Operations Research

Axel Parmentier

January 8, 2020
École des Ponts Paristech – Academic year 2019-2020

Contents

Chapter 1

Introduction

1.1 Operations research

Any student with some scientific training is accustomed to model phenomenons of our world using mathematical equations, be it materials behaviors, planets orbits, chemical reactions speed, etc. Mathematical modeling can also be used to understand and improve "operations" such as the planning of workforce of a large organization, the design of an electric network, or the choice of an investment portfolios. "Operations" in OR comes from military operations: OR was invented to plan the Normandy landings, the most complicated operations of its time. According to Rardin and Rardin [1998], "Operations research (OR) is the study of how to form mathematical models of complex engineering and management problems and how to analyze them to gain insight about possible solutions." Operations Research has notably been applied to vehicle routing, workforce management, electricity/water/communication network design, web platforms management, beam choice for large buildings, metabolic networks in biology, productions management, project management, supply chain, revenue management, etc.

Most of the times, Operations Research models are optimization problems of the form

$$\min_{x \in X} f(x).$$

where X is the set feasible solutions x, and f the objective function. If most of you have been optimizing functions since high school, the optimization problems we are going to work with are pretty different from the ones previously considered. For instance, we will see how to seek the shortest route between two cities, say Paris and Toulouse. Then X is the set of routes between Paris and Toulouse, and $f(x)$ evaluates the travel time of route x.

This optimization problem has two main novelties when compared to those you have seen previously. First, new mathematical objects will be considered. Graph theory will notably be helpful. For instance, routes between Paris and Toulouse will be modeled as paths in graphs. And second, the decision variables are discrete, i.e., in $\mathbb{Z}$ or $\{0,1\}$. For instance, consider for instance the aircraft routing problem, which aims at building the sequences of flight legs operated by airplanes so as to cover all the flight legs of an airline at minimum cost. The decision variable x indicating if a plane operates or not a leg will be equal to 0 (leg not operated) or 1 (leg operated), but not to 0.5.

Such models with variables in $\{0,1\}$ optimization problems with a finite set of solutions X. At first sight, you may think that Operations Research problems are therefore easy to solve: if X is finite and non-empty, then there exists an optimal solution, and it suffices to enumerate all the solutions to find the optimal one. But what if X is finite but contain more that 10^{100} solutions, which is frequently the case on industrial applications. And the best computers won't be able to enumerate all the solutions in a year of computing time.

Operations Research is therefore intrinsically linked to combinatorial optimization, the discipline of mathematics that designs optimization algorithms to solve problems on finite but exponentially large sets. Combinatorial algorithms are fairly different from the ones used in continuous optimization because there is nothing like gradient of derivatives on finite sets.

Two elements are crucial when choosing an operations research model. First, the quality of the description of the reality studied, and second the ability to design algorithm to solve the problem obtained. An optimization framework is class of optimization problems. For instance, those who have followed Frédéric Meunier's optimization course last year have studied linear programming, which contains all the problems which aims at minimizing a linear objective under linear constraints. An optimization framework must strike a balance between two contradictory objectives: flexibility, which enables to model a wide range of applications, and structure, which enables to design efficient algorithms. Good frameworks are those that enable to model a wide range of applications while still retaining enough structure to design efficient solution algorithms. When designing an algorithm to solve a problem, we generally do not search in a vacuum. We try to model our problem as a specific case of well-studied optimization framework, and we use algorithms previously developed for this framework.

To use Operations Research in the industry, it is therefore crucial to know the main optimization frameworks used in Operations Research, and to know how to model new applications within these frameworks.

This course will therefore introduce the main optimization frameworks, and aims at developing the following skills:

- The ability to model new applications within operations research main frameworks;
- The ability to design algorithms to solve these problems, and to prove that these algorithms are correct;
- And the ability to evaluate the results obtained using these algorithms.

1.2 How the course will be evaluated

The course is validated using three evaluations.

1. An *MCQ* (multiple-choice questionnaire) at the end of each lecture.
2. A *project* with a modeling and an implementation part. The project will be on the same subject as the Kiro, the operations research Hackathon organized by the *club informatique* (`http://hackathon.enpc.org/`). If you participate at the Kiro, you will be able to use your code to answer the project. You will also have to write a report on the project, with more academic questions on the project.
3. A *final exam*. The only document allowed during the exam is this textbook. No document will be allowed during the resit exam.

Denoting by p the grade obtained at the project, and by e the grade obtained at the exam, the final grade g of the course is

$$g = \begin{cases} \frac{2p+3e}{5} & \text{if } e > \frac{8}{20}, \\ e & \text{otherwise.} \end{cases}$$

To validate the course, g must be greater than or equal to $\frac{10}{20}$. Remark that it is necessary to have a grade superior to $\frac{8}{20}$ at the exam to validate the course.

In case of a failure, you will not be allowed to retake the exam if you have more than one unjustified absence, or if you have not answered to more than one MCQ.

Finally, *students more that 5 minutes late will not be admitted to the lecture*.

1.3 Prerequisites

1.3.1 Mathematics

The mathematical techniques introduced in this lecture will be fairly new for most of the students. Hence, the lecture requires few mathematical prerequisites except linear algebra. However, the concepts introduced require a certain maturity in mathematics. If this should not pose much difficulty to the students from the "concours commun", it may be harder for students with a less mathematical curriculum.

You will know for sure at the end of the first course if you have or not this mathematical maturity. If you do not, you will have to work much harder than the other students to validate the course. We recommend Oakley [2014]'s book to students seeking advices on how to learn university courses in mathematics.

1.3.2 Programming

Furthermore, using operations research in the industry requires minimum programming skills. Indeed, it is common to solve mathematical programs with hundreds of thousands of variables and/or constraints. Without a programming language, writing such programs would be rather tedious. The project of the course requires some implementation. You are free to choose the programming language you prefer for that project. Since Operations Research algorithms intensively use for-loops, compiled languages (`C`, `C++`, `java`, etc.) lead to better performances than interpreted languages (such as `python`, `matlab`, `scilab`). However, to avoid loosing time, we recommend you to use the language you know best.

If you are not familiar with any language, or if you want to learn a new one, we recommend you to use `julia` (`https://julialang.org/`), which is a scripting language with many packages for optimization and running time nearly as good as those of a compiled language. It is probably the best language for small projects in Operations Research. You can also choose to learn `python` or `C++`. Both languages are widely used, and learning them is never a loss of time. But note that `python` leads to poor performances in Operations Research, while `C++` is much harder to learn.

If you are not familiar with any language, we advise you to start learning one now. A tutorial to learn `julia` is available on the following webpage `https://juliaacademy.com/p/intro-to-julia`. Additional resources are available there: `https://julialang.org/learning/`. Tutorials on other languages can be found for instance on `https://openclassrooms.com/`.

1.4 How to use this textbook

Sections starting with 😎 are extracurricular. Symbol 🔍 at the end of an exercise indicates that a solution to this exercise will be given.

This textbook is not finished. Some exercises are written in French. Symbol ◆ indicates when a section is still under work.

1.5 Acknowledgments

We thank Frédéric Meunier for letting us use part of the content of his textbook [Meunier, 2018].

Chapter 2

Problems, complexity, and algorithms

Addressing a question with the tools of Operations Research requires to model is as a problem, and to design algorithms that solve this problem. We have seen in the introduction that the time needed for the execution of an algorithm determines its usability on practical applications. Complexity theory gives tools to evaluate how difficult a problem is and if it can be solved by fast algorithm. Decision problems play a key role in complexity theory. Informally, a decision problem is defined by a certain type of *input*, as well as a *question* answerable by yes or no on this input.

Example 2.1. The following decision problem is considered in the theory of linear programming that is exposed in Chapter 9.

LINEAR PROGRAMMING INEQUALITIES
Input. A matrix $\boldsymbol{A}$ in $\mathbb{Z}^{m\times n}$, a vertex $\boldsymbol{b}$ in $\mathbb{Z}^n$.
Question. Is there an $x \geq 0$ in $\mathbb{Q}^n$ satisfying $\boldsymbol{Ax} = \boldsymbol{b}$

△

Example 2.2. Consider a firm that wants to open $k \in \mathbb{Z}_+$ factories. It has the choice between m possible sites, and a set of n clients. Let a_{ij} indicate the cost of delivering client j from factory i. A site selection S is a subset of $[m]$ of at most k elements that indicates the site that are opened, where $[m]$ denotes $\{1, \dots, m\}$. The cost of a site selection is

$$c(S) = \sum_{j=1}^{n} \min_{i \in S} a_{ij}.$$

FACILITY LOCATION (DECISION VERSION)
Input. Three integers m, n, and k, a distance matrix $A = (a_{ij}) \in \mathbb{R}_+^{m\times n}$, a cost $c_0 \in \mathbb{R}^+$.
Question. Is there a subset $S \subseteq [m]$ such that $|S| \leq k$ and with cost $c(S) \leq c_0$?

A given realization of an input of a problem is called an *instance* of the problem. For example, 17 is an instance of the PARITY problem, for which the answer is no. And

$$m = 4, \quad n = 5, \quad k = 2, \quad A = \begin{pmatrix} 2 & 5 & 4 & 9 & 15 \\ 7 & 3 & 11 & 6 & 12 \\ 12 & 15 & 1 & 3 & 3 \\ 7 & 5 & 4 & 5 & 5 \end{pmatrix}, \quad c_0 = 15 \tag{2.1}$$

is an instance of FACILITY LOCATION for which it is easy to check that the answer is yes. Indeed $S = \{1, 3\}$ gives an adequate site selection. △

Computer scientists have struggled without success during decades to find "good" algorithms to compute the solution of some important problems. For instance, no "good" algorithm could be found for the FACILITY LOCATION PROBLEM, or for the TRAVELING SALESMAN PROBLEM which we informally state below.

> Given a set of n clients to be visited by a salesman, find the order in which the salesman should visit the client that minimizes the total distance the salesman will travel.

Complexity theory emerged from the acknowledgment that some problem are intrinsically difficult and it may not be possible to find "good" algorithm. This requires to formalize the notions of "problem" and "solution algorithm". The foundations of complexity theory are quite technical and out of the scope of this lecture. Hence, we are not completely formal in the introduction of some notions, and provide the rigorous definitions for the interested reader in extra-curricular sections.

2.1 Algorithm and decision problems

2.1.1 Algorithm

Tasks can be operated on computers using algorithms. We now give an informal idea of what an algorithm is which suffices to understand this lecture. A formal definition is given in the extra-curricular Section 2.1.3.

The tasks operated by a computer can be decomposed into elementary operations such as reading or writing a bit (binary digit) in the memory, or performing arithmetic operations. An instruction is an unambiguous description of what a computer the elementary operations a computer should execute given its current state and memory. An *algorithm* ϕ is a collection of instructions indicating unambiguously what the computer should do given all the possible states and content of the memory. A step of an algorithm is the realization of an instruction. Given an input which specifies the initial state of a computer, an execution of an algorithm is the sequence of state and operations operated by the computer if it follows the instructions of the algorithm. As an instruction may modify the state and the memory of a computer, or stop the algorithm and return the result, the number of steps in the sequence depends on the instructions realized and hence on the input. The sequence can be of infinite length if no instruction terminating the algorithm is met. The *time complexity* $\mathsf{time}(\phi, x) \in \mathbb{Z}_+$ of an algorithm ϕ on an input x is the number of steps in the sequence. An algorithm *converges after a finite number of iterations* on x if $\mathsf{time}(\phi, x) < +\infty$.

Characterizing the input and the output of an algorithm ϕ requires to be slightly more formal. An *alphabet* A is a finite set of at least two elements. Practically, a computer works on the binary digits alphabet $\{0, 1\}$. A *string* over A is a finite sequence of elements of A. A string is possibly empty. The *length* $\mathsf{size}(x)$ of a string x is the number of elements in the sequence. We denote by A^* the set of strings over A. A *language* over A is a subset of A^*. An element of a language is called a *word*.

Let S and T be two languages on an alphabet A. An algorithm ϕ from S to T takes in input a word of S and returns a word of T. Given an input x in S, if $\mathsf{time}(\phi, x) < +\infty$, then ϕ returns an output denoted by $\mathsf{output}(\phi, x)$. The quantity $\mathsf{output}(\phi, x)$ is not defined if $\mathsf{time}(\phi, x) = +\infty$. An algorithm ϕ is a *polynomial time* algorithm if there exists a polynomial P such that $\mathsf{time}(\phi, x) = O(P(\mathsf{size}(x)))$ for any x in S.

Let A be an alphabet, S and T be two languages on an alphabet A, and $f : S \to T$ a map. Then algorithm ϕ *computes* f if $\mathsf{time}(\phi, x) < +\infty$ for any x and $\mathsf{output}(\phi, x) = f(x)$ for any x

in S. A function f is *computable in polynomial time* if there exists polynomial time algorithm that computes it.

2.1.2 Decision problems

Without loss of generality, we now assume that we are on the alphabet $A = \{0, 1\}$

Definition 2.1. *(Simplified)* A decision problem *$\mathcal{P}$ is a pair (X, Y), where X is a language, and $Y \subseteq X$. Language X is the* input*, and its elements are the* instances*. Elements of Y are the instances for which the answer is* yes*, and $X \backslash Y$ those for which the answer is* no*.*

To make this definition coincide with the informal definition given at the beginning of the chapter, we must choose a language $\mathcal{L}$ that is in bijection with the input set, and such that the size of the encoding is minimal. This definition is simplified because X must satisfy an additional condition, which is satisfied by all the problems we will consider in this lecture. This condition is given in Section 2.1.3.

A *solution algorithm* for a decision problem (X, Y) is an algorithm computing the function $f : X \to \{\mathsf{yes}, \mathsf{no}\}$ that associates yes to instances in Y and no to instances in $X \backslash Y$.

Practically, we will use the informal definition given at the beginning of the chapter. We just have to remember that the size needed to encode an integer is $\lceil 1 + \log \rceil$.

2.1.3 Formal definitions: Turing Machine 😎

We now introduce the notion of *Turing machine*, which is a formal definition of an algorithm. Although it seems to be a restricted definition of an algorithm, it is quite powerful and suffices to the analysis of most algorithms, and many alternative "machines" that looks richer have been proved to be equivalent. It is therefore the most widely used mathematical model of an algorithm on a computer. Some more general machines such as random-access stored-program machine are also used by specialists of complexity theory.

Informally, a Turing machine is a composed of tape, that can be seen as a sequence of cells. Each cell contains a symbol of a given alphabet, or the blank symbol ␣ that separates words. The tape is indefinitely extensible. A head can read and write cells of the tape, and move along the tape. A state register stores the state of the machine. The machine can only take a finite number of states. At a given step, the machine reads the content of the tape at the level of the head, and executes a fixed sequence of instructions that depend only on the content of the cell and of the current state of the machine. These instructions are of three types: erase or write a symbol, move the head by a given number of cells, or change of state. These instructions are contained in the table of instructions. The machine starts in an initial state. The initial content of the tape is the input of the algorithm, completed by blank symbols ␣. While the Machine is not in a final state, the machine reads the content of the current cell and executes the corresponding instructions in the table of instructions.

A *Turing machine* ϕ is therefore a 7-tuple $(Q, A, b, \Sigma, \delta, q_{\text{init}}, F)$, where Q is the non-empty and finite set of states, A is the alphabet which does not contain the blank symbol ␣, b is the blank symbol ␣, $\Sigma \subseteq A \backslash \{b\}$ is the set of input symbols that can be initially on the tape, $q_{\text{init}} \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and

$$\delta : Q \times A \to Q \times A \times \{-1, 0, 1\}$$

is the table of instructions.

Let $\overline{A} = A \cup \{␣\}$. The *computation* of ϕ on input x in $\Sigma^{\mathsf{size}(x)}$ is the finite or infinite sequence $(q_i, s_i, \pi_i)_i$ in $Q \times \overline{A}^* \times \mathbb{Z}$, where q_i, s_i, and π_i are respectively the state, the content of the tape, and the position of the head at step i, defined recursively as follows.

- $q_0 = q_{\text{init}}$, $s_0(j) = x(j)$ if $0 \leq j < \mathsf{size}(x)$, and $s_0(j) = \llcorner\!\lrcorner$ otherwise, and $\pi_0 = 0$.

- If q_i is in F, then this is the end of the sequence, and we define $\mathsf{time}(\phi, x) = i$. Furthermore, let $k = \min\{j : s_i(j) = \llcorner\!\lrcorner\}$, and $\mathsf{output}(\phi, x)$ be the string t in A^k defined by $t(j) = s(j)$ for $0 \leq j < k$.

- Otherwise, with $(q', a, m) = \delta(q_i, s_i(\pi_i))$, we have $q_{i+1} = q'$, $s_{i+1}(\pi_i) = a$, and $\pi_{i+1} = \pi_i + m$.

If the sequence is infinite, we set $\mathsf{time}(\phi, x) = +\infty$, and $\mathsf{output}(\phi, x)$ is undefined. We say that ϕ converges after a finite number of steps on a language S on A if $\mathsf{time}(\phi, x) < \infty$ for any input x in S.

Exercise 2.1. What does the algorithm described by the following Turing Machine ? △

Turing machines are algorithms that enable to compute functions. Let A be an alphabet, S and T be two languages on A, and $f : S \to T$ a map. Then ϕ *computes* f if ϕ converges after a finite number of steps on S and $\mathsf{output}(\phi, x) = f(x)$ for any x in S. A Turing machine ϕ is a *polynomial time* Turing machine if there exists a polynomial P such that $\mathsf{time}(\phi, x) = O(P(\mathsf{size}(x)))$ for any x in S. A function f is *computable* if there exists a Turing machine that computes it, and *computable in polynomial time* if there exists polynomial time Turing machine that computes it.

Turing machine can notably be used to check if an element belongs to a language. A Turing machine ϕ *decides* a language L on A if $\mathsf{time}(\phi, x) < \infty$ for any x in A^* and $\mathsf{output}(\phi, x) = \mathsf{yes}$ if $x \in L$ and no otherwise. A language is *decidable* if there exists a Turing machine that decides it, and *decidable in polynomial time* if there exists a polynomial time Turing machine that decides it.

Remark 2.1. △

Definition 2.2. A decision problem *is a pair* (X, Y) *where* X *is a language decidable in polynomial time, and* $Y \subseteq X$. *Language* X *is the input, and its elements are the instances. Elements of* Y *are the instances for which the answer is* yes, *and* $X \backslash Y$ *those for which the answer is* no.

An *algorithm* for a decision problem (X, Y) is a Turing machine computing the function $f : X \to \{\mathsf{yes}, \mathsf{no}\}$ that associates yes to instances in Y and no to instances in $X \backslash Y$.

Practically, when we study an algorithm, we never specify the precise Turing machine describing it. Indeed, we only need to know the size of the input, that is, the size of the string that would be required to encode the input, and an upper bound on the number of steps. To compute this upper bound, it suffices to describe the algorithm in terms of elementary operations, such as arithmetic operations, which are known to be implementable in a fixed/polynomial number of steps on a Turing Machine.

2.2 Complexity classes $\mathscr{P}$ and $\mathscr{NP}$

Recall that given two functions f and g, function f is a $O(g(x))$ if there exists a number $M > 0$ such that $f(x) \leq Mg(x)$ for any x in X. A *polynomial algorithm* ϕ on an input language X is algorithm for which there is a polynomial P such that the time complexity of ϕ is in $O(P(\mathsf{size}(x)))$ for any instance x in X. Hence, a polynomial solution algorithm for a decision problem is a solution algorithm such that there is a polynomial P satisfying $\mathsf{time}(\phi, x) = O\big(P(\mathsf{size}(x))\big)$, where x takes its values in the instances the input language X.

Definition 2.3. *A* polynomial problem *is a decision problem for which there exists a polynomial solution algorithm. We denote by $\mathscr{P}$ the class of polynomial problems.*

For instance, the LINEAR PROGRAMMING INEQUALITIES problem stated in the introduction of the chapter is in $\mathscr{P}$.

Skill 2.1. *How to show that a problem is in $\mathscr{P}$?*

It suffices to exhibit a polynomial algorithm

For difficult problems such as FACILITY LOCATION, we are not able to exhibit a polynomial algorithm proving that they belong to $\mathscr{P}$. But theses problems can be proved to be in a larger class called $\mathscr{NP}$. A problem is in $\mathscr{NP}$ if we can check in polynomial time a "certificate". For instance, for the facility location problem, a certificate is a site selection S, and we can check in polynomial time that $c(S) \leq c_0$.

Definition 2.4. *A decision problem $\mathcal{P} = (X, Y)$ is in $\mathscr{NP}$ if there exists a polynomial P and a decision problem $\mathcal{P}' = (X', Y')$ in $\mathscr{P}$ such that*

$$X' = \left\{x\#c \colon x \in X, c \in A^{P(\lfloor \mathsf{size}(x) \rfloor)}\right\}$$

and

$$Y = \left\{y \in X \colon \textit{there exists a string } c \textit{ in } A^{P(\lfloor \mathsf{size}(y) \rfloor)} \textit{such that } y\#c \in Y'\right\},$$

where $a\#b$ denotes the concatenation of strings a and b.

A string $y\#c \in Y'$ is called a *certificate* for y, as string c enables to prove y in Y. An algorithm for (X', Y') is called a *certificate checking* algorithm.

Proposition 2.5. $\mathscr{P} \subseteq \mathscr{NP}$.

We do not know if $\mathscr{P} = \mathscr{NP}$, but $\mathscr{P} \neq \mathscr{NP}$ is one of the most famous and widely believed conjecture in computer science. The Clay Mathematics Institute offers 1 million dollars to the first person that will solve this conjecture.

Exercise 2.2. Prove that FACILITY LOCATION belongs to $\mathscr{NP}$. △

As $\mathscr{NP}$ contains $\mathscr{P}$ as well as difficult problems such that FACILITY LOCATION, proving that a problem $\mathcal{P}$ belongs to $\mathscr{NP}$ is not a good indication of its difficulty. We therefore introduce a notion of difficulty that amounts to say that "problem $\mathcal{P}$ is as difficult as the most difficult problems in $\mathscr{NP}$".

A *reduction* of a problem $\mathcal{P} = (X, Y)$ to a problem $\mathcal{P}' = (X', Y')$ is a mapping $f : X \to X'$ such that

$$x \in Y \Leftrightarrow f(x) \in Y' \quad \text{for any } x \text{ in } X.$$

It is a *polynomial reduction* if there exist a polynomial P such that $\mathsf{size}\big(f(x)\big) = O\big(P(\mathsf{size}(x))\big)$. We say that a problem $\mathcal{P}$ *polynomially reduces* to a problem $\mathcal{P}'$, or simply *reduces* to $\mathcal{P}_2$, if there exists a polynomial reduction of $\mathcal{P}$ to $\mathcal{P}'$. We say that two problems $\mathcal{P}$ and $\mathcal{P}'$ are *polynomially equivalent* if $\mathcal{P}$ reduces to $\mathcal{P}'$ and $\mathcal{P}'$ reduces to $\mathcal{P}$. The following proposition shows the interest of these definitions.

Proposition 2.6. *Let $\mathcal{P}$ and $\mathcal{P}'$ be two decision problems such that $\mathcal{P}$ reduces to $\mathcal{P}'$. Then if $\mathcal{P}'$ is in $\mathscr{P}$, then $\mathcal{P}$ is in $\mathscr{P}$.*

Hence, the fact that $\mathcal{P}$ reduces to $\mathcal{P}'$ means that $\mathcal{P}'$ is at least as hard as $\mathcal{P}$. This enables us to define the class of the "hardest" problems in $\mathscr{NP}$.

Definition 2.7. *A problem $\mathcal{P}$ is* $\mathscr{NP}$-complete *if it is in $\mathscr{NP}$ and any problem $\mathcal{P}'$ in $\mathscr{NP}$ polynomially reduces to $\mathcal{P}$.*

Proposition 2.8. *Let $\mathcal{P}$ be a problem. If $\mathcal{P}$ is in $\mathscr{NP}$, and there exists an $\mathscr{NP}$-complete problem $\mathcal{P}'$ that polynomially reduces to $\mathcal{P}$, then $\mathcal{P}$ is $\mathscr{NP}$-complete.*

Once some problems has been proved to be $\mathscr{NP}$-complete, Proposition 2.8 can be used to prove new $\mathscr{NP}$-completeness result. But for this kind of proofs to work, we need to prove using another kind of arguments that a first problem is $\mathscr{NP}$-complete. This has been done by Cook [1971], who proved that the SATISFIABILITY problem is $\mathscr{NP}$-complete.

Skill 2.2. *How to prove that a problem $\mathcal{P}$ is $\mathscr{NP}$ complete?*

The proof is in two steps. First, prove that $\mathcal{P}$ is in $\mathscr{NP}$. And second, reduce an $\mathscr{NP}$-complete problem to $\mathcal{P}$.

2.3 Optimization problems

Informally an optimization problem is defined by and input, an output, and an objective to minimize or maximize. For instance, here is the optimization version of the facility location problem.

FACILITY LOCATION
Input. Three integers m, n, and k, a cost matrix $A = (a_{ij})$ in $\mathbb{Q}_+^{m\times n}$.
Output. A site selection $S \subseteq [m]$ satisfying $|S| \leq k$ of minimum $\sum_{j=1}^n \min_{i\in S} a_{ij}$.

As optimization problems are not decision problems, they cannot be in $\mathscr{NP}$. However, they can be at least as hard as any problem in $\mathscr{NP}$. This paragraph introduces the corresponding notion. Again, we will need a formal definition of optimization problems. The details of the definition below are not required for the understanding of the lecture.

Definition 2.9. *An* optimization problem *$\mathcal{P}$ is a quadruple $(X, (S_x)_{x\in X}, c, \mathrm{goal})$ where*

- *X is a language over $\{0,1\}$ decidable in polynomial time.*
- *$S_x \subseteq \{0,1\}^*$ for each x in X, and there exists a polynomial P satisfying $\mathsf{size}(y) \leq \mathsf{size}(x)$ for each $y \in S_x$, and the languages $\{(x,y) \colon x \in X, y \in S_x\}$ and $\{x \in X \colon S_x = \emptyset\}$ are decidable in polynomial times.*
- *$c : \{(x,y) \colon x \in X, y \in S_x\} \to \mathbb{Q}$ is a polynomially computable function.*
- $\mathrm{goal} \in \{\min, \max\}$.

Elements of X are the *instances* of the problem. Given an instance x, the elements S_x is the set of *feasible solutions* and is denoted by $\mathrm{Sol}(x)$. The *value* of an instance is $\mathrm{val}(x) = \mathrm{goal}\{c(x,y) \colon x \in X, y \in S_x\}$. The *optimal solutions* are the elements y of S_x that $c(x,y) = \mathrm{val}(x)$. We denote by $\mathrm{Opt}(x)$ the set of optimal solutions.

A decision problem $\mathcal{P}_1 = (X_1, Y_1)$ *polynomially reduces* to an optimization problem $\mathcal{P} = (X_2, (S_x)_{x \in X}, c, \text{goal})$ if there exists a polynomially computable function $f : X_1 \to X_2 \times \mathbb{Q}$ and a polynomial P such that $\mathsf{size}(f(x)) = O\big(P(\mathsf{size}(x))\big)$, and $x \in Y$ if and only if $\text{val}(x') \leq c$ where $(x', c) = f(x)$.

Definition 2.10. *An optimization problem $\mathcal{P}$ is $\mathscr{NP}$-hard if all the problems in $\mathscr{NP}$ polynomially reduce to $\mathcal{P}$.*

Skill 2.3. *Proving that an optimization problem is $\mathscr{NP}$-hard*

To prove the $\mathscr{NP}$-hardness of an minimization problem,

OPTIMIZATION PROBLEM
Input. An x in X
Output. A feasible solution y in Y_x of minimum $c(x)$

start by considering the decision version of the problem

DECISION PROBLEM
Input. An x in X, a rational c_0 in $\mathbb{Q}$.
Question. Is there a solution y in Y_x such that $c(x) \leq c_0$

and then prove that an $\mathscr{NP}$-complete problem $\mathcal{P}$ reduces to DECISION PROBLEM. It suffices to replace "min" and "$\leq$" by "max" and "$\geq$" to deal with maximization problems. A collection of $\mathscr{NP}$-complete problems $\mathcal{P}$ will be introduced in the following chapter. When none of these are easily reduced to the decision problem we are considering, a good resource is Wikipedia's list of $\mathscr{NP}$-complete problems.

2.4 Further readings

Books on complexity theory.

- Ausiello et al. [2012]
- Garey and Johnson [2002]
- Papadimitriou [2003]

2.5 Exercises

Many decision and optimization problems are introduced, and exercises on their complexity are in the subsequent chapters. See notably the exercises of Chapter 3 on graphs.

Part I

Graphs

Chapter 3

Graphs

This chapter introduces basic notions on graphs, as well as terminology and notations.

3.1 Undirected graphs

3.1.1 Definition

A *graph* G or *undirected graph* is a pair (V, E) where V is a finite set and E is a collection of unordered pairs of V. Elements of v are called *vertices* (or *nodes*), and elements of E edges. The cardinal $|V|$ is generally denoted by n, and $|E|$ by m.

As a graph G can be described by the list of its edges, where each vertex and edge is identified by an integer, the size of a graph $G = (V, E)$ in the sense of complexity theory is in $O\big(m \log(n) + n \log(m)\big)$.

Remark 3.1. More formally, E is a multiset on V^2. A *multiset* M on a set S is a pair (S, m_M) where m_M is a map from S to Z_+. Set S is called the *universe*, and m_M is the *multiplicity* maps. The *support* of M is the set of s in S such that $m_M(s) > 0$. △

There can be edges of the form (v, v). Such edges are called *loops*. A given edge (u, v) can be several times in E. The number of times this occurs in E is called its *multiplicity*. A *simple graph* is a graph with no loops and such that edges in E have multiplicity one. The *complement* $\overline{G}$ of a graph $G = (V, E)$ is the simple graph (V, E') where $E' = \big\{(u, v) \colon u \neq v \text{ and } (u, v) \notin E\big\}$.

Two vertices u and v are *adjacent* if the unordered pair (u, v) belongs to E. A vertex u is a *neighbor* of v if u and v are adjacent. We denote by

$$N(v) \quad \text{the set of neighbors of } v.$$

An edge e is *incident* with a vertex v if e is of the form (u, v) for some u in E. We denote by

$$\delta(v) \quad \text{the set of edges incident with } v.$$

An edge e is incident with a set of vertices U if $e = (u, v)$ for some $u \in U$ and $v \notin U$. We denote by $N(U)$ the set of edges that are incident to U.

The *degree* $\deg_G(v)$ of a vertex v is the number of edges incident with v. We denote it by $\deg(v)$ when graph G is clear from the context.

Exercise 3.1. Prove that $\sum_{v \in V} \deg(v)$ is an even number. △

A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. Given $V' \subseteq V$, the *subgraph induced* by V' is the graph $G' = (V', E')$ where E' is the collection of edges (u, v)

A *complete graph* is a simple graph such that all the vertices are connected. We denote by K_n the complete graph with n vertices.

Exercise 3.2. How many edges has K_n? △

3.1.2 Adjacency and incidence matrix

The *adjacency* matrix of a graph $G = (V, E)$ is the matrix A of $\mathbb{Z}^n \times \mathbb{Z}^n$ such that

$$A_{uv} = \text{number of edges connecting } u \text{ and } v.$$

Remark that A is symmetric.

The *incidence matrix* is the $\mathbb{Z}^{|V|} \times \mathbb{Z}^{|E|}$ matrix such that

$$B_{ve} = \begin{cases} 0 & \text{if } v \notin e, \\ 1 & \text{if } v \in e \text{ and } e \neq (v, v), \\ 2 & \text{if } e = (v, v). \end{cases}$$

3.1.3 Paths, connectedness

A *path* P in a graph $G = (V, E)$ is a sequence

$$(v_0, e_1, v_1, \dots, e_n, v_n)$$

such that v_i is a vertex in V for all i in $\{0, \dots, n\}$, and e_i is an edge in E adjacent to v_{i-1} and v_i for all i in $\{1, \dots, n\}$. A path is *simple* if it visits each edge at most once, and *elementary* if it visits each vertex at most once. We denote respectively by $V(P)$ and $E(P)$ the set of vertices and the set of edges of P.

Remark 3.2. Some authors call a walk what we have defined as a path, a trail a walk in which each edge is contained at most once, and a path a trail in which each vertex is contained at most once. △

Vertex v_0 is the *origin* vertex or *first vertex* of P, and v_n is its *destination* or *last vertex*. An o-d path is a path with origin o and destination d. We write $v \in P$ (resp. $e \in P$) to indicate that vertex v (resp. edge e) belongs to P. Two paths are *vertex-disjoints* (resp. arc disjoints) if there is no vertex v (resp edge e) that belongs to the two paths.

Exercise 3.3. Let A be the adjacency matrix of $G = (V, E)$. Characterize A^k_{uv} in terms of paths in G. △

A graph is *connected* if there exists a v-w path for each (v, w) in V^2. The *connected components* of G are the connected induced subgraph $G' = (V', E')$ of G that are maximal for inclusion (adding an vertex to V' makes it non connected).

Exercise 3.4. Let $u \dashleftarrow\dashrightarrow v$ be the binary relation on V indicating if there exists a u-v path.

1. Prove that $\dashleftarrow\dashrightarrow$ is an equivalence relation.
2. What are the equivalence classes of $\dashleftarrow\dashrightarrow$ on G?

△

3.1.4 Cycles

A *cycle* or *circuit* C in a graph G is a path $v_0, \dots, v_k$ such that $v_1, \dots, v_k$ is simple and $v_0 = v_k$. Some authors keep the term circuit for directed graph and cycle for undirected graphs.

A *Hamiltonian* path in a graph G is an elementary path such that $V(P) = V(G)$. A *Hamiltonian cycle* C in a graph G is an elementary cycle such that $V(C) = V(G)$. A *Hamiltonian graph* is a graph that admits a Hamiltonian cycle.

Hamiltonian cycle problem
Input. An undirected graph G
Question. Is there an Hamiltonian cycle in G?

The Hamiltonian path problem is obtained by replacing "cycle" by "path".

Theorem 3.1. *The* Hamiltonian path problem *and the* Hamiltonian cycle problem *are NP-complete.*

An *Eulerian path* P in a graph G is a simple path such that $E(P) = E(G)$. An *Eulerian cycle* C in a graph G is a simple cycle such that $E(C) = E(G)$. Consider the following problem.

Eulerian cycle problem
Input. An undirected graph G
Question. Is there an Eulerian cycle in G?

Proposition 3.2. *A graph $G = (V, E)$ is Eulerian if and only if $\deg_G(v)$ is even for all v in V.*

Exercise 3.5. Prove Proposition 3.2 △

Exercise 3.6. Is the Eulerian cycle problem NP-complete ? △

3.1.5 Trees

A *forest* is a graph that contains no cycle. A *tree* is a connected forest.

Exercise 3.7. Prove that each connected component of a forest is a tree. △

Exercise 3.8. How many edges has a tree with n vertices? △

3.1.6 Cliques, stable sets, matching, covers

We now introduce sets of vertices and edges of special interest.

A *clique* C in a graph G is a subset of vertices whose induced subgraph is complete. The size of a clique is its number of vertices. A *stable* S is a subset of vertices such that no two vertices of S are adjacent. It is also called an *independent set*. A *matching* M in graph G is a subset of edges such that there are no two edges in M that are incident with the same vertex v. A *vertex cover* C is a subset of V such that such that each edge of E is incident with at least a vertex in C. An *edge cover* C is a subset of E such that any vertex in V has an incident edge in C. Figure 3.1 illustrates these notions.

The following decision problems are associated to these notions.

Maximum clique
Input. A graph G, an integer k
Question. Is there a clique in G of size at least k.

Maximum stable
Input. A graph G, an integer clique
Question. Is there a stable in G with k vertex?

Maximum matching
Input. A graph G, an integer k
Question. Is there a matching with k edges in G?

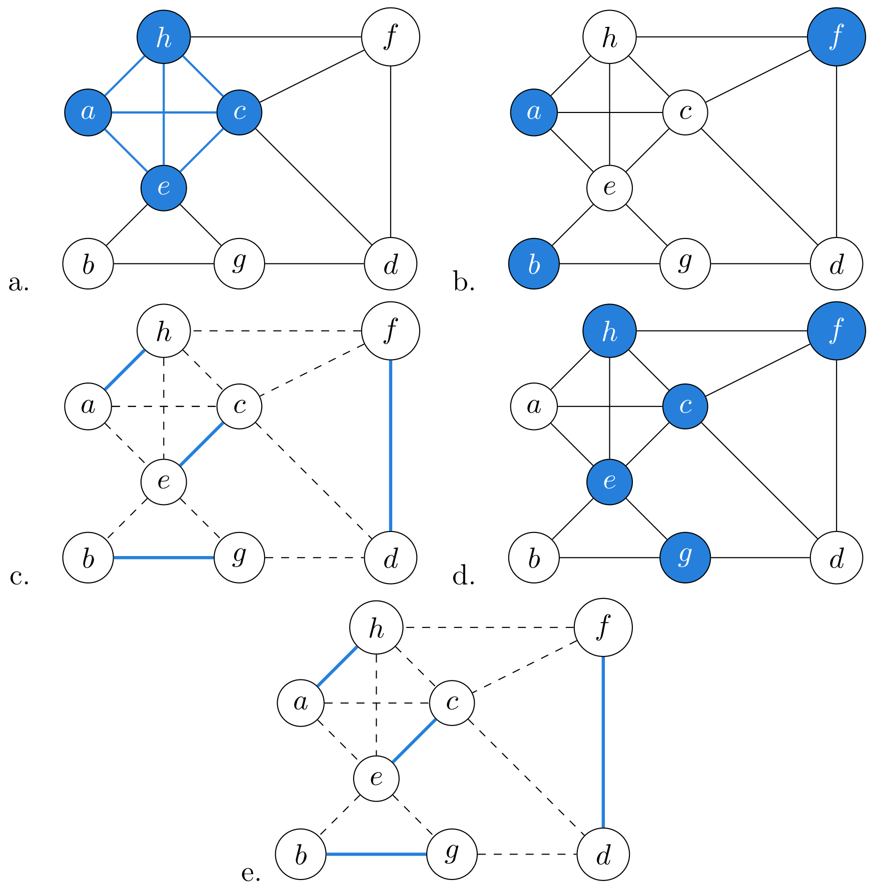


Figure 3.1: Example of a. clique, b. stable set, c. matching, d. vertex cover, and e. edge cover

MINIMUM VERTEX COVER
Input. A graph G, an integer k
Question. Is there an vertex cover with k vertices?

MINIMUM EDGE COVER
Input. A graph G, an integer k
Question. Is there an edge cover with k edges?

The following theorem is admitted.

Theorem 3.3. *The* MAXIMUM CLIQUE, *the* MAXIMUM STABLE, *and the* MINIMUM VERTEX COVER *problems are NP-complete.*

Details on the following algorithm are given in Chapter 7.

Theorem 3.4. *The* MAXIMUM MATCHING *and the* MINIMUM EDGE COVER *problem are polynomial.*

We denote by

$$\omega(G) = \textit{clique number of } G = \text{maximum size of a clique in } G, \tag{3.1a}$$
$$\alpha(G) = \textit{stable set number of } G = \text{maximum size of a stable set in } G. \tag{3.1b}$$
$$\nu(G) = \textit{matching number of } G = \text{maximum size of a matching in } G. \tag{3.1c}$$
$$\tau(G) = \textit{vertex cover number of } G = \text{minimum size of a vertex cover in } G. \tag{3.1d}$$
$$\rho(G) = \textit{edge cover number of } G = \text{minimum size of an edge cover in } G. \tag{3.1e}$$

Exercise 3.9. Prove that $\nu(G) \leq \tau(G)$ and $\alpha(G) \leq \rho(G)$. △

Exercise 3.10. Show that S is a stable in and only if $V \backslash S$ is a vertex cover. Deduce a relation between $\alpha(G)$ and $\tau(G)$. △

Exercise 3.11. Show that $\omega(G) = \alpha(\overline{G})$, where $\overline{G}$ is the complement of G. △

Application 3.1. *Transmitting messages over a noisy channel*

3.1.7 Coloring

A *coloring* of a graph $G = (V, E)$ is partition of V into stable sets. It is sometimes defined as a function $\phi : V \to i$ such that $\phi(u) \neq \phi(v)$ if (u, v) is in E. The number of colors of a coloring is the number of edges in the partition. A graph is *k-colorable* if it has a coloring with at most k colors. Let

$$\chi(G) = \textit{chromatic number of } G \tag{3.2}$$

be the minimum number of colors in a coloring of G.

Exercise 3.12. Show that $\omega(G) \leq \chi(G)$. △

COLORING
Input. A graph G, an integer k.
Output. Is there a k-coloring of G?

Theorem 3.5. *The coloring problem is NP-complete.*

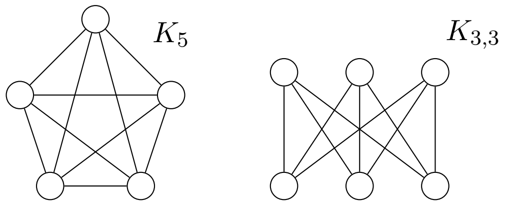


Figure 3.2: The two smallest non-planar graphs

Application 3.2. *Coloring and scheduling*

Exercise 3.13. A set F of formations must be given to employees of a firm. Each employee i must follow a subset F_i of formations. The firm wants to find the minimum number of formation slots it must schedule so that each employee can attend to its formations. Model this problem as a coloring problem. △

3.1.8 Planarity

A *planar graph* is a graph that can be embedded in the plane, i.e., that can be drawn on the plane in such a way that two distinct edges intersect only on vertices. The region delimited by the drawing of the graph are called the *faces*. An edge or a vertex is *incident* to a face if it is contained in the boundary of the face. Two faces are *adjacent* if they are incident with a common edge. There is a unique unbounded face called the *outer face*.

Remark 3.3. This definition can be made more formal, but the formal definition is not required to understand the discussion below. The topological graph associated with a graph G is the topological space consisting of V, and for each edge e, a curve $\phi(e)$ such that $\phi(e) \cap \phi(f) = e \cap f$. An embedding of G in the plane is a continuous injection from G to the plane $\mathbb{R}^2$. △

The importance of planar graphs comes from their numerous applications, such as network visualization or electrical circuits (chip) layout design ◈, and from the fact that many NP-complete problems on graphs become polynomial when restricted to planar graphs. This is for instance the case of MAX-CUT), but not of GRAPH COLORING problem. In particular, practically efficient divide and conquer algorithms can be devised for problems on planar graphs thanks to the planar separator theorem [Lipton and Tarjan, 1979]. ◈.

Figure 3.2 depicts the two smallest non-planar graphs. Proving that these graphs are non-planar is not completely trivial, and the object of Exercise ◈

The *dual graph* of a planar graph G is the graph whose vertices are the faces and whose edges are the pairs of adjacent faces.

◈

The following theorem took 125 years to be proved and is the first major theorem proved with the help of a computer.

Theorem 3.6. *(Appel et al. [1977]) Every planar graph is 4-colorable.*

3.1.9 Minors

Let $G = (V, E)$ and X be a subset of V. By *contracting* X, we mean removing X from V and replacing it by a new vertex x, removing all edges with both extremities in X, and replacing all edges (u, v) with $u \in X$ and $v \in V \backslash X$ by (x, v). Given $e \in E$ and $X \subseteq E$, we denote G/e and G/X the graph obtained by contracting e or X.

A graph H is a *minor* of a graph G if it can be obtained form G by a series of contraction and deletion of edges.

Minors play a key role in theoretical graph theory as many family of graphs can be characterized as graphs that do not contain some minors. The following theorem is a well-known example, the graph cited being illustrated on Figure 3.2.

Theorem 3.7. *(Kuratowski [1930]) A graph is planar if and only if it does not admit K_5 and $K_{3,3}$ as minors.*

3.2 Directed graphs

A *directed graph* D or *digraph* is a pair (V, A) where V is a finite set and A is a multiset on ordered pairs elements of V – a collection of ordered pairs on V^2. V is the set of vertices and A is the set of *arcs* a.

The *underlying undirected graph* of a digraph $D = (V, A)$ is the graph $G = (V, E)$ obtained by removing the orientation of the arcs in A. In that case, we say that D is an *orientation* of V. The *reverse graph* of $D = (V, A)$ is the digraph $D^{-1} = (V, A^{-1})$ where $A^{-1} = \{(v, u)\colon (u, v) \in A\}$

As for undirected graph, the multiplicity of an arc (u, v) is the number of times it occurs in A, a *loop* is an arc (v, v) for some v in V. A digraph is *simple* if arcs have multiplicity at most one and there is no loop.

An arc $a = (u, v)$ *leaves* u or *is outgoing from* u and enters v or is *incoming to* v. If a is in A, then u is an *inneighbor* of v and v is an *outneighbor of u*. We denote by

$$\delta^-(v) \text{ the set of arcs incoming to } v, \tag{3.3a}$$
$$\delta^+(v) \text{ the set of arcs outgoing from } v, \tag{3.3b}$$
$$N^-(v) \text{ the set of inneighbor of } v, \tag{3.3c}$$
$$N^+(v) \text{ the set of outneighbors of } v. \tag{3.3d}$$

Given a set of vertices U, we define similarly $\delta^-(U)$, $\delta^+(U)$, $N^-(U)$, and $N^+(U)$. The *indegree* and *outdegree* of a vertex v are respectively the number of arcs incoming to and outgoing from v, and are denoted by $\deg^-(v)$ and $\deg^+(v)$. A *source* in a digraph is vertex that has no incoming arcs, and a *sink* is a vertex that has no *outgoing* arcs.

The notions of *subgraph* and *induced subgraph* are defined as in the undirected case.

3.2.1 Adjacency and incidence matrices

The *adjacency matrix* M of a digraph is the $V \times V$ matrix where

$$m_{u,v} \text{ is the number of arcs from } u \text{ to } v \text{ in } D. \tag{3.4}$$

The *incidence matrix* is the $V \times A$ matric B where

$$b_{u,a} = \begin{cases} -1 & \text{if } a = (u, v) \text{ with } v \neq u, \\ 1 & \text{if } a = (v, u) \text{ with } v \neq u, \\ 0 & \text{otherwise.} \end{cases} \tag{3.5}$$

3.2.2 Paths, connected components, cuts

A *path* in a digraph $D = (V, A)$ is a sequence

$$(v_0, e_1, v_1, \ldots, e_n, v_n)$$

such that v_i is a vertex in V for all i in $\{0,\dots,n\}$, and a_i is an arc in A from v_{i-1} to v_i for all i in $\{1,\dots,n\}$. An o-d path is a path from o to d. Vertex o is the *origin* of the path, and d its *destination*. A path is *simple* if it visits each edge at most once, and *elementary* if it visits each vertex at most once. We denote respectively by $V(P)$ and $A(P)$ the set of vertices and the set of arcs of P.

A vertex s is connected to a vertex t if there exists an s-t path. We denote by

$$u \dashrightarrow v \tag{3.6}$$

the fact that u is connected to v. In a directed graph, connectedness is no more an equivalence relation. Two vertices s and t are *strongly connected* s is connected to t and t to s. Strong connectedness is an equivalent relations, and its equivalence classes are the strong connected components. The connected components of the underlying undirected graph are called the weak connected components of a digraph.

A *cut* in a directed graph a set of arcs B such that $B = \delta^-(U)$ for some set of vertices U.

Exercise 3.14. Show that a set B is a cut if and only if it is of the form $\delta^+(U)$ for some $U \subseteq V$. $\triangle$

3.2.3 Cycles, acyclic digraphs and directed trees

A *cycle* is a directed graph is a v-v path that contains at least one arc and such that the path obtained by removing the last arc is elementary. A digraph is *acyclic* of it contains no cycle.

Exercise 3.15. Prove that if D is an acyclic digraph and $u \neq v$, then u and v cannot be strongly connected. $\triangle$

A *topological ordering* on a digraph $D = (V, A)$ is a total ordering $\preceq$ on V such that $u \dashrightarrow v$ implies $u \preceq v$.

Exercise 3.16. Give a simple (polynomial) algorithm computing a topological ordering on an acyclic digraph. Prove that it returns a topological ordering. $\triangle$

A *directed tree* is a directed graph whose underlying digraph is a tree. An r-*rooted tree* is a directed tree such that has a unique source, r, that is call its *root*. Sinks of a rooted tree are called *leaves*.

Hamiltonian and *Eularian* paths, cycles, and graphs in a directed graph are defined as in undirected graph, the only difference being that undirected paths are replaces by directed paths.

Exercise 3.17. Prove that a digraph is Eulerian if and only if $\deg^-(v) = \deg^+(v)$ for all v in V. $\triangle$

3.3 Further readings

By increasing difficulty

- Diestel [2018]
- Bondy et al. [1976]
- Bollobás [2013]

See also Schrijver [2003], which is not specifically on graphs but contains much content.

3.4 Exercises

Exercise 3.18. The BAR FIGHT PREVENTION problem can be described as follows. You are a security guard in a bar in a small city. You know the n clients that want to come to your bar on a Friday evening. And you know the pair of clients that, if both admitted, will fight. Your boss wants many client in his bar, and allows you to exclude at most k clients. You want to know if it is possible to prevent fights by excluding only k clients, and if yes to find the (at most) k clients to exclude.

1. Model the BAR FIGHT PREVENTION as a graph problem.

   Solution. VERTEX COVER □

2. What is its complexity?

To avoid excluding nice troublemakers from his bars, your boss comes with a brilliant new plan. He has bought k bars. For the next Friday, you still have a list of pair of clients that will fight. Your job is now to partition this list into k sub-list of clients, one for each bar, in such a way that two clients that would fight are not in the same bar.

3. Model this new version as a graph problem.

   Solution. Coloring □

4. What is it complexity?

△

Exercise 3.19. Characterize the graphs that admits a circuit C that is both Hamiltonian and Eulerian. △

Exercise 3.20. Prove that the HAMILTONIAN PATH PROBLEM is NP-complete using the fact that the HAMILTONIAN CYCLE PROBLEM is NP-complete, and conversely, prove that the HAMILTONIAN CYCLE PROBLEM is NP-complete using the fact that the HAMILTONIAN PATH PROBLEM is NP-complete. △

Exercise 3.21. *Les Ponts de Königsberg* En 1736, les notables de Königsberg demandèrent à Euler s'il était possible de parcourir les 7 ponts de la ville en passant sur chacun d'eux exactement une fois (voir Figure 3.3). C'est le premier problème de Recherche Opérationnelle (à visée non pratique). Ce genre de problème se rencontre maintenant très souvent dans les problèmes de tournées du type facteur ou ramassage de déchets ménagers, dans lesquels il faut parcourir les rues d'une ville de façon optimale. Euler trouva la solution...

Trouvez la solution. △

Exercise 3.22. *Dessiner sans lever le crayon*

Indiquez pour les Figures 3.4 et 3.5 quand il est possible de dessiner la figure sans lever le crayon (et sans repasser sur un trait déjà dessiné) et quand il ne l'est pas. △

Exercise 3.23. *Parcours hamiltonien*

Pour chacune des trois graphes suivants, indiquez s'il est possible de faire un parcours qui passe par chaque commet une fois et une seule. Un tel parcours s'appelle *chaîne hamiltonienne.* Même question sur l'existence d'une parcours fermé (revenant à son point de départ) qui passe par chaque sommet une fois et une seule. Un tel parcours s'appelle *cycle hamiltonien.*

Justifiez votre réponse. △

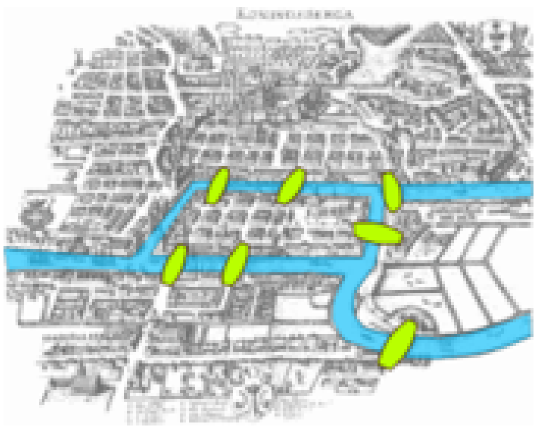


Figure 3.3: Königsberg et ses 7 ponts

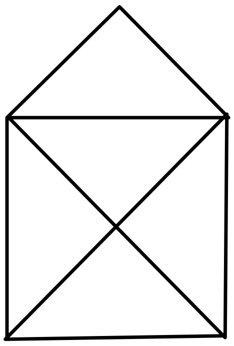


Figure 3.4: Peut-on tracer cette figure sans lever le crayon ?

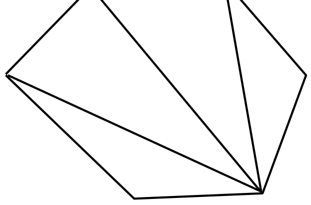


Figure 3.5: Et celle-là ?

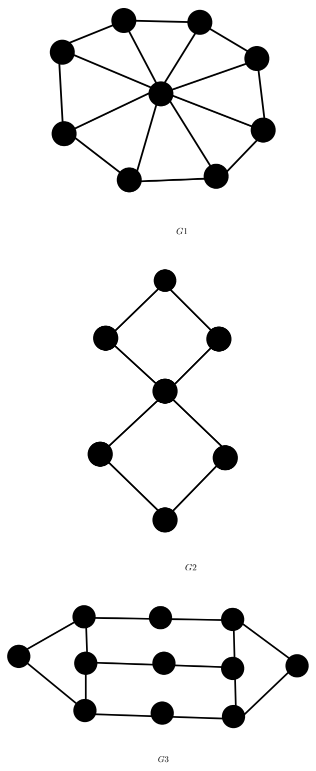


Figure 3.6: Lequel de ces graphes contient une chaîne hamiltonienne ? Lequel contient un cycle hamiltonien ?

Exercise 3.24. *Le jeu Icosian*

La Figure 3.7 représente le jeu Icosian inventé en 1859 par le mathématicien Hamilton. Peut-on mettre les chiffres de 1 à 20 dans les cercles de manière à ce que toute paire de nombres consécutifs soit adjacente, 1 et 20 étant aussi considérés comme adjacents ?

△

Exercise 3.25. *Le voyageur de commerce* Un camion doit effectuer des livraisons depuis un dépôt en un certain nombre de points d'un réseau consitué de routes. On connaît le temps qu'il faut pour parcourir chaque portion de route. On suppose ces temps indépendants du sens de circulation. On veut minimiser le temps qu'il faut au camion pour effectuer ces livraisons et revenir au dépôt. Modéliser ce problème comme un problème de cycle hamiltonien de poids minimum sur un graphe complet.

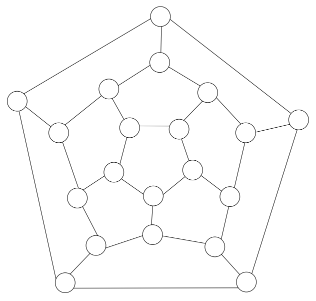
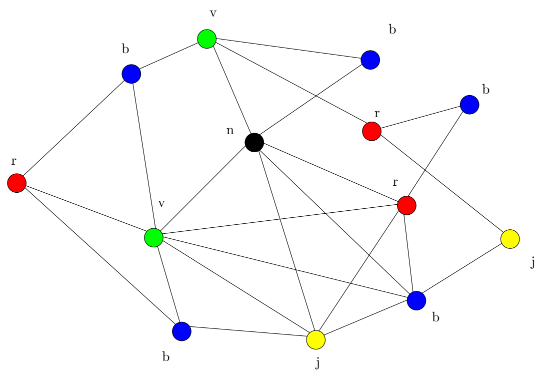


Figure 3.7: Le jeu Icosian.



Graphe coloré avec cinq couleurs: r, b, j, v, n.

Figure 3.8: Peut-on colorer ce graphe avec moins de cinq couleurs ?

△

Exercise 3.26. *Affectation de fréquences* Considérons des relais de téléphonie mobile. Chaque relais reçoit et émet des signaux à une certaine fréquence. Pour éviter les interférences, on veut que deux relais à moins de 200 m fonctionnent avec des fréquences différentes. Le coût de gestion du parc de relais est une fonction croissante du nombre de fréquences utilisées. On souhaite minimiser ce coût.

Modéliser ce problème avec les outils du cours. △

Exercise 3.27. *Organisation de formations* Considérons maintenant un DRH qui doit assurer un certain nombre de formation à ses employés. Les formations sont $F_1, \dots, F_n$. Et les employés sont $E_1, \dots, E_m$. Le DRH sait pour chaque employé E_i quelles sont les formations qu'il doit suivre. Il veut organiser une session de formations – chacune des formation ne peut être dispensée qu'une fois au total dans l'entreprise et un employé peut suivre au plus une formation par jour. En revanche, plusieurs formations peuvent être dispensées le même jour. Le DRH souhaite organiser la session la plus courte possible.

Modéliser ce problème avec les outils du cours. △

Exercise 3.28. *Borne inférieure du nombre chromatique* Montrer l'optimalité de la coloration de la Figure 3.8, et proposer une borne inférieure générale pour les nombre chromatique. △

Exercise 3.29. *Borne supérieure pour les couplages maximum* Montrer l'optimalité du couplage de la Figure 3.9, et proposer une borne supérieure générale pour les couplages maximum. △

Exercise 3.30. **NP**-*complétude de la chaîne hamiltonienne* Montrer que le problème de l'existence de la chaîne hamiltonnienne est **NP**-complet, sachant que celui du cycle hamiltonien l'est. Réciproquement, montrer que le problème de l'existence du cycle hamiltonnien est **NP**-complet, sachant que celui de la chaîne hamiltonienne l'est. △

Exercise 3.31. **NP**-*difficulté du problème du voyageur de commerce* Le problème du voyageur de commerce consiste à trouver un cycle hamiltonien de longueur minimale dans un graphe complet muni d'une distance $d : E \to \mathbb{R}_+$ (satisfaisant l'inégalité triangulaire) : $d(uv) + d(vw) \geq d(uw)$ pour tous sommets u, v, w.

Montrer que le problème du voyageur de commerce est **NP**-difficile en utilisant la **NP**-complétude du problème du cycle hamiltonien. △

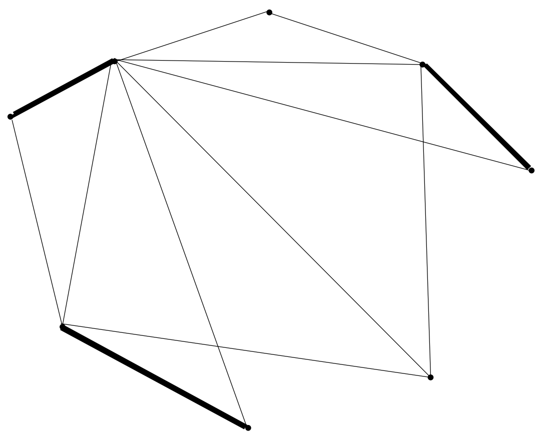


Figure 3.9: Est-ce un couplage maximum ?

Chapter 4

Spanning trees

A *spanning subgraph* in a graph $G = (V, E')$ is a a subgraph $T = (V, E')$ such that each vertex in V has at least an incident edge in E'. A *spanning tree* T is a spanning subgraph that is a tree.

This chapter focuses on the minimum weight spanning tree problem.

Minimum weight spanning tree problem
Input. An undirected graph $G = (V, E)$, a weight function $c : E \to \mathbb{R}$.
Output. A spanning tree $T = (V, E')$ of minimum weight $\sum_{e \in E'} c(e)$

We will see its link with the maximum weight forest problem.

Maximum weight forest problem
Input. An undirected graph $G = (V, E)$, a weight function $c : E \to \mathbb{R}$.
Output. A forest $G' = (V, E')$ of maximum weight $\sum_{e \in E'} c(e)$.

Two problems $\mathcal{P}$ and $\mathcal{P}'$ are *linearly equivalent*, or simply *equivalent* is there exists a linear reduction from $\mathcal{P}$ to $\mathcal{P}'$ and from $\mathcal{P}'$ to $\mathcal{P}$.

Proposition 4.1. *The minimum weight spanning tree problem and the maximum weight forest are equivalent.*

Exercise 4.1. Prove Proposition 4.1.

Solution. Let $(G = (V, E), c)$ be an instance of the maximum weight forest. Let G' be the graph obtained by removing all negative weight edges from G, and $c'(e)$ to be equal to $-c(e)$. Find a minimum weight spanning tree for c' on each connected component of G'. The result is a maximum weight forest in G'. Indeed, any edge with negative weight cannot be in a maximum weight forest, and it is immediate to complete a maximum weight forest on a component of G' into a maximum weight forest that is a tree by adding edges. This maximum weight forest for c that is a tree is a minimum weight spanning tree.

Conversely, let $(G = (V, E), c)$ be an instance of the minimum weight spanning tree. Let $c'(e) = 1 - c(e) + \max_{e' \in E} c(e')$. Then, if G is connected, it is clear that a maximum weight forest $F = (V, E')$ in (G, c') is a tree, and $\sum_{e \in E'} c'(e) = |V - 1|(1 + \max_{e \in E} c(e)) - \sum_{e \in E'} c(e)$. Hence, a maximum weight forest in (G, c') is a minimum weight spanning tree in (G, c). If G is not connected, the maximum weight forest returned is not a tree. □

△

The following characterization of minimum weight spanning trees will be useful.

Proposition 4.2. *A spanning tree $T = (V, E')$ is a minimum weight spanning tree if and only if, for all e in E', edge e is a minimum weight edge of the cut $\delta(V_1)$ between the two connected components V_1 and V_2 of $(V, E'\backslash E)$.*

Proof. Suppose that T is a minimum weight spanning tree. Take any $e \in E'$ and assume there exists an edge e' in $\delta(V_1)$ such that $c(e') < c(e)$. Since $(V, E'\backslash\{e\})$ is the union of a spanning tree on V_1 and a spanning tree on V_2, $(V, E'\backslash\{e\} \cup \{e'\})$ is a spanning tree on G whose weight is strictly smaller than the weight of T, which is contradictory.

Suppose that for all e in E', edge e is a minimum weight edge of the cut $\delta(V_1)$ between the two connected components V_1 and V_2 of $(V, E'\backslash\{e\})$. Let T_0 be a minimum weight spanning tree. Suppose that k edges of T are not in T_0. Let e be such an edge, and V_1 and V_2 the two connected components of $(V, E'\backslash\{e\})$. Since T_0 is a tree, any cycle in $(V, E_0 \cup \{e\})$ contains e. And since T_0 is a tree, it contains a unique path between the endpoints of e. Hence $(V, E_0 \cup \{e\})$ contains a unique cycle, which contains at least an edge e' different from e in $\delta(V_1)$. Furthermore, $T' = (V, E_0\backslash\{e'\} \cup \{e\})$ is a spanning tree. And since $c(e) \leq c(e')$, T' is also a minimum weight spanning tree, and $m-1$ edges of T are not in T'. Iterating this procedure enables to build a sequence $T_0, \dots, T_k$ of minimum weight spanning trees such that for each i, $k-i$ edges of T are not in T_i. Hence $T_k = T$ and T is a minimum weight spanning tree. □

4.1 Greedy algorithm

The minimum weight spanning tree problem is very efficiently solved using Kruskal's algorithm, which we now state.

Algorithm 1 Kruskal's algorithm

```
1: Input: An undirected graph $G = (V, E)$, a weight function $c : E \to \mathbb{R}$
2: Sort $E$ by $c(e)$: $E = \{e_1, \dots, e_m\}$ with $c(e_i) \leq c(e_{i+1})$
3: $E' \leftarrow \emptyset$
4: for $i$ from 1 to $m$ do
5:     if $(G, E' \cup \{e_i\})$ is acyclic then $E' \leftarrow E' \cup \{e_i\}$
6: end for
7: return $(V, E')$
```

Kruskal's algorithm is a greedy algorithm: At each step, we add the best candidate to E'. Greedy algorithm are usually heuristics. A very surprising fact is that, in the case of minimum weight spanning trees, Kruskal's algorithm returns an optimal solution.

Theorem 4.3. *If G is connected, then Kruskal's algorithm returns a minimum weight spanning tree.*

Proof. It is clear that at any time during the algorithm, (V, E') is a forest. Let T be the graph returned by the algorithm. Suppose that T has at least two connected components V_1 and V_2 in T. Since G is connected, $\delta(V_1)$ is not empty. Let e_i be the edge of $\delta(V_1)$ with smallest index i. When e_i was considered, there was no other edge of $\delta(V_1)$ in E'. Hence e_i should have been added to E', which gives a contradiction. Hence T is a tree. Furthermore, each edge e_i added to E' is a minimum weight edge in $\delta(V_1)$. Hence T satisfies the hypotheses of Proposition 4.2, and is therefore a minimum weight spanning tree. □

4.2 Spanning tree polytope

We now introduce a linear programming formulation for the minimum weight spanning tree problem. Since the minimum weight spanning tree problem can be solved efficiently using Kruskal's algorithm, this linear programming formulation should not be used in practice to solve the minimum weight spanning tree problem. However, as we will see on some applications, this linear programming formulation can easily be adapted to deal with variants of the minimum weight spanning tree problem that cannot be handled by Kruskal's algorithm.

Given a subset of vertices $X \subseteq V$ of the graph $G = (V, E)$, we denote by $G[X] = (X, E[X])$ the subgraph induced by X.

Consider the following linear program

$$\begin{aligned}
\min \sum_{e \in E} c(e) x_e & & \quad (4.1a)\\
\text{s.t.} \sum_{e \in E} x_e = |V| - 1, & & \quad (4.1b)\\
\sum_{e \in E[X]} x_e \leq |X| - 1, & \quad \forall \emptyset \subsetneq X \subsetneq V & \quad (4.1c)\\
x_e \geq 0, & \quad \forall e \in E. & \quad (4.1d)
\end{aligned}$$

Theorem 4.4. *A basic optimal solution x_e of the linear program is integer, and gives a minimum weight spanning tree (V, E') where $E' = \{e \in E \colon x_e = 1\}$.*

Exercise (4.3) gives a proof of Theorem 4.4 and is a nice exercise on linear programming duality and integer programming.

Remark that the number of constraints (4.1c) is exponential in the size of the instance. Such a linear program must therefore be solved using the line generation algorithm of Section 9.5.1. The separation problem associated to this problem is

$$\min_{X \subseteq V} \left(|X| - 1 - \sum_{e \in E[X]} x_e \right). \quad (4.2)$$

This separation problem can be solved using a flow approach. Exercise 4.4 shows that we can either find a violated constraint or show that there is no such constraint in polynomial time.

4.3 Exercises

Exercise 4.2. An air-conditioning company wants to equip a new building with their cooling machines. It must install:

- A compressor s outside the building
- One unit in every room that must be air-conditioned. These rooms are numbered with the letters a to g.
- Pipes designed so that each unit inside is linked to the compressor.

The same portion of pipe can be used for several inside units. Figure 4.1 described the pipes that can be built and their construction cost. We want to decide which pipes to install so that the pipe network statisfies the constraints at a minimal cost.

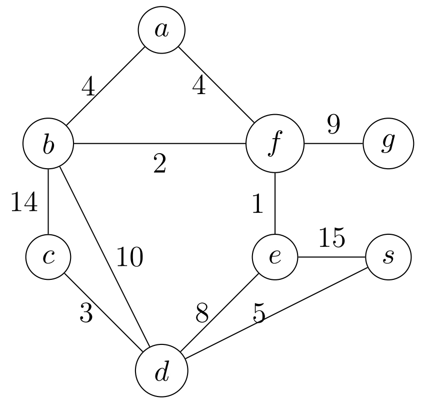


Figure 4.1: Tuyaux disponibles

1. Which tool from the lectures can be used to model this problem? (short answer - 1 pt)

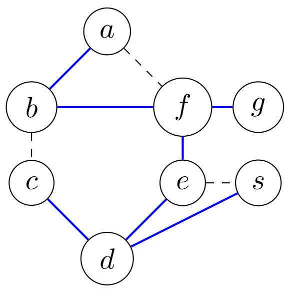
   Solution. This is a minimum weight spanning tree problem. □

2. What is the most efficient algorithm to solve it? (1 pt)

   Solution. Kruskal's algorithm. □

3. Give the optimal solution and its cost (you can draw it on the question sheet). (1 pt)

   Solution. The optimal solution is the following, it has a cost of 32.

   □

△

Exercise 4.3. *Proof of Theorem 4.4* (Requires Chapters 9 and 10)

1. Prove that integer solutions of (4.1) correspond to spanning trees.

   Solution. Let $T = (V, E')$ be a spanning tree, and x_e be equal to 1 if e is in E' and 0 otherwise. Then x_e is a solution of (4.1) because a spanning tree contains $|V| - 1$ edges, and subgraphs induced by trees are forests.

   Let x be a feasible solution of (4.1) such that $x_e \in \{0, 1\}$ for all e. We now prove that $T = (V, E')$ where $E' = \{e \in E : x_e = 1\}$ is a spanning tree. Suppose that T contains a cycle C, and let X be the vertices in C. Then $\sum_{E[X]} x_e = |C|$, which violates constraint (4.1c). Hence T is a tree with $|V| - 1$ edges, and therefore a spanning tree. □

We therefore only have to show that the polytope of (4.1) is integral. Let $z_X \geq 0$ be the dual variable associated with constraint (4.1c), and $z_V \in \mathbb{R}$ the dual variable associated with constraint (4.1b).

2. Prove that the dual of (4.1) is

$$\max \sum_{\emptyset \subsetneq X \subseteq V} -(|X|-1)z_X \tag{4.3a}$$

$$\text{s.t.} \sum_{X \ni e} -z_X \leq c(e), \qquad \forall e \in E \tag{4.3b}$$

$$z_X \geq 0 \qquad \forall \emptyset \subsetneq X \subsetneq V \tag{4.3c}$$

   Remark that in the two first equations, the full vertex set V appears as a set X in the sums.

Let $T = (V, E')$ be the spanning tree produced by Kruskal's algorithm, and x^* be the incidence vector of E'. Let $e_1, \ldots, e_{n-1}$ be the edges of T in the order in which they appear in Kruskal's algorithm. We have $c(e_1) \leq \ldots \leq c(e_{n-1})$. For k in $[n-1]$, let X_k be the connected component of $(V, \{e_1, \ldots, e_k\})$ containing e_k.

3. For k in $1, \ldots, n-2$, show that there exists $k' > k$ such that $X_k \cap e_{k'} \neq \emptyset$.

   Solution. Since $(V, \{e_1, \ldots, e_k\})$ is not a spanning tree, $X_k \neq V$ and Kruskal's algorithm adds at least an edge $e_{k'}$ connecting X_k to the rest of the graph. □

For k in $1, \ldots, n-2$, let $z^*_{X_k} = c(e_\ell) - c(e_k)$ where ℓ is the first index greater than k such that $X_k \cap e_l \neq \emptyset$. Let $z^*_V = -c(e_{n-1})$. And let $z^*_X = 0$ for any X not in $\{X_1, \ldots, X_{n-1}\}$.

4. Show that z^* is a feasible solution of (4.3).

   Solution. Let e be an edge, u, v be its extremities, and i the smallest index such that both extremities of e belong to X_i. We have

$$\sum_{X \ni e} -z^*_X = c(e_i) \leq c(e).$$

   Indeed, the equality following immediately from the definition of z^*. By definition of e_i, the extremities of e are in different connected components of $(V, \{e_1, \ldots, e_{i-1}\})$. Suppose now that $c(e_i) > c(e)$. Edge e is therefore considered before e_i. Hence, there is no u-v path in the graph constructed by Kruskal's algorithm when it considers e. Hence e is added by Kruskal's algorithm. This contradicts the definition of e_i, and gives the result. □

5. Show that x^* and z^* satisfy the complementarity slackness condition. Conclude.

   Solution. Proposition (9.6) then gives that x^* and z^* are optimal solutions of the primal and the dual, which gives the result. □

We show that x is an optimal solution of (4.1).

△

Exercise 4.4. *Polynomial algorithm for the separation problem (4.2) (Requires Chapter 6)* In this exercise, we propose a polynomial time algorithm to separate the family of constraint (4.1c), which we now restate.

$$\sum_{e\in E[X]} x_e \leq |X| - 1, \quad \forall \emptyset \subsetneq X \tag{4.4}$$

We therefore consider the separation problem 4.2, i.e.,

$$\min_{X\subseteq V} |X| - 1 - \sum_{e\in E[X]} x_e$$

1. Prove that (4.4) has the same solutions as the following problem.

$$\min_{X\subseteq V} |X| + \sum_{e\notin E(S)} x_e. \tag{4.5}$$

Solution. Since $\sum_{e\in E} x_e = |V| - 1$, the objective of the two problems are equal up to a constant. □

We define a directed graph $D = (U, A)$ as follows. $U = E \cup A \cup \{o, d\}$. A contains the following arcs

- $a = (o, e)$ for each e in E,
- $a = (e, u)$ and $a' = (e, v)$ for each $e = (u, v)$ in E,
- $a = (v, d)$ for each v in V.

2. Show that (4.4) can be reduced to a minimum capacity cut problem on D for a vector of capacities $(u_a)_a$ on the arcs of D that will be indicated.

Solution. We define the following capacities (u_a) on the arcs

- if $a = (o, e)$ with e in E, x_e.
- if $a = (e, v)$ with e in E and v in V, then $u_a = u_{a'} + \infty$.
- if $a = (v, d)$ with v in V, then $u_a = 1$.

Let φ be the mapping that associates to each subset X of V the set of arcs

$$\varphi(X) = \{((v, d) \colon c \in X\} \cup \{(o, e) \colon e \notin E(x)\}.$$

Then $\varphi(X)$ is an o-d cut. Indeed, consider an o-d path P. Then P is necessarily of the form (o, e, v, d) with v an extremity of e. And the definition of $\varphi(X)$ implies that if (o, e) is not in $\varphi(X)$ then (v, d) is in $\varphi(X)$. Furthermore

$$\sum_{a\in\varphi(X)} u_a = |X| + \sum_{e\notin E(S)} x_e.$$

Consider now a minimum capacity o-d cut $B = \delta^+(S)$ in D, and let $X = B \cap V$. There is not edge of the form e, v with e in E and v in V that is in B because these edges have capacity $+\infty$. Remark that, for any edge e in $E[X]$, there is no e-d path in D that is not cut by B. Hence, since B is a minimum capacity cut, either $x_e = 0$ or (o, e) is not in B, as otherwise we would obtain an o-d cut of strictly smaller capacity by removing (o, e) from B. W.l.o.g., we suppose that if $e = (u, v)$, B contains (u, d) and (v, d), and $x_e = 0$, then B does not contain (o, e). Indeed, we obtain an o-d with the same weight if we remove (o, e). Consider now an edge x_e that is not in $E[X]$. By definition of X, (v, d) is not in B. Hence, one of its extremities v is not in X. Since B is an o-d cut, we can deduce that (o, e) belongs to B. Hence $B = \varphi(X)$.

To summarize, we have shown that for any $X \subseteq V$, $\varphi(X)$ is a cut pf cost $|X| + \sum_{e \notin E(S)} x_e$, and that for any minimum capacity cut B, there is a cut B' with the same capacity that is of the form $\varphi(X)$. This concludes the reduction of (4.5) to a flow problem on D. $\square$

$\triangle$

Chapter 5

Shortest paths and dynamic programming

This chapter focuses on the shortest path problem.

SHORTEST PATH PROBLEM
Input. A digraph $D = (V, A)$, a cost function $c : A \to \mathbb{Q}$.
Output. A simple o-d path of minimum cost $\sum_{a \in P} c(a)$.

It of course has the following analogue in undirected graphs.

SHORTEST PATH PROBLEM (UNDIRECTED VERSION)
Input. An undirected graph $G = (V, E)$, a cost function $c : A \to \mathbb{Q}$.
Output. A simple o-d path of minimum cost $\sum_{e \in P} c(e)$.

The undirected version can be reduced to the directed version by solving the instances in the digraph $D = (V, A)$ where A contains the arcs (u, v) and (v, u) for each (undirected) edge (u, v) in E, which we uses in all cases except on graph with negative cost arcs but no negative cost cycles.

Theorem 5.1. *The* SHORTEST PATH PROBLEM *is $\mathscr{NP}$-hard.*

However, as we will see in this chapter and sum-up in Table 5.1, there are many polynomial cases. In particular, the shortest path problem becomes polynomial in the absence of *negative cost cycles* C, that is, cycles C satisfying

$$\sum_{a \in C} c_a < 0.$$

Acyclic undirected graphs are not mentioned in Table 5.1 because they correspond to trees and forests, and hence there is a unique o-d path if the graph is connected, which makes the problem trivial.

Problem	Algorithm	Complexity
Acyclic digraph	Dynamic programming (topological ordering)	$O(m)$
Digraph, $c(a) \geq 0$	Dijkstra	$O(n^2)$
Digraph, no absorbing cycle	Dynamic programming (Ford-Bellman)	$O(nm)$
Digraph, generic c		$\mathscr{NP}$-complete
Undirected graph, $c(a) \geq 0$	Dijkstra	$O(n^2)$
Undirected graph, no absorbing cycle	T-joints 😎	$O(n^3)$
Undirected graph, generic c		$\mathscr{NP}$-complete

Table 5.1: Shortest path algorithms – applies to directed and undirected graphs if not mentioned

5.1 Dynamic programming

5.1.1 General case: Ford Bellman algorithm

Dynamic programming algorithms follow from the following observation.

Proposition 5.2. *Let P be an o-v path with $k > 0$ arcs, and u be the path before v on P, and P' the o-u subpath of P obtained by removing the last arc of P. If P is a shortest path among the o-v paths with k arcs, then P' is a shortest path among the o-u paths with $k-1$ arcs.*

Proof. Let Q' be an o-u path with $k-1$ arcs such that

$$\sum_{a \in Q'} c_a < \sum_{a \in P'} c_a,$$

and Q be Q' followed by u-v. Then Q is an o-v path satisfying

$$\sum_{a \in Q} c_a < \sum_{a \in P} c_a.$$

□

Let $f(v,k)$ be equal to the length of a shortest o-v path with at most k arcs if such a path exists, and to $+\infty$ if no such path exists. Proposition 5.2 ensures that f satisfies the *Bellman* or *dynamic programming* equation

$$f(v,k+1) = \min_{u \in N^-(v)} c_{(u,v)} + f(u,k), \tag{5.1}$$

which enables to compute f iteratively knowing that

$$f(v,0) = \begin{cases} 0 & \text{if } v = o, \\ +\infty & \text{otherwise.} \end{cases} \tag{5.2}$$

This iterative algorithm is known as the Ford-Bellman algorithm. Keeping in memory an argmin in (5.1) enables to rebuild a shortest path with k vertices. However, if the digraph contains negative cost cycles, the shortest path computed is not necessarily elementary.

Proposition 5.3. *If D has no negative cost cycles, then the shortest path problem can be solved in $O(mn)$ using the Ford-Bellman algorithm.*

Proof. Since D has no negative cost cycles, an elementary shortest path P' can be obtained from a shortest path P by removing cycles from P. Hence, there is an elementary shortest path of length at most $n-1$. Let k_0 be in $\operatorname{argmin}_{k\in[n]} f(d,k)$, let P' be a shortest path of length k_0 obtained from Ford-Bellman algorithm, and let P be the graph obtained by removing cycles from P'. Then P is an elementary shortest path.

Ford-Bellman algorithm enables to compute $f(d,k)$ for $k \in [n]$ in $O(mn)$. All the other steps of the approach are at most in $O(n)$. □

5.1.2 Acyclic digraphs

Proposition 5.2 can be strengthened in acyclic digraphs. We underline that in an acyclic digraph, all the paths are elementary.

Proposition 5.4. *Let D be an acyclic digraph, P be a shortest o-v path with at least one arc, u be the vertex before v on P, and Q be the o-v subpath of P. If P is a shortest o-v path, then Q is a shortest o-u path.*

Exercise 5.1. Prove Proposition 5.4. △

Hence, denoting $f(v)$ the length of a shortest o-v path, we have the following dynamic programming equation.

$$f(v) = \min_{u\in N^-(v)} f(u) + c_{(u,v)} \tag{5.3}$$

Recall that a topological ordering is an ordering $\preceq$ on V such that $(u,v)\in A$ implies $u \preceq v$, and that a digraph is acyclic if and only if it admits a topological ordering. A topological ordering on D can be computed in $O(m+n)$ using Algorithm 2.

Algorithm 2 Compute a topological order on D.

```
1: Input: a digraph $D = (V, A)$
2: Initialization: $L \leftarrow$ empty list, $S \leftarrow \emptyset$.
3: $S \leftarrow \{v : \delta_A^-(v) = \emptyset\}$
4: while $S \neq \emptyset$ do
5:     remove a vertex $v$ from $S$
6:     add $v$ at the end of $L$
7:     remove from $A$ all the arcs in $\delta^+(v)$
8:     add to $S$ all the vertices $w$ in $N^+(v)$ such that $\delta_A^-(w) = \emptyset$
9: end while
10: return $L$                    (L is sorted along a topological order $\preceq$)
```

Let $\preceq$ be a topological ordering on D. We denote by $\prec$ the strict order corresponding to $\preceq$, i.e., $u \prec v$ if $u \preceq v$ and $u \neq v$. The $f(v)$ for $o \preceq v \preceq D$ can be computed iteratively along $\preceq$ in $O(m+n)$ using

$$f(v) = \begin{cases} \infty & \text{if } v \prec o \\ 0 & \text{if } v = o \\ \displaystyle\min_{u\in N^-(v)} f(u) + c_{(u,v)} & \text{otherwise.} \end{cases} \tag{5.4}$$

We have proved the following proposition.

Proposition 5.5. *A topological order on an acyclic digraph can be computed in $O(m+n)$ by dynamic programming.*

5.1.3 Dynamic programming, a general method

The dynamic programming methods introduced in this section enables to solve a wide range of problem that aim at controlling a discrete dynamic system. Consider a dynamic system that over on a horizon T in $\mathbb{Z}_+$. At each time t in $0, 1, \ldots, T$, the system is in a state x in a finite state space $\mathcal{X}$. At $t = 0$ the system is in state s_{ini}. At each time t in $0, 1, \ldots, t-1$, the decision make can take an action u in a finite set $\mathcal{U}$. Let s_t denote the state of the system at time t. Let f be a transition function which gives the state s_{t+1} of the system at time $t+1$ if the system is in s_t at time t and action u_t is taken.

$$s_{t+1} = f(s_t, u_t) \tag{5.5}$$

Taking action u when in state s has cost $c(s, u)$. Being in state s at time T costs $c_T(s)$. The objective is to find a sequence of controls $\boldsymbol{u} = (u_0, \ldots, u_{T-1})$ so as to minimize the total cost. In summary, we consider the following problem.

$$\begin{aligned}
\min_{\boldsymbol{u}} \; & \sum_{t=0}^{T-1} c(s_t, u_t) + c_T(u_T) \\
\text{s.t.} \; & s_0 = s_{\text{ini}} \\
& s_{t+1} = f(s_t, u_t) \qquad \forall t \text{ in } 0, \ldots, t-1
\end{aligned}$$

We define the *value function* $V(t, s)$ to be equal to the cost of an optimal solution of the problem on $t, t+1, \ldots, T$ if the system is in s at t. Using the same argument as for the shortest path problem, we can show

$$V(T, s) = c_T(s) \tag{5.6a}$$

$$V(t, s) = \min_u c(s, u) + V(t+1, f(s, u)) \quad \forall t \in \{0, 1, \ldots, T-1\} \tag{5.6b}$$

Hence, $V(T, s)$ can be computed using a backward recursion in $O(|\mathcal{S}| \cdot |\mathcal{U}| \cdot T)$, and we can rebuild the optimal solution by "following the argmin" in the dynamic programming equation.

Dynamic programming is a general strategy that can be applied to more general problems than discrete dynamic systems. Exercise (5.20) provides an example of application to a problem that cannot be expressed as a discrete dynamic system.

Exercise 5.2. Prove that Equation (5.6) is satisfied. △

Exercise 5.3. *Forward dynamic programing.* An alternative approach consists in using a value function $V(t, s)$ equal to the optimal solution of the problem on $0, \ldots, t$ is the systems ends in state s at time t. Write the dynamic programming equation in that case.

Solution.

$$\begin{aligned}
V(0, s) &= \begin{cases} 0 & \text{if } s = s_0, \\ +\infty & \text{otherwise.} \end{cases} \\
V(t+1, s) &= \min_{(s', u) \colon f(s', u) = s} V(t, s') + c(s', u)
\end{aligned}$$

□

△

Markov Decision Processes

Consider now the stochastic case where there is some uncertainty on the transitions. The state at time t becomes a random variables S_t, as well as the action U_t. To take its decision U_t decision maker has access to the information $S_0, U_0, \dots, S_{t-1}, U_{t-1}, S_t$. A policy is a mapping that associates to the value taken by $(S_0, U_0, \dots, S_{t-1}, U_{t-1}, S_t)$ a decision u_t. The cost of a policy is defined as the expectation

$$c_T(S_T) + \mathbb{E}\Big(\sum_{t=0}^{T-1} c(S_t, U_t)\Big)$$

under that policy. An optimal policy is a policy of minimum cost. The dynamic is *Markovian* if S_{t+1} is independent form $S_0, U_0, \dots, S_{t-1}, U_{t-1}$ given S_t.

Lemma 5.6. *If the dynamic is Markovian, there is an optimal policy where the value of U_t is chosen based only on the value of S_t.*

Exercise 5.4. *Exercise on stochastic processes* 😎. Prove Lemma 5.6. (Indication: introduce the filtration $(\mathcal{F}_t)_t$ induced by the (S_t, U_t).)

Solution. Choose arbitrarily an optimal policy (non-necessarily Markovian). Define a Markovian policy by choosing, for any state s that is visited by S_t with non-zero probability, an action u_t among those chosen with non-zero probability when $S_t = s_t$ by the non-Markovian policy. Then a backward induction on t' enables to prove that, fixing the value of S_t to s, the value of

$$\mathbb{E}\Big(c_T(S_T) + \sum_{t=t'}^{T-1} c(S_t, U_t)\Big)$$

is identical under the Markovian policy and the non-Markovian policy. Two arguments are used of prove the result for t' given the results at $t'+1$: the independence of S_t with respect to $\mathcal{F}_{t-1}$ given S_{t_1}, U_{t-1}, and the optimality of the non-Markovian policy. □

△

In the rest of the section, we place ourselves under the Markovian assumption. The decision problem considered is called a *Markov Decision Process (MDP)* In this setting, $\mathbb{P}(S_{t+1} = s' | S_t = s, U_t = u)$ does not depend on the policy. We denote introduce the notation

$$p^t_{s'|su} = \mathbb{P}(S_{t+1} = s' | S_t = s, U_t = u). \tag{5.7}$$

Equation (5.7) plays in the stochastic setting the role that ways played by Equation (5.5) in the deterministic setting.

Dynamic programming can still be applied to find an optimal policy of a MDP. In that case, we define $V(s,t)$ to be the value of

$$\mathbb{E}\Big(c_T(S_T) + \sum_{t=t'}^{T-1} c(S_t, U_t) | S_t = s\Big)$$

under an optimal policy. The dynamic programming equation becomes

$$\begin{aligned} V(T,s) &= C_T(s) \\ V(t,s) &= \min_u c(s,u) + \textstyle\sum_{s' \in S} p^t_{s'|su} V(t+1, s') \end{aligned} \tag{5.8}$$

Exercise 5.5. Prove that Equation (5.8) is true.

Solution. We have
$$V(t,s) = \min_u \mathbb{E}\Big(\sum_{t'=t}^{T-1} c(S_{t'}, U_{t'}) | S_t = s, U_t = u\Big)$$
□

and the result follows. △

Exercise 5.6. Explain why, in this stochastic setting, we cannot write the forward dynamic programming equation of Exercise (5.3) as in the deterministic case.

Solution. There is no stochastic analogue to the set $\{(s', u) \colon f(s', u) = s\}$ used in the deterministic case. □

△

5.2 Non negative costs and Dijkstra Algorithm

In this section, we consider the case where $c_a \geq 0$ for all $a \in A$.

Proposition 5.7. *Let $D = (V, A)$ be a digraph, $k < |V|$ be an integer, and V_k be a set of k nearest vertices of o (including o). Let v be a vertex in $V \backslash V_k$ minimizing $\min\limits_{u \in N^-(v) \cap V_k} c_{u,v}$. Then $V_k \cup \{v\}$ is a set of $k+1$ nearest vertices to v.*

Exercise 5.7. Prove Proposition 5.7. △

Algorithm 3 Dijkstra algorithm

```
1: Input: a digraph $D = (V, A)$, costs $\boldsymbol{c} \in \mathbb{Q}_+^A$
2: Initialization: $U \leftarrow \emptyset$, $d_v \leftarrow \begin{cases} 0 & \text{if } v = o, \\ +\infty & \text{otherwise.} \end{cases}$
3: while $V \backslash U \neq \emptyset$ do
4:     Let $v$ in $V \backslash U$ be such that $d(v) = \min_{u \in V \backslash u} d_u$.
5:     Add $v$ to $u$
6:     $d_w \leftarrow \min\big(d_w, d_v + c_{(v,w)}\big)$ for all $w \in N^+(v)$.
7: end while
8: return $\boldsymbol{d}$.
```

Algorithm 3 states *Dijkstra* algorithm for graphs with arcs of multiplicity at most one. It is easily generalized to graphs with arbitrary multiplicity.

Proposition 5.8. *Dijkstra algorithm converges in $O(m + n\log(n))$ if the list L of vertices to treat is implemented as a Fibonacci heap, and* $\boldsymbol{d}$ *returned is such $d(v)$ is the length of a shortest o-v path for all v.*

5.3 A* algorithm

The algorithm we introduce enables to speed-up the resolution, at the cost of a long preprocessing time.

5.3.1 Algorithm

Proposition 5.9. *Let c^{ub}_{od} be an upper bound on the cost of a shortest o-d path, P be an o-v path, and b_v be a lower bounds on the cost of a shortest o-d path. If $c_P + b_v > c^{\mathrm{ub}}_{od}$, then P is not the subpath of an optimal path.*

Algorithm 4 A*

```
Input: A digraph $D = (V, A)$, costs $\boldsymbol{c}$ in $\mathbb{Q}^A$, bounds $\boldsymbol{b}$ in $\mathbb{Q}^V$.
Initialization: $c^{\mathrm{ub}}_{od} \leftarrow +\infty$, $L \leftarrow \{$empty path in $o\}$ with key 0
while $L$ is not empty do
  Extract from $L$ a path $P$ of minimum key
  Let $v$ be the destination of $P$
  if $v = d$ and $c_P < c^{\mathrm{ub}}_{od}$ then
    $c^{\mathrm{ub}}_{od} \leftarrow c_P$
  else
    for all $w \in N^+(v)$ such that $c_P + c_{vw} + b_w < c^{\mathrm{ub}}_{od}$ do
      Add $P$ followed by $(v, w)$ to $L$ with key $c_P + c_{vw} + b_w$.
    end for
  end if
end while
return: $c^{\mathrm{ub}}_{od}$
```

Proposition 5.10. *If $\boldsymbol{c} > 0$ and $\boldsymbol{b} \geq 0$, then A* converges after a finite number of iterations. Furthermore, if $b_v \leq c_P$ for all v-d paths P, then c^{ub}_{od} returned is the cost of a shortest o-v path.*

A* can be proved to converge under more general conditions than those of Proposition 5.10.

5.3.2 Generating bounds 😎

5.4 Exercises

Exercise 5.8. Trouver la plus courte s-t chaîne du graphe de la Figure 5.1 à l'aide de l'algorithme de Dijkstra. △

Exercise 5.9. Deux bassins s et t doivent être reliés par une rivière artificielle dans une région vallonnée. Cette rivière doit permettre de faire passer l'eau de s à t. Le graphe orienté de la Figure 18.1 indique les possibilités : un arc est une portion de rivière possible et l'orientation de l'arc indique le sens d'écoulement de l'eau sur cette portion. Le coût attaché à l'arc est le coût d'ouverture de cette portion. Calculer la rivière de coût minimum reliant s à t à l'aide d'un algorithme du cours (essayer de choisir le plus efficace dans ce contexte). △

Exercise 5.10. On se donne un graphe non-orienté $G = (V, E)$, deux sommets s et t, et une fonction $w : E \to \mathbb{R}_+$ telle que $w(e) \geq 1$ pour toute arête e. Montrer que l'on peut trouver en temps polynomial une s-t chaîne P minimisant $\prod_{e \in E(P)} w(e)$. △

Solution. Il suffit d'appliquer la fonction log sur les poids. □

Exercise 5.11. On se donne un graphe non-orienté $G = (V, E)$, deux sommets s et t, et une fonction $w : E \to \mathbb{R}$. Montrer que l'on peut adapter l'algorithme de Dijkstra afin d'obtenir un algorithme polynomial calculant une s-t chaîne P minimisant $\max_{e \in E(P)} w(e)$. △

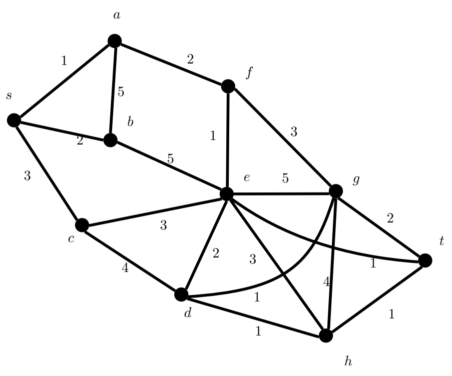


Figure 5.1:

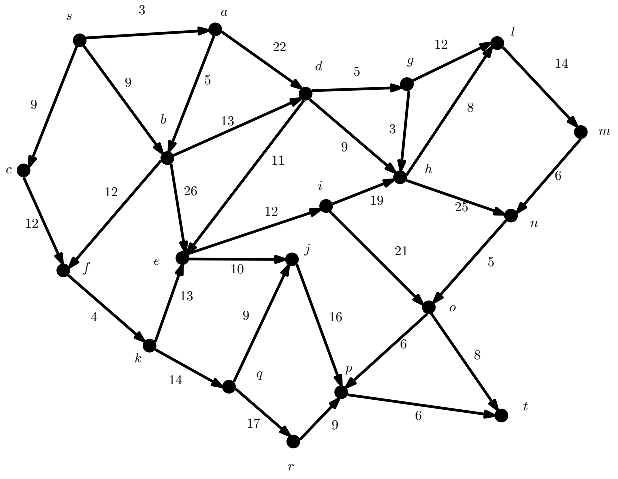


Figure 5.2:

Solution. L'algorithme reste le même, mais on met à jour l'étiquette d'un sommet v en faisant $\lambda(v) := \max(\lambda(u), w(uv))$. La preuve est identique. Noter que pour ce problème, le fait que la fonction max soit croissante permet de travailler avec des poids de signe quelconque. □

Exercise 5.12. *Intervalles disjoints et locations rentables)* On suppose donnée une collection finie $\mathcal{C}$ d'intervalles fermés bornés de la droite réelle. On dispose d'une application poids $w : \mathcal{C} \to \mathbb{R}_+$.

1. Montrer que l'on sait trouver le sous-ensemble d'intervalles de $\mathcal{C}$ deux à deux disjoints de poids maximal en temps polynomial.

2. Des demandes de location pour un appartement sont supposées connues, chacune d'entre elles étant caractérisée par un instant de début, un instant de fin et le gain qu'apporte la satisfaction de cette demande. Montrer que grâce à la question 1., on sait proposer une stratégie rapide maximisant le gain total de la location de l'appartement.

△

Solution.

1. On considère le graphe orienté D dont l'ensemble des sommets V est la collection d'intervalles $\mathcal{C}$ (on identifie alors V et $\mathcal{C}$) et dans lequel on a un arc (I, J) si l'extrémité supérieure de I est strictement inférieure à l'extrémité inférieure de J. C'est un graphe acircuitique, et les sommets des chemins élémentaires sont précisement les sous-ensembles d'intervalles deux à deux disjoints. En mettant comme poids sur tout arc (I, J) la quantité $w(I)$, et en ajoutant un dernier sommet X et tous les arcs (I, X) avec $I \in \mathcal{C}$ que l'on pondère avec $w(I)$, chercher le sous-ensemble d'intervalles de $\mathcal{C}$ deux à deux disjoints de poids maximal revient à chercher le plus long chemin de D pour la pondération w, ce qui se fait par un algorithme de programmation dynamique (le graphe étant acircuitique). Pour se ramener totalement au cas du cours, on peut également ajouter un sommet Y et tous les arcs (Y, I) avec $I \in \mathcal{C}$ que l'on pondère avec 0, et chercher le chemin de Y à X de plus grand poids.

2. Une demande de location se modélise par un intervalle de la droite réelle dont les extrémités sont les debut et fin de la location, pondéré par la gain qui serait obtenue en satisfaisant cette demande. Maximiser le gain revient alors à chercher le sous-ensemble d'intervalles disjoints (les locations ne pouvant se recouvrir) de plus grand poids.

□

Exercise 5.13. *(Plus courts chemins avec contrainte de nombre d'arcs)* On se donne un graphe orienté $D = (V, A)$ sans circuit, muni d'une fonction de coût $c : A \to \mathbb{R}_+$. On se fixe un entier b. Etant donnés deux sommets s et t, on souhaite trouver le s-t chemin de plus petit coût ayant au plus b arcs. Expliquer pourquoi ce problème peut se résoudre à l'aide d'un algorithme polynomial. △

Solution. On définit

$$\pi(v, k) := \text{coût minimal d'un } s\text{-}v \text{ chemin possédant exactement } k \text{ arcs.}$$

On peut alors écrire l'équation de programmation dynamique

$$\pi(v, k+1) = \min\left(\pi(u, k) + c((u, v))\right)$$

où le minimum est pris sur les sommets u antécédents de v. On répond à la question en comparant les valeurs de $\pi(t, k)$ pour k variant de 0 à b. Chacune de ces valeurs se calcule en temps polynomial $O(|V|^2)$, et la dernière se fait en $b + 1$ opérations, ce qui est polynomial car $b \leq |A|$. □

Exercise 5.14. *Achat de poutres* La construction d'une structure métallique nécessite l'utilisation d'un ensemble $\mathcal{P}$ de poutres. On va donc acheter ces poutres. Pour chaque poutre $p \in \mathcal{P}$, on connaît σ_p la valeur minimale que doit prendre sa section. On dispose d'un catalogue dans lequel sont proposés n types de poutres, chaque type $i \in \{1, \ldots, n\}$ étant caractérisé par une section S_i. Le prix d'une poutre de section S_i est $q_i > 0$. On suppose que les S_i et les q_i sont ordonnées par valeurs croissantes, i.e. que si $i < j$, alors $S_i < S_j$ et $q_i < q_j$.

On souhaite pouvoir réaliser la structure en minimisant les coûts liés aux achats de poutres. Pour toute poutre p, tout type i tel que $S_i \geq \sigma_p$ convient. On supposera que $S_n \geq \max_{p \in \mathcal{P}} \sigma_p$.

1. Montrer que ce problème de minimisation peut être résolu en temps polynomial par un algorithme direct tout simple.

On suppose maintenant que pour des raisons de gestion, le nombre de types différents que l'entreprise s'autorise à acheter est borné par un certain k entier. Noter que si $k \geq n$, le problème est identique à celui de la question précédente.

On considère un graphe orienté $D = (V, A)$ dont les sommets sont les types de poutres ainsi que deux autres sommets s et t. Il y a donc $n + 2$ sommets. Les arcs sont de trois types

- les arcs (s, i) pour tout $i \in \{1, \ldots, n\}$,
- les arcs (i, j) pour tous $i, j \in \{1, \ldots, n\}$ tels que $i < j$ et
- l'arc (x, t) où x est le plus petit indice tel que $S_x \geq \max_{p \in \mathcal{P}} \sigma_p$.

2. En mettant des coûts judicieux sur les arcs de D, montrer que ce problème peut se résoudre par l'algorithme proposé à l'exercice (**??**), en choisissant la bonne valeur de b.

3. Le problème reste-t-il polynomial si la condition devient : le type i convient pour p si et seulement si
$$(1 + \theta)\sigma_p \geq S_i \geq \sigma_p$$
pour un certain $\theta > 0$ fixé ? Justifier la réponse.

Solution. 1. Pour chaque poutre p, on achète une poutre de type $i(p)$, où $i(p)$ est le plus petit indice i tel que $S_i \geq \sigma_p$.

2. Sur un arc (s, i), on met un coût égal à $q_j g_i$, où g_i est le nombre de poutres $p \in \mathcal{P}$ telles que $\sigma_p \in]0, S_i]$. Sur un arc (i, j) on met un coût égal à $q_j f_{ij}$, où f_{ij} est le nombre de poutres $p \in \mathcal{P}$ telles que $\sigma_p \in]S_i, S_j]$. Sur l'arc (x, t), on met un coût égal à 0. On pose $b = k + 1$.

   Les types sélectionnés dans une solution optimale du problème constituent avec s et t un s-t chemin dans ce graphe, utilisant au plus b arcs. Noter que le type x est forcément utilisé dans une solution optimale. On peut vérifier que le coût du chemin et le coût de la solution coïncident. Réciproquement, tout s-t chemin donne une solution réalisable de même coût.

3. S'il existe une poutre p telle que $\sigma_p \in]S_i, S_j]$ et $(1 + \theta)\sigma_p < S_j$, on supprime l'arc (i, j) du graphe D. De même pour les arcs (s, i). Dans ce nouveau graphe, la méthode de la question précédente permet encore de résoudre le problème.

□

△

Exercise 5.15. *(Choisir le trajet d'un remonte-pente)* Une entreprise de travaux publics a obtenu le marché pour la construction d'un remonte-pente sur le flanc d'une montagne. Une telle construction consiste à placer des pylônes, parmi des emplacements potentiels, et y faire passer les câbles, de manière à relier une station au bas de la montagne à une station au haut de la montagne. On suppose que ces deux stations sont fixées. L'entreprise connaît les emplacements potentiels et on note V leur ensemble, auquel on adjoint deux emplacements s^{bas} et s^{haut} correspondant aux stations bas et haut. Les contraintes topographiques font que l'on peut ou pas faire se succéder deux pylônes à des emplacements u et v donnés : l'ensemble des couples $(u, v) \in V^2$ tels que v peut succéder directement à u est un ensemble connu noté A. On a donc un graphe orienté $D = (V, A)$ dans lequel on cherche un s^{bas}-s^{haut} chemin. L'entreprise cherchant à minimiser ses coûts, on cherche un s^{bas}-s^{haut} chemin de plus petit coût.

La particularité du problème ici est que le coût d'un chemin se calcule à l'aide d'une fonction f à valeur dans $\mathbb{R}_+$, définie sur les couples (a, a') d'arcs tels que la tête de a est la queue de a' (l'arc a arrive sur le sommet d'où part l'arc a'). Le coût d'un chemin décrit par la séquence de ses arcs $a_1, \ldots, a_\ell$, où $a_1 \in \delta^+(s^{bas})$ et $a_\ell \in \delta^-(s^{haut})$, est

$$\sum_{i=1}^{\ell-1} f\big((a_i, a_{i+1})\big).$$

Montrer que ce problème se résout en temps polynomial.

Solution. On construit un graphe $\mathcal{D} = (\mathcal{V}, \mathcal{A})$ dont les sommets $\mathcal{V}$ sont s^{bas}, s^{haut} et tous les couples d'arcs (a, a') tels que la tête de a est la queue de a'. Les arcs de $\mathcal{A}$ sont de la forme :

- $\big(s^{bas}, (a, a')\big)$ pour tout $(a, a') \in \mathcal{V}$ tel que $a \in \delta^+(s^{bas})$.
- $\big((a, a'), s^{haut}\big)$ pour tout $(a, a') \in \mathcal{V}$ tel que $a' \in \delta^-(s^{haut})$.
- $((a, a'), (a', a''))$ pour tout $(a, a'), (a', a'') \in \mathcal{V}$.

On définit le coût d'un arc $(\cdot, (a, a'))$ de $\mathcal{D}$ comme étant $f(a, a')$ et celui d'un arc $\big((a, a'), s^{haut}\big)$ comme étant nul. Un s^{haut}-s^{bas} chemin dans D induit un s^{haut}-s^{bas} de $\mathcal{D}$ et réciproquement. N'importe quel algorithme de calcul de plus court chemin dans un graphe orienté avec des poids positifs convient. $\square$

$\triangle$

Exercise 5.16. *(Gestion de stock)* Un stock suit une dynamique $x_{k+1} = x_k - d_k + u_k$, avec $x_1 = 2$ et

k	1	2	3
d_k	3	2	4

La quantité d_k est la demande pour la période k (supposée connue). $x_k \in \mathbb{N}$ est nombre d'unités disponibles en début de période k. u_k est le nombre d'unités commandées (et reçues immédiatement) en début de période k . On a une capacité maximale de stockage de 6, i.e que l'on doit toujours avoir $x_k + u_k \leq 6$ (ou de manière équivalente $x_{k+1} + d_k \leq 6$).

Le coût de gestion de stock pour la période k se décompose en un coût d'approvisionnement et un coût de stockage et s'écrit $4u_k + 3 + x_{k+1}$ si $u_k \neq 0$ et x_{k+1} sinon.

En utilisant une approche "programmation dynamique", trouver la stratégie qui minimise le coût total de la gestion du stock. $\triangle$

Exercise 5.17. *(Remplacement de machine)* Une entreprise vient de faire l'acquisition d'une machine neuve, dont elle a besoin pour assurer son bon fonctionnement pour les cinq années à venir. Une telle machine a une durée de vie de trois ans. A la fin de chaque année, l'entreprise peut décider de vendre sa machine et d'en racheter une neuve pour 100 000€. Le bénéfice annuel assuré par la machine est noté b_k, son coût de maintenance c_k et son prix de revente p_k. Ces quantités dépendent de l'âge k de la machine et sont indiquées dans le tableau ci-dessous.

	Age k de la machine en début d'année		
k	0	1	2
b_k	80 000€	60 000€	30 000€
c_k	10 000€	14 000€	22 000€
p_k	50 000€	24 000€	10 000€

△

En utilisant une approche "programmation dynamique", proposer la stratégie de remplacement de la machine maximisant le profit de l'entreprise, sachant qu'à la fin de ces cinq années elle n'aura plus besoin de la machine.

Solution. Ce problème se résout par la programmation dynamique. Les états sont les âges possibles de la machine en début d'année. Les périodes sont les années. On s'intéresse aux quantités

$\pi(t,k) =$ profit maximal possible en terminant la $t$ème année en possédant une machine d'âge k.

On souhaite trouver $\max_{k\in\{1,2,3\}}(\pi(5,k)+p_k)$.

$\pi(t,k)$ satisfait les relations suivantes:

$$\pi(1,k) = \begin{cases} 0 & \text{pour tout } k \in \{2,3\} \\ b_0 - c_0 & \text{pour } k = 0. \end{cases}$$

$$\pi(t+1,k+1) = \pi(t,k) + b_k - c_k \quad \text{si } k \in \{1,2\}$$

$$\pi(t+1,1) = \max_{k\in\{1,2,3\}} (\pi(t,k) + p_{k-1} - 100000 + b_0 - c_0) \text{ si } t \in \{1,2,3,4\}$$

On peut alors remplir le tableau des valeurs de π

t / k	1	2	3	4	5
1	70000	90000	110000	90000	150000
2	0	116000	136000	156000	136000
3	0	0	124000	144000	164000

Le profit maximal de l'entreprise est donc $\max(150000 + 50000, 136000 + 24000, 164000 + 10000) = 200000$. Une stratégie optimale consiste à garder la machine deux ans, puis la vendre et en racheter une nouvelle, encore une fois pour deux ans.

□

Exercise 5.18. *(Un problème d'intérimaires)* Une entreprise a une tâche à réaliser, qui va prendre 5 mois. Pour chaque mois, elle a besoin des nombres suivants d'intérimaires en plus de ses salariés.

Mois	Nombre d'intérimaires
1	10
2	7
3	9
4	8
5	11

Embaucher un intérimaire coûte 800€ (embauche + formation). La fin d'une embauche coûte 1200€. Enfin, le coût mensuel d'un des ces intérimaires est de 1600€ (salaire + charges). En utilisant une approche "programmation dynamique", trouver la stratégie de recrutement qui minimise les coûts. △

Solution. On pose

$$\pi(i, x) := \text{coût optimal d'une stratégie terminant sur le mois } i \text{ avec } x \text{ employés,}$$

et

$$\pi(0, x) := \begin{cases} 0 & \text{si } x = 0 \\ +\infty & \text{si } x > 0. \end{cases}$$

Noter qu'il n'ai jamais intéressant d'avoir strictement plus de 11 employés sur un mois quelconque. Pour $i \geq 1$ on a alors en notant $d(i)$ la demande en intérimaires sur le mois i :

$$\pi(i, x) = \begin{cases} \min\limits_{y \in \{0,1,\ldots,11\}} (\pi(i-1, y) + c(y, x)) & \text{si } x \in \{d(i), \ldots, 11\} \\ +\infty & \text{sinon.} \end{cases}$$

avec

$$c(y, x) := \begin{cases} 1600x + 800(x - y) & \text{si } y \leq x \\ 1600x + 1200(y - x) & \text{sinon.} \end{cases}$$

On calcule alors aisément de proche en proche les valeurs de $\pi(i, x)$ pour tout i et tout x. La stratégie optimale s'obtient en calculant la valeur de $\pi(5, 11)$. (Le coût total optimal est $\pi(5, 11) + 11 \times 1200$ euros, en supposant que l'on doit mettre fin aux contrats des 11 intérimaires du mois 5.)

□

Exercise 5.19. *(Plus long sous-mot commun)* Un *mot* w sur un alphabet Σ est une suite finie $w_1 w_2 \ldots w_n$ d'élément de Σ. La quantité n est appelé *longueur* du mot w. On dit qu'un mot $w' = w'_1, \ldots w'_{n'}$ de longueur n' est un *sous-mot* de w s'il existe une application strictement croissante

$$f : \{1, 2, \ldots, n'\} \longrightarrow \{1, 2, \ldots, n\}$$

telle que $w'_i = w_{f(i)}$ pour tout $i \in \{1, 2, \ldots, n'\}$. En d'autres termes, en effaçant des lettres de w, on peut obtenir w'. Par exemple : ABC est un sous-mot de $AEBBAC$, mais n'est pas un sous-mot de $AECAB$.

On se donne deux mots w_1 et w_2 sur un alphabet Σ. On cherche leur sous-mot commun w' le plus long. Montrer que c'est un problème de type programmation dynamique, et qu'il peut se résoudre en $O(n_1 n_2)$, où n_1 et n_2 sont les longueurs de w_1 et de w_2.

(Ce problème a des applications en biologie, lorsque on veut évaluer la "distance" séparant deux brins d'ADN.) △

Exercise 5.20. *(Problème du voyageur de commerce)* On se donne un graphe complet $K_n = (V, E)$, avec une fonction de coût $c : E \to \mathbb{R}_+$. On cherche le cycle hamiltonien de plus petit coût.

1. En fixant au préalable un sommet s et en utilisant comme états les couples (X, v) avec $X \subseteq V \setminus \{s\}$ et $v \in V \setminus X$, écrire une équation de programmation dynamique permettant de résoudre le problème.

2. Comparer la complexité de l'algorithme qu'elle implique à la complexité d'un algorithme naïf énumérant toutes les solutions possibles.

△

Chapter 6

Flows

6.1 Maximum s-t flows, minimum s-t cuts

6.1.1 Problem statement

Let $D = (V, A)$ be a digraph, s and t be two distinct vertices in V, and $u : \boldsymbol{A} \to \mathbb{R}_+$ be a *capacity* function.

Definition 6.1. *An s-t flow is a function $f : A \to \mathbb{R}_+$ such that*

$$\sum_{a\in\delta^-(v)} f(a) = \sum_{a\in\delta^+(v)} f(a) \quad \textit{for all } v \in V\backslash\{s,t\}.$$

It is an s-t flow subject to u or under u if $f(a) \le u(a)$ for all a in A. The value of a s-t flow f is

$$\mathrm{val}(f) = \sum_{a\in\delta^+(s)} f(a) - \sum_{a\in\delta^-(s)} f(a).$$

MAX FLOW
Input. A digraph $D = (V, A)$, two distinct vertices s and t in V, and a *capacity* function $u : \boldsymbol{A} \to \mathbb{R}_+$.
Output. An s-t flow under u of maximum value.

We recall the definition of cut.

Definition 6.2. *An s-t cut is a set of arcs B such that $B = \delta^+(U)$ where $U \subseteq V$ contains s but not t. The capacity $u(B)$ of a cut b is the sum of the capacity of its arcs.*

$$u(B) = \sum_{a\in B} u(A).$$

MIN CUT
Input. A digraph $D = (V, A)$, two distinct vertices s and t in V, and a *capacity* function $u : \boldsymbol{A} \to \mathbb{R}_+$.
Output. An s-t cut B of minimum capacity $u(B)$.

6.1.2 Max flow min cut theorem

Remark the following link between flows and cut.

Proposition 6.3. *Let f be an s-t flow under u and $B = \delta^+(U)$ be an s-t cut. We have*

$$\mathrm{val}(f) = \sum_{a\in\delta^+(U)} f(a) - \sum_{a\in\delta^-(U)} f(a),$$

and

$$f(B) \leq u(B).$$

Proof. Following the definition of a flow f under u, we have

$$\begin{aligned}\mathrm{val}(f) &= \sum_{a\in\delta^+(s)} f(a) - \sum_{a\in\delta^-(s)} f(a) + \sum_{v\in U\backslash\{s\}} \underbrace{\sum_{a\in\delta(v)} f(a) - \sum_{a\in\delta^-(v)} f(a)}_{0} \\ &= \sum_{a\in\delta^+(U)} f(a) - \sum_{a\in\delta^-(U)} f(a)\end{aligned}$$

which gives the first result. The second result then follows from

$$0 \leq f \leq u.$$

□

Consider and instance of $(D = (V, A), s, t, u)$ of the MAXIMUM FLOW problem. Given an arc $a = (u, v)$ in A, we denote by $\overleftarrow{a}$ a new arc (v, u) in A. Given an s-t flow f, the *residual capacities* capacities $u_f : \overleftrightarrow{A} \to \mathbb{R}_+$ are defined by

$$\begin{cases} u_f(a) &= u(a) - f(a) \\ u_f(\overleftarrow{a}) &= f(a) \end{cases} \quad \text{for all } a \in A, \quad \text{and} \quad \overleftrightarrow{A} = A \cup \{\leftarrow a\colon a \in A\}$$

The *residual graph* is the capacitated graph $\overleftrightarrow{D_f} = (V, A_f)$ where

$$A_f = \{a \in \overleftrightarrow{A} : u_f(a) > 0\}.$$

An *f-augmenting path* in an s-t path in the residual graph. To *augment* f by γ along an s-t path consists means, for each a in A, to increase $f(a)$ by γ if $a \in P$ and to decrease it by γ if $\leftarrow a$ in P.

Theorem 6.4. *An s-t flow is maximum if there is no f-augmenting path.*

Proof. Suppose that there is no augmenting path, and the U denote the connected component of s in the residual graph. Then $\mathrm{val}(f) = u(U)$ and Proposition 6.3 ensures that f is maximum. Conversely, suppose that there is an f-augmenting path P in the residual graph. Then f can be augmented by $\gamma = \min\limits_{a\in P} u_f(a)$. □

As an immediate corollary of Proposition 6.3 and Theorem 4, we obtain the main theorem of flow theory.

Theorem 6.5. (Max-flow min-cut Theorem) *The maximum value of an s-t flow is equal to the minimum value of an s-t cut.*

Exercise 6.1. Prove that given a maximum flow, we can find a minimum cut in $O(m)$. △

Solution. Follows from the proof of Theorem 6.4. □

6.1.3 Edmonds-Karp algorithm

Edmonds-Karp Algorithm (Algorithm 5) is based on this result.

Algorithm 5 Edmonds-Karp Algorithm

```
1: Input: a digraph $D = (V, A)$, $s$, $t \in A$, and $u : A \to \mathbb{R}_+$.
2: Output: an $s$-$t$ flow $f$ of maximum value.
3: $f(a) \leftarrow 0$ for all $a \in A$;
4: Find an $f$-augmenting path $P$ with a minimum number of arc; Stop if there is none;
5: Augment $f$ by $\min_{a \in P} u_f(a)$;
```

Theorem 6.6. *(Edmonds and Karp [1972]) Algorithm 5 converges after at most $\frac{mn}{2}$ augmentation, and therefore solves the* MAXIMUM FLOW *problem in $O(mn^2)$.*

Proof. TO DO □

Remark that if capacities are integral, flow is always augmented by and integer number. Hence, we obtain as a corollary the following theorem.

Corollary 6.7. *If the capacity u is integer, then there exists an integer maximum flow, and Edmonds-Karp algorithm finds one.*

Remark 6.1. The first algorithm for s-t flows is the *Ford and Fulkerson* algorithm, which is obtained by removing "with a minimum number of arc" in Edmonds Karp algorithm. △

6.2 Minimum cost flow

6.2.1 Problem statement

Let $D = (V, A)$ be a digraph, $\ell : A \to \mathbb{R}_+$ and $u : A \to \mathbb{R}_+$ be capacities such that $\ell \leq u$, and $b : V \to \mathbb{R}$ be such that

$$\sum_v b(v) = 0. \tag{6.1}$$

Definition 6.8. *Given D, ℓ, u, and b as above, a b-flow is an application $f : A \to \mathbb{R}_+$ such that*

$$\sum_{a \in \delta^+(v)} f(a) - \sum_{a \in \delta^-(v)} f(a) = b(v) \quad \text{for all } v \text{ in } V,$$

and $\ell(a) \leq f(a) \leq u(a)$ for all a in A. A circulation is a b-flow with $b = 0$.

Given a cost function $c : A \to \mathbb{R}$, the *cost* of a b-flow is

$$\sum_{a \in A} c(a) f(a).$$

MINIMUM COST FLOW
Input. A digraph $D = (V, A)$, $\ell : A \to \mathbb{R}_+$ and $u : A \to \mathbb{R}_+$ such that $\ell \leq u$, $b : V \to \mathbb{R}$ such that $\sum_{v \in V} b(v) = 0$, and $c : A \to \mathbb{R}_+$
Output. A b-flow of minimum cost.

6.2.2 Optimality criterion

Let f be a b-flow. The *residual graph* is the digraph $G_f = (V, A_f)$ where

$$A_f = \big\{a \in A\colon u(a) - f(a) > 0\big\} \cup \big\{\overleftarrow{a} : a \in A \text{ and } f(a) > \ell(a)\big\}$$

Again, we define, for each a in A

$$u_f(a) = u(a) - f(a) \quad \text{and} \quad u_f(\overleftarrow{a}) = f(a) - \ell(a).$$

We extend c to $\overleftarrow{A}$ by $c(\overleftarrow{a}) = -c(a)$. An *$f$-augmenting cycle* is a cycle in G_f, and we define

$$c(C) = \sum_{a \in C} c(a).$$

Theorem 6.9. *A b-flow f is of minimum cost if there is no f-augmenting cycle C such that $c(C) < 0$.*

Exercise 6.2. Proof of Theorem 6.9

1. Let g and f be two b-flows. Show that $g - f$ is a circulation in G_f.
2. Show that any circulation h can be decomposed as $\sum_{C \in \mathcal{C}} a_C \mathbb{1}_C$ where $\mathcal{C}$ is a collection of cycles, $a_C \in \mathbb{R}_+$, and $\mathbb{1}_C : A \to \{0,1\}$ is the indicator function of the arcs of C.
3. Deduce the Theorem 6.9.

△

6.2.3 Minimum mean cycle-canceling algorithm

Given an instance of the minimum cost flow and a b-flow f, the mean cost of a cycle in the residual graph is

$$\bar{c}(C) = \frac{c(C)}{|C|}.$$

Exercise 6.3. Prove that a cycle of minimum mean-cost can be computed in polynomial time (in $O(mn)$). △

Solution. A cycle of minimum mean-cost can be computed in $O(mn)$ using the dynamic programming algorithm of Equation (5.1). □

Goldberg and Tarjan proposed Algorithm 6 to solve the MINIMUM COST FLOW PROBLEM.

Algorithm 6 Minimum mean cycle-canceling algorithm

```
1: input: A digraph $D = (V, A)$, $\ell : A \to \mathbb{R}_+$ and $u : A \to \mathbb{R}_+$ such that $\ell \leq u$, $b : V \to \mathbb{R}$
   such that $\sum_{v \in V} b(v) = 0$, and $c : A \to \mathbb{R}_+$
2: output: A $b$-flow of minimum cost.
3: find a $b$-flow $f$
4: find a minimum mean cost cycle in the residual graph $G_l$. If $C$ has negative total cost, stop
5: augment the flow along $C$ by $\min_{a \in C} u_f(a)$.
```

Exercise 6.4. Show that an b-flow can be found at Step 1 by solving a maximum s-t flow problem in the digraph where vertices s and t have been added. △

Theorem 6.10. *The minimum mean cycle-canceling algorithm find a minimum cost b-flow in $O(m^3 n^2 \log(n))$.*

6.3 Linear programming for flows

The maximum flow problem can be solved using the following linear program

$$\begin{aligned}
\max \quad & \sum_{a\in\delta^+(s)} x_a - \sum_{a\in\delta^-(s)} x_a && && (6.2a)\\
\text{s.t.} \quad & \sum_{a\in\delta^-(v)} x_a = \sum_{a\in\delta^+(v)} x_a, && \forall v \in V\backslash\{s,t\} && (6.2b)\\
& x_a \le u_a && \forall a \in A && (6.2c)\\
& x_a \ge 0 && \forall a \in A && (6.2d)
\end{aligned}$$

Exercise 6.5. Give an LP for the minimum cost flow problem. △

A practical consequence, using a LP solver to deal with a flow problem is a good idea, both in terms of coding time and computing time.

Proposition 6.11. *The flow matrix is totally unimodular.*

Proof. See Corollary 9.11 for total unimodularity. □

Proposition 6.12. *The minimum capacity s-t cut problem is the Lagrangian dual of the maximum s-t flow.*

We prove Proposition 6.12 as an example of the dualization of a linear program (Skill 8.1), which is a skill that must be mastered at the end of the course.

Proof of Proposition 6.12. It will be handy to consider the following equivalent version of (6.2).

$$\begin{aligned}
\max \quad & q && && (6.3a)\\
\text{s.t.} \quad & \sum_{a\in\delta^+(s)} x_a - \sum_{a\in\delta^-(s)} x_a = q && && (6.3b)\\
& \sum_{a\in\delta^+(t)} x_a - \sum_{a\in\delta^-(t)} x_a = -q && && (6.3c)\\
& \sum_{a\in\delta^+(v)} x_a - \sum_{a\in\delta^-(v)} x_a = 0, && \forall v \in V\backslash\{s,t\} && (6.3d)\\
& x_a \le u_a && \forall a \in A && (6.3e)\\
& x_a \ge 0 && \forall a \in A && (6.3f)
\end{aligned}$$

Introducing dual variables $y_v \in \mathbb{R}$ for Equations (6.3b) to (6.3d) and $z_a \ge 0$ for Equation 6.3e, we get the Lagrangian

$$\begin{aligned}
\mathcal{L}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) &= \sum_{a\in\delta^+(s)} x_a + \sum_{a\in A} z_a(u_a - x_a) + \sum_v y_v \left(\sum_{a\in\delta^+(v)} x_a - \sum_{a\in\delta^-(v)} x_a\right)\\
&\quad + y_s\left(\sum_{a\in\delta^+(s)} x_a - \sum_{a\in\delta^-(s)} x_a - q\right) + y_t\left(\sum_{a\in\delta^+(t)} x_a - \sum_{a\in\delta^-(t)} x_a + q\right)\\
&= \sum_{a\in A} z_a u_a + \sum_{a=(v,w)\in A} x_a(y_v - y_w - z_a) + q(1 + y_t - y_s)
\end{aligned}$$

which gives the dual

$$
\begin{aligned}
\min \quad & \sum_a u_a z_a && && (6.4a)\\
\text{s.t.} \quad & z_a \geq y_v - y_w && \forall a = (v,w) \in A && (6.4b)\\
& y_s - y_t \geq 1 && && (6.4c)\\
& z_a \geq 0 && \forall a \in A. && (6.4d)
\end{aligned}
$$

whose matrix is totally unimodular, and whose integer results are naturally interpreted as the minimum capacity cut problem. □

We conclude with two results that play a role in Chapter 11. Let $\mathbb{1}_P$ (resp. $\mathbb{1}_C$) denote the indicator function of a path P (resp. a cycle C). Proving

Proposition 6.13. *Any s-t flow f can be decomposed as*

$$\sum_{C \in \mathcal{C}} \mu_C \mathbb{1}_C + \sum_{P \in \mathcal{P}} \lambda_P \mathbb{1}_P$$

and its value is $\mathrm{val}(f) = \sum_{P \in \mathcal{P}} \lambda_P$, *and a circulation g as*

$$\sum_{C \in \mathcal{C}} \mu_C \mathbb{1}_C$$

where $\mathcal{C}$ denotes the set of cycles in G and $\mathcal{P}$ the set of s-t paths in P.

Corollary 6.14. *The rays of the s-t flow matrix correspond to cycles, the extreme points to s-t paths.*

Exercise 6.6. (not trivial but doable) Prove Proposition 6.13 and Corollary 6.14. △

6.4 Exercises

Exercise 6.7. *(Conception de réseau)* Une entreprise souhaite organiser sa distribution, pour acheminer annuellement de la marchandise depuis des entrepôts jusqu'à des points de vente. Pour chaque entrepôt e, elle connaît la quantité $o(e)$ qui y arrivera dans l'année, et pour chaque point de vente v, elle connaît la quantité $d(v)$ qui doit y parvenir au cours de l'année. On suppose $\sum_e o(e) = \sum_v d(v)$.

L'entreprise doit sélectionner les liens dans le réseau de transport préexistant (routes, voies ferrées, voies fluviales, etc.), vu comme un graphe orienté. Le transport d'une unité de marchandise sur l'année sur le lien (i,j) dans le réseau de transport coûte c_{ij}. Modéliser ce problème d'organisation de la distribution à coût minimal comme un problème de flot.

△

Solution. C'est directement un problème de b-flot de coût minimum, dans le réseau de transport. Les entrepôts e sont des sommets sources avec $b(e) = o(e)$, les points de vente sont des sommets puits avec $b(v) = -d(v)$, et les autres sommets u du réseau sont affecté d'un $b(u) = 0$. La somme des b sur les sommets est bien nulle (condition pour l'application des algorithmes de b-flot). La capacité $u(a)$ est $+\infty$ pour chaque arc a (on peut supposer le réseau de transport surcapacitaire par rapport aux besoins de l'entreprise) et les coûts unitaires sont ceux de l'énoncé. □

Exercise 6.8. *(Problème de transport de Monge)* Le premier problème de recherche opérationnelle à visée pratique a été étudié par Monge en 1781 sous le nom du problème des déblais et remblais. Considérons n tas de sable, devant servir à combler m trous. Notons s_i la masse du $i$ème tas de sable et t_j la masse de sable nécessaire pour combler le $j$ème trou. Pour chaque couple (i,j) on connaît la distance d_{ij} du $i$ème tas au $j$ème trou. Si une masse x_{ij} est déplacée du tas i au trou j, le coût du déplacement est égal à $d_{ij}x_{ij}$. On souhaite savoir quel est le plan de transport qui permet de boucher les trous au coût minimum. Montrer que ce problème se modélise comme un problème de b-flot de coût minimum.

△

C'est un problème de b-flot sur un graphe biparti complet dont les sommets sont :

- les tas i et un tas fictif i^*
- les trous j et un trou fictif j^*,

et dont tous les arcs vont des tas vers les trous. On pose $b(i) = s_i$ et $b(i^*) = \max(0, \sum_j t_j - \sum_i s_i)$. On pose de même $b_j = -t_j$ et $b(j^*) = \max(0, \sum_i s_i - \sum_j t_j)$. Le coût unitaire est d_{ij} sur l'arc (i,j) et 0 sur les arcs (i^*,j) et (i,j^*).

Exercise 6.9. *(Remplissage d'un bus)* Un bus capable d'embarquer au plus B passagers va partir de la ville 1 et va visiter successivement les villes $2, 3, \ldots, n$. Le nombre de passagers voulant voyager de la ville i à la ville j (avec $i < j$) est d_{ij} et le prix de ce trajet est p_{ij}. Combien de passagers faut-il prendre en chaque ville pour maximiser les recettes totales ? Modéliser ce problème comme un problème de flot. △

Solution. On définit le graphe orienté $D = (V, A)$ de la manière suivante. On construit d'abord un chemin élémentaire à n sommets, numérotés de 1 à n depuis la source jusqu'au puits (ces sommets représentent les villes, et les arcs les trajets successifs du bus). Ensuite, pour chaque couple d'entiers (i,j), avec $1 \leq i < j \leq n$, on introduit un sommet v_{ij} et on ajoute un arc (v_{ij}, i) et un arc (v_{ij}, j). Voilà pour le graphe.

On met une capacité B sur les arcs (i,j) du chemin, et une capacité $+\infty$ sur les autres arcs. On met un coût nul partout, excepté sur les arcs (v_{ij}, i) pour lesquels on met un coût $-p_{ij}$. Enfin, on définit $b(v_{ij}) = d_{ij}$ pour tout $i < j$ et $b(j) = -\sum_{i=1}^{j-1} d_{ij}$ pour tout j. Le problème de l'énoncé est précisément le problème de b-flot de coût minimum sur ce graphe. L'équivalence des deux problèmes se prouve formellement comme suit.

Etant donnée une solution au problème de l'énoncé, on construit une solution réalisable de même coût au problème de b-flot : sur un arc (v_{ij}, i) on met un flot égal au nombre de passagers montant en i et souhaitant se rendre en j; sur un arc (v_{ij}, j) on met un flot égal au nombre de passagers souhaitant faire le trajet $i \to j$, mais ne montant pas dans le bus; sur un arc $(i, i+1)$ on met le nombre de passagers dans le bus sur le trajet $i \to i+1$. On vérifie aisément que cela forme un b-flot de même coût.

Réciproquement, étant donnée une solution au problème de b-flot (que l'on peut supposer entière car les paramètres de capacité et les b le sont), on construit une solution réalisable de même coût au problème de départ en faisant monter en i, pour chaque $j > i$, un nombre de passagers égal à la valeur du flot sur l'arc (v_{ij}, i). On ne dépasse pas la capacité du bus car sur l'arc $(i, i+1)$ on a bien alors le nombre total de passagers montés en i ou avant, diminué du nombre de passagers descendus en i (d'après la loi de Kirchoff). C'est donc une solution réalisable au problème de départ, et le coût est trivialement le même. □

Exercise 6.10. *(Le problème des représentants)* Une ville a des clubs et des partis politiques. Chaque citoyen de cette ville appartient à au moins un club, et à au plus un parti. Le jour du renouvellement du conseil de la ville approche. Le nombre de citoyens du parti P_k présents au conseil ne doit pas excéder u_k. Or, chaque club doit nommer un de ces membres pour le représenter au conseil de la ville. Un citoyen siégeant au conseil ne peut représenter qu'un seul club. Proposer un algorithme polynomial permettant de décider si ces contraintes peuvent être satisfaites. △

Solution. Quitte à ajouter un parti des non-affiliés, avec $u_k = +\infty$, on peut suppose que tout citoyen est membre d'un unique parti politique. On note C l'ensemble des clubs, I l'ensemble des citoyens et P l'ensemble des partis. On définit alors un graphe orienté $D = (V, A)$ de la manière suivante. On pose $V = C \cup I \cup P \cup \{t\}$, où t est un sommet additionnel représentant un puits. On introduit ensuite un arc (c, i) pour tout citoyen i membre du club c (avec une capacité égale à 1) et un arc (i, p) pour tout citoyen i membre du parti p (avec une capacité égale à u_k). On met $b(c) = 1$ pour tout $c \in C$, $b(i) = 0$ pour tout $i \in I$, $b(p) = 0$ pour tout $p \in P$ et enfin $b(t) = -|C|$. Le problème de l'énoncé est alors exactement le problème d'existence d'un b-flot dans ce graphe. L'équivalence des deux problèmes se prouve formellement comme suit. (On utilise la propriété d'intégrité des flots). □

Exercise 6.11. *(Théorème de Menger)* Démontrer le théorème suivant, dit "théorème de Menger", à partir des outils du cours.

Soit $D = (V, A)$ une graphe orienté, et s et t deux sommets particuliers. Le nombre maximum de s-t chemins arc-disjoints est égale à la cardinalité minimale d'un sous-ensemble d'arcs intersectant tout s-t chemin.

△

Solution. Munissons chaque arc de D d'une capacité 1. Le théorème d'intégrité combiné avec le théorème de décomposition des flots en chemins assure que le nombre maximum de s-t chemins arc-disjoints est égal à la valeur d'un flot maximum. D'autre part, un sous-ensemble d'arcs intersectant tout s-t chemin est un ensemble déconnectant s de t, et il est prouvé dans le cours qu'un tel ensemble minimum est une s-t coupe de capacité minimum. Le théorème max-flot min-coupe permet alors de conclure. □

Exercise 6.12. *(Flotte de taxis)* Une compagnie de taxis a p courses à satisfaire sur une journée. Ces courses sont supposées connues à l'avance. Pour chacune de ces courses $i = 1, \ldots, p$, on connaît son lieu o_i et son heure de départ h_i, la durée prévisible de la course t_i et le lieu d'arrivée d_i. De plus, le temps τ_{ji} pour se rendre de d_j à o_i est connu pour tout (i, j). La compagnie souhaite minimiser le nombre de taxis nécessaires pour satisfaire toute la demande. Tous les taxis sont supposés situés en un unique dépôt en début de journée. Modéliser ce problème comme un problème de flot. △

Solution. (La solution n'est pas unique) On construit le graphe $D = (V, A)$ avec un sommet s pour le dépôt, un sommet pour chaque o_i et chaque d_i et un sommet t pour la fin de journée. Les arcs sont les couples (s, o_i) et (o_i, d_i) pour toutes les courses i. On ajoute tous les couples (d_j, o_i) s'il est possible à un taxi de faire la course i après la course j (concrètement, si $h_i \geq t_j + \tau_{ji}$). Enfin, on ajoute l'arc (s, t). Les capacités sont les suivantes :

- (s, o_i) : capacité supérieure $= +\infty$ et capacité inférieure $= 0$.
- (o_i, d_i) : capacité supérieure $= +\infty$ et capacité inférieure $= 1$.
- (d_j, o_i) : capacité supérieure $= +\infty$ et capacité inférieure 0.
- (s, t) : capacité supérieure $= +\infty$ et capacité inférieure 0.

Les coûts sont les suivants :

- (s, o_i) : 0.
- (o_i, d_i) : 0.
- (d_j, o_i) : 0
- (s, t) : 1.

Enfin, on définit $b(s) = p = -b(t)$ et $b(o_i) = b(d_i) = 0$ pour toute course i.

Le théorème d'intégrité combiné avec le théorème de décomposition des flots en chemins permet alors de vérifier que le problème de l'énoncé est équivalent au problème de b-flot de coût minimum dans ce graphe. Cela se prouve formellement comme suit. □

Exercise 6.13. *(Flotte de taxis bis)* Même exercice, à ceci près que pour chaque course (i, j), le gain net g_{ij} est connu (frais, salaire, carburant sont déduits). La compagnie souhaite maximiser son profit, sachant qu'elle dispose de K taxis. Modéliser ce problème comme un problème de flot. △

Exercise 6.14. *(Arrondir de manière consistante)* On se donne une matrice réelle $A = (a_{ij}) \in \mathbb{R}^{m \times n}$. On aimerait arrondir chaque entrée de la matrice et arrondir la somme des termes de chaque ligne et de chaque colonne de manière à ce que la somme des arrondis de chaque ligne soit égale à l'arrondi de la ligne et de même pour les colonnes. Lorsqu'on arrondit un nombre a, on a la liberté de choisir si c'est "vers le haut" ou "vers le bas", i.e. on peut remplacer comme on veut a par $\lceil a \rceil$ ou par $\lfloor a \rfloor$, et ce indépendamment de la façon dont on a arrondi les autres nombres. Montrer qu'il est toujours possible d'arrondir ainsi les entrées de A. △

Solution. On considère le graphe biparti complet dont un côté est formé des lignes et l'autre des colonnes. On oriente les arêtes des colonnes vers les lignes. On ajoute un sommet source s avec un arc (s, i) pour chaque ligne i. On ajoute un sommet puits t avec un arc (j, t) pour chaque colonne j. On ajoute enfin un arc (s, t).

On met sur chaque arc

- (i, j) : une capacité inférieure égale à $\lfloor a_{ij} \rfloor$ et une capacité supérieure égale à $\lceil a_{ij} \rceil$,
- (s, i) : une capacité inférieure égale à $\lfloor \sum_j a_{ij} \rfloor$ et une capacité supérieure égale à $\lceil \sum_j a_{ij} \rceil$,
- (j, t) : une capacité inférieure égale à $\lfloor \sum_i a_{ij} \rfloor$ et une capacité supérieure égale à $\lceil \sum_i a_{ij} \rceil$,
- (s, t) : une capacité inférieure égale à 0 et une capacité supérieure égale à 1.

Posons $b(s) = \sum_{i,j} a_{ij} = -b(t)$ et $b(i) = b(j) = 0$ pour toute ligne i et toute colonne j.

Remarquons maintenant que la fonction $f : A \to \mathbb{R}_+$ definie par $f(i, j) = a_{ij}$, $f(s, i) = \sum_j a_{ij}$, $f(j, t) = \sum_i a_{ij}$ et $f(s, t) = 0$ est un b-flot. Le théorème d'intégrité des flots assure alors qu'il existe un b-flot $g : A \to \mathbb{Z}$ sur le même graphe. On vérifie alors immédiatement que $g(i, j)$ est une matrice "arrondie" satisfaisant les conditions de l'énoncé (remarquons que l'on prouve plus : la somme totale des entrées de la matrice varie de moins d'une unité !). □

Exercise 6.15. *(Théorème max-flot min-coupe par la programmation linéaire)* Prouver le théorème max-flot, min-coupe en utilisant le théorème de la dualité forte en programmation linéaire. △

Exercise 6.16. *(Coupe minimum)* On se donne un graphe orienté $D = (V, A)$ avec une capacité $u : A \to \mathbb{R}_+$. Montrer que l'on peut trouver en temps polynomial l'ensemble X non vide et distinct de V tel que $\sum_{a \in \delta^+(X)} u(a)$ soit minimal. (Attention : il n'y a pas de s et de t spécifiés).

△

Exercise 6.17. *(Combat sur un réseau)* Un centre de commandement est situé en un sommet p d'un réseau non-orienté. On connaît la position des subordonnés modélisée par un sous-ensemble S des sommets du réseau. On souhaite détruire un nombre minimum de liens afin d'empêcher toute communication entre le centre de commandement et les subordonnés. Comment résoudre ce problème en temps polynomial ?

△

Exercise 6.18. *(Extraction de mine à ciel ouvert)* Un domaine d'application de la recherche opérationnelle est l'exploitation des mines à ciel ouvert. Un problème important consiste à déterminer les *blocs* à extraire. Dans toute la suite, la mine sera assimilée à une collection de n blocs numérotés de 1 à n. L'extraction du bloc i rapporte la quantité c_i; cette quantité peut être positive ou négative (en fonction de la quantité de minerai présent dans le bloc). Les blocs ne peuvent pas être extraits dans n'importe quel ordre: pour chaque bloc i, on connaît l'ensemble Y_i des blocs qu'il faut avoir extraits pour pouvoir extraire le bloc i. L'objectif est de proposer un sous-ensemble de blocs à extraire de manière à maximiser le profit total.

Dans un graphe orienté, un ensemble S de sommets est *fermé* si tout successeur d'un sommet de S est encore dans S. Considérons le problème suivant.

Le problème du fermé maximum.

Donnée. Un graphe $D = (V, A)$ orienté, des réels c_v pour tout $v \in V$.
Tâche. Trouver un sous-ensemble fermé $S \subseteq V$ tel que $\sum_{v \in S} c_v$ soit maximal.

1. Expliquer pourquoi le problème d'extraction précédent peut se modéliser sous la forme d'un problème de fermé maximum.

L'objectif va maintenant être de modéliser le problème du fermé maximum comme un problème de s-t coupe de capacité minimale. On construit un nouveau graphe $\widetilde{D} = (\widetilde{V}, \tilde{A})$ en ajoutant un sommet source s et un sommet puits t à V. On note $V^+ = \{v \in V : c_v \geq 0\}$ et $V^- = \{v \in V : c_v < 0\}$. L'ensemble $\tilde{A}$ est alors A auquel on ajoute les arcs $\{(s, v) : v \in V^+\}$ et les arcs $\{(v, t) : v \in V^-\}$. On met une capacité $u_{(s,v)} = c_v$ pour $v \in V^+$ et $u_{(v,t)} = -c_v$ pour $v \in V^-$.

2. Quelles capacités u_a mettre sur les arcs $a \in A$ de manière à ce que tout $X \subseteq V$ tel que $\delta^+_{\widetilde{D}}(X \cup \{s\})$ soit une s-t coupe de $\widetilde{D}$ de capacité minimale soit un ensemble fermé de D ?

3. Soit $X \subseteq V$ un ensemble fermé de D. Montrer que
$$\sum_{a \in \delta^+_{\widetilde{D}}(X \cup \{s\})} u_a = \sum_{v \in V^+} c_v - \sum_{v \in X} c_v.$$

4. En déduire que le problème du fermé maximum se modélise comme un problème de coupe minimale. Conclure sur la résolution pratique de problème.

△

Solution. 1. Considérons le graphe $D = (V, A)$ dont les sommets sont les blocs et pour lequel on a un arc (u, v) si $v \in Y_u$. Une solution du cas statique est un fermé de ce graphe et réciproquement tout fermé du graphe est une solution du cas statique.

2. Si on met $u_a := +\infty$ pour tout $a \in A$, on a la propriété recherchée. En effet, soit $X \subseteq V$ tel que $\delta^+_{\tilde{D}}(X \cup \{s\})$ soit une s-t coupe de $\tilde{D}$ de capacité minimale. Aucune arête de A ne peut être dans cette coupe, sinon sa capacité serait $= +\infty$. Donc, si $u \in X$, tout successeur de u est encore dans X. Ce qui est précisément la définition de “fermé”.

3. On note $\bar{X} := V \setminus X$. Comme X est fermé, les seuls arcs qui vont jouer un rôle dans la coupe sont les arcs de la forme (s, v) avec $v \in \bar{X}$ et ceux de la forme (v, t) avec $v \in X$. On a donc
$$\sum_{a \in \delta^+_{\tilde{D}}(X \cup \{s\})} u_a = \sum_{v \in \bar{X} \cap V^+} c_v - \sum_{v \in X \cap V^-} c_v = \sum_{v \in V^+} c_v - \sum_{v \in X} c_v.$$

4. Posons $C := \sum_{v \in V^+} c_v$.

   Soit X un fermé de D de valeur maximale η. D'après ce qui a été montré ci-dessus, on a alors une s-t coupe de $\tilde{D}$ de capacité $C - \eta$.

   Réciproquement, prenons une s-t coupe de $\tilde{D}$ de capacité minimale κ. D'après ce qui a été montré, l'ensemble X induit sur D est alors fermé et de valeur $C - \kappa$.

   On a donc: $C - \kappa = \eta$, et chercher une coupe minimale sur $\tilde{D}$ est équivalent à chercher un fermé maximum sur D. Comme chercher une coupe minimale se fait en temps polynomial, on sait résoudre le problème de manière efficace.

$\square$

Chapter 7

Matchings

Definition 7.1. *A graph $G = (V, E)$ is bipartite if and only if V can be partitioned into two subset (U, W) such that each edge in E is one extremity in U and the other in W.*

Proposition 7.2. *Let $G = (V, E)$ be a graph. The following statements are equivalent.*

1. *G is bipartite*
2. *G is 2-colorable*
3. *G has no odd cycle*

Exercise 7.1. Prove Proposition 7.2. Deduce a polynomial algorithm determining if a graph is bipartite. △

7.1 Maximum matching

We recall the MAXIMUM MATCHING PROBLEM.

MAXIMUM MATCHING
Input. A graph G, an integer k
Question. Is there a matching with k edges in G?

Let G be a graph and M a matching in G. An elementary path P in G is *M-augmenting* it has odd length, its ends and not covered by M, and its edges are alternatively inside and outside M. Figure 7.1 illustrates such a path. Let Δ denote the *symmetric difference*: $X \Delta Y = (X \cup Y) \backslash (X \cap Y)$..

Proposition 7.3. *A matching M in G is maximal if there is no M-augmenting path.*

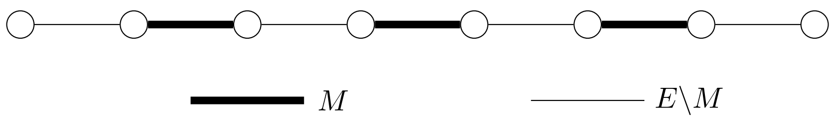


Figure 7.1: An M-augmenting path.

Proof. Let M be a matching. Suppose that there exists an M-augmenting path P with edges $E(P)$. Then $M'\Delta E(P)$ is a matching and $|M'| = |M+1|$.

Conversely, suppose that M' is a matching with $|M'| > M$. Then vertices in the graph $(V, M \cup M')$ have degree at most 2. Hence the connected components of this graph are paths and cycles. And as $|M'| > M$, there is a connected component with more edges in M' than in M. Such a component is an M-augmenting path. □

Hence, maximum matching can be found in graphs where M-augmenting paths can be found. This is the case of bipartite graphs, as we will see in the next section.

7.2 Maximum weight matching

We now consider the more general maximum weight matching, but restrict ourselves to bipartite graphs.

MAXIMUM WEIGHT MATCHING
Input. A bipartite graph G, a weight function $w : E \to \mathbb{R}$.
Output. A matching M of maximum weight $\sum_{e \in M} w(e)$.

This problem typically models cases where machines must be affected to tasks.

Let G be a bipartite graph with vertices sets U and W. Let $H = (V', A)$ be the digraph obtained by orienting all edges in E from U to W, adding two vertices s and t, arcs (s, v) for each v in U and (v, t) for each v in W. Finally, let capacity $u(a)$ be equal to 1 in a in an orientation of an edge in E, and to $+\infty$ otherwise.

Proposition 7.4. *There is a bijection between matchings in G and integer flows under u in H.*

Hence, a maximum weight matching in a bipartite graph can be found in polynomial time using a maximum flow algorithm.

Proposition 7.3 enables to propose a faster algorithm: the *Hungarian method*. Given a matching M, let $D_M = (V, A_M)$ be the directed graph obtained by orienting the edges in M from W to U, and the edges not in M from U to W. And define $\ell_M(e) = w(e)$ if $e \in M$ and $\ell_M(e) = -w(e)$ if $e \notin M$. Let U_M and W_M denote the subsets of vertices of U and W that are note covered by an edge in M. Algorithm 7 states the Hungarian algorithm.

Algorithm 7 Hungarian method
```
1: Input: a bipartite graph $G$ with partition $U, W$, a weight function $w : E \to \mathbb{R}$
2: Output: a maximum weight matching $M$.
3: $M \leftarrow \emptyset$
4: Find a $U_M$-$W_M$ path $P$ of minimum $\sum_{a \in P} \ell_M(a)$. Stop and return $M$ if no such path exist.
5: $M \leftarrow M \Delta E(P)$.
```

Remark that U_M-W_M paths correspond to M-augmenting paths. Hence, by Proposition 7.3, Algorithm 7 ends after at most $n/2$ iterations and returns a maximum cardinality matching. It remains to settle the question of how to compute a shortest U_M-W_M path.

Proposition 7.5. *M is a maximum weight matching among the matching of cardinality $|M|$.*

Corollary 7.6. *There is no cycle C in D_M with $\sum_{a \in C} \ell_M(a) < 0$.*

Proof. $M \Delta C$ would be a matching of cardinality $|M|$ and larger weight. $\square$

Given $N \subseteq E$, let $w(N) = \sum_{e \in M} w(e)$.

Proof of Proposition 7.5. Suppose that M is a maximum weight matching among the matching of cardinality $|M|$, let M' be the next value taken by M, and let N be an arbitrary matching such that $|N| > |M|$. Then $N \Delta M$ is U_M-W_M path and $w(N) = w(M) - \ell(N \Delta M) \leq w(M) - \ell(M' \Delta M) = w(M')$. The result follows by iteration. $\square$

Using an appropriate implementation, we obtain the following result.

Proposition 7.7. *The Hungarian algorithm solves the* MAXIMUM WEIGHT MATCHING *problem in bipartite graphs in $O(n(m + n \log(n)))$.*

Note that both flow and the Hungarian method enable to solve the MAXIMUM MATCHING problem.

7.3 b-matchings

Given a graph $G = (V, E)$ and $b : E \to \mathbb{Z}_+$, a b-matching is a subset M of E such that $\deg_M(v) \leq b(v)$ for all v in V. The flow approach presented in the previous section naturally extends to the following problem.

MAXIMUM WEIGHT MATCHING
Input. A bipartite graph $G = (V, Z)$, $b : E \to \mathbb{Z}_+$, and $w : E \to \mathbb{R}$.
Output. A b-matching M of maximum weight $\displaystyle\sum_{e \in M} w(e)$.

Lower bounds on the degree can also be taken into account in the flow approach.

7.4 Maximum and maximum weight matchings in general graphs 😎

7.5 Stable matchings

Consider a bipartite graph $G = (V, E)$ with V partitioned into U, W. Suppose for each vertex v, we have a total ordering $\preceq_v$ on $\delta(v)$. A matching M is *stable* if, for each (u, v) not in M, at least one of the following conditions is satisfied

- there is e in $\delta(u) \cap M$ such that $e \prec_u (u, v)$,
- there is e in $\delta(v) \cap M$ such that $e \prec_v (v, u)$,

The traditional interpretation is the following one. Given a set U of men and a set W of women, edge (u, w) belongs to E if both u and w could potentially accept to marry the other one. Each woman w (resp. man u) has an order of preference $\preceq_w$ (resp. $\preceq_u$) on their potential partner. A matching M is stable if there are no pair (u, w) such that both u and w would prefer to be with the other one than with its partner in M.

Theorem 7.8. *There exists a stable matching in a bipartite graph, and it can be computed in $O(|U||W|$*

The proof and algorithm,, which we call the *traditional matching algorithm*, can be told as a story.

Proof. Every morning, each man invites to dinner a woman who has never refused on of his invitations – provided that such a woman exists. A woman accepts the invitation from the man she prefers among the men who invited her – provided that she would potentially accept marrying him. Whenever a woman refuses an invitation from a man, he never invites her again. The algorithm continues until all invitations remains identical two successive evenings.

Once a woman has dined with a man, she is guaranteed to dine the next evening with a man she likes at least as much.

Indeed, she will dine with the same man the next evening unless she is invited by a man she prefers. As a consequence, and *as at least one dinner changes every evening*, the algorithms terminates after at most the number of women times the number of men. Similarly

Once a man has dine with a woman, he will dine again only with her or with women that he likes less. We obtain as a consequence that the matching built is stable. Taking a man u, every woman v that u prefers to the woman he is matched with is matched with a man she prefers to u. □

There exists many stable matchings, and the algorithm used in the proof compute a very specific one. We define the *optimal partner* (resp.*pessimal partner*) of an individual the one he/she prefers (resp. likes the less) in all the one he/she is matched to in stable matching.

Proposition 7.9. *Two individuals cannot have the same optimal partner / pessimal partner.*

Proof. Suppose u and u' have the same optimal partner w, and w.l.o.g. suppose that w prefers u' to u. Let M be a stable matching where u and w are matched. By definition of the optimal partner, u' is matched to a partner he likes less that w, which contradicts the fact that M is stable. □

A stable matching is male optimal if every man is matched to his optimal partner and every woman to its pessimal partner.

Lemma 7.10. *Any male optimal matching is female pessimal.*

Proof. Let M be a male optimal stable matching, and M' a stable matching where a woman w is matched to a man u' she likes less that her partner u in M. Then w prefers u to u' in M', and as w is the optimal partner of u, he prefers her to his partner in M', which contradicts the fact that M' is stable. □

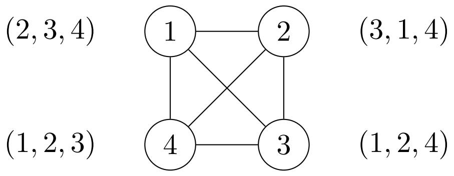
Proposition 7.11. *In the matching computed by the traditional matching algorithm is male optimal and female pessimal.*

Proof. It suffices to prove that it is male optimal. Consider for a contradiction that u is the first man rejected by his optimal partner w, because she prefers u'. By definition of u, man u' has still not been rejected by his optimal woman – each man considers women by decreasing order of preference. By definition of the optimal partner, there exists a matching M where u and w are matched. But in M, man u' is matched at best to his optimal partner, and hence to a woman he likes less than w. Hence, both u' and w would prefer being matched together, which leads to a contradiction. □

Consider now the case of bipartite complete graphs, and suppose that preferences $\preceq_v$ are drawn randomly. Then Pittel [1989] show that the average rank of an optimal partner is asymptotically in $\ln(n)$ while the average rank of a pessimal partner is asymptotically in $\frac{n}{\ln(n)}$. To sum things up, on a matching market, daring is a good strategy.

The *roommate problem* generalizes the stable marriage problem to general graphs. It enables to model matching problems where there is a single population. The objective is find a stable matching in a general graph. Such a matching is not guarantee to exist anymore. If preferences are dranw randomly, Mertens [2005] conjectures that the probability of existence of a stable matching decays algebraically in graphs with connectivity $\Theta(n)$ and algebraically in grids.

Exercise 7.2. Prove that there is no stable matching in the following graph, where list of preferences are given: 1 prefers 2 to 3 and 3 to 4.



△

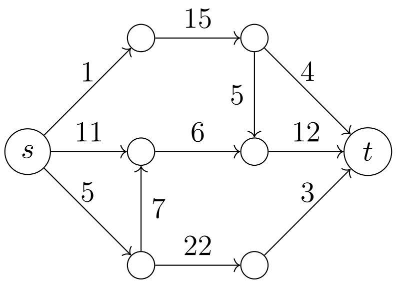
7.6 Further reading

Schrijver [2003, parts II and III] are devoted to matchings and their generalizations.

Roth [2015] vulgarizes the applications of stable matchings to non-financial markets.

7.7 Exercise

Exercise 7.3. What is the value of a maximum s-t flow in the following graphs.

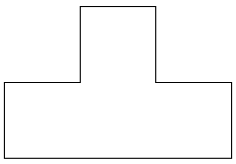


△

Vous dirigez une petite entreprise de soutien scolaire (cours particuliers). Vous proposez différents créneaux horaires. Chaque élève doit avoir au plus un créneau. On demande à chaque élève d'indiquer quels sont les horaires qui lui conviennent. On ne peut pas forcément satisfaire toutes les préférences. On veut maximiser le nombre d'élèves satisfaits. S'il y a k professeurs dans l'entreprise. Modélisez le problème avec les outils du cours.

Solution. □

Exercise 7.4. On souhaite carreler une pièce dont tous les murs sont à angle droits avec des carreaux rectangulaire de taille 10cm × 20cm. On souhaite savoir s'il est possible de carreler la pièce sans découper de carreaux. C'est par exemple impossible pour la pièce (minuscule) suivante



où tous les murs sauf celui du bas font 10cm. Montrer que le problème peut se modéliser comme un problème de couplage dans un graphe biparti. △

Solution. Un sommet par position 10cm × 10cm possible. Une arete entre deux sommets si ils sont voisins. On veut savoir s'il existe un couplage parfait (couvrant tous les sommets) □

Exercise 7.5. *Matching and vertex cover* A vertex cover of a graph $G = (V, E)$ is a set of nodes $U \subseteq V$ such that every edge in E contains at least an element of U .

1. Show that, for every matching M and every vertex cover U , $|M| \leq |U|$.
2. Using linear programming duality and total unimodularity, show the following statement: If G is bipartite, then $\max \{|M| : M \text{ is a matching}\} = \min \{|U| \colon U \text{ is a vertex cover}\}$
3. Give another proof of the statement in 2) using the max-flow/min-cut theorem.
4. Does the statement in 2) hold if G is not bipartite?

△

Part II

Mixed Integer Linear Programming

Chapter 8

Lagrangian duality

This chapter recalls notions seen last year. Consider the optimization problem

$$\min_{\boldsymbol{x}\in X} \ f(\boldsymbol{x}) \tag{8.1a}$$

$$\text{s.t.} \ g_i(\boldsymbol{x}) = 0, \quad \forall i \in [p] \tag{8.1b}$$

$$h_j(\boldsymbol{x}) \leq 0, \ \forall j \in [q] \tag{8.1c}$$

where f, g_i, and h_i are differentiable functions from $\mathbb{R}^n$ to $\mathbb{R} \cup \{+\infty\}$, and $\boldsymbol{X}$ is a subset of $\mathbb{R}^n$.

Remark that we have made no assumptions on the subset X of $\mathbb{R}^n$. In the context of linear programming duality, X is going to be the set of feasible solutions of a mixed integer linear program.

8.1 Lagrangian duality

8.1.1 Lagrangian and weak duality

The *Lagrangian* associated to problem (8.1) is the mapping

$$\begin{array}{rcl} \mathcal{L} : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^q & \to & \mathbb{R} \\ (\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) & \mapsto & f(\boldsymbol{x}) + \sum_{i=1}^{p} \lambda_i g_i(\boldsymbol{x}) + \sum_{j=1}^{q} \mu_j h_j(\boldsymbol{x}) \end{array} \tag{8.2}$$

where $\boldsymbol{\lambda}$ and $\boldsymbol{\mu}$ are the vectors of *Lagrangian multipliers*. Let $\mathcal{X}$ be the set of feasible solutions of (8.1). Remark that

$$\sup_{\boldsymbol{\lambda}\in\mathbb{R}^p, \boldsymbol{\mu}\in\mathbb{R}^q_+} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \begin{cases} f(\boldsymbol{x}) & \text{if } \boldsymbol{x} \in \mathcal{X} \\ +\infty & \text{otherwise.} \end{cases} \tag{8.3}$$

Hence, Problem (8.1) can be reformulated as the following *primal* problem.

$$\inf_{\boldsymbol{x}\in\mathbb{R}^n} \sup_{\boldsymbol{\lambda}\in\mathbb{R}^p, \boldsymbol{\mu}\in\mathbb{R}^q_+} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}). \tag{P}$$

The *dual problem* associated to (P) is

$$\sup_{\boldsymbol{\lambda}\in\mathbb{R}^p, \boldsymbol{\mu}\in\mathbb{R}^q_+} \inf_{\boldsymbol{x}\in X} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}). \tag{D}$$

Theorem 8.1. (weak duality)

$$\text{val}\,(\text{D}) \leq \text{val}\,(\text{P}). \tag{8.4}$$

The quantity val (P) − val (D) is called the *duality gap*. We say that we have *strong duality* is the duality gap is equal to 0.

Skill 8.1. *Computing the dual*

The direction of the inequalities can be arbitrarily chosen. The sign of the dual variable μ_j must only be chosen in such a way that (8.3) remains true. If possible possible, factorize $\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$ in $\boldsymbol{x}$ and turn the optimization problem $\inf_{\boldsymbol{x}\in X} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$ into a set of constraints ensuring that $\inf_{\boldsymbol{x}\in X} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) > -\infty$.

We are going to Lagrangian duality mainly in two contexts:

- linear programming in Chapter 9.
- Lagrangian relaxation in Chapter 11.

8.1.2 Saddle points

A point $(\boldsymbol{x}', \boldsymbol{\lambda}', \boldsymbol{\mu}')$ is a *saddle-point* of the Lagrangian if

$$\mathcal{L}(\boldsymbol{x}', \boldsymbol{\lambda}', \boldsymbol{\mu}') = \inf_{\boldsymbol{x}\in X} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}', \boldsymbol{\mu}') = \sup_{\boldsymbol{\lambda},\boldsymbol{\mu}} \mathcal{L}(\boldsymbol{x}', \boldsymbol{\lambda}, \boldsymbol{\mu}).$$

Remark that if $X = \mathbb{R}^n$ and $(\boldsymbol{x}', \boldsymbol{\lambda}', \boldsymbol{\mu}')$ is a saddle point of the Lagrangian, then

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} = \boldsymbol{0}, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{\lambda}} = \boldsymbol{0}, \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}} = \boldsymbol{0} \quad \text{in} \quad (\boldsymbol{x}', \boldsymbol{\lambda}', \boldsymbol{\mu}'). \tag{8.5}$$

Proposition 8.2. *There exists an optimal solution $\boldsymbol{x}^*$, the dual admits an optimal solution $(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$, and we have strong duality, then $(\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ is a saddle-point of the Lagrangian.*

Conversely, if there exists a saddle-point $(\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ of the Lagrangian, then $\boldsymbol{x}^*$ is an optimal solution of the primal, $\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ an optimal solution of the dual, and we have strong duality.

Exercise 8.1. Prove Proposition 8.2. △

Solution. We have

$$\begin{aligned}
\mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) &\geq \inf_{\boldsymbol{x}\in X} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) \\
&= \sup_{\boldsymbol{\lambda},\boldsymbol{\mu}} \inf_{\boldsymbol{x}\in X} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \inf_{\boldsymbol{x}\in X} \sup_{\boldsymbol{\lambda},\boldsymbol{\mu}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \\
&= \sup_{\boldsymbol{\lambda},\boldsymbol{\mu}} \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\lambda}, \boldsymbol{\mu}) \qquad \geq \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)
\end{aligned}$$

where the first equality comes from the definition of $(\boldsymbol{\lambda}*, \boldsymbol{\mu}^*)$, the second from strong duality, and the third from the definition of $\boldsymbol{x}^*$. The result immediately follows.

Conversely, suppose that $(\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ is a saddle point. Then

$$\begin{aligned}
\inf_{\boldsymbol{x}\in X} \sup_{\boldsymbol{\lambda},\boldsymbol{\mu}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) &\leq \sup_{\boldsymbol{\lambda},\boldsymbol{\mu}} \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\lambda}, \boldsymbol{\mu}) \qquad = \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) \\
&= \inf_{\boldsymbol{x}\in X} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) \qquad \leq \sup_{\boldsymbol{\lambda},\boldsymbol{\mu}} \inf_{\boldsymbol{x}\in X} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \\
&\leq \inf_{\boldsymbol{x}\in X} \sup_{\boldsymbol{\lambda},\boldsymbol{\mu}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}).
\end{aligned}$$

where the first inequality is immediate, the two equalities comes from the definition of a saddle point, the second inequality is immediate, and the last inequality is weak duality. Hence, all inequalities are equalities, which gives the result. □

8.1.3 Economic interpretation of duality

Consider now the following perturbed version of Problem (8.1) where we highlight the dependence of the value of the minimization problem with respect to the right hand side $\boldsymbol{b}^t = (\boldsymbol{b}_g^t, \boldsymbol{b}_h^t)$.

$$
\begin{aligned}
v(\boldsymbol{b}) := \min_{\boldsymbol{x}\in X} \ & f(\boldsymbol{x}) && (8.6a)\\
\text{s.t. } & g_i(\boldsymbol{x}) = b_{g,i}, \quad \forall i \in [p] && (8.6b)\\
& h_j(\boldsymbol{x}) \leq b_{h,j}, \ \forall j \in [q] && (8.6c)
\end{aligned}
$$

Typically, each constraints $h_j(\boldsymbol{x}) \leq b_{h,j}$ model the fact that the solution must not consume more than a given amount $b_{h,j}$ of some resource. In this section, we show that the reduced cost can be interpreted as the marginal effect on the price of an increase in the resource. Consider the Lagrangian where $\boldsymbol{b}$ has been introduced as a variable

$$\overline{\mathcal{L}}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}, \boldsymbol{b}) := f(x) + \sum_{i=1}^{p} \lambda_i\big(g_i(\boldsymbol{x}) - b_{g,i}\big) + \sum_{j=1}^{q} \mu_j\big(h_j(\boldsymbol{x}) - b_{h,j}\big)$$

Theorem 8.3. *Suppose that $X = \mathbb{R}^n$, that $(\boldsymbol{x}^*(\boldsymbol{b}'), \boldsymbol{\lambda}^*(\boldsymbol{b}'), \boldsymbol{\mu}^*(\boldsymbol{b}'), \boldsymbol{b}')$ is the unique saddle point of the Lagrangian $\overline{\mathcal{L}}(\cdot,\cdot,\cdot,\boldsymbol{b}')$ and that $\boldsymbol{b}' \mapsto \boldsymbol{x}^*(\boldsymbol{b}')$, $\boldsymbol{b}' \mapsto \boldsymbol{\lambda}^*(\boldsymbol{b}')$, and $\boldsymbol{b}' \mapsto \boldsymbol{\mu}^*(\boldsymbol{b}')$ are differentiable for $\boldsymbol{b}'$ in a neighborhood of $\boldsymbol{b}$. Then v is differentiable in $\boldsymbol{b}$ and*

$$\frac{\partial v}{\partial b} = -\begin{pmatrix} \boldsymbol{\lambda} \\ \boldsymbol{\mu} \end{pmatrix}.$$

The hypotheses of the theorem are very restrictive. We need them to ensure that v is differentiable. Theorem 8.3 admits a natural generalization without these restrictive hypotheses. However, this generalization requires the notion of sub-differential to deal with the case there the solutions and the value function are neither unique nor differentiable. We now give a simple proof under the restrictive assumption.

Proof. We have $v(\boldsymbol{b}) = \overline{\mathcal{L}}(\boldsymbol{x}^*(\boldsymbol{b}), \boldsymbol{\lambda}^*(\boldsymbol{b}), \boldsymbol{\mu}^*(\boldsymbol{b}), \boldsymbol{b})$. Hence

$$\frac{\partial v}{\partial b} = \frac{\partial \overline{\mathcal{L}}}{\partial \boldsymbol{x}}\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{b}} + \frac{\partial \overline{\mathcal{L}}}{\partial \boldsymbol{\lambda}}\frac{\partial \boldsymbol{\lambda}}{\partial \boldsymbol{b}} + \frac{\partial \overline{\mathcal{L}}}{\partial \boldsymbol{\mu}}\frac{\partial \boldsymbol{\mu}}{\partial \boldsymbol{b}} + \frac{\partial \overline{\mathcal{L}}}{\partial \boldsymbol{b}}\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{b}}$$

where everything is evaluated in $(\boldsymbol{x}^*(\boldsymbol{b}), \boldsymbol{\lambda}^*(\boldsymbol{b}), \boldsymbol{\mu}^*(\boldsymbol{b}), \boldsymbol{b})$. Remark that since $\mathcal{L}(\cdot,\cdot,\cdot) = \overline{\mathcal{L}}(\cdot,\cdot,\cdot,b)$, we have

$$\left.\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}}\right|_{(\cdot,\cdot,\cdot)} = \left.\frac{\partial \overline{\mathcal{L}}}{\partial \boldsymbol{x}}\right|_{(\cdot,\cdot,\cdot,b)}, \quad \left.\frac{\partial \mathcal{L}}{\partial \boldsymbol{\lambda}}\right|_{(\cdot,\cdot,\cdot)} = \left.\frac{\partial \overline{\mathcal{L}}}{\partial \boldsymbol{\lambda}}\right|_{(\cdot,\cdot,\cdot,b)}, \quad \text{and} \quad \left.\frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}}\right|_{(\cdot,\cdot,\cdot)} = \left.\frac{\partial \overline{\mathcal{L}}}{\partial \boldsymbol{\mu}}\right|_{(\cdot,\cdot,\cdot,b)}.$$

The results then follows from (8.5) and $\dfrac{\partial \overline{\mathcal{L}}}{\partial \boldsymbol{b}} = -\begin{pmatrix} \boldsymbol{\lambda} \\ \boldsymbol{\mu} \end{pmatrix}$. □

Remark 8.1. In some books, Theorem 8.3 is written as $\dfrac{\partial v}{\partial b} = \begin{pmatrix} \boldsymbol{\lambda} \\ \boldsymbol{\mu} \end{pmatrix}$. This comes from the fact that the Lagrangian can alternatively be defined by

$$\overline{\mathcal{L}}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}, \boldsymbol{b}) := f(x) + \sum_{i=1}^{p} \lambda_i\big(b_{g,i} - g_i(\boldsymbol{x})\big) + \sum_{j=1}^{q} \mu_j\big(b_{h,j} - h_j(\boldsymbol{x})\big),$$

leading to duals of opposite sign. △

Exercise 8.2. *Sensitivity analysis* Theorem 8.3 gives in fact a general method to prove to analysis the sensibility of an optimal solution to a parameter $\boldsymbol{a}$. Consider the generic parametrized problem

$$v(\boldsymbol{a}) := \min_{\boldsymbol{x}\in X} \; f(\boldsymbol{x},\boldsymbol{a}) \tag{8.7a}$$
$$\text{s.t. } g_i(\boldsymbol{x},\boldsymbol{a}) = 0, \quad \forall i \in [p] \tag{8.7b}$$
$$h_j(\boldsymbol{x},\boldsymbol{a}) \leq 0, \quad \forall j \in [q] \tag{8.7c}$$

Consider the Lagrangian of 8.7 where $\boldsymbol{a}$ has been introduce as a variable.

$$\overline{\mathcal{L}}(\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{\mu},\boldsymbol{a}) = f(\boldsymbol{x},\boldsymbol{a}) + \sum_i \lambda_i g_i(\boldsymbol{x},\boldsymbol{a}) + \sum_j \mu_j h_j(\boldsymbol{x},\boldsymbol{a})$$

1. Adapt the proof of Theorem 8.3 to show that, under similar hypothesis that will be provided,

△

8.2 KKT conditions

8.3 Convex duality and Lagrangian duality 😎

This section is extra-curricular. A simpler treatment of the economic interpretation of Lagrange multipliers in the context of linear programming is available in Section 9.3.

Follows from Theorem 11.39.2 if Rockafellar and Wets, using perturbation $Ax = y$ of the constraint $Ax = 0$.

- Ajouter une explication simple de l'interpretation economique de la dualite (sans passer par Fenchel)

8.4 Further readings

The general case of Theorem 8.3 is detailed by Gilbert [2002, Section 4.5]. [Rockafellar and Wets, 2009, Chapter 11] details link between Lagrangian and convex duality, and gives an in-depth view of perturbation theory, which is a nice framework to give interpretations to duals.

Chapter 9

Linear Programming

A *linear program* is an optimization program of the form

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^n} \quad & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.t.} \quad & A\boldsymbol{x} \leq \boldsymbol{b} \end{aligned} \tag{9.1}$$

where $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$, and $A \in \mathbb{R}^{m\times n}$. The way a linear program is written in (9.1) is called the *general form* or *inequational form*. Alternative forms include the *canonical form*

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^n} \quad & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.t.} \quad & A\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \geq 0, \end{aligned} \tag{9.2}$$

and the *standard form* or *equational form*

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^n} \quad & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.t.} \quad & A\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq 0, \end{aligned} \tag{9.3}$$

These three forms are equivalent, in the sense that a linear program written in one of these forms can be written in any of the other ones.

9.1 Simplex algorithm

Consider a linear programing in standard form 9.3. W.l.o.g. suppose that A is of *full rank*, i.e., that its lines are linearly independent. Recall that $A \in \mathbb{R}^{m\times n}$.

A *base* B is a subset of $[n]$ of m column indices such that the corresponding columns are linearly independent, i.e., such that the square matrix $A_B = (a_{\cdot j})_{j\in B}$ is invertible. The basic solution of 9.3 associated to a base B is

$$\boldsymbol{x}_b = A_B^{-1}\boldsymbol{b} \tag{9.4}$$

A solution is *basic* if it is the basic solution of a base B. It is *basic feasible* if $A_B^{-1}\boldsymbol{b} \geq 0$.

Theorem 9.1. *If the linear program 9.3 admits an optimal solution, then it admits a basic feasible optimal solution.*

Let B be a feasible base, $N = [n]\backslash B$, A_B and A_N be the corresponding submatrices, $\boldsymbol{c}_B$ and $\boldsymbol{c}_N$ the corresponding vectors of variables, and $\boldsymbol{x}_B$ and $\boldsymbol{x}_N$ the corresponding vectors of variables. By multiplying the constraints by A_B^{-1} and substituting in the objective, we obtain the following equivalent of (9.3).

$$\begin{array}{cl} \min\limits_{\boldsymbol{x}\in\mathbb{R}^n} & \boldsymbol{c}_B A_B^{-1}\boldsymbol{b} + (\boldsymbol{c}_N^T - \boldsymbol{c}_B^T A_B^{-1} A_N)\boldsymbol{x}_N \\ \text{s.t.} & \boldsymbol{x}_B = A_B^-1\boldsymbol{b} - A_B^{-1}A_N\boldsymbol{x}_N \\ & \boldsymbol{x} \geq 0, \end{array} \tag{9.5}$$

The vector $\boldsymbol{r}_N = \boldsymbol{c}_N^T - \boldsymbol{c}_B^T A_B^{-1} A_N$ is the vector of *reduced costs* associated to B. The optimality criterion in the following proposition follows immediately from (9.5).

Proposition 9.2. *Let B be a feasible basis. If $\boldsymbol{r}_N \geq 0$, then B is an optimal basis. Otherwise, let k be such that $r_k < 0$. Then one of then exactly one of the following statement is true.*

1. *There exists a feasible basis $B' \subseteq B \cup \{k\}$ with objective at least as small as the one of B.*
2. *There is no feasible basis $B' \subseteq B \cup \{k\}$ different from B, and the value of* (9.3) *is $-\infty$.*

A *pivot rule* is a rule that unambiguously chooses a base in B' in case 1. The *simplex algorithm* solves the linear program as follows. Starting from a feasible basis B, it uses the pivot rule to update B until $\boldsymbol{r}_N \geq 0$ or we are in case 2.

As in case 1, base B' is at least as good as B, and not strictly better, depending on the update rule chosen, the simplex algorithm might return to a base already visited an cycle. The following theorem is not easy to prove, and its proof is out of the scope of this lecture.

Theorem 9.3. *There exists a pivot rule such that the simplex algorithm does not cycle.*

As there is a finite number of basis, Theorem 9.3 guarantees that the simplex algorithm converges after a finite number of iteration. Hence, if (9.3) admits an optimal solution, then it admits an optimal basis $\boldsymbol{B}$ satisfying $\boldsymbol{r}_N \geq 0$.

9.2 Interior point and ellipsoid algorithms

Just put the table with complexity, and add the separation theorem.

9.3 Duality

We recall the linear program in equational form

$$\begin{array}{cl} \min\limits_{x\in\mathbb{R}^n} & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.t.} & A\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq 0, \end{array} \tag{P}$$

Its *dual* is the linear program

$$\begin{array}{cl} \max\limits_{y\in\mathbb{R}^m} & \boldsymbol{b}^T\boldsymbol{y} \\ \text{s.t.} & A^T\boldsymbol{y} \leq \boldsymbol{c} \end{array} \tag{D}$$

Proposition 9.4. (D) *is the Lagrangian dual of* (P), *and* (P) *is the Lagrangian dual of* (D).

Exercise 9.1. Prove Proposition (9.4). △

Theorem 9.5. (Strong duality theorem for linear programming) *One and only one of the following statements is true for* (P) *and* (D)

1. *Neither* (P) *nor* (D) *have a feasible solution.*
2. (P) *is unbounded and* (D) *has no feasible solution.*
3. (D) *is unbounded and* (P) *has no feasible solution.*
4. *Both* (P) *and* (D) *have a feasible solution and*

$$\mathrm{val}\,(\mathrm{P}) = \mathrm{val}\,(\mathrm{D})$$

The standard proof of Theorem (9.5) relies on Farkas Lemma. We give an alternative proof based on simplex algorithm theory.

Proof of case 4. Theorem 8.1 ensures $\mathrm{val}\,(\mathrm{P}) \geq \mathrm{val}\,(\mathrm{D})$. We prove the equality by exhibiting solutions of the two problems with the same value. Let B be an optimal basis of the primal returned by the simplex algorithm, and define $\boldsymbol{x}_B = A_B^{-1}\boldsymbol{b}$ and $\boldsymbol{y}_B = (A_B^{-1})^T\boldsymbol{c}_B$.

We have

$$A^T\boldsymbol{y}_B = \begin{pmatrix} A_B^T \\ A_N^T \end{pmatrix} (A_B^{-1})^T\boldsymbol{c}_B = \begin{pmatrix} \boldsymbol{c}_B \\ A_N^T(A_B^{-1})^T\boldsymbol{c}_B \end{pmatrix} \leq \begin{pmatrix} \boldsymbol{c}_B \\ \boldsymbol{c}_N \end{pmatrix}$$

The last inequality resulting from the fact that, as B is the optimal basis produced by the simplex, reduced costs are positive: $\boldsymbol{r}_N = \boldsymbol{c}_N^T - \boldsymbol{c}_B^T A_B^{-1} A_N \geq 0$. Hence $\boldsymbol{y}_B$ is a feasible solution of (D), and

$$\mathrm{val}\,(\mathrm{P}) \leq \boldsymbol{c}^T\boldsymbol{x}_B = \boldsymbol{c}_B^T A_B^{-1}\boldsymbol{b} = \boldsymbol{b}\boldsymbol{y}_B \leq \mathrm{val}\,(\mathrm{D}), \tag{9.6}$$

which concludes the proof. □

The following corollary enables to prove the optimality of a pair of primal dual solutions.

Proposition 9.6. *Let x and y be respectively feasible solutions of the primal* (P) *and the dual* (D). *Then x and y are both optimal solution if and only if*

$$(\boldsymbol{c} - \boldsymbol{y}A)\boldsymbol{x} = 0 \quad \textit{and} \quad \boldsymbol{y}(\boldsymbol{b} - A\boldsymbol{x}) = 0. \tag{9.7}$$

Equation (9.7) *is known as the* complementarity slackness *condition.*

Proof. Remark that

$$\boldsymbol{y}(\boldsymbol{b} - A\boldsymbol{x}) \leq 0 \leq (\boldsymbol{c} - \boldsymbol{y}A)\boldsymbol{x}.$$

which gives

$$\boldsymbol{y}\boldsymbol{b} \leq \boldsymbol{y}A\boldsymbol{x} \leq \boldsymbol{c}\boldsymbol{x}$$

where the inequalities are equalities if and only if the complementarity slackness conditions are satisfied. The result then follows from strong duality. □

Remark 9.1. Due to strong duality, the *economic interpretation* of duals applies in the case of linear programming. Indeed, under the assumption that B remains an optimal basis in a neighborhood of $\boldsymbol{b}$, the hypotheses of Theorem 8.3 are satisfied.

Furthermore, a simpler proof is available in the case of linear programming. Let $v(\boldsymbol{b}, \boldsymbol{c})$ denotes the value of (P). Supposing that B remains an optimal basis on a neighborhood around $(\boldsymbol{b}, \boldsymbol{c})$, function v is differentiable and we have

$$\frac{\partial v}{\partial \boldsymbol{c}} = \boldsymbol{x}_B = \quad \text{and} \quad \frac{\partial v}{\partial \boldsymbol{b}} = \boldsymbol{y}_B,$$

where the different sign conventions are explained in Remark 8.1. Using the notations in the proof of Theorem 9.5, it appears clearly from Equation (9.6) that

$$v(\boldsymbol{b},\boldsymbol{c}) = \boldsymbol{c}_B^T A_B^{-1}\boldsymbol{b}.$$

As we have supposed that B remains an optimal basis on a neighborhood around $(\boldsymbol{b},\boldsymbol{c})$, we obtain that

$$v(\boldsymbol{b},\boldsymbol{c}) = \boldsymbol{c}_B^T A_B^{-1}\boldsymbol{b},$$

and the results follows.

Again, a more general result that does not require that B remains an optimal basis on a neighborhood can be proved but required the notion of sub-differential. $\triangle$

9.4 Total unimodularity

A polyhedron P is *integral* if $P = \operatorname{conv}(P \cap \mathbb{Z}^n)$, where $\operatorname{conv}(\,\cdot\,)$ denotes the convex hull.

Proposition 9.7. *Let P be a polyhedron. The following statements are equivalent.*

1. *P is integral.*
2. *$\max\{\boldsymbol{c}^T\boldsymbol{x} \colon \boldsymbol{x} \in P\}$ is attained by a vector $\boldsymbol{x}^* \in \mathbb{R}^n$ for any $\boldsymbol{c}$ such that the linear program is finite.*
3. *$\max\{\boldsymbol{c}^T\boldsymbol{x} \colon \boldsymbol{x} \in P\}$ is integer for each $\boldsymbol{c}$ in $\mathbb{Z}^n$ such that the maximum is finite.*

A matrix A in $\mathbb{Z}^{m\times n}$ is *totally unimodular* the determinant of any of its square submatrix is in $\{-1,0,1\}$.

Theorem 9.8. *(Hoffman and Kruskal, 1956) An integral matrix is totally unimodular if and only if the polyhedron $\{\boldsymbol{x} \colon A\boldsymbol{x} \leq \boldsymbol{b}\boldsymbol{x} \geq 0\}$ is integral for each integral vector $\boldsymbol{b}$.*

Exercise 9.2. Prove the following corollary. $\triangle$

Corollary 9.9. *An integral matrix is totally unimodular if and only if, for all integral vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ such that we are in Case 4 of Theorem 9.5, then both the primal and the dual optimal are attained by integral vectors.*

The following sufficient condition for total unimodularity is useful in practice.

Proposition 9.10. *Let A be a matrix with coefficients in $\{-1,0,1\}$. If A contains at most one 1 and one -1 per column, then A is totally unimodular.*

Corollary 9.11. *The incidence matrix of a directed graph is totally unimodular.*

9.5 Line and column generation

9.5.1 Line generation

Line generation is an algorithm that enables to deal with linear program with a tractable number of variables (= columns of the constraint matrix) and an intractable number of constraints (=

lines of the constraint matrix). We index columns by i in $\mathcal{I}$ and lines by ℓ in $\mathcal{L}$. Consider the linear program

$$\min \sum_{i \in \mathcal{I}} c_i x_i \tag{9.8a}$$

$$\text{s.t.} \sum_{i} a_{\ell i} x_i \leq b_\ell, \quad \forall \ell \in \mathcal{L} \tag{9.8b}$$

where $\mathcal{I}$ is of tractable size but not $\mathcal{L}$. Remark that $\mathcal{I}$ variables in $\mathcal{L}$ suffices to define the vertex of the polyhedra that corresponds to the optimal solution. If we denote by $\mathcal{L}^*$ this set of constraints, and we solve the linear program with only these constraints, we would find an optimal solution of the problem. Unfortunately, we do not known $\mathcal{L}^*$. The idea of the line generation algorithm is to restrict ourselves to a set of constraints $\tilde{\mathcal{L}} \subset \mathcal{L}$ of tractable size, and to make this set evolve by adding constraints until we are able to prove that we have an optimal solution.

Algorithm 8 Line generation algorithm

```
1: **Input:** $(a_{\ell i})_{\ell i}$, $(b_\ell)_\ell$, $(c_i)_i$, a subset $\tilde{\mathcal{L}}$ of $\mathcal{L}$ ;
2: Solve the linear program (9.8) restricted to $\tilde{\mathcal{L}}$, denote $\tilde{\boldsymbol{x}}$ its solution;
3: **If** it has no solution **return** "(9.8) has no solution";
4: Solve $\min_{\ell \in \mathcal{L}} b_\ell - \sum_{i \in \mathcal{I}} a_{\ell i} \tilde{x}_i$, denote $\tilde{\ell}$ its solution;        *separation problem*
5: **If** $b_\ell - \sum_{i \in \mathcal{I}} a_{\ell i} \tilde{x}_i \geq 0$ **return** $\tilde{\boldsymbol{x}}$;
6: Add $\tilde{\ell}$ to $\tilde{\mathcal{L}}$; **Go to** step 2;
```

Given a vector $\boldsymbol{x}$ in $\boldsymbol{Q}^{\mathcal{I}}$, the *separation problem* associated to problem (9.8) for $\boldsymbol{x}$ is

$$\min_{\ell \in \mathcal{L}} b_\ell - \sum_{i \in \mathcal{I}} a_{\ell i} x_i.$$

The separation problem is also called *pricing subproblem*. A line separation algorithm will work efficiently only if we are able to solve efficiently the pricing subproblem.

Exercise 9.3.

1. Show that the line generation algorithm converges after a finite number of iterations.

   Solution. Once a line has been added, it is never removed, and thus never added again. Since there is a finite number of lines, the algorithm converges after a finite number of iterations. □

2. Show that it converges after a finite number of iterations even if a violated constraint but not necessarily the most violated one is added at each iteration.

   Solution. The proof above does not use the fact that the line generated is the most violated one. □

△

9.5.2 Column Generation

Column Generation aims at dealing with the case where the number of columns $\mathcal{I}$ is intractably large while the number of lines $\mathcal{L}$ is tractable. It consists in performing a line generation in the dual. Indeed, when we move to the dual, line becomes columns and columns lines, and we end up in the setting of line generation, where we have a tractable number of columns but an intractable number of lines.

9.5.3 Separation problem 😎

The column generation and the line generation algorithms work only if we are able to solve efficiently the separation problem. These statements are grounded on a strong theoretical results known as the *separation theorem* [Korte et al., 2017, see e.g. Theorem 4.21]. It states that we have a linear programming formulation for a problem $\mathcal{P}$ that has a polynomial number of columns but an exponential number of lines (resp. a polynomial number of columns but an exponential number of lines) in the size of $\mathcal{P}$, and if we have a separation algorithm that is polynomial in the size of $\mathcal{P}$, then there is a polynomial algorithm to solve $\mathcal{P}$. This algorithm is a variant of the ellipsoid method.

9.6 Further readings

We recommend the excellent textbook on linear programming by Matousek and Gärtner [2007].

9.7 Exercises

Exercise 9.4. *Ordonnancement et planning de projet*. On considère un projet constitué d'un ensemble de tâches J à executer. On cherche à trouver pour chaque tâche, la date à laquelle la commencer (alors elle s'execute sans interruption). Chaque tâche $i \in J$ a une durée $p_i \in \mathbb{R}_+$. On doit respecter des précédences, càd qu'on a un ensemble A^{prec} de paires (i,j) telles qu'on doit attendre la fin de l'execution de la tache i pour commencer la tâche j. L'objectif est de minimiser la durée totale du projet, c'est-à-dire la date de fin d'execution maximale d'une tâche.

1. Montrer que le problème peut etre modélisé par le programme linéaire (P) suivant, pour $D = (V, A)$ un graphe orienté acircuitique, $s, t \in V$ deux sommets, et w des poids à déterminer.

$$
\begin{array}{llll}
(P): & \min & x_t - x_s & \\
 & \text{s.t.} & x_j - x_i \geq w_{ij} & \forall (i,j) \in A \\
 & & x_i \in \mathbb{R} & \forall i \in V
\end{array}
$$

   Solution. Poser $V = J \cup \{s,t\}$ où s et t sont deux jobs fictifs représentant le début et la fin du projet. Poser $A = A^{prec} \cup \{(s,i), i \in J\} \cup \{(i,t), i \in J\}$. Les poids sont les p_i pour les arcs sortant d'un job de J, et 0 pour les arcs sortants de s. □

2. Calculer le dual (D) de (P). Quel problème du cours le PL (D) modélise-t-il ?

Solution.

$$\begin{array}{llll} (D): & \max & \sum_{(i,j)\in A} w_{ij} f_{ij} & \\ & \text{s.t.} & f^-(i) - f^+(i) = 0 & \forall i \in J \\ & & f^-(s) - f^+(s) = -1 & \\ & & f^-(t) - f^+(t) = 1 & \\ & & f_{ij} \geq 0 & \forall (i,j) \in A \end{array}$$

f est donc un s-t flot de valeur 1. □

3. Montrer que la matrice des contraintes de (D) est totallement unimodulaire.

4. Que dire d'une solution entière de (D) ?

   Solution. C'est un $s - t$ chemin (par ex par le théorème de décomposition des flots / ou bien se voit directement). □

5. Montrer que la valeur optimale de (P) est égale à la longueur maximale d'un $s - t$ chemin dans le graphe D pondéré par les w.

   Solution. Dualité forte PL + totale unimodularité. □

6. En déduire que la durée minimale du projet se calcule en temps polynomial : donner l'algorithme et la complexité.

   Solution. Prog dyn.

   □

7. Prouver que la longueur maximale d'un chemin dans le graphes est égale à la durée minimale du projet sans faire appel à la programmation linéaire.

△

Exercise 9.5. *Markov decision processes via linear programming*.

We consider on a horizon $\{0, \dots, T\}$ a system with finite state space $\mathcal{S}$, and finite decision space $\mathcal{U}$. At each time t, the state is in a state s, and the decision maker takes and action in $\mathcal{U}$, and the system evaluates to a state s' at $t + 1$.

Deterministic transitions We denote by $p^t_{s'|su}$ the binary defined as follows. If, when the system is in state s at t and action u is taken, the system is in state s' at $t+1$, then $p^t_{s'|su} = 1$. Otherwise, $p^t_{s'|su} = 0$. We have $\sum_{s'\in\mathcal{S}} p^t_{s'|su} = 1$, for all $s \in \mathcal{S}, u \in \mathcal{U}$, and $t < T$. Let $\boldsymbol{u}$ denote a vector of decisions $(u_0, \dots, u_{T-1})$, and $S_t(\boldsymbol{u})$ the state s of the system at time t if decisions $\boldsymbol{u}$ are taken.

1. Let $s_0, \dots, s_T$ be a sequence of T elements in S, and suppose that the system is initially in s_0. Show that

$$\prod_{t=0}^{T-1} p^t_{s_{t+1}|s_t u_t} = \begin{cases} 1 & s_t = S_t(\boldsymbol{u}) \text{ for all } t, \\ 0 & \text{otherwise.} \end{cases}$$

   Solution. $S_{t+1}(\boldsymbol{u})$ is the unique s in $\mathcal{S}$ such that $p^t_{s|S_t(\boldsymbol{u}),u_t}$ is equal to 1. The result follows by induction on t. □

Taking decision u when in state s at time $t < T$ costs $c_t(s,u)$. Being in state s at T costs $c_T(s)$. The problem considered is to control the system at minimum cost:

$$\min_{\boldsymbol{u}=(u_0,\ldots,u_{T-1})} C_T(S_T(\boldsymbol{u})) + \sum_{t<T} c_t(S_t(\boldsymbol{u}),u_t).$$

We denote by $V(t,s)$ the cost incurred between t and T if the system is in s at t and optimal decisions $\boldsymbol{u} = (u_t,\ldots,u_{T-1})$ given this initial state are taken.

$$V(t,s) = \min_{\boldsymbol{u}=(u_t,\ldots,u_{T-1})} C_T(S_T(\boldsymbol{u})) + \sum_{t\leq t'<T} c_{t'}(S_{t'}(\boldsymbol{u}),u_{t'}) \quad \text{subject to} \quad S_t = s \tag{9.9}$$

2. Write the dynamic programming equation satisfied by V.

   Solution.
   $$V(t,s) = \min_{u\in\mathcal{U}} c_t(s,u) + \sum_{s'\in\mathcal{S}} p^t_{s'|su} V(t+1,s')$$
   □

Consider the linear program with variables $v_{t,s}$ for $s \in \mathcal{S}$ and $t \in \{0,\ldots,T\}$

$$\max_v \frac{1}{|\mathcal{S}|}\sum_{s\in\mathcal{S}} v_{0,s} \tag{9.10a}$$

$$\text{s.t. } v_{t,s} \leq c(s,u) + \sum_{s'\in\mathcal{S}} v_{t+1,s'}p_{s'|s,u} \quad \forall s\in\mathcal{S}, \forall t\in\{0,\ldots,T-1\}, \forall u\in\mathcal{U} \tag{9.10b}$$

$$v_{T,s} \leq c_T(s) \quad \forall s\in\mathcal{S} \tag{9.10c}$$

A state s is *reachable* at time t if there exists states $s_0,\ldots,s_{t-1}$ and controls $u_0,\ldots,u_{t-1}$ such that

$$p^{t-1}_{s|s_{t-1}u_{t-1}} p^{t-2}_{s_{t-1}|s_{t-2}u_{t-2}} \cdots p^0_{s_1|s_0,u_0} = 1.$$

We suppose that all states are reachable.

3. Explain why, in an optimal solution of (9.10), $v_{t,s}$ is equal to the value of the solution $V(t,s)$ defined in Equation (9.9).

   Solution. Let v^* be the solution obtained by defining $v^*_{ts} = V(t,s)$. Then it is immediate that v^* satisfies Equation (9.10b) and hence is a solution of (9.10). Consider now a solution v of (9.10). Suppose that an inequality is strict for some $v_{t,s}$. Then, under our simplifying assumption, it is easy to show that the cost of $v_{t',s'} < v_{t,s'}$ for any $t' < t$ and $s' \in \mathcal{S}$. □

4. Let $\boldsymbol{v}^*$ be an optimal solution of (9.10). To what corresponds v_{0s}?

   Solution. To the cost of an optimal solution if the system is in s at $t = 0$. □

5. Explain how to deduce an optimal solution from an optimal solution of (9.10)

   Solution. The optimal solution of the linear program provides the optimal solution of the dynamic programming equation. Hence, if the state is in state s at t, we choose u_t in $\arg\min_u c_t(s,u) + \sum_{s'\in\mathcal{S}} v_{t+1,s'}p^t_{s'|s,u}$. □

We denote by $\mu_{Ts} \geq 0$ the dual associated to Constraint (9.10c) and $\mu_{tsu} \geq 0$ the dual associated to Constraint (9.10b).

6. Compute the dual of (9.10).

   Solution. The Lagrangian can be written

$$\begin{aligned}\mathcal{L}(v,\mu) = &\frac{1}{|\mathcal{S}|}\sum_{s\in\mathcal{S}} v_{0,s} + \sum_{t<T}\sum_{s\in\mathcal{S}}\sum_{u\in\mathcal{U}} \mu_{tsu}\Big(c(s,u) + \sum_{s'\in\mathcal{S}} v_{t+1,s'}p_{s'|s,u} - v_{ts}\Big)\\ &+ \sum_{s\in\mathcal{S}} \mu_{Ts}(c_T(s) - v_{Ts})\\ = &\sum_{t<T}\sum_{s\in\mathcal{S}}\sum_{u\in\mathcal{U}} \mu_{tsu}c(s,u) + \sum_{s\in\mathcal{S}}\mu_{Ts}c_T(s) + \sum_{s\in\mathcal{S}} v_{0s}\Big(\frac{1}{|\mathcal{S}|} - \sum_{u\in\mathcal{U}}\mu_{0su}\Big)\\ &+ \sum_{t=1}^{T} v_{ts}\Big(\sum_{s'\in\mathcal{S}}\sum_{u\in\mathcal{U}} \mu_{t-1,s',u}p_{s|s',u} - \sum_{u\in\mathcal{U}}\mu_{tsu}\Big)\end{aligned}$$

   Since v_{ts} belongs to $\mathbb{R}$, the dual $\min\limits_{\mu}\max\limits_{v}\mathcal{L}(v,\mu)$ becomes

$$\min_{\mu} \sum_{t<T}\sum_{s\in\mathcal{S}}\sum_{u\in\mathcal{U}} \mu_{tsu}c(s,u) + \sum_{s\in\mathcal{S}}\mu_{Ts}c_T(s) \tag{9.11a}$$

$$\text{s.t.} \sum_{u\in\mathcal{U}}\mu_{0su} = \frac{1}{|\mathcal{S}|} \qquad \forall s\in\mathcal{S} \tag{9.11b}$$

$$\sum_{u\in\mathcal{U}}\mu_{tsu} = \sum_{s'\in\mathcal{S}}\sum_{u\in\mathcal{U}}\mu_{t-1,s',u}p_{s|s',u} \qquad \forall t>0, \forall s\in\mathcal{S} \tag{9.11c}$$

$$\mu \geq 0 \tag{9.11d}$$

   □

7. Explain why there is a bijection between the integer solutions $\boldsymbol{\mu}$ of the dual (i.e., the solutions such that each variable belongs to $\{0,1\}$ and the solutions $\boldsymbol{u}$ of the initial problem.

   Solution. An immediate induction on t enables to show that $\mu_{st} \neq 0$ if and only if there exists $s_0, u_0, \dots, s_{t-1}, u_{t-1}, s_t$ such that $s_t = s$ and

$$\prod_{0\leq t'<t} p^{t'}_{s_{t'+1}|s_t u_t} = 1.$$

   The result then immediately follows from Question 1. □

In the rest of this exercise, we show that the solution $\boldsymbol{\mu}$ of the dual has a natural interpretation in a stochastic setting.

Markov decision processes *The rest of this exercise should be skipped by students that have never followed a course on Markov chains.*

We now suppose that transitions are non-deterministic: if the system is in state s at time t, and action u is taken, then the systems transits to the state s' with probability $p^t_{s'|s,u}$. In other words, we still have $\sum_{s'} p^t_{s'|s,u} = 1$, but now $p^t_{s'|s,u}$ belongs to the interval $[0,1]$ instead of $\{0,1\}$.

Due to the uncertainty of the transition, even if the sequence $u_0, \dots, u_T$ is known, the decision maker cannot know in advance which sequence of states will be visited. A policy $\boldsymbol{\delta} = (\delta^t_{a|s})_{s\in\mathcal{S},u\in\mathcal{U},t\in\{0,\dots,t-1\}}$ encodes the decisions taken by the decision maker. A policy is *deterministic* if the decision maker takes always the same action u when the system is in state s at t. In a *deterministic policy*, In that case, $\delta^t_{u|s}$ is equal to 1 if the decision maker takes action u when the system is in state s at t. The equation

$$\sum_{u\in\mathcal{U}} \delta^t_{u|s} = 1, \quad \forall s, \forall t$$

ensures that the decision maker takes always the same action if the system is in state s at time t. In a *non-deterministic* policy $\boldsymbol{\delta} = (\delta^t_{u|s})_{s\in\mathcal{S},u\in\mathcal{U},t\in\{0,\dots,t-1\}}$, we still have $\sum_{u\in\mathcal{U}} \delta^t_{u|s} = 1$, but now $\delta^t_{u|s}$ belongs to the interval $[0,1]$

In this stochastic setting, the state S_t and the action U_t become random variables. Let $\pi(s) := 1$ be the probability that the system is in state s at $t = 0$. A policy $\boldsymbol{\delta}$ induces a probability distribution $\mathbb{P}_{\boldsymbol{\delta}}$ on $S_0, U_0, \dots, u_{T-1}, S_T$ defined by

$$\mathbb{P}_{\boldsymbol{\delta}}(S_0 = s_0, U_0 = u_0, \dots, Ut_{t-1} = u_t, S_T = s_T) = \pi(s_0) \prod_{t=0}^{T-1} p^t_{s_{t+1}|s_t u_t} \delta^t_{u_t|s_t}$$

We denote by $\mathbb{E}_{\boldsymbol{\delta}}$ the expectation according to this distribution. The objective is to find a policy $\boldsymbol{\delta}$ that minimizes the expected cost

$$\min_{\boldsymbol{\delta}} \mathbb{E}_{\boldsymbol{\delta}}\Big(C_T(S_T) + \sum_{t<T} c_t(S_t, U_t)\Big)$$

8. Prove that, if we denote by

$$V(t,s) = \min_{\boldsymbol{\delta}} \mathbb{E}_{\boldsymbol{\delta}}\Big(C_T(S_T) + \sum_{t\leq t'<T} c_t(S_t, U_t)\Big| S_t = s\Big), \tag{9.12}$$

V satisfies the same dynamic programming equation as the one obtained at Question 2 in the deterministic setting.

Solution. We denote by $\boldsymbol{\delta}^{>t}$ the restriction of $\boldsymbol{\delta}$ to time indices greater than t. Conditionally to S_t, $\boldsymbol{\delta}^{>t}$ suffices to characterize $\mathbb{P}_{\boldsymbol{\delta}}$ on $S_{t+1}, U_{t+1}, \dots, u_{T-1}, S_T$

$$\begin{aligned}&\mathbb{P}_{\boldsymbol{\delta}}(S_{t+1} = s_{t+1}, U_{t+1} = u_{t+1}, \dots, U_{t-1} = u_t, S_T = s_T | S_t = s_t, U_t = u_t)\\ &= \prod_{t'=t+1}^{T-1} p^t_{s_{t'+1}|s_{t'} u_{t'}} \delta^t_{u_{t'}|s_{t'}}\end{aligned}$$

Hence, in the optimization problem (9.12), we can identify a solution by the devision u taken at t and the policy $\boldsymbol{\delta}^{>t}$, i.e.,

$$\begin{aligned}V(t,s) &= \min_{u\in\mathcal{U},\boldsymbol{\delta}^{>t}} \mathbb{E}_{\boldsymbol{\delta}^{>t}}\Big(C_T(S_T) + \sum_{t\leq t'<T} c_{t'}(S_{t'}, U_{t'})\Big| S_t = s, U_t = u\Big)\\ &= \sum_{u\in\mathcal{U}} c_t(s,u) + \sum_{s'\in\mathcal{S}} \mathbb{P}(S_{t+1} = s_{t+1} | S_t = s, U_t = u)\cdot\\ &\qquad \mathbb{E}_{\boldsymbol{\delta}^{>t}}\Big(C_T(S_T) + \sum_{t+1\leq t'<T} c_{t'}(S_{t'}, U_{t'})\Big| S_{t+1} = s', S_t = s, U_t = u\Big)\\ &= \min_{u\in\mathcal{U}} c_t(s,u) + \sum_{s'\in\mathcal{S}} p^t_{s'|su} V(t+1, s')\end{aligned}$$

□

9. Deduce that the linear program (9.10) and its dual still provide the optimal value of the problem.

   Solution. Immediate. □

10. Prove that there is always an optimal policy that is deterministic.

   Solution. The solution provided by the dynamic programming equation is deterministic. □

11. How to interpret the value of μ_{tsu} for a feasible solution?

   Solution. μ_{tsu} can be interpreted as the probability of being in s at time t and to take action u. Constraint (9.11c) can then be seen as the chain rule. And Constraint (9.11b) sets the initial probabilities at time $t = 0$. The objective is then naturally interpreted as the expectation. □

12. Explain how to deduce a feasible policy from a feasible solution of the dual (and an optimal policy from an optimal solution).

   Solution. If in state s at time t, take action u with probability

$$\frac{\mu_{tsu}}{\sum_{u'} \mu_{tsu'}}$$

   if the denominator is non-zero, and arbitrarily otherwise. Note that the probability of being in s at t is equal to 0 when the denominator of the ratio is equal to 0. □

13. To what corresponds the set of feasible solutions of the dual problem?

   Solution. To the moments on (S_t, U_t) induced by the feasible policies. □

△

Chapter 10

Integer Programming

A *mixed integer linear program* is an optimization problem of the form

$$\begin{aligned} \min \quad & \boldsymbol{cx}, \\ \text{s.t.} \quad & \boldsymbol{Ax} \leq \boldsymbol{b}, \\ & \boldsymbol{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p}. \end{aligned} \tag{MILP}$$

where $\boldsymbol{A}$ is a matrix in $\mathbb{R}^{m\times n}$, and $\boldsymbol{c}$ and $\boldsymbol{b}$ respectively belong to $\mathbb{R}^n$ and $\mathbb{R}^m$. The *linear relaxation* of Problem MILP is the linear program

$$\begin{aligned} \min \quad & \boldsymbol{cx}, \\ \text{s.t.} \quad & \boldsymbol{Ax} \leq \boldsymbol{b}, \\ & \boldsymbol{x} \in \mathbb{R}^n. \end{aligned} \tag{REL}$$

Recall that the value of a problem P is denoted by v(P) and equal to $+\infty$ if P admits no solution. We have

$$\mathrm{val}\,(\mathrm{REL}) \leq \mathrm{val}\,(\mathrm{MILP}). \tag{10.1}$$

Exercise 10.1. Prove Equation (10.1). △

Solution.

$$\mathrm{Sol}\,(\mathrm{MILP}) \subseteq \mathrm{Sol}\,(\mathrm{REL}).$$

□

10.1 Branch and bound algorithm

General method

Solving a combinatorial optimization problem

$$\min_{x\in X} f(x) \tag{10.2}$$

where X is finite by explicit enumeration becomes impractical as soon as the size of X increases. *Branch-and-bound* is a general method to solve combinatorial optimization problems by *implicit enumeration*. Suppose that $\mathcal{Y} \subseteq 2^X$ is a collection of parts of X such that $X \in \mathcal{Y}$, there exists an optimal solution x of (10.2) such that $\{x\} \in \mathcal{Y}$, that we have a *lower bound function* $\lambda : \mathcal{Y} \to \mathbb{R}$ such that

$$\lambda(Y) \leq \min_{x\in Y} f(x).$$

We also suppose to have a *branching function*

$$b : \mathcal{Y} \to \mathcal{Y}\times\mathcal{Y},\quad Y \mapsto Y_1 \times Y_2 \quad \text{s.t.} \quad \left|\begin{array}{l} Y_1 \cup Y_2 = Y, \\ Y_1 \cap Y_2 = \emptyset \end{array}\right.$$

Typically, we have $b(Y_i) \geq b(Y)$. Algorithm 9 states the branch-and-bound algorithm.

Algorithm 9 Branch-and-bound algorithm (general)

```
1:  Input: an instance of (10.2), $\lambda$ and $b$;
2:  Output: an optimal solution $x^*$ or $\emptyset$ if no optimal solution exist;
3:  $\mathcal{L} \leftarrow \{X\}$, $x^* \leftarrow \emptyset$, $u \leftarrow +\infty$;
4:  while $\mathcal{L} \neq \emptyset$ do
5:      extract $Y$ from $\mathcal{L}$;                                   node selection
6:      if $Y = \{y\}$ and $f(y) < u$ then
7:          $u \leftarrow f(y)$ and $x^* \leftarrow y$;
8:      else if $\lambda(Y) < u$ then
9:          $\mathcal{L} \leftarrow \mathcal{L} \cup \{Y_1, Y_2\}$ where $(Y_1, Y_2) = b(Y)$;          branching
10:     end if
11: end while
12: return: $x^*$
```

It is easy to show that Algorithm 9 admits the following invariants. First, if there is an optimal solution and $x = \emptyset$, then $\bigcup_{Y\in\mathcal{Y}}$ contains an optimal solution. Second, if Y and Y' are two elements of $\mathcal{Y}$, then $Y \cap Y' = \emptyset$. And third, u is an upper bound on the value of an optimal solution. These invariants enable to deduce that a part $Y \in \mathcal{Y}$ is considered at most once by the algorithm, and to deduce the following proposition.

Proposition 10.1. *The branch-and-bound algorithm converges after a finite number of iterations, and at the end of the algorithm, $x^* = \emptyset$ if $X = \emptyset$, and x^* is an optimal solution of* (10.2).

At any time during the algorithm, the quantity

$$\ell = \min_{Y\in\mathcal{Y}} \lambda(Y) \tag{10.3}$$

provides a lower bound on the value of the optimal solution. It can therefore be used to assess the quality of the current solution x, and may be used to decide to stop the algorithm before convergence because the current solution is proved to be of sufficient quality.

Remark that an execution of a branch-and-bound algorithm defines a rooted tree as follows. The nodes are the elements of $\mathcal{Y}$ considered by the algorithm – vertices of the tree are traditionally called node in that context. X is the root node. And the children of a node Y are the elements of $b(Y)$. A node Y is *pruned* when it is discarded because $\lambda(Y) > 0$.

Several decisions must be taken along a branch-and bound algorithm.

- *Node selection:* which node of $\mathcal{L}$ must be deal with?
- *Branching strategy selection:* choice of b.

Depending of the goal of the person solving the problem, different strategies can be chosen. If the goal is to find quickly a good quality solution but not to prove the optimality of the solution returned, a *depth-first-search*, which selects first node that are deep in the tree, is a good solution. But it is not a good solution if the goal is to find an optimal solution and prove its optimality. Indeed, in that case, a *breadth-first-search* tends to give better results.

Mixed Integer Linear Programs

Solving mixed integer programs MILP is one of the main applications of branch and bound. Recall that we consider solutions $\boldsymbol{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p}$. We denote by $\boldsymbol{x}_{[p]}$ the integer variables of $\boldsymbol{x}$. For convenience, we describe it on the mixed integer linear program In that case, the elements of $\mathcal{Y}$ are linear programs, and the branching procedure splits the solution space among the solutions such that $x_i \leq k$ and those such that $x_i \geq k+1$ for some integer variable x_i and some integer k. We denote by LP_0 the linear relaxation of (MILP). Algorithm 10 details the branch-and-bound algorithm for MILPs.

Algorithm 10 Branch-and-bound algorithm (MILP)

```
1:  **Input:** an instance of (MILP);
2:  **Output:** an optimal solution $\boldsymbol{x}^*$ or $\emptyset$ if no optimal solution exist;
3:  $\mathcal{L} \leftarrow \{\mathrm{LP}_0\}$, $\boldsymbol{x}^* \leftarrow \emptyset$, $u \leftarrow +\infty$;
4:  **while** $\mathcal{L} \neq \emptyset$ **do**
5:      extract LP from $\mathcal{L}$;                                    *node selection*
6:      solve LP;
7:      **if** LP has feasible solutions **then**
8:          let $\boldsymbol{x}^c$ denote an optimal solution of LP;
9:          **if** $\boldsymbol{x}^c_{[p]} \in \mathbb{Z}^p$ and val(LP) $< u$ **then**
10:             $u \leftarrow$ val(LP) and $\boldsymbol{x}^* \leftarrow \boldsymbol{x}^c$;
11:         **else if** val(LP) $< u$ **then**
12:             let $k \in [p]$ be such that $x_k \notin \mathbb{Z}$
13:             $\mathcal{L} \leftarrow \mathcal{L} \cup \{\mathrm{LP}_1, \mathrm{LP}_2\}$ where $\mathrm{LP}_1$ and $\mathrm{LP}_2$ are respectively obtained by adding con-
                straints $x_k \leq \lfloor \boldsymbol{x}^c_k \rfloor$ and $x_k \geq \lceil \boldsymbol{x}^c_k \rceil$ to LP;            *branching*
14:         **end if**
15:     **end if**
16: **end while**
17: **return:** $\boldsymbol{x}^*$
```

Practical efficiency

The practical efficiency of a branch-and-bound algorithm depends on the quality of the lower bounds used. If the difference between $\min_{x \in Y}(x)$ and $\lambda(Y)$ is small, then nodes can be pruned efficiently, and the number of nodes enumerated remains small. When it is not the case, the branch-and-bound algorithm may be slow.

Several alternative formulations are generally possible when modeling a combinatorial optimization problem as a (MILP). Given two alternative formulations of the same problem, one formulation is better than the other if the polyhedron of its linear relaxation is included in the polyhedron of the other one – there exists a surjection from the solutions of the linear relaxation of the first to the solutions of the second that preserves solution costs. The two next sections introduce good quality formulations and methods to strengthen the quality of the formulation.

10.2 Perfect formulations

A MILP

$$\begin{aligned} \min \quad & \boldsymbol{cx}, \\ \text{s.t.} \quad & \boldsymbol{Ax} \leq \boldsymbol{b}, \\ & \boldsymbol{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p}. \end{aligned}$$

is a *perfect formulation* is the polyhedron of its linear relaxation is the convex hull of its integer solutions, that is

$$\left\{\boldsymbol{x} \in \mathbb{Z}^n \colon \boldsymbol{Ax} \leq \boldsymbol{b}\right\} = \operatorname{conv}\left\{\mathbb{Z}^p \times \mathbb{R}^{n-p} \colon \boldsymbol{Ax} \leq \boldsymbol{b}\right\}.$$

The advantage of perfect formulations comes from the following proposition.

Proposition 10.2. *Any basic optimal solution of the linear relaxation is an optimal solution of the MILP*

In other words, it suffices to solve the linear relaxation of a perfect formulation to obtain its optimal solution. Hence, a perfect formulation can be solved in polynomial time in the size of the formulation. Remark that the size of such a formulation is not necessarily polynomial in the size of the problem it models. Indeed, it suffices to take the convex hull of the integer solution of an arbitrary formulation of a problem to obtain a perfect formulation of this problem. However, the number of faces in the convex hull is typically exponential in the size of the initial formulation.

For purely integer linear program, that is $p = n$, perfect formulations are characterized by totally unimodular matrices, which have been introduced in Section 9.4. The following problems are important combinatorial optimization problems that admit a perfect formulations:

- spanning trees (Chapter 4).
- paths and flows (Chapters 5 and 6).
- bipartite matchings (Chapter 7) and general matchings (Section 7.4 😎).

These perfect formulation naturally arise as subproblems in decomposition methods (Chapter 11).

10.3 Valid inequalities and Branch and Cut

A *valid inequality* or *valid cut* for a MILP

$$\begin{aligned} \min \quad & \boldsymbol{cx}, \\ \text{s.t.} \quad & \boldsymbol{Ax} \leq \boldsymbol{b}, \\ & \boldsymbol{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p}. \end{aligned}$$

is an inequality

$$\boldsymbol{a}^T\boldsymbol{x} \leq b$$

satisfied by any integer feasible solution $\boldsymbol{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p}$ but not by all the feasible solution of the linear relaxation. Valid inequalities typically enable to *strengthen the formulation* a MILP, that is, to improve the quality of its linear relaxation.

The *branch-and-cut* algorithm is an improvement of the *branch-and-bound* algorithm that uses valid inequalities to strengthen the formulation. It is extremely efficient and it is the kind of algorithm in use in state of the art general purpose MILP solvers. Branch-and-cut algorithm is obtained from branch-and-bound by adding the step

- decide whether to strengthen the formulation of LP, and strengthen it if decided;

between steps 6 and 7 of Algorithm 10. LP is strengthened as follows. A family $\mathcal{F}$ of valid inequalities $f = (a_f, b_f)$ is considered. Fixing the current solution $\boldsymbol{x}$, the most violated inequality f^* in $\mathcal{F}$ is identified by solving the SEPARATION PROBLEM

$$\min_{f \in \mathcal{F}} b_f - \boldsymbol{a}_f^T \boldsymbol{x}.$$

If $b_{f^*} - \boldsymbol{a}_{f^*}^T \boldsymbol{x}_{f^*} < 0$, then f^* is added to LP. The current solution $\boldsymbol{x}$ does not satisfy f, and hence adding f enables to strengthen LP.

In practice, solvers use families of inequality $\mathcal{F}$ of exponential size but whose seperation problem is easily solved. Indeed, if there is a small family of valid inequalities that strengthen a lot the formulation, all the inequalities in $\mathcal{F}$ can be added from the beginning. But on difficult $\mathscr{NP}$-complete problems, small families are generally not sufficient to improve a lot the quality of the LP. Families $\mathcal{F}$ of exponential size generally enable to obtain much stronger relaxations, but their size prohibit the addition of the complete families to LP. Solving the separation problem enables to add only valid inequalities in $\mathcal{F}$ that enable to strengthen LP the most. Hence, if the separation problem is well solved, branch-and-cut algorithm enables to benefit from large families of valid inequalities at small computational cost.

10.4 Modeling tricks

10.4.1 Logic constraints

10.4.2 McCormick inequalities

10.5 Meyer's theorem 😎

Theorem 9.5 ensures that a feasible and bounded linear program always admits an optimal solution. This is not the case of general mixed integer linear program, as shown in Exercise 10.2, and comes from the fact that the convex hull of the integer points $Q = \operatorname{conv}\big(\{\boldsymbol{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \colon A\boldsymbol{x} \leq \boldsymbol{b}\}\big)$ in a polyhedron $P = \{\boldsymbol{x} \in \mathbb{R}^n \colon A\boldsymbol{x} \leq \boldsymbol{b}\}$ is not necessarily a polyhedron. Simple sufficient conditions are:

- If $\{x \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \colon A\boldsymbol{x} \leq \boldsymbol{b}\}$ is finite, then Q is a polytope

- If $P = \{x \in \mathbb{R}^n \colon Ax \leq b\}$ is a polytope, then Q is a polytope.

The following theorem provides a much more general results.

Theorem 10.3. *(Meyer) Let A be a rational matrix and $\boldsymbol{b}$ be a rational vector. Then $Q = \operatorname{conv}\big(\{x \in \mathbb{Z}^p \times \mathbb{Q}^{n-p} \colon Ax \leq b\}\big)$ is a polyhedron and there exists a rational matrix A' and a rational vector $\boldsymbol{b}'$ such that*

$$Q = \{x \in \mathbb{R}^n \colon A'\boldsymbol{x} \leq \boldsymbol{b}'\}$$

10.6 Further readings

10.7 Exercises

Exercise 10.2. We recall that $\sqrt{3}$ is not rational. Consider the integer program

$$\begin{aligned} &\min \sqrt{3}x_2 - x_1 \\ &\text{s.t.} 1 \leq x_1 \leq \sqrt{3}x_2 \\ &x_1, x_2 \in \mathbb{Z}_+ \end{aligned}$$

- Construct a sequence $(x^k)_k$ of feasible solutions such that $\sqrt{3}x_2^k - x_1^k \underset{k\to\infty}{\longrightarrow} 0$.
- Deduce that the integer program has no optimal solution.
- Deduce that the convex hull of $\{x_1, x_2 \in \mathbb{Z}_+ : 1 \leq x_1 \leq \sqrt{3}x_2\}$ is not a polyhedron.

$\triangle$

Exercise 10.3. *Approximate Branch-and-bound* When the branch-and-bound algorithm takes too much time to find an optimal solution, one may be willing to settle for an approximate solution. Assume that the optimum value z^* is positive. Modify the branch-and-bound algorithm to find a feasible solution whose objective value $\overline{z}$ is within $p\%$ of the optimum, i.e., $\frac{\overline{z}}{z^*} \leq 1 + p/100$. $\triangle$

Exercise 10.4. A firm is considering project $A, B, \dots, H$. Using binary variables $x_a, \dots, x_h$ and linear constraints, model the following conditions on the projects to be undertaken.

1. At most one of $A, B, \dots, H$.
2. Exactly two of $A, B, \dots, H$.
3. If A then B.
4. If A then not B.
5. If not A then B.
6. If A then B, and if B then A.
7. If A then B and C.
8. If A then B or C.
9. If B or C then A.
10. If B and C then A.
11. If two or more of B, C, D, E then A.
12. If m or more than n projects$B, \dots, H$ then A.

$\triangle$

Exercise 10.5. *Linearization* Suppose that $x \in \{0, 1\}$ and $y, z \in \mathbb{R}$, with $0 \leq y \leq M$ are variables of a MIP. Shot that the quadratic constraint

$$z = xy$$

can be replaced by several linear constraints. $\triangle$

Exercise 10.6. Jobs $1, ..., n$ must be processed on a single machine. Each job is available for processing after a certain time, called release time. For each job we are given its release time r_i , its processing time p_i and its weight w_i . Formulate as an integer linear program the problem of sequencing the jobs without overlap or interruption so that the sum of the weighted completion times is minimized:

- If $r_i \in \mathbb{Z}_+$ and $p_i \in \mathbb{Z}_+$
- If $r_i \in \mathbb{R}_+$ and $p_i \in \mathbb{R}_+$

$\triangle$

Exercise 10.7. On considère une usine de pneus sur un certain horizon, divisé en m périodes. Les pneus sont produits à l'aide de moules. Les moules, placés sur des supports adaptés, peuvent être changés en début de période. Il y a k types de moules. On suppose la demande connue sur l'horizon, i.e. que pour chaque période, on connaît le nombre minimum de moules de chaque type nécessaire pour assurer la production attendue. On connaît aussi pour chaque période le nombre total de moules de chaque type qui seront disponibles. Les coûts proviennent des opérations de placement des moules.

Les notations sont les suivantes. Le nombre de supports est n, i.e. qu'à tout instant, on ne peut avoir strictement plus de n moules en place. Le nombre minimum (resp. maximum) de moules de type i pouvant être en place sur la période j est noté r_{ij} (resp. d_{ij}). Enfin, le coût de placement d'un moule de type i est c_i.

Juste avant la première période, il y a déjà des moules en place et on note p_i le nombre de moules en place du type i. L'objectif de ce problème est de trouver la politique de placement des moules qui minimise le coût sur l'horizon considéré, tout en satisfaisant les contraintes énoncées. On supposera dans tout le problème que les paramètres p_i, r_{ij}, d_{ij} et c_i sont strictement positifs, et que les p_i, r_{ij} et d_{ij} sont de plus entiers.

L'objet de ce problème est de proposer deux approches possibles pour résoudre ce problème, l'une par la programmation linéaire en nombres entiers, l'autre par les flots. Les deux approches sont indépendantes.

Formulation du problème comme un programme linéaire mixte On introduit la variable x_{ij} qui modélise le nombre de moules de type i en place sur la période j pour $i = 1, \ldots, k$ et $j = 1, \ldots, m$. On étend cette variable à $i = 0$ et $j = 0$ en définissant x_{0j} comme étant le nombre de supports non-utilisés sur la période j et x_{i0} comme étant le nombre de moules de type i en place juste avant la première période (on a donc la contrainte triviale $x_{i0} = p_i$ pour $i = 1, \ldots, k$). On introduit aussi la variable y_{ij} qui est le nombre de moules de type i supplémentaires ajoutés en début de période j à ceux déjà en place. Si on n'en place pas ou si on en retire, y_{ij} est nul.

1. Ecrire la fonction objectif du problème.
2. Ecrire l'égalité qui lie n et les x_{ij}, pour $i = 0, \ldots, k$.
3. Ecrire les inégalités qui lient la variable x_{ij} et respectivement r_{ij} et d_{ij}, pour $i = 1, \ldots, k$ et $j = 1, \ldots, m$.
4. Ecrire l'égalité qui lie la variable y_{ij} aux variables x_{ij-1} et x_{ij} pour $i = 1, \ldots, k$ et $j = 1, \ldots, m$. (Attention, cette égalité n'est pas linéaire, cf définition de y_{ij}).
5. En utilisant les égalités et les inégalités écrites aux questions précédentes, et en les complétant si nécessaire, écrire le problème sous la forme d'un programme mathématique, à ce stade non-linéaire à cause de la contrainte liant la variable y_{ij} aux variables x_{ij-1} et x_{ij}.

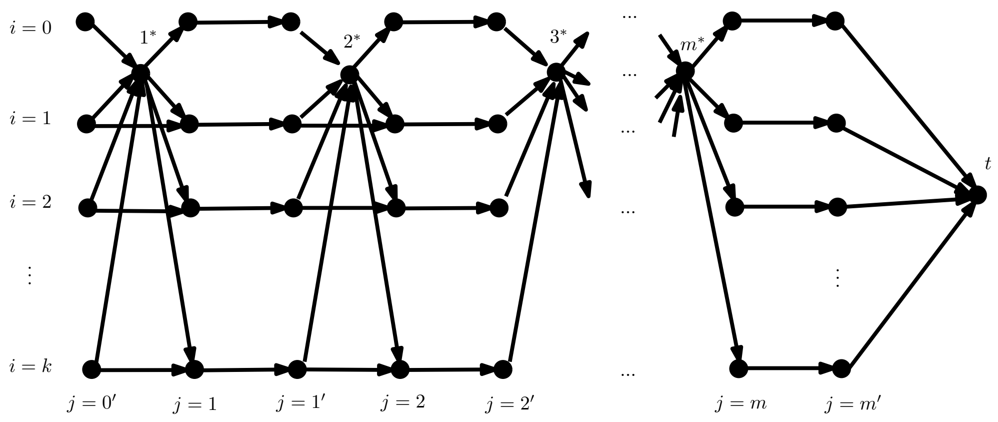


Figure 10.1: Le graphe de la question 7

6. Transformer ce programme mathématique en un programme linéaire en nombres entiers équivalent.

Formulation du problème comme un problème de *b*-flot de coût minimum La construction de cette partie du problème s'appuie sur le graphe dont les sommets sont :

- $(i, 0')$ pour $i = 0, \ldots, k$
- (i, j) et (i, j') pour $i = 0, \ldots, k$ et $j = 1, \ldots, m$
- j^* pour $j = 1, \ldots, m$
- un sommets fictif t,

et les arcs sont

- $\big((i,j),(i,j')\big)$ pour $i = 0, \ldots, k$ et $j = 1, \ldots, m$
- $\big((i,j'),(i,j+1)\big)$ pour $i = 1, \ldots, k$ et $j = 1, \ldots, m-1$
- $\big((i,j'),(j+1)^*\big)$ pour $i = 0, \ldots, k$ et $j = 0, \ldots, m-1$
- $\big(j^*,(i,j)\big)$ pour $i = 0, \ldots, k$ et $j = 1, \ldots, m$
- $\big((i,m'),t\big)$ pour $i = 0, \ldots, k$.

Le graphe est illustré Figure 10.1.

7. Montrer que ce problème se résout comme un problème de b-flot de coût minimum sur ce graphe. Pour cela,
   - on précisera pour chaque arc les capacités inférieures et supérieures et le coût, et pour chaque sommet la valeur de b correspondante,
   - on indiquera la signification du flot sur chaque arc
   - on justifiera le fait que résoudre le problème de b-flot permet de résoudre le problème de l'énoncé.

   Le problème peut donc se résoudre en temps polynomial, ce qui n'était pas forcément évident de prime abord.

$\triangle$

Exercise 10.8. *Interception d'un commando* $\triangle$

Préliminaire : formulation du problème du plus court chemin comme programme linéaire Soit $D = (V, A)$ un graphe orienté, s et t deux sommets particuliers de ce graphe, et des coûts $c_a > 0$ définis pour tout $a \in A$. Considérons le programme linéaire suivant :

$$\begin{array}{rll} \max & y_t & \\ \text{s.c.} & y_v - y_u \leq c_{(u,v)} & \forall (u,v) \in A \\ & y_s = 0. & \end{array} \tag{10.4}$$

L'objectif de cette sous-section est de démontrer que ce programme linéaire (de maximisation) a comme valeur optimale le coût minimum d'un s-t chemin. Notons P un tel s-t chemin de coût minimum.

1. Montrer que si on définit y_v comme le coût minimal d'un s-v chemin, on obtient une solution réalisable du programme (10.4) donnant comme valeur à la fonction objectif le coût de P.

   Réciproquement, soit y^* une solution optimale du programme (10.4). Notons $v_0, v_1, \ldots, v_\ell$ la suite des sommets visités par P (dans cet ordre ; on a donc $v_0 = s$ et $v_\ell = t$).

2. Montrer par récurrence sur i que $y^*_{v_i} \leq \sum_{j=1}^{i} c_{(v_{j-1},v_j)}$, en déduire que y^*_t est inférieur ou égal au coût de P, et conclure.

Le problème Votre mission est de positionner des guetteurs en différents points d'une zone afin de repérer une intrusion d'un commando. On suppose que l'on dispose de k guetteurs. L'ensemble des mouvements possibles dans la zone est modélisé par un graphe orienté $D = (V, A)$. Les sommets représentent les positions que peut occuper le commando, et les arcs représentent les mouvements possibles qu'il peut faire. On note s le sommet représentant l'entrée de la zone et t le sommet représentant sa sortie.

Soit X l'ensemble des positions possibles pour les guetteurs. On note $p_{a,i}$ la probabilité de repérer le commando s'il passe sur l'arc a lorsqu'on a un ou plusieurs guetteurs en $i \in X$. On supposera dans toute la suite que $0 \leq p_{a,i} < 1$ pour tout a et tout i. Choisir les positions des guetteurs, c'est choisir un sous-ensemble $Y \subseteq X$ tel que $|Y| \leq k$.

Supposons que l'ensemble Y des positions des guetteurs ait été choisi.

1. Soit P un s-t chemin élémentaire suivi par le commando. Justifier que la probabilité que le commando se fasse repérer est (sous une hypothèse d'indépendance naturelle)

   $$1 - \prod_{a \in A(P)} \prod_{i \in Y} (1 - p_{a,i}).$$

   Soit P^Y un s-t chemin qui minimise la probabilité que le commando se fasse repérer (lorsque les positions des guetteurs sont les positions dans Y).

2. Montrer que P^Y est un s-t chemin P qui minimise $\sum_{a \in A(P)} \gamma_a$, avec

   $$\gamma_a = - \sum_{i \in Y} \ln(1 - p_{a,i}).$$

   Le problème que vous souhaitez résoudre consiste à trouver l'ensemble Y tel que

   $$1 - \prod_{a \in A(P^Y)} \prod_{i \in Y} (1 - p_{a,i})$$

   est le plus grand possible : vous voulez choisir Y de manière à ce que le meilleur chemin pour le commando ait le plus grande probabilité possible de repérage.

3. En utilisant le résultat prouvé à la section 10.7, écrire un programme linéaire mixte (avec des variables entières et des variables réelles) qui modélise ce problème, en utilisant d'une part des variables réelles $(y_v)_{v \in V}$ et d'autre part des variables binaires $(x_i)_{i \in X}$, qui valent 1 si un ou plusieurs guetteurs sont positionnés en i, et 0 sinon.

Chapter 11

Relaxations and decompositions

Modern MILP solvers can tackle formulations with up to hundreds of thousands of variables and constraints, provided their matrix are sparse and they have a good linear relaxation. An matrix is *sparse* if it has few non-zero coefficients. An MILP is sparse if its constraint matrix is sparse. For large but *sparse* instances that cannot be dealt with using present day solvers, decomposition and relaxation techniques provide powerful alternatives. They exploit the specific structure of the instance to ease the resolution.

The two first techniques we introduce are Lagrangian-Relaxation and Dantzig-Wolfe decomposition. Both apply to MILP whose constraint matrix has a block-diagonal structure like the one illustrated on Figure 11.1. The lines, i.e. the constraints, of such matrices can be partitioned into

- Linking constraints, represented in blue on Figure 11.1.
- Blocks of constraints $B_1, \dots, B_k$, such that, if $i \neq j$, the variables that intervene in constraints of block B_i are different from those that intervene in the constraints of block B_j. Such blocks are illustrate in red on Figure 11.1.

The linking constraints are "complicating constraints". If these constraints were not there, we would have several smaller and simpler MILP problems, one for each block. Lagrangian relaxation and DW decomposition can also apply to problem with linking constraints, as the one illustrated on Figure 11.2.a: several block of constraints, illustrated in red, would be independent if they were not coupled by some linking variables, illustrated in purple. An equivalent problem with linking constraints is naturally obtained: it suffices to introduce a copy of the linking variable for each block, illustrated in cyan on Figure 11.2.b, and to add linking constraints enforcing the equality of the copies, illustrated in blue on the figure.

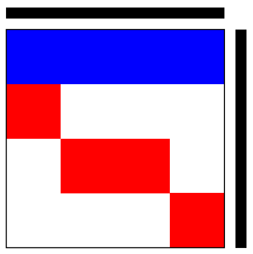


Figure 11.1: Block diagonal decomposition of a MILP

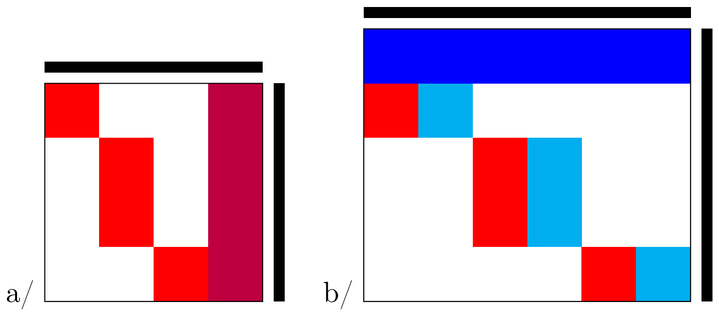


Figure 11.2: Turning linking variables into linking constraints

Dantzig-Wolfe decomposition and Lagrangian relaxation exploit the block diagonal structure to solve more efficiently the initial problem in two ways.

1. They give a tractable relaxation of the problem that is better than the linear relaxation – both give the same bound.

2. Pricing subproblems corresponding to each blocks, which are identical for the two methodologies, can be solved with ad hoc solvers.

If the bound is not improves, or if the pricing subproblems cannot be solved efficiently these approaches are probably not good options. The relative advantages of both approaches are the following ones. Lagrangian relaxation is easy to implement, and does not require an LP solver. It is a good option when the objective is to compute one bound or to design a matheuristic. It is less appropriate to solve the problem to optimality, or when finding a feasible solution of the initial problem is difficult. Dantzig-Wolfe decomposition machinery is more involved, and when implemented, it requires more memory than the Lagrangian-relaxation approach. However, it has the advantage of being purely combinatorial when Lagrangian relaxation is numerical, and is better adapted when finding a feasible solution is difficult. Using Branch-and-Price, it enables to prove optimality.

Benders decomposition applies directly to the block diagonal structure with linking constraints of Figure 11.2.a., where in addition the block variables must be continuous. The linking variables can be integer. Benders decomposition enables to solve a problem only in the initial variable, solving each pricing subproblem separately to generate cuts. Contrary to the previous approaches, its objective is not to improve the lower bound used to cut. It is only to reduce the number of variables. If the initial problem is a linear program, Benders decomposition can be seen as a dual version of Dantzig-Wolfe decomposition. However, the analogy stops when mixed integer variables are introduced, and these decompositions are relevant on very different kinds of problem.

Lagrangian relaxation and Dantzig-Wolfe decomposition are more of success stories than Benders decomposition, as we will see in Section 11.3. This comes probably from the fact that, first, integer variables can be considered in the subproblems, and second, they provide an improved bound. However, Benders decomposition has one very important application, where it is probably the best option: stochastic optimization problems.

11.1 Lagrangian relaxation

11.1.1 Definition and interest

Consider the “Primal” Mixed Integer Linear Program, that we write w.l.o.g. in its equational form.

$$\begin{aligned} z_{\mathrm{I}} = \min\ & \boldsymbol{cx} && (11.1a)\\ \text{s.t.}\ & A_1\boldsymbol{x} = \boldsymbol{b}_1 && (11.1b)\\ & A_2\boldsymbol{x} = \boldsymbol{b}_2 && (11.1c)\\ & \boldsymbol{x} \in \mathbb{Z}_+^p \times \mathbb{R}_+^{n-p} && (11.1d)\end{aligned}$$

where we have partitioned the rows of the constraints matrix A into $(A_1, B_1)^t$, and z denotes the value of the program. Typically, $A_1\boldsymbol{x} = b_1$ contains “complicating constraints”, and solving (11.1) without these constraints would be much easier. On Figure 11.1, constraints $A_1\boldsymbol{x} = \boldsymbol{b}_1$ corresponds to the blue block, while $A_2\boldsymbol{x} = \boldsymbol{b}_2$ corresponds to the red blocks.

The *Lagrangian relaxation* is an application of the Lagrangian duality introduced in Section 8.1. It is obtained by taking the Lagrangian dual (D) obtained by dualizing Constraints (11.1c), and using

$$X = \left\{\boldsymbol{x} \in \mathbb{Z}_+^p \times \mathbb{R}_+^{n-p} \colon A_2\boldsymbol{x} = \boldsymbol{b}_2\right\}.$$

Denoting $\boldsymbol{\lambda}$ the vector of duals in $\mathbb{R}^q$ corresponding to the q constraints in (11.1c),

$$z_{\mathrm{LR}}(\boldsymbol{\lambda}) = \min_{\boldsymbol{x}\in X} \boldsymbol{cx} + \boldsymbol{\lambda}^T(A_1\boldsymbol{x} - \boldsymbol{b}_1) \tag{11.2}$$

the dual is

$$\sup_{\boldsymbol{\lambda}\in\mathbb{R}^q} z_{\mathrm{LR}}(\boldsymbol{\lambda}). \tag{11.3}$$

Theorem 8.1 gives the following proposition.

Proposition 11.1. *$z_{\mathrm{LR}}(\boldsymbol{\lambda}) \leq z_{\mathrm{LD}} \leq z_{\mathrm{I}}$ for all $\lambda \in \mathbb{R}^q$.*

11.1.2 Quality of the bound

The following results characterizes the bound z_{LD}. Recall that $\mathrm{conv}(X)$ is the convex hull of X.

Theorem 11.2. *(Geoffrion) If X is finite or A_2 and $\boldsymbol{b}_2$ have rational coefficients, then*

$$z_{\mathrm{LD}} = \min\left\{\boldsymbol{cx} \colon \boldsymbol{x} \in \mathrm{conv}(X), A_1\boldsymbol{x} = \boldsymbol{b}_1\right\}.$$

An immediate corollary of Theorem 11.2 is that the Lagrangian relaxation z_{LR} provides a better lower bound on the value of the value z_{I} of the integer program (11.1) than its linear relaxation z_{lin}

$$z_{\mathrm{lin}} \leq z_{\mathrm{LR}} \leq z_{\mathrm{I}}.$$

Figure 11.3 illustrates this fact and Theorem 11.2. The red polyhedron in plain line is $\{\boldsymbol{x} \colon A_2\boldsymbol{x} = \boldsymbol{b}_2\}$. The dashed red polyhedron is the polyhedron $\mathrm{conv}(X)$, i.e., the convex hull of the integer solutions in the plain red polyhedron. The quality of the Lagrangian relaxation comes from the fact that the dashed red polyhedron is contained in and strictly smaller than the plain red one.

The main interest of the Lagrangian relaxation comes from this improved lower bound. Indeed, using z_{LR} instead of z_{lin} in a branch-and-bound algorithm enable to cut more nodes, and hence to explore a smaller part of the branch-and-bound tree.

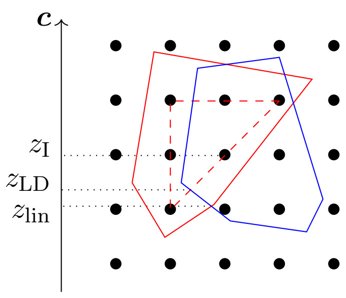


Figure 11.3: Value of the primal z_{I} and of the Lagrangian relaxation z_{LD}

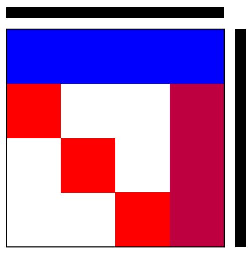


Figure 11.4: Linking constraints and variables

Proof of Theorem 11.2. Meyer's theorem ensures that $\operatorname{conv}(X)$ is a polyhedron, there exists rational A_2' and $\boldsymbol{b}_2'$ such that

$$\operatorname{conv}(X) = \{\boldsymbol{x} \in \mathbb{R}^n \colon A_2'\boldsymbol{x} \leq \boldsymbol{b}_2'\}$$

and hence

$$\min_{\boldsymbol{x}\in X} \boldsymbol{c}\boldsymbol{x} + \boldsymbol{\lambda}^T(A_1\boldsymbol{x} - \boldsymbol{b}_1) = \min_{\boldsymbol{x}} \big\{\boldsymbol{c}\boldsymbol{x} + \boldsymbol{\lambda}^T(A_1\boldsymbol{x} - \boldsymbol{b}_1) \colon A_2'\boldsymbol{x} \leq \boldsymbol{b}_2'\big\},$$

where the right hand side is a linear program. Applying twice linear programming strong duality, we obtain

$$\begin{aligned}
z_{\mathrm{LD}} &= \max_{\boldsymbol{\lambda}} z_{\mathrm{LR}}(\boldsymbol{\lambda}) \\
&= \max_{\boldsymbol{\lambda}} \min_{\boldsymbol{x}} \big\{\boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{\lambda}^T(A_1\boldsymbol{x} - \boldsymbol{b}_1) \colon A_2'\boldsymbol{x} \leq \boldsymbol{b}_2'\big\} \\
&= \max_{\boldsymbol{\lambda}} -\boldsymbol{\lambda}^T\boldsymbol{b}_1 + \min_{\boldsymbol{x}} \big\{(\boldsymbol{c}^T + \boldsymbol{\lambda}^T A_1)\boldsymbol{x} \colon A_2'\boldsymbol{x} \leq \boldsymbol{b}_2'\big\} \\
&= \max_{\boldsymbol{\lambda}} -\boldsymbol{\lambda}^T\boldsymbol{b}_1 + \max_{\boldsymbol{y}} \big\{\boldsymbol{y}^T\boldsymbol{b}_2' \colon \boldsymbol{y}_2 \leq 0 \text{ and } \boldsymbol{y}_2^T A_2' = (\boldsymbol{c}^T + \boldsymbol{\lambda}^T A_1)\big\} \\
&= \max_{\boldsymbol{\lambda},\boldsymbol{y}} \big\{ -\boldsymbol{\lambda}^T\boldsymbol{b}_1 + \boldsymbol{y}^T\boldsymbol{b}_2' \colon \boldsymbol{y}_2 \leq 0 \text{ and } \boldsymbol{y}_2^T A_2' - \boldsymbol{\lambda}^T A_1 = \boldsymbol{c}^T\big\} \\
&= \min_{\boldsymbol{x}} \big\{\boldsymbol{c}\boldsymbol{x} \colon A_1\boldsymbol{x} = \boldsymbol{b}_1 \text{ and } A_2'\boldsymbol{x} \leq \boldsymbol{b}_2\big\} \\
&= \min \big\{\boldsymbol{c}\boldsymbol{x} \colon \boldsymbol{x} \in \operatorname{conv}(X), A_1\boldsymbol{x} = \boldsymbol{b}_1\big\},
\end{aligned}$$

which gives the theorem. □

Remark 11.1. **Linking variables.** Consider now the case with linking constraints and linking variables illustrated in Figure 11.2.a. As explained in the introduction and illustrated on

Figure 11.2, copies of linking variables are added in each block, and we relax the linking constraints enforcing the equality of these copies. Suppose that the initial problem (without copies) is composed of linking constraints

$$A_1\boldsymbol{x} \leq \boldsymbol{b}_1,$$

and of blocks

$$A_i\boldsymbol{x} \leq \boldsymbol{b}_i, \quad \text{for } i \geq 2,$$

where the A_i include the linking variables. Let

$$X_i = \{\boldsymbol{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \colon A_i\boldsymbol{x} \leq \boldsymbol{b}_i\}.$$

Then an extended version of Geoffrion theorem shows that, if A has rational coefficients, the Lagrangian relaxation bound is equal to

$$\min \left\{ \boldsymbol{c}^T\boldsymbol{x} \colon A_1\boldsymbol{x} \leq \boldsymbol{b}_1 \text{ and } \boldsymbol{x} \in \bigcap_{i\geq 2} \operatorname{conv}(X_i) \right\}.$$

linking variable case illustrated in Figure 11.2.a., which is dealt with by adding copies of linking variables in each block, as illustrated in Figure 11.2.b., and relaxing the linking constraints enforcing the equality of these copies. Suppose that each block is of the form

$$A_i\boldsymbol{x} \leq \boldsymbol{b}_i, \quad \text{for } i \geq 2.$$

We suppose that the initial problem still has linking constraint

$$A_1\boldsymbol{x} \leq \boldsymbol{b}_1.$$

Geoffrion theorem has a nice interpretation: the Lagrangian relaxation bound is equal to the solution of △

11.1.3 Computing the bound: subgradient algorithm

We now introduce an algorithm to compute z_{LD}.

Proposition 11.3. *$\boldsymbol{\lambda} \mapsto z_{\mathrm{LR}}(\lambda)$ is concave.*

Proof. It is an infimum of concave functions. □

The following proposition is immediate.

Proposition 11.4. *If X is finite, $\boldsymbol{\lambda} \mapsto z_{\mathrm{LR}}(\lambda)$ is piecewise affine.*

Given a concave function h on a part Y of $\mathbb{R}^q$, a *supergradient* $\boldsymbol{p}$ of a concave function h in $\boldsymbol{\lambda}$ is a vector $\boldsymbol{p}$ such that

$$h(\boldsymbol{\mu}) - h(\boldsymbol{\lambda}) \leq \boldsymbol{p}^T(\boldsymbol{\mu} - \boldsymbol{\lambda}) \qquad \text{for all } \mu \text{ in } Y.$$

Proposition 11.5. *Given $\boldsymbol{\lambda}$, if $\boldsymbol{x}$ is an optimal solution of $\mathcal{L}(\boldsymbol{x},\boldsymbol{\lambda})$, then $A_1\boldsymbol{x}-\boldsymbol{b}_1$ is a subgradient of $\boldsymbol{\lambda} \mapsto z_{\mathrm{LR}}(\boldsymbol{\lambda})$ in $\boldsymbol{\lambda}$.*

Proof. Given such an x, we have

$$z_{\mathrm{LR}}(\boldsymbol{\mu}) - z_{\mathrm{LR}}(\boldsymbol{\lambda}) \leq \mathcal{L}(\boldsymbol{x},\boldsymbol{\mu}) - \mathcal{L}(\boldsymbol{x},\boldsymbol{\mu}) = (A_1\boldsymbol{x} - \boldsymbol{b}_1)^T(\boldsymbol{\mu} - \boldsymbol{\lambda}).$$

□

The *subgradient algorithm* can be described as follows. First, choose $\boldsymbol{\lambda}_0$ arbitrarily. At each step k, compute a subgradient $\boldsymbol{p}_k$ of $z_{\mathrm{LR}}(\,\cdot\,)$ in $\boldsymbol{\lambda}_k$ using Proposition 11.5. Then set

$$\boldsymbol{\lambda}_{k+1} = \boldsymbol{\lambda}_k + \frac{\rho_k}{\|\boldsymbol{p}_k\|} p_k,$$

where $(\rho_k)_{k\in\mathbb{Z}_+}$ is a sequence in $\mathbb{R}^+$. The algorithm stops when $\boldsymbol{p}_k = 0$. If $(\rho_k)_{k\in\mathbb{Z}_+}$ is such that

$$\lim_{k\to\infty} \rho_k = 0 \quad \text{and} \quad \sum_{k=0}^{+\infty} \rho_k = \infty,$$

the concavity of $z_{\mathrm{LR}}(\,\cdot\,)$ ensures the convergence of $\boldsymbol{\lambda}_k$ to a supremum.

Remark 11.2. Subgradients are the analogues of a supergradients for convex functions. The name "subgradient algorithm" comes from the fact that the theory has been developed for convex functions. △

More advanced convex optimization algorithms such as bundle methods can of course be used instead of the subgradient algorithm.

11.1.4 Branch and price

Branch-and-Bound algorithm can be adapted to use Lagrangian bound instead of the linear relaxation bound. The resulting algorithm, Branch-an-price, is discussed in more details in the context of Dantzig-Wolfe decomposition.

11.1.5 Heuristic

Lagrangian heuristics are problem specific. However two general principles can be used:

- rounding variables to restore integrality.
- approximate branching.

11.2 Dantzig Wolfe decomposition

11.2.1 Definition and link with Lagrangian relaxation

11.2.2 Branch and Price

11.2.3 Avoid branching when integrality gap is small

11.2.4 Matheuristics

11.3 Applications of Lagrangian relaxation and Dantzig-Wolfe decomposition

11.3.1 Bin packing

Suppose that we have K bins of size W available, and n objects of size $\{a_1, \dots, a_n\}$. The bin-packing problem can be written

$$\begin{array}{lll} \min & \sum_{j=1}^K z_j & \\ \text{s.t.} & \sum_{j=1}^K y_{ji} = 1 & i = 1, \dots, n \\ & \sum_{i=1}^n a_i y_{ji} \le W z_j & j = 1, \dots, K \\ & y_{ji}, z_i \in \{0,1\} & i = 1, \dots, n\,;\, j = 1, \dots, K \end{array} \tag{11.4}$$

where $z_j = 1$ if bin j is used, and $y_{ji} = 1$ if object i is in j. Dualizing the constraint $\sum_{j=1}^K y_{ji} = 1$, we obtain

$$\begin{aligned} z_{\mathrm{LR}}(\boldsymbol{\lambda}) := \quad \min \quad & \sum_{j=1}^K z_j + \sum_{i=1}^n \lambda_i (\sum_{j=1}^K y_{ji} - 1) & \\ \text{s.t.} \quad & \sum_{i=1}^n a_i y_{ji} \le W z_j & j = 1, \dots, K \\ & y_{ji}, z_i \in \{0,1\} & i = 1, \dots, n\,;\, j = 1, \dots, K \end{aligned}$$

which can be rewritten

$$z_{\mathrm{LR}}(\boldsymbol{\lambda}) := -\sum_{i=1}^n \lambda_i + \sum_{j=1}^K S_j(\boldsymbol{\lambda})$$

where $S_j := \min_{\boldsymbol{y}, z \in \{0,1\}} \left\{ z + \sum_{i=1}^n \lambda_i y_i : \sum_{i=1}^n a_i y_i \le W z \right\}$. Computing $z_{\mathrm{LR}}(\boldsymbol{\lambda})$ therefore only requires to solve K knapsack problems.

We now show that the bound z_{LD} is better than the bound $\left\lceil \frac{\sum_{i=1}^n a_i}{W} \right\rceil$ provided by the linear relaxation. Let α he the optimal solution of

$$\begin{aligned} \max & \frac{1}{W} \sum_{i=1}^n a_i x_i \\ \text{s.t.} & \sum_{i=1}^n a_i x_i \le W \\ & x_i \in \{0,1\}. \end{aligned}$$

We have $\alpha \le 1$. Let $\overline{\boldsymbol{\lambda}}$ be defined by $\overline{\lambda}_i = -\frac{a_i}{\alpha W}$.

Proposition 11.6. *We have* $z_{\mathrm{LR}}(\overline{\boldsymbol{\lambda}}) = \frac{1}{\alpha W} \sum_{i=1}^n a_i$.

Proof. The definition of α ensures that $S_j(\overline{\boldsymbol{\lambda}}) = 0$ for all j in $[K]$. □

Using $a_1 = 3$, $a_2 = 5$, $a_3 = 5$, and $W = 7$, we have $\frac{\sum_{i=1}^n a_i}{W} = 13/7$, and $\frac{1}{\alpha W} \sum_{i=1}^n a_i = 13/5$, hence the bound provided by the linear relaxation is 2 and the bound provided by the Lagrangian relaxation is 3.

11.3.2 Facility location

11.3.3 Vehicle routing problems

11.3.4 Unit commitment

11.4 Benders decomposition

11.5 Exercises

Exercise 11.1. *Traveling Salesman Problem*

Consider the traveling salesman problem on a complete graph $G = (V, E)$ with edge costs $c(e)$ – See Chapter 16 for its definition.

1. Show that an optimal solution of the following MILP

$$\begin{array}{lll} \text{Min} & \sum_{e\in E} c(e)x_e & \\ & 0 \le x_e \le 1 & e \in E \\ & x_e \in \mathbb{Z} & e \in E \\ & \sum_{e\in\delta(v)} x_e = 2 & v \in V \\ & \sum_{e\in\delta(X)} x_e \ge 2 & X \subseteq V,\ X \ne \varnothing, V. \end{array} \tag{11.5}$$

2. This problem must be solved by cut generation. What is the pricing subproblem?

Solution. Min-cut. □

Lagrangian enables to improve the linear relaxation bounds.

3. Show that we can rewrite the constraints

$$\begin{array}{ll} 0 \le x_e \le 1 & e \in E \\ x_e \in \mathbb{Z} & e \in E \\ \sum_{e\in\delta(v)} x_e = 2 & v \in V \\ \sum_{e\in E[X]} x_e \le |X|-1 & X \subseteq V,\ X \ne \varnothing, V. \end{array}$$

4. We now give a special role to vertex 1. Show that the constraints of the MILP can be rewritten

$$\begin{array}{ll} 0 \le x_e \le 1 & e \in E \\ x_e \in \mathbb{Z} & e \in E \\ \sum_{e\in\delta(1)} x_e = 2 & \\ \sum_{e\in E[X]} x_e \le |X|-1 & X \subseteq \{2,\dots,n\},\ X \ne \varnothing \\ \sum_{e\in E} x_e = n & \\ \sum_{e\in\delta(v)} x_e = 2 & v \in \{2,\dots,n\}. \end{array}$$

   Solution. If we consider $X \subseteq V$ containing 1. Suppose that X contains a cycle. As $X\backslash\{1\}$ contains at most $|X|-2$ edges, this cycle contains all the vertices. Hence there is no edge of $\delta(X)$ in the solution, and contains $|X|$ edges. Hence $\sum_{e\in E} x_e = n$ implies that its complementary must contain $n-|X|$. This contradicts the constraint that enforces that X^c contains at most $n-|X|-1$ edges. □

5. Show that if we proceed to the Lagrangian relaxation of constraints $\sum_{e\in\delta(v)} x_e = 2$ for $v \in \{2,\dots,n\}$, we obtain a subproblem that can be solved in polynomial time.

   Solution. Subproblem is a spanning tree problem. □

6. What is the value of the Lagrangian relaxation?

   Solution. It is the value of the linear relaxation. Indeed, the

$$\begin{array}{ll} 0 \le x_e \le 1 & e \in E \\ \sum_{e\in E[X]} x_e \le |X|-1 & X \subseteq \{2,\dots,n\},\ X \ne \varnothing \\ \sum_{e\in E\backslash\delta(1)} x_e = n-2 & \end{array}$$

   is a perfect formulation of the spanning tree polytope (conforti p.154). Geoffrion's theorem enables to conclude. □

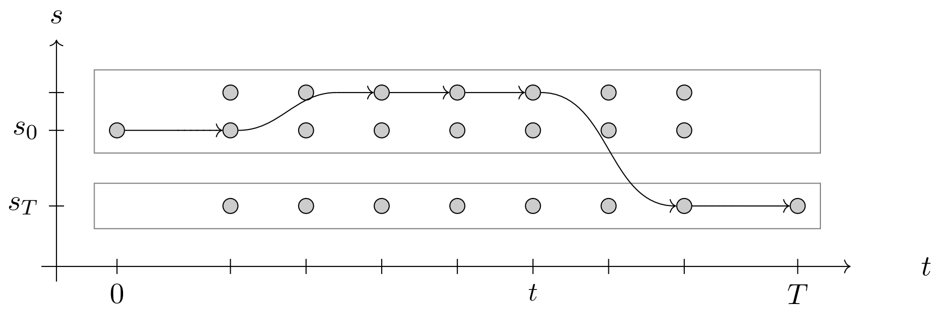


Figure 11.5: Un planning de production

△

Exercise 11.2. *Facility location*

Consider again the MILP formulation of the facility location problem.

$$\begin{array}{lll}
\text{Min} & \sum_{i\in\mathcal{F}} f_i y_i & \\
& \quad + \sum_{i\in\mathcal{F}}\sum_{j\in\mathcal{D}} c_{ij}x_{ij} & \\
\text{s.c.} & x_{ij} \leq y_i & i\in\mathcal{F},\, j\in\mathcal{D} \quad (1)\\
& \sum_{i\in\mathcal{F}} x_{ij} = 1 & j\in\mathcal{F} \quad (2)\\
& x_{ij}\in\{0,1\} & i\in\mathcal{F},\, j\in\mathcal{D} \quad (3)\\
& y_i\in\{0,1\} & i\in\mathcal{F}. \quad (4)
\end{array}$$

- Show that the dual function is easy to compute if we relax the constraints of Equation (2).
- Show that it is again easy to compute if we relax the constraints of Equation (1).

△

Exercise 11.3. *Unit commitment*

Un producteur d'électricité possède un ensemble U de centrales thermiques u. Il souhaite planifier sa production sur un horizon $\{0,\dots,T\}$. Chaque centrale u peut opérer à différents niveaux de production s dans l'ensemble fini S_u. À $t=0$, la centrale doit produire au niveau s_{u0}, et au temps T, au niveau s_{uT}. Seules certaines transitions $(s,s')\in S_u\times S_u$ sont admissibles. Entre deux instants t et $t+1$, une centrale peut soit rester au même niveau de production, et $s_{t+1}=s_t$, soit changer de niveau de production, et (s_t,s_{t+1}) doit dans ce cas être une transition admissible. À chaque niveau de production et transition est associé un coût de production $c_{(s,s')}$. Un planning de production P est une séquence $(s_t^P)_{t\in\{0,\dots T\}}$ telle que $s_t^P\in S_u$ pour tout t, les transitions (s_{t-1}^P,s_t^P) sont admissibles pour tout $t>1$. Un exemple de planning de production est illustré sur la Figure 11.5. À chaque niveau de transition est associée une consommation de gaz $w_{(s,s')}\in\mathbb{Z}_+$. La quantité de gaz consommée sur l'horizon par une centrale u ne doit pas excéder la capacité $W_u\in\mathbb{Z}_+$ du réservoir de la centrale. Un planning $P=(s_t^P)_{t\in\{0,\dots T\}}$ pour la centrale u est *admissible* si

$$\sum_{t=1}^{T} w_{(s_{t-1}^P,s_t^P)} \leq W_u.$$

A chaque instant t, la quantité totale d'électricité produite par l'ensemble des centrales doit être égale à la demande D_t du producteur. Le problème de *unit commitment* consiste à trouver les plannings de production des centrales de manière à minimiser le coût total.

- Justifier brièvement pourquoi le problème peut être modélisé par le programme linéaire en nombre entier suivant

$$\begin{aligned}
\min \quad & \sum_{a\in A} c_a x_a && && (11.6a)\\
\text{s.t.} \quad & \sum_{a\in A_t} s_a x_a = D_t, && \forall t \in \{1,\dots T-1\} && (11.6b)\\
& \sum_{a\in A_u} w_a x_a \leq W_u && \forall u \in U && (11.6c)\\
& \sum_{a\in\delta^-(v)} x_a = \sum_{a\in\delta^+(v)} x_a, && \forall v \in V\backslash(V_0 \cup V_t) && (11.6d)\\
& \sum_{a\in\delta^+(v_{0u})} x_a = \sum_{a\in\delta^-(v_{Tu})} x_a = 1 && \forall u \in U && (11.6e)\\
& x_a \in \{0,1\} && \forall a \in A. && (11.6f)
\end{aligned}$$

où l'on précisera en particulier la définition du graphe orienté acyclique $G = (V, A)$, ainsi que des ensembles A_t, A_u, V_0 et V_t, des sommets v_{0u} et v_{Tu}, de c_a, s_a et w_a, et des variables x_a.

Solution. Pour chaque centrale, un graphe (V_u, A_u) avec $V_u = S_u \times [T-1] \cup \{v_{0u}, v_{Tu}\}$, et A_u les transitions admissibles.

– c_a et w_a les coûts et consommations de gaz de chaque arc (transition).

– V_t l'ensemble des sommets de chaque graphe de la forme $(\cdot, t)$.

Contraintes :

– (11.6b) contraintes de demande,

– (11.6c) consommation de gaz

– (11.6d) contraintes de flot

– (11.6e) origine est destination

□

- En introduisant les variables duales réelles $\lambda = (\lambda_t)_{t\in\{1,\dots T-1\}}$ et les variables duales positives $\mu = (\mu_u)_{u\in U}$,

1. Procéder à la relaxation Lagrangienne des contraintes (11.6b) et (11.6c). On notera $\mathcal{G}(\lambda, \mu)$ la borne obtenue.
2. Montrer que $\mathcal{G}(\lambda, \mu)$ peut-être calculée en $O(\sum_u |A_u|)$ en résolvant, pour chaque centrale, un problème de graphe que l'on précisera.

Solution. 1. En notant le Lagrangien $\mathcal{L}(x, \lambda, \mu)$, on a $\mathcal{G}(\lambda, \mu) = \min_x \mathcal{L}(x, \lambda, \mu)$.

$$\begin{aligned}
\mathcal{G}(\lambda, \mu) = & \min_x \mathcal{L}(x, \lambda, \mu)\\
& \text{s.t. } (11.6d), (11.6e), (11.6f).
\end{aligned}$$

2. Problème de plus court chemin dans un graphe acyclique pour chaque centrale, résolu par programmation dynamique.

□

- En introduisant les variables duales réelles $\lambda = (\lambda_t)_{t\in\{1,\dots T-1\}}$,
  1. Procéder à la relaxation Lagrangienne des contraintes (11.6b). On notera $\mathcal{H}(\lambda)$ la borne ainsi obtenue.
  2. Montrer que $\mathcal{H}(\lambda)$ peut-être calculée en $O(\sum_u |A_u| W_u)$.

Solution. 1. En notant $\mathcal{L}'(x,\lambda,\mu)$ le Lagrangien, on a

$$\mathcal{H}(\lambda) = \min_x \mathcal{L}'(x,\lambda)$$
$$\text{s.t. (11.6c), (11.6d), (11.6e), (11.6f).}$$

2. Problème de plus court chemin sous contraintes, résolu par programmation dynamique.

□

- Laquelle des deux inégalités suivantes est toujours satisfaite : $\max_{\lambda,\mu} \mathcal{G}(\lambda,\mu) \leq \max_\lambda \mathcal{H}(\lambda)$ et $\max_\lambda \mathcal{H}(\lambda) \leq \max_{\lambda,\mu} \mathcal{G}(\lambda,\mu)$? Justifier.

Solution. $\max_{\lambda,\mu} \mathcal{G}(\lambda,\mu) \leq \max_\lambda \mathcal{H}(\lambda)$ (Immédiat par le théorème de Geoffrion.)

On peut aussi le prouver directement en utilisant le Lagrangien $\mathcal{L}(x,\lambda,\mu)$.

$$\begin{aligned}\max_\lambda \mathcal{H}(\lambda) &= \max_\lambda \min_x \max_\mu \mathcal{L}(x,\lambda,\mu) \\ &\geq \max_\lambda \max_\mu \min_x \mathcal{L}(x,\lambda,\mu) && \text{(dualité faible)} \\ &= \max_{\lambda,\mu} \mathcal{G}(\lambda,\mu).\end{aligned}$$

□

△

Exercise 11.4. We continue on the setting of the previous exercise. We now suppose that gas is delivered to units through a network of pipelines, and that units reservoirs can be refilled using pipelines during the horizon. Let S be a set of additional reservoirs. The capacity of each reservoir is W_s. A quantity b_{st} of gaz can be added to each reservoir at time t. Let $A \subseteq (U \cup S)^2$ be a set of pipelines. Gas can be sent in a pipeline in only one direction. Hence, $(U \cup S, A)$ is a digraph. At each timestep, a quantity κ_a can be sent in pipeline a. Given a pipeline between v_1 and v_2, gas sent from v_1 at t arrives at $t + \tau_a$ in v_2. Sending a unit of gas through pipeline a has cost γ_a. The unit commitment problem with pipelines consists in choosing production planning for units and gas deliveries using the pipelines so as to meet the demand at minimum cost.

- Give a MILP for the unit commitment problem of serving the demand at minimum cost.
- Which Lagrangian relaxation approach would you suggest for this problem?

△

Part III

Heuristics

Chapter 12

Metaheuristics

In this chapter, we introduce generic heuristic algorithms to solve the problem

$$\min_{x \in \mathcal{X}} c(x). \tag{12.1}$$

12.1 Generalities on heuristics

12.1.1 What is a heuristic?

A *heuristic* for an optimization problem $\mathcal{P}$ is an algorithm ϕ which, given and instance x of an optimization problem, returns a feasible solution in S_x (if S_x is non-empty). It should be contrasted with an *exact-algorithm*, which returns an optimal solution.

12.1.2 Evaluating a heuristic

When using a heuristic, there is *no guarantee on the quality of the solution returned*. It is therefore crucial to *evaluate experimentally the performance of a heuristic* before using it on an industrial problem. The experiments must be of course designed in a way to produce *reproducible results*

To evaluate the quality of a heuristic, one must first define the *goals that should be met* by the algorithm. The two main criteria are:

- the quality of the solution returned
- the time needed to return a good solution

but many other goals can be of interest.

- Ability to tackle large instances
- Easiness of implementation
- Robustness in terms of instances
- Flexibility and robustness to change in the modeling of the problem
- etc.

On practical problems, we generally do not know the optimal solution of the problem. *A good lower bound on the value of the optimal solution is the best way to evaluate the solution returned by the heuristics*. Indeed, the *gap* between the lower bound and the value returned by the algorithm is a good evaluation of the problem. General strategies to obtain a good lower bound are:

- solve to optimality an easier relaxed version of the problem, typically obtained by relaxing some complicating constraints
- solve a dual problem, not necessarily to optimality.

Second one must build a library of *testing instances*, which should contain a diverse panel of real-life and constructed instances. Heuristics generally depend on a wide range of "parameters", and one must build a panel of parameters, and launch the heuristic on each instance with each set of parameter.

Ideally, state of the art algorithms for the problem studied must also be tested on the same instances, and the results should be presented in a way that enables to compare the performance of the different algorithm.

Remark 12.1. One flaw of the literature on heuristics is that similar algorithms have been presented many times under different names using "nature inspired metaphors". When searching a good heuristic in the literature, avoiding papers with shinny metaphorical name is generally a good idea. And checking the quality of the experimental design evaluating the performance of a heuristic is essential. △

12.1.3 Families of heuristics

There are two kinds of heuristics:

- *specific heuristics*, or simple *heuristics*, which are problem dependent
- *metaheuristics* are general purpose algorithms that can be applied to a wide range of optimization problems. They can be viewed as general recipes to build good heuristics on some specific problems.

This lecture being general, we focus on metaheuristics.

A first way to classify heuristic is to distinguish *constructive heuristics* from *iterative heuristics*.

- Constructive heuristics, also called *greedy algorithms*, start from an empty solution, and complete at each step the solution with the "best" variable given the current partial solution until a full solution is built. Constructive heuristics are problem specific and generally ends in a "bad" local minimum.
- Iterative heuristics starts from a (population of) feasible solution(s) and transform it at each iteration using search operators.

Most metaheuristics are iterative heuristics. However, using a simple greedy algorithm is sometimes a good way to find the initial solution. In this chapter, we focus on iterative metaheuristic.

The key objective when designing an iterative metaheuristic is too find a way to find quickly a good solution without being "trapped" in a local minimum. One therefore seeks a *tradeoff between intensification and diversification*. The objective of intensification is to find quickly a good solution near the current one, with the risk of ending in a local minimum. On the contrary, diversification aims at leaving a local minimum by moving in another region of the solution space.

The three next sections introduce the main families of metaheuristics: single solution neighborhood based metaheuristics, population based metaheuristics, and hybrid metaheuristics.

12.2 Neighborhood based meta-heuristics

12.2.1 Neighborhood and local search

A *neighborhood* $\mathcal{N}(x)$ of a solution x is a set of solutions "near" x. Typically, it is obtained by modifying some variables in x.

Algorithm 11 Local search algorithm

```
Input: a feasible solution $x_0 \in \mathcal{X}$;
Output: a feasible solution $x \in \mathcal{X}$ satisfying $c(x) \leq c(x_0)$;
Initialize $x \leftarrow x_0$;
while $\min_{x' \in \mathcal{N}(x)} c(x') < c(x)$ do
    $x \leftarrow x''$ with $x'' \in \operatorname{argmin}_{x' \in \mathcal{N}(x)} c(x')$;
end while
return $x$;
```

The *local search* algorithm, stated in Algorithm 11, iteratively seeks in the neighborhood $\mathcal{N}(x)$ of the current solution x a solution that improves x until no such solution exists.

On standard neighborhood, the optimization problem $\min_{x' \in \mathcal{N}(x)} c(x')$ is solved by enumerating all the solutions in $\mathcal{N}(x)$. The update policy of Step 5 is called *best improvement*: the complete neighborhood is tested in order to find the best solution in the neighborhood. One alternative that is generally faster and leads to solutions of comparable quality is *first improvement*, and consists in stopping the enumeration of $\mathcal{N}(x)$ as soon as a solution x' with $c(x') < c(x)$ is found, and update with $x \leftarrow x'$.

A solution x is a *local minimum for neighborhood $\mathcal{N}$* if

$$c(x) = \min_{x' \in \mathcal{N}(x)} c(x).$$

Once a local search reaches a local minimum, the algorithm stops. Hence, a local search typically ends up in a local minimum that is not a global minimum.

12.2.2 From local search to metaheuristic: getting out off local minima

The metaheuristics we now introduce are modifications of the local search that allow the value $c(x)$ of the current solution to increase in order to get out of local minima.

Simulated annealing

Algorithm 12 is the *simulated annealing* algorithm. Its main idea is to randomly search the neighborhood space, accept improving solutions, and accept solution that increase the objection function with low probability.

Many parameters must be set:

- The initial temperature $T^{\max}$ is typically chosen in such a way that the probability of acceptance at the beginning is between 40% and 50%.

Algorithm 12 Simulated annealing

```
Input: initial solution $x_0$, initial temperature $T^{\max}$, parameter $\alpha$
Output: a solution $x \in \mathcal{X}$
Initialize: $x \leftarrow x_0$, $x^{\mathrm{b}} \leftarrow x_0$, $T \leftarrow T^{\max}$
repeat
  repeat
    if $c(x) < c(x^{\mathrm{b}})$ then $x^{\mathrm{b}} \leftarrow x$
    Generate a random neighbor $x'$ in $\mathcal{N}(x)$
    if $c(x') < c(x)$ then $x \leftarrow x'$
    else do $x \leftarrow x'$ with probability $e^{\frac{f(x')-f(x)}{T}}$
  until Equilibrium condition satisfied
  $T \leftarrow \alpha T$                                        temperature update
until Stopping criterion satisfied
return $x^{\mathrm{b}}$
```

- Parameter α is typically chosen between 0 and 0.99. Alternative temperature updates rules can be used. Typically, T slowly decreases to 0.

- Typical choices for the equilibrium condition is to update temperature after $y \cdot |N(s)|$ iterations. The larger y, the better the solution and the higher the computational cost. Another alternative is too decrease the objective when objectives improves, and increase it after a number of iterations without accepting the move.

- The stopping criterion is generally that a given number of iterations have been performed, or that a small temperature has been reached.

It is generally a good idea to run a local search after a simulated annealing to guarantee that the solution obtained is a local minimum.

Taboo

One characteristic of the previous methods is that they are *memory-less*: only the current state is used in the search. *Taboo search*, on the contrary, uses memory to diversify, and is stated in Algorithm 13.

Algorithm 13 Taboo search

```
Input: feasible solution $x_0 \in S$
Initialize: $T \leftarrow \emptyset$, $x \leftarrow x_0$, $x^{\mathrm{b}} \leftarrow x_0$
repeat
  if $\min_{x' \in \mathcal{N}(x)} c(x') < c(x_b)$ then          aspiration criterion
    $x \leftarrow \min_{x' \in \mathcal{N}(x)} c(x)$;
    $x^{\mathrm{b}} \leftarrow x$;
  else $x \leftarrow \operatorname{argmin}_{x' \in \mathcal{N}(x') \setminus T} c(x')$;          non-taboo move
  Update $T$;
until stopping criterion holds
return $x^{\mathrm{b}}$
```

The main idea of taboo search is to maintain a list T of taboo solutions that correspond to the last solutions visited. The objective is to force diversification by forbidding solutions previously tested. Practically, T *must not be implemented as the list of the n previous solutions visited*, but as the set of solutions satisfying certain conditions. Typically, when solutions are vectors and neighborhood consists in changing some components of the vector, the taboo lists forbids to change again the components modified during the n previous moves. The aspiration criterion allows to accept taboo solutions that improves the best solution known. Other aspiration criterion can be used. Of course, it is not necessary to solve the minimization problems to optimality, and some randomness can be introduced.

The taboo list T is typically a "short term memory": it contains information on the last solutions visited. Advanced taboo techniques also use.

- A *medium term memory* for *intensification*. This medium term memory contains statistics of attributed of the best solution founds during the previous part of the search (on a longer term than the taboo list). If an intensification criterion holds, search is intensified around the current solution using a local search where solutions are generated randomly using a distribution that bias the search toward solutions having the same attributes as the best solutions in the medium term memory.
- A *long term memory* for *diversification*. This memory stores statistics on the attributes of all the solutions considered during the search. When a diversification criterion is met, this memory is used to move the current solution to a region of the search space that has not yet been explored.

12.2.3 Practical aspects

A metaheuristic will be efficient if it can explore efficiently the solution space. The two elements that are critical for the performance are

- the quality of the neighborhoods: a solution may be a local minimum for one neighborhood but not for another.
- the "speed" at which the algorithm moves from one solution to another, which requires a careful implementation.

Section 12.2.4 details how to build neighborhoods of good quality. We now elaborate on how to make a heuristics that moves fast, and how to handle constraints.

Moving fast

Two aspects are critical to build a heuristic that moves fast. We illustrate them on the facility location problem introduced on page 18.

First, the encoding of the solution must

- avoid redundancy: there should not be several encoded solutions that correspond to the same solution (avoid symmetry if possible)
- be compact
- allow fast computations

Computing in a preprocessing quantities that will be needed many times is a good idea.

Typically, on the facility location problem, a solution is described by the set of facilities opened. It is not necessary to store the client-facility association, as, given a set of opened facilities, it is immediate to find which facility will serve which client. This computation will be faster if, in a preprocessing, the ordered list of nearest facilities have been computed for each client.

Second, the evaluation of a solution must be incremental. Computing from scratch the cost of a solution takes time. Computing the costs of a neighbor x' of the current solution x is generally much faster. For instance, a typical neighborhood for the facility location problem consists in opening a facility. When such a move is done, only the costs the clients that are near the new facility must be updated.

Given the speed of the current computers, a heuristic is considered "fast" if it moves one million times every second. This is generally achievable for academic problems that are relatively "pure". On industrial ones, it might be one or two order of magnitude slower.

Handling constraints

Several strategies are used to handle constraints. The most common ones are the following.

- *reject strategies* consists in using only feasible solutions as the current solution
- *penalizing strategies* include a penalty for violating the constraint in the objective function. Typically, the penalty is a constant times a distance to feasibility. Like temperature in the simulated annealing, these penalties can be static, dynamic, or adaptive.
- *repairing strategies* that rebuild a feasible solution from an infeasible can be useful.

12.2.4 Very large and variable neighborhood search

Large neighborhoods have a clear advantage over using small neighborhoods: a local minimum for a small neighborhood may not be a local minimum for a large neighborhood. And a clear drawback: enumerating the solution of a large neighborhood may take time. This section details techniques to improve metaheuristics using very large neighborhoods.

Using mixed integer programming to build large neighborhood

Metaheuristics need to optimize over neighborhoods $\mathcal{N}(x)$. For small neighborhoods, this can be done by complete enumeration. For large neighborhoods, this requires a practically efficient algorithm to optimize over the neighborhood. Exact algorithms such as dynamic programming, shortest path problems, flows, linear programming or integer programming can typically be used, as well as practically efficient heuristics to search the neighborhood.

A typical way of building large neighborhoods is the following. If there exists a MIP formulation that is non-tractable for large instances of interest, but practically well solved for small instances, a typical strategy to build a neighborhood is to fix a given proportion of the solution, and solve the small resulting instance using a MIP. For instance, for a time-tabling problem on a horizon of one month, using a small MIPs on rolling horizons of three days is generally a good idea to build a good neighborhood.

Variable neighborhood search

Using many neighborhoods along a metaheuristic is generally a good idea. At the beginning, the current solution is easy to improve, and small neighborhoods generally enable to improve it faster than large ones. At the end of the algorithm, the current solution is generally a local minimum for small neighborhoods, and it is therefore better to use large neighborhoods.

A technique that performs generally well is to select randomly the neighborhood using a distribution that bias the selection toward the neighborhoods that were the "most efficient" on a medium term (e.g. last 1000 iterations).

Machine learning tricks 😎

Use machine learning to identify the properties of good solutions.

12.3 Population based heuristics

Genetic or *evolutionary* and *ant colonies* algorithms are the most used population based heuristics. Algorithm 14 introduces a prototype of evolutionary algorithm.

Algorithm 14 Evolutionary algorithms

```
Initialize with a population $P$ of feasible solutions
repeat
    Generate a new population $P'$ from $P$
    Update $P$ by selecting a population of desired size in $P \cup P'$
until Stopping criterion satisfied
return best solution(s) found;
```

Population generation in evolutionary (or genetic) algorithms rely on *cross-over* operators $g : \mathcal{X} \times \mathcal{X} \rightarrow \mathcal{X}$, which given two solutions x_1 and x_2 generate a third one $g(x_1, x_2)$ by combining these two solutions. For instance, on the facility location problem, one can partition the set of facilities into F_a and F_b, and building $g(x_1, x_2)$ by opening the opened facilities of x_1 in F_a and the opened facilities of x_2 in F_b? A new population P' is generally produced by randomly selecting two elements x_1 and X_2 in P and adding $g(x_1, x_2)$ to P. This operation is repeated until P' has reached the desired size.

Typically, the P solutions of minimum cost are selected from $P \cup P'$ in the population selection step.

12.4 Hybrid heuristics 😎

12.4.1 Matheuristics

Part IV

Applications

Chapter 13

Bin packing

Chapter 14

Facility location

14.1 Exercises

Exercise 14.1. On reprend le problème de positionnement d'entrepôts vu en cours, mais avec cette fois une contrainte supplémentaire: chaque entrepôt ne peut desservir qu'un nombre limité de clients. Modéliser le problème suivant comme un problème linéaire en nombres entiers.

Données: Un ensemble fini de clients $\mathcal{D}$, un ensemble fini d'entrepôts potentiels $\mathcal{F}$, un coût fixe $f_i \in \mathbb{R}_+$ d'ouverture et une capacité K_i pour chaque entrepôt $i \in \mathcal{F}$, et un coût de service $c_{ij} \in \mathbb{R}_+$ pour chaque $i \in \mathcal{F}$ et $j \in \mathcal{D}$.

Demande: Trouver un sous-ensemble $X \subseteq \mathcal{F}$ (dits *entrepôts ouverts*) et une affectation $\sigma : \mathcal{D} \to X$ des clients aux entrepôts ouverts, tel que pour tout i, l'entrepôt i ne desserve pas plus de K_i clients, de façon à ce que la quantité

$$\sum_{i\in X} f_i + \sum_{j\in\mathcal{D}} c_{\sigma(j)j}$$

soit minimale.

Solution.

$$\begin{array}{lll}
\text{Min} & \displaystyle\sum_{i\in\mathcal{F}} f_i y_i + \sum_{i\in\mathcal{F}}\sum_{j\in\mathcal{D}} c_{ij}x_{ij} & \\
\text{s.c.} & \displaystyle\sum_{j\in\mathcal{D}} x_{ij} \le K_i y_i & i \in \mathcal{F} \\
& \displaystyle\sum_{i\in\mathcal{F}} x_{ij} = 1 & j \in \mathcal{D} \\
& x_{ij} \in \{0,1\} & i \in \mathcal{F},\, j \in \mathcal{D} \\
& y_i \in \{0,1\} & i \in \mathcal{F}.
\end{array}$$

□

△

Chapter 15

Network design

Chapter 16

Routing

Chapter 17

Scheduling

According to `Wikipedia`, a “schedule is a time management tool consisting of a list of times at which events are to occur, or an order in which they are to occur.” Optimization of schedules has therefore countless applications, among which

- project management (see Section 17.1)
- production scheduling (Section 2 onwards)
- manpower scheduling (intersects production scheduling, but with specificities due to working rules)
- transport scheduling (see chapter on routing)
- computer scheduling (how to schedule threads, processes, and data flows)
- etc.

In this lecture, we will focus on the two first applications.

17.1 Minimum duration of a project

Let J be a set of tasks j representing a project. Each task has a processing time p_j. Furthermore, we have some precedence constraints: some tasks must be finished before others can be started. The objective is to find the minimum duration of the project.

1. Explain why this problem can be solved as a longest s-t path problem on a digraph $D = (V, A)$ where $V = J \cup \{s, t\}$, where s is a source vertex, t is a sink vertex, and A is a set of arcs which should be described, as well as the arc lengths.

   Solution. There is an arc (j, j') is task j must be performed before task j', as well as arc (s, j) and (j, t) for all t. The length of arc (s, j) is 0, the length of an arc $(j, \cdot)$ is p_j. □

2. Which algorithm should be used to solve that problem?

   Solution. The most efficient algorithm to compute a longest path is an acyclic digraph is dynamic programming. □

3. How can you identify the critical tasks, that is, the tasks for which there is no margin on their ending time: if they are just a little late, then the project will be late.

Tasks	Description	Processing time	Preceding tasks
A	foundations	6	–
B	wall constructions	10	A
C	exterior plumbing	4	B
D	interior plumbing	5	A
E	electricity	7	A
F	roof	6	B
G	exterior painting	16	B,C,F
H	panels	8	D,E
I	floor	4	D,E
J	interior	11	H,I

Table 17.1: House construction project

Solution. The tasks on a critical path, i.e., a longest s-t path. □

4. What is the minimum time on the house construction project on Table 17.1? Identify critical tasks.

   Solution. Length: 38. Critical tasks: A, B, F, G □

17.2 Production scheduling terminology

Production scheduling is the problem of affecting jobs to machines. Jobs and machines are generic terms: jobs can be any kind of tasks to be accomplished, and machine can be anything that is required for some task.

Jobs are indexed by j and machines by i. There are m machines and n jobs. If explicitly mentioned, a job j must be operated between its *release date* r_j and its *due date* d_j. Job j has sometimes a *weight*, denoted by w_j. The *processing time* of job j on machine i is denoted by p_{ij}: it is the time needed to operate j on i. The processing time of j may not depend on the machine, and we denote it in this case p_j.

On complicated problems, a job may require several *processing steps* on different machines. Pair (i, j) then refers to the processing step or operation of job j on machine i, and its duration is again denoted by p_{ij}.

In the literature, a scheduling problem is described by a triplet $\alpha|\beta|\gamma$ where α is the *machine environment*, β contains additional constraints, and γ indicated the *objective to minimize*.

The most frequent machine environment are

- *Single machine* (1)
- *Parallel machines*: identical machines in parallel (Pm), parallel machines with different speeds (Qm), i.e., $\frac{p_{ij}}{p_{i'j}}$ does not depend on j, or unrelated machines in parallel (Rm).
- *Open shop* (Om) each job must be processed on each of the m machines in any order.
- *Flow shop* (Fm): there are m machine in series, and each job has to be processes on machines $1, \dots, m$ in this order. Generally, the order in which jobs are processed is identical on all machines. If one job can pass another, β contains the entry *prmu*

- *Job shop* (Jm): each job must be processes on each machine, and the order in which it must be processed is fixed but job dependent.
- *Flexible flow shop* (FFc) and *flexible job shop* (FJc) are analogues of flow shop and job shop, the main difference being that, if there are c types of machines, there are several machines of each type that are available and can work in parallel.

Parameter β can contain

- *Release dates* (r_j)
- *Preemptions* ($prmp$). Jobs are generally assumed to be completed at once on a machine. Parameter $prmp$ indicates that, on the contrary, a job can be stopped and restarted.
- *Precedence constraints* ($prec$): some jobs must be performed before others. Such constraints are easily modeled using an acyclic digraph.
- etc.

We denote by C_{ij} the completion time of job j on i; and by C_j the *completion time* of j on the last machine it visits. The *lateness* L_j of a job is

$$L_j = C_j - D_j$$

and its *tardiness* is $T_j = \max(L_j, 0)$.

Frequent minimized objective γ include

- Makespan ($C_{\max}$), defined as $C_{\max} = \max_j(C_j)$,
- Total weigthed completion time ($\sum_j w_j C_j$)
- Maximum lateness ($L_{\max}$), defined as $L_{\max} = \max_j(L_j)$
- Total weighted tardiness ($\sum_j w_j T_j$)

17.3 Single machine problems

17.3.1 Minimum weighted completion time

Consider the problem $1||\sum_j w_j C_j$: there is no release date, each job has processing time p_j and weight w_j.

5. Show that processing the jobs in decreasing $\frac{w_j}{p_j}$ order gives an optimal solution.

Solution. Consider a solution such that j' is right after j, and $\frac{w_j}{p_j} < \frac{w_j'}{p_j'}$. Consider the solution obtained by exchanging j and j'. Then the difference of the cost of the new solution minus the one of the solution is between the two solutions $w_j p_{j'} - w_{j'} p_j < 0$, and the solution is not optimal. Hence an optimal solution is obtained by processing the jobs in decreasing $\frac{w_j}{p_j}$ order. $\square$

17.3.2 Precedence constraints – dynamic programming

We consider the problem $1|prec|\max_j T_j$, where each job j has a given due date d_j, tardiness T_j is defined in Section 17.2 and precedence constraints indicated by an acyclic digraph $D = ([n], A)$. Let h_j be the mapping $t \mapsto \max(t - d_j, 0)$, such that $T_j = h_j(C_j)$.

6. Show that there exists an optimal schedule such that, for each integer $k < n$, we have

$$h_{j_k}\Big(\sum_{j' \in J^c} p_{j'}\Big) = \min_{j \in J^c \colon \delta^+(j) \subseteq J} h_j\Big(\sum_{j' \in J^c} p_{j'}\Big). \tag{17.1}$$

where j_k denotes the kth jobs operated in the schedule, J denotes the jobs after k in the schedule, and J^c its complementary: $J^c = [n]\backslash J$.

Solution. Consider an optimal schedule $s = j_1, \dots, j_n$. Let k be the largest integer such that (17.1) is not satisfied in s. We are going to build an optimal schedule $\overline{s}$ such that the largest integer $\overline{k}$ such that (17.1) is not satisfied in $\overline{s}$ is such that $\overline{k} < k$. Iterating this procedure at most n times gives an optimal schedule satisfying the desired property. We now explain how to build schedule $\overline{s}$. We denote by C_j and T_j the completion time and tardiness for schedule s. Then schedule $\overline{s}$ defined by $j_1, \dots, j_{\overline{k}-1}, j_{\overline{k}+1}, \dots, j_k, j_{\overline{k}}, j_{k+1}, \dots, j_n$ is a schedule that respects the precedence constraints. We denotes by $\overline{C_j}$ and $\overline{T_j}$ the completion time and tardiness of job j under this new schedule.

- If $\ell < \overline{k}$ or $\ell > k$, we have $\overline{T}_{j_\ell} = T_{j_\ell}$
- If $\overline{k} < \ell \leq k$, we have $\overline{T}_{j_\ell} \leq T_{j_\ell}$ by monotonicity of h_ℓ
- $\overline{T}_{j_{\overline{k}}} \leq T_{j_k}$ by hypothesis.

Hence $\max_j \overline{T}_j \leq \max_j T_j$, and $\overline{s}$ satisfies the desired property. □

7. Give an algorithm solving $1|prec|\max_j T_j$ to optimality.

Solution. Initialize $J = \emptyset$. For k from n to 1, do

- select j_k in the argmin of $\displaystyle\min_{j \in J_c \colon \delta^+(j) \subseteq J} h_j\Big(\sum_{j' \in J^c} p_{j'}\Big)$,
- $J \leftarrow J \cup \{j_k\}$
- $k \leftarrow j - 1$

We can then prove that the algorithm returns the optimal result by induction on the number of jobs. If there is one job, the result is trivial. Suppose now that there is $n+1$ jobs. Let $\tilde{s} = \tilde{j}_1, \dots, \tilde{j}_{n+1}$ be the schedule produce by that algorithm. Let $s = j_1, \dots, j_n$ be an arbitrary schedule, and k be such that $j_k = \tilde{j}_{n+1}$. Using the same proof as in the previous question, we obtain that $\overline{s} = j_1, \dots, j_{k-1}, \dots, j_{n+1}, j_k$ has a smaller maximum tardiness than s. Furthermore by, induction hypothesis, $\tilde{j}_1, \dots, \tilde{j}_n$ is an optimal scheduling of $\{j_1, \dots, \tilde{j}_n\}$, and hence $\tilde{s}$ has smaller maximum tardiness that $\overline{s}$. This gives the induction hypothesis and concludes the proof. □

17.3.3 Precedence constraints – mathematical programming

We consider now $1|prec|\sum_j w_j T_j$ with $p_j \in \mathbb{R}_+$, where we recall that T_j is the tardiness.

8. Give a MILP modeling this problem (indication: "big M" constraints may be helpful).

Solution. Let $M = \sum_{j\in[n]} p_j$ We use continuous variables T_j and C_j, and binary variables x_{jk} indicating if task j is operated before task k.

$$\begin{aligned}
\min_{C_j} \quad & \sum_{j\in[n]} w_j T_j && && (17.2a)\\
\text{s.c.} \quad & C_j \geq C_k + p_j - M x_{jk}, && \forall j < k && (17.2b)\\
& C_k \geq C_j + p_k - M(1 - x_{jk}), && \forall j < k && (17.2c)\\
& C_k \geq C_j + p_k && \text{for all } (j,k) \in A && (17.2d)\\
& C_j - p_j \geq 0 && \text{for all } j \text{ in } [n] && (17.2e)\\
& T_j \geq C_j - d_j && && (17.2f)\\
& t_j \geq 0 && && (17.2g)\\
& C_j \geq 0 && && (17.2h)\\
& x_{jk} \in \{0,1\} && && (17.2i)
\end{aligned}$$

□

We now consider the simpler case where $p_j \in \mathbb{Z}_+$ for all j. Let $T = \sum_j p_j$ Consider the following MILP.

$$\begin{aligned}
\min \quad & \sum_{j\in J} w_j \sum_{t\in[T]} \max(t - d_j, 0) x_{jt} && && (17.3a)\\
\text{s.t.} \quad & \sum_{t=0}^{T} x_{jt} = 1 && \forall j \in J && (17.3b)\\
& \sum_{t'=0}^{t+p_k} x_{kt'} \leq \sum_{t'=0}^{t} x_{jt'} && \forall (j,k) \in A, \forall t \in [T - p_k] && (17.3c)\\
& \sum_{j\in J} \sum_{t'=t}^{t+p_j-1} x_{jt'} \leq 1 && \forall t \in [T] && (17.3d)\\
& x_{jt} \in \{0,1\} && \forall j \in J, \forall t \in \{0,\dots,T\} && (17.3e)
\end{aligned}$$

9. Explain the meaning of the binary variable x_{jt} and of the different constraints.

Solution. Binary variable x_{jt} indicates if $C_j = t$. □

10. An interval matrix is a matrix such that each line is of the form $(0,\dots,0,1,\dots,1,0,\dots,0)$. show that an interval matrix is totally unimodular.

Solution. Subtract to each column (except the first) the previous column. The matrix obtained has at most one 1 and one -1 in its non-zero terms. □

11. Show that the matrix defined by constraints (17.3d) is totally unimodular

    Solution. It is an interval matrix, and therefore totally unimodular. □

12. Explain why the subproblem of the Lagrangian Relaxation of constraints (17.3b) and (17.3c) can be solved in polynomial time.

    Solution. The matrix of the pricing subproblem is totally unimodular, hence an optimal solution of its linear relaxation gives an optimal solution of the subproblem. Such a solution can be computed in polynomial time using the ellipsoid or interior point algorithms. □

17.4 Easy multiple machines problems

17.4.1 Operating scheduled jobs with a minimum number of machines

Suppose that we have a set of n jobs with fixed starting and ending times $d_j = r_j + p_j$ that have to be processed by identical machines.

13. Show that the minimum number of machines required to operate all these jobs (and the schedules of these machines) can be computed in polynomial time.

    Solution. We identify the set of jobs by $[n]$. Let D be the digraph with vertex set $([n] \times \{0,1\}) \cup \{s,t\}$, where s and t are respectively a source and a sink. Let A be the arc set obtained by adding, for all j in $[n]$

    - an arc $a = \big(s,(j,0)\big)$ for all j in $[n]$ with lower capacity $\ell_a = 0$, upper capacity $u_a = +\infty$, and cost $c_a = 0$,
    - an arc $a = \big((j,0),(j,1)\big)$ with $\ell_a = 1$, $u_a = 1$, and $c_a = 0$,
    - an arc $a = \big((j,1),t\big)$ with $\ell_a = 0$ $u_a = +\infty$, and $c_a = 0$,
    - arc $a = \big((j,1),(j',0)\big)$ with $\ell_a = 0$ $u_a = +\infty$, and $c_a = 0$ for all j' such that $d_j \leq r_{j'}$,

    as well as an arc $a = (t,s)$ with $\ell_a = 0$, $u_a = \infty$, and $c_a = 1$. A minimum cost circulation with lower capacity (l_a), upper capacity u_a, and cost (c_a) gives an optimal solution – we recall that a circulation is a $\boldsymbol{b}$-flow with $\boldsymbol{b} = \mathbf{0}$. Indeed, as capacity are integers, such a circulation can be decomposed into cycles, and $\ell_a = 1$, $u_a = 1$ for $a = \big((j,0),(j,1)\big)$ of the form $s,(j_1,0),(j_1,1),\ldots,(j_k,0),(j_k,1),t$ ensures that this decomposition partitions the job set: given a job j, $(j,0)$ and $(j,1)$ is on a unique cycle. Furthermore, the definition of arcs $\big((j,1),(j',0)\big)$ ensure that there is a bijection between sequences of jobs that can be operated by one machine and cycles.

    Such a circulation can be found in polynomial time using a minimum cost flow algorithm. □

17.4.2 Minimum makespan with preemption allowed

Consider the problem $Pm|prmp|C_{\max}$, where m jobs are processed on unrelated machines with processing times p_j, each job has release date r_j.

14. Let C be an upper bound on the value of an instance of $Pm|prmp|C_{\max}$. Give a simple algorithm to rebuild a solution with value C.

Solution. Let m_j and r_j be the quotient and the rest of the division of $\sum_{j'=1}^{j-1} p_j$ by $C_{\max}$, and m'_j and r'_j the quotient of the division of $\sum_{j'=1}^{j} p_j$ by $C_{\max}$. If $m_j = m'_j$, we schedule j on m_j between r_j and r'_j. Otherwise, we schedule j on j between r_j and C, and on m'_j between 0 and r'_j. The solution is a schedule (no two machines on the same job at the same time) as C is larger than p_j. □

15. Prove that the following linear program

$$\min \; z \tag{17.4a}$$

$$\text{s.t.} \; \sum_{i=1}^{n} x_{ij} = p_j, \quad \forall j \in [n], \tag{17.4b}$$

$$\sum_{i=1}^{n} x_{ij} \leq z, \quad \forall j \in [n], \tag{17.4c}$$

$$\sum_{j=1}^{m} x_{ij} \leq z, \quad \forall i \in [m], \tag{17.4d}$$

$$x_{ij} \geq 0, z \geq 0 \quad \forall i \in [m], \forall j \in [n]. \tag{17.4e}$$

gives the optimal value of $Pm|prmp|C_{\max}$, and explain how to reconstruct an optimal solution from this value.

Solution. It is immediate the a solution of $Pm|prmp|C_{\max}$ gives a solution of the linear program. Indeed, it suffices to define x_{ij} as the time spent by machine i in job j.

Conversely, given the optimal value z^* of the linear program, and using the same technique as in the previous question, we rebuild a solution of $Pm|prmp|C_{\max}$ with value z: constraint (17.4c) ensures that a job is not scheduled at the same time on two machines, and constraint (17.4d) that we do not use more machine than we have. □

17.5 Branch and Bound and heuristics for the job shop problem

To conclude, we introduce a powerful tool that enables to solve many scheduling problems, the *disjunctive graphs*. We illustrate it on the job shop problem $Jm||C_{\max}$.

This graph contains

- A vertex for each processing step of each job, as well as a source vertex s and a sink vertex t
- An arc between successive steps of each job.
- An $\big((i,j),(i,j')\big)$ edge between processing steps of different jobs on using the same machine.

An orientation of the edges of the disjunctive graph is an orientation (turning each edge into an arc) such that the resulting digraph is acyclic.

16. Explain why there is bijection between acyclic orientations of the edges of the disjunctive graph and solutions of the scheduling problem.

Solution. Orienting an arc enables to indicate if a processing step is performed before another on a given machine. □

17. Given an orientation, how do we efficiently compute the makespan $C_{\max}$ of the corresponding scheduling.

    Solution. Longest s-t path. □

18. Explain how the disjunctive graph can be used to build a Branch and Bound algorithm.

    Solution. Branch by orienting the edges. A bound is obtained by searching the longest path in the digraph where no-oriented edges are removed. □

19. Propose a metaheuristic to solve the problem:

    (a) How is a solution encoded

    Solution. Orientation of the arcs □

    (b) Propose several neighborhoods

    Solution. Reverse orientation of an arc. Reverse orientation of the disjunctive arcs of a critical path. □

20. Based on the disjunctive graph, propose a MILP modeling the problem.

    Solution. For each i, j, and j', let $x_{ijj'}$ be a binary indicating if step (i,j) is performed before (i,j'). Let $M = \sum_i \sum_j p_{ij}$ The following MILP solves the problem.

$$\begin{aligned}
\min\ & C_{\max} && && \text{(17.5a)}\\
\text{s.t.}\ & C_{ij} \le C_{\max} && \forall i \in [m], j \in [n] && \text{(17.5b)}\\
& x_{ijj'} = 1 - x_{ij'j} && \forall i \in [m], \forall j \in [n], \forall j' \in [n] && \text{(17.5c)}\\
& C_{i'j} \ge C_{ij} + p_{i'j} && \forall \big((i,j),(i',j)\big) \in A && \text{(17.5d)}\\
& C_{ij'} \ge C_{ij} + p_{ij'} - M x_{ij'j} && \forall \big((i,j),(i,j')\big) \in E && \text{(17.5e)}\\
& x_{ijj'} \in \{0,1\} && \forall \big((i,j),(i,j')\big) \in E && \text{(17.5f)}\\
& C_{ij} \ge 0 && \forall i \in [m], j \in [n] && \text{(17.5g)}
\end{aligned}$$

□

Part V

Previous exams

Chapter 18

Exam 2018

- Le contrôle est noté sur 23. La note sera laissée sur 20, et éventuellement remontée en fonction des résultats globaux.
- Polycopié de cours autorisé. Tout autre document interdit. Objets electroniques interdits.
- Les questions marquées d'une * sont un peu plus difficiles.

18.1 Creuser une rivière artificielle à moindre coût – 3 pts

Deux bassins s et t doivent être reliés par une rivière artificielle dans une région vallonnée. Cette rivière doit permettre de faire passer l'eau de s à t. Le graphe orienté de la Figure 18.1 indique les possibilités : un arc est une portion de rivière possible et l'orientation de l'arc indique le sens d'écoulement de l'eau sur cette portion. Le coût attaché à l'arc est le coût d'ouverture de cette portion. Donner la rivière de coût minimum reliant s à t. Justifier la réponse de façon détaillée – une partie des points sera accordée uniquement si l'algorithme le plus efficace est utilisé.

18.2 Identification des tâches critiques et calcul des marges d'un projet de construction d'un pavillon – 4 pts

Dans le tableau suivant sont indiquées les tâches à effectuer pour réaliser un pavillon, leurs durées, et leurs identifiants. La dernière colonne indique pour, chaque tâche, les tâches qu'il faut avoir complètement achevées pour pouvoir la débuter. S'il est indiqué "Début", c'est qu'aucune tâche particulière doit être effectuée au préalable.

Identifiant	Tâche	Durée (en jours)	Tâches devant être achevées
A	Déblaiement	5	Début
B	Baraquements	4	Début
C	Fondations	10	A,B
D	Murs	22	C
E	Canalisations extérieures	3	Début
F	Charpente	5	D
G	Toit	6	D,F
H	Canalisations intérieures	4	E,D
I	Electricité	3	D
J	Peinture	3	I,H,G

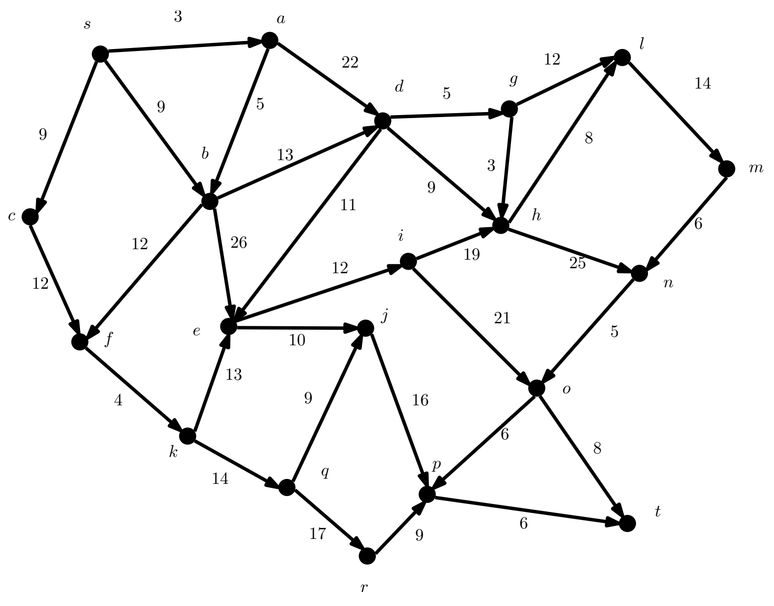


Figure 18.1: Région vallonée : il faut creuser une rivière de s à t au moindre coût

On considère que l'instant $t = 0$ correspond au début du chantier.

1. Pour toute tâche X, déterminer η_X l'instant au plus tôt auquel on peut commencer cette tâche. Justifier la réponse. (1 pt)

2. En déduire la durée minimale de ce projet. (1 pt)

3. Pour toute tâche X, déterminer π_X l'instant au plus tard auquel on peut commencer cette tâche. Justifier la réponse. (1 pt)

4. En déduire les tâches critiques et les marges pour chaque tâche. (1 pt)

18.3 Ordonnancement de voitures sur une chaîne de montage, d'après Estellon et al. – 10 pts

On s'intéresse à un problème central de l'industrie automobile : celui de la détermination de la séquence des voitures sur une chaîne de montage. Une voiture est caractérisée par une *couleur* et un sous-ensemble d'*options*. L'ensemble des couleurs est noté $\mathcal{C}$ et l'ensemble des options est noté $\mathcal{O}$. Une voiture est donc caractérisée par un couple (c, O) avec $c \in \mathcal{C}$ et $O \subseteq \mathcal{O}$. On appelle *classe* l'ensemble des voitures de même couple (c, O).

On suppose que les voitures produites par l'usine se partitionnent en un ensemble de classes $\mathcal{K}$. On note (c_k, O_k) le couple qui caractérise la classe k. Etant donné un ensemble de n voitures à produire, avec n_k voitures par classe k ($\sum_{k\in\mathcal{K}} n_k = n$), il s'agit de fixer la séquence des voitures sur la chaîne de montage, en satisfaisant la contrainte donnée au paragraphe suivant, et en minimisant le coût dont le calcul est précisé plus bas.

La contrainte que doit satisfaire la séquence de voitures est la suivante : il doit y avoir au plus r voitures de même couleur placées consécutivement sur la chaîne. En effet, si trop de voitures consécutives sont peintes de la même couleur, il est plus difficile d'évaluer la qualité de la peinture.

Le coût se décompose quant à lui en deux termes. L'un des termes est le coût de changement de couleurs. A chaque fois qu'il y a changement de couleur sur la chaîne de montage, une pénalité $\rho \geq 0$ est ajoutée. On dit qu'il y a *changement de couleur* si deux voitures consécutives doivent être peintes de couleurs différentes. Noter que si $n > r$, on est certain que la pénalité ρ sera payée au moins une fois pour toute solution réalisable. L'autre terme est le coût lié au déséquilibre des charges de travail. A toute option o est associé un couple d'entiers (p_o, q_o) qui signifie la chose suivante : Sur toute suite de q_o voitures consécutives, il serait bien qu'au plus p_o aient l'option o. Une pénalité est ajoutée en cas de dépassement. Cette pénalité est proportionnelle au dépassement : Si l'on a x voitures qui ont l'option o sur la suite de q_o voitures consécutives, avec $x > p_o$, une pénalité égale à $(x - p_o)\gamma_o$, avec $\gamma_o \geq 0$, est ajoutée. Ce calcul doit être fait pour toute suite de q_o voitures consécutives (toute "fenêtre").

Par exemple, supposons que l'on a deux options X et Y, deux couleurs `bleu` et `rouge` et 3 classes numérotées $1, 2, 3$ décrites par le tableau suivant, un 1 indiquant que l'option est présente et 0 qu'elle ne l'est pas.

Classe	1	2	3
Couleur	`bleu`	`bleu`	`rouge`
Option X	1	0	1
Option Y	0	1	1

Supposons de plus que les paramètres sont $r = 3$, $(p_{\mathrm{X}}, q_{\mathrm{X}}) = (2, 5)$, $(p_{\mathrm{Y}}, q_{\mathrm{Y}}) = (3, 4)$ et que $n_1 = 4$, $n_2 = n_3 = 3$. Le coût de la séquence (pour chaque voiture, on n'indique que sa classe)

$$1113222331$$

est égal à

$$4 \times \rho + (2 + 1 + 0 + 0 + 0 + 1) \times \gamma_{\mathrm{X}} + (0 + 0 + 0 + 1 + 1 + 1 + 0) \times \gamma_{\mathrm{Y}} = 4\rho + 4\gamma_{\mathrm{X}} + 3\gamma_{\mathrm{Y}}.$$

Les termes entre parenthèses correspondent aux différentes "fenêtres" de q_o voitures consécutives. Le premier 2 correspond aux 4 options X dans les 5 premières voitures de la suite, alors qu'il n'en fallait idéalement que 2. Le +1 qui le suit correspond aux 3 options X dans les 5 voitures aux positions 2 à 6 de la suite, alors qu'il n'en fallait idéalement que 2.

L'objectif est donc d'ordonner les n voitures en respectant la contrainte sur la peinture et en minimisant les coûts.

18.3.1 Cas général – 6 pts

On va construire un programme linéaire en nombres entiers modélisant ce problème. Pour cela, on va utiliser les variables suivantes.

$x_{i,k}$	vaut 1 si la voiture en position $i \in \{1, \ldots, n\}$ est de classe $k \in \mathcal{K}$ et vaut 0 sinon.
$y_{i,c}$	vaut 1 si la voiture en position $i \in \{1, \ldots, n\}$ est de couleur $c \in \mathcal{C}$ et vaut 0 sinon.
$z_{i,o}$	vaut 1 si la voiture en position $i \in \{1, \ldots, n\}$ a l'option $o \in \mathcal{O}$ et vaut 0 sinon.
u_i	vaut 1 si les voitures en position i et $i+1$ sont de couleurs différentes et vaut 0 sinon.
$v_{i,o}$	vaut 0 si le nombre de voitures en positions $i, i+1, \ldots, i+q_o-1$ avec l'option o est $\leq p_o$ et vaut ce nombre moins p_o sinon.

1. Ecrire la fonction objectif. (0.5 pt)

2. Ecrire des égalités mettant en jeu les $x_{i,k}$ et qui décrivent toutes les suites possibles – sachant qu'il y a n_k voitures de la classe k pour tout k – sans prise en compte de la contrainte qui limite le nombre de voitures consécutives de même couleur. (1 pt)

3. En introduisant le paramètre $\epsilon_{k,c}$ si la classe k est de couleur c et 0 sinon, et le paramètre $\delta_{k,o}$ qui vaut 1 si la classe k possède l'option o et 0 sinon, écrire des égalités liant d'une part les variables $x_{i,k}$ et $y_{i,c}$ et d'autre part les variables $x_{i,k}$ et $z_{i,o}$. (1 pt)

4. Ecrire les inégalités qui assurent la prise en compte de la contrainte qui limite le nombre de voitures consécutives de même couleur. (1 pt)

5. Ecrire des inégalités liant les u_i et les $y_{i,c}$ d'une part, et les $v_{i,o}$ et les $z_{i,o}$ d'autre part. (1 pt)

6. Ecrire le programme linéaire en nombres entiers complet. (0.5 pt)

7. Pour cette question, on suppose que le nombre $N_o = \sum_{k\in\mathcal{K}} \delta_{k,o}n_k$ de voitures ayant l'option o est tel que $N_o/n \leq p_o/q_o$. Déterminer alors la valeur optimale du programme linéaire obtenu par relaxation continue. La valeur de cette relaxation vous semble-t-elle adaptée à un branch-and-bound ? (1 pt)

18.3.2 Cas particuliers – 4 pts

Si $r = +\infty$ et $q_o = 2$ pour tout $o \in \mathcal{O}$

On suppose pour les deux questions suivantes que $r = +\infty$ et $q_o = 2$ pour tout $o \in \mathcal{O}$.

8. Montrer que le problème peut se formuler comme le problème de la chaîne hamiltonienne de plus petit poids dans un graphe complet pondéré à n sommets. (1 pt)

On suppose maintenant de plus que toutes les voitures doivent être peintes de la même couleur ($|\mathcal{C}| = 1$).

9. Résoudre le cas suivant. Chaque classe k contient une seule voiture ($n_k = 1$) et l'on prendra $\gamma_o = p_o = 1$. (1 pt)

Classe	1	2	3	4	5	6	7
Option A	1	1	0	0	0	0	0
Option B	0	1	1	1	1	0	1
Option C	1	0	1	1	0	1	1
Option D	1	1	1	0	1	1	0
Option E	0	0	0	1	1	0	0
Option F	0	0	0	0	0	1	1

Si $\mathcal{O} = \emptyset$

10*. On suppose uniquement pour cette question qu'il n'y a pas d'option sur les voitures. La chaîne de montage ne traite donc que des couleurs. On introduit pour toute couleur c la quantité $N_c = \sum_k n_k\delta_{k,c}$, qui est donc le nombre de voitures devant être peintes de la couleur c, et la quantité $m_c = \lceil N_c/r \rceil$.

Ecrire alors une condition nécessaire et suffisante qui assure l'existence d'une solution. Montrer de plus que s'il y a une solution réalisable, l'optimum vaut

$$\max\left(2m_{c^*}-2,\sum_{c\in\mathcal{C}}m_c-1\right),$$

où $c^*=\arg\max_{c\in\mathcal{C}}m_c$. (2 pts)

18.4 Voyageur de commerce et programmation dynamique – 6 pts

Considérons l'instance suivante: un graphe complet $K_n=(V,E)$ à n sommets ($n\geq 3$), des coûts positifs $c(vw)$ définis pour toute paire v,w de sommets, et un sommet particulier $s\in V$. On cherche la chaîne hamiltonienne de plus petit coût dont une des extrémités est s.

1. En prenant comme états les couples (X,v) tels que $v\in X\subseteq V\setminus\{s\}$, montrer que l'on peut écrire une équation de programmation dynamique permettant le calcul de la chaîne optimale. (2 pts)

Pour les questions suivantes, on peut se servir des identités indiquées à la fin de l'examen.

2. Quel est le nombre d'états possibles ? (1 pt)

On prend comme opération élémentaire l'addition.

3. Estimez le nombre d'additions que ferait un algorithme exploitant cette équation de programmation dynamique. Comparez ce nombre au nombre d'additions que ferait un algorithme qui énumérerait toutes les solutions. (1 pt)

4. Si votre ordinateur est capable de faire 1 million d'opérations élémentaires par seconde, pour chacune des valeurs suivantes de n, indiquez (par un calcul "à la louche") si vous serez capable de résoudre le problème avec cet algorithme en 1 seconde, 1 heure, 1 jour, 1 semaine, 1 mois, 1 an, 1 siècle (1 pt):

$$n=15\quad n=30\quad n=45.$$

5. Comment utiliser l'algorithme pour résoudre le problème du voyageur de commerce (cycle hamiltonien de plus petit coût) sur K_n ? (1 pt)

Quelques formules

$$\sum_{i=0}^{m}\binom{m}{i}=2^m.$$

$$\sum_{i=0}^{m}\binom{m}{i}i=m2^{m-1}.$$

$$\sum_{i=0}^{m}\binom{m}{i}i(i-1)=m(m-1)2^{m-2}.$$

$$m!\sim\sqrt{2\pi m}\left(\frac{m}{e}\right)^m.$$

Chapter 19

Exam 2019

- Durée 2h30. Documents papiers autorisés. Objets électroniques interdits.
- Le contrôle est noté sur 32. Un élève ayant 28 points aura 20/20. Les exercices 1 à 3 et la question 11 sont des applications directes du cours. Les autres sont un peu plus difficiles.

19.1 Climatisation d'un bâtiment (3 pts)

Voir Exercice 4.2.

19.2 Affectation de projets à des stagiaires (5 pts)

Let cabinet de conseil MacBG a recruté un ensemble S de stagiaires s en césure pour réaliser un ensemble de missions M. Les missions demandant des qualifications, pour chaque mission m, seulement un sous-ensemble S_m des stagiaires peut réaliser m. Chaque mission $m \in M$ demande $t_m \in \mathbb{Z}_+$ semaines de travail. Une mission peut-être réalisée par plusieurs stagiaires, mais si un stagiaire travaille sur une mission, il devra y passer au moins une semaine de manière à avoir le temps de se l'approprier. Pour chaque stagiaire s, on note $w_s \in \mathbb{Z}_+$ le nombre de semaines qu'il va passer chez MacBG. On cherche à affecter les stagiaires sur les missions de manière à ce que toutes les missions soient réalisées.

21. Modéliser ce problème avec les outils du cours, et montrer qu'il peut se résoudre en temps polynomial. (3 pts)

    Solution. On adapte la méthode de résolution par les flots proposée pour le problème de b-couplage. Soit $(\{o\} \cup S \cup M \cup \{d\}, A)$, $(\ell_a)_{a\in A}$, et $(u_a)_{a\in A}$ le graphe orienté et les vecteurs de capacité minimales et maximales définis de la manière suivante.

    - Un arc $a = (d, o)$ de capacité minimale $\ell_a = 0$ et maximale $u_a = +\infty$,
    - Un arc $a = (o, s)$ pour tout s dans s avec $\ell_a = u_a = w_s$,
    - Un arc $a = (s, d)$ pour tout s dans s avec $\ell_a = 0$ et $u_a = w_s$,
    - Un arc $a = (s, m)$ pour toute paire $(s, m) \in S \times M$ telle que le stagiaire s peut réaliser la mission m de capacité minimale $\ell_a = 0$ et maximale $u_a = +\infty$,
    - Un arc $a = (m, d)$ pour tout m dans M, avec $\ell_a = u_a = t_m$.

Le problème revient à trouver une circulation (un b-flot avec $b = 0$) de coût minimum dans ce graphe, ce qui peut être trouvé en temps polynomial. En effet, par théorème d'intégrité des flots, la solution sera entière, et décomposable en somme $\sum_C \lambda_C \mathcal{W}_c$ d'indicatrices de cycles C de la forme o, s, m, d, qui indiquent que s travailler sur m pendant λ_C semaines, et de cycles C de la forme o, s, d qui indiquent le nombre de semaine que s passe à ne pas travailler. □

On note maintenant δ_s la durée maximum que le stagiaire passe inoccupé (sans avoir de mission).

22. Donner un programme linéaire en nombre entier permettant de trouver une affectation qui minimise la durée maximum qu'un stagiaire passe inoccupé, i.e., telle que $\max_{s\in S} \delta_s$, soit minimum. Peut-on relâcher les contraintes d'intégrité ? (2 pts)

Solution. On reprend le graphe de la section précédente, indexe les sommets par v et les arcs par a.

$$\begin{array}{lll}
\min & z & \\
\text{s.t.} & \displaystyle\sum_{a\in\delta^-(v)} x_a = \sum_{a\in\delta^+(v)} x_a \forall v \in V & \\
& \ell_a \leq x_a \leq u_a & \forall a \in A \\
& z \geq x_{(s,d)} & \forall s \in S \\
& x_a \in \mathbb{Z}_+ & \forall a \in A
\end{array}$$

On ne peut enlever les contraintes d'intégrité car la matrice n'est plus TU à cause de la contrainte $z \geq x_{(s,d)}$, $\forall s \in S$. □

19.3 Production de boites de chocolat (5 pts)

Une chocolaterie souhaite planifier sa production pour répondre à la demande suivante (Pâques est en mars cette année là).

Demande en millier de boites			
Décembre	Janvier	Février	Mars
5	1	2	7

Les boites de chocolats sont produites par lot de 1000. La chocolaterie peut produire au plus 5 lots par mois. Le tableau suivant donne le cout de production mensuel suivant le nombre de lot produit.

Production mensuelle q (lots)	Coût $c_p(q)$ (k€)
0	0
1	25
2	35
3	45
4	60
5	80

Par ailleurs, les chocolats étant périssables, ils peuvent être stockés au plus un mois, et stocker 1 lot coût c_s de 2 k€. Il n'y a pas de chocolat en stock début décembre. La chocolaterie souhaite planifier sa production pour répondre à la demande à moindre coût.

23. Modéliser ce problème en utilisant la programmation dynamique. On précisera la fonction valeur et l'équation de Bellman utilisées. (2.5 pts)

    Solution. On indexe par $t \in \{1, \dots, 4\}$ les mois, et on note D_t la demande pendant le mois t. On notera $V(e,t)$ le coût d'un planning optimal à partir de t si le nombre lots dans le stock au mois t avant production est e. Comme une boite ne peut être gardée plus d'un mois et que la production maximale est de 5 lots, on a $e \in \{1, \dots, 4\}$. A chaque temps t, on peut décider de produire $a \in \{0, \dots, 5\}$. En notant

    $$e_{t+1}(a,e) = \min(a, e + a - D_t)$$

    l'équation de Bellman est

    $$\begin{aligned} V(e,t) &= \min_{a:\, e+a \geq D_t} c_p(a) + c_s(e_{t+1}(a,e)) + V(e_{t+1}(a,e), t+1), && \text{pour } t < 4 \\ V(e,t) &= \min_{a:\, e+a \geq D_t} c_p(a), && \text{pour } t = 4 \end{aligned}$$

    □

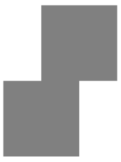
24. Résoudre le problème. (2.5 pts – attention, question calculatoire)

    Solution. Une solution optimale a pour coût 240 et consiste à produire 5,3,3,4, ce qui permet de passer par les états 0,0,2,2. Le table suivant donne la fonction valeur.

	Mois t			
e	Décembre	Janvier	Février	Mars
0	*240*	*160*	144	–
1	–	–	126	–
2	–	–	*111*	80
3	–	–	98	*60*
4	–	–	80	45
5	–	–	–	35

□

19.4 Pavage par des polynominos (9 pts)

Pour carreler un salon, rien n'est plus à la mode que des pavages par autre chose que des carreaux. On considère dans cet exercice le problème de pavage par des polynominos. Un *polynomino* est un assemblage de carrés unitaires qui se touchent par une arête, sans se recouvrir, et dont les extremités sont à des coordonnées entières du plan. Par exemple,  n'est pas

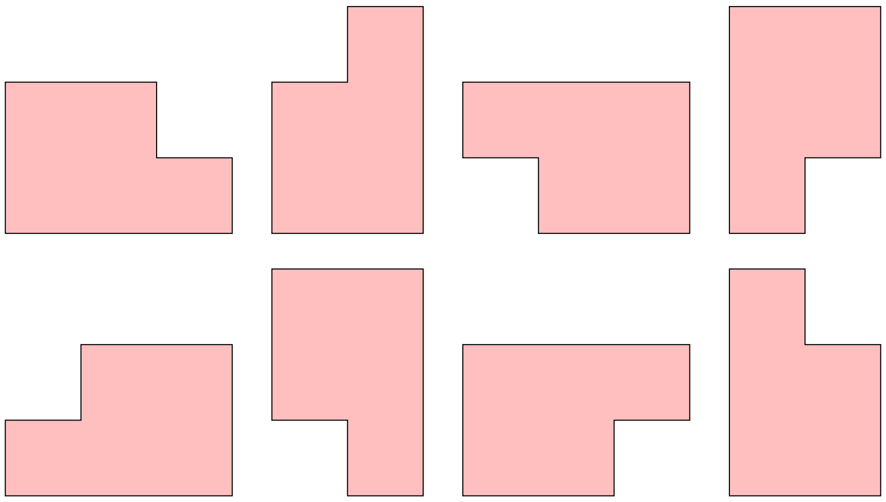


Figure 19.1: Rotations et retournement donnant les positions possibles d'un type de polynomino

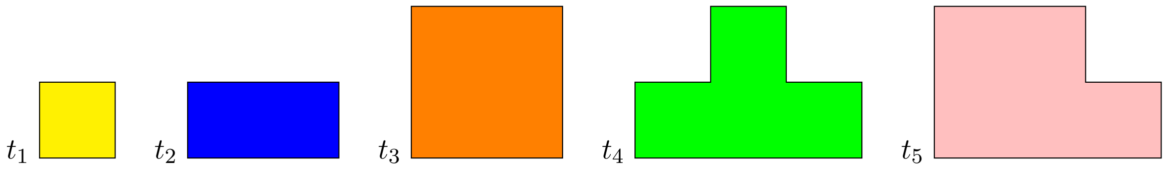
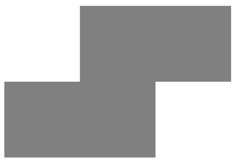


Figure 19.2: Quelques exemples de polynominos

un polynomino car certains carrés unitaires ne sont pas sur des coordonnées entières du plan, tandis que  est un polynomino.

Lorsque l'on utilise des polynominos pour paver une pièce, comme illustré sur la Figure 19.1, chaque polynomino peut être posé dans 8 positions différentes, obtenues par rotation de 90 degrés ou retournement. Selon les symétries des polynominos, certains positions sont identiques. Deux polynominos sont dits du même types d'ils peuvent être obtenus par rotation ou symétrie. La Figure 19.2 donne quelques exemples de types de polynominos.

Les pièces à paver ont le même type de forme : murs à angles droits, dont les angles sont dans des coordonnées entières du plan. La Figure 19.3 donne un exemple de pièce à paver. On note $P = \{p_1, \ldots, p_\ell\}$ l'ensemble des carrés unitaires composant la pièces. L'ensemble P permet de se repérer dans la pièce à paver. La Figure 19.3 donne un exemple de pièce à paver.

Un *pavage* d'une pièce par des polynominos est composé d'un assemblage de polynominos, posés de manière ce que les carrés unitaires d'un polynomino recouvre des pavés unitaires de la pièce, de manière à ce que tous les carrés unitaires de la pièce soient couverts par un et un seul polynomino. La Figure 19.4 illustre un pavage de la pièce de la Figure 19.3 avec les polynominos de la Figure 19.2.

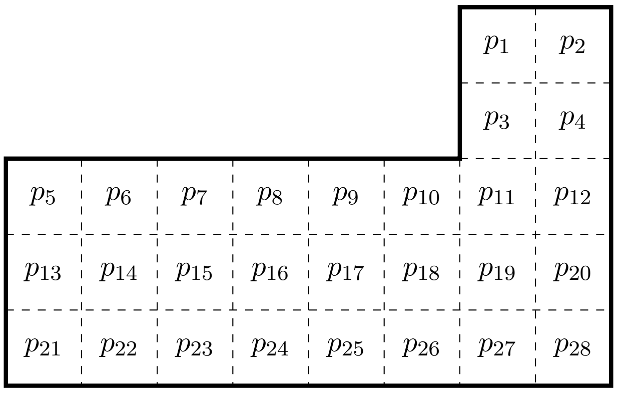

p_1
p_2
p_3
p_4
p_5 p_6 p_7 p_8 p_9 p_{10} p_{11} p_{12}
p_{13} p_{14} p_{15} p_{16} p_{17} p_{18} p_{19} p_{20}
p_{21} p_{22} p_{23} p_{24} p_{25} p_{26} p_{27} p_{28}

Figure 19.3: Une pièce à paver

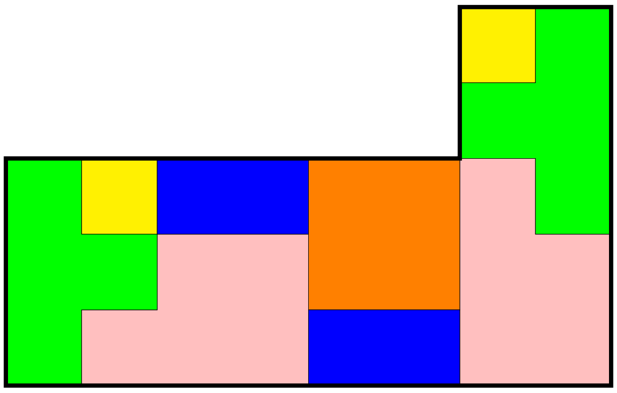


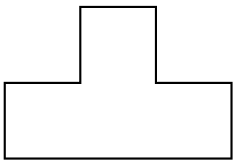
Figure 19.4: Une pièce pavée par des polynominos

19.4.1 Algorithme polynomial pour les dominos

On commence par les cas simple où l'on a que des dominos , c'est à dire de polynominos rectangulaires composé de deux carrés unitaires.

25. Donner un exemple de pièce composée d'un nombre pair de carrés unitaires qui ne peut être pavée exclusivement avec des dominos. (1 pt)

    Solution. Un exemple

    □

26. Montrer que le problème de savoir si une pièce peut être pavée avec des dominos (sans en découper) peut être résolu en temps polynomial avec un algorithme vu en cours. (3 pts)

    Solution. On construite le graphe suivant. Un sommet par possible. Une arête entre deux sommets si ils sont voisins (séparés par une arête). On veut savoir s'il existe un couplage parfait dans ce graphe.

    Un tel couplage peut être trouvé en temps polynomial car le graphe est biparti. Notons $i \in \mathbb{Z}$ et $j \in \mathbb{Z}$ les coordonnées d'un sommet. Considérant deux sommets voisins (i_1, j_1) et (i_2, j_2) pris dans leur ordre lexicographique, on a nécessairement $i_2 = i_1 + 1$ et $j_2 = j_1$ ou $i_1 = i_2$ et $j_2 = j_1 + 1$. Le graphe est biparti entre les sommets de $i + j$ pair et les sommets de $i + j$ impair. □

19.4.2 PLNE pour les polynominos

On s'intéresse maintenant au cas général. On souhaite paver une pièce avec des polynominos. On note toujours P l'ensemble des carrés unitaires composant la pièce à paver. On rappelle qu'un pavage doit couvrir toute la pièce: tout élément de P doit être recouvert par un et un seul polynomino. On a un ensemble fini T de types t de polynominos disponibles. Par exemple, pour le pavage de la pièce de la Figure 19.4, on a $T = \{t_1, \dots, t_5\}$ où $t_1, \dots, t_5$ sont illustrés sur la Figure 19.2.

On souhaite trouver un pavage

- de coût minimum (le coût est la somme du coût des polynominos utilisés),
- tel que au minimum ℓ_t et au plus u_t polynominos du type t sont utilisés dans le pavage,
- et tel que, deux polynominos de même type ne peuvent se toucher (ni par une arête, ni dans un angle) – ceci pour des raisons esthétiques.

Remarquez qu'un polynomino de type t peut être placé à plusieurs endroits de la pièce et dans plusieurs positions différentes (rotations, retournements). Ces endroits et positions peuvent être calculés en prétraitement.

27. Modéliser ce problème sous la forme d'un programme linéaire en nombres entiers dont le nombre de variables est polynomial en la taille de l'instance. Justifier (5 pts)

Solution. Pour chaque type de polynominos t, on note S_t l'ensemble des configuration s dans lesquelles on peut poser t dans la pièce (au plus 8 fois $|P|$). On note E_t l'ensemble des paires de position $s,'$ dans S_t telles que s et s' se touchent (ont une arête commune ou un angle commun). On note $S = \bigcup_t S_t$. Pour tout s dans S_t, on introduit une variable binaire x_s indiquant si un polynominos t est installé dans la position s. Pour tout p dans P, on note S'_P le sous-ensemble de S de configurations s couvrant p. Le PLNE suivant modélise le problème

$$\begin{aligned}
\min \sum_{t\in[m]} c_t \sum_{s\in S_t} c_t x_s & \\
\text{s.t.} \sum_{s\in S_t} x_s \geq \ell_t, \quad & \forall t \in [m] \\
\sum_{s\in S_t} x_s \leq u_t, \quad & \forall t \in [m] \\
\sum_{s\in S'_p} x_s = 1 \quad & \forall p \in P \\
x_s + x_{s'} \leq 1 \quad & \forall (s,s') \in S_t, \forall t \in [m] \\
x_s \in \{0,1\} \quad & \forall s \in S.
\end{aligned}$$

□

19.5 Positionnement d'ambulances – 10 pts

Pour diminuer le temps mis par les ambulances pour atteindre les victimes d'accidents, certains hôpitaux envisagent de les positionner de manière optimisée sur le département dont ils ont la charge. Considérons donc un certain hôpital ayant K ambulances. L'ensemble des communes est noté C. On dispose d'un ensemble S de points de stationnement potentiel et on suppose que $|S| \geq K$. Pour $c \in C$ et $s \in S$, on note $t_{s,c}$ le temps nécessaire à une ambulance positionné en s pour atteindre c. Si $S' \subseteq S$ est l'ensemble des points où sont positionnées les ambulances (avec bien sûr $|S'| \leq K$), dans le pire des cas, une ambulance atteindra la commune où a eu lieu l'appel en un temps égal à $\max_{c\in C} \min_{s\in S'} t_{s,c}$. C'est ce temps maximum que l'hôpital souhaite rendre le plus petit possible.

28. Montrez que ce problème est NP-difficile. Indication: soit $G = (V,E)$ un graphe non-orienté; on appelle *dominant* un sous-ensemble $Y \subseteq V$ tel que tout $v \in V$ est soit dans Y, soit a un voisin dans Y; décider s'il existe un dominant de taille $\leq K$ est NP-complet. (3 pts)

Solution. Prenons un graphe $G = (V,E)$ et construisons l'instance suivante du problème des ambulances: $S = C = V$, K ambulances, et $t_{u,v}$ = nombre minimum d'arêtes de u à v. Il y a un dominant à $\leq K$ sommets dans G si et seulement si le temps maximum pour le problème des ambulances associé est $= 1$. Si le problème des ambulances admettait un algorithme polynomial, on pourrait alors s'en servir pour tester l'existence en temps polynomial de dominant de taille $\leq K$ dans le graphe. Ce qui est impossible, ce dernier problème étant NP-complet. □

29. Démontrez que le programme linéaire suivant modélise le problème. Pour cela, procédez en deux temps: montrez que toute solution optimale du problème donne une solution de même valeur au programme linéaire; puis montrez que toute solution optimale du programme linéaire donne une solution de même valeur au problème. (2 pts)

$$
\begin{array}{llllr}
\min & h & & & \\
\text{s.c.} & \sum_{s\in S} y_s & \leq & K & \text{(i)} \\
& \sum_{c\in C} x_{s,c} & \leq & |C|y_s \quad \text{pour tout } s\in S & \text{(ii)} \\
& \sum_{s\in S} x_{s,c} & = & 1 \quad \text{pour tout } c\in C & \text{(iii)} \\
& t_{s,c}x_{s,c} & \leq & h \quad \text{pour tout } s\in S \text{ et } c\in C & \text{(iv)} \\
& x_{s,c} & \in & \{0,1\} \quad \text{pour tout } s\in S \text{ et } c\in C & \text{(v)} \\
& y_s & \in & \{0,1\} \quad \text{pour tout } s\in S & \text{(vi)} \\
& h & \in & \mathbb{R}_+ & \text{(vii)}
\end{array}
$$

Solution. Considérons une solution optimale au problème des ambulances. On construit alors de la manière suivante une solution au programme linéaire: $y_s = 1$ s'il y a au moins une ambulance située en s et $y_s = 0$ sinon; $x_{s,c} = 1$ si l'ambulance la plus proche de c est située en s, et $x_{s,c} = 0$ sinon. Si plusieurs sites sont les plus proches, on en choisit arbitrairement un. Enfin, on pose $h =$ temps maximum pour qu'une commune soit atteinte par une ambulance. Cette solution est une solution réalisable du programme linéaire, et de même valeur du critère à optimiser.

Réciproquement, soit $\boldsymbol{x}, \boldsymbol{y}, h$ une solution optimale du programme linéaire. On positionne une (ou plusieurs) ambulance en s si $y_s = 1$. Comme on cherche à minimiser h, h est forcément égal à $\max_{c\in C}\min_{s\in S\colon x_{s,c}=1} t_{s,c}$ d'après l'inégalité (iv). Combiné avec (ii), cela indique que toute les communes peuvent être ralliées en un temps maximum h.

Le problème et le programme linéaire ont donc même valeur optimale et on peut passer directement d'une solution du problème à une solution du programme linéaire et vice-versa. □

Un tel programme linéaire se résout par branch-and-bound. La borne dont on se sert alors peut être celle obtenue en relâchant les contraintes d'intégrité de $\boldsymbol{x}$ et de $\boldsymbol{y}$. On se propose de voir si l'on peut également obtenir des bornes par relaxation lagrangienne.

30. Ecrire le programme d'optimisation obtenu lorsqu'on procède à la relaxation lagrangienne de la contrainte (ii). (1 pt)

Solution. En notant λ_s le multiplicateur de Lagrange assiocié à la contrainte (ii), on obtient

$$\begin{array}{llllll}
\min & h + \sum_{s\in S}(-\lambda_s)|C|y_s + \sum_{s\in S}\sum_{c\in C}\lambda_s x_{s,c} & & & & \\
\text{s.c.} & \sum_{s\in S} y_s & \leq & K & & \text{(i)} \\
& \sum_{s\in S} x_{s,c} & = & 1 & \text{pour tout } c \in C & \text{(iii)} \\
& t_{s,c}x_{s,c} & \leq & h & \text{pour tout } s \in S \text{ et } c \in C & \text{(iv)} \\
& x_{s,c} & \in & \{0,1\} & \text{pour tout } s \in S \text{ et } c \in C & \text{(v)} \\
& y_s & \in & \{0,1\} & \text{pour tout } s \in S & \text{(vi)} \\
& h & \in & \mathbb{R}_+ & & \text{(vii)}
\end{array}$$

□

31. Montrez que ce programme se ramène à deux programmes distincts et indépendants, l'un ne mettant en jeu que les variables $\boldsymbol{y}$, l'autre ne mettant en jeu que les variables $\boldsymbol{x}$ et h. (1 pt)

Solution. Les variables $\boldsymbol{y}$ d'une part, et les variables $\boldsymbol{x}$ et h d'autre part sont indépendantes dans les contraintes du programme linéaire de la question précédente. Quelque soit le choix de $\boldsymbol{y}$, on peut fixer $\boldsymbol{x}$ et h indépendament. On optimise donc les deux programmes

$$\begin{array}{llllll}
\min & h + \sum_{s\in S}\sum_{c\in C}\lambda_s x_{s,c} & & & & \\
\text{s.c.} & \sum_{s\in S} x_{s,c} & = & 1 & \text{pour tout } c \in C & \text{(iii)} \\
& t_{s,c}x_{s,c} & \leq & h & \text{pour tout } s \in S \text{ et } c \in C & \text{(iv)} \\
& x_{s,c} & \in & \{0,1\} & \text{pour tout } s \in S \text{ et } c \in C & \text{(v)} \\
& h & \in & \mathbb{R}_+ & & \text{(vii)}
\end{array}$$

et

$$\begin{array}{llllll}
\min & \sum_{s\in S}(-\lambda_s)y_s & & & & \\
\text{s.c.} & \sum_{s\in S} y_s & \leq & K & & \text{(i)} \\
& y_s & \in & \{0,1\} & \text{pour tout } s \in S & \text{(vi)}
\end{array}$$

indépendamment.

□

32. Montrez que chacun de ces deux programmes se résout en temps polynomial. (2 pts)

Solution. Optimiser le deuxième programme est simple: il suffit de trier les s par valeurs de λ_s décroissantes; ensuite on met $y_s = 1$ pour les K premiers s, et $y_s = 0$ pour les suivants. Complexité $O(S \log S)$.

Optimiser le premier programme est presque aussi simple. Noter que les λ_s sont positifs. A l'optimum, (iv) est une égalité. On essaie successivement dans h toutes les valeurs de $t_{s,c}$, et pour chacune de ces valeurs, on fait la chose suivante: pour chaque c, on cherche parmi les $t_{s,c} \leq h$ celui tel que λ_s est le plus petit, et c'est ce couple (s, c) pour lequel $x_{s,c} = 1$. Pour chacune des SC valeurs de h essayées, on obtient une valeur de la fonction objectif; on garde alors la plus petite. Complexité $O(S^2C^2)$. □

33. Expliquez pourquoi la polynomialité de ces deux programmes permet d'espérer un calcul efficace d'une borne inférieure au programme linéaire de la question 2. (1 pt)

Solution. Avec les notations de cours, pour tout $\boldsymbol{\lambda} \in \mathbb{R}^S$, on sait calculer $\mathcal{G}(\boldsymbol{\lambda})$ et ses surgradients en $O(S^2C^2)$ (donc rapidement), et donc optimiser par un algorithme de surgradient $\mathcal{G}(\boldsymbol{\lambda})$, fournissant ainsi des bornes au programme linéaire en nombres entiers. □

Chapter 20

Rattrappage 2019

- Durée 2h. Documents interdits. Objets électroniques interdits.
- Le contrôle est noté sur 20. Le barème ne sera pas réévalué.

20.1 Activités dans un club de vacances (5 points)

Un club de vacances souhaites planifier les activités (sports, loisirs, etc.) qu'il organise pour ses clients pour la semaine à venir. Avant de venir, chaque client indique la liste des activités auquel il souhaite participer. Il y a 10 créneaux disponibles dans la semaine : un le matin, un l'après midi, chaque jour du lundi au vendredi. Chaque activité ne peut être planifiée qu'une seule fois. Deux activités peuvent être placées sur les même créneau, mais dans ce cas là, un client ne pourra effectuer qu'une seule des deux activités. On cherche à savoir s'il est possible de planifier les activités de manière à ce que toutes les demandes soient satisfaites.

34. Modéliser ce problème comme un problème de graphe. *(2.5 points)*

    Solution. Coloration dans le graphe dont les sommets les activités et les arêtes les paires d'activités (u, v) telles qu'au moins un client souhaite réaliser u et v. □

Pour la saison suivante, le club décide de monter en gamme et de satisfaire toutes les demandes des clients. On note I l'ensemble des clients, et S_i l'ensemble des activités demandées par le client i. Un client peut demander au plus 10 activités (le nombre total de créneaux). On note J l'ensemble des activités. Une activité j peut maintenant être planifiée sur plusieurs créneaux. Planifier l'activité j sur un créneau coute c_j. On note T l'ensemble des créneaux t. On souhaite planifier les activités de manière à répondre aux demandes des clients à moindre coût.

35. Modéliser le problème sous la forme d'un programme linéaire en nombres entiers *(2.5 points)*

    Solution. Variable binaires x_{ijt} indiquant si i réalise j sur le créneau t, et y_{jt} indiquant si j est planifiée à t. □

20.2 Convoi militaire (5 points)

Un convoi militaire doit rejoindre une destination d depuis un origine o. Les routes empruntables sont modélisées par une graphe non-orienté $G = (V, E)$ dont les sommets sont les carrefours et les arêtes les routes. L'origine o et la destination d sont des sommets dans V. Un convoi passant sur une arête e est attaqué avec probabilité $1 - p_e$, et n'est pas attaqué avec probabilité p_e. Les attaques sur les différents arêtes sont supposées indépendantes.

36. Expliquez pourquoi la probabilité de ne pas se faire attaquer sur un o-d chemin est $\prod_{e \in P} p_e$. *(0.5 point)*

    Solution. Variables aléatoires indépendantes. □

On cherche donc un o-d chemin minimisant la probabilité d'attaque, c'est à dire maximisant $\prod_{e \in P} p_e$.

37. Expliquer comment, en utilisant la fonction logarithme, on peut modéliser ce problème comme un problème de plus court chemin. *(1 point)*

    Solution.
    $$\min_P \sum_{a \in P} - \log(p_a)$$
    □

38. Donner le chemin minimisant la probabilité d'attaque pour le graphe sur la Figure. *(1 point)*

    Solution. Chemin o-3-5-8-11-d de coût 19. □

39. Quel est l'algorithme le plus efficace pour calculer ce plus court chemin ? Justifier. *(1 point)*

    Solution. Dijkstra car $-\log(p_a) \geq 0$ pour tout a. □

40. Adapter cet algorithme pour calculer directement le chemin $\prod_{e \in P} p_e$ *sans utiliser la fonction logarithme*. *(1.5 points)*

    Solution. Remplace + par × dans Dijsktra. □

20.3 Stratégie de révision (5 points)

Un étudiant dispose de 5 jours pour préparer ses examens dans 3 matières. Chaque matière a le même coefficient. Quand il travaille une matière, il la travaille une journée complète. Il souhaite allouer ses journées de travail pour maximiser sa moyenne finale. Le tableau suivant donne la note qu'il espère avoir dans la matière selon le temps passé à réviser.

41. Modéliser le problème à l'aide de la programmation dynamique. (Attention, plusieurs solutions possibles, qui rendront plus ou moins difficile la question suivante) *(2.5 points)*

    Solution. Plusieurs solution possible. Une solution efficace: instant $t \in \{0, \ldots, 3\}$. A l'étape t les matières $0, 1, \ldots, t$ ont été considérées. Etat: paire nombre de jour travaillés jusque là en tout. Mettre le nombre de points récoltés en état donne des espaces d'état plus grands. □

42. Donner la stratégie de révision optimale, et la note espérée. *(2.5 points)*

    Solution. Score maximal: 17, moyenne 5.67, stratégie (par exemple): 4 jours sur m_1, 1 sur m_2 et 0 sur m_3 □

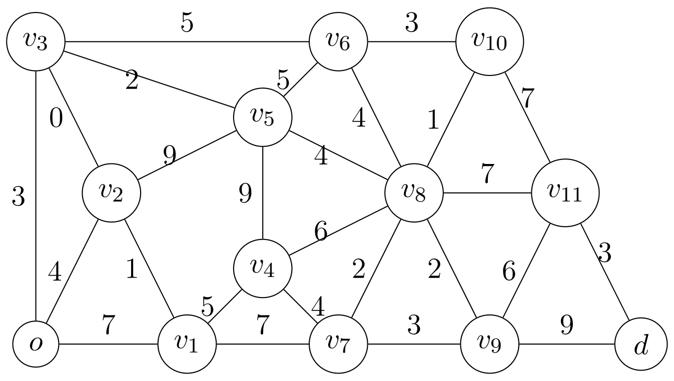


Figure 20.1: Probabilité d'attaque. Pour chaque arête e, la quantité $-\log(p_e)$ est indiquée.

Jours de travail	Matière		
	m_1	m_2	m_3
0	2	1	3
1	4	4	5
2	6	6	6
3	8	7	7
4	10	7	8

Table 20.1: Note attendue dans chaque matière en fonction du temps de révision. Par exemple, 3 jours de travail dans la matière m_1 rapporte 7 points

20.4 Course de VTC (5 points)

Une compagnie de VTC opérant pour des trajets professionnels connait chaque soir l'ensemble T des trajets qu'elle doit opérer le lendemain. Chaque trajet est caractérisé par une heure de début, une heure de fin. Il existe deux sortes de coûts. Les coûts liés au déplacement effectués, et les coûts liés aux nombres de chauffeurs embauché pour la journée. Le déplacement d'un chauffeur de son domicile à la première course, et de la dernière course à son domicile n'est pas rémunéré. Embaucher un chauffeur pour la journée coute α. Etant donné deux trajets t et t', on note c_t la rémunération perçue par une chauffeur pour effecteur la course t, et $c_{tt'}$ la rémunération perçue par une chauffeur pour aller du lieu de fin du trajet t au lieu de début de trajet t'. On cherche à effectuer l'ensemble des courses à coût minimal.

43. Modéliser le problème comme un problème de flot dans un graphe que l'on précisera. *(1 point)*

    Solution. Deux sommets u_t et v_t par courses, avec un arc de capacité min et max 1 entre les deux. Arcs $(v_t, u_{t'})$ si t et t' peuvent être enchainées, de cout $c_{tt'}$. Un sommet origine o, un sommet destination d, un arc d-o de coût α, des arcs (o, u) et (v, d) pour tout u et v. □

44. Donner un programme linéaire modélisant ce problème. *(1 point)*

    Solution. Cours □

45. Calculer le dual de ce programme linéaire. Justifier. *(1.5 point)*

Solution. Cours □

On suppose maintenance que le coût $c_{tt'}$ est nul.

46. A quoi correspond une solution admissible entière du problème dual? *(1 point)*

    Solution. Coupe. □

47. A t'on dualité faible? Dualité forte? *(0.5 point)*

    Solution. Cours. □

Bibliography

Kenneth Appel, Wolfgang Haken, et al. Every planar map is four colorable. part i: Discharging. *Illinois Journal of Mathematics*, 21(3):429–490, 1977.

Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, Alberto Marchetti-Spaccamela, and Marco Protasi. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media, 2012.

Béla Bollobás. *Modern graph theory*, volume 184. Springer Science & Business Media, 2013.

John Adrian Bondy, Uppaluri Siva Ramachandra Murty, et al. *Graph theory with applications*, volume 290. Citeseer, 1976.

Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

Reinhard Diestel. *Graph theory*. Springer Publishing Company, Incorporated, 2018.

Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

Michael R Garey and David S Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.

Jean-Charles Gilbert. Optimisation différentiable: théorie et algorithmes. *INRIA Rocquencourt*, 2002.

Bernhard Korte, Jens Vygen, B Korte, and J Vygen. *Combinatorial optimization*, volume 2. Springer, 6 edition, 2017.

Casimir Kuratowski. Sur le probleme des courbes gauches en topologie. *Fundamenta mathematicae*, 15(1):271–283, 1930.

Richard J Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

Jiri Matousek and Bernd Gärtner. *Understanding and using linear programming*. Springer Science & Business Media, 2007.

Stephan Mertens. Random stable matchings. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(10):P10008, 2005.

Frédéric Meunier. *Introduction à la Recherche Opérationnelle*. 2018.

Barbara A Oakley. *A mind for numbers: How to excel at math and science (even if you flunked algebra)*. TarcherPerigee, 2014.

Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

Boris Pittel. The average number of stable matchings. *SIAM Journal on Discrete Mathematics*, 2(4):530–549, 1989.

Ronald L Rardin and Ronald L Rardin. *Optimization in operations research*, volume 166. Prentice Hall Upper Saddle River, NJ, 1998.

R Tyrrell Rockafellar and Roger J-B Wets. *Variational analysis*, volume 317. Springer Science & Business Media, 2009.

Alvin E Roth. *Who Gets What—and Why: The New Economics of Matchmaking and Market Design*. Houghton Mifflin Harcourt, 2015.

Alexander Schrijver. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer Science & Business Media, 2003.

Index