Optimization methods

FAMNIT
University of Primorska
Koper, Slovenia

Lecturer: Dr. Clément Dallard
Author: Dr. Clément Dallard

Contents

Chapter 1

Computational complexity

We refer the interested reader to [37] for a more complete tutorial and to [31] for a rigorous book on the subject.

1.1. Problem, instance and algorithm

A *problem* is a generic question to be solved and an *instance* of a problem corresponds to the input for the problem. The *size* of an instance I is the number of bits required to represent I. A *solution* to a problem is the output corresponding to the given input of the problem.

Example 1.1.1. Consider the problem of *primality testing*. An instance to this problem is an integer. The solution is "yes" if the number is prime and "no" otherwise.

An *algorithm* for a problem Π is a step-by-step procedure describing how to compute a solution for every instance of Π. We can assume that an algorithm always returns an output, even if constant. Algorithms fall into two categories: *deterministic* and *nondeterministic*.

Definition 1.1.1 (Deterministic and nondeterministic algorithms)**.**

- A *deterministic algorithm* always goes through the same sequence of states and return the same output for a fixed given input.
- A *nondeterministic algorithm* may go through a different sequence of states and return different outputs for a fixed given input.

To compare the efficiency of different algorithms for a same problem Π, we usually compare the maximum number of operations required by each algorithm to solve any instance of Π (which of course may depends on the size of the instance). The *running time* of an algorithm A is the number of elementary operations performed by A expressed as a function of the size of the instance. The *(worst-case) complexity* of an algorithm is the maximum number of operations it needs to solve an instance I over all instances of size n.

In this course, we express the complexity of an algorithm with the $\mathcal{O}$ *notation*.

Definition 1.1.2 ($\mathcal{O}$ notation)**.** Given two functions f and g, we say that $f = \mathcal{O}(g)$ ("f is big Oh of g") if there exist two positive constants α and x_0 such that, for all $x \geq x_0$, $f(x) \leq \alpha \cdot g(x)$.

1.2. Decision problem and P vs. NP

A *decision problem* is a problem that can be posed as a yes-no question of the input values. We are particularly interested in problems for which a *certificate* (also called *witness* and *proof* in the literature) can be verified in polynomial time. Informally, if someone gives us an "argument" that a solution for a specific instance is "yes," then we want to be able to check whether the "argument" is valid in polynomial time.

Definition 1.2.1 (NP class)**.** The class NP is the set of all decision problems for which the problem instances with answer "yes" have certificates verifiable in polynomial time.

The *complement* of a decision problem is the decision problem resulting from reversing the "yes" and "no" answers, and the class coNP contains the decision problems whose complements are in NP.

Definition 1.2.2 (P class)**.** The class P is the set of all decision problems that can be solved in polynomial time with a deterministic algorithm.

While some problems in NP are proved to belong to P, others are not, despite decades of research. There lies the P versus NP problem; there seems to exist a dichotomy of the problems in NP between problems that are "easy to solve" (in P) and problems that are "hard to solve." As described by Cook in [4], the P versus NP problem is "*to determine whether every language accepted by some nondeterministic algorithm in polynomial time is also accepted by some (deterministic) algorithm in polynomial time*." It is one of the seven Millennium Prize Problems.[1]

To describe this possible complexity dichotomy in further details, we need to define the concept of *polynomial-time reduction*.

1.3. Polynomial-time reduction

Definition 1.3.1 (Polynomial-time reduction)**.** Let A and B be two decision problems. A *polynomial-time reduction* is a function f that maps instances of B into instances of A in polynomial time such that, for any instance x of B, x is a yes-instance if and only if $f(x)$ is a yes-instance. We say that B (polynomial-time) *reduces* to A and write $B \propto A$.

1.4. NP-hardness and NP-completeness

We can now formally define the notions of NP-*hardness* and NP-*completeness*.

Definition 1.4.1 (NP-hardness and NP-completeness)**.** A decision problem A is NP-hard if every problem B in NP can be reduced in polynomial time to A. Furthermore, if A is NP-hard and belongs to NP, then A is NP-complete.

Informally, an NP-hard problem is at least as hard as any problem in NP. A polynomial-time reduction is one of the most useful tools at the disposal of the researcher for characterizing the complexity of a problem. Its main use is to prove that a problem A is NP-hard by showing that there exists an NP-hard problem B and a polynomial-time reduction from B to A. But a polynomial-time reduction can also prove that a problem is in P: If a problem A can be reduced to a problem B in P, then A is obviously in P as well.

Although not discussed in this course, it is interesting to mention that, under the assumption that $\mathsf{P} \neq \mathsf{NP}$, there exist problems within NP that are not in P and not NP-complete [25] (these problems are called NP-intermediate).

[1] https://www.claymath.org/millennium-problems/p-vs-np-problem

The existence of an NP-hard problem is, a priori, not clear. In 1971, Cook, an American-Canadian computer scientist and mathematician, proved that the BOOLEAN SATISFIABILITY problem, commonly known as SAT, is an NP-hard problem by showing that any problem in NP can be reduced to SAT [5]. Interestingly, Levin, a Soviet-American mathematician and computer scientist, independently proved the same result in 1973 [26]. This result is now known as the *Cook-Levin theorem.*

An instance of SAT is a formula in conjunctive normal form (CNF), also called a *CNF formula*, which is a conjunction of one or more clauses, where a clause is a disjunction of literals. When each clause contains at most ℓ variables, we say that the formula is a *ℓ-CNF formula.* If it is possible to assign *True* or *False* values to all variables such that the formula evaluates to *True*, then the formula is *satisfiable* and the assignment is *satisfying.* On the other hand, if there is no satisfying assignment of the formula, then the formula is *unsatisfiable.* Hence, SAT is the problem of deciding if a given formula is satisfiable. One can check in polynomial-time if a given assignment of the variables satisfies the formula, and hence SAT is NP-complete. The 3-SAT problem, where each clause contains *exactly* 3 variables, is also NP-complete.

3-SAT
Input: A 3-CNF formula ϕ with clauses of size exactly 3.
Question: Is there a satisfying assignment of ϕ?

In 1972, Karp used the Cook-Levin theorem to show that there is a polynomial-time (many-one) reduction from SAT to 21 different combinatorial and graph theoretical problems, thereby proving that they are NP-complete [19]. Nowadays, thousands of problems are known to be NP-complete.

Let us illustrate the concept of polynomial-time reduction by reducing 3-SAT to the MIN VERTEX COVER problem, and thus proving that the latter is NP-hard. A *vertex cover* is a subset of vertices S such that each edge contains at least one endpoint in S. The proof we present here can comes from [16]. We safely assume that a clause does not contain the positive and negative literals of a same variable, otherwise we can simply remove the clause from the formula.

MIN VERTEX COVER
Input: A graph G and an integer k.
Question: Is there a vertex cover of size at most k in G?

The general idea of Construction 1.4.1 is to create a graph from a 3-CNF formula where each variable is represented by a *variable gadget* and each clause is represented by a *clause gadget.*

Construction 1.4.1 (3-SAT to MIN VERTEX COVER)**.** *Let ϕ be a 3-CNF formula with exactly 3 variables per clause, that is, a set of m clauses $C_1, C_2, \dots, C_m$ on n variables $x_1, x_2, \dots, x_n$. We construct the graph $G = (V, E)$ as follows:*

- *for each variable x_i, create a* variable gadget *made of two adjacent vertices v_i and $\bar{v}_i$;*
- *for each clause C_j, create a* clause gadget *T_j, which is a clique of size 3 (a triangle);*
- *finally, for each variable x_i of a clause C_j, connect one vertex in T_j to v_i if the literal $x_i \in C_j$ or to $\bar{v}_i$ if $\bar{x}_i \in C_j$, and such that the vertices in T_j have degree 3 (connected to exactly one vertex outside the clause gadget).*

Of course, Construction 1.4.1 can be done in polynomial time (see Fig. 1.1 for an example).

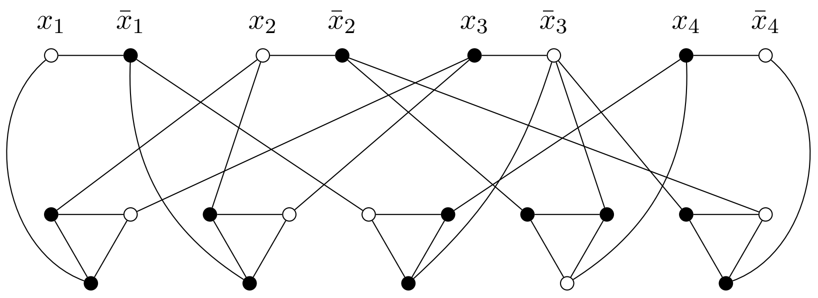


Figure 1.1 Example of a graph obtained through Construction 1.4.1 with the 3-CNF formula $(x_1 \lor x_2 \lor x_3) \land (\bar{x}_1 \lor x_2 \lor x_3) \land (\bar{x}_1 \lor \bar{x}_3 \lor x_4) \land (\bar{x}_2 \lor \bar{x}_3 \lor x_4) \land (\bar{x}_2 \lor \bar{x}_3 \lor \bar{x}_4)$. Vertices in black belong to a vertex cover of size $n + 2m = 14$, and hence $x_1 := False$, $x_2 := False$, $x_3 := True$ and $x_4 := True$.

Theorem 1.4.1. *Min Vertex Cover is* NP*-complete.*

Proof. Of course, Min Vertex Cover is in NP. Let ϕ be a 3-CNF formula with exactly 3 variables per clause, that is, a set of m clauses $C_1, C_2, \ldots, C_m$ on n variables $x_1, x_2, \ldots, x_n$, and consider the graph $G = (V, E)$ obtained through Construction 1.4.1. We claim that ϕ is satisfiable if and only if there exists a vertex cover of size at most $n + 2m$ in G.

Let β be a satisfying assignment of ϕ and S be an empty set of vertices. For each variable x_i, if x_i is set to *True* in β, then add the vertex v_i to S, otherwise add the vertex $\bar{v}_i$ to S. At this point, $|S| = n$ and S covers all the edges between the vertices representing the literals of the variables. Now, consider a clause gadget T_j representing a clause C_j and note that there exists $u_j \in T_j$ adjacent to a vertex in S, since β satisfies ϕ. Then, add the vertices in $T_j \setminus \{u_j\}$ to S. Note that all the edges of the clause gadget and between the clause gadget and the corresponding variable gadgets are covered by S. Once all the clause gadgets have been processed, S is a vertex cover of size $n + 2m$.

Let S be a vertex cover in G of size at most $n+2m$. It is easy to see that S contains at least two vertices for each clause gadget and at least one vertex for each variable gadget. Therefore, S has size exactly $n + 2m$ and contains exactly two vertices per clause gadget and one vertex per variable gadget. We set a variable x_i to *True* in β if the vertex v_i belongs to S, otherwise we set it to *False*. Note that doing so, each variable is assigned one (and only one) value because S contains exactly one vertex per variable gadget. Now, suppose that there exists a clause C_j which is not satisfied by β and let u_j be the only vertex in T_j which does not belong to S. Without loss of generality, let v_i be the vertex from a variable gadget adjacent to u_j. Since S is a vertex cover and $u_j \notin S$, then $v_i \in S$, which implies that $x_i = True$ in β. A contradiction with the fact that the clause is not satisfied. □

1.5. Optimization problem

An *optimization problem* is, informally, the problem of finding the "best" solution among all feasible solutions. Usually, the "best" solution either *minimizes* or *maximizes* some value derived from the solution, and therefore we talk about *minimization* and *maximization* problems. It is quite common that the computational complexity of the minimization variant of a problem differs from the maximization variant of this same problem. For instance, Shortest Path is polynomial-time computable but Longest Path is NP-hard (so if $\mathsf{P} \neq \mathsf{NP}$, the latter cannot be solved in polynomial time).

1.6. Practical issue of exponential algorithms

It seems natural to expect that an algorithm returns a valid solution to a decision problem, and in case of an optimization problem, an optimal solution. Among all algorithms that are able to return such a solution, the one with the smallest complexity is usually preferred. Clearly, the time required to solve a problem does not decrease if the size of the instance increases; at best, it remains constant (e.g., check if a given integer represented as a binary string is even), but often it increases.

Example 1.6.1. Suppose we are given a CNF formula with n variables $x_1, \ldots, x_n$. It is clear that we can answer whether the expression is satisfiable by listing all binary vectors $(x_1, \ldots, x_n)$ and checking for each of them if it satisfies the expression. However, we may have to check 2^n vectors before being able to decide whether the expression is satisfiable or not. For $n = 50$, this means $2^{50} > 10^{15}$ vectors. Even if our computer is able to check 10^6 solutions per second, the whole procedure could take up to 10^9 seconds, or about 30 years. And for $n = 60$, this would take 30000 years. Note that for each step of 10 variables, the time to solve the instance is multiplied by a factor of $2^{10} = 1024$. So, if $n = 80$ (which is not such a large number of variables in practice), the same algorithm would run for (at least) 30 billion years, which is about twice the age of the universe.

To avoid such issues, the researcher (or programmer) tries to come with efficient and exact algorithms that run in polynomial time, if they exists. If the solutions (of an optimization problem) do not need to be optimal but only "close to being optimal," then approximation algorithms can be an efficient alternative. If optimality is required and there is no known polynomial-time algorithm for solving the problem, then two options are still possible: considering specific kinds of instances that can be solved efficiently (if possible in polynomial time), or optimizing the naive exponential-time algorithm by decreasing the values of the exponent and the base in the function expressing the computational complexity of the algorithm (this often implies the design of complex algorithms and fine complexity analysis). On the other hand, one can prove complexity lower bounds on the given problem. For a short paper on the subject, see [32].

1.7. Exercises

Exercise 1 (Polynomial-time reductions)**.**

- Show that SAT and 3-SAT are equivalent.
- Show that Hamiltonian Path and Hamiltonian Cycle problems are equivalent.

Chapter 2

Dynamic programming

In this section we describe a powerful algorithmic design technique known as *dynamic programming* (abbreviated as DP) to develop polynomial-time algorithms for optimization problems. A large class of seemingly hard problems are in fact polynomial-time solvable thanks to dynamic programming.

2.1. Principle

Dynamic programming, a concept introduced by *Richard Bellman* in the 1950s, can be seen as a “careful brute-force” or a “clever exhaustive search”, where the instance is broken down into smaller instances and solutions to such instances help computing an optimal solution to the main instance. For simplicity and consistency with many textbooks and online lectures,[1] the smaller instances are referred to as “subproblems.”

2.1.1. Fibonacci numbers

We first consider a simple example: computing the Fibonacci numbers.

$$\begin{aligned} F_1 &= F_2 = 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

The goal is to compute F_n.

Algorithm 1: Naive Fibonacci

```
Fib($n$):
1 if $n \leq 2$ then
2     return 1
3 return Fib($n-1$) + Fib($n-2$)
```

The complexity of Algorithm 1 is *exponential*. Indeed, we can write the running time as the following recurrence, where $T(n)$ is the time to compute the nth Fibonacci number (denoted by F_n):

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + \mathcal{O}(1) \geq F_n \approx \varphi^n \\ &\geq 2T(n-2) + \mathcal{O}(1) \geq \mathcal{O}\left(2^{n/2}\right) . \end{aligned}$$

The recursion tree has exponentially many nodes (see Fig. 2.1 for an example). However, observe that the same values are computed several times. The idea of dynamic programming is to avoid computing those same values several times and, instead, store them for (possible) future use. This technique is called *memoization*.

[1]We recommend the reader the following lectures of Erik Demaine at the MIT: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/.

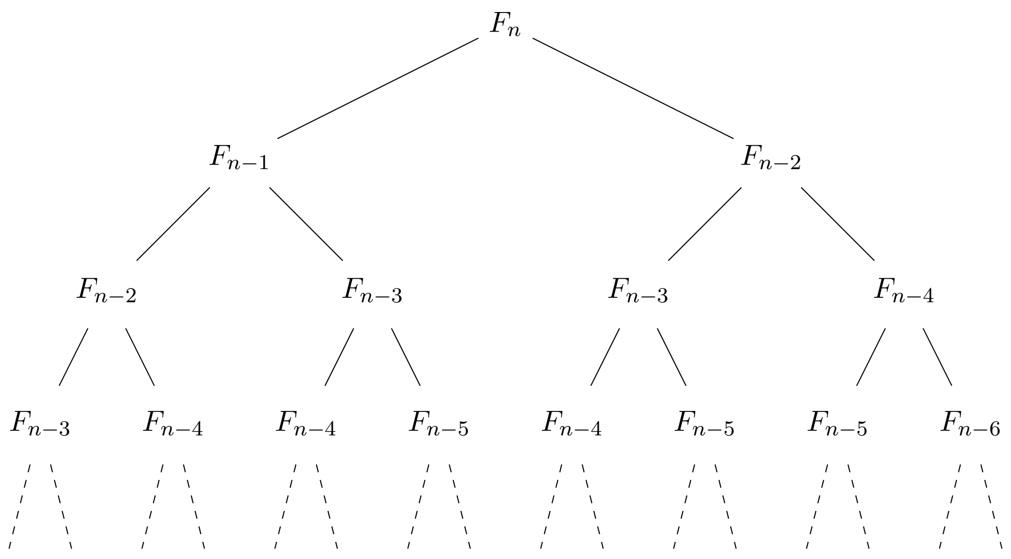


Figure 2.1 A schematic representation of the recursion tree of Algorithm 1.

Algorithm 2: Memoized DP Fibonacci

```
memo[$k$] = $-1$, for all $k \in \{1, \dots, n\}$
FIB($n$):
1 if memo[$n$] $\neq -1$ then
2     return memo[$n$]
3 $f$ = FIB($n-1$) + FIB($n-2$)
4 memo[$n$] = $f$
5 return $f$
```

Algorithm 2 has many advantages over Algorithm 1:

- The function Fib(k) recurses only if it is the first time it is called for such a value k.
- It means that there are only n non-memoized calls of the function, for $k \in \{1, \dots, n\}$.
- Memoized calls cost constant time ($\mathcal{O}(1)$ time).

Hence, Algorithm 2 is a *linear*-time algorithm!

In general, the running time depends on the number of subproblems multiplied by the time to solve each subproblem ignoring when recursive calls; in Algorithm 2, each subproblem can be solved in constant time.

Algorithm 3 can be seen as a version of Algorithm 2 where the recursion is "unrolled." It also has a linear time complexity.

In general, the bottom-up version performs the same computation steps as as the memoized version, and and hence same running time. The main difference between the recursive and the bottom-up versions is the order in which the subproblems are considered. While a recursive DP algorithm breaks each problem into smaller subproblems until it reaches the base case and then

Algorithm 3: Bottom-up DP Fibonacci

```
for $k \in \{1, 2, \dots, n\}$ do
    if $k \leq 2$ then
        $f = 1$
    else
        $f = \mathsf{memo}[k-1] + \mathsf{memo}[k-2]$
    $\mathsf{memo}[k] = f$
return $\mathsf{memo}[n]$
```

goes up the recursion tree, a bottom-up algorithm starts with the smallest (nontrivial) subproblems. More specifically, in order to develop a bottom-up DP algorithm, one first needs to compute the *topological sort* of the dependencies between the subproblems, that is, the directed acyclic graph (DAG) whose vertices correspond to the subproblems and where an edge (uv) means that the subproblem u has to be solved before the subproblem v; we call this graph the *dependency graph* (of the subproblems). The topological sort gives us a way to compute the subproblems such that if a subproblem (or the main problem) depends on some other subproblems, then the latter have already been solved (see Fig. 2.2).

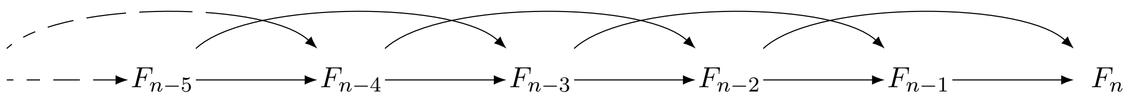

$$\begin{tikzcd}
\cdots \arrow[r, dashed] & F_{n-5} \arrow[r] \arrow[rr, bend left] & F_{n-4} \arrow[r] \arrow[rr, bend left] & F_{n-3} \arrow[r] \arrow[rr, bend left] & F_{n-2} \arrow[r] \arrow[rr, bend left] & F_{n-1} \arrow[r] & F_n
\end{tikzcd}$$

Figure 2.2 A schematic representation of the dependency DAG for computing the Fibonacci number F_n.

The bottom-up technique usually allows to save space in the algorithm. Indeed, in Algorithm 3, instead of using a table memo of length n, we can only store the last two values. Hence, this algorithm needs constant space. It is also sometimes easier to analyze the running time of a bottom-up DP algorithm than the one of a memoized DP algorithm.

2.2. Finding a shortest path

First, let's consider a simplified variant where we want to compute a shortest path between two given vertices s and t in a directed acyclic graph (DAG).

What do you think of Algorithm 4? Is it exponential, polynomial, linear? And what if, as in Algorithm 5, we implement memoization?

In fact, both Algorithms 4 and 5 would fail on graph with cycles.[2] For example, consider the graph depicted in Fig. 2.3. To compute the shortest path from a to d, the algorithm needs to compute the shortest path from a to b, which depends on the shortest path from a to c. However, the latter depends on the shortest path from a to a and, more problematically, on the shortest path from a to d. In other words, the dependency DAG contains a (directed) cycle, and thus the algorithm does not terminate.

Nonetheless, if the input graph does not contain a cycle, then Algorithm 5 has a running time of $\mathcal{O}(|V(G)| + |E(G)|)$: there are $|V(G)|$ subproblems and for each subproblem the algorithm

[2]For simplicity, *directed cycles*, also known as *circuits*, are simply called *cycles*.

Algorithm 4: DAG recursive shortest path

```
   Data: a directed graph $G$, $s, t \in V(G)$ and a weight function $w : E(G) \to \mathbb{R}$
   Result: the minimum distance between $s$ and $t$
   return ShortestPath($t$)
   // ShortestPath function definition
   ShortestPath($v$):
1  if $v = s$ then
2  |   return 0
3  return $\min_{(u,v)\in E(G)}$ ShortestPath($u$) $+ w(u, v)$
```

Algorithm 5: DAG recursive shortest path with memoization

```
   Data: a directed graph $G$, $s, t \in V(G)$ and a weight function $w : E(G) \to \mathbb{R}$
   Result: the minimum distance between $s$ and $t$
   // Initialization
   dist[$s$] $= 0$
   dist[$v$] $= +\infty$, for all $v \in V(G) \setminus \{s\}$
   return ShortestPath($t$)
   // ShortestPath function definition
   ShortestPath($v$):
1  if dist[$v$] $\neq +\infty$ then
2  |   return dist[$v$]
3  dist[$v$] $= \min_{(u,v)\in E(G)}$ ShortestPath($u$) $+ w(u, v)$
4  return dist[$v$]
```

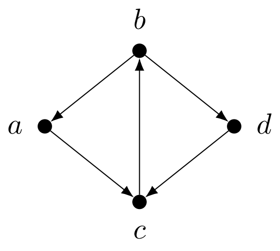


Figure 2.3 A simple graph for which Algorithms 4 and 5 would fail to compute the shortest paths.

checks every incoming edge to the considered vertex. The sum of the indegrees of the vertices is $\mathcal{O}(|E(G)|)$—by the handshaking lemma—and thus we obtain the claimed complexity.

The dependency graph of the subproblems must be acyclic (a DAG), otherwise we have an infinite algorithm. The question is "how to get the dependency graph to be a DAG when the input graph contains cycles?"

2.2.1. From a single source vertex

We want to compute the shortest path from a vertex s (the source) to every other vertex in $V(G)$. This is equivalent to computing a shortest-path tree rooted at s (see Exercise 4).

In this section, we study two algorithms with different complexities and different applications. The first algorithm, known as *Dijkstra's algorithm* [8], is an efficient greedy algorithm but requires nonnegative weights on the edges of the input graph. The second algorithm, known as *Bellman-Ford algorithm* [1, 12, 29, 35], uses a dynamic programming approach and, although less efficient (in the worst case), can detect negative weight cycles.

Dijkstra's algorithm

We consider that the input graph G is a directed graphs with nonnegative weights on its edges.

Algorithm 6 uses the following property: *If w is a vertex on a shortest path P from u to v, then the subpath of P from u to w is a shortest path. ($\star$)*

Although the dependency graph of the subproblems for the (single source) shortest path problem may contain cycles (if the input graph is not acyclic), property ($\star$) provides a way to bypass the issue by simplifying the dependency graph into a dependency tree[3] on which we can apply the dynamic programming concepts and claim the correctness of the returned solution. However, this dependency tree is not constructed *before* but *while* solving the problem. This is why *Dijkstra's algorithm* is usually not considered as a dynamic programming algorithm but only as a greedy algorithm.

Lemma 2.2.1. *At the end of Algorithm 6, for every $u \in V(G)$, $\mathsf{dist}[u]$ corresponds to the distance from s to u and* previous *corresponds to a shortest-path tree rooted at s.*

Proof. We denote by $\delta(s,u)$ the minimum distance in the graph from s to u, for any $u \in V(G)$. We say that a vertex u is *visited* at a given step of Algorithm 6 if $u \notin \mathsf{Q}$.
Invariant hypothesis: At any step of Algorithm 6, $\mathsf{dist}[u] = \delta(s,u)$ for every visited vertex u.

[3]This dependency tree corresponds exactly to the shortest-path tree returned by Algorithm 6.

Algorithm 6: Dijkstra's algorithm, 1956 (published in 1959)

```
   Data: a (directed) graph $G$, $s \in V(G)$ and a weight function $w : E(G) \to \mathbb{R}^+$
   Result: a shortest-path tree of $G$ rooted at $s$ (as a table)
   // Initialization
 1 forall $v \in V(G)$ do
 2     dist$[v] = +\infty$
 3     previous$[v] =$ undefined
 4 dist$[s] = 0$
 5 Q $= V(G)$
   // Main loop
 6 while Q is not empty do
 7     $u = \arg\min_{u \in Q}$ dist$[u]$
 8     remove $u$ from Q
 9     foreach $v \in V(G)$ such that $(u, v) \in E(G)$ do
10         alt = dist$[u] + w(u, v)$
11         if alt < dist$[v]$ then
               // This step is often referred to as "relaxation" of the edge $(u, v)$
12             dist$[v] =$ alt
               // If Q is a priority queue, update Q with the new dist$[v]$
13             previous$[v] = u$
14 return previous
```

The base case when $Q = V(G)$ is vacuously true. In fact, s is the first vertex to be visited and $\mathsf{dist}[s] = \delta(s, s) = 0$, so the invariant holds for 1 visited vertex. We prove the statement by contradiction and consider the sate of Algorithm 6 just before the invariant fails for the first time. We denote by $v \in \mathsf{Q}$ the vertex about to be visited (*i.e.*, about to be removed from Q) such that $\mathsf{dist}[v] \neq \delta(s, v)$. Note that the assumption $\mathsf{dist}[v] \neq \delta(s, v)$ implies that there exists a path from s to v; otherwise $\delta(s, v) = \mathsf{dist}[v] = +\infty$. Let u be the visited vertex such that $(u, v) \in E(G)$ and $\mathsf{dist}[v] = \mathsf{dist}[u] + w(u, v)$ (the edge (u, v) has been relaxed). Then $\delta(s, v) < \mathsf{dist}[v] = \mathsf{dist}[u] + w(u, v) < +\infty$. Let P be a shortest path from s to v in G and $(x, y) \in P$ be the first edge in P such that just before v gets removed from Q, the vertex x is visited ($x \notin \mathsf{Q}$) but y is not yet visited ($y \in \mathsf{Q}$).

We first claim that $\mathsf{dist}[y] = \delta(s, y)$ when v is visited. Observe that x is visited before v, and thus by hypothesis we have $\mathsf{dist}[x] = \delta(s, x)$. Also, notice that since x is visited, we have that $\mathsf{dist}[y] \leq \mathsf{dist}[x] + w(x, y) = \delta(s, x) + w(x, y)$. Using property ($\star$) and the fact that P is a shortest path, we get $\mathsf{dist}[y] = \delta(s, x) + w(x, y) = \delta(s, y)$, as claimed.

If $y = v$, then $\mathsf{dist}[v] = \mathsf{dist}[y] = \delta(s, y) = \delta(s, v)$, a contradiction with the assumption that $\mathsf{dist}[v] \neq \delta(s, v)$. Hence, $y \neq v$. Recall that y appears before v in P and all edges have a nonnegative weight. Consequently, we have $\mathsf{dist}[y] = \delta(s, y) \leq \delta(s, v) < \mathsf{dist}[v]$. However, this contradicts the choice of v at Line 9.

To conclude, at the end of Algorithm 6, the queue Q is empty, and hence every vertex has been visited. As we just showed, $\mathsf{dist}[u] = \delta(s, u)$ for every vertex $u \in V(G)$. Furthermore, for each vertex $u \in V(G) \setminus \{s\}$, the vertex $\mathsf{previous}[u]$ corresponds to the penultimate vertex on a shortest path from s to u in G, with the additional property that $\mathsf{previous}[u]$ is always set to a visited vertex. In particular, previous corresponds to a shortest-path tree rooted at s. □

As mentioned earlier, Algorithm 6 is a *greedy algorithm* in the sense that it follows the locally optimal choice at each stage with the hope of finding a global optimum. The optimality of the computed solution is guaranteed by Lemma 2.2.1. The requirement that the edges must have nonnegative weight follows the facts that Lemma 2.2.1 does not necessarily hold in the case of negative weight edges, and that Algorithm 6 is greedy and never updates its choices.

The complexity of Algorithm 6 greatly depends on the data structure used to represent Q. In fact, it depends on the (worst-case) complexity of adding a new vertex to Q (Line 5), retrieving the vertex u with minimum $\mathsf{dist}[u]$ from Q (Line 7), deleting u from Q (Line 8) and, possibly, updating Q with a new value for one of its element (Line 12), which is known as the "decrease key" method of a priority queue.

Let G be the input graph and assume that G is given as an adjacency list. We denote by n the number of vertices in G and m the number of edges in G. If Q is represented as an array, or a linked list, then retrieving the minimum and deleting it can be done in $\mathcal{O}(|Q|)$ time (linear search through the vertices in Q). Hence, the complexity of Algorithm 6 when Q is an array is $\mathcal{O}(m + n^2) = \mathcal{O}(n^2)$.

Instead of an array, Q can be represented as a priority queue implemented as a *self-balancing binary search tree*, a *binary heap*, a *pairing heap* or a *Fibonacci heap*. The implementation of Algorithm 6 must reflect the choice of the data structure representing Q. In particular, one needs to update Q when $\mathsf{dist}[v]$ changes for some v at Line 12. Using a Fibonacci heap improves the complexity of Algorithm 6 to $\mathcal{O}(m + n \log n)$.

Exercise 2. Prove property ($\star$) on Page 12.

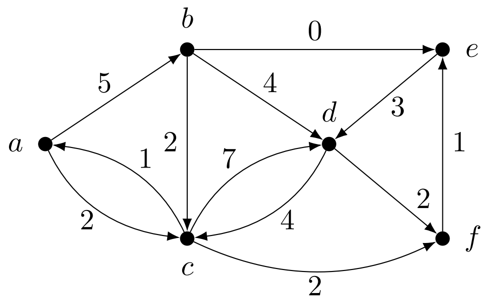
Exercise 3. Suppose we change Line 6 of Algorithm 6 to the following:

"**while** $|\mathsf{Q}| > 1$ **do**"

This changes the while loop to execute $|V(G)| - 1$ times instead of $|V|$ times. Explain whether this new algorithm is correct.

Exercise 4. Given the table previous obtained from Algorithm 6, write an algorithm with complexity at most $\mathcal{O}(|V(G)|)$ that constructs and returns the corresponding shortest-path tree with root s, that is, an arborescence T with root s such that every path in T is a shortest path in G.

Exercise 5. Give a trace of Algorithm 6 with the following graph and vertex a as input. In case of a tie at Line 7, choose the vertex with the lexicographically smallest label.



Bellman-Ford algorithm

Dijkstra's algorithm limitation on nonnegative weight edges comes from the fact that property ($\star$) may fail if the input graph contains negative weight cycles. To overcome this limitation, the greedy approach of *Dijkstra's algorithm* has to be generalized to a more general one, namely, dynamic programming. Unlike *Dijkstra's algorithm*, *Bellman-Ford algorithm* (see Algorithm 7) uses a dynamic programming approach and is able detect cycles of negative length, and thus to handle negative weight edges.

Lemma 2.2.2. *At the end of Algorithm 7, for every $u \in V(G)$, either $\mathsf{dist}[u]$ corresponds to the distance from s to u and* previous *corresponds to a shortest-path tree rooted at s, or a negative weight cycle has been detected. The running time of Algorithm 7 is $\mathcal{O}(|V(G)| \cdot |E(G)|)$.*

Proof. Let C be a negative weight cycle of length k, reachable from s, with vertex set $V(C) = \{x_0 = x_k, x_1, \dots, x_{k-1}\}$. Then, we have that $\sum_{i=1}^{k} w(x_{i-1}, x_i) < 0$. We proceed by contradiction and suppose that Algorithm 7 has not detected C or any other cycle. This implies that the condition at Line 11 always evaluates to *False*, and in particular that $\mathsf{dist}[x_i] \le \mathsf{dist}[x_{i-1}] + w(x_{i-1}, x_i)$, for all $i \in \{1, \dots, k\}$. Then, summing up the inequality for each vertex in C, we get

$$\sum_{i=1}^{k} \mathsf{dist}[x_i] \le \sum_{i=1}^{k} \mathsf{dist}[x_{i-1}] + \sum_{i=1}^{k} w(x_{i-1}, x_i)\,.$$

Since $x_0 = x_k$, it is readily observed that $\sum_{i=1}^{k} \mathsf{dist}[x_i] = \sum_{i=1}^{k} \mathsf{dist}[x_{i-1}]$. Hence, $\sum_{i=1}^{k} w(x_{i-1}, x_i) \ge 0$, contradicting the fact that C is a negative weight cycle. We conclude that if there exists a negative weight cycle in G, then the condition at Line 11 evaluates to *True* and Algorithm 7 detects the presence of such a cycle.

Algorithm 7: Bellman-Ford algorithm

```
Data: a (directed) graph $G$, $s \in V(G)$ and a weight function $w : E(G) \to \mathbb{R}$
Result: a shortest-path tree of $G$ rooted at $s$ (as a table), if $G$ does not contain a negative
        weight cycle
// Initialization
1  foreach $v \in V(G)$ do
2      dist$[v] = +\infty$
3      previous$[v] =$ undefined
4  dist$[s] = 0$
// Iterative edge relaxation
5  for $i \in \{1, \dots, |V(G)| - 1\}$ do
6      foreach $(u, v) \in E(G)$ do
7          if dist$[v] >$ dist$[u] + w(u, v)$ then
8              dist$[v] =$ dist$[u] + w(u, v)$
9              previous$[v] = u$
// Check for negative weight cycles
10 foreach $(u, v) \in E(G)$ do
11     if dist$[v] >$ dist$[u] + w(u, v)$ then
12         print "The input graph contains a negative weight cycle"
13 return previous
```

We claim that after k iterations, for every vertex $v \in V(G)$, $\mathsf{dist}[v]$ is the length of a shortest path from s to v with at most k edges. We proceed by induction. The claim holds for the base case when $k = 0$ since $\mathsf{dist}[s][s] = 0$. Suppose it holds until iteration $k-1$, for $k \geq 1$. Let P be a shortest path from s to v with at most k edges and let (u, v) be the last edge of P. Then, the subpath P' of P from s to u is a shortest path from s to u with at most $k-1$ edges, and by induction hypothesis $\mathsf{dist}[u] = w(P')$ after $k-1$ iterations. At iteration k, the edge (u, v) is also considered and so $\mathsf{dist}[v] \leq \mathsf{dist}[u] + w(u, v) \leq w(P)$. Since P is a shortest path, we in fact $\mathsf{dist}[v] = w(P)$, and thus the claim holds at iteration k. As no path has more than $|V(G)| - 1$ edges, the claim implies that, at the end of the algorithm, $\mathsf{dist}[v]$ corresponds to the distance from s to v. Note that the value $\mathsf{dist}[v]$ is updated only if it is strictly decreased, and thus previous corresponds to a shortest-path tree (as a table) at the end of the algorithm.

The $\mathcal{O}(|V(G)| \cdot |E(G)|)$ running time comes from the two nested loops at Lines 5 and 6. □

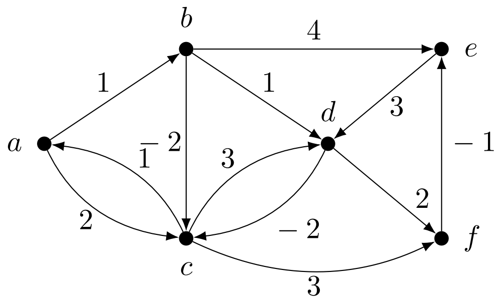
Exercise 6. Write down a (polynomial-time) dynamic programming algorithm that returns the length of a longest path between two given vertices s and t in a directed acyclic graph. Prove its complexity.

Exercise 7. Give a trace of Algorithm 7 with the following graph and vertex a as input. If the algorithm detects a negative weight cycle, describe which one.

The order in which the edges are considered at Lines 6 and 10 is fixed as follows:

$$(a, b) < (a, c) < (b, c) < (b, d) < (b, e) < (c, a) < (c, d) < (c, f) < (d, c) < (d, f) < (e, d) < (f, e)\,.$$

It is suggested to represent the trace as a table where the rows represent the iterations of the loop at Line 5, the columns represent the vertices, and a cell at position (i, u) contains two values: $\mathsf{dist}[u]$ and $\mathsf{previous}[u]$ as at the end of iteration i.



Exercise 8. For simplicity, this exercise only considers undirected simple graphs.

We want to determine the computational complexity of the problem of finding a shortest path P of length at most k between two vertices s and t in a graph (the value k and the vertices s and t are given as input, together with the graph). Recall that a vertex may appear only once in a path. Note that the input graph may contain negative weight cycles. We call this problem General Shortest Path.

a. Show that if the input graph does not contain negative weight cycles, then we can solve the problem in polynomial-time.

b. Let G be a graph and G' be the edge-weighted complete graph with weight function $w : E(G) \to \{-1, 0\}$ such that:

- $V(G') = V(G)$;
- for all $u, v \in V(G')$, if $(u, v) \in E(G)$, then $w(u, v) = -1$, else $w(u, v) = 0$.

Show that G contains a Hamiltonian path if and only if G' contains a path P with weight $w(P) = \sum_{e \in E(P)} w(e) \leq 1 - |V(G)|$.

c. Let G^* be the graph obtained from G' (as defined in b.) by adding two new nonadjacent vertices s and t such that s and t are adjacent to every vertex in $V(G^*) \setminus \{s, t\}$; the weight of the edges incident to s or to t is zero.

(a) Show that G contains a Hamiltonian path if and only if G^* contains an s,t-path P with weight $w(P) = \sum_{e \in E(P)} w(e) \leq 1 - |V(G)|$.

(b) Prove that General Shortest Path is NP-complete.

2.2.2. Between any two vertices

Floyd-Warshall algorithm is able to find a shortest path between any pair of vertices of a graph with real weights on its edges. Of course, one could run Algorithm 7 for every vertex of the input graph G, which would give a time complexity of $\mathcal{O}(|V(G)|^2 \cdot |E(G)|)$. However, as we show next, Floyd-Warshall algorithm (see Algorithm 8) computes these shortest paths in time $\mathcal{O}(|V(G)|^3)$.

Fix a graph G and a linear ordering of its vertices. For simplicity, given a vertex $x \in V(G)$, we denote by $x - 1$ the vertex that precedes x in the ordering and if no such vertex exists, then $x - 1$ is set to $NULL$. Furthermore, we write $V(G)_{\leq x}$ to denote the subset of vertices in $V(G)$ that are not greater than x with respect to the ordering, and if $x = NULL$, then $V(G)_{\leq x} = \emptyset$.

Suppose that we have a function $\mathsf{ShortestPath}(u, v, x)$ that returns a shortest from u to v whose internal vertices (that is, vertices of the path that are neither u nor v) are from the set $V(G)_{\leq x}$. Our goal is to compute a shortest path from u to v for all $u, v \in V(G)$ using any vertex in $V(G)$, that is, a shortest path in G.

For any $u, v, x \in V(G)$, the function $\mathsf{ShortestPath}(u, v, x)$ returns a shortest path P from u to v such that

- either P does not contain x, that is, its internal vertices are from the set $V(G)_{\leq x-1}$,
- or P goes through x and can be seen as a path P_1 from u to x and a path P_2 from x to v, such that the internal vertices of P_1 and P_2 belong to $V(G)_{\leq x-1}$.

In both cases, we see that our knowledge of P only depends on our knowledge of shortest paths with internal vertices in $V(G)_{\leq x-1}$, and thus we can use recursion. The base case is

$$\mathsf{ShortestPath}(u, v, NULL) = \begin{cases} w(u,v) & \text{if } (u,v) \in E(G)\,, \\ +\infty & \text{otherwise}\,. \end{cases}$$

The recursion is as follows:

$$\mathsf{ShortestPath}(u, v, x) = \min \left\{ \begin{array}{l} \mathsf{ShortestPath}(u, v, x-1), \\ \mathsf{ShortestPath}(u, x, x-1) + \mathsf{ShortestPath}(x, v, x-1) \end{array} \right\}.$$

This recursion is the main idea of the Floyd-Warshall algorithm, which iterates over all $x \in V(G)$, with respect to the order on the vertices, and computes $\mathsf{ShortestPath}(u, v, x)$ for all $u, v \in V(G)$. Algorithm 8 is a bottom-up implementation of Floyd-Warshall algorithm.

Lemma 2.2.3. *At the end of Algorithm 8, for every $u, v \in V(G)$, either $\mathsf{dist}[u][v]$ corresponds to the distance from u to v and $\mathsf{previous}$ corresponds to a shortest-path matrix, or a negative weight cycle has been detected. The running time of Algorithm 8 is $\mathcal{O}(|V(G)|^3)$.*

Proof. The $\mathcal{O}(|V(G)|^3)$ time complexity is due to the three nested loops at Lines 11 to 13.

First, suppose that the input graph G does not contains negative weight cycles.
Invariant hypothesis: For each $x \in V(G)$ that has already been considered in the loop at Line 11, the value of $\mathsf{dist}[u][v]$ is the length of a shortest path P from u to v with internal vertices in $V(G)_{\leq x}$ and $\mathsf{previous}[u][v]$ contains the predecessor of v in P.

The invariant is clearly true when there is no such x defined (*i.e.*, before reaching the loop), since it follows from the initialization of the values.

Suppose the invariant holds for some $x-1 \in V(G)$ and consider the case when the loop at Line 11 runs with x. Fix $u, v \in V(G) \setminus \{x\}$. If $\mathsf{dist}[u][v] \leq \mathsf{dist}[u][x] + \mathsf{dist}[x][v]$, then Line 14 evaluates to *False* and $\mathsf{dist}[u][v]$ remains unchanged. So let's consider the case where $\mathsf{dist}[u][v] > \mathsf{dist}[u][x] + \mathsf{dist}[x][v]$. Then there exists a path P_1 from u to x and a path P_2 from x to v such that the sum of their distance is smaller than $\mathsf{dist}[u][v]$. If P_1 and P_2 do not share any vertex other than x, then clearly the union of P_1 and P_2 forms a path P. By hypothesis, both P_1 and P_2 are shortest paths with internal vertices in $V(G)_{\leq x-1}$, and thus P is a shortest path with internal vertices in $V(G)_{\leq x}$. Now, suppose for a contradiction that both P_1 and P_2 have a vertex $y \in V(G)$ as internal vertex. Then,

Algorithm 8: Floyd-Warshall algorithm

```
Data: a (directed) graph $G$ and a weight function $w : E(G) \to \mathbb{R}$
Result: a shortest-path matrix of $G$
// Initialization
1  foreach $u \in V(G)$ do
2      $\mathsf{dist}[u][u] = 0$
3      $\mathsf{previous}[u][u] = u$
4      foreach $v \in V(G) \setminus \{u\}$ do
5          if $(u, v) \in E(G)$ then
6              $\mathsf{dist}[u][v] = w(u, v)$
7              $\mathsf{previous}[u][v] = u$
8          else
9              $\mathsf{dist}[u][v] = +\infty$
10             $\mathsf{previous}[u][v] = \mathit{undefined}$
// Main loop
11 foreach $x \in V(G)$ do
12     foreach $u \in V(G) \setminus \{x\}$ do
13         foreach $v \in V(G) \setminus \{x\}$ do
14             if $\mathsf{dist}[u][v] > \mathsf{dist}[u][x] + \mathsf{dist}[x][v]$ then
15                 $\mathsf{dist}[u][v] = \mathsf{dist}[u][x] + \mathsf{dist}[x][v]$
16                 $\mathsf{previous}[u][v] = \mathsf{previous}[x][v]$
// Check for negative weight cycles
17 foreach $x \in V(G)$ do
18     if $\mathsf{dist}[x][x] < 0$ then
19         print “The input graph contains a negative weight cycle”
20 return previous
```

the union of P_1 and P_2 forms a graph G' containing a path P' from u to v, which shortcuts the maximal closed walk between y and x in G'. Note that this closed walk is made out of cycles that, by assumption, are of nonnegative weight. We obtain that P' is a path from u to v whose internal vertices belong to $V(G)_{\leq x-1}$ and has weight at most the weight of P. Hence, at this stage of the algorithm, we have $\mathsf{dist}[u][v] \leq \mathsf{dist}[u][y] + \mathsf{dist}[y][v] \leq \mathsf{dist}[u][x] + \mathsf{dist}[x][v]$, a contradiction with our assumption. Thus, we conclude that if G does not contain negative weight cycles, then Algorithm 8 computes the shortest paths correctly.

The graph G contains a negative weight cycle C if and only if there exists some $x \in V(C)$ for which $\mathsf{dist}[x][x] < 0$, and thus the algorithm is able to detect it (see Exercise 9). □

Exercise 9. Prove (with more details than in the proof of Lemma 2.2.3) that, at the end of Algorithm 8, there exists a vertex $x \in V(G)$ with $\mathsf{dist}[x][x] < 0$ if and only if there exists a negative weight cycle C in G such that $x \in V(C)$.

2.3. Other examples

2.3.1. $0-1$ Knapsack

In the $0-1$ Knapsack problem, the input consists of a list X of n items such that each item is associated with a size and a value, and a bound S on the size of the knapsack; the goal is to find a subset of items maximizing the sum of their values and such that the sum of their sizes is at most S (they all fit together in the knapsack).

It is easily observed that the following recursive equation returns the maximum value that can fit in the knapsack, where i is position of the item in the list X and R is the remaining space in the knapsack:

$$\mathsf{Knapsack}(i,R) = \begin{cases} \max\{\mathsf{Knapsack}(i-1,R), \mathsf{Knapsack}(i-1,R-s(x_i)) + v(x_i)\} & \text{if } s(x_i) \leq R\,, \\ \mathsf{Knapsack}(i-1,R) & \text{otherwise}\,. \end{cases}$$

We can directly derive Algorithm 9 from this equation.

As mentioned already in this chapter, to analyze the running time of a dynamic programming algorithm, one needs to multiply the number of subproblems by the time to compute each subproblem (ignoring the recursive calls). If we ignore the recursive calls (since, for each pair (i,R) there is only one recursive call before it is stored in $\mathsf{memo}[i][R]$), then each subproblem takes constant time. The initialization step takes $\mathcal{O}(n \cdot S)$ time, and there could be up to $\mathcal{O}(n \cdot S)$ subproblems. Thus, the complexity of Algorithm 9 is $\mathcal{O}(n \cdot S)$. However, this is not a polynomial-time algorithm!

To understand why Algorithm 9 does not run in polynomial time, recall that the running time of an algorithm is expressed in the size of its input. Algorithm 9's input consists of n items, each being associated with a size and a value, and an integer S. We can assume that the size of each item is bounded by S, otherwise such an item does not fit in the backpack, and thus can be ignored from the input. To encode the size of an item in binary, one needs at most $\log S$ bits. Hence, the input has size $\mathcal{O}(n \cdot \log S)$. Since S is exponential in $\log S$, this means that the running time $\mathcal{O}(n \cdot \log S)$ is exponential in the size of the input. For small values of S, this algorithm remains quite efficient.

An algorithm whose complexity is a polynomial in the numeric value of the input (the largest integer present in the input) but not necessarily in the length of the input (the number of bits required to represent the input) is said *pseudo-polynomial*.

Algorithm 9: Recursive DP $0-1$ Knapsack

```
Data: a set of items $X = \{x_1, x_2, \dots, x_n\}$, a size function $s : X \to \mathbb{N}$ and a value function
      $v : X \to \mathbb{R}^+$, an integer $S \in \mathbb{N}$
Result: the maximum value of a set of items with total size at most $S$
// Initialization
1  foreach $i \in \{1, \dots, n\}$ do
2      foreach $j \in \{0, \dots, S\}$ do
3          memo[$i$][$j$] = undefined
4  return Knapsack($n, S$)
// Knapsack function definition
Knapsack($i, R$):
5  if $i \leq 0$ or $R \leq 0$ then
6      return 0
7  if memo[$i$][$R$] $\neq$ undefined then
8      return memo[$i$][$R$]
9  if $s(x_i) \leq R$ then
10     memo[$i$][$R$] = $\max\{$Knapsack($i-1, R$), Knapsack($i-1, R-s(x_i)$) $+ v(x_i)\}$
11 else
12     memo[$i$][$R$] = Knapsack($i-1, R$)
13 return memo[$i$][$R$]
```

Exercise 10. Rewrite Algorithm 9 as a bottom-up dynamic programming algorithm (without recursion).

SOLUTION:

Exercise 11. The SUBSET SUM problem is the problem of, given a set (or multiset) of nonnegative integers and an integer k, deciding whether there exists a nonempty subset whose sum is k. It is known that SUBSET SUM is NP-hard. Give a polynomial-time reduction from SUBSET SUM to (the decision variant of) $0-1$ KNAPSACK, proving that $0-1$ KNAPSACK is NP-hard.

2.3.2. Longest Common Subsequence

In the LONGEST COMMON SUBSEQUENCE problem, we are given two strings S and T, and the goal is to find (one of) their longest common subsequence (LCS), that is, the longest sequence of characters that appear in the same order (but not necessarily contiguously) in both S and T. For example, the LCS of $S =$ FAFMINTINT and $T =$ AFTAMANIAT is FAMNIT. Finding the LCS of strings is a common problem in genomics, when comparing DNA fragments.

We denote by $S[1 \dots i]$ and $R[1 \dots j]$ the prefix of S of length i and the prefix of R of length j, respectively. Let $\mathsf{LCS}(i, j)$ be a function that returns the length of a LCS of $S[1 \dots i]$ and $T[1 \dots j]$. Observe the following:

$$\mathsf{LCS}(i,j) = \begin{cases} 1 + \mathsf{LCS}(i-1, j-1) & \text{if } S[i] = T[j]\,, \\ \max\{\mathsf{LCS}(i-1, j), \mathsf{LCS}(i, j-1)\} & \text{if } S[i] \neq T[j]\,. \end{cases}$$

Algorithm 10: Bottom-up DP $0-1$ Knapsack

```
Data: a set of items $X = \{x_1, x_2, \dots, x_n\}$, $s : X \to \mathbb{N}$ and $v : X \to \mathbb{R}^+$, an integer $S \in \mathbb{N}$
Result: the maximum value of a set of items with total size at most $S$
// Initialization
foreach $i \in \{1, \dots, n\}$ do
    foreach $j \in \{0, \dots, S\}$ do
        memo$[i][j] = 0$
// Main loop
foreach $i \in \{1, \dots, n\}$ do
    foreach $j \in \{0, \dots, S\}$ do
        if $s(x_i) \le j$ then
            memo$[i][j] = \max\{$memo$[i-1][j]$, memo$[i-1][j - s(x_i)] + v(x_i)\}$
        else
            memo$[i][j] =$ memo$[i-1][j]$
return memo$[n][S]$
```

The idea of the algorithm should now be clear: two nested loops, one incrementing i, the other incrementing j, so that we can compute $\mathsf{LCS}(i, j)$ in terms of $\mathsf{LCS}(i-1, j-1)$, $\mathsf{LCS}(i-1, j)$ and $\mathsf{LCS}(i, j-1)$.

Algorithm 11: Bottom-up DP Longest Common Subsequence

```
Data: two strings $S$ and $T$
Result: a longest common subsequence of $S$ and $T$
// Initialization
foreach $i \in \{1, \dots, |S|\}$ do
    LCS$[i][0] = 0$
foreach $j \in \{1, \dots, |T|\}$ do
    LCS$[0][j] = 0$
// Main loop
foreach $i \in \{1, \dots, |S|\}$ do
    foreach $j \in \{1, \dots, |T|\}$ do
        if $S[i] = T[j]$ then
            LCS$[i][j] =$ LCS$[i-1][j-1] + 1$
        else
            LCS$[i][j] = \max\{$LCS$[i-1][j]$, LCS$[i][j-1]\}$
return LCS$[|S|][|T|]$
```

Note that Algorithm 11 does not compute an LCS of S and T, but only the length of such an LCS. However, once the table LCS has been entirely filled with the lengths of the LCS for each pair of prefixes of S and T, one can backtrack the table to get an LCS. This backtracking takes linear time, since the length of the LCS is at most $\min\{|S|, |T|\}$.

Example 2.3.1. Consider the two strings $S =$ FAIMINTINT and $T =$ AFTAMANITA. Then, we obtain the following LCS table:

	A	F	T	A	M	A	N	I	T	A
F	0	①	1	1	1	1	1	1	1	1
A	1	1	1	②	2	2	2	2	2	2
I	1	1	1	2	2	2	2	3	3	3
M	1	1	1	2	③	3	3	3	3	3
I	1	1	1	2	3	3	3	4	4	4
N	1	1	1	2	3	3	④	4	4	4
T	1	1	2	3	3	3	4	4	5	5
I	1	1	3	3	3	3	4	⑤	5	5
N	1	1	3	3	3	3	4	5	5	5
T	1	1	3	3	3	3	4	5	⑥	6

Even though we only computed the length of an LCS (for every prefixes of S and T), we can use the table LCS from Algorithm 11 to get an LCS. Such an LCS is represented with circled number in the table (a number is circled if the letter on the same line/column belongs to the LCS).

2.3.3. Longest Increasing Subsequence

The LONGEST INCREASING SUBSEQUENCE problem is to find the longest increasing subsequence (LIS) of a given sequence. In other words, given a sequence $S = (s_1, s_2, \ldots, s_n)$, the goal is to return the largest subset S' of S such that for all $s_i, s_j \in S'$, if $i < j$, then $s_i < s_j$.

Let's denote by $\mathsf{LIS}(i)$ be the length of an LIS of $S[1, \cdots, i]$, with $S[1, \cdots, i]$ the prefix of S containing the first i characters. It is quite easy to write the corresponding recursive equation:

$$\mathsf{LIS}(i) = \max_{\substack{0 \le j < i \\ S[j] < S[i]}} \mathsf{LIS}(j) + 1 \,.$$

We directly obtain Algorithm 12.

2.3.4. Edit Distance

The EDIT DISTANCE problem is to, given two strings S and T, compute the minimum number of *character edits* (insert a character, delete a character, replace a character) to turn S into T. This is also a common problem in genomics.

Especially when considering the motivation from genomics, it is interesting to give different costs to character edits. Indeed, some mutations are more likely than others (for instance, $C \to G$ is more likely than $C \to A$), and thus unlikely mutations are given a higher weight than common mutations.

Similarly as for LONGUEST COMMON SUBSEQUENCE, we can define the subproblems on prefixes of the input strings. Let's $\mathsf{ED}(i, j)$ be the cost of transforming $S[1 \ldots i]$ into $T[1 \ldots j]$. Then, we can write the following recursive equation:

$$\mathsf{ED}(i,j) = \begin{cases} \mathsf{ED}(i-1, j-1) & \text{if } S[i] = T[j]\,, \\ \min \left\{ \begin{array}{l} \mathsf{ED}(i, j-1) + \text{cost of inserting } T[j]\,, \\ \mathsf{ED}(i-1, j) + \text{cost of deleting } S[i]\,, \\ \mathsf{ED}(i-1, j-1) + \text{cost of replacing } S[i] \to T[j] \end{array} \right\} & \text{otherwise}\,. \end{cases}$$

Algorithm 12: Recursive DP Longest Increasing Subsequence

```
Data: a strings $S$
Result: a longest increasing subsequence of $S$
// Initialization
1  foreach $i \in \{0, \dots, |S|\}$ do
2      memo$[i]$ = undefined
3  return LIS($S$)
// LIS function definition
LIS($S$):
4  $i = |S|$
5  if $i = 0$ then
6      return 0
7  foreach $j \in \{0, \dots, i-1\}$ do
8      if $S[j] < S[i]$ then
9          if memo$[j] \neq$ undefined then
10             memo$[i] = \max\{$memo$[i]$, memo$[j] + 1\}$
11         else
12             memo$[i] = \max\{$memo$[i]$, LIS($S[1 \dots j]$) $+ 1\}$
13 return memo$[i]$
```

Exercise 12. Write down a dynamic programming algorithm (bottom-up or recursive) that solves Edit Distance in time $\mathcal{O}(|S| \cdot |T|)$, with S and T being the two input strings.

Exercise 13. Explain how Longuest Common Subsequence can be reduced in polynomial time to Edit Distance.
Hint: Fix the weights for character edits to specific values.

Chapter 3

Matchings in bipartite graphs

All graphs in this chapter are undirected.

3.1. Basic definitions

Definition 3.1.1. A *matching* in a graph is a set of pairwise disjoint edges.

We focus on *bipartite graphs* and consider, in particular, the following two problems.

MAX CARDINALITY MATCHING
Input: A graph G.
Output: A matching of maximum size in G.

MAX WEIGHT MATCHING
Input: A graph G with a weight function $w : E(G) \to \mathbb{R}$.
Output: A matching of maximum weight in G.

It is clear that the complexity of an algorithm solving the MAX WEIGHT MATCHING problem must be *at least* the complexity solving the MAX CARDINALITY MATCHING problem. Indeed, choosing the weight function $w : E(G) \to c$ for some fixed positive constant c (for example $c = 1$) in the instance of MAX WEIGHT MATCHING would return a matching of maximum cardinality. As we will see, these problems have different complexity, even on bipartite graphs.

Definition 3.1.2. Given a graph G and a matching M in G, a vertex $v \in V(G)$ is *covered* (by M) if $v \in e$ for some $e \in M$; otherwise v is *exposed* (by M). This terminology extends to sets $S \subseteq V(G)$ of vertices: S is covered by M if every vertex in S is covered by M; otherwise S is exposed.

Definition 3.1.3. A matching M in a graph G is a *perfect matching* if $V(G)$ is covered by M.

Observe that a perfect matching, if it exists, is a maximum cardinality matching.

Exercise 14. Let G be a bipartite graph with at least two vertices and bipartition $\{A, B\}$. Show that the bipartition is unique if and only if G is connected.

3.2. Maximum cardinality matching

Finding a matching of maximum size has several applications, even when restricted to bipartite graphs, including assignment problems.

Example 3.2.1. Suppose for instance that you want to assign a set B of tasks with same completion time, say one hour, to a set A of workers. It is possible that some workers may not be able to perform the same tasks. Consider the following question: *Is there an assignment of the tasks in B to workers in A such that all tasks can be done in one hour?* Note that such an assignment implies that all every tasks are processed at the same time (by different workers).

A natural way to model this problem is to represent it as a bipartite graph G with bipartition $\{A, B\}$ where vertices in A represent the workers and vertices in B represent the tasks, and there exists an edge $\{a, b\} \in E(G)$, with $a \in A$ and $b \in B$, if and only if worker a is able to perform task b. Then, the question becomes the following: *Is there a matching $M \subseteq E(G)$ in G covering B?* If such a matching exists, then we readily obtain the desired assignment and all tasks can be done in one hour. Conversely, if there exists an assignment of the tasks in B to workers in A such that all tasks can be done in one hour, then this implies that every task has been assigned to a different worker and thus we obtain a matching M in G directly from the assignment. Thus, the problems are in fact equivalent. We note that since B is covered, the matching M has maximum cardinality.

3.2.1. Structural results

A *vertex cover* in a graph G is a set $S \subseteq V(G)$ of vertices such that every edge in G contains a vertex in S. We denote $\nu(G)$ the maximum cardinality of a matching in G and by $\tau(G)$ the minimum cardinality of a vertex cover in G.

To prove Theorem 3.2.1, we introduce the notion of *alternating path*.

Definition 3.2.1. Let G be a graph and M be a matching in G. A path P is an *M-alternating path*, or simply an *alternating path*, if $E(P) \setminus M$ is a matching.

Theorem 3.2.1 (König, 1931 [22])**.** *If G is a bipartite graph, then $\nu(G) = \tau(G)$.*

Proof. Fix a bipartition $\{A, B\}$ of G. Let M be a matching of maximum size in G. Clearly, for any vertex cover S, no vertex in S can cover two edges in M, otherwise M would not be a matching. Hence, every vertex cover in of G is of size at least $|M|$. It remains to show that there exists a vertex cover of size at most $|M|$.

Let $X_A \subseteq A$ be the set of exposed vertices in A and Y be the set of vertices that belong to an alternating path with an endpoint in X_A (note that such a path can be of length 0, and hence $X_A \subseteq Y$). Observe that if X_A is empty, then A is covered by M, which would imply that A is a vertex cover of size $|A| = |M|$. So let's consider the case when X_A is nonempty. Let $K = (A \setminus Y) \cup (B \cap Y)$.

We first claim that K is a vertex cover. Fix an edge $\{a, b\} \in E(G)$, with $a \in A$ and $b \in B$, and suppose that $a \notin K$ and $b \notin K$. Then $a \in A \cap Y$, and hence a belongs to an alternating path with an endpoint in X_A. Furthermore, $b \in B \setminus Y$, and thus b does not belong to an alternating path with an endpoint in X_A. Note that $a \notin X_A$; otherwise b would belong to an alternating path with an endpoint in X_A. We obtain that a is covered by M and that a belongs to an alternating path P with an endpoint $x \in X_A$. Since a is incident to exactly one edge in M, whether or not the edge ab belongs to M, we can find an alternating path with an endpoint in X_A containing the edge ab, which contradicts the definition of b. Hence, K is a vertex cover of G.

We now argue about the cardinality of K. Fix $u \in K$. Clearly, if $u \in A \cap K = A \setminus Y$, then by definition of Y the vertex u does not belong to X_A and is thus covered by an edge in M. Suppose that there exists a vertex $u \in B \cap K = B \cap Y$, such that u is not covered by M. Then u belongs to an alternating path P with an endpoint $x \in X_A$. Since u and x are not covered by M, then they are the endpoints of P. Therefore, as the first and last edge in P do not belong to M, we obtain that $|E(P) \cap M| = |E(P) \setminus M| - 1$. However, the symmetric difference of M and $E(P)$,

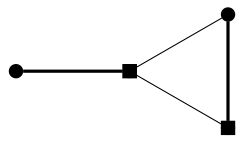


Figure 3.1 The paw. Square vertices form a minimum vertex cover. Thick edges form a maximum matching.

that is, the set $M \triangle E(P) = (M \cup E(P)) \setminus (M \cap E(P))$, is a matching containing one more edge than M, a contradiction with the fact that M is maximum. Hence, every vertex $u \in K$ is covered by M. Finally, suppose that there exists an edge $\{a, b\} \in M$ such that $a, b \in K$. Then $a \in A \setminus Y$ and $b \in B \cap Y$. By definition of Y, b must belong to some alternating path with an endpoint in X_A. In particular, b is the endpoint of an edge in $E(P) \setminus M$. But then, a would also belong to an alternating path with an endpoint in X_A, a contradiction with the fact that $a \in A \setminus Y$. Thus, every vertex in K is covered by M and every edge in M contains at most one vertex from K. In particular, we obtain that $|K| \leq |M|$, which concludes the proof. □

Note that the condition $\nu(G) \leq \tau(G)$ holds for any graph, bipartite or not. This follows the fact that if S is a vertex cover of a graph G, then $V(G) - S$ is an independent set. However, the equality does not hold in general, for instance on the graph $C_3 \cong K_3$, that is, the complete graph on 3 vertices. On the other hand, there exist graphs that are not bipartite but for which the equality holds, that is, graphs G for which $\nu(G) = \tau(G)$. The *paw* is such a graph (see Fig. 3.1).

Theorem 3.2.2 (Hall, 1935 [17])**.** *Let G be a bipartite graph with bipartition $\{A, B\}$. Then G has a matching that covers A if and only if $|X| \leq |N(X)|$, for all $X \subseteq A$.*

Proof. Let M be a matching in G that covers A and fix $X \subseteq A$. Denote by $M_X \subseteq M$ the set of edges in M with an endpoint in X. Then, M_X covers a set $X' \subseteq B$ of vertices in B such that $|X'| = |X|$. Also, $X' \subseteq N(X)$, and thus $|X| = |X'| \leq |N(X)|$.

Now suppose that there is no matching that covers A, that is, $\nu(G) < |A|$. Theorem 3.2.1 implies that $\tau(G) < |A|$. Let $A' \subseteq A$ and $B' \subseteq B$ such that $A' \cup B'$ is a vertex cover and $|A' \cup B'| < |A|$. Clearly, $N(A \setminus A') \subseteq B'$; otherwise some edge would not be covered by $A' \cup B'$. Therefore, $|N(A \setminus A')| \leq |B'| < |A| - |A'| = |A \setminus A'|$. Thus, for $X = A \setminus A'$, we have that $|N(X)| < |X|$. □

Our proof of Theorem 3.2.2 uses Theorem 3.2.1, but a direct proof is also possible. For completeness, we give such a proof.

Direct proof of Theorem 3.2.2. Let G be a bipartite graph with bipartition $\{A, B\}$ such that $|X| \leq |N(X)|$, for all $X \subseteq A$; we say that G satisfies Hall's condition. We claim that G has a matching covering A by induction on $|A|$. If $|A| = 0$ or $|A| = 1$, then the statement holds. Suppose the statement holds for $|A| \leq k$ for some $k \in \mathbb{N}$ and consider the case where $|A| = k + 1$.

- If $|N(X)| > |X|$ for every nonempty proper subset X of A, then choose an edge $\{a, b\} \in E(G)$, with $a \in A$ and $b \in B$, and delete the two vertices a and b from G. The obtained graph satisfies Hall's condition because $|N(X)| - |X|$ has decreased by at most one for all $X \subseteq A \setminus \{a\}$, and thus contains a matching M' covering $A \setminus \{a\}$. Clearly, $M' \cup \{a, b\}$ is a matching covering A in G.

- If there exists a nonempty proper subset X of A with $|N(X)| = |X|$, then, by induction hypothesis, there exists a matching M_X covering X in the graph $G[X \cup N(X)]$. Consider the graph $G' = G[(A \setminus X) \cup (B \setminus N(X))]$. Since G satisfies Hall's condition, for all $Y \subseteq A \setminus X$, we have that $|N_G(Y) \setminus N_G(X)| = |N_G(X \cup Y)| - |N_G(X)| \geq |X \cup Y| - |X| = |Y|$. In particular, $|N_{G'}(Y)| \geq |Y|$, and the induction hypothesis implies that there exists a matching M_Y in G' covering Y. Thus, the matching $M_X \cup M_Y$ is a matching covering A in G.

For the converse direction, let M be a matching in G that covers A and fix $X \subseteq A$. Denote by $M_X \subseteq M$ the set of edges in M with an endpoint in X. Then, M_X covers a set $X' \subseteq B$ of vertices in B such that $|X'| = |X|$. Also, $X' \subseteq N(X)$, and thus $|N(X)| \geq |X'| = |X|$. □

A special case of Theorem 3.2.2 is the following theorem, usually called "Marriage theorem."

Theorem 3.2.3 (Frobenius, 1917 [13]). *Let G be a bipartite graph with bipartition $\{A, B\}$. Then G has a perfect matching in and only if $|A| = |B|$ and $|X| \leq |N(X)|$, for all $X \subseteq A$.*

Exercise 15. Let G be a k-regular bipartite graph, with $k \in \mathbb{N}^*$ (k-regular means that every vertex has degree k). Show that G contains a perfect matching.

3.2.2. Algorithmic results

In fact, the proof of Theorem 3.2.1 already uses a property of alternating paths when the matching M is maximum: if P is an M-alternating path, then P contains at least as many edges in M than outside M; otherwise $M \triangle E(P) = (M \cup E(P)) \setminus (M \cap E(P))$ is a matching of size $|M| + 1$ and M would not be maximum.

Definition 3.2.2. Let G be a graph and M a matching in G. A path P is an *M-augmenting path*, or simply *augmenting path*, if it is alternating and its endpoints are exposed by M.

It follows from the definition of an augmenting path P that every vertex in P which is not an endpoint of P must be covered by $E(P) \cap M$. Besides, $|E(P) \cap M| = |E(P) \setminus M| - 1$.

As we have already noticed in the proof of Theorem 3.2.1, finding an augmenting path in a bipartite graph allows to increase the size of the matching by 1. In fact, this argument holds for any graph.

Theorem 3.2.4 (Petersen, 1891 [33], Berge, 1957 [2]). *Let G be a graph and M a matching in G. Then M is a maximum matching if and only if there are no M-augmenting paths.*

Proof. Clearly, if there exists an augmenting path P, then the symmetric difference $M \triangle E(P)$ is a matching and has cardinality $|M| + 1$, so M is not maximum.

Now, suppose that M is not of maximum size and let M^* be a matching of maximum size. Let H be the subgraph of G induced by the edges in $M \triangle M^*$ and note that, since M and M^* are both matchings, every vertex in H has degree at most 2. Hence, H is a collection of disjoint cycles and paths whose edges alternate between M^* and M. Besides, for each path P in H, either P contains the same number of edges in M^* and in M, or P contains more edges in M^* than in M (the case where P contains more edges in M than in M^* cannot happen, otherwise P would be an M^*-augmenting path, contradicting the maximality of M^*). Cycles and paths with the same number of edges in M^* and M do not contribute to the difference in size between M^* and M. Hence, there exists at least one path P in H containing more edges from M^* than from M, and thus P is an M-augmenting path in G. □

This previous theorem is the core concept of *Hopcroft-Karp algorithm* [18, 21] (see Algorithm 13).

Algorithm 13: Hopcroft-Karp algorithm

```
Data: a bipartite graph $G$
Result: a matching of maximum cardinality
1 $\mathsf{M} = \emptyset$
2 do
3     $\mathsf{H}$ = a maximal set of vertex-disjoint $M$-augmenting paths
4     $\mathsf{M} = \mathsf{M} \triangle E(\mathsf{H})$
5 while $\mathsf{H} = \emptyset$;
6 return $M$
```

The complexity of Algorithm 13 depends on the complexity of finding a maximal set of vertex-disjoint M-augmenting paths.

We describe a technique to find augmenting paths in a bipartite graph G with bipartition $\{A, B\}$. The idea is to orient the edges of G as follows.

Construction 3.2.1. *Let G be a bipartite graph with bipartition $\{A, B\}$ and M be a matching in G. For each edge $\{a, b\} \in E(G)$, with $a \in A$ and $b \in B$, if $\{a, b\} \in M$, the edge is oriented from b towards a; otherwise, the edge is oriented from a towards b.*

Let D be the directed graph obtained from Construction 3.2.1. Clearly, D can be obtain in $\mathcal{O}(|E(G)|)$ time. It is easily observed that there is a bijection between the set of directed paths in D and the set of alternating paths in G. Let $A' = A \setminus V(M)$ and $B' = B \setminus V(M)$, where $V(M)$ corresponds to the vertices covered by M in G. Finding a directed path from A' to B' can be done $\mathcal{O}(|E(G)|)$ time using a *breadth-first search* (BFS) from A' and a *depth-first search* (DFS) from some exposed vertex reached this way. Such a path corresponds to an augmenting path in G, since it is an alternating path that starts and ends in an exposed vertex. Each augmenting path implies that the matching can be increased in size, and hence there can be at most $\frac{|V(G)|}{2}$ augmenting paths. Thus, Algorithm 13 has complexity at most $\mathcal{O}(|V(G)| \cdot |E(G)|)$. In fact, some clever argument on the maximum length of an augmenting path allows us to conclude that the complexity of the algorithm is $\mathcal{O}(\sqrt{|V(G)|} \cdot |E(G)|)$.[1]

Exercise 16. Let G be a bipartite graph with bipartition $\{A, B\}$, $S \subseteq A$ and $T \subseteq B$. Suppose that there exists a matching covering S and a matching covering T. Prove that there exists a matching covering $S \cup T$.

3.3. Maximum weight matching

In this section, we consider that the input bipartite graph G has weights (real numbers) on its edges, and the goal is to find a matching of maximum weight (the weight of the maximum is the sum of the weights of its edges). As for the case of matchings in unweighted graphs, the notion of augmenting path is crucial.

[1]To improve the running time, one has to compute a maximal set of vertex-disjoints augmenting paths of minimum length (in number of edges). This can be done in the same complexity, and one can show that the algorithm terminates after at most $2\sqrt{|V(G)|}$ iterations.

Definition 3.3.1. Let G be a graph and M be a matching in G. We define the "length" function $\ell : E(G) \to \mathbb{R}$ such that for each edge $e \in E(G)$ we have:

$$\ell(e) = \begin{cases} w(e) & \text{if } e \in M\,, \\ -w(e) & \text{otherwise}\,. \end{cases}$$

The length of a graph H is defined as $\ell(H) = \sum_{e \in E(H)} \ell(e)$.

Observe that if P is M-augmenting, then the matching $M \triangle E(P)$ has weight $w(M) - \ell(P)$. Intuitively, the goal is to iteratively update a matching M by finding an augmenting path P with minimum length and consider the matching $M \triangle E(P)$. Note that we want P to have minimum length so that $w(M \triangle E(P)) = w(M) - \ell(P)$ is as large as possible. However, nothing, *a priori*, guarantees that we can reach a maximum weight matching in this way.

Definition 3.3.2. A matching M is *extreme* if M has maximum weight among all matchings of cardinality $|M|$.

The goal is to find an extreme matching of cardinality of every possible cardinality and keep the one with maximum weight.

Lemma 3.3.1. *Let G be a graph and M be an extreme matching in G. If P is an M-augmenting path of minimum length, then $M \triangle E(P)$ is an extreme matching.*

Proof. Let M^* be an extreme matching of cardinality $|M| + 1$. Then, $M \triangle M^*$ is a collection of disjoint cycles and paths (with edges alternatively in M and M^*). Since $|M^*| = |M| + 1$, there must exist a path P with $E(P) \subseteq M \triangle M^*$ containing one more edge in M^* than in M. Necessarily, since the edges of P alternate between M and M^*, the path P starts and ends with edges in M^*. Hence, P is an M-augmenting path in G. Suppose that there exists another M-augmenting path P' with $\ell(P') < \ell(P)$. Then the matching $M \triangle E(P')$ has weight $w(M) - \ell(P') > w(M) - \ell(P) = w(M^*)$, a contradiction with the fact that M^* is extreme. Hence, P must be an M-augmenting path of minimum length. □

Lemma 3.3.1 implies that if we can find a minimum-length augmenting path in polynomial time, then we can find a maximum weight matching in polynomial time. Indeed, it suffices to iteratively find extreme matchings $M_0, M_1, \ldots$ such that $|M_k| = k$ for each k, and to return the one with the largest weight. By definition, such a matching is a matching with maximum weight.

Exercise 17. Give an edge-weighted graph G and three extreme matchings M_1, M_2 and M_3 of G with $|M_i| = i$, for all $i \in \{1, 2, 3\}$, such that $w(M_2) < w(M_1) < w(M_3)$.

If the input graph G is bipartite, then one can find a minimum-length M-augmenting path as follows. First, we construct D using Construction 3.2.1. Here again, consider the sets $A' = A \setminus V(M)$ and $B' = B \setminus V(M)$, where $V(M)$ corresponds to the vertices covered by M in G. Then a minimum-length M-augmenting path in G can be obtained by computing a minimum-length directed path in D from a vertex in A' to a vertex in B'. To argue that such a directed path can be found in polynomial-time, let's prove that D does not contain negative length cycles.

Lemma 3.3.2. *Let M be an extreme matching of a bipartite graph G, D the oriented graph obtained from Construction 3.2.1 given G, and ℓ the length function as defined in Definition 3.3.1. Then D has no negative length cycle.*

Proof. Suppose for a contradiction that D contains a negative length cycle C_D. Let C be the cycle corresponding to C_D in G. We partition the edges of C into two sets: the set $E_M = E(C) \cap M$ and the set $E_{\bar{M}} = E(C) \setminus E_M$. By construction of D, for two vertices $a \in A$ and $b \in B$, if $(a, b) \in E(D)$ then $\{a, b\} \notin M$, and if $(b, a) \in E(D)$ then $\{a, b\} \in M$. This implies that the edges of C alternatively belong to M and to $E(G) \setminus M$. In particular, we get that $E_{\bar{M}}$ is a matching. Note that, by definition of the function ℓ, we have $\ell(C) = \ell(E_M) + \ell(E_{\bar{M}}) = w(E_M) - w(E_{\bar{M}}) < 0$. In particular, $w(E_{\bar{M}}) > w(E_M)$. Let $M' = M \triangle E(C)$ and observe that M' is a matching in G with $|M'| = |M|$. Furthermore, $w(M') = w(M) - w(E_M) + w(E_{\bar{M}}) = w(M) - \ell(C) > w(M)$, a contradiction with the fact that M is extreme. $\square$

Since D does not contain negative length cycles, we can use any algorithm able to find a shortest path in a directed graph without negative weight cycles but possibly with negative weight edges; for instance, the Bellman-Ford algorithm (see Algorithm 7). If G contains n vertices and m edges, this gives us an algorithm with complexity $\mathcal{O}(n^2 \cdot m)$, as we may have to do $\mathcal{O}(n)$ iterations of the Bellman-Ford algorithm.[2] In fact, a refinement of this method is possible so that all the weights are non-negative, and thus use Dijkstra's algorithm (see Algorithm 6) instead of the Bellman-Ford algorithm. It brings down the complexity of finding an augmenting path of minimum length to $\mathcal{O}(n \cdot (m + n \log n))$, which is $\mathcal{O}(n^2 \cdot \log n)$.

Theorem 3.3.1 (Galil, 1986 [15])**.** *Let G be an edge weighted bipartite graph with n vertices and m edges. Then the* MAX WEIGHT MATCHING *problem can be solved in $\mathcal{O}(n^2 \cdot \log n)$ time on G.*

Several other algorithms are known to efficiently solve the MAX WEIGHT MATCHING problem on bipartite graphs. This problem is usually referred to as the ASSIGNMENT problem.[3] Most notably, three decades earlier and with a different technique than ours, Kuhn proved the following theorem.

Theorem 3.3.2 (Kuhn, 1955 [23])**.** *Let G be an edge weighted bipartite graph with n vertices. Then the* MAX WEIGHT MATCHING *problem can be solved in $\mathcal{O}(n^4)$ time on G.*

Kuhn's algorithm is commonly known as the *Hungarian method* (the algorithm was largely based on the earlier works of two Hungarian mathematicians: Dénes König and Jenö Egerváry). Although not as efficient as Theorem 3.3.1, the Hungarian method has been extended by Ford and Fulkerson [11] to solve general maximum flow problems, which we explain in the next chapter. The famous $\mathcal{O}(n^3)$ algorithm of Edmonds and Karp [10], and the one from Tomizawa [36], are in fact adaptations of the $\mathcal{O}(n^4)$ algorithm of Kuhn [23, 24].

3.3.1. Minimum weight perfect matching

MIN WEIGHT PERFECT MATCHING
Input: A graph G with a weight function $w : E(G) \to \mathbb{R}$.
Output: A perfect matching of minimum weight in G, if it exists.

[2] We can avoid recomputing the directed graph D at every iteration by computing a table that answers in $\mathcal{O}(1)$ wether a given edge belongs to the matching. This table can be computed in $\mathcal{O}(m)$ time and using it rather than orienting the edges of G doesn't affect the overall complexity.

[3] Carl Gustav Jacobi had solved the assignment problem in the 19th century and the solution had been published posthumously in 1890 in Latin: `http://www.lix.polytechnique.fr/~ollivier/JACOBI/presentationlEngl.htm`

Proposition 3.3.1. *The* Max Weight Matching *problem and the* Min Weight Perfect Matching *problem are polynomial-time equivalent.*

Proof. Consider a graph G and a weight function $w : E(G) \to \mathbb{R}$, given as input of the Min Weight Perfect Matching problem. We define the function $w' : E(G) \to \mathbb{R}$ such that $w'(e) = K - w(e)$, for every edge $e \in E(G)$, where $K = 1 + \sum_{e \in E(G)} |w(e)|$. Then any maximum weight matching with respect to w' is a maximum cardinality matching. If such a matching is perfect, it is easily seen that it has minimum weight with respect to w.

For the converse direction, consider a graph G and weight function w given as input of the Max Weight Matching problem. Let H be such that $V(H) = \{(v, i) \mid v \in V(G), i \in \{1, 2\}\}$ and $E(H) = \{\{(u, i), (v, i)\} \mid \{u, v\} \in E(G), i \in \{1, 2\}\} \cup \{\{(v, 1), (v, 2)\} \mid v \in V(G)\}$. In other words, H is obtained from two disjoint copies of G by adding an edge between the two copies of each vertex. Clearly, H has a perfect matching. Let $w' : E(H) \to \mathbb{R}$ such that $w'(\{(u, 1), (v, 1)\}) = -w(e)$, for every edge $\{u, v\} \in E(G)$, and $w'(e) = 0$ for every other edge in $E(H)$. Then a minimum weight perfect matching M in H with respect to w' corresponds to a matching of maximum weight in G with respect to w by selecting the edges in $\{\{u, v\} \in E(G) \mid \{(u, 1), (v, 1)\} \in M\}$. □

Note that the proof of Proposition 3.3.1 provides two polynomial-time reductions, and that both of them have a time complexity of at most $\mathcal{O}(n + m)$ if the input graph has n vertices and m edges. Hence, Proposition 3.3.1 and Theorem 3.3.1 imply the following.

Corollary 3.3.1. *Let G be a bipartite graph with n vertices and m edges. Let $w : E(G) \to \mathbb{R}$. Then the* Min Weight Perfect Matching *problem can be solved in $\mathcal{O}(n^2 \cdot \log n)$ on G.*

3.3.2. Stable matching

The Stable Matching problem, also known as the Stable Marriage problem, is the problem of finding a *stable matching* between two equally sized sets A and B of elements given an ordering of preferences for each element. Here, the meaning of "matching" is slightly generalized from the one given in Definition 3.1.1; we now think of a matching as a bijection from the elements of one set to the elements of the other set.

Definition 3.3.3. Let A and B be two equally sized sets of elements and an ordering of preferences for each element. A matching f for $\{A, B\}$ is *stable* if there is no $a \in A$ and $b \in B$ such that a and b both prefer each other over their current assignment in f.

Gale and Shapley proposed an algorithm (see Algorithm 14) for finding a solution to the stable matching problem [14]. Their algorithm, commonly known as Gale-Shapley algorithm, has a linear time complexity in the size of the input (which is quadratic in $|A| = |B|$). Interestingly, there always exists a stable matching (assuming that the sets A and B have the same cardinality).

Theorem 3.3.3 (Gale and Shapley, 1962 [14])**.** *Let A and B be two equally sized sets of elements and an ordering of preferences for each element. Then there exists a perfect matching for $\{A, B\}$ and it can be computed in linear time in the size of the input (which is $\mathcal{O}(|A|^2)$).*

Algorithm 14: Gale-Shapley algorithm

```
Data: Two equally sized sets $A$ and $B$, an ordering of preferences $p(x)$ for each element $x$
Result: a stable matching $f : A \to B$ for $\{A, B\}$
for $a \in A$ do
    $f(a) = NULL$
for $b \in B$ do
    $f^{-1}(b) = NULL$
while $\exists a \in A$ with $f(x) = NULL$ do
    $b = first(p(a))$;                                  // first element w.r.t. $p(a)$
    if $f^{-1}(b) \neq NULL$ then
        if $a >_{p(b)} f^{-1}(b)$ then                  // if $b$ prefers $a$ over $f^{-1}(b)$
            // Match $a$ and $b$
            $f(f^{-1}(b)) = NULL$;
            $f^{-1}(b) = a$;
            $f(a) = b$;
        else
            remove $b$ from $p(a)$
    else
        $f(a) = b$;
    return $f$
```

Chapter 4

Maximum flow and minimum cut

In this chapter, we consider *flows in networks*. The input is a directed graph G with *edge capacities* $c : E(G) \to \mathbb{R}^+$ and two vertices $s, t \in V(G)$, where s is the *source* and t is the *sink*. The goal is to transport as many units as possible, simultaneously, from s to t, respecting the capacity constraints on the edges. A solution to this problem is called a *maximum flow*.

4.1. Definitions

For a vertex set $S \subseteq V(G)$, let $\delta^+(S) = \{(u, v) \in E(G) \mid u \in S, v \notin S\}$ (the set of edges "leaving" S) and $\delta^-(v) = \{(u, v) \in E(G) \mid u \notin S, v \in S\}$ (the set of edges "entering" S). When $S = \{v\}$, we may simply write $\delta^+(v)$ and $\delta^-(v)$.

We assume that the input graph is simple, that is, does not contain loops or multiple edges.

4.1.1. Maximum flow

Definition 4.1.1. Given a directed graph G with edge capacities $c : E(G) \to \mathbb{R}^+$ and two vertices $s, t \in V(G)$, a function $f : E(G) \to \mathbb{R}^+$ is an *s-t-flow* if:

- $0 \le f(e) \le c(e)$ for all $e \in E(G)$ and
- $\sum_{e \in \delta^-(v)} f(e) = \sum_{e \in \delta^+(v)} f(e)$ for all $v \in V(G) \setminus \{s, t\}$ (*flow conservation law*).

The *value* of f, denoted by *value*$(f) = \sum_{e \in \delta^+(s)} f(e) - \sum_{e \in \delta^-(s)} f(e)$, is the net amount of flow leaving s. A *maximum s-t-flow* is an s-t-flow of maximum value.

Note that the value of f is also equal to the net amount of flow entering t. We say that an edge $e \in E(G)$ is *saturated* by f if $f(e) = c(e)$. For simplicity, given an edge $(u, v) \in E(G)$ we write $f(u, v)$ instead of $f((u, v))$ and $c(u, v)$ instead of $c((u, v))$.

The Maximum Flow problem consists in finding a flow of maximum value.

Maximum Flow (Max Flow)

Input:	A directed graph G, $c : E(G) \to \mathbb{R}^+$, and $s, t \in V(G)$.
Output:	An s-t-flow of maximum value.

4.1.2. Minimum cut

Definition 4.1.2. Given a directed graph G with edge capacities $c : E(G) \to \mathbb{R}^+$ and two vertices $s, t \in V(G)$, an *s-t-cut* in G is a set of edges X such that there are no paths from s to t in $G - X$. The *capacity* of X is the sum of the capacities of its edges. A *minimum s-t-cut* in G is an s-t-cut of minimum capacity in G.

4.2. Max Flow–Min Cut duality and more

Lemma 4.2.1. *Let $S \subset V(G)$ such that $s \in S$ and $t \in T = V(G) \setminus S$. Then for any s-t-flow f, the following holds:*

a. *$value(f) = \sum_{e \in \delta^+(S)} f(e) - \sum_{e \in \delta^-(S)} f(e)$, and*

b. *$value(f) \le \sum_{e \in \delta^+(S)} c(e)$.*

Proof. To prove (a), recall that the flow conservation law fold for every vertex $v \in S \setminus \{s, t\}$. Thus,

$$\begin{aligned} value(f) &= \sum_{e \in \delta^+(s)} f(e) - \sum_{e \in \delta^-(s)} f(e) \\ &= \sum_{v \in S} \left(\sum_{e \in \delta^+(v)} f(e) - \sum_{e \in \delta^-(v)} f(e) \right) \\ &= \sum_{e \in \delta^+(S)} f(e) - \sum_{e \in \delta^-(S)} f(e) \,. \end{aligned}$$

Note that (b) directly follows from (a) by using $0 \le f(e) \le c(e)$ for $e \in E(G)$. □

In other words, Lemma 4.2.1 states that the value of a maximum s-t-flow cannot exceed the capacity of a minimum s-t-cut. Our goal is to show that we have equality between the value of a maximum s-t-flow and the capacity of a minimum s-t-cut. First, we define the concepts of residual graph and augmenting paths in the context of flows.

Definition 4.2.1. For a directed graph G, we define the multigraph $\overleftrightarrow{G} = (V(G), E(G) \cup \{\overleftarrow{e} \mid e \in E(G)\})$, that is, for every $e = (u, v) \in E(G)$, the multigraph $\overleftrightarrow{G}$ contains the edges e and $\overleftarrow{e} = (v, u)$. We call $\overleftarrow{e}$ the *reverse edge* of e. Note that if $e = (u, v)$ and $e' = (v, u)$ both belong to $E(G)$, then the edges e' and $\overleftarrow{e} = (v, u)$ are two distinct parallel edges in $\overleftrightarrow{G}$.

Let $c : E(G) \to \mathbb{R}^+$ be the edge capacities of G and f be a flow in G. The *residual capacities* $c_f : E(\overleftrightarrow{G}) \to \mathbb{R}^+$ are such that, for all $e \in E(G)$, we have $c_f(e) = c(e) - f(e)$ and $c_f(\overleftarrow{e}) = f(e)$. The *residual graph* G_f is the graph $(V(G), \{e \in E(\overleftrightarrow{G}) \mid c_f(e) > 0\})$. In other words, G_f is obtained from $\overleftrightarrow{G}$ by removing its edges with residual capacity zero.

An *f-augmenting path*, or simply *augmenting path*, is a path from s to t in G_f.

Given an augmenting path P in G_f, we say that we *augment* f along P by γ by doing the following procedure: for each $e \in E(P)$, if $e \in E(G)$, then we increase $f(e)$ by γ; otherwise, if e is the reverse edge of some edge $e' \in E(G)$, then we decrease $f(e')$ by γ.

Using this notion of augmenting paths, it is now quite natural to find an augmenting path P, augment along the flow along P, and repeat this procedure as long as possible. This is exactly the idea of Algorithm 15 due to Ford and Fulkerson [11]. Note that we consider that the capacities are rational.

The choice of γ in Algorithm 15 guarantees that the flow on each edge is at most its capacity, and since P is a path from s to t, the flow conservation rule is preserved; so f remains a flow at each loop of the algorithm.

Algorithm 15: Ford-Fulkerson algorithm

```
  Data: A directed graph $G$ with edge capacities $c : E(G) \to \mathbb{Q}^+$ and two vertices $s, t \in V(G)$
  Result: A maximum $s$-$t$-flow
  // Initialization
1 Set $f(e) = 0$ for all $e \in E(G)$
  // Main loop
2 P = some $f$-augmenting path
3 while $P$ is not empty do
4     $\gamma = \min_{e \in E(P)} c_f(e)$
5     augment $f$ along $P$ by $\gamma$
6     P = some $f$-augmenting path
7 return $f$
```

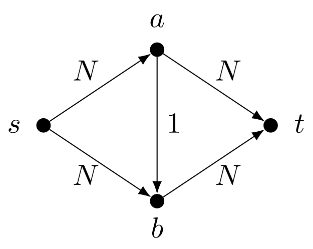


Figure 4.1 Example of a graph on which Algorithm 15 may take $\mathcal{O}(N)$ time to return a maximum flow. The capacities are written next to the edges and $N \in \mathbb{N}$.

To find an f-augmenting path in G, it suffices to find a path from s to t in G_f. However, the value γ may be very small and thus the flow may only slightly increase. Even in the case of integer capacities, if $\gamma = 1$, then we may have an exponential number of loops in the algorithm: the value of the flow depends on the capacities, which are exponential in the number of bits needed to represent them. An example is illustrated in Fig. 4.1: if we choose an augmenting path of length 3 at each iteration (containing the edge (a, b) or its reverse edge (b, a)), then the flow increases by only 1. In fact, if the capacities are real numbers, then there are graphs on which Algorithm 15 does not terminates (even though the flow strictly increases at each loop) [38].

Theorem 4.2.1. *Let G be a directed graph with rational edge capacities $c : E(G) \to \mathbb{Q}^+$ and two vertices $s, t \in V(G)$, instance of Algorithm 15. Then Algorithm 15 terminates.*

Proof. Since all capacities are rational, there exists $K \in \mathbb{N}$ such that $K \cdot c(e)$ is an integer, for every $e \in E(G)$. For instance, we can take K to be the least common multiple of the denominators of the capacities. Then in the main loop, the flow value increases by at least $1/K$. Since the flow value cannot exceed $\sum_{e \in \delta^+(s)} c(e)$, the algorithm must terminate. □

To show the correctness of Algorithm 15, we prove the following.

Theorem 4.2.2. *An s-t-flow is maximum if and only if there are no f-augmenting paths.*

Proof. Clearly, if there exist an f-augmenting path, then we can compute a flow of strictly greater value (as in Algorithm 15), so f is not maximum.

If there is no f-augmenting path, then t is not reachable from s in the residual graph G_f. Let S be the set of vertices reachable from s in G_f. Consider an edge $e \in \delta^+_{\overleftrightarrow{G}}(S)$ and notice that $c_f(e) = 0$, as otherwise e would exists in G_f, which could contradict the definition of S. Hence,

- either $e \in E(G)$ and $f(e) = c(e)$, that is, e is saturated by f,
- or $e \notin E(G)$ and thus $e = \overleftarrow{g}$ is the reversed edge of some edge $g \in E(G)$, and thus $c_f(e) = f(g) = 0$.

Therefore, by definition of G_f, we get that every edge in $\delta^+_G(S)$ is saturated and every edge in $\delta^-_G(S)$ does not transit flow. Equivalently we have $f(e) = c(e)$ for all $e \in \delta^+_G(S)$ and $f(e) = 0$ for all $e \in \delta^-_G(S)$. By Lemma 4.2.1 (a), we have that $value(f) = \sum_{e\in\delta^+(S)} f(e) - \sum_{e\in\delta^-(S)} f(e) = \sum_{e\in\delta^+(S)} c(e)$, which by Lemma 4.2.1 (b) implies that f is a maximum flow. □

In particular, for any maximum s-t-flow we have an s-t-cut whose capacity equals the value of the flow. Together with Lemma 4.2.1 (b), we obtain the central result of network flow theory, commonly referred to as the *Max-Flow–Min-Cut Theorem*:

Theorem 4.2.3 (Ford and Fulkerson, 1956 [11], Dantzig and Fulkerson, 1956 [6])**.** *In a directed graph with capacities on its edges, the maximum value of an s-t-flow equals the minimum capacity of an s-t-cut.*

Note that if all the capacities are integers, then γ in Algorithm 15 is an integer and thus the returned flow is integral. The next result is sometimes called the *Integral Flow Theorem.*

Corollary 4.2.1 (Dantzig and Fulkerson, 1956 [6])**.** *If the capacities of a graph are integers, then there exists an integral maximum flow.*

Exercise 18. Let G be a directed graph, and S and T be two disjoint nonempty subsets of vertices of G. Explain how to compute in polynomial time a cut X with minimum cardinality in G such that no edge in X has both endpoints in S or both endpoints in T.

4.3. Edmonds-Karp algorithm

We noticed that Algorithm 15 does not necessarily run in polynomial-time since its complexity depends on the capacities. Instead of choosing an arbitrary augmenting path in the algorithm, it may be a good idea to look for a shortest one, *i.e.*, an augmenting path with a minimum number of edges. In fact, if we choose a shortest augmenting path, then the number of iterations is at most $|V(G)| \cdot |E(G)|$. This was shown by Dinic in 1970 [9] who developed the first polynomial-time algorithm for the MAX FLOW problem. A variant of this algorithm is Edmonds-Karp algorithm (see Algorithm 16), published in 1972 [10].

Proposition 4.3.1. *Let G be a directed graph, $s,t \in V(G)$ and $e \in E(G)$ such that e belongs to some shortest path P from s to t in G. Then there is no shortest path from s to t in $G + \overleftarrow{e}$ containing the reverse edge $\overleftarrow{e}$.*

Proof. Suppose that there exists a shortest path Q in $G + \overleftarrow{e}$ containing $\overleftarrow{e}$. Then the graph $E(P) \cup E(Q) \setminus \{e, \overleftarrow{e}\}$ contains a path P' from s to t which is strictly shorter than P, a contradiction. □

Algorithm 16: Edmonds-Karp algorithm

```
  Data: A directed graph $G$ with edge capacities $c : E(G) \to \mathbb{R}^+$ and two vertices $s, t \in V(G)$
  Result: An $s$-$t$-flow of maximum value
  // Initialization
1 Set $f(e) = 0$ for all $e \in E(G)$
  // Main loop
2 P = a shortest $f$-augmenting path
3 while $P$ is not empty do
4     $\gamma = \min_{e \in E(P)} c_f(e)$
5     augment $f$ along $P$ by $\gamma$
6     P = a shortest $f$-augmenting path
7 return $f$
```

Theorem 4.3.1. *The number of iterations of Algorithm 16 is at most $|V(G)| \cdot |E(G)|$.*

Proof. After augmenting a flow f along a shortest path P from s to t, obtaining a flow f', the residual graph $G_{f'}$ is a subgraph of the graph $H_{f'} = (V(G), E(G_f) \cup \{\overleftarrow{e} \mid e \in E(P)\})$. By Proposition 4.3.1, the shortest paths from s to t in $H_{f'}$ have length at least the length of P. Furthermore, Proposition 4.3.1 also guarantees that if there exists a shortest path Q from s to t of length $\ell(P)$ in $G_{f'}$, then Q exists in G_f; this follows from the fact that no reverse edge $\overleftarrow{e}$ of some $e \in E(P)$ can belong to a shortest path in $G_{f'}$. Since P contains at least one edge that belongs to G_f but not to $G_{f'}$ (all edges $e \in E(P)$ for which the value $c_f(e) = \gamma$, and thus $c_{f'}(e) = 0$), P does not exists in $G_{f'}$. Hence, the set of shortest augmenting paths in $G_{f'}$ is a proper subset of the set of augmenting paths in G_f. As there are at most $|E(G)|$ edge-disjoint augmenting paths and at most $|V(G)|$ possible lengths for a path, Algorithm 16 iterates at most $|V(G)| \cdot |E(G)|$ times. □

We point out that, with a different proof, one can show that the number of iterations is in fact bounded by $\frac{1}{4}|V(G)| \cdot |E(G)|$; see [10] for details.

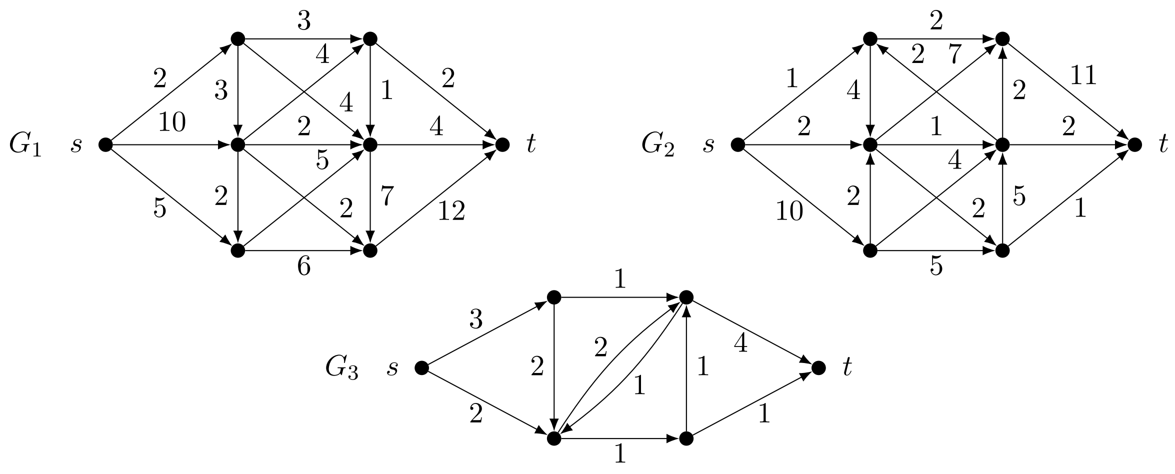
Theorem 4.3.2 (Edmonds and Karp, 1972 [10])**.** *The* Maximum Flow *problem can be solved in time $\mathcal{O}(nm^2)$ on graphs with n vertices and m edges.*

Proof. To find an augmenting path, we can perform a breadth-first search (BFS) from s, which takes $\mathcal{O}(m)$ time. Together with Theorem 4.3.1 which bounds the number of iterations to nm, we get the claimed complexity. □

Independently, Dinic proposed a $\mathcal{O}(n^2m)$ algorithm in 1970 [9]. Then, in 1974, Karzanov showed that Edmonds-Karp algorithm can be further improved: as long as the length of an augmenting path does not increase, one can use the data-structure of the previous BFS to find the next augmenting path faster.

Theorem 4.3.3 (Karzanov, 1974 [20])**.** *The* Maximum Flow *problem can be solved in time $\mathcal{O}(n^3)$ on graphs with n vertices.*

Exercise 19. Determine, using Algorithm 16, a maximum s-t-flow and a minimum s-t-cut in the following graphs G_1, G_2 and G_3 (the numbers on the edges are the capacities).



4.4. Minimum-cost flow

Now, we consider a generalization of the Maximum Flow problem, known as the Minimum-Cost Flow problem. In this variant, an instance consists in an input graph G with edge capacities $c : E(G) \to \mathbb{R}$, a cost function $k : E(G) \to \mathbb{R}^+$, and two vertices $s, t \in V(G)$. The *cost* of a flow f in G is $cost(f) = \sum_{e \in E(G)} k(e) \cdot f(e)$. The goal is to find a maximum s-t-flow of minimum cost.

Minimum-Cost Flow (Min-Cost Flow)
Input: A directed graph G, $c : E(G) \to \mathbb{R}^+$, $k : E(G) \to \mathbb{R}^+$ and $s, t \in V(G)$.
Output: An s-t-flow with minimum cost among all s-t-flows of maximum value.

Our goal is to adapt Algorithm 15 to solve this problem.

Definition 4.4.1. An s-t-flow f is *extreme* if f has minimum cost among all s-t-flows with the same value as f.

So an extreme flow does not necessarily have maximum value. The parallel with the notion of extreme matching (see Definition 3.3.2)is immediate.

Definition 4.4.2. Let G be a directed graph with edge capacities $c : E(G) \to \mathbb{R}$ and f be a flow in G. Consider the residual graph G_f, with respect to G, f and c. We define the "length" function $\ell : E(G_f) \to \mathbb{R}$ as follows:

$$\ell(e) = \begin{cases} k(e) & \text{if } e \in E(G)\,, \\ -k(e) & \text{otherwise (if } e \text{ is a reverse edge)}\,. \end{cases}$$

The length of a graph H is defined as $\ell(H) = \sum_{e \in E(H)} \ell(e)$.

Since the edges of G_f can have negative length, there could be negative length cycles in G_f. As the following result shows, this is not the case if f is an extreme matching. We refer the reader to [34] for a proof of Proposition 4.4.1.

Proposition 4.4.1. *An s-t-flow f in a directed graph G is extreme if and only if the residual graph G_f has no cycles of negative length (with respect to ℓ).*

In the same flavor as Lemma 3.3.1, we can now show the following result.

Proposition 4.4.2. *Let f be an extreme s-t-flow in G. Let f' be the flow obtained from f in G_f after augmenting f along an augmenting path P with minimum length with respect to ℓ. Then f' is an extreme flow.*

Proof. Suppose that $G_{f'}$ has a cycle C of negative length with respect to ℓ. Since f is an extreme s-t-flow, Proposition 4.4.1 implies that C does not exist in G_f. Hence, there is an edge $(x, y) \in E(C)$ such that $(x, y) \notin E(G_f)$, which implies that $(y, x) \in E(P)$. But then there exists an augmenting path P' in G_f with $\ell(P') < \ell(P)$, a contradiction with the fact that P has minimum length. (see Exercise 20). □

This implies that the Min-Cost Flow problem can be solved by choosing a shortest augmenting path (a shortest path from s to t in G_f) with respect to ℓ in Algorithm 15 (we assume that the capacities are rational). The first flow f_0, with value zero, is trivially a min-cost flow. Hence, all further flows $f_1, f_2, \ldots$ are extreme flows too, by Proposition 4.4.2. The last flow obtained this way is a flow of maximum value which is also extreme, and thus of minimum cost among all maximum flows.

To find a shortest path from s to t in the residual graph, we can use the Bellman-Ford algorithm (see Algorithm 7)since the residual graph does not contain negative-length cycles, as stated in Proposition 4.4.1.

The running time of such an algorithm is $\mathcal{O}(M(m + n \log n))$ where M is the value of the maximum flow (assuming that the capacities are integers), and thus is not polynomial but pseudo-polynomial. Nonetheless, Orlin proposed a polynomial-time algorithm for the Min-Cost Flow problem.

Theorem 4.4.1 (Orlin, 1993 [30]). *The Minimum-Cost Flow problem can be solved in time $\mathcal{O}(m \log n(m + n \log n))$ on graphs with n vertices and m edges.*

Exercise 20. In the proof of Proposition 4.4.2, we claim that if $G_{f'}$ contains a cycle C of negative length, then there exists an augmenting path P' of length $\ell(P') < \ell(P)$, resulting in a contradiction. Explain with more details than in the proof why such a path P' exists, assuming that C exists.

4.5. Using flows to solve other problems

4.5.1. Maximum cardinality matching in bipartite graphs

In Chapter 3 we have studied the Maximum Cardinality Matching in bipartite graphs and learned that there exists $\mathcal{O}(\sqrt{n}m)$ algorithms to solve the problem, where n is the number of vertices and m the number of edges of the input graph. In fact, the Maximum Cardinality Matching in bipartite graphs can also be solved using flows.

Construction 4.5.1. *Given a bipartite graph G with bipartition $\{A, B\}$, construct the directed graph D obtained from G by:*

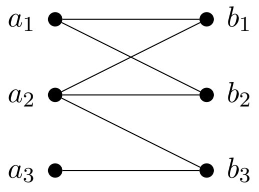
- *orienting each edge $\{a, b\} \in E(G)$, $a \in A$ and $b \in B$, from a to b;*
- *creating two new vertices s and t;*

- *adding edges (s, a) for all $a \in A$;*
- *adding edges (b, t) for all $b \in B$.*

That is, $D = (V(G) \cup \{s, t\}, \{(a, b) \mid \{a, b\} \in E(G), a \in A, b \in B\} \cup \{(s, a) \mid a \in A\} \cup \{(b, t) \mid b \in B\})$. Then, set the capacities $c : E(D) \to 1$.

Exercise 21. Show that, given a bipartite graph G, the graph D obtained from Construction 4.5.1 contains a flow of value k if and only if G contains a matching of size k.

Exercise 22. Find a matching of maximum size in the following graph using Algorithm 16 and the aforementioned construction, starting with the augmenting path $s \to a_2 \to b_3 \to t$. Clearly describe the augmenting paths, the value of γ and the value of the flow at each iteration.



4.5.2. Assignment problem

We further generalize the Assignment problem by allowing that several workers can contribute to a same task, as follows.

The input consists of n jobs with processing time $t_1, \dots, t_n \in \mathbb{R}^+$ and, for each $i \in \{1, \dots, n\}$, a nonempty subset $W_i \subseteq \{w_1, \dots, w_m\}$ of workers that can contribute to job i. The goal is to find numbers $x_{i,j} \in \mathbb{R}^+$ corresponding to the time worker j works on job i such that all jobs are finished, that is, $\sum_{w_j \in W_i} x_{i,j} = t_i$, for $i \in \{1, \dots, n\}$.

Construction 4.5.2. *Given an instance of the Assignment problem, create a directed graph G with vertex set $\{v_1, \dots, v_n\} \cup \{w_1, \dots, w_m\} \cup \{s, t\}$ such that every $v_i \in V(G)$ corresponds to the job i, every $w_i \in V(G)$ corresponds to the worker w_i. Add a directed edge from w_j to v_i if and only if $w_j \in W_i$, that is, if worker w_j can contribute to task i. Then add an edge from s to each w_j, and and edge from each v_i to t. Finally, for each edge (v_i, t), set the capacity $c(v_i, t) = t_i$ (the completion time of job i), and the capacity of every other edge to $+\infty$ (or to any value $K \geq \sum_{i \in \{1,\dots,n\}} t_i$).*

A maximum s-t-flow in G saturates the edges (v_i, t), and thus corresponds to an assignment: we have $\sum_{w_j \in W_i} f(w_j, v_i) = t_i$, and since f respects the flow conservation law, then we can set $x_{i,j} = f(w_j, v_i)$. Conversely, an assignment readily gives us a flow in D, which is maximum since $\sum_{w_j \in W_i} x_{i,j} = \sum_{e \in \delta^-(w_j)} f(e) = \sum_{e \in \delta^+(w_j)} f(e) = t_i = c(v_i, t)$.

Notice that we can set a cost for each unit of time worker w_j works on job v_i by setting a cost on the edge (w_j, v_i) and solve the Min-Cost Flow problem on the obtained graph.

Exercise 23. Suppose that the instance of the Assignment problem contains time constraints of the type:

a. maximum working time of worker w_j (specific to each worker);

b. maximum working time of worker w_j on job v_i (specific to each pair worker-job);

c. maximum sum of the working time of each worker.

Explain what to change in Construction 4.5.2 so that a maximum s-t-flow in G corresponds to a solution of the ASSIGNMENT problem in which those new constraints are also respected?

4.5.3. Transportation problem

Suppose that there are ℓ factories producing a same product and k customers that use the product. Each month, factory i can produce s_i tons of the product. Customer j needs d_j tons of the product each month (the demand). However, there are constraints on how much product one factory can send to a client each month. In particular, at most $c_{i,j}$ tons can be sent from factory i to client j each month. The question corresponding to the TRANSPORTATION problem is: can the needs of the customers be fulfilled?

Construction 4.5.3. *Given an instance of the Transportation problem, create a directed graph D where factory i is associated with a vertex f_i, customer j is associated with a vertex b_j, and D contains two other vertices s and t. The edges in D are from s to every f_i, from each f_i to every b_j, and from each b_j to t. Then set the capacities of the edges as follows:*

- $c(s, f_i) = s_i$ *(amount produced by factory i each month);*
- $c(fi, b_j) = c_{i,j}$ *(maximum amount that can be sent from factory i to customer j);*
- $c(b_j, t) = d_j$ *(demand of customer j).*

Since there cannot exists an s-t-flow in G of value larger than $d_1 + d_2 + \cdots + d_n$, the problem can be solved with a maximum flow algorithm. If there exists a flow of value $d_1 + d_2 + \cdots + d_n$, then the flow on each edge (f_i, b_i) gives the amount that should be transported each month from factory i to customer j. The flow on arc (s, f_i) gives the amount to be produced each month by factory i.

Clearly, this problem can be further generalized by assigning a cost $w_{i,j}$ to each unit of product that is sent from factory f_i to customer b_j. This variant, called MINIMUM-COST TRANSPORTATION problem can then be solved using algorithms for the MIN-COST FLOW problem.

Exercise 24. Solve the transportation problem, using Algorithm 15, for the following data:

$c_{i,j}$	$j=1$	$j=2$	$j=3$
$i=1$	0	8	2
$i=2$	8	3	3
$i=3$	2	5	1

	$i=1$	$i=2$	$i=3$
s_i	15	11	8

	$j=1$	$j=2$	$j=3$
d_j	7	12	3

Clearly describe the augmenting paths, the value of γ and the value of the flow at each iteration.

Exercise 25. Using the same data as in Exercise 24 and the following costs, solve the mininimum-cost transportation problem:

$w_{i,j}$	$j=1$	$j=2$	$j=3$
$i=1$	3	1	4
$i=2$	2	3	2
$i=3$	3	1	4

Clearly describe the augmenting paths, the value of γ and the value of the flow at each iteration.

4.5.4. Minimum vertex cut

Now, given a directed graph G and two nonadjacent vertices $x, y \in V(G)$, we are interested in finding a set $S \subseteq V(G) \setminus \{x, y\}$ of minimum size such that there is no path from x to y in $G - S$. The set S is called a *minimum x-y-vertex cut.*

Two paths P_1 and P_2 are *edge-disjoint* if $E(P_1) \cap E(P_2) = \emptyset$; if P_1 and P_2 and *internally vertex-disjoint* if the sets of internal vertices of P_1 and P_2 are disjoint.

Lemma 4.5.1. *Let x and y be two nonadjacent vertices of a directed graph G. Then the maximum number of edge-disjoint paths from x to y in G is equal to the number of edges in a minimum x-y-cut in G.*

Proof. See Exercise 26. □

Lemma 4.5.2. *Let x and y be two nonadjacent vertices of a directed graph G. Then the maximum number of internally vertex-disjoint paths from x to y in G is equal to the number of vertices in a minimum x-y-vertex cut in G.*

Proof. We construct a new directed graph G' as follows:

- for each $v \in V(G) \setminus \{x, y\}$, we create two vertices v^- and v^+ and the edge (v^-, v^+);
- for each edge $(u, v) \in E(G)$, $u, v \notin \{x, y\}$, we create an edge (u^+, v^-).

To each path from x^- (or x^+) to y^+ (or y^-) in G', there corresponds a path from x to y in G, obtained by contracting all edges of the type (v^-, v^+). Conversely, by definition of G', to each path from x to y in G there corresponds a path from x^- to y^+ in G'. Furthermore, two paths from x to y in G' are edge-disjoint if and only if the corresonding paths from x to y in G are internally vertex-disjoint. It therefore follows that the maximum number of edge-disjoint paths from x to y in G' is equal to the maximum number of internally vertex-disjoint paths from x to y in G. Similarly, the minimum number of edges in G' whose deletion destroys all paths from x to y is equal to the minimum number of vertices in G whose deletion destroys all paths from x to y (see Exercise 27). The statement now follows directly from Lemma 4.5.1. □

In fact, we just proved *Menger's theorem.*

Theorem 4.5.1 (Menger, 1927 [27]). *Let G be a directed graph and x and y two nonadjacent vertices in G. Then the maximum number of pairwise internally vertex-disjoint paths from x to y is equal to the size of the minimum x-y-vertex cut.*

As a corollary of Lemma 4.5.2, we obtain that we can compute a minimum x-y-vertex cut in polynomial time, using a maximum-flow algorithm.

Corollary 4.5.1. *Let x and y be two nonadjacent vertices of a directed graph G. Then one can compute a minimum x-y-vertex cut in time $\mathcal{O}(|V(G)|^3)$.*

Exercise 26. Prove Lemma 4.5.1.

Exercise 27. Show that, in the proof of Lemma 4.5.2, the minimum number of edges in G' whose deletion destroys all paths from x to y is equal to the minimum number of vertices in G whose deletion destroys all paths from x to y.

Chapter 5

Linear programming

Linear programming is a generic algebraic method for solving *linear programs*. A linear program is a linear *objective function* subject to linear *constraints* (equalities or inequalities) on *decision variables*. Such programs can model many different real world optimization problems.

In this chapter, we see how to model, solve and prove the optimality of solutions of linear programs.[1]

5.1. The graphical method

Let's start with an example of real world optimization problem that we would like to solve with linear programming.

Example 5.1.1 (Gepetto's problem)**.** Gepetto is a businessman: he wants to maximize his profits. In his store, Gepetto sells two kinds of toys: trains and soldiers. Two kinds of processes are needed: carving and painting. These two processes are done in different parts of the store, and different skills are needed for both.

	soldiers	trains	max. time/week
carving	1h	1h	80h
painting	2h	1h	100h
max. number / week	≤ 40	∞	
profit	3€/piece	2€/piece	

We define the decision variables x_1 = number of soldiers and x_2 = number of trains, that can be produced in a week. The goal is to maximize the profit while respecting every constraint.

For instance $(x_1, x_2) = (40, 20)$ is a feasible solution, that is, all constraints are respected with these values. The corresponding profit is $40 \times 3€ + 20 \times 2€ = 160€$.

We can rewrite our problem as follows.

Trains and soldiers

$$\begin{array}{llrl}
 & \max z = 3x_1 + 2x_2 & & \text{(profit)} \\
\text{Subject to} & & & \\
(1) & x_1 + x_2 \leq 80 & & \text{(carving time)} \\
(2) & 2x_1 + x_2 \leq 100 & & \text{(painting time)} \\
(3) & x_1 \leq 40 & & \text{(max. number of soldiers)} \\
 & x_1, x_2 \geq 0 & & \text{(positivity)}
\end{array}$$

[1]These notes are inspired by the following lecture notes: https://math.mit.edu/~goemans/18310S15/lpnotes310.pdf (although some terms do not have the same meaning, *e.g.* "standard form" and "canonical form").

Notice the nonnegativity constraint on x_1 and x_2 that we included in this formulation. It is *very important*, even though implicit in the problem.

Since we have only two variables, we can graphically solve the problem.

Each constraints gives a linear function:

- carving time constraint: $x_2 = -x_1 + 80$;
- painting time constraint: $x_2 = -2x_1 + 100$;
- max. number of soldiers constraint: $x_1 = 40$;
- nonnegativity constraints: $x_1 = 0$ and $x_2 = 0$.

These functions define a polytope of feasible solutions. We can graphically check that the optimal solution is $(x_1, x_2) = (20, 60)$, which gives a profit of $z = 3 \times 20 + 2 \times 60 = 180$€. The optimality comes from the fact that if we want to increase the profit, then we leave the feasible region. However, this graphical technique only works in 2 dimensions (maybe 3, but it would be cumbersome).

Notice that in our example, the optimal solution is integral. However, in general, the optimal solutions are not necessarily integral. Also, the decision variables can take any real value that would satisfy the inequalities (constraints).

5.2. Convexity, polytope and vertices

Linear programming deals with maximizing or minimizing a linear objective function of finitely many variables subject to finitely many linear inequalities.[2] So the set of feasible solutions is the intersection of finitely many half-spaces (corresponding to the inequalities). Such a set is called a polyhedron.

Definition 5.2.1. A *polyhedron* in $\mathbb{R}^n$ is a set of type $K = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ for some matrix $A \in \mathbb{R}^{m \times n}$ and some vector $b \in \mathbb{R}^m$. A bounded polyhedron is also called a *polytope*.

Definition 5.2.2. A *region* K in $\mathbb{R}^n$ (that is, a set $K \subseteq \mathbb{R}^n$), is *convex* if for any two point $x, y \in K$, and for any $\lambda \in \mathbb{R}$ such that $0 \leq \lambda \leq 1$, it holds that $\lambda x + (1-\lambda)y \in K$ (the convex combination of x and y is always in K).

If $A, B \subseteq \mathbb{R}^n$ are convex, then $A \cap B$ is convex. To see this, fix $x, y \in A \cap B$ and let $\lambda \in \mathbb{R}$ such that $0 \leq \lambda \leq 1$. Clearly, $x, y \in A$, and since A is convex then the point $(\lambda x + (1-\lambda)y) \in A$. Similarly, $x, y \in B$, and since B is convex then the point $(\lambda x + (1-\lambda)y) \in B$. Thus, $(\lambda x + (1-\lambda)y) \in A \cap B$, which implies that $A \cap B$ is convex. From this observation, we can define the *convex hull* of a set $V \subset \mathbb{R}^n$ as

$$conv\langle V \rangle = \bigcap_{\substack{K \subseteq R^n \\ K \text{ is convex} \\ V \subseteq K}} K\,.$$

We can see the convex hull of V as the smallest convex set K containing V.

[2]If a constraint is an equality, then we can rewrite as two inequalities.

Theorem 5.2.1 (Minkowski, 1968 [28]). *A set P is a polytope if and only if it is the convex hull of a finite set of points.*

The *vertices* of the polytope are the corner points formed by the intersection of edges or faces of the polytope. There can be an exponential gap between the number of vertices in a polytope and the number of inequalities defining it. For example, consider the polytope $\{x \in \mathbb{R}^n \mid 0 \leq x_i \leq 1, \forall i = 1, 2, \ldots, n\} = [0, 1]^n$ (the *n-dimensional unit cube*). The vertices of the polytope are exactly the binary vectors $\{0, 1\}^n$ and there are 2^n of them. However, the polytope is defined with only $2n$ inequalities. We can also consider the polytope defined as the the convex hull of the points in the middle of the faces of the n-dimensional unit cube. This polytope has only $2n$ vertices (number of faces of the n-dimensional unit cube), but there exists a mapping between the faces of the polytope and the vertices of the cube (each face has a normal vector that goes through a *unique* vertex of the cube). Hence, we need 2^n inequalities to describe its convex hull.

Claim 5.2.1. *Every linear program has an optimal solution that is a vertex of the feasible solution set (if such set is not empty).*

5.3. Representation of linear programs

A linear program can take many different forms. First, we have a *minimization* or a *maximization* problem, depending on whether the objective function is to be minimized or maximized. The *constraints* can either be inequalities ($\leq$ or $\geq$) or equalities. Some variables might be unrestricted in sign while others might be restricted to be nonnegative. In this lecture notes, we consider two forms of linear programs: the *canonical form* (which we use in Section 5.4) and the *standard form* (which we use in Section 5.5).

5.3.1. Canonical form

Definition 5.3.1. A linear program is said to be in *canonical form* if:

- it is a maximization program,
- there are only equalities (no inequalities) and
- all variables are restricted to be nonnegative.

For instance, the following linear program is in canonical form.

LP in canonical form

$$
\begin{array}{rl}
 & \max z = c_0 + c_1x_1 + c_2x_2 + \cdots + c_nx_n \\
\text{Subject to} & \\
(1) & a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = b_1 \\
\vdots & \vdots \qquad \vdots \qquad \vdots \qquad \vdots \\
(i) & a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,n}x_n = b_i \\
\vdots & \vdots \qquad \vdots \qquad \vdots \qquad \vdots \\
(m) & a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n = b_m \\
 & x_i \geq 0
\end{array}
$$

In matrix form, a linear program in canonical form can be written as follows.

LP in matrix canonical form

$$
\begin{array}{rl}
 & \max z = c^\intercal x \\
\text{Subject to} & \\
 & Ax = b \\
 & x \geq 0
\end{array}
$$

In this formulation, $b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$, $c = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}$ and $x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$ are columns vectors, $c^\intercal$ denotes the transpose of the vector c and $A = [a_{i,j}]$ is the $m \times n$ matrix whose i, j-element is $a_{i,j}$.

Any linear program can be transformed into an equivalent linear program in canonical form.

From minimization to maximization:
Minimizing the function $z = c^\intercal \cdot x$ is equivalent to maximizing $z = -c^\intercal \cdot x$.

From "$\leq$" to "$=$":
A constraint of the form $a_{j,1}x_1 + a_{j,2}x_2 + \cdots + a_{j,n}x_n \leq b_j$ can be rewritten as $a_{j,1}x_1 + a_{j,2}x_2 + \cdots + a_{j,n}x_n + s_j = b_j$, by introducing a new variable s_j, called a *slack variable.* We also add the nonnegativity constraint $s_j \geq 0$.

From "$\geq$" to "$=$":
A constraint of the form $a_{j,1}x_1 + a_{j,2}x_2 + \cdots + a_{j,n}x_n \geq b_j$ can be rewritten as $a_{j,1}x_1 + a_{j,2}x_2 + \cdots + a_{j,n}x_n - s_j = b_j$, by introducing a new variable s_j, called a *surplus variable.* We also add the nonnegativity constraint $s_j \geq 0$.

From unrestricted variable to nonnegative variable:
Suppose that there exists a variable x_j that does not have a nonnegativity constraint (can be negative). Then, create two new variables x'_j and x''_j, replace x_j with $x'_j - x''_j$, and add the inequalities $x'_j \geq 0$ and $x''_j \geq 0$ to the linear program.

Example 5.3.1. Convert the following linear program to canonical form.

P

$$\begin{aligned}
\min z = \ & 3x_1 - 2x_2 - x_3 + x_4 - 87 \\
\text{Subject to} & \\
(1) \quad & 4x_1 - x_2 + x_4 \le 6 \\
(2) \quad & -7x_1 + 8x_2 + x_3 \ge 7 \\
(3) \quad & 2x_1 + x_2 + 4x_4 = 12 \\
& x_1, x_2, x_3 \ge 0 \\
& x_4 \text{ unrestricted}
\end{aligned}$$

We obtain the following linear program.

***P* in canonical form**

$$\begin{aligned}
\max z = \ & -3x_1 + 2x_2 + x_3 - x_4' + x_4'' + 87 \\
\text{Subject to} & \\
(1) \quad & 4x_1 - x_2 + x_4' - x_4'' + s_1 = 6 \\
(2) \quad & -7x_1 + 8x_2 + x_3 - s_2 = 7 \\
(3) \quad & 2x_1 + x_2 + 4x_4' - 4x_4'' = 12 \\
& x_1, x_2, x_3, x_4', x_4'', s_1 \ge 0
\end{aligned}$$

5.3.2. Standard form

In some cases, another form of linear program is used. A linear program is in *standard form* if it is of the following form.

LP in matrix standard form

$$\begin{aligned}
\max z = c^\intercal x \\
\text{Subject to} \\
Ax \le b \\
x \ge 0
\end{aligned}$$

A linear program in standard form can be replaced by a linear program in canonical form by just replacing $Ax \le b$ by $Ax + Is = b$, where $s \ge 0$ is a vector of slack variables and I is the $m \times m$ identity matrix. Similarly, a linear program in canonical form can be replaced by a linear program in standard form by replacing $Ax = b$ by $A'x \le b'$ where

$$A' = \begin{bmatrix} A \\ -A \end{bmatrix} \text{ and } b' = \begin{pmatrix} b \\ -b \end{pmatrix} .$$

5.4. The simplex algorithm

In 1951, Dantzig developed a technique to solve linear programs, nowadays commonly referred to as the *simplex method* or *simplex algorithm* [7].

5.4.1. The pivot operation

Two systems of equations $Ax = b$ and $\bar{A}x = \bar{b}$ are said to be equivalent if $\{x \mid Ax = b\} = \{x \mid \bar{A}x = \bar{b}\}$. Let E_i denote equation i of the system $Ax = b$, *i.e.* $a_{i,1}x_1 + \cdots + a_{i,n}x_n = b_i$. Given a system $Ax = b$, an *elementary row operation* consists in replacing E_i either by αE_i, where α is a non-zero scalar, or by $E_i + \beta E_k$, where β is a non-zero scalar and $k \neq i$. Clearly, if $\bar{A}x = \bar{b}$ can be obtained from $Ax = b$ by elementary row operations, then the two systems are equivalent. Notice that an elementary row operation is reversible.

Let $a_{r,s}$ be a non-zero element of A. A *pivot* on $a_{r,s}$ consists in performing the following sequence of elementary row operations:

a. replacing E_r by $\bar{E}_r = \frac{1}{a_{r,s}}E_r$,

b. for $i \in \{1, \ldots, m\}$, $i \neq r$, replacing E_i by $\bar{E}_i = E_i - a_{i,s}\bar{E}_r = E_i - \frac{a_{i,s}}{a_{r,s}}E_r$.

After pivoting on $a_{r,s}$, all coefficients in column s are equal to 0 except the one in row r, which is now equal to 1. Since a pivot consists of elementary operations, the resulting system $\bar{A}x = \bar{b}$ is equivalent to the original system $Ax = b$.

5.4.2. An example: given a linear program with a basic feasible solution

For simplicity, we assume that the linear program we try to solve is such that:

- it is in canonical form,
- $b \geq 0$,
- there exists a collection B of m variables called a *basis* such that
  - the submatrix A_B of A consisting of the columns of A corresponding to the variables in B is the $m \times m$ identity matrix and
  - the cost coefficients corresponding to the variables in B are all equal to 0.

For example, the following linear program has the required form.

LP with a basic feasible solution

$$\max z = 10 + 20x_1 + 16x_2 + 12x_3$$

Subject to

$$\begin{array}{lrrrrrrl}
(1) & x_1 & & & +x_4 & & & = 4 \\
(2) & 2x_1 + & x_2 + & x_3 & & +x_5 & & = 10 \\
(3) & 2x_1 + & 2x_2 + & x_3 & & & +x_6 & = 16 \\
 & & & & & & x_1, x_2, x_3, x_4, x_5, x_6 & \geq 0
\end{array}$$

In this example, $B = \{x_4, x_5, x_6\}$, and the variables in B are called *basic variables* while the other variables are *non-basic*. The set of non-basic variables is denoted by N. In the example, $N = \{x_1, x_2, x_3\}$.

The advantage of having $A_B = I$ is that we can quickly infer the values of the basic variables given the values of the non-basic variables. For instance, if $x_1 = 1$, $x_2 = 2$ and $x_3 = 3$, then

$$\begin{aligned} x_4 &= 4 - x_1 = 3\,, \\ x_5 &= 10 - 2x_1 - x_2 - x_3 = 3\,, \\ x_6 &= 16 - 2x_1 - 2x_2 - x_3 = 7\,. \end{aligned}$$

Furthermore, we do not need to know the values of the basic variables to evaluate the cost of the solution. Here, we have $z = 10 + 20x_1 + 16x_2 + 12x_3 = 98$. Notice that there is no guarantee that the constructed solution is feasible. For instance, if $x_1 = 5$, $x_2 = 2$ and $x_3 = 1$, then $x_4 = 4 - x_1 = -1$ does not respect the nonnegativity constraint $x_4 \geq 0$.

There is an assignment of values to the non-basic variables that needs special consideration. By just letting all non-basic variables to be equal to 0, we see that the values of the basic variables are just given by the right-hand-sides of the constraints and the cost of the resulting solution is just the constant term in the objective function. In our example, if $x_1 = x_2 = x_3 = 0$, we obtain $x_4 = 4$, $x_5 = 10$, $x_6 = 16$ and $z = 10$. Such a solution is called a *basic feasible solution*. The feasibility of this solution comes from the fact that $b \geq 0$. Later, we shall see that, when solving a linear program, we can restrict our attention to basic feasible solutions. The simplex method is an iterative method that generates a sequence of basic feasible solutions (corresponding to different bases) and eventually stops when it has found an optimal basic feasible solution.

Instead of always writing explicitly these linear programs, we adopt what is known as the *tableau format*. First, in order to have the objective function play a similar role as the other constraints, we consider z to be a variable and the objective function as a constraint. Putting all variables on the same side of the equality sign, we obtain:

$$-z + 20x_1 + 16x_2 + 12x_3 = -10\,.$$

We also get rid of the variable names in the constraints to obtain the tableau format (empty cells have value 0 but are left empty to emphasize that they belong to a column of a basic variable).

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1	20	16	12				-10
	1	0	0	1			4
	2	1	1		1		10
	2	2	1			1	16

Our basic feasible solution is currently $(x_1, x_2, x_3, x_4, x_5, x_6) = (0, 0, 0, 4, 10, 16)$ and $z = 10$. Since the cost coefficient c_1 of x_1 is positive (it is equal to 20), we notice that we can increase z by increasing x_1 and keeping x_2 and x_3 at the value 0. But in order to maintain the feasibility of the system, we must have that $x_4 = 4 - x_1 \geq 0$, $x_5 = 10 - 2x_1 \geq 0$ and $x_6 = 16 - 2x_1 \geq 0$. This implies that $x_1 \leq 4$. So we can set $x_1 = 4$, $x_2 = 0$ and $x_3 = 0$, and we obtain $x_4 = 0$, $x_5 = 2$, $x_6 = 8$ and $z = 90$. This new solution $(x_1, x_2, x_3, x_4, x_5, x_6) = (4, 0, 0, 0, 2, 8)$ is also a basic feasible solution and corresponds to the basis $B = \{x_1, x_5, x_6\}$. We say that "x_1 has entered the basis and, as a result, x_4

has left the basis." Notice that there is a *unique* basic solution associated with any basis. This (not necessarily feasible) solution is obtained by setting the non-basic variables to zero and deducing the values of the basic variables from the m constraints.

Now, we would like that our tableau reflects this change by showing the dependence of the new basic variables as a function of the non-basic variables. This can be accomplished by pivoting on the element $a_{1,1}$: we need to pivot on an element of column 1 because x_1 is entering the basis; the choice of the row to pivot on is dictated by the variable which leaves the basis, which in this case is x_4. After pivoting on $a_{1,1}$, we obtain the following tableau.

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1		16	12	−20			−90
	1	0	0	1			4
		1	1	−2	1		2
		2	1	−2		1	8

Notice that while pivoting, we also modified the objective function row, just like if it was another constraint. We now have a linear program which is equivalent to the original one and from which we can easily extract a (basic) feasible solution of value 90. Still z can be improved by increasing x_s for $s = 2$ or $s = 3$ since these variables have a positive cost coefficient[3] $\bar{c}_s$. Let us choose the one with the greatest $\bar{c}_s$; in our case x_2 enters the basis. The maximum value that x_2 can take while x_3 and x_4 remain nonnegative is dictated by the constraints $x_1 = 4 \geq 0$, $x_5 = 2 - x_2 \geq 0$, and $x_6 = 8 - 2x_2 \geq 0$. As the tightest of these inequalities is $x_5 = 2 - x_2 \geq 0$, variable x_5 must leave the basis. Therefore, pivoting on $\bar{a}_{2,2}$, we obtain the tableau:

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1			−4	12	−16		−122
	1		0	1	0		4
		1	1	−2	1		2
			−1	2	−2	1	4

The current basis is $B = \{x_1, x_2, x_6\}$ and its value is 122. Since $12 > 0$, we can improve the current basic feasible solution by having x_4 enter the basis. Instead of writing explicitly the constraint on x_4 to compute the value at which x_4 can enter the basis, we perform the *min ratio test.* If x_s is the variable that is entering the basis, we compute

$$\underset{i \text{ s.t. } \bar{a}_{i,s}>0}{\arg\min} \left\{ \bar{b}_i / \bar{a}_{i,s} \right\}.$$

The argument of the minimum gives the index of the row of the basic variable leaving the basis. In our example, we obtain that the minimum is obtained with $i = 3$, the value of the minimum being $4/2 = 2$. Therefore, variable x_6 must leave the basis, since it is the basic variable corresponding to row 3. Moreover, in order to get the updated tableau, we need to pivot on $\bar{a}_{3,4}$. Doing so, we obtain

[3]By simplicity, to differentiate the vectors and matrix from the original linear program from the new ones obtained after pivoting, we always denote the data corresponding to the new tableau by $\bar{c}$, $\bar{A}$ and $\bar{b}$.

the following tableau.

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1			2		-4	-6	-146
	1		$1/2$		1	$-1/2$	2
		1	0		-1	1	6
			$-1/2$	1	-1	$1/2$	2

Our current basic solution is $(x_1, x_2, x_3, x_4, x_5, x_6) = (2, 6, 0, 2, 0, 0)$ and $z = 146$ (the value of the solution).

Remark 5.4.1. If $a_{r,s} > 0$ and $\frac{b_r}{a_{r,s}} \leq \frac{b_i}{a_{i,s}}$ for all $a_{i,s} > 0$, then after pivoting on $a_{r,s}$ we obtain:

- $\bar{b}_r = \frac{b_r}{a_{r,s}} \geq 0$,
- $\bar{b}_i = b_i - \frac{a_{i,s}}{a_{r,s}} \cdot b_r \geq b_i \geq 0$ if $a_{i,s} \leq 0$, and
- $\bar{b}_i = b_i - \frac{a_{i,s}}{a_{r,s}} \cdot b_r = a_{i,s}(\frac{b_i}{a_{i,s}} - \frac{b_r}{a_{r,s}}) \geq 0$ if $a_{i,s} > 0$.

By Remark 5.4.1, we obtain that the new vector $\bar{b} \geq 0$, which means that the solution is feasible (since every basic variable respects the nonnegativity constraint).

Remark 5.4.2. When pivoting on $a_{r,s} > 0$, the constant term c_0 in the objective function becomes $c_0 + b_r \cdot \frac{c_s}{a_{r,s}}$.

By Remark 5.4.2, we notice that if $b_r > 0$ when we pivot on $a_{r,s}$, then we strictly increase c_0, that is, $\bar{c}_0 > c_0$. In other words, we striclty increase the value of the solution. This comes from the fact that $c_s > 0$. We shall see later how to deal with the case when $b_r = 0$.

Going back on our example, we notice that the basic solution $B = \{x_1, x_2, x_4\}$ is not yet optimal, since there is still a positive cost coefficient. The variable x_3 is the only candidate to enter the basis, and since there is only one positive element in row 3, we directly infer that x_1 has to leave the basis. Thus, we pivot on $\bar{a}_{1,3}$ and obtain the following tableau.

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1	-4				-8	-4	-154
	2		1		2	-1	4
	0	1			-1	1	6
	1			1	0	0	4

The basis is $B = \{x_2, x_3, x_4\}$ and the associated basic feasible solution is $(x_1, x_2, x_3, x_4, x_5, x_6) = (0, 6, 4, 4, 0, 0)$ with value $z = 154$. This basic feasible solution is *optimal* since the objective function reads $z = 154 - 4x_1 - 8x_5 - 4x_6$ and cannot be more than 154 due to the nonnegativity constraints.

5.4.3. Some more cases

Through a sequence of pivots, the simplex algorithm goes from one linear program to another equivalent linear program which is trivial to solve. This is possible from the crucial observation that a pivot operation yields an equivalent system of equations, and thus does not alter the feasible region.

The example of Section 5.4.2 was quite simple. In general, some more complicated situations can occur.

First, in the min ratio test, several terms might produce the minimum. In that case, we can select arbitrarily one of them. For example, suppose that the current tableau is the following.

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1		16	12	-20			-90
	1	0	0	1			4
		1	1	-2	1		**2**
		2	1	-2		1	**4**

If x_2 is entering the basis, the min ratio test gives $2 = \min\{2/1, 4/2\}$. Thus, either x_5 or x_6 can leave the basis. If we decide to have x_5 leave the basis, we pivot on $\bar{a}_{2,2}$; otherwise, if x_6 leaves the basis, we pivot on $\bar{a}_{3,2}$. Notice that, in any case, the pivot operation creates a zero coefficient among the values in b. For instance, pivoting on $\bar{a}_{2,2}$, we obtain the following tableau.

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1			-4	12	-16		-122
	1		0	1	0		4
		1	1	-2	1		2
			-1	2	-2	1	**0**

A basic feasible solution with $\bar{b}_i = 0$ for some $i \in \{1, \ldots, m\}$ is called *degenerate*. A linear program is *non-degenerate* if no basic feasible solution is degenerate (so the goal of Phase I of the simplex algorithm is to give a basic feasible solution that is non-degenerate). Pivoting now on $\bar{a}_{3,4}$, we obtain the following tableau.

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1			2		-4	-6	-122
	1		$1/2$		1	$-1/2$	4
		1	0		-1	1	2
			$-1/2$	1	-1	$1/2$	0

This pivot is degenerate. A pivot on $\bar{a}_{r,s}$ is called *degenerate* if $\bar{b}_r = 0$. Notice that a degenerate pivot alters neither the $\bar{b}_i$'s nor $\bar{c}_0$. In the example, the basic feasible solution is $(x_1, x_2, x_3, x_4, x_5, x_6) = (4, 2, 0, 0, 0, 0)$ in both tableaus. We thus observe that several bases can correspond to the same basic feasible solution.

Another situation that may occur is when x_s is entering the basis but $\bar{a}_{i,s} \leq 0$, for all $i \in \{1, \ldots, m\}$. In this case, there is no term in the min ratio test. This means that, while keeping the other non-basic variables at value 0, x_s can take an arbitrarily large value without violating feasibility. Since $\bar{c}_s > 0$, this implies that z can be made arbitrarily large. In this case, the linear program is said to be *unbounded*, or *unbounded from above* if we want to emphasize the fact that we are dealing with a maximization problem. For example, consider the following tableau.

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1		16	12	20			-90
	1	0	0	-1			4
		1	1	0	1		2
		2	1	-2		1	8

If x_4 enters the basis, we have that $x_1 = 4 + x_4$, $x_5 = 2$ and $x_6 = 8 + 2x_4$. As a result, for *any* nonnegative value of x_4, the solution $(4 + x_4, 0, 0, x_4, 2, 8 + 2x_4)$ is a feasible solution with value $z = 90 + 20x_4$. Thus, there is no *finite optimum*.

5.4.4. In general: given a linear program with a basic feasible solution

In this section, we summarize the different steps of the simplex algorithm that we used and described in Section 5.4.2. We assume that the input linear program has a basic feasible solution, which is obtained through Phase I of the algorithm (see Section 5.4.7). Suppose the initial or current tableau is as follows.

$-z$	x_1	$\dots$	x_s	$\dots$	x_n	
1	$\bar{c}_1$	$\dots$	$\bar{c}_s$	$\dots$	$\bar{c}_n$	$-\bar{c}_0$
	$\bar{a}_{1,1}$	$\dots$	$\bar{a}_{1,s}$	$\dots$	$\bar{a}_{1,n}$	$\bar{b}_1 \geq 0$
	$\vdots$		$\vdots$		$\vdots$	$\vdots$
	$\bar{a}_{r,1}$	$\dots$	$\bar{a}_{r,s}$	$\dots$	$\bar{a}_{r,n}$	$\bar{b}_r \geq 0$
	$\vdots$		$\vdots$		$\vdots$	$\vdots$
	$\bar{a}_{m,1}$	$\dots$	$\bar{a}_{m,s}$	$\dots$	$\bar{a}_{m,n}$	$\bar{b}_n \geq 0$

and the variables can be partitioned in $B = \{x_{j_1}, \dots, x_{j_m}\}$ and N with the property that

$$\bar{c_{j_i}} = 0 \text{ for } i \in \{1, \dots, m\} \text{ and } \bar{a}_{k,j_i} = \begin{cases} 0 & \text{if } k \neq i\,, \\ 1 & \text{if } k = i\,. \end{cases}$$

The *basic feasible solution* is given by $x_{j_i} = \bar{b}_i$ for $i \in \{1, \dots, m\}$ and $x_j = 0$ otherwise. The objective function value of this solution is $\bar{c}_0$.

Phase II of the simplex algorithm

a. *Check for optimality.*
   If $\bar{c}_j \leq 0$ for all $j \in \{1, \dots, n\}$, the the current basic feasible solution is optimal: **STOP**.

b. *Fix the variable entering the basis.*
   Find a column s for which $\bar{c}_s > 0$. The variable x_s is entering the basis.

c. *Check for unboundedness.*
   If $\bar{a}_{i,s} \leq 0$ for all $i \in \{1, \dots, m\}$, then the linear program is unbounded (from above): **STOP**.

d. *Min ratio test.*
   Find a row r such that $\bar{b}_r/\bar{a}_{r,s} = \min_{i \text{ s.t. } \bar{a}_{i,s} > 0} \left\{\bar{b}_i/\bar{a}_{i,s}\right\}$. The basic variable x_{j_r} is leaving the basis.

e. *Pivot on $\bar{a}_{r,s}$.*
Replace the current tableau by the following and replace x_{j_r} by x_s in B.

$$
\begin{array}{r|ccccc|c}
-z & \dots & x_s & \dots & x_j & \dots & \\
\hline
1 & \dots & 0 & \dots & \bar{c}_j - \frac{\bar{a}_{r,j}\bar{c}_s}{\bar{a}_{r,s}} & \dots & -\bar{c}_0 - \frac{\bar{b}_r\bar{c}_s}{\bar{a}_{r,s}} \\
\hline
 & & \vdots & & \vdots & \vdots & \vdots \\
(\text{row } r) & \dots & 1 & \dots & \frac{\bar{a}_{r,j}}{\bar{a}_{r,s}} & \dots & \frac{\bar{b}_r}{\bar{a}_{r,s}} \\
 & & \vdots & & \vdots & \vdots & \vdots \\
(\text{row } i) & \dots & 0 & \dots & \bar{a}_{i,j} - \frac{\bar{a}_{r,j}\bar{a}_{i,s}}{\bar{a}_{r,s}} & \dots & \bar{b}_i - \frac{\bar{b}_r\bar{a}_{i,s}}{\bar{a}_{r,s}} \\
 & & \vdots & & \vdots & \vdots & \vdots
\end{array}
$$

f. *Go to Step a.*

5.4.5. Convergence of the simplex algorithm

As we have seen in the previous subsections, the simplex algorithm is an iterative algorithm, that generates a sequence of basic feasible solutions. However, we have not yet discussed whether the algorithm terminates. We have however noticed that the value of the solution strictly increases if the problem is not degenerate. Hence, if the linear program is bounded (the corresponding polyhedral region is bounded), then the algorithm must terminate.

Theorem 5.4.1. *The simplex algorithm solves a non-degenerate linear program in finitely many steps.*

Proof. For non-degenerate linear programs, there is an improvement of value $\frac{\bar{b}_r\bar{c}_s}{\bar{a}_{r,s}} > 0$ in the objective function at each iteration. Hence, in the sequence of basic feasible solutions produced by the simplex algorithm, each basic feasible solution can appear at most once. Therefore, for non-degenerate linear programs, the number of iterations is bounded by the number of basic feasible solutions. This latter number is bounded by $\binom{n}{m}$ since any basic feasible solution corresponds to m variables being basic (although not all choices of basic variables give rise to *feasible* solutions). □

We note however that, for degenerate linear programs, it is possible that the simplex algorithm loops infinitely (if unfortunate choices of entering and leaving variables are made).

Example 5.4.1. Here is an example of a degenerate linear program on which the simplex algorithm may never terminate (read from left to right and notice that pivoting on $\bar{a}_{1,5}$ in the last tableau yields the first tableau—circled values represent the pivots).

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1	4	1.92	-16	-0.96			0
	-12.5	-2	12.5	1	1		0
	①	0.24	-2	-0.24		1	0

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1		0.96	-8	0		-4	0
		①	-12.5	-2	1	12.5	0
	1	0.24	-2	-0.24		1	0

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1			4	1.92	-0.96	-16	0
		1	-12.5	-2	1	12.5	0
	1		①	0.24	-0.24	-2	0

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1	-4			0.96	0	-8	0
	12.5	1		①	-2	-12.5	0
	1		1	0.24	-0.24	-2	0

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1	-16	-0.96			1.92	4	0
	12.5	1		1	-2	-12.5	0
	-2	-0.24	1		0.24	①	0

$-z$	x_1	x_2	x_3	x_4	x_5	x_6	
1	-8	0	-4		0.96		0
	-12.5	-2	12.5	1	①		0
	-2	-0.24	1		0.24	1	0

5.4.6. Bland's anticycling rule

The simplex method, as described in Section 5.4.4, is ambiguous (not fully deterministic). If we have several variables with a positive $\bar{c}_s$ at Step b, then we have not specified which enters the basis. Moreover, there might be several variables attaining the minimum in the minimum ratio test at Step d. If so, we need to specify which of these variables leaves the basis.

A pivoting rule consists of an entering variable rule and a leaving variable rule that unambiguously decide what are the entering and leaving variables. The most classical entering variable rule and leaving variable rule are the following.

- **Largest coefficient entering variable rule:** Select the variable x_s with the largest $\bar{c}_s > 0$. In case of ties, select the one with the smallest index s.
- **Largest coefficient leaving variable rule:** Among all rows attaining the minimum in the minimum ratio test, select the one with the largest pivot $\bar{a}_{r,s}$. In case of ties, select the one with the smallest index r.

As we have seen in Example 5.4.1, the use of the largest coefficient entering and leaving variable rule does not prevent cycling. There are two rules that avoid cycling: the *lexicographic rule* and *Bland's rule* (named after Bland who discovered the rule in 1977 [3]). We only describe the latter one, which is conceptually simpler.

- **Bland's anticycling pivoting rule:** Among all variables x_s with positive $\bar{c}_s$, select the one with the smallest index s. Among the possible leaving variables x_r, which are equivalent with respect to the min ratio test, select the one with the smallest index r.

Theorem 5.4.2. *The simplex algorithm with Bland's anticycling pivoting rule terminates after a finite number of iterations.*

Proof. The proof is by contradiction. If the method does not stop after a finite number of iterations, then there is a cycle of tableaus that repeats. If we delete from the tableau that initiates this cycle the rows and columns not containing pivots during the cycle, the resulting tableau has a cycle with the same pivots. For this tableau, all $\bar{b} = 0$ throughout the cycle since all pivots are degenerate. Let t be the largest index of the variables remaining. Consider the tableau T_1 in the cycle with x_t leaving. Let $B = \{x_{j_1}, \dots, x_{j_m}\}$ be the corresponding basis (say $j_r = t$), x_s be the associated entering variable and, $a^1_{i,j}$ and c^1_j the constraint and cost coefficients. On the other hand, consider the tableau T_2 with x_t entering and denotes by $a^2_{i,j}$ and c^2_j the corresponding constraint and cost coefficients. Let x be the (infeasible) solution obtained by letting the non-basic variables in T_1 be zero except for $x_s = -1$. Since $\bar{b} = 0$, we deduce that $x_{j_i} = a_{i,s}$ for $i \in \{1, \dots, m\}$. Also, as T_2 is obtained from T_1 by elementary row operations, x must have the same objective function value in T_1 and T_2. This means that

$$c^1_0 - c^1_s = c^2_0 - c^2_s + \sum_{i=1}^{m} a^1_{i,s} c^2_{j_i} \,.$$

Because we have no improvement in objective function in the cycle, we have $c^1_0 = c^2_0$. Moreover, $c^1_s > 0$ and, by Bland's rule, $c^2_s \leq 0$, otherwise x_t would not be the entering variable in T_2. Hence,

$$\sum_{i=1}^{m} a^1_{i,s} c^2_{j_i} < 0 \,,$$

implying that there exists k with $a^1_{k,s} c^2_{j_k} < 0$. Notice that $k \neq r$, *i.e.*, $j_k < t$, since the pivot element in T_1, that is, $a^1_{r,s}$, must be positive and $c^2_t > 0$. However, in T_2, all cost coefficients c^2_j except c^2_t are nonnegative; otherwise x_j would have been selected as entering variable. Thus $c^2_{j_k} < 0$ and $a^1_{k,s} > 0$. This is a contradiction because Bland's rule should have selected $x_{j,k}$ rather than x_t in T_1 as leaving variable. □

5.4.7. Find a basic feasible solution

In this section, we consider Phase I of the simplex algorithm, which consists in finding a basis of a given linear program that leads to a basic feasible solution. Note that if the given problem have constraints of the form $Ax \leq b$ with $b \geq 0$, then the slack variables (which must be introduced when transforming the problem in canonical form) constitute the basis.

Consider a program in canonical form with $b \geq 0$. Note that this latter restriction is without loss of generality: since we have $Ax = b$, we may multiply some constraints by -1. So we have a linear program of the following form.

Linear program P

$$\max z = c_0 + c^\intercal x$$

Subject to

$$Ax = b$$
$$x \geq 0$$

Instead of solving this linear program (which may not have a basic feasible solution), we add some artificial variables $\{x_i^a \mid 1 \leq i \leq m\}$ and consider the following linear program.

Linear program P'

$$\min w = \sum_{i=1}^{m} x_i^a$$

Subject to

$$\begin{aligned} &Ax + Ix^a = b \\ &x, x^a \geq 0 \end{aligned}$$

Clearly, this new linear program is not in the required form to apply Phase II of the simplex algorithm. It can nonetheless easily be transformed to this form by changing $\min w$ by $\max w' = -w$ and expressing the objective function in terms of the initial variables. Eventually, we obtain a linear program with the following form.

Linear program Q

$$\max w' = -\sum_{i=1}^{m} x_i^a$$

Subject to

$$\begin{aligned} &Ax + Ix^a = b \\ &x, x^a \geq 0 \end{aligned}$$

We artificially created a basic feasible solution: $x = 0$ and $x^a = b$. Then, we use Phase II of the simplex algorithm, as described in Section 5.4.4, to solve the linear program Q. There are three possible outcomes:

a. w' is reduced to zero and no artificial variables remain in the basis, *i.e.* we are left with a basis consisting only of original variables. In this case, we simply delete the columns corresponding to the artificial variables, replace the objective function by the objective function of the original linear program P after having expressed it in terms of the non-basic variables, and use Phase II of the simplex algorithm to solve the new linear program.

b. $w' < 0$ at optimality. This means that the original linear program P is infeasible. To see this, notice that if x is feasible in P, then $(x, x^a = 0)$ is feasible in Q with value $w' = 0$.

c. w' is reduced to zero but some artificial variables remain in the basis. These artificial variables must be at zero since, for this solution, $-w' = \sum_{i=1}^{m} x_i^a = 0$. Suppose that the ith variable of the basis is artificial We may pivot on any non-zero (not necessarily positive) element $\bar{a}_{i,j}$ of row i corresponding to a non-artificial variable x_j. Since $\bar{b}_i = 0$, there is no change in the solution or in w'. We say that we are *driving the artificial variables out of the basis.* By repeating this for all artificial variables in the basis, we obtain a basis consisting only of original variables. We have thus reduced this case to case 1.

However, we might be unsuccessful in driving one artificial variable out the basis if $\bar{a}_{i,j} = 0$ for $j \in \{1, \ldots, n\}$. This means that we have arrived at a zero row in the original matrix by performing elementary row operations, implying that the constraint is redundant. We can delete this constraint and continue in Phase II of the simplex algorithm with a basis of lower dimension.

Example 5.4.2. Consider the following linear program, already in canonical form and written as a tableau.

$-z$	x_1	x_2	x_3	x_4	
1	20	16	12	5	0
	1	0	1	2	4
	0	1	2	3	2
	0	1	0	2	2

We first observe that we don't need to add an artificial variable for the first constraint, since we can use x_1 as basic variable. In general, if a variable x_s is such that $a_{r,s} \neq 0$ for some $r \in \{1, \ldots, m\}$ and $a_{j,s} = 0$ for all $j \neq r$, $j \in \{1, \ldots, m\}$, then we can use x_s as basic variable by replacing E_r by $\bar{E}_r = \frac{1}{a_{r,s}} E_r$.

As x_1 is a basic variable (with coefficient equal to 1 in row 1), it remains to add two *artificial variables* for the remaining two rows (2 and 3). We obtain the following linear program, whose objective function is to maximize $w' = -x_1^a - x_2^a$ (which is equivalent to minimizing $w = x_1^a + x_2^a$) and, as a result, the objective function coefficients of the non-basic variables as well as $-\bar{c}_0$ are obtained by taking the negative of the sum of all rows corresponding to artificial variables. This is done by expressing the artificial variables x_1^a and x_2^a with the non-basic variables x_2, x_3 and x_4:

$$\left.\begin{array}{l} x_1^a = -x_2 - 2x_3 - 3x_4 + 2 \\ x_2^a = -x_2 - 2x_4 + 2 \end{array}\right\} \implies w' = -x_1^a - x_2^a = 2x_2 + 2x_3 + 5x_4 - 4\,.$$

We can thus consider the following tableau.

$-w'$	x_1	x_2	x_3	x_4	x_1^a	x_2^a	
1		2	2	5			4
	1	0	1	2			4
		1	2	3	1		2
		1	0	2		1	2

Pivoting on $\bar{a}_{2,2}$, which makes x_2 enter the basis and x_1^a leave the basis, we obtain the following tableau.

$-w'$	x_1	x_2	x_3	x_4	x_1^a	x_2^a	
1			-2	-1	-2		0
	1		1	2	0		4
		1	2	3	1		2
			-2	-1	-1	1	0

This tableau is optimal, since the costs coefficients $c \leq 0$; this follows from the fact that all variables, including the artificial variables, are nonnegative. The objective function value is $w' = 0$, which implies that the original linear program is feasible. However, x_a^2 is still in the basis of the current tableau. To obtain a basic feasible solution to the original program, we need to drive x_2^a out of the basis. This can be done by choosing some arbitrary non-basic variable from the original program and pivot according to the usual min ratio test. Here, if we pivot on $\bar{a}_{3,4}$, then we obtain the following tableau.

$-w'$	x_1	x_2	x_3	x_4	x_1^a	x_2^a	
1			0		-1	-1	0
	1		-3		-2	2	4
		1	-4		-2	3	2
			2	1	1	-1	0

We are not interested anymore in the artificial variables (they are not basic and thus their value is zero), so we remove the two corresponding columns. Hence, we obtain the following tableau.

$-w'$	x_1	x_2	x_3	x_4	
1			0		0
	1		-3		4
		1	-4		2
			2	1	0

We now express z as a function of the basic variables x_1, x_2 and x_4.

$$\left.\begin{aligned} x_1 &= 3x_3 + 4 \\ x_2 &= 4x_3 + 2 \\ x_4 &= -2x_3 \end{aligned}\right\} \implies z = 20x_1 + 16x_2 + 12x_3 + 5x_4 = 126x_3 + 112\,.$$

This gives us the following tableau.

$-z$	x_1	x_2	x_3	x_4	
1			126		-112
	1		-3		4
		1	-4		2
			2	1	0

It remains to solve this tableau with Phase II of the simplex algorithm. Note that this linear program is degenerate since there exists $i \in \{1, \ldots, m\}$ such that $\bar{b}_i = 0$ (in this example, $i = 3$). In other words, there exists a basic variable (in this example, x_4) whose value in the basic feasible solution is zero.

If we continue to solve this linear program with Phase II of the simplex algorithm, then we must pivot on $\bar{a}_{3,3}$, with x_3 entering and x_4 leaving the basis. We obtain the following tableau.

$-z$	x_1	x_2	x_3	x_4	
1				-63	-112
	1			$3/2$	4
		1		2	2
			1	$1/2$	0

Since all cost coefficients are non-positive, the basic feasible solution $(x_1, x_2, x_3, x_4) = (4, 2, 0, 0)$ is optimal with an objective function value $z = 112$.

Example 5.4.3. Consider the following linear program.

$$
\begin{aligned}
&\min z = -2x_1 - 3x_2 \\
&\text{Subject to} \\
&(1) \quad 2x_1 + x_2 \leq 32 \\
&(2) \quad -2x_1 - 2x_2 \leq -20 \\
&(3) \quad x_1 + 3x_2 \leq 48 \\
&\qquad x_1, x_2 \geq 0
\end{aligned}
$$

First, we transform the linear program into an equivalent linear program whose objective function is to maximize and whose vector $b \geq 0$.

$$
\begin{aligned}
&\max z = 2x_1 + 3x_2 \\
&\text{Subject to} \\
&(1) \quad 2x_1 + x_2 \leq 32 \\
&(2) \quad 2x_1 + 2x_2 \geq 20 \\
&(3) \quad x_1 + 3x_2 \leq 48 \\
&\qquad x_1, x_2 \geq 0
\end{aligned}
$$

Now, we can transform our linear program into canonical form by the introduction of slack and surplus variables.

$$
\begin{aligned}
&\max z = 2x_1 + 3x_2 \\
&\text{Subject to} \\
&(1) \quad 2x_1 + x_2 + s_1 = 32 \\
&(2) \quad 2x_1 + 2x_2 - s_2 = 20 \\
&(3) \quad x_1 + 3x_2 + s_3 = 48 \\
&\qquad x_1, x_2, s_1, s_2, s_3 \geq 0
\end{aligned}
$$

Now, we need to find a basis for our linear program. We notice that s_1 appears with coefficient 1 in $\bar{E}_1$ (the first constraint) and with coefficient 0 everywhere else, including the objective function. So s_1 can be used as basic variable associated with $\bar{E}_1$. The same hods for s_3, which has coefficient 1 in $\bar{E}_3$ (the third constraint) and coefficient 0 everywhere else, including the objective function. However, we still need to introduce an artificial variable for the second constraint.

$$
\begin{array}{llrl}
\max z = & 2x_1 + 3x_2 & & \\
\text{Subject to} & & & \\
(1) & 2x_1 + x_2 + s_1 & & = 32 \\
(2) & 2x_1 + 2x_2 - s_2 & + x_a^1 & = 20 \\
(3) & x_1 + 3x_2 + s_3 & & = 48 \\
 & & x_1, x_2, s_1, s_2, s_3, x_a^1 & \geq 0
\end{array}
$$

In this new linear program, we have a basis $B = \{s_1, s_3, x_a^1\}$. We want to get rid of x_a^1 from the basis, so we minimize its value in a new linear program with the same constraints. The objective function is $\min w = x_a^1$, which is equivalent to $\max w' = -x_a^1$. Since x_a^1 is a basic variable, we can rewrite it in terms of the non-basic variables: $x_a^1 = 20 - 2x_1 - 2x_2 + s_2$. Thus, $\max w' = -20 + 2x_1 + 2x_2 - s_2$, and we get the following linear program.

$$
\begin{array}{llrl}
\max w' = & 2x_1 + 2x_2 - s_2 - 20 & & \\
\text{Subject to} & & & \\
(1) & 2x_1 + x_2 + s_1 & & = 32 \\
(2) & 2x_1 + 2x_2 - s_2 & + x_a^1 & = 20 \\
(3) & x_1 + 3x_2 + s_3 & & = 48 \\
 & & x_1, x_2, s_1, s_2, s_3, x_a^1 & \geq 0
\end{array}
$$

For simplicity, we use the tableau form of the program.

$-w'$	x_1	x_2	s_1	s_2	s_3	x_a^1	
1	2	2		-1			20
	2	1	1	0			32
	2	2		-1		1	20
	1	3		0	1		48

The goal is to find an optimal solution of value 0 (if it exists) such that x_a^1 is out of the basis, which would give us a basic feasible solution to our original linear program.

As long as a pivot increasing the value of the solution is possible, we apply such a pivot. If no pivot can be done but the basis still contains an artificial variable, in this case x_a^1, we drive such artificial variable out of the basis. We apply Bland's rule to avoid cycling.

In this example, x_1 and x_2 have positive cost coefficients, and Bland's rule says that x_1 should be entering the basis. We then apply the min ratio test, which gives us that x_1^a must be leaving the basis, with a pivot on $\bar{a}_{2,1}$.

$-w'$	x_1	x_2	s_1	s_2	s_3	x_a^1	
1		0		0		-1	0
		-1	1	1		-1	12
	1	1		$-1/2$		$1/2$	10
		2		$1/2$	1	$-1/2$	38

Since all cost coefficient are non-positive, this tableau is optimal. Besides, x_a^1 is not part of the current basis $\{x_1, s_1, s_3\}$. The corresponding solution is $(x_1, x_2, s_1, s_2, s_3, x_a^1) = (10, 0, 12, 0, 38, 0)$ and has value 0. This corresponds to the end of Phase I of the simplex algorithm.

Since the basis does not contain any artificial variable, we can translate this solution into a basic feasible solution to our original linear program (in canonical form). First, we need to express z in terms of non-basic variables (ignoring artificial variables). We have $z = 2x_1 + 3x_2$ and $x_1 = 10 - x_2 + \frac{s_2}{2}$. Hence, $z = 20 + x_2 + s_2$. We create the following tableau, with the objective function $z = 20 + x_2 + s_2$ and similar constraints as in the previous one (with which we obtained the basic feasible solution).

$-z$	x_1	x_2	s_1	s_2	s_3	
1		1		1		-20
		-1	1	1		12
	1	1		$-1/2$		10
		2		$1/2$	1	38

Following Bland's rule, we pivot on $\bar{a}_{2,2}$, with x_2 entering the basis and x_1 leaving the basis.

$-z$	x_1	x_2	s_1	s_2	s_3	
1	-1			$3/2$		-30
	1		1	$1/2$		22
	1	1		$-1/2$		10
	-2			$3/2$	1	18

Then, we pivot on $\bar{a}_{3,4}$, with s_2 entering the basis and s_3 leaving the basis.

$-z$	x_1	x_2	s_1	s_2	s_3	
1	1				-1	-48
	$5/3$		1		$-1/3$	16
	$1/3$	1			$1/3$	16
	$-4/3$			1	$2/3$	12

Since x_1 still has a positive cost coefficient, we pivot so that x_1 enters the basis. The min ratio test implies that we should pivot on $\bar{a}_{1,1}$.

$-z$	x_1	x_2	s_1	s_2	s_3	
1			$-3/5$		$-4/5$	$-288/5$
	1		$3/5$		$-1/5$	$48/5$
		1	$-1/5$		$2/5$	$64/5$
			$4/5$	1	$2/5$	$124/5$

This tableau is optimal since all cost coefficients are non-positive (there is no way to increase the value of the solution). The solution is $(x_1, x_2, s_1, s_2, s_3) = (48/5, 64/5, 0, 124/5, 0)$ and has value $288/5 = 57.6$. As the original linear program does not have slack or surplus variables, this means that an optimal solution to the original linear program is $(x_1, x_2) = (48/5, 64/5)$ and has the same objective function value of 57.6.

Exercise 28. Solve the following linear program P, that is:

a. convert it to canonical form,

b. find a basic feasible solution with Phase I of the simplex algorithm and

c. find an optimal solution of the obtained linear program with Phase II of the simplex algorithm, or explain why it is unbounded or infeasible.

Linear program P

$$\begin{aligned}
&\max z = 2x_1 + 2x_2 - x_3\\
&\text{Subject to}\\
&(1)\quad 2x_1 + x_2 + x_3 \le 16\\
&(2)\quad x_2 - x_3 \ge 8\\
&(3)\quad 2x_1 + 2x_2 + 2x_3 = 20\\
&\qquad x_1, x_2, x_3 \ge 0
\end{aligned}$$

<u>SOLUTION</u>:

a. We transform P into canonical form. As P is already a maximization program and every variable is nonnegative, it suffices to transform every inequality into an equality by adding slack or surplus variables. We obtain the following linear program.

P in canonical form

$$\begin{aligned}
&\max z = 2x_1 + 2x_2 - x_3\\
&\text{Subject to}\\
&(1)\quad 2x_1 + x_2 + x_3 + s_1 = 16\\
&(2)\quad x_2 - x_3 - s_2 = 8\\
&(3)\quad 2x_1 + 2x_2 + 2x_3 = 20\\
&\qquad x_1, x_2, x_3, s_1, s_2 \ge 0
\end{aligned}$$

b. In order to find a feasible solution to P, we first need a basis. Variable s_1 can act as basic variable, since it as coefficient 1 in (1) and 0 in constraints (2) and (3). We add two artificial variables x_1^a and x_2^a to constraint (2) and (3), respectively.

P with artificial variables

$$\begin{aligned}
\max z =\ & 2x_1 + 2x_2 - x_3 \\
\text{Subject to} & \\
(1)\quad & 2x_1 + x_2 + x_3 + s_1 = 16 \\
(2)\quad & x_2 - x_3 - s_2 + x_1^a = 8 \\
(3)\quad & 2x_1 + 2x_2 + 2x_3 + x_2^a = 20 \\
& x_1, x_2, x_3, s_1, s_2, x_1^a, x_2^a \geq 0
\end{aligned}$$

We thus obtain the basis $B = \{s_1, x_1^a, x_2^a\}$. Note that $x_1^a = 8 - x_2 + x_3 + s_2$ and $x_2^a = 20 - 2x_1 - 2x_2 - 2x_3$. We now convert this into a linear program that minimizes the value $w = x_1^a + x_2^a$, which is equivalent to maximize $w' = -x_1^a - x_2^a = -28 + 2x_1 + 3x_2 + x_3 - s_2$. For simplicity, we write this linear program in its tableau format, and we start solving it using the simplex algorithm.

$-w'$	x_1	x_2	x_3	s_1	s_2	x_1^a	x_2^a	
1	2	3	1		−1			28
	2	1	1	1	0			16
	0	1	−1		−1	1		8
	2	2	2		0		1	20

Following Bland's anticylcing rule and the min ratio test, we must pivot on $\bar{a}_{1,1}$, with x_1 entering and s_1 leaving the basis. We obtain the following tableau.

$-w'$	x_1	x_2	x_3	s_1	s_2	x_1^a	x_2^a	
1		2	0	−1	−1			12
	1	1/2	1/2	1/2	0			8
		1	−1	0	−1	1		8
		1	1	−1	0		1	4

We continue with a pivot on $a_{3,2}$, with x_2 entering and x_2^a leaving the basis.

$-w'$	x_1	x_2	x_3	s_1	s_2	x_1^a	x_2^a	
1			−2	1	−1		−2	4
	1		0	1	0		−1/2	6
			−2	1	−1	1	−1	4
		1	1	−1	0		1	4

We now pivot on $\bar{a}_{2,4}$, with s_1 entering and, conveniently, x_1^a leaving the basis.

$-w'$	x_1	x_2	x_3	s_1	s_2	x_1^a	x_2^a	
1			0		0	−1	−1	0
	1		2		1	−1	1/2	2
			−2	1	−1	1	−1	4
		1	−1		−1	1	0	8

Note that all costs coefficients are nonpositive, so we reached the optimal value for w', which is 0. This implies that the original linear program is feasible, and gives the feasible solution $(x_1, x_2, x_3, s_1, s_2) = (2, 8, 0, 4, 0)$.

We can express z in terms of the non-basic variables:

$$z = 2x_1 + 2_x2 - x_3 = 2(2 - 2x_3 - s_2) + 2(8 + x_3 + s_2) + x_3 = 20 - 3x_3\,.$$

And we obtain the following tableau for the original linear program P.

$-z$	x_1	x_2	x_3	s_1	s_2	
1			-3		0	-20
	1		2		1	2
			-2	1	-1	4
		1	-1		-1	8

c. We observe that all cost coefficients in the expression of z are nonpositive, and hence we already reached an optimal solution $(x_1, x_2, x_3) = (2, 8, 0)$ for P with value $z = 20$.

5.5. Duality

Duality is one of the most important and useful structural properties of linear programs.

5.5.1. A first example

To illustrate the concept of duality, consider the following linear program.

$$\begin{aligned}
\max z = {} & 5x_1 + 4x_2 \\
\text{Subject to} & \\
(1) \quad & x_1 \leq 4 \\
(2) \quad & x_1 + 2x_2 \leq 10 \\
(3) \quad & 3x_1 + 2x_2 \leq 16 \\
& x_1, x_2 \geq 0
\end{aligned}$$

This linear program is the *primal*. By exhibiting any feasible solution, say $x_1 = 4$ and $x_2 = 2$, one derives a lower bound (since we are maximizing) on the optimum value z^* of the linear program; in this case, we have $z^* \geq 28$. How could we derive upper bounds on z^*? Multiplying inequality (3) by 2, we derive that $6x_1 + 4x_2 \leq 32$, for any feasible solution (x_1, x_2). Since $x_1 \geq 0$, this in turn implies that $z = 5x_1 + 4x_2 \leq 6x_1 + 4x_2 \leq 32$ for any feasible solution and, thus, $z^* \leq 32$. One can even combine several inequalities to get upper bounds. Adding up all three inequalities, we get $5x_1 + 4x_2 \leq 30$, implying that $z^* \leq 30$.

In general, one would multiply each inequality (i) by some nonnegative scalar y_i and then add the inequalities together. In our example, we derive that

$$(y_1 + y_2 + 3y_3)x_1 + (2y_2 + 2y_3)x_2 \leq 4y_1 + 10y_2 + 16y_3\,.$$

To derive an upper bound on the value of an optimal solution z^*, one would then impose that the coefficients on the x_i's in this implied inequality are at least the cost coefficients in the original linear program. In our example, this would mean that we need the constraints $y_1 + y_2 + 3y_3 \geq 5$ and $2y_2 + 2y_3 \geq 4$. To derive the *best* upper bound (*i.e.* the smallest upper bound), one should solve the following *dual* linear program:

$$\begin{array}{lr}
\min w = & 4y_1 + 10y_2 + 16y_3 \\
\text{Subject to} & \\
(1) & y_1 + \quad y_2 + \ 3y_3 \geq 5 \\
(2) & 2y_2 + \ 2y_3 \geq 4 \\
 & y_1, y_2, y_3 \geq 0
\end{array}$$

Observe how the dual is constructed from the primal: one is a maximization problem and the other a minimization problem; the cost coefficients of one are the right hand side of the other and vice versa; the constraint matrix is transposed.

The optimum solution to this linear program is $(y_1, y_2, y_3) = (0, 0.5, 1.5)$, giving an upper bound of 29 on z^*. We shall show that this upper bound is in fact equal to the optimum value of the primal (when the problem is bounded). Here, $(x_1, x_2) = (3, 3.5)$ is a feasible solution for the primal of value 29 as well. Because of our upper bound of 29, this solution *must* be optimal, and thus duality is a way to prove optimality.

5.5.2. In general

In order to find the dual of any linear program P, we first transform P into an equivalent linear program in standard form (see Section 5.3.2) and then write its dual. Hence, we first explain how to obtain the dual of linear programs in standard form.

Consider a linear program P in standard form.

Primal linear program P

$$\begin{array}{l}
\max z = c^\intercal x \\
\text{Subject to} \\
Ax \leq b \\
x \geq 0
\end{array}$$

We define its *dual* linear program D as follows.

Dual linear program D

$$\begin{aligned} &\min w = b^\intercal y \\ \text{Subject to}& \\ &A^\intercal y \geq c \\ &y \geq 0 \end{aligned}$$

The linear program P is called the *primal*.

Notice that there is a bijection between variables of the dual and constraints of the primal, and between constraints of the dual and variables of the primal. In fact, the primal and dual are indistinguishable.

Proposition 5.5.1. *The dual of the dual is the primal.*

Proof. To construct the dual of the dual, we first need to put the dual D in standard form.

D': D in canonical form

$$\begin{aligned} &\max w' = -w = -b^\intercal y \\ \text{Subject to}& \\ &-A^\intercal y \leq -c \\ &y \geq 0 \end{aligned}$$

Therefore the dual DD' of D' is the following linear program.

DD': dual of D'

$$\begin{aligned} &\min z' = -c^\intercal x \\ \text{Subject to}& \\ &-Ax \geq -b \\ &x \geq 0 \end{aligned}$$

Once DD' is transformed into standard form, we obtain the primal P. □

Theorem 5.5.1 (Weak duality)**.** *If x is a feasible solution to the primal P with value z and y is a feasible solution to the dual D with value w, then $z \leq w$.*

Proof.

$$\begin{aligned}
z &= c^\intercal x && \text{(by definition)}\\
&= x^\intercal c && \text{(since this is a scalar product of two vectors)}\\
&\leq x^\intercal (A^\intercal y) && \text{(since } A^\intercal y \geq c \text{ by the dual constraints and the fact that } x \geq 0)\\
&= (x^\intercal A^\intercal) y && \text{(by associativity)}\\
&= (Ax)^\intercal y && \text{(by the properties of transpose)}\\
&\leq b^\intercal y = w && \text{(since } Ax \leq b \text{ by the primal constraints)}
\end{aligned}$$

□

From Theorem 5.5.1, we get that any *dual feasible solution* (*i.e.* a feasible solution to the dual D) gives an upper bound on the optimal value z^* of the primal P and vice versa (*i.e.* any primal feasible solution gives a lower bound on the optimal value w^* of the dual D).

In order to take care of infeasible linear programs, we adopt the convention that the maximum value of any function over an empty set is defined to be $-\infty$ while the minimum value of any function over an empty set is $+\infty$.

Corollary 5.5.1 (Weak duality)**.** *If x is an optimal solution to the primal P with value z^* and y is an optimal solution to the dual D with value w^*, then $z^* \leq w^*$.*

More interestingly, if the primal is bounded, that is, if the value of an optimal solution is finite, then the inequality is in fact an equality.

Theorem 5.5.2 (Strong duality)**.** *If z^* is finite, then $w^* = z^*$.*

Proof. The proof uses the simplex algorithm. In order to solve the primal P with the simplex algorithm, we transform it in canonical form.

P in canonical form

$$\begin{aligned}
&\max z = c^\intercal x\\
\text{Subject to}\quad&\\
&Ax + Is = b\\
&x, s \geq 0
\end{aligned}$$

Let $\tilde{A} = (AI)$, $\tilde{x} = \begin{pmatrix} x \\ s \end{pmatrix}$ and $\tilde{c} = \begin{pmatrix} c \\ 0 \end{pmatrix}$. Let B be the optimal basis obtained by the simplex algorithm. The optimality conditions imply that $\tilde{A}^\intercal y \geq \tilde{c}$ where $y^\intercal = (\tilde{c}_B)^\intercal \tilde{A}_B^{-1}$ (the cost coefficients c_B and the matrix $\tilde{A}_B$ are with respect to B, obtained by the simplex algorithm).

Replacing $\tilde{A}$ by (AI) and $\tilde{c}$ by $\begin{pmatrix} c \\ 0 \end{pmatrix}$, we obtain $A^\intercal y \geq c$ and $y \geq 0$. Hence, y is a dual feasible solution. Moreover, the value of y is precisely $w = y^\intercal b = (\tilde{c}_B)^\intercal \tilde{A}_B^{-1} b = (\tilde{c}_B)^\intercal \tilde{x}_B = z^*$. Therefore, by Corollary 5.5.1, we have $z^* = w^*$. □

Since the dual of the dual is the primal, we have that if either the primal or the dual is feasible and bounded, then so are both of them and their values are equal. From weak duality, we know that if the primal P is unbounded (*i.e.* $z^* = +\infty$), then the dual D is infeasible ($w^* = +\infty$). Similarly, if D is unbounded (*i.e.* $w^* = -\infty$), then P is infeasible ($z^* = -\infty$). However, the converse to these statements are not true: There exist dual pairs of linear programs for which both the primal and the dual are infeasible. Here is a summary of the possible alternatives.

Primal / Dual	bounded (z^* finite)	unbounded ($z^* = +\infty$)	infeasible ($z^* = -\infty$)
bounded (w^* finite)	$z^* = w^*$	impossible	impossible
unbounded ($w^* = -\infty$)	impossible	impossible	possible
infeasible ($w^* = +\infty$)	impossible	possible	possible

As mentioned in the beginning of the section, to obtain the dual of a linear program P, one can first put P into standard form and get the dual as described before. But one can notice that if the primal P (not necessarily in standard form) is a maximization linear program, then inequalities with a $\leq$ sign correspond to nonnegative dual variables, inequalities with a $\geq$ sign correspond to negative dual variables, and equalities correspond to unrestricted dual variables. By performing similar transformations corresponding to the constraints on the primal variables, we obtain the following set of rules for constructing the dual linear program.

Primal	$\longleftrightarrow$	**Dual**
Max	$\longleftrightarrow$	Min
$\sum_j a_{i,j}x_j \leq b_i$	$\longleftrightarrow$	$y_i \geq 0$
$\sum_j a_{i,j}x_j \geq b_i$	$\longleftrightarrow$	$y_i \leq 0$
$\sum_j a_{i,j}x_j = b_i$	$\longleftrightarrow$	y_i is unrestricted
$x_j \geq 0$	$\longleftrightarrow$	$\sum_i a_{i,j}y_i \leq c_j$
$x_j \leq 0$	$\longleftrightarrow$	$\sum_i a_{i,j}y_i \geq c_j$
x_j is unrestricted	$\longleftrightarrow$	$\sum_i a_{i,j}y_i = c_j$

Exercise 29. Consider the following two linear programs.

Linear program P_1

$$\max z = 2x_1 + 2x_2 - x_3$$

Subject to

$$\begin{array}{ll} (1) & 2x_1 + x_2 + x_3 \leq 16 \\ (2) & x_2 - x_3 \geq 8 \\ (3) & 2x_1 + 2x_2 + 2x_3 = 20 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

Linear program P_2

$$\max z = 3x_1 + 2x_3 - x_4$$

Subject to

$$\begin{array}{ll} (1) & x_1 - 5x_2 - x_3 + 2x_4 \leq 50 \\ (2) & 2x_1 + 3x_2 - x_3 + 3x_4 \geq 10 \\ (3) & 2x_1 + 2x_3 - 3x_4 = 30 \\ & x_1, x_2, x_3 \geq 0 \\ & x_4 \text{ unrestricted} \end{array}$$

For $i \in \{1, 2\}$, transform P_i into standard form and write its dual D_i.

Solution: Let us first consider the linear program P_1. Its standard form is as follows.

P_1 in standard form

$$\begin{array}{lrcl} \max z = & 2x_1+2x_2-x_3 & & \\ \text{Subject to} & & & \\ (1) & 2x_1+x_2+x_3 & \leq & 16 \\ (2) & -x_2+x_3 & \leq & -8 \\ (3) & 2x_1+2x_2+2x_3 & \leq & 20 \\ (4) & -2x_1-2x_2-2x_3 & \leq & -20 \\ & x_1,x_2,x_3 & \geq & 0 \end{array}$$

The dual of P_1 is then the following linear program.

Dual of P_1

$$\begin{array}{lrcl} \min w = & 16y_1-8y_2+20y_3-20y_4 & & \\ \text{Subject to} & & & \\ (1) & 2y_1+2y_3-2y_4 & \geq & 2 \\ (2) & y_1-y_2+2y_3-2y_4 & \geq & 2 \\ (3) & y_1+y_2+2y_3-2y_4 & \geq & -1 \\ & y_1,y_2,y_3,y_4 & \geq & 0 \end{array}$$

We now consider the linear program P_2, and put it in standard form as follows.

P_2 in standard form

$$\begin{array}{lrcl} \max z = & 3x_1+2x_3-x_4'+x_4'' & & \\ \text{Subject to} & & & \\ (1) & x_1-5x_2-x_3+2x_4'-2x_4'' & \leq & 50 \\ (2) & -2x_1-3x_2+x_3-3x_4'+3x_4'' & \leq & -10 \\ (3) & 2x_1+2x_3-3x_4'+3x_4'' & \leq & 30 \\ (4) & -2x_1-2x_3+3x_4'-3x_4'' & \leq & -30 \\ & x_1,x_2,x_3,x_4',x_4'' & \geq & 0 \end{array}$$

The dual of P_2 is thus the following linear program.

Dual of P_2

$$\min w = 50y_1 - 10y_2 + 30y_3 - 30y_4$$

Subject to

$$\begin{array}{lrrrrl}
(1) & y_1- & 2y_2+ & 2y_3- & 2y_4 & \geq 3 \\
(2) & -5y_1- & 3y_2 & & & \geq 0 \\
(3) & -y_1+ & y_2+ & 2y_3- & 2y_4 & \geq 2 \\
(4) & 2y_1- & 3y_2- & 3y_3+ & 3y_4 & \geq -1 \\
(5) & -2y_1+ & 3y_2+ & 3y_3- & 3y_4 & \geq 1 \\
 & & & & y_1, y_2, y_3, y_4 & \geq 0
\end{array}$$

5.5.3. Checking for optimality

Consider the following linear programs: P the primal and D the dual.

Primal linear program P

$$\max z = c^\intercal x$$

Subject to

$$Ax \leq b$$
$$x \geq 0$$

Dual linear program D

$$\min w = b^\intercal y$$

Subject to

$$A^\intercal y \geq c$$
$$y \geq 0$$

Strong duality (see Theorem 5.5.2) implies the existence of a simple test for optimality.

Theorem 5.5.3 (Complementary slackness)**.** *If x is a feasible solution to the primal P and y a feasible solution to the dual D, then x is optimal for P and y is optimal for D if only if $y^\intercal(b-Ax) = 0$ and $x^\intercal(A^\intercal y - c) = 0$.*

Proof. By Theorem 5.5.1, we always have $c^\intercal x \overset{x\geq 0}{\leq} (A^\intercal y)^\intercal x = y^\intercal Ax \overset{y\geq 0}{\leq} y^\intercal b = b^\intercal y$. Furthermore, by strong duality (see Theorem 5.5.2), we know that x is optimal for P and y is optimal for D if and only if $z^* = c^\intercal x = b^\intercal y = w^*$. Hence, since x and y are optimal for P and D, respectively, the chain of inequalities is a chain of equalities. Therefore, $c^\intercal x = b^\intercal y$ is equivalent to $c^\intercal x = y^\intercal Ax$ and $y^\intercal Ax = y^\intercal b$. Rearranging these expressions, we conclude that $x^\intercal(A^\intercal y - c) = 0$ and $y^\intercal(b - Ax) = 0$. ☐

From Theorem 5.5.3, one can derive the following theorem, which provides a simple way to check the otimality of a given solution.

Theorem 5.5.4. *A feasible solution x for the primal P is optimal if and only if there exists a feasible solution y for the dual of P such that*

$$\begin{array}{ll}
\sum_{i=1}^m a_{i,j}y_i = c_j & \text{if } x_j > 0 \text{ and} \\
y_i = 0 & \text{if } \sum_{j=1}^n a_{i,j}x_j \neq b_i\,.
\end{array}$$

Example 5.5.1. Consider the following linear program.

Primal P

$$\begin{aligned}
&\min z = -2x_1 - 3x_2\\
&\text{Subject to}\\
&(1) \quad 2x_1 + x_2 \leq 16\\
&(2) \quad -x_1 + 2x_2 \geq 20\\
&(3) \quad -3x_1 + 2x_2 \geq 10\\
&\qquad x_1, x_2 \geq 0
\end{aligned}$$

We want to show that the solution the solution $(x_1, x_2) = (0, 16)$ is optimal for P. To compute its dual D, we first transform P in standard form.

Primal P (in standard form)

$$\begin{aligned}
&\max z = 2x_1 + 3x_2\\
&\text{Subject to}\\
&(1) \quad 2x_1 + x_2 \leq 16\\
&(2) \quad x_1 - 2x_2 \leq -20\\
&(3) \quad 3x_1 - 2x_2 \leq -10\\
&\qquad x_1, x_2 \geq 0
\end{aligned}$$

Then, we write its dual.

Dual D

$$\begin{aligned}
&\min w = 16y_1 - 20y_2 - 10y_3\\
&\text{Subject to}\\
&(1) \quad 2y_1 + 2y_2 + 3y_3 \geq 2\\
&(2) \quad y_1 - 2y_2 - 2y_3 \geq 3\\
&\qquad y_1, y_2, y_3 \geq 0
\end{aligned}$$

Now, we apply Theorem 5.5.4: we construct a solution to the dual from the given solution to the primal; if it is feasible, then the solution to the primal is optimal.

Since $x_2 > 0$, we get that $y_1 - 2y_2 - 2y_3 = 3$. Furthermore, as $x_1 - 2x_2 \neq -20$ and $3x_1 - 2x_2 \neq -10$ for $(x_1, x_2) = (0, 16)$, we get that $y_2 = y_3 = 0$. Hence, $y_1 = 3$. It remains to check whether $(y_1, y_2, y_3) = (3, 0, 0)$ is a feasible solution to the dual D. As $2y_1 = 6 \geq 2$ and $y_1 \geq 3$, the solution $(3, 0, 0)$ is a feasible to D. By Theorem 5.5.4, we conclude that $(x_1, x_2) = (0, 16)$ is an optimal solution to P.

Note that $w = 16 \times 3 = 48$ and $z = 3 \times 16 = 48$; in other words, the solution x for the primal gives a solution y for the dual such that both objective functions values are equal. Thus, the sufficiency condition of Theorem 5.5.4 follows from Theorem 5.5.2: we have optimality of the solutions $(0, 16)$ for P and $(3, 0, 0)$ for D since their objective function values are equal.

Example 5.5.2. We want to solve the following linear program.

Linear program P

$$\begin{aligned} &\max z = 5x_1 + 6x_2 + 9x_3 + 8x_4 \\ &\text{Subject to} \\ &(1) \quad x_1 + 2x_2 + 3x_3 + x_4 \leq 5 \\ &(2) \quad x_1 + x_2 + 2x_3 + 3x_4 \leq 3 \\ &\qquad x_1, x_2, x_3, x_4 \geq 0 \end{aligned}$$

We can solve it with the simplex algorithm: first transforming P in canonical form, then finding a feasible basic solution, and finally finding the optimal solution (if any).

Another option is to solve its dual. Since P has only two constraints, then its dual has only two variables; so the graphical method could be used to solve it.

Dual D of P

$$\begin{aligned} &\min w = 5y_1 + 3y_2 \\ &\text{Subject to} \\ &(1) \quad y_1 + y_2 \geq 5 \\ &(2) \quad 2y_1 + y_2 \geq 6 \\ &(3) \quad 3y_1 + 2y_2 \geq 9 \\ &(4) \quad y_1 + 3y_2 \geq 8 \\ &\qquad y_1, y_2 \geq 0 \end{aligned}$$

Notice that some constraints of the dual are redundant. For instance $(1) + (2) = 3y_1 + 2y_2 \geq 5 + 6 = 11$, and thus constraint (3) is redundant.

Suppose that we are given the feasible solution $(1, 2, 0, 0)$ for P, which has value $z = 5 + 6 \times 2 = 17$. To decide whether this is an optimal solution, we use Theorem 5.5.4. Since only x_1 and x_2 are positive in the solution, the first two constraints of D must be equalities: $y_1 + y_2 = 5$ and $2y_1 + y_2 = 6$. We readily obtain that $y_1 = 1$ and $y_2 = 4$. Since all constraints are respected for these values, we know that this is a feasible solution to the dual and therefore that the solution $(x_1, x_2, x_3, x_4) = (1, 2, 0, 0)$ is optimal. Note also that the value of w for $(y_1, y_2) = (1, 4)$ is $w = 5 \times 1 + 3 \times 4 = 17$, as expected.

Exercise 30. Consider the following linear program.

Linear program P

$$\begin{aligned} \max z = {} & 2x_1 + 2x_2 - x_3 \\ \text{Subject to} & \\ (1) \quad & 2x_1 + x_2 + x_3 \leq 16 \\ (2) \quad & x_2 - x_3 \geq 8 \\ (3) \quad & 2x_1 + 2x_2 + 2x_3 = 20 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Apply Theorem 5.5.4 to show that $(x_1, x_2, x_3) = (2, 8, 0)$ is an optimal solution for P.

<u>Solution</u>: We already have computed the standard form of P and its dual D in Exercise 29.

P in standard form

$$\begin{aligned} \max z = {} & 2x_1 + 2x_2 - x_3 \\ \text{Subject to} & \\ (1) \quad & 2x_1 + x_2 + x_3 \leq 16 \\ (2) \quad & -x_2 + x_3 \leq -8 \\ (3) \quad & 2x_1 + 2x_2 + 2x_3 \leq 20 \\ (4) \quad & -2x_1 - 2x_2 - 2x_3 \leq -20 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Dual D of P

$$\begin{aligned} \min w = {} & 16y_1 - 8y_2 + 20y_3 - 20y_4 \\ \text{Subject to} & \\ (1) \quad & 2y_1 + 2y_3 - 2y_4 \geq 2 \\ (2) \quad & y_1 - y_2 + 2y_3 - 2y_4 \geq 2 \\ (3) \quad & y_1 + y_2 + 2y_3 - 2y_4 \geq -1 \\ & y_1, y_2, y_3, y_4 \geq 0 \end{aligned}$$

As $x_1 > 0$ and $x_2 > 0$, Theorem 5.5.4 implies that constraints (1) and (2) in D, corresponding respectively to variables x_1 and x_2, must be tight, that is, must reach equality with their bounds. So we know that $2y_1 + 2y_3 - 2y_4 = 2$ and $y_1 - y_2 + 2y_3 - 2y_4 = 2$. Furthermore, we observe that in the primal P, constraint (1) is not tight (with respect to the solution) since $2 \times 2 + 8 + 0 = 12 < 16$;

hence $y_1 = 0$. However, constraints (2), (3) and (4) in the primal P are tight:

$$\begin{aligned}
&(2) \quad -8 + 0 = -8\\
&(3) \quad 2 \times 2 + 2 \times 8 = 20\\
&(4) \quad -2 \times 2 + -2 \times 8 = -20\,,
\end{aligned}$$

and hence we cannot derive anything using Theorem 5.5.4.

It remains to solve the following system of equations:

$$\begin{aligned}
&2y_1 + 2y_3 - 2y_4 = 2\\
&y_1 - y_2 + 2y_3 - 2y_4 = 2\\
&y_1 = 0\,,
\end{aligned}$$

which gives us $(y_1, y_2, y_3, y_4) = (0, 0, y_4 + 1, y_4)$.[4] Fixing, for instance, y_4 to 0, we obtain the solution $(0, 0, 1, 0)$, which is a feasible solution to D. Thus, Theorem 5.5.4 implies that the solution $(x_1, x_2, x_3) = (2, 8, 0)$ is an optimal solution for P.

[4]There are infinitely many feasible solutions.

Bibliography

[1] R. Bellman, “On a routing problem,” *Quart. Appl. Math.*, vol. 16, pp. 87–90, 1958. DOI: 10.1090/qam/102435.

[2] C. Berge, “Two theorems in graph theory,” *Proc. Nat. Acad. Sci. U.S.A.*, vol. 43, pp. 842–844, 1957. DOI: 10.1073/pnas.43.9.842.

[3] R. G. Bland, “New finite pivoting rules for the simplex method,” *Math. Oper. Res.*, vol. 2, no. 2, pp. 103–107, 1977. DOI: 10.1287/moor.2.2.103.

[4] S. Cook, “The P versus NP problem,” in *The millennium prize problems*, Clay Math. Inst., Cambridge, MA, 2006, pp. 87–104.

[5] S. A. Cook, “The complexity of theorem-proving procedures,” in *Proceedings of the third annual ACM symposium on Theory of computing*, 1971, pp. 151–158.

[6] G. B. Dantzig and D. R. Fulkerson, “On the max-flow min-cut theorem of networks,” in *Linear inequalities and related systems*, Princeton University Press, Princeton, N. J., 1956, pp. 215–221.

[7] G. B. Dantzig, “Maximization of a linear function of variables subject to linear inequalities,” in *Activity Analysis of Production and Allocation*, John Wiley & Sons, Inc., New York, N. Y.; Chapman & Hall, Ltd., London, 1951, pp. 339–347.

[8] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numer. Math.*, vol. 1, pp. 269–271, 1959. DOI: 10.1007/BF01386390.

[9] E. A. Dinic, “Algorithm for solution of a problem of maximum flow in networks with power estimation,” in *Soviet Math. Doklady*, vol. 11, 1970, pp. 1277–1280.

[10] J. Edmonds and R. M. Karp, “Theoretical improvements in algorithmic efficiency for network flow problems,” *Journal of the ACM (JACM)*, vol. 19, no. 2, pp. 248–264, 1972.

[11] L. R. Ford Jr. and D. R. Fulkerson, “Maximal flow through a network,” *Canadian J. Math.*, vol. 8, pp. 399–404, 1956. DOI: 10.4153/CJM-1956-045-5.

[12] L. R. Ford Jr, “Network flow theory,” Rand Corp Santa Monica Ca, Tech. Rep., 1956.

[13] G. Frobenius, *Über zerlegbare determinanten.* Reimer, 1917.

[14] D. Gale and L. S. Shapley, “College Admissions and the Stability of Marriage,” *Amer. Math. Monthly*, vol. 69, no. 1, pp. 9–15, 1962. DOI: 10.2307/2312726.

[15] Z. Galil, “Efficient algorithms for finding maximum matching in graphs,” *Comput. Surveys*, vol. 18, no. 1, pp. 23–38, 1986.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979.

[17] P. Hall, “On representatives of subsets,” *J. London Math. Soc*, vol. 10, no. 1, pp. 26–30, 1935.

[18] J. E. Hopcroft and R. M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM J. Comput.*, vol. 2, pp. 225–231, 1973. DOI: 10.1137/0202019.

[19] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations (Proc. Sympos., IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., 1972)*, 1972, pp. 85–103.

[20] A. V. Karzanov, "The problem of finding the maximal flow in a network by the method of preflows," *Dokl. Akad. Nauk SSSR*, vol. 215, pp. 49–52, 1974.

[21] A. V. Karzanov, "An exact estimate of an algorithm for finding a maximum flow, applied to the problem "on representatives"," *Problems in Cybernetics*, vol. 5, pp. 66–70, 1973.

[22] D. König, "Gráfok és mátrixok," *Matematikai és Fizikai Lapok*, vol. 38, pp. 116–119, 1931.

[23] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Res. Logist. Quart.*, vol. 2, pp. 83–97, 1955. DOI: 10.1002/nav.3800020109.

[24] H. W. Kuhn, "Variants of the Hungarian method for assignment problems," *Naval Res. Logist. Quart.*, vol. 3, 253–258 (1957), 1956. DOI: 10.1002/nav.3800030404.

[25] R. E. Ladner, "On the structure of polynomial time reducibility," *J. Assoc. Comput. Mach.*, vol. 22, pp. 155–171, 1975. DOI: 10.1145/321864.321877.

[26] L. A. Levin, "Universal enumeration problems," *Problemy Peredači Informacii*, vol. 9, no. 3, pp. 115–116, 1973.

[27] K. Menger, "Zur allgemeinen kurventheorie," *Fundamenta Mathematicae*, vol. 10, no. 1, pp. 96–115, 1927.

[28] H. Minkowski, *Geometrie der Zahlen.* Johnson Reprint Corp., New York-London, 1968, pp. vii+256.

[29] E. F. Moore, "The shortest path through a maze," in *Proc. Internat. Sympos. Switching Theory 1957, Part II*, Harvard Univ. Press, Cambridge, Mass., 1959, pp. 285–292.

[30] J. B. Orlin, "A faster strongly polynomial minimum cost flow algorithm," *Oper. Res.*, vol. 41, no. 2, pp. 338–350, 1993. DOI: 10.1287/opre.41.2.338.

[31] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity.* Courier Corporation, 1998.

[32] C. H. Papadimitriou, "Computational complexity," in *Encyclopedia of Computer Science.* GBR: John Wiley and Sons Ltd., 2003, pp. 260–265. DOI: 10.5555/1074100.1074233.

[33] J. Petersen, "Die Theorie der regulären graphs," *Acta Math.*, vol. 15, no. 1, pp. 193–220, 1891. DOI: 10.1007/BF02392606.

[34] A. Schrijver, *Combinatorial Optimization. Polyhedra and Efficiency (3 volumes).* Springer-Verlag, 2003, vol. 24.

[35] A. Shimbel, "Structure in communication nets," in *Proceedings of the symposium on information networks, New York, April, 1954*, Polytechnic Institute of Brooklyn, Brooklyn, N.Y., 1955, pp. 199–203.

[36] N. Tomizawa, "On some techniques useful for solution of transportation network problems," *Networks*, vol. 1, pp. 173–194, 1971/72. DOI: 10.1002/net.3230010206.

[37] C. A. Tovey, “Tutorial on computational complexity,” *Interfaces*, vol. 32, no. 3, pp. 30–61, 2002.

[38] U. Zwick, “The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate,” *Theoret. Comput. Sci.*, vol. 148, no. 1, pp. 165–170, 1995. DOI: 10.1016/0304-3975(95)00022-O.