Operations Research

Axel Parmentier

November 30, 2018
École des Ponts Paristech – Academic year 2018-2019

Contents

Chapter 1

Introduction

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 $\{1, \ldots, m\}$. 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

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

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 $\overline{A} = A \cup \{\lrcorner\}$. 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 $s_0(j) = \lrcorner$ 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 $k = \min\{j : s_i(j) = \lrcorner\}$, 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.

`A:gives example of the successive states`

Exercise 2.1. What does the algorithm described by the following Turing Machine ? `A:todo` △

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. `A:todo: non decidable and non polynomial time Turing machine.` △

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

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

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 $\mathcal{P}$. `A:give an exercise`

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 $(X, (S_x)_{x\in X}, c, \text{goal})$ 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.*
- $\text{goal} \in \{\min, \max\}$.

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 $\text{Sol}(x)$. The *value* of an instance is $\text{val}(x) = \text{goal}\{c(x,y)\colon x \in X, y \in S_x\}$. The *optimal solutions* are the elements y of S_x that $c(x,y) = \text{val}(x)$. We denote by $\text{Opt}(x)$ 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 $\mathcal{NP}$-hard if all the problems in $\mathcal{NP}$ polynomially reduce to $\mathcal{P}$.*

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

To prove the $\mathcal{NP}$-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 $\mathcal{NP}$-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 $\mathcal{NP}$-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 $\mathcal{NP}$-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 $|A|$ 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 is 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

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

such that v_i is a vertex in V for all i in $\{0, \ldots, n\}$, and e_i is an edge in E adjacent to v_{i-1} and v_i for all i in $\{1, \ldots, n\}$. 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 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 e_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 $u \leftrightarrow v$ be the binary relation on V indicating if there exists a u-v path.

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

△

3.1.4 Cycles

A *cycle* or *circuit* C in a graph G is a path $v_0, \ldots, v_k$ such that $v_1, \ldots, v_k$ 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 a 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 $V(P) = V(G)$. An *Eulerian cycle* C in a graph G is a 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 are incident with the same vertex v. A *vertex cover* C is a subset of V such that such that each edge of A 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?

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.*

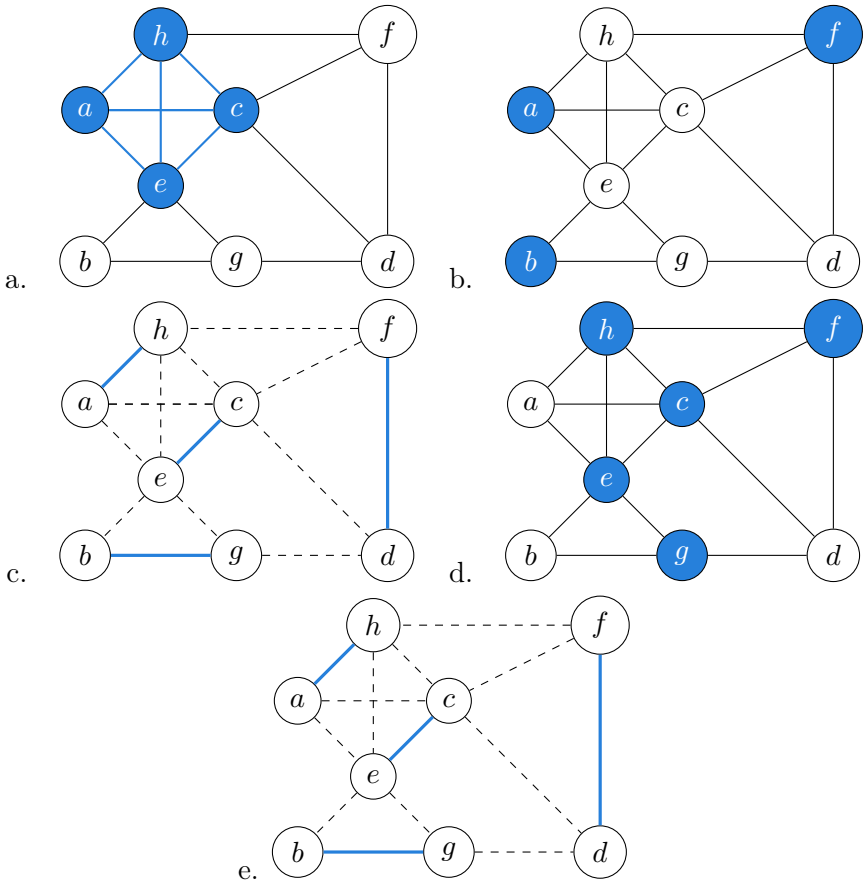


Figure 3.1: Example of a. clique, b. stable set, c. matching, d. vertex cover, and e. edge cover

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*

A:todo: an introduction to Shannon capacity. Finish with an exercise

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.*

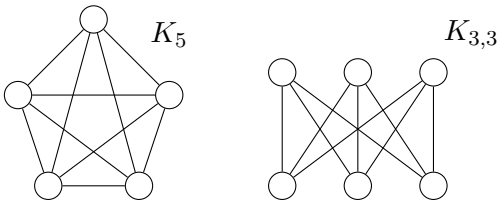


Figure 3.2: The two smallest non-planar graphs

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 `A:todo: an exercise on the topic`, 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]. `A:todo: an exercise on the topic`.

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 `A:todo`

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.

A:todo: figure

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

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

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 $\{0, \ldots, n\}$, and a_i is an arc in A from v_{i-1} to v_i for all i in $\{1, \ldots, n\}$. An *o-d* path is a path from o to d. 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 $U \subseteq V$. △

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. △

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. △

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. △

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

Exercise 3.19. 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. △

Chapter 4

Spanning trees

Chapter 5

Shortest paths and dynamic programming

This chapters 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 $\mathcal{NP}$-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 note mentioned in Table 5.1 because they correspond to trees and forests, and hence there is a unique o-d path is 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, and to $+\infty$ if no such path exists. Proposition 5.2 ensures that f satisfies *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. As 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 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 1.

Let $\preceq$ be a topological ordering on D. The $f(v)$ for $o \preceq v \preceq D$ can be computed iteratively along $\preceq$ in $O(m+n)$ using

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

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.*

Algorithm 1 Compute a topological order on D.

Input: a digraph $D = (V, A)$
Initialization: $L \leftarrow$ empty list, $S \leftarrow \emptyset$.
$S \leftarrow \{v : \}$
while $S \neq \emptyset$ **do**
  remove a vertex v from S
  add v at the end of L
  remove from A all the arcs in $\delta^+(v)$
  add to S all the vertices w in $N^+(v)$ such that $\delta^-_A(w) = \emptyset$
end while
return L *(L is sorted along a topological order $\preceq$)*

5.1.3 Dynamic programming, a general method

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 5.6. *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 $\min_{u \in N^-(v) \cap V_k} c_{u,v}$. Then $V_k \cup \{v\}$ is a set of $k+1$ nearest vertices to v.*

Proof. □

Algorithm 2 Dijkstra algorithm

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

Algorithm 2 states *Dijkstra* algorithm for graphs with arcs of multiplicity at most one. It is easily generalized to graphs with arbitrary multiplicity.

Proposition 5.7. *Dijkstra algorithm converges in $O(m + n \log(n))$ if the L the list 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 5.8. *Let c_{od}^{ub} 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 $c_P + b_v > c_{od}^{\text{ub}}$, then P is not the subpath of an optimal path.*

Algorithm 3 A*

Input: A digraph $D = (V, A)$, costs $\boldsymbol{c}$ in $\mathbb{Q}^A$, bounds $\boldsymbol{b}$ in $\mathbb{Q}^V$.
Initialization: $c_{od}^{\text{ub}} \leftarrow +\infty$, $L \leftarrow \{\text{empty path in } o\}$ 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 $c_P < c_{od}^{\text{ub}}$ **then**
    $c_{od}^{\text{ub}} \leftarrow c_P$
  else
    for all $w \in N^+(v)$ such that $c_P + b_v < c_{od}^{\text{ub}}$ **do**
      Add P followed by (v, w) to L with key $c_P + b_v$.
    end for
  end if
end while
return: c_{od}^{ub}

Proposition 5.9. *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_{od}^{ub} returned is the cost of a shortest o-v path.*

A* can be proved to converge under more general conditions than those of Proposition 5.9.

5.3.2 Generating bounds

5.4 Exercises

Chapter 6

Flows

Teaching remark: This chapter is long. And maximum s-t flows and minimum cost flow are somehow redundant. Suggestions:
- Teach maximum s-t flows on the blackboard. (1h)
- Teaching minimum cost flow and linear programming using the video-projector (30min)
- In the linear programming, do not prove the max-flow min-cut Lagrangian duality during the lesson. The dualization of the max flow is not completely easy. Put it as a corrected exercise.
- Do exercises on flows also during the next sessions.

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*

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

It is an s-t flow subject to u or under u if $f(a) \leq u(a)$ 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

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

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 $\text{val}(f) = u(U)$ 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 $\gamma = \min_{a \in P} u_f(a)$. □

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 4) is based on this result.

Algorithm 4 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 4 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

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

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

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

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 5 to solve the MINIMUM COST FLOW PROBLEM.

Algorithm 5 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

$$\max \sum_{a \in \delta^+(s)} x_a - \sum_{a \in \delta^-(s)} x_a \tag{6.2a}$$

$$\text{s.t.} \sum_{a \in \delta^-(v)} x_a = \sum_{a \in \delta^+(v)} x_a, \quad \forall v \in V \backslash \{s,t\} \tag{6.2b}$$

$$x_a \leq u_a \qquad \forall a \in A \tag{6.2c}$$

$$x_a \geq 0 \qquad \forall a \in A \tag{6.2d}$$

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 9.10 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).

$$\max \; q \tag{6.3a}$$

$$\text{s.t.} \sum_{a \in \delta^+(s)} x_a - \sum_{a \in \delta^-(s)} x_a = q \tag{6.3b}$$

$$\sum_{a \in \delta^+(t)} x_a - \sum_{a \in \delta^-(t)} x_a = -q \tag{6.3c}$$

$$\sum_{a \in \delta^+(v)} x_a - \sum_{a \in \delta^-(v)} x_a = 0, \quad \forall v \in V \backslash \{s,t\} \tag{6.3d}$$

$$x_a \leq u_a \qquad \forall a \in A \tag{6.3e}$$

$$x_a \geq 0 \qquad \forall a \in A \tag{6.3f}$$

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

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

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. △

Chapter 7

Matchings

Teaching remark: This chapter is pretty short, and should remain short, to save time for exercises on flows. When teaching:
- not needed to introduce the Hungarian algorithm
- Only state the problem, and give as an exercise to give the flow formulations.
- Prove Theorem 7.8. Just tell the story for the remaining on stable marriages (why not slides)

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

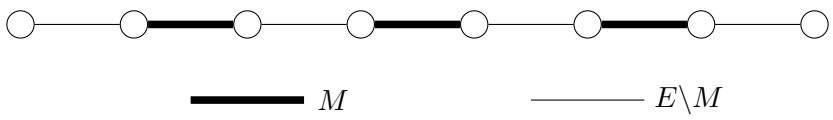


Figure 7.1: An M-augmenting path.

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.*

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 6 states the Hungarian algorithm.

Algorithm 6 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 6 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. □

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. □

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 $\sum_{e \in M} w(e)$.

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.*

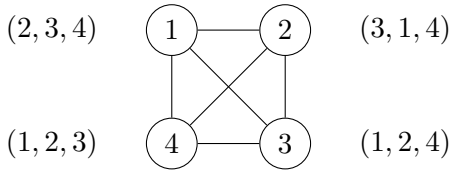
Proof. It suffices to prove that it is male optimal. Consider for a contradiction that u is the first man rejected by his optimal partner w, because she prefers u'. By definition of u, man u' has still not been rejected by his optimal woman – each man considers women by decreasing order of preference. 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.



△

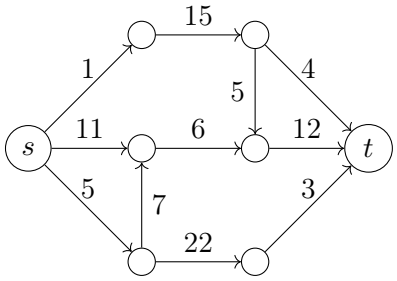
7.6 Further reading

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.


15
1
4
5
11
6
12
s
t
5
7
3
22

△

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 $\mathbb{R}^n$.

8.1 Lagrangian duality

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

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

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 $\text{val}\,(\text{P}) - \text{val}\,(\text{D})$ is called the *duality gap*.

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.

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.

8.2 KKT conditions

A:todo

Chapter 9

Linear Programming

Teaching remark:
- The reminder on duality can be done during the lecture on bipartite matching (bipartite matching and vertex cover are pretty similar)
- For a quick reminder during a lecture:
  - geometric interpretation of simplex
  - simplex equivalent form of a LP
  - If enough time, strong duality by linear programming
  - just mentioning the algorithm
- Line and column generation might be done as an exercise.

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).

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

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

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

Its *dual* is the linear program

$$\begin{array}{ll} \min\limits_{y\in\mathbb{R}^m} & \boldsymbol{b}^T\boldsymbol{y} \\ \text{s.t.} & A^T\boldsymbol{y} \leq \boldsymbol{c} \\ & \boldsymbol{x} \geq 0, \end{array} \tag{D}$$

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) *us unbounded and* (P) *has no feasible solution.*
4. *Both* (P) *and* (D) *have a feasible solution and*

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

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 val (P) $\geq$ val (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

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

which concludes the proof. □

9.4 Total unimodularity

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

Proposition 9.6. *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 9.7. *(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. △

Corollary 9.8. *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 9.9. *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 9.10. *The incidence matrix of a directed graph is totally unimodular.*

9.5 Line and column generation

Conclude with the separation theorem. Starting from the separation theorem

9.6 Further readings

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

Chapter 10

Integer Programming

A *mixed integer linear program* is an optimization problem of the form

$$\begin{array}{rl} \min & \boldsymbol{c}\boldsymbol{x}, \\ \text{s.t.} & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \\ & \boldsymbol{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p}. \end{array} \tag{MILP}$$

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

$$\begin{array}{rl} \min & \boldsymbol{c}\boldsymbol{x}, \\ \text{s.t.} & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \\ & \boldsymbol{x} \in \mathbb{R}^n. \end{array} \tag{REL}$$

Recall that the value of a problem P is denoted by v(P) and equal to $+\infty$ if P admits no solution. We have

$$\text{val}\,(\text{REL}) \leq \text{val}\,(\text{MILP}). \tag{10.1}$$

Exercise 10.1. Prove Equation (10.1). △

Solution.

$$\text{Sol}\,(\text{MILP}) \subseteq \text{Sol}\,(\text{REL}).$$

□

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*

$$\begin{array}{rcl} b : \mathcal{Y} & \to & \mathcal{Y} \times \mathcal{Y} \\ Y & \mapsto & Y_1 \times Y_2 \end{array} \quad \text{s.t.} \quad \left| \begin{array}{l} Y_1 \cup Y_2 = Y, \\ Y_1 \cap Y_2 = \emptyset \end{array} \right.$$

Typically, we have $b(Y_i) \geq b(Y)$. Algorithm 7 states the branch-and-bound algorithm.

Algorithm 7 Branch-and-bound algorithm (general)

Input: an instance of (10.2), λ and b;
Output: an optimal solution x^* or $\emptyset$ if no optimal solution exist;
$\mathcal{L} \leftarrow \{X\}$, $x^* \leftarrow \emptyset$, $u \leftarrow +\infty$;
while $\mathcal{L} \neq \emptyset$ **do**
  extract Y from $\mathcal{L}$; *node selection*
  if $Y = \{y\}$ and $f(y) < u$ **then**
    $u \leftarrow f(y)$ and $x^* \leftarrow y$;
  else if $\lambda(Y) < u$ **then**
    $\mathcal{L} \leftarrow \mathcal{L} \cup \{Y_1, Y_2\}$ where $(Y_1, Y_2) = b(Y)$; *branching*
  end if
end while
return: x^*

It is easy to show that Algorithm 7 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 8 details the branch-and-bound algorithm for MILPs.

A:Add here a theorem saying that branch and bound converges for MILP with infinitely many solutions and rational matrix, as a corollary of Meyer theorem. Say also that it is trivially true if the polyhedra is bounded, as there are finite solutions. But not true if the polyhedra is unbounded

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

Algorithm 8 Branch-and-bound algorithm (MILP)

Input: an instance of (MILP);
Output: an optimal solution $\boldsymbol{x}^*$ or $\emptyset$ if no optimal solution exist;
$\mathcal{L} \leftarrow \{\mathrm{LP}_0\}$, $\boldsymbol{x}^* \leftarrow \emptyset$, $u \leftarrow +\infty$;
while $\mathcal{L} \neq \emptyset$ **do**
  extract LP from $\mathcal{L}$; *node selection*
  solve LP;
  if LP has feasible solutions **then**
    let $\boldsymbol{x}^c$ denote an optimal solution of LP;
    if $\boldsymbol{x}^c_{[p]} \in \mathbb{Z}^p$ and $\mathrm{val}(\mathrm{LP}) < u$ **then**
      $u \leftarrow \mathrm{val}(\mathrm{LP})$ and $\boldsymbol{x}^* \leftarrow \boldsymbol{x}^c$;
    else if $\mathrm{val}(\mathrm{LP}) < u$ **then**
      let $k \in [p]$ be such that $x_k \notin \mathbb{Z}$
      $\mathcal{L} \leftarrow \mathcal{L} \cup \{\mathrm{LP}_1, \mathrm{LP}_2\}$ where LP_1 and LP_2 are respectively obtained by adding constraints $x_k \leq \lfloor \boldsymbol{x}^c_k \rfloor$ and $x_k \geq \lceil \boldsymbol{x}^c_k \rceil$ to LP; *branching*
    end if
  end if
end while
return: $\boldsymbol{x}^*$

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

$$\begin{aligned} \min \quad & \boldsymbol{c}\boldsymbol{x}, \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \\ & \boldsymbol{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p}. \end{aligned}$$

is a *perfect formulation* is the polyhedron of its linear relaxation is the convex hull of its integer solutions, that is

$$\left\{\boldsymbol{x} \in \mathbb{Z}^n \colon \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\right\} = \mathrm{conv}\left\{\mathbb{Z}^p \times \mathbb{R}^{n-p} \colon \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\right\}.$$

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

$$\begin{array}{rl} \min & \boldsymbol{c}\boldsymbol{x}, \\ \text{s.t.} & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \\ & \boldsymbol{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p}. \end{array}$$

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 8. 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

$$\min_{f \in \mathcal{F}} b_f - \boldsymbol{a}_f^T \boldsymbol{x}_f.$$

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 $\mathcal{NP}$-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

Teaching remark: teach as an exercise

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 $Q = \text{conv}\left(\{\boldsymbol{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \colon A\boldsymbol{x} \leq \boldsymbol{b}\}\right)$ 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 $Q = \text{conv}\left(\{x \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \colon Ax \leq b\}\right)$ 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 $\{x_1, x_2 \in \mathbb{Z}_+ \colon 1 \leq x_1 \leq \sqrt{3}x_2\}$ is not a polyhedron.

△

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 $B_1, \ldots, B_k$, 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 con-

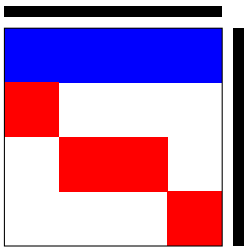


Figure 11.1: Block diagonal decomposition of a MILP

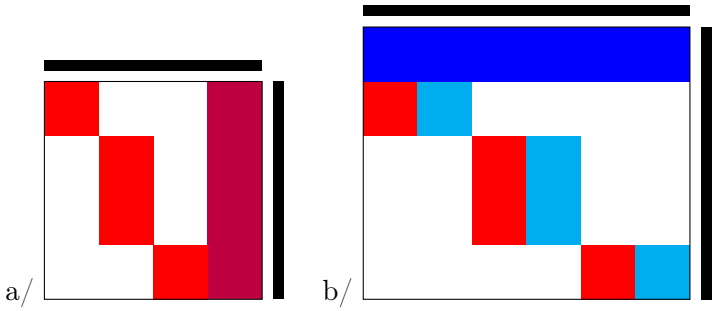


Figure 11.2: Turning linking variables into linking constraints

straints 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.

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, solv-

ing 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.

$$z_{\mathrm{I}} = \min \ \boldsymbol{cx} \tag{11.1a}$$
$$\text{s.t.} \ A_1\boldsymbol{x} = \boldsymbol{b}_1 \tag{11.1b}$$
$$A_2\boldsymbol{x} = \boldsymbol{b}_2 \tag{11.1c}$$
$$\boldsymbol{x} \in \mathbb{Z}^p_+ \times \mathbb{R}^{n-p}_+ \tag{11.1d}$$

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. *$z_{\mathrm{LR}}(\boldsymbol{\lambda}) \le z_{\mathrm{LD}} \le z_{\mathrm{I}}$ for all $\lambda \in \mathbb{R}^q$.*

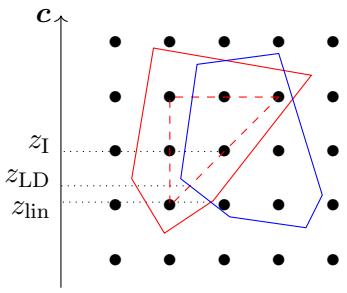


Figure 11.3: Value of the primal z_{I} and of the Lagrangian relaxation z_{LD}

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*

$$z_{\mathrm{LD}} = \min\left\{\boldsymbol{c}\boldsymbol{x} \colon \boldsymbol{x} \in \mathrm{conv}(X), A_1\boldsymbol{x} = \boldsymbol{b}_1\right\}.$$

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.

Proof of Theorem 11.2. Meyer's theorem ensures that $\mathrm{conv}(X)$ is a polyhedron, there exists rational A_2' and $\boldsymbol{b}_2'$ such that

$$\mathrm{conv}(X) = \{\boldsymbol{x} \in \mathbb{R}^n \colon A_2'\boldsymbol{x} \leq \boldsymbol{b}_2'\}$$

and hence

$$\min_{\boldsymbol{x}\in X} \boldsymbol{c}\boldsymbol{x} + \boldsymbol{\lambda}^T(A_1\boldsymbol{x} - \boldsymbol{b}_1) = \min_{\boldsymbol{x}} \left\{\boldsymbol{c}\boldsymbol{x} + \boldsymbol{\lambda}^T(A_1\boldsymbol{x} - \boldsymbol{b}_1) \colon A_2'\boldsymbol{x} \leq \boldsymbol{b}_2'\right\},$$

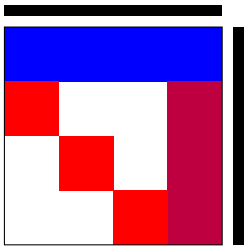


Figure 11.4: Linking constraints and variables

where the right hand side is a linear program. Applying twice linear programming strong duality, we obtain

$$\begin{aligned}
z_{\mathrm{LD}} &= \max_{\boldsymbol{\lambda}} z_{\mathrm{LR}}(\boldsymbol{\lambda}) \\
&= \max_{\boldsymbol{\lambda}} \min_{\boldsymbol{x}} \{\boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{\lambda}^T(A_1\boldsymbol{x} - \boldsymbol{b}_1) \colon A_2'\boldsymbol{x} \leq \boldsymbol{b}_2'\} \\
&= \max_{\boldsymbol{\lambda}} -\boldsymbol{\lambda}^T\boldsymbol{b}_1 + \min_{\boldsymbol{x}} \{(\boldsymbol{c}^T + \boldsymbol{\lambda}^T A_1)\boldsymbol{x} \colon A_2'\boldsymbol{x} \leq \boldsymbol{b}_2'\} \\
&= \max_{\boldsymbol{\lambda}} -\boldsymbol{\lambda}^T\boldsymbol{b}_1 + \max_{\boldsymbol{y}} \{\boldsymbol{y}^T\boldsymbol{b}_2' \colon \boldsymbol{y}_2 \leq 0 \text{ and } \boldsymbol{y}_2^T A_2' = (\boldsymbol{c}^T + \boldsymbol{\lambda}^T A_1)\} \\
&= \max_{\boldsymbol{\lambda},\boldsymbol{y}} \{ -\boldsymbol{\lambda}^T\boldsymbol{b}_1 + \boldsymbol{y}^T\boldsymbol{b}_2' \colon \boldsymbol{y}_2 \leq 0 \text{ and } \boldsymbol{y}_2^T A_2' - \boldsymbol{\lambda}^T A_1 = \boldsymbol{c}^T\} \\
&= \min_{\boldsymbol{x}} \{\boldsymbol{c}\boldsymbol{x} \colon A_1\boldsymbol{x} = \boldsymbol{b}_1 \text{ and } A_2'\boldsymbol{x} \leq \boldsymbol{b}_2\} \\
&= \min \{\boldsymbol{c}\boldsymbol{x} \colon \boldsymbol{x} \in \mathrm{conv}(X), A_1\boldsymbol{x} = \boldsymbol{b}_1\},
\end{aligned}$$

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

$$\min \left\{ \boldsymbol{c}^T\boldsymbol{x} \colon A_1\boldsymbol{x} \leq \boldsymbol{b}_1 \text{ and } \boldsymbol{x} \in \bigcap_{i \geq 2} \mathrm{conv}(X_i) \right\}.$$

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. *$\boldsymbol{\lambda} \mapsto z_{\text{LR}}(\lambda)$ is concave.*

Proof. It is an infimum of concave functions. □

The following proposition is immediate.

Proposition 11.4. *If X is finite, $\boldsymbol{\lambda} \mapsto z_{\text{LR}}(\lambda)$ 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

$$h(\boldsymbol{\mu}) - h(\boldsymbol{\lambda}) \leq \boldsymbol{p}^T(\boldsymbol{\mu} - \boldsymbol{\lambda}) \quad \text{for all } \mu \text{ in } Y.$$

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 $\boldsymbol{\lambda} \mapsto z_{\text{LR}}(\boldsymbol{\lambda})$ in $\boldsymbol{\lambda}$.*

Proof. Given such an x, we have

$$z_{\text{LR}}(\boldsymbol{\mu}) - z_{\text{LR}}(\boldsymbol{\lambda}) \leq \mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu}) - \mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu}) = (A_1\boldsymbol{x} - \boldsymbol{b}_1)^T(\boldsymbol{\mu} - \boldsymbol{\lambda}).$$

□

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 $z_{\text{LR}}(\cdot)$ 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 $z_{\text{LR}}(\cdot)$ 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-and-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 $\{a_1, \ldots, a_n\}$. The bin-packing problem can be written

$$\begin{array}{lll} \min & \sum_{j=1}^{K} z_j & \\ \text{s.t.} & \sum_{j=1}^{K} y_{ji} = 1 & i = 1, \ldots, n \\ & \sum_{i=1}^{n} a_i y_{ji} \leq W z_j & j = 1, \ldots, K \\ & y_{ji}, z_i \in \{0,1\} & i = 1, \ldots, n\,;\, j = 1, \ldots, K \end{array} \tag{11.4}$$

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

$$
\begin{array}{rlll}
z_{\mathrm{LR}}(\boldsymbol{\lambda}) := & \min & \displaystyle\sum_{j=1}^{K} z_j + \sum_{i=1}^{n} \lambda_i (\sum_{j=1}^{K} y_{ji} - 1) & \\
& \text{s.t.} & \displaystyle\sum_{i=1}^{n} a_i y_{ji} \leq W z_j & j = 1, \ldots, K \\
& & y_{ji}, z_i \in \{0,1\} & i = 1, \ldots, n\,;\, j = 1, \ldots, K
\end{array}
$$

which can be rewritten

$$
z_{\mathrm{LR}}(\boldsymbol{\lambda}) := -\sum_{i=1}^{n} \lambda_i + \sum_{j=1}^{K} S_j(\boldsymbol{\lambda})
$$

where $S_j := \min\limits_{\boldsymbol{y}, z \in \{0,1\}} \left\{ z + \sum\limits_{i=1}^{n} \lambda_i y_i : \sum\limits_{i=1}^{n} a_i y_i \leq W z \right\}$. 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

$$
\begin{aligned}
& \max \frac{1}{W} \sum_{i=1}^{n} a_i x_i \\
& \text{s.t.} \sum_{i=1}^{n} a_i x_i \leq W \\
& x_i \in \{0,1\}.
\end{aligned}
$$

We have $\alpha \leq 1$. 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

11.5.1 Traveling salesman problem

Consider the traveling salesman problem on a complete graph $G = (V, E)$ with edge costs $c(e)$ – See Chapter 17 for its definition.

1. Show that an optimal solution of the following MILP

$$\begin{array}{lll} \text{Min} & \sum_{e\in E} c(e)x_e & \\ & 0 \le x_e \le 1 & e \in E \\ & x_e \in \mathbb{Z} & e \in E \\ & \sum_{e\in\delta(v)} x_e = 2 & v \in V \\ & \sum_{e\in\delta(X)} x_e \ge 2 & X \subseteq V,\, X \neq \varnothing, V. \end{array} \tag{11.5}$$

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

$$\begin{array}{ll} 0 \le x_e \le 1 & e \in E \\ x_e \in \mathbb{Z} & e \in E \\ \sum_{e\in\delta(v)} x_e = 2 & v \in V \\ \sum_{e\in E[X]} x_e \le |X| - 1 & X \subseteq V,\, X \neq \varnothing, V. \end{array}$$

4. We now give a special role to vertex 1. Show that the constraints of the MILP can be rewritten

$$\begin{array}{ll} 0 \le x_e \le 1 & e \in E \\ x_e \in \mathbb{Z} & e \in E \\ \sum_{e\in\delta(1)} x_e = 2 & \\ \sum_{e\in E[X]} x_e \le |X| - 1 & X \subseteq \{2, \ldots, n\},\, X \neq \varnothing \\ \sum_{e\in E} x_e = n & \\ \sum_{e\in\delta(v)} x_e = 2 & v \in \{2, \ldots, n\}. \end{array}$$

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 $v \in \{2, \ldots, n\}$, 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

$$\begin{array}{ll} 0 \leq x_e \leq 1 & e \in E \\ \sum_{e \in E[X]} x_e \leq |X| - 1 & X \subseteq \{2, \ldots, n\},\ X \neq \varnothing \\ \sum_{e \in E \backslash \delta(1)} x_e = n - 2 & \end{array}$$

is a perfect formulation of the spanning tree polytope (conforti p.154). Geoffrion's theorem enables to conclude. □

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 9 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 9, 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 10 Simulated annealing

Input: initial solution x_0, initial temperature $T^{\max}$, parameter α
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^{b}

Algorithm 10 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%.
- 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 11.

Algorithm 11 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^{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 16.

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 12 introduces a prototype of evolutionary algorithm.

Algorithm 12 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 $g : \mathcal{X} \times \mathcal{X} \to \mathcal{X}$, 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

Managing an Operations Research project in the industry

Chapter 14

Bin packing.tex

Chapter 15

Facility location

Chapter 16

Network design

Chapter 17

Routing

Chapter 18

Scheduling

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.

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.

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.

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

English French

matching couplage.

FrenchEnglish

couplage matching.