP-GCD-Descend: Finding the greatest common divisor of two numbers

input: a, b: natural numbers
output: $\gcd(a, b)$
1 G: array[1..2] of int initially $G[1] := a$; $G[2] := b$;
2 **forbidden**(j): $\exists i : G[j] > G[i]$
3 $\Rightarrow$ **advance**: **if** $G[j] \bmod G[i] = 0$ **then** $G[j] := G[i]$ **else** $G[j] := G[j] \bmod G[i]$;
4 **return** $G[1]$;

The algorithm terminates because each advance strictly reduces a positive integer, and both entries stay at least 1.

Example 19.2 Let $a = 30$ and $b = 18$, so $\gcd(a, b) = 6$. Algorithm LLP-GCD-Descend starts with $G = [30, 18]$.

- $G[1] = 30 > G[2] = 18$, and 18 does not divide 30, so $G[1] := 30 \; mod \; 18 = 12$. Now $G = [12, 18]$.
- $G[2] = 18 > G[1] = 12$, and 12 does not divide 18, so $G[2] := 18 \; mod \; 12 = 6$. Now $G = [12, 6]$.
- $G[1] = 12 > G[2] = 6$, and 6 divides 12, so $G[1] := 6$. Now $G = [6, 6]$.

No index is forbidden, and the algorithm returns $G[1] = 6 = \gcd(30, 18)$. This is exactly the trace of Euclid's algorithm.

Let $M = \max(a, b)$. By the standard analysis of Euclid's algorithm, two consecutive advances of $G[1]$ and $G[2]$ reduce $\max(G[1], G[2])$ by at least a factor of two. Hence the number of advance steps is $O(\log M)$, each advance is $O(1)$ arithmetic operations on integers of bit length $O(\log M)$, so

$$T_{\text{seq}}(\text{LLP-GCD-Descend}) \;=\; O(\log M),$$

and $O((\log M)^2)$ bit operations if we charge each arithmetic operation its actual bit cost.

From two numbers to n numbers. For an array A of n natural numbers, $\gcd(A)$ can be computed by the standard *generic reduce* pattern: combine the entries pairwise using the two-number algorithm above. Sequentially this performs $n - 1$ pairwise GCDs in $O(n \log M)$ arithmetic operations.

Least common multiple. Because gcd and lcm are dual under the divisibility lattice and satisfy the elementary identity

$$\operatorname{lcm}(a, b) \;=\; \frac{a \cdot b}{\gcd(a, b)},$$

the GCD algorithm above yields a polynomial-time LCM algorithm at no extra asymptotic cost: one GCD computation, one multiplication, and one division suffice for two numbers, and

an array's LCM is obtained by the same generic-reduce pattern. We therefore do not develop a separate LLP loop for LCM.

19.3 An alternative GCD algorithm

We now give an alternative GCD algorithm based on lattice-linearity. Let $a, b \geq 1$ be the two given natural numbers and let $d = \gcd(a, b)$. Dividing a and b by d yields the quotient pair $(a/d,\ b/d)$, again natural numbers. Finding d is therefore equivalent to finding the minimum vector G of size 2 such that $A[i]/G[i]$ is the same value for both i, where $A = [a, b]$.

We formulate this as the predicate

$$B_{gcd}(G) \ \equiv \ A[1]/G[1] \ = \ A[2]/G[2],$$

which is equivalent to $\forall j : G[j] \geq \max_i\{A[j]/A[i] * G[i]\}$. The right-hand side is a monotone function, so the predicate is lattice-linear. We apply the *LLP* algorithm with $forbidden(G, j) \equiv G[j] < \max_i\{A[j]/A[i] * G[i]\}$ and $\alpha(G, j) = \lceil \max_i\{A[j]/A[i] \cdot G[i]\}\rceil$.

Algorithm LLP-GCD-Ascend: Finding the greatest common divisor of two numbers

input: a, b: natural numbers; let $A := [a, b]$
output: $\gcd(a, b) = A[1]/G[1]$

```
1 G: array[1..2] of int initially G[1] := 1; G[2] := 1;
2 forbidden(j): ∃i : G[j] · A[i] < A[j] · G[i]
3    ⇒ advance: G[j] := (A[j] · G[i] + A[i] − 1)/A[i];
4 return A[1]/G[1];
```

The above algorithm was mechanically derived from the LLP algorithm.

Example 19.3 [LLP-GCD-Ascend on $a = 10$, $b = 15$] Let $A = [10, 15]$. Initially $G = [1, 1]$.

- $G[2] \cdot A[1] < G[1] \cdot A[2]$ (i.e., $10 < 15$), so index 2 is forbidden. Update $G[2] := \lceil (A[2]/A[1]) \cdot G[1]\rceil = \lceil 15/10\rceil = 2$. Now $G = [1, 2]$.
- $G[1] \cdot A[2] < G[2] \cdot A[1]$ (i.e., $15 < 20$), so index 1 is forbidden. Update $G[1] := \lceil (A[1]/A[2]) \cdot G[2]\rceil = \lceil 20/15\rceil = 2$. Now $G = [2, 2]$.
- $G[2] \cdot A[1] < G[1] \cdot A[2]$ (i.e., $20 < 30$), so index 2 is forbidden. Update $G[2] := \lceil (A[2]/A[1]) \cdot G[1]\rceil = \lceil 30/10\rceil = 3$. Now $G = [2, 3]$.

At $G = [2, 3]$, we have $A[1]/G[1] = 10/2 = 5 = 15/3 = A[2]/G[2]$, so the predicate holds and the algorithm terminates. It returns $A[1]/G[1] = 5$, the gcd of 10 and 15.

Let $M = \max(a, b)$, $d = \gcd(a, b)$, and $Q = M/d$. Each advance increases $G[j]$ geometrically, so the number of advances is $O(\log Q) = O(\log(M/d))$, each costing $O(1)$ arithmetic operations. Hence

$$T_{\text{seq}}(\text{LLP-GCD-Ascend}) \ = \ O(\log(M/d))$$

arithmetic operations on integers of bit length $O(\log Q)$.

From two numbers to n numbers. The same pairwise-reduce pattern applies: sequentially $O(n \log(M/d))$ arithmetic operations.

19.4 Extended GCD Algorithm

The *extended Euclidean algorithm* strengthens LLP-GCD-Descend by computing, in addition to the greatest common divisor d of two numbers a and b, integer coefficients x and y such that

$$ax + by = d = \gcd(a, b).$$

Such a pair (x, y) is called a Bézout pair for (a, b). The existence of Bézout pairs is guaranteed by Bézout's identity and is essential for many applications, including: computing multiplicative inverses modulo m (needed in the Chinese Remainder Theorem and in RSA key generation), solving linear Diophantine equations of the form $ax + by = c$, and lifting relations in quotient rings.

We cast the extended algorithm as a lattice-linear one with auxiliary Bézout coefficients. We maintain two pairs of integers (x_1, y_1) and (x_2, y_2) alongside the array G, with the invariant

$$I \;\equiv\; G[1] \;=\; x_1 \cdot a \;+\; y_1 \cdot b \quad \text{and} \quad G[2] \;=\; x_2 \cdot a \;+\; y_2 \cdot b.$$

Initially, $G[1] = a$ and $G[2] = b$, with $(x_1, y_1) = (1, 0)$ and $(x_2, y_2) = (0, 1)$, so the invariant holds. Whenever LLP-GCD-Descend would replace $G[j]$ by $G[j] - q \cdot G[i]$ for some integer quotient q, we perform the same subtraction on the Bézout pair:

$$G[j] \leftarrow G[j] - q \cdot G[i], \qquad (x_j, y_j) \leftarrow (x_j - q \cdot x_i,\; y_j - q \cdot y_i).$$

A direct computation shows that the invariant I is preserved: the new $G[j]$ equals $(x_j - q\,x_i)\,a + (y_j - q\,y_i)\,b$, which matches the new pair (x_j, y_j). Thus when LLP-GCD-Descend terminates with $G[1] = G[2] = d$, the pair (x_1, y_1) is a Bézout pair satisfying $ax_1 + by_1 = d$.

To make the reduction fit a uniform subtractive form, we set the quotient to $q = \lfloor G[j]/G[i] \rfloor$ when $G[i]$ does not divide $G[j]$, and $q = G[j]/G[i] - 1$ when $G[i]$ divides $G[j]$; in the latter case the new $G[j]$ equals $G[i]$, matching LLP-GCD-Descend's update rule. Both cases are captured by the single expression $q := \lfloor (G[j] - 1)/G[i] \rfloor$. We obtain the following algorithm:

Algorithm LLP-ExtGCD: Extended Euclidean algorithm

input: a, b: natural numbers
output: (d, x, y) such that $d = \gcd(a, b)$ and $a \cdot x + b \cdot y = d$

1 G: array[1..2] of int initially $G[1] := a;\; G[2] := b;$
2 $(x_1, y_1) := (1, 0); \quad (x_2, y_2) := (0, 1);$
3 **forbidden**(j): $\exists i : G[j] > G[i]$
4 $\quad \Rightarrow$ **advance**: int $q := \lfloor (G[j] - 1)/G[i] \rfloor$; $G[j] := G[j] - q \cdot G[i]$;
$(x_j, y_j) := (x_j - q \cdot x_i,\; y_j - q \cdot y_i)$;
5 **return** $(G[1],\; x_1,\; y_1)$;

Example 19.4 Let $a = 30$ and $b = 18$. Initially $G = [30, 18]$, $(x_1, y_1) = (1, 0)$, $(x_2, y_2) = (0, 1)$.

- $G[1] = 30 > G[2] = 18$, and 18 does not divide 30, so $q = \lfloor 30/18 \rfloor = 1$. Update: $G[1] := 30-18 = 12$, $(x_1, y_1) := (1-0,\ 0-1) = (1, -1)$. Check: $12 = 1\cdot 30+(-1)\cdot 18$. Now $G = [12, 18]$.
- $G[2] = 18 > G[1] = 12$, 12 does not divide 18, so $q = 1$. Update: $G[2] := 18-12 = 6$, $(x_2, y_2) := (0-1,\ 1-(-1)) = (-1, 2)$. Check: $6 = (-1)\cdot 30+2\cdot 18$. Now $G = [12, 6]$.
- $G[1] = 12 > G[2] = 6$, and 6 divides 12, so $q = 12/6 - 1 = 1$. Update: $G[1] := 12 - 6 = 6$, $(x_1, y_1) := (1-(-1),\ -1-2) = (2, -3)$. Check: $6 = 2\cdot 30+(-3)\cdot 18 = 60-54$. Now $G = [6, 6]$, and the algorithm terminates.

We return $(6,\ 2,\ -3)$: $\gcd(30, 18) = 6$, with Bézout coefficients $(x, y) = (2, -3)$.

For two input numbers a, b with $M = \max(a, b)$, the number of advance steps is the same as in classical Euclid, namely $O(\log M)$. Each advance performs a constant number of arithmetic operations on integers bounded by M in the G array and by M in the Bézout pairs (the coefficients produced by extended Euclid are bounded in absolute value by the larger input). Sequential time is therefore $O(\log M)$ arithmetic operations, or $O((\log M)^2)$ bit operations. In parallel, the algorithm has no useful parallelism beyond the two indices and runs in the same $O(\log M)$ depth.

For $n > 2$ inputs, one can either iterate LLP-ExtGCD on pairs (producing an n-entry Bézout decomposition in $O(n \log M)$ arithmetic operations), or run the full n-entry version of the algorithm with n Bézout coefficient vectors, at sequential cost $O(n^2 \log M)$ arithmetic operations.

19.5 Chinese Remainder Theorem

The Chinese Remainder Theorem (CRT) states that if $m_1, m_2, \ldots, m_r$ are pairwise coprime positive integers and $b_1, \ldots, b_r$ are any integers with $0 \le b_i < m_i$, then the system of congruences $x \equiv b_i \pmod{m_i}$ has a unique solution modulo $M = m_1 m_2 \cdots m_r$. The theorem is more than two thousand years old — it appears in Sun Tzu's *Sun Zi Suan Jing* (3rd century CE) — and it is one of the most useful structural results in elementary number theory. It underpins algorithms for modular arithmetic with large integers, secret sharing schemes, and efficient implementations of polynomial arithmetic.

We present a polynomial-time CRT algorithm built on LLP-ExtGCD: for any system of pairwise coprime moduli, the construction delivers the unique solution in time polynomial in the input bit size. The algorithm uses LLP-ExtGCD to compute modular inverses $M_j^{-1} \bmod m_j$ for each j, and the r inverses are independent and may be computed in parallel.

A polynomial-time CRT via LLP-ExtGCD

The classical constructive CRT, built on the extended Euclidean algorithm of Section 19.4, delivers the solution in $O(r \log M)$ arithmetic operations on integers bounded by M, equivalently

$O(r \log^2 M)$ bit operations using schoolbook multiplication. This is genuinely polynomial in the input bit size and is appropriate for arbitrary moduli, including the large moduli that appear in cryptographic applications.

The construction uses the LLP-ExtGCD of Section 19.4 as a subroutine. For each j, let $M_j = M/m_j$. Since the m_j are pairwise coprime, $\gcd(m_j, M_j) = 1$, and LLP-ExtGCD on (m_j, M_j) returns coefficients (s_j, y_j) with $s_j m_j + y_j M_j = 1$. Reducing modulo m_j gives $y_j M_j \equiv 1 \pmod{m_j}$, so y_j is the multiplicative inverse of M_j modulo m_j. The combination

$$x \;=\; \Big(\sum_{j=1}^{r} b_j \, M_j \, y_j\Big) \bmod M$$

satisfies $x \equiv b_j \pmod{m_j}$ for every j: in the sum, the j-th term reduces to b_j modulo m_j (since $M_j y_j \equiv 1$), and every other term contains the factor $M_k = M/m_k$ for some $k \neq j$, which is divisible by m_j and so vanishes mod m_j.

Algorithm CRT-via-ExtGCD: Polynomial-time CRT via LLP-ExtGCD

input: m: array[1..r] of pairwise-coprime int; b: array[1..r] of int with $0 \le b[j] < m[j]$
output: the unique $x \in \{0, 1, \ldots, M-1\}$ with $\forall j : x \equiv b[j] \pmod{m[j]}$, where $M = \prod_j m[j]$

```
1 M := ∏_{j=1}^{r} m[j];
2 forall j ∈ [1..r] in parallel do
3 |   M_j := M/m[j];
4 |   (s_j, y_j) := LLP-ExtGCD(m[j], M_j) ;        // satisfies s_j m[j] + y_j M_j = 1
5 end
6 x := ( Σ_{j=1}^{r} b[j] · M_j · y_j ) mod M;
7 return x;
```

Each LLP-ExtGCD call costs $O(\log M)$ arithmetic operations (or $O(\log^2 M)$ bit operations with schoolbook multiplication); the final summation does r multiplications and additions on integers of bit length $O(\log M)$. Total sequential cost is $O(r \log^2 M)$ bit operations. The r LLP-ExtGCD calls are independent and may all run in parallel, yielding parallel time $O(\log M)$ arithmetic operations on r processors.

Example 19.5 For the system $x \equiv 2 \pmod 3$, $x \equiv 3 \pmod 5$, $x \equiv 2 \pmod 7$, we have $M = 105$, and $M_1 = 35$, $M_2 = 21$, $M_3 = 15$. LLP-ExtGCD gives $35 \cdot 2 \equiv 1 \pmod 3$, so $y_1 = 2$; $21 \cdot 1 \equiv 1 \pmod 5$, so $y_2 = 1$; $15 \cdot 1 \equiv 1 \pmod 7$, so $y_3 = 1$. Hence

$$x \;=\; (2{\cdot}35{\cdot}2 + 3{\cdot}21{\cdot}1 + 2{\cdot}15{\cdot}1) \bmod 105 \;=\; (140+63+30) \bmod 105 \;=\; 233 \bmod 105 \;=\; 23,$$

the unique CRT solution for this system. Indeed $23 \equiv 2 \pmod 3$, $23 \equiv 3 \pmod 5$, and $23 \equiv 2 \pmod 7$.

19.6 Viewing Numbers as a Distributive Lattice

The set of all natural numbers forms a distributive lattice under the divides relation. This distributive lattice (infinite) has the following representation. For every prime p, we have a chain of elements $p^0, p^1, p^2, \ldots$. This infinite poset has as many chains as the number of distinct primes. There is a 1-1 correspondence between the lattice of natural numbers and the cross product of all chains in the poset (by the unique factorization of numbers). We can go from any element n in the number lattice to an element in the cross product by choosing $k+1$ elements from the chain for the prime p if k is the largest integer such that p^k divides n.

Let us define some arithmetical functions on the set of natural numbers. We are interested in functions that can be defined on the poset representation of natural numbers, i.e., the function is defined only for the prime powers. The value of a function on any number is simply the product of its value on each of the prime powers.

Definition 19.6 (Multiplicative Functions) *We call an arithmetical function f* multiplicative *if it is not a zero function and $f(mn)$ equals $f(m)f(n)$ whenever* $\gcd(m,n)$ *equals* 1.

Note that we require that $f(mn)$ equal $f(m)f(n)$ only when $\gcd(m,n)$ equals 1. If we drop the requirement that the gcd be 1, then the function is called *completely* multiplicative.

We now introduce some multiplicative functions.

Definition 19.7 (Möbius function) *The Möbius function μ is defined on any prime power p^k as 1 if $k = 0$, -1 if $k = 1$, and 0 otherwise.*

Here, μ is defined to be multiplicative. It follows that if $n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$, then $\mu(n)$ equals $(-1)^k$ if $\forall i \in [k] : a_i = 1$ and $\mu(n)$ equals 0 otherwise.

Example 19.8 A few values of μ: $\mu(1) = 1$ (empty product of primes); $\mu(2) = -1$, $\mu(3) = -1$, $\mu(5) = -1$ (single primes); $\mu(6) = \mu(2)\mu(3) = (-1)(-1) = 1$ (two distinct primes); $\mu(12) = \mu(4)\mu(3) = 0 \cdot (-1) = 0$ (since $4 = 2^2$); $\mu(30) = \mu(2)\mu(3)\mu(5) = (-1)^3 = -1$ (three distinct primes); $\mu(100) = \mu(4)\mu(25) = 0 \cdot 0 = 0$.

We now show a simple formula for the divisor sum of $\mu(n)$.

Theorem 19.9 *For any $n > 1$, $\sum_{d|n} \mu(d) = 0$.*

Proof: Let $n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$. It is easy to see that d divides n iff $d = p_1^{b_1} p_2^{b_2} \cdots p_k^{b_k}$, where for all $i \in [k] : b_i \leq a_i$. Consider the poset $P(n)$. It is easy to verify that $\sum_{d|n} \mu(d)$ is the sum of all cross-products of chains in $P(n)$. This sum is also equal to the product of sum of $\mu(d)$ where d is a prime power. However, $\sum_{d|n,\, d \text{ is a power of } p_i} \mu(d)$ is zero because for any prime p that divides n, $\mu(p^0) = 1, \mu(p^1) = -1$, and $\mu(p^k) = 0$ for $k \geq 2$. Hence, $\sum_{d|n} \mu(d) = 0$.

■

Example 19.10 Let $n = 12 = 2^2 \cdot 3$. Its divisors are $\{1, 2, 3, 4, 6, 12\}$, and $\sum_{d|12} \mu(d) = \mu(1) + \mu(2) + \mu(3) + \mu(4) + \mu(6) + \mu(12) = 1 + (-1) + (-1) + 0 + 1 + 0 = 0$, as the theorem predicts.

We now show another multiplicative arithmetical function defined on the set of natural numbers.

Definition 19.11 (Euler's Totient function) *Euler's totient function ϕ is defined on any prime power p^k for $k > 0$ as $p^k - p^{k-1}$. Also, $\phi(1)$ is defined as 1.*

For example, $\phi(3^2) = 3^2 - 3^1 = 9 - 3 = 6$. The function ϕ can be extended to any natural number n by using the multiplicative property. We now show that

Lemma 19.12 *For any $n > 1$, $\phi(n)$ equals the number of positive integers less than or equal to n which are relatively prime to n.*

Proof: Let $n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$ for some $k \geq 1$. When k equals 1, $n = p_1^{a_1}$. There are $p_1^{a_1}$ positive integers less than or equal to n. Of these, $p_1^{a_1 - 1}$ are multiples of p_1 and hence not coprime to n. Therefore, $\phi(n)$ equals the number of positive integers less than or equal to n that are relatively prime to n. Let A_i denote the numbers that are relatively prime to $p_i^{a_i}$. Now, suppose that $k > 1$. From the multiplicative property, $\phi(n) = \phi(p_1^{a_1})\phi(p_2^{a_2}) \cdots \phi(p_k^{a_k})$. For each i, let A_i be the set of integers in $\{1, \ldots, p_i^{a_i}\}$ that are coprime to $p_i^{a_i}$; we have $|A_i| = \phi(p_i^{a_i})$. By the Chinese Remainder Theorem (since $p_1^{a_1}, \ldots, p_k^{a_k}$ are pairwise coprime), the map $x \mapsto (x \bmod p_1^{a_1}, \ldots, x \bmod p_k^{a_k})$ is a bijection from $\{1, \ldots, n\}$ to $\{1, \ldots, p_1^{a_1}\} \times \cdots \times \{1, \ldots, p_k^{a_k}\}$, and $\gcd(x, n) = 1$ iff $\gcd(x, p_i^{a_i}) = 1$ for every i. Hence the number of integers coprime to n is $|A_1| \cdots |A_k| = \phi(p_1^{a_1}) \cdots \phi(p_k^{a_k}) = \phi(n)$.

■

Example 19.13 A few values of ϕ: $\phi(1) = 1$, $\phi(2) = 1$, $\phi(3) = 2$, $\phi(4) = 4 - 2 = 2$, $\phi(5) = 4$, $\phi(6) = \phi(2)\phi(3) = 1 \cdot 2 = 2$, $\phi(10) = \phi(2)\phi(5) = 1 \cdot 4 = 4$, $\phi(12) = \phi(4)\phi(3) = 2 \cdot 2 = 4$. For instance, the integers in $\{1, 2, \ldots, 12\}$ that are coprime to 12 are $\{1, 5, 7, 11\}$, and indeed $|\{1, 5, 7, 11\}| = 4 = \phi(12)$.

We now show that

Theorem 19.14 *For any $n > 1$, $\sum_{d|n} \phi(d) = n$.*

Proof: Let $n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$ for some $k \geq 1$. Clearly, $\sum_{d|n} \phi(d) = \prod_{i=1}^{k} \sum_{j=0}^{a_i} \phi(p_i^j)$. Since $\phi(p_i^j) = p_i^j - p_i^{j-1}$ for $j \geq 1$ and $\phi(p_i^0) = 1$, by the telescoping argument we get $\sum_{j=0}^{a_i} \phi(p_i^j) = p_i^{a_i}$. Hence, $\sum_{d|n} \phi(d) = \prod_{i=1}^{k} p_i^{a_i} = n$.

■

Example 19.15 Let $n = 12$. Its divisors are $\{1, 2, 3, 4, 6, 12\}$, and $\sum_{d|12} \phi(d) = \phi(1) + \phi(2) + \phi(3) + \phi(4) + \phi(6) + \phi(12) = 1 + 1 + 2 + 2 + 2 + 4 = 12 = n$, as the theorem predicts.

Some additional multiplicative functions are defined below.

1. The arithmetical function N is defined as $N(p^k) = p^k$.
2. The arithmetical function *Identity*, I, is defined as $I(p^k) = 1$ if $k = 0$ and 0 otherwise.
3. The arithmetical function *Unit*, u, is defined as $u(p^k) = 1$ for all p, k.
4. The arithmetical function λ is defined as $\lambda(p^k) = (-1)^k$ for all k.

The following Lemma gives the summation results for these functions.

Lemma 19.16 *For any $n > 1$,*
(a) $\sum_{d|n} N(d) = \prod_{p^k \| n} (p^{k+1} - 1)/(p - 1)$, where $p^k || n$ denotes that p^k is the largest power of p dividing n.
(b) $\sum_{d|n} I(d) = 1$.
(c) $\sum_{d|n} u(d) =$ the number of divisors of n.
(d) $\sum_{d|n} \lambda(d) = 1$ if n is a square and 0 otherwise.

Proof: Since all these functions are multiplicative, we will simply compute them for $n = p^k$.
(a) $\sum_{d|p^k} N(d) = (p^{k+1} - 1)/(p - 1)$ since we have a geometric series. From multiplicative property, the result follows.
(b) $\sum_{d|p^k} I(d) = 1$ since only $I(1) = 1$. It is 0 for all $k \geq 1$. By multiplicative property, we get that $\sum_{d|n} I(d) = 1$.
(c) $\sum_{d|p^k} u(d) = k + 1$. By invoking the multiplicative property, we get that $\sum_{d|n} u(d) =$ the number of divisors of n.
(d) $\sum_{d|p^k} \lambda(d) = 1$ if k is even and 0 otherwise. Therefore, $\sum_{d|n} \lambda(d) = 1$ if for each prime the corresponding exponent is even and 0 otherwise. Thus, $\sum_{d|n} \lambda(d) = 1$ if n is a square and 0 otherwise.

■

Example 19.17 Let $n = 12 = 2^2 \cdot 3$. The divisors are $\{1, 2, 3, 4, 6, 12\}$. Then: (a) $\sum_{d|12} N(d) = 1 + 2 + 3 + 4 + 6 + 12 = 28$, which agrees with $((2^3 - 1)/(2 - 1)) \cdot ((3^2 - 1)/(3 - 1)) = 7 \cdot 4 = 28$. (c) $\sum_{d|12} u(d) = 6$, the number of divisors of 12. (d) Since 12 is not a square, $\sum_{d|12} \lambda(d) = 0$; indeed $\lambda(1) + \lambda(2) + \lambda(3) + \lambda(4) + \lambda(6) + \lambda(12) = 1 - 1 - 1 + 1 + 1 - 1 = 0$.

19.7 Summary

This chapter applied the Lattice-Linear Predicate framework to a range of number-theoretic computations: the greatest common divisor of two numbers (LLP-GCD-Descend and LLP-GCD-Ascend), the extended Euclidean algorithm (LLP-ExtGCD), and a polynomial-time Chinese Remainder Theorem solver (CRT-via-LLP-ExtGCD). The least common multiple is obtained directly from the GCD via $\operatorname{lcm}(a, b) = ab/\gcd(a, b)$ and so does not require a separate LLP loop. In each case the LLP framework delivered not just a sequential algorithm but also an immediately parallelizable one: since the feasibility predicate is lattice-linear, any set of forbidden indices can be advanced simultaneously, and correctness is independent of the order in which they are processed.

The algorithms illustrate two recurring themes. First, *ascending / descending duality*: every feasibility predicate has both an ascending and a descending lattice-linear formulation, giving rise to complementary algorithms with dual performance profiles. LLP-GCD-Descend descends from A using Euclidean mod, while LLP-GCD-Ascend ascends from the vector of ones using ceiling updates. Second, *separation of concerns*: the lattice-linear predicate captures the search target abstractly, while the advance rule supplies the arithmetic. The same LLP loop that descends in the divisibility lattice for GCD furnishes Bézout coefficients (LLP-ExtGCD) by augmenting the advance with a coefficient ledger, and the CRT solver reuses LLP-ExtGCD as a modular-inverse subroutine.

Algorithm	**Problem**	**Direction**	**Sequential Time**
LLP-GCD-Descend	GCD of n numbers	descending	$O(n \log M)$
LLP-GCD-Ascend	GCD of n numbers	ascending	$O(n \log(M/d))$
LLP-ExtGCD	GCD + Bézout pair	descending	$O(\log M)$ (for $n = 2$)
CRT-via-LLP-ExtGCD	CRT (polynomial-time)	—	$O(r \log^2 M)$ bit ops

Table 19.1: Summary of the LLP-based number-theory algorithms of this chapter. Here n is the number of inputs to the GCD problem, $M = \max_i A[i]$, $d = \gcd(A)$, r is the number of congruences in the CRT problem, and $m_1, \ldots, m_r$ are the moduli with product $M = \prod_j m_j$.

The second half of the chapter developed the poset-of-prime-powers representation of natural numbers and used it to prove the classical summation identities for the Möbius function, Euler's totient function, and several other multiplicative arithmetical functions. The proofs exploited the fact that each multiplicative function is determined by its values on prime powers, so that a divisor sum over n factors as a product of divisor sums over each prime-power chain.

Throughout the chapter, the LLP framework served as a unifying bridge: number-theoretic structure on one side (divisibility, residues, prime factorization), and a uniform parallel-algorithmic recipe on the other (identify the feasibility predicate, verify lattice-linearity, read off the algorithm). This mirrors the treatment of graph and optimization problems in earlier chapters and sets the stage for the remaining chapters of the book.

19.8 Problems

1. **(LLP GCD Descend Trace)** Trace Algorithm LLP-GCD-Descend on $A = [24, 36, 60, 84]$. Write down the value of G after each advance step, and verify that the algorithm terminates with $G[i] = \gcd(A) = 12$ for every i.

2. **(GCD Invariant Preservation)** Prove carefully that if $G[j] > G[i]$ and $G[i]$ does not divide $G[j]$, then the set of common divisors of $G[i]$ and $G[j]$ equals the set of common divisors of $G[i]$ and $G[j]$ *mod* $G[i]$. Use this to conclude that the invariant "every common divisor of A is a common divisor of G" is preserved by advance in LLP-GCD-Descend.

3. **(GCD Descend Step Bound)** Prove that Algorithm LLP-GCD-Descend performs at most $O(n \log M)$ advance steps, where $M = \max_i A[i]$. (Hint: show that the potential function $\Phi(G) = \sum_i \log G[i]$ strictly decreases by a constant after any sequence of advances by a fixed pair (i, j).)

4. **(GCD Identity Proofs)** Show that if $\gcd(a, b) = 1$, then $\gcd(a + b, ab) = 1$, and that $\gcd(a + b,\ a - b)$ divides 2.

5. **(CRT System Solution)** Use the Chinese Remainder Theorem to solve the system $x \equiv 1 \pmod 5$, $x \equiv 2 \pmod 7$, $x \equiv 3 \pmod{11}$. Consider a pseudopolynomial LLP approach (call it LLP-CRT): start with $G[j] = b_j$ for each j; repeatedly find a forbidden index j where $G[j] < \max_i G[i]$ and advance $G[j]$ to the next value $\equiv b_j \pmod{m_j}$ that is at least $\max_i G[i]$; terminate when all entries are equal. Trace LLP-CRT on this input, and compare with the classical constructive proof of CRT.

6. **(CRT Non-Coprime Moduli)** Adapt LLP-CRT to the case where the moduli are not pairwise coprime. Show that the algorithm still runs, but that it may fail to terminate. Characterize the inputs on which it does terminate, and give the resulting solution.

7. **(Euler Totient Multiplicativity)** Prove that Euler's totient function satisfies $\phi(mn) = \phi(m)\phi(n)$ whenever $\gcd(m, n) = 1$, by giving an explicit bijection between the residues coprime to mn and pairs of residues coprime to m and to n, respectively. (This is essentially the Chinese Remainder Theorem.)

8. **(Mobius Inversion Formula)** Möbius inversion formula. Let f and g be arithmetical functions. Prove that if $g(n) = \sum_{d|n} f(d)$ for all $n \geq 1$, then $f(n) = \sum_{d|n} \mu(d)\, g(n/d)$ for all $n \geq 1$. Use this to derive the formula $\phi(n) = \sum_{d|n} \mu(d)\, (n/d)$.

9. **(Mobius Totient Table)** Compute $\mu(n)$ and $\phi(n)$ for all n from 1 to 30 using the poset representation of numbers as prime powers. Verify numerically that $\sum_{d|n} \mu(d) = [n = 1]$ and $\sum_{d|n} \phi(d) = n$.

10. **(Sigma Divisor Sum)** Let $\sigma(n) = \sum_{d|n} d$ denote the sum of the divisors of n. Show that σ is multiplicative, and derive a closed-form formula for $\sigma(p^k)$. What does the Lemma in this chapter say about $\sigma(n)$ in terms of N?

11. **(Incremental GCD)** The greatest common divisor d of an array $A[1..n]$ has been computed. A new value a_{n+1} is appended. Update d in $O(\log M)$ arithmetic operations.

12. **(Dynamic CRT)** A CRT system $x \equiv b_j \pmod{m_j}$ has been solved. A new congruence $x \equiv b_{r+1} \pmod{m_{r+1}}$ (with m_{r+1} coprime to all existing m_j) is added. Update the solution.

13. **(Constrained GCD With Forbidden Divisors)** Find the greatest common divisor of a and b that is coprime to a given integer k (i.e., $\gcd(d, k) = 1$).

14. **(GCD With Tied Inputs)** Given an array $A[1..n]$ where multiple entries may be equal, compute both $\gcd(A)$ and the multiplicity of the GCD (how many $A[i]$ equal $\gcd(A)$).

15. **(Online GCD)** A stream of integers $a_1, a_2, \ldots$ arrives one at a time. Maintain $d_n = \gcd(a_1, \ldots, a_n)$ at each step, using $O(\log M)$ work per arrival.

19.9 Bibliographic Remarks

Euclid's algorithm for the greatest common divisor appears in Book VII of Euclid's *Elements*; it is one of the oldest algorithms known. Knuth's *The Art of Computer Programming*, Volume 2 [Knu97b], provides a thorough modern analysis of Euclid's algorithm, its extended form, and variants such as the binary GCD algorithm, and is the standard reference for algorithmic number theory. Complexity-theoretic aspects and randomized variants are discussed in detail by Shoup [Sho09] and by Bach and Shallit [BS96].

The Chinese Remainder Theorem, in its arithmetic form, was first stated by Sun Tzu in the third century CE. Its modern formulation and the extension to general principal ideal domains were developed in the nineteenth century; [Gau01] treats it systematically in his *Disquisitiones Arithmeticae*. Modern number-theory textbooks such as [Ros11] and [HW08] give accessible treatments with many applications, including secret sharing (Mignotte's and Asmuth–Bloom's schemes) and fast modular multiplication for large integers. Applications to cryptography, in particular the RSA cryptosystem, are discussed in Rivest, Shamir, and Adleman's original paper [RSA78].

Multiplicative arithmetical functions, including μ, ϕ, N, u, and λ, are classical. A systematic treatment can be found in Apostol [Apo76] and in Hardy and Wright [HW08]. The poset-of-prime-powers view that we emphasize in this chapter is implicit in many treatments of multiplicative functions and is closely related to Rota's theory of Möbius functions on posets [Rot64], which generalizes the number-theoretic Möbius function to arbitrary locally finite partially ordered sets. *Concrete Mathematics* by Graham, Knuth, and Patashnik [GKP94] offers a friendly introduction to these ideas at the undergraduate level.

The lattice-linear predicate framework used here to derive parallel versions of Euclid's algorithm and the Chinese Remainder Theorem is due to Garg [Gar20b]; it unifies a variety of parallel and distributed algorithms, including those for shortest paths, stable matching, and scheduling, within a single algorithmic paradigm.

Chapter 20

The Predicate Detection Problem

20.1 Introduction

Debugging and testing multithreaded software is widely acknowledged to be a hard task. Sometimes it takes a programmer days to locate a single bug, especially when the bug appears in one thread schedule but not in others. The current debugging and testing method for multithreaded programs is as follows. The programmer tries the program with multiple inputs in the hope of finding a faulty execution. However, the behavior of a multithreaded program depends not only on the external user input, but also on the thread schedule and the order in which locks are obtained by the program. It is easy for the testing process to miss a bug that arises with an alternate schedule. One of the fundamental problems in debugging these systems is to check if the user-specified condition exists in *any* global state of the system that can be reached by an alternative thread schedule. This problem, called *predicate detection*, takes a concurrent computation (in an online or offline fashion) and a condition that denotes a bug (for example, violation of a safety constraint), and outputs a schedule of threads that exhibits the bug if possible. Predicate detection is predictive because it generates inferred reachable global states from the computation; an inferred reachable global state might not be observed during the execution of the program, but is possible if the program is executed in a different thread interleaving.

This chapter is organized as follows. We first discuss the complexity of the predicate detection problem, establishing NP-completeness results for general boolean predicates. Section 20.3 presents a work-efficient algorithm for detecting conjunctive predicates using the State Rejection Graph technique. We show that detecting a conjunctive predicate at a given level is NP-complete. Finally, we address the problem of recognizing meet-closed and sublattice predicates, showing that these recognition problems are co-NP-complete. (Lattice-linear and regular predicates are strictly stronger than their structural counterparts meet-closed and sublattice, so the recognition problems for the former are at least as hard.)

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

20.2 Complexity of Predicate Detection Problem

In this section, we explore the complexity-theoretic results for predicate detection. It is not surprising that given a parallel program computation and a boolean predicate, it is computationally hard to determine if the execution went through a global state in which the predicate became true. We also show that given a boolean predicate b and an execution, determining whether b is meet-closed for that execution is co-NP-complete; the same holds for the sublattice property. The classes of *lattice-linear* and *regular* predicates — the algorithmic strengthenings of meet-closed and sublattice predicates respectively, in which an efficient advancement procedure is also available — are strict subclasses, so their recognition is at least as hard.

Since there are n processes, the total number of global states possible is m^n, where m is the number of state intervals at any process. Consider a boolean predicate B. Even when B is a boolean expression, and processes do not communicate, the problem of detecting *possibly*: B is NP-complete.

We show in this section that the problem of global predicate detection is NP-complete. In fact, we show that it is NP-complete even in the absence of messages between processes.

The global predicate detection problem is a decision problem. It can be written as:

> **Input instance**: a poset S of N sequences, a set of variables X partitioned into N subsets $X_1, \ldots, X_N$, and a predicate B defined on X.
> **Problem:** Determine whether there exists a consistent cut $G \in S$ such that $B(G)$ has the value true.

We now show:

Theorem 20.1 *The global predicate detection problem is* NP*-complete for polynomial-time computable predicates.*

Proof: First note that the problem is in NP. The verification that the cut is consistent can easily be done in polynomial time (for example, using vector clocks and examining all pairs of states from the cut). Therefore, if the predicate itself can be evaluated in polynomial time, then the detection of that predicate belongs to the set NP.

We show NP-completeness of the simplified predicate detection problem where all program variables are restricted to taking the values "true" or "false", and at most one variable from each X_i can appear in B. We reduce the satisfiability problem of a boolean expression (SAT) to the global predicate detection problem by constructing an appropriate poset.

The poset is constructed as shown in Figure 20.1. For each variable $u_i \in U$, we define a process P_i that hosts variable x_i (i.e., $X_i = \{x_i\}$). Let the sequence S_i consist of exactly two states. In the first state, x_i has the value false. In the second state, x_i has the value true.

It is easily verified that the predicate B is true for some cut in S if and only if the expression is satisfiable.

■

The above result shows that detection of a general global predicate is intractable even for simple distributed computations.

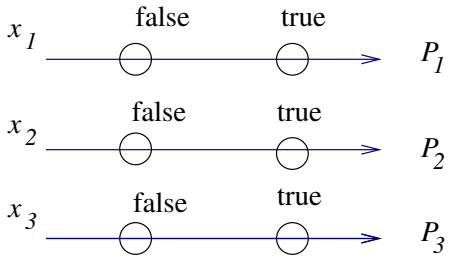


Figure 20.1: Transformation from SAT to global predicate detection

20.3 Detecting Conjunctive Predicates

In this section we describe a work-efficient algorithm to detect a conjunctive predicate $B = l_1 \wedge l_2 \wedge \cdots \wedge l_n$. To detect B, we need to determine if there exists a consistent global state G such that B is true in G. Note that given a computation on n processes each with m states, there can be as many as m^n possible consistent global states. Therefore, enumerating and checking the condition B for all consistent global states is not feasible. Since B is conjunctive, it is easy to show [GW92] that B is true iff there exists a set of states $s_1, s_2, ..., s_n$ such that (1) for all i, s_i is a state on P_i, (2) for all i, l_i is true on s_i and (3) for all i, j: $s_i \| s_j$. Our detection algorithm will either output such local states or guarantee that it is not possible to find them in the computation. When the global predicate B is true, there may be multiple G such that B holds in G. We are interested in algorithms that return the minimum G that satisfies B. The minimum G corresponds to the smallest counter-example to a programmer's understanding because B typically represents the violation of a safety constraint.

Our algorithm is based on the setting where the execution traces for all processes have been collected at one process. For example, in the centralized algorithm for conjunctive predicate detection, one process serves as a checker and collects the traces. All other processes involved in detecting the conjunctive predicate, referred to as application processes, check for local predicates during the computation. Each process P_i also maintains the vector clock algorithm. Whenever the local predicate of a process becomes true for the *first* time since the most recently sent message (or the beginning of the trace), it generates a debug message containing its local timestamp vector and sends it to the checker process [GW92].

The checker process uses queues of incoming messages to hold incoming local snapshots from application processes. We require that messages from an individual process be received in FIFO order. If the underlying system is non-FIFO, then sequence numbers can be attached with messages to ensure FIFO delivery. At the end of the computation, the checker process has a sequence of local states from each process where its local predicate is true. We now describe a sequential and a parallel algorithm to detect B on these traces. The sequential algorithm is an adaptation of the algorithm from [GW92]. We include it here because it is instrumental in understanding the algorithm. Moreover, the correctness of the algorithm is shown by assuming the correctness of the sequential algorithm.

The algorithm in Figure ConjunctiveDetection takes as input n traces each of size m as shown in Figure 20.2. Since conjunctive predicates are lattice-linear, we simply use LLP algorithm to detect them. We use G for the consistent global state. A local state $G[j]$ is forbidden if it is less than $G[i]$ for some i. Since we are interested only in global states where

the local predicate is true for all $G[i]$, we assume that for any i, we only consider $G[i]$ such that the local predicate is true in $G[i]$.

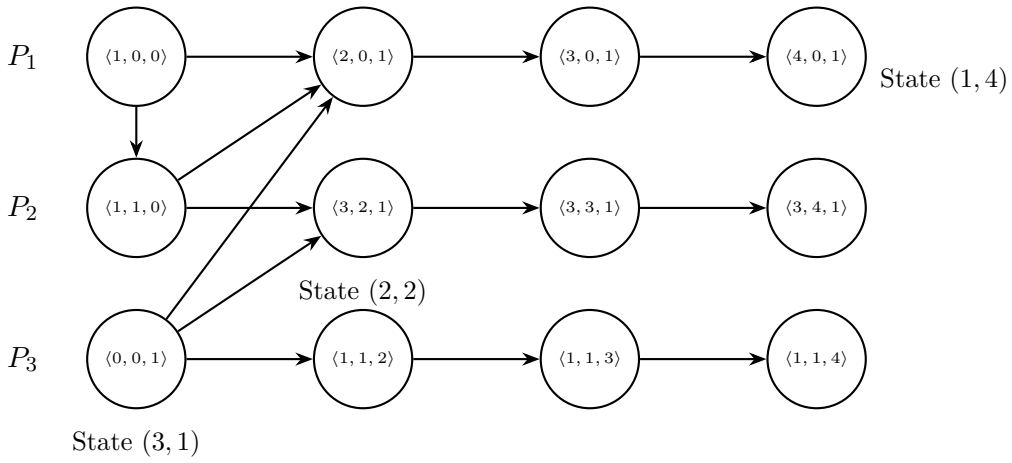


Figure 20.2: State-based model of a distributed computation.

Algorithm ConjunctiveDetection: Conjunctive predicate detection algorithm.

input: traces on n processes; $T[i]$: max states at P_i; each trace contains only states where the local predicate l_i is true
output: a consistent global state satisfying B, or null

```
1 G: array[1..n] of int initially 1;
2 forbidden(G, j, B) ≡ ∃i : G[j] → G[i];
3 while ∃j : forbidden(G, j, B) do
4 |   forall j such that forbidden(G, j, B) in parallel do
5 |   |   if G[j] = T[j] then return null;
6 |   |   else G[j] := G[j] + 1;
7 |   end
8 end
9 return G ;                                   // satisfying global state
```

20.4 Detecting a Conjunctive Predicate at the Given Level

We now show that, in general, asking for a conjunctive predicate on a particular level is NP-complete.

Theorem 20.2 *Given a distributed computation, deciding whether there exists a global state with k events satisfying a given conjunctive predicate is* NP*-complete.*

Algorithm ParallelCut: The ParallelCut algorithm to find the first consistent cut.

input: *states*: array$[1 \dots n][1 \dots m]$ of vectorClock
output: consistent global state as array $cut[1 \dots n]$, or null
`// Step 1: Create` F`, the set of initially rejected states`
F: array$[1 \dots n]$ of $\{0, 1\}$ initially 0;
forall $(i \in 1 \dots n,\ j \in 1 \dots n)$ **in parallel do**
  if $(i, 1) \rightarrow (j, 1)$ **then** $F[i] := 1$;
end
`// Step 2: Create` R`, the State Rejection Graph`
R: array$[(1 \dots n, 1 \dots m) \times (1 \dots n, 1 \dots m)]$ of $\{0, 1\}$;
forall $(i \in 1 \dots n,\ j \in 1 \dots m)$ **in parallel do**
  $R[(i, j), (i, j)] := 1$;
end
forall (i, j, i', j') *with* $j < m$ *and* $i \neq i'$ **in parallel do**
  if $(i', j') \rightarrow (i, j{+}1)$ **then** $R[(i, j), (i', j')] := 1$;
  else $R[(i, j), (i', j')] := 0$;
end
`// Step 3: Transitive closure of` R
$RT :=$ TransitiveClosure(R);
`// Step 4: Mark invalid states`
valid: array$[1 \dots n][1 \dots m]$ of $\{0, 1\}$ initially 1;
forall $(i \in 1 \dots n,\ i' \in 1 \dots n,\ j' \in 1 \dots m)$ **in parallel do**
  if $F[i] = 1$ *and* $RT[(i, 1), (i', j')] = 1$ **then** $valid[i'][j'] := 0$;
end
`// Step 5: Extract the first consistent global state`
cut: array$[1 \dots n]$ of $0 \dots m$ initially 0;
forall $(i \in 1 \dots n,\ j \in 1 \dots m)$ **in parallel do**
  if $valid[i][j] \neq 0$ *and* $(j = 1$ *or* $valid[i][j{-}1] = 0)$ **then**
    $cut[i] := j$;
  end
end
forall $i \in 1 \dots n$ **in parallel do**
  if $cut[i] = 0$ **then return** null;
end
return *cut*;

Proof: We first show that the problem is in NP. The global state itself provides a succinct certificate. We can check that all local predicates are true in that global state and that the global state is at level k.

For hardness, we use the subset sum problem. Given a subset problem on n positive integers, $x_1, x_2, \ldots, x_n$ with the requirement to choose a subset that adds up to k, we create a computation on n processes as follows. Each process P_i has x_i events. The local predicate on P_i is true initially and also after it has executed x_i events. Thus, the local predicate is true at each process exactly twice (the local predicate is false at all intermediate states). The problem asks us if there is a global state with k events in which all local predicates are true. Since the local predicate is true only at the initial and final states, such a global state, if it exists, would choose for every process either the initial local state or the final local state. All the final states that are chosen correspond to the numbers that have been chosen.

To avoid the expansion of the numbers in binary to unary construction, we encode the representation of events on a process as follows: Since the conjunctive predicates can only be true when the local predicates are true, we keep only local states which satisfy their corresponding local predicate and store the number of local events executed so far with them. This leaves two local states at each process: The initial state with zero events executed until that point, and the final state of the process with the number of events equal to x_i for the i^{th} process. Now, the construction of the computation from the subset sum problem is polynomial in the size of the input.

■

20.5 Recognizing Meet-Closed and Sublattice Predicates

As discussed earlier, efficient detection algorithms exist for various classes of predicates. Thus, given a boolean expression B, one would like to determine if it belongs to a tractable subclass, in which case detection of the predicate may be performed efficiently.

Before stating the recognition problems, we recall the relationship between the structural and algorithmic properties used in earlier chapters.

- B is *meet-closed* on L iff the satisfying set $\{G \in L : B(G)\}$ is closed under componentwise meet. Equivalently, whenever $B(G)$ and $B(H)$ both hold, $B(G \sqcap H)$ also holds. This is a structural property of the satisfying set.

- B is *lattice-linear* on L (Chapter 2) iff B is meet-closed *and* there is an efficient procedure that, given any G with $\neg B(G)$, returns a forbidden index together with its advance value. Lattice-linearity is therefore strictly stronger than meet-closure: every lattice-linear predicate is meet-closed, but a meet-closed predicate need not come with an efficient advancement oracle.

- B is a *sublattice* predicate iff its satisfying set is a sublattice of L — closed under both meet and join. This is the structural strengthening of meet-closed.

- B is *regular* iff it is lattice-linear and dual lattice-linear. Regular is therefore strictly stronger than the sublattice property: every regular predicate is a sublattice predicate, but a sublattice predicate need not admit efficient advancement in either direction.

The recognition problems studied in this section concern the *structural* properties — meet-closure and the sublattice property — which are easier to check than their algorithmic strengthenings. We show that even these structural recognition problems are co-NP-complete. Since lattice-linear $\subsetneq$ meet-closed and regular $\subsetneq$ sublattice, the recognition problems for the algorithmic classes are at least as hard.

We define the decision problems as follows.

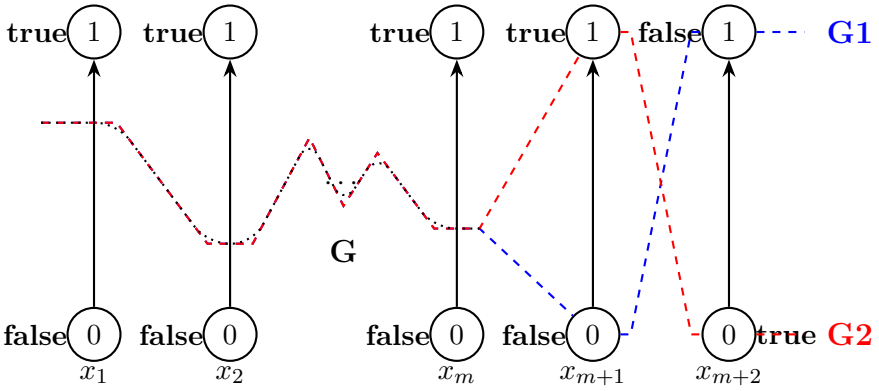


Figure 20.3: Transformation for Theorems 20.3 and 20.4.

Meet-Closure: Given a boolean expression b and a program computation, is the satisfying set of b closed under componentwise meet?

Sublattice: Given a boolean expression b and a program computation, is the satisfying set of b a sublattice (closed under both componentwise meet and componentwise join)?

Theorem 20.3 Meet-Closure *is co-NP-complete. Consequently, recognizing lattice-linearity is also co-NP-hard.*

Proof: *Meet-Closure is in co-NP*: Given a pair of candidate global states G and H in which the predicate is true, it can be easily verified in polynomial time that the predicate is false in the global state $G \cap H$. Such a triple is a certificate that the satisfying set is not closed under meet.

Meet-Closure is co-NP-hard: To show co-NP-hardness, we transform an arbitrary instance of TAUTOLOGY to an instance of Meet-Closure.

Let b be a boolean expression involving variables $x_1, x_2, \ldots, x_m$. For $i = 1, \ldots, m$ we place each x_i on a separate process P_i. Each of these m processes has two local states, a *true* state and a *false* state, which correspond to the value taken by the variable x_i in that local state. We also define two new variables x_{m+1} and x_{m+2} and place them on processes P_{m+1} and P_{m+2} respectively. Process P_{m+1} has two local states: an initial *false* state and a final *true* state, and process P_{m+2} has two local states: an initial *true* state and a final *false* state. Figure 20.3 shows this transformation. It is evident that this transformation can be achieved in polynomial time.

We define

$$B = b \vee x_{m+1}x_{m+2} \vee \overline{x}_{m+1}\overline{x}_{m+2}$$

We claim that B is meet-closed iff b is a tautology. If b is a tautology, then B is trivially meet-closed, since B will be true for all global states. Conversely, if b is not a tautology, then there exists a subcut involving processes $P_1, \ldots, P_m$ in which b evaluates to false. Let us call this subcut G. We can now extend the subcut G to form two cuts G_1 and G_2 in which the predicate B is true, as shown in Figure 20.3.

$$G_1 = (G, 0, 1) \text{ and } G_2 = (G, 1, 0)$$

However, $G_1 \cap G_2 = (G, 0, 0)$ in which the predicate B is false. Thus, B is not meet-closed.

Finally, since every lattice-linear predicate is meet-closed (lattice-linearity adds an efficient advancement requirement on top of meet-closure), recognizing lattice-linearity is also co-NP-hard: an oracle for lattice-linearity recognition would in particular decide Meet-Closure on the family of predicates above (each of which is meet-closed iff lattice-linear in this construction, since the trivially-true predicate is lattice-linear and the constructed counterexample is not even meet-closed).

■

Theorem 20.4 Sublattice *is co-NP-complete. Consequently, recognizing regularity is also co-NP-hard.*

Proof: *Sublattice is in co-NP*: Given two candidate global states in which the predicate is true, and their union and intersection such that the predicate is false in either the union or intersection, it can be easily verified in polynomial time that the predicate is not a sublattice predicate. Thus, Sublattice is in co-NP.

Sublattice is co-NP-hard: The transformation in Theorem 20.3 holds for Sublattice as well. That is, b is a tautology iff $B = b \vee x_{m+1}x_{m+2} \vee \overline{x}_{m+1}\overline{x}_{m+2}$ is a sublattice predicate. If b is a tautology, then B is trivially a sublattice predicate since B is true for all global states. Conversely, if b is not a tautology, then B is not a sublattice predicate since both the intersection and union of G_1 and G_2 result in a global state in which B is false. Thus, Sublattice is co-NP-complete. Since every regular predicate is a sublattice predicate, recognizing regularity is also co-NP-hard.

■

Note that the above transformation can also be used to show that the problem of deciding whether the satisfying set of a given boolean predicate is closed under componentwise join (the dual of meet-closure) is co-NP-complete. Consequently, recognizing dual lattice-linearity is also co-NP-hard.

20.6 Detection without Efficient Advancement

The LLP algorithm of Chapter 2 delivers a strongly polynomial detection procedure under *two* hypotheses on the predicate B:

- B is meet-closed — its satisfying set is closed under componentwise meet (a purely structural condition);

- B admits an *efficient advancement* oracle — given a G with $\neg B(G)$, a forbidden index j and the next value $\alpha(G, j, B)$ can be computed in polynomial time.

The two hypotheses together are what we have been calling *lattice-linearity*: a meet-closed predicate equipped with an efficient advancement oracle. The first hypothesis is structural; the second is computational. A natural question is whether the second hypothesis is really needed: could there be a single algorithm that detects every meet-closed predicate in polynomial time, without an efficient-advancement oracle being supplied externally? We show in this section that the answer is no: meet-closure alone does not buy tractability, and consequently lattice-linearity is a strict strengthening of meet-closure. The argument uses a celebrated result of Valiant and Vazirani [VV85].

Valiant–Vazirani: SAT reduces to USAT

A boolean expression B is a *USAT* instance if it has at most one satisfying assignment. Valiant and Vazirani showed that USAT is just as hard as general SAT under randomized reductions:

Theorem 20.5 (Valiant–Vazirani) *There is a randomized polynomial-time reduction from SAT to USAT. Consequently, if USAT admits a randomized polynomial-time decision procedure then* NP = RP.

The reduction takes a SAT instance and, by hashing the variable assignments through a random parity-check matrix, produces a USAT instance that is satisfiable with probability $\Omega(1)$ whenever the original SAT instance was. We refer the reader to [VV85] for the proof.

Every USAT instance is meet-closed (and a sublattice predicate)

Lemma 20.6 *Let B be a boolean expression with at most one satisfying assignment. Then B, viewed as a predicate on the lattice of consistent global states of any distributed computation, is meet-closed. Moreover the satisfying set of B is a sublattice.*

Proof: Meet-closure asks that the satisfying set be closed under componentwise meet. When B has at most one satisfying global state, any two satisfying states must coincide, so their meet is again that state and the satisfying set is trivially closed. The same argument applies to componentwise join, so the satisfying set is a sublattice as well.

■

Note that the lemma asserts only the *structural* properties. It does not assert lattice-linearity in the algorithmic sense: even though a USAT instance is meet-closed, computing the unique satisfying assignment (equivalently, the forbidden index together with its advance value at any given G) is not known to be easy.

Detection is hard for meet-closed predicates without efficient advancement

Given a USAT instance B on variables $x_1, \ldots, x_m$, build the two-state-per-process computation of Figure 20.1 (the same construction used in the NP-completeness proof of general predicate detection). Each consistent global state of this computation corresponds to exactly one assignment of the x_i, and detecting *possibly*: B is equivalent to deciding USAT on B. Combining this construction with Lemma 20.6 and Theorem 20.5 gives the main result.

Theorem 20.7 (Meet-closure alone is not enough) *There exists a family of meet-closed predicates (in fact, predicates whose satisfying set is a sublattice) whose detection problem is NP-hard under randomized reductions. In particular, no algorithm can detect every meet-closed predicate in polynomial time unless* NP = RP.

Proof: Apply the construction above to a SAT instance Φ. Valiant–Vazirani produces, in randomized polynomial time, a USAT instance B such that B is satisfiable with probability $\Omega(1)$ iff Φ is satisfiable. By Lemma 20.6, the satisfying set of B is meet-closed (and a sublattice). A polynomial-time detection algorithm for meet-closed predicates would then yield a randomized polynomial-time algorithm for SAT, forcing NP = RP.

■

Corollary. The same argument applies verbatim to sublattice predicates, since a USAT instance has a sublattice satisfying set by Lemma 20.6. Hence detection of sublattice predicates is also NP-hard under randomized reductions.

Reading of the result

Theorem 20.7 sharpens the LLP framework's tractability claim. The strongly polynomial bounds proved in Chapter 2 and applied throughout the book do not follow from meet-closure *per se*. They require, in addition, an explicit polynomial-time advancement oracle for the predicate at hand — exactly the algorithmic strengthening that distinguishes lattice-linearity from plain meet-closure (and regularity from the sublattice property). When we say "problem X is in P because its feasibility predicate is lattice-linear," the unstated companion claim is that the forbidden index and its advance value are computable in polynomial time — which is something one demonstrates separately for each problem (shortest path, stable marriage, MST, assignment, etc.). Without this companion claim the predicate's detection can be as hard as SAT.

In short: the strength of the LLP framework lies not in meet-closure of the satisfying set but in the combination of meet-closure *and* an efficient advancement procedure — which together constitute lattice-linearity. An analogous statement holds for regularity, which is the sublattice property strengthened by efficient advancement in both directions.

20.7 Summary

This chapter studied the complexity of predicate detection and predicate recognition in distributed computations. We showed that general predicate detection is NP-complete and pre-

sented efficient algorithms for the important subclass of conjunctive predicates. We also established co-NP-completeness results for recognizing tractable predicate classes.

Problem/Topic	Algorithm/Result	Time Complexity
General predicate detection	NP-complete	—
Conjunctive predicate detection	LLP-based (sequential)	$O(mn)$
Conjunctive predicate at level k	NP-complete	—
Recognizing Meet-Closed	co-NP-complete	—
Recognizing Sublattice	co-NP-complete	—

20.8 Problems

1. **(Conjunctive Predicate Backward Search)** In this chapter, we discuss finding the least consistent cut that satisfies a conjunctive predicate. (a) Show that conjunctive predicates are also closed under joins. (b) Give the predicate $rForbidden(G)$ to detect the largest consistent cut that satisfies a conjunctive predicate.

2. **(Bidirectional Conjunctive Search)** Give an algorithm that finds a consistent cut satisfying a conjunctive predicate by combining the forward search with the backward search.

3. **(Incremental Predicate Detection)** A ParallelCut-style algorithm has produced the least consistent cut $G^\star$ satisfying a conjunctive predicate $B = l_1 \wedge \cdots \wedge l_n$ on a computation of n processes. A new event is now appended to process P_k. Show that $G^\star$ can be extended to the least cut satisfying B on the augmented computation in $O(n)$ time, without rebuilding the state-rejection graph from scratch.

4. **(Dynamic Predicate Detection)** Each process maintains a *local* predicate l_i which may be edited by the developer (the literal swapped, a new conjunct added) in between detection queries. The trace itself is unchanged. Describe how much of the ParallelCut pipeline must be recomputed after such an edit, and explain why the recomputation cost depends on which step the edited literal influences.

5. **(Constrained Predicate Detection)** The user adds an external constraint of the form "the satisfying cut G must include event $e^\star$." Show that this is captured by conjoining B with a lattice-linear predicate, and that the ParallelCut algorithm still applies with no asymptotic overhead.

6. **(Approximate Predicate Detection at a Given Level)** Recall that conjunctive detection at level k is NP-complete by reduction from subset sum. Give a pseudo-polynomial algorithm that decides whether a satisfying cut exists at level exactly k when the target k is given in unary.

20.9 Bibliographic Remarks

The detection of conjunctive predicates was discussed by Garg and Waldecker in [GW92]. Distributed on-line algorithms for detecting conjunctive predicates were presented in Garg and

Chase [GC95]. Observer-independent predicates were introduced by Charron-Bost, Delporte-Gallet, and Fauconnier [CBDGF95]. Hurfin, Mizuno, Raynal and Singhal [HMRS95] gave a distributed algorithm that does not use any additional messages for predicate detection. Distributed algorithms for offline evaluation of global predicates are also discussed in Venkatesan and Dathan [VD92]. Stoller and Schneider [SS95] have shown how Cooper and Marzullo's algorithm can be integrated with that of Garg and Waldecker's to detect a conjunction of global predicates.

The second algorithm for detecting conjunctive predicates is from [GG19]. The NP-completeness of detecting a conjunctive predicate at the level k is shown in [GS24].

The complexity of detecting a boolean predicate is taken from [CG98]. The complexity of checking whether a predicate is lattice-linear (or regular) is from [KG05].

Chapter 21

Appendix: Lattice Theory

This appendix covers the definitions of partial orders and lattices used throughout this book. For a thorough treatment, see [DP90, Grä03, Gar15].

21.1 Relations

A partial order is simply a relation with certain properties. A **relation** R over a set X is a subset of $X \times X$. For example, on $X = \{a, b, c\}$, one possible relation is

$$R = \{(a, c), (a, a), (b, c), (c, a)\}.$$

It is sometimes useful to visualize a relation as a directed graph on the vertex set X with an edge from x to y whenever $(x, y) \in R$.

A relation is **reflexive** if $(x, x) \in R$ for every $x \in X$, and **irreflexive** if $(x, x) \notin R$ for every $x \in X$. On the natural numbers $\mathbb{N}$, the relation *divides* is reflexive, while *less than* is irreflexive. Note that a relation need not be either reflexive or irreflexive.

A relation R is **symmetric** if $(x, y) \in R$ implies $(y, x) \in R$, and **antisymmetric** if $(x, y) \in R$ and $(y, x) \in R$ implies $x = y$. On $\mathbb{N}$, *less than or equal to* is antisymmetric. R is **transitive** if $(x, y) \in R$ and $(y, z) \in R$ imply $(x, z) \in R$; the relations *less than* and *equal to* on $\mathbb{N}$ are both transitive.

A relation that is reflexive, symmetric, and transitive is an **equivalence** relation. When R is an equivalence, we write $x \equiv_R y$ for $(x, y) \in R$, and the **equivalence class** $[x]_R = \{y \mid y \equiv_R x\}$. The equivalence classes form a **partition** of X. For example,

$$R = \{(x, y) \mid x \bmod 5 = y \bmod 5\} \tag{21.1}$$

is an equivalence on $\mathbb{N}$ that partitions it into five classes.

21.2 Partial Orders

A relation R is a **reflexive partial order** (or **non-strict partial order**) if it is reflexive, antisymmetric, and transitive. A relation R is an **irreflexive partial order** (or **strict partial**

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

order) if it is irreflexive and transitive. The *divides* relation on $\mathbb{N}$ is a reflexive partial order, and *less than* on $\mathbb{N}$ is an irreflexive partial order. When R is a reflexive partial order, we write $x \leq_R y$ to denote $(x, y) \in R$. A reflexive partially ordered set, **poset** for short, is denoted by $(X, \leq)$. When R is irreflexive, we write $x <_R y$. Given an irreflexive partial order, we can define $x \leq y$ as $x < y$ or $x = y$, and vice versa. A relation is a **total order** if it is a partial order and for all distinct $x, y \in X$, either $(x, y) \in R$ or $(y, x) \in R$. The natural order on the integers is a total order, but *divides* on $\mathbb{N}$ is only a partial order.

Finite posets are often depicted using **Hasse diagrams**. For any $x, y \in X$, y **covers** x if $x < y$ and there is no z with $x < z < y$. We write $x <_c y$ to denote that y covers x (y is an **upper cover** of x; x is a **lower cover** of y). A Hasse diagram draws an edge from x to y whenever $x <_c y$, with x below y. For example, consider the poset

$$X \stackrel{\text{def}}{=} \{p, q, r, s\}; \quad \leq \stackrel{\text{def}}{=} \{(p, q), (q, r), (p, r), (p, s)\}, \tag{21.2}$$

whose Hasse diagram is shown in Figure 21.1.

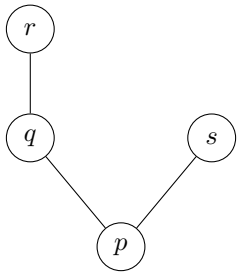


Figure 21.1: Hasse diagram of a poset

Elements x, y are **comparable** if $x < y$ or $y < x$; otherwise they are incomparable, written $x \| y$. A poset is a **chain** if every pair is comparable, and an **antichain** if every pair is incomparable. In Figure 21.1, $\{p, q, r\}$ is a chain and $\{q, s\}$ is an antichain. The **height** of a poset is the size of a longest chain, and the **width** is the size of a largest antichain. We use $\underline{n}$ for a chain of n elements and A_n for an antichain of n elements. A total order Q that extends a poset P is called a **linear extension** of P. For example, ordering the poset of Figure 21.1 as $p < q < r < s$ gives one linear extension.

An element x is a **bottom element** (denoted $\bot$) if $x \leq y$ for all y, and a **top element** (denoted $\top$) if $y \leq x$ for all y. An element x is **minimal** if there is no y with $y < x$, and **maximal** if there is no y with $y > x$.

21.3 Lattices

Let $(X, \leq)$ be a poset. We define two operators on subsets of X: **meet** (also called **infimum**, or **inf**) and **join** (also called **supremum**, or **sup**). For $Y \subseteq X$ and $m \in X$, we say that $m = \inf Y$ iff

1. $\forall y \in Y : m \leq y$, and
2. $\forall m' \in X : (\forall y \in Y : m' \leq y) \Rightarrow m' \leq m$.

Condition (1) says that m is a lower bound of Y, and condition (2) says that any other lower bound is at most m. For this reason, m is also called the **greatest lower bound** (glb) of Y. The infimum is unique whenever it exists, and m need not itself belong to Y. Dually, for $s \in X$, we say that $s = \sup Y$ iff

1. $\forall y \in Y : y \leq s$, and
2. $\forall s' \in X : (\forall y \in Y : y \leq s') \Rightarrow s \leq s'$.

s is also called the **least upper bound** (lub) of Y. We denote the glb of $\{a, b\}$ by $a \sqcap b$ and the lub of $\{a, b\}$ by $a \sqcup b$. In the natural numbers ordered by *divides*, the meet of two numbers is their gcd and the join is their lcm. The meet or join need not always exist: in Figure 21.1, $\{q, s\}$ has no upper bound; in poset (iii) of Figure 21.2 below, $\{b, c\}$ has upper bounds d and e but no least upper bound.

Definition 21.1 (Lattice) *A poset $(X, \leq)$ is a* **lattice** *iff $\forall x, y \in X$: $x \sqcup y$ and $x \sqcap y$ exist.*

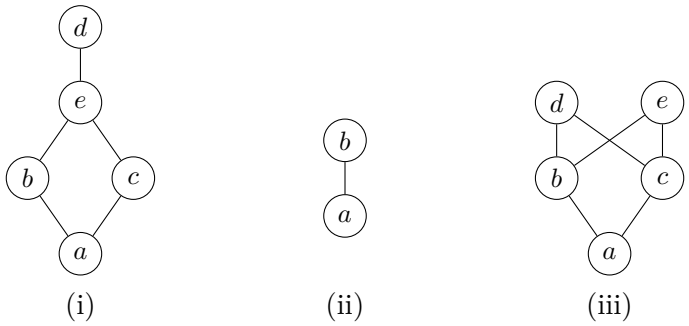


Figure 21.2: Only the first two posets are lattices. In (iii), $\{b, c\}$ has no least upper bound.

The first two posets in Figure 21.2 are lattices; the third is not, since $\{b, c\}$ has upper bounds d and e but no least upper bound. If only $\sqcup$ (resp. $\sqcap$) exists for all pairs, we have a *sup semilattice* (resp. *inf semilattice*). A lattice in which every subset has both a sup and inf is a *complete lattice*.

A nonempty $S \subseteq L$ is a **sublattice** of a lattice L iff $\forall a, b \in S$: $\sup(a, b) \in S$ and $\inf(a, b) \in S$, where sup and inf are computed in L.

A subset $Y \subseteq X$ of a poset is a **down-set** (or **order ideal**) if $z \in Y$ and $y \leq z$ implies $y \in Y$. The set $D[x] = \{y \mid y \leq x\}$ is a **principal order ideal**. Dually, Y is an **up-set** (or **order filter**) if $y \in Y$ and $y \leq z$ implies $z \in Y$; $U[x] = \{y \mid x \leq y\}$ is a **principal order filter**. In Figure 21.1, $D[r] = \{p, q, r\}$ and $U[p] = \{p, q, r, s\}$. We also write $D(x) = \{y \mid y < x\}$ and $U(x) = \{y \mid x < y\}$.

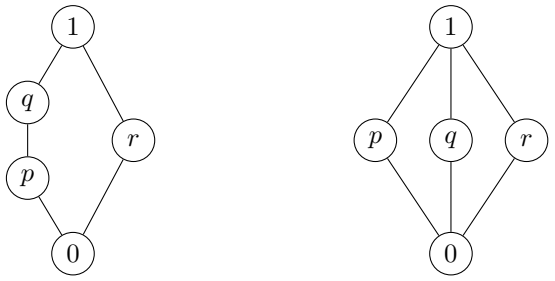
21.4 Distributive Lattices and Birkhoff's Theorem

An element x in a poset P is **join-irreducible** if $\forall Y \subseteq P$: $x = \sup Y$ implies $x \in Y$. The bottom element (if it exists) is not join-irreducible. The set of join-irreducible elements is

denoted $J(P)$; in a finite lattice, $x \in J(P)$ iff x has exactly one lower cover. In poset (i) of Figure 21.2, $J = \{b, c, d\}$ (and $e \notin J$ since $e = b \sqcup c$). By duality, **meet-irreducible** elements are denoted $M(P)$. The join-irreducibles form a basis: for any $x \in P$, $x = \sup(D[x] \cap J(P))$ (see [DP90]).

Definition 21.2 (Distributive Lattice) *A lattice L is* **distributive** *if $\forall a, b, c \in L$: $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$.*

Any power-set lattice is distributive, as is the lattice of natural numbers under *divides*. A finite lattice is nondistributive iff it contains a sublattice isomorphic to the Pentagon N_5 or the Diamond M_3 [Bir67, DP90]:



(a) Pentagon N_5 (b) Diamond M_3

Figure 21.3: The two smallest nondistributive lattices.

Let $L_I(P)$ denote the set of order ideals of a poset P, ordered by inclusion. Birkhoff's Theorem establishes that every finite distributive lattice is isomorphic to the lattice of order ideals of its join-irreducible elements.

Theorem 21.3 *[Birkhoff's Theorem [Bir67]] Let L be a finite distributive lattice. Then the mapping*

$$f : L \to L_I(J(L)), \quad a \mapsto f(a) = \{x \in J(L) : x \leq a\}$$

is an isomorphism from L onto $L_I(J(L))$.

The poset $J(L)$ can be exponentially more succinct than L itself. An example is shown in Figure 21.4: the five-element lattice L has three join-irreducibles a, b, c (with $a < b$ and $a < c$ in $J(L)$), and $L_I(J(L)) \cong L$.

Birkhoff's Theorem also translates properties between L and $P = J(L)$: the height of L equals $|P| + 1$, L is Boolean iff P is an antichain, and L is a chain iff P is a chain. See [DP90, Grä03] for proofs.

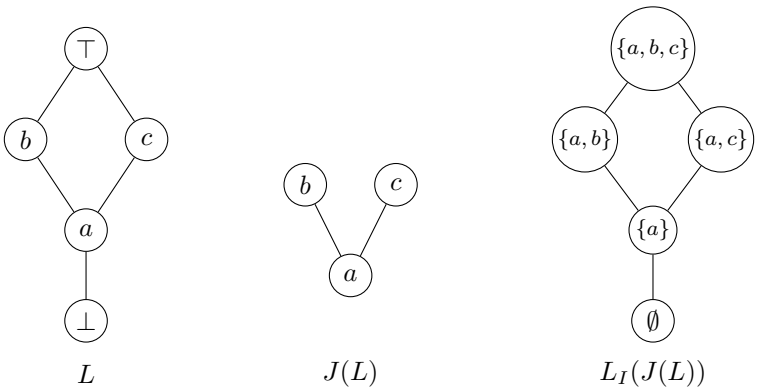


Figure 21.4: A lattice L, its join-irreducibles $J(L)$, and the lattice of ideals $L_I(J(L))$.

21.5 Problems

21.1. Give an example of a nonempty binary relation which is symmetric and transitive but not reflexive.

21.2. Show that if P and Q are posets defined on set X, then so is $P \cap Q$.

21.3. Draw the Hasse diagram of all natural numbers less than 10 ordered by the relation *divides*.

21.4. Show that if C_1 and C_2 are down-sets for any poset $(E, <)$, then so is $C_1 \cap C_2$.

21.5. Show that the join and meet operators are associative, commutative, and satisfy the absorption laws: $a \sqcup (a \sqcap b) = a$ and $a \sqcap (a \sqcup b) = a$.

21.6. Show that in a finite lattice, an element is join-irreducible iff it has only one lower cover.

21.7. Show that for a finite distributive lattice, L is Boolean iff $J(L)$ is an antichain.

21.6 Bibliographic Remarks

The first book on lattice theory was written by Garrett Birkhoff [Bir40, Bir48, Bir67]. The reader will find a discussion of origins of lattice theory and an extensive bibliography in the book by Grätzer [Grä71, Grä03]. The book by Davey and Priestley [DP90] provides an easily accessible account of the field. A treatment with an emphasis on computer science applications is given by Garg [Gar15]. Other books include those by Caspard, Leclerc, and Monjardet [CLM12], and Roman [Rom08].

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[Ada00] Hiroyuki Adachi. On a characterization of stable matchings. *Economics Letters*, 68(1):43–49, 2000.

[AG22] David R. Alves and Vijay K. Garg. Parallel minimum spanning tree algorithms via lattice linear predicate detection. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS Workshops 2022, Lyon, France, May 30 - June 3, 2022*, pages 774–782. IEEE, 2022.

[AHU74a] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AHU74b] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AKG20] David R. Alves, Madan S. Krishnakumar, and Vijay K. Garg. Efficient parallel shortest path algorithms. In *19th International Symposium on Parallel and Distributed Computing, ISPDC 2020, Warsaw, Poland, July 5-8, 2020*, pages 188–195. IEEE, 2020.

[ALM+98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, 1998.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[AMOT90] Ravindra K Ahuja, Kurt Mehlhorn, James Orlin, and Robert E Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM (JACM)*, 37(2):213–223, 1990.

[Apo76] Tom M. Apostol. *Introduction to Analytic Number Theory*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, 1976.

[APT79] Bengt Aspvall, Michael F. Plass, and Robert E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979. Introduces a linear-time algorithm for 2SAT.

[AS98] Atila Abdulkadiroğlu and Tayfun Sönmez. Random serial dictatorship and the core from random endowments in house allocation problems. *Econometrica*, 66(3):689–701, 1998.

[AS99] Atila Abdulkadiroğlu and Tayfun Sönmez. House allocation with existing tenants. *Journal of Economic Theory*, 88(2):233–260, 1999.

[BC06] David A. Bader and Guojing Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *Journal of Parallel and Distributed Computing*, 66(11):1366–1378, 2006.

[BCH90] Endre Boros, Yves Crama, and Peter L Hammer. Polynomial-time inference of all valid implications for horn and related formulae. *Annals of Mathematics and Artificial Intelligence*, 1(1-4):21–32, 1990.

[BDM12] Rainer Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems*. SIAM, revised reprint edition, 2012.

[Bel52] Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716, 1952.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[Ber92] Dimitri P. Bertsekas. Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications*, 1(1):7–66, 1992.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[Bir40] G. Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, R.I., 1940. first edition.

[Bir46] Garrett Birkhoff. Three observations on linear algebra. *Universidad Nacional de Tucumán Revista A*, 5:147–151, 1946.

[Bir48] G. Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, R.I., 1948. second edition.

[Bir67] G. Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, R.I., 1967. third edition.

[Bla77] Robert G. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2(2):103–107, 1977.

[Bor26] Otakar Borůvka. O jistém problému minimálním. *Práce Moravské Přírodovědecké Společnosti*, 3(3):37–58, 1926. In Czech. English translation available in: *Sources in the History of Mathematics and Physical Sciences*, vol. 1, 1985.

[BS96] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory, Volume 1: Efficient Algorithms*. MIT Press, Cambridge, MA, 1996.

[BYE85] Reuven Bar-Yehuda and Shimon Even. A local-ratio theorem for approximating the weighted vertex cover problem. *Annals of Discrete Mathematics*, 25:27–46, 1985.

[CBDGF95] B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and temporal predicates in distributed systems. *ACM Trans. on Programming Languages and Systems*, 17(1):157–179, January 1995.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[CG98] Craig M Chase and Vijay K Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[Cha00] Bernard Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[Chv79] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CLM12] Nathalie Caspard, Bruno Leclerc, and Bernard Monjardet. *Finite ordered sets: concepts, results and uses*, volume 144. Cambridge University Press, 2012.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 2001. second edition.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition, 2009.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC)*, pages 151–158, 1971.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[Dan63] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963. Introduces the simplex method and foundational linear programming theory.

[Dav13] Manlove David. *Algorithmics of matching under preferences*, volume 2. World Scientific, 2013.

[DdFdFS03] Vânia M.F. Dias, Guilherme D. da Fonseca, Celina M.H. de Figueiredo, and Jayme L. Szwarcfiter. The stable marriage problem with restricted pairs. *Theoretical Computer Science*, 306(1):391 – 405, 2003.

[DG84] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984. Presents a linear-time algorithm for Horn formula satisfiability.

[DGS86] Gabrielle Demange, David Gale, and Marilda Sotomayor. Multi-item auctions. *Journal of Political Economy*, 94(4):863–872, 1986.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Dil50] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Ann. Math. 51*, pages 161–166, 1950.

[Din70] Yefim A. Dinitz. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11(5):1277–1280, 1970.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[DPV08] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008.

[DPW10] David B Shmoys David P Williamson. *The Design of Approximation Algorithms*. Cambridge University Press, 2010.

[Edm65] Jack Edmonds. Maximum matching and a polyhedron with $0, 1$-vertices. *Journal of Research of the National Bureau of Standards B*, 69:125–130, 1965.

[Ege31] E. Egervary. On combinatorial properties of matrices. *Mat. Lapok*, 38:16–28, 1931.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[Eri19a] Jeff Erickson. *Algorithms*. Self-published, 2019. Available at http://jeffe.cs.illinois.edu/teaching/algorithms/.

[Eri19b] Jeff Erickson. *Algorithms*. http://jeffe.cs.illinois.edu/teaching/algorithms/, 2019.

[For56] L. A. Ford. Network flow theory. Technical report, The Rand Corporation, 1956.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[Ful56] D.R. Fulkerson. Note on dilworth's decomposition theorem for partially ordered sets. *Proc. Americal Math. Society*, pages 701–702, 1956.

[Gar15] Vijay K Garg. *Lattice Theory with Computer Science Applications*. Wiley, New York, NY, 2015.

[Gar17] Vijay K. Garg. Brief announcement: Applying predicate detection to the stable marriage problem. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 52:1–52:3. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[Gar20a] Vijay K. Garg. A generalization of teo and sethuraman's median stable marriage theorem, 2020.

[Gar20b] Vijay K. Garg. Predicate detection to solve combinatorial optimization problems. In Christian Scheideler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 235–245. ACM, 2020.

[Gar21] Vijay K. Garg. A lattice linear predicate parallel algorithm for the housing market problem. In Colette Johnen, Elad Michael Schiller, and Stefan Schmid, editors, *Stabilization, Safety, and Security of Distributed Systems - 23rd International Symposium, SSS 2021, Virtual Event, November 17-20, 2021, Proceedings*, volume 13046 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2021.

[Gar22] Vijay K. Garg. A lattice linear predicate parallel algorithm for the dynamic programming problems. In *Proc. of the Int'l Conf. on Distributed Computing and Networking (ICDCN)*, Delhi, India, 2022. Springer-Verlag.

[Gar23] Vijay Kumar Garg. Keynote talk: Lattice linear predicate algorithms for the constrained stable marriage problem with ties. In *24th International Conference on Distributed Computing and Networking, ICDCN 2023, Kharagpur, India, January 4-7, 2023*, pages 2–11. ACM, 2023.

[Gau01] Carl Friedrich Gauss. Disquisitiones arithmeticae. *English Translation*, 1801. English translation, 1966. Introduces the Chinese Remainder Theorem in modern form.

[GC95] V. K. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. In *Proc. of the IEEE Intnatl. Conf. on Distributed Computing Systems*, pages 423–430, Vancouver, Canada, June 1995.

[GG19] Vijay K. Garg and Rohan Garg. Parallel algorithms for predicate detection. In R. C. Hansdah, Dilip Krishnaswamy, and Nitin H. Vaidya, editors, *Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019, Bangalore, India, January 04-07, 2019*, pages 51–60. ACM, 2019.

[GH20] Vijay Kumar Garg and Changyong Hu. Improved paths to stability for the stable marriage problem, 2020.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Languages and Systems*, 5(1):66–77, January 1983.

[GI89] Dan Gusfield and Robert W Irving. *The stable marriage problem: structure and algorithms*. MIT press, 1989.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[GK23] Arya Tanmay Gupta and Sandeep S Kulkarni. Lattice linearity of multiplication and modulo, 2023.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, MA, 2 edition, 1994.

[Grä71] George Grätzer. *Lattice theory*. W.H. Freeman and Company, San Francisco, 1971.

[Gra79] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 8(1):47–50, 1979.

[Grä03] George Grätzer. *General Lattice Theory*. Birkhäuser, Basel, 2003.

[GS62] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[GS24] Vijay K. Garg and Robert P. Streit. Parallel algorithms for equilevel predicates. In *Proceedings of the 25th International Conference on Distributed Computing and Networking, ICDCN 2024, Chennai, India, January 4-7, 2024*, pages 104–113. ACM, 2024.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[GW92] V. K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 253–264. Springer Verlag, December 1992. Lecture Notes in Computer Science 652.

[Hås01] Johan Håstad. Some optimal inapproximability results. *Journal of the ACM*, 48(4):798–859, 2001.

[HG21] Changyong Hu and Vijay K. Garg. Characterization of super-stable matchings. In Anna Lubiw and Mohammad R. Salavatipour, editors, *Algorithms and Data Structures - 17th International Symposium, WADS 2021, Virtual Event, August 9-11, 2021, Proceedings*, volume 12808 of *Lecture Notes in Computer Science*, pages 485–498. Springer, 2021.

[HK56] A. J. Hoffman and J. B. Kruskal. Integral boundary points of convex polyhedra. *Linear Inequalities and Related Systems (Annals of Mathematics Studies 38)*, pages 223–246, 1956.

[HK71] John E. Hopcroft and Richard M. Karp. A $n^{5/2}$ algorithm for maximum matchings in bipartite. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 122 –125, oct. 1971.

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[HMRS95] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient distributed detection of conjunction of local predicates. Technical Report 2731, IRISA, Rennes, France, November 1995.

[Hoa61] Charles Antony Richard Hoare. Algorithm 64: quicksort. *Communications of the ACM*, 4(7):321, 1961.

[Hoa62] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–15, 1962.

[HS74] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM (JACM)*, 21(2):277–292, 1974.

[HSR01] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran. *Fundamentals of Computer Algorithms (Computer software engineering series)*. Galgotia Publications, new edition edition, 2001.

[Huf52] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[HW08] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, Oxford, 6 edition, 2008. Revised by D. R. Heath-Brown and J. H. Silverman.

[HZ79] Aanund Hylland and Richard Zeckhauser. The efficient allocation of individuals to positions. *Journal of Political economy*, 87(2):293–314, 1979.

[IK75] Oscar H Ibarra and Chul E Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM (JACM)*, 22(4):463–468, 1975.

[IM08] Kazuo Iwama and Shuichi Miyazaki. A survey of the stable marriage problem and its variants. In *Informatics Education and Research for Knowledge-Circulating Society, 2008. ICKS 2008. International Conference on*, pages 131–136. IEEE, 2008.

[JF56] L. R. Ford Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[Joh74] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, 1974.

[Kan39] L. V. Kantorovich. Mathematical methods of organizing and planning production. *Management Science (English translation, 1960)*, 6(4):366–422, 1939.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Kar84] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984. Presents the interior-point method, a polynomial-time alternative to simplex.

[KG05] Sujatha Kashyap and Vijay K. Garg. Intractability results in predicate detection. *Inf. Process. Lett.*, 94(6):277–282, 2005.

[Kha79] L. G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

[KKT95] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[KM72] Victor Klee and George J. Minty. How good is the simplex algorithm? In *Inequalities III: Proceedings of the Third Symposium on Inequalities*, pages 159–175, 1972.

[Knu71] Donald E. Knuth. Optimum binary search trees. *Acta informatica*, 1(1):14–25, 1971.

[Knu97a] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 3rd edition, 1997.

[Knu97b] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1997. Provides a detailed treatment of Euclid's algorithm and its applications.

[Knu97c] Donald Ervin Knuth. *Stable marriage and its relation to other combinatorial problems: An introduction to the mathematical analysis of algorithms*, volume 10. American Mathematical Soc., 1997.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[KO62] Anatoly Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1962.

[Kon31] D. Konig. Graphen und matrizen. *Mat. es Fiz. Lapok*, 38:116–119, 1931.

[Kru56] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[KT06] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison Wesley, 2006.

[KV18] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 6th edition, 2018.

[Lev73] Leonid A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973. Translated from Russian: Problemy Peredachi Informatsii, 9(3), 115–116.

[Lew78] Harry R. Lewis. Renaming a set of clauses as a Horn set. *Journal of the ACM*, 25(1):134–135, 1978.

[LNS82] J-L. Lassez, V. L. Nguyen, and E. A. Sonenberg. Fixed point theorems and semantics: a folk tale. *Information Processing Letters*, 14(3):112–116, 1982.

[Lov75] László Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13(4):383–390, 1975.

[LP86] László Lovász and Michael D. Plummer. *Matching Theory*, volume 121 of *North-Holland Mathematics Studies*. Elsevier, 1986.

[LRK76] Eugene L. Lawler and A. H. G. Rinnooy Kan. A single-machine scheduling problem with deadlines and lateness penalties. *Management Science*, 22(11):1237–1242, 1976.

[Man13] David F. Manlove. *Algorithmics of Matching Under Preferences*, volume 2 of *Series on Theoretical Computer Science*. World Scientific, Singapore, 2013.

[Men27] Karl Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10:96–115, 1927.

[MM11] Cristopher Moore and Stephan Mertens. *The Nature of Computation*. Oxford University Press, 2011.

[Mun57] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, 5(1):32–38, 1957.

[NMN01] Jaroslav Nesetril, Eva Milkova, and Helena Nesetrilova. Otakar Boruvka on minimum spanning tree problem Translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1-3):3–36, 2001.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PH77] Franco P. Preparata and Se June Hong. Convex hulls of finite sets of points in two and three dimensions. *Communications of the ACM*, 20(2):87–93, 1977.

[PQ80] Jean-Claude Picard and Maurice Queyranne. On the structure of all minimum cuts in a network and applications. *Mathematical Programming Studies*, 13:8–16, 1980.

[PR02] Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1):16–34, 2002.

[Pri57] Robert C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[PS08] Parag A. Pathak and Tayfun Sönmez. Leveling the playing field: Sincere and sophisticated players in the Boston mechanism. *American Economic Review*, 98(4):1636–1652, 2008.

[Ram97] Rajeev Raman. Recent results on the single-source shortest paths problem. *SIGACT News*, 28(2):81–87, June 1997.

[Rom08] Steven Roman. *Lattices and ordered sets*. Springer, 2008.

[Ros11] Kenneth H. Rosen. *Elementary Number Theory and Its Applications*. Pearson, 6th edition, 2011. Offers an accessible explanation of the Chinese Remainder Theorem.

[Rot64] Gian-Carlo Rota. On the foundations of combinatorial theory I: Theory of Möbius functions. *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 2(4):340–368, 1964.

[Rot82] Alvin E Roth. Incentive compatibility in a market with indivisible goods. *Economics letters*, 9(2):127–132, 1982.

[RP77] Alvin E Roth and Andrew Postlewaite. Weak versus strong domination in a market with indivisible goods. *Journal of Mathematical Economics*, 4(2):131–137, 1977.

[RS92] Alvin E Roth and Marilda Sotomayor. Two-sided matching. *Handbook of game theory with economic applications*, 1:485–541, 1992.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[RSU04] Alvin E. Roth, Tayfun Sönmez, and M. Utku Ünver. Kidney exchange. *Quarterly Journal of Economics*, 119(2):457–488, 2004.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

[Sch03] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics*. Springer, 2003.

[SG25] Robert Streit and Vijay K. Garg. Constrained cuts, flows, and lattice-linearity. *CoRR*, abs/2512.18141, 2025. arXiv preprint.

[SH75] Michael Ian Shamos and Dan Hoey. Closest-point problems. *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 151–162, 1975.

[Sho09] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, Cambridge, 2 edition, 2009.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2013.

[Ski08] Steven S. Skiena. *The Algorithm Design Manual*. Springer, 2nd edition, 2008.

[SMDD19] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019.

[SS71] Lloyd S Shapley and Martin Shubik. The assignment game i: The core. *International Journal of game theory*, 1(1):111–130, 1971.

[SS74] Lloyd Shapley and Herbert Scarf. On cores and indivisibility. *Journal of mathematical economics*, 1(1):23–37, 1974.

[SS95] S. D. Stoller and F. B. Schneider. Faster possibility detection by combining two approaches. In *Proc. of the 9th Intnatl. Workshop on Distributed Algorithms*, pages 318–332, France, September 1995. Springer-Verlag.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011.

[Tar55] Alfred Tarski. A lattice-theoretic fixed point theorem and its applications. *Pacific J Math*, 5:285–309, 1955.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[Tho00] Mikkel Thorup. On ram priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000.

[TS98] Chung-Piaw Teo and Jay Sethuraman. The geometry of fractional stable matchings and its applications. *Mathematics of Operations Research*, 23(4):874–891, 1998.

[TV85] Robert E. Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[Van20] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer, 5th edition, 2020. Covers simplex, duality, and interior-point methods with modern applications.

[Vaz01] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin, Germany, 2001.

[Vaz03] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[VD92] S. Venkatesan and B. Dathan. Testing and debugging distributed programs using global predicates. In *30th Annual Allerton Conf. on Commun., Control and Computing*, pages 137–146, Allerton, Illinois, October 1992.

[vN47] John von Neumann. Discussion of a maximum problem. *Institute for Advanced Study, Princeton (manuscript)*, 1947.

[VV85] Leslie G. Valiant and Vijay V. Vazirani. NP is as easy as detecting unique solutions. In Robert Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 458–463. ACM, 1985.

[VV89] John H. Vande Vate. Linear programming brings marital bliss. *Operations Research Letters*, 8(3):147–153, 1989.

[WS11] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

[ZG19] Xiong Zheng and Vijay K. Garg. Parallel and distributed algorithms for the housing allocation problem. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland*, volume 153 of *LIPIcs*, pages 23:1–23:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[Zho90] Lin Zhou. On a conjecture by Gale about one-sided matching problems. *Journal of Economic Theory*, 52(1):123–135, 1990.

Algorithm Index

General Index

Algorithms through the lens of lattice-linear predicates

This book presents the classical algorithms of computer science from two complementary perspectives. The first is the traditional one—greedy choices, divide-and-conquer recursions, dynamic-programming tables, augmenting paths. The second recasts each problem as the search for an extremal element of a distributive lattice that satisfies a lattice-linear predicate. From this single, unifying viewpoint, the book derives algorithms for sorting, graph traversal, shortest paths, minimum spanning trees, dynamic programming, network flow, bipartite matching, stable matching, housing allocation, assignment, satisfiability, number theory, and more.

The lattice-search formulation makes correctness proofs shorter, exposes natural parallelism, and turns algorithmic design into a systematic activity: identify the lattice, write down the predicate, check lattice-linearity, and read off the advance rule. Every algorithm in the book ships as runnable code on a companion website—in Java, Python, C++, and JavaScript—together with interactive demos that step through each LLP loop on small concrete instances. A solution manual is available to instructors on request.

Vijay K. Garg

Cullen Trust for Higher Education Endowed Professor
Department of Electrical and Computer Engineering,
The University of Texas at Austin

Vijay K. Garg leads research on parallel and distributed computing, lattice theory, and algorithms at The University of Texas at Austin. He is the author of *Introduction to Lattice Theory with Computer Science Applications* (Wiley, 2015), *Concurrent and Distributed Computing in Java* (Wiley, 2004), *Elements of Distributed Computing* (Wiley, 2002), *Principles of Distributed Systems* (Springer, 1996), and *Modeling and Control of Logical Discrete Event Systems* (Springer, 1995), and has published extensively on predicate detection, distributed snapshots, and combinatorial algorithms.