Lecture Notes on Algorithms and their Complexity

Michael A. Bekos

Department of Mathematics
University of Ioannina

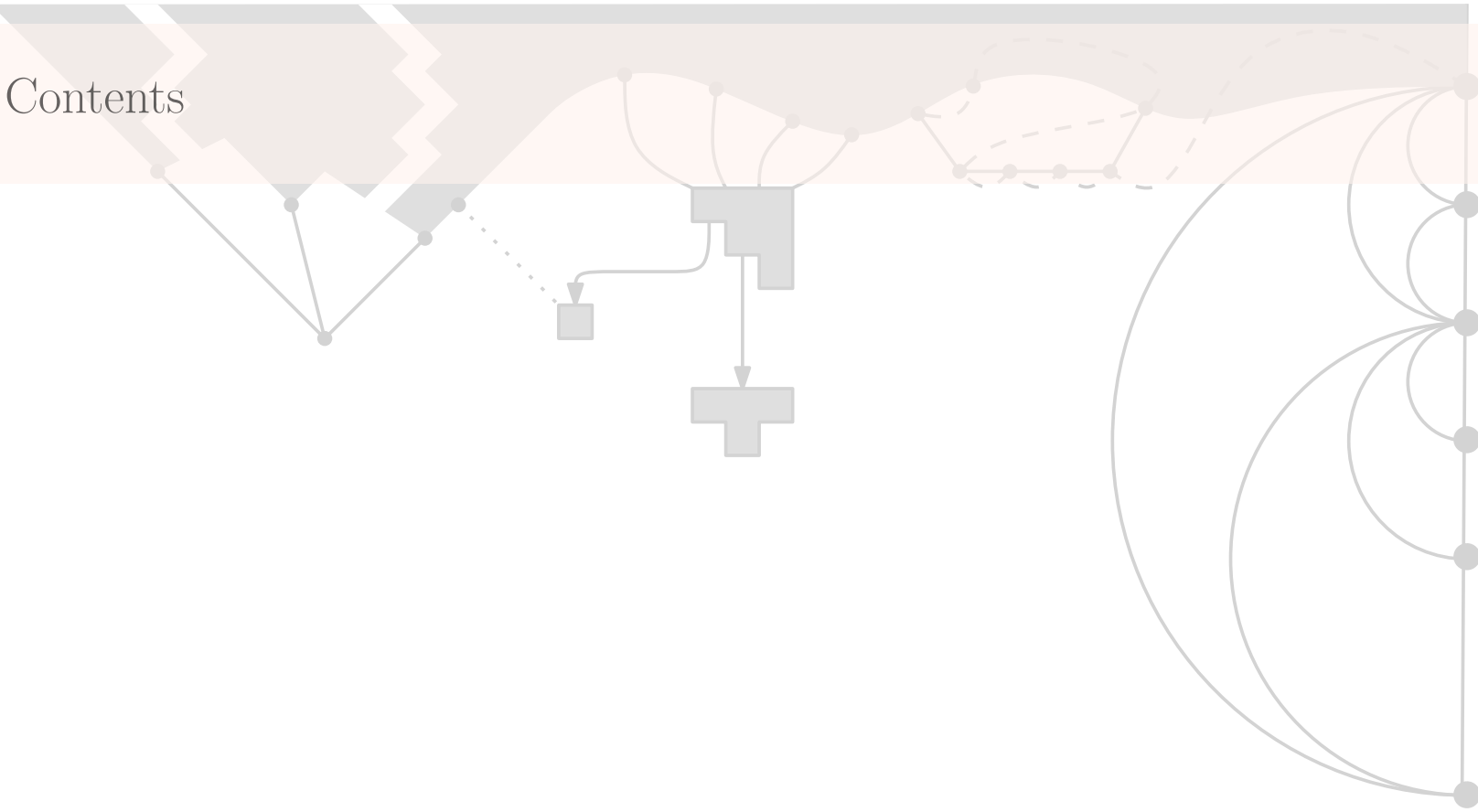2022

Contents

DRAFT

DRAFT

1. Introduction

This chapter will focus on an overview of the topics that are covered in the next chapters, as well as on an introduction of preliminary notions that are used in the following chapters. The rest of this chapter is organized as follows: Section 1.1 reviews standard graph theoretic definitions and notation that is used throughout these notes. In Section 1.2, we present a short description of the topics that are covered in these lecture notes. Section 1.4 recaps basic concepts for the analysis of the complexity of an algorithm and showcases them on simple examples. In Section 1.5, the standard definitions of big-O notation are given, and few examples of algorithm analyses are studied. Section 1.6 focuses on recalling a standard technique to develop algorithms, namely divide and conquer, through the well-known matrix multiplication problem. Finally, Section 1.7 recalls known lower bounds for standard problems.

1.1 Preliminaries

A graph $G = (V, E)$ consists of a finite set V of vertices and a set E of edges. Given a graph G, we denote by n and m its number of vertices and edges, that is, $n = |V|$ and $m = |E|$. Each edge $(u, v) \in E$ expresses a binary relationship between vertices u and v of V. We say that the edge (u, v) connects vertex u to vertex v and that the vertices u and v are the endvertices of the edge (u, v). Two vertices are adjacent or neighboring if they are connected by an edge, while two edges are adjacent if they share a common endvertex. Each edge in a graph can be either directed or undirected, i.e., an ordered or an unordered pair of vertices, respectively. Accordingly, a graph containing only undirected (directed) edges is referred to as undirected (directed or digraph). Unless otherwise specified, we assume that the graphs that we consider are simple, i.e., they contain neither self-loops (i.e., edges connecting vertices to themselves) nor parallel edges (i.e., edges with the same endvertices).

A graph is connected if there is a path between every pair of vertices of it. Otherwise, the graph is disconnected. A graph is called k-connected if it contains at least $k + 1$ vertices, but does not contain a set of $k - 1$ vertices whose removal disconnects the graph.
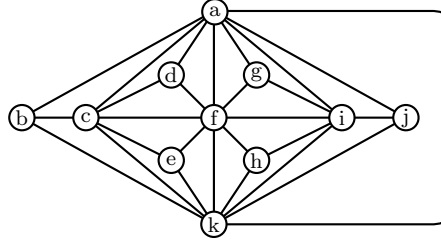
Figure 1.1: Illustration of a graph with 11 vertices and 27 edges.

Simplicity implies that the number of edges of a graph is bounded. In particular, a simple graph with n vertices that contains the maximum number of edges (overall graphs with n vertices) is called complete and is usually denoted by K_n.

> **Theorem 1.1** The complete graph K_n contains exactly $\frac{n(n-1)}{2}$ edges.

Proof. The number of edges of K_n equals to the number of handshakes possible at a party with n people, which is $\binom{n}{2}$. Equivalently, each of the n people can shake hands with $n-1$ people (they would not shake their own hand), and the handshake between two people is not counted twice; thus, $\frac{n(n-1)}{2}$ possible handshakes. □

A graph $G = (V, E)$ is bipartite if and only if its vertex-set V can be partitioned into two sets A and B, such that each edge of it connects a vertex from A to a vertex from B, namely, $E \subseteq A \times B$. If $E = A \times B$, then G is called complete bipartite. A complete bipartite graph whose parts have size n and m is denoted by $K_{n,m}$.

> **Theorem 1.2** The complete bipartite graph $K_{n,m}$ contains exactly $n \cdot m$ edges.

Proof. Each vertex belonging to the part, whose cardinality is n, is incident to exactly m edges. Since there exist n such vertices, it follows that $K_{n,m}$ has exactly $n \cdot m$ edges. □

A path of length k in a graph is formed by k vertices $v_0, v_1, \ldots, v_{k-1}$, such that for each $0 \le i < k-1$ vertices v_i and v_{i+1} are adjacent. A cycle of length k in a graph is formed by k vertices $v_0, v_1, \ldots, v_{k-1}$, such that for each $0 \le i \le k-1$ vertices v_i and $v_{i+1 \bmod k}$ are adjacent. A path or a cycle is simple if it consists of pairwise different vertices. A graph is acyclic if it contains no cycle. A tree is a connected acyclic graph.

> **Theorem 1.3** A tree with n vertices has exactly $n-1$ edges.

DRAFT

Proof. We give a simple proof by induction. The base of the induction corresponds to a tree with a single vertex. Since such a tree contains no edges, the statement holds in the base of the induction. Assume now that each tree with n vertices has exactly $n-1$ edges and consider a tree $\mathcal{T}$ with $n+1$ vertices. Tree $\mathcal{T}$ contains at least two vertices of degree 1. To see this, consider a longest path P in $\mathcal{T}$. Since $\mathcal{T}$ is finite, path P starts at some vertex v and ends at some vertex w of $\mathcal{T}$. Suppose, for a contradiction, that v is not of degree 1; then v has at least two neighbors, x and y, and one of them (say x) is not in P (as otherwise, $\mathcal{T}$ has a cycle). Let P' be the path obtained by concatenating P and x. This is a longer path than P, contradicting the choice of P. Thus, v is of degree 1. Similarly, w is also of degree 1. By removing v and its incident edge from T, we obtain a new tree $\mathcal{T}'$ with n vertices, for which we know that it contains exactly $n-1$ edge (by the inductive hypothesis). Since $\mathcal{T}$ and $\mathcal{T}'$ differ by one vertex and one edge, it follows that $\mathcal{T}$ has $n+1$ vertices and n edges, as desired.   □

1.2   Topics overview

The next chapters focus on different algorithms (and their analyses) for solving different optimization problems. An overview of the topics is given below followed by a brief description for each of them:

Minimum Spanning Trees (Chapter 3).   The input in this problem is a connected graph G, whose edges are weighted, where the weight of an edge is a positive number. The goal is to determine a spanning tree of G whose total weight (i.e., the sum of the weights of its edges) is minimum over all spanning trees of G. Two greedy algorithms are assumed to be known, which determine a minimum spanning tree of a graph:

Prim's algorithm, also known as Jarnik's algorithm, constructs a minimum spanning tree starting from an arbitrary vertex of the graph. At each step, the algorithm adds a new vertex to the tree, such that its connection to the formerly constructed tree is the cheapest possible.

Kruskal's algorithm constructs a minimum spanning tree by iterating over all edges of the graph in ascending order of their weights. An edge in some iteration is added to the minimum spanning tree if and only if it does not introduce a cycle with previously added edges.

In Chapter 3, a unification of both algorithms is presented and analyzed.

Minimum Cuts (Chapter 4).   The input in this problem is a connected graph G, whose edges are weighted (the weight of each edge is a positive number). The goal of this problem is to determine a partition of the vertices of G into two non-empty sets, such that the total weight of the edges with endvertrices to different sets in the partition is minimum. In Chapter 4, the algorithm by Stoer & Wagner is introduced to efficiently solve this problem.

Maximum Flows (Chapter 5).   The input in this problem is a connected, directed, and edge-weighted graph G, containing two distinct vertices s and t, which signify the source and the corresponding target of G. The weight of each edge, called capacity, is a positive

number corresponding to the maximum amount of flow that can pass through it. The goal is to determine a function which maps a flow to each edge of G such that the total flow out of the source-vertex s of G is maximized, without deviating the capacity constraints. After an investigation of the relationship between the minimum cut problem and the maximum flow problem, several algorithms are presented to solve the maximum flow problem, namely the algorithm by Ford and Fulkerson, the algorithm by Edmonds and Karp, Dinitz's algorithm as well as the Preflow-Push algorithm.

Maximum matchings (Chapter 6). Given a graph, the maximum matching problem asks for a subset of the edges of it, called matching, such that no two edges of the matching share a common endvertex. If the input graph is unweighted (weighted), the objective is to compute a matching of maximum cardinality (of maximum total weight, respectively). In Chapter 6 we focus on algorithms for solving different variants of this problem.

Algorithms for planar graphs (Chapter 7). A graph is planar if and only if it can be drawn on the plane so that no two edges of it cross (except possibly at common endpoints); for an example of a planar graph refer to Fig. 1.1. In Chapter 7, we prove several properties of planar graphs, and we also study algorithms which use these properties.

Linear Programming (Chapter 8). The term linear programming refers to the problem of optimizing (i.e., either maximizing or minimizing) a linear objective function with respect to a set of variables, subject to a number of constrains that are linear equalities or linear inequalities of the variables. As an example consider the following:

$$\text{maximize} \quad 3x_1 + 5x_2 \quad \text{subject to} \quad \begin{cases} x_1 + 3x_2 \geq 1 \\ x_1 - x_2 \geq 5 \\ x_1, x_2 \geq 0 \end{cases}$$

In Chapter 8, we discuss the simplex algorithm to solve linear programs.

Approximation Algorithms (Chapter 9). For an NP-hard problem, it is rather ambitious to try to derive a polynomial time algorithm (unless $P = NP$). A more reasonable goal is to approximate the problem's solution in polynomial time. This means that the computed output are most likely not optimal, but there is a guarantee to be close to the optimal. The primal goal here it to keep the computation need low so that the corresponding algorithm is practically useful. In Chapter 9 few such algorithms are discussed.

Randomized Algorithms (Chapter 10). In contrast to all formerly mentioned algorithms, which are deterministic (that is, given a particular input, each such algorithm always produces the same output), there also exist the so-called randomised algorithms, which are non-deterministic algorithms with several different applications. Randomised algorithms are usually employed to solve NP-complete problems. However, they are also useful for solving problems for which a deterministic (but rather inefficient) algorithm exists, since they are usually quite efficient. The main characteristic of randomised algorithms is that at certain steps of their executions "choices" are made at random, which means that their outputs are no longer solely dependent on their inputs. In Chapter 10 a few examples of those algorithms are discussed.

1.3   Useful Properties

In the next two theorems, we recall simple properties of sums and logarithms that will be useful in the following chapters.

Theorem 1.4   The following properties hold:

(a) $b^{\log_b x} = x$.

(b) $\log_b(x \cdot y) = \log_b x + \log_b y$.

(c) $\log_b(\frac{x}{y}) = \log_b x - \log_b y$.

(d) $\log_b x^k = k \cdot \log_b x$.

(e) $\log_b x = \frac{\log_a x}{\log_a b}$.

Proof. Let $m = \log_b x$ and $n = \log_b y$. By definition, $x = b^m$ and $y = b^n$. Hence:

(a) $b^{\log_b x} = b^m = x$.

(b) $\log_b(x \cdot y) = \log_b(b^m \cdot b^n) = \log_b(b^{n+m}) \stackrel{(a)}{=} n + m = \log_b x + \log_b y$.

(c) $\log_b(\frac{x}{y}) = \log_b(\frac{b^m}{b^n}) = \log_b(b^{n-m}) \stackrel{(a)}{=} n - m = \log_b x - \log_b y$.

(d) $\log_b x^k = \log_b (b^m)^k = \log_b b^{k \cdot m} \stackrel{(a)}{=} k \cdot m = k \cdot \log_b x$.

(e) $\log_a x = \log_a b^m \stackrel{(d)}{=} m \cdot \log_a b = \log_b x \cdot \log_a b$. $\square$

Theorem 1.5   The following properties hold:

(a) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

(b) For $x \neq 1$, $\sum_{k=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$.

(c) For $|x| < 1$, $\sum_{k=0}^{\infty} x^i = \frac{1}{1-x}$.

Proof. There are several proofs of these properties. In the following, we give simple ones.

(a) Let $S_n = 1+2+\ldots+(n-1)+n$. If we write S_n backwards, then $S_n = n+(n-1)+\ldots+2+1$. Adding the two equations, term by term, yields: $2S_n = (n+1)+(n+1)+\ldots+(n+1)+(n+1) = n(n+1)$, which implies that $S_n = \frac{n(n+1)}{2}$.

(b) Let $T_n = 1 + x + x^2 + \ldots + x^n$. Then $x \cdot T_n = x + x^2 + \ldots + x^n + x^{n+1}$. Thus, $x \cdot T_n - T_n = x^{n+1} - 1$, which implies that $T_n = \frac{x^{n+1}-1}{x-1}$.

(c) $\sum_{k=0}^{\infty} x^i = \lim_{n \to \infty} \left(\sum_{k=0}^{n} x^i \right) = \lim_{n \to \infty} \left(\frac{x^{n+1}-1}{x-1} \right) = \frac{-1}{1-x} = \frac{1}{x-1}$. $\square$

1.4   Algorithms' analysis

The field of algorithms' analysis describes different methods for analysis and comparison of various algorithms. Three main aspects can be highlighted within the field:

Correctness deals with formal proofs to determine that a given algorithm is guaranteed to produce the desired output to any properly formatted input. To accomplish this task, a multitude of techniques can be employed, e.g., a proof by induction, or a formalization of an invariant that holds true after every step of the algorithm.

Time complexity describes the running time of an algorithm as a function of the input size. For comparison purposes, sometimes the inputs are usually chosen to describe an average or worst case scenario. We provide more details on this aspect of algorithms in Section 1.5.1.

Space complexity describes the allocated memory space needed to run an algorithm as a function of the input size (refer to Section 1.5.2 for a more complete introduction). Note that in general, time and space complexities are not independent of each other and many algorithms can be modified to trade off between time and space constraints.

Our analysis focuses on algorithms having polynomial complexity. In general, this means that the computation of a correct solution is described by a polynomial function of the input size.

1.5   Big O notation: A recap

To analyse the (time or space) complexity of an algorithm, the following definitions of upper and lower bounds are central, as one needs to accurately assess different complexity classes.

Definition 1.1 We write $f(n) \in \mathcal{O}(g(n))$ to denote that function g is an asymptotic upper bound of function f, namely:

$$f(n) \in \mathcal{O}(g(n)) \Leftrightarrow \exists c > 0, \exists n_0 > 0, \forall n > n_0 : f(n) \leq cg(n)$$

Definition 1.2 We write $f(n) \in o(g(n))$ to denote that function g is an asymptotic strict upper bound of function f, namely:

$$f(n) \in o(g(n)) \Leftrightarrow \exists c > 0, \exists n_0 > 0, \forall n > n_0 : f(n) < cg(n)$$

Definition 1.3 We write $f(n) \in \Theta(g(n))$ to denote that function g is an asymptotic tight bound for function f, namely:

$$f(n) \in \Theta(g(n)) \Leftrightarrow \exists c_1, c_2 > 0, \exists n_0 > 0, \forall n > n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Definition 1.4 We write $f(n) \in \Omega(g(n))$ to denote that function g is an asymptotic lower bound of function f, namely:

$$f(n) \in \Omega(g(n)) \Leftrightarrow \exists c > 0, \exists n_0 > 0, \forall n > n_0 : f(n) \geq cg(n)$$

Definition 1.5 We write $f(n) \in \omega(g(n))$ to denote that function g is an asymptotic strict lower bound of function f, namely:

$$f(n) \in \omega(g(n)) \Leftrightarrow \exists c > 0, \exists n_0 > 0, \forall n > n_0 : f(n) > cg(n)$$

For simplicity reasons it is considered equally valid to use an equality expression instead of being an element part of a certain upper/lower bound class, as follows:

$$f(n) \in \mathcal{O}(g(n)) \quad \equiv \quad f(n) = \mathcal{O}(g(n))$$

Property 1.1 The following properties directly follow from Definitions 1.1–1.5:

- If $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$, then $f(n) = \Theta(g(n))$

- If $f_1(n) = \mathcal{O}(g_1(n))$ and $f_2(n) = \mathcal{O}(g_2(n))$, then $f_1(n) + f_2(n) = \mathcal{O}(g_1(n) + g_2(n))$

- If $f_1(n) = \mathcal{O}(g_1(n))$ and $f_2(n) = \mathcal{O}(g_2(n))$, then $f_1(n) \cdot f_2(n) = \mathcal{O}(g_1(n) \cdot g_2(n))$

- If $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(h(n))$, then $f(n) = \mathcal{O}(h(n))$

Example 1.1 The following relationships also follow from Definitions 1.1–1.5:

(a) $5n + 100 = \Theta(n)$.

(b) $10n^2 + 20 = \Theta(n^2)$.

(c) $7\log n + 2n^2 = \Theta(n^2)$.

(d) $7n = o(n\log n)$.

(e) $7\log n + 12 = o(n)$.

(f) $3n^2 = \omega(n)$.

Solution. We identify appropriate constants to show the desired relationships.

(a) $\forall n \geq 0 : n \leq 5n + 100 \leq 105n \Rightarrow 5n + 100 = \Theta(n)$

(b) $\forall n \geq 0 : n^2 \leq 10n^2 + 20 \leq 30n^2 \Rightarrow 10n^2 + 20 = \Theta(n^2)$

(c) $\forall n \geq 0 : 2n^2 \leq 7\log n + 2n^2 \leq 9n^2 \Rightarrow 7\log n + 2n^2 = \Theta(n^2)$.

(d) $\forall n \geq 0 : 7n < 7n\log n \Rightarrow 7n = o(n\log n)$.

(e) $\forall n \geq 0 : 7\log n + 12 < 19n \Rightarrow 7\log n + 12 = o(n)$.

(f) $\forall n \geq 0 : 3n^2 > n \Rightarrow 3n^2 = \omega(n)$. $\qquad\square$

Example 1.2   If $P(n)$ is a polynomial of degree k, then $P(n) = \Theta(n^k)$.

Solution. Since $P(n)$ is a polynomial of degree k, we may assume that $P(n)$ is written as:

$$P(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0, \text{ with } a_k > 0.$$

Since $P(n) \le \left(\sum_{i=0}^{k} |a_i| \right) n^k$, it follows that $P(n) = O(n^k)$. To show that $P(n) = \Omega(n^k)$, we rewrite $P(n)$ as follows:

$$P(n) = \tfrac{1}{2} a_k n^k + \tfrac{1}{2} a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$
$$= \tfrac{1}{2} a_k n^k + \left(\tfrac{a_k}{2k} n^k + a_{k-1} n^{k-1} \right) + \left(\tfrac{a_k}{2k} n^k + a_{k-2} n^{k-2} \right) + \cdots + \left(\tfrac{a_k}{2k} n^k + a_1 n \right) + \left(\tfrac{a_k}{2k} n^k + a_0 \right).$$

Since $a_k > 0$, for large enough values of n, we may assume that $\tfrac{a_k}{2k} n^k + a_i n^i \ge 0$, for each $i = 0, 1, \ldots, k-1$, which implies that $P(n) \ge \tfrac{1}{2} a_k n^k$.   □

Example 1.3   Assume that the time complexity of an algorithm is given by the following recursive formula:

$$T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + n \tag{1.1}$$

Give a closed formula for the algorithm's time complexity.

Solution. Before giving a closed formula for the algorithm's time complexity, we first discuss how one can interpret a recursive formula as the one of Eq. (1.1). Usually, such equations appear, when solving a problem using recursion (as we will shortly see in Section 1.6). A possible interpretation of Eq. (1.1) is that in order to solve a problem of size n, one needs to solve 3 subproblems of size $\lfloor \frac{n}{4} \rfloor$ and additionally perform a series of operations, whose total cost is n. In such case, $T(n)$ would describe the time complexity of the algorithm.

To solve Eq. (1.1), assume $n = 4^k$ to avoid working with the floor function (this assumption is without loss of generality since we seek for an asymptotic bound). Then, by expanding the recursive formula of Eq. (1.1), we obtain:

$$T(n) = 3 \left(3T \left(\frac{n}{4^2} \right) + \frac{n}{4} \right) + n$$

or equivalently:

$$T(n) = 3^2 T \left(\frac{n}{4^2} \right) + \frac{3n}{4} + n$$

By our assumption above, the recursion terminates in k steps and the result is:

$$T(n) = 3^k T \left(\frac{n}{4^k} \right) + n \sum_{i=0}^{k} \left(\frac{3}{4} \right)^i \overset{T(1)=1;n=4^k}{=} 3^k + n \sum_{i=0}^{k} \left(\frac{3}{4} \right)^i$$

The expression can then be simplified and estimated thusly:

$$T(n) = 3^{\log n} + n \sum_{i=0}^{k} \left(\frac{3}{4} \right)^i \le 4^{\log_4(n)} + n \sum_{i=0}^{\infty} \left(\frac{3}{4} \right)^i \overset{Thm.1.5}{=} n + n \cdot \frac{1}{1 - \frac{3}{4}} \in \mathcal{O}(n) \qquad □$$

Example 1.4 Assume that the time complexity of an algorithm is given by the following recursive formula:

$$T(n) = 2T(\sqrt{n}) + \log n \tag{1.2}$$

Give a closed formula for the algorithm's time complexity.

Solution. By setting $n = 2^m$ (or equivalently $m = \log n$), Eq. (1.2) can be rewritten as follows:

$$T(2^m) = 2T(2^{\frac{m}{2}}) + m \tag{1.3}$$

To further simplify the expression, we now substitute $T(2^m)$ by $S(m)$ in Eq. (1.3). Thus, the recursive formula expressing the time complexity of the algorithm becomes as follows:

$$S(m) = 2S(\frac{m}{2}) + m \tag{1.4}$$

Now observe that Eq. (1.4) describes the time complexity of the well-known merge sort algorithm, which has an upper bound of $\mathcal{O}(m \log m)$; see Section 1.7. Hence, the resulting complexity of our algorithm is:

$$T(n) = T(2^m) = S(m) = \mathcal{O}(m \log m) \overset{m=\log n}{=} \mathcal{O}(\log n \log \log n) \qquad \square$$

Definition 1.6 In mathematics, the n-th harmonic number is the sum of the reciprocals of the first n natural numbers:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots = \sum_{j=1}^{n} \frac{1}{j}$$

Lemma 1.1 For the n-th harmonic number H_n, it holds $\frac{1}{2} \log n - \frac{1}{2} < H_n < 2 + \log n$, that is, $H_n = \Theta(\log n)$.

Proof. Assume that n is a power of s, that it, $n = 2^k$ for some $k \in \mathbb{N}^+$. Then:

$$H_n = 1 +$$
$$\frac{1}{2} + \frac{1}{3} +$$
$$\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} +$$
$$\dots$$
$$\frac{1}{2^{k-1}} + \frac{1}{2^{k-1}+1} + \dots + \frac{1}{2^k - 1} +$$
$$\frac{1}{2^k}$$

Each group in the sum above can be bounded from below and above as follows:

$$
\begin{array}{ccccc}
\frac{1}{2} & \leq & 1 & \leq & 1 \\[4pt]
\frac{1}{4}+\frac{1}{4} & \leq & \frac{1}{2}+\frac{1}{3} & \leq & \frac{1}{2}+\frac{1}{2} \\[4pt]
\frac{1}{8}+\frac{1}{8}+\frac{1}{8}+\frac{1}{8} & \leq & \frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7} & \leq & \frac{1}{4}+\frac{1}{4}+\frac{1}{4}+\frac{1}{4} \\[4pt]
& & \cdots & & \\[4pt]
\frac{1}{2^k}+\frac{1}{2^k}+\cdots+\frac{1}{2^k} & \leq & \frac{1}{2^{k-1}}+\frac{1}{2^{k-1}+1}+\cdots+\frac{1}{2^k-1} & \leq & \frac{1}{2^{k-1}}+\frac{1}{2^{k-1}}+\cdots+\frac{1}{2^{k-1}} \\[4pt]
\frac{1}{2^k} & \leq & \frac{1}{2^k} & \leq & \frac{1}{2^k}
\end{array}
$$

Each row of the left-hand sums (except the last one) sums up to $\frac{1}{2}$, while each row of the right-hand sums (again except the last one) sums up to 1. Therefore:

$$
\frac{1}{2}k + \frac{1}{2^k} \leq H_n \leq k + \frac{1}{2^k}
$$

Since $n = 2^k$ and since $0 \leq \frac{1}{n} \leq 1$, the inequality above can be written in terms of n as follows:

$$
\frac{1}{2}\log n \leq H_n \leq 1 + \log n \tag{1.5}
$$

Thus, the statement holds, when n is a power of 2. Assume now that n is not a power of 2 and let n^- and n^+ be the surrounding powers of 2 of n. Then:

$$
\frac{n}{2} < n^- < n < n^+ < 2n
$$

By Eq. (1.5), the latter implies:

$$
\frac{1}{2}\log\frac{n}{2} < \frac{1}{2}\log n^- < H_{n^-} < H_n < H_{n^+} < 1 + \log n^+ < 1 + \log 2n
$$

Hence by Theorem 1.4:

$$
\frac{1}{2}\log n - \frac{1}{2} < H_n < 2 + \log n. \qquad \square
$$

1.5.1   Time Complexity Analysis

In this section, we recall few important fundamentals of time complexity analysis through a simple example. For our analysis, we will adopt the Random Access Machine (RAM) model of computation, which allows us to compare algorithms under the following assumptions:

- each simple operation takes unit time,
- loops and subroutines are not simple operations,
- each memory access takes unit time, and there is no shortage of memory.

Under these assumptions, in order to compute the time needed by an algorithm to solve a problem of size n (or equivalently, to compute its time complexity), one needs to identify the simple operations performed by the algorithm and express their total number as a function of n. To clarify this concept, we give two indicative examples.

Example 1.5 If an algorithm computes the result of the multiplication of two $n \times n$ matrices, then in order to estimate its time complexity, we need to estimate how many primitive operations (i.e., additions and multiplications of elements) are performed in total as a function of n.

Example 1.6 If an algorithm sorts n given elements (say, in ascending order), then in order to estimate its time complexity, we need to estimate how many comparisons between pairs of elements it performs in total (again as a function of n).

In the remainder of this section, we focus on the sorting problem. In particular, we will study the well-known bubble-sort algorithm (see Algorithm 1.1) and discuss its time complexity.

Algorithm 1.1: The bubble-sort algorithm

Input : An array A of n elements.
Output : The array sorted in ascending order.

```
1 Function BubbleSort(A[1,…,n])
2     for i = 1 to n do
3         for j = 1 to n do
4             if A[j] > A[j + 1] then
5                 swap (A[j], A[j + 1]);
6             end
7         end
8     end
9     return A;
10 end
```

Theorem 1.6 The time needed to sort n elements using bubble-sort is $\mathcal{O}(n^2)$.

Proof. We estimate the total number of comparisons. In Lines 2 and 3 of Algorithm 1.1, the algorithm consists of two nested for-loops iterating over the elements of the array. In the i-th iteration of the outer loop, exactly $n - i$ comparisons are executed. Hence, the total number of comparisons is:

$$\sum_{i=1}^{n}(n - i) = \sum_{i=0}^{n-1} i = \sum_{i=1}^{n-1} i \overset{\text{Thm. 1.5}}{=} \frac{n(n-1)}{2} \in \mathcal{O}(n^2) \qquad \square$$

The correctness of the bubble-sort algorithm can be proved by showing that the invariant described in the following theorem is maintained at each iteration of the algorithm.

Theorem 1.7 Let A be an array containing n elements. After i-th iteration of the bubble-sort algorithm the part $A[n - i, n]$ of the array A is sorted.

DRAFT

Proof. Assume that the invariant holds up to the $(i-1)$-th iteration of the bubble-sort algorithm. Consider now the i-th iteration. As soon as any element is compared against the largest element of $A[1, n-i]$, the latter is bound to be swapped until the end of the part $A[1, n-i]$ of the array A is reached. Thus after the i-th iteration the invariant holds.   □

1.5.2 Space Complexity Analysis

In the following, we focus on space complexity analysis, where the goal is to express the amount of required space to store the input and the data generated by an algorithm as a function of the size of the input of the problem. By definition, it immediately follows that the size of the input problem is always a lower bound for the space complexity of the algorithm.

In the following, we discuss the space requirements of two basic representations of graphs: the adjacency-lists representation and the matrix representation. We also discuss how they are related to the time complexity of corresponding simple graph algorithms. To this end, consider a graph $G = (V, E)$ with n vertices and m edges, and assume that graph G is undirected (even though the representations are defined similarly also for directed graphs).

The adjacency-lists representation of graph G stores the n vertices of graph G in a list. Each entry in this list, i.e., each vertex in graph G, is further linked to another list containing its neighboring vertices. This implies that for each edge (u, v) of graph G connecting two vertices u and v, vertex u is added in the list of neighbors of v, while vertex v is added in the list of neighbors of u. The adjacency-lists representation of the example graph of Fig. 1.1 is given in Fig. 1.2a.

The matrix representation of graph G uses a symmetric $n \times n$ matrix A to store G. If there exists an edge (v_i, v_j) connecting two vertices v_i and v_j in graph G, the respective entries $A[i, j]$ and $A[j, i]$ are set to 1; otherwise they are set to 0. The matrix representation of the graph illustrated in Fig. 1.1 is given in Fig. 1.2b.

$$
\begin{bmatrix}
0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0
\end{bmatrix}
$$

a →   b - c - d - f - g - i - j - k
b →   a - c - k
c →   a - b - d - e - f - k
d →   a - c - f
e →   c - f - k
f →   a - c - d - e - g - h - i - k
g →   a - f - i
h →   f - i - k
i →   a - g - h - j - k
j →   a - i - k
k →   a - b - c - e - f - h - i - j

(a) Adjacency-list representation.   (b) Matrix representation.

Figure 1.2: Two different representations of the graph seen in Fig. 1.1.

> **Theorem 1.8** The space requirement of the adjacency-lists representation of a graph with n vertices and m edges is $\mathcal{O}(n+m)$.

Proof. The adjacency-lists representation of a graph with n vertices and m edges consists of exactly n lists, and thus is directly dependant on the number of vertices. To estimate the total amount of entries in all lists, we observe that each edge (u,v) the graph contributes two entries in the representation; one in the list of u and one in the list of v. Thus the total amount of entries in all lists is upper bounded by $2m$. This implies that the space requirement is $n + 2m \in \mathcal{O}(n+m)$. In other words, the space requirement of the adjacency-lists representation is asymptotically equal to the size of the graph. $\square$

> **Theorem 1.9** The space requirement of the matrix representation of a graph with n vertices and m edges is $\mathcal{O}(n^2)$.

Proof. All information in a matrix representation of graph is maintained into a $n \times n$ matrix. Thus, its space requirement is exactly $n^2 \in \mathcal{O}(n^2)$. $\square$

> **Observation 1.1** The space requirement of the matrix representation of a graph is independent from its number of edges.

> **Observation 1.2** By an isolated comparison of the space requirements of the two representations, if a graph is sparsely connected (i.e., it contains $\mathcal{O}(n)$ edges), then a clear preference towards the adjacency-lists could emerge.

While the latter observation appears as a clear advantage of the adjacency-lists representation over the matrix representation, it is important to consider the interactions of an algorithm with the proposed representations and the resulting influence on the time complexity of it.

To exemplify the influence of a chosen representation on the time complexity of an algorithm, consider the following simple algorithm that determines whether two given vertices v_i and v_j in the graph are connected by an edge or not. An algorithm's behavior to perform this simple query would be dependent on the data that must be accessed (i.e., on the underlying representation of the graph). To answer this query, if the graph is represented as adjacency-lists, the list corresponding to vertex v_i must accessed, and iterated through until v_j is either found or the list is exhausted. Therefore, reporting whether v_i and v_j are adjacent is done in $\mathcal{O}(n)$ time (in worst case). On the other hand, if the graph is represented as a matrix, to report whether v_i and v_j are adjacent, it is enough to access the corresponding entry $A[i,j]$ of the matrix, which can be done in $\mathcal{O}(1)$ time.

In conclusion, the example above indicates a trade off between space and time complexity. Additionally different representations can be dependant on a variety of properties of the input data, which can inform the optimal method to solve a specific problem.

1.6   The matrix multiplication problem

This section recalls a quite common technique to develop algorithms, namely divide and conquer. The technique is recalled through a quite common algorithmic problem, the one of computing the product of two matrices. Recall that a matrix in mathematics is simply a 2-dimensional array of elements (most commonly numbers). Matrices are usually referred to by the amount of rows and columns they have, e.g., the matrix in Fig. 1.3 is a "two by three" matrix, because it consists of two rows and three columns. By convention, we use rows before columns to identify elements in an array, so an element a_{ij} is at the intersection of row i and column j.

$$\begin{bmatrix} 2 & 0 & 1 \\ 3 & -1 & 4 \end{bmatrix}$$

Figure 1.3: A 2×3 matrix

Matrices have numerous applications in computer science, e.g., for manipulating 3D models and projecting them onto a 2D screen or in the ranking algorithms of search engines. In almost every application where matrices are used, matrix multiplication is a central operation which is performed very frequently. Hence, finding an efficient algorithm to solve this problem is critical.

To be able to multiply two matrices, the number of columns in the first matrix has to be equal to the number of rows in the second matrix. For simplicity, in the following we will only consider square matrices, i.e., $n \times n$ matrices. Given two square $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$, the product matrix $C = A \cdot B$ consists of entries c_{ij} which are defined by the following equation.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}, \ i,j = 1,2,\ldots,n. \tag{1.6}$$

In other words, entry c_{ij} in the product matrix C is computed by multiplying the entries in the i-th row of A with the entries in the j-th column of B on a term-by-term basis and then summing up the individual products.

1.6.1   The Naive Algorithm

The most straightforward approach to compute the product of two matrices is to follow the definition very closely. First, iterate over all rows of the first matrix. For a given row i, iterate over all columns of the second matrix and for every combination of row i and column j, compute the corresponding entry c_{ij}. This entry c_{ij} is calculated as the sum of n products.

In other words, n^2 matrix entries have to be computed and for every one of those entries we have to sum up n terms. The pseudocode for such an algorithm is presented in Algorithm 1.2.

Algorithm 1.2: A naive algorithm to compute the product of two martices.

Input   : Two $n \times n$ matrices A and B.
Output : The product of A and B.

1 for $i = 1$ to n do
2 | for $j = 1$ to n do
3 | | $c_{ij} = 0$;
4 | | for $k = 1$ to n do
5 | | | $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
6 | | end
7 | end
8 end
9 return C;

Since the algorithm consists of three nested loops, each of which runs for exactly n iterations, the complexity of this algorithm is in $\Theta(n^3)$. Note that $\Omega(n^2)$ is a trivial lower bound for the matrix multiplication problem.

1.6.2   A simple divide-and-conquer algorithm

The naive algorithm for solving the matrix multiplication problem has cubic time complexity. In the following, we seek for an improved algorithm in terms of the time complexity. To this end, we will try to solve the matrix multiplication problem with a divide-and-conquer algorithm. Notice that for such an algorithm the matrix multiplication problem is ideal, since it can be easily partitioned into non-overlapping sub-problems, which are of identical nature, and their solutions can be easily combined. For simplicity, we assume that n is a power of 2 in our $n \times n$ matrices, so that we can partition them evenly into sub-matrices. The idea behind this algorithm is to partition the input matrices A and B as well as the product matrix C into four $n/2 \times n/2$ matrices each as follows.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}. \tag{1.7}$$

The product $C = A \cdot B$ is then equal to

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \tag{1.8}$$

where each of C_{11}, C_{12}, C_{21} and C_{22} is given by one of the following four equations:

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \quad (1.9) \qquad\qquad C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \quad (1.11)$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \quad (1.10) \qquad\qquad C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \quad (1.12)$$

In the base case, in which $n = 1$, only the scalar multiplication $c_{11} = a_{11} \cdot b_{11}$ has to be computed, which can be clearly done in constant time such that:

$$T(1) = \Theta(1). \tag{1.13}$$

In the recursive case, the algorithm is called on eight sub-instances each of size half of the size of the original instance. We also must perform four additions of matrices of size $n/2 \times n/2$, each of which can be done in $\Theta(n^2)$. Together, the time complexity in the recursive case is

$$T(n) = 8\,T\left(\frac{n}{2}\right) + cn^2. \tag{1.14}$$

Combining Equations (1.13) and (1.14) yields the following recurrence relation:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1. \\ 8\,T(\frac{n}{2}) + cn^2, & \text{if } n > 1. \end{cases} \tag{1.15}$$

We proceed to solve this recurrence. Since we have assumed that n is a power of 2, i.e., $n = 2^k$ for some k, it follows that $k = \log n$. Hence:

$$\begin{aligned}
T(n) &= 8\,T\left(\frac{n}{2}\right) + cn^2 \\
&= 8^2\,T\left(\frac{n}{2^2}\right) + c8\left(\frac{n}{2}\right)^2 + cn^2 \\
&= \ldots \\
&= 8^k\,T\left(\frac{n}{2^k}\right) + c8^{k-1}\left(\frac{n}{2^{k-1}}\right)^2 + \cdots + c8\left(\frac{n}{2}\right)^2 + cn^2 \\
&\overset{k=\log n}{=} 8^{\log n}\,T(1) + cn^2 \cdot \sum_{i=0}^{k-1} 8^i\left(\frac{1}{2^i}\right)^2 \\
&= (2^3)^{\log n} + cn^2 \cdot \sum_{i=0}^{k-1} \left(\frac{8}{2^2}\right)^i \\
&= (2^{\log n})^3 + cn^2 \cdot \sum_{i=0}^{k-1} 2^i \\
&= n^3 + cn^2(2^k - 1) \\
&= n^3 + cn^2(n - 1) \\
&= \Theta(n^3)
\end{aligned} \tag{1.16}$$

Equation (1.16) proves that the recurrence relation (1.15) has the solution $T(n) = \Theta(n^3)$. Therefore, the simple divide-and-conquer algorithm also has cubic time complexity. Thus, asymptotically is not more efficient than the naive algorithm.

1.6.3   Strassen's Algorithm

As we have seen, the simple divide-and-conquer algorithm is not enough to out-perform the naive algorithm for matrix multiplication. This stems from the fact that the time complexity of the divide-and-conquer algorithm is dominated by the number of multiplications that are performed recursively. In particular, observe that $\mathcal{O}(n^3)$ is actually $\mathcal{O}(n^{\log 8})$, where 8 is the number of multiplications performed at each recursive call. The key idea of Strassen's algorithm is to reduce the number of multiplications down to seven at the cost of having to perform more additions but still only a constant number of them.

More precisely, Strassen's algorithm also partitions matrices A, B and C into sub-matrices as in Equation (1.7). Afterwards, however, it computes more complicated combinations of the sub-matrices. First, the seven auxiliary matrix products in Equations (1.17) to (1.23) are computed.

$$P_1 = A_{11} \cdot (B_{12} - B_{22}) \qquad (1.17) \qquad P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \qquad (1.21)$$

$$P_2 = (A_{11} + A_{12}) \cdot B_{22} \qquad (1.18) \qquad P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \qquad (1.22)$$

$$P_3 = (A_{21} + A_{22}) \cdot B_{11} \qquad (1.19) \qquad P_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \qquad (1.23)$$

$$P_4 = A_{22} \cdot (B_{21} - B_{11}) \qquad (1.20)$$

The entries of C are then computed by adding and subtracting the aforementioned matrices as seen in Equations (1.24) to (1.27) .

$$C_{11} = P_5 + P_4 - P_2 + P_6 \qquad (1.24)$$

$$C_{12} = P_1 + P_2 \qquad (1.25)$$

$$C_{21} = P_3 + P_4 \qquad (1.26)$$

$$C_{22} = P_5 + P_1 - P_3 - P_7 \qquad (1.27)$$

The correctness of this method can be easily verified. For instance, replacing P_1 and P_2 with their expansions in Equation (1.25) gives the following:

$$C_{12} = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} + A_{11} \cdot B_{22} + A_{12} \cdot B_{22} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22},$$

which is equal to Equation (1.10). The other equalities can be verified similarly.

Equations (1.17) to (1.27) imply that Strassen's algorithm performs 7 multiplications and 18 additions in each recursive call. The base case, which corresponds to multiplying two matrices of size 1×1 each, consists of one scalar multiplication. Thus, the recurrence relation expressing the time complexity of Strassen's algorithm is as follows

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1. \\ 7\,T(\frac{n}{2}) + cn^2, & \text{if } n > 1. \end{cases} \qquad (1.28)$$

Solving this recurrence analogous to (1.16) gives us a running time of $T(n) = \mathcal{O}(n^{\log 7}) \approx \mathcal{O}(n^{2.807})$. Thus, Strassen's algorithm is asymptotically faster than the naive algorithm and the simple divide-and-conquer algorithm. However, the $\mathcal{O}$-notation hides a rather big constant,
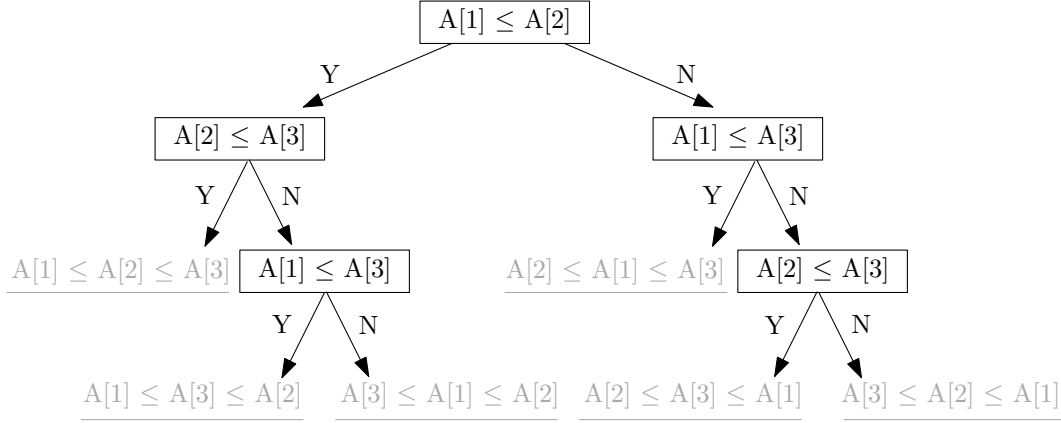
Figure 1.4: Illustration of a decision tree of a sorting algorithm with input size $n = 3$

which is the reason why for small matrices it is often preferred to use the naive algorithm. For large matrices, a common practice is to start with Strassen's algorithm and once the matrices become relatively small compute their product with the naive algorithm.

1.7   Lower bounds

In this section, we recall a standard approach to prove lower bounds on the time needed to solve a problem with certain size. To this end, we consider the problem of sorting an array with n elements using comparisons and we seek to prove that this problem cannot be solved in time $o(n \log n)$.

> **Theorem 1.10**   The problem of sorting an array A with n elements (with comparisons) is in $\Omega(n \log n)$.

Proof. Consider any comparison-based algorithm that sorts array A. We explicitly remark that we do not consider some particular algorithm here (e.g., merge-sort or bubble-sort), but any such algorithm that sorts array A using comparisons. The process of sorting array A by such an algorithm can be represented by a decision tree, as illustrated in Fig. 1.4 for the case $n = 3$. Every internal node in this decision tree corresponds to a comparison (of the form "is element $A[i]$ smaller or equal to element $A[j]$?") that the algorithm may perform. Depending on the output of a certain comparison (i.e., yes or no) the algorithm proceeds by either comparing another pair of elements or by reporting the sorted array. Therefore, the decision tree is a binary tree, whose leaves corresponds to the different outputs (i.e., different permutations of the n elements of A) of it. Hence, the decision tree has $n!$ leaves. As a result, in the worst case the algorithm may need to perform as many comparisons as the height of the tree. Thus, its complexity is upper bounded by the height of the tree, which is at least:

$$\lceil \log n! \rceil \geq \log(\frac{n}{2})^{\frac{n}{2}} = \frac{n}{2} \log(\frac{n}{2})$$

Algorithm 1.3: The merge-sort algorithm

   Input   : An array A of n elements.
   Output : The array sorted in ascending order.

1 Function MergeSort(A, ℓ, r)
2   if $\ell < r$ then
3       $m = \lfloor (\ell + r)/2 \rfloor$;
4       $A_1 = $ MergeSort(A, ℓ, m);
5       $A_2 = $ MergeSort($A, m + 1, r$);
6   return Merge(A_1, A_2);   // Method to merge two sorted arrays
7   end
8   return $A[\ell]$;
9 end

Hence, the problem of sorting n elements using comparisons is in $\Omega(n \, log n)$.   □

> **Theorem 1.11**   The problem of sorting an array with n elements (with comparisons) is in $\Theta(n \log n)$

Proof. In view of Theorem 1.10, it suffices to provide an algorithm that sorts an array with n elements using $\mathcal{O}(n \log n)$ comparisons. For this, consider the merge-sort algorithm. Algorithm 1.3 provides corresponding pseudocode; the implementation of the method merge, which given two sorted arrays A_1 and A_2 returns in $\mathcal{O}(|A_1| + |A_2|)$ time a new sorted array containing all elements of A_1 and A_2, is left as an exercise to the reader. It follows that if we denote by $T[n]$ the time needed by merge-sort to sort the elements of an array of size n, then:

$$T[n] = T[\lfloor \frac{n}{2} \rfloor] + T[\lceil \frac{n}{2} \rceil] + \mathcal{O}(n); \quad T[1] = 1$$

This is because the time needed to sort an array of size n is equal to the time needed to sort two arrays of size approximately $n/2$ plus the time needed to merge these two sorted arrays. To solve the aforementioned recursive formula, we assume that $n = 2^k$ and $c = \mathcal{O}(1)$. Hence, we obtain:

$$T(n) = 2T[\frac{n}{2}] + cn = 2^2(T(\frac{n}{2^2}) + c\frac{n}{2}) + cn = \ldots = 2^k(T(\frac{n}{2^k})) + kcn = nT(1) + kcn = n + cn \log n$$

Hence, $T[n] \in \mathcal{O}(n \log n)$. Note that the correctness of the merge-sort algorithm follows by induction and by the correctness of the method merge (which is again left as an exercise to the reader).   □

In the following, we prove a lower bound for another problem by a reduction to the problem of sorting n elements with comparisons, which by Theorem 1.10 belongs to $\Omega(n \log n)$. Recall that problem reduction is a method to transform an input (i.e., a specific instance) of a problem to an input (i.e., another specific instance) of a different problem (usually with a known solution). Important is that this transformation can be done in considerably less time

(a) Illustration of the convex hull on a set of nine points on a Euclidean plane

(b) Illustration of the convex hull of the following set of points $\{(1,1),(2,4),(3,9),(4,16)\}$
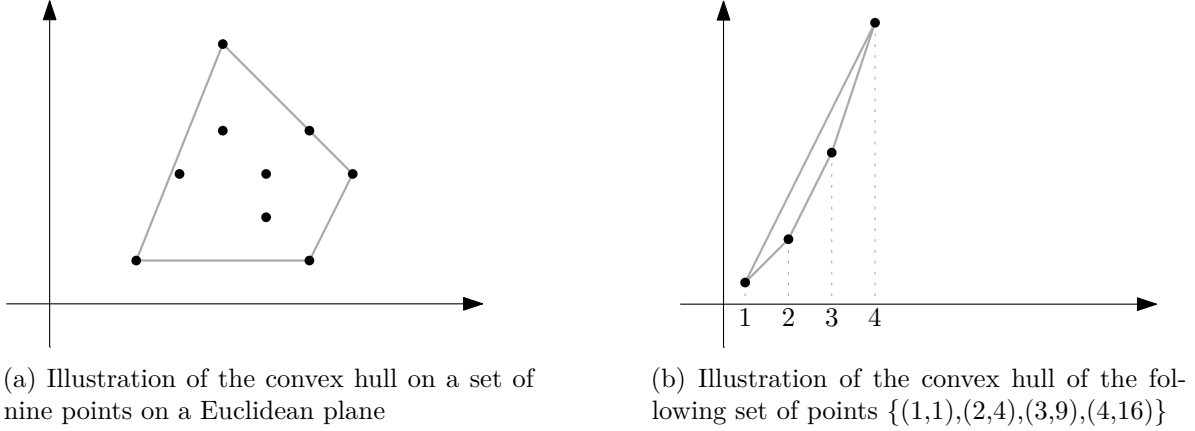
Figure 1.5: Collection of convex hull illustrations

than the actual computation of the solution of the transformed instance. In other words, the time needed for the transformation is strictly upper bounded by the time needed to compute the solution of the transformed instance. To express that instances of size n of a problem A can be transformed to instances of a problem B in time $\mathcal{O}(f(n))$, we usually write:

$$\text{Problem A} \leq_{O(f(n))} \text{Problem B}$$

As an example, we consider the convex hull problem, which given a set of n points on the plane, asks for a convex polygon of minimum area that contains all input points either in its interior or on its boundary. Recall that a polygon is convex if the line segment connecting any two points either on the boundary or in the interior of the polygon lies completely in the interior of the polygon; see, e.g., Fig. 1.5a.

The convex hull problem seems to be considerably different from the problem of sorting elements with comparisons, as the latter is defined on a set of pairwise comparable elements (e.g., real numbers) while the former is defined on a set of points on the plane (which, e.g., are identified by their coordinates).

Furthermore, notice that there exist several algorithms to solve the convex hull problem in time $\mathcal{O}(n \log n)$; we will also study such an algorithm in a later chapter. To determine a lower bound for the convex hull problem though, it is necessary to consider any possible algorithm. To circumvent this issue, we perform a linear-time transformation from the sorting problem with comparisons to the convex hull problem.

> Lemma 1.2 Let A be an array with n elements $x_1, \ldots, x_n$. The convex hull of the points in $\{(x_1, x_1{}^2), \ldots, (x_n, x_n{}^2)\}$ determines a sorting of the elements of A.

Proof. Clearly, the transformation described in the statement of the theorem can be done in linear time as every element needs to be processed once to create the coordinates of the points for the instance of the convex hull problem. As illustrated in Fig. 1.5b, the convex hull of these points approximate a quadratic parabola. Since the latter is convex, all constructed

points lie on its boundary and thus on their convex hull. Furthermore, in a counter-clockwise traversal of this convex hull (x_i, x_i^2) appears before (x_j, x_j^2) if and only if $x_i < x_j$. □

Corollary 1.1 Sorting with comparisons $\leq_{O(n)}$ Convex hull

---

Theorem 1.12 The problem of computing the convex hull of n points on the plane is in $\Omega(n \log n)$

Proof. Assume to the contrary that there exists an algorithm to compute the convex hull in $o(n \log n)$ time. By Lemma 1.2 and Corollary 1.1 this would imply that the sorting problem can be solved in time

$$o(n \log n) + \mathcal{O}(n) = o(n \log n)$$

However, this contradicts Theorem 1.11. □

Exercises

Exercise 1.1 Prove or disprove the following statements:

(a) $4^{n-1} \in \mathcal{O}(2^n)$

(b) $2^{\log(n^n)} \in \Omega(2^n)$

(c) $3^n \in o(5^n)$

(d) $\sqrt{2n^2 + 3n} - n \in \Theta(n)$

(e) $\sqrt{n^2 + 3n} - n \in \Theta(1)$

Exercise 1.2 Order the following functions by their asymptotic growth. Prove the correctness for each of the six successive pairs.

$$A : \log n \quad B : 2^{\sqrt{\log n}} \quad C : (\log n)^{\log n} \quad D : \log(n!)$$
$$E : (\log n)^{\sqrt{n}} \quad F : (\log(\log n))^n \quad G : n$$

Exercise 1.3 Let A be a sorted array with n elements. Given an input element x, the goal is to determine whether x belongs to A. Prove that in worst-case one needs to perform $\Omega(\log n)$ comparisons for this problem (independent from the algorithm that one will use).

Exercise   1.4   In binary search, the input is a sorted array A with n elements and a given object x. The task is to test whether x belongs to A. As you know, this can be done asymptotically in $\mathcal{O}(\log n)$ time, by splitting the array into two equal parts and by recursively driving the search in one of these two parts. Another alternative called ternary search is to split the array into three parts of equal size and recursively drive the search in one of these three parts.

(a) Give the recurrence relation that expresses the time requirement of ternary search.

(b) Prove that ternary search asymptotically needs the same time as binary search.

(c) Which of the two methods is performing more comparisons, the binary or the ternary search?

Exercise   1.5   Let $u, v, w \in \Sigma^*$ be three words over an alphabet Σ. Word w is called a mixture of words u and v, if there exist decompositions $u = u_1 \ldots u_k$ and $v = v_1 \ldots v_l$ such that $u_i, v_j \in \Sigma^*$ for $i = 1, 2, \ldots, k$ and $j = 1, 2, \ldots, l$ and $w = u_1 v_1 u_2 v_2 \ldots$
As for an example consider the words $u = INFORMATIK$ and $v = algorithmen$. Then, the word $w = INaFOlgoRMrithAmTIenK$ is a mixture of u and v.

(a) Give a dynamic programming based algorithm, which given three words u, v, w determines whether w is a mixture of u and v.

(b) Discuss the running time of your algorithm w.r.t. the lengths of the input words.

Further reading

– Chapters 3 and 4 from: T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009.

DRAFT

2. Amortized Analysis

In this chapter, we continue our discussion on the complexity analysis of algorithms. While in Chapter 1 we mainly focused on the worst-case complexity analysis, here we will focus on the average-case complexity analysis and on amortized analysis. In particular, in Section 2.1 we deal with analyzing the complexity of an algorithm on average. Amortized analysis is discussed in Section 2.2. Finally, in Section 2.3 we use the method to analyze the amortised time complexity requirements of an efficient implementation of the priority-queue data structure.

2.1 Complexity analysis on average

As already mentioned, analysing the complexity of an algorithm in worst case is a standard approach to determine the efficiency of the algorithm. Sometimes, however, we also seek to know the complexity of an algorithm in the "average case". To clarify this, let us look at a rather simple algorithm, which given a (not-necessarily sorted) array of length n, returns its minimum element.

Algorithm 2.1 provides pseudocode for such an algorithm. At the beginning, it initializes the minimum to the first element of the array and then iterates over the rest of the array. At each iteration, an new element is encountered; if this element is smaller than the current minimum, then the algorithm sets this element to the minimum.

In the following, we will analyze the complexity of this algorithm, first in the worst case, and then in the average case. Let c_1 be the cost of executing the assignment in line 2 of Algorithm 2.1, c_2 the cost of performing the comparison in line 4, and c_3 the cost of the assignment in line 5. Hence, in worst case, the algorithm's complexity is:

$$c_1 + c_2 \cdot (n-1) + c_3 \cdot (n-1).$$

However, performing the assignment of line 5 of Algorithm 2.1 at every iteration of the algorithm seems quite unlikely. Indeed, if this were the case, then the array would have to be

---

Algorithm 2.1: An algorithm to find the minimum element of an unsorted array.

   Input   : An array A of n elements.
   Output : The minimum element contained in the array.

1 Function FindMin($A[1,\ldots,n]$)
2   min = A[1];
3   for $i = 2$ to n do
4   if $A[i] < min$ then
5   min = A[i];
6   end
7   end
8   return min;
9 end

---

sorted in decreasing order. However, if the size of the array is n, the probability for having a decreasingly sorted array is $\frac{1}{n!}$, which is extremely unlike even for relatively small values of n. This observation motivates the complexity analysis on average, in which the study focuses on the input case that one expects to happen.

To this end, we first define a discrete random variable X_i which takes the value 1 if and only if, among the first i elements, the last one is the smallest one; otherwise it takes the value 0. In other words:

$$X_i = \begin{cases} 1, & \text{if } A[i] < A[1],\ldots,A[i-1]. \\ 0, & \text{otherwise.} \end{cases}$$

Since X_i is a discrete 0/1-variable, the expected value of X_i is the probability of X_i taking the value 1. Hence, we obtain:

$$E[X_i] = \mathcal{P}(X_i = 1) = \mathcal{P}(A[i] < A[1],\ldots,A[i-1]) = \frac{1}{i}.$$

Using the linearity of expected values (i.e., $E[\sum_{i=0}^{n} X_i] = \sum_{i=0}^{n} E[X_i]$), and the fact that the n-th harmonic number (i.e., $\frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n}$) is upper bounded by $2 + \log n$ (by Lemma 1.1), we can rewrite the time complexity of Algorithm 2.1 as follows:

$$\begin{aligned} T(n) &= c_1 + c_2 \cdot (n-1) + c_3 \cdot E[X_2 + \cdots + X_n] \\ &= c_1 + c_2 \cdot (n-1) + c_3 \cdot (E[X_2] + \cdots + E[X_n]) \\ &= c_1 + c_2 \cdot (n-1) + c_3 \cdot (\frac{1}{2} + \cdots + \frac{1}{n}) \\ &\leq c_1 + c_2 \cdot (n-1) + c_3 \cdot (2 + \log n) \end{aligned}$$

The latter confirms our intuition that the assignment of line 5 of Algorithm 2.1 is most likely not executed at every iteration of the algorithm.
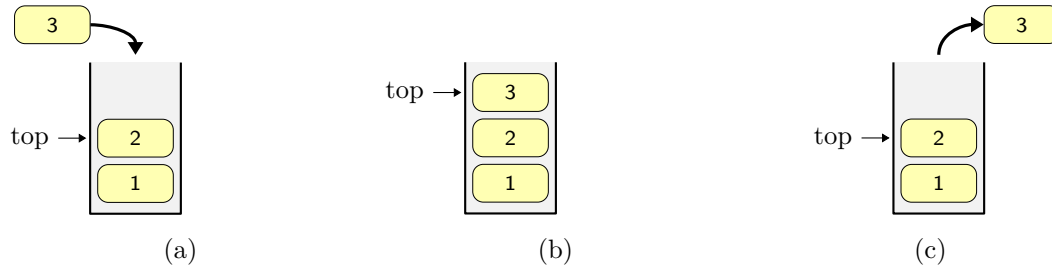
Figure 2.1: Illustration of a Push operation (of element 3) followed by a Pop operation (of element 3 again, which was the most recently added).

2.2   Amortized analysis

The goal of an amortized analysis is to determine the average cost per operation in a sequence of n operations under the assumption that in this sequence of operations several of them are "cheap", while a few are "expensive". Under this assumption, the goal is to prove that the average cost per operation (called amortised cost) is small for any such sequence of operations. The principle idea in this approach is to show that the cost of the expensive operations is "cancelled out" by the cheap operations (due to their number) or in other words that the expensive operations are so few the their total cost does not exceed the corresponding one of the cheap operations.

Note that the amortized analysis is different from the average-case analysis, which we studied in Section 2.1. In an average-case analysis, we have to estimate probabilities for how often we expect an operation to be performed. The amortized analysis, however, gives a guarantee for the average performance of each operation in the worst case, i.e., it specifies an upper bound for the average cost of an operation.

There are three main techniques to perform an amortized analysis. We will introduce and study each of them using the example of a multipop stack.

2.2.1   The Multipop Stack

A stack is a data structure which operates under the Last In First Out principle; the element which was most-recently added to the stack is the one which will be returned first. A regular stack has two fundamental operations:

- Push(S,x) pushes an element x in stack S.

- Pop(S) removes the most-recently added element from S and returns it.

Clearly, both of these operations can be implemented in constant time, i.e., $\mathcal{O}(1)$ time. For an illustration, refer to Fig. 2.1, which depicts a Push operation followed by a Pop operation.
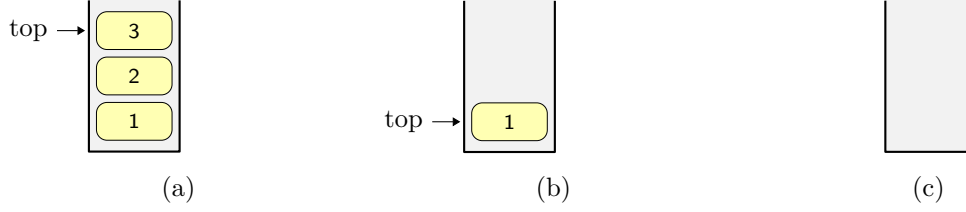
Figure 2.2: The effect of a Multipop operation. Calling Multipop(S,2) on the initial stack shown in (a) causes the top two elements to be removed. Calling Multipop(S,2) again on the stack from (b) empties the stack, as shown in (c).

In this section, we consider a variant of a regular stack, which is called multipop stack, as it supports the following additional operation:

- Multipop(S,k) removes the k most-recently added elements from S or all elements of S if they are fewer than k.

Clearly, the Multipop(S,k) operation can be implemented in $\mathcal{O}(\min\{k,|S|\})$ time, that is, in time linear in the number of elements which have to be removed. For an illustration, refer to Fig. 2.2.

Consider now a sequence of n operations supported by a multipop stack. To determine the worst-case time-complexity of the sequence, one has to consider the worst-case run-time of an operation and multiply it with the number of these operations in the sequence, namely by n. This implies that in worst case one may conclude a time complexity of $\mathcal{O}(n^2)$ for a sequence of n operations, since a Multipop operation may need $\mathcal{O}(n)$ time. However, as one can readily see, it is simply not possible to remove $\mathcal{O}(n)$ objects at each application of a Multipop operation. In the following, we will use amortized analysis to take this observation into account aiming to improve the total time complexity of the sequence of operations.

2.2.2   The Accounting Method

The idea behind the accounting method is to express the actual cost of each operation in terms of a number of coins, e.g., if the actual cost of an operation is $\mathcal{O}(1)$, we assume that it costs a single coin. The goal is to find an upper bound on the total cost of a sequence of n operations in terms of coins (if a sequence of n operations costs n coins, then the amortised cost per operation is unit, i.e., $\mathcal{O}(1)$). In the accounting method, this upper bound is found by developing an accounting scheme which charges each "cheap" operation with a few coins more than those needed to cover its actual cost (the amount of coins an operation is charged is called its amortized cost). The difference in coins is then deposited as credit into a "bank" and is subsequently used to "pay" the actual costs of "expensive" operations that follow. Quite important in this method is to guarantee that (i) the credit in the bank never becomes negative, and (ii) it is enough to cover the cost of the expensive operations.

Assuming that c_i denotes the actual cost of the i-th operation and $\hat{c}_i$ denotes the amortized cost of the i-th operation (in terms of coins), then Requirement (i) is guaranteed by ensuring

Operation	Actual Cost	Amortized Cost
Push	1	2
Pop	1	0
Multipop	min{k, |S|}	0

(a)
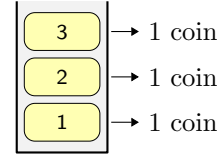
(b)

Figure 2.3: (a) The actual and amortized costs for the stack operations. (b) Illustration of the stored credit in the stack.

that the following relation holds for any sequence of n operations:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \tag{2.1}$$

Requirement (ii) is guaranteed by distributing the total credit in the bank to specific objects (e.g., to elements of our multipop stack). To clarify these concepts, we turn our attention back to the multipop stack example. Fig. 2.3a recalls the actual cost of each operation supported by the multipop stack and provides an accounting scheme, according to which we overcharge each Push operation so as to allow to charge nothing for a Pop or Multipop operation.

In particular, when we perform a Push operation, we are charged an amortized cost of two coins. One of these two coins "pays" the actual cost of the operation, while the other coin is stored as credit in the bank. This means that every object in the stack has an associated coin as illustrated in Fig. 2.3b. This further implies that the total credit in the bank equals the number of objects in the stack and since the stack cannot accommodate a negative number of objects, the credit in the bank never becomes negative as required.

Since each object, which has to be removed from the stack, has been associated with a coin in the bank, the actual cost of each Pop or Multipop operation can be easily covered by the credit in the bank. In other words, every object in the stack "pays" for itself (with the one coin that has been associated with) in order to be removed from the stack.

For a sequence of n operations, the total amount of coins which has to be spent is at most $2n$. Thus, the cost for a sequence of n operations is in $\mathcal{O}(n)$ and the average cost of a single operation is in $\mathcal{O}(1)$ (i.e., $2n/n = 2$).

2.2.3   The Aggregate Method

The aggregate method estimates the amortized cost of an operation by providing an upper-bound on the total cost of a sequence of n operations. More precisely, if we denote by $T(n)$ the worst-case cost of any sequence of n operations, then it follows that the amortized cost per operation is:

$$\frac{T(n)}{n}$$

We will now see how to apply the aggregate method to the multipop stack example. Let x and y be the number of Push and Pop operations performed in a sequence of n operations, respectively. Assuming that each Multipop(S,k) operation is composed by $\min\{k, |S|\}$ Pop operations, we can conclude that $x + y = n$. The critical observation here is that in order to Pop an object from the stack, it is required to Push it first into the stack. Therefore, $y \leq x$ holds. Assuming that each Push and each Pop operation has unit cost, we can deduce:

$$T(n) = \sum_{i=1}^{x} 1 + \sum_{i=1}^{y} 1 = x + y = n$$

Therefore, any sequence of n operations has a total cost of n. This implies that the amortized cost per operation is unit, that is, $\mathcal{O}(1)$.

2.2.4   The Potential Method

The potential method is similar to the accounting method. More precisely, whereas in the accounting method we stored credit in a bank and we associated it with individual objects, in the potential method we store credit as potential energy, or just potential, in a data structure. So, if the actual cost of an operation is lower than its amortized cost, then the potential increases, while if the actual cost of an operation is higher than the amortized cost, then some of the stored potential in the data structure is released (to cover the cost of the operation).

More precisely, denote by D_0 the initial state of the data structure and by D_i the state of the data structure after the i-th operation. The actual cost of the i-th operation is denoted by c_i. In the potential method, the challenge is to define a so-called potential function Φ, which maps each state D_i of the data structure to a real-valued number representing the stored potential in that state. The amortized cost $\hat{c}_i$ of the i-th operation is then defined as:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}), \tag{2.2}$$

that is, the amortized cost of an operation is its actual cost plus the change in potential caused by the operation. If the change in potential is positive, then the operation was overcharged. Otherwise, the operation was undercharged, which means that potential had to be released (to cover the cost of the operation). Eq. (2.2) implies that the total amortized cost of a sequence of n operations is:

$$\sum_{i=1}^{n} \hat{c}_i \overset{(2.2)}{=} \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \left(\sum_{i=1}^{n} c_i\right) + \Phi(D_n) - \Phi(D_0). \tag{2.3}$$

 The goal (and, simultaneously, the main difficulty in the potential method) is to define the potential function Φ in such a way that the total amortized cost of a sequence of n operations is an upper bound for its total actual cost, namely, $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$. By Eq. (2.3), we can deduce that the goal is achieved if and only if:

$$\Phi(D_n) \geq \Phi(D_0),$$

An alternative way to achieve the aforementioned goal is by defining Φ such that:

$$\Phi(D_i) \geq \Phi(D_0), \quad \forall i = 1, 2, \ldots, n.$$

This way, we guarantee that the potential of the data structure is never negative. Since, in most cases, the potential in the data structure in initially zero (that is, $\Phi(D_0) = 0$), it suffices to guarantee that, in subsequent states, Φ stays non-negative, that is,

$$\Phi(D_0) = 0, \quad \Phi(D_i) \geq 0, \forall i = 1, 2, \ldots, n$$

Let us now return one more time back to the multipop-stack example. As already mentioned, the challenge is to define the potential function. Inspired by the fact that, in the accounting method, the total credit stored in the bank equals the number of objects in the stack, we define the potential function Φ such that $\Phi(D_i)$ equals the number of objects in the stack after the i-th operation. Since initially the stack is empty and since a stack cannot contain a negative number of objects, it follows that:

$$\Phi(D_0) = 0, \quad \Phi(D_i) \geq 0, \forall i = 1, 2, \ldots, n$$

Therefore, the total amortized cost of n operations is an upper bound for their total actual cost, as desired. Having defined Φ, we analyze in the following the amortized cost of each operation of the multipop-stack; see Eq. (2.2).

- If the i-th operation is a Push operation, then the change $\Phi(D_i) - \Phi(D_{i-1})$ in the potential is $+1$ (since after the i-th operation the stack contains one element more than before). This implies that the amortised cost $\hat{c}_i$ of the i-th operation is:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2.$$

- If the i-th operation is a Pop operation, then the change $\Phi(D_i) - \Phi(D_{i-1})$ in the potential is -1 (since after the i-th operation the stack contains one element less than before). This implies that the amortised cost $\hat{c}_i$ of the i-th operation is:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0.$$

- If the i-th operation is a Pop operation, then the change $\Phi(D_i) - \Phi(D_{i-1})$ in the potential is $-min\{k, s\}$ (since after the i-th operation the stack contains $\min\{k, s\}$ element less than before). This implies that the amortised cost $\hat{c}_i$ of the i-th operation is:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = \min\{k, s\} - \min\{k, s\} = 0.$$

As explained above, the total actual cost for a sequence of n operations is upper-bounded by the total amortized cost. Since by our case-analysis above, the total amortized cost is at most $2n$, then the amortized cost per operation is $2 = \mathcal{O}(1)$.

DRAFT

Algorithm 2.2: A general algorithm to solve the single-source shortest path problem.

    Input   : An edge-weighted directed graph $G = (V, E, w)$ and a designated vertex $s \in V$.
    Output: For every vertex v of G, the length $d[v]$ of the shortest path from s to v.

1 $d[s] = 0$;
2 foreach $v \in V \setminus \{s\}$ do
3 | $d[v] = \infty$;
4 end
5 while $\exists (u, v) \in E$ with $d[v] > d[u] + w(u, v)$ do
6 | $d[v] = d[u] + w(u, v)$;
7 | $\pi[v] = u$;   // to restore the path
8 end

2.3   Dijkstra's algorithm: A more advanced application

Dijkstra's algorithm solves a very important algorithmic problem, the single-source shortest path problem. The input in this problem is an edge-weighted directed graph $G = (V, E, w)$ and a designated vertex s in V, where $w : E \rightarrow \mathbb{R}^+$. In the output, we seek to compute the (length of the) shortest path from s to each vertex of G, where the length of a path is the sum of the weights of its constituent edges. Equivalently, the output can be encapsulated by a function $\delta : \{s\} \times V \rightarrow \mathbb{R}^+$ such that:

$$\delta(s, v) = \text{ length of the shortest path from } s \text{ to } v.$$

Algorithm 2.2 provides pseudocode for a general approach to solve the single-source shortest path problem that is, in a sense, the core of any shortest path algorithm. Note that $\pi[v]$ is only needed if we additionally want to return the actual shortest path, and not only its length, as the shortest path can then be restored by using the pointers $\pi[v]$ with $v \in V \setminus \{s\}$. The correctness of Algorithm 2.2 can be proved by showing that the following invariant holds true at any step of the algorithm:

$$d[v] \geq \delta(s, v) \ \forall v \in V$$

Indeed, the invariant holds true prior to the first iteration of the while-loop, since $d[s] = 0 \geq \delta(s, s)$ and $d[v] = \infty \geq \delta(s, v)$ for every vertex v different than s. Furthermore, the invariant is maintained at each iteration of the while-loop, since:

$$d[v] = d[u] + w(u, v) \geq \delta(s, u) + w(u, v) \geq \delta(s, u) + \delta(u, v) \geq \delta(s, v)$$

The last inequality holds by the triangle inequality. Having proved the correctness of the algorithm, we note that the algorithm will terminate once the length of the shortest path from s to every vertex of the input graph is computed, since at each iteration of the algorithm there exists a vertex v, whose $d[v]$-value is decreased.

Dijkstra's algorithm (refer to Algorithm 2.3) is an adjustment of Algorithm 2.2. Its main idea lies on maintaining a set S of vertices such that $\delta(s, v)$ has been computed for each vertex v in S. Then, the next vertex to be considered by the algorithm is the one from $V \setminus S$ that

---

Algorithm 2.3: Dijkstra's algorithm

Input   : An edge-weighted directed graph $G = (V, E, w)$ and a designated vertex $s \in V$.
Output : For every vertex v of G, the length $d[v]$ of the shortest path from s to v.

1 $d[s] = 0$;
2 PQ.insert(s, $d[s]$);
3 foreach $v \in V \setminus \{s\}$ do
4 $\quad$ $d[v] = \infty$;
5 $\quad$ PQ.insert($v, d[v]$);
6 end
7 while !PQ.isEmpty() do
8 $\quad$ $u = $ PQ.delete_min();
9 $\quad$ foreach $(u, v) \in E$ do
10 $\quad\quad$ if $d[v] > d[u] + w(u, v)$ then
11 $\quad\quad\quad$ $d[v] = d[u] + w(u, v)$;
12 $\quad\quad\quad$ $\pi(v) = u$;
13 $\quad\quad\quad$ PQ.decrease_key($v, d[v]$);
14 $\quad\quad$ end
15 $\quad$ end
16 end

---

is "closest" to (some vertex in) S. The correctness of Dijsktra's algorithm can be proved by showing that the following invariant is maintained in the course of the algorithm: "when a vertex u is extracted from the priority queue PQ, it holds that $d[u] = \delta(s, u)$". In other words, set S consists of the vertices of G which have been extracted from PQ. The detailed proof of this claim is left as an exercise to the reader (see Exercise 2.3) and we proceed to discuss the time complexity of the algorithm.

The algorithm is implemented using the priority queue data structure, which supports the following operations.

- insert(x, k): inserts an element x with key k to the data structure.

- find_min(): returns the element with the minimum key.

- delete_min(): returns and deletes from the data structure the element with the minimum key.

- decrease_key(x, k): decreases the key of the element x to k.

Let n be the number of vertices and m the number of edges of the input graph G. Since each vertex is inserted in the priority queue exactly once, and the algorithm terminates, when the priority queue is empty, there exist in total n calls to insert and extract_min operations. Furthermore, since each edge of the graph is examined exactly once during the algorithm, there exist m calls to decrease_key in the course of the algorithm. Hence, the time complexity of Dijkstra's algorithm is as follows:

$$\mathcal{O}(n \cdot (T_{\text{insert}} + T_{\text{extract\_min}}) + m \cdot T_{\text{decrease\_key}})$$

where $T_{\text{operation}}$ denotes the complexity of the specified operation. Thus, the complexity depends on the implementation of the priority queue PQ. An easy way to implement a priority queue is by using a doubly-connected list. A more involved method is using a binary heap with the min-heap property, that is, of an almost-balanced binary tree such that the key of a node in the tree is not smaller than the key of its parent. The following table gives an overview of the complexities of the operations of a priority queue using different implementations.

	Doubly-connected list	Binary heap	Fibonacci heap
insert	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
find_min	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
delete_min	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
decrese_key	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

In particular, the complexity of Dijkstra's algorithm using a doubly-connected list to implement the priority queue is $\mathcal{O}(n^2 + mn)$, while using a binary heap it can be decreased to $\mathcal{O}(n \log n + m \log n)$ time. The goal is to decrease the complexity even further. We will see that this can be achieved by using Fibonacci heaps; the complexity will be decreased to $\mathcal{O}(n \log n + m)$, since in a Fibonacci heap all operations have constant amortized costs except for delete_min, whose amortised cost is logarithmic in n.

2.3.1   Fibonacci Heaps

Fibonacci heaps were introduced by Fredman and Tarjan back in 1984 as an efficient implementation of a priority queue. Their actual name comes from the fact that the Fibonacci numbers appear during the analysis; we will see this later in this section.

A Fibonacci heap is a collection of trees with the min-heap property, that is, the key of a node is not smaller than the key of its parent. Each node in a tree can be marked and has a pointer to its parent, pointers to its children, and two pointers to its left and right siblings (if any), that is, siblings are connected via circular doubly-linked lists. The roots of all trees are also connected via a circular, doubly-linked list, which is called root list. By the min-heap property, the element with the minimum key is contained in the root list. To guarantee constant access to it, the data structure maintains a point to this element, called min-pointer (see Fig. 2.4).

Note that doubly-linked lists support additions and deletions of elements as well as list-mergings in constant time. Also, note that the marking of the vertices plays an important role in the analysis of the required amortised time of each operation.

In the following, we describe how each operation is implemented and employ the potential methods to prove the amortized costs. Let H be an instance of a Fibonacci heap at a certain state (that is, after the application of a particular number of operations). We denote by $t(H)$ the number of trees in H, and by $m(H)$ the number of marked vertices in H. We define the
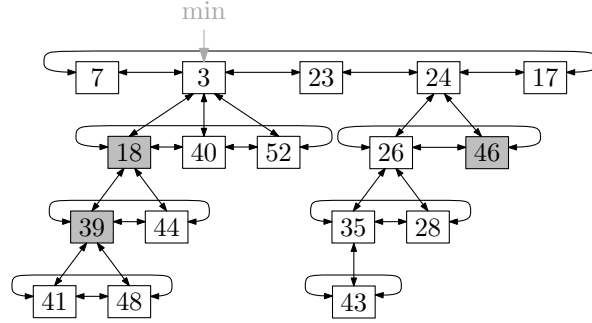
Figure 2.4: An example of a Fibonacci heap.

potential function as follows:

$$\Phi(H) = t(H) + 2m(H). \tag{2.4}$$

Since $m(H) \geq 0$ and since $t(H) \geq 0$, it follows that $\Phi(H) \geq 0$. If H corresponds to the initial state, then $\Phi(H) = 0$ (since H is empty). Let H' be the instance of Fibonacci heap after a specific operation of actual cost c has been applied to H. Then, the amortized cost $\hat{c}$ of this operation is defined as

$$\hat{c} = c + \Phi(H') - \Phi(H), \tag{2.5}$$

that is, the actual cost plus the change in the potential. In the following, we distinguish cases for the type of operation performed in the transition from H to H'. In our analysis, we assume that a unit of potential can pay for a constant amount of work. Hence, this constant is sufficiently large to cover the cost of any constant time operation, which will be needed later.

Finding the minimum: We first assume that the operation performed in the transition from H to H' is find_min(). Since the element with the minimum key can be identified through the min-pointer, the actual cost of this operation is $\mathcal{O}(1)$. Furthermore, this operation does not imply a change in the potential, since both the number of trees and the number of marked vertices remain unchanged (that is, $m(H') = m(H)$ and $t(H') = t(H)$). So, by Eq. (2.5), the amortized cost of this operation is equal to its actual cost, which is $\mathcal{O}(1)$.

Inserting an element: Next, we assume that the operation performed in the transition from H to H' is insert(x, k). To insert a new element x with key k in the Finonacci heap, we create a new tree consisting of a single node that contains the element x with key k. Then, we insert this tree into the root list and we update the min-pointer (if needed); see Fig. 2.5 for an illustration. It follows that the actual cost for this operation is $\mathcal{O}(1)$, because creating a tree consisting of one node can be done in constant time and updating the pointer to the minimum needs only one comparison. Since $m(H') = m(H)$ and $t(H') = t(H) + 1$, it follows that the change in the potential is unit, which by Eq. (2.5) implies that the amortized cost of this operation is $\mathcal{O}(1)$.

Deleting the element with minimum key: In this case, the operation performed in the transition from H to H' is delete_min(). To delete the element with the minimum key, we identify its corresponding node in the Fibonacci heap using the min-pointer. Then, we remove it
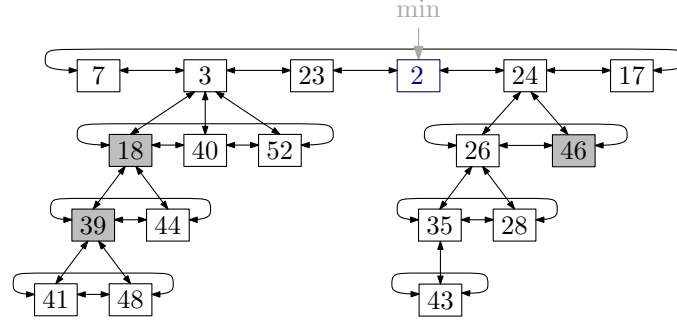
Figure 2.5: The result of inserting an element with key 2 into the Fibonacci heap of Fig. 2.4.

from the root list and add its children in the root list (if any). Afterwards, we "tidy up" the Fibonacci heap, by consolidating the root list, so that no two roots have the same degree as follow (see Fig. 2.6). As long as there exist two trees with the same degree i, we merge them into a tree with degree $i + 1$ by making the tree with the larger root-key a child of the root of the other tree (if this root was marked, we also clear the mark). Using an auxiliary array of length $d_{\max}(H) + 1$, it can be easily proved that the actual cost of this operation is $\mathcal{O}(d_{max}(H) + t(H))$, where $d_{\max}(H)$ denotes the maximum degree of any node in H; proving the exact time bound is left as an exercise for the reader (see Exercise 2.4).
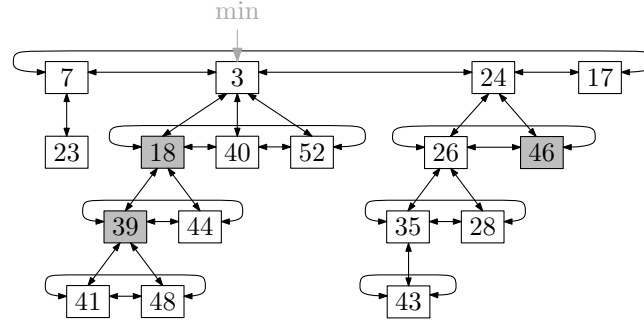
Figure 2.6: The result of deleting the element with key 2 from the Fibonacci heap of Fig. 2.5.

Since no node becomes marked, $m(H') \leq m(H)$ holds. Since no two roots have the same degree in the root list after the application of the operation, $t(H') \leq d_{\max}(H) + 1$ holds. Hence, the change in the potential is:

$$\Phi(H') - \Phi(H) = t(H') + 2m(H') - (t(H) + 2m(H)) \leq d_{\max}(H) + 1 - t(H).$$

Hence, by Eq. (2.5), the amortised cost of the operation is in worst case:

$$\begin{aligned}
\hat{c} &= c \cdot (d_{\max}(H) + t(H)) + d_{\max}(H) + 1 - t(H) \\
&= (c + 1) \cdot d_{\max}(H) + (c - 1) \cdot t(H) + 1 \\
&= \mathcal{O}(d_{\max}(H)),
\end{aligned}$$

Decreasing the key of an element: In this case, the operation performed in the transition from H to H' is decrease_key(x, k). This operation may result in restructuring the data structure, in order to preserve the min-heap property at each tree of the root list. In particular, a node

may loose a child, as we will shortly see. The first time a node looses a child, it becomes marked. The second time it looses a child, it becomes a root. This becomes clear in the following, where we distinguish three main cases.

C.1 Decreasing the key of node x does not violate the min-heap property. In this case, we simple decrease the key value of x and we are done. Hence, the actual cost of this operation is $\mathcal{O}(1)$. Since there is no change in the potential, the amortized cost of the operation is equal to its actual cost, that is, $\mathcal{O}(1)$.

C.2 Decreasing the key of node x violates the min-heap property; the parent p of node x is not marked. In this case, we proceed by decreasing the key-value of x and by cutting the parent edge (p, x) of x, where the operation of cutting the edge (p, x) consists of the following steps (see Fig. 2.7):

– remove edge (p, x) from the tree containing x,

– add the subtree rooted at x as a new tree in the root list,

– update the min-pointer, if necessary,

– unmark x, if it previously was marked, and

– mark p (since p lost a child).

It follows that the actual cost of this operation is $\mathcal{O}(1)$. Since $m(H') \leq m(H) + 1$ and $t(H') = t(H) + 1$, the change in the potential is as follows:

$$\Phi(H') - \Phi(H) = t(H') - t(H) + 2(m(H') - m(H)) \leq 1 + 2 = 3,$$

Thus, the amortized cost is $\mathcal{O}(1)$.
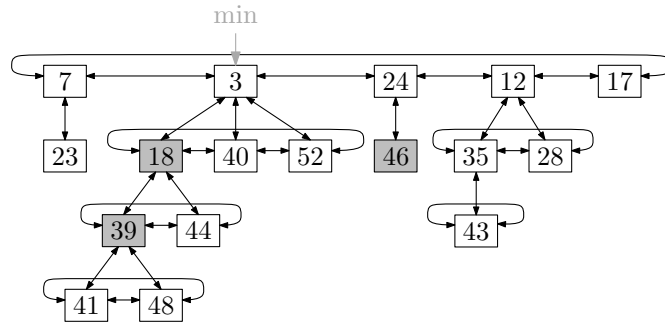

Figure 2.7: The result of decreasing key-value 26 to 12 of the Fibonacci heap of Fig. 2.6.

C.3 Decreasing the key of node x violates the min-heap property; the parent p of node x is marked. In this case, we first decrease the key-value of x and then we cut the edges of the path from x to the root of the tree containing x, until we encounter an unmarked
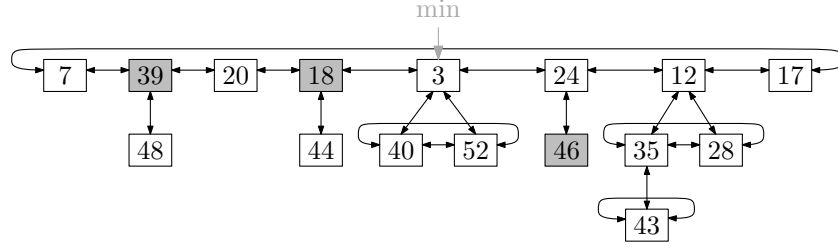
Figure 2.8: The result of decreasing key-value 41 to 20 of the Fibonacci heap of Fig. 2.7.

node of this path (see Fig. 2.8). Formally, this is achieved by the following procedure.

> while x is not marked do
> | $p \leftarrow$ the parent of x;
> | Cut edge (p, x);
> | $x \leftarrow p$;
> end
> if x is not the root then
> | Mark x;
> end

If we denote by ℓ the number of edges that are cut by the procedure above, then the actual cost of this operation is linear in ℓ, say $\ell + 1$ where the additional unit corresponds to the cost of decreasing the key of x. Since $t(H') = t(H) + \ell$ and since $m(H') \leq m(H) - (\ell - 1)$, the change in the potential is:

$$\Phi(H') - \Phi(H) = t(H') - t(H) + 2(m(H') - m(H)) \leq \ell + -2(\ell - 1) = 2 - \ell,$$

Therefore, by Eq. (2.5) the amortized cost is at most $\ell + 1 + 2 - \ell = 3 = \mathcal{O}(1)$.

The following observation follows from the cases that we distinguished above.

> Observation 2.1 The first time a node looses a child, it becomes marked. The second time it looses a child, it becomes a root.

So far, we guarantee amortized costs of $\mathcal{O}(1)$ for insert, find_min, and decrease_key. The amortized cost for delete_min is $\mathcal{O}(d_{\max}(H))$. In the following, we prove that $d_{\max}(H) = \mathcal{O}(\log n)$, where n is the number of nodes in H. Since Fibonacci numbers will be central in our analysis, we first recall their formal definition.

> Definition 2.1 (Fibonacci numbers) The k-th Fibonacci number is defined as follows:
>
> $$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k > 1 \end{cases}$$

We start with two auxiliary properties.

> Property 2.1 $F_{k+2} \geq \phi^k$, where ϕ is the golden ratio, that is, $\phi = \frac{(1+\sqrt{5})}{2} \approx 1.618$.

Proof. We prove the property by induction. The induction base correspond to the cases $k = 0$ and $k = 1$, which obviously hold true, since:

$$F_2 = 1 = \phi^0 \text{ and } F_3 = 3 \geq \phi^1.$$

For the inductive hypothesis, we assume $F_k \geq \phi^{k-2}$ and $F_{k+1} \geq \phi^{k-1}$. In the inductive step, we consider F_{k+2} and we argue as follows:

$$F_{k+2} = F_{k+1} + F_k \overset{I.H.}{\geq} \phi^{k-1} + \phi^{k-2} = \phi^{k-2} \cdot (\phi + 1) \overset{\phi^2 = \phi+1}{=} \phi^{k-2} \cdot \phi^2 = \phi^k \qquad \square$$

> Property 2.2 $F_{k+2} = 1 + \sum_{j=0}^{k} F_j$.

Proof. We prove the property by induction. The induction base corresponds to the case $k = 0$, which obviously holds true, since:

$$F_2 = 1 + \sum_{j=0}^{0} F_j = 1 + F_0 = 1.$$

For the inductive hypothesis, we assume $F_{k+1} = 1 + \sum_{j=0}^{k-1} F_j$. In the inductive step, we consider F_{k+2} and we argue as follows:

$$F_{k+2} = F_{k+1} + F_k = 1 + \sum_{j=0}^{k-1} F_j + F_k = 1 + \sum_{j=0}^{k} F_j \qquad \square$$

> Lemma 2.1 Let x be a node of degree k and let $y_1, y_2, \ldots, y_k$ be the children of x in the order in which they were linked to x. Then,
>
> $$\deg(y_i) \geq \begin{cases} 0 & \text{if } i = 1 \\ i - 2 & \text{if } i > 1 \end{cases}$$

Proof. Consider the moment in which y_i was linked to x. In this moment, we know that the degree of x is exactly $i - 1$, since the children $y_1, \ldots, y_{i-1}$ were already linked to x. It follows that the degree of y_i is also $i - 1$ at this moment, since the algorithm links nodes of the same degree. Afterwards, y_i may loose at most one child, as otherwise y_i would have been in the root list (by Observation 2.1). Hence, $\deg(y_i) \geq i - 2$ holds, as desired. $\qquad \square$

By Lemma 2.1, it follows that the i-th child of any node in a Fibonacci heap has degree at least $i - 2$. Let s_k be the number of a nodes of a minimum-sized tree rooted at a node of degree k, with the property just mentioned. Then, the following property holds by definition.

> Property 2.3   If $i \geq j$, then $s_i \geq s_j$.

Clearly, $s_0 = 1$ and $s_1 = 2$ holds. In general, it holds that

$$
\begin{aligned}
s_k \quad &= \quad 1 + \sum_{i=1}^{k} s_{\deg(y_i)} \\
&= \quad 1 + s_{\deg(y_1)} + \sum_{i=2}^{k} s_{\deg(y_i)} \\
&\underset{\substack{\text{Prp.2.3} \\ \text{Lem.2.1}}}{\geq} \quad 1 + s_0 + \sum_{i=2}^{k} s_{i-2}
\end{aligned}
$$

Since $s_0 = 1$, we conclude that:

$$
s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \tag{2.6}
$$

Having expressed s_k as in Eq. (2.6), we prove in the following property that s_k is lower-bounded by F_{k+2}.

> Property 2.4 $s_k \geq F_{k+2}$

Proof. We prove the property by induction. The induction base correspond to the case $k = 0$, which obviously holds true, since:

$$
s_0 = F_2 = 1.
$$

For the inductive hypothesis, we assume $s_i \geq F_{i+2}$ for all $i = 0, \ldots, k - 1$. In the inductive step, we consider s_k and we argue as follows:

$$
s_k \overset{(2.6)}{\geq} 2 + \sum_{i=2}^{k} s_{i-2} \overset{I.H.}{\geq} 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i \overset{Prp.2.2}{=} F_{k+2} \qquad \square
$$

The following property follows by combining Property 2.1 and Property 2.4

> Property 2.5 $s_k \geq \phi^k$

We are now ready to prove that $d_{\max}(H) = \mathcal{O}(\log n)$.

> **Lemma 2.2** Let H be a Fibonacci heap with n nodes and let $d_{\max}(H)$ be the maximum degree of any node in H. Then, $d_{\max}(H) = \mathcal{O}(\log n)$.

Proof. Denote by x the highest-degree node in H and, for simplicity, call k the degree of x in H, that is, $k = d_{\max}(H)$. Let $\text{size}(x)$ be the number of nodes in the subtree rooted at x. It follows that $n \geq \text{size}(x)$. By definition of s_k, it follows that $\text{size}(x) \geq s_k$. Therefore:

$$n \;\geq\; \text{size}(x) \;\geq\; s_k \;\overset{Prp.2.5}{\geq}\; \phi^k$$

Thus:

$$k \leq \lfloor \log_\phi n \rfloor = \mathcal{O}(\log n) \qquad \square$$

Exercises

> **Exercise 2.1** Suppose we are maintaining a data structure under a series of operations. Let i be the running time of the i-th operation, if i is a power of 2; otherwise the running time is 1. Compute the amortized costs $\hat{c} = \frac{T(n)}{n}$ for a single operation, where $T(n)$ denotes the total costs for n operations.

> **Exercise 2.2** Consider a dynamic data structure (implemented using an array) for storing objects, which supports two operations delete and insert. Since the total number of objects is unknown in advance, the size of the array must be increased (or eventually decreased) dynamically. Compute the running time for the two operations (insert, delete) in following 3 scenarios (I) first in worst case and (II) second with amortized analysis. You may assume that the initial size of the array is 1.
>
> a) First scenario: The insert operation follows one of the two following cases. If the array is not full, then the new element is simply inserted. Otherwise we allocate a new array of double size, copy all elements from the old array to the new array and insert the new element finally. The delete operation removes an element without changing the size of the array.
>
> b) The insert operation of the second scenario is the same as in the first scenario, while the delete operation is different: after removing an element we check whether the number of objects is at least half of the array size. In the negative case we allocate a new array of half size and copy all elements from the old array to the new array.
>
> c) The third scenario is the same as the second scenario, but we half the array not until the number of objects is a quarter of the array size.

Exercise 2.3   Consider Dijkstra's algorithm as described in Algorithm 2.3.

a) Prove that when a vertex u is extracted from the priority queue PQ, it holds that $d[u] = \delta(s, u)$, where $\delta(s, u)$ corresponds to the length of the shortest part of u from s.

b) Give an example of an edge-weighted digraph, whose weight function is not positive, for which Dijkstra's algorithm produces an incorrect output. Point out where your proof from (a) uses the fact that $w(e)$ is nonnegative for each edge e.

Exercise 2.4   Consider an instance H of a Fibonacci heap.

a) Prove or disprove: If H contains n nodes, then the height of H is $\mathcal{O}(\log n)$.

b) Prove that the consolidation of the root list can be done in time

$$\mathcal{O}(d_{\max}(H) + t(H)),$$

where $d_{\max}(H)$ is an upper bound on the maximum degree of any node in H and, and $t(H)$ is the number of trees in the root-list.

Further reading

– Chapter 17 from: T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009.

3. Minimum Spanning Trees

This chapter will focus on the minimum spanning tree (MST) problem. In particular, the focus will be on an greedy algorithm, which unifies two popular algorithms to find a minimum spanning tree of a connected, edge-weighted undirected graph, namely, the Kruskal's algorithm and the Prim's algorithm. The rest of this chapter is organized as follows: In Section 3.1, we formally define the minimum spanning tree problem. In Section 3.2, we present a generic greedy method to find a minimum spanning tree. In Section 3.3, following the generic method, we introduce Kruskal's and Prim's algorithms, which are the two most popular algorithms to find minimum spanning trees. Both follow the greedy adopt that we introduce in the Section 3.1.

3.1   The minimum spanning tree problem

Consider a connected, edge-weighted undirected graph $G = (V, E)$ and let $c : E \to \mathbb{R}^+$ be a cost function over its edges. In the minimum spanning tree problem, the goal is to find an acyclic connected subgraph T (whose vertex-set is denoted by $V[T]$ and whose edge-set is denoted by $E[T]$) of G that spans all the vertices of G and whose total weight given by the following expression is minimized.

$$w(T) = \sum_{(u,v) \in E[T]} w(u, v)$$

More formally, a minimum-cost spanning tree T has the following properties:

- T is connected and acyclic,

- $V[T] = V$ and $E[T]$ is subset of E,

- $c(T) = \sum_{e \in E[T]} c(e) \leq c(T^*) = \sum_{e \in E[T^*]} c(e)$, for each spanning tree T^* of G.

In the next section, we are going to investigate a greedy approach to find a minimum spanning tree. In particular, a minimum spanning tree is greedily constructed in steps, such that one
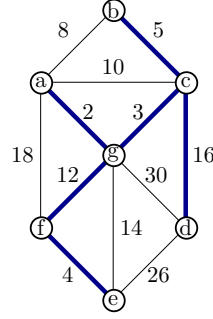
Figure 3.1: Example of a minimum spanning tree drawn in bold.

edge is added at each step, each step is locally optimal (according to some criterion that we greedily seek to optimize) and no step can be reverted. In other words, if we put an edge into the minimum spanning tree, that edge will be part of the minimum spanning tree when the algorithm will terminate, i.e., the locally-optimal choice becomes globally-optimal. Note that in general, a greedy strategy does not always result in a globally-optimal solution, but the case of the minimum spanning tree problem is exceptional. In particular, for the minimum spanning tree problem, there exist several greedy approaches to solve it.

3.2 The generic approach

As already stated, in this section we will investigate a generic greedy approach, which forms a unification of both the well-known greedy algorithms by Prim and by Kruskal's (for details, refer to Section 3.3). Consider a graph G with the properties that we have mentioned in Section 3.1. The generic approach computes a minimum spanning tree by coloring one edge of G at a time either blue or red. During its execution, it maintains the following invariant:

"there is some minimum spanning tree of G that contains all blue edges but no red".

It remains to consider the coloring rules that will allow us to maintain this invariant. To define the coloring rules and prove that the invariant holds for these rules, we need to introduce the notion of a cut in a graph.

Definition 3.1 A cut of $G = (V, E)$ is a partition of V into two sets X and $V \setminus X$ such that $X \neq \emptyset$ and $V \setminus X \neq \emptyset$. An edge (u, v) crosses the cut if $u \in X$ and $v \in V \setminus X$.

Example 3.1 Fig. 3.2 shows an example of a cut of a graph in which the cut separates the graph into two sets of vertices $X = \{a, f\}$ and $V \setminus X = \{b, c, d, e, g\}$. With this definition in mind, we are now ready to define the blue rule and the red rule.
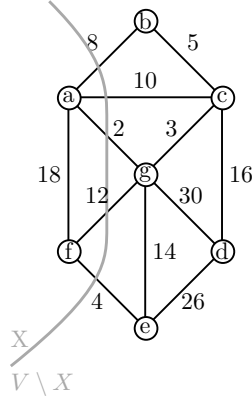
Figure 3.2: Example of a cut $(X, V \setminus X) = (\{a, f\}, \{b, c, d, e, g\})$.

The blue rule: Let $(X, V \setminus X)$ be a cut which is not crossed by a blue edge (if any). Color blue the min-cost uncolored edge crossed by $(X, V \setminus X)$.

The blue rule seeks in finding a cut such that there is no blue edge crossing it (if it exists), and among all the uncolored edges that cross that cut, the rule chooses to color blue the edge with the minimum cost. In other words, the blue subgraph is extending by one edge using the blue rule.

The red rule: Let C be a simple cycle without red edges (if any). Color red the max-cost uncolored edge of C.

Recall that a simple cycle is a cycle in which each vertex appears only once. The red rule seeks in finding such a cycle if it exists. Once such a cycle is found, the red subgraph is extended by the uncolored edge of maximum cost of this cycle, which is then colored red.

In the following, we proceed by proving two properties:

1. As along as there is at least one uncolored edge, at least one of the two rules can always be applied, i.e., the algorithm terminates after exactly m applications of the coloring rules.

2. When the algorithm will terminate, i.e., when all edges will have received a color, the subgraph of G induced by the blue edges is a minimum spanning tree of G.

Example 3.2   In Fig. 3.3, we demonstrate by an example the process of applying the coloring rules on a particular graph. Note that the rules can be applied in any order.

Lemma 3.1   As along as there is at least one uncolored edge, at least one of the two coloring rules can always be applied.
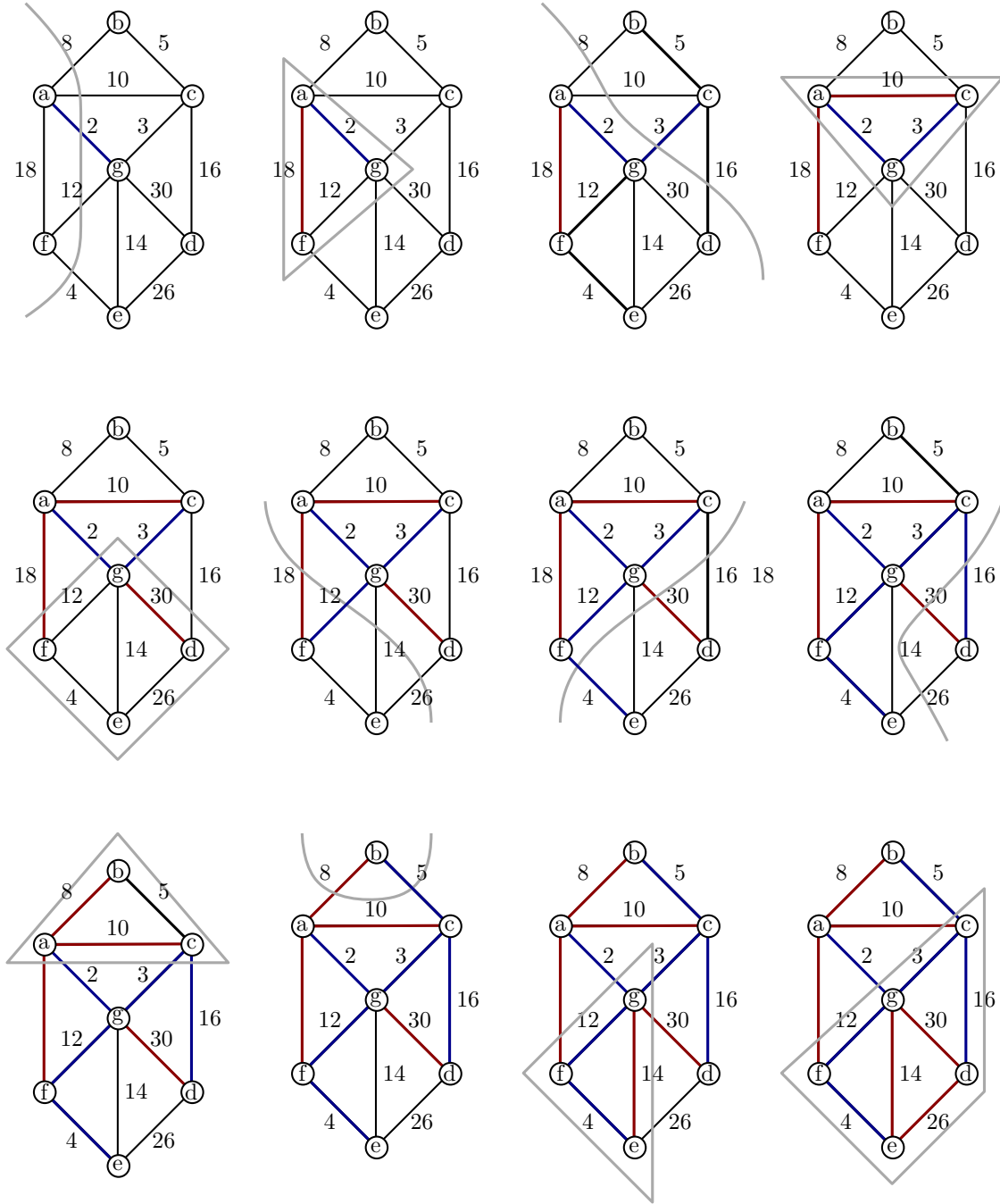
Figure 3.3: Using the coloring rules to find the minimum spanning tree.

Proof. Assume that a particular number of edges have been colored until a certain step in the execution of the algorithm and consider an uncolored edge e. We proceed by considering two cases:

– If this edge connects two blue subtrees (see Fig. 3.4a), then there is always a cut $(X, V − X)$ that does not contain any blue edge (e is among the edges that cross this cut). In this case, we can apply the blue rule and color blue the min-cost edge crossed by $(X, V − X)$.
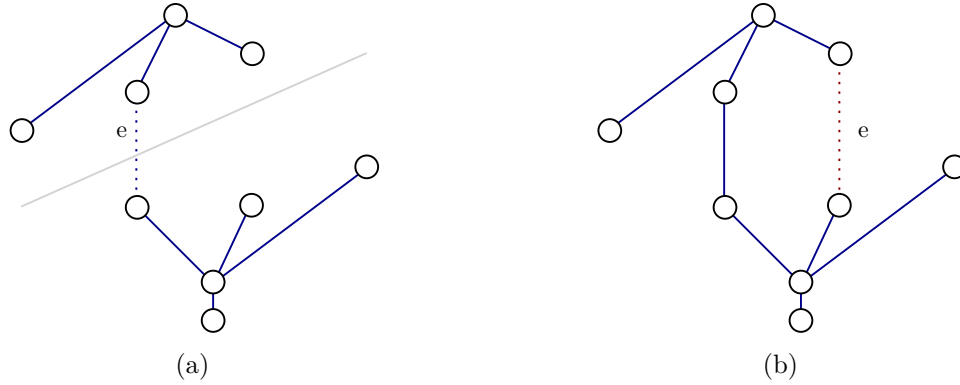
Figure 3.4: Applying (a) the blue rule when e connects two disconnected graphs, and (b) the red rule on edge e that forms a cycle.

– If we cannot find a cut as the one described above (i.e., e does not connect two blue subtrees), then edge e closes a cycle of blue edges (see Fig. 3.4b). Since e is the only uncolored edge in this cycle, then the red rule can be applied and thus e will be colored red.

In either case, we color the edge blue or red. In other words, we can always choose one of the two coloring rules and proceed with the algorithm until all edges are colored, which also implies that the algorithm always terminates.   □

Having proved that we can always proceed with one of the coloring rules, it remains to prove the correctness of the algorithm.

> **Theorem 3.1** Let $G = (V, E)$ be a connected, undirected graph together with a cost function $c : E \to \mathbb{R}^+$ over its edges. If we color the edges of G according to the two coloring rules, then the subgraph of G induced by the blue edges is a minimum spanning tree of G.

Proof. We prove the theorem by showing that the invariant that we defined earlier is maintained each time a coloring rule is applied: "there is an minimum spanning tree of G that contains all blue but no red edge". To do so, we use induction on the number of colored edges. In the base of the induction, no edge has been colored and the invariant trivially holds.

For the inductive step, we assume that we have colored a certain number of the edges of G by maintaining the invariant of the algorithm, that is, there is a minimum spanning tree of G, which we denote by T in the following, containing all edges that we have colored blue and none of the edges that we have colored red so far. For the inductive step, let (u, v) be an uncolored edge. By Lemma 3.1, we know that (u, v) will be colored either blue or ref. We will prove that after the coloring of the edge (u, v), the invariant still holds, that is, there is a minimum spanning tree, say T', of G that contains all edges that we have colored blue and none of the edges that we have colored red once the coloring of (u, v) has been completed.

DRAFT

Note that T and T' are not necessarily equal. We distinguish two cases depending on whether (u, v) is colored blue or red.

Assume first that edge (u, v) is colored blue, that is, the blue rule has been applied for the coloring of edge (u, v). Let $(X, V \setminus X)$ be the involved cut. If (u, v) belongs to T, then we set $T' = T$ and the invariant is trivially maintained. So, in this case we do not have to prove anything else. Consider now the more involved case in which the edge (u, v) does not belong to tree T. In this case, we need to define T' differently. Since T is a spanning tree of G, there exists a path, say P, from u to v in T (in particular, for any two vertices of a tree, there is a unique path connecting them). Since u and v belong to different partitions of the cut $(X, V \setminus X)$, there exists an edge e of P that has one endpoint in X and its other endpoint in $V \setminus X$. In other words, P has an edge e that crosses the cut $(X, V - X)$. We make the following observations regarding the color that the edge e has just before we color the edge (u, v) (using the blue rule):

- We first observe that edge e cannot be red, since e belongs to T, and T does not contain red edges.

- We further observe that e cannot be blue, since the cut $(X, V \setminus X)$ crosses no blue edge, i.e., among the edges that cross the cut $(X, V \setminus X)$, there is no blue edge, which includes also the case of edge e.

The two observations made above imply that, when (u, v) is colored blue, the edge e is uncolored. Since the cost of (u, v) is the smallest one among the edges that cross the cut $(X, V - X)$ and since the edge e is uncolored and crosses the cut $(X, V - X)$, the cost of the edge e is at least the cost of (u, v). Since T is a minimum spanning tree of G, the cost of (u, v) should be equal to the cost of edge e (as otherwise, we could exchange e and (u, v) in T and obtain a new spanning tree whose total cost is strictly smaller than the one of T, contradicting the fact that T is a minimum spanning tree of G). We proceed by setting T' to be the tree obtained from T by exchanging edge e with edge (u, v). Tree T' is a new minimum spanning tree of G with the same cost as T, which contains all edges that we have colored blue (including the edge (u, v)), and no edge that we have been colored red so far, as required by the invariant.

To complete the proof of correctness, it remains to consider the case in which the edge (u, v) is colored red, that is, the red rule is employed for its coloring. Let C be the involved simple cycle that contains no other red edges. Recall that in cycle C, the edge (u, v) has the maximum cost among all uncolored edges. If (u, v) does not belong to T, then we set $T' = T$ and the invariant is trivially maintained. Consider now the more involved case in which the edge (u, v) belongs to tree T. In this case, we need to define T' differently. Since T is a tree, there is an edge e in cycle C that does not belong to T (since T is acyclic). We make the following observations regarding the color that the edge e has just before we color the edge (u, v) (using the red rule):

- We first observe that edge e cannot be red, as e belongs to C, and C has no red edges.

- We further observe that edge e cannot be blue, as e does not belong to T.

From the two observation above, it follows that edge e is uncolored, when (u,v) is colored red. Also, by the choice of (u,v), the cost of e is less than or equal to the cost of (u,v). As T is a minimum spanning tree of G, we can conclude (similarly to the case above) that the cost of (u,v) is equal to the cost of the edge e. We proceed by setting T' to be the tree obtained from T by exchanging edge e with edge (u,v). Tree T' is a new minimum spanning tree of G with the same cost as T, which does not contain (u,v), as required by the invariant, while at the same time it contains all edges that we have colored blue so far and none of the edges that we have colored red. This completes the proof of the correctness of the algorithm. $\qquad\square$

3.3 Kruskal's algorithm and Prim's algorithm

In this section, we briefly recall the well-known algorithms by Prim and by Kruskal and we show that both can be considered as special cases of the generic algorithm that we demonstrated in the previous section.

3.3.1 Kruskal's algorithm

The algorithm by Kruskal is computing a minimum spanning tree by maintaining acyclicity in the edges that are chosen to be added to the minimum spanning tree at each step, i.e., an edge is only added to the computed minimum spanning tree if it does not form a cycle with previously added edges. Simultaneously, it has the minimum cost among all available edges that we could choose to add; see Algorithm 3.1.

Algorithm 3.1: Kruskal's greedy algorithm.

Input : A connected, edge-weighted undirected graph $G = (V, E)$ and a cost function
$\quad\quad c : E \to \mathbb{R}^+$ over its edges.
Output : A minimum spanning tree T of G.

1 Sort the edges by increasing cost;
2 $T \longleftarrow \varnothing$;
3 for each edge e in increasing cost order do
4 $\quad$ if adding e to T does not form a cycle then
5 $\quad\quad$ add e to T;
6 $\quad$ end
7 end

Correctness: The correctness of the algorithm by Kruskal follows from the following two observations for the edge e that is considered at a certain step of the algorithm:

- If edge e closes a cycle, then by the sorting, e has maximum cost, which implies that by not adding it to T, we apply the red rule.

- Otherwise, e connects two connected components, and by the sorting e has the minimum cost in the corresponding cut. In other words, we apply the blue rule.

Complexity analysis: Consider a data structure that maintains several sets and supports the following two operations: (i) union(A,B): unions two disjoint sets A and B (ii) find(x): retains the name of the set containing x.

With the find(x) operation, we can determine whether two vertices u and v belong to the same tree by testing whether find(u) equals find(v). The union is used to combine two subtrees if indeed find(u) equals find(v). Both of these functions can be implemented in $\mathcal{O}(\alpha)$ where α is the Ackermann function which is almost a constant. With this data structure in mind, we can rewrite the algorithm by Kruskal as in Algorithm 3.2. Regarding the time complexity, we first note that the sorting of the edges by increasing cost can be done in $\mathcal{O}(m \log m)$ time, while the rest of the algorithm needs $\mathcal{O}(m \cdot \alpha)$ time (by the discussion above), where m denotes the number of edges of G.

Algorithm 3.2: Kruskal's greedy algorithm: revised.

Input   : A connected, edge-weighted undirected graph $G = (V, E)$ and a cost function
$c : E \to \mathbb{R}^+$ over its edges.
Output : A minimum spanning tree T of G.

1 Sort the edges by increasing cost;
2 $T \longleftarrow \varnothing$;
3 for each edge (u, v) in increasing cost order do
4 | if $find(u) \neq find(v)$ then
5 | | add (u, v) to T;
6 | | $Union(find(u), find(v))$;
7 | end
8 end

3.3.2   Prim's algorithm

The algorithm by Prim also iterates through the edges of graph G and at each iteration it adds an edge to the minimum spanning tree when it is directly connected to the ones computed so far and has the minimum cost among all edges with this property. In other words, the algorithm by Prim maintains connectivity; see Algorithm 3.3.

Algorithm 3.3: Prim's greedy algorithm

Input   : A connected, edge-weighted undirected graph $G = (V, E)$ and a cost function
$c : E \to \mathbb{R}^+$ over its edges.
Output : A minimum spanning tree T of G.

1 Initialize T to a vertex of G;
2 while T is not spanning do
3 | Add to T the cheapest uncolored edge e incident to T;
4 | color e blue;
5 end

Correctness: The correctness of the algorithm by Prim follows from the fact that the algorithm repeatedly applies the blue rule.

Algorithm 3.4: Prim's greedy algorithm: Revised

---

1   $D \longleftarrow \varnothing$;

2   for each vertex v do

3      $key(v) = +\infty$;   // This for loop costs $\mathcal{O}(n \log n)$

4      $\pi(v) = null$;

5      $D.insert(v, key(v))$;

6   end

7   $s \leftarrow$ a vertex of G ;

8   $D.decrease\_key(s, 0)$;

9   while $!D.isEmpty()$ do

10     $u = D.delete\_min()$;   // All delete_min calls cost $\mathcal{O}(n \log n)$

11   foreach neighbor v of u do

12   if $(key(v) > c(u, v))$ then

13          $key(v) = c(u, v)$;   // This for loop costs $\mathcal{O}(m \log n)$

14          $\pi(v) = u$;

15          $D.decrease\_key(v, c(u, v))$;

16   end

17   end

18 end

Complexity analysis: To estimate the time complexity of the algorithm, we maintain a data structure of elements, each of which is associated with a key, that supports the following operations: (i) insert(x, key): inserts element x with a certain key into the data structure, (ii) delete_min(): deletes the element with the minimum key from the data structure, and (iii) decrease_key(x,k): decreases the key of element x to k. Each of the aforementioned operations can be implemented in $\mathcal{O}(\log n)$ assuming that the data structure is implemented as a binary heap.

With this data structure in mind, we can rewrite the algorithm by Prim as in Algorithm 3.4. The idea is to maintain in the data structure (denoted by D in the following) the vertices that have not been spanned yet (i.e., the vertices that have been spanned are those in $V - D$). Initially, all vertices are added to the data structure, such that their keys are $+\infty$, except for one whose key is 0. This can be clearly done in $\mathcal{O}(n \log n)$ time. At each iteration, the element (i.e., vertex) with the minimum key in D is added to the minimum spanning tree. The idea here is that the key of a vertex in D corresponds to the weight of its lightest edge towards a vertex that has been spanned (i.e., belonging to $V - D$). That said, once a vertex v has been added to the minimum spanning tree, we must remove it from D. More importantly, however, we need to update the keys of the vertices in D that are incident to v so as to maintain the aforementioned property for the keys of the vertices in D. Since the total number of such update operations is $\mathcal{O}(m)$, the time complexity of the algorithm by Prim is $\mathcal{O}(m \log n)$, as this step also dominates the total cost for the insertion of the vertices in D.

Exercises

Exercise 3.1 Prove or disprove the following statements.

(a) Let e be a minimum-cost edge in a graph G, i.e., $\forall e_i \in E : c(e_i) \geq c(e)$. Then, e belongs to some minimum spanning-tree.

(b) There is a weighted graph with exactly two minimum spanning trees.

(c) Let G be a weighted graph with distinct weights. Then, G has a unique MST.

Exercise 3.2 Let T be a minimum spanning tree of a graph G and let $L(T)$ be the sorted list of the edge weights of T. Show that for any two minimum spanning trees S and T of G, it holds that $L(S) = L(T)$.

Exercise 3.3 Let T be a minimum spanning tree of a graph $G = (V, E)$ and let V' be a subset of V. Let T' be the subgraph of T induced by V' and let G' be the subgraph of G induced by V'. Show that if T' is connected, then T' is a minimum spanning tree of G'.

Exercise 3.4 The first algorithm for finding a minimum spanning tree of a connected, weighted and undirected graph was due to Boruvka and dates back to 1926. The algorithm is greedy and proceeds in steps as follows.

Algorithm 3.5: Boruvka's algorithm to compute a minimum spanning tree.

    Input   : A connected, weighted and undirected graph G.
    Output: A minimum spanning tree $\mathcal{T}$ of G.

1 Initialize all vertices of G as individual components (or sets);
2 $\mathcal{T} \leftarrow \emptyset$;
3 **while** there are more than one components **do**
4   **foreach** component C **do**
5         $e \leftarrow$ a minimum-weight edge that connects C to any other component C';
6         $\mathcal{T} \leftarrow \mathcal{T} \cup \{e\}$; // e might be already in $\mathcal{T}$
7   **end**
8   Merge all pairs of components C and C' that were linked by an edge;
9 **end**
10 **return** $\mathcal{T}$;

(a) Apply Boruvka's algorithm to the following graph to compute a minimum spanning tree.
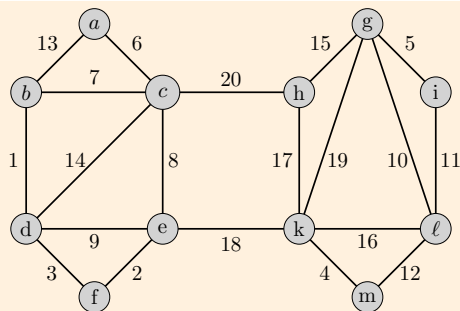
(b) Use coloring rules of Section 3.2 to prove the correctness of Boruvka's algorithm.

Exercise 3.5 Compute with the blue-red rules a minimum spanning tree for the following graph. Give all intermediate steps.

Further reading

– Chapter 23 from: T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009.

– http://www.cs.rpi.edu/~goldberg/15-GT/05-minspan.pdf

– https://courses.cs.washington.edu/courses/cse326/04wi/lectures/mst.pdf

4. Minimum Cuts

This chapter will focus on the minimum cut problem, which given an edge-weighted graph asks for a partition of its vertices into two non-empty sets S and T, such that the sum of the weights of the edges with one endpoint in S and one in T is minimum. The rest of this chapter is organized as follows: In Section 4.1 we introduce the formal definition of the problem and overview some applications of it. In Section 4.3, we present an algorithm to compute a minimum cut of a graph.

4.1   The minimum cut problem

Let $G = (V,E)$ be an undirected graph and let $c : E \to \mathbb{R}^+$ be a cost function over the edges of graph G. A cut (S,T) of graph G is a partition of its vertex-set V in two non-empty sets S and T, namely:

$$S \subset V, \quad T \subset V$$
$$S \cup T = V, \quad S \cap T = \emptyset$$
$$S \neq \emptyset, \quad T \neq \emptyset$$

We denote such a cut by (S,T), which can equivalently be written as $(S, V \setminus S)$, since $T = V \setminus S$. We say that an edge (u,v) of G crosses the cut (S,T) if either $u \in S$ and $v \in T$ or $v \in S$ and $u \in T$. Clearly, the removal of all edges that cross the cut (S,T) from G results in disconnecting graph G. The cost $c(S,T)$ of the cut (S,T) is the sum of the costs of the edges that cross (S,T), namely:

$$c(S,T) = \sum_{\substack{(u,v)\in E \\ u\in S, v\in T}} c(u,v)$$

If each edge has a unit cost, that is, $c(e) = 1, \forall e \in E$, then the cost of cut (S,T) is actually equal to the number of edges that cross (S,T). For an illustration refer to Fig. 4.1, where we list all cuts of a particular graph and their corresponding costs. Note that in the illustration we do list cuts in which the cardinality of S is 3, since these are in one to one correspondence with those in which the cardinality of S is 1 (that are listed).

DRAFT

S	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{a,b\}$	$\{a,c\}$	$\{a,d\}$	$\{b,c\}$	$\{b,d\}$	$\{c,d\}$
$c(S, V \setminus S)$	7	9	8	5	8	14	8	8	14	8

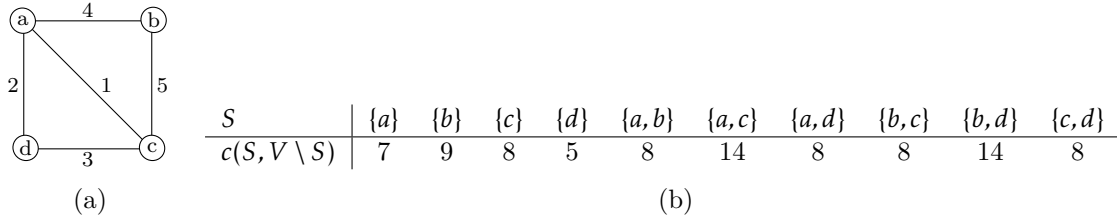(a)   (b)

Figure 4.1: Illustrating all $(S, V \setminus S)$ cuts of a graph and their costs. The minimum cut of the graph is $(\{d\}, \{a, b, c\})$. On the other hand, the minimum (a, b)-cut is $(\{a\}, \{b, c, d\})$.

> **Definition 4.1** (Minimum-cut problem) Given an n-vertex undirected graph $G = (V, E)$ and a cost function $c : E \to \mathbb{R}^+$ over its edges, the minimum cut problem asks for a minimum cut, that is, a cut (S, T) of graph G which has the minimum cost over all cuts of G, namely:
> $$c(S, T) \le c(S', T'), \ \forall (S', T') \text{ cut of } G$$

Note that a naive algorithm to find the minimum cut of an n-vertex graph computes all possible cuts $(S, V \setminus S)$ with $S \ne \emptyset$ and reports the one of minimum cost. Since the cuts of graph are in one to one correspondence with the powerset of its vertex-set (if we ignore its empty set and set the vertex-set itself), and since the cardinality of the powerset of the vertex-set is 2^n, it follows that the time complexity of the naive algorithm is exponential in n.

The minimum (s, t)-cut problem

A problem that is closely related to the minimum cut problem is the so-called minimum (s, t)-cut problem. The input of this problem specifies additionally two designated vertices s and t. The goal is to compute a so-called (s, t)-cut of minimum cost, that is, one in which s and t must belong to different parts in the partition. Formally, the problem is defined as follows:

> **Definition 4.2** (Minimum (s,t)-cut problem) Given an n-vertex undirected graph $G = (V, E)$, a cost function $c : E \to \mathbb{R}^+$ over its edges and two distinct vertices s and t of G, the minimum (s, t)-cut problem asks for the cut (S, T) of graph G which has the minimum cost $c_{st}(S, T)$ over all cuts of G under the additional requirement that $s \in S$ and $t \in T$, that is,
> $$c_{st}(S, T) \le c_{st}(S', T'), \ \forall (S', T') \text{ cut of } G \text{ with } s \in S' \text{ and } t \in T'$$

Since the minimum (s, t)-cut of G is a cut of G, the following observation directly follows from Definitions 4.1 and 4.2 (e.g., in the example illustrated in Fig. 4.1 the cost of the minimum (a, b)-cut is 7, while the corresponding cost of the minimum cut is 5).

> Observation 4.1   The cost $c(S,T)$ of a minimum cut (S,T) of a graph G is a lower bound for the cost $c_{st}(S',T')$ of a minimum (s,t)-cut (S',T') of G, namely,
>
> $$c(S,T) \le c_{st}(S',T').$$

4.2   Relating the minimum (s,t)-cut and the minimum cut problems

In this section, we discuss the connection between the minimum (s,t)-cut problem and the minimum cut problem. To see this connection, assume that an algorithm $\mathcal{A}$ to compute a minimum (s,t)-cut of a graph G is known. Using algorithm $\mathcal{A}$ as a basis, one can compute a minimum cut of the input graph G, using the approach described in Algorithm 4.1. Since a minimum cut (S,T) of graph G must contain at least one vertex in S and at least one vertex in T, it follows that Algorithm 4.1 is correct, as it considers all such pairs of vertices. Algorithm 4.1 further proves that one can reduce the minimum cut problem to the problem of finding a minimum (s,t)-cut in $O(n^2)$ time, or in other words, the complexity of Algorithm 4.1 is $O(n^2)$ times the complexity of algorithm $\mathcal{A}$.

Algorithm 4.1: Computing a minimum cut of graph G assuming the existence of an algorithm $\mathcal{A}$ to compute a minimum (s,t)-cut.

---

1 Function minimumCut(G)
2   minimum $\leftarrow +\infty$;
3      $(S_{min}, T_{min}) \leftarrow (\emptyset, \emptyset)$;
4   for $s \in V$ do
5   for $t \in V$, $t \ne s$ do
6              $(S,T) \leftarrow$ minimumCut(G,s,t) ;   // Compute a minimum (s,t)-cut using $\mathcal{A}$
7   if $(c(S,T) <$ minimum) then
8   minimum $\leftarrow c(S,T)$;
9                  $(S_{min}, T_{min}) \leftarrow (S,T)$;
10   end
11   end
12   end
13   return (S_{min}, T_{min});
14 end

To improve the quadratic factor in the aforementioned reduction, we introduce the operation of merging two vertices of a graph G; for an example, refer to Fig. 4.2. Formally, let u and v be distinct two vertices of G and recall that graph G is equipped with a cost function $c : E \to \mathbb{R}^+$ over its edges. The operation of merging u and v results in a new graph

$$G|\{u,v\} = (V', E'),$$

which is also equipped with a cost function $c' : E' \to \mathbb{R}^+$. This graph is derived from G as follows:
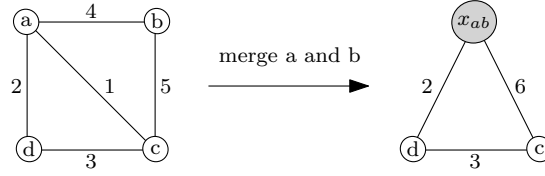
Figure 4.2: Merging vertices a and b of the graph of Fig. 4.1

- graph $G|\{u,v\}$ contains a designated vertex x_{uv} and all vertices of G except for vertices u and v, that is,

$$V' = V \cup \{x_{uv}\} \setminus \{u,v\}$$

- graph $G|\{u,v\}$ contains all edges of G except those having as an endpoint either u or v or both,

- for each edge (u,w) of G with $w \neq v$, graph $G|\{u,v\}$ contains an edge (x_{uv},w),

- for each edge (v,w) of G with $w \neq u$, graph $G|\{u,v\}$ contains an edge (x_{uv},w),

- the cost of each edge (w,z) of $G|\{u,v\}$, which is not incident to x_{uv}, equals $c(w,z)$, that is,

$$c'(w,z) = c(w,z), \quad \text{if } x_{u,v} \notin \{w,z\}$$

- the cost of each edge (x_{uv},w) of $G|\{u,v\}$ incident to x_{uv} is given by the following relation:

$$c'(x_{uv},w) = \begin{cases} c(u,w) & \text{if } (u,w) \in E, (v,w) \notin E, \\ c(v,w) & \text{if } (v,w) \in E, (v,w) \notin E, \\ c(u,w) + c(v,w) & \text{if} (u,w) \in E, (v,w) \in E. \end{cases}$$

> **Observation 4.2** The graph $G|\{u,v\}$ obtained from graph G by merging vertices u and v has one vertex less than graph G.

With the definition of the operation of merging two vertices in mind, we are now ready to give the following theorem.

> **Theorem 4.1** Let $G = (V,E)$ an undirected graph and a cost function $c : E \to \mathbb{R}^+$ over its edges. If G contains at least two vertices s and t, then the lighter of the following two cuts is a minimum cut of G:
>
> - (S_1, T_1) is a minimum (s,t)-cut of G.
>
> - (S_2, T_2) is a minimum cut of $G|\{s,t\}$.

Proof. Let (S,T) be a minimum cut of G. Then, either s and t both belong to S or both belong to T or one belongs to S and the other belongs to T. In the first two cases, the cost of (S,T) is equal to the cost of (S_2, T_2), while in the latter case the cost of (S,T) is equal to the cost of (S_1, T_1). □

Assuming the existence of algorithm $\mathcal{A}$ for computing a minimum (s,t)-cut of a graph G, Theorem 4.1 gives rise to a more efficient algorithm for computing a minimum cut of graph G than the one given in Algorithm 4.1. This algorithm is recursive. At the base of the recursion, the graph has only two vertices. Thus, it has only one cut, which is the one that the algorithm reports. At each recursive step, the algorithm selects two vertices s and t and proceeds to compute a minimum (s,t)-cut (using algorithm $\mathcal{A}$) and a minimum cut of the graph $G|\{s,t\}$ obtained from graph G by merging vertices s and t (using recursion). As Theorem 4.1 suggests, the algorithm reports the minimum of the two computed cuts; refer to Algorithm 4.2 for a summary. Therefore, the correctness of the algorithm is trivially implied.

Algorithm 4.2: Improved computation of a minimum cut of graph G assuming the existence of an algorithm $\mathcal{A}$ to compute a minimum (s,t)-cut.

---

1 Function minimumCut(G)
2   if ($|V(G)| = 2$, i.e., $V(G) = \{v_1, v_2\}$) then
3   return ($\{v_1\}, \{v_2\}$)
4   end
5   else
6   s,t ← any two vertices of G
7   (S_1, T_1) ← minimumCut(G,s,t) // compute a minimum (s,t)-cut using $\mathcal{A}$
8   (S_2, T_2) ← minimumCut($G|\{s,t\}$) // recursive call
9   if $c(S_1, T_1) < c(S_2, T_2)$ then return (S_1, T_1);
10   else return (S_2, T_2);
11   end
12 end

By Observation 4.2, we can further imply that if graph G has n vertices, then the total number of recursive calls performed by Algorithm 4.2 (until it reaches the base case) is $n-2$, since each merge operation decreases the number of vertices of the graph by 1. It follows that one can reduce the minimum cut problem to the problem of finding a minimum (s,t)-cut in $O(n)$ time, or in other words, the complexity of Algorithm 4.2 is $O(n)$ times the complexity of algorithm $\mathcal{A}$.

4.3   The algorithm by Stoer and Wagner

The algorithm by Stoer and Wagner is based on the recursive procedure of Algorithm 4.2 with a crucial difference; the vertices s and t to be merged are not chosen arbitrarily (as in Algorithm 4.2), but they are carefully chosen. In particular, the algorithm constructs a certain ordering $v_1, v_2, \ldots, v_n$ of the vertices of graph G such that the cut

$$(\{v_1, v_2, \ldots, v_{n-1}\}, \{v_n\})$$

is a minimum (v_{n-1}, v_n)-cut of G. Having guaranteed this property, the vertices to be merged are v_{n-1} and v_n. In the following, we first describe how this ordering is computed and then we prove that it has the desired property.

Algorithm 4.3: The core of the algorithm by Stoer and Wagner

  Input : An undirected graph $G = (V, E)$ and a function $c : E \to \mathbb{R}^+$.

  Output : An ordering $v_1, v_2, \ldots, v_n$ of the vertices of G such that $(\{v_1, v_2, \ldots, v_{n-1}\}, \{v_n\})$

       is a minimum (v_{n-1}, v_n)-cut of G.

 1 Function minimumCutCore(G)

 2    $v_1 \leftarrow$ a random vertex G

 3    $L \leftarrow \{v_1\}$

 4   for $i = 2, \ldots, n$ do

 5      $max \leftarrow -\infty$

 6   foreach v in $V \setminus L$ do

 7   if $max < c(\{v\}, L)$ then

 8         $max \leftarrow c(\{v\}, L)$

 9         $v_i \leftarrow v$

10   end

11   end

12      $L \leftarrow L \oplus \{v_i\}$

13   end

14   return $(\{v_1, v_2, \ldots, v_{n-1}\}, \{v_n\})$

15 end

The ordering $v_1, v_2, \ldots, v_n$ is computed as follows. Assume that we have computed an ordering $L : v_1, v_2, \ldots, v_i$ for $1 \leq i < n$ vertices of G. The next vertex v_{i+1} is chosen from $V \setminus L$, such that the total cost of the edges incident to it and towards the vertices of L is maximum; refer to Algorithm 4.3. To do so, we consider each vertex v in $V \setminus L$ and we compute the cost $c(\{v\}, L)$ of the cut $(\{v\}, L)$ which corresponds to the total cost of the edges that have v as one endpoint and a vertex in L as their other endpoint. Among these vertices, we then choose as v_{i+1} the one which maximizes the aforementioned cost. In the next lemma, we prove that the ordering $v_1, v_2, \ldots, v_n$ computed by the algorithm by Stoer and Wagner has the property that the cut $(\{v_1, v_2, \ldots, v_{n-1}\}, \{v_n\})$ is a minimum (v_{n-1}, v_n)-cut of G.

> **Lemma 4.1** Let $G = (V, E)$ an undirected graph and a cost function $c : E \to \mathbb{R}^+$ over its edges. Let also $v_1, v_2, \ldots, v_n$ be an ordering of the vertices of G computed by Algorithm 4.3. Then, $(\{v_1, v_2, \ldots, v_{n-1}\}, \{v_n\})$ is a minimum (v_{n-1}, v_n)-cut of G.

Proof. To prove the lemma, it suffices to prove that for the ordering $v_1, v_2, \ldots, v_n$ computed by Algorithm 4.3 and for any (v_{n-1}, v_n)-cut (S, T) of G, it holds that:

$$c(\{v_1, v_2, \ldots, v_{n-1}\}, \{v_n\}) \leq c(S, T) \tag{4.1}$$

We consider such a (v_{n-1}, v_n)-cut (S, T) and let a vertex v_i with $1 \leq i \leq n$ be critical if and only if either $v_i \in S$ and $v_{i-1} \in T$, or $v_i \in T$ and $v_{i-1} \in S$ (see Fig. 4.3 for an illustration). Since (S, T) is a (v_{n-1}, v_n)-cut of G, vertex v_n is by definition critical. Since v_n is critical, since $\{v_1, v_2, \ldots, v_n\} \cap S = S$ and since $\{v_1, v_2, \ldots, v_n\} \cap T = T$, to prove that Eq. (4.1) holds, it suffices
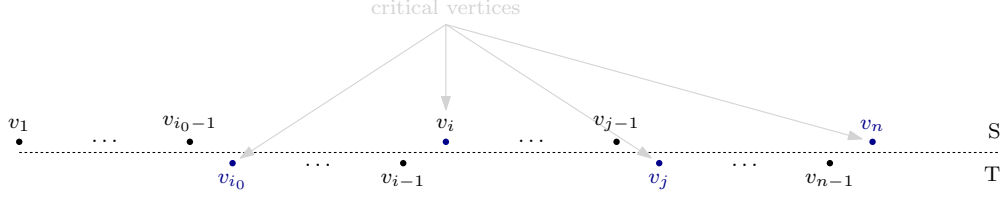
Figure 4.3: Illustration for the proof of Lemma 4.1; critical vertices are illustrated in blue.

to prove that for every critical vertex v_i the following inequality holds:

$$c(\{v_1, v_2, \ldots, v_{i-1}\}, \{v_i\}) \leq c(\{v_1, v_2, \ldots, v_i\} \cap S, \{v_1, v_2, \ldots, v_i\} \cap T) \tag{4.2}$$

We prove this property by induction on the number of critical vertices. Clearly, for the first critical vertex v_{i_0} Eq. (4.2) holds as equality, because either:

$$\{v_1, v_2, \ldots, v_{i_0}\} \cap S = \{v_1, v_2, \ldots, v_{i_0-1}\}, \quad \{v_1, v_2, \ldots, v_{i_0}\} \cap T = \{v_{i_0}\},$$

or

$$\{v_1, v_2, \ldots, v_{i_0}\} \cap S = \{v_{i_0}\}, \quad \{v_1, v_2, \ldots, v_{i_0}\} \cap T = \{v_1, v_2, \ldots, v_{i_0-1}\}$$

For the inductive hypothesis, assume that claimed property holds for some critical vertex v_i:

$$c(\{v_1, v_2, \ldots, v_{i-1}\}, \{v_i\}) \leq c(\{v_1, v_2, \ldots, v_i\} \cap S, \{v_1, v_2, \ldots, v_i\} \cap T) \tag{4.3}$$

In the inductive step we prove that Eq. (4.2) holds for the next critical vertex v_j. Since $j > i$, the following equality trivially holds.

$$c(\{v_1, \ldots v_{j-1}\}, \{v_j\}) = c(\{v_1, \ldots, v_{i-1}\}, \{v_j\}) + c(\{v_i, v_2, \ldots, v_{j-1}\}, \{v_j\}) \tag{4.4}$$

Since v_j appears after v_i in the ordering computed by the algorithm by Stoer and Wagner:

$$c(\{v_1, \ldots, v_{i-1}\}, \{v_j\}) \leq c(\{v_1, \ldots v_{i-1}\}, \{v_i\}) \tag{4.5}$$

We are now ready to put everything together:

$$
\begin{aligned}
c(\{v_1, \ldots v_{j-1}\}, \{v_j\}) &\overset{4.4}{=} c(\{v_1, \ldots, v_{i-1}\}, \{v_j\}) + c(\{v_i, \ldots, v_{j-1}\}, \{v_j\}) \\
&\overset{4.5}{\leq} c(\{v_1, \ldots, v_{i-1}\}, \{v_i\}) + c(\{v_i, \ldots, v_{j-1}\}, \{v_j\}) \\
&\overset{4.3}{\leq} c(\{v_1, \ldots, v_i\} \cap S, \{v_1, \ldots, v_i\} \cap T) + c(\{v_i, \ldots, v_{j-1}\}, \{v_j\}) \\
&\leq c(\{v_1, \ldots, v_j\} \cap S, \{v_1, \ldots, v_j\} \cap T)
\end{aligned}
$$

The last inequality follows from the fact that v_j is the last critical vertex. Therefore, it may have connections to vertices in $\{v_1, v_2, \ldots, v_{i-1}\}$, whose costs are not captured in the cost $c(\{v_i, \ldots, v_{j-1}\}, \{v_j\})$. □

DRAFT

Algorithm 4.4: The algorithm by Stoer and Wagner

---

1 Function minimumCut(G)
2   if ($|V(G)| = 2$, i.e., $V(G)) = \{v_1, v_2\}$ then
3 | return ($\{v_1\}, \{v_2\}$)
4   end
5   else
6   ($\{v_1, \ldots, v_{n-1}\}, \{v_n\}$) $\leftarrow$ minimumCutCore(G)
7   (S, T) $\leftarrow$ minimumCut($G|\{v_{n-1}, v_n\}$)
8   if $c(S, T) < c(\{v_1, \ldots, v_{n-1}\}, \{v_n\})$ then
9 | return (S, T)
10   end
11   else
12 | return ($\{v_1, \ldots, v_{n-1}\}, \{v_n\}$)
13   end
14   end
15 end

By Lemma 4.1, it follows that indeed Algorithm 4.3 returns a minimum (v_{n-1}, v_n)-cut of G. To compute the minimum cut of graph G, it suffices to appropriate adjust Algorithm 4.2 to use vertices v_{n-1} and v_n in the recursive call. The adjustment is presented in Algorithm 4.4, while a sample execution of the algorithm on the example of Fig. 4.1 is given in Fig. 4.4.
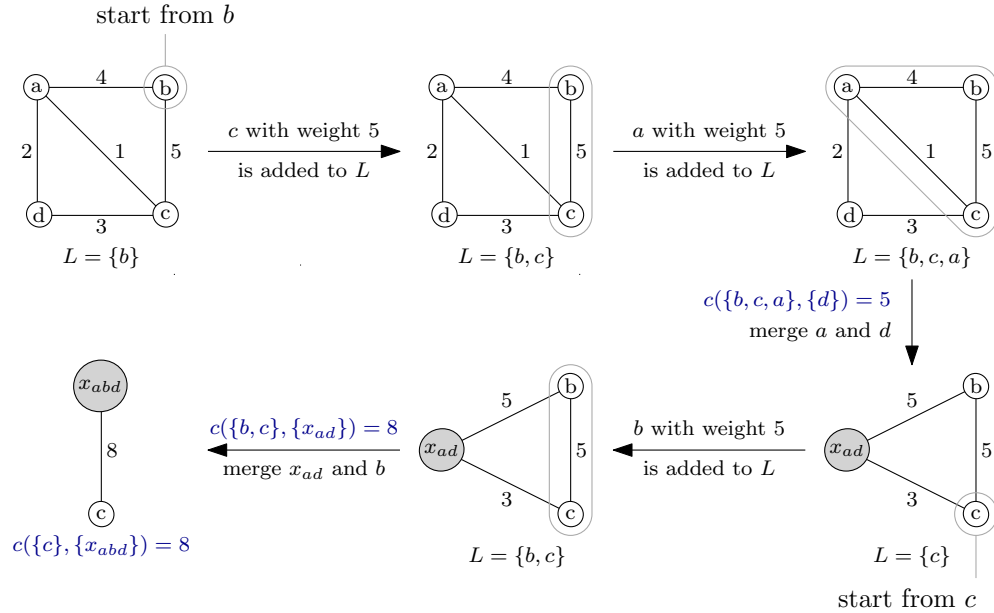

Figure 4.4: An example execution of the algorithm by Stoer and Wagner. In the example, three cuts indicated in blue have been recursively computed. The cut with $S = \{b, c, a\}$ and $T = \{d\}$ is the minimum one with $c(S, T) = 5$.
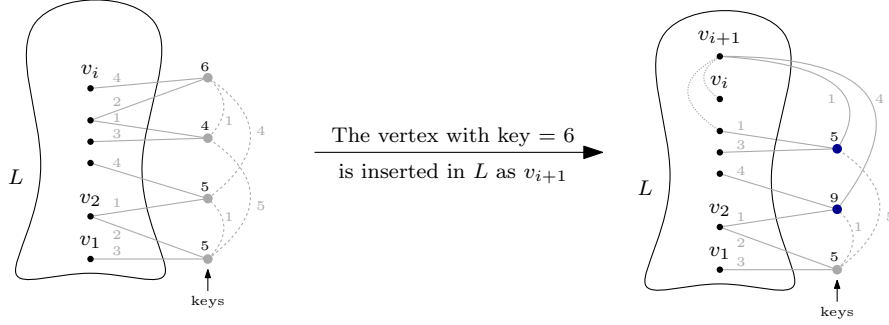
Figure 4.5: Illustration of the process of inserting the next vertex v_{i+1} into the ordering $L = \{v_1, \ldots, v_i\}$ according to the algorithm by Stoer and Wagner.

Time complexity analysis

We conclude this section by analyzing the time complexity of the algorithm by Stoer and Wagner. To this end, we employ a data structure of elements, each of which is associated with a key, supporting the following operations:

– insert(e,k): inserts element e with key k into the data structure.

– delete_max(): deletes and returns the element with maximum key from the data structure.

– increase_key(e,k): increases the key of element e to k.

Assuming that the data structure is implemented as a Fibonacci heap, the first two operations can be implemented in $\mathcal{O}(\log n)$ time, while the last one in $\mathcal{O}(1)$ time (see Section 2.3.1).

Since at each step of Algorithm 4.3, the vertex in $V \setminus L$ whose cumulative cost to L is maximum has to be computed, it is natural to associate each vertex u in $V \setminus L$ with a key $key(u)$, such that (see the left part of Fig. 4.5 for an illustration):

$$key(u) = \sum_{\substack{(u,v) \in E \\ v \in L}} c(u,v)$$

By doing so, the vertex with the largest key in the data structure is the next in the ordering L. Therefore, it can be reported in $\mathcal{O}(\log n)$ time. At the beginning of Algorithm 4.3, each vertex of the graph is inserted into the data structure with key $-\infty$. This costs in total $\mathcal{O}(n \log n)$ time. Since each vertex of the graph will be extracted from the data structure exactly once (by means of the delete_max operation), it follows that the total time needed to perform all extractions is also $\mathcal{O}(n \log n)$. However, once a vertex is extracted from the data structure (to enter the ordering L), the keys of several vertices in $V \setminus L$ will have to be necessarily increased, since they may be adjacent to the vertex entering L (for an illustration, refer to be blue colored vertices in the right part of Fig. 4.5, whose keys were increased due to the insertion of v_{i+1} into L). Since increasing the key of a vertex (by means of the increase_key operation) needs constant time, it follows that the total time needed to perform all key-increments is $\mathcal{O}(m)$,

because the number of times a key-increment operation is performed is at most the number m of edges of the graph. So, in total the time complexity of Algorithm 4.3 is $\mathcal{O}(n \log n + m)$.

Since the time complexity of the algorithm by Stoer and Wagner (see Algorithm 4.4) is $\mathcal{O}(n)$ times the time complexity of Algorithm 4.3, it follows that the overall time complexity of the algorithm by Stoer and Wagner is $\mathcal{O}(n^2 \log n + nm)$ as summarized in the following theorem.

> **Theorem 4.2** (Stoer and Wagner, 1997) Let $G = (V, E)$ an undirected graph with n vertices and m edges and let $c : E \rightarrow \mathbb{R}^+$ be a cost function over the edges of G. Then, the time needed to compute a minimum cut of G is $\mathcal{O}(n^2 \log n + nm)$.

Exercises

Exercise 4.1 Apply the algorithm by Stoer & Wagner to compute a minimum cut for the following graph; use vertex a as starting vertex.
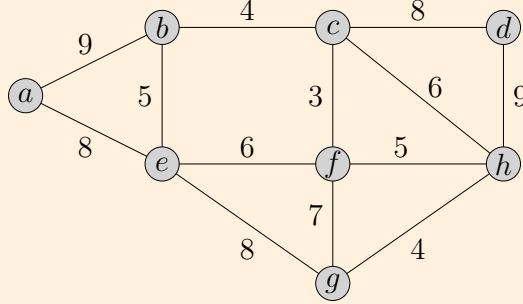
Exercise 4.2 Let $G = (V, E, c)$ be a graph with a cost function $c : E \rightarrow \mathbb{N}^+$. Let also $G' = (V, E, c')$ be a graph on the same set of vertices and edges as graph G whose cost function is defined as follows:

$$c'(u, v) := |E| \cdot c(u, v) + 1, \forall (u, v) \in E$$

Recall that given a cut $S \subset V$ of graph G, its total cost is defined as follows:

$$c(S) = \sum_{\substack{(u,v) \in E: \\ u \in S, v \notin S}} c(u, v).$$

(a) Let S_1 and S_2 be two cuts of G with $c(S_1) < c(S_2)$. Prove that the following holds in G': $c'(S_1) < c'(S_2)$.

(b) Let S_1 and S_2 be two cuts of G with $c(S_1) = c(S_2)$. Prove the following. If S_1 has fewer edges than S_2, then the following holds in G': $c'(S_1) < c'(S_2)$.

(c) In general, a graph may have more than one minimum cuts. Using (a) and (b), derive an algorithm that finds a minimum cut with minimum number of edges in G. Briefly explain why and how your algorithm works.

Exercise 4.3 Besides the algorithm by Stoer and Wagner, there exists a really simple alternative algorithm that at least sometimes returns a correct minimum cut, assuming that the input graph G is unweighted:

For each vertex v of graph G, denote by $S(v)$ the vertices that are contracted into v. Initially $S(v) = \{v\}$ for each vertex v of G. If G has only two vertices v_1 and v_2, then the algorithm returns the cut $(S(v_1), S(v_2))$. Otherwise, the algorithm chooses an edge $e = (u, v)$ at random and let G' be the graph resulting from the contraction of e with a new node z_{uv} replacing u and v. The algorithm then sets $S(z_{uv})$ to $S(u) \cup S(v)$ and is recursively applied to G'.

Suppose that G has a minimum cut (A, B) of size k and let F be the set of edges crossed by this cut (since G is unweighted, the size of the cut is equivalent to the number of edges having one end in A and the other in B). We want to show that the contraction algorithm returns this cut with probability at least

$$\frac{1}{\binom{n}{2}}$$

To achieve this, prove the following intermediate properties:

(a) Graph G does not contain a vertex whose degree is smaller than k.

(b) Using (a), give a lower bound on the number of edges of G.

(c) Show that in the first iteration of the algorithm, the probability to select an edge of F to be contracted is at most $\frac{2}{n}$.

(d) Assume that in all previous i iterations, no edge of F was selected to be contracted. Under this assumption, show that in the next iteration $i + 1$ of the algorithm the probability to select an edge of F to be contracted is at most $\frac{2}{n-i}$.

(e) Our algorithm will return the cut (A, B) if and only if no edge of F was contracted in iterations $1, 2, \ldots, n - 2$. Let us denote by E_j the event that no edge of F was contracted in the j-th iteration. Formulate the probability that our algorithm will return the cut (A, B) based on the events $E_1, \ldots, E_{n-2}$ in every iteration and obtain the desired bound of $\frac{1}{\binom{n}{2}}$ using your previous results.

Exercise 4.4 In this exercise, we are interested in obtaining (lower and upper) bounds on the number of minimum cuts that a graph may contain.

(a) Describe a graph with n vertices that has 2^{n-2} minimum (s, t)-cuts.

(b) Describe a graph with n vertices that has $\binom{n}{2}$ global minimum cuts.

(c) Use the probability bound of Exercise 4.3 to show that an unweighted graph cannot have more than $\binom{n}{2}$ global minimum cuts.

Further reading

– M. Stoer and F. Wagner. A simple min-cut algorithm. J. ACM, 44(4):585–591, 1997.

DRAFT

5. Maximum Flows

This chapter will focus on the maximum flow problem and its relationship to the minimum (s,t)-cut problem. We will also investigate three algorithms for computing a maximum flow in a flow network, namely, the one by Ford and Fulkerson, the one by Edmonds and Karp and the preflow-push algorithm. The rest of this summary is organized as follows: In Section 5.1, the maximum flow problem is formally introduced together with the necessary terminology. Section 5.2 focuses on the relationship between flows and (s,t)-cuts. In Sections 5.3 and 5.4 the algorithms by Ford and Fulkerson and by Edmonds and Karp for computing maximum flows are presented and analyzed. Sections 5.5 and 5.6 focus on Dinitz's and preflow-push algorithms.

5.1   The maximum flow problem

In this chapter, we will focus on an algorithmic problem, called the maximum flow problem, which is important because it can be used to express a wide variety of different kinds of problems, e.g., traffic in computer networks, circulations with demands, fluids in pipes and currents in electrical circuits. To define formally the maximum flow problem, we need first to define the notions (i) of a flow network and (ii) of a flow in a flow network.

> Definition 5.1 (Flow network) A flow network is a 4-tuple $\mathcal{N} = (G, c, s, t)$ for which the following apply:
>
> – $G = (V, E)$ is a connected and directed graph,
>
> – $c : E \to \mathbb{R}^+$ is a so-called capacity function, which maps each arc e of G to a positive number $c(e)$, called the capacity of e,
>
> – graph G has a designated source-node $s \in V$, with $indegree(s) = 0$, and
>
> – a designated target-node $t \in V$, with $outdegree(t) = 0$.
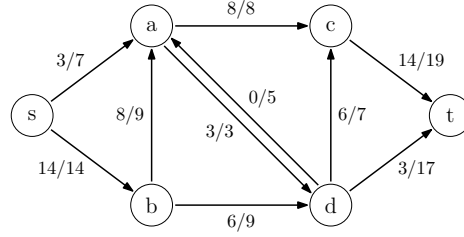
DRAFT

Figure 5.1: Illustration of a flow network together with a legit flow of it; each arc (u,v) has a label "a/b" meaning that the flow from u to v is a, while the capacity of arc (u,v) is b (namely, $f(u,v) = a$ and $c(u,v) = b$).

In this content, graph G of flow network $\mathcal{N}$ is called a network, the vertices of G are called nodes, while the edges of G are called arcs. Nodes s and t are often called terminals of network G, while the remaining nodes of G are called interior nodes or non-terminals. The flow in a flow network is formally defined as follows.

> **Definition 5.2** (Flow) A flow in a flow network $\mathcal{N} = (G, c, s, t)$ is defined as a function $f : E \to \mathbb{R}^+$ over the arcs of G such that the following two conditions hold:
>
> – Capacity constraint: $0 \le f(u,v) \le c(u,v)$, $\forall (u,v) \in E$
>
> – Flow conservation constraint: $\sum_{(v,u) \in E} f(v,u) = \sum_{(u,v) \in E} f(u,v)$, $\forall u \in V - \{s, t\}$
>
> The value $|f|$ of a flow f is defined as follows:
>
> $$|f| = \sum_{(s,v) \in E} f(s,v)$$

For an arc (u,v) of network G, we refer to $f(u,v)$ as the flow from u to v. By the capacity constraint it follows that the flow from u to v can be neither negative nor greater than the capacity of arc (u,v). The flow conservation constraint implies that for every internal node u of G the flow that enters node u must be the same as the flow that exists from u. In other words, flow cannot be stored at internal nodes of the network. The value of a flow accounts for the total flow that exists from the source-terminal s of $\mathcal{N}$. Since flow cannot be stored at internal nodes of the network, one naturally expects that the value of a flow can be equivalently defined as the total flow that enters the target-terminal t of $\mathcal{N}$ (we formally prove this property in Lemma 5.2).

> **Example 5.1** Fig. 5.1 illustrates a legit flow of a flow network, whose terminals are labeled s and t. Observe that s has no incoming arcs, while t has no outgoing arcs (the directions of the arcs of this network are indicated by arrows). Each arc in the example is labeled with two numbers. The left one indicates the flow, while the right one the capacity of the corresponding arc.
>
> Clearly, the flow of each arc is lower than or equal to its capacity. Also, it is not difficult to see that for every internal node, the incoming flow equals to the outgoing flow. Thus, both the capacity and the flow conservation constraints are satisfied.
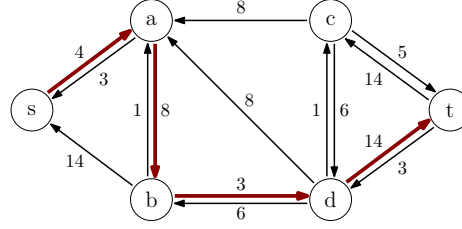
Figure 5.2: Illustration of the residual network of the flow network of Fig. 5.1. The red-drawn path corresponds to an augmenting path whose bottleneck is 3.

We are now ready to introduce the maximum flow problem as follows:

Definition 5.3 (Maximum Flow Problem) The input of the maximum flow problem is a flow network $\mathcal{N} = (G, c, s, t)$. In the output we seek to compute a maximum flow, namely, a flow whose value is greater than or equal to the value of any other flow of $\mathcal{N}$.

There exists several algorithms for solving the maximum flow problem. In Sections 5.3 and 5.4 we focus on two such algorithms, which both rely on finding augmenting paths (i.e., paths from s to t that allows augmenting a flow) in a auxiliary network, called residual network. Intuitively, the residual network of a flow network describes for each arc (u, v) the residual capacity of (u, v), i.e., how much flow one further can pass through (u, v). On the other hand, an augmenting path corresponds to a path from s to t in the residual network along which additional flow can be pushed in the network. Both notions are formally introduced bellow.

Definition 5.4 (Residual Network) Given a flow f of a flow network $\mathcal{N} = (G, c, s, t)$, the residual network of G with respect to flow f is a directed graph $G_f = (V, E_f)$ such that:

$$E_f = \{(u, v) \in V^2 : c_f(u, v) > 0\}$$

where $c_f : V \times V \to \mathbb{R}^2$ is a function defined as follows:

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & if\,(u, v) \in E \\ f(u, v) & if\,(v, u) \in E \\ 0 & otherwise \end{cases}$$

Definition 5.5 (Augmenting Path) Given a flow f of a flow network $\mathcal{N} = (G, c, s, t)$, an augmenting path is a path from s to t in the residual network G_f.

Example 5.2 Fig. 5.2 illustrates the residual network of the flow network of Fig. 5.1. The red drawn path from s to t is an augmenting path, which allows to increase the value of the flow by 3 units.

DRAFT

By definition, G and G_f are defined on the same set of nodes. In the following, we observe that G and G_f have a comparable number of arcs as well.

Observation 5.1 Given a flow f of a flow network $\mathcal{N} = (G, c, s, t)$, the number of arcs of the residual network G_f is at most twice the number of arcs of network G, namely:

$$|E_f| \leq 2 \cdot |E|$$

Proof. By definition of c_f, only arcs in E and their opposite arcs can have values greater than zero. Therefore, E_f contains at most $2|E|$ arcs.   □

5.2   Relation with (s,t)-cuts

Consider a flow network $\mathcal{N} = (G, c, s, t)$ and let (S, T) be an arbitrary (s,t)-cut of network G (that is, $s \in S$ and $t \in T$; see Section 4.1). Consider all arcs (u, v) of G with $u \in S$ and $v \in T$. Intuitively, one expects that the total capacity of these arcs forms an upper bound for the value of any legit flow in $\mathcal{N}$, as the flow that can be pushed from to s to t in G must necessarily use these arcs. This intuition applied to the (s,t)-cut (S, T) illustrated in Fig. 5.3 implies that the value of any legit flow of this network is at most 20.
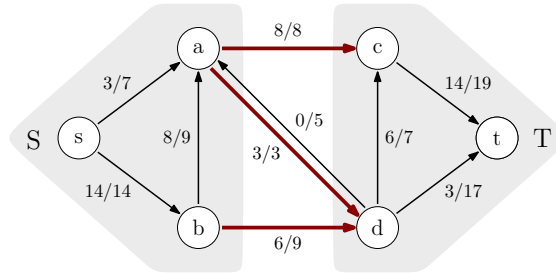
Figure 5.3: Illustration of an (s,t)-cut (S, T) of capacity $8 + 3 + 9 = 20$ in a flow network.

To confirm the intuition stated above, we introduce the notion of the capacity of an (s, t)-cut.

Definition 5.6 (Capacity of an (s, t)-cut) Given a flow network $\mathcal{N} = (G, c, s, t)$, the capacity $c(S, T)$ of an (s, t)-cut (S, T) is defined as the sum of the capacities of all arcs (u, v) of G with $u \in S$ and $v \in T$, namely:

$$c(S, T) := \sum_{\substack{u \in S, v \in T \\ (u,v) \in E}} c(u, v)$$

In this context, an (s, t)-cut of network G is minimum if and only if its capacity is smaller than or equal to the capacity of any other cut of G.

With Definition 5.6 at hand, it suffices to prove that the value of any flow f is upper bounded by the capacity $c(S,T)$ of any (s,t)-cut (S,T) of G.

> **Lemma 5.1** For every flow f of a flow network $\mathcal{N} = (G, c, s, t)$ and every (s,t)-cut (S,T), the value of f is upper bounded by the capacity $c(S,T)$ of (S,T), namely:
>
> $$|f| \le c(S,T)$$

Proof. By definition, the value $|f|$ of flow f is as follows:

$$|f| = \sum_{(s,v) \in E} f(s,v) \tag{5.1}$$

Since s has no incoming arcs, we can rewrite Eq. (5.1), as follows:

$$|f| = \sum_{(s,v) \in E} f(s,v) - \sum_{(v,s) \in E} f(v,s) \tag{5.2}$$

By the flow conservation constraint, we can further rewrite Eq. (5.2), as follows:

$$|f| = \sum_{(s,v) \in E} f(s,v) - \sum_{(v,s) \in E} f(v,s) + \sum_{\substack{u \in S - \{s\} \\ (u,v) \in E}} f(u,v) - \sum_{\substack{u \in S - \{s\} \\ (v,u) \in E}} f(v,u) \tag{5.3}$$

The sums of Eq. (5.3) can be combined as follows to obtain:

$$|f| = \sum_{u \in S} \left(\sum_{(u,v) \in E} f(u,v) - \sum_{(v,u) \in E} f(v,u) \right) \tag{5.4}$$

The sums of Eq. (5.4) can be split among those arcs with both endpoints in S and those with one endpoint in S and the other in T. So, we obtain:

$$|f| = \sum_{\substack{u \in S, v \in S \\ (u,v) \in E}} f(u,v) - \sum_{\substack{u \in S, v \in S \\ (v,u) \in E}} f(v,u) + \sum_{\substack{u \in S, v \in T \\ (u,v) \in E}} f(u,v) - \sum_{\substack{u \in S, v \in T \\ (v,u) \in E}} f(v,u) \tag{5.5}$$

By the flow conservation constraint, Eq. (5.5) can be rewritten as follows:

$$|f| = \sum_{\substack{u \in S, v \in T \\ (u,v) \in E}} f(u,v) - \sum_{\substack{u \in S, v \in T \\ (v,u) \in E}} f(v,u) \tag{5.6}$$

Since by the capacity constraint the flow of an arc can be neither negative nor greater than its capacity, it follows:

$$\sum_{\substack{u \in S, v \in T \\ (u,v) \in E}} f(u,v) \le \sum_{\substack{u \in S, v \in T \\ (u,v) \in E}} c(u,v) \quad \text{and} \quad - \sum_{\substack{u \in S, v \in T \\ (v,u) \in E}} f(v,u) \le 0$$

The last two inequalities together with Eq. (5.6) imply the desired property:

$$|f| \le \sum_{\substack{u \in S, v \in T \\ (u,v) \in E}} c(u,v) \qquad \qquad \square$$

Since a maximum flow is a flow, the following is a direct consequence of Lemma 5.1.

> **Corollary 5.1** Let f^* be a maximum flow of a flow network $\mathcal{N} = (G, c, s, t)$. Then, the value of f^* is upper bounded by the capacity of any (s, t)-cut of G, namely:
>
> $$|f^*| \leq c(S, T) \ \ \forall (s, t)\text{-cut } (S, T)$$

In Definition 5.2, we defined the value of a flow as the sum of the flows of the arcs exiting terminal s. In the following, lemma we prove that one can equivalently define the value of a flow as the sum of the flows of the arcs entering terminal t.

> **Lemma 5.2** Given a flow f of a flow network $\mathcal{N} = (G, c, s, t)$, the sum of the flows of the arcs exiting terminal s equals to the sum of of the flows of the arcs entering terminal t, namely:
>
> $$\sum_{(s,u)\in E} f(s, u) = \sum_{(u,t)\in E} f(u, t)$$

Proof. By Eq. (5.6) and by the fact that the value $|f|$ of f is the sum of the flows of the arcs exiting terminal s, we obtain:

$$\sum_{(s,u)\in E} f(s, u) \ = \ \sum_{\substack{u\in S, v\in T \\ (u,v)\in E}} f(u, v) \ - \ \sum_{\substack{u\in S, v\in T \\ (v,u)\in E}} f(v, u) \tag{5.7}$$

Note that Eq. (5.7) holds for any (s, t)-cut (S, T). To prove the lemma, we consider a specific (s, t)-cut (S, T) defined as follows:

$$S = V \setminus \{t\} \quad \text{and} \quad T = \{t\}$$

Then, Eq. (5.7) can be written as follows:

$$\sum_{(s,u)\in E} f(s, u) \ = \ \sum_{\substack{u\in V\setminus\{t\}, v\in\{t\} \\ (u,v)\in E}} f(u, v) \ - \ \sum_{\substack{u\in V\setminus\{t\}, v\in\{t\} \\ (v,u)\in E}} f(v, u) \ = \ \sum_{(u,t)\in E} f(u, t) \ - \ \sum_{(t,u)\in E} f(t, u)$$

Since terminal t has no outgoing arcs, the last equality yields:

$$\sum_{(s,u)\in E} f(s, u) = \sum_{(u,t)\in E} f(u, t) \qquad \qquad \square$$

Another consequence of Lemma 5.1 is given in the following lemma.

> **Lemma 5.3** If f is a flow of a flow network $\mathcal{N} = (G, c, s, t)$ and (S, T) is an (s, t)-cut of G such that $|f| = c(S, T)$, then f is a maximum flow and (S, T) is a minimum (s, t)-cut.

Proof. The fact that f is maximum can be easily seen by contradiction. In particular, if f were not maximum, then the value of the maximum flow would not be upper bounded by $c(S,T)$; a contradiction. So, f is maximum. The fact that (S,T) is a minimum cut can also be easily seen by contradiction. Assume to the contrary that there exists another (s,t)-cut (S',T') such that $c(S',T') < c(S,T)$. By Corollary 5.1, the maximum flow f is upper bounded by $c(S',T')$. This is a contradiction, as the value of the maximum flow per definition equals to $c(S,T)$; a contradiction.   □

5.3   The algorithm by Ford and Fulkerson

The algorithm by Ford and Fulkerson is an iterative algorithm to solve the maximum flow problem. It starts with a zero-flow and at each iteration, it first searches for an augmenting path. Recall that an augmenting path is simply a path from s to t in the residual network (see Definition 5.5). Therefore, finding an augmenting path or determining that one does not exist is an easy task, which can be done, e.g., by executing either a breath-first-search (BFS) or a depth-first-search (DFS) starting from terminal s. If an augmenting path exists, then the flow can be augmented; in particular, the value of the flow can be increased by the bottleneck (i.e., the smallest residual capacity of the arcs) of this path. On the other hand, if an augmenting path does not exist, then the algorithm reports that the flow computed in the previous iteration is maximum (which is a correct statement, as we will shortly show).

Algorithm 5.1: The core idea of the Ford Fulkerson algorithm

  Input   : A flow network $\mathcal{N} = (G,c,s,t)$.
  Output : The maximum flow.

1 Function FordFulkerson(G,s,t)
2 │   Initialize f to 0;
3 │   while (there is augmenting path P) do
4 │   │   augment f by the bottleneck of P;
5 │   end
6 │   return f;
7 end

Example 5.3   Fig. 5.2 illustrates the residual network of the flow network of Fig. 5.1 and an augmenting path of it. Fig. 5.5 shows the flow network after the flow augmentation. In particular, due to this path one can push 3 additional units of flow from s to t by changing the flow of the arcs of this path. Note that an augmenting path uses the option to push flow in the opposite direction of an arc by reducing the flow of the arc. Note that any path from s to t in the residual network is an augmenting path.

Example 5.4   Fig. 5.4 illustrates a sample execution of the algorithm by Ford and Fulkerson (starting from a zero-flow). The computed flow has value 20, which by Lemma 5.3 is maximum because the capacity of the (s,t)-cut $(\{s,a,b\},\{c,d,t\})$ is 20.
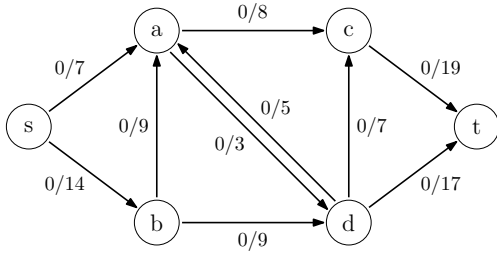
(a) The initial flow network of zero flow.
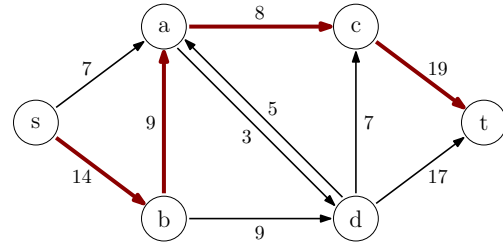
(b) Augmenting path (s, b, a, c, t) with bottleneck 8.
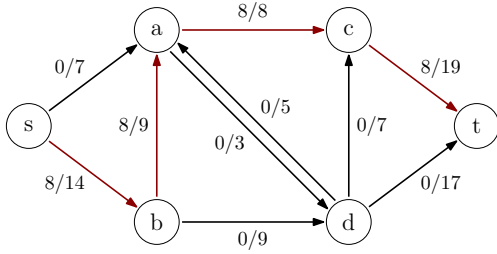
(c) Augmenting the flow by 8 along (s, b, a, c, t).
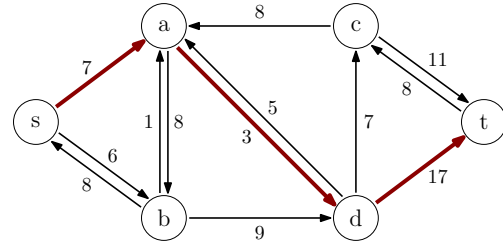
(d) Augmenting path (s, a, d, t) with bottleneck 3.
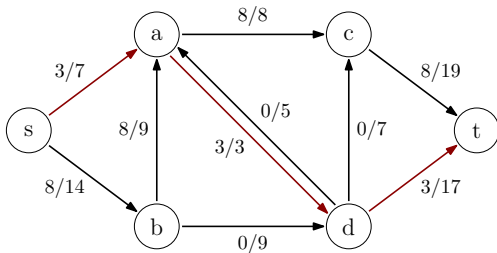
(e) Augmenting the flow by 3 along (s, a, d, t).
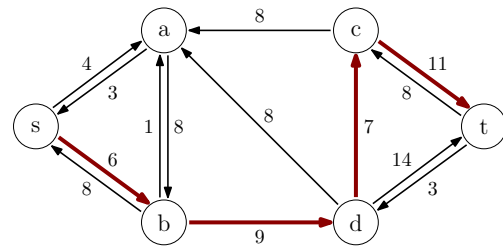
(f) Augmenting path (s, b, d, c, t) with bottleneck 6.
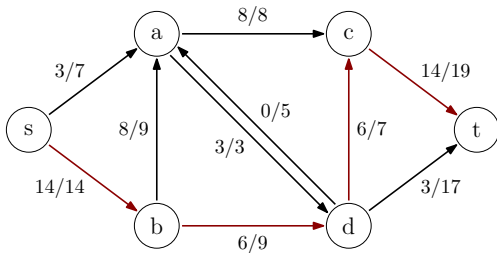
(g) Augmenting the flow by 6 along (s, b, d, c, t).
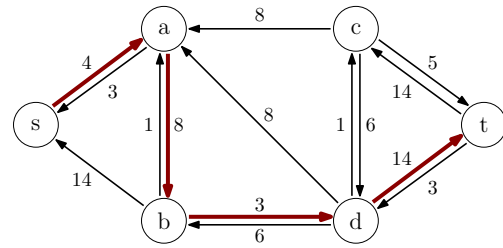
(h) Augmenting path (s, a, b, d, t) with bottleneck 3.
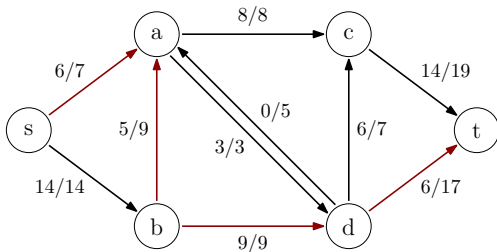
(i) Augmenting the flow by 3 along (s, a, b, d, t).
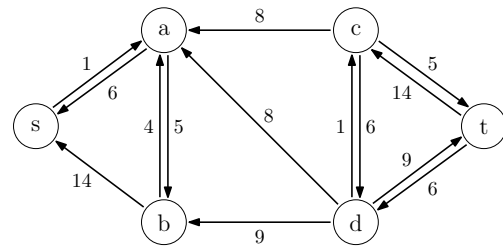
(j) A residual network with no augmenting path.

Figure 5.4: Illustration of a sample execution of the algorithm by Ford and Fulkerson: the left column illustrates the flow network, while the right one the residual network and a corresponding augmenting path.
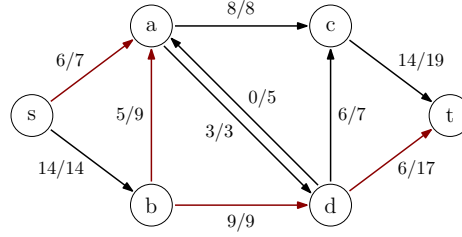
Figure 5.5: Illustration of the augmentation of the flow by using the augmenting path of the residual network of Fig. 5.2.

The correctness of the algorithm by Ford and Fulkerson directly follows from the following two theorems:

Theorem 5.1 (Augmenting paths theorem) A flow f of a flow network $\mathcal{N} = (G, c, s, t)$ is maximum if and only if there is no augmenting path in the residual network G_f.

Theorem 5.2 (Max-flow min-cut theorem) A flow f of a flow network $\mathcal{N} = (G, c, s, t)$ is maximum if and only if the value of f is equal to the capacity of an (s, t)-cut (S_0, T_0) in G, namely, $|f| = c(S_0, T_0)$.

We prove both theorems in the following theorem.

Theorem 5.3 Let f be a flow of a flow network $\mathcal{N} = (G, c, s, t)$. Then, the following are equivalent:

  (i) f is maximum

 (ii) there is no augmenting path in G_f

(iii) there is an (s,t)-cut (S_0, T_0) such that $|f| = c(S_0, T_0)$

Proof. We prove the equivalency by proving the following implications $(i) \Rightarrow (ii) \Rightarrow (iii) \Rightarrow (i)$.

$(i) \Rightarrow (ii)$: It follows directly from the definition of the residual network G_f.

$(ii) \Rightarrow (iii)$: Let S_0 be the set of nodes that are reachable from s in G_f, that is,

$$S_0 = \{v \in V : \exists \text{ path from } s \text{ to } v \text{ in } G_f\}$$

Since $s \in S_0$, it follows that $S_0 \neq \emptyset$. Denote by T_0 the remaining nodes of G, that is, $T_0 = V \setminus S_0$. Since there is no augmenting path in G_f (i.e., there is no path from s to t in G_f), it follows that t is not reachable from s in G_f. Therefore, $t \in T_0$, which implies that $T_0 \neq \emptyset$. Further, since $s \in S_0$, $t \in T_0$ and $S_0 \neq \emptyset \neq T_0$, the cut (S_0, T_0) is an (s, t)-cut of G. We proceed by identifying the following two properties:

P.1 $f(u,v) = c(u,v)$, for each arc (u,v) of G with $u \in S_0$ and $v \in T_0$: We prove this prop-
erty by contradiction. Assume that there exists an arc (u,v) of G with $u \in S_0$ and
$v \in T_0$ such that $f(u,v) < c(u,v)$. Since the residual capacity $c(u,v) - f(u,v)$ of (u,v)
is strictly positive, it follows that G_f contains an arc from u to v whose weight is
$c(u,v) - f(u,v)$ (see Definition 5.4). Since $u \in S_0$, it follows that there is a path P
from s to u in G_f. Since G_f has an arc from u to v, it follows that $P \cup \{(u,v)\}$ is
a path from s to v in G_f. Therefore, $v \in S_0$. This conclusion, however, contradicts
the fact that v belongs to T_0.

P.2 $f(v,u) = 0$, for each arc (v,u) of G with $u \in S_0$ and $v \in T_0$: We prove this property
by contradiction. Assume that there exists an arc (u,v) of G with $u \in S_0$ and $v \in T_0$
such that $f(v,u) > 0$. Since Definition 5.4, it follows that G_f has an arc from u to
v whose weight is $f(v,u)$. Since $u \in S_0$, it follows that there is a path P from s to
u in G_f. Since G_f has an arc from u to v, it follows that $P \cup \{(u,v)\}$ is a path from
s to v in G_f. Therefore, $v \in S_0$. This conclusion, however, contradicts the fact that
v belongs to T_0.

By Eq. (5.6) of the proof of Lemma 5.1 applied on the (s,t)-cut (S_0, T_0) we obtain:

$$|f| \;=\; \sum_{\substack{u \in S_0, v \in T_0 \\ (u,v) \in E}} f(u,v) - \sum_{\substack{u \in S_0, v \in T_0 \\ (v,u) \in E}} f(v,u) \qquad (5.8)$$

Property P.1 implies that the first of the two sums of Eq. (5.8) is equal to the capacity
$c(S_0, T_0)$ of the (s,t)-cut (S_0, T_0), while Property P.2 implies that the second sum of
Eq. (5.8) is zero. Hence, $|f| = c(S_0, T_0)$

(iii) $\Rightarrow$ (i): It follows by Lemma 5.3 $\square$

The detailed algorithm (see Algorithm 5.2) includes the computation of the residual network
for finding the necessary augmenting paths. The bottleneck of each augmenting path is used
for updating the computed flow. The update of the residual network is done in the same way
as the update of the flow.

Complexity analysis: We will analyze the time complexity of the algorithm by Ford and
Fulkerson under the following assumption:

Assumption: The capacity of each arc of the network is a positive integer, i.e., $c : E \to \mathbb{N}$.

Under this assumption, at every iteration of the algorithm by Ford and Fulkerson, the bottle-
neck of the augmenting path is an integer. This implies that the value of the maximum flow
is also an integer. Since the algorithm starts with a zero-flow and since each flow augmenta-
tion increases the value of the flow by at least one unit, the algorithm will terminate after at
most $|f^*|$ flow augmentations, where f^* denotes the maximum flow (see Example 5.5). As an
augmenting path in G_f can be found in $\mathcal{O}(m)$ (e.g., by BFS or DFS), the complexity of the
algorithm is:

$$\mathcal{O}(m|f^*|)$$

Algorithm 5.2: The algorithm by Ford Fulkerson

Input   : A flow network $\mathcal{N} = (G, c, s, t)$.
Output : The maximum flow.

1 Function FordFulkerson(G, s, t)
2 | foreach arc (u, v) of G do
3 | | $f(u, v) = f(v, u) = 0$;
4 | end
5 | Compute G_f;
6 | while there is a path P from s to t in G_f do
7 | | $x = \min\{c_f(u, v); (u, v) \in P\}$;
8 | | foreach arc (u, v) of P do
9 | | | if (u, v) belongs to G then
10 | | | | $f(u, v) = f(u, v) + x$;
11 | | | end
12 | | | else
13 | | | | $f(u, v) = f(u, v) - x$;
14 | | | end
15 | | end
16 | | Update G_f;
17 | end
18 | return f;
19 end

Note that this is an output-sensitive analysis, whereby the time complexity of the algorithm depends on the size of the output.

> Example 5.5 Fig. 5.6 shows an example, in which the algorithm by Ford and Fulkerson may perform exactly $|f^*|$ flow augmentations (the example illustrates only the residual network and not the corresponding flow network). Clearly, the maximum flow can be calculated by only two flow augmentations along the paths $s \to a \to t$ and $s \to b \to t$, each having a bottleneck of 1.000. Since the algorithm by Ford and Fulkerson does not choose the augmenting paths using a particular rule, it may happen that it always chooses one of the paths $s \to a \to b \to t$ or $s \to b \to a \to t$ whose bottlenecks are unit, thus resulting in 2.000 flow augmentations, which equals to $|f^*|$.
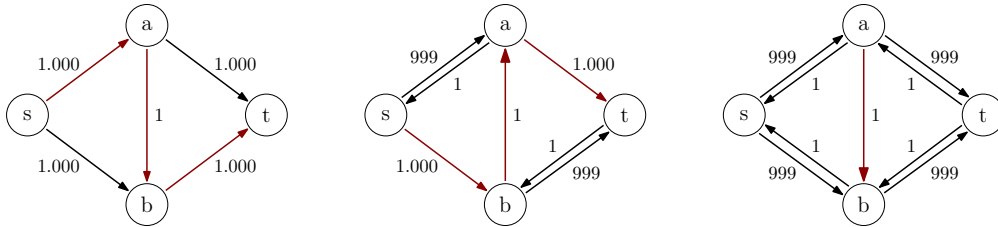
Figure 5.6: Illustration of an example, in which the algorithm by Ford and Fulkerson may perform $|f^*|$ flow augmentations.

5.4   The algorithm by Edmonds and Karp

The algorithm by Edmonds and Karp is an refinement of the algorithm by Ford and Fulkerson. Sometimes it is also referred to as an implementation of the algorithm by Ford and Fulkerson, as it addresses the aforementioned problem of choosing a "good" augmenting path (for a more efficient computation of a maximum flow). More specifically, while the algorithm by Ford and Fulkerson does not impose any constraint on the choice of the augmenting path (when more than one such paths exist in the residual network), the refinement by Edmonds and Karp chooses a shortest one; see Definition 5.7.

> **Definition 5.7**   Given a flow f of a flow network $\mathcal{N} = (G, c, s, t)$, we say that an augmenting path in the residual graph G_f is shortest if it has the least number of arcs among all augmenting paths in G_f.

Since the algorithm by Edmonds and Karp is a special case of the algorithm by Ford and Fulkerson, the correctness of this algorithm follows directly from Theorem 5.3 In the following, we analyze the time complexity of the algorithm by Edmonds and Karp. Central in this analysis is the following notion.

> **Definition 5.8** (Node distance in G_f)   Given a flow f of a flow network $\mathcal{N} = (G, c, s, t)$ and an internal node v of G, the distance $d_f(s, v)$ between s and v is the minimum number of arcs between s and v in G_f.
>
> $$d_f(s, v) = \text{minimum number of arcs from s to v in } G_f, v \in V \setminus \{s, t\}$$
>
> If there is no path from s to v in G_f, then the distance between s and v in G_f is $+\infty$, that is, $d_f(s, v) = +\infty$.

In the following lemma, we prove that for every internal node v of G, the distance between s and v does not decrease with flow augmentations.

> **Lemma 5.4**   Given a flow network $\mathcal{N} = (G, c, s, t)$, let f and f^* be two flows of $\mathcal{N}$ such that f is obtained from f^* by augmenting f along path P_f, where P_f is a (shortest) augmenting path in G_f. Then, for every internal node $v \in V \setminus \{s, t\}$ of network G it holds:
> $$d_f(s, v) \leq d_{f^*}(s, v)$$

Proof. Assume, for a contradiction, that there exists an internal node v of G such that:

$$d_{f^*}(s, v) < d_f(s, v) \tag{5.9}$$

If network G contains more than one internal node with the aforementioned property, then we choose v such that its distance from s in G_{f^*} is minimum among all these internal nodes,

namely:

$$d_{f^*}(s,v) \le d_{f^*}(s,x) \quad \text{for all } x \in V \setminus \{s,t\} \text{ with } d_{f^*}(s,x) < d_f(s,x) \tag{5.10}$$

In other words, v is chosen to be closest to s in G_{f^*}. This implies that for any internal vertex z of G whose distance to s in G_{f^*} is strictly smaller than the one of v, it holds $d_f(s,z) \le d_{f^*}(s,z)$. Namely:

$$d_{f^*}(s,z) < d_{f^*}(s,v), \ z \in V \setminus \{s,t\} \ \overset{(5.10)}{\Rightarrow} \ d_f(s,z) \le d_{f^*}(s,z) \tag{5.11}$$

Let $P : s \to \ldots \to u \to v$ be the shortest path from s to v in G_{f^*}, where (u,v) is the last arc in P. It follows:

$$d_{f^*}(s,v) = d_{f^*}(s,u) + 1 \tag{5.12}$$

By Eq. (5.11), we obtain:

$$d_f(s,u) \le d_{f^*}(s,u) \tag{5.13}$$

We proceed by considering two cases. Assume first that (u,v) is an arc of G_f (i.e., $f(u,v) < c(u,v)$). Since the length of the shortest path from s to v in G_f is at least the length of the path from s to u plus one (i.e., since there is an arc from u to v), we obtain:

$$d_f(s,v) = d_f(s,u) + 1 \overset{(5.13)}{\le} d_{f^*}(s,u) + 1 \overset{(5.12)}{=} d_{f^*}(s,v)$$

Assume now that (u,v) is not an arc of G_f (i.e., $f(u,v) = c(u,v)$). Since (u,v) is an arc of G_{f^*}, arc (v,u) is necessarily on a shortest path $s \to \ldots \to v \to u \to \ldots \to t$ in G_f. Hence, we can conclude:

$$d_f(s,v) = d_f(s,u) - 1 \overset{(5.13)}{\le} d_{f^*}(s,u) - 1 = d_{f^*}(s,v) - 2$$

Both cases form a contradiction by Eq. (5.9), which completes the proof. $\qquad\square$

Lemma 5.5 Let $\mathcal{N} = (G,c,s,t)$ be a flow network. In the algorithm by Edmonds and Karp, each arc of network G can be bottleneck of an augmenting path at most $n/2 - 1$ times, where n denotes the number of nodes of G.

Proof. Consider an arc (u,v) of network G and assume that (u,v) is the bottleneck of a (shortest) augmenting path $s \to \ldots \to u \to v \to \ldots \to t$ in residual network G_f. Since this path is shortest, it follows:

$$d_f(s,v) = d_f(s,u) + 1 \tag{5.14}$$

After the flow augmentation, arc (u,v) disappears from the resulting residual network. It appears again when the arc (v,u) belongs to a shortest path $s \to \ldots \to v \to u \to \ldots \to t$ in some residual network G_{f^*}. Hence:

$$d_{f^*}(s,u) = d_{f^*}(s,v) + 1 \tag{5.15}$$

By Lemma 5.4, it follows:

$$d_{f^*}(s,v) \ge d_f(s,v) \tag{5.16}$$

Putting all together, we obtain:

$$d_{f^*}(s,u) \overset{(5.15)}{=} d_{f^*}(s,v) + 1 \overset{(5.16)}{\geq} d_f(s,v) + 1 \overset{(5.14)}{=} d_f(s,u) + 2$$

In other words, for (u,v) to appear again in some residual network, the distance of u from s has to increase by at least 2 with respect to its distance, when (u,v) was a bottleneck. As the distance from s to u is at most $n-2$ in the residual network, the lemma follows.   □

We are now ready to estimate the complexity of the algorithm by Edmonds and Karp.

> **Theorem 5.4** Given a flow network $\mathcal{N} = (G,c,s,t)$ with n nodes and m arcs, the algorithm by Edmonds and Karp computes a maximum flow of $\mathcal{N}$ in $\mathcal{O}(m^2 n)$ time.

Proof. The bound directly follows from the following observations:

– By definition, each augmenting path has a bottleneck.

– By Lemma 5.5, there are at most $(n/2 - 1)m$ augmenting paths.

– Each augmenting path can be found in $\mathcal{O}(m)$ time, e.g., using BFS.

Thus, the algorithm by Edmonds and Karp computes a maximum flow in $\mathcal{O}(m^2 n)$ time.   □

5.5   Dinitz's algorithm

Dinitz's algorithm forms an extension of the algorithm by Edmonds and Karp (see Section 5.4), which by introducing an additional technique to compute multiple augmenting paths simultaneously reduces the running time from $\mathcal{O}(m^2 n)$ to $\mathcal{O}(n^2 m)$.

The core of Dinitz's algorithm is the notion of a blocking flow. However, in order to formally define this notion, we need to first introduce a few concepts.

> **Definition 5.9** (Admissible network) Let f be a flow of a network (G,c,s,t), and let G_f denote the corresponding residual network. An arc (v,w) is admissible if (v,w) lies on some shortest path from s to t in G_f. The subgraph of G_f induced by the admissible arcs is called admissible network and is denoted by L_f.

> **Definition 5.10** (Admissible path) A path from s to t in G_f is called admissible if and only if all its arcs also belong to L_f, i.e., they are admissible.
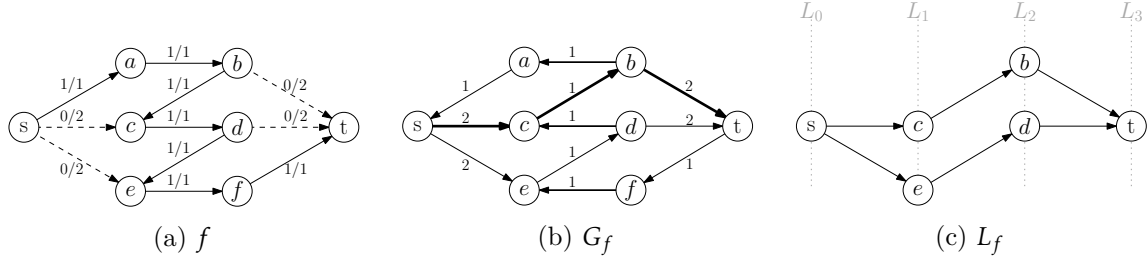
Figure 5.7: Illustration of: (a) a blocking flow f, (b) an augmenting path (drawn bold) in the residual network G_f, and (c) the admissible network L_f.

> **Observation 5.2** Let f be a flow of a network (G, c, s, t) that is not maximum, and let G_f denote the corresponding residual network. Then, any augmenting path computed by the algorithm by Edmond and Karp in G_f is admissible.

Given a flow f on a network (G, c, s, t), one can visualize the admissible network L_f as a set of layers $L_0, L_1, \dots$ as follows (see Fig. 5.7): layer L_0 contains s, layer L_1 contains the nodes of G_f whose distance (according to the number of arcs in G_f) from s is 1, and so on; the final layer, obviously, contains only t. Then, every admissible arc goes from one layer to the next, and there is no arc (v, w) in G_f such that $v \in L_i$ and $w \in L_j$ with $j \geq i + 2$. We formalize the layering of the nodes in the following definition.

> **Definition 5.11** (Layering) Let f be a flow of a network (G, c, s, t), and let G_f denote the corresponding residual network. A layering of the nodes of the admissible network L_f is a partition of its nodes into layers according to their distance from s in L_f.

With these definitions in mind, we are now ready to introduce the notion of a blocking flow.

> **Definition 5.12** (Blocking flow) A flow f is blocking if and only if every admissible path from s to t in L_f has at least one arc saturated by f.

Observe that a blocking flow is not necessarily maximum. For an example, refer to the flow f of Fig. 5.7a. The fact that f is blocking follows from the observation that there exist two admissible paths in the admissible network L_f as depicted in Fig. 5.7c and each of them has at least one arc saturated by f. On the other hand, Fig. 5.7b shows the corresponding residual network G_f, which contains an augmenting path thereby establishing that f is not maximum. We summarize this observation in the following.

> **Observation 5.3** A blocking flow is not necessarily maximum.

On the other hand, it is not difficult to see that a maximum flow is always blocking. This property follows from the fact that a maximum flow implies that there is no path (and thus

no admissible path) from s to t in the residual network.

Even though we have not described how one can compute a blocking flow, we can describe the principle idea of the algorithm by Dinitz; see Algorithm 5.3. Observe that the algorithm has the same structure as the one by Ford and Fulkerson and its variant by Edmonds and Karp: start with an zero flow, and augment this flow until no residual s-t path can be found. The main difference lies in the way the flow is augmented, i.e., by blocking flows.

Algorithm 5.3: The core idea of the algorithm by Dinitz

   Input   : A flow network $\mathcal{N} = (G, c, s, t)$.
   Output : The maximum flow.

1 Function Dinitz(G, s, t)
2   Initialize f to 0;
3   while G_f has an s-t path do
4          $g \leftarrow$ a blocking flow in L_f;
5          $f \leftarrow f + g$;
6   end
7   return f;
8 end

The key property of blocking flows is that augmenting along a blocking flow always increases the shortest s-t path distance in the residual network as described in the following lemma.

> **Lemma 5.6** At every augmentation by a blocking flow in Dinitz's algorithm, the distance from s to t in the residual network (strictly) increases.

Proof. Consider a flow augmentation by a blocking flow g in Dinitz's algorithm. Let f and f^* be the flows before and after the augmentation, and let G_f and G_{f^*} be the corresponding residual networks. Denote by $d_f(s, t)$ and by $d_{f^*}(s, t)$ the shortest s-t path distance in G_f and in G_*, respectively. We seek to prove that $d_f(s, t) < d_{f^*}(s, t)$.

Consider an admissible path P from s to t in G_{f^*}. Note that P cannot be admissible in G_f, because g saturates some arc of every admissible s-t path in G_f. Thus, some arc $(v, w) \in P$ is a "new" arc in G_{f^*}. In order for (v, w) to be a new arc, either g sent flow from w to v or (v, w) was an arc of G_f but not of L_f. In both cases:

$$d_f(s, w) \leq d_f(s, v) \tag{5.17}$$

On the other hand, we know from Lemma 5.4 that the distance of a node from s does not decrease with flow augmentations. Symmetrically, one proves that the distance of a node from t does not decrease with flow augmentations. Hence:

$$d_f(s, v) \leq d_{f^*}(s, v) \quad \text{and} \quad d_f(w, t) \leq d_{f^*}(w, t) \tag{5.18}$$

We are now ready to put everything together:

$$d_f(s, t) = d_f(s, w) + d_f(w, t) \overset{(5.17)}{\leq} d_f(s, v) + d_f(w, t) \overset{(5.18)}{\leq} d_{f^*}(s, v) + d_{f^*}(w, t) < d_{f^*}(s, t) \qquad \square$$
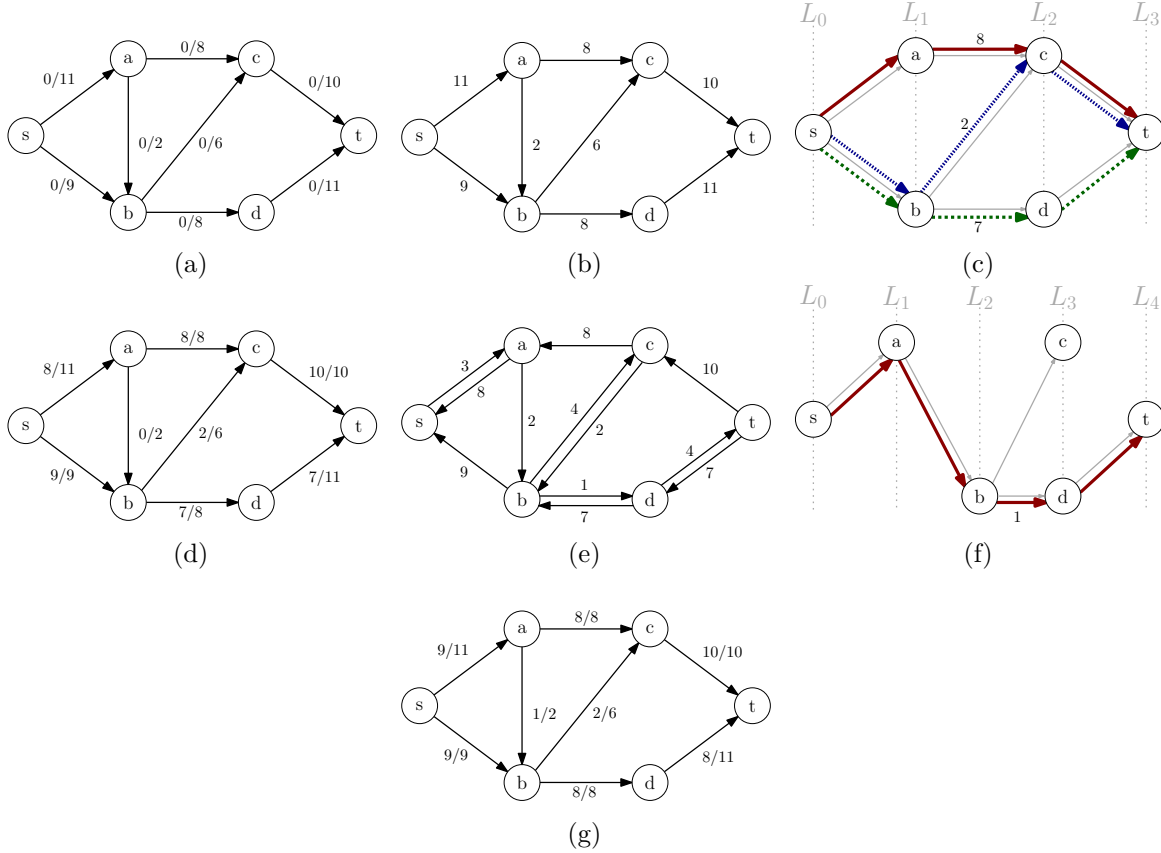
Figure 5.8: Illustration of Dinitz's algorithm by an example: (a) the initial (zero) flow, (b) the corresponding residual network, (c) the admissible network and a blocking flow (consisting of three blocking paths; solid, dotted and dashed), (d) the flow augmentation, (e) the corresponding the residual network obtained after the augmentation, (f) the admissible network and a blocking flow (consisting of a single blocking path), and (g) the final (maximum) flow.

As a direct corollary of Lemma 5.6 we obtain.

> **Corollary 5.2** Dinitz's algorithm performs at most $n-1$ blocking flow augmentations.

Recall that the upper bound on the number of flow augmentations that we had provided for the algorithm by Edmonds and Karp in Lemma 5.5 was $\mathcal{O}(nm)$, which suggests that Dinitz's algorithm is much more efficient in terms of the required number of flow augmentations. However, for this to be worth it, the extra time that we need to find a blocking flow must not be too much worse than finding an augmenting path.

Dinitz's idea to compute a blocking flow is to repeat the following procedure as long as a path from s to t in the admissible network L_f is possible to be found (see also Algorithm 5.4):

"Find a path P from s to t in L_f (if any) and saturate one arc e of P by sending flow via P from s to t. Then, remove e (and other arcs that are potentially saturated) and repeat."

Algorithm 5.4: The algorithm by Dinitz to find a blocking flow.

> Input   : The residual network G_f and the admissible network L_f of a network
> (G, c, s, t).
> Output : A blocking flow.

1 <u>Initialize</u>: $P \leftarrow \{s\}$, $v \leftarrow s$, Go to Advance;

2 <u>Advance</u>:
3 if $\nexists\, e = (v, w)$ then
4 $\quad$ Go to Retreat;
5 else
6 $\quad$ $e \leftarrow (v, w)$;
7 $\quad$ $P \leftarrow P \cup \{w\}$;
8 $\quad$ $v \leftarrow w$;
9 $\quad$ if $v \neq t$ then
10 $\quad\quad$ Go to Advance;
11 $\quad$ else
12 $\quad\quad$ $\delta \leftarrow \min_{e \in P} \{c(e) - f(e)\}$;
13 $\quad\quad$ Augment the flow along P by δ;
14 $\quad\quad$ Delete all saturated arcs of P from L_f;
15 $\quad\quad$ Go to Initialize;
16 $\quad$ end
17 end

18 <u>Retreat</u>:
19 if $v = s$ then
20 $\quad$ Stop;
21 else
22 $\quad$ $(u, v) \leftarrow$ the last arc of P;
23 $\quad$ Delete v from P and (u, v) from L_f;
24 $\quad$ $v \leftarrow u$;
25 $\quad$ Go to Advance;
26 end

More precisely, the algorithm by Dinitz to compute a blocking flow uses DFS on L_f by continuously applying two operations, starting at the source s:

- Advance: Follow an admissible arc of the current node in the forward direction (if any).

- Retreat: If there is no outgoing admissible arc at the current node, go back along the arc that lead to the current node, and delete it.

Once t is reached, we add the flow of the path lead to it to the blocking flow, update the residual capacities, and repeat from the source. The procedure is terminated once a search retreats all the way back to s.

> **Theorem 5.5** Dinitz's algorithm computes a maximum flow in $\mathcal{O}(n^2 m)$ time.

Proof. The correctness of the algorithm follows from the Augmenting Paths theorem (see Theorem 5.1) and from the correctness of Algorithm 5.4. For the later, observe that an arc (v, w) is deleted from L_f when it is either saturated or all paths from w to t pass through some already saturated arcs. So, the deletion of (v, w) does not influence augmenting paths to be discovered later. Since Algorithm 5.4 searches (finds) paths from s to t in L_f as long as such paths exists, by induction, the union of these paths form a blocking flow at the end.

To complete the proof of the theorem, it remains to discuss the time complexity. To this end, we first note that the creation of each admissible network can be done in $\mathcal{O}(m)$ time by applying BFS on G_f. So, all admissible networks can be computed in $\mathcal{O}(nm)$ time by Lemma 5.6. Thus, to prove the time complexity it suffices to prove that Algorithm 5.4 needs $\mathcal{O}(nm)$ time, since by Lemma 5.6 it is called $\mathcal{O}(n)$ times. We will prove this by showing that the number of Advance and Retreat operations at a single call of Algorithm 5.4 is bounded by $\mathcal{O}(nm)$.

We first prove an asymptotically worse bound. In particular, we can easily observe that the number of Advance and Retreat operations that can be performed before an augmentation occurs (or before the algorithm stops) is $\mathcal{O}(m)$, since in worst case all arcs of L_f will be traversed and/or deleted, depending on the operation. To bound the number of flow augmentations performed before the algorithm stops, we observe that each flow augmentation results in an arc-deletion in L_f. Hence, the number of flow augmentations is $\mathcal{O}(m)$. This implies that in worst case the number of Advance and Retreat operations at a single call of Algorithm 5.4 is bounded from $\mathcal{O}(m^2)$.

To improve this bound, we use amortised analysis (see Section 2.2). Recall that every Advance operation either ends in a flow augmentation or it is followed by a Retreat operation. Here, we charge each Advance operation to the flow augmentation or to the Retreat operation that follows it. Hence, we can make the following observations:

– Since each Retreat operation deletes an arc of L_f, there are at most $\mathcal{O}(m)$ such operations per blocking flow. Every Retreat operation has exactly one Advance operation charged to it. So, the total amount of charge to Retreat operations per blocking flow is $\mathcal{O}(m)$.

– Since each flow augmentation deletes (at least) one arc of L_f, there are at most $\mathcal{O}(m)$ such operations per blocking flow. However, there can be $\mathcal{O}(n)$ Advance operations charged to a flow augmentation, since there are at most n arcs in an path from s to t in L_f. So, the total amount of charge to flow augmentations per blocking flow is $\mathcal{O}(mn)$.

Adding these up overall blocking flows, we have $\mathcal{O}(nm)$ time per blocking flow, and $\mathcal{O}(n^2 m)$ for the algorithm, as desired.   □

We conclude this section by mentioning that the bound of Lemma 5.6 in the number of block flow augmentations is tight, in the sense that there exist *n*-node networks that require exactly
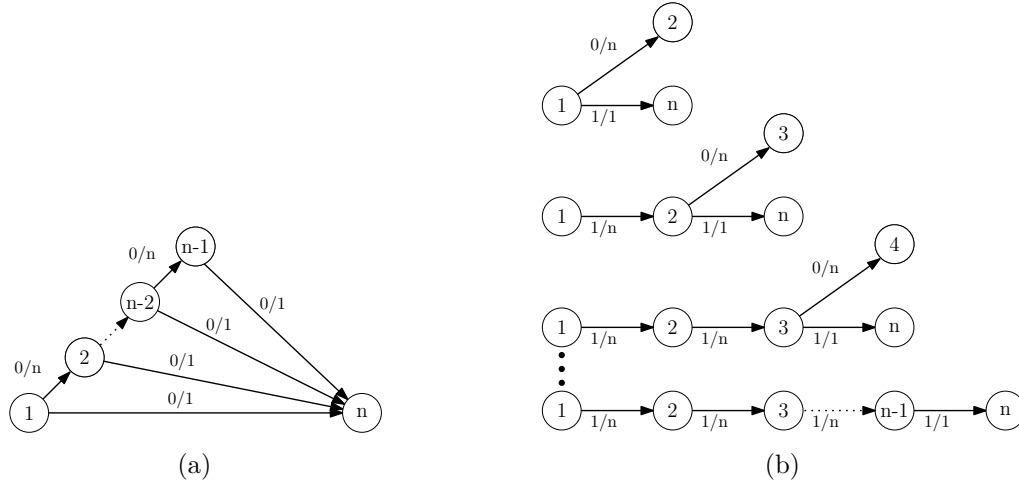
Figure 5.9: Illustration of: (a) a sample network introduced by Waissi back in 1991, and (b) $n-1$ block flow augmentations (in the corresponding admissible networks) performed by Dinitz's algorithm .

$n-1$ blocking flow augmentations. Such an example is illustrated in Fig. 5.9a. As it can be seen in Fig. 5.9b, this particular network requires exactly $n-1$ block flow augmentations.

5.6   The Preflow-Push algorithm

The main idea of the preflow-push algorithm is to relax the flow conservation constraint by allowing the incoming flow to a node to be greater than its incoming flow. In other words, instead of maintaining a flow throughout the algorithm (as, e.g., both the algorithms by Ford and Fulkerson and by Endmond and Karp do), the preflow-push algorithm maintains a so-called preflow, which is formally defined (similarly to a flow) as follows.

> Definition 5.13 (Preflow) A preflow in a flow network $\mathcal{N} = (G, c, s, t)$ is a function $f : E \to \mathbb{R}^+$, which maps every arc to a positive number and meets following constraints:
>
> Capacity constraint. For each arc (u, v) of G, it holds: $0 \le f(u, v) \le c(u, v)$.
>
> Non-negative excess constraint. For each node u of G different from s and t, the difference between the preflow that enters node u and the preflow that exists node u is called the excess of u, is denoted as excess(u) and must stay non-negative, that is,
> $$\text{excess}(u) = \sum_{(v,u)\in E} f(u, v) - \sum_{(u,v)\in E} f(v, u) \ge 0, \quad \forall u \in V - \{s, t\}$$

The algorithmic idea of maintaining a preflow (instead of a flow) will be formalized later in Section 5.6.1. Before doing this, however, we need one additional concept, namely, the one of a labeling and in particular of a labeling that is compatible with a preflow. Both concepts

DRAFT
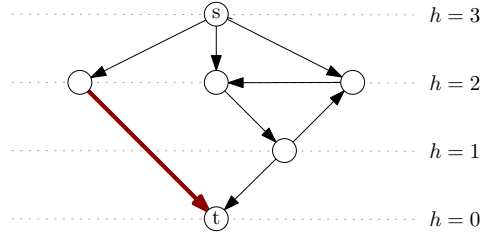
Figure 5.10: Illustration of the concept of labeling. In this case, any preflow f would not be compatible with labeling h, because of the red-colored arc.

are formally introduced in the following.

> **Definition 5.14** (node labeling) A node-labeling or simply labeling in a flow network $\mathcal{N} = (G, c, s, t)$ is a function $h : V \to \mathbb{N}^+$, which maps every node of G to a non-negative integer number.

Intuitively, one can think of a labeling as a function, which maps each node to a certain "height". Accordingly, each arc is pointing into a certain direction based on the heights of its endnodes; upwards (if the height of its source is smaller than the one of its target), downwards (if the height of its source is greater than the one of its target) or straight (if the height of its source is equal to the one of its target); see Fig. 5.10.

> **Definition 5.15** (Compatibility) A preflow f and a labeling h are compatible if, and only if, the following conditions are met:
>
> C.1 $h(t) = 0$.
>
> C.2 $h(s) = n$.
>
> C.3 $h(v) \le h(w) + 1$ for each arc (v, w) in the residual network G_f.

Condition C.1 requires the height of the sink t of the network to be zero, while Condition C.2 requires the height of the source s to be equal to the number of nodes in graph G. Finally, Condition C.3 requires the arcs of G_f not to be too steep downwards, that is, if an arc is pointing downwards, then the maximum height difference of its endnodes is 1.

We are now ready to prove a central property for the preflow-push algorithm.

> **Theorem 5.6** If f is a preflow and h is a compatible labeling, then there is no s-t path in G_f.

Proof. Assume to the contrary that there is an s-t path $P : s \to \dots \to v_{n-2} \to v_{n-1} \to t$ in G_f; see Fig. 5.11. The length of P is at least 1 (i.e., P is one arc) and at most $n-1$ (i.e., P spans

all nodes of the graph).

Figure 5.11: Illustration of s-t path P.

According to the definition of compatibility, the height of t is 0. For node v_{n-1}, we know that its height is at most 1, as otherwise the arc (v_{n-1}, t) would be too steep downwards and thus f and h would not be compatible. Similarly, the height of node v_{n-2} is at most 2. Since the length of path P is at most $n-1$, the height of node s is at most $n-1$. However, this is a contradiction to the definition of compatibility which requires the height of s to be n. Hence, we can conclude that there is indeed an s-t path in G_f, if f is a preflow and h is a compatible labeling. $\qquad\square$

The property described in Theorem 5.6 is essential for the preflow-push algorithm. In particular, assume for the time being that the preflow-push algorithm maintains a preflow f, which is always compatible with a labeling h. Then, if at some point in the course of the algorithm this preflow (that is, $\text{excess}(v) \geq 0$, $\forall v \in V \setminus \{s, t\}$) is transformed into a flow (that is, $\text{excess}(v) = 0$, $\forall v \in V \setminus \{s, t\}$), then we know by the termination condition of the Ford and Fulkerson algorithm that the maintained preflow is actually a maximum flow. This is the core idea of the preflow-push algorithm.

5.6.1   The Preflow-Push algorithm

As already mentioned, the algorithm maintains at any point of its execution a preflow f and a labeling h, which are compatible (we will prove this property soon). Initially, the algorithm sets the preflow $f(u, v)$ of each arc (u, v) as follows:

$$f(u, v) = \begin{cases} c(u, v), & \text{if } u = s \\ 0, & \text{otherwise} \end{cases} \tag{5.19}$$

Accordingly, the labeling $h(u)$ of each node u is initially set as follows:

$$h(u) = \begin{cases} n, & \text{if } u = s \\ 0, & \text{otherwise} \end{cases} \tag{5.20}$$

In other words, the preflow of each arc starting from s is set to its capacity, while for each other arc it is set to zero. Accordingly, the heights of all nodes are set to zero, with the only exception of node s whose height is set to n.

The preflow-push algorithm consists of two main operations, the push and the relabel, which work complementary to each other, namely, when one cannot be applied, then the other can be applied, as long as the algorithm has not terminated. The pseudocode in Algorithm 5.5 outlines this property.

Algorithm 5.5: The Preflow-Push algorithm.

---

1 Initialize f and h as in Eqs. (5.19) and (5.20);
2 while $\exists v \in V \setminus \{s, t\}$ with excess$(v) > 0$ do
3   pick a node $v \in V \setminus \{s, t\}$ with $excess(v) > 0$;
4   if Push (f,h,v,w) applies then
5 | Push (f,h,v,w);
6   end
7   else
8 | Relabel (f,h,v);
9   end
10 end

Once the preflow and the labeling have been initialized, the preflow-push algorithm iteratively selects a node v with excess$(v) > 0$. Of course, if there is no such node v, then the algorithm will terminate, and by Theorem 5.6 and by the Ford and Fulkerson termination condition the computed preflow is a maximum flow (assuming that f and h are compatible). In a high-level description, if there is a node v with excess$(v) > 0$, then the algorithm checks whether the push operation can be applied, namely, whether there is an arc (v, w), such that the height of w is less than the height of v (see Algorithm 5.6). In the positive case, preflow can be pushed along the arc (v, w). Otherwise, it is not possible to push preflow along any arc (v, w) incident to v. So, node v will be relabelled by increasing its height (see Algorithm 5.7), in order to push preflow in some next iteration.

The push operation. This method only applies, if there exist a node v with excess$(v) > 0$ and a direct neighbour w of v (i.e., $(v, w) \in E$), such that $h(v) > h(w)$; see Algorithm 5.6. In particular, consider first the case in which the arc (v, w) is a forward arc in G_f (i.e., it is part of the original network). In this case, it is not possible to push more than the difference of the capacity $c(v, w)$ of the arc (v, w) and the preflow $f(v, w)$, as otherwise it would be a violation of the capacity constraint of arc (v, w) and in course of the algorithm the preflow f and height h should be always compatible. In addition, it is also not possible to push more than excess(v). So, the upper bound for pushing from node v to node w is the minimum of these two quantities. Consider now the case in which the arc (v, w) is a backward arc (i.e., it has been introduced to G_f due to some push along the reverse arc (w, v), which existed in the original network). In this case, we would ideally like to push from node v to node w the residual capacity of the original arc (w, v), which corresponds to the preflow $f(v, w)$. Similar to the previous case, however, it is not possible to push more than excess(v). So, the upper bound in this case is again the minimum of these two quantities. Thus, we may conclude that in both cases (i.e., when the arc (v, w) is forward or backward), δ is the minimum of the residual capacity of the arc (v, w) and the excess of v.

The relabel operation. This operation applies whenever it is not possible to apply the push operation, namely, when there is a node v with excess$(v) > 0$ but for every arc (v, w) that is incident to v, the height of w is greater or equal to the one of v. In this case, the relabel operation results in increasing the height of v by one. For an illustration refer to Fig. 5.12 and observe that after the relabeling operation has been applied there is an arc (v, w) incident to v, which allows pushing preflow along it.

Algorithm 5.6: The Push operation

---

1 Applies only if: $(v,w) \in E; v \notin \{s,t\}; h(v) > h(w)$ and $\text{excess}(v) > 0$
2 Function Push(f,h,v,w)
3   if (v,w) is a forward arc then
        // push forward as much as possible
4          $\delta = \min\{c(v,w) - f(v,w), \text{excess}(v)\}$;
5          $f(v,w) = f(v,w) + \delta$;
6   end
7   else
        // push backward as much as possible
8          $\delta = \min\{f(v,w), \text{excess}(v))\}$;
9          $f(v,w) = f(v,w) + \delta$;
10   end
11 end

Algorithm 5.7: The Relabel operation

---

1 Applies only if: $v \notin \{s,t\}, \text{excess}(v) > 0$ and $\forall (v,w) \in E \ h(w) \geq h(v)$
2 Function Relabel(f,h,v)
    // increase the height of v by 1
3      $h(v) = h(v) + 1$;
4 end

We conclude by observing that the push operation does not affect the height of the nodes, while the relabel operation does not affect the preflow.

5.6.2   Proof of correctness

In this section, we start by proving that f and h are compatible in the course of the algorithm. To this end, we first prove that f and h are compatible at the beginning of the algorithm (refer to Eqs. (5.19) and (5.20)).

> Lemma 5.7 f and h as defined by Eqs. (5.19) and (5.20) are compatible.

Proof. The algorithm starts by setting $h(t) = 0$ and $h(s) = n$. These heights never change in course of the algorithm, because the relabeling method is only applied to a nodes that are different from s and t. So, Conditions C.1 and C.2 clearly hold. To prove that Condition C.3 holds, we need to show that for every arc (v,w) of the residual network G_f the following property holds:

$$h(v) \leq h(w) + 1.$$

Consider any arc (s,v) that is outgoing from s. By Eq. (5.19), the preflow of (s,v) equals its capacity $c(s,v)$. So, the residual network G_f contains the reverse arc (v,s). Since by
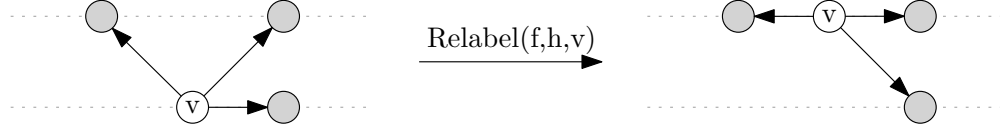
Figure 5.12: Illustration of the operation of relabeling a node v.

Eq. (5.20), $h(s) = n$ and $h(v) = 0$, the aforementioned property holds for (v, s). Consider now any arc (v, w) that is not an outgoing arc of s. By Eq. (5.20), $h(v) = h(w) = 0$ holds, which implies that Condition C.3 is trivially satisfied. Hence, f and h are initially compatible, as claimed.   □

Next, we prove that f and h are compatible in the course of the algorithm.

> Lemma 5.8 f and h are compatible throughout the algorithm.

Proof. The proof is by induction on the number of operations (push or relabel) performed by the algorithm. By Lemma 5.7, f and h are initially compatible (i.e., before applying any operation). In the inductive hypothesis, we assume that after the application of a certain number of operations, f and h are compatible. We next show that f and h are compatible once the next operation is applied. Since neither the push operation nor the relabel operation modifies $h(s)$ and $h(t)$, Conditions C.1 and C.2 hold. So, we can focus the proof on Condition C.3. We consider two cases depending on the type of the next operation in the algorithm.

Push(f, h, v, w): The push operation does not change the height of any node, but it may result in a new arc in G_f. It follows that Condition C.3 holds for every arc of the residual network which was part of it before the application of the push operation. To prove that Condition C.3 holds, we may focus on the case of a new arc in the residual network. In particular, if $f(v, w) = 0$ holds before the application of the push operation, then the arc (w, v) will appear in the residual network after the application of the push operation. Since node v is such that $h(v) > h(w)$ (as otherwise, the push operation would not be applied), we can conclude that for the new arc (w, v) the height relation is $h(w) \leq h(v) + 1$, which implies that Condition C.3 holds also for the new arc.

Relabel(f, h, v): The relabel operation is applied to a node v with excess$(v) > 0$ if for each arc (v, w) incident to v, it holds that $h(v) \leq h(w)$. After the application of the relabel operation, the new height of v is $h(v) + 1$, which is clearly upper bounded by $h(w) + 1$, where (v, w) is an arc of G_f. Thus, after the application of the relabel operation on node v, Condition C.3 holds.

In summary, we have shown that for both operations C.3 holds. So, f and h are compatible throughout the algorithm.   □

In the following, we prove that f is always a preflow in the course of the algorithm. To this end, we prove that f is a preflow at the beginning of the algorithm (refer to Eq. (5.19)).

> **Lemma 5.9** f is initially a preflow (i.e., as defined by Eq. (5.19)).

Proof. To prove that f is initially a preflow, it suffices to show that it meets both the capacity constraint and the non-negative excess constraint. The former is clearly met by Eq. (5.19). So, it suffices to show that for each node v different from s and t has non-negative excess. Assume first that v is a neighbor of s, that is (v,s) in an outgoing arc of s. By Eq. (5.19), it follows that $\text{excess}(v) = c(s,v)$. Since the remaining vertice of the network do have zero excess zero, we can conclude that the non-negative excess constraint is met. Hence, f is initially a preflow, as desired. □

Next, we prove that f is a preflow in the course of the algorithm.

> **Lemma 5.10** f is a preflow in the course of the algorithm.

Proof. The proof is by induction on the number of operations (push or relabel) performed by the algorithm. By Lemma 5.9, f is initially a preflow (i.e., before applying any operation). In the inductive hypothesis, we assume that after the application of a certain number of operations, f is a preflow. We next show that f is a preflow once the next operation is applied. We consider two cases depending on the type of the next operation in the algorithm.

Relabel(f, h, v): The relabel operation does not have any affect to f. So, since f was a preflow before the application of the relabel operation, it follows that f is a preflow after application of it.

Push(f, h, v, w): The capacity constraint is clearly met after the application of this operation, since the algorithm never pushes more than the capacity or to negative (see Algorithm 5.6). It remains to show that the non-negative excess constraint is met for every node of the network. This is clear for all nodes different from v and w. Since the excess of node w is increased, the non-negative excess constraint is met for node w. Finally, the same holds for node v, because $\text{excess}(v)$ is an upper bound on the preflow pushed to w.

In summary, we have shown that for both containts of a preflow hold for f. So, f is a preflow throughout the algorithm. □

By Lemmata 5.7 and 5.10, it follows that f and h are compatible and that f is a preflow through the algorithm. If we prove that the algorithm terminates, then we know that the preflow at termination will be a flow and by Theorem 5.6 actually maximum flow. So, it remains to prove that the algorithm indeed terminates, which is the subject of the next section.

5.6.3   Time Complexity

In this section, we first prove that the preflow-push algorithm terminates and then we provide an estimation of its time complexity. For the former, we show that the number of push and relabel operations are rebounded by polynomials in n and m, i.e., in the number of nodes and arcs of the network.

Bounding the numbers of relabel operations: To bound the number of relabel operation we first need to prove two auxiliaries properties.

> Claim 5.1   If f and h are compatible and for a node v it holds $\text{excess}(v) > 0$, then there is a v-s path in G_f.

Proof. The claim trivially holds at the beginning of the algorithm, since the only nodes with positive excess are directly connected to s in G_f, i.e., in the residual network G_f there is an arc from each of these nodes to s. In order to prove that the claim holds, consider a node v with positive excess. Let U be the set of nodes for which there is a simple path from v to u in G_f. Clearly, set U and its compliment U^c form a partition of V, namely, $U \cup U^c = V$ and $U \cap U^c = \emptyset$.

For a contradiction, assume that $s \notin U$. It follows that node $s \in U^c$. We have:

$$
\begin{aligned}
\sum_{u \in U} \text{excess}(u) &= \sum_{u \in U} \left(\sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) \right) \\
&= \sum_{u \in U} \left(\sum_{v \in U} f(v, u) + \sum_{v \in U^c} f(v, u) - \sum_{v \in U} f(v, u) - \sum_{v \in U^c} f(v, u) \right) \\
&= \sum_{u \in U} \sum_{v \in U} f(v, u) + \sum_{u \in U} \sum_{v \in U^c} f(v, u) \\
&\quad - \sum_{u \in U} \sum_{v \in U} f(u, v) - \sum_{u \in U} \sum_{v \in U^c} f(u, v) \\
&= \sum_{u \in U} \sum_{v \in U^c} f(v, u) - \sum_{u \in U} \sum_{v \in U^c} f(u, v)
\end{aligned}
$$

Since set U contains node v for which $\text{excess}(v) > 0$ holds, since U does not contain s and since all other nodes have non-negative excess, it follows:

$$
\sum_{u \in U} \text{excess}(u) > 0.
$$

By the last equality from above and since $f(u, v) \geq 0$ for each arc (u, v), this implies:

$$
\sum_{u \in U} \sum_{v \in U^c} f(v, u) > 0.
$$

Hence, there is a $u' \in U$ and a node $v' \in U^c$ such that $f(v', u') > 0$. However, in this case the arc (u', v') belongs to G_f. So, node v' is reachable from node v in G_f. Hence, $v' \in U$. This is a contradiction to the fact that $v' \in U^c$. Hence, there is a v-s path in G_f, as desired. □

> **Claim 5.2** For each node v of G, it holds $h(v) \le 2n - 1$.

Proof. Since $h(s) = n$ and $h(t) = 0$, this claim clearly holds for the nodes s and t. Consider now a node $v \notin \{s, t\}$. When a relabel operation is performed at node v it holds that $\text{excess}(v) > 0$. Thus, by Claim 5.1 there is a v-s path $P : v \to \ldots \to v_2 \to v_1 \to s$ in G_f, as illustrated in Fig. 5.13.

Figure 5.13: v-s path in G_f

Since $h(s) = n$ and since f and h are compatible, for node v_1 it holds that $h(v_1) \le n + 1$. Similarly, for node v_2 it holds that $h(v_2) \le n + 2$. Since path path P has length at most $n - 1$, for node v it holds that $h(v_{n-1}) \le 2n - 1$. □

Initially, each node $v \in V \setminus \{s, t\}$ is such that $h(v) = 0$. With every relabel operation applied to v, the height of v gets increased by one. So, for a single node, the algorithm applies at most $2n - 1$ relabel operations. As the relabel operation can be applied to at most $n - 2$ nodes (i.e., nodes s and t are not relabeled), we can conclude that the number of relabel operations is $\mathcal{O}(n^2)$, as summarized in the following corollary.

> **Corollary 5.3** The number of relabel operations is $\mathcal{O}(n^2)$

Bounding the numbers of push operations To complete the proof that the algorithm terminates, we have to prove that the number of push operations is also bounded. To this end, we distinguish between two types of push operations.

> **Definition 5.16** A saturating (non-saturating) push from v to w uses all (does not use all, respectively) the residual capacity of (v, w).

With these definitions in mind, we can make the following observations.

> **Observation 5.4** A saturating push from v to w results in (v, w) being no longer present in G_f (since there is no residual capacity in arc (v, w)).

> **Observation 5.5**   A non-saturating push from v to w reduces excess(v) to zero.

With these observations in mind we can now bound the number of saturating pushes.

> **Lemma 5.11**   The number of saturating pushes is $\mathcal{O}(nm)$.

Proof. Consider a saturating push from v to w. It follows that $h(v) > h(w)$ holds. Since f and h are compatible, it further holds that $h(v) \leq h(w) + 1$. These two relations together implies that $h(v) = h(w) + 1$. By Observation 5.4, after the push from v to w is completed, the arc (v, w) will be no longer present in G_f. In order to apply a saturating push from v to w again, the arc (v, w) needs to appear in G_f. To this end, a push from w to v is needed, which implies that $h(w) > h(v)$ must hold. So, the algorithm needs at least two relabel operations at w so to apply another saturating push from v to w. By Claim 5.2, it follows that the number of saturating pushes from v to w is at most $n - 1$. Since the network has m arcs, the number of saturating pushes is $\mathcal{O}(nm)$ □

To prove that the number of non-saturated pushes is also bounded, we introduce the following additional rule:

Max-height rule: If there is a choice from which node v to push from, i.e, there are more than one nodes with positive excess, then choose the one with maximum height:

$$H = \max_{\text{excess}(v) > 0} h(v).$$

With this additional rule in mind, we make few observations.

> **Observation 5.6**   A push operation cannot increase the maximum height H (because it does not change the height of any node).

> **Observation 5.7**   A relabel operation can increase the maximum height H by one (if the node with maximum height is relabelled, which implies that its excess is positive).

The next two observations are slightly more difficult but also not so difficult to be proven.

> **Observation 5.8**   A saturating push cannot decrease the maximum height H.

Proof. Consider a saturating push from v to w which potentially decreases the maximum height H. Since H is decreased, node v must define the maximum height H, namely, $h(v) = H$;

see Fig. 5.14.

$$h(v) > h(w) \qquad \text{Push(f,h,v,w)} \qquad \text{excess}(v) > 0$$
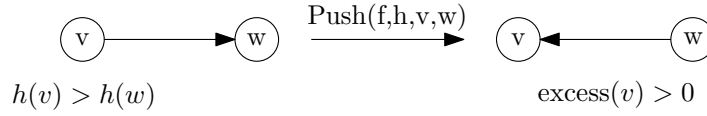
Figure 5.14: Illustration of a saturating push from v to w.

Since the push from v to w is saturating, it follows from Observation 5.4 that $\text{excess}(v) > 0$ holds once the saturating push from v to w is completed. This implies that node v has still positive excess and thus still defines the maximum height H. In other words, H does not decrease. $\qquad \square$

> **Observation 5.9** A non-saturating push can decrease the maximum height H by one.

Proof. Consider a non-saturating push from v to w, which results in decreasing the maximum height H. Since H is decreased, node v must define the maximum height H, namely, $h(v) = H$; see Fig. 5.15.

$$h(v) > h(w) \qquad \text{Push(f,h,v,w)} \qquad \text{excess}(v) = 0$$
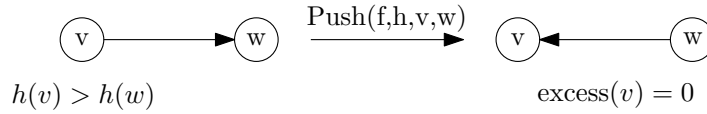
Figure 5.15: Illustration of a non-saturating push from v to w.

Since the push from v to w is non-saturating, it follows from Observation 5.5 that $\text{excess}(v) = 0$ holds once the non-saturating push from v to w is completed. Hence, H may indeed decrease but the decrement is at most one, since $h(v) = h(w) + 1$ holds and $\text{excess}(w) > 0$. Note that H will not decrease, if there is a tie, i.e., another node v' with $h(v') = H$. $\qquad \square$

Before we give a bound on the number of non-saturating pushes we prove an important property in the following claim.

> **Claim 5.3** The number of non-saturating pushes that can be performed consecutively without decreasing H is $\mathcal{O}(n)$.

Proof. As already mentioned, a non saturating push does not decrease H if and only if there is a tie, i.e., two nodes v and v' so that $h(v) = h(v') = H$. Since there can be at most $\mathcal{O}(n)$ ties, the claim follows. $\qquad \square$

We are now ready to give a bound on the number of non-saturating pushes.

Lemma 5.12   The number of non-saturated pushes is $\mathcal{O}(n^3)$

Proof. To prove the claim, we treat the maximum height H as a potential function and we ask how many times H can change. Recall that:

$$H = \max_{\text{excess}(v)>0} h(v).$$

To estimate how many times the maximum height H can change, we recall:

– By Observations 5.6 and 5.9, H can increase only by relabeling.

– By Corollary 5.3, H can increase at most $\mathcal{O}(n^2)$ times.

– Therefore, H can decrease at most $\mathcal{O}(n^2)$ times.

Hence, H can change $\mathcal{O}(n^2)$ times. By Claim 5.3, between two changes of H there is at most a linear number of non-saturated pushes. So, the number of non-saturated pushed is $\mathcal{O}(n^3)$, as desired.   □

Corollary 5.3, Lemma 5.11 and Lemma 5.12 imply that the maximum number of the operations performed by the push-relabel algorithm is in fact bounded by polynomials in n and m, as desired. Hence, the algorithm terminates. In the following theorem, we give few more insights on the time complexity of the preflow-push algorithm.

Theorem 5.7   The preflow-push algorithm computes a maximum flow in $\mathcal{O}(n^3)$ time.

Proof. The correctness of the algorithm follows from Theorem 5.6, Corollary 5.3, Lemma 5.11 and Lemma 5.12. For the time complexity, one can prove that:

– Each relabel operation can be implemented in $\mathcal{O}(n)$.

– Each push operation can be implemented in $\mathcal{O}(1)$.

– Selecting the applicable operation can be done in $\mathcal{O}(1)$.

Since by Corollary 5.3 the number of relabel operations is $\mathcal{O}(n^2)$ and since a single push operation can be implemented in $\mathcal{O}(n)$, the total cost of all relabel operations is $\mathcal{O}(n^3)$. Similarly, since by Lemmata 5.11 and 5.12 the number of push operations is $\mathcal{O}(n^3)$ and since a single push operation can be implemented in $\mathcal{O}(1)$, the total cost of all push operations is $\mathcal{O}(n^3)$. Therefore, the time complexity of the push-relabel algorithm is $\mathcal{O}(n^3)$, since selecting the applicable operation can be done in $\mathcal{O}(1)$.   □

We conclude this section by mentioning that the preflow-push algorithm has asymptotically faster running time than the algorithm by Edmonds and Karp. Recall that the time complexity of the algorithm by Edmonds and Karp is $\mathcal{O}(m^2 n)$, while the one of the preflow-push algorithm $\mathcal{O}(n^3)$; see Theorems 5.4 and 5.7.

Exercises

Exercise 5.1 Let $\mathcal{N} = (G, c, s, t)$ be a flow network such that $c : E \to \mathbb{N}^+$ is the capacity function of $\mathcal{N}$, s is the source of $\mathcal{N}$ and t is the target of $\mathcal{N}$. Let also f^* be a maximum flow of $\mathcal{N}$ (e.g., computed by the algorithm by Ford and Fulkerson).

(a) Describe an algorithm, which finds a minimum (s,t)-cut in $\mathcal{O}(|E|)$ time.

(b) Argue shortly about the correctness of your algorithm.

Exercise 5.2 The edge-connectivity of an undirected graph $G = (V, E)$ is defined as the minimum number of edges that must be removed from graph G such that the resulting graph G' is disconnected (i.e., there exist at least two vertices which are not connected by a path in G'). For example the edge-connectivity of a tree is one, while the edge-connectivity of a cycle is two. The goal of this exercise is to determine the edge-connectivity of an undirected graph $G = (V, E)$ using an algorithm for computing a maximum flow. To this end, this algorithm must be applied to a maximum of $|V|$ networks and each such network can have at most $\mathcal{O}(|V|)$ nodes and $\mathcal{O}(|E|)$ arcs.

(a) Describe an algorithm that calculates the edge-connectivity of an undirected graph according to the above specifications.

(b) Argue shortly about the correctness of your algorithm.

Exercise 5.3 A matching of an undirected graph G is a subset M of the edges of G such that for each vertex u of G there is at most one edge in M having u as an endpoint. The size of a matching is defined as the number of the edges it contains. A matching M of a graph G is called maximum if and only if its size is greater or equal to the size of every other matching of G, i.e, for every matching M' of G it holds that $|M| \geq |M'|$. In this exercise, we are interested in finding a maximum matching of a bipartite graph $G = (A, B, E)$, i.e., of a graph whose vertex set can be partitioned into two sets A and B such that $E \subseteq A \times B$. To this end, based on G we construct a flow network $\mathcal{N} = (H, c, s, t)$ such that:

– $H = (V_H, E_H)$

– $V_H = A \cup B \cup \{s, t\}$

– $E_H = \{(a, b) \in E; a \in A, b \in B\} \cup \{(s, a); a \in A\} \cup \{(b, t); b \in B\}$

– $c(e) = 1$, $\forall e \in E_H$

(a) Describe the network $\mathcal{N}$ corresponding to the complete bipartite graph $K_{3,3}$.

(b) Prove that if G has a matching of size k, then there exists a flow in $\mathcal{N}$ with value k.

(c) Prove that if there exists a maximum flow in $\mathcal{N}$ with value k, then G has a matching of size k.

Exercise 5.4 Let $\mathcal{N} = (G, c, s, t)$ be a flow network with the special property that all arcs have unit capacity, i.e., $c : E \to \{1\}$. Prove that in this special case Dinitz's algorithm requires $\mathcal{O}(nm)$ time.

Exercise 5.5 Let $f : E \to \mathbb{N}^+$ be a maximum integral flow in a flow network $\mathcal{N} = (G, c, s, t)$, such that $G = (V, E)$ is the underlying directed graph, $c : E \to \mathbb{N}^+$ is the (integral) capacity function of $\mathcal{N}$, s is the source of $\mathcal{N}$ and t is the target of $\mathcal{N}$.

Let $e_0 \in E$ be an arc of G and let $\mathcal{N}' = (G, c', s, t)$ be the flow network derived from $\mathcal{N}$ by increasing the capacity of arc e_0 by one, that is,

$$c'(e) = \begin{cases} c(e) + 1 & \text{if } e = e_0 \\ c(e) & \text{otherwise} \end{cases}$$

(a) Use the maximum-flow minimum-cut theorem to prove that the value of the maximum flow of $\mathcal{N}'$ is either $|f|$ or $|f| + 1$, where $|f|$ denotes the value of the maximum flow of $\mathcal{N}$.

(b) Present an $\mathcal{O}(n + m)$-time algorithm to find a maximum-flow of $\mathcal{N}'$, where n is the number of nodes in G and m is the number of arcs in G.

Further reading

– Chapter 26 from: T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009.

– Chapter 7 from: J. M. Kleinberg and É. Tardos. Algorithm design. Addison-Wesley, 2006.

– http://people.seas.harvard.edu/~cs125/fall14/lec19.pdf

– https://www2.cs.sfu.ca/CourseCentral/705/ramesh/maxflow3.pdf

– https://www2.cs.duke.edu/courses/fall15/compsci532/scribe_notes/lec02.pdf

DRAFT

6. Maximum Matchings

In this chapter, we will study the maximum matching problem, which given a graph, asks for a subset of the edges of it, called matching, such that no two edges of the matching share a common endvertex. If the input graph is unweighted (weighted), the objective is to compute a matching of maximum cardinality (of maximum total weight, respectively). In Section 6.1, we introduce basic concepts for coping with the maximum matching problem. In the remaining three sections, the focus is on algorithms for solving there different variants of maximum matching problem; (i) the maximum matching problem for bipartite graphs (Section 6.2), (ii) the maximum matching problem for general graphs (Section 6.3), and (iii) the maximum-weight matching problem for bipartite graphs (Section 6.4).

6.1   The maximum matching problem: Definitions and key concepts

In this section, we formally introduce definitions and key concepts that are central in the maximum matching problem. We start with the formal definition of a (maximum or maximum-weight) matching.

> **Definition 6.1** (Matching) Given a graph $G = (V, E)$, a subset M of the edge-set E of G is called matching if and only if there is no vertex v in V that is an endvertex of more than one edge in M, i.e,
>
> $$(u, v), (u', v') \in M \Rightarrow \{u, v\} \cap \{u', v'\} = \emptyset.$$

Example 6.1 Consider the following graph $G = (V, E)$ with $V = \{v_1, v_2, v_3, v_4\}$ and $E = \{(v_1, v_2), (v_2, v_3), (v_3, v_1), (v_1, v_4)\}$, that is, G is a 3-cycle plus an edge (see Fig. 6.1). Then, $M = \{(v_1, v_4), (v_2, v_3)\}$ is a matching of G.

Given a graph, the maximum matching problem asks for a so-called maximum matching, that
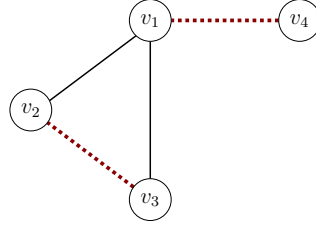
Figure 6.1: Illustration of a perfect matching (highlighted in red).

is, of a matching of maximum cardinality (formally defined in the following).

Definition 6.2 (Maximum matching) Given a graph $G = (V, E)$, a matching M of G is called maximum if and only if M is of maximum cardinality over all different matchings of G, i.e, for every matching M' of G, it holds that

$$|M| \geq |M'|.$$

A special type of a maximum matching is one that spans all vertices of the graph, e.g., as the one of Example 6.1. Such a matching is called perfect.

Definition 6.3 (Perfect matching) Given a graph $G = (V, E)$, a matching M of G is called perfect if and only if each vertex v in V is incident to an edge of M, that is, $|M| = \frac{n}{2}$.

Observation 6.1 A graph has a perfect matching, only if its order is even, that is, it has an even number of vertices.

Given an edge-weighted graph, the maximum-weight matching problem asks for a so-called maximum-weight matching, that is, of a matching whose total weight is maximized (formally defined in the following).

Definition 6.4 (Maximum-weight matching) Given an edge-weighted graph $G = (V, E, w)$, where $w : E \to \mathbb{R}^+$, a matching M of G is called maximum weighted if and only if the total weight of M is maximum over all different matchings of G, i.e, for every matching M' of G, it holds that

$$\sum_{e \in M} w(e) \geq \sum_{e \in M'} w(e).$$

A matching of a graph partitions its vertices into two sets; the set of matched vertices, which are spanned by the edges of the matching, and the set of free vertices, which are not spanned.
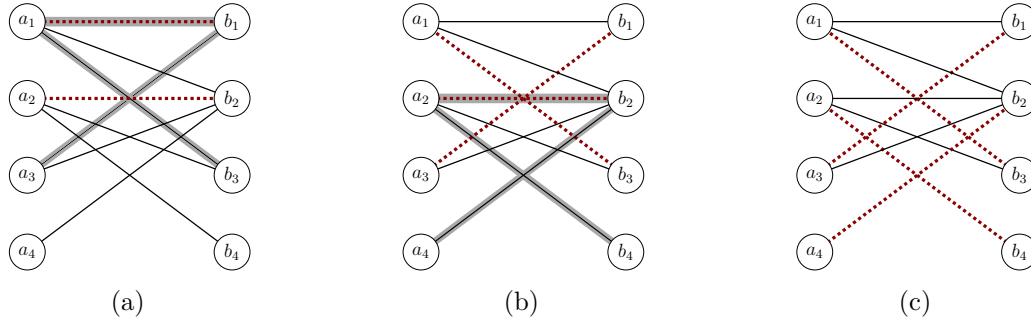
Figure 6.2: (a) A graph with a matching (dashed) and an augmenting path (highlighted in gray) with respect to this matching, and (b)-(c) the resulting matchings after two applications of Lemma 6.1.

Definition 6.5 (Free and matched vertices) Given a graph $G = (V, E)$ and a matching M of G, a vertex v in V is called free with respect to M if and only if there is no edge in M having v as an endvertex. Otherwise, v is called matched with respect to M.

A crucial component to compute maximum matchings is the so-called augmenting path, which is formally introduced in the following definition.

Definition 6.6 (Augmenting path) Given a graph $G = (V, E)$ and a matching M of G, an odd-length path $P = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{2k-1}, v_{2k})\}$ in G is called augmenting with respect to M if and only if path P satisfies the following properties:

P.1 P is simple,

P.2 the first vertex v_1 of P is free with respect to M,

P.3 the last vertex v_{2k} of P is free with respect to M, and

P.4 the edges of P alternate between $E \setminus M$ and M.

Property 6.1 Combining Properties P.1-P.4, we obtain:

$$\{(v_1, v_2), (v_3, v_4), \ldots, (v_{2k-1}, v_{2k})\} \subseteq E \setminus M, \quad \{(v_2, v_3), (v_4, v_5), \ldots, (v_{2k-2}, v_{2k-1})\} \subseteq M.$$

Definition 6.7 Given a graph $G = (V, E)$ and a matching M of G, a path satisfying Properties P.1 and P.4 of Definition 6.7 is called alternating with respect to M.

Example 6.2 Consider the bipartite graph $G = (V, E)$ with $V = \{v_1, v_2, \ldots, v_8\}$ and $E = \{(v_1, v_2), (v_1, v_4), (v_1, v_6), (v_3, v_4), (v_3, v_8), (v_5, v_2), (v_5, v_4), (v_7, v_4)\}$ together with matching $M = \{(v_1, v_2), (v_3, v_4)\}$ (see Fig. 6.2a). Then, $P = \{(v_5, v_2), (v_2, v_1), (v_1, v_6)\}$ is an augmenting path with respect to M.

Observation: If we remove from M the edge (v_1, v_2) and add to it the edges (v_5, v_2) and (v_1, v_6) of P, then we will obtain a new matching that is of greater cardinality than the one of M (see Fig. 6.2b) and of course we can repeat this procedure (see Fig. 6.2c). We formalize this idea in the following.

In the following definition we recall the notion of symmetric difference and then we demonstrate how one can use it to conclude that an already computed matching is not maximum.

Definition 6.8 (Symmetric difference) Given two finite sets A and B of elements, the symmetric difference $A \oplus B$ between A and B is defined as:

$$A \oplus B = (A \cup B) \setminus (A \cap B) = (A \setminus B) \cup (B \setminus A)$$

Property 6.2 For a finite set A, the following hold: (i) $A \oplus \emptyset = A$, (ii) $A \oplus A = \emptyset$.

Given a graph and a matching, we demonstrate in the following lemma how an augmenting path can be used to compute a new matching of higher cardinality.

Lemma 6.1 Let $G = (V, E)$ be a graph and let M be a matching of G. If P is an augmenting path with respect to M, then $M \oplus P$ is also a matching of G such that:

$$|M \oplus P| = |M| + 1.$$

Proof. Let $P = \{(v_1, v_2), \ldots, (v_{2k-1}, v_{2k})\}$. Since P is augmenting with respect to M, it follows that each of the edges in $\{(v_2, v_3), (v_4, v_5), \ldots, (v_{2k-2}, v_{2k-1})\}$ belongs to M. Additionally, each of the edges in $\{(v_1, v_2), (v_3, v_4), \ldots, (v_{2k-1}, v_{2k})\}$ does not belong to M. Since $M \oplus P = (M \setminus P) \cup (P \setminus M)$, it follows that $M \oplus P$ is obtained from M by removing the former edges (which are $k - 1$ in total) from it and by adding the latter ones (which are k in total) to it. Hence, $|M \oplus P| = |M| + 1$.

To prove that $M \oplus P$ is a matching, we first observe that v_1 and v_{2k} are free with respect to M, which implies that $M \oplus P$ can contain both edges (v_1, v_2) and (v_{2k-1}, v_{2k}). Next, we observe that each of the remaining vertices of P (that is, $v_2, \ldots, v_{2k-1}$) is matched both in M and in $M \oplus P$. Hence, if there exists a vertex which is incident to two edges of $M \oplus P$, then it must be inevitably incident to two edges in M as well; a contradiction.   □

6.2   The algorithm by Hopcroft and Karp

The algorithm by Hopcroft and Karp is computing a maximum matching of an input bipartite graph. The idea of the algorithm is rather simple. Start with an empty matching and as long

---

Algorithm 6.1: The core idea of the algorithm by Hopcroft and Karp.

Input : A bipartite graph G.
Output : A maximum matching of G.

1 Function HopcroftKarp(G)
2   Initialize M to $\emptyset$;
3   while there is an augmenting path P with respect to M do
4         $M \leftarrow M \oplus P$;
5   end
6   return M;
7 end

---

as there is an augmenting path, augment the matching as in Lemma 6.1 (see Algorithm 6.1). Before we formalize the algorithm, we first recall the definition of a bipartite graph.

> Definition 6.9 (Bipartite graph) A graph $G = (V, E)$ is called bipartite if and only if its vertex-set V can be partitioned into two subsets A and B (i.e., $V = A \cup B$ and $A \cap B = \emptyset$) such that for every edge (u, v) in E either $u \in A$ and $v \in B$, or $v \in A$ and $u \in B$ holds.

6.2.1 The general principles of the algorithm by Hopcroft and Karp

In the following, we introduce the main principles of the algorithm by Hopcroft and Karp (which are general and do not hold only for bipartite graphs). We start with an auxiliary lemma that will help us to prove that if one cannot find an augmenting path with respect to a matching, then this matching is maximum.

> Lemma 6.2 Let M and N be two matchings of a graph $G = (V, E)$ such that $|M| = \mu$, $|N| = \nu$ and $\nu > \mu$. Then, $M \oplus N$ contains at least $\nu - \mu$ vertex-disjoint augmenting paths with respect to M.

Proof. Let $\bar{G} = (V, M \oplus N)$. It follows that the degree of each vertex in $\bar{G}$ is at most 2, since each vertex in $\bar{G}$ is incident to edges in M and/or in N. This implies that, for each connected component C of $\bar{G}$, one of the following holds:

- C is an isolated vertex, or

- C is a cycle of even length, whose edges alternative between $M \setminus N$ and $N \setminus M$, or

- C is a path, whose edges alternative between $M \setminus N$ and $N \setminus M$.

Let $C_1, C_2, \ldots, C_q$ be the connected component of $\bar{G}$, where $C_i = (V_i, E_i)$ for $i = 1, 2, \ldots, q$. If we let $\delta(C_i) = |E_i \cap N| - |E_i \cap M|$, then it is not difficult to see that $\delta(C_i) \in \{-1, 0, 1\}$. In particular,

$\delta(C_i) = 1$ implies that C_i is an alternating path with respect to M. In the following, we are summing up over i:

$$\sum_{i=1}^{q} \delta(C_i) = \sum_{i=1}^{q} (|E_i \cap N| - |E_i \cap M|) = |N \setminus M| - |M \setminus N| = \nu - \mu.$$

If follows that there exists at least $\nu - \mu$ connected components in $\bar{G}$ whose δ-value is 1. Hence, G has at least $\nu - \mu$ vertex-disjoint augmenting paths with respect to M, as desired.   □

Corollary 6.1 (Berge) A matching M of a graph G is maximum if and only if there is no augmenting path in G with respect to M.

Corollary 6.1 implies that the termination condition of Algorithm 6.1 is reasonable. The following lemma focuses on providing an upper bound of the length of an augmenting path.

Lemma 6.3 Let M and N be two matchings of a graph $G = (V, E)$ such that $|M| = \mu$, $|N| = \nu$ and $\nu > \mu$. If N is a maximum matching, then there exists an augmenting path P with respect to M whose length $|P|$ is such that

$$|P| \leq 2 \cdot \lfloor \frac{\mu}{\nu - \mu} \rfloor + 1.$$

Proof. From Lemma 6.2, $M \oplus N$ has $\nu - \mu$ vertex-disjoint augmenting paths with respect to M (and thus also edge-disjoint). Since $|M| = \mu$, all these paths contain at most μ edges of M. It follows that there exists one augmenting path P containing at most $\lfloor \frac{\mu}{\nu-\mu} \rfloor$ edges of M. Hence, the length of P is at least $2 \cdot \lfloor \frac{\mu}{\nu-\mu} \rfloor + 1$.   □

Lemma 6.4 Let M be a matching of a graph $G = (V, E)$. Let also P be a shortest augmenting path with respect to M and P' be any augmenting path with respect to $M \oplus P$. Then:
$$|P'| \geq |P| + |P \cap P'|.$$

Proof. Let $N = M \oplus P \oplus P'$. By Lemma 6.1, it follows that N is a matching with $|N| = |M| + 2$. By Lemma 6.2, $M \oplus N$ contains (at least) two vertex-disjoint augmenting paths P_1 and P_2 with respect to M. Hence:

$$|M \oplus N| \geq |P_1| + |P_2| \tag{6.1}$$

Since P is a shortest augmenting path with respect to M, it follows that:

$$|P_1| \geq |P|, \quad |P_2| \geq |P| \tag{6.2}$$

Hence, combining Eqs. (6.1) and (6.2), we obtain:

$$|M \oplus N| \geq 2|P| \tag{6.3}$$

By definition of matching N, we have $M \oplus N = M \oplus M \oplus P \oplus P'$. Hence, by Property 6.2, $M \oplus N = P \oplus P'$ follows. Thus, by Eq. (6.3) we obtain:

$$|P \oplus P'| \geq 2|P| \tag{6.4}$$

Since by definition $P \oplus P' = (P \cup P') \setminus (P \cap P')$ holds, we can rewrite Eq. (6.4) as follows:

$$|P| + |P'| - |P \cap P'| \geq 2|P|$$

Thus, we get $|P'| \geq |P| + |P \cap P'|$, as desired. $\qquad\square$

We now slightly adjust Algorithm 6.1 as follows. The algorithm starts with an empty matching M_0. In subsequent steps, it computes a sequence of matchings $M_1, M_2, \ldots, M_i, M_{i+1}, \ldots, M_s$, such that P_i is a shortest augmenting path with respect to M_i and $M_{i+1} = M_i \oplus P_i$, where s is the cartinality of a maximum matching of the input graph G. In this regard, the following two lemmas are immediate corollaries of Lemma 6.4.

> **Lemma 6.5** For any $0 \leq i < s - 1$, it holds that $|P_{i+1}| \geq |P_i|$.

Proof. From Lemma 6.4, it follows that $|P_{i+1}| \geq |P_i| + |P_i \cap P_{i+1}| \geq |P_i|$. $\qquad\square$

> **Lemma 6.6** For any $0 \leq i < j < s - 1$, if $|P_i| = |P_j|$, then P_i and P_j are vertex-disjoint.

Proof. Assume to the contrary that there exists i, j with $0 \leq i < j < s - 1$, such that $|P_i| = |P_j|$ and P_i and P_j are not vertex-disjoint. Without loss of generality, we may further assume there is no k with $i < k < j$ such that P_k shares a vertex with P_i. Under this assumption, P_j is an augmenting path with respect to $M_i \oplus P_i$. By Lemma 6.5, it follows that for any $i < k < j$, P_k has the same length as P_i (and thus as P_j), which implies that P_i and P_k are vertex-disjoint and thus edge-disjoint.

By Lemma 6.4, we know that $|P_j| \geq |P_i| + |P_i \cap P_j|$. Since by assumption $|P_i| = |P_j|$ holds, it follows that $|P_i \cap P_j| = 0$, which means that P_i and P_j are edge-disjoint.

Let v be a common vertex of P_i and P_j. Since v belongs to P_i, there is an edge incident to v in M_{i+1}. Call it (u, v). Since v is a vertex of P_i and since (u, v) belongs to M_{i+1}, it follows that (u, v) is an edge of P_i. We now claim that (u, v) is also an edge of P_j, which is the desired contradiction, since P_i and P_j are edge-disjoint. Since for any $i < k < j$, P_i and P_k are edge-disjoint, edge (u, v) belongs to matching M_j. Since v belongs to P_j and since (u, v) belongs to M_j, it follows that (u, v) indeed is an edge of P_j, as claimed. This completes the proof. $\qquad\square$

> **Corollary 6.2** If $0 \leq i < j < s - 1$ and $|P_i| = |P_j|$, then P_j is a shortest augmenting path with respect to M_i.

> **Lemma 6.7** The distinct integers in the sequence $|P_0|, |P_1|, \ldots, |P_s|$ are no more than
> $$2\lfloor \sqrt{s} \rfloor + 2.$$

Proof. Let $r = \lfloor s - \sqrt{s} \rfloor < s$. Then, the cardinality of matching M_r is r. We next provide an upper bound on the length of the augmenting path P_r as follows:

$$
\begin{aligned}
|P_r| &\leq 2\lfloor \frac{r}{s-r} \rfloor + 1 && \text{by Lemma 6.3} \\
&\leq 2\frac{r}{s-r} + 1 && \text{since } \lfloor x \rfloor \leq x, \ \forall x \in \mathbb{R} \\
&\leq 2\frac{\lfloor s - \sqrt{s} \rfloor}{s - (\lfloor s - \sqrt{s} \rfloor)} + 1 && \text{since } r = \lfloor s - \sqrt{s} \rfloor \\
&\leq 2\frac{s - \sqrt{s}}{\sqrt{s}} + 1 && \text{since } s - \lfloor s - \sqrt{s} \rfloor \geq s - (s - \sqrt{s}) \\
&\leq 2(\sqrt{s} - 1) + 1 \\
&= 2\sqrt{s} - 1 \\
&\leq 2\lfloor \sqrt{s} \rfloor + 1 && \text{since } x \leq \lfloor x \rfloor + 1, \ \forall x \in \mathbb{R}
\end{aligned}
$$

For every $i < r$, we know from Lemma 6.5 that $|P_i| \leq |P_r| \leq 2\lfloor \sqrt{s} \rfloor + 1$. Since each augmenting path has odd length, $|P_i|$ must be one of the $\lfloor \sqrt{s} \rfloor + 1$ odd integers that are less than or equal to $2\lfloor \sqrt{s} \rfloor + 1$. For $r + 1 \leq i \leq s$, we observe that $|P_i|$ corresponds to the length of one of the paths in $\{P_{r+1}, \ldots, P_s\}$, which are $s - r = \lceil \sqrt{s} \rceil$ in total. Hence, the distinct integers in the sequence $|P_0|, |P_1|, \ldots, |P_s|$ are no more than

$$\lfloor \sqrt{s} \rfloor + 1 + \lceil \sqrt{s} \rceil \leq 2\lfloor \sqrt{s} \rfloor + 2. \qquad \square$$

By Lemma 6.5, Corollary 6.2 and Lemma 6.7, we can conclude that in order to compute $M_1, M_2, \ldots, M_s$ we can consider at most $2\lfloor \sqrt{s} \rfloor + 2$ phases within of which all augmenting paths found (i) are vertex-disjoint, (ii) have the same length, and (iii) it is not possible to find another augmenting path with the same length (see Algorithm 6.2).

The main difference between the algorithm-sketch that we provided at the begging of this section (see Algorithm 6.1) and the algorithm that we just introduced (see Algorithm 6.2) lies mainly on the number of shortest augmenting paths that are used at each iteration. In particular, at each iteration of Algorithm 6.2, the matching is augmented by a maximal set of vertex-disjoint augmenting paths of same length instead of just one in Algorithm 6.1.

Algorithm 6.2: The algorithm by Hopcroft and Karp

    Input   : A bipartite graph $G = \{V, E\}$

    Output : A maximum matching of G

1 Function HopcroftKarp(G)

2     $M = \emptyset$

3   repeat

4         $\ell(M) \leftarrow$ the length of a shortest augmenting path with respect to M;

5   Find a maximal set of vertex-disjoint paths $Q_1, Q_2, \ldots, Q_t$ such that:

6   − $|Q_i| = \ell(M)$;   // See Algorithm 6.3

7   − Q_i is an augmenting path with respect to M;

8         $M \leftarrow M \oplus Q_1 \oplus \ldots \oplus Q_t$;

9   until $S = \emptyset$;

10   return M;

11 end

The question that remains to be answered is how to compute the augmenting paths of each phase. An answer to this question is given in the following section for the case where the input graph is bipartite.

6.2.2 Adjusting for bipartite graphs

In this section, we assume that the input graph $G = (V, E)$ is a bipartite graph, i.e., its vertex-set V is partitioned into two subsets A and B such that $E \subseteq A \times B$. Under this assumption, we proceed as follows. Assume that we have performed a particular number of iterations (that is, phases) of Algorithm 6.2 and let M be the matching for graph G that we have computed so far. Of course, M is not necessarily maximum.

To compute a maximal set of vertex-disjoint augmenting paths for the next iteration of Algorithm 6.2, we first observe that the augmenting paths of G with respect to M start from a free vertex of A and end at a free vertex of B (or vice versa), since G is bipartite and augmenting paths are of odd-length by definition. Next, we direct the non-matching edges of G from A to B, and the matched ones from B to A (see Fig. 6.3a). In this way, we obtain a new directed graph, which we denote by $\hat{G} = (V, \hat{E})$. Then, we assign (potentially a subset of) the vertices of $\hat{G}$ to layers $L_1, L_2, \ldots, L_k$ (see Fig. 6.3b) such that:

- $L_1 = \{v \in A : v$ is free$\}$, i.e, L_1 contains only free vertices in A

- $L_i = \{v \in V : v \notin L_1 \cup L_2 \cup \ldots \cup L_{i-1}, u \in L_{i-1}$ and $\exists (u, v) \in \hat{E}\}$

- $k = \min\{i : L_i$ contains a free vertex of B and $i \mod 2 \equiv 0\}$.

Clearly, the layering of $\hat{G}$ can be constructed in $\mathcal{O}(n + m)$ time by applying BFS to $\hat{G}$. The following properties are immediate implications of the constructed layering.
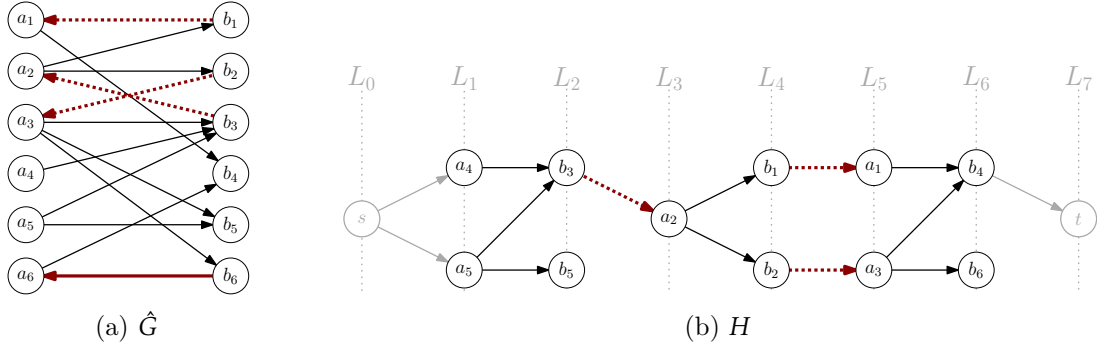
Figure 6.3: (a) A bipartite graph with a matching (dashed), which has been directed, and (b) the obtained s-t layered network of it.

> **Property 6.3** $L_1 \cup L_3 \cup \dots L_{k-1} \subseteq A$ and $L_2 \cup L_4 \cup \dots L_k \subseteq B$.

> **Property 6.4** If (u,v) is an edge of the layering of $\hat{G}$, then $u \in L_i$ and $v \in L_{i+1}$ for some $1 \le i < k$.

> **Property 6.5** The shortest augmenting paths with respect to M are in one-to-one correspondence with the paths in the layering of G that start at a free vertex in L_1 (i.e, in A) and end at a free vertex in L_k (i.e, in B).

> **Property 6.6** The length of each path from a free vertex in L_1 to a free vertex in L_k is $k-1$.

Note that the layering of $\hat{G}$ may have several free vertices in A (which reside in L_1) and several free vertices in B (which reside in L_k). For convenience, we introduce two vertices s and t, and we connect s to all free vertices in L_1 and all the free vertices in L_k to t (refer to the gray colored vertices in Fig. 6.3b). We refer to the obtained graph as the s-t layered network of G with respect to M, and we denote it by H.

In order to compute a maximal set of vertex-disjoint augmenting paths with respect to M, Hopcroft and Karp's idea is to repeat the following procedure as long as a path from s to t in H is possible to be found (see also Algorithm 6.3):

"Find a path P from s to t in H (if any) and add $P \backslash \{s,t\}$ to the set of vertex-disjoint augmenting paths with respect to M. Then, remove all vertices of $P \setminus \{s,t\}$ from H and repeat."

More precisely, this is achieved by continuously applying one of the two operations described below, starting at the source s:

Algorithm 6.3: The single phase of the algorithm by Hopcroft and Karp.

    Input   : The s-t layered network H of G with respect to M.
    Output: A maximal set of equal-length vertex-disjoint augmenting paths w.r.t. M.

1 <u>Initialize</u>: $S \leftarrow \emptyset$, $P \leftarrow \{s\}$, $v \leftarrow s$, Go to Advance;

2 <u>Advance</u>:
3 if $\nexists\, e = (v, w)$ then
4 | Go to Retreat;
5 else
6 | $e \leftarrow (v, w)$;
7 | $P \leftarrow P \cup \{w\}$;
8 | $v \leftarrow w$;
9 | if $v \neq t$ then
10 | | Go to Advance;
11 | else
12 | | $S \leftarrow S \cup P \setminus \{s, t\}$;
13 | | Delete the first and the last edge of P from H;
14 | | foreach vertex v of $P \setminus \{s, t\}$ do
15 | | | Delete all edges incident to v from H;
16 | | end
17 | | Go to Initialize;
18 | end
19 end

20 <u>Retreat</u>:
21 if $v = s$ then
22 | Return S;
23 else
24 | $(u, v) \leftarrow$ the last edge of P;
25 | Delete v from P and (u, v) from H;
26 | $v \leftarrow u$;
27 | Go to Advance;
28 end

- Advance: Follow an edge of the current vertex in H (if any).

- Retreat: If there is no outgoing edge at the current vertex, go back along the edge that lead to the current vertex, and delete it.

Once t is reached, an augmenting path $P \setminus \{s, t\}$ has been found, $P \setminus \{s, t\}$ is added to the set of vertex-disjoint augmenting paths, all edges incident to each vertex of P are removed from H, and repeat from the source. The procedure is terminated once a search retreats all the way back to s. Note that this procedure is very similar to the corresponding one in Dinitz's algorithm to find a blocking flow (see Algorithm 5.4).
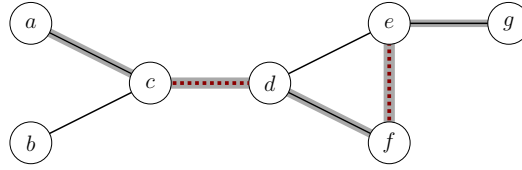
Figure 6.4: A non-bipartite graph with a matching (dashed).

> **Theorem 6.1** The algorithm by Hopcroft and Karp computes a maximum matching of a bipartite graph in $\mathcal{O}(n^{5/2})$ time.

Proof. The correctness of the algorithm follows from Property 6.5 and the correctness of Algorithm 6.1, which has already been shown. Since the construction of the s-t layered network can be done in $\mathcal{O}(n + m)$ time and since a single execution of Algorithm 6.3 needs $\mathcal{O}(n+m)$ time, it follows that in total the algorithm by Hopcroft and Karp needs $\mathcal{O}(\sqrt{s}(m+n))$ Since $s = \mathcal{O}(n)$ and since in the worst case $m = \mathcal{O}(n^2)$, the proof of the theorem follows. □

Note that, as described in Exercise 5.3, one can compute a maximum matching of a bipartite graph by solving a flow problem on a network that is asymptotically of same size. Since the value of the maximum flow corresponds to the cardinality of the maximum matching in this approach, the flow problem (and thus the matching problem) can be solved in $\mathcal{O}(n^3)$ time, e.g., by using the algorithm by Ford and Fulkerson (see Theorem 5.3). Hence, the algorithm described in this section forms an improvement by a factor of $\mathcal{O}(\sqrt{n})$.

We conclude this section by mentioning that Algorithm 6.3 is simply an adjustment of DFS. We also mention that if the input graph is not bipartite, then Algorithm 6.3 may fail to compute an augmenting path. An example is illustrated in Fig. 6.4, which contains a single augmenting path with respect to the depicted matching from vertex a (highlighted in gray). In particular, if DFS visits the vertices of the graph by following the path $a \to c \to d \to f \to e \to g$, then the augmenting path will be found. However, if after d, DFS visits first e, then the augmenting path will not be found.

6.3   The blossom shrinking algorithm by Edmonds

To generalize the main principles of the algorithm by Hopcroft and Karp to general (i.e., non-bipartite) graphs, Edmonds' algorithm identifies special substructures (like to one of Fig. 6.4) in the input graph and contracts each of them to a single vertex. To formalize this concept, however, we need to introduce a few concepts.

Definition 6.10 (Flower) A flower in a graph $G = (V, E)$ with respect to a matching M and a free vertex r, called root, is a subgraph of G consisting of two components:

- a stem: that is, a simple even-length alternating path starting at r and terminating at a vertex w, called base ($r = w$ is possible),

- a blossom: that is, a simple odd-length alternating cycle starting and terminating at w such that w is the only common vertex with the stem.

Equivalently, we refer to such a flower as a blossom B rooted at r and based at w.
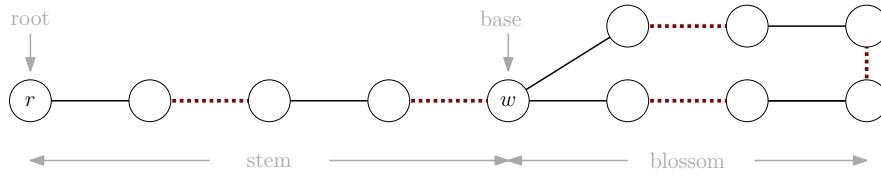
Figure 6.5: Illustration of a flower; the edges of the matching are dashed.

The following properties directly follow from Definition 6.10.

Property 6.7 The stem contains $2\ell + 1$ vertices and ℓ edges in M, for some $\ell \geq 0$.

Property 6.8 The blossom contains $2k + 1$ vertices and k edges in M, for some $k \geq 1$.

Property 6.9 Every vertex v in the blossom that is different from its base, that is, $v \neq w$, is reachable from w (and thus from r) through two distinct alternating paths; one with even length and one with odd length; the former path terminates with a matching edge, while the latter terminates with a non-matching edge.

We next define the operation of shrinking a blossom (see Fig. 6.6) and prove an important property of this operation, i.e., a graph contains an augmenting path with respect to a matching if and only if the graph obtained by shrinking a blossom contains an augmenting path.

Definition 6.11 (Blossom shrinking) Let $G = (V, E)$ be a graph with a blossom B rooted at r and based at w. The graph $G' = (V', E')$ obtained by shrinking B is such that:

- $V' = V \setminus V(B) \cup \{b\}$, and

- $E' = E \setminus E(B) \cup \{(b, v) : (u, v) \in E, u \in V(B), v \notin V(B)\}$

In other words, G' is derived by contracting B to a single vertex b, called pseudo-vertex.
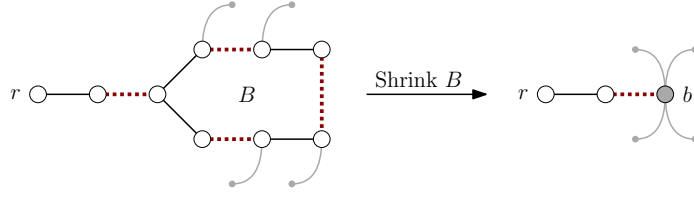
Figure 6.6: Illustration of the operation of blossom shrinking.

(a)

(b)
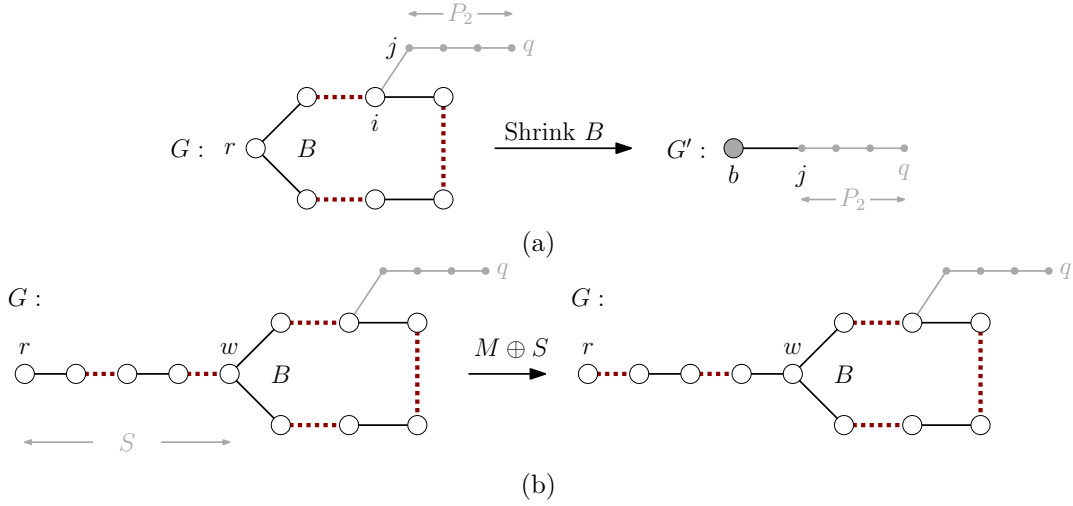
Figure 6.7: Illustrations for the proof of Lemma 6.8.

> **Lemma 6.8** Let G be a graph with a matching M and a blossom B rooted at r and based at w. Let G' be the graph obtained by shrinking B, and M' be the restriction of M in G', i.e., $M' = M \setminus E(B)$. If G contains an augmenting path P with respect to M from r to q, then G' contains an augmenting path P' with respect to matching M' from r to q.

Proof. If P does not contain any pseudo-vertex of G', then clearly $P' = P$ is an augmenting path with respect to M' from r to q. Hence, we may assume without loss of generality that P contains (at least one) pseudo-vertex b of G'. We proceed by distinguishing two cases.

We first consider the case, in which r is a pseudo-vertex in G', i.e., $r = b$. Assuming that P starts at r and ends at q, let i be the last vertex of B in P, and let j be its successor in P ($j = q$ is possible; see Fig. 6.7a). We may assume that P is the results of concatenating two paths P_1 and P_2, where $P_1 = r \to \ldots \to i$ and $P_2 = j \to \ldots \to q$. Since i is incident to an edge of M in B, (i, j) does not belong to M. Hence, $P' = b \to j \to P_2$ is an augmenting path with respect to M'.

To complete the proof consider now the case, in which r is not a pseudo-vertex in G', i.e., $r \neq b$. W.l.o.g., we may assume that r and q are the only free vertices in G. Let S be the stem of blossom B in G, i.e., S is an even-length alternating part from r to w; recall that w is the base of B. Let $N = M \oplus S$ and let N' be the restriction of N in G'. Since S is of even length,

$|M| = |N|$ holds (see Fig. 6.7b). Hence, G has an augmenting path with respect to N, which has to be between w and q, because w and q are the only free vertices in G with respect to N. Since w is matched in M, it follows that w is not matched in N (thus, pseudo-vertex b is also unmatched in N). Hence, the first case of the lemma applies for G and N. $\square$

Lemma 6.9 Let G be a graph with a matching M and a blossom B rooted at r and based at w. Let G' be the graph obtained by shrinking B, and M' be the restriction of M in G', i.e., $M' = M \setminus E(B)$. If G' contains an augmenting path P' with respect to M' from r to q, then G contains an augmenting path P with respect to matching M from r to q.

Proof. If P' does not contain any pseudo-vertex of G', then clearly $P = P'$ is an augmenting path with respect to M from r to q. Hence, we may assume without loss of generality that P' contains (at least one) pseudo-vertex b of G'. We proceed by distinguishing two cases.

We first consider the case in which neither r nor q is a pseudo-vertex in G', i.e., $r \neq b$ and $q \neq b$. Assuming that P' starts at r and ends at q, let i and j be the predecessor and the successor of b in P', respectively. Without loss of generality, we assume that (i, b) belongs to M', which implies that (b, j) does not belong to M'. Note that this assumption is indeed without loss of generality, since the roles of the free vertices r and q in P' are interchangeable. We denote by P'_1 and by P'_2 the subpaths of P' from r to i and from j to q, respectively, that is, $P'_1 = r \to \ldots \to i$ and $P'_2 = j \to \ldots \to q$. The fact that the edge (b, j) does not belong to M' implies that there is a vertex k in blossom B in G such that (k, j) is an edge of G. If k is identified with the base w of blossom B (that is, $w = k$), then $P = P'_1 \to w \to P'_2$ is an augmenting path in G with respect to M. Hence, we may assume that k is not identified with the base w of blossom B (that is, $w \neq k$). Let P_e be the even-length path from w to k in B. By Property 6.9, P_e ends with an edge in M. Therefore, $P = P'_1 \to w \to P_e \to P'_2$ is an augmenting path of G with respect to M.

To complete the proof consider now the case, in which r is a pseudo-vertex in G', i.e., $r = b$. Let j be the successor of b in P', and let P'_2 be the subpath of P' from j to q. It follows that (b, j) does not belong to M'. Since (b, j) belongs to G', there is a vertex i of blossom B in G, such that (i, j) is an edge of G. Let P_e be the even-length path in B from the base w of B to vertex j. By Property 6.9, it follows that P_e ends with an edge in M. Therefore, $P = P_e \to P'_2$ is an augmenting path with respect to M in G. $\square$

Corollary 6.3 Let G be a graph with a matching M. Then, one of the following holds:

- M is maximum,

- M contains an augmenting path,

- M contains a blossom.

(a)

(b)
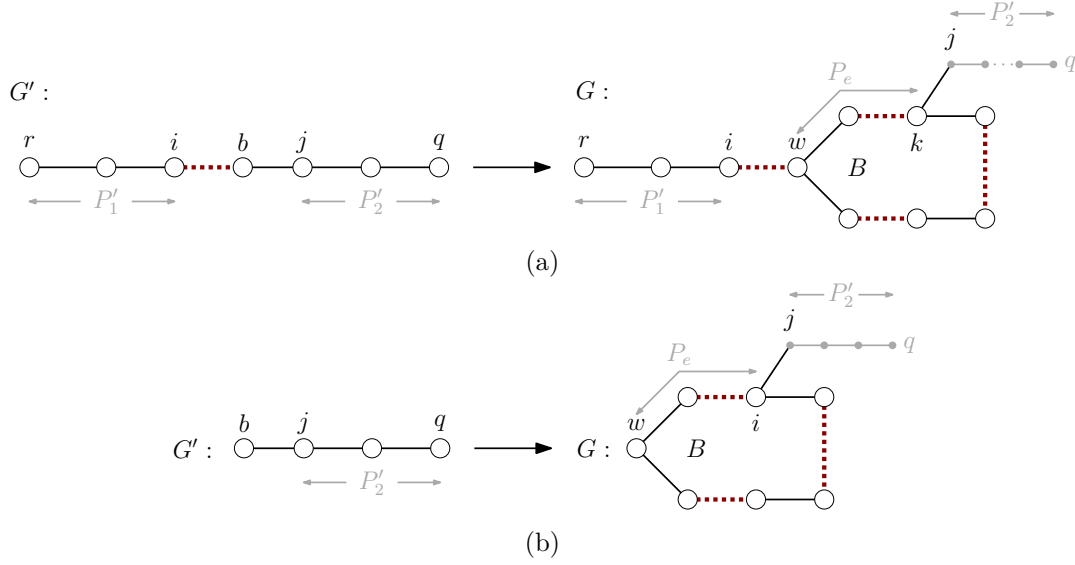
Figure 6.8: Illustrations for the proof of Lemma 6.8.

Corollary 6.3 implies a very natural extension of Algorithm 6.1; see Algorithm 6.4. The correctness of this extension directly follows from Corollary 6.3, assuming (i) the existence of an algorithm to determine whether there exists a blossom to be shrunk or an augmenting path with respect to the already computed matching (or none of the two, in which case we can safely conclude that the already computed matching is maximum), and (ii) an efficient implementation of the blossom shrinking and blossom expansion operations.

The latter can be clearly implemented in time linear in the size of each found blossom, assuming that each blossom is stored, e.g., as a doubly connected list, and that each pseudo-vertex maintains a reference-pointer towards its associated blossom. For the former refer to Algorithm 6.5, where we describe the main ingredients needed to determine either the existence of a blossom or of an augmenting path.

Algorithm 6.5 classifies each of the vertices of the input graph G into two categories (called "even" and "odd"; see also Fig. 6.9) based on the parity of its distance from a free vertex in G. This information is stored as a label "[r, odd]" or "[r, even]" at each vertex, where r is the reference free vertex, and "odd" or "even" is the corresponding category. The classification is computed by traversing the vertices of G in a BFS-like way, starting from all free vertices of G.

In particular, each free vertex r of G is labeled as "[r, even]" and is stored in a priority queue Q. The algorithm maintains two invariants. First, Q contains only even-labeled vertices. Second, each even-labeled (odd-labeled) vertex has been explored by a vertex that is already labeled as odd (even) via an edge that has been marked as examined and does not belong to M (belongs to M, respectively). We store the information that a vertex x has been explored by a vertex y in the traversal by using a pointer $\pi(x)$ that points to y.

Assume now that the algorithm has performed some steps maintaining the aforementioned invariants, and neither a blossom nor an augmenting path has been detected. In the next

---

Algorithm 6.4: The blossom shrinking algorithm by Edmonds.

Input : A graph G.
Output : A maximum matching of G.

1 Function Edmonds(G)
2 | Initialize M to $\emptyset$;
3 | repeat
4 | | Start a parallel search from all free vertices of G;
5 | | if a blossom B is encountered then
6 | | | Add B to $\mathcal{B}$;
7 | | | Shrink B to b;
8 | | | Start a new search in the derived graph;
9 | | end
10 | | if an augmenting path P is encountered then
11 | | | Expand all blossoms in $\mathcal{B}$;
12 | | | Update P accordingly;
13 | | | Delete all blossoms from $\mathcal{B}$;
14 | | | $M \leftarrow M \oplus P$;
15 | | end
16 | until there is no augmenting path with respect to M;
17 | return M;
18 end

---

step, a vertex x is extracted from Q. If this is not possible, then M is maximum (as neither a blossom nor an augmenting path was found). Otherwise, by the first invariant of the algorithm, x is labeled as $[r, even]$, where r is the reference free vertex for x. Note that $r = x$ is possible. Similarly to BFS, the traversal continues by exploring all vertices that are neighboring x. Let y be such a vertex. If the edge (x, y) is marked as examined (i.e., x was reached from y in some previous step of the algorithm), then we ignore y and proceed with the next neighbor of x. Otherwise, we mark the edge (x, y) as examined and we distinguish few cases.

– If y is unlabeled (i.e., y has not been explored in the traversal so far), then we know that y is not free (since all free vertices have been labeled at the beginning of the algorithm), which implies that y is incident to a vertex z such that the edge (y, z) belongs to M. Since y was explored from x, whose label is even, y is labeled as "[r, odd]", i.e., its reference free vertex is the one of x, while the parity of its distance to r is odd (that is, different from the one of x). Similarly, vertex z is labeled as "[r, even]" and is stored in Q, in order to satisfy the first invariant of the algorithm.

– If y is labeled as "[r, even]", then we claim that a blossom B has been found(refer to the green edges in Fig. 6.9). To see this, recall that the distance of x to the free vertex r is even. Of course, the same holds for vertex y, since it has the same label. The blossom is then formed in the presence of the edge (x, y). Namely, by the second invariant of the algorithm, there are two alternating paths P_1 and P_2 from x and from y to the free vertex r. Let w be a common vertex in these two paths ($r = w$ is possible). Then, blossom B is rooted at r, is based at w and it can be compute in time linear to the size of B using the π pointers.

Algorithm 6.5: The algorithm by Edmonds to detect blossoms and augmenting paths.

   Input   : A graph $G = \{V, E\}$ and a matching M.
   Output : Either an augmenting path with respect to M or a blossom.

1 $Q \leftarrow \emptyset$;
2 foreach free vertex r in G do
3 | label r as [r, even];
4 | add v to Q;
5 | $\pi(r) \leftarrow$ null;
6 end
7 while Q is not empty do
8 | $x \leftarrow Q.\text{extract}()$;
9 | Let [r, even] be the label of x;
10 | foreach neighbor y of x do
11 | | if (x, y) is examined then
12 | | | continue;
13 | | else
14 | | mark (x, y) examined;
15 | | if y is unlabeled then
16 | | | $z \leftarrow$ the vertex matched to y in M;
17 | | | label y as [r, odd]; $\pi(y) \leftarrow x$;
18 | | | label z as [r, even]; $\pi(z) \leftarrow y$;
19 | | | add z to Q;
20 | | end
21 | | if y is labeled [r, even] then
22 | | | $B \leftarrow$ blossom found;   // Compute it using π-pointers.
23 | | | return B;
24 | | end
25 | | if y is labeled [ρ, even] with $r \neq \rho$ then
26 | | | $P \leftarrow$ augmenting path found;   // Compute it using π-pointers.
27 | | | return P;
28 | | end
29 | end
30 | end
31 end
32 Report that there is no augmenting path;
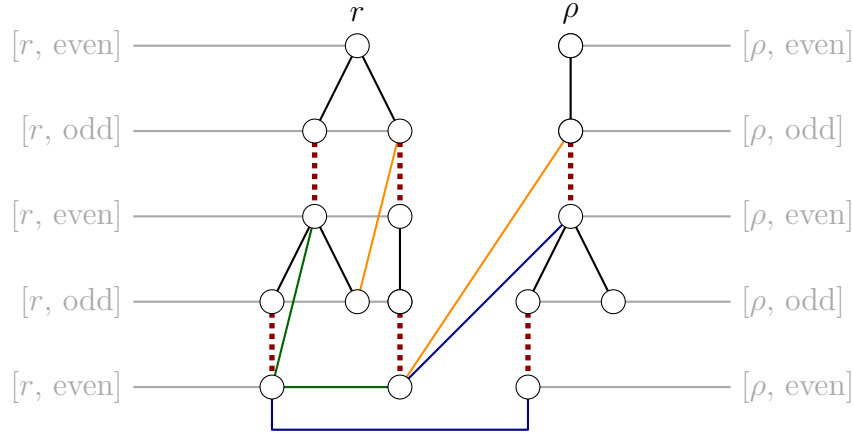
Figure 6.9: Illustration of different cases that may occur in the labeling procedure.

– If y is labeled as "$[\rho, \text{even}]$" with $\rho \neq r$, then we claim that an augmenting path P has been found(refer to the blue edges in Fig. 6.9). The proof of the claim is similar as above. Since x is labeled as "[r, even]", by the second invariant of the algorithm there is an alternating path P_1 from x to the free vertex r. Similarly, since y is labeled as "$[\rho, \text{even}]$", by the second invariant of the algorithm there is an alternating path P_2 from y to the free vertex ρ. Since the edges incident to x and y in P_1 and P_2, respectively, belong to M (by the second invariant of the algorithm), it follows that the concatenation of P_1 and P_2 is an alternating path from r to ρ. Since r and ρ are both free, this path is in fact an augmenting path, proving the claim.

– If y is odd-labeled, then by the invariants of the algorithm neither an augmenting path nor a blossom can be found, since x is even-labeled(refer to the orange edges in Fig. 6.9). So, in this case no further action is taken.

Theorem   6.2   The blossom shrinking algorithm by Edmonds computes a maximum matching of a bipartite graph in $\mathcal{O}(nm)$ time.

Proof. The correctness of the algorithm follows from Corollary 6.3 and from the correctness of Algorithm 6.5, which follows from the discussion above. To analyze the time complexity of the algorithm, consider a phase of the algorithm between two augmenting paths computations. In such a phase, the algorithm performs at most $\mathcal{O}(n)$ blossom shrinkings, while a single blossom shrinking can be done in time linear in the size of it, which is at most $\mathcal{O}(m)$. By Algorithm 6.4, each blossom shrinking is followed by a new execution of Algorithm 6.5, which is essentially a modification of BFS and thus can be implemented in $\mathcal{O}(n+m)$ time. Combining these observations, we can conclude that a single phase of the algorithm can be implemented in $\mathcal{O}(nm)$ time. This analysis yields a total time complexity of $\mathcal{O}(n^2 m)$, since the algorithm consists of at most $\mathcal{O}(n)$ phases.

To achieve an improvement by a factor of n, we note that Algorithm 6.4 can be easily modified to perform only a single execution of Algorithm 6.5 at each phase. In particular, once a

blossom is shrunk, it enough to assign to the pseudo-vertex corresponding to the shrunk blossom the same label as its base vertex, which is even; thus, by the first invariant of the algorithm, the new pseudo-vertex must also be added to Q. Since, as already mentioned, a blossom shrinking operation can be implemented in time linear in the size of it, it follows that all blossom shrinkings of a single phase of the algorithm can be done in $\mathcal{O}(m)$ time. This yields a total time of $\mathcal{O}(nm)$ over all phases of the algorithm, as desired.   □

6.4   The Hungarian method

In this section, we turn our attention to edge-weighted bipartite graphs and we seek to compute matchings that are of maximum weight (see Fig. 6.10 for an example). Let $G = (A \cup B, E, w)$ be such a graph, where $w : E \to \mathbb{R}^+$. Recall that the weight of matching M is the sum of the weights of edges in M, that is, $w(M) = \sum_{e \in M} w(e)$.

To ease the description, we make two simplifying assumptions:

(i) the bipartitions A and B of G are of the same cardinality, that is, $|A| = |B| = n$, and

(ii) graph G is a complete bipartite graph, that is, $E = A \times B$.

Note that both assumptions are without loss of generality. Indeed, the former can be achieved by adding $\mathcal{O}(n)$ vertices to G, while the second by adding edges of zero weight. Under these two simplifying assumptions, the maximum-weight matching is perfect (recall Definition 6.3). Next, we introduce two key concepts of the Hungarian method.

> **Definition 6.12** (Feasible labeling) Let $G = (A \cup B, E, w)$ be an edge-weighted complete bipartite graph with $|A| = |B| = n$. A vertex-labeling $\ell : A \cup B \to \mathbb{R}$ of G is called feasible if and only if $\ell(a) + \ell(b) \geq w(a, b)$ holds for any two vertices a and b in A and B, respectively.

Figure 6.10: (a) An edge-weighted complete bipartite graph, and (b) a maximum-weight perfect matching of it.
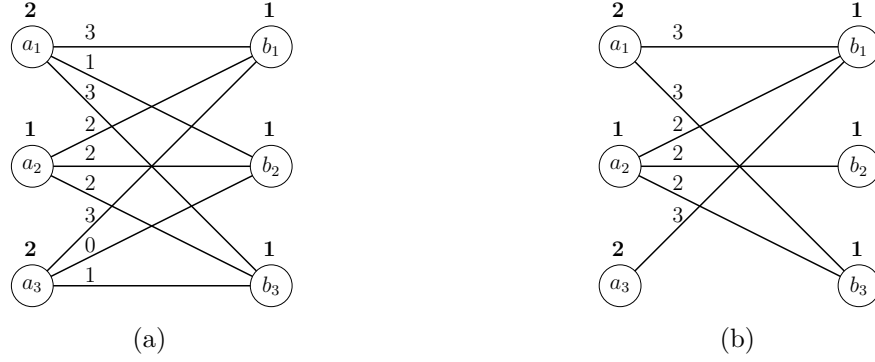
Figure 6.11: (a) A feasible labeling of the graph of Fig. 6.10a, and (b) its equality graph.

Definition 6.13 (Equality graph) Let $G = (A \cup B, E, w)$ be an edge-weighted complete bipartite graph with $|A| = |B| = n$, and let $\ell : A \cup B \to \mathbb{B}$ be a feasible labeling of G. The equality graph $G_\ell = (V_\ell, E_\ell)$ with respect to ℓ is defined as follows:

- $V_\ell = A \cup B$, and

- $E_\ell = \{(a, b) \in E : \ell(a) + \ell(b) = w(a, b)\}$.

Example 6.3 For an example of a feasible labeling and of its corresponding equality graph refer to Fig. 6.11. The labeling highlighted in bold is feasible, since $\ell(a) + \ell(b) \geq w(a, b)$ holds for any two vertices a and b. The equality graph of Fig. 6.11b is then obtain by keeping the edges for which the relationship holds as equality.

The following theorem is central in the Hungarian method, since it transforms the optimization problem of finding a maximum-weight matching in G into a purely combinatorial one, where the goal is to find a feasible labeling ℓ, such that a perfect matching exists in the equality graph G_ℓ.

Theorem 6.3 (Kuhn-Munkres) Let $G = (A \cup B, E, w)$ be an edge-weighted complete bipartite graph with $|A| = |B| = n$. If ℓ is any feasible labeling of G and M is any perfect matching in the equality G_ℓ of G, then M is a maximum-weight matching in G.

Proof. Since M is a perfect matching in the equality graph G_ℓ, we have:

$$w(M) = \sum_{(a,b)\in M} w(a, b) \overset{\text{by def. of } G_\ell}{=} \sum_{(a,b)\in M} \ell(a) + \ell(b) \overset{M \text{ is perfect}}{=} \sum_{v\in A\cup B} \ell(v) \qquad (6.5)$$

On the other hand, for any perfect matching M' of G, we have:

$$w(M') = \sum_{(a,b)\in M'} w(a, b) \overset{\ell \text{ is feasible}}{\leq} \sum_{(a,b)\in M'} \ell(a) + \ell(b) \overset{M' \text{ is perfect}}{=} \sum_{v\in A\cup B} \ell(v) \overset{(6.5)}{=} w(M)$$

Hence, M is a maximum-weight matching in G, as desired. $\qquad \square$

As already mentioned, Theorem 6.3 suggests that in order to compute a maximum-weight matching in G, it suffices to compute a feasible labeling ℓ of G, such that a perfect matching exists in the equality graph G_ℓ. Algorithm 6.6 suggests a natural approach to compute such a labeling ℓ and a corresponding perfect matching in G_ℓ. Note that at each iteration of the algorithm either the cardinality of M or the cardinality of E_ℓ is increased. So, the approach given in Algorithm 6.6 inevitably terminates. Furthermore, at termination, M must be a perfect matching in G_ℓ for some feasible labeling ℓ. So, by Theorem 6.3, M is a maximum-weight matching for G.

Algorithm 6.6: The core idea of the Hungarian method.

Input   : An edge-weighted complete bipartite graph $G = (A \cup B, E, w)$ such that
$\qquad\qquad w : E \to \mathbb{R}^+$ and $|A| = |B| = n$.

Output : A maximum-weight perfect matching of G.

1 Start with any feasible labeling ℓ and some matching M in E_ℓ, e.g.:
    - $\ell(a) = \max_{b \in B} w(a, b) \, \forall a \in A$;
    - $\ell(b) = 0 \, \forall b \in B$;
    - $M \leftarrow \emptyset$;

2 Construct the equality graph $G_\ell = (V_\ell, E_\ell)$ of G with respect to ℓ;

3 while M is not perfect do
4   if there is an augmenting path P with respect to M in G_ℓ then
5          $M \leftarrow M \oplus P$;
6   else
7   "Improve" ℓ to ℓ' such that $E_\ell \subset E_{\ell'}$;
8          $\ell \leftarrow \ell'$;
9   end
10 end
11 return M;

It remains to describe how a feasible labeling ℓ is improved, i.e., given a feasible labeling ℓ of G, we seek to compute a new feasible labeling ℓ' of G such that the number of edges of $G_{\ell'} = (V_{\ell'}, E_{\ell'})$ is strictly greater than the corresponding number of edges of $G_\ell = (V_\ell, E_\ell$, that is, $E_\ell \subset E_{\ell'}$. To this end, we need to introduce some useful notation. Let a be a vertex in A and let S be a subset of A. Then, we denote by $N_\ell(a)$ the neighborhood of a in G_ℓ, and by $N_\ell(S)$ the union of the neighborhoods of all vertices in S. Formally:

$$N_\ell(a) = \{b \in B : (a, b) \in E_\ell\} \text{ and } N_\ell(S) = \cup_{a \in S} N_\ell(a).$$

> **Lemma 6.10** Let $G = (A \cup B, E, w)$ be an edge-weighted complete bipartite graph with $|A| = |B| = n$, and let $\ell : A \cup B \to \mathbb{B}$ be a feasible labeling of G. Let also $S \subseteq A$ and $T = N_\ell(S)$. Finally, let:
>
> $$\lambda = \min\{\ell(a) + \ell(b) - w(a, b); a \in S, b \in B \setminus T\}$$

Then, labeling $\ell' : A \cup B \to \mathbb{R}^+$ with

$$\ell'(v) = \begin{cases} \ell(v) - \lambda & \text{if } v \in S \\ \ell(v) + \lambda & \text{if } v \in T \\ \ell(v) & \text{otherwise} \end{cases}$$

is a feasible labeling such that:

  (i) if $(a, b) \in E_\ell$ with $a \in S$ and $b \in T$, then $(a, b) \in E_{\ell'}$,

  (ii) if $(a, b) \in E_\ell$ with $a \notin S$ and $b \notin T$, then $(a, b) \in E_{\ell'}$, and

  (iii) there exists some edge $(a, b) \in E_{\ell'}$ with $a \in S$ and $b \notin T$.

Proof. Consider an edge (a, b) of G. Then, by definition of λ, we have:

$$\lambda \le \ell(a) + \ell(b) - w(a, b) \tag{6.6}$$

To prove that ℓ' is feasible, we consider cases based on whether a and b belong to S and T:

$$\text{If } a \in S \text{ and } b \in T, \text{ then: } \ell'(a) + \ell'(b) = \ell(a) - \lambda + \ell(b) + \lambda \ge w(a, b) \tag{6.7}$$

$$\text{If } a \notin S \text{ and } b \notin T, \text{ then: } \ell'(a) + \ell'(b) = \ell(a) + \ell(b) \ge w(a, b) \tag{6.8}$$

$$\text{If } a \notin S \text{ and } b \in T, \text{ then: } \ell'(a) + \ell'(b) = \ell(a) + \ell(b) + \lambda > w(a, b) \tag{6.9}$$

$$\text{If } a \in S \text{ and } b \notin T, \text{ then: } \ell'(a) + \ell'(b) = \ell(a) - \lambda + \ell(b) \overset{(6.6)}{\ge} w(a, b) \tag{6.10}$$

Eqs. (6.7) and (6.10) imply that ℓ' is feasible. Eqs. (6.7) and (6.8) imply that Properties (i) and (ii) of ℓ' hold. To prove Property (iii) of ℓ', let (a_0, b_0) be the edge for which $\lambda = \ell(a_0) + \ell(b_0) - w(a_0, b_0)$, where $a_0 \in S$ and $b_0 \in B \setminus T$. For this edge, we can conclude $\ell'(a_0) + \ell'(b_0) = \ell(a_0) - \lambda + \ell(b_0) = w(a_0, b_0)$. Hence, (a_0, b_0) belongs to $E_{\ell'} \setminus E_\ell$, as desired. $\square$

Algorithm 6.7 is an adjustment of the algorithmic idea presented in Algorithm 6.6, where the improvement of the feasible labeling is done using Lemma 6.10. Hence, the correctness of Algorithm 6.7 directly follows from Theorem 6.3 and Lemma 6.10. To complete the description of the Hungarian method, it remains to discuss the time complexity. Note that since at each iteration of the algorithm one of the cardinalities of S, M and E_ℓ is increased and since each of the first two of these sets has at most $\mathcal{O}(n)$ elements, while the third one at most $\mathcal{O}(n^2)$ elements, it follows that the algorithm terminates. Actually, a straight-forward analysis implies an upper bound of $\mathcal{O}(n^4)$ time in total. To improve this bound, for each vertex b in $B \setminus T$, we maintain a so-called slack value, denoted by $\text{slack}(b)$, such that:

$$\text{slack}(b) = \min_{a \in S}\{\ell(a) + \ell(b) - w(a, b)\}$$

These slack values allow to implement Step 3 of Algorithm 6.7 in $\mathcal{O}(n)$ time, since the value of λ can be computed in $\mathcal{O}(n)$ time, as follows:

$$\lambda = \min_{b \in B \setminus T} \text{slack}(b).$$

Of course, once Step 3 has been performed, the slack values of vertices in $B \setminus T$ must be updated in order to conform with the obtained labeling. This can be easily achieved in $\mathcal{O}(n)$ time, by updating the slack value of each vertex b in $B \setminus T$ as follows:

$$\text{slack}(b) = \text{slack}(b) - \lambda.$$

Step 4 also takes $\mathcal{O}(n)$ time. In particular, vertex b in $N_\ell(S) \setminus T$ can be found in $\mathcal{O}(n)$ time, assuming that, when adding a vertex of A in S, we mark its neighbors in B. By doing so, b corresponds to an unmarked vertex of B, and can be found in $\mathcal{O}(n)$ time, as claimed. Thus, the augmentations of M or S (depending on whether vertex b is free or not, respectively) can be done in $\mathcal{O}(n)$ time. Note that in the latter case, the addition of a' to S implies that we need to update the slack value of each vertex b in $B \setminus T$, as follows:

$$\text{slack}(b) = \min\{\text{slack}(b), \ell(a') + \ell(b) - w(a', b)\}.$$

Since each of Steps 3 and 4 of Algorithm 6.7 can be implemented in $\mathcal{O}(n)$ time and since in worst case the carinality of S is $|A| = \mathcal{O}(n)$, it follows that augmenting matching M by one edge can be done in $\mathcal{O}(n^2)$ time. Since $|M| = \mathcal{O}(n)$, Algorithm 6.7 can be implemented in $\mathcal{O}(n^3)$ time. We summarize this discussion in the following theorem.

> **Theorem 6.4** The Hungarian method computes a maximum-weight matching of a bipartite graph in $\mathcal{O}(n^3)$ time.

We conclude this section by mentioning that the problem of finding a maximum-weight matching of a complete bipartite graph cannot be solved in $o(n^2)$ time, since the size of such a graph is $\mathcal{O}(n^2)$, when its bipartitions are of asymptotically equal cardinality.

Exercises

> **Exercise 6.1** A matching M of a graph G is maximal if for every edge e of G that does not belong to M it holds that $M \cup \{e\}$ is not a matching of G.
>
> a) Prove or disprove: A maximal matching is a maximum matching.
>
> b) Prove or disprove: A maximum matching is a maximal matching.
>
> c) Show how to construct a maximal matching in linear time.
>
> d) Show that the cardinality of a maximal matching is at least half of the cardinality of a maximum matching.

Exercise 6.2 An edge coloring of an undirected graph G is a coloring of its edges, so that no two edges incident to a common vertex are of the same color. A k edge-coloring is an edge coloring with k different colors, where $k \geq 1$.

a) Prove that any k edge-coloring of a graph is a partition of its edges into k edge-disjoint matchings.

b) Prove that any k-regular bipartite graph admits a k edge-coloring; recall that a k-regular is a graph whose vertices have degree exactly k.

c) Prove that any bipartite graph G admits a $\mathbf{deg}(G)$ edge-coloring, where $\mathbf{deg}(G)$ denotes the maximum degree of G.

Exercise 6.3 Let G be a graph with n vertices. A subset $\mathcal{E}$ of the edge-set of G is an arc cover of G if and only if for every vertex v of G there exists at least one edge in $\mathcal{E}$ that is incident to v. Let s be the cardinality of a maximum matching of G and let t be the cardinality of a minimum arc cover of G. Prove that $t + s = n$.

Exercise 6.4 A vertex cover of a graph G is a subset $\mathcal{V}$ of its vertices such that every edge of G is incident to at least one vertex of $\mathcal{V}$.

a) Show that the cardinality of a maximum matching is a lower bound for the cardinality of any vertex cover.

b) Show that for any maximal matching M of a graph G, there is a vertex cover of G of cardinality $2|M|$.

Exercise 6.5 Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n points in the interior of a rectangle R, which represent a set of n cities of a map. On the right boundary edge of R we have already placed n uniform-size boxes that act as placeholders for the names of the cities. We want to associate each point with a box (where we will write the name of the corresponding city) such that no two points are associated to the same box. To visualize an association we draw a straight-line segment (called connection) from each point to the mid-point of the left side of its associated box. For an example refer to the following figure.
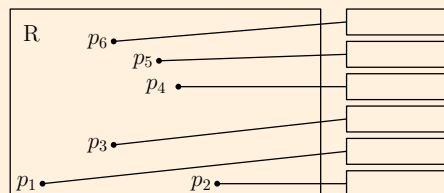
a) Present a polynomial time algorithm to find a crossing-free association, that is, no two connections cross.

b) Model the problem of finding an association in which the total length of the connections is minimum as a matching problem.

DRAFT

c) Prove or disprove that an association in which the total length of the connections is minimum is crossing-free.

d) Prove or disprove that in a crossing-free association the total length of the connections is minimum.

Exercise 6.6 Let $G = (A \cup B, E)$ be a bipartite graph with bipartitions A and B, i.e., $E \subseteq A \times B$. Prove the following statements.

a) G has a perfect matching if and only if it satisfies the Hall's condition:

$$|N(S)| \geq |S|, \text{ for every subset } S \text{ of } A \text{ or } B,$$

where $N(S)$ denotes the neighborhood of S in G.

b) If $\deg(a) \geq 1$ for every vertex a in A, and $\deg(a) \geq \deg(b)$ for every edge (a, b) in E with $a \in A$ and $b \in B$, then G admits a matching in which every vertex of A is matched.

c) Let M and N be two arbitrary matchings of G. Construct a matching that matches all the vertices in A matched by M and all the vertices in B matched by N.

Exercise 6.7 Let $G = (A \cup B, E)$ be a bipartite graph with a matching M. Hall's condition is a necessary and sufficient condition for M to be a perfect matching.

Suppose that M is not perfect. Then, there exists a subset S of A or B, such that $|S| > |N(S)|$. Such a set S is called a short certificate that M is not perfect. Give an efficient algorithm to compute such a short certificate.

Exercise 6.8 Consider the graph of the following figure, which has a matching that is indicated in bold.
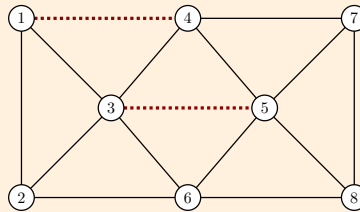
a) Specify a blossom in the graph whose stem contains two edges.

b) Contract this blossom and show the contracted graph.

c) List all augmented paths of length 3 in the contracted graph and the corresponding paths in the original graph.

DRAFT

Exercise 6.9 Let $G = (A \cup B, E)$ be the complete weighted bipartite graph with bipartitions $A = \{a_1, \ldots, a_5\}$ and $B = \{b_1, \ldots, b_5\}$, whose edge weights are as in the following table.

	a_1	a_2	a_3	a_4	a_5
b_1	6	7	8	1	3
b_2	5	3	4	5	3
b_3	3	4	7	2	7
b_4	9	6	9	4	5
b_5	1	5	6	3	2

Exercise 6.10 Let $G = (A \cup B, E, c)$ be a complete bipartite graph, where $c : E \to [1, \infty)$. A maximum product maximum matching of G is a maximum matching $M \subseteq E$ such that the product of its edge weights $\prod_{e \in M} c(e)$ is maximized.

a) Show that neither the solution of the minimum cost maximum matching problem nor the solution of the maximum cost maximum matching problem is necessarily a maximum product maximum matching.

b) Present an algorithm that computes the maximum product maximum matching of G. Prove the correctness of your algorithm.

Exercise 6.11 Lemma 6.3 provides an upper bound on the length of the shortest augmenting path for a given matching of cardinality μ.

a) Derive from this lemma what is the percentage of matching edges with respect to the cardinality of the maximum matching that can be found by considering augmenting paths of odd lengths only in the algorithm by Hopcroft and Karp. Discuss the case of a perfect matching.

b) Assume that you want to compute at least 99% of the edges of a maximum matching. Determine an upper bound on the length of the shortest augmenting paths that one has to consider.

c) Theorem 6.1 provides an upper bound of $O(\sqrt{n}(n + m))$ for the time complexity of the algorithm by Hopcroft and Karp. Can this bound be improved if one seeks in computing a matching that contains at least 99% of the edges of the corresponding maximum matching?

Further reading

– R. Tarjan, "Sketchy Notes on Edmonds' Incredible Shrinking Blossom Algorithm for General Matching," Course Notes, Department of Computer Science, Princeton University, 2002. https://www.cs.dartmouth.edu/~ac/Teach/CS105-Winter05/Handouts/tarjan-blossom.

pdf

– http://www.cs.tau.ac.il/~zwick/grad-algo-0910/match.pdf

– U. Zwick, "Lecture Notes on: Maximum Matching in Bipartite and Non-Bipartite Graphs,"
  2009. https://www2.cs.duke.edu/courses/fall15/compsci532/scribe_notes/lec02.pdf

– H. W. Kuhn. The hungarian method for the assignment problem. Naval Research Logistics
  Quarterly, 2(1-2):83–97, 1955

– http://www.columbia.edu/~cs2035/courses/ieor6614.S16/GolinAssignmentNotes.pdf

---

Algorithm 6.7: The Hungarian method.

Input   : An edge-weighted complete bipartite graph $G = (A \cup B, E, w)$ such that
$w : E \rightarrow \mathbb{R}^+$ and $|A| = |B| = n$.

Output : A maximum-weight perfect matching of G.

1 Step 1: Initialization.

2 $M \leftarrow \emptyset$;   // Initial matching
3 foreach vertex a in A do $\ell(a) \leftarrow \max_{b \in B} w(a, b)$;   // Initial labeling
4 foreach vertex b in B do $\ell(b) \leftarrow 0$;
5 Construct the equality graph $G_\ell = (V_\ell, E_\ell)$ of G with respect to ℓ;

6 Step 2: Advance.

7 if M is not perfect then
8 $\quad$ $a \leftarrow$ free vertex in A;
9 $\quad$ $S \leftarrow \{a\}$; $T \leftarrow \emptyset$;
10 else return M ;

11 Step 3: Improve ℓ.

12 if $N_\ell(S) = T$ then
13 $\quad$ $\lambda \leftarrow \min\{\ell(a) + \ell(b) - w(a, b); a \in S, b \in B \setminus T\}$;   // Apply Lemma 6.10
14 $\quad$ foreach vertex a in S do $\ell(a) \leftarrow \ell(b) - \lambda$;
15 $\quad$ foreach vertex b in T do $\ell(b) \leftarrow \ell(b) + \lambda$;
16 $\quad$ Update G_ℓ;

17 Step 4: Augment M or S.

18 if $N_\ell(S) \neq T$ then
19 $\quad$ $b \leftarrow$ a vertex in $N_\ell(S) \setminus T$;
20 $\quad$ if b is free then
21 $\quad\quad$ $M \leftarrow M \cup \{(a, b)\}$;   // Augment M
22 $\quad\quad$ Go to Step 2;
23 $\quad$ else
24 $\quad\quad$ $a' \leftarrow$ the vertex matched to b;   // $(a', b) \in M \Rightarrow (a', b) \in E_\ell$
25 $\quad\quad$ $S \leftarrow S \cup \{a'\}$; $T \leftarrow T \cup \{b\}$;   // Augment S
26 $\quad\quad$ Go to Step 3;
27 $\quad$ end

---

7. Planar Graphs

This chapter will focus on properties of planar graphs, i.e., on graphs that admit drawings on the Euclidean plane $\mathbb{R}^2$ such that no two edges cross (except probably at common endpoints). In Section 7.1, we introduce several notions and terminology related to planar graphs. Section 7.2 focuses on the proof of the so-called Euler's formula, which implies that planar graphs with n vertices cannot have more than $3n - 6$ edges. Section 7.3 is devoted on proving that the compete graph K_5 on five vertices as well as the complete bipartite graph $K_{3,3}$ are not planar. Section 7.5 discusses an algorithm to compute a planar straight-line drawing of a planar graph on an integer grid of quadratic area. Section 7.6 focuses on the Planar Separator Theorem, which provides an upper bound on the number of vertices that are required to be removed such that each connected component of the derived graph has a certain size. Section 7.7 goes beyond planar graphs, by providing a lower bound on the number of crossings for such graphs. We conclude in Section 7.8 with a few remarks on the covered topics.

7.1 Basic definitions

As already mentioned, a planar graph is one admitting a drawing on the Euclidean plane, such that no two of its edges cross. The formal definition follows. However, note that there exist several relaxations of this definition in the literature.

> **Definition 7.1** (Planar graphs) A graph G is planar if it admits a drawing on the Euclidean plane $\mathbb{R}^2$, such that
>
> (i) the vertices of G are distinct points on $\mathbb{R}^2$, i.e., no two vertices share the shame point,
>
> (ii) the edges are simple curves connecting their endpoints, and
>
> (iii) no two curves cross (except at their endpoints)

Accordingly, a drawing of graph G with the aforementioned properties (i)–(iii) is called planar.

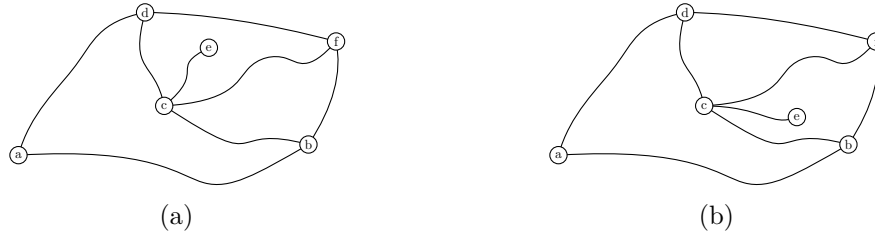(a)   (b)

Figure 7.1: Two different planar embeddings of a graph with 5 vertices, 7 edges, and 4 faces.

A planar drawing partitions the plane into topologically connected regions, which are called faces. The unbounded face is called the outer face; for an illustration refer to Fig. 7.1. A face can be described by a clockwise enumeration of the edges that bound it. Note that this enumeration does not necessarily imply a simple cycle, e.g., if there is a vertex with degree one on a boundary of a face, then its incident edge appears twice in the enumeration of the edges that bound the face. Further, note that each face can become the outer face (by appropriately changing the drawing).

Several planar drawings of a graph are similar, in the sense that they define the same set of faces. To make this more evident consider the planar drawing of Fig. 7.1a and replace each edge connecting two vertices with one drawn as a straight-line segment between them. The resulting drawing will be clearly a planar grawing of the graph of Fig. 7.1a, which is in a sense "equivalent" to the initial one. The notion of an embedding introduced in the following definition formalizes this concept.

> Definition 7.2   An embedding of a planar graph is an equivalence class of planar drawings that define the same set of faces.

 A planar embedding is equivalently defined by the cyclic order of the edges around each vertex. Given a graph, testing whether it is planar and computing a planar embedding can be done in linear time [3]. However, the corresponding algorithms are too technical and thus beyond the scope of this chapter.

> Definition 7.3   A planar graph together with a planar embedding (i.e., a cyclic order of the edges around each vertex) of it is called a plane graph.

Note that not any two planar drawings of a planar graph have the same embedding, as illustrated in Fig. 7.1.

Motivation: There exist several reasons that justify why planar graphs are of interest. In the following, we name a few.

- Several non-tractable problems admit efficient algorithms, if the input is a planar graph.

- Planar graph are not dense, i.e., their maximum number of edges is at most linear in

their number of vertices, i.e., $m = \mathcal{O}(n)$; see Section 7.2.

- Many structures (i.e., real-world networks) are planar or almost planar, e.g., social networks form such an example, as they are locally dense but globally sparse.

- Planar graphs can be characterized in terms of forbidden minors and recognized in linear time; see Section 7.3.

- Planar graphs have several remarkable graph-theoretic properties, e.g., their chromatic number is at most 4; see Section 7.4.

Subclasses of planar graphs. We conclude this section by mentioning some important subclasses of planar graphs. First, observe that every tree (i.e., a connected, acyclic graph) is a planar graph, as a certificate planar drawing, in which all vertices are incident to its outer face, is not difficult to be constructed. The class of outerplanar graphs, formally introduced in the following, forms a super class of the class of trees.

> **Definition 7.4** (Outerplanar graph) An outerplanar graph is a graph that admits a planar drawing, in which all vertices are incident to its outer face.

Another important subclass of the class of planar graphs is the one of the maximal planar graphs, which as we will shortly see have maximum edge density.

> **Definition 7.5** (Maximal planar graph) A maximal planar graph is a planar graph such that the addition of an edge between any two vertices of it yields a non-planar graph.

Clearly, all faces of a maximal planar graph are triangular. Also, note that any planar graph can be augmented to maximal planar. A straight-forward such linear-time augmentation introduces a new vertex into each face of the original planar graph together with all connections to the vertices bounding the face. More advanced augmentations introduce only edges and can also be done in linear time.

An important property of maximal planar graphs is 3-connectivity, as shown in the following.

> **Property 7.1** Every maximal planar graph is 3-connected.

Proof. It is straight-forward to see that a maximal planar graph cannot have a cut-vertex. To prove the property, assume for a contradiction that there exists a 2-connected maximal planar graph G, i.e., G has a separation pair $\langle u, v \rangle$. It follows that the removal of u and v yields at least two connected components C_1 and C_2. Let Γ_1 and Γ_2 be the subdrawings of the planar drawing Γ of G restricted to $C_1 \cup \{u, v\}$ and $C_2 \cup \{u, v\}$, respectively. The outer face of each of Γ_1 and Γ_2 consists of u, v and at least one additional vertex. Let w_1 and w_2 be

these additional vertices in Γ_1 and Γ_2, respectively. It follows that (w_1, w_2) can be added to Γ without deviating its planarity; a contradiction to the fact that G is maximal planar. $\qquad\square$

Similarly, one can prove the following.

> **Property 7.2** Every maximal outerplanar graph is biconnected.

7.2   Euler's formula

This section focuses on a formula relating the number of vertices, faces, and edges of a simply connected planar graph, which was first discovered by Euler back in 1752. Note that the formula holds even for non-simple planar graphs (i.e., containing self-loops or parallel edges). To simplify the presentation in the following, we do not distinguish between a planar graph and its corresponding drawing. In other words, we assume that the considered graphs are topological, i.e., drawn on the Euclidean plane under the Conditions (i)–(iii) of Definition 7.1.

> **Theorem 7.1** (Euler 1752) Let G be a connected planar graph with n vertices, m edges, and f faces. Then,
> $$n + f = m + 2$$

Proof. Let $\mathcal{T}$ be a spanning tree of G. Let also $\mathcal{T}^*$ be the graph having one vertex v_ϕ for each face ϕ of G such that for every pair of faces ϕ and ϕ' of G there is an edge $(v_\phi, v_{\phi'})$ in $\mathcal{T}^*$ if and only if ϕ and ϕ' share an edge of G that does not belong to $\mathcal{T}$. It follows that $\mathcal{T}^*$ is acyclic (as otherwise $\mathcal{T}$ would not be spanning) and connected (as otherwise $\mathcal{T}$ would contain a cycle). Thus $\mathcal{T}^*$ is a tree. Denote by n_T and m_T the number of vertices and edges of $\mathcal{T}$, respectively. Similarly, denote by n_{T^*} and m_{T^*} the number of vertices and edges of $\mathcal{T}^*$, respectively. Since $\mathcal{T}$ and $\mathcal{T}^*$ are trees, it follows:

$$m_T = n_t - 1 \quad \text{and} \quad m_{T^*} = n_{T^*} - 1 \tag{7.1}$$

Thus, by summing up the two equalities of Eq. (7.1), we obtain:

$$m_T + m_{T^*} = n_{T^*} + n_t - 2 \tag{7.2}$$

Since T is a spanning tree of G, we have:

$$n_T = n \tag{7.3}$$

Since by definition $\mathcal{T}^*$ has a vertex for each face of G, we further have:

$$n_{T^*} = f \tag{7.4}$$

Finally, since all edges of G that do not belong to $\mathcal{T}$ are in one-to-one correspondence with the edges of $\mathcal{T}^*$, it follows:

$$m_T + m_{T^*} = m \tag{7.5}$$

Thus, by plugging Eqs. (7.3)–(7.5) to the one of Eq. (7.2), we obtain:

$$m = n + f - 2$$

$\square$

In the following, we study implications of Euler's formula.

> **Corollary 7.1** A maximal planar graph with n vertices has exactly $3n - 6$ edges.

Proof. Let G be a maximal planar graph with n vertices, and in accordance to Theorem 7.1 denote by m and by f the number of edges and faces of G, respectively. Observe that if for each face of G we count the (three) edges on its boundary, then we will eventually count each edge of G exactly twice (i.e., once from its of the two faces that it bounds). Therefore:

$$3f = 2m. \tag{7.6}$$

Thus, by Euler's formula, we obtain:

$$n + \frac{2}{3}m = m + 2 \Rightarrow m = 3n - 6.$$

$\square$

In the following theorem, we prove that a general (i.e., not necessarily maximal) planar graph with n vertices has at most $3n - 6$ edges, which implies that the maximal planar graphs form the densest members of the class of planar graphs.

> **Theorem 7.2** A planar graph with n vertices has at most $3n - 6$ edges.

Proof. Let G be a maximal planar graph with n vertices, m edges and f faces, and denote by f_i the number of faces with length i. Then, the number f of faces of G is equal to the number f_3 of faces of length 3 plus the number f_4 of faces of length 4 and so on. Thus:

$$f = f_3 + f_4 + f_5 + \ldots \tag{7.7}$$

As we observed in the previous corollary, if for each face of G we count the edges on its boundary, then we will eventually count each edge of G exactly twice (i.e., once from its of the two faces that it bounds), that is:

$$2m = 3f_3 + 4f_4 + 5f_5 + \ldots \tag{7.8}$$

Since:

$$3f_3 + 3f_4 + 3f_5 + \ldots \quad \leq \quad 3f_3 + 4f_4 + 5f_5 + \ldots$$

Eqs. (7.7) and (7.8) imply:

$$3f \leq 2m$$

Thereby, we have obtained a relationship analogue to the one of Eq. (7.6) for maximal planar graphs and by Euler's formula we can conclude:

$$m + 2 \leq n + \frac{2}{3}m \Rightarrow m \leq 3n - 6$$

$\square$

> **Corollary 7.2** Every planar graph has a vertex of degree at most 5.

The following theorem is left as an exercise to the reader (see Exercise 7.5).

> **Theorem 7.3** Every outerplanar graph with n vertices has at most $2n - 3$ edges.

7.3   Forbidden subgraphs

In this section, we study substructures that are not allowed in a planar graph. Intuitively, since a planar graph has a linear number of vertices with respect to its number of vertices (as proved in Theorem 7.2), it cannot contain as a subgraph a large clique (recall that a clique on n vertices has $\frac{n(n-1)}{2}$ edges) or a large bipartite graph (recall that K_{nm} contains nm edges). This intuition is formalized in the following theorem, which directly implies that a planar graph contain as a subgraph neither the complete graph K_5 on five vertices nor the complete bipartite graph $K_{3,3}$.

> **Theorem 7.4** Graphs K_5 and $K_{3,3}$ are not planar.

Proof. First, assume to the contrary that K_5 is planar. Since K_5 has $n = 5$ vertices and $m = 10$ edges, by Theorem 7.2 it follows that it has at most $3 \cdot 5 - 6 = 9$ edges. However, this is a contradiction, since K_5 has 10 edges.

The corresponding proof for $K_{3,3}$ is slightly more involved. Again, we assume for a contradiction that $K_{3,3}$ is planar. Since $K_{3,3}$ has $n = 6$ vertices and $m = 9$ edges, by Euler's formula it follows that it has $f = m + 2 - n = 5$ faces. Hence, the average face length (defined as the total length of all faces over the number of faces of G), denoted by $\overline{f}$, of $K_{3,3}$ is:

$$\overline{f} = \frac{3f_3 + 4f_4 + 5f_5 + \dots}{f} \overset{(7.8)}{=} \frac{2m}{f} = \frac{18}{5} < 4.$$

Hence, $K_{3,3}$ must have a face of length at most 3, which is a contradiction since $K_{3,3}$ is simple and bipartite. □

Theorem 7.4 can easily be generalized through the operation of edge-subdivision, which is defined as follows. The subdivision of an edge (u, v) of a graph G consists of deleting (u, v) from G and adding a new vertex w along with two new edges (u, w) and (w, v). A graph which has been derived from G by a sequence of edge subdivision operations is called a subdivision of G. In view of Theorem 7.4, it is not difficult to see that a planar graph can contain as a subgraph neither a subdivision of K_5 nor a subdivision of $K_{3,3}$. Kurantowski back in 1930 showed that the reverse of this observation also holds (the proof however is beyond the

scope of this chapter), thus providing a complete characterization of planar graphs in terms of forbidden subgraphs. We summarize this result in the following theorem.

> **Theorem 7.5** (Kuratowski's theorem, 1930) A graph is planar if and only if it does not contain a subgraph that is a subdivision of K_5 or $K_{3,3}$.

7.4 The chromatic number of planar graphs

A vertex coloring of a graph is a coloring of its vertices with a certain number of colors, such that adjacent vertices have different colors. The minimum required number of colors to obtain a vertex coloring of a graph is its chromatic number, e.g., the chromatic number of K_4 is 4. A well-known theorem, known in the literature as 4-color theorem, states that every planar graph admits a vertex coloring with at most four colors. The proof, however, is complicated and several parts of it have been verified by computer. On the other hand, Corollary 7.2 directly implies that every planar graph admits a vertex coloring with at most six colors. In the following, we give a relatively easy proof for the existence of a corresponding vertex coloring with at most five colors. The proof is given via a recursive $\mathcal{O}(n)$ algorithm.

> **Theorem 7.6** Every planar graph admits a vertex coloring with at most 5 colors, such that adjacent vertices have different colors.

Proof. Let G be a planar graph with n vertices. The base of the recursive algorithm corresponds to the case $n \leq 5$, in which a vertex coloring with at most 5 colors is trivial to be computed. Hence, we assume $n \geq 6$. Since G is planar, by Corollary 7.2 there exists a vertex, say v, of G with degree at most 5. If vertex v has degree at most 4, then we proceed as follows. We remove v from G and obtain a new graph G', that is, $G' = G \setminus \{v\}$, which is obviously planar. Then, we recursively compute a vertex coloring of G' with at most five colors. To obtain a vertex coloring of G with at most five colors, it is enough to specify the color of vertex v. To this end, we observe that vertex v is adjacent to at most four vertices, say $v_1, \ldots, v_4$ in G. Hence, we can assign to v the color not assigned to $v_1, \ldots, v_4$.

In the following , we may assume w.l.o.g. that all vertices of G have degree at least 5. Consider now vertex v, whose degree is exactly 5. Let $v_1, \ldots, v_5$ be the neighbors of v in G. Since by Theorem 7.4 graph G cannot contain the complete graph K_5 as a subgraph, not all neighbors of v are pairwise adjacent. Assume w.l.o.g. that v_1 and v_2 are not adjacent. We proceed by removing v from G and by contracting v_1 and v_2 into a single vertex x. In this way, we obtain a new graph G'. It is see to see that G' is planar. We recursively compute a vertex coloring of G' with at most five colors. To extend this coloring to a vertex coloring of G with at most five colors, it is enough to specify the colors of vertices v, v_1 and v_2 in G. To this end, we first set the colors of v_1 and v_2 to the color of vertex x of G', which is allowed since v_1 and v_2 are not adjacent. Since the neighbors of v use together at most four colors, this allows us to assign to v the "spare" color of its neighbors again, namely, the one not assigned to $v_1, \ldots, v_5$. $\square$
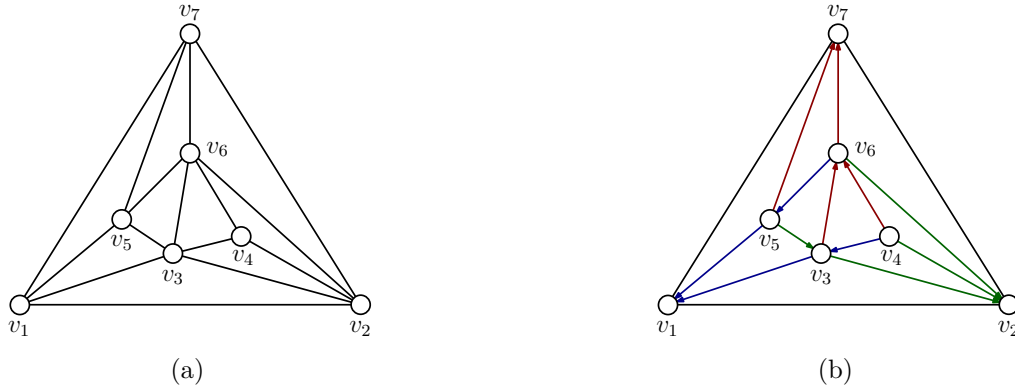
Figure 7.2: Illustration of (a) a canonical order of a maximal planar graph, and (b) a corresponding Schnyder realizer.

7.5   How to draw a planar graph

In this section, we describe an algorithm to compute a planar straight-line drawing of an input n-vertex maximal planar graph on an integer grid of size $\mathcal{O}(n) \times \mathcal{O}(n)$, assuming that a planar embedding (that is, a cyclic order of the edges around each vertex) of the input graph is specified as part of the input, e.g., computed by a planarity testing algorithm. Note that the assumption that the input graph is maximal planar is without loss of generality, since, as already mentioned, any planar graph can be augmented to maximal planar by only adding edges in linear time. A key concept of this algorithm is the so-called canonical order, which is introduced formally in the following.

7.5.1   Canonical orders

A canonical order of a maximal planar graph is a particular permutation of its vertices, which is formally defined as follows.

Definition 7.6 (Canonical order) Let G be a maximal planar graph with n vertices, and let $\mathcal{E}$ be a planar embedding of G, such that v_1, v_2 and v_n are the vertices of G that are delimiting the outer face of $\mathcal{E}$.

A canonical order of G is a total order $v_1, v_2, \ldots, v_n$ of its vertices that meets the following conditions for every $3 \le k \le n$:

(C.1) the subgraph G_{k-1} of G induced by $v_1, v_2, \ldots, v_{k-1}$ is 2-connected, and the boundary of its outer face is a cycle C_{k-1} containing the edge (v_1, v_2), and

(C.2) the vertex v_k is in the outer face of G_k, and its neighbors in G_{k-1} are at least two and appear consecutively along the path $C_{k-1} \setminus (v_1, v_2)$.

> **Example 7.1** Fig. 7.2a illustrates a sample canonical order of a maximal planar graph. Observe that the neighbors v_5, v_3, v_4 and v_2 of v_6 in G_5 appear consecutive along the cycle $C_5 = \langle v_1, v_5, v_3, v_4, v_2 \rangle$ delimiting the outer face of G_5.

Our goal is to prove that every maximal plane graph admits a canonical order, which can be computed in linear time. To this end, we need the definition of a chord of a cycle (see Definition 7.7) and an auxiliary lemma regarding possible separating pairs (see Lemma 7.1).

> **Definition 7.7** (Chord) Let C be a cycle in a graph G. A chord of C is an edge of G connecting non-adjacent vertices of C.

> **Lemma 7.1** Let G be a maximal plane graph with a canonical order $v_1, v_2, \ldots, v_n$. Then, for every $3 \leq i \leq n-1$, any separating pair $\langle u, w \rangle$ of G_i is a chord of C_i.

Proof. Since G is a maximal plane graph and since each vertex v_j with $j > i$ is in the outer face of G_j, all the internal faces of G_i are triangles. Now, add a dummy vertex d in the outer face of G_i and connect d to each vertex on C_i. The obtained graph $\hat{G}_i$ is maximal planar, and thus, by Property 7.1, 3-connected. For each separation pair $\langle u, w \rangle$ of G_i, $\langle u, v, d \rangle$ is a separating triplet of $\hat{G}_i$, since $G_i = \hat{G}_i \setminus d$. Since $\langle u, v, d \rangle$ is a separating triangle in $\hat{G}_i$, the edge (u, v) is a chord of C_i. $\square$

> **Theorem 7.7** Every maximal plane n-vertex graph admits a canonical order, which can be computed in $\mathcal{O}(n)$ time.

Proof. Let G be a maximal planar n-vertex graph and let $\mathcal{E}$ be a planar embedding of it. Without loss of generality, we assume that $n > 3$ holds, as otherwise (i.e., $n = 3$) a canonical of G is trivially defined. We prove the existence of a canonical order for G by constructing a vertex order $v_n, \ldots, v_1$ that complies with Conditions C.1 and C.2 of Definition 7.6, that is, we construct the reverse of a canonical order.

For vertex v_n, we observe that $G_{n-1} = G \setminus \{v_n\}$ is 2-connected, because G is 3-connected (by Property 7.1). Since G is maximal planar, the set of vertices adjacent to v_n forms a cycle C_{n-1}, which is the boundary of the outer face of G_{n-1}. Thus Conditions C.1 and C.2 hold for v_n.

Assume that vertices $v_n, v_{n-1}, \ldots, v_{k+1}$, with $k \geq 3$, have been computed. Next, we show how to compute vertex v_k. If there is a vertex x on C_k different than v_1 and v_2, which is not an end-vertex of a chord of C_k, then we can choose x as v_k. Indeed if deleting x from G_k violated the 2-connectivity, then the cut-vertex y of $G_k \setminus \{x\}$, together with x would form a separating pair for G_k and hence (x, y) would be a chord in G_k (by Lemma 7.1); a contradiction to the

choice of x. In the following, we show that there is a vertex on C_k, which is not an end-vertex of a chord of C_k.

Let $w_1,\ldots,w_p$ be the vertices of $C_k \setminus (v_1, v_2)$ consecutively starting from v_1 and ending at v_2, where $w_1 = v_1$ and $w_p = v_2$. By definition, for any chord (w_i, w_j) of C_k with $i < j$, it holds that $i < j - 1$. We say that a chord (w_i, w_j) of C_k with $i < j$ includes another chord $(w_{i'}, w_{j'})$ of C_k with $i' < j'$ if and only if $i \le i' < j' \le j$. Consider an inclusion-minimal chord (w_i, w_j) of C_k. Then, any vertex w_ℓ for $i < \ell < j$ can be chosen as v_k. Since v_k is not an end-vertex of a chord for C_{k-1}, $G_{k-1} = G_k \setminus \{v_k\}$ remains 2-connected. Furthermore due to the maximal planarity of G, the neighborhood of v_k on C_{k-1} are consecutive along $C_{k-1} \setminus \{(u,v)\}$.

Note that the algorithm described above can be implemented to run in $\mathcal{O}(n)$-time by maintaining a variable for each vertex of C_k that corresponds the number of chords incident to it. Once we determine the next vertex v_k in the reverse canonical order, the variables for all its neighbors can be updated in $\mathcal{O}(\deg(v_k))$ time. Summing over all vertices of G leads to an overall linear running time. □

7.5.2 The shift algorithm by de Fraysseix, Pach and Pollack

The shift-method [2] is a well-known incremental algorithm, which constructs in linear time a plane drawing Γ of a maximal planar graph $G = (V, E)$. Drawing Γ has integer grid coordinates and occupies quadratic area. More precisely, based on a canonical order $v_1,\ldots,v_n$ of G, drawing Γ is constructed as follows. Initially, vertices v_1, v_2 and v_3 are placed at points $(0,0)$, $(2,0)$ and $(1,1)$. For $k = 4,\ldots,n$, assume that a planar drawing Γ_{k-1} of G_{k-1} has been constructed such that $C_{k-1} \setminus \{(v_1, v_2)$ is x-monotone and each edge of C_{k-1} is drawn as a straight-line segment with slope ± 1, except for the edge (v_1, v_2), which is drawn as a horizontal line segment (contour condition; see Fig. 7.3a). We further assume that each of the vertices $v_1,\ldots,v_{k-1}$ has been associated with a so-called shift-set, which for v_1, v_2 and v_3 are singletons containing only themselves, that is, $S(v_1) = \{v_1\}$, $S(v_2) = \{v_2\}$ and $S(v_3) = \{v_3\}$.

(a) contour condition (b) placement of v_k in Γ_{k-1}
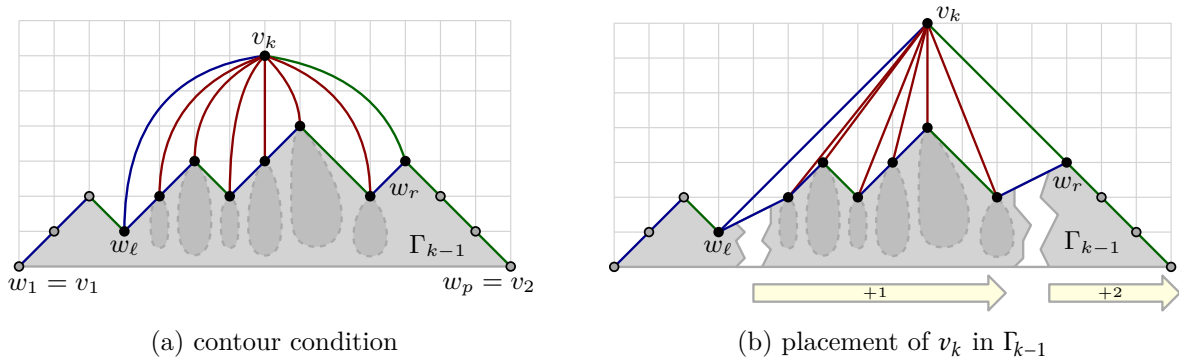
Figure 7.3: Illustration of the shift-method by de Fraysseix, Pach and Pollack.

Let $w_1,\ldots,w_p$ be the vertices of C_{k-1} from left to right in Γ_{k-1}, where $w_1 = v_1$ and $w_p = v_2$. Let also $w_\ell,\ldots,w_r$, with $1 \le \ell < r \le p$ be the neighbors of v_k from left to right along C_{k-1} in Γ_{k-1}. To avoid edge-overlaps, the algorithm first translates each vertex in $\bigcup_{i=\ell+1}^{r-1} S(w_i)$ one unit to the right and each vertex in $\bigcup_{i=r}^{p} S(w_i)$ two units to the right; see Fig. 7.3b. Then, the

algorithm places vertex v_k at the intersection of the line of slope $+1$ through w_ℓ with the line of slope -1 through w_r (which is a grid point, since by the contour condition the Manhattan distance between w_ℓ and w_r is even) and sets

$$S(v_k) = \{v_k\} \bigcup_{i=\ell+1}^{r-1} S(w_i). \tag{7.9}$$

It follows that the contour condition is maintained in Γ_k. Furthermore, once v_k is placed, vertices $w_{\ell+1}, \ldots, w_{r-1}$ are covered by v_k. Since each inner vertex of G is covered exactly once, the cover-relationship defines a tree in $G_n = G$ rooted at v_n. In G_k with $k < n$, this relationship defines a forest, such that the roots of each tree in the forest are vertices of C_k. Observe that Eq. (7.9) encapsulates exactly this cover-relationship. The following lemma implies that drawing Γ_n is a planar straight-line drawing of $G_n = G$, since it shows that the performed shifts do not introduce crossings[1].

> **Lemma 7.2** Let Γ_k be a planar straight-line drawing of G_k and let $w_1, \ldots, w_q$ be the vertices of C_k. Let $a_1, \ldots, a_q$ be positive integer numbers such that $a_1 \leq \ldots \leq a_q$. If we shift $S(w_i)$ by a_i unit to the right for each $1 \leq i \leq q$, then the obtained drawing of Γ_k is a planar straight-line of G_k.

Proof. We prove the statement by induction on k. The base of the induction corresponds to the case $k = 3$, for which the statement trivially holds. Assume in the inductive hypothesis that the statement holds for G_{k-1} for some k greater than 4. Consider now the drawing Γ_k of graph G_k obtained by placing vertex v_k in the drawing of Γ_{k-1}. Let $a_1 \leq \ldots \leq a_\ell \leq a \leq a_r \leq a_p$ be a fixed order of integer number, where $w_1, \ldots, w_p$ are the vertices of C_{k-1} from left to right in Γ_{k-1} (with $w_1 = v_1$ and $w_p = v_2$), such that $w_\ell, \ldots, w_r$, with $1 \leq \ell < r \leq p$ are the neighbors of v_k from left to right along C_{k-1} in Γ_{k-1}. We proceed by shifting $S(w_i)$ by a_i unit to the right for each $i \in \{1, \ldots, \ell\} \cup \{r, \ldots, p\}$, and v_k by a units to the right. To show that the obtained drawing is planar straight-line, we let

$$a_i' = \begin{cases} a_i, & \text{for } i = 1, \ldots, \ell \\ a, & \text{for } i = \ell+1, \ldots, r-1 \\ a_i, & \text{for } i = r, \ldots, p \end{cases}$$

By the induction hypothesis, G_{k-1} has planar straight-line drawing $\hat{\Gamma}_{k-1}$, in which $S(w_i)$ has been shifted by a_i', where $i = 1, \ldots, p$. It follows that drawing Γ_k stays planar straight-line after the shifting, since $\hat{\Gamma}_{k-1}$ is planar straight-line and since v_k moves together with $w_{\ell+1}, \ldots, w_{r-1}$. □

We conclude the description of the shift algorithm by de Fraysseix, Pach and Pollack with two remarks. The first regards the area of the produced drawing, which is at most $(2n-4) \times (n-2)$. This follows by the contour condition and by the fact that the width of the drawing cannot exceed $2n - 4$, since each vertex in the canonical order contributes two units in the width,

---

[1] Note that Lemma 7.2 considers an even stronger form of the property we need to prove planarity.

except for v_1, v_2 and v_3 which together contribute in total two units. The second remark regards the time complexity of the algorithm. A trivial upper bound is $\mathcal{O}(n^2)$. However, with a careful relative offset computation the algorithm can be implemented in linear time. The idea is to maintain at each vertex an offset-value which corresponds to the amount of shift due this particular vertex. Since the cover-relationship is a tree rooted at v_n, the total amount of shift of each vertex can then be computed by a top-down traversal of the cover-relationship tree.

> **Theorem 7.8** Let G be a maximal plane graph with n vertices. A planar straight-line grid drawing of G on a grid of size $(2n-4) \times (n-2)$ can be computed in $\mathcal{O}(n)$ time.

7.5.3 Schnyder realizers

While constructing drawing Γ_n, it is also possible to compute a 3-coloring and an orientation of the inner edges of G, known as Schnyder realizer. Initially, color (v_1, v_3) blue and direct it towards v_1, color (v_2, v_3) green and direct it towards v_2. When a vertex v_k, with $k = 4, \ldots, n$, is placed, color edge (w_ℓ, v_k) blue and direct it toward w_ℓ, color edge (v_k, w_r) green and direct it toward w_r, and color the remaining edges incident to v_k in G_k (that is, (w_i, v_k) with $i = \ell + 1, \ldots, r - 1$) red and direct them towards v_k. Note that since the coloring and the orientation is limited to inner edges of G, for $k = n$, the edges (v_1, v_n) and (v_2, v_n) incident to v_n are not colored/oriented (see Fig. 7.2b). It follows that the incoming edges of v_1, v_2 and v_n are blue, green and red, respectively. By definition, for each interior vertex v_k, the following properties hold:

- v_k has out-degree exactly one in blue, green and red; the former two are by its placement, while the latter by its coverage.

- by planarity, the clockwise order of edges incident to v_k is: outgoing blue, incoming green (if any), outgoing red, incoming blue (if any), outgoing green, incoming red (if any).
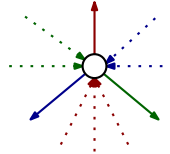
Let T_b, T_g and T_r be the subgraphs of G induced by the blue, green and red edges, respectively. Since the blue-outdegree of each interior vertex is one and since each blue edge is oriented from a vertex with a higher index in the canonical order to a vertex with a lower index in the canonical order, it follows that T_b is acyclic. The same symmetrically holds for T_g and T_r. Since a maximal plane graph has exactly $n - 3$ internal vertices and exactly $3n - 9$ internal edges (by Euler's formula) and since each of T_b, T_g and T_r spans at most $n - 2$ vertices, it follows that each of T_b, T_g and T_r has exactly $n - 3$ edges, which implies that each of T_b, T_g and T_r is a tree which spans exactly $n - 2$ vertices. This implies that every internal vertex has three edge-disjoint paths in blue, green and red towards v_1, v_2 and v_n, respectively.

7.6   The Planar Separator Theorem

Several algorithms on planar graphs are based on the divide-and-conquer approach (see Section 1.6). Such algorithms usually split an instance of a problem to subproblems that have roughly the same size, which are recursively solved and their solutions are combined to obtain a solution of the original instance. To adopt this approach to a problem, in which the input contains a planar graphs, one needs a technique to slit the input planar graph in subinstances (i.e., subgraphs) that have roughly the same size (e.g., number of vertices). We formalize this concept in the following definition.

Definition 7.8 (Separator) Let $G = (V, E)$ be an n-vertex graph. Let also a be a positive number, such that $0 < a < 1$, and $f : \mathbb{N} \to \mathbb{R}^+$ be a function. A subset $\mathcal{C}$ of the vertex-set of G is a $(f(n), a)$-separator of G if and only if

1. The cardinality of $\mathcal{C}$ is upper bounded by $f(n)$, that is, $|\mathcal{C}| \le f(n)$, and

2. Each connected component of the subgraph $G[V \setminus \mathcal{C}]$ of G induced by the vertices of $V \setminus \mathcal{C}$ has at most $a \cdot n$ vertices.

Example 7.2 A $k \times \ell$ grid graph is a planar graph whose vertices are in one-to-one correspondence with the points of a $k \times \ell$ integer grid whose bottom left corner coincides with $(1, 1)$, such the vertex corresponding to grid point (a, b), with $0 < a \le k$ and $0 < b \le \ell$ is connected to the vertex corresponding to $(a + 1, b)$, if $a + 1 \le l$, and to the vertex corresponding to $(a, b + 1)$, if $b + 1 \le \ell$.

An $\sqrt{n} \times \sqrt{n}$ grid graph has a $(\sqrt{n}, \frac{1}{2})$-separator. To see this observe that the removal of the vertices mapped to $(1, \sqrt{n}/2), \ldots, (\sqrt{n}, \sqrt{n}/2)$ results in exactly two connected components, each of which contains at most $\frac{n}{2}$ vertices; see Fig. 7.4a.

Example 7.3 Every tree has a $(1, 1/2)$-separator.

Any non-leaf vertex in a tree is a separator, that is, its removal yields at least two connected components. To find a vertex whose removal yields connected components of size at most $n/2$, we assume that our tree has at least three vertices (as otherwise, the instance is trivial) and we proceed as follows.

S.1 Let v be a non-leaf vertex, which exists by our assumption.

S.2 If the removal of v yields connected components of size at most $n/2$, then the separator has been found.

S.3 Otherwise, the removal of v yields exactly one connected component C containing more than $n/2$ vertices. Let w be the vertex of C adjacent to v. Set $v \leftarrow w$ and go to Step 2.

Since in Step 3 the size of the largest component due the removal of w is strictly smaller than the size of the largest component due the removal of v, the procedure above terminates, thus finding the claimed separator. For an example, see Fig. 7.4b.
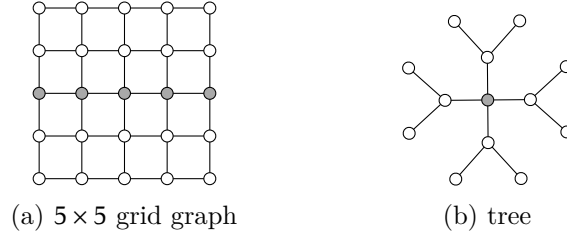
(a) 5×5 grid graph   (b) tree

Figure 7.4: Illustration of different separators (colored in gray).

To find a separator of a general planar graph, we need to introduce first few auxiliary concepts. We start with the definition of the diameter of a graph.

> **Definition 7.9** (Graph diameter) The diameter $\mathbf{diam}(G)$ of a graph $G = (V, E)$ is defined as the longest shortest distance between any two vertices of it, that is,
>
> $$\mathbf{diam}(G) = \max_{u,v \in V} \mathbf{dist}(u, v),$$
>
> where $\mathbf{dist}(u, v)$ denotes the shortest distance between u and v in G.

> **Lemma 7.3** Let $G = (V, E)$ be an n-vertex planar graph and $T = (V, E_r)$ be a spanning tree of G with $\mathbf{diam}(T) = s$. Then, G has an $(s+1, 2/3)$-separator, which can be computed in $\mathcal{O}(n)$ time.

Proof. Without loss of generality, we assume that G is maximal planar, as otherwise G can be augmented to maximal planar in $\mathcal{O}(n)$ time by only introducing edges. Let Γ be a planar drawing of G. Given a simple cycle C in G, we denote by $\mathbf{ext}(C)$ the number of vertices outside C in Γ, and by $\mathbf{int}(C)$ the number of vertices inside C in Γ.

Let e be an edge of G that does not belong to T, that is $e \in E \setminus E_r$. It follows that the subgraph of G induced by $E_r \cup \{e\}$ has a cycle $C(e)$; see Fig. 7.5a. Since the diameter of G is s, cycle $C(e)$ at most has $s + 1$ vertices (and edges). Our goal is to find an edge $e \in E \setminus E_r$, such that

$$\mathbf{ext}(C(e)) \leq 2/3n \text{ and } \mathbf{int}(C(e)) \leq 2/3n. \tag{7.10}$$

Consider an arbitrary edge e in $E \setminus E_r$. If Eq. (7.10) holds for edge e, then the proof of the lemma follows, as $C(e)$ is an $(s+1, 2/3)$-separator for G. Otherwise, either $\mathbf{ext}(C(e)) > 2/3n$ or $\mathbf{int}(C(e)) > 2/3n$ holds. By symmetry, assume without loss of generality that $\mathbf{int}(C(e)) > 2/3n$ holds. Our goal is to find an edge e' in $E \setminus E_r$, such that the following conditions hold:

C.1 the interior of $C(e')$ lies in the interior of $C(e)$ in Γ,

C.2 if $\mathbf{int}(C(e)) = \mathbf{int}(C(e'))$, then $C(e')$ contains in its interior at least one face less than $C(e)$,
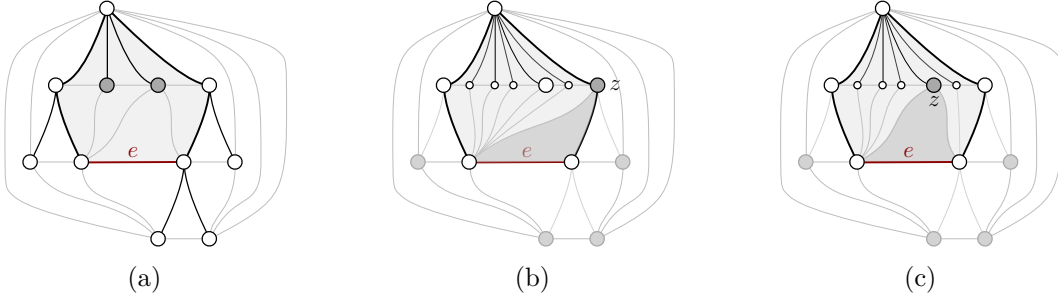
C.3 $\mathbf{ext}(C(e')) \leq 2/3n$

Figure 7.5: (a) A spanning tree (in black) T of a maximal planar graph and an edge e, which forms a cycle $C(e)$ in T. The two gray-colored vertices in the gray-highlighted region form $\text{int}(C(e))$.

Assume for the time being that there is a way to find edge e' (we will describe how to do this soon). If Eq. (7.10) holds for e', then $E_r \cup \{e'\}$ is an $(s+1, 2/3)$-separator for G. Otherwise, we set $e \leftarrow e'$ and we repeat the searching procedure above. Since $C(e)$ contains $\mathcal{O}(n)$ faces in its interior (by Euler's formula) and since $C(e)$ shrinks at each iteration of this procedure, the searching algorithm described above terminates by finding an $(s+1, 2/3)$-separator for G after at most $\mathcal{O}(n)$ iterations. The question that remains to be answered is how to compute e' in constant time.

Let x and y be the endvertices of edge e, that is, $e = (x, y)$. Let z be the vertex of G such that $\langle x, y, z \rangle$ is a face of Γ in the interior of $C(e)$. Note that vertex z is uniquely defined, since we have assumed that Γ is maximal planar and can be computed in $\mathcal{O}(1)$ time by the planar embedding of G. Since $\text{int}(C(e)) > 2/3n$, it follows that at most one of (z, x) and (z, y) may belong to $C(e)$. Based on this observation, we proceed by considering two cases:

Case 1: Either (z, x) or (z, y) belongs to $C(e)$; see Fig. 7.5b

Case 2: Neither (z, x) nor (z, y) belongs to $C(e)$; see Fig. 7.5c.

We first consider Case 1 assuming, without loss of generality, that (z, x) belongs to $C(e)$. This implies that (z, y) is a chord of $C(e)$ and therefore (z, y) does not belong to T, that is, $(z, y) \notin E_r$. In this case, we set e' to be (z, y), which implies that Conditions C.1–C.3 hold $C(e')$, since $\text{ext}(C(e)) = \text{ext}(C(e'))$, $\text{int}(C(e)) = \text{int}(C(e'))$ and $C(e')$ contains in its interior one face too few than $C(e)$.

We now consider Case 2, in which neither (z, x) nor (z, y) belongs to $C(e)$. In this case, we observe that at most one of (z, x) and (z, y) may belong to T, as otherwise $C(e) \backslash \{e\} \cup \{(z, x), (z, y)\}$ would be a cycle in T; a contradiction. This observation gives rise to two sub-cases: (a) either (z, x) or (z, y) belongs to T, and (b) neither (z, x) nor (z, y) belongs to T.

We first consider sub-case 2.(a), assuming, without loss of generality, that (z, x) belongs to T, that is, $e \in E_r$. In this case, we set e' to be (z, y), which implies $\text{ext}(C(e)) = \text{ext}(C(e'))$, $\text{int}(C(e)) = \text{int}(C(e'))$ and $C(e')$ contains in its interior one face too few than $C(e)$. Thus, $C(e')$ complies with Conditions C.1–C.3, as desired.

Next, we consider the sub-case 2.(b) in which neither (z,x) nor (z,y) belong to T, that is $(z,x) \notin E_r$ and $(z,y) \notin E_r$. Note that in this case vertex z may or may not belong to $C(e)$. Assume without loss of generality:

$$\text{int}(C(z,x)) \geqslant \text{int}(C(z,y)). \tag{7.11}$$

Then:

$$2/3n \overset{(7.10)}{<} C(e) = \text{int}(C(x,z)) + \text{int}(C(z,y)) + |C(x,z) \cap C(z,y)| - 1$$
$$\overset{(7.11)}{\leq} 2 \cdot \text{int}(C(x,z)) + |C(x,z)|$$

We proceed by setting e' to be (z,x). Thus, Conditions C.1 and C.2 are satisfied. Further, the inequality above can be written as follows:

$$\text{int}(C(e')) \geq 1/3n - 1/2|C(e')|.$$

Since $\text{ext}(C(e')) = n - \text{int}(C(e')) - |C(x,z)|$, we can conclude that Conditions C.3 is also satisfied, since:

$$\text{ext}(C(e')) \leq n - 1/3n - 1/2|C(e')| \leq 2/3n$$

.

To complete the proof of the lemma, it remains to prove that edge e' can be computed in $\mathcal{O}(1)$ time. This is clear in Cases 1 and 2.(a). However, in Case 2.(b), we have to determine whether $\text{int}(C(z,x)) \geq \text{int}(C(z,y))$ or $\text{int}(C(z,x)) < \text{int}(C(z,y))$ holds; see Eq. (7.11). To prove that this check can be done in amortised constant time, we make an observation first.

> **Observation 7.1** $\text{int}(C(e))$ can be computed by counting how many edges of T are in the interior of $C(e)$.

In view of Observation 7.1, to determine whether $\text{int}(C(z,x)) \geq \text{int}(C(z,y))$ or $\text{int}(C(z,x)) < \text{int}(C(z,y))$ holds, we visit simultaneously (e.g., with DFS) the edges of T in $C(z,x)$ and in $C(z,y)$ until of one of the two regions is completely explored. Since for this region these edges will not be visited again (in subsequent searches of edges) and since T has $\mathcal{O}(n)$ edges, it follows that edge e' can be computed in amortised constant time. $\square$

We are now ready to prove the main theorem of this section, which is commonly known as Planar Separator Theorem.

> **Theorem 7.9** (Planar Separator Theorem) An n-vertex planar graph $G = (V,E)$ admits a $(\sqrt{8n}, 2/3)$–separator, which can be computed in $\mathcal{O}(n)$ time.

Proof. Let v_0 be any vertex of G. For $1 \leq i \leq n - 1$, let L_i be the set of vertices of G whose distance from v_0 is exactly i, that is, $L_i = \{v \in V : \text{dist}(v, v_0) = i\}$. Finally, let ℓ be
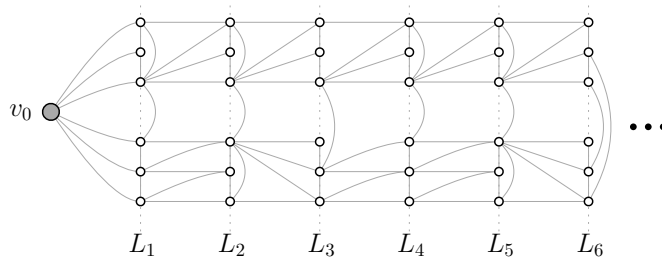
Figure 7.6: Illustration of a layering produced starting from vertex v_0.

the maximum index, such that $L_\ell \neq \emptyset$ and $L_{\ell+1} = \emptyset$, that is, $\ell = \max\{i : L_i \neq \emptyset\}$. Clearly, $L_0, L_1, \ldots, L_\ell$ is a layering of G (see Definition 5.11 and Fig. 7.6). We proceed by considering two cases:

Case 1: $2\ell + 1 \leq \sqrt{8n}$, and

Case 2: $2\ell + 1 > \sqrt{8n}$.

In Case 1, any BFS tree of G (computed starting from vertex v_0) is a spanning tree of G whose diameter is at most $2\ell + 1$. Since $2\ell + 1 \leq \sqrt{8n}$, by Lemma 7.3, G has a $(\sqrt{8n}, 2/3)$-separator.

In Case 2, we set s to be $\lceil \sqrt{n/2} \rceil$, that is $s = \lceil \sqrt{n/2} \rceil$, and we define s sets $S_0, \ldots, S_{s-1}$ as follows:

$$S_j = \bigcup_{i=1}^{\ell} \{L_i : i \equiv j \bmod s\}, \quad j = 0, 1, \ldots, s-1.$$

Let S_{j_0} be the set of minimum cardinality among $S_0, \ldots, S_{s-1}$, that is, $|S_{j_0}| \leq |S_j|$, for each $j = 0, 1, \ldots, s-1$. It follows that S_{j_0} contains at most $\lfloor n/s \rfloor = \lfloor \sqrt{2n} \rfloor$ vertices.

Consider the connected components of $G \setminus S_{j_0}$. If each of these connected components has at most $2/3n$ vertices, then S_{j_0} is a $(\lfloor \sqrt{2n} \rfloor, 2/3)$-separator. Otherwise, there exists exactly one connected component C with strictly more than $2/3n$ vertices. Assume w.l.o.g. that C is formed by $\cup_{j>j_0} S_j$. We contract the vertices of S_{j_0} in the layer above C into a single vertex. This yields a new planar graph G'. Any BFS tree of G' (computed starting from vertex v_0) has height at most $s - 1$ and diameter at most $2s - 1$. Hence, by Lemma 7.3, G' has a $(2s - 1, 2/3)$-separator. This separator together with all vertices of S_{j_0}, which are at most $\lfloor \sqrt{2n} \rfloor$, yield an $(f(n), 2/3)$-separator for G, where:

$$f(n) \leq \lfloor n/s \rfloor + 2s - 1 = \lfloor \sqrt{2n} \rfloor + 2\lceil \sqrt{n/2} \rceil - 1 \leq \sqrt{8n}. \qquad \square$$

Note that with more complicated arguments one can further improve Theorem 7.9 as follows.

> **Theorem 7.10** Let $G = (V, E)$ be an n-vertex planar graph and $\varepsilon : \mathbb{N} \to \mathbb{R}$ be a function such that $\varepsilon(n) \leq 1$ and $\varepsilon(n) \cdot n \geq 24$. Then, G has a $(\sqrt{24n/\varepsilon(n)}, \varepsilon(n))$–separator, which can be computed in $\mathcal{O}(n \log \frac{1}{\varepsilon(n)})$ time.

Example 7.4 For $\varepsilon(n) = 1/2$, the theorem above provides a $(\sqrt{48n}, 1/2)$–separator.

7.6.1 An application of the Planar Separator Theorem

In this section, we present an application of the Planar Separator Theorem to the maximum independent set problem, which is formally defined as follows.

Definition 7.10 (Maximum Independent Set Problem) The input of the maximum independent set problem consists of an undirected graph $G = (V, E)$. The output specifies a set I of vertices of G of maximum cardinality, called maximum independent set, such that no two vertices in I are adjacent, that is, each edge of G has at most one endvertex in I.

Note that the maximum independent set problem is a well-known NP-complete. In the following theorem, we consider a special case of it, called maximum planar independent set, in which the input graph is planar.

Theorem 7.11 A maximum independent set of a planar graph with n vertices can be computed in $2^{\mathcal{O}(\sqrt{n})}$ time.

Proof. Consider Algorithm 7.1, whose correctness follows from the fact that the algorithm considers all independent subsets of separator C and from the divide and conquer principle.

To estimate the time complexity of Algorithm 7.1, we denote by $T[n]$ the time needed to compute a maximum independent set for a planar instance G of the problem with n vertices. The computation of separator C of G can be done in time linear in n (by Theorem 7.9), say in time $c_1 n$. All subsets of C are at most $2^{\sqrt{8n}}$. Determining whether each of them is an independent set (that is, no two vertices in the set are connected by an edge) can be done in time linear in n, say in time $c_2 n$. Finally, each of the subsets of C that are independent yields two recursive calls of the algorithm on instances that are size at most $\frac{2}{3}n$. Thus:

$$T[n] \leq c_1 n + 2^{\sqrt{8n}} \cdot c_2 n \cdot 2T[\frac{2}{3}n] \tag{7.12}$$

To simplify the expression of Eq. (7.12), we may assume that for sufficiently large n, say for $n \geq n_0$, we have:

$$c_1 n + 2c_2 n \leq 2^{(4-\sqrt{8})\sqrt{n}}$$

while for $n < n_0$, we have:

$$c_1 n + 2c_2 n \leq c_3$$

In either case, Eq. (7.12) can be rewritten as follows:

$$T[n] \leq c_3 \cdot 2^{4\sqrt{n}} \cdot T[\frac{2}{3}n], \quad \text{with } c_3 = \mathcal{O}(1) \tag{7.13}$$

Algorithm 7.1: Maximum independent set for planar graphs.

    Input   : A planar graph G with n vertices.
    Output: A maximum independent set of G.

1 Function PlanarIS($G = (V, E)$)
2   if $|V| \leq 1$ then
3         $I \leftarrow V(G)$;
4   else
5         $I \leftarrow \emptyset$;
6         $C \leftarrow (\sqrt{8|V|}, 2/3)$-separator of G;

        // $G[X]$ denotes the subgraph of G induced by $X \subseteq V$
        // $N(X)$ denotes the vertices of G neighboring $X \subseteq V$.
7         $G[V \setminus C]$ is decomposed into two sets A and B of size at most $\frac{2}{3}|V(G)|$;

8   foreach subset S of C such that no two vertices in S are adjacent do
9             $I_A \leftarrow$ PlanarIS ($G[A - N(S)]$);
10            $I_B \leftarrow$ PlanarIS ($G[B - N(S)]$);
11   if $|S| + |I_A| + |I_B| > |I|$ then
12               $I \leftarrow S \cup I_A \cup I_B$;
13   end
14   end
15   end
16   return I;
17 end

For $n < n_0$, we may assume that $T[n] \leq c_4$, that is, for instances of the maximum independent set problem whose size is constant, a maximum independent set can be computed in constant time (e.g., by a brute force approach). Under these assumptions, the recursive formula of Eq. (7.13) can be solved as follows:

$$
\begin{aligned}
T[n] &\leq c_3 \cdot 2^{4\sqrt{n}} \cdot T[\frac{2}{3}n] \\
&\leq c_3 \cdot 2^{4\sqrt{n}} \cdot 2^{4\sqrt{2/3n}} \cdot T[\left(\frac{2}{3}\right)^2 n] \\
&\leq \ldots \\
&\leq c_3 \cdot c_4 \cdot 2^{4\sqrt{n} \cdot (1 + \sqrt{2/3} + \sqrt{2/3}^2 + \sqrt{2/3}^3 + \ldots)} \\
&\leq c_3 \cdot c_4 \cdot 2^{4\sqrt{n} \cdot \sum_{i=0}^{\infty} \left(\sqrt{2/3}\right)^i} \\
&= c_3 \cdot c_4 \cdot 2^{\frac{4\sqrt{n}}{1 - \sqrt{2/3}}}
\end{aligned}
$$

Hence, the proof of the theorem follows.   □
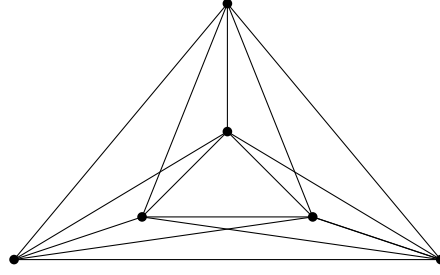
Figure 7.7: A drawing of K_6 with 3 crossings.

7.7   The Crossing Lemma

Usually, when a graph is not planar, one seeks in drawing it with as few crossings as possible. However, the problem of determining a drawing with minimum number of crossings for a (non-planar) graph is an NP-complete problem. Thus, the interest is on an estimation of the crossings that are unavoidable in any drawing of a given non-planar graph.

This is guaranteed by the so-called Crossing Lemma. In particular, for a graph G we refer to the minimum number of crossings required by any drawing of it as the crossing number of G, which is commonly denoted by $cr(G)$. Then, the Crossing Lemma provides a lower bound on $cr(G)$. The general form of this lemma is stated in the following.

> **Lemma 7.4** (Crossing Lemma) Let G be a graph with n vertices and m edges, such that $m \geq c_1 \cdot n$ edges. Then:
> $$cr(G) \geq c_2 \cdot \frac{m^3}{n^2}.$$
> where $c_1, c_2 \in \mathcal{O}(1)$.

Note that the dependency of $cr(G)$ to m^3/n^2 is unavoidable, since there exist graphs with n vertices and m edges that asymptotically require $\mathcal{O}(m^3/n^2)$ crossings in any drawing of them. Hence, the greater the value of constant c_2 in the crossing lemma is, the more valuable the corresponding lower bound is. The currently best-known value for constant c_2 is $\approx 0,0354$. However, the corresponding proof is rather involved. In the following, we give an relatively easy proof for a slightly worse constant. To this end, we need the following auxiliary lemma, whose proof is based on the following intuition. Consider a graph which has $3n - 6 + k$ edges. Then, by Theorem 7.2 (and by induction) this graph necessarily contains k edges, each of which has at least one crossing. Thus, k is a lower bound for $cr(G)$. We formalize this intuition in the following lemma.

> **Lemma 7.5** Let G be a graph with n vertices and m edges. Then:
> $$cr(G) \geq m - 3n + 6$$

Proof. Let H be the graph obtained from G by replacing the crossings of G with dummy-vertices. Clearly, graph H is planar with $n + cr(G)$ vertices and $m + 2cr(G)$ edges. Hence, by Theorem 7.2, we obtain:

$$m + 2cr(G) \le 3(n + 2cr(G)) - 6$$

Hence:

$$cr(G) \ge m - 3n + 6 \qquad \square$$

As an example, consider the complete graph K_6 on six vertices. By Lemma 7.5, it follows that

$$cr(K_6) \ge m - 3n + 6 = 15 - 18 + 6 = 3.$$

Fig. 7.7 provides a drawing with exactly 3 crossings, which implies that $cr(K_6) = 3$.

We are now ready to provide a proof for the Crossing Lemma when $c_1 = 4$ and $c_2 = 1/64 \approx 0,016$.

Theorem 7.12 Let G be a graph with n vertices and m edges, such that $m \ge 4n$ edges. Then:

$$cr(G) \ge \frac{1}{64} \cdot \frac{m^3}{n^2}.$$

Proof. Let $p \in [0,1]$ be a parameter, which we will be specified later. We proceed by defining a probabilistic subgraph G_p of G as follows. We choose every vertex of G with probability p and we let G_p to be the graph induced by the chosen vertices. We further introduced three discrete random variables n_p, m_p and c_p, which correspond to the number of vertices, edges and crossings of G_p, respectively, namely:

$$n_p = |V(G_p)|, \quad m_p = |E(G_p)|, \quad c_p = cr(G_p).$$

Since each vertex of G appears in G_p with probability p, since each edge of G appears in G_p with probability p^2, and finally since each crossing of G appears in G_p with probability p^4, the expected values of n_p, m_p and c_p are as follows:

$$E[n_p] = pn, \quad E[m_p] = p^2 m, \quad E[c_p] = p^4 cr(G).$$

By Lemma 7.5 applied on graph G_p, we obtain:

$$cr(G_p) \ge m_p - (3n_p - 6) \ge m_p - 3n_p.$$

By linearity of expectations, we further obtain:

$$E[c_p] \ge E[m_p] - 3E[n_p] \quad \Rightarrow \quad p^4 cr(G) \ge p^2 m - 3pn$$

Thus:

$$cr(G) \ge \frac{m}{p^2} - \frac{3n}{p^3}$$

To complete the proof, we choose $p = \frac{4n}{m} \le 1$, which implies:

$$cr(G) \ge \frac{1}{64} \cdot \frac{m^3}{n^2} \qquad \square$$

7.8   Concluding remarks

We conclude this chapter with a couple of remarks. First, we remark that there are several algorithms for testing whether a given graph is planar. The first one was given by Even, Lempel and Cederbaum back in 1967. Almost a decade afterwards, Hopcroft and Tarjan (in 1974) presented the first linear time such algorithm. Independently, Booth and Luecker in 1976 observed that the first algorithm by Even, Lempel and Cederbaum can also be implemented in linear time. However, all algorithm that are currently known in the literature for testing whether a given graph is planar are quite complicated (and thus beyond the scope of this chapter). The second remark is related to the crossing numbers of the complete graphs and of the complete bipartite graphs, for which there exist two long-standing open conjectures by Guy and by Zaradkiewicz, respectively, asserting the following:

Conjecture 7.1 (Guy 1972) $cr(K_n) = \frac{1}{4}\lfloor\frac{n}{2}\rfloor\lfloor\frac{n-1}{2}\rfloor\lfloor\frac{n-2}{2}\rfloor\lfloor\frac{n-3}{2}\rfloor$

Conjecture 7.2 (Zaradkiewicz 1954) $cr(K_{n,m}) = \lfloor\frac{n}{2}\rfloor\lfloor\frac{n-1}{2}\rfloor\lfloor\frac{m}{2}\rfloor\lfloor\frac{m-1}{2}\rfloor$

Exercises

Exercise 7.1   For each of the following graphs prove or disprove that the graph is planar.
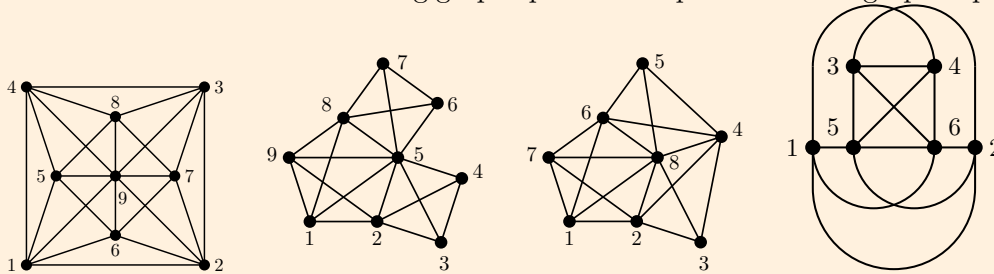
Exercise 7.2   A k-regular graph is a connected graph whose vertices have degree exactly $k \geq 1$.

a) What is the number of edges of a k-regular graph with n vertices?

b) Prove that a k-regular graph with $k \geq 6$ cannot be planar.

c) Present an infinite class of 3-regular planar graphs.

d) Present an infinite class of 4-regular planar graphs.

e) Present an infinite class of 5-regular planar graphs.

Exercise 7.3 Consider a planar embedded graph with n vertices and m edges, which has no trianglular faces (that is, it does not have a face containing less than 4 vertices).

a) Prove that $m \leq 2n - 4$.

b) What is the maximum number of faces that such a planar graph can have?

c) Present an infinite class of planar graphs, whose faces are all of length 4.

Exercise 7.4 The complement of a graph $G = (V, E)$ is the graph $G^c = (V, V \times V - E)$ that is defined on the same set of vertices such that any two vertices are adjacent in G^c if and only if they are not adjacent in G.

a) Prove that if G is a planar graph with at least 11 vertices then G^c is non-planar.

b) Present a planar graph on 8 vertices whose complement is planar.

Exercise 7.5 Recall that an outerplanar graph is a graph that admits a planar drawing in which all vertices belong to the unbounded face of the drawing (see Definition 7.4).

a) Prove that an outerplanar graph with n vertices has at most $2n - 3$ edges.

b) Prove that a biconnected outerplanar graph has one vertex of degree 2.

c) Prove that every biconnected outerplanar graph is 3-vertex colorable.

d) Prove that every biconnected outerplanar n-vertex graph has a $(3, \frac{1}{2})$-separator.

e) Prove that every biconnected outerplanar n-vertex graph has a $(2, \frac{2}{3})$-separator.

Hint: If an outerplanar graph is biconnected, then its outerface is delimited by a simple cycle.

Exercise 7.6 Compute a planar straight-line drawing of the graph of Fig. 7.2a using the shift method by de Fraysseix, Pach and Pollack.

Exercise 7.7 Give a family of maximal planar n-vertex graphs with approximately $(\frac{n}{2} - 2)!$ canonical orderings.

Exercise 7.8 Consider a planar graph, which does not contain faces of length 3 and 4.

a) Show that such an n-vertex graph has at most $\frac{5}{3}(n - 2)$ edges.

b) Show that there exist infinitely many values of n such that there exists an n-vertex planar graph, which does not contain faces of length 3 and 4, and which has exactly $\frac{5}{3}(n - 2)$ edges.

Exercise 7.9   Use dynamic programming to determine a maximum independent set of an n-vertex tree in time polynomial in n. Analyze the running time and argue about the correctness of your algorithm.

Exercise 7.10   Let $T = (V, E, w)$ be an edge-weighted n-vertex tree, where $w : V \to \mathbb{R}^+$.

a) Present an $\mathcal{O}(n)$-time algorithm to determine a vertex of T whose removal yields a number of connected components, such that the total weight of each of them is at most half the total weight of T.

b) Analyze the running time and argue about the correctness of your algorithm.

Exercise 7.11   Let P be a simple polygon with n vertices $p_1, \dots, p_n$. Present an algorithm that computes in $\mathcal{O}(n \log n)$ time a pair of vertices p_i and p_j with $i < j$ such that the polygon with vertices $p_1, \dots p_i, p_j, \dots p_n$ and the polygon with vertices $p_i, \dots p_j$ are simple polygons, each with at most $\lfloor 2n/3 \rfloor + 2$ vertices.
Hint: Triangulating the interior of a simple polygon can be done in $\mathcal{O}(n \log n)$ time.

Exercise 7.12   A 1-planar drawing of a graph is a straight-line drawing, in which each edge is crossed at most once. A 1-planar graph is a graph that admits a 1-planar drawing.

a) Prove or disprove: K_6 and $K_{2,2,2,2}$ admit 1-planar drawings with 3 and 6 crossings, respectively.

b) Prove or disprove: The crossing number of K_6 is 2 (without using R. Guy's conjecture).

Further reading

– Chapter 9 from: T. Nishizeki and N. Chiba. Planar Graphs: Theory and Algorithms. North-Holland, 1988.

– https://www.ti.inf.ethz.ch/ew/lehre/GA10/lec-separators.pdf

DRAFT

8. Linear Programming

This summary will focus on introducing linear programming (LP), an approach which deals with the problem of optimizing a linear objective function with respect to a set of variables, subject to a number of constrains that are linear equalities or inequality of the variables. We will now discuss a well-known algorithm for solving linear programs, which is called simplex. The rest of this chapter is organized as follows: In Section 8.1, we introduce the linear programming problem formally, and we further demonstrate how one can solve a linear program with up to two variables via a graphical representation of it. Section 8.2 introduces the general form of a linear program. In Section 8.2.1, we formulate the maximum flow problem that we discussed in Chapter 5 as a linear program. In Section 8.3, we discuss the first step of the simplex algorithm, which focuses on transforming the input linear program into an equivalent one in a so-called standard or canonical form. In Section 8.4, we first present the main ideas of the simplex algorithm by an example and then we formalize them in the form of an algorithm. In Section 8.5, we introduce an auxiliary linear program that forms the so-called Phase I of the simplex algorithm, whose goal is to find an initial feasible solution of a linear program. In Section 8.6, we discuss issues related to the time-complexity of the simplex algorithm. In Section 8.7, we introduce duality first by an example (to get the intuition of the program) and then formally (to demonstrate how one can use it to verify optimal solutions). We discuss integer linear programs and show $\mathcal{NP}$-completeness in Section 8.9.

8.1   Linear programming

Linear programming (LP, for short; also called linear optimization) is a method to achieve the best outcome (such as maximum profit or lowest cost) in a mathematical model whose requirements are represented by linear relationships. To make this consent clear, let us consider an example.

> Example 8.1   Assume that a factory produces two products; tables and chairs. Each table that is produced requires one unit of metal and three units of wood; each chair, on the other hand, requires two units of metal and one unit of wood. The factory has 600 units of metal and 900 units of wood available. The profit of a single table is 100 €;

the profit of a single chair is 100 €. The goal is to compute the number of tables and the number of chairs that the factory has to produce, in order to maximize its profit. Note that there is a capacity on the materials which do not allow to produce infinitely many tables or chairs.

To formulate this problem as an optimization problem, denote by x_1 and by x_2 the total number of tables and the total number of chairs, respectively, that the factory has to produce. Since the goal is to maximum the profit, one would naturally seek in maximizing the following expression, which describes the total profit of the company:

$$100x_1 + 100x_2$$

Of course, the total number of tables and the total number of chairs to be produced should not be negative, which can be guaranteed by a constraint of the form:

$$x_1, x_2 \geq 0$$

Since each table that is produced requires one unit of metal and three units of wood, each chair requires two units of metal and one unit of wood, and the factory has 600 units of metal and 900 units of wood available, we also need to guarantee the following two constraints:

$$x_1 + 2x_2 \leq 600$$

$$3x_1 + x_2 \leq 900$$

Hence, by putting everything together we obtain the following formulation of Example 8.1, which is our first linear program:

$$\begin{aligned}
\text{maximize} \quad & 100x_1 + 100x_2 \\
\text{subject to} \quad & x_1 + 2x_2 \leq 600 \\
& 3x_1 + x_2 \leq 900 \\
& x_1, x_2 \geq 0
\end{aligned}$$

More in general, a linear program contains a linear objective function, which should be minimized or maximized (e.g., in cases of costs or profits, respectively), subject to several linear constraints.

8.1.1   A solution through graphical representation

The next natural question to ask is how would one solve a linear program, e.g., as the one we obtained in the previous section. The most natural approach for a mathematician to follow is to visualize the constraints and then to try to find the optimal solution. In particular, since our linear program has only two variables, each constraint represents a half-plane in the Euclidean plane. To visualize these half-planes, one needs to find the line that bounds each half-plane, i.e., where the constraint holds by equality.

– The line that bounds the first constraint $x_1 + 2x_2 \leq 600$ of our linear program is defined as $x_1 + 2x_2 = 600$, and it is not difficult to see that it passes through the points $(600, 0)$ and $(0, 300)$; blue colored in Fig. 8.1.
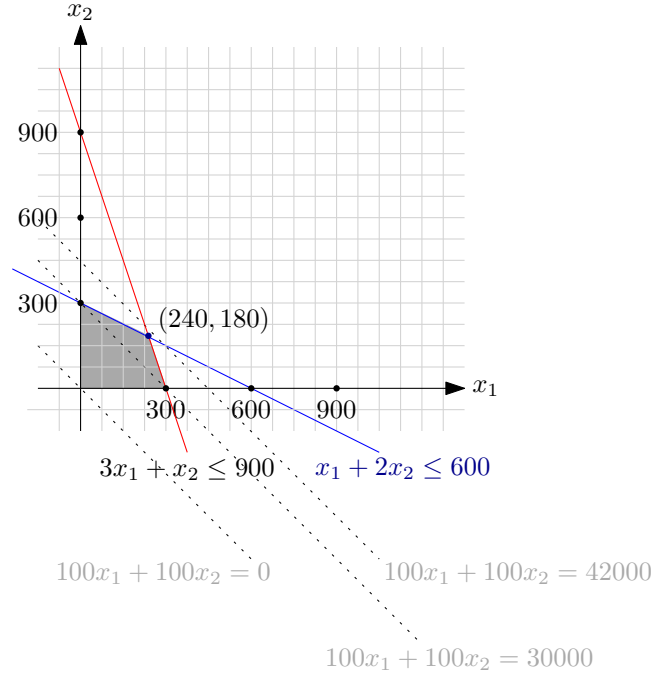
DRAFT

Figure 8.1: Graphical solution of the model.

– Similarly, the line that bounds the second constraint $3x_1 + x_2 \leq 900$ of our linear program is defined as $3x_1 + x_2 = 900$, and it is not difficult to see that it passes through the points $(0, 900)$ and $(300, 0)$; red colored in Fig. 8.1.

– To determine which side of the line defines the half-plane of each constrain, we can easily try a point in the constraint to see if it is in the half-plane or not. For instance, if use the point $(0, 0)$, we will quickly realize that this points belongs to both half-planes.

It follows that any (i.e., not necessarily the optimal) solution of our linear program must lie within the gray-colored region of Fig. 8.1. This region in a sense describes all feasible solutions, i.e., all solutions that conform with the constraints of the program.

The next question to answer is how to determine the optimal solution. To this end, we have to find the maximum z for which the line $z = 100x_1 + 100x_2$ (which expresses the profit) has a non-empty intersection with the gray shaded region (which corresponds to the feasible solution space). As you can see from Fig. 8.1, the maximum possible value is the intersection point $(240, 180)$ of the two constraints $x_1 + 2x_2 \leq 600$ and $3x_1 + x_2 \leq 900$. In this case, the total profit is equal to 42000.

We conclude this section by investigating different cases that may occur while searching for the optimal solution of a linear program through its graphical representation. In particular, consider the three cases illustrated in Fig. 8.2.

– Case a: In this case, the feasible solution space (gray shaded) is bounded and there is a unique optimal solution given at the intersection point of the blue line and the red line.
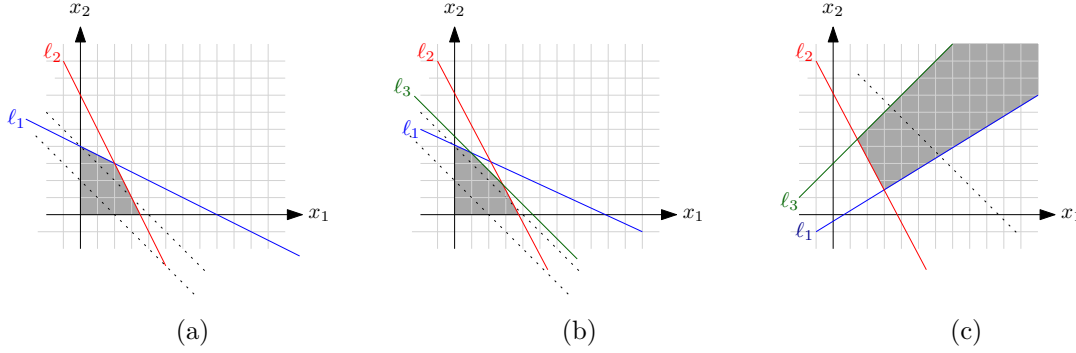
Figure 8.2: Different cases that may arise in a graphical representation of a linear program.

– Case b: This case is similar to Case a, but instead of a single optimal solution there exist infinitely many ones. In particular, all points that are along the green line and between the two intersection points of it with the blue line and the red line correspond to an optimal solution. As a result, the solution space is bounded and there are infinitely many optimal solutions.

– Case c: In this case, the feasible solution space is unbounded. Here, an optimal solution may exist or may not exist. In particular, notice that in the case where a solution exists, we may have infinitely many optimal solutions as in Case b.

The conclusion out of the three cases described above is that whenever an optimal solution exists, then there exists also one that corresponds to a corner of the feasible solution space. Such a solution is called basic. Another observation that is immediate is that when the linear program has more than two variables then finding the optimal solution through the graphical representation of the program becomes difficult.

8.2 The general form of a linear program

As already mention, a linear program, in its general form, may either maximize or minimize a liner objective function. In the following, we assume that the program consists of n variables, which we denote by $x_1, x_2, \ldots, x_n$. Another assumption that we make is that there exist in total m constraints, each of which has in its left-hand-side a linear function on $x_1, x_2, \ldots, x_n$ and in its right-hand-side a constant value. Furthermore, each constraint may be in the form of less than or equal ($\leq$), greater than or equal ($\geq$), or just equal ($=$). Besides these constraints, there may also exist a non-negativity constraint for each of the variables of the program. Putting everything together we obtain.

$$
\begin{aligned}
\text{maximize} \quad & c_1 x_1 + c_2 x_2 + \cdots + c_n x_n \\
\text{subject to} \quad & a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n \quad (\leq, =, \geq)\ b_1 \\
& \cdots \\
& a_{m1} x_1 + a_{m2} x_2 + \cdots + a_{mn} x_n \ (\leq, =, \geq)\ b_m \\
& x_1, \cdots, x_n \geq 0
\end{aligned}
$$

8.2.1   Formulating problems as linear programs

In the following, we demonstrate by an example how one can formulate a problem as a linear program. We use as example the maximum flow program, which we have introduced in Chapter 5. To do so, for each edge $(u,v) \in E$ of the network G, we introduce a variable $f(u,v)$. To formulate the capacity constraint, we introduce the following two (linear) constraints:

$$f(u,v) \geq 0 \text{ and } f(u,v) \leq c(u,v), \quad \forall (u,v) \in E$$

The flow conservation constraint is similarly formulated as a linear constraint as follows:

$$\sum_{(u,v) \in E} f(u,v) = \sum_{(v,u) \in E} f(v,u), \quad \forall u \in V \setminus \{s,t\}$$

Hence, putting everything together we obtain:

$$\text{maximize} \sum_{(s,v) \in E} f(s,v)$$

$$\text{subject to}$$

$$f(u,v) \leq c(u,v), \forall (u,v) \in E$$

$$\sum_{(u,v) \in E} f(u,v) = \sum_{(v,u) \in E} f(v,u), \forall u \in V \setminus \{s,t\}$$

$$f(u,v) \geq 0, \forall (u,v) \in E$$

8.3   Linear programs in standard or canonical form

A quite common algorithm to solve linear programs is the so-called simplex algorithm (see Section 8.4), which assumes that the input linear program (i.e., the one to be solved) is in a so-called standard or canonical form. A linear program in this form has three special properties:

P.1 The objective function is a maximization one.

P.2 Each constraint is of the form less or equal ($\leq$).

P.3 All variables satisfy the non-negativity constraint, i.e., all variables should not be negative.

Hence, a linear program in standard form is as follows:

$$
\begin{aligned}
\text{maximize} \quad & c_1 x_1 + c_2 x_2 + \cdots + c_n x_n \\
\text{subject to} \quad & a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n \leq b_1 \\
& \cdots \\
& a_{m1} x_1 + a_{m2} x_2 + \cdots + a_{mn} x_n \leq b_m \\
& x_1, \cdots, x_n \geq 0
\end{aligned}
$$

A linear program in standard form can be easily rewritten in the so-called matrix form, as follows.

$$
\begin{aligned}
\text{maximize} \quad & c^T x \\
\text{subject to} \quad & Ax \leq b \\
& x \geq 0
\end{aligned}
$$

where:

$$
A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{2,2} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}
$$

The next reasonable question to ask is whether one can always transform an input linear program to one in canonical form. To this end, we need the following definition of equivalence between two linear programs.

> **Definition 8.1** Two maximization (or minimization) linear programs L_1 and L_2 are equivalent if and only if for each feasible solution of L_1 with objective value z, there is a feasible solution of L_2 with objective value z, and vice versa.

> **Definition 8.2** A maximization linear program L_1 is equivalent to a minimization problem L_2 if and only if for each feasible solution of L_1 with objective value z, there is a feasible solution of L_2 with objective value z, and vice versa.

With these two definitions in mind, we are now ready to prove the following theorem, which answers in the positive the aforementioned question.

> **Theorem 8.1** Every linear program has an equivalent one in standard form.

Proof. To prove the theorem, we show how to transform an input linear program, which is not necessarily in standard form, to an equivalent one in standard form, i.e., that complies with Properties P.1–P.3.

P.1 If the input linear program is a minimization one, we can easily rewrite its objective function as a maximization one by multiply it with -1, in other words:

$$
\text{minimize } c_1 x_1 + \cdots + c_n x_n \quad \Longleftrightarrow \quad \text{maximize } -c_1 x_1 - \cdots - c_n x_n
$$

P.2 If the input linear program has an opposite direction inequality, we can multiply by -1 both its sides to transform it to one that conforms with the ones allowed by the standard form, namely:

$$
a_{j1} x_1 + a_{j2} x_2 + \ldots + a_{jn} x_n \geq b_j \quad \Longleftrightarrow \quad -a_{j1} x_1 - a_{j2} x_2 - \ldots - a_{jn} x_n \leq -b_j
$$

If the input linear program contains an equality, we can replace it by two inequalities as
following:

$$a_{j1}x_1 + a_{j2}x_2 + \ldots + a_{jn}x_n = b_j \iff \begin{aligned} a_{j1}x_1 + a_{j2}x_2 + \ldots + a_{jn}x_n \leq b_j \\ -a_{j1}x_1 - a_{j2}x_2 - \ldots - a_{jn}x_n \leq -b_j \end{aligned}$$

P.3 Finally, if a variable x_j of the input linear program does not obey the non-negativity
constraint, we can replace it with the difference $x_j' - x_j''$ between two variables x_j' and x_j''
which are both non-negative, i.e., $x_j' \geq 0$ and $x_j'' \geq 0$.

☐

> **Example 8.2**   Transform the following linear program in standard form:
>
> $$\begin{aligned} \text{minimize} \quad & -x_1 - x_2 \\ \text{subject to} \quad & 2x_1 - x_2 \geq 3 \\ & x_1 \geq 0 \end{aligned}$$

Answer. By applying the rules described in the proof of Theorem 8.1, we obtain:

$$\begin{aligned} \text{maximize} \quad & x_1 + x_2' - x_2'' \\ \text{subject to} \quad & -2x_1 + x_2' - x_2'' \leq -3 \\ & x_1, x_2', x_2'' \geq 0 \end{aligned}$$

☐

8.4   The simplex method

In this section, we are introducing the simplex algorithm for solving linear programs. We first
demonstrate the principle ideas of this algorithm by the following example of a linear program
(note that we neglect the example of Section 8.1, because it contains only two variables).

$$\begin{aligned} \text{maximize} \quad & 5x_1 + 4x_2 + 3x_3 \\ \text{subject to} \quad & 2x_1 + 3x_2 + x_3 \leq 5 \\ & 4x_1 + x_2 + 2x_3 \leq 11 \\ & 3x_1 + 4x_2 + 2x_3 \leq 8 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

As already mentioned, the first step of the simplex algorithm is to transform the input problem
into a standard form; for our example, it is not difficult to see that the program is already in
the standard form.

DRAFT

The second step is to introduce a name for the objective function. This is not an obligatory step, but it will be convenient for the description. So, in the following we call the objective function ζ, namely:

$$\zeta = 5x - 1 + 4x_2 + 3x_3$$

In the third step, we introduce a so-called slack variable for each constraint of the input program, which expresses the difference of the right-hand side to the left-hand side of the inequality corresponding to the constraint. Hence, each of these new variables is not negative. For example, the slack variable corresponding to the first constraint of our example would be defined as follows:

$$x_4 = 5 - 2x_1 - 3x_2 - x_3; \; x_4 \geq 0$$

By following the aforementioned three steps, we can rewrite the linear program of our example in the following form, which is called slack form or initial dictionary:

$\zeta = 5x_1 + 4x_2 + 3x_3$
$x_4 = 5 - 2x_1 - 3x_2 - x_3$
$x_5 = 11 - 4x_1 - x_2 - x_3$
$x_6 = 8 - 3x_1 - 4x_2 - 2x_3$
$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0$

Assuming that an initial feasible solution of the linear program is known, the idea of the simplex algorithm is to find a new feasible solution, which is at least as good as the current one, and to iterate this procedure as long as possible.

So for this algorithm to start, we need an initial feasible solution. To simplify this step, we make a simple observation that is tailored for our example: If the right side of each inequality is non negative (i.e., $b_1, b_2, \ldots, b_n \geq 0$), then a feasible solution is easy to be found, since $x_1 = x_2 = \ldots = x_n = 0$ is trivially such one (for which the value of ζ becomes 0). For our program, this implies that the following solution is feasible:

$$x_1 = x_2 = x_3 = 0 \Rightarrow x_4 = 5, x_5 = 11, x_6 = 8$$

Since we now have a feasible solution for the simplex algorithm to start, the next reasonable question to ask is how one can improve this solution (as this is the idea of the simplex algorithm), namely, how can we have to find a better solution?

In our example, we can see that if we increase x_1, the value of ζ will increase as well. This is because x_1 has a positive coefficient in ζ:

$$\zeta = 5x_1 + 4x_2 + 3x_3$$

However, we cannot arbitrarily increase x_1. We have to ensure that all variables (in particular, x_4, x_5, x_6) of our program stay non-negative. So, the question is how much we can increase x_1 such that all other variables stay non-negative. We know that in our solution $x_2 = x_3 = 0$ holds. By this, we obtain:

$x_4 \geq 0 \iff 5 - 2x_1 \geq 0 \iff x_1 \leq 5/2 = 2.5$
$x_5 \geq 0 \iff 11 - 4x_1 \geq 0 \iff x_1 \leq 11/4 = 2.75$
$x_6 \geq 0 \iff 8 - 3x_1 \geq 0 \iff x_1 \leq 8/3 \approx 2.66$

Hence, we cannot increase x_1 more than 2.5, as otherwise one of x_4, x_5, x_6 will become negative. Since we seek to maximize the value of ζ, we proceed to increase x_1 to 2.5, which implies a new solution, as follows:

$$x_1 = 2.5, x_2 = x_3 = x_4 = 0, x_5 = 1 \Longrightarrow \zeta = 25/2$$

So, at this point we have an improved solution with respect to the one that we started the simplex algorithm. So, now could again ask whether this solution be further improved. To find a way to answer this question, we make a critical observation, which is central to the simplex algorithm:

– the first step was easy, because we had a set of zero-value variables, and

– the remaining ones (and also the objective function) were expressed in terms of them.

So, it reasonable to try to do the same. In this case, we have to express x_1, x_5, x_6 in terms of x_2, x_3, x_4. First, we do so for variable x_1:

$x_4 = 5 - 2x_1 - 3x_3 - x_3 \Longleftrightarrow$
$2x_1 = 5 - x_4 - 3x_2 + x_3 \Longleftrightarrow$
$x_1 = 5/2 - 3/2x_2 - 1/2x_3 - 1/2x_4$

To express x_5 and x_6 in terms of x_2, x_3, x_4, we replace in the equalities of x_5 and x_6 variable x_1 with the computed expression above; we do the same also with the objective function ζ.

$x_5 = 11 - 4x_1 - x_2 - 2x_3 \Longleftrightarrow$
$x_5 = 11 - 4(5/2 - 3/2x_2 - 1/2x_3 - 1/2x_4) - x_2 - 2x_3 \Longleftrightarrow$
$x_5 = 1 + 5x_2 + 2x_4$

$x_6 = 8 - 3x_1 - 4x_2 - 2x_3 \Longleftrightarrow$
$x_6 = 8 - 3(5/2 - 3/2x_2 - 1/2x_3 - 1/2x_4) - 4x_2 - 2x_3 \Longleftrightarrow$
$x_6 = 1/2 + 1/2x_2 - 1/2x_3 + 3/2x_4$

$\zeta = 5x_1 = 4x_2 + 3x_3 \Longleftrightarrow$
$\zeta = 5(5/2 - 3/2x_2 - 1/2x_3 - 1/2x_4) + 4x_2 + 3x_3 \Longleftrightarrow$
$\zeta = 25/2 - 7/2x_2 + 1/2x_3 - 5/2x_4$

The aforementioned computations give rise to a new dictionary:

$\zeta = 25/2 - 7/2x_2 + 1/2x_3 - 5/2x_4$
$x_1 = 5/2 - 3/2x_2 - 1/2x_3 - 1/2x_4$
$x_5 = 1 + 5x_2 + 2x_4$
$x_6 = 1/2 + 1/2x_2 - 1/2x_3 + 3/2x_4$
$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0$

Again, we observe that variable x_3 has positive coefficient in ζ. Hence, if we increase its value, then this will also lead to an increment of the objective function ζ. Note that this will not happen if we increase the value of x_2 or x_4, because they have negative coefficients in ζ.

However, we have to ensure again that variables x_1, x_5 and x_6 must stay non-negative. Recall that $x_2 = x_4 = 0$. Hence:

$x_1 \geq 0 \iff 5/2 - 1/2x_3 \geq 0 \iff x_3 \leq 5$
$x_5 \geq 0$: -
$x_6 \geq 0 \iff 1/2 - 1/2x_3 \geq 0 \iff x_3 \leq 1$

Note that since there is no dependency between x_3 and x_5, x_3 can increase infinitely without making x_5 negative. Hence, by the other two conditions x_3 can increase at most up to 1. Thus, we obtain a new solution:

$$x_1 = 2, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 1, x_6 = 0 \implies \zeta = 13$$

Now, we know how to continue. To determine whether the solution above can be further improved, we will express x_1, x_3 and x_5 in terms of x_2, x_4 and x_6, since x_1, x_3 and x_5 have non-zero values, while x_2, x_4 and x_6 have indeed zero values.

$x_6 = 1/2x + 1/2x_2 - 1/2x_3 + 3/2x_4 \iff$
$1/2x_3 = 1/2 + 1/2x_2 + 3/2x_4 - x_6 \iff$
$x_3 = 1 + x_2 + 3x_4 - 2x_6$

$x_1 = 5/2 - 3/2x_2 - 1/2x_3 - 1/2x_4 \iff$
$x_1 = 5/2 - 3/2x_2 - 1/2(1 + x_2 + 3x_4 - 2x_6) - 1/2x_4 \iff$
$x_1 = 2 - 2x_2 - 2x_4 + x_6$

$x_5 = 1 + 5x_2 + 2x_4$

$\zeta = 25/2 - 7/2x_2 + 1/2x_3 - 5/2x_4 \iff$
$\zeta = 25/2 - 7/2x_2 + 1/2(1 + x_2 + 3x_4 - 2x_6) - 5/2x_4 \iff$
$\zeta = 13 - 3x_2 - x_4 - x_6$

The aforementioned computations give rise to a new dictionary:

$\zeta = 13 - 3x_2 - x_4 - x_6$
$x_1 = 2 - 2x_2 - 2x_4 + x_6$
$x_3 = 1 + x_2 + 3x_4 - 2x_6$
$x_5 = 1 + 5x_2 + 2x_4$
$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0$

As we can see here, we cannot further increase the value of the objective function ζ. In fact, increasing x_2, x_4, or x_6 leads to a decrement of the objective function ζ, while increasing x_2, x_4, or x_6 is not possible, because of the last constraint and the fact that their value is zero. As a result, $\zeta = 13$ is the optimal solution.

At this point, the principle ideas of the simplex algorithm should be clear. So, in the following we generalize them. As already mentioned at the beginning, we assume that we have a linear

program in standard form.

$$
\begin{aligned}
\text{maximize} \quad & c_1 x_1 + c_2 x_2 + \cdots + c_n x_n \\
\text{subject to} \quad & a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n \ \le b_1 \\
& \cdots \\
& a_{m1} x_1 + a_{m2} x_2 + \cdots + a_{mn} x_n \le b_m \\
& x_1, \cdots, x_n \ge 0
\end{aligned}
$$

We assume that for each $1 \le i \le m$ it holds that $b_i \ge 0$, which implies that $x_i = 0$, $1 \le i \le n$ is a feasible solution. We will relax this assumption later, when we will demonstrate how one can compute an initial feasible solution with a different approach. The next step is to turn the standard form to slack form by introducing m slack variables as follows:

$$
\begin{aligned}
\zeta &= c_1 x_1 + c_2 x_2 + \ldots + c_n x_n \\
x_{n+i} &= b_i - a_{i1} x_1 - a_{i2} x_2 - \ldots - a_{in} x_n; \quad i = 1, \ldots, m \\
x_i &\ge 0; \quad i = 1, \ldots, n+m
\end{aligned}
$$

The aforementioned is the initial dictionary. In general, each dictionary consists of m basic (i.e., non-zero valued) and n non-basic variables (i.e., zero valued), such that each basic variable is expressed as a linear combination of the non-basic variables. For the initial dictionary, this is illustrated in the following.

$$
\underset{\text{basic}}{\underbrace{x_{n+i}}} = b_i - \underset{\text{non-basic}}{\underbrace{a_{i1} x_1 - a_{i2} x_2 - \ldots - a_{in} x_n}}; \quad i = 1, \ldots, m
$$

More general, assuming that B is the set of the indices of the basic variables, while N is the set of the indices of the non-basic variables (initially $B = \{1, \ldots, n\}$, $N = \{n+1, \ldots, m\}$), each intermediate dictionary can be written as follows, where $\hat{\zeta}_0$ is the value of the objective function at this dictionary:

$$
\begin{aligned}
\zeta &= \hat{\zeta}_0 + \sum_{i \in N} \hat{c}_i x_i \\
x_i &= \hat{b}_i - \sum_{j \in N} \hat{a}_{ij} \, x_j \ \forall i \in B \\
x_i &\ge 0, \ \forall i \in N \cup B
\end{aligned}
$$

In each step of the simplex method, exactly one variable from non-basic becomes basic (entering variable) and exactly one variable from basic becomes non-basic (leaving variable). The entering variable was in our example chosen so to increase the value of the objective function, i.e., it had a positive coefficient in the objective function ζ. So, if we consider the following set of indices of variables, the index of the entering variable must belong to this set.

$$
N^+ = \{j \in N; \hat{c}_j > 0\}
$$

Clearly, if $N^+ = \emptyset$, then $\zeta = \hat{\zeta}_0$ is optimal. Notably, if $|N^+| > 1$, there is even a choice on the selection of the entering variable. A common rule by Bland suggests that whenever there is a choice on the entering variable, then the one with the smallest index must be chosen. In the following, we assume that there is an entering variable, which we denote by x_k.

To determine the leaving variable, we have to determine the maximum increment of x_k, such that none of the variables (in particular, none of the basic variables) of the problem becomes negative. In other words, since each non-basic variable has zero value, we have to ensure the following:

$$x_i \geq 0, \ \forall i \in B \Longrightarrow \hat{a}_{ik} x_k \leq \hat{b}_i \ \forall i \in B$$

Then the leaving variable is the one that restricts x_k the most. In particular, this variable corresponds to the minimum entry of the following set:

$$\{\hat{b}_i / \hat{a}_{ik}; i \in B \text{ and } \hat{a}_{ik} > 0\}$$

Of course, if none of the constraints upper bounds x_k (that is, $\hat{a}_{ik} \leq 0, \ \forall i \in B$), then x_k can increase arbitrary without making any of the basic variables negative, and thus we may conclude that the problem is unbounded. This completes the description of the simplex method, under the assumption that an initial feasible solution can be easily found by take advantage of the fact that $b_1, b_2, \ldots, b_m \geq 0$, which of course does not hold for all linear programs.

8.5   Phase I of simplex algorithm

Phase I of simplex algorithm focuses on finding an initial feasible solution for the simplex algorithm; recall that the main idea of the simplex algorithm is to iteratively improve upon an already computed solution. Hence, one needs an initial such solution in order to put forward the simplex algorithm.

Recall that in the previous sectio, we introduced the simplex algorithm and we showed how to apply it in order to solve a linear program with the additional restriction that $b_j \geq 0$ holds for each $1 \leq j \leq m$ when the program is written in its canonical form. With this restriction, it was easy to compute an initial feasible solution; namely $x_i = 0$ for each $1 \leq i \leq n$. In the following, we will relax this constraint so that each b_j might be negative as well. Thus, finding an initial feasible solution is no more trivial.

The idea is that we introduce an auxiliary linear program for which it is easy to find a feasible solution. The optimal solution of it should then imply a feasible solution for the original problem. Next, we formalize this idea.

The auxiliary linear program is defined as follows:

$$
\begin{aligned}
\text{maximize} \quad & -x_0 \\
\text{subject to} \quad & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n - x_0 \leq b_1 \\
& a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n - x_0 \leq b_2 \\
& \cdots \\
& a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n - x_0 \leq b_m \\
& x_0, \cdots, x_n \geq 0
\end{aligned}
$$

We can observe that an initial feasible solution of the auxiliary linear program can easily be found, e.g.,

$$\text{set } x_1 = \cdots = x_n = 0 \text{ and pick } x_0 \text{ arbitrarily large.}$$

Furthermore, it is not difficult to see that the optimal solution has value at most 0, since x_0 is non-negative but $-x_0$ has to be maximized. In the following theorem, we prove that the original linear program is feasible if the auxiliary program is feasible with $x_0 = 0$.

Theorem 8.2 Let L be a linear program in its standard form. Then, L is feasible if and only if the auxiliary program L_a to L is feasible with $x_0 = 0$.

Proof. We show both directions of the theorem separately. For the forward direction, let $(x_1, \cdots, x_n)$ be a feasible solution to the original program L. Then of course $(x_0, x_1, \cdots, x_n)$ with $x_0 = 0$ is a feasible solution for the auxiliary linear program L_a, since for every constraint it holds that:

$$
\begin{aligned}
& a_{i1}x_1 + \cdots + a_{in}x_n - x_0 \leq b_i \\
\Leftrightarrow\ & a_{i1}x_1 + \cdots + a_{in}x_n - 0\ \leq b_i \\
\Leftrightarrow\ & a_{i1}x_1 + \cdots + a_{in}x_n \qquad \leq b_i
\end{aligned}
$$

Hence, all constraints of L_a are satisfied, since $(x_1, \cdots, x_n)$ is a feasible solution to L. For the other direction, let $(x_0, x_1, \cdots, x_n)$ be a feasible solution of L_a with $x_0 = 0$. Again with the same conversion as before just in the opposite direction, we may conclude that every constraint of L is satisfied since $x_0 = 0$ and the solution of L_a is feasible. $\qquad\square$

Example 8.3 Find an initial feasible solution for the following linear program:

$$
\begin{aligned}
\text{maximize} \quad & x_1 + 3x_2 \\
\text{subject to} \quad & -x_1 - x_2 \leq -3 \\
& -x_1 + x_2 \leq -1 \\
& x_1 + 2x_2 \leq 2 \\
& x_1, x_2 \quad \geq 0
\end{aligned}
$$

We first observe that the given linear program is in canonical form, and that there are two negative values -3 and -1 on the right hand side of its constraints. Therefore, to compute a feasible solution for it we need to solve the auxiliary linear problem of Phase I of simplex algorithm. By definition, the auxiliary linear program is as follows:

$$
\begin{aligned}
\text{maximize} \quad & -x_0 \\
\text{subject to} \quad & -x_1 - x_2 - x_0 \leq -3 \\
& -x_1 + x_2 - x_0 \leq -1 \\
& x_1 + 2x_2 - x_0 \leq 2 \\
& x_0, x_1, x_2 \quad \geq 0
\end{aligned}
$$

To solve this linear program, we apply the simplex algorithm. So, initially we convert it to its slack form and derive the following dictionary:

$$\hat{\zeta} = -x_0$$
$$x_3 = -3 \;\; + x_1 + x_2 \;\; +x_0$$
$$x_4 = -1 \;\; + x_1 - x_2 \;\; +x_0$$
$$x_5 = \;\;\; 2 \;\; - x_1 - 2x_2 +x_0$$
$$x_0, \cdots, x_5 \; \geq 0$$

Since the non-basic variables have zero value, we can conclude that the solution of this dictionary is $x_0 = x_1 = x_2 = 0$, $x_3 = -3$, $x_4 = -1$ and $x_5 = 2$. However, this solution is not met by all constraints, since x_3 and x_4 are negative. Therefore, this dictionary is not feasible.

As we have already mentioned, we can find an initial feasible solution of the auxiliary linear program by setting $x_1 = \cdots = x_n = 0$ and by picking x_0 arbitrarily large. To this end, we search the "most infeasible" variable (i.e., the variable with the lowest value in the current solution); in our example this variable is x_3. Then, we choose x_0 and x_3 to be our entering and leaving variables, respectively, we rearrange the first equality to express x_0 in terms of x_1, x_2 and x_3, and substitute it in the other constraints. By this we obtain the following dictionary:

$$\hat{\zeta} = -3 \;\;\; +x_1 + x_2 \;\; -x_3$$
$$x_0 = 3 \;\;\;\; -x_1 - x_2 \;\; +x_3$$
$$x_4 = 2 \;\;\;\;\;\;\;\;\;\;\; - 2x_2 +x_3$$
$$x_5 = 5 \;\;\;\;\;\;\;\;\;\;\; - 3x_2 +x_3$$
$$x_0, \cdots, x_5 \geq 0$$

With this trick, we obtain a dictionary for the auxiliary linear problem, which is feasible, since all variables have non-negative values (the non-basic ones x_1, x_2 and x_3 have of course zero value, while the basic ones are such that $x_0 = 3$, $x_4 = 2$ and $x_5 = 5$). Hence, from this point on, we can find the optimal solution by applying the simplex algorithm as in Section 8.4. Variable x_1 has a positive coefficient in $\hat{\zeta}$. Therefore, if we increase the value of x_1, $\hat{\zeta}$ increases. But we have to guarantee that none of the other variables becomes negative by increasing the value of x_1. In particular, $x_1 \leq 3$, as otherwise x_0 becomes negative. Hence, x_1 is the entering variable and x_0 is the leaving variable. Next, we express x_1 as a linear combination of x_0, x_2 and x_3. In this particular case, we do not have to substitute x_1 in the constraints, since x_1 only appears in the first constraint. As a result we get following dictionary:

$$\hat{\zeta} = -x_0$$
$$x_1 = 3 \;\;\;\; -x_0 - x_2 \;\; +x_3$$
$$x_4 = 2 \;\;\;\;\;\;\;\;\;\;\; - 2x_2 +x_3$$
$$x_5 = 5 \;\;\;\;\;\;\;\;\;\;\; - 3x_2 +x_3$$
$$x_0, \cdots, x_5 \geq 0$$

Now we can see that the only variable appearing in the objective function $\hat{\zeta}$ is varialbe x_0 which has a negative coefficient. Hence, this dictionary is optimal with $x_0 = 0$. In particular, the optimal solution is $x_0 = x_2 = x_3 = 0$, $x_1 = 3$, $x_4 = 2$ and $x_5 = 5$. By Theorem 8.2, this solution is feasible for the original problem; in fact, $x_1 = 3$, $x_2 = 0$ meets all the constraints.

To describe the first dictionary of Phase II of simplex algorithm, where we seek to solve the original linear program, we make use of the fact that $x_0 = 0$. In other words, we can neglect variable x_0 completely from our last dictionary and by this derive a dictionary for the original problem. The only issue left is to define appropriately the objective function. Recall that the original objective function was $\zeta = x_1 + 3x_2$. Since x_1 is a basic variable, we need to substitute x_1 by $3 - x_2 + x_3$, which results in $\zeta = 3 - x_2 + x_3$. Hence, we obtain the following dictionary as the first dictionary of Phase II of simplex algorithm.

$$
\begin{aligned}
\zeta &= 3 & -x_2 + x_3 \\
x_1 &= 3 & -x_2 + x_3 \\
x_4 &= 2 & -2x_2 + x_3 \\
x_5 &= 5 & -3x_2 + x_3 \\
x_1, \cdots, x_5 &\geq 0
\end{aligned}
$$

8.6   Bland's Rule

In this section, we show that the simplex method terminates after a finite number of pivots. We start by making a simple observation regarding the maximum amount of distinct dictionaries, which follows from the fact that each dictionary is specified by its basic variables, which are in total m, out of $n + m$ variables that exist in total

> **Observation 8.1**   For a linear program with n variables and m constraints there are $\binom{n+m}{m}$ distinct dictionaries.

So, now we have an upper bound on the number of distinct dictionaries that the simplex algorithm can visit. It follows that if the algorithm does not terminate, then there must exist a cycle of dictionaries that the algorithm visits. We call this behavior degeneracy.

> **Observation 8.2**   (Degeneracy) If the simplex algorithm does not terminate, then some dictionaries have been visited more than once.

In the following, we formalize this observation.

> **Definition 8.3**   (Degenerate dictionary)   A dictionary
>
> $$
> \begin{aligned}
> z &= v + \sum_{i \in \mathcal{N}} \bar{c}_i x_i \\
> x_i &= \bar{b}_i - \sum_{j \in \mathcal{N}} \bar{a}_{ij} x_j \quad \forall i \in \mathcal{B} \\
> x_i &\geq 0 \qquad\qquad\qquad \forall i \in \mathcal{B} \cup \mathcal{B}
> \end{aligned}
> $$
>
> is called degenerate if $\bar{b}_i$ vanishes for some $i \in \mathcal{B}$, that is, $\bar{b}_i = 0$.

Example 8.4 The following is an example of a degenerate dictionary:

$$\zeta = 3 \quad -\frac{1}{2}x_1 + 2x_2 - \frac{3}{2}x_4$$
$$x_3 = 1 \quad -\frac{1}{2}x_1 \qquad -\frac{1}{2}x_4$$
$$x_5 = \ 0 + \ x_1 - \ x_2 + \ x_4$$

Definition 8.4 (Degenerate pivot) A pivot is degenerate, if the objective function value does not change.

Example 8.5 Consider the degenerate dictionary of Example 8.4. Observe that x_2 is the only variable with a positive coefficient in the objective function. Hence, x_2 is the entering variable in the simplex method. The leaving variable is x_5 and the new dictionary is:

$$\zeta = 3 + \frac{3}{2}x_1 - 2x_5 + \frac{1}{2}x_4$$
$$x_3 = 1 - \frac{1}{2}x_1 \qquad -\frac{1}{2}x_4$$
$$x_2 = 0 + \ x_1 - \ x_5 + \ x_4$$

Note that the value of the objective function did not change.

Definition 8.5 (Bland's rule.) The choice of the entering and leaving variables is according to the smallest subscript, i.e., if there is a choice in which entering or leaving variable to choose, then always choose the one with the smallest subscript consistently.

Theorem 8.3 (Bland 1977) The simplex method always terminates if the choice of the variables to enter and to leave the basis obey Bland's rule.

Proof. Assume for a contradiction that the simple method does not terminate. By Observation 8.2, it follows that some dictionaries are visited more than once. In particular, there exists a sequence of dictionaries $D_0, D_1, \ldots, D_{k-1}, D_k$ such that:

$$D_k = D_0.$$

Note that each of the dictionaries $D_0, D_1, \ldots, D_{k_1}$ defines the same solution; thus $D_0, D_1, \ldots, D_{k_1}$ are degenerate. Also, each variable that leaves the basis in one of $D_0, D_1, \ldots, D_{k_1}$ has to enter it again in one of these dictionaries.

> Definition 8.6 (Frickle) A variable is called fickle if it enters and leaves the basis in one of dictionaries $D_0, D_1, \ldots, D_{k_1}$.

In subsequent steps, we use the following notation:

- We denote by $\mathcal{F}$ the set of all fickle variables.

- Let ℓ be the largest index in $\mathcal{F}$.

- We denoted by D the dictionary in $D_0, D_1, \ldots, D_{k-1}$, in which x_ℓ leaves the basis.

- Let x_e be the corresponding entering variable in D.

- We denote by D^* the dictionary in which x_ℓ enters the basis.

> Observation 8.3 Since $x_e \in \mathcal{F}$, it follows that $e < \ell$.

Assume that dictionary D is written as follows:

$$\zeta = v + \sum_{j \in \mathcal{N}} c_j x_j \tag{8.1}$$

$$x_i = b_i - \sum_{j \in \mathcal{N}} a_{ij} x_j \ \forall i \in \mathcal{B} \tag{8.2}$$

Since the value of the objective function is the same in D and D^*, we can assume that dictionary D^* is written as follows (where $c_j^* = 0$ for all $j \in \mathcal{B}^*$):

$$\zeta = v + \sum_{i=1}^{n+m} c_i^* x_i \tag{8.3}$$

$$x_i = b_i^* - \sum_{j \in \mathcal{N}^*} a_{ij}^* x_j \ \forall i \in \mathcal{B}^* \tag{8.4}$$

Since x_ℓ enters the basis in D^* since and ℓ is the largest index in $\mathcal{F}$, the following must hold:

$$c_j^* \leq 0, \ \forall j \in \{1, \ldots, n+m\} \text{ such that } x_j \in \mathcal{F} \setminus \{x_\ell\}, \text{ and } c_\ell^* > 0. \tag{8.5}$$

Now consider the solution obtained by letting x_e increase (e.g., $x_e = t$) and all other variables in $\mathcal{N}$ remain zero (i.e., $x_j = 0$ for all $j \in \mathcal{N} \setminus \{e\}$). Then, by Section 8.6), we obtain:

$$\zeta = v + c_e t, \text{ and } x_i = b_i - a_{ie} t, \ \forall i \in \mathcal{B} \tag{8.6}$$

By substituting Eq. (8.6) into Section 8.6, we obtain:

$$\begin{aligned} \zeta &= v + c_e^* t + \sum_{i \in \mathcal{B}} c_i^* x_i \\ &= v + c_e^* t + \sum_{i \in \mathcal{B}} c_i^* (b_i - a_{ie} t) \end{aligned}$$

Since the value of the objective function is the same in D and D^*, we can conclude that, for every value of t, the following holds:

$$v + c_e^* t + \sum_{i \in \mathcal{B}} c_i^*(b_i - a_{ie}t) = v + c_e t \Leftrightarrow$$
$$(c_e - c_e^* + \sum_{i \in \mathcal{B}} c_i^* a_{ie})t = \sum_{i \in \mathcal{B}} c_i^* b_i$$

The equation above has to hold for all t. Hence:

$$c_e - c_e^* + \sum_{i \in \mathcal{B}} c_i^* a_{ie} = 0 \tag{8.7}$$

We proceed by making the following observations:

- Since x_e enters in D, it holds $c_e > 0$.

- Since x_ℓ enters in D^* and since $e < \ell$, by Bland's rule it follows that $c_e^* \leq 0$ holds.

Combining the two observations above, we obtain $c_e - c_e^* > 0$. Therefore, Section 8.6 implies:

$$\sum_{i \in \mathcal{B}} c_i^* a_{ie} < 0.$$

Since the sum above is strictly negative, it follow that there exist an element s in $\mathcal{B}$ such that $c_s^* a_{se} < 0$. Hence, x_s is basic in D. Since $c_s^* \neq 0$, variable x_s is non-basic in D^*. This implies that x_s is a fickle, that is, $x_s \in \mathcal{F}$.

Note that, since ℓ is the largest index in $\mathcal{F}$, it holds that $s \leq \ell$. We now claim that $s < \ell$ holds. To see this, observe first that x_ℓ is the leaving variable in D and x_e the entering variable. Hence, $a_{\ell e} > 0$. Also, x_ℓ is the entering variable in D^*, which implies that $c_\ell^* > 0$. It follows that $c_\ell^* a_{\ell e} > 0$, which implies that $s \neq \ell$, as we showed earlier that $c_s^* a_{se} < 0$.

Since $s < \ell$, by Bland's rule x_s cannot be a candidate to enter the basis in $\mathcal{D}^*$. Hence, it holds that $c_s^* < 0$. Since $c_s^* a_{se} < 0$, it follows that $a_{se} > 0$.

Recall that each of the dictionaries $D_0, D_1, \ldots, D_{k_1}$ defines the same solution. This implies that all fickle variables are zero. The next observation is since $x_s \in \mathcal{F}$ and $s \in \mathcal{B}$, the value of x_s is zero, i.e., $b_s = 0$. Since $b_s = 0$ and $a_{se} > 0$, variable x_s is a candidate for leaving $\mathcal{B}$ in D. But x_ℓ leaves with $\ell > s$; this is a contradiction. $\qquad \square$

Bland's theorem implies that simple algorithm terminates after at most $\binom{n+m}{m}$ iterations. Regarding the time complexity of the simplex algorithm, two notable results are by Dantzig and by Klee and Minty, which we briefly discuss in the following.

> **Property 8.1** (Dantzig '80) The simplex algorithm has in average polynomial-time complexity.

In other words, if one considers the space of all possible linear programs with n variables and m constraints, then on average they can be solved in polynomial time. In contrast to this
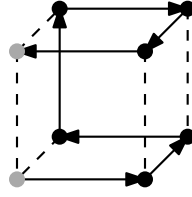
Figure 8.3: In the linear program by Klee and Minty, the feasible solution space is a 3-dimensional cube with $n = 3$ (which implies that it consists of $2^n = 8$ corners). The simplex algorithm needs $2^n - 1 = 7$ iterations to solve the program; it starts at the gray lower-left corner and finds the optimal solution at the gray upper-left corner having visited first all intermediate corners of the solution space.

positive result, Klee and Minty found a linear program which requires an exponential amount of iterations in order to be solved by the simple algorithm.

> **Property 8.2** (Klee and Minty '72) There is a linear program with n variables and n constraints, where simplex needs $2^n - 1$ iterations to solve it.

Without entering too many details, the feasible solution space of the program by Klee and Minty is a n-dimensional hypercube, and simplex algorithm has to iterate every single corner of it in order to compute the optimal solution; see Fig. 8.3 for an illustration of the case $n = 3$. So, while the time-complexity of the simplex algorithm is on average polynomial, there exist instances which require an exponential number of iterations.

8.7   Duality

In this section, we formally introduce the dual program of a linear program, which, as already mentioned, can be used to verify results from the simplex algorithm (among others). We initially introduce the principle ideas by an example.

$$
\begin{aligned}
\text{minimize} \quad & 3x_1 + 5x_2 \\
\text{subject to} \quad & 2x_1 + x_2 \geq 3 \\
& 2x_1 + 2x_2 \geq 5 \\
& x_1 + 4x_2 \geq 4 \\
& x_1, x_2 \geq 0
\end{aligned}
$$

The optimal solution of this linear program is $(x_1, x_2) = (2, 1/2)$ with value 8.5. The question that we will try to answer is why this solution is optimal. We can observe that for any feasible solution all five constraints of the program must hold. If we add up constraints one and three, we obtain $3x_1 + 5x_2 \geq 7$. The left hand side of the equation is exactly the objective function, which implies that for this linear program 7 is a lower bound for the optimal solution.

We can derive a better bound if we multiply the second constraint by $3/2$ resulting in $3x_1 +$

$3x_2 \geq 7.5$. After that we substitute $3x_2$ by $5x_2$ which we are allowed to do because $x_2 \geq 0$. Hence, we obtain $3x_1 + 5x_2 \geq 7.5$, which sets a new lower bound of 7.5 for the optimal solution.

In the following, we will approach this search for lower bounds more systematically. We seek to find a linear combination of all the constraints to get as close to the coefficients of the objective function as possible. It is obviously fine to have a smaller coefficient than the one in the objective function, since every variable is non-negative and we can just add the needed rest up without violating the inequality (as we just did when we upper bounded $3x_5$ by $5x_5$).

So we multiply each of the three constraints by one new variable y_1, y_2 and y_3 respectively and sum all the constraints up (except the non-negativity ones). If we then extract the variables x_1, x_2 we get

$$(2y_1 + 2y_2 + y_3)x_1 + (y_1 + 2y_2 + 4y_3)x_2 \geq 3y_1 + 5y_2 + 4y_3$$

In the objective function we see that the coefficients of x_1 and x_2 are upper bounded by 3 and 5. So, we obtain two new constraints $2y_1 + 2y_2 + y_3 \leq 3$ and $y_1 + 2y_2 + 4y_3 \leq 5$, respectively. We can also conclude that the right hand side of the inequality should be as high as possible, so we can find the best lower bound of the optimal solution and therefore we seek to maximize $3y_1 + 5y_2 + 4y_3$. Now, we also require every new variable y_i to be non-negative. Thus, putting everything together, we obtain a new linear program, the so-called dual of the original one:

$$
\begin{aligned}
\text{maximize} \quad & 3y_1 + 5y_2 + 4y_3 \\
\text{subject to} \quad & 2y_1 + 2y_2 + y_3 \leq 3 \\
& y_1 + 2y_2 + 4y_3 \leq 5 \\
& y_1, y_2, y_3 \geq 0
\end{aligned}
$$

The optimal solution of this linear program is $y_1 = 0$, $y_2 = 7/6$ and $y_3 = 2/3$. Naturally, now we wonder what is the lower bound for the value of the solution of the original problem that one obtains by using this solution. To this end, we observe:

$$
\begin{aligned}
7/6*(2): \quad & 14/6x_1 + 14/6x_2 \geq 35/6 \\
2/3*(3): \quad & +\ 4/6x_1 + 16/6x_2 \geq 16/6 \\
\hline
& 3x_1 + 5x_2 \geq 8.5
\end{aligned}
$$

It follows that the original objective function is now lower bounded by 8.5 which justifies the optimal solution for the original program, since it can obviously not be lower than that.

Formalization. After introducing duality intuitively by the example, we now seek to formalise the concept accordingly. We first define the so-called primal linear program (which corresponds to the original program) in maximum standard form and then build the dual linear program in minimum standard form as follows. Note that in the previous example the primal was in minimum standard form and the dual was in maximum standard form. But this actually is not important, since the dual of the dual is the primal (this is left as an exercise to the reader).

Primal linear program:

$$
\begin{aligned}
\text{maximize} \quad & c_1 x_1 + c_2 x_2 + \cdots + c_n x_n \\
\text{subject to} \quad & a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n \leq b_1 \\
& \cdots \\
& a_{m1} x_1 + a_{m2} x_2 + \cdots + a_{mn} x_n \leq b_m \\
& x_1, \cdots, x_n \geq 0
\end{aligned}
$$

Note that we can easily write the primal in matrix form as follows.

$$
\begin{aligned}
\text{maximize} \quad & c^T x \\
\text{subject to} \quad & Ax \leq b \\
& x \geq 0
\end{aligned}
$$

Dual linear program:

$$
\begin{aligned}
\text{minimize} \quad & b_1 y_1 + b_2 y_2 + \cdots + b_m y_m \\
\text{subject to} \quad & a_{11} y_1 + a_{12} y_2 + \cdots + a_{1n} y_m \geq c_1 \\
& \cdots \\
& a_{1n} y_1 + a_{m2} y_2 + \cdots + a_{mn} y_m \geq c_m \\
& y_1, \cdots, y_m \geq 0
\end{aligned}
$$

Also for the dual, it is easy to obtain the matrix form as follows.

$$
\begin{aligned}
\text{minimize} \quad & y^T b \\
\text{subject to} \quad & y^T A \geq c^T \\
& y \geq 0
\end{aligned}
$$

> **Observation 8.4** The dual of the dual of a linear program is the primal linear program (see Exercise 8.6).

In the following, we prove that finding a feasible solution of the primal and finding a feasible solution of the dual that are equal verifies the optimality of both solutions.

> **Theorem 8.4** (Weak duality theorem) If $\bar{x}^T = (\bar{x}_1, \cdots, \bar{x}_n)$ is a feasible solution for the primal linear program and $\bar{y}^T = (\bar{y}_1, \cdots, \bar{y}_m)$ is a feasible solution for the dual linear program, then it holds that
> $$ c^T \bar{x} \leq \bar{y}^T b $$

Proof. Since $\bar{x}$ is a feasible solution for the primal linear program, we obtain:

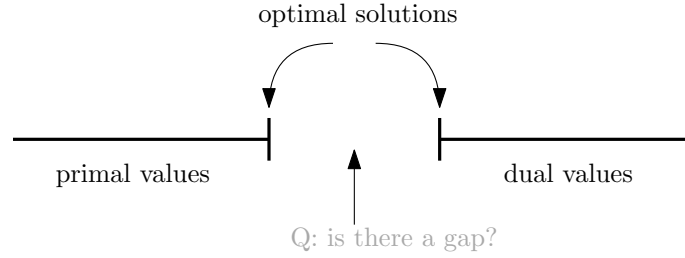$$ A\bar{x} \leq b; \ \bar{x} \geq 0 $$

Figure 8.4: Strong duality theorem implies that there is no gap.

Similarly, since $\bar{y}$ is a feasible solution for the dual linear program, we obtain:

$$\bar{y}^T A \geq c^T; \ \bar{y} \geq 0$$

Now by multiplying $A\bar{x} \leq b$ by $\bar{y}^T$ on the left we get $\bar{y}^T A\bar{x} \leq \bar{y}^T b$. Since $\bar{y} \geq 0$ holds, the inequality sign does not change. Now we do the same for the second inequality and multiply $\bar{y}^T A \geq c^T$ by $\bar{x}$ on the right side and get $\bar{y}^T A\bar{x} \leq c^T \bar{x}$, while again $\bar{x} \geq 0$ implies that the inequality sign stays the same. Now, we can just write the whole inequality as $c^T\bar{x} \leq \bar{y}^T A\bar{x} \leq \bar{y}^T b$ and by ignoring the middle part we get $c^T\bar{x} \leq \bar{y}^T b$. $\qquad\square$

Theorem 8.4 directly implies that the value of any feasible solution (and thus also of the optimal one) of the primal linear program is upper bounded by the value of any feasible solution (and thus also by the optimal one) of dual linear program. Therefore, we can quickly conclude the following corollary of Theorem 8.4.

> **Corollary 8.1** If $\bar{x}$ is a feasible primal solution and $\bar{y}$ is a feasible dual solution such that $c^T\bar{x} = \bar{y}^T b$, then $\bar{x}$ and $\bar{y}$ are optimal.

Proof. Assume to the contrary that $\bar{x}$ is not optimal. It follows that there exists another solution $\bar{x}'$ such that $c^T\bar{x} < c^T\bar{x}'$. Since we already know that $c^T\bar{x} = \bar{y}^T b$ it follows that $c^T\bar{x} > \bar{y}^T b$ holds. Therefore, the weak duality theorem is violated. The same argument applies if $\bar{y}$ is not optimal; so both are optimal. $\qquad\square$

In view of Theorem 8.4, there is still a question to be answered, namely, whether there is a gap between the optimal primal value and the optimal dual value such that the two values are really different (as illustrated in Fig. 8.4). This is what the strong duality theorem is about, which we prove in the following.

> **Theorem 8.5** (Strong duality theorem) If $\bar{x}^T = (\bar{x}_1, \cdots, \bar{x}_n)$ is optimal for the primal and $\bar{y}^T = (\bar{y}_1, \cdots, \bar{y}_m)$ is optimal for the dual, then:
>
> $$c^T\bar{x} = \bar{y}^T b$$

Proof. The proof strategy is as follows. Given the optimal solution $(\bar{x}_1, \cdots, \bar{x}_n)$ of the primal linear program, we describe a feasible solution $(\bar{y}_1, \cdots, \bar{y}_m)$ of the dual linear program, and we prove that these two solutions have the same value in the primal and in the dual linear programs, respectively. By Corollary 8.1, we can then conclude that $(\bar{y}_1, \cdots, \bar{y}_m)$ is an optimal feasible solution of the dual linear program. Let the primal and the dual linear programs be as follows:

$$
\begin{array}{ll}
\text{maximize} \quad \sum_{j=1}^{n} c_j x_j & \qquad\qquad \text{minimize} \quad \sum_{i=1}^{m} b_i y_i \\[2em]
\text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_i \le b_i \quad i = 1, \ldots, m & \qquad\qquad \text{subject to} \quad \sum_{i=1}^{m} a_{ij} y_i \ge c_j \quad j = 1, \ldots, n \\[2em]
\qquad\qquad x_1, \cdots, x_n \ge 0 & \qquad\qquad\qquad\qquad y_1, \cdots, y_m \ge 0
\end{array}
$$

We next introduce m slack variables for the primal linear program as follows:

$$
x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j, \text{ for } i = 1, \ldots, m \tag{8.8}
$$

By doing so, using the simplex algorithm, we obtain that the value of optimal solution is:

$$
\zeta = \hat{\zeta} + \sum_{k=1}^{n+m} \bar{c}_k x_k \tag{8.9}
$$

where $\bar{c}_k = 0$ if x_k is basic, and $\bar{c}_k \le 0$ if x_k is non-basic.

We next claim that $(\bar{y}_1, \ldots, \bar{y}_m) = (\bar{c}_{n+1}, \ldots, \bar{c}_{n+m}) \ge 0$ is a feasible solution for the dual linear program, that is, $\bar{y}_i = -\bar{c}_{n+i}$ for $i = 1, \ldots, m$. We argue as follows:

$$
\begin{aligned}
\sum_{j=1}^{n} c_j \bar{x}_j \;=\; \zeta \;&\overset{(8.9)}{=}\; \hat{\zeta} + \sum_{k=1}^{n+m} \bar{c}_k x_k \\[1em]
&= \hat{\zeta} + \sum_{j=1}^{n} \bar{c}_j x_j + \sum_{i=1}^{m} \bar{c}_{n+i} x_{n+i} \\[1em]
&\overset{(8.8)}{=} \hat{\zeta} + \sum_{j=1}^{n} \bar{c}_j x_j - \sum_{i=1}^{m} \bar{y}_i \Big(b_i - \sum_{j=1}^{n} a_{ij} x_j \Big) \\[1em]
&= \hat{\zeta} - \sum_{i=1}^{m} \bar{y}_i b_i + \sum_{j=1}^{n} \Big(\bar{c}_j - \sum_{i=1}^{m} a_{ij} \bar{y}_i \Big) x_j
\end{aligned}
$$

From the equality above, we obtain:

$$
\hat{\zeta} = \sum_{i=1}^{m} \bar{y}_i b_i \tag{8.10}
$$

Furthermore:

$$
c_j = \bar{c}_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i, \quad \text{ for } j = 1, \ldots, n
$$

Since $\bar{c}_j \leq 0$ for $j = 1, \ldots, n + m$, we can conclude:

$$\sum_{i=1}^{m} a_{ij} \bar{y}_i \geq c_j, \quad \text{for } j = 1, \ldots, n \tag{8.11}$$

Since $(\bar{y}_1, \ldots, \bar{y}_m) \geq 0$, Eq. (8.11) implies that $(\bar{y}_1, \ldots, \bar{y}_m)$ is a feasible solution of the dual linear program. By Eq. (8.10) and by Theorem 8.4, it follows that $(\bar{y}_1, \ldots, \bar{y}_m)$ is optimal solution for the dual linear program. $\qquad \square$

Corollary 8.2 If (P) is a linear program and (D) its dual, then the following hold.

i. (P) has an optimal solution if and only if (D) has an optimal solution.

ii. If (P) is unbounded, then (D) is infeasible.

iii. If (D) is unbounded, then (P) is infeasible.

Note that (P) and (D) can be both infeasible (see Exercise 8.5).

Strong duality theorem has an important algorithmic relevance. Assume that $x^T = (x_1, \cdots, x_n)$ and $y^T = (y_1, \cdots, y_m)$ are two potential solutions for the primal and the dual linear programs, respectively. Then, we prove optimality for both by assessing the following: (i) $Ax \leq b$, $x \geq 0$, (ii) $y^T A \geq c^T$, $y \geq 0$, and (iii) $c^T x = y^T b$.

In the following, we introduce a method to determine whether a given solution of a linear program is optimal. To this end, we first introduce the so-called complementary slackness theorem.

Theorem 8.6 (Complementary slackness) Let $\bar{x}^T = (\bar{x}_1, \cdots, \bar{x}_n)$ be a feasible solution for the primal and $\bar{y}^T = (\bar{y}_1, \cdots, \bar{y}_m)$ be a feasible solution for the dual, respectively. Then, $\bar{x}$ and $\bar{y}$ are optimal if and only if all of the following hold:

(i) $\bar{x}_j = 0$ or $\sum_{i=1}^{m} a_{ij} \bar{y}_i = c_j$ for all $j = 1, \ldots, n$, and

(ii) $\bar{y}_i = 0$ or $\sum_{j=1}^{n} a_{ij} \bar{x}_j = b_i$ for all $i = 1, \ldots, m$.

Proof. By the weak duality theorem, we know that:

$$c^T \bar{x} \leq (\bar{y}^T A) \bar{x} \leq \bar{y}^T b \tag{8.12}$$

By the strong duality theorem, it follows that $\bar{x}$ and $\bar{y}$ are optimal if and only if:

$$c^T \bar{x} = \bar{y}^T b$$

or equivalently, by Eq. (8.12):

$$c^T \bar{x} = (\bar{y}^T A)\bar{x} \quad \text{and} \quad \bar{y}^T b = \bar{y}^T (A\bar{x})$$

The above equalities can be equivalently rewritten as follows:

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{j=1}^{n} (\sum_{i=1}^{m} \bar{y}_i a_{ij})\bar{x}_j \quad \text{and} \quad \sum_{i=1}^{m} \bar{y}_i b_i = \sum_{i=1}^{m} \bar{y}_i (\sum_{j=1}^{n} a_{ij}\bar{x}_j)$$

The latter are equivalent to (i) and (ii) of the theorem, respectively.   □

Theorem 8.7 A feasible solution $\bar{x}^T = (\bar{x}_1, \cdots, \bar{x}_n)$ for the primal linear program is optimal if and only if there exists a feasible solution $\bar{y}^T = (\bar{y}_1, \cdots, \bar{y}_m)$ of the dual linear program such that:

(CS.1) If $\bar{x}_j > 0$, then $\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j$, for all $j = 1, \ldots, n$

(CS.2) If $\sum_{j=1}^{n} a_{ij}\bar{x}_j < b_i$, then $\bar{y}_i = 0$, for all $i = 1, \ldots, m$.

Proof. Assume first that there exists a feasible solution $\bar{y}^T = (\bar{y}_1, \cdots, \bar{y}_m)$ of the dual linear program such that (CS.1) and (CS.2) hold. We prove that $\bar{x}$ and $\bar{y}$ are optimal for the primal and for the dual linear programs, respectively. Since $\bar{x}$ is feasible, for all $j = 1, \ldots, m$, either $\bar{x}_j = 0$ or $\bar{x}_j > 0$ holds. By (CS.1), the latter implies $\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j$. Hence, Condition (i) of complementary slackness theorem holds. Similarly, since $\bar{y}$ is feasible, for $i = 1, \ldots, m$, either $\bar{y}_i = 0$ or $\bar{y}_i > 0$ holds. By (CS.2), the latter implies $\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j$. Hence, Condition (ii) of complementary slackness theorem also holds. Therefore, $\bar{x}$ and $\bar{y}$ are optimal, as desired.

Assume now that $\bar{x}$ is an optimal solution for the primal linear program. We will prove that the optimal solution $\bar{y}$ of the dual linear program, which exists by Corollary 8.2, satisfies (CS.1) and (CS.2). Indeed, if $\bar{x}_j > 0$, then, by Condition (i) of complementary slackness theorem, it follows that $\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j$ holds, for $j = 1, \ldots, n$. Hence, (CS.1) holds. On the other hand, if $\sum_{j=1}^{n} a_{ij}\bar{x}_j < b_i$, then, by Condition (ii) of complementary slackness theorem, it follows that $\bar{y}_i > 0$, for $i = 1, \ldots, m$. Hence, (CS.2) also holds, completing the proof.   □

In the following example, we demonstrate how Theorem 8.7 can be employed to determine whether a given solution is optimal for a linear program.

Example 8.6 Determine whether $(\bar{x}_1, \bar{x}_2, \bar{x}_3) = (0, \frac{7}{6}, \frac{2}{3})$ is the optimal solution for the following linear program:

$$
\begin{aligned}
\text{maximize} \quad & 3x_1 + 5x_2 + 4x_3 \\
\text{subject to} \quad & 2x_1 + 2x_2 + x_3 \leq 3 \\
& x_1 + 2x_2 + 4x_3 \leq 5 \\
& x_1, x_2, x_3 \geq 0
\end{aligned}
$$

Answer. It is not difficult to see that $(0, \frac{7}{6}, \frac{2}{3})$ is a feasible solution of the given linear program. Condition (CS.1) of Theorem 8.7 applied to $\bar{x}_2 > 0$ and to $\bar{x}_3 > 0$, implies $2\bar{y}_1 + 2\bar{y}_2 = 5$, and $\bar{y}_1 + 4\bar{y}_2 = 4$, respectively. These two equalities imply $(\bar{y}_1, \bar{y}_2) = (2, \frac{1}{2})$, which is not difficult to see that it is feasible for the dual of the given linear program:

$$
\begin{aligned}
\text{minimize} \quad & 3y_1 + 5y_2 \\
\text{subject to} \quad & 2y_1 + y_2 \geq 3 \\
& 2y_1 + 2y_2 \geq 5 \\
& y_1 + 4y_2 \geq 4 \\
& y_1, y_2 \geq 0
\end{aligned}
$$

Therefore, we can conclude that $(0, \frac{7}{6}, \frac{2}{3})$ is optimal for the primal linear program. $\qquad \square$

8.8 The primal-dual simplex method

Consider a primal linear program P and for the sake of simplicity assume that (i) the objective function of P is a minimization one, while (ii) its constraint system consists exclusively of equalities. Hence, P can be written as follows:

$$
\begin{aligned}
\text{minimize} \quad & c^T x \\
\text{subject to} \quad & Ax = b \\
& x \geq 0
\end{aligned}
$$

Observe that both assumptions (i) and (ii) are without loss of generality, as any program in normal form can be easy transformed into an equivalent one, such that both (i) and (ii) are satisfied. Let D be the dual program of P, which is written as follows:

$$
\begin{aligned}
\text{maximize} \quad & y^T b \\
\text{subject to} \quad & A^T y \leq c \\
& y \geq 0
\end{aligned}
$$

Let $\bar{x} = (\bar{x}_1, \ldots, \bar{x}_n)$ and $\bar{y} = (\bar{y}_1, \ldots, \bar{y}_m)$ be optimal solutions for P and D, respectively. By massaging the complementary slackness theorem, we obtain:

$$
\begin{aligned}
0 &= c^T \bar{x} - \bar{y}^T b \\
&= c^T \bar{x} - \bar{y}^T A\bar{x} + \bar{y}^T A\bar{x} - \bar{y}^T b \\
&= (c^T - \bar{y}^T A)\bar{x} + \bar{y}^T(A\bar{x} - b) \\
&= (c^T - \bar{y}^T A)\bar{x} \\
&= (c - A^T \bar{y})^T \bar{x}
\end{aligned}
$$

Since $\bar{x} \geq 0$ holds by P and since $(c - A^T \bar{y})^T \geq 0$ holds by D, it follows:

$$
\sum_{i=1}^{m} a_{ij} \bar{y}_i < c_j \Rightarrow \bar{x}_j = 0, \quad \forall j = 1, \ldots, n \tag{8.13}
$$

To further formalize the property of Eq. (8.13), we define the following set of indices:

$$I = I(\bar{y}) = \{j : 1 \leq j \leq n, \sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j\} \quad \text{and} \quad \bar{I} = \bar{I}(\bar{y}) = \{j : 1 \leq j \leq n, \sum_{i=1}^{m} a_{ij}\bar{y}_i \neq c_j\} \quad (8.14)$$

Note that $\bar{I}$ is the complement of I. Having introduced sets I and $\bar{I}$, we can rewrite Eq. (8.13) as follows:

$$\bar{x}_j = 0, \quad \forall j \in \bar{I} \tag{8.15}$$

The algorithmic idea in the dual-simplex method is based on the following observation.

> **Observation 8.5** If we have computed two feasible solutions $x = (x_1,\ldots,x_n)$ and $y = (y_1,\ldots,y_m)$ for P and D, respectively, with $x_j = 0$ for each j in $\bar{I}(y)$, then x and y are optimal for P and D, respectively

Hence, the goal is to find a solution $\bar{y}$ of D, such that Eq. (8.15) holds. In the dual-simplex method, this is achieved by iteratively "improving" an already computed feasible solution of D (hence, the method focuses only in computing feasible solutions of the dual program). At each step in this iterative procedure, given a solution $y \in \mathbb{R}^m$ of D, we seek to find an $x \in \mathbb{R}^n$ that fulfills Eq. (8.15) and is 'as feasible as possible' in P. Of course, if x is a feasible solution for P, then, by the observation above, the optimal solutions for both P and D have been found.

Let $y \in \mathbb{R}^m$ be a feasible solution of D (initially, we assume $y = 0$). To cope with the task of finding x, we first consider the following linear program, which we denote by R:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{m} s_i \\
\text{subject to} \quad & \sum_{j \in I} a_{ij}x_j + s_i = b_i, \quad \forall i = 1,\ldots,m \\
& x_j \geq 0, \quad \forall j \in I \\
& s_i \geq 0, \quad \forall i = 1,\ldots,m
\end{aligned}$$

Note that, in the constraint system of R, variables in $\bar{I}$ are absent, since they are of zero value. Note also that R consists of m slack variables $s_1,\ldots,s_m$, such that s_i expresses the difference between b_i and $\sum_{j \in I} a_{ij}x_j$, for $i = 1,\ldots,m$. Thus, in a feasible solution of P all these variables must be zero (while in general they are non-negative).

> **Claim 8.1** Linear program R always admits a (non-negative) optimal solution.

Proof. Since R is restricted (that is, its optimal solution is lower-bounded by zero) and since R is a minimization program, to prove the claim it suffices to prove that R has a feasible

solution. Such a solution is the following:

$$x_j = 0, \forall j \in I \quad \text{and} \quad (s_1, \ldots, s_m) = (b_1, \ldots, b_m). \qquad \square$$

Let $\bar{z} \geq 0$ be the optimal solution of R, which exists by Claim 8.1. This solution consists of two components, which we denote by $s \in \mathbb{R}^m$ and x_I. The former corresponds to the values of the slack variables $s_1, \ldots, s_m$ of R, while the second to the values of the variables x_j with $j \in I$. For completeness, we denote by $x_{\bar{I}}$ the zero-values of the variables x_j with $j \in \bar{I}$.

We distinguish two cases. Consider first the case, in which the value of the optimal solution of R is zero, that is, $\bar{z} = 0$. Since $s_i \geq 0$ for each $i = 1, \ldots, m$, it follows that the value of each slack variable of R is zero in the optimal solution of R, that is, $s = 0$. This implies that the solution $(x_1, \ldots, x_n)$ obtained by the union of x_I and $x_{\bar{I}}$ is a feasible solution for P, and thus optimal by Observation 8.5.

Hence, we focus on the case, in which the value of the optimal solution of R is non-zero, that is $\bar{z} > 0$. In this case, we consider the dual of R, which we denote by DR and write as follows:

$$\text{maximize} \quad \sum_{i=1}^{m} b_i \pi_i$$

$$\text{subject to} \quad \sum_{i=1}^{m} a_{ij} \pi_i \leq 0, \quad \forall j \in I$$

$$\pi_i \leq 1, \quad \forall i = 1, \ldots, m$$

By the complementary slackness theorem, DR admits an optimal solution $\bar{\pi}$ such that $b^T \bar{\pi} = \bar{z}$. Our goal is to improve the solution y of D to $y + t\bar{\pi}$, where $t > 0$ is appropriately chosen. By doing so, the value of the solution is increased from $b^T y$ to:

$$b^T(y + t\bar{\pi}) = b^T y + t b^T \bar{\pi} = b^T y + t\bar{z}$$

However, by the constraints of D, we need to guarantee that:

$$A^T(y + t\bar{\pi}) \leq c$$

which can be equivalently written as follows:

$$\sum_{i=1}^{m} a_{ij}(y_i + t\bar{\pi}_i) \leq c_j, \quad \forall j = 1, \ldots, n \tag{8.16}$$

> **Claim 8.2** Eq. (8.16) holds for every $j \in I$.

Proof. Since $y = (y_1, \ldots, y_m)$ is a feasible solution of D, it follows that $\sum_{i=1}^{m} a_{ij} y_i \leq c_j$ holds (for every $j = 1, \ldots, n$, and thus) for every $j \in I$. Since $\bar{\pi} = (\bar{\pi}_1, \ldots, \bar{\pi}_m)$ is the optimal solution of DR, $\sum_{i=1}^{m} a_{ij} \bar{\pi}_i \leq 0$ holds, for every $j \in I$. Since $t > 0$, we obtain:

$$\sum_{i=1}^{m} a_{ij}(y_i + t\bar{\pi}_i) \leq c_j, \quad \forall j \in I \qquad \square$$

By Claim 8.2, it follows that Eq. (8.16) has to be guaranteed for $j \in \bar{I}$, in order to guarantee that $y + t\bar{\pi}$ is a feasible solution for D.

> Claim 8.3   If $\sum_{i=1}^{m} a_{ij} \bar{\pi}_i \leq 0$, for each $j \in \bar{I}$, then D is unrestricted.

Proof. Since $y = (y_1, \ldots, y_m)$ is a feasible solution of D, it follows that $\sum_{i=1}^{m} a_{ij} y_i \leq c_j$ holds (for every $j = 1, \ldots, n$, and thus) for every $j \in \bar{I}$. Since $\sum_{i=1}^{m} a_{ij} \bar{\pi}_i \leq 0$ holds for each $j \in \bar{I}$, by Claim 8.2, it follows we can increase t arbitrarily without deviating Eq. (8.16). Thus, D is unrestricted, as desired. $\square$

If $\sum_{i=1}^{m} a_{ij} \bar{\pi}_i \leq 0$, for each $j \in \bar{I}$, then the primal-dual simplex method will terminate by reporting that D is unrestricted, and thus, by Corollary 8.2, P has no feasible solution. Hence, in the following we assume without loss of generality that there exist at least one $j \in \bar{I}$ such that $\sum_{i=1}^{m} a_{ij} \bar{\pi}_i > 0$. In this case, we guarantee Eq. (8.16) by setting t as follows:

$$t = \min_{\substack{j \in \bar{I} \\ \sum_{i=1}^{m} a_{ij} \bar{\pi}_i > 0}} \frac{c_j - \sum_{i=1}^{m} a_{ij} y_i}{\sum_{i=1}^{m} a_{ij} \bar{\pi}_i} \tag{8.17}$$

This guarantees that $y + t\bar{\pi}$ is a new feasible solution for D, and the primal-dual simplex method iteratively continues, by setting y to $y + t\bar{\pi}$. Fig. 8.5 provides a schematization of the basic steps of the primal-dual simplex method.
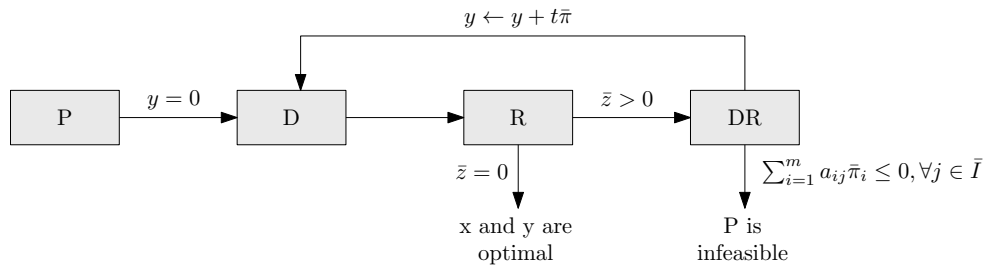

Figure 8.5: Schematization of the basic steps of the primal-dual simplex method.

8.8.1   An application to minimum-cost bipartite matching

In this section, we present an application of the primal-dual simple method. Let $G = (U \cup W, E, c)$ be an edge-weighted complete bipartite graph, such that $E = U \times W$ and $|U| = |W| = n$. We assume that the weights of the edges are costs (that is, $c : E \to \mathbb{R}^+$) and our goal is to compute a minimum-cost perfect matching of G. This problem can be solved efficiently using (an adjustment of) the Hungarian method in time $\mathcal{O}(n^3)$; see Section 6.4. In the following, we first formulate this problem as a linear program. Then, we apply the primal-dual simplex

method to solve it and in this way we identify several similarities with the Hungarian method.

$$\text{minimize} \quad \sum_{u \in U} \sum_{w \in W} c(u,w) x_{u,w}$$

$$\text{subject to} \quad \sum_{(u,v) \in E} x(u,v) = 1, \quad \forall u \in U$$

$$\sum_{(v,w) \in E} x(w,v) = 1, \quad \forall w \in W$$

$$x(u,w) \in \{0,1\}, \qquad \forall (u,w) \in E$$

Note that the linear program given above contains binary variables. As a result, the primal-dual simplex method cannot be directly applied. To overcome this issue, we consider a relaxation of this program but we will pay attention to keep the computed solutions integer. We denote the relaxation by P and we write it as follows:

$$\text{minimize} \quad \sum_{u \in U} \sum_{w \in W} c(u,w) x_{u,w}$$

$$\text{subject to} \quad \sum_{(u,v) \in E} x(u,v) = 1, \quad \forall u \in U$$

$$\sum_{(v,w) \in E} x(w,v) = 1, \quad \forall w \in W$$

$$x(u,w) \geq 0, \qquad \forall (u,w) \in E$$

We proceed with the application of the primal-dual simplex method by computing the dual program D of P, as follows:

$$\text{maximize} \quad \sum_{v \in U \cup W} y_v$$

$$\text{subject to} \quad y_u + y_w \leq c(u,w), \quad \forall (u,w) \in E$$

$$y_v \geq 0, \qquad \forall v \in U \cup W$$

Observe that any feasible solution of D corresponds to a feasible labeling of G in the Hungarian method; see Definition 6.12[1]. Let y be a feasible solution of D (e.g., initially $y = 0$) and define set $I = I(y)$ as in Eq. (8.14):

$$I = I(y) = \{(u,w) \in W : y_u + y_w = c(u,w)\}$$

An immediate observation is that set I contains the edges of the equality graph of the Hungarian method; see Definition 6.13. The restricted linear program R of the primal-dual simple method can be written as follows:

$$\text{minimize} \quad \sum_{v \in U \cup W} s_v$$

$$\text{subject to} \quad \sum_{(v,z) \in I} x_{v,z} + s_v = 1, \quad \forall v \in U \cup W$$

$$x_{v,z} \geq 0, \qquad \forall (v,z) \in I$$

$$s_v \geq 0, \qquad \forall v \in U \cup W$$

---

[1]Note that in Definition 6.12, the direction of the inequality is reversed, since the version of the problem that we solved was the maximization one. The method, however, can be easily adjusted for the corresponding minimization problem.

We now observe that the objective function of R can be equivalently written as follows:

$$\sum_{v \in U \cup W} s_v = \sum_{v \in U \cup W} \left(1 - \sum_{(v,z) \in I} x_{v,z}\right) = 2n - 2 \sum_{(v,z) \in I} x_{v,z} \tag{8.18}$$

Therefore, R can be equivalently written as follows:

$$\begin{aligned}
\text{maximize} \quad & \sum_{(v,z) \in I} x_{v,z} \\
\text{subject to} \quad & \sum_{(v,z) \in I} x_{v,z} \leq 1, \quad \forall v \in U \cup W \\
& x_{v,z} \geq 0, \qquad \forall (v,z) \in I
\end{aligned}$$

Having written R as above, we observe that solving R corresponds to finding a maximum-cardinality matching in the equality graph (assuming that the variable of R are restricted in $\{0, 1\}$). Therefore, instead of using simplex method to solve R, we can use the algorithm by Hopcroft and Karp (see Algorithm 6.2 in Section 6.2), which needs $\mathcal{O}(n^{5/2}m)$ time. In particular, if M is a solution of R, then by Eq. (8.18) we obtain:

$$\sum_{v \in U \cup W} s_v = 2n - 2|M| \tag{8.19}$$

Further, if M is perfect (that is, $|M| = n$ holds in the equality graph), then the corresponding solution of R has zero value, which implies that M is a minimum-cost perfect matching for G (by the primal-dual algorithm). If M is not perfect, then we proceed by considering the dual DR of R, which is written as follows:

$$\begin{aligned}
\text{maximize} \quad & \sum_{v \in U \cup W} \pi_v \\
\text{subject to} \quad & \pi_u + \pi_w \leq 0, \quad \forall (u,w) \in I \\
& \pi_v \leq 1, \quad \forall v \in U \cup W
\end{aligned}$$

Note that if we could restrict the value of each variable π_v (with $v \in U \cup W$) of DR in $\{-1, 1\}$, then the solution of DR would correspond to a maximum vertex cover of the equality graph (recall that a vertex cover of a graph is a subset of its vertices such that each edge has at least one endpoint in this set; see Exercise 6.4). To see this, observe that under this assumption maximizing $\sum_{v \in U \cup W} \pi_v$ is equivalent to minimizing the cardinality of the set $\{v \in U \cup W : \pi_v = -1\}$, which is a vertex cover by the first constraint of DR. In the following, we use this observation to find a feasible solution of DR, whose value is $2n - 2|M|$ (without applying the simplex method to solve DR). Hence, by Eq. (8.19) and by the strong duality theorem, this solution is optimal for DR.

Lemma 8.1 Let M be a maximum matching of the equality graph induced by the edges of I in $G = (U \cup W, E, c)$. Let B be the set containing the vertices of the equality graph that have been marked by the following rule:

"For each edge (u, w) in M with $u \in U$ and $w \in W$, if (u, w) belongs to an alternating path in the equality graph that starts at a free vertex in U, then mark vertex w ; otherwise mark vertex u."

> Set B is a vertex cover of the equality graph whose cardinality is $|M|$. Furthermore, B can be computed in $\mathcal{O}(n^2)$ time.

Proof. Since M is a matching and since B contains exactly one endpoint of each edge in M, it follows that $|B| = |M|$. To prove the lemma, it remains to prove that each edge in I has at least one marked endvertex (that is, in B). For a proof by contradiction, assume that there exists an edge (u, w) in I with $u \in U$ and $w \in W$ such that neither u nor w is marked. Since neither u nor w is marked, (u, w) does not belong to M. Hence, (u, w) belongs to $I \setminus M$.

If u is free, then w cannot be free, as otherwise (u, w) is an augmenting path with respect to M, which is a contradiction to the fact that M is maximum (by Corollary 6.1). Hence, w is matched in M. Let (u', w) be the edge of M incident to w. Now observe that $u \to w \to u'$ is an alternating path that starts at vertex u, which is free in U, and contains the edge (u', w). This implies that w is marked; a contradiction to the assumption that neither u nor w is marked.

It follows that u is matched in M. Let (u, w') be the edge of M incident to vertex u. Since (u, w') belongs to M and since u is not marked, vertex w' is marked, which implies that there exists an alternating path P that starts from a free vertex, say s, in U, and contains the edge (u, w'). Let P' be the subpath of P from s to u. Since P is alternating and since s belongs to U, edge (u, w') belongs to P'. If w is free, then $P' \cup \{(u, w)\}$ is an augmenting path with respect M, which is a contradiction to the fact that M is maximum (by Corollary 6.1). Hence, w is matched in M. Let (u', w) be the edge of M incident to w. Since w is not marked and since (u', w) belongs to M, it follows u' is necessarily marked. Hence, there exist no alternating path in the equality graph that starts from a free vertex in U containing the edge (u', w). However, this is a contradiction since $P' \cup \{(u, w), (w, u')\}$ is an alternating path that starts at s (which is free in U) and contains the edge (w, u'). We conclude the proof by mentioning that set B can be computed in time $\mathcal{O}(n + m) = \mathcal{O}(n^2)$ by applying an algorithm similar to Algorithm 6.3. $\qquad\square$

By Lemma 8.1, the solution of DR that we obtain by setting for each vertex v in $V \cup W$:

$$\pi_v = \begin{cases} -1, & \text{if } v \in B \\ 1, & \text{otherwise.} \end{cases}$$

is feasible for DR such that its value is $2n - 2|M|$. Thus, it is optimal by the discussion that we had earlier. Hence, the primal-dual method may proceed by setting y_v to $y_v + t\pi_v$ for each v in $V \cup W$, where t is defined as in Eq. (8.17) as follows:

$$t = \min_{\substack{j \in \bar{I} \\ \sum_{i=1}^m a_{ij}\bar{\pi}_i > 0}} \frac{c_j - \sum_{i=1}^m a_{ij} y_i}{\sum_{i=1}^m a_{ij}\bar{\pi}_i} = \min_{\substack{y_u + y_w < c(u,w) \\ \pi_u + \pi_w > 0}} \frac{c(u,w) - y_u - y_w}{\pi_u + \pi_w} \tag{8.20}$$

that is, labeling y_v of v is reduced by t, if v is marked (i.e., in B); otherwise, it is increased by t.

DRAFT

> **Theorem 8.8** Let $G = (U \cup W, E, c)$ be an edge-weighted complete bipartite graph with $|U| = |W| = n$. A minimum-cost perfect matching of G can be found in polynomial time by the primal-dual simplex method.

Proof. One iteration of the algorithm, involves computing the equality graph, a maximum matching of it and a vertex cover of equal cardinality. Hence, by Theorem 6.1 and Lemma 8.1, a single iteration can be implemented in $\mathcal{O}(n^{5/2})$ time. To bound the total number of iterations performed by the algorithm, we first observe that matching M can increase at most n times. Consider now a phase of the algorithm between two consecutive enlargements of M. Since at each iteration in such a phase, the equality graphs gets increased by one edge, it follows that the algorithm may perform at most $\mathcal{O}(n^2)$ iterations during a single phase. With a slightly more involved analysis, one can prove an upper bound of n iterations during a single phase. This actually follows from two facts; once a new edge enters the equality graph, both its endvertices become marked, and once a vertex becomes marked, then it stays marked in subsequent iterations of the same phase. This improved upper bound implies an overall time complexity of $\mathcal{O}(n^{9/2})$.   □

8.9 NP-completeness of integer linear programming

In several applications, one seeks for an integer solution to a linear program. Such linear programs are called integer linear programs (or ILPs, for short). In the following, we prove that integer linear programming is an $\mathcal{NP}$-complete problem by a reduction from the minimum vertex-cover problem, which is known to be $\mathcal{NP}$-complete.

Formally, the input of the vertex-cover problem is a connected, undirected graph $G = (V, E)$. The output consists of a minimum-cardinality subset U of the vertex-set V of G, such that for each edge $(u, v) \in E$ either $u \in U$ or $v \in U$ holds (see also Definition 9.6 in Section 9.2). Given an instance $G = (V, E)$ of the vertex-cover problem, one can easily formulate the problem of finding a minimum vertex-cover of graph G as an integer linear program as follows:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{v \in V} x(v) \\
\text{subject to} \quad & x(u) + x(v) \geq 1,\ \forall (u, v) \in E \\
& x(v) \in \{0, 1\}, \quad \forall v \in V
\end{aligned}
$$

The idea of the reduction is to introduce for every vertex $v \in V$ a binary variable $x(v) \in \{0, 1\}$, which describes whether v is in the vertex-cover or not. Now, since every edge $(u, v) \in E$ should be covered by at least one vertex, we require that the sum $x(v) + x(u)$ is at least 1. With these constraints, we now seek to minimize the cardinality of the vertex-cover and therefore the sum over all variables $x(v)$. Since this reduction can be obviously done in polynomial time and a solution of the constructed ILP implies a solution of the original vertex-cover instance, we can conclude that integer linear programming is NP-complete in general.

Exercises

Exercise 8.1   For each of the following linear programs, determine graphically whether the program is infeasible, unbounded, or has an optimal solution. Argue why your answer is correct.

a) $\quad$ maximize $\qquad x_1 \ + \ x_2$

$\qquad$ subject to $\qquad 3x_1 \ + \ x_2 \ \leq \ 6$

$\qquad\qquad\qquad - \ x_1 \ + \ 3x_2 \ \leq \ 3$

$\qquad\qquad\qquad\quad x_1 \qquad\qquad \leq \ 3$

b) $\quad$ maximize $\qquad x_1 \ + \ x_2$

$\qquad$ subject to $\qquad 3x_1 \ + \ x_2 \ \leq \ 6$

$\qquad\qquad\qquad - \ x_1 \ + \ 3x_2 \ \leq \ 3$

$\qquad\qquad\qquad\quad x_1 \qquad\qquad \geq \ 3$

c) $\quad$ maximize $\qquad x_1 \ + \ x_2$

$\qquad$ subject to $\qquad 3x_1 \ + \ x_2 \ \geq \ 6$

$\qquad\qquad\qquad - \ x_1 \ + \ 3x_2 \ \leq \ 3$

$\qquad\qquad\qquad\quad x_1 \qquad\qquad \geq \ 3$

Exercise 8.2   Transform the linear program given below into an equivalent linear program in standard form.

$$
\begin{aligned}
\text{minimize} \quad & 3\ x_1 \ + \ 4\ x_2 \ - \ 9\ x_3 \\
\text{subject to} \quad & 3\ x_1 \ - \ 1\ x_2 \ + \ x_3 \ \geq \ -7 \\
& - \ 6\ x_1 \ + \ 5\ x_2 \ - \ 5\ x_3 \ = \ 13 \\
& x_1 \ \geq \ 0, \ 2 \ \leq \ x_2 \ \leq \ 8
\end{aligned}
$$

Exercise 8.3   Solve the following linear program using the simplex algorithm following the instructions below.

$$
\begin{aligned}
\text{maximize} \quad & x_1 \ + \ 2\ x_2 \ + \ x_3 \\
\text{subject to} \quad 2\ & x_1 \ + \ 2\ x_2 \ + \ x_3 \ \leq \ 10 \\
& x_1 \ + \ 4\ x_2 \ + \ 2\ x_3 \ \leq \ 11 \\
& x_1, x_2, x_3 \ \geq \ 0
\end{aligned}
$$

- Introduce the slack variables and specify the new form of the equations.

- Describe all individual intermediate steps of the algorithm in detail.

- Describe the entering and the leaving variables of each step of the algorithm and how you computed them. If there exist more than one options for choosing an entering variable, then choose the one with the lowest index.

Exercise 8.4 Consider the following linear program.

$$
\begin{array}{rrrrcr}
\text{maximize} & -2\ x_1 & - & x_2 & & \\
\text{subject to} & -\ x_1 & - & x_2 & \leq & -1 \\
& -\ x_1 & -2\ x_2 & & \leq & -2 \\
& & & x_2 & \leq & 1 \\
& x_1, & x_2 & & \geq & 0
\end{array}
$$

a) Find a basic feasible solution of the linear program using phase I of the simplex algorithm.

b) Write the dual problem.

Exercise 8.5 Prove that both the following linear program and its dual are infeasible.

$$
\begin{array}{rrrrcr}
\text{maximize} & 2\ x_1 & - & x_2 & & \\
\text{subject to} & x_1 & - & x_2 & \leq & 1 \\
& -\ x_1 & + & x_2 & \leq & -2 \\
& x_1, x_2 & & & \geq & 0
\end{array}
$$

Exercise 8.6 Prove formally that the dual of the dual of a linear program is the primal linear program.

Exercise 8.7 In Section 8.2.1, we formulated the problem of computing a maximum flow as a linear program.

a) Compute the dual program of this linear program.

b) Argue why the dual program is a formulation of the minimum-cut problem.

Exercise 8.8 Consider the following linear program.

$$
\begin{array}{rrrrrcr}
\text{maximize} & 2\ x_1 & - & 6\ x_2 & & & \\
\text{subject to} & -\ x_1 & - & x_2 & - & x_3 & \leq & -2 \\
& 2\ x_1 & - & x_2 & + & x_3 & \leq & 1 \\
& x_1, x_2, x_3 & & & & & \geq & 0
\end{array}
$$

a) Using Phase I of simplex algorithm, compute a basic feasible solution of the linear program given below.

b) Using Phase II of simplex algorithm, compute the optimal solution of the linear program given below. In your solution, always apply Bland's rule.

Exercise 8.9 Solve the following linear programming problem:

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j$$

subject to

$$
\begin{aligned}
\sum_{j=1}^{n} q_j x_j &\leq \beta \\
x_j &\leq 1 \qquad \forall j = 1,\ldots,n \\
x_j &\geq 0 \qquad \forall j = 1,\ldots,n
\end{aligned}
$$

under the following conditions:

– $q_j > 0$ and $p_j > 0$, $\forall j = 1,\ldots,n$.

– $\sum_{j=1}^{n} p_j = \sum_{j=1}^{n} q_j = 1$.

– $\frac{p_1}{q_1} < \frac{p_2}{q_2} < \ldots < \frac{p_n}{q_n}$.

– β is a small number, such that $\frac{\beta}{q_j} < 1, \forall j = 1,2,\ldots,n$.

Hint: Write down the slack form of the system and consider what is a good option for the first entering variable.

Exercise 8.10 Use the complementary slackness theorem for testing the optimality of the given solutions.

a) $x^T = (1,0,3,4,0)$

$$
\begin{array}{rrrrrrrrrrrrl}
\text{maximize} & 14 & x_1 & + & 22 & x_2 & + & 4 & x_3 & - & 18 & x_4 & + & 11 & x_5 & \\
\text{subject to} & 4 & x_1 & + & 5 & x_2 & & & & & & & - & 6 & x_5 & \leq & 4 \\
& 3 & x_1 & + & & x_2 & + & 2 & x_3 & + & & x_4 & + & 3 & x_5 & \leq & 15 \\
& 2 & x_1 & - & 2 & x_2 & - & 3 & x_3 & & & & + & 7 & x_5 & \leq & -7 \\
& & x_1 & + & 8 & x_2 & + & 5 & x_3 & - & 9 & x_4 & + & 4 & x_5 & \leq & -20 \\
& & & & & & & & x_1, & x_2, & x_3, & x_4, & x_5 & \geq & 0
\end{array}
$$

b) $x^T = (2,1,0)$

$$
\begin{array}{rrrrrrrrl}
\text{maximize} & 9 & x_1 & + & 2 & x_2 & + & 4 & x_3 & \\
\text{subject to} & 4 & x_1 & & & & + & & x_3 & \leq & 8 \\
& 3 & x_1 & + & & x_2 & + & 2 & x_3 & \leq & 9 \\
& & x_1 & + & 2 & x_2 & + & 3 & x_3 & \leq & 4 \\
& & & & x_1, & x_2, & x_3 & \geq & 0
\end{array}
$$

Exercise 8.11 Solve the following linear program using the primal-dual simplex method; as an initial solution for the dual problem, use $y = (0,0)^T$.

$$
\begin{array}{rrrrrrl}
\text{minimize} & 2 & x_1 & + & & x_2 & \\
\text{subject to} & & x_1 & - & 2 & x_2 & = & 2 \\
& - & x_1 & + & 3 & x_2 & = & 1 \\
& & & x_1, & x_2 & \geq & 0
\end{array}
$$

Exercise 8.12 Let $p_1, p_2, \ldots, p_n$ be n points with integer coordinates on the plane $\mathbb{Z}^2$. Every point p_i is associated with a circle C_i that is centered at p_i. For each of the following cases, the task is to determine a radius r_i for each circle C_i, such that no two circles C_i and C_j corresponding to two distinct points p_i and p_j intersect:

a) The minimum radius among all circles is maximized.

b) The sum of the circumferences of all circles is maximized.

c) All circles must have the same radius which is maximized.

Formulate a linear program that computes for the given set of points the set of corresponding circles that optimizes each of the objectives above.

Exercise 8.13 Let n be a positive integer. Now consider an arrangement of n vertices $v_1, v_2, \ldots, v_n$ along a horizontal line ℓ in this order. The goal is to determine the maximum number of edges that one can draw as half-circles in the half-plane above ℓ so that no two half-circles corresponding to two independent edges nest each other. Recall that two edges that do not share a common endpoint are called independent. Note that a nesting occurs when both endpoints of one edge are located in between the two endpoints of another edge. An example for $n = 6$ which achieves 7 edges is given below.
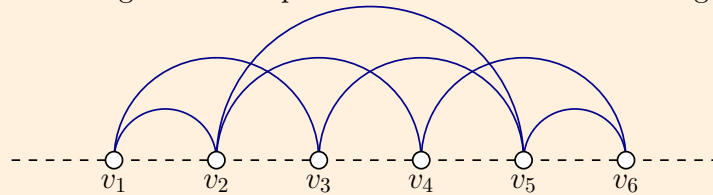

a) Give a solution to the problem that achieves at least $2n - 3$ edges, for any $n \geq 3$.

b) Formulate the problem as a linear programming problem, where binary variables are allowed.

c) Give a formulation of the problem with the ZIMPL programming language (see http://zimpl.zib.de), where n must be a parameter in your formulation.

d) We conjecture that one cannot achieve more than $2n - 3$ edges. Test this conjecture for small values of n using the SCIP framework to compute the optimal solution for the different values of n as given in the following table.

$n = 5$	$n = 6$	$n = 7$	$n = 8$	$n = 9$	$n = 10$

Hints: For your formulation, you will need so-called binary variables that can only have one of two values, either zero or one. The ZIMPL manual contains several useful examples: http://zimpl.zib.de/download/zimpl.pdf. To get a list of all available commands of SCIP, simply type help in the console of SCIP. SCIP is able to read files in ZIMPL format, if the input file has .zpl extension.

Exercise 8.14 Let n and k be two positive integers such that $k \leq n$. Now consider an arrangement of n vertices $v_1, v_2, \ldots, v_n$ along the boundary of a circle $\mathcal{C}$ in this order. The goal is to determine the maximum number of edges that one can draw as straight-line segments in the interior of $\mathcal{C}$ so that no edge is intersected more than k times. An example for $n = 6$ and $k = 3$ is given in the following figure, where one can see that it is possible to draw 6 edges between consecutive vertices along the boundary of $\mathcal{C}$ (black colored) plus 8 additional edges (gray colored) leading to a solution with 14 edges in total (observe that edge (v_2, v_5) is not present in the figure).
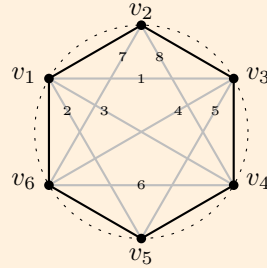
a) Formulate this problem as a linear programming problem, where integer and binary variables are allowed.

b) Give a formulation of this problem with the ZIMPL programming language (see http://zimpl.zib.de), where n and k must be parameters in your formulation.

c) Use the SCIP framework to compute the optimal solution for the different values of n and k as given in the following table.

	$n = 6$	$n = 7$	$n = 8$	$n = 9$	$n = 10$
$k = 2$					
$k = 3$					
$k = 4$					
$k = 5$					

Hint: Given two positive variables X and Y, a constraint of the form $X \leq Y$ which must hold if (binary) variable $Z \in \{0,1\}$ is 1 can be expressed by a constraint of the following form:

$$X \leq Y + M \cdot (1 - Z),$$

where M is the maximum value that can be assigned to either X or Y. Indeed, if Z equals to 0, then the constraint is: $X \leq Y + M$, which is always true. On the other hand, if Z equals to 1, then the constraint is: $X \leq Y$, which is exactly the initial constraint that must be fulfilled.

Further reading

– Chapter 1-5 from: R. J. Vanderbei. Linear programming - foundations and extensions, volume 4 of Kluwer international series in operations research and management service. Kluwer, 1998.

9. Approximation Algorithms

This chapter will focus on approximation algorithms. In general, these are special types of algorithms, which run in polynomial-time and guarantee that the computed solution is close to the optimum. They are of high importance especially for solving $\mathcal{NP}$-complete problems, since an optimal solution of such a problem cannot be computed efficiently, unless $\mathcal{P} = \mathcal{NP}$ holds. In this regards, approximation algorithms often produce solutions that are sufficiently good. The rest of this chapter is organized as follows: In Section 9.1, we give important definitions and notations. In Sections 9.3–9.5, we study simple approximation algorithms for a variant of the well-known travelling salesman problem, for the knapsack problem and for the bin-packing problem, respectively.

9.1 Definitions

Let Π be a problem and let I be an instance of problem Π. We further assume that Π is an optimization problem, that is, either a maximization (e.g., of the profit) problem or a minimization (e.g., of the total cost) problem. In most of the examples that we will consider in this chapter, problem Π is usually an $\mathcal{NP}$-complete problem.

Definition 9.1 (Approximation factor) An algorithm ALG for a problem Π has an approximation factor of $a(n)$ if for any input instance I of problem Π with size n, the returned value of the solution produced by the algorithm, denoted by $ALG(I)$, is within a factor of $a(n)$ of the value $OPT(I)$ of the optimal solution.

If Π is a maximization problem, then:

$$ALG(I) \geq \frac{1}{a(n)} \cdot ALG(I).$$

If Π is a minimization problem, then:
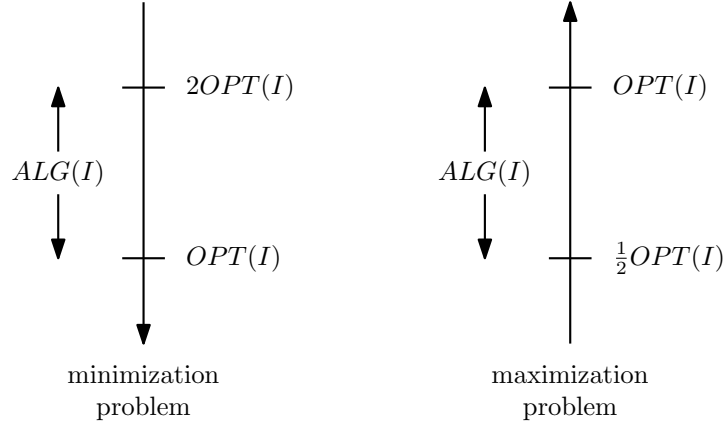
$$ALG(I) \leq a(n) \cdot OPT(I).$$

Figure 9.1: Illustration for an algorithm with approximation factor $a = 2$ applied on an instance I of a problem Π, assuming that Π is a minimization problem (left) or a maximization problem (right). The range $ALG(I)$ contains the possible values that can be returned by the algorithm, where $OPT(I)$ denotes the value of the solution of I.

Definition 9.2 (Approximation algorithm) We refer to an algorithm whose approximation factor is $a(n)$ as an $a(n)$-approximation algorithm. If $a(n)$ is a function independent of n (i.e., a constant), then we say that the algorithm has a constant approximation factor of a.

Observation 9.1 A 1-approximation algorithm for a problem always returns an optimal solution.

In the following, we introduce special types of approximation algorithms.

Definition 9.3 (Approximation scheme) An approximation scheme for a problem Π is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\epsilon > 0$, such that for any fixed ϵ, the scheme is a $(1 + \epsilon)$-approximation algorithm.

Definition 9.4 (PTAS) An algorithm is called polynomial-time approximation scheme (or PTAS, for short), if it is an approximation scheme and its running time is polynomial in the size n of the instance for every fixed $\epsilon > 0$.

Definition 9.5 (FPTAS) An algorithm is called fully polynomial-time approximation scheme (FPTAS) if it is an approximation scheme and its running time is polynomial in the size n of the instance and $\frac{1}{\epsilon}$, i.e., the time complexity of the algorithm is in $\mathcal{O}((\frac{1}{\epsilon})^d \cdot n^c)$ for some constants $c, d > 0$.

> **Observation 9.2** A FPTAS provides a tradeoff between time complexity and accuracy. If the value of ϵ is small, then the accuracy improves but the runtime order increases. If the value of ϵ is large, then the accuracy weakens but the runtime order improves.

Let I be an instance of a maximization problem Π and let $OPT(I)$ be the value of the optimal solution of I. To prove that an algorithm has approximation factor is $a(n)$, we need to compare against $OPT(I)$, which is a value that we do not know. So, how can we justify the approximation factor of the algorithm? The critical observation here is that a lower bound for $OPT(I)$ suffices. More precisely, suppose that $B(I) \leq OPT(I)$ holds for some function B. If we can prove that the value $ALG(I)$ returned by our algorithm is such that $ALG(I) \leq a(n) \cdot B(I)$, then $ALG(I) \leq a(n) \cdot OPT(I)$ follows. If Π is minimization, an upper bound is needed.

9.2 Vertex Cover

In this section, we study a well-known combinatorial problem, called vertex cover problem, which asks the following question: "Given a graph, find a subset of its vertices that is of minimum cardinality such that each edge is covered, that is, it has at least one endvertex in the set" (see Fig. 9.2 for an example). Formally, the problem is defined as follows.

> **Definition 9.6** (Vertex cover problem) The input of the vertex cover problem is a connected, undirected graph $G = (V, E)$. The output specifies a set $S \subseteq V$ of vertices that is of minimum cardinality such that for each edge (u, v) in E, $u \in S$ or $v \in S$ holds.
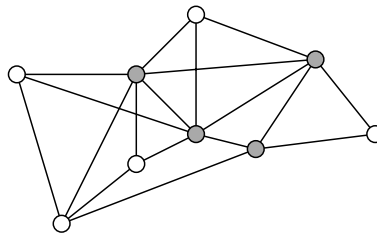

Figure 9.2: Illustration of a vertex cover (highlighted in gray) of a graph.

It is well-known that the vertex cover problem is an $\mathcal{NP}$-complete problem (as also mentioned in Section 8.9). In the following, we give two different 2-approximation algorithms.

> **Theorem 9.1** There exists a 2-approximation algorithm for vertex cover.

Proof. As already mentioned, we will prove the theorem by giving two different algorithms. Let I be an instance of the vertex cover problem consisting of a graph $G = (V, E)$. The first

algorithm is a rather simple greedy algorithm. Pick an edge of G and add its end-vertices to the solution. Then, remove all the edges incident to the added vertices from G and repeat as long as G has some edges. If we denote by $e_1, \ldots, e_m$ the edges of G chosen by the algorithm, then the cardinality of the returned set is $2m$, that is $ALG(I) = 2m$. On the other hand, we observe that $e_1, \ldots, e_m$ form a maximal matching in G, which implies that m is a lower bound for $OPT(I)$, that is, $m \leq OPT(I)$ (given a matching of a graph, one cannot use the same vertex to cover more than one matching edge). Combining these two observations together we conclude $ALG(I) \leq 2OPT(I)$.

The second algorithm is based on the following relaxation of the integer linear program that we presented in Section 8.9 for solving the vertex cover problem optimally.

$$
\begin{aligned}
\text{minimize} \quad & \sum_{v \in V} x(v) \\
\text{subject to} \quad & x(u) + x(v) \geq 1, \ \forall (u,v) \in E \\
& 0 \leq x(v) \leq 1, \quad \forall v \in V
\end{aligned}
$$

Let $\bar{x} = \{\bar{x}(v) : v \in V\}$ be an optimal solution of the relaxed linear program. Based on this solution, we computer the following subset of the vertex-set of G:

$$
S = \{v \in V : \bar{x}(v) \geq \frac{1}{2}\}.
$$

It is not difficult to see that S covers all edges of G, since for each edge (u,v) of G, it holds that $\bar{x}(u) + \bar{x}(v) \geq 1$, which implies that $\bar{x}(u) \geq \frac{1}{2}$ or $\bar{x}(v) \geq \frac{1}{2}$, or equivalently that $u \in S$ or $v \in S$. Let $ALG(I)$ be the cardinality of S and let $OPT(I)$ be the cardinality of a minimum vertex cover S_{opt}. Since an optimal solution for the relaxed linear program lower-bounds any solution of the corresponding integer linear program (and thus $|S_{opt}|$), we obtain

$$
OPT(I) \quad = \quad \sum_{v \in S_{opt}} 1 \quad \geq \quad \sum_{v \in S} \bar{x}(v) \quad \geq \quad \frac{1}{2} \cdot \sum_{v \in S} 1 \quad = \quad \frac{1}{2} \cdot ALG(I) \qquad \square
$$

9.3   The traveling salesman problem

In this section, we focus on a well-known combinatorial problem, called traveling salesman problem, which asks the following question: "Given a set of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?". Formally, the problem is defined as follows.

> **Definition 9.7** (TSP) Let $G = (V, E, w)$ be a complete weighted graph, where $w : E \to \mathbb{N}^+$ is a weight function and $k \in \mathbb{N}$ an integer. The decision version of the traveling salesman problem (or TSP, for short) asks whether there exist a cycle visiting every vertex of G exactly once such that its weight is at most k? The optimization version of the problem asks for the minimum value of k for which a corresponding cycle exists.

DRAFT

It is well-known that the traveling salesman problem is an $\mathcal{NP}$-complete problem. In the following, we prove that it cannot be even approximated by a computable function $a(n)$, unless $\mathcal{P} = \mathcal{NP}$.

> **Theorem 9.2** For any polynomial computable function $a(n)$, TSP cannot be approximated by a factor $a(n)$, unless $\mathcal{P} = \mathcal{NP}$.

Proof. Our proof is by a reduction from the Hamiltonian cycle problem, which is known to $\mathcal{NP}$-complete. Recall that given a graph G, the Hamiltonian cycle problem asks whether there exists a cycle in G passing through each vertex of G exactly once.

For the proof, assume to the contrary that there exists a polynomial-time approximation algorithm ALG with factor $a(n)$, i.e.,

$$ALG(I) \leq a(n) \cdot OPT(I), \ \forall \text{ instance } I \text{ of TSP.}$$

We show that ALG can be used to decide the Hamiltonian cycle problem in polynomial-time (which is $\mathcal{NP}$-complete). The idea is to reduce the Hamiltonian cycle problem to TSP. Let $G = (V, E)$ be an instance of the Hamiltonian cycle problem. We construct an instance $G' = (V, V \times V, w)$ of TSP as follows:

$$w(e) = \begin{cases} 1, & \text{if } e \in E \\ a(n) \cdot n, & \text{otherwise.} \end{cases}$$

Then, it is not difficult to see that the following hold:

- If G is Hamiltonian, the weight of TSP in G' is exactly n.

- If G is not Hamiltonian, the weight of TSP in G' is strictly greater than $a(n) \cdot n$.

Hence, we can decide the Hamiltonian cycle problem based on the output of ALG, as follows:

- If the weight of TSP in G' is $\leq a(n) \cdot n$, then G is Hamiltonian.

- If the weight of TSP in G' is $> a(n) \cdot n$, then G is not Hamiltonian.   □

Theorem 9.2 suggests that one should not hope for an approximation algorithm for the traveling salesman problem, unless $\mathcal{P} = \mathcal{NP}$. In the following, we study a natural restriction of the traveling salesman problem which allows constant factor approximations.

> **Definition 9.8** (Metric TSP) The metric TSP problem is a restriction of the traveling salesman problem, in which the edges of the input graph G satisfy the triangle inequality, that is, for every triple of edges (u, v), (v, w) and (u, w) of G the following holds:
>
> $$w(u, w) \leq w(u, v) + w(v, w).$$

$$H : 1 \to 2 \to 1 \to 3 \to 4 \to 3 \to$$
$$5 \to 3 \to 1 \to 6 \to 1$$
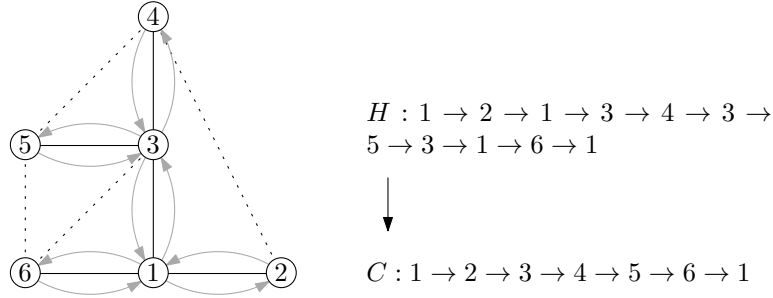
$$C : 1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 1$$

Figure 9.3: Illustration for the proof of Theorem 9.3.

Theorem 9.3 There exists a 2-approximation algorithm for metric TSP.

Proof. We start by describing the algorithm, which first computes a minimum spanning tree T of the input graph G; see Chapter 3. Then, it computes a (non-simple) graph H that is obtained by walking around T. Finally, it returns the Hamiltonian cycle C that is obtained by short-cutting H to remove vertex-duplicates.

Example 9.1 Fig. 9.3 illustrates the construction of graphs T, H and C for an input graph G containing $n = 6$ vertices (illustrated by black solid and dotted edges). Initially, a minimum spanning tree T of G is found (illustrated by black solid edges). Then, graph H is constructed by walking around T (illustrated by gray solid edges) and finally cycle C is obtained by short-cutting H.

Then, by definitions of graphs T and H:

$$w(H) = 2w(T).$$

Now, consider cycle C. Since each short-cut only decrease the total weight, by the triangle inequality we obtain:

$$w(C) \le w(H) = 2w(T). \tag{9.1}$$

We now claim that $w(T) \le OPT$ holds, where OPT denotes the value of the optimal solution. To prove the claim, let C^* be the (optimal) cycle of G for TSP, i.e., $w(C^*) = OPT$. Remove one edge of C^* and obtain a path P. Then, since T is an MST and P is a trivial tree, we obtain:

$$w(T) \le w(P).$$

On the other hand, since C^* contains all edges of P plus one, we obtain:

$$w(P) \le w(C^*).$$

Hence, $w(T) \le w(P) \le w(C^*) = OPT$ and the claim follows.

Combining with Eq. (9.1), we obtain $w(C) \le 2w(T) \le 2OPT$ □

Proof. Consider the complete graph consisting of n vertices $v_0, \ldots, v_{n-1}$ whose edges have unit weight except for $n-1$ edges in $\{(v_1, v_2), (v_2, v_3) \ldots, (v_{n-1}, v_1)\}$, which have weight 2; see Fig. 9.4a. A solution for TSP with weight n is clearly possible. Fig. 9.4b illustrates the result of the first two steps of the algorithm of Theorem 9.3, where the black edges forming a star at v_0 are those of the minimum spanning tree and the grey ones form the associated graph H. Fig. 9.4c illustrates the cycle returned by the algorithm, whose total weight is $2n-2$, thus proving that $w(C) \approx 2OPT$. $\square$
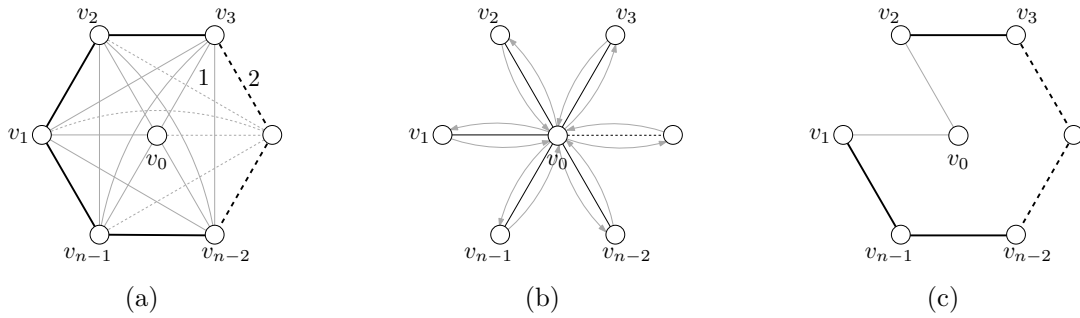
Figure 9.4: Illustration for Observation 9.3.

At this point, it is natural to ask whether it is possible to obtain an algorithm whose approximation factor is better (i.e., smaller) than the one provided by the algorithm of Theorem 9.3. The following theorem, by Christofides, answers this question in the positive.

Theorem 9.4 (Christofides) There exists a $\frac{3}{2}$-approximation for metric TSP.

Proof. Consider the following Algorithm 9.1, which first computes a minimum spanning tree T of the input graph G and then a minimum cost perfect matching M of G on the set V_o of vertices of odd degree of T. Finally, the algorithms computes that union H of T and M and outputs the vertices in their first appearance in a Euler tour of H (i.e., a cycle that passes through each edge exactly once).

Algorithm 9.1: The algorithm by Christofides

Input   : An edge-weighted complete graph $G = (V, E)$ such that the edge-weights
             satisfy the triangle inequality.
Output : A Hamiltonian cycle of G.

1 $T \leftarrow$ a minimum spanning tree T of G;
2 $M \leftarrow$ a minimum cost perfect matching on the set V_o of odd degree vertices of T;
3 $H \leftarrow T \cup M$;
4 return the vertices in their first appearance in a Euler tour of H;

Note that Christofides' algorithm applied to the graph of Fig. 9.4 results in an optimal solution, as a minimum-cost perfect matching on the odd-degree vertices of the star of Fig. 9.4b does contain edges with weight 2. To prove formally the correctness of Algorithm 9.1, we make few observations.

> **Observation 9.4** $|V_o| \geq 1$ is even.

Proof. Since each leaf in T has degree 1 and since T has at least two leaves, $|V_o| \geq 1$ holds. The fact that the sum of the degrees of the vertices of a graph is even implies that $|V_o|$ is even. $\square$

> **Corollary 9.1** There exists a minimum cost perfect matching on the set of odd degree vertices of T.

> **Observation 9.5** All vertices of the graph H have an even degree by construction.

Proof. It follows by construction. In particular, the odd-degree vertices of T are bridged by in edge in H. Hence, they have even degree. On the other hand, the even-degree vertices of T do not have any other incident edge in H. This completes the proof. $\square$

Since a graph is Eulerian if and only if each vertex of it has even degree, the following follows as an immediate consequence of Observation 9.5.

> **Corollary 9.2** There exists an Euler tour of H.

Let OPT be the value of the optimal solution. Then, as in the proof of Theorem 9.3 we can prove that $w(T) \leq OPT$. So, in order to complete the theorem, it suffices to prove that $w(M) \leq \frac{OPT}{2}$, which is proven in the following claim.

> **Claim 9.1** $w(M) \leq \frac{OPT}{2}$.

Proof. Let C^* be the optimal cycle of G for TSP, i.e., $w(C^*) = OPT$. Short-cut C^* on V_o to obtain C, i.e., short-cut every vertex of C^* in $V \setminus V_o$. Then, we get $w(C) \leq OPT$. The cycle C can be decomposed into two matchings M_1 and M_2. Assume w.l.o.g. that $w(M_1) \leq w(M_2)$. Then, $w(M_1) \leq \frac{OPT}{2}$. Since M is a min-cost perfect matching on $V_o \subseteq V$, $w(M) \leq w(M_1)$. Hence, $w(M) \leq \frac{OPT}{2}$. $\square$
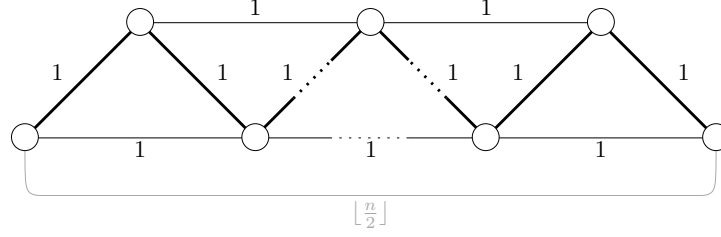
Figure 9.5: Illustration for Observation 9.6.

We are now ready to put everything together. Let ALG be the cost of the cycle returned by Algorithm 9.1. Then, $ALG \leq w(H)$ holds. By triangle inequality, we know that $w(H) \leq w(T) + w(M)$. Finally, as already stated $w(T) \leq OPT$ holds, while by Claim 9.1 $w(M) \leq \frac{OPT}{2}$ holds. Hence:

$$ALG \leq w(H) \leq w(M) + w(T) \leq \frac{OPT}{2} + OPT = \frac{3}{2}OPT. \qquad \square$$

> **Observation 9.6**   The analysis of Algorithm 9.1 in Theorem 9.4 is tight.

Proof. Consider the graph illustrated Fig. 9.5, where each black edge has weight 1, while the gray edge has weight $\lfloor \frac{n}{2} \rfloor$. Algorithm 9.1 may compute as a minimum spanning tree the path given by the bold black edges of Fig. 9.5, which has weight $(n-1)$ and contains exactly two odd-degree vertices (the endpoints of the path). Algorithm 9.1 then adds the edge joining the two odd-degree vertices and returns the computed TSP cycle, whose total weight is: $(n-1) + \lfloor \frac{n}{2} \rfloor$, while a Hamiltonian cycle with weight n clearly is possible. $\square$

9.4   The knapsack problem

In this section, we present an approximation scheme for the knapsack problem, which is known to be an $\mathcal{NP}$-complete problem. In the following, we start by recalling the formal definition of the knapsack problem.

> **Definition 9.9**   (Knapsack problem) The input of the knapsack problem consists of a "knapsack" of size B and a set $I = \{a_1, \ldots, a_n\}$ of n items, such that item a_i has size $s(a_i)$ and profit $p(a_i)$, $i = 1, \ldots, n$. In the output, we seek to compute the most profitable subset of items that can fit in the knapsack; formally, a non-empty subset $S \subseteq I$ whose size $s(S) = \sum_{x \in S} s(x)$ is at most B and whose profit $p(S) = \sum_{x \in S} p(x)$ is maximum.

Before trying to derive an algorithm to solve the knapsack problem, we give first an example of an instance of this problem.

Example 9.2 Let $I = \{a_1, ..., a_5\}$ be a set of five items, which size and profit is as given in Fig. 9.6. Assume that the knapsack has size $B = 11$. Observe that the subset $S = \{a_3, a_4\}$ of I has maximum profit $p(S) = 40$ over all subsets of I that can fit within the knapsack and its size is exactly equal to 11, i.e., the knapsack size.

item	size	profit
a_1	1	1
a_2	2	2
a_3	5	18
a_4	6	22
a_5	7	28
	$B = 11$	

Figure 9.6: Example of an instance of the knapsack problem with five items.

We first give a solution to the knapsack problem via dynamic programming. For this, we first observe that if P is the profit of the most profitable item, i.e., $P = \max\limits_{1 \leq i \leq n} p(a_i)$, then the value OPT of the optimum solution is clearly upper bounded by nP, i.e.:

$$OPT \leq nP.$$

Let $S(i, p)$ be a subset of $\{a_1, ..., a_n\}$ whose total profit is exactly p, if any. Let also $A(i, p)$ be the size of $S(i, p)$, i.e.:

$$A(i, p) = \begin{cases} \sum\limits_{a \in S(i,p)} s(a), & \text{if } S(i,p) \text{ exists} \\ +\infty, & \text{otherwise} \end{cases}$$

Then, in the base case of our dynamic program algorithm, in which $i = 1$ holds, it is not difficult to compute $A(i, p)$, for each $1 \leq p \leq nP$, as follows:

$$A(1, p) = \begin{cases} s(a_1), & p = s(a_1) \\ +\infty, & \text{otherwise} \end{cases}$$

Assume now that we have computed $A(j, p)$ for each $1 \leq j \leq i$ and $1 \leq p \leq nP$. Then, we can compute $A(i + 1, p)$ for each $1 \leq p \leq nP$, as follows:

$$A(i + 1, p) = \begin{cases} \min\{A(i, p), s(a_{i+1}) + A(i, p - p(a_{i+1}))\}, & \text{if } p(a_{i+1}) \leq p \\ A(i, p), & \text{otherwise} \end{cases}$$

Having computed with the recursive formula above all entries of table A, the algorithm will clearly return the maximum value of p for which $A(n, p) \leq B$ holds, namely:

$$\max \{p \mid A(n, p) \leq B\}.$$

Since A is of size $n \times nP$ and each entry of it can be computed in $\mathcal{O}(1)$ time, the time complexity

DRAFT

of the algorithm is $\mathcal{O}(n^2 P)$. However, observe that the profit P of the most profitable item might not be polynomial in n.

> **Definition 9.10** An algorithm whose time complexity depends on a parameter that might not be polynomial in the size of the input is commonly refereed to as pseudo-polynomial.

The critical observation to derive an approximation scheme for the knapsack problem is that if all the profits of the items are small, i.e., they are bounded by a polynomial in n, then the algorithm that we gave above has polynomial running time, i.e., its time complexity becomes bounded by a polynomial in n.

Algorithm 9.2: An approximation scheme for knapsack.

   Input   : n items $a_1, \ldots, a_n$, a knapsack-size B and a real number $\varepsilon > 0$.
   Output : A subset of the items that can fit in the knapsack.

1 $K \leftarrow \varepsilon \cdot \frac{P}{n}$;
2 foreach $i = 1$ to n do
3     $p^*(a_i) = \lfloor \frac{p(a_i)}{K} \rfloor$;
4 end
5 With the new profits, find the most profitable set S using the dynamic program;
6 return S;

Clearly, the time complexity of Algorithm 9.2 is $\mathcal{O}(n^2 \cdot \lfloor \frac{P}{K} \rfloor) = \mathcal{O}(n^2 \cdot \lfloor \frac{n}{\varepsilon} \rfloor)$, that is, polynomial in n and $\frac{1}{\varepsilon}$. The following theorem further establishes that Algorithm 9.2 is an FPTAS for knapsack (see Definition 9.5).

> **Theorem 9.5** The profit $p(S)$ of the set S returned by Algorithm 9.2 is at least $(1 - \varepsilon)$ times the value OPT of the optimal solution of the knapsack instance, namely:
>
> $$p(S) \geq (1 - \varepsilon)OPT$$

Proof. Let S_{opt} be the indices of the items of an optimal set for the knapsack instance. Since for every $x \in \mathbb{R}$ it holds that $x - 1 \leq \lfloor x \rfloor$, we have:

$$p^*(a_i) = \left\lfloor \frac{p(a_i)}{K} \right\rfloor \Rightarrow \frac{p(a_i)}{K} - 1 \leq p^*(a_i)$$
$$\Leftrightarrow p(a_i) - K \leq K p^*(a_i)$$
$$\Leftrightarrow p(a_i) - K p^*(a_i) \leq K.$$

Summing up over i in S_{opt}:

$$\sum_{i \in S_{opt}} (p(a_i) - K p^*(a_i)) \leq K \cdot |S_{opt}|.$$

Since $|S_{opt}| \leq n$, we obtain:
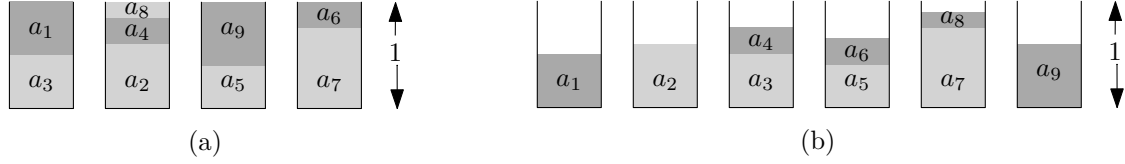
$$p(S_{opt}) - K p^*(S_{opt}) \leq K \cdot n.$$

(a)   (b)

Figure 9.7: Different packings of nine items $a_1,\ldots,a_9$ such that $s(a_1) = 0.5$, $s(a_2) = 0.7$, $s(a_3) = 0.5$, $s(a_4) = 0.2$, $s(a_5) = 0.4$, $s(a_6) = 0.2$, $s(a_7) = 0.5$, $s(a_8) = 0.1$ and $s(a_9) = 0.6$: (a) the optimal one, and (b) the one produced with the Next-Fit algorithm.

Since for every $x \in \mathbb{R}$ it holds that $\lfloor x \rfloor \leq x$ and since $p^*(S) \geq p^*(S_{opt})$, by the inequality above, we obtain:

$$p(S) \geq K \cdot p^*(S) \geq K \cdot p^*(S_{opt}) \geq p(S_{opt}) - K \cdot n = OPT - \epsilon \cdot P$$
$$\geq OPT - \epsilon \cdot OPT = (1 - \epsilon) \cdot OPT. \qquad \square$$

> **Corollary 9.3**   The approximation scheme given above is an FPTAS for knapsack.

9.5   Bin packing

In this section, we will discuss the bin packing problem, which is another well-known $\mathcal{NP}$-complete problem. Before we proceed with the introduction of a simple 2-approximation algorithm for this problem, we recall the formal definition of the bin packing problem.

> **Definition 9.11** (Bin packing problem) The input of the bin packing problem consists of n items $a_1,\ldots,a_n$ such that a_i has size $s(a_i) \in (0,1]$, $i = 1,\ldots,n$. In the output, we seek to compute a packing of the input items into unit-sized bins such that the number of bins used is minimum.

> **Example 9.3**   Let $I = \{a_1,\ldots,a_9\}$ be a set of nine items such that $s(a_1) = 0.5$, $s(a_2) = 0.7$, $s(a_3) = 0.5$, $s(a_4) = 0.2$, $s(a_5) = 0.4$, $s(a_6) = 0.2$, $s(a_7) = 0.5$, $s(a_8) = 0.1$ and $s(a_9) = 0.6$. Fig. 9.7a shows an optimal packing of these items into four bins. The fact that the packing is optimal easily follows from the observation that all bins are full.

As usual, in the following we denote by OPT be the value of an optimal solution of an instance of the bin packing problem with n items. Then, the following observation immediately follows from the problem definition

> **Observation 9.7** $\sum_{i=1}^{n} s(a_i) \leq OPT$

The first algorithm that we are going to study for the bin packing problem is a greedy one, called next-fit; see Algorithm 9.3. The algorithm is processing the items in any order. If the current item fits into the most-recently used bin, then the algorithm places it there. Otherwise, it starts a new bin and places the item into the new bin.

Algorithm 9.3: The Next-Fit algorithm.

Input   : n items $a_1,\ldots,a_n$ such that a_i has size $s(a_i) \in (0,1]$, $i = 1,\ldots,n$.
Output : A packing of the items into bins of unit size.

1 for $i = 1$ to n do
2 | if a_i fits into the most-recently used bin then
3 | | place a_i into the most-recently used bin;
4 | end
5 | else
6 | | introduce a new bin B and place a_i into B;
7 | end
8 end

> **Theorem 9.6** The next-fit algorithm is a 2-approximation algorithm for the bin-packing problem.

Proof. Let k be the number of bins used by the next-fit algorithm, and assumed that k is even (in order to simplify the presentation). To prove that the next-fit algorithm is a 2-approximation algorithm for the bin-packing problem, it suffices to show that $k \leq 2 \cdot OPT$. Denote by $B_1,\ldots,B_k$ the actual bins used by the next-fit algorithm. For $i = 1,\ldots,k$, let $s(B_i)$ be the sum of the sizes of the items assigned to bin B_i, namely:

$$s(B_i) = \sum_{x \in B_i} s(x)$$

Then, we observe that the sum of the sizes of the items of every pair of consecutive bins is strictly greater than 1 (as otherwise the next-fit algorithm wouldn't have started a new bin). This implies:

$$1 \;<\; s(B_1) + s(B_2)$$
$$\ldots$$
$$1 \;<\; s(B_{k-1}) + s(B_k)$$

By summing the inequalities above, we obtain:

$$\frac{k}{2} < \sum_{i=1}^{n} s(a_i) \quad \Rightarrow \quad k < 2 \cdot \sum_{i=1}^{n} s(a_i)$$

Hence, the proof follows from Observation 9.7. Let n be a multiple of 4, δ be $2/n$ and consider the following instance $\{a_1, a_2, \ldots, a_n\}$ of bin-packing, in which a_i is 0.5, if i is odd, and δ, otherwise. Since all even-indexed items fit in a single bin, while pairwise all odd-indexed items fit in a single bin, the optimal solution uses $\frac{n}{4}$ foul bins. If next-tit algorithm processes the items in the order $a_1, a_2, \ldots, a_n$, then it will need $\frac{n}{2}$ bins. Hence, our analysis is tight. $\square$

Theorem 9.6 implies that the bin packing problem admits a constant factor approximation algorithm with a factor of 2. So, it is natural to ask whether a better approximation factor is possible. In the following theorem, we show that this factor cannot be strictly less that $\frac{3}{2}$.

> **Theorem 9.7** For any $\varepsilon > 0$, the bin packing problem cannot be approximated by a factor of $3/2 - \varepsilon$, unless $\mathcal{P} = \mathcal{NP}$.

Proof. We give the proof of the theorem by a reduction from the well-known $\mathcal{NP}$-complete partition problem. In the following, we first recall the formal definition of the partition problem.

> **Definition 9.12** (Partition problem) The input of the partition problem consists of n items a_1, ..., a_n such that a_i has size $s(a_i)$, $i = 1,\ldots,n$. In the output, we seek to compute a subset A of $\{a_1,\ldots,a_n\}$ (if any) such that
> $$\sum_{a \in A} s(a) = \sum_{a \notin A} s(a)$$

For a contradiction, assume that there exists a $(3/2 - \varepsilon)$ approximation algorithm $\mathcal{A}$ for bin packing, and consider an instance $\langle a_1,\ldots, a_n \rangle$ of the partition problem. Then, it is not difficult to see that the following hold:

- If $\langle a_1,\ldots, a_n \rangle$ is a no-instance, then $\mathcal{A}$ will use at least two bins;

- Otherwise, $\mathcal{A}$ will use at most $(3/2 - \varepsilon) \cdot 2 < 3$ bins.

Thus, we can conclude that there exists a polynomial-time algorithm for the partition problem, which is a contradiction (unless $\mathcal{P} = \mathcal{NP}$). $\qquad\square$

> **Corollary 9.4** There is no PTAS for bin packing ,unless $\mathcal{P} = \mathcal{NP}$

In view of Corollary 9.4, in the following we seek in finding an asymptotic PTAS for the bin packing problem, i.e., an algorithm which for any $\varepsilon > 0$ runs in time polynomial in n and $\frac{1}{\varepsilon}$ and returns a solution whose value is $(1 + \varepsilon) \cdot OPT + c$, where $c \in \mathcal{O}(1)$. We start with an auxiliary property.

> **Property 9.1** The number of non-negative integer solutions of the inequality $x_1 + \ldots + x_k \le n$ is $\mathcal{O}(n^k)$

Proof. Denote by $S(n)$ the number of non-negative integer solutions of the inequality $x_1 + \ldots +$

$x_k \leq n$. Then, $S(n)$ equals to $S(n-1)$ plus the number of the non-negative integer solutions of the equality $x_1 + \ldots + x_k = n$. Hence, as a first step we need to compute the number of the non-negative integer solutions of the equality $x_1 + \ldots + x_k = n$. This problem is known as "stars and bars", because one can represent n by a sequence of n objects, in this case stars, as follows:

$$\star \ \star \ \star \ \star \ \ldots \ \star \ \star \ \star \ \star$$

Then assuming that we have chosen, out of the k variables, b that will be non-zero (which can be done in $\binom{k}{b}$ different ways), the number of non-negative integer solutions of the equality $x_1 + \ldots + x_k = n$ equals to the number of possible ways of putting $b-1$ bars in the sequence of n stars:

$$\star \ \star \ | \ \star \ | \ \star \ \ldots \ \star \ | \ \star \ \star \ \star$$

In other words, the number of stars between the $(i-1)$-th bar and the i-th bar correspond to the value of i-th variable out of the chosen b ones. Since there are $n-1$ gaps between the stars, the number of possible ways of putting $b-1$ bars in the sequence of n stars is:

$$\binom{n-1}{b-1}$$

Thus, in total that the number of non-negative integer solutions of the equality $x_1 + \ldots + x_k = n$ is:

$$\sum_{b=0}^{k} \binom{k}{b} \cdot \binom{n-1}{b-1}$$

This further implies that the number $S(n)$ of non-negative integer solutions of the inequality $x_1 + \ldots + x_k \leq n$ is given by the following recursive formula:

$$S(n) = S(n-1) + \sum_{b=0}^{k} \binom{k}{b} \cdot \binom{n-1}{b-1}$$

which can be solved as follows:

$$S(n) \quad = \quad \sum_{v=1}^{n} \sum_{b=0}^{k} \binom{k}{b} \cdot \binom{v-1}{b-1} \quad = \quad \sum_{b=0}^{k} \sum_{v=1}^{n} \binom{k}{b} \cdot \binom{v-1}{b-1} \quad = \quad \sum_{b=0}^{k} \binom{k}{b} \cdot \left(\sum_{v=1}^{n} \binom{v-1}{b-1} \right) \quad = \quad \sum_{b=0}^{k} \binom{k}{b} \cdot \binom{n}{b}$$

The latter is clearly in $\mathcal{O}(n^k)$, since the highest order term in the sum above is $\binom{n}{k}$ which is in $\mathcal{O}(n^k)$. Note that in the last equality we used the following property of binomial coefficients, which can be easily proved by induction on n:

$$\sum_{v=1}^{n} \binom{v-1}{b-1} = \binom{n}{b} \qquad \qquad \square$$

Lemma 9.1 There is a polynomial time algorithm for instances of the bin packing problem, in which:

- there exist at most K different item-sizes,

- at most L items can fit in a bin, and

> - K and L are constants, i.e., $K, L \in \mathcal{O}(1)$.

Proof. We start by asking how many different types of bins exist, under the assumptions of the lemma. Each bin-type is described in terms of the items it contains (which are at most L) and then for each item one has also to account its size (which are at most K). In other words, each bin-type is specified by the size of each of the items in it. Therefore, there exist at most K^L different bin-types.

We are now asking how many different solutions an instance of the bin packing problem can have, under the assumptions of the lemma. If they are bounded by a polynomial in n, then a brute-force algorithm can simply list all of them and keep the optimum. To answer the initial question, let x_i be the number of bins of type i in a solution, where $0 < i \leq K^L$. Since the number of used bins is upper bounded by n, it follows that the number of different solutions of an instance of the bin packing problem under the assumptions of the lemma equals to the non-negative integral solutions to the equation:

$$x_1 + x_2 + \ldots + x_{K^L} \leq n$$

which by Property 9.1 are:

$$\mathcal{O}(n^{K^L})$$

Since $K, L \in \mathcal{O}(1)$, the number of different solutions of an instance of the bin packing problem under the assumptions of the lemma is indeed at most a polynomial in n. Hence, a brute force algorithm is a trivial polynomial time algorithm for the special instances of the bin packing problem of the lemma.   □

> **Theorem 9.8**   For any $\varepsilon > 0$, there is a polytime approximation algorithm for the bin packing problem, that uses at most $(1 + \varepsilon) \cdot OPT + 1$ bins.

Proof. Consider an instance $I = \{a_1, a_2, \ldots, a_n\}$ of bin packing, and partition I into I_L (i.e., large items) and I_S (i.e., small items), such that:

- $I_L = \{a \in L : s(a) \geq \varepsilon\}$

- $I_S = \{a \in L : s(a) < \varepsilon\}$

If we consider only the large items in I_L, the number of items in each bin is bounded by $\frac{1}{\varepsilon}$ (i.e., a constant). Hence, if the number k of different sizes is also constant, we can solve the problem in poly-time by enumerating all the possible assignments for all the bins with a brute-force algorithm (there are more efficient ways, but this is already polynomial).

Given an instance I_L of large items, we create a new one I_L' in which the number of different sizes is constant: Let k be a parameter that we will specify soon. We first group the items into
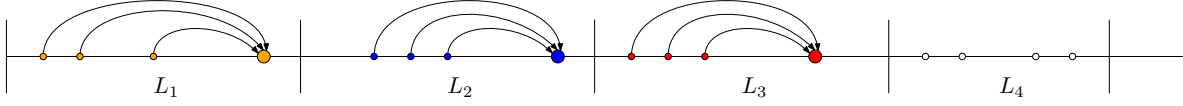
Figure 9.8: Sort and split

k different groups $L_1,\dots,L_k$, each of size at most $\frac{|L|}{k}$, so that all items in L_i are smaller than all items in L_{i+1} (i.e., we first sort them by their size and then we split them into k groups; see Fig. 9.8). For each $i = 1,\dots,k-1$, we round the size of each item in L_i to the one of the largest item in L_i (note that we exclude L_k here). This implies that if $x \in L_i$, $y \in L_j$ and $i < j$, then $x < y$. By Lemma 9.1, we can find an optimal solution $OPT(I'_L)$ of I'_L in time polynomial in k and $\lceil 1/\varepsilon \rceil$.

Next, we find a solution $ALG(I_L)$ for I_L by using at most $OPT(I'_L) + \frac{|L|}{k}$ bins. Namely, we use the bins that are hosting the items of L_i in $OPT(I'_L)$ to fit the items of L_i, for $i = 1,\dots,k-1$, and then use at most $\frac{|L|}{k}$ additional bins to fit the (at most $\frac{|L|}{k}$) items of L_k. Note that $OPT(I'_L) \le OPT(I_L)$. In fact, given a solution for I_L, we can use the bins that are hosting the items of L_i to fit the items of L_{i-1} (recall that L_k does not exist in I'_L). By setting $k = 1/\varepsilon^2$, we obtain:

$ALG(I_L) = OPT(I'_L) + |I_L|/k$

$\qquad \le OPT(I_L) + \varepsilon^2 |I_L| \qquad\qquad$ since $OPT(I'_L) \le OPT(I_L)$ and $k = 1/\varepsilon^2$

$\qquad \le OPT(I_L) + \varepsilon OPT(I_L) \qquad$ since $\varepsilon \cdot |I_L| \le OPT(I_L)$

$\qquad = (1 + \varepsilon) \cdot OPT(I_L) \qquad\qquad$ note that this is a PTAS for the large items

To complete the algorithm, we now put the small items back, by filling the gaps with a greedy approach. Let $ALG(I)$ be the number of used bins. If the small items fit in the gaps, then we have:

$$ALG(I) = ALG(I_L) = \le (1 + \varepsilon) \cdot OPT(I_L) = \le (1 + \varepsilon) \cdot OPT(I).$$

Otherwise, all bins (except one, i.e., the last one) are filled with an amount of at least $(1 - \varepsilon)$. Thus, we obtain:

$$(ALG(I) - 1)(1 - \varepsilon) < \sum_{i=1}^{n} s(a_i)$$

By Observation 9.7, we obtain:

$$(ALG(I) - 1)(1 - \varepsilon) \le OPT(I),$$

which we can rewrite as follows:

$$ALG(I) \le OPT(I)/(1 - \varepsilon) + 1$$

To complete the proof, we assume that $\varepsilon \le 1/2$, which implies that $1/(1 - \varepsilon) \le (1 + 2\varepsilon)$. Hence:

$$ALG(I) \le (1 + 2\varepsilon) \cdot OPT(I) + 1$$

This completes the description and the analysis of the algorithm. $\qquad\qquad\qquad\qquad\qquad\square$

Exercises

Exercise 9.1 The input of the weighted vertex cover problem is a connected, undirected, vertex-weighted graph $G = (V, E, w)$, where $w : V \to \mathbb{R}^+$. In the output, we seek for a set $S \subseteq V$ of vertices that is of minimum total weight, such that for each edge (u, v) in E, $u \in S$ or $v \in S$ holds.

Present what modifications are needed in the second algorithm supporting Theorem 9.1 in order to derive a corresponding 2-approximation algorithm for the weighted vertex cover problem.

Exercise 9.2 The input of the subset-sum problem is a set $A = \{a_1, a_2, \ldots, a_n\}$ of positive natural numbers and an integer $B \in \mathbb{N}$. We call a subset S of A admissible if and only if the sum of the numbers in S, denoted by $\mathrm{SUM}(S)$, does not exceed B, that is,

$$\mathrm{SUM}(S) := \sum_{a_i \in S} a_i \leq B.$$

In the output of the subset-sum problem, we seek to compute an admissible subset S such that $\mathrm{SUM}(S)$ is maximum. For example, for $A = \{8, 2, 4\}$ and $B = 11$, the optimum would be $S = \{8, 2\}$. Consider the following algorithm:

Algorithm 9.4: A greedy algorithm for the subset-sub problem.

---

1 $S \leftarrow \emptyset$;
2 $T \leftarrow 0$;
3 for $i = 1, \ldots, n$ do
4 $\quad$ if $T + a_i \leq B$ then
5 $\quad \quad$ $S \longleftarrow S \cup \{a_i\}$;
6 $\quad \quad$ $T \longleftarrow T + a_i$;
7 $\quad$ end
8 end
9 return S

(a) Let $S' \subseteq A$ be an admissible set. Give an example where the value S returned by the algorithm above is such that $\mathrm{SUM}(S) < \frac{1}{2}\mathrm{SUM}(S')$.

(b) Describe an approximation algorithm for this problem with the following properties:

- The time complexity is $\mathcal{O}(n \log n)$;

- The algorithm returns an admissible set $S \subseteq A$ such that $\mathrm{SUM}(S) \geq \frac{1}{2}\,\mathrm{SUM}(S^*)$, where $S^* \subset A$ is the optimal solution for the problem.

Discuss the correctness and the complexity of your algorithm.

Exercise 9.3 Consider the following heuristic for solving bin packing:

Algorithm 9.5: First-fit heuristic for bin packing.

---
1 Start with a sequence $B_1, B_2, \ldots, B_n$ of empty bins of unit size;
2 for $i = 1$ to n do
3 | Place item a_i into the bin of smallest index in which it can fit;
4 end
5 return the packing of items into the non-empty bins

(a) Give an example of an instance of the problem, and an execution of the first-fit heuristic on this instance, where the packing returned by the heuristic does not use the minimum possible number of bins.

(b) Prove that in any execution of the first-fit heuristic, on any instance, the number of bins used in the packing returned by the heuristic is at most twice the minimum possible number of bins.

Exercise 9.4 Another heuristic for solving bin packing is the following:

Algorithm 9.6: A merging heuristic for bin packing.

---
1 Start with each item in a different bin;
2 while there exist two bins whose contents have total size ≤ 1 do
3 | Empty the contents of both bins;
4 | Place all these items in a single bin;
5 end
6 return the current packing of items in bins

Of course, the merging heuristic sometimes has the freedom to choose among several pairs of bins to merge. Thus, on a given instance there might exist different possible executions of the heuristic.

(a) Prove that the merging heuristic always terminates in time polynomial to the size of the input.

(b) Give an example of an instance of the problem, and an execution of the merging heuristic on this instance, where the packing returned by the heuristic does not use the minimum possible number of bins.

(c) Prove that in any execution of the merging heuristic, on any instance, the number of bins used in the packing returned by the heuristic is at most twice the minimum possible number of bins.

Exercise 9.5 Let $G = (V, E)$ be a $\sqrt{n} \times \sqrt{n}$ grid graph (see Section 7.6 for the definition), such that each vertex v of G is associated with a non-negative integer weight $w(v)$. You may assume that the weights of all vertices are distinct. The goal is to compute a maximum-weight independent set of G, that is, a set $S \subseteq V$ such that no two vertices in

DRAFT

S are adjacent in G and whose total total weight is as large as possible. Consider the following greedy algorithm for this problem:

Algorithm 9.7: The heaviest-first greedy algorithm.

---
1 $S \leftarrow \emptyset$;
2 while some vertex remains in G do
3 | $v \leftarrow$ the node of maximum weight;
4 | Add v to S;
5 | Delete v and its neighbors from G;
6 end
7 return S

(a) Let S be the indepedent set returned by the "heaviest-first" greedy algorithm, and let T be any other independent set in G. Show that for each node $v \in T$, either $v \in S$, or there is a node $v' \in S$ so that $w(v) \le w(v')$ and (v, v') is an edge of G.

(b) Show that the "heaviest-first" greedy algorithm returns an independent set of total weight at least $1/4$ times the maximum total weight of any independent set in the grid graph G.

Further reading

– V. V. Vazirani. Approximation algorithms. Springer, 2001.

10. Randomized Algorithms

This chapter will focus on introducing randomized algorithms. In particular, we will formally introduce randomized algorithms, explain why they are useful (in Section 10.1) as well as showcase and analyze few randomized algorithms. Before doing so, we will recall some basic definitions and properties of probabilities and expected values (in Section 10.2). We will also discuss some properties used in the analysis of randomized algorithms (including the so-called linearity of expectations and the Markov inequality). In Section 10.3 we study a very simple randomized algorithm for the problem of verifying polynomial identities, according to which, given two polynomials, the task is to determine whether these two polynomials are equivalent. In Section 10.4, the focus will be on a technique, called random walk, to explore one dimensional solution spaces using randomization. We will apply this technique to derive a randomized algorithm for the well-known 2SAT problem in Section 10.5. Section 10.6 mainly focuses on the correctness and time complexity analysis (using a method called backward analysis) of a randomized incremental algorithm to solve the convex hull problem introduced in Chapter 1.

10.1   Randomized algorithms

A randomized algorithm is an algorithm that makes random choices in its execution. As a result, different executions of such an algorithm on the same input may result in different outputs (which is not the case for the algorithms that we have encountered so far).

In this regard, the following question naturally arises: why a randomized algorithm becomes useful, especially when a corresponding efficient deterministic algorithm for solving the same problem exists? We discuss three replies to this question. The first one is that randomized algorithms are usually quite simple and easy to perceive or implement. On the contrary, a corresponding efficient deterministic algorithm solving the same problem may be considerably more complex. The second one is that randomized algorithms are, in most cases, more efficient than the corresponding deterministic ones for the same problem. And finally sometimes there is no deterministic algorithm for solving the problem.

There are two main categories of randomized algorithms, namely Monte-Carlo and Las Vegas,

which differ in two aspects: running time and correctness. More precisely:

> **Definition 10.1** (Monte-Carlo) A Monte-Carlo algorithm is a polynomial-time randomized algorithm whose output may be incorrect with a certain (typically small) probability.

> **Definition 10.2** (Las Vegas) A Las Vegas algorithm is a randomized algorithm whose output is always correct but its running time differs depending on the input; typically, however, the expected running time of a Las Vegas algorithm is finite.

> **Example 10.1** (The unweighted minimum-cut problem) In Exercise 4.3 of Chapter 4, we already came across a simple randomized algorithm to computed a minimum cut of a graph G, under the assumption that the edges of G are unweighted (or equivalently, that they all have the same weight, e.g., unit). What is the type of this algorithm?

Answer. The algorithm is rather easy, as it is based on edge contractions. More precisely, for each vertex v of graph G, denote by $S(v)$ the vertices that are contracted into v. Initially $S(v) = \{v\}$ for each vertex v of G. If G has only two vertices v_1 and v_2, then the algorithm returns the cut $(S(v_1), S(v_2))$. Otherwise, the algorithm chooses an edge (u, v) at random and computes the graph G' resulting from the contraction of (u, v) with a new node z_{uv} replacing u and v. The algorithm then sets $S(z_{uv})$ to $S(u) \cup S(v)$ and is recursively applied to G'. The exercise gives the main steps to prove that this simple algorithm returns a minimum cut with probability $\frac{2}{n(n-1)}$ in polynomial time. Thus the type of this algorithm is Monte-Carlo. □

10.2   Basics

Before introducing and analyzing our first randomized algorithm, we recall some elementary properties of probabilities and expected values, which are useful in analyzing the behavior of randomized algorithms. We start with the definition of probability.

> **Definition 10.3**   Given a finite set Ω of events, each event $\omega \in \Omega$ is associated with a probability $Pr(w)$ such that:
> $$0 \le Pr(w) \le 1 \quad \text{and} \quad \sum_{\omega \in \Omega} Pr(w) = 1$$

> **Example 10.2**   Let $\Omega = \{1, \ldots, 6\}$. What is the probability of event $w = \{2, 4, 6\}$?

Answer. $Pr(\omega) = Pr(\{2, 4, 6\}) = 3 \cdot \frac{1}{6} = \frac{1}{2}$ □

DRAFT

In the following, we recall two important properties of probabilities. The first is commonly referred to as conditional probability and provides a closed formula for the computation of the probability of the intersection of two events. The latter is known as inclusion-exclusion principle and provides a closed formula for the computation of the probability of the union of two events.

Property 10.1 (Conditional probability) For two events ω_1 and ω_2 in Ω, it holds that $Pr(\omega_1 \cap \omega_2) = Pr(\omega_1) \cdot Pr(\omega_2|\omega_1)$.

Property 10.2 (Inclusion-exclusion principle) For two events ω_1 and ω_2 in Ω, it holds that $Pr(\omega_1 \cup \omega_2) = Pr(\omega_1) + Pr(\omega_2) - Pr(\omega_1 \cap \omega_2)$.

Note that both properties generalize to more than two events as follows.

Property 10.3 For n events $\omega_1, \omega_2, \ldots, \omega_n$ in Ω, it holds that $Pr(\cap_{i=1}^n \omega_i) = Pr(\omega_1) \cdot Pr(\omega_2|\omega_1) \cdots Pr(\omega_n|\omega_1 \cap \ldots \cap \omega_{n-1})$.

Property 10.4 For n events $\omega_1, \omega_2, \ldots, \omega_n$ in Ω, it holds that $Pr(\cup_{i=1}^n \omega_i) = \sum_{i=1}^n Pr(\omega_i) - \sum_{1 \le i \le j \le n} Pr(\omega_i \cap \omega_j) + \sum_{1 \le i \le j \le k \le n} Pr(\omega_i \cap \omega_j \cap \omega_k) - \cdots + (-1)^{n-1} \cdot Pr(\cap_{i=1}^n \omega_i)$.

Recall that a random variable is defined as a variable whose values depend on outcomes of a random phenomenon. In that context, a random variable forms a measurable function defined on a probability space that maps from the sample space to the real numbers. We mainly focus on discrete random variables, which have a countable number of possible values, and binary random variables which can take only two values, 0 or 1.

Definition 10.4 (Expected value) Given a discrete random variable X, the expected value of X, i.e., the value that we expect on average, is

$$E[X] = \sum_{j=0}^{\infty} j \cdot Pr(X = j), \tag{10.1}$$

An immediate consequence of Definition 10.4 is the following property, commonly known as 0-1 property.

Property 10.5 (0-1 property) If X is a binary random variable, then:

$$E[X] = Pr(X = 1)$$

Example 10.3   What is the expected value of a dice, after one rolls it once?

Answer. Let $X \in \{1,\dots,6\}$ be the random variable expressing the amount of the rolled dice. Then, the probability that X equals j is:

$$Pr(X = j) = \frac{1}{6}, \forall j = 1,\dots,6 \quad \text{and} \quad Pr(X = j) = 0, \forall j > 6$$

Therefore, the expected value of X is:

$$E[X] = \sum_{j=0}^{\infty} j \cdot Pr(X = j) = \sum_{j=1}^{6} j \cdot Pr(X = j) = \frac{1}{6}.(1 + \dots + 6) = \frac{21}{6} = 3.5 \qquad \square$$

Example 10.4   Suppose that we toss a fair coin until a head comes. What is the expected number of tosses to be made?

Answer. Let $X \in \{0,1,\dots\}$ be a random variable expressing the number of tosses until a head comes. Let $Pr(H_j)$ and $Pr(T_j)$ denote the probabilities of getting a head or a tail on the j-th toss, respectively, which is always $\frac{1}{2}$ for $j > 0$. Then, the probability of getting a head on the j-th toss for $j > 0$

$$Pr(X = j) = Pr(T_1) \cdots Pr(T_{j-1}) \cdot Pr(H_j)$$
$$= \frac{1}{2} \cdots \frac{1}{2} \cdot \frac{1}{2}$$
$$= \left(\frac{1}{2}\right)^j$$

Therefore by definition of the expected value of X

$$E[X] = \sum_{j=0}^{\infty} j \cdot Pr(X = j) = \sum_{j=1}^{\infty} j \cdot \left(\frac{1}{2}\right)^j$$

To compute $E[X]$, we can either give a closed formula for the sum above, as follows:

$$E[X] = 2E[X] - E[X] = (1 + \frac{2}{2^1} + \frac{3}{2^2} + \frac{4}{2^3} + \frac{5}{2^4} \dots) \quad - \quad (\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} \dots)$$
$$= 1 + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \dots = \frac{1}{1 - \frac{1}{2}} = 2.$$

or we can make the following observation:

$$E[X] = 1 + \frac{1}{2}E[X]$$

which directly implies $E[X] = 2$. The 1 comes from the fact that we have to flip the coin at least once to get a head. The second part of the formula is due to the fact that there is a $\frac{1}{2}$ chance that we get a tail in which case we will have to flip the coin again and thus recursively proceed. In other words, we flip the coin once (hence the 1), and there is a 1/2 chance that we get tail and have to continue flipping the coin (hence the $+\frac{1}{2}E[X]$). $\qquad \square$

In the following, we recall two important properties, namely the linearity of expectations and the inequality by Markov, which we are going to use in the analysis of randomized algorithms.

Property 10.6 (Linearity of expectations) Given two discrete random variable X and Y, the following holds:

$$E[X + Y] = E[X] + E[Y]$$

Proof. The property follows by Definition 10.4 as follows:

$$
\begin{aligned}
E[X + Y] &= \sum_{x=0}^{\infty} \sum_{y=0}^{\infty} (x + y) Pr(X = x, Y = y) \\
&= \sum_{x=0}^{\infty} \sum_{y=0}^{\infty} x \cdot Pr(X = x, Y = y) + \sum_{y=0}^{\infty} \sum_{x=0}^{\infty} y \cdot Pr(X = x, Y = y) \\
&= \sum_{x=0}^{\infty} x \cdot Pr(X = x) + \sum_{y=0}^{\infty} y \cdot Pr(Y = y) \\
&= E[X] + E[Y]
\end{aligned}
$$
$\square$

Property 10.7 (Markov inequality) For a discrete random variable X and a value $a > 0$, the following holds:

$$Pr(X \geq a) \leq \frac{E[X]}{a}$$

Proof. The property follows by Definition 10.4 as follows:

$$
\begin{aligned}
E[X] &= \sum_{j=0}^{\infty} j \cdot Pr(X = j) \\
&= \sum_{j<a}^{\infty} j \cdot Pr(X = j) + \sum_{j\geq a}^{\infty} j \cdot Pr(X = j) \\
&\geq \sum_{j\geq a}^{\infty} j \cdot Pr(X = j) \\
&\geq a \sum_{j\geq a}^{\infty} Pr(X = j) \\
&= a \cdot Pr(X \geq a)
\end{aligned}
$$
$\square$

10.3 Verifying polynomial identities

In this section, we present our first randomized algorithm for a relatively simple problem, namely, the one of verifying polynomial identities. The input of this problem is two polynomials $P(x)$ and $Q(x)$. In the output we seek to determine whether P and Q are equivalent (that is, whether $P(x) = Q(x), \forall x \in \mathbb{R}$).

It is rather clear that this problem can be solved in polynomial time, e.g., by bringing P and Q in a standard form, such as sum of monomials. This observation implies that one can solve the problem of verifying polynomial identities in time $\mathcal{O}(d^2)$, where d is the maximum degree of the input polynomials P and Q.

> **Example 10.5** Determine whether the following two polynomials are equivalent.
>
> $$P(x) = (x+1)(x-2)(x+3)(x-4)(x+5)(x-6)$$
> $$Q(x) = x^6 - 7x^3 + 25$$

Answer. Polynomials P and Q are not equivalent, because if one brings P in standard form, then its constant term is $1 \cdot (-2) \cdot 3 \cdot (-4) \cdot 5 \cdot (-6)$, which is different from the constant term of Q, which is 25. □

To solve the problem of verifying polynomial identities with a randomized algorithm, we make some further observations. First, we observe that if there exists a value $x_0 \in \mathbb{R}$ for which $P(x_0)$ and $Q(x_0)$ are not equal, then it directly follows that the two polynomials P and Q are not equivalent. On the other hand, if there exists a value $x_0 \in \mathbb{R}$ for which $P(x_0) = Q(x_0)$ holds, then this does not necessarily imply that P and Q are equivalent, because it can simply mean that the value x_0 is a root of the polynomial $P - Q$ expressing the difference between P and Q. In particular, P and Q are equivalent if and only if the polynomial $P - Q$ is equivalent to the zero polynomial. We summarize these observations in the following.

> **Observation 10.1** For two polynomials P and Q, the following hold:
>
> (i) If $\exists x_0 \in \mathbb{R}$ such that $P(x_0) \neq Q(x_0) \implies P \not\equiv Q$
>
> (ii) If $\exists x_0 \in \mathbb{R}$ such that $P(x_0) = Q(x_0) \not\implies P \equiv Q$
>
> (iii) $P \equiv Q \iff F = P - Q \equiv 0$, i.e., $P(x) - Q(x) = 0, \forall x \in \mathbb{R}$.

With Observation 10.1 in mind, we are now ready to give a randomized algorithm for the problem of verifying polynomial identities; refer to Algorithm 10.1.

It is rather clear that the complexity of Algorithm 10.1 is $\mathcal{O}(d)$, since the algorithm computes the value for $F(r)$ and polynomial F is of degree d. However, by Observation 10.1.i, the algorithm may report a wrong answer. However, since F has at most d roots, the probability

---

Algorithm 10.1: A randomized algorithm to solve the problem of verifying polynomial identities.

Input   : Two polynomials $P(x)$ and $Q(x)$.
Output : A boolean value indicating whether $P \equiv Q$.

1 $d \leftarrow$ the degree of $F = P - Q$;
2 $r \leftarrow$ a random integer from $[1, 2, \ldots, 100d]$;
3 if $F(r) = 0$ then
4 | report $P \equiv Q$
5 end
6 else
7 | report $P \not\equiv Q$
8 end

---

of reporting a wrong answer is at most:

$$\frac{d}{100d} = 0.01$$

In the following, we are interested in further decreasing the probability of Algorithm 10.1 to report a wrong answer (but without changing the interval $[1, 2, \ldots, 100d]$). The most natural way to achieve this, is by running Algorithm 10.1 k times. In particular, if at each run the algorithm reports $P \equiv Q$, then we conclude $P \equiv Q$. Otherwise, there is at least one run in which the algorithm reported $P \not\equiv Q$, and thus we conclude $P \not\equiv Q$. Hence, the probability to report a wrong answer is at most:

$$\left(\frac{d}{100d}\right)^k = 0.01^k$$

Note that the same number can be chosen multiple times at different iterations of the algorithm. Hence, it is natural to ask whether the probability can be further improved, if the algorithm is modified such that any integer from the interval $[1, 2, \ldots, 100d]$ can be chosen at most once during the k runs of the algorithm. The intuition is that if we apply the algorithm $d+1$ times (that is, $k = d+1$), then under this assumption we can decide whether $P \equiv Q$ (since $P - Q$ has at most d roots). However, for reasons of efficiency, we prefer to keep k less than $d + 1$.

Let A_i be the event that the algorithm reports a wrong answer after the i-th run. Then, after k runs of the algorithm, a wrong answer is reported with probability:

$$Pr(\cap_{i=1}^{k} A_i) = Pr(A_1) \cdot Pr(A_2|A_1) \cdots Pr(A_k|A_1 \cap \cdots \cap A_{k-1})$$
$$= \frac{d}{100d} \cdot \frac{d-1}{100d-1} \cdot \frac{d-2}{100d-2} \cdots \frac{d-(k-1)}{100d-(k-1)}$$

It follows that the complexity of the modified algorithm is $\mathcal{O}(kd)$, because the algorithm will run at most k times and each time computing $F(r)$ needs $\mathcal{O}(d)$ time. We conclude by mentioning that the algorithm is Monte-Carlo, since it has a polynomial time complexity with a low probability of giving a wrong result.

10.4   Random walks

A random walk describes a path that consists of a sequence of random steps on some space such as the two dimensional space of the integer grid points of the Euclidean plane $\mathbb{R}^2$. In the following, we focus on the one dimensional case.

To make the setting more evident, consider a person P on a line (e.g., the x-axis); see Fig. 10.1. At some point of time, person P is at position i. If the person has to move, either P will move to position $i - 1$ (i.e., to the left) with probability p or to position $i + 1$ (i.e., to the right) with probability $1 - p$. To simplify the presentation, we assume $p = \frac{1}{2}$, i.e., P moves to the left and to the right with probability $\frac{1}{2}$. We further assume that P is initially at position 0 (and cannot move to -1 or more in general to negative). The task is to compute the expected number of moves that person P has to make in order to reach position n.
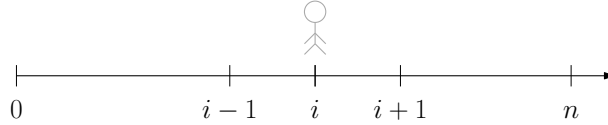
Figure 10.1: Illustration of random walk.

To solve this task, we introduce a discrete random variable X_i, which corresponds to the number of moves that person P has to make in order to reach position i. Of course, we are interested in computing the expected value of X_n. To simplify the presentation, in the following we set $T[i]$ to be equal to $E[X_i]$.

Since P is initially at position 0 and cannot move to negative, it follows:

$$T[0] = 0 \tag{10.2}$$

To reach position 1, either P is at position 0 and performs a move to the right with probability $\frac{1}{2}$ or P is at position 2 and performs a move to the left with probability $\frac{1}{2}$. Hence:

$$T[1] = 1 + \frac{1}{2}T[0] + \frac{1}{2}T[2] \tag{10.3}$$

Symmetrically, we argue for position i as follows:

$$T[i] = 1 + \frac{1}{2}T[i - 1] + \frac{1}{2}T[i + 1] \tag{10.4}$$

We finally observe that the only way for P to reach position n is to be at position $n - 1$ and to move to the right. Hence:

$$T[n] = 1 + T[n - 1] \tag{10.5}$$

Backward substitution: To solve Eqs. (10.2) and (10.5), we first substitute backwards as follows:

$$
\begin{aligned}
T[n - 1] &= 1 + \frac{1}{2}T[n - 2] + \frac{1}{2}T[n] \\
&= 1 + \frac{1}{2}T[n - 2] + \frac{1}{2}(1 + T[n - 1]) \\
&= 3 + T[n - 2]
\end{aligned}
$$

Similarly:

$$T[n-2] = 1 + \frac{1}{2}T[n-3] + \frac{1}{2}T[n-1]$$
$$= 1 + \frac{1}{2}T[n-3] + \frac{1}{2}(3 + T[n-2])$$
$$= 5 + T[n-3]$$

Therefore, we may conclude:

$$T[n-i] = 2i + 1 + T[n-i-1]$$

which, since $T[0] = 0$ by Eq. (10.2), implies that:

$$T[1] = 2(n-1) + 1$$

Forward substitution: Since we have a closed formula for $T[1]$, we proceed by substituting forward as follows:

$$T[2] = 1 + 2(n-2) + T[1]$$
$$= 1 + 2(n-2) + 2(n-1) + 1$$

Similarly:

$$T[3] = 1 + 2(n-3) + T[2]$$
$$= 1 + 2(n-3) + 1 + 2(n-2) + 2(n-1) + 1$$

Therefore, we can conclude that $T[n]$ is as follows:

$$T[n] = 1 + 2(n-n) + \ldots + 1 + 2(n-2) + 2(n-1) + 1$$
$$= n + 2 \cdot (1 + 2 + \ldots + (n-2) + (n-1))$$
$$= n + n(n-1)$$
$$= n^2$$

Since by definition $T[n]$ corresponds to the expected value of variable X_n, we may conclude that:

$$E[X_n] = n^2$$

In other words, the expected number of steps that person P has to make in order to reach position n from position 0 is n^2.

10.5   The 2SAT problem

In this section, we study a randomized algorithm for the well-known 2-satisfiability (or 2SAT, for short) problem, which asks for an assignment of the truth values (i.e., T (true) or F (false)) to the variables of a Boolean formula consisting of clauses with two literals. Note that in contrast to the well-known $\mathcal{NP}$-complete problem 3SAT, the 2SAT problem is polynomially time solvable (as we will also shortly see). The 2SAT problem is formally defined as follows.

Definition 10.5 (2SAT problem) The input of the 2SAT problem consists of a Boolean formula ϕ with n variables and m clauses, each with two literals. In the output, we seek to compute an assignment of truth values to the variables of ϕ that makes it satisfiable (if any).

Example 10.6 Determine whether the following formula is satisfiable.

$$\phi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2) \wedge (\neg x_3 \vee \neg x_4)$$

Answer. Formula ϕ is indeed satisfiable, since the following truth assignment clearly makes it satisfiable.
$$x_1 = x_2 = \text{true}, \quad x_3 = x_4 = \text{false} \qquad \square$$

10.5.1   A randomized algorithm

The randomized algorithm to solve the 2SAT problem is rather simple (refer to Algorithm 10.2). It starts with an arbitrary assignment of truth values to the variables of the input formula ϕ. Then, as long as there is an unsatisfied clause in ϕ, the algorithm is choosing such a clause and it changes the value of one of its literals uniformly at random.

Algorithm 10.2: A randomized algorithm for the 2SAT problem

Input   : A Boolean 2SAT formula ϕ with n variables $x_1, \ldots, x_n$ and an upper bound on the maximum number of iterations.
Output : A true assignment for the variable of ϕ.

1 for $i = 1$ to n do
2 | $x_i \leftarrow false$
3 end
4 while there is an unsatisfied clause and the max. no. of iterations is not exceeded do
5 | c $\leftarrow$ some clause that is unsatisfied in ϕ;
6 | choose a variable of c at random by flipping a coin;
7 | change the value of this variable;
8 end

To analyse Algorithm 10.2, consider an assignment S of truth values to the variables of ϕ; note that we do not assume that S makes ϕ satisfiable. We denote by A_j the assignment of truth values to the variables of ϕ after the first j-th iteration of Algorithm 10.2, and by X_j the number of variables with same value in S and A_j. Since ϕ consists of n variables, if for some j it holds that $X_j = n$, then Algorithm 10.2 has found S.

To understand the meaning of the aforementioned variables consider the following truth assignment for Example 10.6

$$S = \{x_1 = x_2 = \text{true}, x_3 = x_4 = \text{false}\}$$

DRAFT

Then, we know that $A_0 = \{x_1 = x_2 = x_3 = x_4 = \text{false}\}$, which implies that $X_0 = 2$. We are not ready to analyse Algorithm 10.2.

> **Lemma 10.1** If ϕ admits a satisfying truth assignment S, then Algorithm 10.2 is expected to find S after at most n^2 iterations

Proof. Assume without loss of generality that $X_0 = 0$, that is, the number of variables with the same value in S and in the initial truth assignment is zero. Note that this assumption is indeed without loss of generality, since if $X_0 > 0$, then Algorithm 10.2 is expected to find S in fewer iterations. Now, recall that if for some j it holds that $X_j = n$, then Algorithm 10.2 has found S. Hence, we can simulate Algorithm 10.2 by a random walk, where the objective is to reach the state in which the value of X_j becomes n for some value of j, while initially $X_0 = 0$ holds, i.e., the person is initially at position 0.
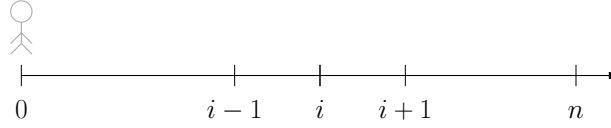
Figure 10.2: Illustration of a random walk for the 2SAT problem.

We are now asking what is the probability of moving closer or away from S at iteration j? These two probabilities are clearly defined as follows.

$$Pr(X_{j+1} = k + 1 \mid X_j = k)$$

$$Pr(X_{j+1} = k - 1 \mid X_j = k)$$

To give an estimation for each of them, let $(x_k \vee x_\ell)$ be the clause of ϕ chosen at iteration j. Then, x_k or x_ℓ or both have different values in A_j and S, as otherwise the clause would not be unsatisfied.

- If both x_k and x_ℓ have the same value, then $Pr(X_{j+1} = k + 1 \mid X_j = k) = 1$ and $Pr(X_{j+1} = k - 1 \mid X_j = k) = 0$.

- Otherwise, $Pr(X_{j+1} = k + 1 \mid X_k = k) = Pr(X_{j+1} = k - 1 \mid X_j = k) = \frac{1}{2}$.

In both cases the probability of moving closer to S is at least $\frac{1}{2}$, while the probability of moving away from S is at most one half. By random walk theory, the expected number of iterations of Algorithm 10.2 to find S is at most n^2. $\square$

In the following theorem, we present an upper bound on the probability that Algorithm 10.2 does not find a truth assignment making the input formula satisfiable (if one exists), in the case where $2n^2$ iterations are performed.

> **Lemma   10.2**   If ϕ admits a satisfying truth assignment, then the probability that
> Algorithm 10.2 does not find a truth assignment making the input formula satisfied
> after $2n^2$ iterations is at most $\frac{1}{2}$.

Proof. Let X be a discrete random variable corresponding to the number of iterations that Algorithm 10.2 needs to perform to compute a truth assignment making the input formula ϕ satisfiable. It follows that the probability that we seek to upper bound equals to $Pr(X \geq 2n^2)$, which by Markov's inequality is at most $\frac{E[X]}{2n^2}$, and thus by the previous lemma at most $\frac{1}{2}$.   □

10.5.2   A deterministic polynomial-time algorithm

We conclude with a polynomial-time (deterministic) algorithm, which provides a certificate that 2SAT is indeed in P, as we claimed earlier in this chapter. In particular, given an instance ϕ of the 2SAT problem with n variables and m clauses we will construct in $\mathcal{O}(n+m)$ time an instance of a graph problem that asks whether specific paths exists in a directed graph, and we will prove that the two problems are equivalent. Since searching for paths in a graph is clearly in P and since the reduction can be done in $\mathcal{O}(n+m)$ time, this will be sufficient to prove that 2SAT is in P. More precisely, given a 2SAT formula ϕ, we create a graph, which we denote by G_ϕ, as follows:

– for each variable x of ϕ, we introduce two vertices in G_ϕ that we denote by x and $\neg x$,

– for each clause $(a \vee b)$ of ϕ, we introduce two directed edges $(\neg a, b)$ and $(\neg b, a)$ in G_ϕ.

The intuition under this construction is that $(x \vee y)$ is equivalent to $\neg x \Rightarrow y$ and to $\neg y \Rightarrow x$. Also, observe that the size of graph G_ϕ is twice the size of formula ϕ, which implies that symptomatically it is of the same size. Hence, graph G_ϕ can be constructed in time $\mathcal{O}(n+m)$.

> **Example   10.7**   The graph G_ϕ corresponding to formula $\phi = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (z \vee y)$
> is the one show in the figure below:
>
>
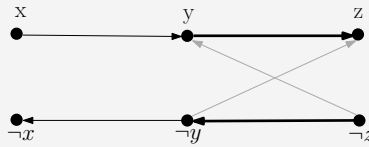
> **Observation   10.2**   If graph G_ϕ has a path from a to b, then it also has a path from $\neg b$
> to $\neg a$.

Proof. The observation follows from the fact that $(a, b) \in G_\phi$ implies that $(\neg b, \neg a) \in G_\phi$ (by the definition of the edges of graph G_ϕ).   □

(a) $x = $ true   (b) $x = $ false

Figure 10.3: Illustration for the proof of Theorem 10.1

We are now ready to prove the following theorem, which proves the equivalence between the problem of finding a truth assignment for ϕ and the problem of testing whether specific paths exists in G_ϕ.

> **Theorem 10.1** Formula ϕ is unsatisfiable if and only if there is a variable x of ϕ such that in G_ϕ there are two directed paths, one from x to $\neg x$ and one from $\neg x$ to x.

Proof. For the forward direction let ϕ be unsatisfiable and assume to the contrary that there exist no such paths, i.e., for each variable x of ϕ either the path from x to $\neg x$ or the path from $\neg x$ to x does not exist in G_ϕ. We then consider the following procedure, which we claim to be correct and to return a truth assignment for ϕ making it satisfiable (i.e., we obtain a contradiction).

> while not all variables of ϕ have been assigned do
> $a \leftarrow$ unassigned literal with no path from a to $\neg a$;
>   set a and all vertices reachable by a in G_ϕ to true;
>   set all vertices reachable by $\neg a$ in G_ϕ to false;
> end

The procedure is not correct if there are two paths, one from a to b and one from a to $\neg b$, for some pair of vertices a and b. By Observation 10.2, however, the second path implies that there is a path from $\neg b$ to $\neg a$, which together with the first path implies that there exists a path from a to $\neg a$; a contradiction to the fact that "a is an unassigned literal with no path from a to $\neg a$". Hence, the procedure is correct as claimed, contradicting the fact that ϕ is unsatisfiable.

For the other direction, assume that there is a variable x of ϕ such that in G_ϕ there are two directed paths, one from x to $\neg x$ and one from $\neg x$ to x. Denote these paths by P_1 and P_2, respectively. For a contradiction let ϕ be satisfiable. Then, in the corresponding truth assignment of ϕ either $x = $ true or $x = $ false. If $x = $ true, then in path P_1 there is an edge (a, b) such that $a = $ true and $b = $ false. On the other hand, if $x = $ false, then in path P_2 there is an edge (a, b) such that $a = $ true and $b = $ false; see Fig. 10.3. In both cases, however, the existence of the edge (a, b) implies the existence of clause $(\neg a \lor b)$ in ϕ, which contradicts our assumption that ϕ is satisfiable. $\square$

In view of Theorem 10.1, to determine whether formula ϕ is unsatisfiable, it suffices to check whether for every variable x of ϕ there exist two directed graphs in G_ϕ, one from x to $\neg x$ and

one of from $\neg x$ to x. This problem can be easily solved in polynomial time (e.g., using BFS), which proves that the 2SAT problem is in P.

10.6 The convex hull problem

To introduce the convex hull problem formally we need the standard definition of a convex polygon; intuitively, a convex polygon has no corner bending inwards the polygon.

> **Definition 10.6** (Convex polygon) A convex polygon is a simple polygon, such that the line segment connecting any two points (either on the boundary or in the interior) of the polygon lies completely within the polygon. Equivalently,
>
> $$\text{Polygon } P \text{ is convex} \iff \forall \, p, q \in P : \overline{pq} \subseteq P$$

The convex hull of a set of points of the Euclidean plane is defined as the smallest convex polygon that encloses all the points in the set. Accordingly, the convex hull problem is seeking for computing the convex hull of a set of given points. Its formal definition follows.

> **Definition 10.7** (Convex hull problem) The input of the convex hull problem consists of a set P of n points $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$ of the Euclidean plane $\mathbb{R}^2$. In the output, the goal is to determine a convex polygon of minimum area containing $p_1, \ldots, p_n$ either on its boundary or in its interior.

For an example, refer to Fig. 10.4a, which illustrates the convex hull of $n = 8$ points $p_1, \ldots, p_8$ on the plane. In this example, the convex hull is formed by the points p_2, p_5, p_8, p_1 and p_4. The remaining points, namely p_3, p_6 and p_7, lie within the interior of the convex hull.

Simplifying assumptions: In order to avoid degenerate cases in our analysis, we make two simplifying assumptions. First, we assume that no two points of P share the same x-coordinate, and second that no three points of P are collinear.

> **Observation 10.3** The following points of P are (by the definition of the convex hull) always on the convex hull: (i) the topmost, (ii) the bottommost, (iii) the leftmost, and (iv) the rightmost.

Note that not all aforementioned points are pairwise different, e.g., in Fig. 10.4a the bottommost and the rightmost points coincide. Further, recall that we have already proven in Chapter 1 that the convex hull problem is in $\Omega(n \log n)$, that is, there is no algorithm to solve the convex hull problem in time $o(n \log n)$. This property is recalled in the following theorem.
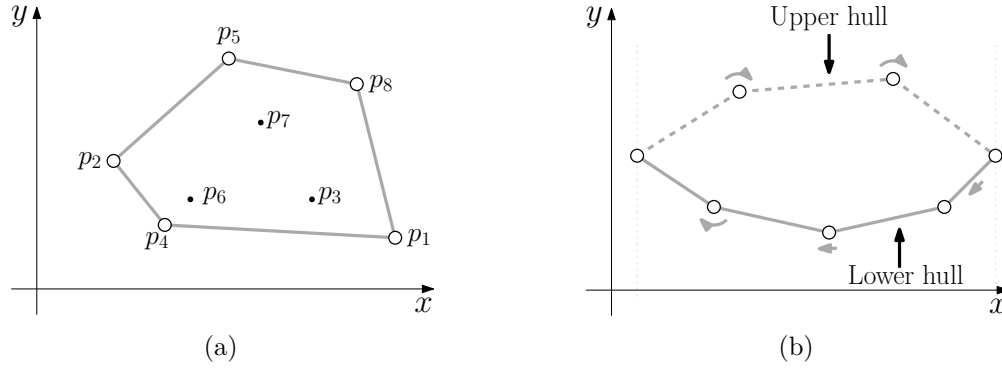
Figure 10.4: Illustration of (a) the convex hull of eight points on the plane, and (b) the upper and lower hull.

Theorem 10.2 The convex hull problem is in $\Omega(n \log n)$

Before trying to solve the convex hull problem, we discuss first how to specify the output of the problem. In particular, how we seek to represent the convex hull. The most natural answer to this question is to specify the input points that are on the boundary by listing these points as they appear in a clockwise traversal of it (e.g., for the example of Fig. 10.4a, we would like to report p_2, p_5, p_8, p_1 and p_4 in this order). We further refer to the part of the convex hull that is delimited between the leftmost and the rightmost point in a clockwise (counterclockwise) traversal of the convex hull as upper hull (lower hull, respective); see Fig. 10.4b for an illustration. Clearly, the union of the upper and lower hulls forms the convex hull of the input points.

10.6.1 An incremental algorithm

As the name suggests, the incremental algorithm builds the convex hull of the input points incrementally by adding to it one point at a time. More precisely, the algorithm is computing independently each of the upper and the lower hulls. In the following, we describe the algorithm for the case of computing the upper hull; the computation of the lower hull is done symmetrically. The algorithm is processing the input points $p_1, \ldots, p_n$, assuming that they are sorted from left to right (recall that no two points p_i and p_j share the same x-coordinate). As an invariant, we assume that the algorithm has computed correctly the upper hull of the first $i-1$ points $p_1, \ldots, p_{i-1}$ and that the points of the computed upper hull have been stored in a stack H, such that the most recently added point in H is the rightmost point of the upper hull; see Figs. 10.5b and 10.5c.

Note that at the beginning of the algorithm, stack H contains points p_1 and p_2 and the invariant is trivially satisfied (see Algorithm 10.3). In the following, we describe how to maintain the invariant after processing of the next point p_i. This point must be appropriately added to the already computed upper hull of the first $i-1$ points $p_1, \ldots, p_{i-1}$, which in turn may also need to be updated (i.e., in the presence of p_i). Since p_i is by definition the rightmost
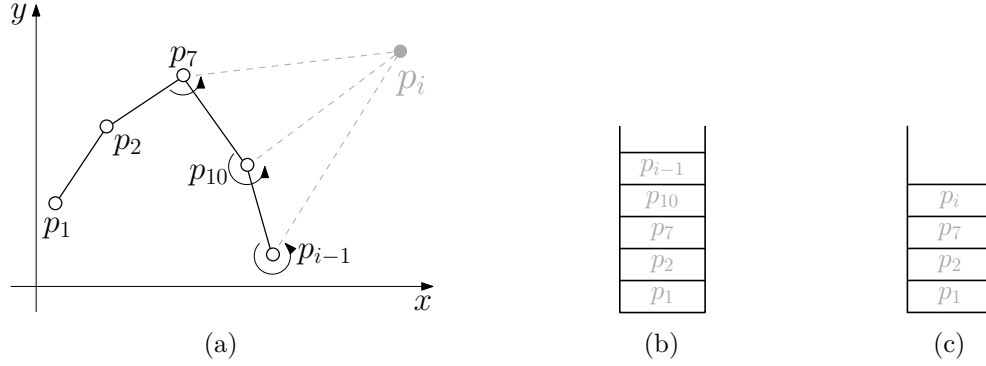
Figure 10.5: Illustration of (a) the computation of the upper hull by the incremental algorithm, (b) the content of H before the addition of point p_i, and (c) the content of H after the addition of point p_i in the upper hull.

point of the first i points, it follows by Observation 10.3 that it necessary belongs to the upper hull of them; see also Fig. 10.5. Hence, we add it to stack H, once we first we remove the points which will not be part of the upper hull after the processing of point p_i. To determine these points, we process the points of the upper hull of the first $i-1$ points in the order that appear in the stack H. In particular, if the angle formed by point p_i, the first point in stack H (denoted by $first(H)$) and the second point in stack H (denoted by $second(H)$) is greater than $180°$, then clearly $first(H)$ does not belong to the upper hull in the presence of point p_i (by convexity). To determine all such points, we pop $first(H)$ from stack H and we recursively apply the same procedure, until we reach a state where p_i, $first(H)$ and $second(H)$ form an angle that is less than or equal to $180°$.

To prove that the aforementioned computation of the upper hull correctly returns the upper hull, it suffices to show that the invariant of the algorithm is maintained in the course of the algorithm. This holds true due to the following two properties. First, point p_i is on the upper hull of the first i points, since it is the rightmost point of them. Secondly, all the popped points do not belong to the upper convex hull as per definition of convexity. Hence, the algorithm is correct.

To estimate the complexity of Algorithm 10.3, we first observe that sorting the input points from left to right takes $\mathcal{O}(n \log n)$ time. Regarding the actual computation of the upper hull, we can easily argue that it takes $\mathcal{O}(n)$ time, since each point is pushed in stack H exactly once and therefore poped at most once. Hence, the time complexity of this algorithm is $\mathcal{O}(n \log n)$ and in conjunction with Theorem 10.2 we can derive the following theorem.

> **Theorem 10.3**   The convex hull problem is in $\Theta(n \log n)$

Algorithm 10.3: An incremental algorithm to compute the upper hull

---

   Input   : A set P of n points of $\mathbb{R}^2$.
   Output : The upper hull of P.

1 $p_1, p_2, \ldots, p_n \leftarrow$ sort the points of P from left to right;
2 Push(p_1, H);
3 Push(p_2, H);
4 for $i = 3$ to n do
5 $\quad$ while $|H| \geq 2$ and the angle formed by $p_i, first(H), second(H)$ is $> 180°$ do
6 $\quad\quad$ Pop(H);
7 $\quad$ end
8 $\quad$ Push(p_i, H);
9 end
10 return H;

10.6.2   A randomized incremental algorithm

In this subsection, we present a randomized incremental algorithm for solving the convex hull problem. To this end, let $p_1, \ldots, p_n$ be a random permutation of the input points and denote by S_i the subset of the first i points $p_1, \ldots, p_i$ in the permutation, where $i = 1, 2, \ldots, n$.
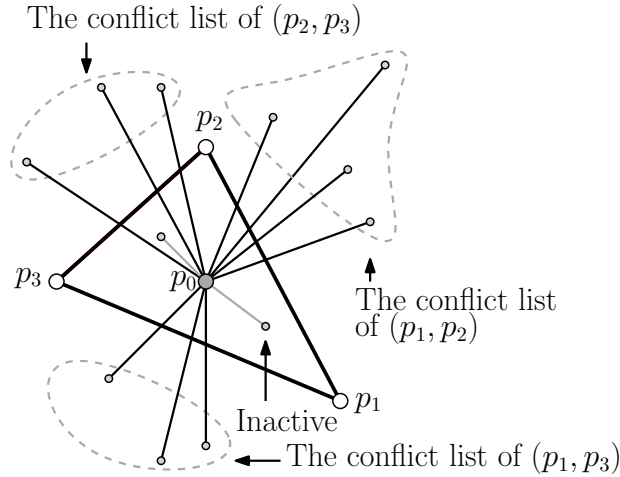
Figure 10.6: Illustration for the base case of the randomized incremental algorithm.

At the beginning the algorithm constructs the triangle, denoted by $conv(S_3)$, which has p_1, p_2 and p_3 as its corners, and considers a fixed point p_0 inside this triangle; see Fig. 10.6. Note that point p_0 is not necessarily one of the points $p_1, \ldots, p_n$. Then, the algorithm draws a straight-line segment, called arc, from point p_0 to each point in $P \setminus S_3$. Note that each such arc may or may not intersect the edges of triangle $conv(S_3)$. The points whose arcs intersect an edge e of triangle $conv(S_3)$ form the so-called conflict list of edge e. The points of $P \setminus S_3$ in a conflict list are marked as active, while the remaining ones are marked as inactive. It is not difficult to see that inactive points cannot be part of the convex hull of $p_1, \ldots, p_n$.

As invariant properties of our algorithm, assume that we have computed the convex hull,
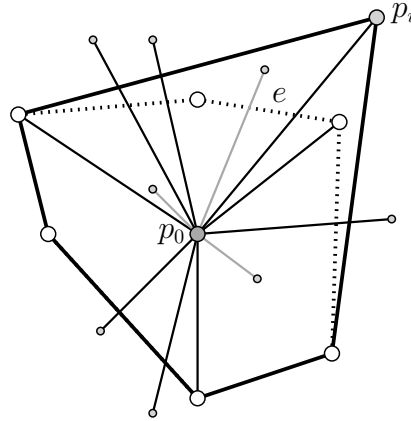
Figure 10.7: Illustration for the inductive step of the randomized incremental algorithm.

denoted by $conv(S_{i-1})$, of the first $i-1$ points in S_{i-1}. We further assume that each edge of $conv(S_{i-1})$ is associated with a conflict list of active points in $P \setminus S_{i-1}$, and that each active point in $P \setminus S_{i-1}$ is associated with an edge of $conv(S_{i-1})$. Points of $P \setminus S_{i-1}$ that lie in the interior of $conv(S_{i-1})$ are marked as inactive. Hence, all invariant properties are preserved at the beginning of the algorithm, as required.

Consider now the next point p_i. If p_i is inside $conv(S_{i-1})$, then clearly nothing is needed to be done and we proceed to consider the point. Hence, we may assume that p_i does not lie inside $conv(S_{i-1})$. Let e be the edge of $conv(S_{i-1})$ associated with p_i. We proceed in there steps (see also Algorithm 10.4).

Step 1: In the presence of point p_i, some edges of $conv(S_{i-1})$ may not be part of $conv(S_i)$; see the dotted edges of Fig. 10.7. We refer to such edges as visible edges. Formally, an edge (u,v) of $conv(S_{i-1})$, such that u appears before v in a clockwise traversal of $conv(S_{i-1})$ starting from its leftmost vertex, is visible if and only if the angle formed by p_i, u and v is greater than $180°$. To determine the visible edges, we start from edge e, and we traverse the edges of $conv(S_i)$ both clockwisely (first) and counterclockwisely (afterwards). We stop each of these traversal, once we identify an edge that is no longer visible from p_i.

Step 2: We replace all the visible edges that we identified in Step 1 with a new pair of edges incident to p_i in $conv(S_i)$. In particular, these edges connect p_i to the two vertices of $conv(S_{i-1})$ incident to the two visible edges that were the last ones to be removed from $conv(S_{i-1})$ at each of the two traversals of Step 1.

Step 3: The points associated with the visible edges in $conv(S_{i-1})$ must be associated to one of the newly introduced edges of $conv(S_i)$ or must be marked as inactive (if they lie in in $conv(S_i)$).

Steps 1, 2 and 3 guarantee that all invariant properties of our algorithm are maintained in the course of the algorithm. Hence, Algorithm 10.4 will compute the convex hull of the input points. The next question that remains to be answered regards the time complexity of the algorithm. For Steps 1 and 2, observe that every (active) point introduces two edges in the convex hull and for an edge to be removed from the convex hull, it must be introduced first.

Algorithm 10.4: A randomized incremental algorithm to compute the upper hull

---

   Input   : A set P of n points of $\mathbb{R}^2$.
   Output : The convex hull of P.

1   $p_1, p_2, \ldots, p_n \leftarrow$ a random permutation of P;
2   $conv(S_3) \leftarrow$ the triangle formed by p_1, p_2, p_3;
3   Compute the conflict list of each edge of $conv(S_3)$;

4   for $i = 4$ to n do
5   if p_i is active then
6         $e \leftarrow$ the edge of $conv(S_{i-1})$ associated with p_i;
7   S.1: Go left / right from e to find all visible edges from p_i;
8   S.2: Replace visible edges with a new pair of edges in $conv(S_i)$;
9   S.3: Move the points associated to the visible edges to the new edges of
            $conv(S_i)$ and filter for inactivity;
10   end
11 end
12 return $conv(S_n)$;

Also, an edge is deleted at most once. So, the introductions and deletions of edges in Steps 1 and 2 can be done in $\mathcal{O}(n)$ time. Step 3, however, where we perform pointer updates (of the points associated to the visible edges of $conv(S_{i-1})$ to the new edges of $conv(S_i)$), is more challenging. In particular, at each iteration of the algorithm we may have to perform $\mathcal{O}(n)$ pointer updated, which results in a total of $\mathcal{O}(n^2)$ time in worst case. We improve this upper bound with backward analysis.

Backward analysis. In backward analysis, we estimate the number of pointer updates (which define the time complexity of the algorithm) by assuming that point p_i is deleted from $conv(S_i)$ to derive $conv(S_{i-1})$. In other words, we consider the randomized incremental construction of the convex hull by the algorithm backwards, i.e., we follow the reversed procedure of the one that the algorithm followed in order to compute the convex hull (which also justifies the name of this analysis). In this case, the expected number of pointer updates is given by the following formula:

$$\sum_{e \in conv(S_i)} (\text{size of } e\text{'s conflict list}) \cdot Pr(e \text{ is removed})$$

To bound the aforementioned expression, we first observe that the probability that an edge e of $conv(S_i)$ is deleted from $conv(S_i)$ in order to derive $conv(S_{i-1})$ is $\frac{2}{i}$, since one of its endpoints must be point p_i. Further, the size of the conflict lists of all edges of $conv(S_i)$ is at most n. By the formula above, it follows that the expected number of pointer updates is upper bounded by $\frac{2n}{i}$ and since each of them has a constant cost, it follows that the expected cost of the pointer updates of the algorithm is $\mathcal{O}(\frac{2n}{i})$. Summing over all i, we obtain an expected total cost of:

$$\sum_{i=1}^{n} \frac{2n}{i} = 2n \sum_{i-1}^{n} \frac{1}{i} = 2 \log n$$

This implies that the expected running time of our algorithm is $\mathcal{O}(n \log n)$.

Exercises

Exercise 10.1 Consider 25 consecutive flips of a fair coin. What is the expected number of consecutive pairs of heads?

For example, in a sequence $H - H - T - H - H - T - T - H - H - H - T$ of flips, there are 4 consecutive pairs of heads.

Exercise 10.2 Consider the following probabilistic process:

Initialize a counter c to 0 and then repeat n time the following procedure. Flip a fair coin and if the result is head then increase c by one; Otherwise, decrease c by one.

Compute the expected value of counter c.

Exercise 10.3 Let $\mathcal{I}$ be an instance of the 3-satisfiability problem with n variables $x_1, x_2, \ldots, x_n$ and m clauses $c_1, c_2, \ldots, c_m$, each with exactly three literals. Now, consider the following randomized algorithm.

For each $1 \le i \le n$, flip a fair coin and if the result is head, then set x_i to true; otherwise set x_i to false.

a) What is the probability that clause c_i is satisfied?

b) What is the expected number of satisfied clauses in $\mathcal{I}$?

c) Argue why for any instance of the 3-satisfiability problem, there is a truth assignment in which at least $\frac{7}{8}$ of its clauses are satisfied.

Exercise 10.4 Let G be a connected graph with n vertices and m edges. Let also c_1, c_2 and c_3 be three colors. Consider the following randomized algorithm.

For each vertex v of G, color v with a color chosen uniformly at random from $\{c_1, c_2, c_3\}$.

An edge (u, v) of graph G is called monochromatic if its endvertices u and v have received the same color by the procedure above, and bichromatic otherwise.

a) What is the probability of an edge of G to be bichromatic?

b) What is the expected number of bichromatic edges?

c) Argue why for any graph, there is a coloring of its vertices in which at least $\frac{2}{3}$ of its edges are bichromatic.

Exercise 10.5 Let P be a set of n points in general position on the Euclidean plane $\mathbb{R}^2$. A point (x, y) in P is dominating another point (x', y') in P if and only if $x \ge x'$ and $y \ge y'$. A dominating point in P is one that is dominated by no other point in P.

a) Design a randomized incremental algorithm that computes the dominating points of P in $\mathcal{O}(n \log n)$. Explain why your algorithm is correct.

b) Use backward analysis, to analyse the time complexity of your algorithm.

Exercise 10.6 A simple polygon P is called star-shaped, if it contains a point q, such that for every corner p of P the line-segment $\overline{pq}$ that connect p and q lies completely in the interior of P. Assume that polygon P is specified by a set of n closed line-segments which are ordered according to the order that they appear along the boundary of P such that two consecutive line-segments share exactly one common endpoint.

a) Present a deterministic algorithm to decide whether a given simple polygon is star-shaped.

b) Present a randomized incremental algorithm to decide whether a simple polygon is star-shaped and use backward analysis to analyse its time complexity.

Further reading

– Chapter 5 from: T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009.

– Chapter 13 from: J. M. Kleinberg and É. Tardos. Algorithm design. Addison-Wesley, 2006.

– http://www.cs.mun.ca/~wang/courses/cs6901-17f/n6-Randomised2DCH.pdf

Bibliography

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009.

[2] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. Comb., 10(1):41–51, 1990.

[3] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. J. ACM, 21(4):549–568, 1974.

[4] J. M. Kleinberg and É. Tardos. Algorithm design. Addison-Wesley, 2006.

[5] H. W. Kuhn. The hungarian method for the assignment problem. Naval Research Logistics Quarterly, 2(1-2):83–97, 1955.

[6] T. Nishizeki and N. Chiba. Planar Graphs: Theory and Algorithms. North-Holland, 1988.

[7] M. Stoer and F. Wagner. A simple min-cut algorithm. J. ACM, 44(4):585–591, 1997.

[8] R. J. Vanderbei. Linear programming - foundations and extensions, volume 4 of Kluwer international series in operations research and management service. Kluwer, 1998.

[9] V. V. Vazirani. Approximation algorithms. Springer, 2001.

Index

2SAT, 222
3SAT, 233

accounting method, 32
adjacency-lists representation, 18
adjacent edges, 7
adjacent vertices, 7
admissible network, 83
admissible path, 83
aggregate method, 33
algorithms' correctness, 12
alternating path, 105
amortised analysis, 31
approximation algorithm, 195
approximation factor, 194
approximation scheme, 195
arc cover, 127
augmenting path, 72, 105
augmenting paths theorem, 78
average complexity analysis, 29

bin packing, 205
binary random variable, 216
bipartite graph, 8, 107
Bland's rule, 170, 171
blocking flow, 84
blossom, 115
blossom shrinking, 115
bottleneck, 76

canonical form, 160
canonical order, 139

Christofides' algorithm, 200
complement graph, 154
complementary slackness, 179
complete graph, 8
conditional probability, 216
connected graph, 7
convex polygon, 227
covex hull, 227
crossing lemma, 151
crossing number, 151

degenerate dictionary, 170
degenerate pivot, 171
Dijkstra's algorithm, 36
Dinitz's algorithm, 83
directed edge, 7
directed graph, 7
disconnected graph, 7
discrete random variable, 216
divide-and-conquer, 21
dual linear program, 174

edge, 7
edge coloring, 127
edge-connectivity, 101
Edmonds and Karp's algorithm, 81
Edmonds' algorithm, 114
endvertex, 7
equality graph, 123
expected value, 216

feasible labeling, 122

236

DRAFT