Foundations of Computer Science
Logic, models, and computations

Course CSC_41012_EP

of l'Ecole Polytechnique

Olivier Bournez

bournez@lix.polytechnique.fr

Version of August 20, 2024

Contents

Chapter 1

Introduction

Objectives

This course is about algorithms and their efficiency.

More precisely, the objective of this course is to answer to the following questions: What are the limits of algorithms, and of today's computers?

Algorithm?

The word "algorithm" comes from the name of mathematician Al-Khwârizmî (Latinised at middle age as Algorithmi), who at 9th century wrote several books on the resolution of equations. We will discuss the notion of algorithm, and the notion of problem solvable by an algorithm, or of function computed by some algorithm.

We will first prove that there are problems that cannot be solved by an algorithm.

Out of the problems that admit a solution by an algorithm, we will then try to determine those which admit a solution with reasonable resources: We will discuss the resources (time, memory, etc) necessary to solve a problem.

Thanks The author of this document would like to thank strongly Stefan Mengel for many comments on previous versions of this document. I also would like to thanks warmly the students of the École Polytechnique Bachelor course CSE-304 for the year 2019-2020 for comments and feedback. Some special thanks to Louis de Benoist De Gentissart, Agathe De Vulpian, Guillaume Lainé and Skander Moalla for some detailed feedback, or bugs about previous versions of some of the chapters of this document, or about related slides.

Some parts of this document are very strongly inspired from a French version, that has been used for the course INF423, and then INF412 at École Polytechnique. The author of this document would like to thank strongly Johanne Cohen, Bruno Salvy and David Monniaux for their comments on preliminary versions of this latter document in French. I also thank the promotions 2011-2012, 2012-2013, 2013-2014,

2014-2015, 2015-2016, 2016-2017, 2017-2018, 2018-2019, 2019-2020, 2020-2021, 2021-2022, 2022-2023 of École Polytechnique for their feedback on INF423 and then INF412. Some special thanks to Louis Abraham, Sariah Al Saati, Olivier Bailleux, Juliette Buet, Ismaël Cahu, Carlo Ferrari, Léo Gaspard, Estienne Granet, Pierre-Jean Grenier, Roberto Moura, Alexis Le Dantec, Denis Langevin, Emmanuel Lazard, Stéphane Lengrand, Arnaud Lenoir, Louis-François Rigano, Louis Rustenholz, Matthieu Vermeil, and Zigfrid Zvezdin, for some detailed feedback, with precise suggestions of improvement, or for having pointed out some problems about preliminary versions of previous French versions of some parts of this document. Thanks also to Romain Cosson and Rodrigue Lelotte for feedback on corrections of previous exams for INF423 and INF412.

This document is still in some non-perfect form.

All comments (even language, typographic, orthographic, etc) on this document are welcome and should be be sent to bournez@lix.polytechnique.fr.

On the exercises Some of the exercises are corrected. The solutions are found and the end of the document in a chapter devoted to the solutions. The exercises marked with a star require more thought.

1.1 Mathematical concepts

1.1.1 Sets, Functions

Let E be a set and e an element. We write $e \in E$ to mean that e is an element of set E. If A and B are two sets, we write $A \subset B$ to mean that every element of A is an element of B. We say in that case that A is a subset of B. When E is a set, the collection of all the subsets of E constitutes a set, called the *power set of* E, that we will denote by $\mathscr{P}(E)$. We will write $A \cup B$, $A \cap B$ for respectively the *union* and *intersection* of the sets A and B. When A is a subset of E, we will write A^c for the *complement* of A in E.

Exercise 1.1 *Let A, B be two subsets of E. Prove the Morgan laws: $(A \cup B)^c = A^c \cap B^c$ and $(A \cap B)^c = A^c \cup B^c$.*

Exercise 1.2 *Let A, B, C three subsets of E. Prove that $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ and $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.*

Exercise 1.3 *(solution on page 227) Let A, B, C three subsets of E. Prove that $A \cap B^c = A \cap C^c$ if and only if $A \cap B = A \cap C$.*

We call *Cartesian product* of two sets E and F, denoted by $E \times F$, the set of all the pairs made of an element of E and of an element of F:

$$E \times F = \{(x, y) | x \in E \text{ and } y \in F\}.$$

Given some integer $n \geq 1$, we write $E^n = E \times \cdots \times E$ for the Cartesian product of E by itself n times: E^n can also be defined[1] recursively by $E^1 = E$, and $E^{n+1} = E \times E^n$.

Intuitively, a *application* f from a set E to a set V is an object which associates to every element e of a set E a unique element $f(e)$ in F. Formally, a function f (one also talks of *partial function*) from a set E to a set F is a subset Γ of $E \times F$, such that for all $x \in E$ there is at most one $y \in F$ with $(x, y) \in \Gamma$. Its *domain* is the set of the $x \in E$ such that $(x, y) \in \Gamma$ for a certain $y \in F$. Its *image* is the set of the $y \in F$ such that $(x, y) \in \Gamma$ for a certain $x \in E$. An *application* f (this is also called a *total function*) from a set E to a set F is a function whose domain is E.

A *family* $(x_i)_{i \in I}$ *of elements of a set* X is some application from a set I to X. I is called the *set of indices* and the image by its application of element $i \in I$ is denoted x_i.

The Cartesian product generalizes to a family of sets:

$$E_1 \times \cdots \times E_n = \{(x_1, \ldots, x_n) | x_1 \in E_1, \cdots, x_n \in E_n\}.$$

The union and intersection generalize to some arbitrary family of subsets of a set E. Let $(A_i)_{i \in I}$ be a family of subsets of E.

$$\bigcup_{i \in I} A_i = \{e \in E | \exists i \in I \; e \in A_i\};$$

$$\bigcap_{i \in I} A_i = \{e \in E | \forall i \in I \; e \in A_i\}.$$

Exercise 1.4 *Let A be a subset of E, and $(B_i)_{i \in I}$ a family of subsets of E. Prove the two following equalities:*

$$A \bigcup \left(\bigcap_{i \in I} B_i \right) = \bigcap_{i \in I} (A \cup B_i)$$

$$A \bigcap \left(\bigcup_{i \in I} B_i \right) = \bigcup_{i \in I} (A \cap B_i)$$

We will write $\mathbb{N}$ for the set of natural integers, $\mathbb{Z}$ for the set of (positive, null, or negative) integers, $\mathbb{R}$ for the set of reals, and $\mathbb{C}$ for the set of complex numbers. $\mathbb{Z}$ is a *ring*. $\mathbb{R}$ and $\mathbb{C}$ are *fields*. We will write $\mathbb{R}^{>0}$ for the set of non-negative reals.

[1] There is a bijection between the objects defined by the two definitions

1.1.2 Alphabets, Words, Languages

We now recall some basic definitions about *words* and *languages*. The terminology, borrowed from linguistics, remind that historically first works on the concepts of formal languages were on the modeling of natural language.

A finite set Σ is fixed: In this context, such a set is also called an *alphabet*. and the elements of Σ are called *letters* or *symbols*.

Example 1.1

- $\Sigma_{bin} = \{0,1\}$ *is the* binary alphabet.
- $\Sigma_{Latin} = \{A,B,C,D,\ldots,Z,a,b,c,d,\ldots,z\}$ *is the alphabet which consists of the letters of the Latin alphabet.*
- $\Sigma_{number} = \{0,1,2,\cdots,9\}$ *is the alphabet which consists of digits in radix* 10.
- *The set of the printable ASCII*[a] *characters, or set of printed characters is an alphabet, that one can write* Σ_{ASCII}.
- $\Sigma_{exp} = \{0,1,2,\cdots,9,+,-,*,/,(,)\}$ *is the alphabet of arithmetic expressions.*

[a] We will not go here to discussions about whether this is precisely what is called the ASCII characters in all generality. We assume Σ_{ASCII} is the set of symbols that can be printed with a keyboard of a computer. It contains symbols such as é, ö, etc. Actually the original 7-bit version of ASCII did it fact not contain accents and those were added later on and there were lots of different incompatible version for different languages, and we do not intend in this document to go to these discussions: For us, it contains symbols that can be printed with a keyboard of a computer.

A *word* w on alphabet Σ is a finite sequence $w_1 w_2 \cdots w_n$ of letters (i.e. elements) of the alphabet Σ. The integer n is called the *length* of word w. It will be denoted length(w).

Example 1.2

- 10011 *is a word on alphabet* Σ_{bin} *of length* 5.
- 9120 *is a word on alphabet* Σ_{number}*,but not a word on the alphabet* Σ_{bin}.
- $Bonjour$ *is a word of length* 6 *on alphabet* Σ_{Latin}*;* $azrddfb$ *is also a word of length* 7 *on the same alphabet.* $;-)$ *is not a word on this alphabet, since the symbol* ; *is not in the alphabet* Σ_{Latin} *defined above.*
- $Student$, $Elephant$ *and* $\pounds\pounds z'!!!"$ *are words on the alphabet* Σ_{ASCII}.
- $243+(5*(1+6))$ *is a word on alphabet* Σ_{exp}.
- $24*((((5/+)/)+$ *is a word on alphabet* Σ_{exp}.

A *language* on alphabet Σ is a set of words on alphabet Σ. The set of all the words on alphabet Σ is denoted by Σ^*. The empty word ϵ is the unique word of length 0. The empty word is a particular word: Similarly to what happens for any other word, It is possible that a language contains the empty word (which is a particular word), or that a language doesn't contain the empty word. Σ^* contains by definition the empty word.

Example 1.3
- $\{0,1\}^*$ *denotes the set of words over alphabet* $\Sigma_{bin} = \{0,1\}$. *For example,* $00001101 \in \{0,1\}^*$. *We have also* $\epsilon \in \{0,1\}^*$.
- $\{hello, goodbye\}$ *is a language on* Σ_{Latin}. *This language contains two words.*
- *The set of words of English dictionary is a language on the alphabet* Σ_{Latin}.
- *The set of words of French dictionary is a language on the alphabet* Σ_{ASCII}, *since a word such as élève can be written using accentuated letters.*
- *The set of the phrases of this document is a language on the alphabet of ASCII characters. Note that the character " ", that is to say the blank (space) character, used to separate the words in a sentence is a particular character of ASCII alphabet.*
- ${\Sigma_{exp}}^*$ *contains words such as* $24 * ((((5/+)//+$ *which is not a valid arithmetic expression. The set of words which are valid arithmetic expressions, such as* $5 + (2 * (1 - 3) * 3)$, *is a particular language on alphabet* Σ_{exp}.

One then defines an operation of *concatenation* on words: The concatenation of word $u = u_1 u_2 \cdots u_n$ and of word $v = v_1 v_2 \cdots v_m$ is the word denoted by $u.v$ defined by

$$u_1 u_2 \cdots u_n v_1 v_2 \cdots v_m,$$

that is to say the words whose letters are obtained by appending the letters of v after those of u. The operation of concatenation denoted by . is associative, but not commutative. The empty word is a right and left neutral element for this operation. Σ^* is also called the free *monoid* on alphabet Σ (since the operation of concatenation provides a structure of monoid.

We will also write uv for the concatenation $u.v$. Actually, every word $w_1 w_2 \cdots w_n$ can be seen as $w_1.w_2 \cdots .w_n$, where w_i represents the word of length 1 consisting only of the letter w_i. This interpretation of letters as words of length 1 is often very useful.

Example 1.4 *If* Σ *is the set* $\{a,b\}$, *then* $aaab$ *is the word of length* 4 *whose first three letters are a, and the last is b. It is also the concatenation of four words of length one: a, a, a and b.*

When i is some integer, and w is a word, we write w^i for the word obtained by concatenating the word w i times: If you prefer, w^0 is the empty word ϵ, and $w^{i+1} = w^i w = w w^i$ for every integer i.

Example 1.5 *By interpreting letters as words of length* 1, $aaabbc$ *can also be written* $a^3 b^2 c$.

A word u is some *prefix* of a word w, if there exists a word z such that $w = u.z$. This is a *proper!prefix* if $u \neq w$. A word u is a *suffix* of a word w if there exists some word z such that $w = z.u$. This is a *proper!suffix* if $u \neq w$.

1.1.3 Change of alphabet

It is often useful to rewrite a word on a given alphabet into a word on some other alphabet. For example, in computer science one often needs to code in binary, that is to say with alphabet $\Sigma = \{0,1\}$.

One way to change the alphabet is to proceed one letter after the other.

Example 1.6 *If Σ is the alphabet $\{a,b,c\}$, and $\Gamma = \{0,1\}$, then one can encode Σ^* in Γ^* by the function h such hat $h(a) = 01$, $h(b) = 10$, $h(c) = 11$. The word $abab$ is then encoded by $h(abab) = 01100110$, that is to say by the word encoded by coding letter after letter.*

Very formally, given two alphabets Σ and Γ, an *homomorphism* is an application from Σ^* into Γ^* such that

- $h(\epsilon) = \epsilon$
- $h(u.v) = h(u).h(v)$ for every words u and v.

Obviously, every homomorphism is perfectly determined by its image on the letters of Σ. It then extends to words of Σ^* by

$$h(w_1 w_2 \cdots w_n) = h(w_1).h(w_2).\ldots.h(w_n)$$

for every word $w = w_1 w_2 \cdots w_n$.

1.1.4 Graphs

A *graph* $G = (V,E)$ consists of a set V, whose elements are called *vertices* and a subset of $E \subset V \times V$, whose elements are called *arcs*. In some books, vertices are called *nodes*.

If the arcs are undirected, that is say, if one assumes that every time that there is the arc (u,v) there is also the arc (v,u), one says that the graph G is *undirected* and the elements of E are called *edges*.

By default, all considered graphs will all be undirected. An edge will then be denoted uv or $\{uv\}$.

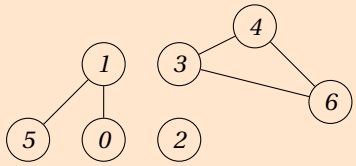
When there is an edge between u and v, that is to say when $\{u,v\} \in E$, one says that u and v are neighbours. The *degree of a vertex u* is the number of its neighbours.

A *path* from s to t is a sequence $(s = s_0, \ldots, s_n = t)$ of vertices such that, for all $1 \leq i \leq n$, (s_{i-1}, s_i) is an arc. A *simple path* is a path that does not go twice through the same vertex, i.e. $s_i \neq s_j$ for $i \neq j$. Its origin is the vertex $s = s_0$. Its end is the vertex $s_n = t$. A *circuit* is a path of non-null length whose origin coincides with its end, i.e. $s_0 = s_n$.

Example 1.7 *The (undirected) graph $G = (V,E)$ with*

- $V = \{0,1,\ldots,6\}$
- $E = \{(0,1),(3,4),(5,1),(6,3),(6,4)\}$.

is represented as below.

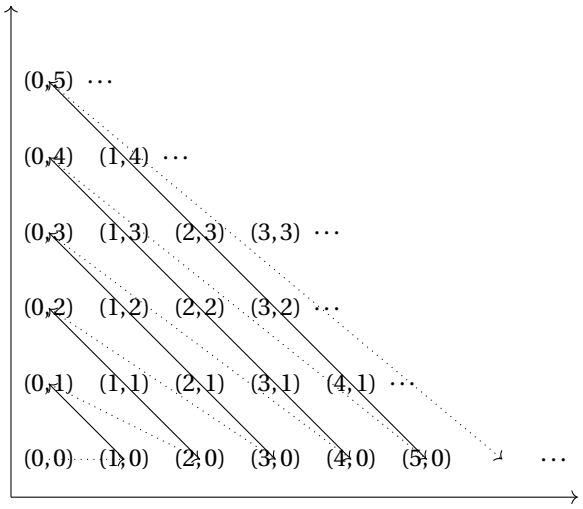


A graph is said to be *connected* if any two vertices are connected by a path.

Example 1.8 *The graph of Example 1.7 is not connected since there is no path between vertices* 1 *and* 6.

1.2 The diagonalisation method

Remember that $\mathbb{N}^2 = \mathbb{N} \times \mathbb{N}$ is *countable*: It is possible to build a bijection between $\mathbb{N}$ and $\mathbb{N}^2$. Below, we illustrate one way of running through all the pairs of integers, in order to realize a bijection between $\mathbb{N}$ and $\mathbb{N}^2$.

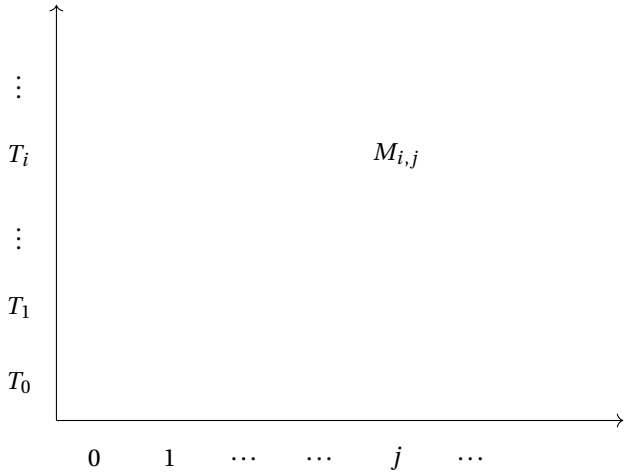


Exercise 1.5 *(solution on page 227) Prove formally that* $\mathbb{N} \times \mathbb{N}$ *is countable by giving the bijection* $f : \mathbb{N}^2 \to \mathbb{N}$ *of the above figure.*

By contrast, the set of subsets of $\mathbb{N}$ is not countable: This can be shown with the *diagonalisation method* due to Cantor.

The reasoning is as follows: Suppose for contradiction that we can enumerate the subsets of $\mathbb{N}$. Then write these subsets as T_1, T_2, $\ldots T_n$ $\ldots$. Every subset T_i of

$\mathbb{N}$ can be seen as the row i of an (infinite) matrix $M = (M_{i,j})_{i,j}$ whose entries are in $\{0,1\}$ and whose element $M_{i,j}$ is 1 if and only if element j is in the ith subset of $\mathbb{N}$.



We consider then the subset T^* obtained by "*inverting the diagonal of M*". Formally, one considers $T^* = \{j | M_{j,j} = 0\}$. This subset of $\mathbb{N}$ is not among the enumeration, since otherwise it would have some index j_0: if $j_0 \in T_{j_0} = T^*$, then we should have $M_{j_0,j_0} = 1$ by definition of M, and $M_{j_0,j_0} = 0$ by definition of T^*, which is impossible. If $j_0 \notin T^*$, then we should have $M_{j_0,j_0} = 0$ by definition of M, and $M_{j_0,j_0} = 1$ by definition of T^*, which is again impossible.

This argument is at the basis of various reasoning in computability theory, as we will see.

Exercise 1.6 *Prove that the set of sequences $(u_n)_{n\in\mathbb{N}}$ with values in $\{0,1\}$ is not countable.*

Exercise 1.7 *Prove that the set $\mathbb{R}$ of real numbers is not countable.*

Chapter 2

Recurrence and induction

2.1 Motivation

The *recursive definitions* are ubiquitous in computer science. There are present both in programming languages, but also in many concepts what we consider in computer science.

Example 2.1 (Lists in JAVA) *In JAVA, with*

```
class List {
  int content;
  List next;
  }
List lst;
```

*the class List is defined in a recursive (*inductive*) manner: by using in the definition of the class, the field "next' whose type is the class List itself.*

Example 2.2 (Ranked trees) *We have defined the ranked trees in the previous chapter by using the notion of graph. A natural alternative would be to describe the ranked trees through a recursive definition: A ranked tree is either empty, or reduced to a vertex (a root), or made of a vertex (a root) and a (ranked) list of ranked trees (its sons).*

In this chapter, we spent some time on the *recursive definitions* of sets and functions. This will be used to give some clean meaning to recursive definitions in next chapters.

We will furthermore define in this chapter how it is possible to do some proofs on structures defined inductively, by introducing the *proofs by (structural) induction*.

2.2 Reasoning by recurrence over $\mathbb{N}$

The *structural induction* is a generalization of the *proof by recurrence*: Let's come back first to this later to have clear ideas.

When reasoning on the integers, the *first principle of induction*, also called *principle of mathematical recurrence* is a reasoning mode particular useful.

Theorem 2.1 *Let $P(n)$ be* predicate *(a property) depending on the integer n. If the the two following conditions are satisfied:*

(B) *$P(0)$ is true;*

(I) *$P(n)$ implies $P(n+1)$ for all n;*

then for all integer n, $P(n)$ is true.

Proof: The reasoning is done by contradiction. Consider

$$X = \{k \in \mathbb{N} | P(k) \text{ is false}\}.$$

If X is non-empty, it admits some least element n. From condition, (B), $n \neq 0$, and hence $n-1$ is some integer, and $P(n-1)$ is true by definition of X. We then get a contradiction with the property (I) applied for integer $n-1$. □

To do a *proof by recurrence*, we prove a property for 0 (basis case), and we prove that the property is *hereditary*, or *inductive*: $P(n)$ implies $P(n+1)$ for all n.

The concept of *inductive proof* generalises this idea to other sets than the integers, namely to sets that are defined inductively.

Exercise 2.1 *Consider $S_n = 1^3 + 3^3 + \cdots + (2n-1)^3$. Prove by recurrence that $S_n = 2n^4 - n^2$.*

Exercise 2.2 *Prove by recurrence that $\sum_{k=1}^{n} \frac{1}{4k^2-1} = \frac{n}{2n+1}$.*

Exercise 2.3 *(solution on page 227) The theorem above is sometimes called the "first principle of induction". Prove the "second principle of induction": Let $P(n)$ be a property depending on integer n. If the following property is satisfied: For all $n \in \mathbb{N}$, if assuming that for all $k < n$ the property $P(k)$, one can deduce $P(n)$, then for all $n \in \mathbb{N}$, the property $P(n)$ is true.*

Exercise 2.4 *(solution on page 228) An alphabet Σ is fixed. Recall that a language over Σ is a subset of Σ^*. If L_1 and L_2 are two languages of Σ^*, their concatenation is defined by $L_1.L_2 = \{u.v | u \in L_1, v \in L_2\}$. The concatenation is an associative operation that admits $\{\epsilon\}$ as neutral element. One can then define the powers of a language L in the following way: $L^0 = \{\epsilon\}$, and for all integer $n > 0$, $L^{n+1} = L^n.L = L.L^n$. The star of a language L is defined by $L^* = \bigcup_{n\in\mathbb{N}} L^n$.*

Let L and M two languages over Σ, with $\epsilon \notin L$. Prove that in $\mathscr{P}(\Sigma^*)$ (the languages over alphabet Σ), the equation $X = L.X \cup M$ admits for unique solution the language $L^*.M$.

2.3 Inductive definitions

The inductive definitions aims at defining some subsets of a set E.

Remark 2.1 *This remark is for purists. It can be avoided in a first reading of this document.*

We restrict in this document to the framework where one wants to define by induction some objects that correspond to subsets to an already known set E. We avoid this to avoid the subtleties and the paradoxes of the set theory.

The very attentive reader will observe that we will often consider the syntactic writing of some objects more than the objects themselves. Indeed, by doing so, we guarantee that we are living in a set $E = \Sigma^*$ for a certain alphabet Σ, and we avoid to have to worry about the existence of the set E in the following reasoning's.

For example, to formalise completely the Example 2.1 above, we would try to define some syntactic representation of lists instead of lists.

When one wants to define a set, or a subset of a set, one way to do it is by giving some *explicit definition*, that is by describing precisely which are its elements.

Example 2.3 *The even integers can be defined by $P = \{n | \exists k \in \mathbb{N}\ n = 2 * k\}$.*

Unfortunately, this is not always as easy. It is often easier to define a set by some *inductive definition*. A typical example of *inductive definition* is a definition like this one:

Example 2.4 *The even integers also correspond to the least set that contains 0 and such that if n is even, then $n+2$ is even.*

Remark 2.2 *Observe that the set of integers $\mathbb{N}$ satisfies that 0 is some integer, and that if n is some integer, then so is $n+2$. It is hence necessary to say that this is*

the least set with this property.

2.3.1 General principle of an *inductive definition*

Intuitively, a subset X is defined inductively if it can be defined from some explicitly given elements of X, and a mean to construct some new elements of X from elements of X.

More generally, in an inductive definition,

- some elements of the set X are explicitly given (that is to say, a set B of elements b of X). They correspond to the *base set* of the inductive definition;
- The other elements of the set X are defined, as a function of elements that already belong to the set X, according to some rules: that is to say we are given a set of rules R for the formation of new elements. This constitutes the *inductive steps* of the inductive definition.

One considers then the least set that contains B and that is *stable* (one says also *closed*) by the rules of R.

2.3.2 Formalisation: First fix point theorem

Formally, all of this is justified by the following theorem.

Definition 2.1 (Inductive definition) *Let E be a set. An* inductive definition *of a subset X of E consists of:*

- *a non-empty subset B of E (called the* base set*)*
- *and a set of* rules *R: each rule $r_i \in R$ is a function (possibly partial) r_i from $E^{n_i} \to E$, for some integer $n_i \geq 1$.*

Theorem 2.2 (Fix point theorem) *To a* inductive definition *corresponds a least set that satisfies the following properties:*

(B) *it contains B: $B \subset X$;*

(I) *it is stable by the rules rules of R: for every rule $r_i \in R$, for every $x_1, \cdots, x_{n_i} \in X$, we have $r_i(x_1, \cdots, x_{n_i}) \in X$.*

One says that this set is inductively defined.

Proof: Let $\mathscr{F}$ be the set of subsets of E satisfying (B) and (I). The set $\mathscr{F}$ is non empty as it contains at least one element: Indeed, the set E satisfies the conditions (B) and (I) and hence $E \in \mathscr{F}$.

We can then consider X defined as the intersection of all the elements of $\mathscr{F}$. Formally:

$$X = \bigcap_{Y \in \mathscr{F}} Y. \tag{2.1}$$

Since B is included in each $Y \in \mathscr{F}$, B is included in X. So X satisfies the condition (B).

The obtained set satisfies also (I). Indeed, consider a rule $r_i \in R$, and some $x_1, \cdots, x_{n_i} \in X$. We have $x_1, \cdots, x_{n_i} \in Y$ for every $Y \in \mathscr{F}$. For every such Y, since Y is stable by the rule r_i, we must hat $r(x_1, \cdots, x_{n_i}) \in Y$. Since this is true for every $Y \in \mathscr{F}$, we also have $r(x_1, \cdots, x_{n_i}) \in X$, which proves that X is stable by the rule r_i.

X is the least set that satisfies the conditions (B) and (I), since it is by definition included in every other set that satisfies the conditions (B) and (I).

□

2.3.3 Various notations of an inductive definition

Notation 2.1 *We often denote some* inductive definition *using the notation*

(B) $x \in X$

with a line like this one for every $x \in B$

(possibly one writes $B \subset X$ *);*

(I) $x_1, \cdots x_{n_i} \in X \Rightarrow r_i(x_1, \cdots, x_{n_i}) \in X$

with such a rule for every rule $r_i \in R$.

Example 2.5 *According to this convention, the inductive definition of even integers (of the Example 2.4) is denoted by:*

(B) $0 \in P$;

(I) $n \in P \Rightarrow n+2 \in P$.

Example 2.6 *Let* $\Sigma = \{(,)\}$ *be the alphabet made of the open parenthesis and of the closing parenthesis. The set* $D \subset \Sigma^*$ *of well founded parenthesising, called the Dyck language, is defined inductively by*

(B) $\epsilon \in D$;

(I) $x \in D \Rightarrow (x) \in D$;

(I) $x, y \in D \Rightarrow xy \in D$.

Notation 2.2 *One sometimes prefers to write some inductive definition as* deduction rules*:*

$$\frac{}{B \subset X} \qquad \frac{x_1 \in X \quad \ldots \quad x_{n_i} \in X}{r_i(x_1, \ldots, x_{n_i}) \in X}$$

The principle of such a notation is that an horizontal line — means some deduction rule. *What is written above the line is some* hypothesis. *What is written under the line is some conclusion. If what is above is empty this means that the conclusion is true without any hypothesis.*

Notation 2.3 *We sometimes write also directly:*

$$\frac{}{b} \qquad \frac{x_1 \quad \dots \quad x_{n_i}}{r_i(x_1,\dots,\ x_{n_i})}$$

for every $b \in B$,
or

$$\frac{}{b \in B} \qquad \frac{x_1 \quad \dots \quad x_{n_i}}{r_i(x_1,\dots,\ x_{n_i})}$$

or even:

$$\frac{}{B} \qquad \frac{x_1 \quad \dots \quad x_{n_i}}{r_i(x_1,\dots,\ x_{n_i})}$$

2.4 Applications

2.4.1 A few examples

Example 2.7 ($\mathbb{N}$) *The subset X of $\mathbb{N}$ defined inductively by*

$$\frac{}{0} \qquad \frac{n}{n+1}$$

is nothing but the whole set $\mathbb{N}$ of the integers.

Example 2.8 (Σ^*) *The subset X of Σ^*, where Σ is an alphabet, defined inductively by*

(*B*) $\epsilon \in X$;

(*I*) $w \in X \Rightarrow wa \in X$, *for every* $a \in \Sigma$;

is nothing but the whole set Σ^*.

Example 2.9 (Language $\{a^n bc^n\}$) *The language L on the alphabet $\Sigma = \{a, b, c\}$ of words of the form $a^n bc^n$, $n \in \mathbb{N}$, is defined inductively by*

(*B*) $b \in L$;

(*I*) $w \in L \Rightarrow awc \in L$.

Exercise 2.5 *(solution on page 228) Define inductively the set of well parenthesised expressions formed from identifiers taken in a set A and using operators $+$ and $\times$.*

2.4.2 Labeled binary trees

Let's recall here the text of the course INF421 (version 2010-2011): "the notion of *binary tree* is rather different from the notion of *free tree* and *ranked tree*. A *binary tree* on a finite set of vertices is either empty, or the disjoint union of a vertex, called its root„ of a binary tree, called its *left sub-tree*, and of a binary tree, called its *right sub-tree*. It is useful to represent such a binary tree on the form of a triplet $A = (A_g, r, A_d)$."

We obtain immediately an *inductive definition* of *labeled binary trees* from this text.

Example 2.10 (Labeled binary trees) *The set AB of labeled binary trees on the set A is the subset of Σ^*, where Σ is the alphabet $\Sigma = A \cup \{\emptyset, (,), ,\}$, defined inductively by*

(B) $\emptyset \in AB$;

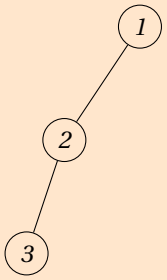
(I) $g, d \in AB \Rightarrow (g, a, d) \in AB$, *for every* $a \in A$.

Remark 2.3 *In the expression above, (g, a, d) denotes the concatenation of the word of length 1 (, of word g, of word , of length 1, of word a, of word , of length 1, of word d and of mot) of length 1. All these words are indeed words over the alphabet Σ that contains all the required symbols.*

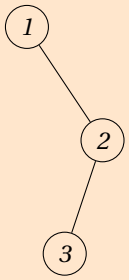
Remark 2.4 *$g, d \in AB \Rightarrow (g, a, d) \in AB$, for every $a \in A$ denotes the fact that one repeats, for every $a \in A$, the rule $g, d \in AB \Rightarrow (g, a, d) \in AB$. This is actually not a rule, but a family of rules: one for each element a of A.*

Remark 2.5 *Be careful: a* binary tree *is not a* ranked tree *such that all the nodes are of arity at most* 2.

Example 2.11 *For example, the* labeled binary tree



and the labeled binary tree



are not the same, since the first corresponds to the word $(((\emptyset,3,\emptyset),2,\emptyset),1,\emptyset)$ *and the second to the word* $(\emptyset,1,((\emptyset,3,\emptyset),2,\emptyset))$. *However, if these trees are considered as ranked trees, they are the same.*

Exercise 2.6 *(solution on page 228) Let A be an alphabet. One defines recursively the sequence of sets $(AB_n)_{n\in\mathbb{N}}$ by*

- $AB_0 = \{\emptyset\}$.
- $AB_{n+1} = AB_n \cup \{(a,g,d) | a \in A, g,d \in AB_n\}$

Prove that $X = \bigcup_{n\in\mathbb{N}} AB_n$ corresponds also to the set AB of labeled binary trees for the set A.

2.4.3 Arithmetic expressions

One can define the well formed arithmetic expression on the alphabet Σ_{exp} of example 1.1. Recall that we have defined the alphabet

$$\Sigma_{exp} = \{0,1,2,\cdots,9,+,-,*,/,(,)\}.$$

Let's start by defining what is a number in radix 10. Usually, one doesn't write an integer in radix 10 by starting by a 0 (except for 0). For example, 000192 is not authorized. By opposition, 192 is a valid expression.

We obtain the following inductive definition.

Example 2.12 *The set $\mathcal{N}$ of non-null integers written in radix* 10 *is the subset of $\Sigma_{exp}{}^*$, defined inductively by*

(*B*) $a \in \mathcal{N}$ *for each* $a \in \{1,2,\ldots,9\}$;

(*I*) $g \in \mathcal{N} \Rightarrow ga \in \mathcal{N}$, *for each* $a \in \{0,1,2,\ldots,9\}$.

One can then defined the arithmetic expression in the following way:

Example 2.13 *The set $Arith$ of* arithmetic!expressions *is the subset of $\Sigma_{exp}{}^*$, defined inductively by*

(*B*) $0 \in Arith$;

(*B*) $\mathcal{N} \subset Arith$;

(*I*) $g, d \in Arith \Rightarrow g + d \in Arith$;

(*I*) $g, d \in Arith \Rightarrow g * d \in Arith$;

(*I*) $g, d \in Arith \Rightarrow g / d \in Arith$;

(*I*) $g, d \in Arith \Rightarrow g - d \in Arith$;

(*I*) $g \in Arith \Rightarrow (g) \in Arith$;

For example, we have $(1 + 2 * 4 + 4 * (3 + 2)) \in Arith$ that corresponds to some valid expression. By opposition, $+1 - /2($ is not in $Arith$.

2.4.4 Terms

Les *terms* are particular labeled ranked trees. They play an essential role in many structures in computer science.

Let $F = \{f_0, f_1, \cdots, f_n, \cdots\}$ be a set of symbols, called *function symbols*. To each such symbol f is associated some integer $a(f) \in F$, that is called its *arity*, and this represents the number of arguments of function symbol f. one writes F_i for the subset of symbols of functions of arity i. The function symbols of arity 0 are called *constants*.

Let Σ the alphabet $\Sigma = F \cup \{(,), ,\}$ constituted of F, of the opening parenthesis, the closing parenthesis, and of comma.

Definition 2.2 (Terms over F) *The set T of terms built over F is the subset of Σ^* defined inductively by:*

(*B*) $F_0 \subset T$
(that is to say: the constants are some terms)

(*I*) $t_1, t_2, \cdots, t_n \in T \Rightarrow f(t_1, t_2, \cdots, t_n) \in T$
for every integer n, for every symbol $f \in F_n$ of arity n.

Remark 2.6 *In the definition above, we are indeed talking about words over the alphabet Σ: $f(t_1, t_2, \cdots, t_n)$ denotes the word whose first letter is f, the second $($, the following the ones of t_1, etc.*

Example 2.14 *For example, we can fix $F = \{0, 1, f, g\}$, with 0 and 1 of arity 0 (these are constants), f of arity 2 and g of arity 1. Then $f(0, g(1))$ is a term over F. $f(g(g(0)), f(1, 0))$ is a term over F. $f(1)$ is not a term over F.*

The terms over F correspond to particular ranked labeled trees: The nodes are labeled by symbols of functions from F, and a node labeled by a symbol of arity k has exactly k sons.

2.5 Proofs by induction

We will need regularly to prove some properties on a the elements of a set X defined inductively. This turns out to be possible by using what is called a *proof!by (structural) induction*, sometimes also called *proof!by (structural) induction*, which generalises the principle of the proof by recurrence.

Theorem 2.3 (Proof by induction) *Let $X \subset E$ be a set defined inductively from a base set B and some rules R. Let $\mathscr{P}$ be a* predicate *expressing some property of an element $x \in E$: That is to say a property $\mathscr{P}(x)$ that is either true or false in a given element $x \in E$.*

If the following conditions are satisfied:

(B) *$\mathscr{P}(x)$ is satisfied for every element $x \in B$;*

(I) *$\mathscr{P}$ is* hereditary, *that is to say stable by the rules of R: Formally, for every rue $r_i \in R$, for every $x_1, \cdots, x_{n_i} \in E$, $\mathscr{P}(x_1), \cdots, \mathscr{P}(x_{n_i})$ true implies $\mathscr{P}(x)$ true in $x = r_i(x_1, \cdots, x_{n_i})$.*

Then $\mathscr{P}(x)$ is true for every element $x \in X$.

Proof: Consider the set Y of elements $x \in E$ that satisfy the property $\mathscr{P}(x)$. Y contains B by the property (B). Y is stable by the rules of R by the property (I). The set X, that corresponds to the least set that contains B and that is stable by the rules of R, is hence included in Y. □

Remark 2.7 *The proof by induction indeed generalises the proof by recurrence; Indeed, $\mathbb{N}$ is defined inductively as in Example 2.7. A proof by induction on this inductive definition of $\mathbb{N}$ corresponds to a proof by recurrence, that is to say to hypotheses of Theorem 2.1.*

Example 2.15 *To prove by induction that all the words of the language defined inductively in Example 2.9 have as many a's as c's, it is sufficient to observe that this is true for the word reduced to a b: Indeed, this is 0 times the letter a and the letter c; and that if this holds for the word w, then the word awc has as many times the letter a than the letter c, namely exactly one more than in word w.*

Exercise 2.7 *(solution on page 228) We consider the subset ABS of strict binary trees defined as the subset of language AB (of labeled by A binary trees) defined inductively by:*

(B) $(\emptyset, a, \emptyset) \in ABS$, *for every* $a \in A$.

(I) $g, d \in ABS \Rightarrow (g, a, d) \in ABS$, *for every* $a \in A$.

Prove that

- *an element of ABS is always non-empty and without a vertex with only one non-empty son.*
- *that in a strict binary tree, the number of vertices n satisfies $n = 2f - 1$, where f is the number of leaves.*

Exercise 2.8 *Prove that any word of the Dyck language has as many open parenthesis than closing parentheses.*

Exercise 2.9 *Prove that any arithmetic expression, that is to say any word of language $Arith$, has as many open parenthesis than closing parentheses.*

Exercise 2.10 *A binary tree is said to be* balanced *if for every vertex of the tree, the difference between the height of its right subtree and the height of its left subtree value either -1, 0 or 1 (i.e. at most one in absolute value).*

- *Provide an inductive definition of the set AVL of balance binary trees.*
- *Define the sequence $(u_n)_{n\in\mathbb{N}}$ by $u_0 = 0$, $u_1 = 1$, and for all $n \geq 2$, $u_{n+2} = u_{n+1} + u_n + 1$.*

  Prove that for every $x \in AVL$, $n \geq u_{h+1}$ where h and n are respectively the height and the number of vertices of a tree.

2.6 Derivations

2.6.1 Explicit expression of the elements: Second fix point theorem

We have seen up to now several examples of sets X defined inductively. The existence of each set X follows from Theorem 2.2, and actually from the Equation (2.1) used in its proof.

This is a *bottom-up definition* of X, as Equation (2.1) defines X from super-sets of X. This has the clear advantage to show easily the existence of sets defined inductively, a fact that we used abundantly up to now.

However, this has the default that it does not say what the elements of obtained sets X exactly are.

It is actually also possible to define each set X defined inductively from a *bottom-up definition*. One obtains then an explicit definition of the elements of X, with in addition a way to describe them explicitly.

This is what states the following result:

Theorem 2.4 (Explicit definition of a set defined inductively) *Every set X defined inductively from the base set B and from the rules R can also be written*

$$X = \bigcup_{n\in\mathbb{N}} X_n,$$

where $(X_n)_{n\in\mathbb{N}}$ is the family of subsets of E defined by recurrence by

- $X_0 = B$
- $X_{n+1} = X_n \cup \{r_i(x_1,\cdots,x_{n_i}) | x_1,\cdots,x_{n_i} \in X_n \text{ and } r_i \in R\}$.

In other words, every element of X is obtained by starting from elements of B and by applying a finite number of times the rules of R to obtain new elements.

Proof: It is sufficient to prove that this set is the least set that contains B and that is stable by the rules of R.

First since $X_0 = B$, B is indeed in the union of the X_n. Second, if one takes some rule $r_i \in R$, and some elements $x_1,\cdots,x_{n_i}$ in the union of the X_n, by definition each x_j is in X_{k_j} for some integer k_j. Since the sets X_i are increasing (i.e. $X_i \subset X_{i+k}$ for all k, which can be proved easily by recurrence over k), all the $x_1,\cdots,x_{n_i}$ are in X_{n_0} for $n_0 = \max(k_1,\cdots,k_{n_i})$. We obtain immediately that $r(x_1,\cdots,x_{n_i})$ is in X_{n_0+1}, which proves that it in the union of the X_n.

Finally, this is the least set, since every set that contains B and that is stable by the rules of R must contain each of the X_n. This is proved by recurrence over n. This is true at the rank $n = 0$, as such a set must contain X_0 since it contains B. Suppose the hypothesis at rank n, that is to say X contains X_n. Since the elements of X_{n+1} are obtained from elements of $X_n \subset X$ by applying some rule $r_i \in R$, X contains each of these elements. □

2.6.2 Derivation trees

The *bottom-up* definition of X from the previous theorem invite to attempt to keep the trace on how each element is obtained, starting from X and by applying the rules of R.

Example 2.16 *The word $1+2+3$ corresponds to some arithmetic expression. Here is a proof.*

$$\dfrac{\dfrac{1 \in \mathcal{N} \qquad 2 \in \mathcal{N}}{1+2 \in Arith} \qquad 3 \in Arith}{1+2+3 \in Arith}$$

This is not the only one possible. Indeed, we can also write:

$$\dfrac{1 \in Arith \qquad \dfrac{2 \in \mathcal{N} \qquad 3 \in \mathcal{N}}{2+3 \in Arith}}{1+2+3 \in Arith}$$

To encode each trace, the notion of term, on a set F of well-chosen symbols appears naturally: One considers that each element b of the base set B is symbol of arity 0. To each rule $r_i \in R$ is associated some symbol of arity n_i. A term t on this set of symbols is called a *derivation.* derivation

To each derivation t is associated some element $h(t)$ as expected: To t of arity 0, is associated the corresponding element b of B. Otherwise t is of the form $r_i(t_1, \cdots, t_{n_i})$, for some rule $r_i \in R$ end for some terms $t_1, \cdots, t_{n_i}$. To such a t is associated the result of the application of the rule r_i to elements $h(t_1), \cdots, h(t_{n_i})$.

Example 2.17 *For arithmetic expressions denote by the symbol + of arity* 2 *the rule* $g, d \in Arith \Rightarrow g+d \in Arith$*;*

The first proof of Example 2.16 corresponds to derivation $+(+(1,2),3)$*. The second to derivation* $+(1,+(2,3))$*. The image by the function h of these derivations is the word* $1+2+3$*.*

We can then reformulate the previous theorem in the following way.

Proposition 2.1 *Let X be a set defined inductively from the base set B and from the rules of R. Let D be the set of the derivations corresponding to B and to R. Then*

$$X = \{h(t) | t \in D\}.$$

In other words, X is precisely the set of the elements of E that have a derivation.

We see in the previous example, that an element of X may have several derivations.

Definition 2.3 *An inductive definition of X is said* non ambiguous *if the previous function h is injective.*

Intuitively, this means that there exists a unique way to build every element of X.

Example 2.18 *The following definition of* $\mathbb{N}^2$ *is ambiguous:*

(B) $(0,0) \in \mathbb{N}^2$;

(I) $(n,m) \in \mathbb{N}^2 \Rightarrow (n+1,m) \in \mathbb{N}^2$;

(I) $(n,m) \in \mathbb{N}^2 \Rightarrow (n,m+1) \in \mathbb{N}^2$.

Indeed, one can obtain for example $(1,1)$ *by starting from* $(0,0)$ *and by applying the second rule, and then the third, but also by applying the third rule, and then the second.*

Example 2.19 *The definition of* $Arith$ *of example 2.13 is ambiguous since* $1+2+3$ *has several derivations.*

Example 2.20 *This problem is intrinsic to arithmetic expressions, since when we write* $1+2+3$*, we do not precise if we are talking about the result of the addition of* 1 *to* $2+3$ *or of* 3 *to* $1+2$*, the idea being that since addition is associative, this is not important.*

Example 2.21 *To avoid this potential problem, let's define the set* $Arith'$ *of the parenthesised arithmetic expressions as the subset of* ${\Sigma_{exp}}^*$*, defined inductively by*

(B) $0 \in Arith$;

(B) $\mathcal{N} \subset Arith'$;

(I) $g,d \in Arith' \Rightarrow (g+d) \in Arith'$;

(I) $g,d \in Arith' \Rightarrow (g*d) \in Arith'$;

(I) $g,d \in Arith' \Rightarrow (g/d) \in Arith'$;

(I) $g,d \in Arith' \Rightarrow (g-d) \in Arith'$;

(I) $g \in Arith' \Rightarrow (g) \in Arith'$;

This times, $1+2+3$ *is not a word of* $Arith'$*. By opposition* $(1+(2+3)) \in Arith'$ *and* $((1+2)+3) \in Arith'$.

The interest of this writing is that we now have some non-ambiguous *rules.*

2.7 Functions defined inductively

We will need sometimes to defined functions on some sets X defined inductively. This can be done easily when X admits some *non-ambiguous definition.*

Theorem 2.5 (Inductively defined function) *Let* $X \subset E$ *be a set defined inductively in a non-ambiguous way from a the base set* B *and from rules* R*. Let* Y *be a set.*

For an application f *from* X *to* Y *is perfectly defined, it suffices that the following are given:*

(*B*) *the value of* $f(x)$ *for each of the elements* $x \in B$*;*

(*I*) *for reach rule* $r_i \in R$*, the value of* $f(x)$ *for* $x = r_i(x_1, \cdots, x_{n_i})$ *as a function of the values* $x_1, \ldots, x_{n_i}$, $f(x_1), \ldots,$ *and* $f(x_{n_i})$.

In other words, informally, if one knows how to "program recursively", that is to say "describe in a recursive way the function" then the function is perfectly defined on the inductive set X.

Proof: The statement above means that there exists a unique application f from X to Y that satisfies these constraints. It suffices to prove that for every $x \in X$, the value of f in x is defined in a unique way. This is proved easily by induction: This is true for the elements $x \in B$. If this is true in $x_1, \cdots, x_{n_i}$, this is true in $x = r_i(x_1, \cdots, x_{n_i})$: The definition of X being non-ambiguous, x can obtained only by the rule r_i from $x_1, \cdots, x_{n_i}$. Its value is hence perfectly defined by the constraint for the rule r_i. □

Example 2.22 *The factorial function* $Fact$ *from* $\mathbb{N}$ *into* $\mathbb{N}$ *is defined inductively by*

(*B*) $Fact(0) = 1$;

(*I*) $Fact(n+1) = (n+1) * Fact(n)$.

Example 2.23 *The height* h *of a labeled binary tree is defined inductively by*

(*B*) $h(\emptyset) = 0$;

(*I*) $h((g,a,d)) = 1 + \max(h(g), h(d))$.

Example 2.24 *The value* v *of an arithmetic expression of* $Arith'$ *is defined inductively by (*v *is a function that goes from words to the rational numbers)*

(*B*) $v(0) = 0$*;*

(*B*) $v(x) = h(x)$ *pour* $x \in \mathcal{N}$*;*

(*I*) $v((g+d)) = v(g) + v(d)$*;*

(*I*) $v((g*d)) = v(g) * v(d)$*;*

(*I*) $v((g/d)) = v(g)/v(d)$*, if* $v(d) \neq 0$*;*

(*I*) $v((g-d)) = v(g) - v(d)$*;*

(*I*) $v((g)) = v(g)$*;*

where h *is the function that, to a word of* $\mathcal{N}$ *maps its value as a rational number:* h *is defined inductively by*

(B) $h(a) = a$ *for each* $a \in \{1,2,\ldots,9\}$*;*

(I) $h(ga) = 10 * h(g) + a$ *for each* $a \in \{0,1,2,\ldots,9\}$.

We observe that all these definitions are essentially only the translation on how their can be programmed in some recursive way. The use of non-ambiguous definitions avoids any ambiguity on the evaluation.

Remark 2.8 *For the arithmetic expressions,* $1+2*3 \in Arith$ *is also ambiguous. A computer program that would take as input a word of* $Arith$ *and supposed to return the value of the expression would have to manage the priorities, and understand that* $1+2*3$ *is not the result of the multiplication of* $1+2$ *by* 3*. By using the definition of* $Arith'$*, we avoid completely this difficulty, since the expressions are encoding explicitly how their must be evaluated, and an inductive definition becomes possible. At first sight, the value of an expression of* $Arith$ *cannot be defined simply inductively, if only because the value of* $x+y*z$ *is not obtained directly from the one of* $x+y$ *and from* z *only.*

2.8 Bibliographic notes

Suggested readings To go further on the notions of this chapter, we suggest to read [Arnold & Guessarian, 2005]. For a more general presentation of inductive definitions, and of fix point theorems, we suggest to read [Dowek, 2008].

Bibliography This chapter has been written by using [Dowek, 2008] as well as [Arnold & Guessarian, 2005].

Chapter 3

Propositional calculus

The *propositional logic* provides means to discuss logical grammatical connectors such as *negation*, *disjunction* or *negation*, by composition starting some Boolean propositions. These connectors are sometimes called *Aristotelian* as they have been pointed out by Aristotle.

The *propositional logic* permits essentially to talk about *Boolean functions*, that is to say about functions from $\{0,1\}^n \to \{0,1\}$. Indeed, the variables, that is to say the *propositions* can only take two values, *true* or *false*.

The propositional calculus has an important position in computer science. A first reason is because today's computers are digital and working in *binary*. This has the consequence that our processors are essentially made of binary gates of the type that we will study in this chapter.

From a point of view of expressive power, propositional calculus remains very limited. For example, one cannot express in propositional calculus the existence of an object with a given property. The predicate calculus, more general, that we will study in Chapter 5, provides means to express some properties of objects and relations between objects, and more generally to formalise the mathematical reasoning.

Since the propositional calculus provides anyway the common basis to numerous logical systems, we will take some time on it in this chapter.

3.1 Syntax

To define formally and properly this language, we must distinguish the *syntax* from the *semantic*: The *syntax* describes how formulas are written. The *semantic* describes their meaning.

Fix a finite or denumerable set $\mathscr{P} = \{p_0, p_1, \cdots\}$ of symbols that are called *propositional variables*.

Definition 3.1 (Propositional formulas) *The set of propositional formulas $\mathscr{F}$ over $\mathscr{P}$ is the language over the alphabet $\mathscr{P} \cup \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, (,)\}$ defined inductively by the following rules: ,,,,*

(B) *it contains* $\mathscr{P}$*: Every propositional variable is a propositional formula;*

(I) *If* $F \in \mathscr{F}$ *then* $\neg F \in \mathscr{F}$*;*

(I) *If* $F, G \in \mathscr{F}$ *then* $(F \wedge G) \in \mathscr{F}$, $(F \vee G) \in \mathscr{F}$, $(F \Rightarrow G) \in \mathscr{F}$, *and* $(F \Leftrightarrow G) \in \mathscr{F}$.

It is an inductive definition that is valid by the considerations of the previous chapter: It is a non-unambiguous definition. This can be reformulated by the following proposition, that is sometimes called *unique reading theorem of propositional calculus*.

Remark 3.1 *The non-ambiguity comes essentially from the explicit parentheses. We use here the trick in the previous chapter that was considering* $Arith'$ *instead of* $Arith$ *to allow to write some expressions without any reading ambiguity.*

Proposition 3.1 (Decomposition / Uniqueness reading) *Let* F *be a propositional formula. Then* F *is of one, and exactly one of the following forms:*

1. *a propositional formula* $p \in \mathscr{P}$*;*
2. $\neg G$*, where* G *is a propositional formula;*
3. $(G \wedge H)$ *where* G *and* H *are some propositional formulas;*
4. $(G \vee H)$ *where* G *and* H *are some propositional formulas;*
5. $(G \Rightarrow H)$ *where* G *and* H *are some propositional formulas;*
6. $(G \Leftrightarrow H)$ *where* G *and* H *are some propositional formulas.*

Moreover, in the cases 2., 3., 4., 5. and 6., there is unicity of the formula G *and unicity of the formula* H *with these properties.*

The fact that a formula can always be decomposed into one of the 6 cases above is easy to establish inductively. The unicity of the decomposition follows from the following exercise:

p	$\neg p$	q	$p \vee q$	$p \wedge q$	$p \Rightarrow q$	$p \Leftrightarrow q$
0	1	0	0	0	1	1
1	0	0	1	0	0	0
0	1	1	1	0	1	0
1	0	1	1	1	1	1

Figure 3.1: Truth value.

Exercise 3.1 *Prove that the previous inductive definition is* non-ambiguous, *that is that G and H are uniquely defined in each of the cases above.*
We can proceed in the following way.

- *Prove that in any formula F the number of open parentheses is equal to the number of closing parentheses.*
- *Prove that in any word M prefix of the word F, we have $o(M) \geq f(M)$, where $o(M)$ is the number of open parentheses, and $f(M)$ the number of closing parentheses.*
- *Prove that in any formula F whose first symbol is some open parenthesis, and for any word M proper prefix of F, we have $o(M) > f(M)$.*
- *Prove that any word M proper prefix of F is not a formula.*
- *Deduce the result.*

We call *subformula* of F a formula that appears in the recursive decomposition of F.

3.2 Semantic

We are going now to define the *semantic* of a propositional formula, that is to say, the meaning that is assigned to such a formula.

The *truth value* of a formula is defined as the interpretation of this formula, after having fixed the truth value of the propositional variables: The principle is to interpret the symbols $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$ by the logic $negation$, the logical or (also called disjunction), the logical and (also called conjunction), the implication and the *double implication* (also called *equivalence*)).

Formally,

Definition 3.2 (Valuation) *A* valuation *is a distribution of truth value to the propositional variables, that is to say a function from $\mathscr{P}$ to $\{0,1\}$.*

In all what follows, 0 represents false, and 1 represents true.

The conditions in the following definition are often represented as a *truth value*: See Figure 3.1.

Proposition 3.2 *Let v be a valuation.*
By Theorem 2.5, there exists a unique function $\overline{v}$ defined on all $\mathscr{F}$ that satisfies the following conditions:

- (B) *$\overline{v}$ extends v: for every propositional variable $p \in \mathscr{P}$, $\overline{v}(p) = v(p)$;*
- (I) *the negation is interpreted by logic negation: if F is of the form $\neg G$, then $\overline{v}(F) = 1$ if and only if $\overline{v}(G) = 0$;*
- (I) *$\wedge$ is interpreted as the logical and: if F is of the form $G \wedge H$, then $\overline{v}(F) = 1$ if and only if $\overline{v}(G) = 1$ and $\overline{v}(H) = 1$;*
- (I) *$\vee$ is interpreted as the logical or: if F is of the form $G \vee H$, then $\overline{v}(F) = 1$ if and only if $\overline{v}(G) = 1$ or $\overline{v}(H) = 1$;*
- (I) *$\Rightarrow$ is interpreted as the logical implication: if F is of the form $G \Rightarrow H$, then $\overline{v}(F) = 1$ if and only if $\overline{v}(H) = 1$ or $\overline{v}(G) = 0$;*
- (I) *$\Leftrightarrow$ is interpreted as the logical equivalence: if F is of the form $G \Leftrightarrow H$, then $\overline{v}(F) = 1$ if and only if $\overline{v}(G) = \overline{v}(H)$.*

We write $v \models F$ for $\overline{v}(F) = 1$, and we say that v is *a model* of F, or that v satisfies F. We write $v \not\models F$ otherwise. The value of $\overline{v}(F)$ for the valuation v is called the *truth value of F on v*.

3.3 Tautologies, equivalent formulas

We would like to classify the formulas according to their interpretations. A particular class of formulas are those that are always trues, and that are called the *tautologies*.

Definition 3.3 (Tautology) *A* tautology *is a formula F that is satisfied by any valuation. We write in this case $\models F$.*

Definition 3.4 (Equivalence) *Two formulas F and G are said to be* equivalent *if for every valuation v, $\overline{v}(F) = \overline{v}(G)$. We write in this case $F \equiv G$.*

Example 3.1 *The formula $p \vee \neg p$ is a tautology. The formulas p and $\neg\neg p$ are equivalent.*

Remark 3.2 *It is important to understand that $\equiv$ is a symbol that is used to write a relation between formulas, but that $F \equiv G$ is not a propositional formula.*

Exercise 3.2 *Prove that $\equiv$ is some equivalence relation on the formulas.*

3.4 Some elementary facts

Exercise 3.3 *Prove that for any formulas F and G, the following formulas are tautologies:*

$$(F \Rightarrow F),$$

$$(F \Rightarrow (G \Rightarrow F)),$$

$$(F \Rightarrow (G \Rightarrow H)) \Rightarrow ((F \Rightarrow G) \Rightarrow (F \Rightarrow H)).$$

Exercise 3.4 *[Idempotence] Prove that for any formula F we have the equivalences:*

$$(F \vee F) \equiv F,$$

$$(F \wedge F) \equiv F.$$

Exercise 3.5 *[Associativity] Prove that for any formulas F, G, H we have the equivalences:*

$$(F \wedge (G \wedge H)) \equiv ((F \wedge G) \wedge H),$$

$$(F \vee (G \vee H)) \equiv ((F \vee G) \vee H).$$

Because of associativity, one often denotes $F_1 \vee F_2 \vee \cdots \vee F_k$ for $(((F_1 \vee F_2) \vee F_3) \cdots \vee F_k)$, and $F_1 \wedge F_2 \wedge \cdots \wedge F_k$ for $(((F_1 \wedge F_2) \wedge F_3) \cdots \wedge F_k)$.

Remark 3.3 *Exactly as we do for arithmetic expression: We write $1+2+3$ for $((1+2)+3)$ as well as for $(1+(2+3))$. See all the discussions on $Arith$ and $Arith'$ in the previous chapter.*

Exercise 3.6 *[Commutativity] Prove that for any formulas F and G we have the equivalences:*

$$(F \wedge G) \equiv (G \wedge F),$$

$$(F \vee G) \equiv (G \vee F).$$

Exercise 3.7 *[Distributivity] Prove that for any formulas F, G, H we have the equivalences:*

$$(F \wedge (G \vee H)) \equiv ((F \wedge G) \vee (F \wedge H)),$$

$$(F \vee (G \wedge H)) \equiv ((F \vee G) \wedge (F \vee H)).$$

Exercise 3.8 *[Morgan's law] Prove that for any formulas F and G we have the equivalences:*

$$\neg(F \wedge G) \equiv (\neg F \vee \neg G),$$

$$\neg(F \vee G) \equiv (\neg F \wedge \neg G).$$

Exercise 3.9 *[Absorption] Prove that for any formulas F and G we have the equivalences:*

$$(F \wedge (F \vee G)) \equiv F,$$

$$(F \vee (F \wedge G)) \equiv F.$$

3.5 Replacement of a formula by some equivalent formula

We know now some *equivalences* between formulas, but we are going to convince ourselves that one can use these equivalences in a *compositional* way: If one replaces in some formula some subformula by some equivalent formula, then one obtains an equivalent formula.

3.5.1 A simple remark

Observe first that the truth value of a formula is depending only on the propositional formulas that appear in the formula: When F is a formula, we will write $F(p_1, \cdots, p_n)$ to say that the formula F is written with the propositional formulas $p_1, \cdots, p_n$ only.

Proposition 3.3 *Let $F(p_1, \cdots, p_n)$ be a formula. Let v be some valuation. The truth value of F on v is depending only on the value of v on $\{p_1, p_2, \cdots, p_n\}$.*

Proof: The property can be established easily by structural induction. □

3.5.2 Substitutions

We have to defined what means replacing p by G in a formula F, denoted by $F(G/p)$.

This provides the rather pedantic definition, but we have to go through this:

Definition 3.5 (Substitution of p by G in F) *The formula $F(G/p)$ is defined by induction on the formula F:*

(*B*) *If F is the propositional formula p, then $F(G/p)$ is the formula G;*

(*B*) *If F is a propositional formula q, with $q \neq p$, then $F(G/p)$ is the formula F;*

(*I*) *If F is of the form $\neg H$, then $F(G/p)$ is the formula $\neg H(G/p)$;*

(*I*) *If F is of the form $(F_1 \vee F_2)$, then $F(G/p)$ is the formula $(F_1(G/p) \vee F_2(G/p))$;*

(*I*) *If F is of the form $(F_1 \wedge F_2)$, then $F(G/p)$ is the formula $(F_1(G/p) \wedge F_2(G/p))$;*

(*I*) *If F is of the form $(F_1 \Rightarrow F_2)$, then $F(G/p)$ is the formula $(F_1(G/p) \Rightarrow F_2(G/p))$;*

(*I*) *If F is of the form $(F_1 \Leftrightarrow F_2)$, then $F(G/p)$ is the formula $(F_1(G/p) \Leftrightarrow F_2(G/p))$.*

3.5.3 Compositionality

We obtain the promised result: If one replaces in a formula some subformula by some equivalent formula, then one obtains an equivalent formula;

Proposition 3.4 *Let F, F', G and G' be formulas. Let p be a propositional variable.*

- *If F is a tautology, then $F(G/p)$ is also a tautology.*
- *If $F \equiv F'$, then $F(G/p) \equiv F'(G/p)$.*
- *If $G \equiv G'$ then $F(G/p) \equiv F(G'/p)$.*

Exercise 3.10 *Prove the result by structural induction.*

3.6 Complete systems of connectors

Proposition 3.5 *Every propositional formula is equivalent to a propositional formula that is built only with the connectors $\neg$ and $\wedge$.*

Proof: This results from a proof by induction on the formula. This is true for the formulas that correspond to some propositional variable. Suppose the property true for the formulas G and H, that is to say suppose that G (respectively H) is equivalent to some formula G' (respectively H') built only with the connectors $\neg$ and $\wedge$.

If F is of the form $\neg G$, then F is equivalent to $\neg G'$, and the induction hypothesis is preserved.

If F is of the form $(G \wedge H)$, then F is equivalent to $(G' \wedge H')$, and the induction hypothesis is preserved.

If F is of the form $(G \vee H)$, by using the second Morgan's law, and the fact that $K \equiv \neg\neg K$ to eliminate the double negations, we obtain that $F \equiv \neg(\neg G' \wedge \neg H')$, which is indeed built using only the connectors $\neg$ and $\wedge$.

If F is of the form $(G \Rightarrow H)$, then F is equivalent to $(\neg G' \vee H')$ that is equivalent to a formula build uniquely with the connectors $\neg$ and $\wedge$ by the previous cases.

If F is of the form $(G \Leftrightarrow H)$, then F is equivalent to $(G' \Rightarrow H') \wedge (H' \Rightarrow G')$ that is equivalent to a formula build uniquely with the connectors $\neg$ and $\wedge$ by the previous cases. □

A set of connectors with the above property for $\{\neg, \wedge\}$ is called a complete system of connectors.

Exercise 3.11 *Prove that $\{\neg, \vee\}$ is also a complete system of connectors.*

Exercise 3.12 *Give a binary logic connector that, alone, constitutes a complete system of connectors.*

3.7 Functional completeness

Suppose that $\mathscr{P} = \{p_1, p_2, \cdots, p_n\}$ is finite. Let V be the set of valuations on $\mathscr{P}$. Since a valuation is a function from $\{1, 2, \ldots, n\}$ to $\{0, 1\}$, V contains 2^n elements.

Each formula F over $\mathscr{P}$ can be seen as a function from V to $\{0, 1\}$, that is called its *truth value of F*: This function is the function that, to a valuation v associates the truth value of this formula on the valuation.

There are 2^{2^n} functions from V to $\{0, 1\}$. The question that one may ask is to know if all these functions can be written as formulas. The answer is positive:

Theorem 3.1 (Functional completeness) *Suppose $\mathscr{P} = \{p_1, p_2, \cdots, p_n\}$ is finite. Let V be the set of valuations over $\mathscr{P}$. Every function f from V to $\{0, 1\}$ is the truth value of some formula F over $\mathscr{P}$.*

Proof: The proof is done by recurrence on the number of propositional variables n.

For $n = 1$, there are four functions from $\{0,1\}^1$ to $\{0,1\}$, that are represented by the formulas p, $\neg p$, $p \vee \neg p$, $p \wedge \neg p$.

Suppose that the property is true for $n-1$ propositional variables. Consider $\mathscr{P} = \{p_1, \cdots, p_n\}$ and let f be a function from $\{0,1\}^n$ to $\{0,1\}$. Each valuation v' over $\{p_1, p_2, \cdots, p_{n-1}\}$ can be seen as the restriction of a valuation on $\{p_1, \cdots, p_n\}$. Let f_0 (respectively f_1) the restriction of f to the valuation v such that $v(p_n) = 0$ (resp. $v(p_n) = 1$). The functions f_0 and f_1 are some functions defined on valuations over $\{p_1, \cdots, p_{n-1}\}$ to $\{0,1\}$ and can be represented by formulas $G(p_1, \cdots, p_{n-1})$ and $H(p_1, \cdots, p_{n-1})$ respectively by recurrence hypothesis. The function f can then be represented by the formula

$$(\neg p_n \wedge G(p_1, \cdots, p_{n-1})) \vee (p_n \wedge H(p_1, \cdots, p_{n-1}))$$

which proves the recurrence hypothesis at rank n. □

Remark 3.4 *Our attentive reader will have observed that the Proposition 3.5 can be seen as a consequence of this proof.*

3.8 Normal forms

3.8.1 Conjunctive and disjunctive normal forms

One often seeks to transform the formulas into some equivalent form as simple as possible.

Definition 3.6 *A* literal *is a propositional formula or its negations, i.e. of the form p, or $\neg p$, for $p \in \mathscr{P}$.*

Definition 3.7 *A* disjunctive normal form *is a disjunction $F_1 \vee F_2 \cdots \vee F_k$ of k formulas, $k \geq 1$ where each formula F_i, $1 \leq i \leq k$ is a conjunction $G_1 \wedge G_2 \cdots \wedge G_\ell$ of ℓ literals (ℓ can possibly depend on i).*

Example 3.2 *The following formulas are in disjunctive normal form*

$$((p \wedge q) \vee (\neg p \wedge \neg q)$$

$$((p \wedge q \wedge \neg r) \vee (q \wedge \neg p))$$

$$(p \wedge \neg q)$$

Definition 3.8 *A* conjunctive normal form *is a conjunction $F_1 \wedge F_2 \cdots \wedge F_k$ of k formulas, $k \geq 1$ where each formula F_i, $1 \leq i \leq k$ is a disjunction $G_1 \vee G_2 \cdots \vee G_\ell$ of ℓ literals (ℓ can possibly depend on i).*

Example 3.3 *The following formulas are in conjunctive normal form*

$$(\neg p \vee q) \wedge (p \vee \neg q)$$

$$(\neg p \vee q) \wedge \neg r$$

$$(\neg p \vee q)$$

Theorem 3.2 *Every formula on a finite number of propositional variables is equivalent to some formula in conjunctive normal form.*

Theorem 3.3 *Every formula on a finite number of propositional variables is equivalent to some formula in disjunctive normal form.*

Proof: These two theorems are proved by recurrence on the number n of propositional formulas.

In the case where $n = 1$, we have already considered in the previous proofs some formulas covering all the possible cases, and which are actually both in conjunctive normal form and disjunctive normal form.

We suppose the property true for $n-1$ propositional variables. Let f be the truth value associated to the formula $F(p_1, \cdots, p_n)$. As in the previous proof, we can build some formula that represents f, by writing a formula of the form

$$(\neg p_n \wedge G(p_1, \cdots, p_{n-1})) \vee (p_n \wedge H(p_1, \cdots, p_{n-1})).$$

By recurrence hypothesis, G and H are equivalent to formulas in disjunctive normal form

$$G \equiv (G_1 \vee G_2 \vee \cdots \vee G_k)$$

$$H \equiv (H_1 \vee H_2 \vee \cdots \vee H_\ell)$$

We can then write

$$(\neg p_n \wedge G) \equiv (\neg p_n \wedge G_1) \vee (\neg p_n \wedge G_2) \vee \cdots \vee (\neg p_n \wedge G_k)$$

which is in disjunctive normal form and

$$(p_n \wedge H) \equiv (p_n \wedge H_1) \vee (p_n \wedge H_2) \vee \cdots \vee (p_n \wedge H_k)$$

which is also in disjunctive normal form. The function f is hence represented by the disjunction of these two formulas, and hence by a formula in disjunctive normal form.

If we want to obtain F in conjunctive normal form, then the hypothesis induction produces two conjunctive normal form G and H. The equivalence that is used is then

$$F \equiv ((\neg p_n \vee H) \wedge (p_n \vee G)).$$

□

Remark 3.5 *Our attentive reader would have observed that the previous theorem, as well as Proposition 3.5 can also be seen as the consequences of this proof.*

3.8.2 Transformation methods

In practise, there exist two main methods to determine the disjunctive normal form, or the conjunctive normal form of a given formula. The first method consists in transforming the formula by successive equivalence by using the following rules applied in this order:

1. elimination of connectors $\Rightarrow$ by

$$(F \Rightarrow G) \equiv (\neg F \vee G)$$

2. entering the negation in the innermost position:

$$\neg(F \wedge G) \equiv (\neg F \vee \neg G)$$

$$\neg(F \vee G) \equiv (\neg F \wedge \neg G)$$

3. distributivity of $\vee$ and $\wedge$ one with respect to the other

$$F \wedge (G \vee H)) \equiv ((F \wedge H) \vee (F \wedge H))$$

$$F \vee (G \wedge H)) \equiv ((F \vee H) \wedge (F \vee H))$$

Example 3.4 *Put the formula $\neg(p \Rightarrow (q \Rightarrow r)) \vee (r \Rightarrow q)$ in disjunctive and conjunctive normal form.*

We use the successive equivalences

$$\neg(\neg p \vee (\neg q \vee r)) \vee (\neg r \vee q)$$

$$(p \wedge \neg(\neg q \vee r)) \vee (\neg r \vee q)$$

$$(p \wedge q \wedge \neg r) \vee (\neg r \vee q)$$

that is a disjunctive normal form.

$$(p \wedge q \wedge \neg r) \vee (\neg r \vee q)$$

$$(p \wedge \neg r \vee q) \wedge (\neg r \vee q)$$

The other method consists in determining the valuations v such that $\overline{v}(F) = 1$, and to write a disjunction of conjunctions, each conjunction corresponding to a valuation for which $\overline{v}(F) = 1$.

The determination of a conjunctive normal form follows the same principle, by exchanging the valuations that value 1 with the valuations giving the value 0, by exchanging conjunctions and disjunctions.

Exercise 3.13 *Prove that the conjunctive and disjunctive normal form of a formula can be exponentially longer than the size of the formula. The* size of a formula *is defined as the length of the formula seen as a word.*

3.9 Compactness theorem

3.9.1 Satisfaction of a set of formulas

We are given this times a set Σ of formulas. One wants to know when it is possible to satisfy all the formulas of Σ.

Let's start by fix the terminology.

Definition 3.9 *Let Σ be a set of formulas.*

- *A valuation* satisfies Σ *if it satisfies each formula of Σ. One also says in that case that this valuation is* a model *of Σ.*
- *Σ is said* consistent *(this is also called* satisfiable*) if it has a model. In other words, if there exists some valuation that satisfies Σ.*
- *Σ is said* inconsistent*, or* contradictory*, in the opposite case*

Definition 3.10 (Consequence) *Let F be a formula. The formula F is said to a* consequence *of Σ if every model of Σ is a model of F. We then write $\Sigma \models F$.*

Example 3.5 *The formula q is a consequence of the set of formulas $\{p, p \Rightarrow q\}$. The set of formulas $\{p, p \Rightarrow q, \neg q\}$ is inconsistent.*

We can get convinced first of the following results, that follows from a game on definitions.

Proposition 3.6 *Every formula F is a consequence of a set Σ of formulas if and only if $\Sigma \cup \{\neg F\}$ is inconsistent.*

Proof: If every valuation that satisfies Σ satisfies F, there there is no valuation satisfying $\Sigma \cup \{\neg F\}$. Conversely, by contradiction: If there is a valuation that satisfies Σ and not satisfying F, then this valuation satisfies Σ and $\neg F$. □

Exercise 3.14 *Prove that for any formulas F and F', $\{F\} \models F'$ if and only if $F \Rightarrow F'$ is a tautology.*

More fundamentally, we ave the following rather surprising and fundamental result.

Theorem 3.4 (Compactness theorem (first version)) *Let Σ be a set of formulas built on a denumerable set $\mathscr{P}$ of propositional variables.*
Then Σ is consistent if and only if every finite subset of Σ is consistent.

Remark 3.6 *Observe that the hypothesis $\mathscr{P}$ countable is not necessary, if we accept to use Zorn hypothesis (the axiom of choice). We will restrict to the case where $\mathscr{P}$ is denumerable in all the proofs that follow.*

Actually, this theorem can be reformulated as follows:

Theorem 3.5 (Compactness theorem (second version)) *Let Σ be a set of formulas built on a denumerable set $\mathscr{P}$ of propositional variables.*
Then Σ is inconsistent if and only if Σ has some finite inconsistent subset.

Or even under the following form:

Theorem 3.6 (Compactness theorem (third version)) *For every set Σ of propositional formulas, and for every propositional formula F built on a denumerable set $\mathscr{P}$ of propositional variables, F is a consequence of Σ if and only if F is a consequence of a finite subset of Σ.*

The equivalence of the three formulations is a simple exercise of manipulations of definitions. We will prove the first version of the theorem.

One of the implication is trivial: If Σ is consistent, then every subset of Σ is consistent, and in particular the finite subsets.

We will provide two proofs of the other implication.

A first proof that makes references to notions of topologies, in particular compactness, and that is addressed to readers who know theses notion, and who like topological arguments.

Proof:[Topological proof] The topological space $\{0,1\}^{\mathscr{P}}$ (with the product topology) is a compact space, since it is obtained as the product of compact spaces (Tychonoff theorem).

For every propositional formula $F \in \Sigma$, the set $\overline{F}$ of the valuations which satisfy it is open in $\{0,1\}^{\mathscr{P}}$, as the truth value of a formula is depending only from a finite number of propositional variables, namely those appearing in the formula. It is also closed, since those that are not satisfying F are those satisfying t $\neg F$.

The hypothesis of the theorem implies that any finite intersection of $\overline{F}$ for $F \in \Sigma$ is non-empty. Since $\{0,1\}^{\mathscr{P}}$ is compact, the intersection of all the $\overline{F}$ for $F \in \Sigma$ is hence non-empty. □

Here is a proof that avoid topology.

Proof:[Direct proof] Consider $\mathscr{P} = \{p_1, p_2, \cdots, p_k, \cdots\}$ an enumeration of $\mathscr{P}$.

We will prove the following lemma: Suppose that there exists some application v from $\{p_1, p_2, \cdots, p_n\}$ to $\{0,1\}$ such that any finite subset of Σ has a model in which $p_1, \cdots, p_n$ take the values $v(p_1)$, ..., $v(p_n)$. Then v can be extended to $\{p_1, p_2, \cdots, p_{n+1}\}$ with the same property.

Indeed, if $v(p_{n+1}) = 0$ does not fit, then there exists some finite set U_0 of Σ that cannot be satisfied when $p_1, \cdots, p_n, p_{n+1}$ take respective values $v(p_1), \ldots, v(p_n)$ and 0. If U is any finite subset of Σ, then from the hypothesis made on v, $U_0 \cup U$ has a model in which $p_1, \cdots, p_n$ take the values $v(p_1), \cdots, v(p_n)$. In this model, the proposition p_{n+1} takes the value 1. In other words, every finite subset U of Σ has a model in which $p_1, \cdots, p_n, p_{n+1}$ take the respective values $v(p_1)$, ..., $v(p_n)$ and 1. Stated in another way, either $v(p_{n+1}) = 0$ is fine with the property, in which case, we can fix $v(p_{n+1}) = 0$, or $v(p_{n+1}) = 0$ is not fine, in which case, we can set $v(p_{n+1}) = 1$ which is fine with the property.

By using this lemma, we hence define some valuation v such that, by recurrence over n, for every n, every finite set of Σ has a model in which $p_1, \cdots, p_n$ take the values $v(p_1), \ldots, v(p_n)$.

It follows that v satisfies Σ: Indeed, let F be a formula of Σ. F is depending only a finite set of propositional formulas $p_{i_1}, p_{i_2}, \cdots, p_{i_k}$ (the one appearing in F). By considering $n = \max(i_1, i_2, \cdots, i_k)$, each of these propositional variables p_{i_j} is among $\{p_1, \cdots, p_n\}$. We then know that the finite subset $\{F\}$ reduced to the formula F admits a model in which $p_1, \cdots, p_n$ take the value $v(p_1), \ldots, v(p_n)$, i.e. F is satisfied by v.

□

3.10 Exercises

Exercise 3.15 *Relate the equivalent propositions:*

1. $\neg(p \wedge q)$	1. $(\neg p \wedge \neg q)$
2. $\neg(p \vee q)$	2. $q \rightarrow (\neg p)$
3. $p \rightarrow (\neg q)$	3. $(\neg p \vee \neg q)$
4. $\neg(p \rightarrow q)$	4. $p \wedge (\neg q)$

Exercise 3.16 *By adding two numbers whose binary expression uses at most two digits, say ab and cd, we obtain a number of at most three digits pqr. For example, $11 + 01 = 100$. Give an expression of p, q and r as a function of a, b, c and d using the usual connectors.*

Exercise 3.17 *(solution on page 229) Let F and G two formulas with no propositional variable in common. Prove that the two following properties are equivalent:*

- *The formula $(F \Rightarrow G)$ is a tautology;*
- *At least one of $\neg F$ and G is a tautology.*

***Exercise 3.1** [Interpolation theorem] Let F and F' such that $F \Rightarrow F'$ is a tautology. Prove that there exists some propositional formula C, whose propositional variables appear in F and F', such that $F \Rightarrow C$ and $C \Rightarrow F'$ are two tautologies (one can reason on recurrence on the number of variables that have at least one occurrence in F without any in F').*

Exercise 3.18 *(solution on page 229) [Application of compactness to graph colouring] A graph $G = (V, E)$ is k-colorable if there exists some function f from V in $\{1, 2, \ldots, k\}$ such that for all $(x, y) \in E$, $f(x) \neq f(y)$. Prove that a graph is k-colorable if and only if any of its finite sub-graphs is k-colorable .*

***Exercise 3.2** [Applications of compactness to group theory] A group G is said to be totally ordered if we have on G some total order relation such that $a \leq b$ implies $ac \leq bc$ and $ca \leq cb$ for all $a, b, c \in G$. Prove that for some Abelian group G can be ordered, it is sufficient and necessary that any subgroup of G spanned by a finite set elements of G can be ordered.*

3.11 Bibliographic notes

Suggested readings To go further on the notions of this chapter, we suggest [Cori & Lascar, 1993a] and [Lassaigne & de Rougemont, 2004].

Bibliography This chapter has been written by using essentially the books [Cori & Lascar, 1993a] and [Lassaigne & de Rougemont, 2004].

Chapter 4

Proofs

The objective of this chapter is to start to address the fundamental following question: what is a *demonstration* (i.e. a (mathematical) *proof*).

To to this, more precisely, we will focus on this chapter on the following problem: Given some *propositional formula* F, how to decide if F is a *tautology*? A tautology is also called a *theorem*.

This will lead us to describe some particular *algorithms*.

4.1 Introduction

A first method to solve this problem is the one that we have used in the previous chapter: If F is of the form $F(p_1, \cdots, p_n)$, we can test for each of the 2^n valuations v, i.e. for the 2^n functions from $\{1,2,\ldots,n\}$ to $\{0,1\}$, if v is indeed a *model* of F. If this is the case, then F is a tautology. In any other case, F is not a tautology. It is easy to program such a method in your favorite programming language.

The good news is that this method exists: The problem to determine if a given formula is a tautology is *decidable*, using the terminology that we will see in the next chapters.

Remark 4.1 *This observation can seem strange, and, in some sense, to expect little, but we will see that when we consider more general logic, even simple logic, this becomes problematic: There does not always exist some algorithm to determine if a given formula F is a tautology.*

However, this method is particularly inefficient. It has the main inconvenient to guarantee that when F is a tautology, we will do 2^n times a similar test of type "is the valuation v a model of F?". When n is big, 2^n explodes very quickly: If this method can indeed be programmed, it is in practise useless, since it takes a huge time, as soon as one considers some formulas F with a high number of variables.

Let's then come back to our problem: One can say that in the classical reasoning in mathematics, the usual method to prove that some assertion is a theorem is to *prove* it.

If one wants to do better than the previous exhaustive method, there are two angles of attacks. The first angle of attack is to try to come close to the notion of *demonstration* in the usual reasoning: Proof methods in the spirit of the coming sections will appear. The second angle of attack is to try to produce algorithms as efficient as possible: Methods such as *proof!by resolution* method or *tableau method* then appear.

In general, one expects that a proof method is always *valid*: It produces only correct deductions. In any case, the question of the *completeness* of the proof method makes sense: Can all the theorems (tautology) be proved using this proof method?

We will see in what follows, four deductions systems that are valid and complete: The proofs *à la Hilbert*, the *natural deduction*, the *resolution method* and the *tableau method*. We will prove the validity and the completeness only for the tableau method. Every time, we will denote by $\vdash$ the underlying notion of proof: $T \vdash F$ means that the formula F can be proved starting from a set of propositional formulas T. We write $\vdash T$ if $\emptyset \vdash T$.

At first sight, one needs a different symbol $\vdash$ for each notion of demonstration.

However, the validity and completeness theorem that follow will prove that, every time, what is provable for each notion of demonstration is exactly the same, that is to say the tautologies of the propositional calculus.

In summary, the symbol $\models$ and the symbol $\vdash$ denotes exactly the same notion: For each of the variants of $\vdash$ mentioned in what follows, we have $\vdash F$ if and only if $\models F$, that is to say if and only if F is a tautology.

4.2 Proofs à la Frege and Hilbert

In this deduction system, we start from a set of axioms from propositional logic, that are tautologies, and we use a unique *deduction rule*, the *modus ponens*, also called *cut rule* that aims to capture a very usual type of reasoning in mathematics.

The modus ponens states that from a formula F and from a formula $F \Rightarrow G$, we deduce G.

Graphically:

$$\frac{F \quad (F \Rightarrow G)}{G}$$

Example 4.1 *For example, starting from* $(A \wedge B)$ *and from* $(A \wedge B) \Rightarrow C$ *we deduce* C.

We consider then a set of *axioms*, that are actually some instances of a finite number of axioms.

Definition 4.1 (Instance) *A formula F is said to be an instance of a formula G if F is obtained by substituting certain propositional variables in G by some formulas F_i.*

Example 4.2 *The formula* $((C \Rightarrow D) \Rightarrow (\neg A \Rightarrow (C \Rightarrow D)))$ *is an instance of* $(A \Rightarrow (B \Rightarrow A))$*, by taking* $(C \Rightarrow D)$ *for* A*, and* $\neg A$ *for* B*.*

Definition 4.2 (Axioms of boolean logic) *An* axiom of Boolean logic *is any instance of the following formulas:*

1. $(X_1 \Rightarrow (X_2 \Rightarrow X_1))$ *(axiom 1 for the implication);*
2. $((X_1 \Rightarrow (X_2 \Rightarrow X_3)) \Rightarrow ((X_1 \Rightarrow X_2) \Rightarrow (X_1 \Rightarrow X_3)))$ *(axiom 2 for the implication);*
3. $(X_1 \Rightarrow \neg\neg X_1)$ *(axiom 1 for the negation);*
4. $(\neg\neg X_1 \Rightarrow X_1)$ *(axiom 2 for the negation);*
5. $((X_1 \Rightarrow X_2) \Rightarrow (\neg X_2 \Rightarrow \neg X_1))$ *(axiom 3 for the negation);*
6. $(X_1 \Rightarrow (X_2 \Rightarrow (X_1 \wedge X_2)))$ *(axiom 1 for the conjunction);*
7. $((X_1 \wedge X_2) \Rightarrow X_1)$ *(axiom 2 for the conjunction);*
8. $((X_1 \wedge X_2) \Rightarrow X_2)$ *(axiom 3 for the conjunction);*
9. $(X_1 \Rightarrow (X_1 \vee X_2))$ *(axiom 1 for the disjunction);*
10. $(X_2 \Rightarrow (X_1 \vee X_2))$ *(axiom 2 for the disjunction);*
11. $((((X_1 \vee X_2) \wedge (X_1 \Rightarrow C)) \wedge (X_2 \Rightarrow C)) \Rightarrow C)$ *(axiom 3 for the disjunction).*

We obtain a notion of demonstration.

Definition 4.3 (Demonstration by modus ponens) *Let* T *be a set of propositional formulas, and* F *be some propositional formula. A* proof (by modus ponens) of F from T *is a finite sequence* $F_1, F_2, \cdots, F_n$ *of propositional formulas such that* F_n *is equal to* F*, and for all* i*, either* F_i *is in* T*, or* F_i *is some axiom of Boolean logic, or* F_i *is obtained by modus ponens from two formulas* F_j, F_k *with* $j < i$ *and* $k < i$*.*

We write $T \vdash F$ if F is provable (by modus ponens) from T. We write $\vdash F$ if $\emptyset \vdash F$, and we say that F is *provable (by modus ponens)*

Example 4.3 *Let* F, G, H *three propositional formulas. Here is a proof of* $(F \Rightarrow H)$ *from* $\{(F \Rightarrow G), (G \Rightarrow H)\}$*:*

- $F_1 : (G \Rightarrow H)$ *(hypothesis);*
- $F_2 : ((G \Rightarrow H) \Rightarrow (F \Rightarrow (G \Rightarrow H)))$ *(instance of axiom 1.);*
- $F_3 : (F \Rightarrow (G \Rightarrow H))$ *(modus ponens from* F_1 *and* F_2*);*

- $F_4 : ((F \Rightarrow (G \Rightarrow H)) \Rightarrow ((F \Rightarrow G) \Rightarrow (F \Rightarrow H)))$ *(instance of axiom 2.);*
- $F_5 : ((F \Rightarrow G) \Rightarrow (F \Rightarrow H))$ *(modus ponens from F_3 and F_4);*
- $F_6 : (F \Rightarrow G)$ *(hypothesis);*
- $F_7 : (F \Rightarrow H)$ *(modus ponens from F_6 and F_5).*

Exercise 4.1 *(solution on page 230) Prove $(F \Rightarrow F)$.*

In the following exercises, you can use the previous exercises to solve each of the questions.

Exercise 4.2 *(solution on page 230) [Deduction theorem] Let T be a family of propositional formulas, and let F and G be two propositional formulas. Prove that $T \vdash F \Rightarrow G$ is equivalent to $T \cup \{F\} \vdash G$.*

Exercise 4.3 *(solution on page 231) Prove the following assertions:*

- *$T \cup \{F\} \vdash G$ is equivalent to $T \cup \{\neg G\} \vdash \neg F$.*
- *If we have both $T \vdash F$ and $T \vdash \neg F$, then we have $T \vdash G$ for any formula G.*

Exercise 4.4 *(solution on page 231) Prove that $\{(\neg G \Rightarrow G)\} \vdash G$, for any formula G.*

Exercise 4.5 *(solution on page 231) Prove that if we have both $T \cup \{F\} \vdash G$ and $T \cup \{\neg F\} \vdash G$ then we have $T \vdash G$.*

Exercise 4.6 *Prove the following assertions:*

- $\{F\} \vdash \neg\neg F$
- $\{F, G\} \vdash F \vee G$
- $\{\neg F\} \vdash \neg(F \wedge G)$
- $\{\neg G\} \vdash \neg(F \wedge G)$
- $\{F\} \vdash F \vee G$
- $\{G\} \vdash F \vee G$
- $\{\neg F, \neg G\} \vdash \neg(F \vee G)$
- $\{\neg F\} \vdash (F \Rightarrow G)$
- $\{G\} \vdash (F \Rightarrow G)$
- $\{F, \neg G\} \vdash \neg(F \Rightarrow G)$

Exercise 4.7 *For v some partial function from $\{X_i\}$ in $\{0,1\}$, we set*

$$T_V = \{X_i | v(X_i) = 1\} \cup \{\neg X_i | v(X_i) = 0\}.$$

Prove that any formula H whose variables are among the domain of V, the relation $v \models H$ implies $T_V \vdash H$ and the relation $v \not\models H$ implies $T_V \vdash \neg H$.

This proof method is valid: By checking that all the axioms are tautologies, it is easy to get convinced by recurrence on the length of a proof that the following results are true.

Theorem 4.1 (Validity) *Every provable propositional formula is a tautology.*

What is less trivial, and more interesting is the converse: Any tautology has a proof of this type.

Theorem 4.2 (Completeness) *Every tautology is provable (by modus ponens).*

We will not do the proof of this result here, but this corresponds to the following exercise:

Exercise 4.8 *Prove this result by using the previous exercises: The key is the possibility to reason by cases (Exercise 4.5) and the Exercise 4.7 which make the relation between semantic and syntax.*

We have just described a deduction system that is very closed to the usual notion of proof in mathematics. However, this system is not easily exploitable to build an algorithm that would determine if a given formula F is a tautology.

This is easy to be convinced of that by trying to do the previous exercises, and by observing how hard it is to find a proof using this method.

4.3 Demonstrations by natural deduction

4.3.1 Rules from natural deduction

The previous notion of demonstration is in practise hard to use. Indeed, in the previous system, we are, somehow, constrained to keep the hypotheses during all the demonstration. We can not easily express some however common reasoning. We want to prove that $A \Rightarrow B$, supposing A and proving B under this hypothesis. This remarks leads to introduce a notion of couple made of a finite set of hypotheses and a conclusion. Such a couple is called a *sequent*.

We consider in this section that the propositional formulas also include $\bot$, interpreted by false, and $\top$ interpreted by true.

Definition 4.4 (Sequent) *A* sequent *is a couple $\Gamma \vdash A$, where Γ is a finite set of propositional formulas, and A is a propositional formula.*

The deduction rules of natural deduction are then the following:

$$\frac{}{\Gamma \vdash A}\ \text{axiom for each } A \in \Gamma$$

$$\frac{}{\Gamma \vdash \top}\ \top\text{-intro}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot\text{-elim}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\text{-intro}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\text{-elim}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\text{-elim}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\text{-intro}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\text{-intro}$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee\text{-elim}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow\text{-intro}$$

$$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \Rightarrow\text{-elim}$$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \neg\text{-intro}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot} \neg\text{-elim}$$

$$\frac{}{\Gamma \vdash A \vee \neg A} \text{ exclusive middle}$$

The rules $\top$-intro, $\wedge$-intro, $\vee$-intro, $\Rightarrow$-intro, $\neg$-intro, $\forall$-intro and $\exists$-intro are termed *introduction rules* and the rules $\bot$-elim, $\wedge$-elim, $\vee$-elim, $\Rightarrow$-elim, $\neg$-elim, $\forall$-elimand $\exists$-elimare termed *elimination rules*. The rules of natural deduction are hence classified in four groups: the introduction rules, the elimination rules, the *axiom* rule and the *exclusive middle* rule.

A *demonstration* of a sequent $\Gamma \vdash A$ is a derivation of this sequent, that is to say a tree whose nodes are labeled by a sequent, whose root is labeled by $\Gamma \vdash A$, and such that if a node is labeled by some sequent $\Delta \vdash B$, then its children are labeled by sequent $\Sigma_1 \vdash C_1, \ldots, \Sigma_n \vdash C_n$ such that there exists some natural deduction rule, that permits to deduce $\Delta \vdash B$ of $\Sigma_1 \vdash C_1, \ldots, \Sigma_n \vdash C_n$.

A sequent $\Gamma \vdash A$ is hence provable if there exists some demonstration of this sequent.

4.3.2 Validity and completeness

We can prove the following results:

Theorem 4.3 (Validity) *For any set of propositional formulas* Γ *and for any propositional formula* A*, if* $\Gamma \vdash A$ *is provable, then* A *is* a consequence *of* Γ.

Theorem 4.4 (Completeness) *Let* Γ *be any set of propositional formulas. Let* A *be some propositional formula that is* a consequence *of* Γ.
Then $\Gamma \vdash A$ *is provable.*

4.4 Proofs by resolution

We present briefly the notion of proof by resolution. This proof methods is maybe less natural, but is simpler to be implemented on a computer.

The resolution applies to a formula in conjunctive normal form. Since any propositional formula can be put in an equivalent conjunctive normal form, this is not restrictive.

Remark 4.2 *At least in appearance. Indeed, it requires to be more clever than in the previous chapter to transform a propositional formula in conjunctive normal form, if we want to avoid the problem of the explosion of the size of the*

formulas, and if we want to implement efficiently the method.

We call *clause* a disjunction of literals. Remember that a *literal* is a propositional variable or its negation. We represent a clause c by the set of the literals on which the disjunction applies.

Example 4.4 *We hence write* $\{p, \neg q, r\}$ *instead of* $p \vee \neg q \vee r$.

Given some literal u, we write $\overline{u}$ for the literal equivalent to $\neg u$: In other words, if u is the propositional variable p, $\overline{u}$ values $\neg p$, and if u is $\neg p$, $\overline{u}$ is p. Finally, we introduce some empty clause, denoted by $\square$, whose value is 0 for any valuation.

Definition 4.5 (Resolvant) *Let* C_1, C_2 *be two clauses. The clause* C *is a resolvent of* C_1 *and* C_2 *if there exists some literal* u *such that:*

- $u \in C_1$;
- $\overline{u} \in C_2$;
- C *is given by* $(C_1 \setminus \{u\}) \cup (C_2 \setminus \{\overline{u}\})$.

Example 4.5 *The clauses* $\{p, q, r\}$ *and* $\{\neg r, s\}$ *have* $\{p, q, s\}$ *as a resolvent.*

Example 4.6 *The clauses* $\{p, q\}$ *and* $\{\neg p, \neg q\}$ *have two resolvents, namely* $\{q, \neg q\}$ *and* $\{p, \neg p\}$. *The clauses* $\{p\}$ *and* $\{\neg p\}$ *have the resolvent* $\square$.

This provides a notion of demonstration:

Definition 4.6 (Proof by resolution) *Let* T *be a set of clauses. A* proof by resolution of T *is a finite sequence* $F_1, F_2, \cdots, F_n$ *of clauses such that* F_n *is equal to* $\square$, *and for every* i, *either* F_i *is a clause in* T, *or* F_i *is a resolvent of two clauses* F_j, F_k *with* $j < i$ *and* $k < i$.

Remark 4.3 *The modus ponens, at the heart of the previous Hilbert-Frege proof systems, consists in stating that from a formula* F *and a formula* $(F \Rightarrow G)$, *we deduce* G. *If we consider that the formula* $(F \Rightarrow G)$ *is equivalent to the formula* $(\neg F \vee G)$, *the modus ponens can also be seen as stating that from a formula* F *and a formula* $(\neg F \vee G)$, *we deduce* G, *which reads similar to the concept of resolvent. The resolvent of* $\{f\}$ *and of* $\{\neg f, g\}$ *is* $\{g\}$.

In some way, the resolvent is some generalized modus ponens, even if this analogy is only an analogy, and if a proof in a given proof system can not be translated directly into another.

Exercise 4.9 *Prove by resolution*

$$T = \{\{\neg p, \neg q, r\}, \{\neg p, \neg q, s\}, \{p\}, \{\neg s\}, \{q\}, \{t\}\}.$$

This proof method is valid (easy direction).

Theorem 4.5 (Validity) *Every clause that appears in a resolution proof of t is a consequence of T.*

Actually, to prove a formula, one reasons in general in this proof method on its negations, and one searches to prove that the negation is contradictory with the hypotheses. The validity is in general expressed in this way:

Corollary 4.1 (Validity) *If a set of clauses T admits some resolution proof, then T is contradictory.*

It turns out to be complete (harder direction).

Theorem 4.6 (Completeness) *Let T be a set of contradictory clauses. It admits some proof by resolution.*

4.5 Proofs by tableau method

We have considered up to now some valid and complete proofs systems, without providing a proof of our theorems. We will study more completely the tableau method. We have chosen to develop this method, since it is very algorithmic, and based on the notion of tree. This will contribute to our recurring argumentation that the notion of tree is everywhere in computer science.

4.5.1 Principle

We can first consider that our formulas are written using only the connectors $\neg, \wedge, \vee, \Rightarrow$, since the formula $(F \Leftrightarrow G)$ can be considered as an abbreviation of formula $((F \Rightarrow G) \wedge (G \Rightarrow F))$.

Suppose that we want to prove that F is a tautology. If the formula F is of the form $(F_1 \wedge F_2)$, then one can try to prove F_1 and to prove F_2. If the formula F is of the form $(F_1 \vee F_2)$, we write F_1, F_2, and we will explore two possibilities, one for the case F_1, and one for the case F_2.

We will basically bring all the possibilities to these two configurations: If the formula F is of the form $(F_1 \Rightarrow F_2)$, using the fact that it can be considered as $(F_2 \vee \neg F_1)$, we will use the rule of $\vee$, and if F is of the form $\neg(F_1 \Rightarrow F_2)$, using that it can be considered as $(F_1 \wedge \neg F_2)$, we will use the rule of $\wedge$. All other cases can be dealt similarly using de Morgan's laws.

Doing so systematically, we will build a tree, whose root is labeled by the negation of the formula F. In other words, to prove a formula F, the method starts from the negation of formula F.

Let's do it on the example of the following formula

$$(((p \wedge q) \Rightarrow r) \Rightarrow ((p \Rightarrow r) \vee (q \Rightarrow r))).$$

that we want to prove.

We start from $\neg F$, that is to say:

$$\neg(((p \wedge q) \Rightarrow r) \Rightarrow ((p \Rightarrow r) \vee (q \Rightarrow r))).$$

- by transforming the implication $\neg(F_1 \Rightarrow F_2)$ into equivalent formula $(F_1 \wedge \neg F_2)$, we obtain $((p \wedge q) \Rightarrow r) \wedge \neg((p \Rightarrow r) \vee (q \Rightarrow r))$ we get to the rule of $\wedge$.

- We then apply the rule of $\wedge$: We consider the formulas $((p \wedge q) \Rightarrow r)$ and $\neg((p \Rightarrow r) \vee (q \Rightarrow r))$.

- Let's consider the latter formula. From de Morgan's law, it can be considered as $(\neg(p \Rightarrow r) \wedge \neg(q \Rightarrow r))$: We can then associate to this formula $\neg(p \Rightarrow r)$ and $\neg(q \Rightarrow r)$ by the $\wedge$ rule.

- We consider then $\neg(p \Rightarrow r)$, and we get the formulas p and $\neg r$.

- We then obtain q and $\neg r$ from $\neg(q \Rightarrow r)$.

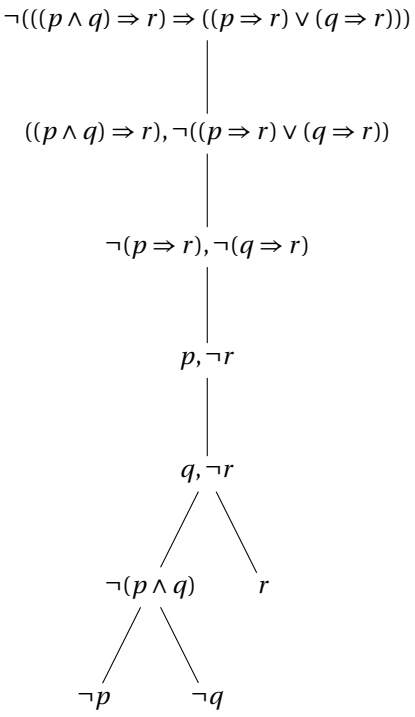
- We consider now $((p \wedge q) \Rightarrow r)$, that can be seen as $(r \vee \neg(p \wedge q))$. Thanks to the $\vee$ rule, we have the choice between the formula $\neg(p \wedge q)$ or r.

- The case of r is excluded by the previous step, where we had $\neg r$.

- In the first case, we still have the choice between $\neg p$ or $\neg q$. The two cases are excluded, since we had before p and q.

Since all branches lead to a contradiction, there is no possibility in which F could be false. Consequently, we deduce that F is a tautology.

The computation we have just done is naturally corresponding to the following tree:

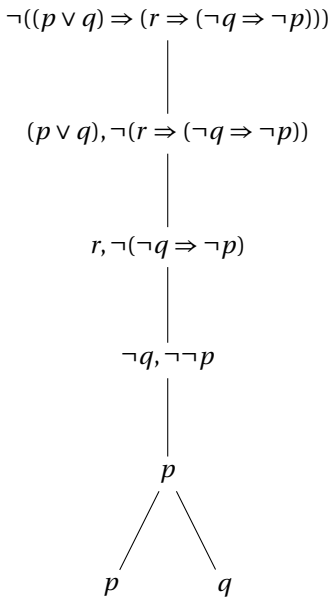


Each branch is a possible scenario. If a branch has a node labeled by some formula A such that $\neg A$ is appearing on the same branch (or their respective opposites), then one stops to develop this branch, and the branch is said to be *closed*: This means that we know that a contradiction is reached. If all the branches are closed, then we say that the *tree is closed*, and we are sure that all the possible scenarios are excluded.

Consider now the example of the formula G given by

$$((p \vee q) \Rightarrow (r \Rightarrow (\neg q \Rightarrow \neg p))).$$

With the same method, we build a tree whose root is $\neg G$.



On this example, we obtain a tree with two branches. The right branch is closed. The left branch is not: The propositional variables on this branch are $r, \neg q$ and p. By taking the valuation v with $v(r) = 1$, $v(q) = 0$, $v(p) = 1$, this provides the value 1 to $\neg G$, and hence 0 to G. In other words, we know that G is not a tautology. We say that the tree is *open. open* .

4.5.2 Description of the method

Let's now formalize the method. A *tableau* is a binary tree whose nodes are labeled by sets of propositional formulas, and that is built recursively from its root, vertices after vertices, by using two types of rules: The α rules and the β rules.

Remember that, in order to simplify the discussion, we have considered that the propositional formulas are written using only the connectors $\neg, \wedge, \vee, \Rightarrow$.

Formulas are divided in two groups, the α-group, and the β-group. To each formula, we associate inductively two new formulas according to the following rules:

- The formulas of the following form are α-formula:
  1. $\alpha = (A \wedge B)$. To such a formula is associated $\alpha_1 = A$ and $\alpha_2 = B$.
  2. $\alpha = \neg(A \vee B)$. To such a formula is associated $\alpha_1 = \neg A$ and $\alpha_2 = \neg B$.
  3. $\alpha = \neg(A \Rightarrow B)$. To such a formula is associated $\alpha_1 = A$ and $\alpha_2 = \neg B$.
  4. $\neg\neg A$: To such a formula is associated $\alpha_1 = \alpha_2 = A$.
- The formulas of the following form are β-formulas:
  1. $\beta = \neg(A \wedge B)$. To such a formula is associated $\beta_1 = \neg A$, $\beta_2 = \neg B$.

2. $\beta = (A \vee B)$. To such a formula is associated $\beta_1 = A$, $\beta_2 = B$.
3. $\beta = (A \Rightarrow B)$. To such a formula is associated $\beta_1 = \neg A$, $\beta_2 = B$.

If B a branch of a tableau, we denote by $\bigcup B$ the set of the formulas that appear on a vertex of B.

The two recursive rules to construct a tableau are the following:

1. An α rule consists in extending a finite branch of tableau T by the vertex labeled $\{\alpha_1, \alpha_2\}$, where α is some α-formula that appears on a vertex of B.
2. A β rule consists in extending a finite branch of a tableau T by two sons labeled respectively by $\{\beta_1\}$ and $\{\beta_2\}$, where β is some β-formula that appears in some vertex of B.

Remark 4.4 *Observe that this is not necessarily the last vertex of a branch B that is developed at each step, but a formula somewhere on the branch.*

A branch B is said to be *closed* if there exists some formula A such that A and $\neg A$ appears on the branch B. In the opposite case, the branch is said to be *open*.

A branch B is *developed* if

1. for any α-formula of $\bigcup B$, $\alpha_1 \in \bigcup B$ and $\alpha_2 \in \bigcup B$.
2. for any β-formula of $\bigcup B$, $\beta_1 \in \bigcup B$ or $\beta_2 \in \bigcup B$.

A tableau is said to be *developed* if all its branches are either closed or developed. A tableau is said to be *closed* if all its branches are closed. A tableau is said to be *open*open tree in tableau method if it has some open branch.

Finally, a *tableau* for a formula A (respectively for a set of formulas Σ) is a tableau whose root is labeled by $\{A\}$ (respectively by $\{A | A \in \Sigma\}$).

4.5.3 Termination of the method

First, observe that it is always possible to apply some α or β rules until a developed tableau is reached.

Proposition 4.1 *If Σ is a finite set of formulas, then there is some (finite) developed tableau for Σ.*

Proof: This is proved by recurrence on the number n of elements of Σ.

For the case $n = 1$, observe that the length of the formulas α_1, α_2, β_1, and β_2 is always strictly less than the length of α and β. The process of extension of branches that are not closed hence eventually terminates after finitely many steps. The array that is obtained a the end is developed, since otherwise it would have some extension.

For the case $n > 1$, we can write $\Sigma = \{F_1, \cdots, F_n\}$. Consider by recurrence hypothesis a developed tableau for $\Sigma = \{F_1, \cdots, F_{n-1}\}$. If this tableau is closed or if F_n is some

propositional variable, then this tableau is a developed tableau for Σ. Otherwise, we can extend all the open branches by applying all the rules corresponding to formula F_n, and by developing all the obtained branches. The process is terminating for the same reason as for the case $n = 1$. □

Remark 4.5 *Of course, from a given root, there are many ways to build some developed tableau.*

4.5.4 Validity

The previous method provides a proof method.

Definition 4.7 *A formula F said to be* provable by tableau *if there exists some closed tableau with the root $\{\neg F\}$. We then write $\vdash F$ when this holds.*

Exercise 4.10 *Prove that A is a consequence of $((A \vee \neg B) \wedge B)$ by the tableau method, i.e. $\vdash (((A \vee \neg B) \wedge B) \Rightarrow A)$.*

Exercise 4.11 *Prove that $\neg C$ is a consequence of $((H \wedge (P \vee C)) \Rightarrow A) \wedge H \wedge \neg A \wedge \neg P)$ by the tableau method.*

The method is valid.

Theorem 4.7 (Validity) *Any provable formula is a tautology.*

Proof: We will say that a branch B of a tableau is *realizable* if there exists some valuation v such that $v(A) = 1$ for any formula $A \in \bigcup B$ and $v(A) = 0$ if $\neg A \in \bigcup B$. A tableau is said to be *realizable* if it has some realizable branch.

We just need to prove the following result.

Lemma 4.1 *Let T' be some immediate extension of the tableau T: That is to say the tableau obtained by applying either an α or a β-rule to T. If T is realizable, then so does T'.*

This lemmas is sufficient to prove the theorem. Indeed, if F is provable, then there is some closed tableau whose root is $\neg F$. That means that in every branch, there is a formula A such that A and $\neg A$ appear on this branch, and hence none of the branches of T is realizable. By this lemma, this means that we started from a tree reduced to $\neg F$ that was not realizable. In other words, that F is a tautology.

It remains to prove the lemma. Let B be some realizable branch of T, and let B' the branch of T that is extended in T'. If $B \neq B'$, then B remains a realizable branch of T. If $B = B'$, then B is extended in T',

1. either in a branch B_α by some α-rule;
2. or by two branches B_{β_1} and B_{β_2} by some β-rule.

In the first case, let α be the formula used by the rule, and let v be a valuation that realizes B: From $v(\alpha) = 1$, we deduce that $v(\alpha_1) = 1$ and $v(\alpha_2) = 1$. Hence v is a valuation realizing B_α and the tableau T' is realizable.

In the second case, let β be the formula used by the β-rule. From $v(\beta) = 1$, we deduce that at least one of the values $v(\beta_1)$ and $v(\beta_2)$ values 1. Hence v realizes one of the branches B_{β_1} and B_{β_2}, and the tableau T' is realizable. □

4.5.5 Completeness

The method is complete. In other words, the converse of the previous theorem is true.

Theorem 4.8 (Completeness) *Every tautology is provable.*

Corollary 4.2 *Let F be some propositional formula.*
F is a tautology if and only if F is provable.

The rest of this subsection is devoted to prove this theorem.

Observe first that if B is a branch that is both developed and open in some tableau T, then then set $\bigcup B$ of the formulas that appear in B have the following properties:

1. there is no propositional variable such that $p \in \bigcup B$ and such that $\neg p \in \bigcup B$;
2. for every α-formula $\alpha \in \bigcup B$, $\alpha_1 \in \bigcup B$ and $\alpha_2 \in \bigcup B$;
3. for every β-formula $\beta \in \bigcup B$, $\beta_1 \in \bigcup B$ or $\beta_2 \in \bigcup B$.

Lemma 4.2 *Every developed and open branch is realizable.*

Proof: Let B some developed and open branch of tableau T. We defined a valuation v by:

1. if $p \in \bigcup B$, then $v(p) = 1$;
2. if $\neg p \in \bigcup B$, then $v(p) = 0$;
3. if $p \notin \bigcup B$ and $\neg p \notin \bigcup B$, then set (arbitrarily) $v(p) = 1$.

We prove by structural induction on a that: if $A \in \bigcup B$, then $v(A) = 1$, and if $\neg A \in \bigcup B$, then $v(A) = 0$.

Indeed, this is true for propositional variables.

If A is a α-formula, then $\alpha_1 \in \bigcup B$ and $\alpha_2 \in \bigcup B$. By induction hypothesis, $v(\alpha_1) = 1$, $v(\alpha_2) = 1$, and hence $v(\alpha) = 1$.

If A is a β-formula, then $\beta_1 \in \bigcup B$ or $\beta_2 \in \bigcup B$. By induction hypothesis, $v(\beta_1) = 1$ or $v(\beta_2) = 1$, and hence $v(\beta) = 1$. □

Proposition 4.2 *If there exists some closed tableau whose root is $\neg A$, then any developed tableau whose root is $\neg A$ is closed.*

Proof: By contradiction. Let T be some developed and open tableau whose root is $\neg A$, and let B be some open branch of T. By previous lemma, B is realizable, sand since $\neg A$ is in B, $\neg A$ is satisfiable. A is hence not a tautology, and hence not provable by tableau. There is no closed tableau with the root $\neg A$. □

We have all the ingredients to prove Theorem 4.8.

Suppose that A is not provable by tableau. Let T be a developed tableau whose root is $\neg A$. T is not closed. As in the previous proof, if B is some open branch of T, then B is realizable, and hence $\neg A$ is satisfiable. In other words, A is not a tautology.

4.5.6 One consequence of compactness theorem

Definition 4.8 *We will say that a set Σ of formulas is* refutable by tableau *if there exists some closed tableau with the root Σ.*

Corollary 4.3 *Every set Σ of formulas that is not satisfiable is refutable by tableau.*

Proof: By the compactness theorem, a set of formulas Σ that is not satisfiable has a finite subset Σ_0 that is not satisfiable. This finite set of formulas as a refutation by tableau, i.e. there is a closed table with the root Σ_0. This tableau also provides a closed tableau with the root Σ. □

4.6 Bibliographic notes

Suggested readings To go further on the notions of this chapter, we suggest to read [Cori & Lascar, 1993a], and [Mendelson, 1987] for the demonstration based on modus ponens, of [Stern, 1994] for a simple presentation of the proof methods based on resolution, and to [Lassaigne & de Rougemont, 2004] and [Nerode & Shore, 1997] for the tableau based methods.

Bibliography This chapter has been written by using [Cori & Lascar, 1993a], and [Dehornoy, 2006] for the part on proof methods based on modus ponens, and [Stern, 1994] for the presentation of proofs by resolution method. The part on natural deduction is taken from [Dowek, 2008]. The section on the tableau method is taken from book [Lassaigne & de Rougemont, 2004].

Chapter 5

Predicate calculus

Propositional calculus remains very limited, and permits essentially only to talk about Boolean operations on propositions.

If we want to reason about mathematical assertions, we need some richer constructions. For example, one may want to talk about statements like

$$\forall x((Prime(x) \wedge x > \mathbf{1}+\mathbf{1}) \Rightarrow Odd(x)). \tag{5.1}$$

Such a statement is not captured by propositional logic. First of all, since it uses some *predicates*, such as $Prime(x)$, whose truth value is depending on some variable x, which is not possible in propositional logic. Furthermore, we use here some *quantifiers*, such as $\exists$, $\forall$ which are not present in propositional logic.

The previous statement is an example of a formula from predicate calculus of *first order*. In this course, we will only talk about *first order logic*. The terminology *first order* makes reference to the fact that the existential and universal quantifiers are authorized only on variables.

A statement of *second order* (and one talks more generally of *higher order logic*) would be a statement where quantifications over functions or relations would be authorized. For example, we may want to write $\neg\exists f(\forall x(f(x) > f(x+\mathbf{1})))$ to mean that there does not exist some infinitely decreasing sequence. We will not attempt to understand the theory under this type of statements in this document, as we will see, the problems and difficulties with first order are already sufficiently numerous.

The objective of this chapter is then to define first order logic. As for propositional logic, we will do it by talking of the *syntax*, that is to say the way formulas are written, and then of their *semantic*, that is to say, their meanings.

The *predicate calculus*, remains the most usual formalism to express mathematical properties. This is also a formalism very often used in computer science to describe objects. For example, the request languages in data bases are essentially based on this formalism, applied to some finite objects, representing data.

5.1 Syntax

To write a *formula* of a first order language, we will use certain symbols that are common to all the languages, and certain symbols that change from a language to the other. The symbols that are common to all the languages are:

- the connectors $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$;
- the parentheses (and) and the comma ,;
- the universal quantifier $\forall$ and the existential quantifier $\exists$;
- an infinite denumerable set of symbols $\mathcal{V}$, called variables.

The symbols that may vary from a language to the other are captured by the notion of *signature*. A signature fixes the symbols of constants, the symbols of functions and the symbols of relations that are authorized.

Formally:

Definition 5.1 (Signature of a first order language) *The* signature

$$\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{R})$$

of a first order language is given by:

- *a first set $\mathcal{C}$ of symbols, called* constant symbols*;*
- *a second set $\mathcal{F}$ of symbols, called* function symbols*; To each symbol of this set is associated a strictly positive integer, that is called its* arity.
- *a third set $\mathcal{R}$ of symbols, called* relation symbols. *To each symbol of this set is associated a strictly positive integer, that is called its* arity.

We suppose that $\mathcal{V}$, $\mathcal{C}$, $\mathcal{F}$, $\mathcal{R}$ are pairwise disjoint sets.

A formula of first order will then be some particular word on the alphabet

$$\mathcal{A}(\Sigma) = \mathcal{V} \cup \mathcal{C} \cup \mathcal{F} \cup \mathcal{R} \cup \{\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, (,), , , \forall, \exists\}.$$

Remark 5.1 *In what follows, we will use the following conventions: We consider that x, y, z, u and v denotes some variables, that is to say some elements of $\mathcal{V}$. a, b, c, d will denote some constants, that is to say some elements of $\mathcal{C}$.*

The intuition is that the constant, functions and relation symbols will be interpreted (in what we will call *structures*). The *arity* of a function symbol or relation symbol will correspond to its number of arguments.

Example 5.1 *For example, we can consider the signature*

$$\Sigma = (\{\mathbf{0},\mathbf{1}\},\{s,+\},\{Odd, Prime,=,<\})$$

that has the constant symbols $\mathbf{0}$ *and* $\mathbf{1}$*, the function symbol* $+$ *of arity* 2*, the function symbol* s *of arity* 1*, the relation symbols* Odd *and* $Prime$ *of arity* 1*, the relation symbols* $=$ *and* $<$ *of arity* 2*.*

Example 5.2 *We can also consider the signature* $\mathscr{L}_2 = (\{c,d\},\{f,g,h\},\{R\})$ *with* c,d *two constant symbols,* f *a function symbol of arity* 1*,* g *and* h *two function symbols of arity* 2*,* R *a relation symbol of arity* 2*.*

We will define by successive steps, first the *terms*, that intend to represent objects, then the *atomic formulas* that intend to represent some relations between objects, and then the formulas.

5.1.1 Terms

We have already defined the terms in Chapter 2: What we call here *terms over a signature* Σ, is a term built on the union of function and constant symbols of the signature, and of the variables.

To be more clear, let's express again our definition:

Definition 5.2 (Termes sur une signature) *Let* $\Sigma = (\mathscr{C},\mathscr{F},\mathscr{R})$ *be a signature.*

The set T *of* terms *on the signature* Σ *is the language over alphabet* $\mathscr{A}(\Sigma)$ *inductively defined by:*

- (B) *every variable is a term:* $\mathscr{V} \subset T$*;*
- (B) *every constant is a term:* $\mathscr{C} \subset T$*;*
- (I) *if* f *is a function symbol of arity* n *and if* $t_1, t_2, \cdots, t_n$ *are terms, then* $f(t_1,\cdots,t_n)$ *is a term*

Definition 5.3 *A* closed term *is a term without any variable.*

Example 5.3 $+(x,s(+(\mathbf{1},\mathbf{1})))$ *is a term built over the signature of Example 5.1 that is not closed.* $+(+(s(\mathbf{1}),+(\mathbf{1},\mathbf{1})),s(s(\mathbf{0})))$ *is a closed term.*

Example 5.4 $h(c,x)$*,* $h(y,z)$*,* $g(d,h(y,z))$ *and* $f(g(d,h(y,z)))$ *are terms over the signature* $\mathscr{L}_2$ *of Example 5.2.*

5.1.2 Atomic formulas

Definition 5.4 (Atomic formulas) *Let $\Sigma = (\mathscr{C}, \mathscr{F}, \mathscr{R})$ be a signature.*

An atomic formula *on the signature Σ is a word on the alphabet $\mathscr{A}(\Sigma)$ of the form $R(t_1, t_2, \cdots, t_n)$, where $R \in \mathscr{R}$ is a relation symbol of arity n, and where $t_1, t_2, \cdots, t_n$ are terms over Σ.*

Example 5.5 *$>(x, +(\mathbf{1}, \mathbf{0}))$ is some atomic formula on the signature of Example 5.1. So is $=(x, s(y))$.*

Example 5.6 *$R(f(x), g(c, f(d)))$ is some atomic formula over $\mathscr{L}_2$.*

Remark 5.2 *We will agree to write sometimes $t_1 R t_2$ for some binary symbols, such as $=, <, +$ to avoid too heavy notations: For example, we will write $x > \mathbf{1} + \mathbf{1}$ for $>(x, +(\mathbf{1}, \mathbf{1}))$.*

5.1.3 Formulas

Definition 5.5 (Formules) *Let $\Sigma = (\mathscr{C}, \mathscr{F}, \mathscr{R})$ be a signature.*

The set of (of first order) formulas on the signature Σ is the language over alphabet $\mathscr{A}(\Sigma)$ inductively defined by:

- *(B) every atomic formula is a formula;*
- *(I) if F is a formula, then $\neg F$ is a formula;*
- *(I) if F and G are two formulas, then $(F \wedge G)$, $(F \vee G)$, $(F \Rightarrow G)$, and $(F \Leftrightarrow G)$ are formulas;*
- *(I) if F is a formula, and if $x \in \mathscr{V}$ is a variable, then $\forall x F$ is a formula, and $\exists x F$ is a formula.*

Example 5.7 *The statement $\forall x((Prime(x) \wedge x > \mathbf{1} + \mathbf{1}) \Rightarrow Odd(x))$ is a formula on the signature of Example 5.1.*

Example 5.8 *So does $\exists x(s(x) = \mathbf{1} + \mathbf{0} \vee \forall y\ x + y > s(x))$*

Example 5.9 *Examples of formulas over the signature $\mathscr{L}_2$:*

- $\forall x \forall y \forall z((R(x, y) \wedge R(y, z) \Rightarrow R(x, z))$
- $\forall x \exists y(g(x, y) = c \wedge g(y, x) = c)$;
- $\forall x \neg f(x) = c$;

- $\forall x \exists y \neg f(x) = c$.

5.2 First properties and definitions

5.2.1 Decomposition / Uniqueness reading

As for the propositional formulas, one can always decompose a formula, and in a unique way.

Proposition 5.1 (Decomposition / Unique reading) *Let F be a formula. Then F is of one, and exactly one of the following forms:*

1. *an atomic formula;*
2. *$\neg G$, where G is a formula;*
3. *$(G \wedge H)$ where G and H are formulas;*
4. *$(G \vee H)$ where G and H are formulas ;*
5. *$(G \Rightarrow H)$ where G and H are formulas;*
6. *$(G \Leftrightarrow H)$ where G and H are formulas;*
7. *$\forall x G$ where G is a formula and x is a variable;*
8. *$\exists x G$ where G is a formula and x is a variable.*

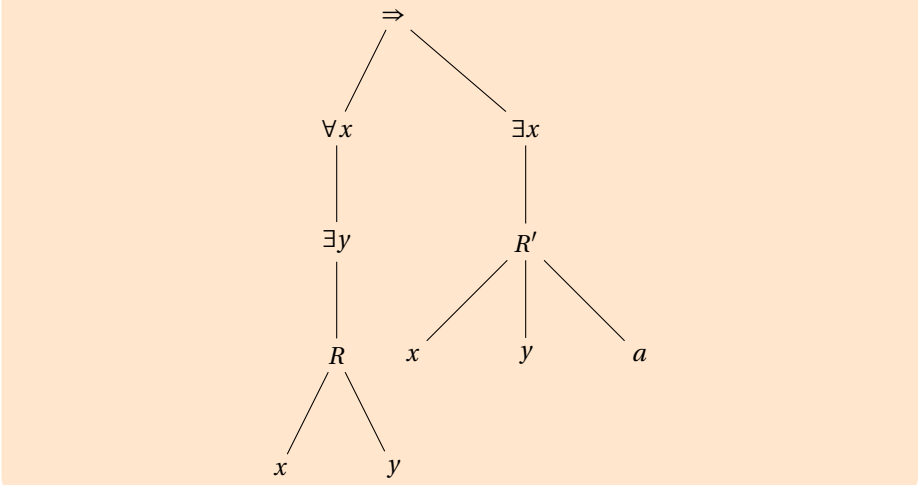
Furthermore, in the first case there is a unique way to "read"" the atomic formula. In all the other cases, there is unicity of the formula G and of the formula H with this property.

One can then represent each formula by a tree (its *decomposition tree*), that is in immediate correspondence with its derivation tree in the sense of Chapter 2): Each vertex is labeled by some constant, function or relation symbol, or by the symbols $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ or a quantifier or universal quantifier.

Example 5.10 *For example, the formula*

$$(\forall x \exists y R(x,y) \Rightarrow \exists x R'(x,y,a)) \tag{5.2}$$

is represented by the following tree



Each subtree of such a tree represents a *subformula* of F. If one prefers:

Definition 5.6 (Subformula) *A formula G is a subformula of a formula F if it appears in the decomposition of F.*

Exercise 5.1 *(solution on page 231) Let us fix a signature containing the relation symbols R_1, R_2 or respective arity 1 and 2. Let us fix the set of variables $\mathcal{V} = \{x_1, x_2, x_3\}$. Which of the following words are formulas?*

- $(R_1(x_1) \wedge R_2(x_1, x_2, x_3))$
- $\forall x_1 (R_1(x_1) \wedge R_2(x_1, x_2, x_3))$
- $\forall x_1 \exists R (R(x_1) \wedge R_2(x_1, x_1))$
- $\forall x_1 \exists x_3 (R_1(x_1) \wedge R_3(x_1, x_2, x_3))$

5.2.2 Free variables

The intuition of what follows is to distinguish the *free variables* from the other: All of this is about the "$\forall x$" and "$\exists x$" which are binders *binders*: When we write $\forall x F$ or $\exists x F$, then x become some bound variable. In other words, when we will talk about the semantic of formulas, the truth value of $\forall x F$ or $\exists x F$ will intend not to depend on x: We could well write $\forall y F(y/x)$ (respectively: $\exists y F(y/x)$) where $F(y/x)$ denotes intuitively the formula that is obtained by replacing x by y in formula F.

Remark 5.3 *We have exactly the same phenomenon in symbols such as the integral symbol in mathematics: In the expression $\int_a^b f(t)dt$, the variable t is some*

bound (dummy) variable. In particular $\int_a^b f(u)\,du$ *is exactly the same integral.*

Let's do this very properly. A same variable can appear several times in a given formula, and we need to be able to locate every occurrence, taking care to $\exists$ and $\forall$.

Definition 5.7 (Occurrence) *An* occurrence *of a variable* x *in some formula* F *is an integer* n *such that the* n*th symbol of word* F *is* x *and such that the* $(n-1)$*th symbol is not* $\forall$ *nor* $\exists$.

Example 5.11 8 *and* 17 *are occurrences of* x *in the formula* (5.2). 7 *and* 14 *are not:* 7 *because the* 7*th symbol of* F *is not an* x *(this is an open parenthesis) and* 14 *because the* 14*th symbol of* F *that is indeed a* x *is quantified by a* $\exists$.

Definition 5.8 (Free, bounded Variable)

- *An occurrence of a variable* x *in a formula* F *is a* bounded occurrence *if this occurrence appears in some subformula of* F *that is not starting by some quantifier* $\forall x$ *or* $\exists x$. *Otherwise the occurrence is said to be* free.
- *A variable is* free *in a formula if it has at least one free occurrence in the formula.*
- *A formula* F *is closed if it does not have any free variable.*

Example 5.12 *In the formula* (5.2)*, the occurrences* 8, 17 *and* 10 *of* x *and* y *are bounded. The occurrence* 19 *of* y *is free.*

Example 5.13 *In the formula* $(R(x,z) \Rightarrow \forall z(R(y,z) \vee y = z))$*, the only occurrence of* x *is free, the two other occurrences of* y *are free. The first (least) occurrence of* z *is free, and the others are bounded. The formula* $\forall x \forall z(R(x,z) \Rightarrow \exists y(R(y,z) \vee y = z))$ *is closed.*

The notation $F(x_1, \cdots, x_k)$ means that the free variables of the formula F are among $x_1, \cdots, x_k$.

Exercise 5.2 *(solution on page 231) Find all the free and the bounded occurrences in the following formulas:*

- $\exists x(l(x) \wedge m(x))$
- $(\exists x l(x)) \wedge m(x)$

Exercise 5.3 *Prove that the* free!variable $\ell(F)$ *of a formula* F *can be obtained by the following inductive definition:*

- $\ell(R(t_1,\cdots,t_n)) = \{x_i | x_i \in \mathcal{V}$ *and* x_i *appears in* $R(t_1,\cdots,t_n)\}$;
- $\ell(\neg G) = \ell(G)$;
- $\ell(G \vee H) = l(G \wedge H) = \ell(G \Rightarrow H) = \ell(G \Leftrightarrow H) = \ell(G) \cup \ell(H)$;
- $\ell(\forall x F) = \ell(\exists x F) = \ell(F) \setminus \{x\}$.

5.3 Semantic

We can now talk about the meaning that we give to formulas. Actually, to provide a meaning to formulas, we need to fix some meaning of the symbols of the signature, and this is the purpose of the notion of structure.

Definition 5.9 (Structure) *Let* $\Sigma = (\mathscr{C}, \mathscr{F}, \mathscr{R})$ *be a signature.*
A structure $\mathfrak{M}$ of signature Σ *is given by:*

- *a non-empty set* M*, called its* base set*, or* domain *of the structure;*
- *an element, denoted by* $c^{\mathfrak{M}}$*, for each constant symbol* $c \in \mathscr{C}$;
- *a function, denoted by* $f^{\mathfrak{M}}$*, of* $M^n \to M$ *for each function symbol* $f \in \mathscr{F}$ *of arity* n ;
- *a subset, denoted by* $R^{\mathfrak{M}}$*, of* M^n *for each relation symbol* $R \in \mathscr{F}$ *of arity* n.

We say that the constant c (respectively the function f, the relation R) is *interpreted* by $c^{\mathfrak{M}}$ (resp. $f^{\mathfrak{M}}$, $R^{\mathfrak{M}}$). A structure is sometimes also called a *realisation* of the signature.

Example 5.14 *A realisation of the signature* $\Sigma = (\{\mathbf{0},\mathbf{1}\},\{+,-\},\{=,>\})$ *corresponds to the domain* $\mathbb{N}$ *of natural integers, with* $\mathbf{0}$ *interpreted by the integer* 0, $\mathbf{1}$ *interpreted by* 1, $+$ *interpreted by addition,* $-$ *interpreted by subtraction, and* $=$ *by equality on the integers: That is to say by the subset* $\{(x,x)|x \in \mathbb{N}\}$*, and* $>$ *by the order on the integers, that is to say by the subset* $\{(x,y)|x > y\}$*. It can be denoted by* $(\mathbb{N},=,<,+,-,0,1)$.

Example 5.15 *Another realisation of this signature corresponds to the domain* $\mathbb{R}$ *of the reals, where* $\mathbf{0}$ *is interpreted by the real* 0, $\mathbf{1}$ *by the real* 1, $+$ *by addition,* $-$ *by subtraction, and* $=$ *by equality on the reals, and* $>$ *by the order on the reals. It can be denoted by* $(\mathbb{R},=,<,+,-,0,1)$.

Example 5.16 *We can obtain a realisation of the signature $\mathscr{L}_2$ by considering the base set $\mathbb{R}$ of the reals, by interpreting R as the order relation $\leq$ on the reals, the function f as the function that to x associates $x+1$, the functions g and h as respectively the addition and the multiplication on the reals, the constants c and d as the reals 0 and 1. It can be denoted by $(\mathbb{R}, \leq, s, +, \times, 0, 1)$.*

We will then use the notion of structure to interpret the terms, the atomic formulas, and then inductively the formulas as one may expect.

5.3.1 Interpretation of terms

Definition 5.10 (Valuation) *Fix a structure $\mathfrak{M}$. A* valuation *v is a distribution of values to the variables, that is to say a function from $\mathscr{V}$ to the domain M of the structure $\mathfrak{M}$.*

Definition 5.11 (Interprétation des termes) *Let $\mathfrak{M}$ be a structure of signature $\Sigma = (\mathscr{C}, \mathscr{F}, \mathscr{R})$.*

Let t be a term of the form $t(x_1, \cdots, x_k)$ over Σ whose free variables are $x_1, \cdots, x_k$.

Let v be a valuation.

The interpretation *$t^{\mathfrak{M}}$* of term *t* for the valuation *v, also denoted by $t^{\mathfrak{M}}[v]$, or $t^{\mathfrak{M}}$ is defined inductively as follows:*

(*B*) *every variable is interpreted as its value by the valuation: if t is the variable $x_i \in v$, then $t^{\mathfrak{M}}$ is $v(x_i)$;*

(*B*) *every constant is interpreted as its interpretation in the structure: if t is the constant $c \in \mathscr{C}$, then $t^{\mathfrak{M}}$ is $c^{\mathfrak{M}}$;*

(*I*) *each function symbol is interpreted as its interpretation int the structure: if t is the term $f(t_1, \cdots, t_n)$, then $t^{\mathfrak{M}}$ est $f^{\mathfrak{M}}(t_1^{\mathfrak{M}}, \cdots, t_n^{\mathfrak{M}})$, where $t_1^{\mathfrak{M}}, \cdots, t_n^{\mathfrak{M}}$ are the respective interpretations of the terms $t_1, \cdots, t_n$.*

Remark 5.4 *The interpretation of a term is an element of M, where M is the base set of the structure $\mathfrak{M}$. In other words, the terms denote some elements of the structure.*

Example 5.17 *Let $\mathscr{N}$ be the structure $(\mathbb{N}, \leq, s, +, \times, 0, 1)$ of signature*

$$\mathscr{L}_2 = (\{c, d\}, \{f, g, h\}, \{R\}) :$$

- *the interpretation of $h(d, x)$ for a valuation such that $v(x) = 2$ is 2.*
- *the interpretation of term $f(g(d, h(y, z)))$ for a valuation such that $v(y) = 2$, $v(z) = 3$ is 8.*

5.3.2 Interpretation of atomic formulas

An atomic formula $F = F(x_1, \cdots, x_k)$ is an object that is interpreted either by *true* or by *false* in some valuation v. When F is interpreted by true, we say that the *valuation v satisfies F*, and this fact is denoted by $v \models F$. We denote $v \not\models F$ in the contrary case.

There only remain to define formally this notion:

Definition 5.12 (Interpretation of some atomic formula) *Let $\mathfrak{M}$ be a structure of signature $\Sigma = (\mathscr{C}, \mathscr{F}, \mathscr{R})$.*

The valuation v satisfies the atomic formula $R(t_1, t_2, \cdots, t_n)$ of free variables $x_1, \cdots, x_k$ if $(t_1^{\mathfrak{M}}[v], t_2^{\mathfrak{M}}[v], \cdots, t_n^{\mathfrak{M}}[v]) \in R^{\mathfrak{M}}$, where $R^{\mathfrak{M}}$ is the interpretation of relation symbol R in the structure.

Example 5.18 *For example, on the structure of Example 5.14, $x > \mathbf{1} + \mathbf{1}$ is interpreted by 1 (true) in the valuation $v(x) = 5$, and by 0 (false) in the valuation $v(x) = 0$. The atomic formula $\mathbf{0} = \mathbf{1}$ is interpreted by 0 (false).*

Example 5.19 *On the structure $\mathscr{N}$ of Example 5.17, the atomic formula $R(f(c), h(c, f(d)))$ is interpreted by false.*

5.3.3 Interpretation of formulas

More generally, a formula $F = F(x_1, \cdots, x_k)$ is an object that is interpreted either by *true* or by *false* in some valuation v. When F interprets to true, we say that *the valuation v satisfies F*, and we write this fact by $v \models F$, and $v \not\models F$ for the contrary case.

Definition 5.13 (Interpretation of some formula) *Let $\mathfrak{M}$ be a structure of signature $\Sigma = (\mathscr{C}, \mathscr{F}, \mathscr{R})$.*

The expression "the valuation v satisfies the formula $F = F(x_1, \cdots, x_k)$", denoted by $v \models F$, is defined inductively in the following way:

(B) *it has already been defined for some atomic formula;*

$\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow$ are interpreted exactly as in the propositional calculus:

(I) *the negation is interpreted by the logical negation:*
if F is of the form $\neg G$, then $v \models F$ if and only if $v \not\models G$;

(I) *$\wedge$ is interpreted as the logical conjunction:*
if F is of the form $(G \wedge H)$, then $v \models F$ if and only if $v \models G$ and $v \models H$;

(I) *$\vee$ is interpreted as the logical or*
if F is of the form $(G \vee H)$, then $v \models F$ if and only if $v \models G$ or $v \models H$;

(I) *$\Rightarrow$ is interpreted as the logical implication:*
if F is of the form $(G \Rightarrow H)$, then $v \models F$ if and only if $v \models H$ or $v \not\models G$;

(*I*) *$\Leftrightarrow$ is interpreted as the logical equivalence:*
if F is of the form $(G \Leftrightarrow H)$, then $v \models F$ if and only if ($v \models G$ and $v \models H$) or ($v \not\models G$ and $v \not\models H$).

$\exists x$ and $\forall x$ are interpreted as existential and universal quantifications:

(*I*) *if F is of the form $\forall x_0 G(x_0, x_1, \cdots, x_k)$, then $v \models F$ if and only if for all $a_0 \in M$ $v' \models G$, where v' is the valuation such that $v'(x_0) = a_0$, and $v'(x) = v(x)$ for all $x \neq x_0$;*

(*I*) *if F is of the form $\exists x_0 G(x_0, x_1, \cdots, x_k)$, then $v \models F$ if and only for a certain element $a_0 \in M$, we have $v' \models G$, where v' is the valuation such that $v'(x_0) = a_0$, and $v'(x) = v(x)$ for every $x \neq x_0$.*

Example 5.20

- *The formula $F(x)$ defined by $\forall y R(x, y)$ is true in the structure $\mathcal{N}$ for 0 (i.e. for a valuation such that $v(x) = 0$), but false for all the other integers.*
- *The formula $G(x)$ defined by $\exists y\, x = f(y)$ is true in the structure $\mathcal{N}$ for the integers distinct from 0 and false for 0.*
- *The closed formula $\forall x \forall z \exists y (x = c \vee g(h(x, y), z) = c)$ of language $\mathcal{L}_2$ is true in $(\mathbb{R} \leq, s, +, \times, 0, 1)$ and false in $\mathcal{N} = (\mathbb{N}, \leq, s, +, \times, 0, 1)$.*

In the case where the valuation v satisfies the formula F, one also says that F is *true* in v. In the contrary, we say that F is *false* in v.

Definition 5.14 (Model of a formula) *For a closed formula F, the satisfaction of F in a structure $\mathfrak{M}$ is not depending on the valuation v. In the case where the formula F is true, we say that the structure $\mathfrak{M}$ is a model of F, and we write $\mathfrak{M} \models F$.*

Exercise 5.4 *(solution on page 231) Let Σ be a signature made of some binary relation R and of the predicate $=$. Write some formula that is valid if and only if R is some order (we can suppose that $=$ is interpreted by equality).*

5.3.4 Substitutions

Definition 5.15 (Substitution in a term) *Given some term t and some variable x appearing in this term, we can replace all the occurrences of x by some other term t'. The new term is said to be obtained by* substitution *of t' to x in t, and is denoted by $t(t'/x)$.*

Example 5.21 *The result of the substitution of $f(h(u,y))$ to x in $g(y,h(c,x))$ is $g(y,h(c,f(h(u,y))))$. The result of the substitution of $g(x,z)$ to y in this new term is*

$$g(g(x,z),h(c,f(h(u,g(x,z))))).$$

To do a substitution of a term to some free variable in some formula, it is necessary to does it carefully: Otherwise the meaning of the formula can be completely modified by the phenomenon of capture of variables.

Example 5.22 *Let $F(x)$ be the formula $\exists y(g(y,y)=x)$. In the structure $\mathcal{N}$ where g is interpreted by addition the meaning of $F(x)$ is clear: $F(x)$ is true in x if and only if x is even.*

If we replace the variable x by z, the obtained formula has the same meaning that the formula $F(x)$ (up to the renaming of the free variable). $F(z)$ is true in z if and only if z is even.

But if we replace x by y, the obtained formula $\exists y(g(y,y)=y)$ is a closed formula that is true in the structure $\mathcal{N}$. The variable x have been replaced by a variable that is quantified in the formula F.

Definition 5.16 (Substitution) *The* Substitution *of a term t to a free variable x in some formula F is obtained by replacing all the free occurrences of this variable by the term t,under the reserve that the following condition is satisfied: For every variable y appearing in t, y has no free occurrence appearing in a subformula of F starting by a $\forall$ or $\exists$ quantifier. The result of this substitution, if it is possible, is denoted by $F(t/x)$.*

Example 5.23 *The result of the substitution of the term $f(z)$ to the variable x in the formula $F(x)$ given by*

$$(R(c,x)\wedge\neg x=c)\wedge(\exists y g(y,y)=x))$$

is the formula

$$(R(c,f(z))\wedge\neg f(z)=c)\wedge(\exists y g(y,y)=f(z))).$$

Proposition 5.2 *If F is a formula, x is some free variable in F, and t is a term such that the substitution of t in x in F is defined, then the formulas $(\forall xF\Rightarrow F(t/x))$ and $(F(t/x)\Rightarrow\exists xF)$ are valid.*

Proof: We prove by induction on the formula F that the satisfaction of the formula $F(t/x)$ by the valuation v is equivalent to the one of formula $F(x)$ by the valuation v_1 where v_1 is obtained from v by giving to x the interpretation of t for the valuation v.

The only cases requiring a justification are those where the formula F is of the form $\forall G$ and $\exists xG$. From the hypothesis for the substitution of t to x, the considered quantification is about some variable y distinct both from x and from all the variables from t. It suffices then to examine the satisfaction of the formula $G(t/x)$ by a valuation v' equals to v but on y. By induction hypothesis on G, the formula $G(t/x)$ is satisfied by v' if and only if G is satisfied by the valuation v'_1 where v'_1 is obtained from v' by giving to x the interpretation of t for the valuation v': Indeed, v and v' are equal on all the variables appearing in the term t. □

5.4 Equivalence, Normal forms

5.4.1 Equivalent formulas

Definition 5.17 *Let $\Sigma = (\mathscr{C}, \mathscr{F}, \mathscr{R})$ be as signature.*

- *A structure $\mathfrak{M}$ satisfies the formula $F(x_1, \cdots, x_k)$ if it satisfies the closed formula $\forall x_1 \cdots \forall x_k F(x_1, \cdots, x_k)$. This latter formula is called the* universal closure.
- *A closed formula F is said* valid *if it is satisfied by any structure $\mathfrak{M}$.*
- *A formula F is said* valid *if its universal closure is valid.*
- *Two formulas F and G are equivalent if for any structure, and for any valuation v, the formulas F and G take the same truth value. We write $F \equiv G$ in this case.*

Exercise 5.5 *Prove that the relation $\equiv$ is an equivalence relation.*

Proposition 5.3 *Let F be a formula. We have the following equivalences*

$$\neg \forall x F \equiv \exists x \neg F$$

$$\neg \exists x F \equiv \forall x \neg F$$

$$\forall x \forall y F \equiv \forall y \forall x F$$

$$\exists x \exists y F \equiv \exists y \exists x F$$

Proposition 5.4 *Suppose that the variable x is not free in the formula G. Let F*

be a formula. We have then the following equivalences:

$$\forall x G \equiv \exists x G \equiv G \tag{5.3}$$
$$(\forall x F \vee G) \equiv \forall x (F \vee G) \tag{5.4}$$
$$(\forall x F \wedge G) \equiv \forall x (F \wedge G) \tag{5.5}$$
$$(\exists x F \vee G) \equiv \exists x (F \vee G) \tag{5.6}$$
$$(\exists x F \wedge G) \equiv \exists x (F \wedge G) \tag{5.7}$$
$$(G \wedge \forall x F) \equiv \forall x (G \wedge F) \tag{5.8}$$
$$(G \vee \forall x F) \equiv \forall x (G \vee F) \tag{5.9}$$
$$(G \wedge \exists x F) \equiv \exists x (G \wedge F) \tag{5.10}$$
$$(G \vee \exists x F) \equiv \exists x (G \vee F) \tag{5.11}$$
$$(\forall x F \Rightarrow G) \equiv \exists x (F \Rightarrow G) \tag{5.12}$$
$$(\exists x F \Rightarrow G) \equiv \forall x (F \Rightarrow G) \tag{5.13}$$
$$(G \Rightarrow \forall x F) \equiv \forall x (G \Rightarrow F) \tag{5.14}$$
$$(G \Rightarrow \exists x F) \equiv \exists x (G \Rightarrow F) \tag{5.15}$$

Each of the equivalence is rather simple to be established, but tedious, and we leave the proofs a exercises.

Exercise 5.6 *Prove Proposition 5.4.*

Exercise 5.7 *(solution on page 231) Are the following propositions equivalent ? If not, does the proposition on the left implies the one one the right?*

1. $\neg(\exists x P(x))$ *and* $(\forall x \neg P(x))$
2. $(\forall x P(x) \wedge Q(x))$ *and* $((\forall x P(x)) \wedge (\forall x Q(x)))$
3. $((\forall x P(x)) \vee (\forall x Q(x)))$ *and* $(\forall x P(x) \vee Q(x))$
4. $(\exists x P(x) \vee Q(x))$ *and* $((\exists x P(x)) \vee (\exists x Q(x)))$
5. $(\exists x P(x) \wedge Q(x))$ *and* $((\exists x P(x)) \wedge (\exists x Q(x)))$
6. $(\exists x \forall y P(x,y))$ *and* $(\forall y \exists x P(x,y))$

5.4.2 Prenex normal form

Definition 5.18 (Prenex form) *A formula F is said to be* in prenex form *if it is of the form*

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n F'$$

where each of the Q_i is either a $\forall$ quantifier or a $\exists$ quantifier, and F' is a formula not containing any quantifier.

Proposition 5.5 *Every formula F is equivalent to some formula in prenex normal form G.*

Proof: By structural induction on F.

Base case. If F is of the form $R(t_1, \cdots, t_n)$, for some relation symbol R, then F is in prenex normal form.

Inductive case:

- If F is of the form $\forall xG$ where $\exists xG$, by induction hypothesis G is equivalent to G' in prenex normal form, and so F is equivalent to $\forall xG'$ or $\exists xG'$ that is in prenex normal form.
- If F is of the form $\neg G$, by induction hypothesis G is equivalent to G' in prenex normal form $Q_1x_1Q_2x_2\cdots Q_nx_nG''$. By using the equivalences of the Proposition 5.3, F is equivalent to $Q'_1x_1Q'_2x_2\cdots Q'_nx_n\neg G''$, by taking $Q'_i = \forall$ if $Q_i = \exists$ and $Q'_i = \exists$ if $Q_i = \forall$.
- If F is of the form $(G \wedge H)$, by induction hypothesis G and H are equivalent to formulas G' and H' in prenex normal form. By applying the equivalences (5.4) à (5.11), we can "bring up" the quantifiers in front of the formula: We need to proceed with care, since for example $F = (F_1 \wedge F_2) = ((\forall xF'_1) \wedge F'_2)$ with x free in F'_2, we need first to rename the variable x in F_1 by replacing x by some new variable z not appearing nor in F_1 nor in F'_2, in order to be able to use the required equivalence among the equivalences (5.4) à (5.11).
- The other cases are treated in a similar way, by using the equations of the two previous propositions.

□

By using the idea of the conjunctive and disjunctive normal form of propositional calculus, we can even go further:

Definition 5.19
- *A* literal *is some atomic formula or the negation of some atomic formula.*
- *A* clause *is a disjunction of literals.*
- *A prenex formula $Q_1x_1Q_2x_2\cdots Q_nx_nG$ is in* conjunctive normal form *if the quantifier free formula G is a clause or a conjunction of clauses.*

The notion of *disjunctive normal form* can be defined in a dual way by considering disjunctions of conjunctions of atomic formulas instead of conjunctions of disjunctions of atomic formulas.

Proposition 5.6 *Every formula F is equivalent to some prenex formula*

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n G,$$

where G is in conjunctive normal form.

Proposition 5.7 *Every formula F is equivalent to some prenex formula*

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n G,$$

where G is in disjunctive normal form.

Proof: Let F be a formula and $Q_1 x_1 Q_2 x_2 \cdots Q_n x_n G$ a prenex equivalent formula equivalent to F. We denote by $A_1, A_2, \ldots, A_k$ the atomic formulas that appear in G. We can define a formula $H$of propositional calculus that uses the variables $\{p_1, p_2, \ldots, p_k\}$ such that the formula G corresponds to the formula $H(A_1/p_1, A_2/p_2, \ldots, A_k/p_k)$. Let H' be a conjunctive (respectively: disjunctive) normal form equivalent to H, obtained int he propositional calculus.

The formula G is equivalent to the formula G' given by expression $H'(A_1/p_1, A_2/p_2, \ldots, A_k/p_k)$ and then F is equivalent to $Q_1 x_1 Q_2 x_2 \cdots Q_n x_n G'$ in conjunctive (resp. disjunctive) normal form. □

Exercise 5.8 *(solution on page 232) Determine an equivalent prenex normal form equivalent to*

$$(\exists x P(x) \wedge \forall x(\exists y Q(y) \Rightarrow R(x))).$$

Exercise 5.9 *Determine an equivalent normal form equivalent to*

$$(\forall x \exists y R(x,y) \Rightarrow \forall x \exists y (R(x,y) \wedge \forall z (Rxz \Rightarrow (Ryz \vee y = z))))$$

and to

$$\forall x \forall y ((R(x,y) \wedge \neg x = y) \Rightarrow \exists z (y = g(x, h(z,z)))).$$

5.4.3 Skolem form

The previous results where about transformations on formulas preserving the equivalence.

We will now focus on weaker transformations in order to eliminate the existential quantifiers. Starting form some closed formula F, we will obtain some formula F' that will not be necessarily equivalent. The formula F' will be written on a signature where possibly some new constant and function symbols have been added. It will have a model if and only if the initial formula has one.

Definition 5.20 *Let $\Sigma = (\mathscr{C}, \mathscr{F}, \mathscr{R})$ be a signature.*

- *A formula F is said to be* universal *it it is prenex and all the quantifiers appearing in F are $\forall$ quantifiers.*
- *A signature $\Sigma' = (\mathscr{C}', \mathscr{F}', \mathscr{R}')$ is a* Skolem extension *of Σ if it is obtained by adding to Σ some function symbols (possibly infinitely many) and some constant symbols (possibly infinitely many).*

A closed prenex formula of F of Σ' is either universal or of the form

$$\forall x_1 \forall x_2 \ldots \forall x_k \exists x G$$

where G is prenex. In the latter case, it may happen that $k = 0$ and F is then of the form $\exists x G$.

The transformation that we will apply consists in associating to F a formula F_1 given by $\forall x_1 \forall x_2 \ldots \forall x_k G(f(x_1, \ldots, x_k)/x)$ where f is some function symbol not appearing in formula G. In the particular case where F is $\exists x G$ (i.e. the case $k = 0$), we will associate some formula F_1 given by $G(c)$ where c is a constant symbol not appearing in formula G.

The formula F_1 obtained in this way has one less existential quantifier than the formula F.

Example 5.24 *To the formula F given by*

$$\forall x \forall y \exists z (R(f(x), g(z, y)) \Rightarrow (R(f(x), z) \wedge R(z, h(x, y))))$$

on the signature $\Sigma = (\{a, b\}, \{f, g, h\}, \{R\})$, we will associate the formula F_1 given by

$$\forall x \forall y (R(f(x), g(k(x, y), y)) \Rightarrow (R(f(x), k(x, y)) \wedge R(k(x, y), h(x, y))))$$

on the signature $\Sigma' = (\{a, b\}, \{f, g, h, k\}, \{R\})$ where we have added the symbol k or arity 2.

F has a model if and only if F' has a model.

Definition 5.21 *Let F be a closed prenex formula on the signature Σ' that has n existential quantifiers.*

- *A Skolem form of F is a formula obtained by applying n times successively the previous transformation.*
- *The new functions and constants introduced in these transformations are called the* Skolem functions and constants.

By construction, the Skolem form of F is some universal formula.

Example 5.25 *Staring from F given by*

$$\exists x \forall y \forall x' \exists y' \forall z (R(x,y) \Rightarrow (R(x',y') \wedge (R(x',z) \wedge (R(x',z) \Rightarrow (R(y',z) \vee y' = z)))))$$

a Skolem form of F is the formula

$$\forall y \forall x' \forall z (R(e,y) \Rightarrow (R(x',k(y,x')) \wedge (R(x',z) \Rightarrow (R(k(y,x'),z) \vee (k(y,x') = z)))))$$

The interest of this transformation lies in the following result:

Theorem 5.1 *Let F' be a Skolem form of F. Then F' has a model if and only if F has a model.*

Proof: We only need to prove that the property is true when F' is obtained from F by some of the transformation above (and repeat n times the argument in the general case): If F is given by

$$\forall x_1 \forall x_2 \ldots \forall x_k \exists x G$$

then F_1 is given by $\forall x_1 \forall x_2 \ldots \forall x_k G(f(x_1,\ldots,x_k)/x)$. If F_1 has a model, then F has a model: This comes from the validity of the formula

$$\forall x_1 \forall x_2 \ldots \forall x_k G(f(x_1,\ldots,x_k)/x) \Rightarrow \forall x_1 \forall x_2 \ldots \forall x_k \exists x G$$

The case $k = 0$ follows from the validity of the formula

$$G(c) \Rightarrow \exists x G(x).$$

To prove the converse direction, suppose that F has a model $\mathfrak{M}$ of base set M. It suffices to define the interpretation of the corresponding Skolem constant or function. If $F = \forall x_1 \forall x_2 \ldots \forall x_k \exists x G$ the interpretation of the Skolem function f is given by taking for each sequence $a_1, a_2, \ldots, a_k$ of elements of M an element $f^{\mathfrak{M}}(a_1, a_2, \ldots, a_k)$ among the $a \in M$ such that

$$\mathfrak{M} \models G(a_1, a_2, \ldots, a_k),$$

which is possible since $\mathfrak{M}$ is a model of F.

If F is of the form $\exists x G$, the interpretation of the Skolem constant c is taken by taking an element $c^{\mathfrak{M}}$ among the $b \in M$ satisfying G in $\mathfrak{M}$. □

5.5 Bibliographic notes

Suggested readings To go further on the notions of this chapter, we suggest [Cori & Lascar, 1993a], [Dowek, 2008] or [Lassaigne & de Rougemont, 2004].

Bibliography This chapter has been written by using essentially [Cori & Lascar, 1993a] and [Lassaigne & de Rougemont, 2004].

Chapter 6

Models. Completeness.

We can now describe various objects, and talk about their properties. We have indeed all the ingredients to talk about models and theories. In this chapter, after a few examples, we will then focus on the *completeness theorem*.

The basic concept is the the concept of theory.

Definition 6.1 (Theory)
- *A* theory $\mathcal{T}$ *is a set of closed formulas over some given signature. The formulas of a theory are called the* axioms *of this theory.*
- *A structure* $\mathfrak{M}$ *is a* model of the theory $\mathcal{T}$ *if* $\mathfrak{M}$ *is a model of each of the formulas of the theory.*

Definition 6.2 (Consistant theory) *A theory is said to be* consistent *if it has a model. It is said* inconsistent *if it is not consistent.*

Of course, the inconsistent theories have less interest.

Remark 6.1 *From a computer science point of view, one can see a theory as a* specification *of an object: We describe the object thanks to first order logic, i.e. thanks to axioms that describe it.*

A consistent specification (theory) is hence nothing but a theory that specifies at least one object.

Remark 6.2 *In this context, the question of completeness is to know if one describes correctly the object in question, or the class of objects in question: The completeness theorem states that this is indeed the case for a consistent theory, as long as one want to talk about the whole class of all the models of these specifications.*

We are going to start by giving several example of theories, in order to make our discussion less abstract.

6.1 Examples of theories

6.1.1 Graphs

An *oriented graph* can be seen as a model of the theory without any axiom over the signature $\Sigma = (\emptyset, \emptyset, \{E\})$, where the relation symbol E is of arity 2: $E(x, y)$ means that there is an arc between x and y.

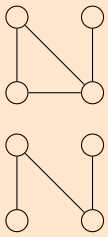
Example 6.1 *The formula $\exists y(E(x, y) \wedge \forall z(E(x, z) \Rightarrow x = y))$ is true in x if and only if y is of exterior degree 1 (modulo the comment of subsection that follows about equality).*

A non-oriented graph can be seen as a model of the theory with the unique axiom

$$\forall x \forall y \, (E(x, y) \Leftrightarrow E(y, x)), \tag{6.1}$$

on the same signature. This axioms states that if there is an arc between x and y, then there is an arc between y and x and conversely.

Example 6.2 *Here are two (non-oriented) graphs*



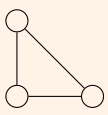
The formula $\exists x \forall y(\neg(x = y) \Rightarrow E(x, y))$ is true on the first and not on the second.

6.1.2 Simple remarks

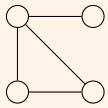
Remark 6.3 *On the signature $\Sigma = (\emptyset, \emptyset, \{E\})$, there is no term. We hence cannot designate any particular vertex but using some free variable, or via some quantifiers.*

If one wants to designate one or some particular vertex, we can add one or several constant symbols. We can hence for example consider the signature $(V, \emptyset, \{E\})$ where $V = \{a, b, c\}$.

For example, the graph



is a model of $E(a,b) \wedge E(b,c) \wedge E(a,c)$.
But be careful, this is not the only one: The graph



is indeed also a model: The domain of a model can contains some elements that are not corresponding to any term.

Furthermore, the interpretation of a, b *or* c *could be the same element.*

Example 6.3 *One can sometimes avoid constants. The formula*

$$\exists x \exists y \exists z (\neg(x=y) \wedge \neg(y=z) \wedge \neg(x=z) \wedge E(x,y) \wedge E(y,z) \wedge E(x,z) \wedge \forall t (t=x \vee t=y \vee t=z)) \tag{6.2}$$

characterizes the triangles such as the graph above (modulo the comment of the following subsection concerning equality).

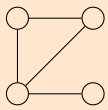
Remark 6.4 *Be careful: All the properties cannot be expressed easily. For example, one can prove that this is not possible to write a (first order) formula which characterizes the connected graphs. Exercise: Try to write it in order to feel where the problem is.*

Remark 6.5 *This is the presence of other models that the one that we intend to describe, and that is sometimes unavoidable, that would be at the heart of the difficulties about the axiomatisation of the integers.*

6.1.3 Equality

Be careful,the previous discussion is not totally correct: We have used at several times the equality symbol. The above discussion was supposing that the interpretation of equality is indeed equality.

Example 6.4 *Actually,*



is indeed a model of (6.2)*, and this is consequently perfectly false that* (6.2) *characterizes the triangles.*

Actually, let's call $\{a,b,c,d\}$ *the vertices from bottom to top and from left to right; we can consider the interpretation* $\equiv$ *of* $=$ *with* $a \equiv a, b \equiv b, c \equiv c, d \equiv b$

and $a \not\equiv b, a \not\equiv c, b \not\equiv c$. Such a model satisfies indeed (6.2)*. However, $\equiv$, the interpretation of $=$ is not the equality. Observe that we have an edge between a and b, $b = d$ that is true, but no edge between a and d.*

To make the above discussion fully correct, we can add a symbol = to the signature to all the examples, and add the axioms satisfied by equality.

Let $\mathscr{R}$ be a set of relation symbols that contains at least the symbol =.

Definition 6.3 (Axioms of equality) *The axioms of equality for a signature $\Sigma = (\mathscr{C}, \mathscr{F}, \mathscr{R})$, with $= \in \mathscr{R}$, are*

- *the axiom $\forall x\ x = x$;*
- *for every function symbol $f \in \mathscr{F}$ of arity n, the axiom*

  $$\forall x_1 \cdots \forall x_i \forall x'_i \cdots \forall x_n (x_i = x'_i \Rightarrow f(x_1, \cdots, x_i, \cdots, x_n) = f(x_1, \cdots, x'_i, \cdots, x_n));$$

- *for every relation symbol $R \in \mathscr{R}$ of arity n, the axiom*

  $$\forall x_1 \cdots \forall x_i \forall x'_i \cdots \forall x_n (x_i = x'_i \Rightarrow (R(x_1, \cdots, x_i, \cdots, x_n) \Rightarrow R(x_1, \cdots, x'_i, \cdots, x_n)).$$

All these axioms specify that the equality is reflexive, and is preserved by the relation and function symbols.

Exercise 6.1 *(solution on page 232) Prove that we then necessarily have $\forall x \forall y\ (x = y \Rightarrow y = x)$.*

Exercise 6.2 *Prove that we then necessarily have for each relation symbol $R \in \mathscr{R}$ or arity n,*

$$\forall x_1 \cdots \forall x_i \forall x'_i \cdots \forall x_n (x_i = x'_i \Rightarrow (R(x_1, \cdots, x_i, \cdots, x_n) \Leftrightarrow R(x_1, \cdots, x'_i, \cdots, x_n)).$$

Exercise 6.3 *Prove that we then necessarily have for each formula $F(x_1, x_2, \ldots, x_n)$*

$$\forall x_1 \cdots \forall x_i \forall x'_i \cdots \forall x_n (x_i = x'_i \Rightarrow (F(x_1, \cdots, x_i, \cdots, x_n) \Leftrightarrow F(x_1, \cdots, x'_i, \cdots, x_n)).$$

Exercise 6.4 *Prove that we then necessarily have $\forall x \forall y \forall z\ ((x = y \wedge y = z) \Rightarrow x = z)$.*

We deduce from the two previous exercises, that = (and its interpretation) is some equivalence relation.

6.1.4 Small digression

Definition 6.4 *A model $\mathfrak{M}$ of a theory $\mathcal{T}$ over a signature with the relation symbol $=$ is said to respect equality if the interpretation of $=$ in $\mathfrak{M}$ is equality.*

In other terms, the interpretation of symbol $=$ in $\mathfrak{M}$ is the subset $\{(x,x)|x \in M\}$ where M is the base set of $\mathfrak{M}$.

It turns out if this is not the case, and if the axioms of equality are among the theory $\mathcal{T}$, we can come back to this case.

Proposition 6.1 *Let $\mathcal{T}$ be a theory with a signature Σ, with at least the symbol $=$ as a relation symbol, which contains the axioms of equality for Σ.*

If $\mathcal{T}$ has a model, then $\mathcal{T}$ has also some model that respects equality.

Proof: We can quotient the domain M of any model $\mathfrak{M}$ of $\mathcal{T}$ by the equivalence relation that puts in the same equivalence class x and y when the interpretation of $x = y$ is true in $\mathfrak{M}$ (i.e. the interpretation of $=$). The quotient model,that is to say the model whose elements are the equivalence classes for this equivalence relation, is by definition, respecting equality. □

As a consequence, a theory $\mathcal{T}$ has a model that respects equality if and only if the theory plus all the axioms of equality (for the corresponding signature) has a model.

Example 6.5 *In the example 6.3, the sentence should be: The models that respects equality of the formula* (6.2) *characterize the triangles.*

Or possibly: The theory made of the formula (6.2) *and the axioms of equality (in that case $\forall x\ x = x, \forall x \forall x' \forall y (x = x' \Rightarrow (R(x,y) \Rightarrow R(x',y))), \forall x \forall y \forall y' (y = y' \Rightarrow (R(x,y) \Rightarrow R(x,y'))))$) characterize the triangles.*

6.1.5 Groups

Let's start by talking about groups, in group theory.

Example 6.6 (Group) *A group is a model of the theory made of the axioms of equality and of the two formulas:*

$$\forall x \forall y \forall z\ x * (y * z) = (x * y) * z \tag{6.3}$$

$$\exists e \forall x\ (x * e = e * x = x \wedge \exists y (x * y = y * x = e)) \tag{6.4}$$

on the signature $\Sigma = (\emptyset, \{*\}, \{=\})$, where $*$ and $=$ are of arity 2.

The first property asserts that the law $*$ of the group is associative, and the second that there is some neutral element, e, and that any element has some inverse.

Example 6.7 (Commutative group) *A commutative group (also called an* Abelian group*) is a model of the theory made of the axioms of equality and of the three*

formulas:

$$\forall x \forall y \forall z \; x * (y * z) = (x * y) * z \tag{6.5}$$
$$\exists e \forall x \; (x * e = e * x = x \wedge \exists y (x * y = y * x = e)) \tag{6.6}$$
$$\forall x \forall y \; x * y = y * x \tag{6.7}$$

over the same signature.

6.1.6 Fields

Example 6.8 (Commutative field) *A* commutative field *is a model of the theory made of the axioms of equality and of the formulas*

$$\forall x \forall y \forall z \; (x + (y + z) = (x + y) + z) \tag{6.8}$$
$$\forall x \forall y (x + y = y + x) \tag{6.9}$$
$$\forall x (x + \mathbf{0} = x) \tag{6.10}$$
$$\forall x \exists y (x + y = \mathbf{0}) \tag{6.11}$$
$$\forall x \forall y \forall z \; x * (y + z) = x * y + x * z \tag{6.12}$$
$$\forall x \forall y \forall z \; ((x * y) * z) = (x * (y * z)) \tag{6.13}$$
$$\forall x \forall y \; (x * y = y * x) \tag{6.14}$$
$$\forall x \; (x * \mathbf{1} = x) \tag{6.15}$$
$$\forall x \exists y (x = \mathbf{0} \vee x * y = \mathbf{1}) \tag{6.16}$$
$$\neg \mathbf{1} = \mathbf{0} \tag{6.17}$$

over a signature with two symbols of constants $\mathbf{0}$ *and* $\mathbf{1}$*, two symbols of functions* $+$ *and* $*$ *of arity* 2*, and the relation symbol* $=$ *of arity 2.*

For example, $\mathbb{R}$ and $\mathbb{C}$ with the usual interpretation are models of these theories.

If we add to the theory the formula F_p defined by $\mathbf{1} + \cdots + \mathbf{1} = \mathbf{0}$, where $\mathbf{1}$ is repeated p times, the models are the fields of characteristic p: For example, $\mathbb{Z}_p$, when p is some prime integer.

If we want to describe a field of characteristic 0, we must consider the theory made of the previous axioms, and the union of the negation of the axioms F_p for all prime integer p.

Example 6.9 (Algebraically closed field) *For every integer* n*, we consider the formula* G_n

$$\forall x_0 \forall x_1 \cdots \forall x_{n-1} \exists x (x_0 + x_1 * x + x_2 * x^2 + \cdots + x_{n-1} * x^{n-1} + x^n = 0)$$

where the reader would have guessed that x^k *is* $x * \cdots * x$ *with* x *repeated* k *times.*

An algebraically closed field is a model of the theory of commutative fields and of the union of the formulas G_n *for* $n \in \mathbb{N}$*.*

For example, $\mathbb{C}$ is algebraically closed. $\mathbb{R}$ is not algebraically closed, since x^2+1 has no real root.

6.1.7 Robinson Arithmetic

We can also try to axiomatise the integers. Here is a first attempt.

Example 6.10 (Robinson arithmetic) *Consider the signature made of the constant symbol* $\mathbf{0}$*, of the unary function symbol* s*, and of two binary function symbols* $+$ *and* $*$*, and of binary relation symbols* $<$ *and* $=$*.*

The axioms of Robinson arithmetic are the axioms of equality and

$$\forall x\ \neg s(x) = \mathbf{0} \tag{6.18}$$
$$\forall x\ \forall y\ (s(x) = s(y) \Rightarrow x = y) \tag{6.19}$$
$$\forall x\ (x = \mathbf{0} \vee \exists y\ \ s(y) = x) \tag{6.20}$$
$$\forall x\ \mathbf{0} + x = x \tag{6.21}$$
$$\forall x\ s(x) + y = s(x + y) \tag{6.22}$$
$$\forall x\ \mathbf{0} * x = \mathbf{0} \tag{6.23}$$
$$\forall x\ s(x) * y = x * y + y \tag{6.24}$$
$$\tag{6.25}$$

The structure whose base set is the integers, and where $+$ is interpreted by addition, $*$ by multiplication, and $s(x)$ by $x+1$ is a model of this theory. We call this model the *standard model of the integers.*

Observe that we can define in any model of the previous axioms some order, by the rule $x < y$ if and only if $\exists z\ (x + s(z) = y)$.

An alternative is to take $<$ as a primitive relation symbol of arity 2 and add the axioms

$$\forall x\ \neg x < \mathbf{0} \tag{6.26}$$
$$\forall x\ \mathbf{0} = x \vee \mathbf{0} < x \tag{6.27}$$
$$\forall x\ \forall y\ (x < y \Leftrightarrow (s(x) < y \vee s(x) = y)) \tag{6.28}$$
$$\forall x\ \forall y\ (x < s(y) \Leftrightarrow (x < y \vee x = y)) \tag{6.29}$$

Exercise 6.5 *Prove that the order defined by the rule* $x < y$ *if and only if* $\exists z\ (x + s(z) = y)$ *satisfies these formulas.*

Exercise 6.6 *(solution on page 232) Let n and m two integers. We write $s^n(\mathbf{0})$ for $s(s(\cdots s(\mathbf{0})))$ with s repeated n times, with the convention that $s^{(0)} = \mathbf{0}$.*

Prove by recurrence that

$$s^n(\mathbf{0}) + s^m(\mathbf{0}) = s^{n+m}(\mathbf{0}).$$

Find some model of Robinson axioms where two elements a and b are such that $a + b \neq b + a$.

Deduce that Robinson axioms are not sufficient to axiomatise the integers: There are other models that the standard model of the integers to these axioms.

Exercise 6.7 *Add $\forall x \forall y (x + y = y + x)$ to previous axioms to guarantee the commutativity of addition. Produce a model of the previous axioms that is not the standard model of the integers: For example, with tow elements a and b such that $a * b \neq b * a$.*

Instead of trying to add certain axioms in order to guarantee that properties such as commutativity of addition and of multiplication, we will consider a family of axioms.

6.1.8 Peano arithmetic

Example 6.11 (Peano arithmetic) *Consider a signature made of the constant symbol $\mathbf{0}$, for the unary function symbol s, and of two binary function symbols $+$ and $*$, and of binary relation symbol $=$.*

The axioms of Peano arithmetic are the axioms of equality and

$$\forall x \neg(s(x) = \mathbf{0}) \tag{6.30}$$
$$\forall x \forall y \ (s(x) = s(y) \Rightarrow x = y) \tag{6.31}$$
$$\forall x \ (x = \mathbf{0} \vee \exists y \ s(y) = x) \tag{6.32}$$
$$\forall x \ \mathbf{0} + x = x \tag{6.33}$$
$$\forall x \ s(x) + y = s(x + y) \tag{6.34}$$
$$\forall x \ \mathbf{0} * x = \mathbf{0} \tag{6.35}$$
$$\forall x \ s(x) * y = x * y + y \tag{6.36}$$
$$\tag{6.37}$$

and the set of all the formulas of the form

$$\forall x_1 \cdots \forall x_n ((F(\mathbf{0}, x_1, \cdots, x_n) \wedge \forall x_0 (F(x_0, x_1, \cdots, x_n) \Rightarrow F(s(x_0), x_1, \cdots, x_n)))$$

$$\Rightarrow \forall x_0 F(x_0, x_1, \cdots, x_n)) \tag{6.38}$$

where n in any integer, and $F(x_0, \cdots, x_n)$ is any formula of free variables $x_0, \cdots, x_n$.

There are hence an infinity of axioms. The last axioms aims at capturing reasoning's by recurrence that are usually done on the integers.

Of course, these axioms guarantee the following property: The standard model of the integers is model of these axioms

Exercise 6.8 *Prove that the axiom $\forall x\ (x = \mathbf{0} \vee \exists y\ s(y) = x)$ is actually useless: This formula is a consequence of the others.*

One clear interest is that we have now:

Exercise 6.9 *(solution on page 233) Prove that in any model of Peano axioms, the addition is commutative: The formula $\forall x \forall y (x + y = y + x)$ is true.*

Exercise 6.10 *Prove that in any model of Peano axioms, the multiplication is commutative: The formula $\forall x \forall y (x * y = y * x)$ is true.*

In other words, this family of axioms is sufficient to guarantee a huge number of properties that are true on the integers.

We will see later on (incompleteness theorem) that there remain some other models than the standard integers to Peano axioms.

6.2 Completeness

The *completeness theorem*, due to Kurt Gödel, sometimes called the *first Gödel theorem*, is relating the notion of completeness to the notion of provability, by demonstrating that the two notions are the same.

6.2.1 Consequences

The notion of consequence is easy to define.

Definition 6.5 (Consequence) *Let F be a formula. The formula F is said to be a (semantic)* consequence *of a theory $\mathcal{T}$ if any model of the theory $\mathcal{T}$ is a model of F. We write in this case $\mathcal{T} \models F$.*

Example 6.12 *For example, the formula $\forall x \forall y\ x * y = y * x$, which expresses the commutativity , is not a consequence of the theory of groups (Definition 6.6),*

since there are groups which are not commutative.

Example 6.13 *We can prove that the formula* $\forall x\ \mathbf{0} + x = x$ *is a consequence of Peano axioms.*

Example 6.14 *The exercise 6.6 proves that the formula* $\forall x \forall y (x+y = y+x)$ *(commutativity of addition) is not a consequence of Robinson axioms.*

6.2.2 Demonstration

We need to fix a notion of demonstration. We will do it, but let's first say that we have a notion of demonstration, such that we write $\mathscr{T} \vdash F$ if one can prove the closed formula F from the axioms of theory $\mathscr{T}$.

We expect at minimum from this notion of proof to be valid: That is to say to derive uniquely consequences: If F is a closed formula, and if $\mathscr{T} \vdash F$, then F is a consequence of $\mathscr{T}$.

6.2.3 Statement of completeness theorem

The completeness theorem states that actually we can succeed to reach all the consequences: The relations $\models$ and $\vdash$ are the same.

Theorem 6.1 (Completness theorem) *Let* $\mathscr{T}$ *be a theory over a denumerable signature. Let* F *be some closed formula.* F *is a consequence of* $\mathscr{T}$ *if and only if* F *is provable from* $\mathscr{T}$.

6.2.4 Meaning of the theorem

Let's take some time to understand what it does mean: In other words, the **provable statements are precisely those which are true in every model of the theory.**

This means in particular that:

- if some closed formula F is not provable, then there must exists a model that is not a model of F.
- if a closed formula F is true in any model of the axioms of the theory, then F is provable.

Example 6.15 *For example, the formula* $\forall x \forall y\ x * y = y * x$*, which expresses the commutativity, is not provable from the axioms of the theory of groups.*

Example 6.16 *The formula* $\forall x\ \mathbf{0} + x = x$ *is provable from the Peano axioms.*

6.2.5 Other formulation of the theorem

We say that a theory $\mathscr{T}$ is *coherent* if there is no formula F such that $\mathscr{T} \vdash F$ and $\mathscr{T} \vdash \neg F$.

We will see while doing the proof that the following also holds:

Theorem 6.2 (Théorème de complétude) *Let $\mathscr{T}$ be a theory over some denumerable signature. $\mathscr{T}$ has a model if and only if $\mathscr{T}$ is coherent.*

6.3 Proof of completeness theorem

6.3.1 A deduction system

We need to define a notion of demonstration. We choose to consider a notion of demonstration based on the notion of proof à la Frege and Hilbert, that is to say based on the modus ponens.

With respect to propositional calculus, we are not using anymore only the modus ponens rule, but also a *generalisation rule*: If F is a formula and if x is some variable, the generalisation rule deduces $\forall x F$ from F.

One ca be troubled by this rule, but this is nothing but what is regularly done in the common reasoning: If we succeed to prove $F(x)$ without any particular hypothesis on x, then we know that $\forall x F(x)$.

We then consider a certain number of axioms:

Definition 6.6 (Axiomes logiques du calcul des prédicats) *The* logical axioms of the predicate calculus *are:*

1. *every instance of the tautologies of propositional calculus;*
2. *the axioms of quantifiers, that is to say:*
   (a) *the formulas of the form* $(\exists x F \Leftrightarrow \neg\forall x \neg F)$*, where F is any formula and x is an arbitrary variable;*
   (b) *the formulas of the form* $(\forall x(F \Rightarrow G) \Rightarrow (F \Rightarrow \forall x G))$ *where F and G are arbitrary formulas and x is a variable that has no free occurrence in F;*
   (c) *the formulas of the form* $(\forall x F \Rightarrow F(t/x))$ *where F is a formula, t is a term and no free occurrence of x in F is covered by some quantifier bounding a variable of t, where $F(t/x)$ denotes the substitution of x by t.*

Exercise 6.11 *Prove that the logical axioms are valid.*

Remark 6.6 *We could not have put all the tautologies of propositional calculus, and, as we did for propositional calculus, restrict to certain axioms, essentially the axioms of Boolean logic. We does so here only to make the proofs simpler, but this could be possible and it would still work.*

We obtain the notion of demonstration.

Definition 6.7 (Demonstration by modus ponens and generalisation) *Let $\mathcal{T}$ be at theory and let F be some formula. A* proof of F from $\mathcal{T}$ *is a finite sequence $F_1, F_2, \cdots, F_n$ of formulas such that F_n is equal to F, and for all i, either F_i is in $\mathcal{T}$, or F_i is some logical axiom, or F_i is obtained by modus ponens from two formulas F_j, F_k with $j < i$ and $k < i$, or F_i is obtained by generalisation from a formula F_j with $j < i$.*

We write $\mathcal{T} \vdash F$ if F is provable from $\mathcal{T}$.

6.3.2 Finiteness theorem

We obtain first easily through the proof the finiteness theorem.

Theorem 6.3 (Finiteness theorem) *For every theory $\mathcal{T}$, and for any formula F, if $\mathcal{T} \vdash F$, then there exists a finite subset $\mathcal{T}_0$ of $\mathcal{T}$ such that $\mathcal{T}_0 \vdash F$.*

Proof: A demonstration is a finite sequence of formulas F_1, F_2, $\cdots, F_n$. Consequently, it is using only a finite number of formulas, hence a finite subset $\mathcal{T}_0$ of formulas of $\mathcal{T}$. This demonstration is also a demonstration of F in the theory $\mathcal{T}_0$. □

Corollary 6.1 *If $\mathcal{T}$ is a theory such that all finite subsets are coherent, then $\mathcal{T}$ is coherent.*

Proof: Otherwise $\mathcal{T}$ proves $(F \wedge \neg F)$, for some formula F, and by the finiteness formula, we deduce that there exists a finite subset $\mathcal{T}_0$ of $\mathcal{T}$ that also proves $(F \wedge \neg F)$. □

6.3.3 Some technical results

We need the following results, whose proofs are coming from a game on writing and rewriting on the demonstrations.

First of all an observation, but that has its importance:

Lemma 6.1 *If a theory $\mathcal{T}$ is not coherent, then any formula is provable in $\mathcal{T}$.*

Proof: Indeed, suppose that $\mathcal{T} \vdash F$ and that $\mathcal{T} \vdash \neg F$, and let G be some arbitrary formula. One can then put one after the other a demonstration of F and a demonstration of $\neg F$. To obtain a demonstration of G, it suffices to add the following formulas to this sequence: The tautology $F \Rightarrow (\neg F \Rightarrow G)$. The formula $\neg F \Rightarrow G$ which can then be obtained by modus ponens, since F has already appeared. Then the formula G, which can be obtained by modus ponens, since $\neg F$ has already appeared. □

Lemma 6.2 (Deduction lemma) *Suppose that $\mathcal{T} \cup \{F\} \vdash G$, with F some closed formula. Then $\mathcal{T} \vdash (F \Rightarrow G)$.*

Proof: From a demonstration $G_0 G_1 \cdots G_n$ of G in $\mathcal{T} \cup \{F\}$, we construct a demonstration of $(F \Rightarrow G)$ in $\mathcal{T}$ by inserting in the sequence $(F \Rightarrow G_0)(F \Rightarrow G_1) \cdots (F \Rightarrow G_n)$.

If G_i is a tautology, then there is nothing to do, since $(F \Rightarrow G_i)$ is also a tautology.

If G_i is F, then there is nothing to do, since $(F \Rightarrow G_i)$ is a tautology.

If G_i is an axiom of quantifiers or an element of $\mathcal{T}$, then it suffices to insert [1] between $(F \Rightarrow G_{i-1})$ and $(F \Rightarrow G_i)$ the formulas G_i and $(G_i \Rightarrow (F \Rightarrow G_i))$ (which is a tautology).

Suppose now that G_i is obtained by modus ponens: There are some integers $j, k < i$ such that G_k is $(G_j \Rightarrow G_i)$. We insert then between $(F \Rightarrow G_{i-1})$ and $(F \Rightarrow G_i)$ the formulas;

1. $((F \Rightarrow G_j) \Rightarrow ((F \Rightarrow (G_j \Rightarrow G_i)) \Rightarrow (F \Rightarrow G_i)))$ (a tautology);

2. $(F \Rightarrow (G_j \Rightarrow G_i)) \Rightarrow (F \Rightarrow G_i)$ that is obtained from modus ponens from the previous and thanks to $(F \Rightarrow G_j)$ which has already appeared;

3. $(F \Rightarrow G_i)$ is then deduced by modus ponens from this last formula and from $(F \Rightarrow (G_j \Rightarrow G_i))$, that has already appeared since it is $(F \Rightarrow G_k)$.

Suppose at last that G_i is obtained by generalisation from G_j with $j < i$. We insert in this case between $(F \Rightarrow G_{i-1})$ and $(F \Rightarrow G_i)$ the formulas:

1. $\forall x (F \Rightarrow G_j)$ obtained by generalisation starting from $(F \Rightarrow G_j)$;

2. $(\forall x (F \Rightarrow G_j) \Rightarrow (F \Rightarrow \forall x G_j))$ (a quantifier axiom). F being a closed formula, x is not free;

3. $(F \Rightarrow G_i)$ is then deduced by modus ponens from the two previous.

□

The corollary that follows can be seen as the justification of reasoning by contradictions.

Corollary 6.2 *$\mathcal{T} \vdash F$ if and only if $\mathcal{T} \cup \{\neg F\}$ is not coherent.*

[1] For $i = 0$, it suffices to position this formula at the beginning.

Proof: It is clear that if $\mathcal{T} \vdash F$ then $\mathcal{T} \cup \{\neg F\}$ is not coherent. Conversely, if $\mathcal{T} \cup \{\neg F\}$ is not coherent, it proves any formula, and in particular F by Lemma 6.1. Now, by the deduction lemma above, we obtain that $\mathcal{T} \vdash \neg F \Rightarrow F$. Now, $(\neg F \Rightarrow F) \Rightarrow F$ is a tautology, which proves that we have $\mathcal{T} \vdash F$. □

Lemma 6.3 *Let $\mathcal{T}$ be at theory, and let $F(x)$ be a formula whose only free variable is x. Let c be some constant symbol that is not appearing in F nor in $\mathcal{T}$. If $\mathcal{T} \vdash F(c/x)$ then $\mathcal{T} \vdash \forall x F(x)$.*

Proof: Consider a demonstration $F_1 F_2 \cdots F_n$ of $F(c/x)$ in $\mathcal{T}$. We consider a variable w that is in none of the formulas F_i and we call K_i the formula obtained by replacing in F_i the symbol c by w.

It turns out that this provides a proof of $F(w/x)$: If F_i is some logical axiom, then so does K_i; If F_i is deduced by modus ponens, and if $F_i \in \mathcal{T}$ then K_i is F_i.

By generalisation, we hence obtain a proof of $\forall w F(w/x)$, and by the remark that follows, we can then obtain a proof of $\forall x F(x)$. □

Remark 6.7 *If w is a variable that has no occurrence in F (nor free, nor bound), then we can prove $\forall w F(w/x) \Rightarrow \forall x F$: Indeed, since w has no occurrence in F, we can then prove $\forall w F(w/x) \Rightarrow F$, (axiom (c) of quantifiers, observing that $(F(w/x))(x/w) = F$ with these hypotheses). By generalisation, we obtain $\forall x(\forall w F(w/x) \Rightarrow F)$, and since x is not free in $\forall w F(w/x)$, the formula $\forall x(\forall w F(w/x) \Rightarrow F) \Rightarrow (\forall w F(w/x) \Rightarrow \forall x F)$ is among the axioms (b) of quantifiers, which allow to obtain $\forall w F(w/x) \Rightarrow \forall x F$ by modus ponens.*

6.3.4 Validity of the deduction system

The validity of the proof method is easy to obtain.

Theorem 6.4 (Validity) *Let $\mathcal{T}$ be a theory. Let F be some formula.*
If $\mathcal{T} \vdash F$, then any model of $\mathcal{T}$ is a model of the universal closure of F.

Proof: It suffices to check that the logical axioms are valid, and that modus ponens and generalisation can only infer some valid facts in any model of $\mathcal{T}$. □

This is the easy direction of the completeness theorem.

6.3.5 Completeness of the deduction system

The other direction consists in proving that if F is a consequence of $\mathcal{T}$, then F can be proved by our proof method.

Definition 6.8

We say that a theory $\mathcal{T}$ is complete *if for any closed formula F, we have $\mathcal{T} \vdash F$ or $\mathcal{T} \vdash \neg F$.*

We say that a theory $\mathcal{T}$ admits some Henkin witnesses *if for any formula $F(x)$ with some free variable x, there exists some constant symbol c in the signature*

such that $(\exists x F(x) \Rightarrow F(c))$ *is a formula of the theory* $\mathcal{T}$.

The proof of completeness theorem due to *Henkin* that we will present runs in two steps.

1. We prove that any coherent theory, complete, with Henkin witnesses admits a model.

2. We prove that any consistent theory admits some with these three properties.

Lemma 6.4 *If* $\mathcal{T}$ *is some coherent, complete, with Henkin witnesses, then* $\mathcal{T}$ *has a model.*

Proof: The trick is to construct from scratch a model, whose base set (domain) is the set M of closed terms on the signature of the theory: This domain is non-empty, since the signature has at least the constants.

The structure $\mathfrak{M}$ is defined in the following way:

1. If c is a constant, the interpretation $c^{\mathfrak{M}}$ of c is the constant c itself.

2. If f is a function symbol of arity n, its interpretation $f^{\mathfrak{M}}$ is the function that to closed terms $t_1, \cdots, t_n$ associate the closed term $f(t_1, \cdots, t_n)$.

3. If R is a relation symbol of arity n, its interpretation $R^{\mathfrak{M}}$ is the subset of M^n made of the $(t_1, \cdots, t_n)$ such that $\mathcal{T} \vdash R(t_1, \cdots, t_n)$.

We observe that the structure that is obtained satisfies the following property For any closed formula F, $\mathcal{T} \vdash F$ if and only if $\mathfrak{M}$ is a model of F. This is proved by structural induction on F.

The property is true for the atomic formulas.

Because of the properties of the quantifiers and connectors, and because of the possibility of using occurrences of tautologies of propositional calculus in our proof method, it suffices to get convinced of this fact inductively on the formulas of type $\neg G$, $(G \vee H)$ and $\forall x G$.

1. Case $\neg G$: Since $\mathcal{T}$ is complete, $\mathcal{T} \vdash \neg G$ if and only if $\mathcal{T} \nvdash G$, which means inductively $\mathfrak{M} \not\models G$, or if one prefers $\mathfrak{M} \models \neg G$.

2. Case $(G \vee H)$: Suppose $\mathfrak{M} \models (G \vee H)$, and so $\mathfrak{M} \models G$ or $\mathfrak{M} \models H$. In the first case for example, by induction hypothesis, we have $\mathcal{T} \vdash G$, and since $(G \Rightarrow (G \vee H))$ is a tautology, we have $\mathcal{T} \vdash (G \vee H)$. Conversely, suppose that $\mathcal{T} \vdash (G \vee H)$. Si $\mathcal{T} \vdash G$ then by the induction hypothesis $\mathfrak{M} \models G$ and so $\mathfrak{M} \models (G \vee H)$. Otherwise, this is because $\mathcal{T} \nvdash G$, and since the theory is complete, we have $\mathcal{T} \vdash \neg G$. But since $(G \vee H \Rightarrow (\neg G \Rightarrow H))$ is a tautology, we obtain that $\mathcal{T} \vdash H$ and by the induction hypothesis, $\mathfrak{M} \models H$ and so $\mathfrak{M} \models (G \vee H)$.

3. Case $\exists x G(x)$: If $\mathfrak{M} \models \exists x G(x)$ this is because there is some closed term t such that $\mathfrak{M} \models G(t/x)$. By induction hypothesis, $\mathcal{T} \vdash G(t/x)$. But it is easy to find

a demonstration of $\exists x G(x)$ from a demonstration of $G(t/x)$. Conversely, suppose that $\mathcal{T} \vdash \exists x G(x)$. Thanks to Henkin witnesses, we deduce that there exists some constant c such that $\mathcal{T} \vdash G(c/x)$, and by induction hypothesis $\mathfrak{M} \models G(c/x)$, and so $\mathfrak{M} \models \exists x G(x)$.

□

There remains the second step. An *extension of at theory* $\mathcal{T}$ is a theory $\mathcal{T}'$ that contains $\mathcal{T}$.

Proposition 6.2 *Every coherent theory $\mathcal{T}$ on a countable signature Σ has some extension $\mathcal{T}'$ on a denumerable signature Σ' (with Σ' that contains Σ) that is coherent, complete and with Henkin witnesses.*

Before proving this property, let us discuss what we are obtaining: Since a model of $\mathcal{T}'$ is a model of $\mathcal{T}$, the previous lemma and the previous proposition permit first to obtain:

Corollary 6.3 *A denumerable coherent theory has a model.*

The following remark is obtained by playing with definitions:

Proposition 6.3 *For every theory $\mathcal{T}$ and for every closed formula F, F is a consequence of T if and only if $\mathcal{T} \cup \{\neg F\}$ has no model.*

Proof: If F is a consequence of $\mathcal{T}$, then by definition every model of $\mathcal{T}$ is a model of F, in other words, there is no model of $\mathcal{T} \cup \{\neg F\}$. The converse is trivial. □

We obtain with this remark exactly the completeness theorem (or the missing direction of what we called the completeness theorem).

Theorem 6.5 *Let F be some closed formula. If F is a consequence of the theory $\mathcal{T}$, then $\mathcal{T} \vdash F$.*

Proof: If $\mathcal{T}$ does not prove F, then $\mathcal{T} \cup \{\neg F\}$ is coherent: By the previous corollary, $\mathcal{T} \cup \{\neg F\}$ has a model. This means that F is not a consequence of the theory $\mathcal{T}$. □

There remains to prove Proposition 6.2.

Proof: The signature Σ' is obtained by adding some denumerable number of new constants to the signature Σ. The obtained signature Σ' remains denumerable and we can enumerate the closed formulas $(F_n)_{n\in\mathbb{N}}$ of Σ'. The theory $\mathcal{T}'$ is obtained as the union of a increasing sequence of theories $\mathcal{T}_n$, defined by recurrence, starting from $\mathcal{T}_0 = \mathcal{T}$. Suppose that $\mathcal{T}_n$ is constructed and coherent. To construct $\mathcal{T}_{n+1}$ we consider the formula F_{n+1} in the enumeration of the closed formulas of Σ'. If $\mathcal{T}_n \cup F_{n+1}$ is coherent, then we let $G_n = F_{n+1}$, otherwise we let $G_n = \neg F_{n+1}$. In the two cases $\mathcal{T}_n \cup \{G_n\}$ is coherent.

The theory $\mathcal{T}_{n+1}$ is defined by:

1. $\mathcal{T}_{n+1} = \mathcal{T}_n \cup \{G_n\}$ if G_n is not of the form $\exists x H$.

2. otherwise: $\mathcal{T}_{n+1} = \mathcal{T}_n \cup \{G_n, H(c/x)\}$ where c is a new constant symbol that is not appearing in any formula of $T_n \cup \{G_n\}$: There is always such a symbol, there there is a finite number of symbols in $T_n \cup \{G_n\}$.

The theory $\mathcal{T}_{n+1}$ is coherent: Indeed, if it were not, this would mean that G_n would be of the form $\exists x H$, and that $T_n \cup \{\exists x H\} \vdash \neg H(c/x)$. By the choice of the constant c, and by Lemma 6.3, we obtain that $T_n \cup \{\exists x H\} \vdash \forall x \neg H(x)$, which is impossible since otherwise $\mathcal{T}_n$ would not be coherent.

The theory $\mathcal{T}' = \bigcup_{n\in\mathbb{N}} \mathcal{T}_n$ defined as the union of the theories $\mathcal{T}_n$ is coherent since any finite subset of it is contained in one of the theories $\mathcal{T}_n$, and hence is coherent.

The theory $\mathcal{T}'$ is also complete: If F is some closed formula of Σ', it appears at some moment in the enumeration of the formulas F_n, and by construction, either $F_n \in \mathcal{T}_n$ or $\neg F_n \in \mathcal{T}_n$.

Finally the theory $\mathcal{T}'$ has some Henkin witnesses: If $H(x)$ is a formula with the free variable x, then the formula $\exists x H$ appears as a formula in the enumeration of the formulas F_n. There are then two cases: either $\neg F_n \in \mathcal{T}_{n+1}$ or there is some constant c such that $H(c/x) \in \mathcal{T}_{n+1}$. In the two cases, $\mathcal{T}_{n+1} \vdash \exists x H(x) \Rightarrow H(c/x)$, which proves that $(\exists x H(x) \Rightarrow H(c/x))$ is in $\mathcal{T}'$ (otherwise its negation would be there, and $\mathcal{T}'$ would not be coherent). □

6.4 Compactness

Observe that we have also established some other facts.

Theorem 6.6 (Compactness theorem) *Let $\mathcal{T}$ a theory on some denumerable signature such that any finite subset of $\mathcal{T}$ has a model. Then $\mathcal{T}$ has a model.*

Proof: Consider a finite subset of such a theory $\mathcal{T}$. This subset is coherent since it has a model. $\mathcal{T}$ is hence a theory such that any finite subset is coherent. By finiteness theorem, this means that the theory itself is coherent.

By Corollary 6.3, this means that $\mathcal{T}$ has a model. □

Exercise 6.12 *(solution on page 233) Use compactness theorem to prove that there exists some non-standard model of Peano axioms.*

6.5 Other consequences

Theorem 6.7 (Löwenheim-Skolem) *If $\mathcal{T}$ is a theory on some denumerable signature that has a model, then it has a model whose base set is denumerable.*

Exercise 6.13 *(solution on page 233) Prove the theorem.*

6.6 Bibliographic notes

Suggested readings To go further on the notions of this chapter, we suggest to read [Cori & Lascar, 1993a], [Dowek, 2008] or [Lassaigne & de Rougemont, 2004].

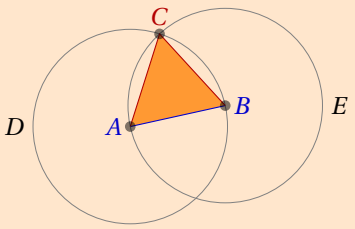
Bibliography This chapter has been written by essentially using the books [Cori & Lascar, 1993a] and [Lassaigne & de Rougemont, 2004].

Chapter 7

Turing machines

We have used many times up to know the notion of algorithm but without having provided a formal definition. Intuitively, one can say that an algorithm is an automatic method, that can be implemented on some computer, to solve a given problem. For example, the familiar techniques to perform an addition, or multiplication or a division on numbers learned at elementary school are algorithms. The techniques discussed to evaluate the truth value of a propositional formula from the value of its variables are also algorithms. More generally, we have discussed some proof methods for propositional calculus or for predicate calculus that can be seen as algorithms.

Example 7.1 *The following example taken from the manual of package* TikZ-PGF *version 2.0, inspired in turn from the* Elements *of* Euclid*, can be considered as an algorithm. It provides a method to draw an equilateral triangle with edge AB.*



***Algorithm to construct a equilateral triangle with edge** AB: draw a circle of center A and radius AB; draw the circle of center B of radius AB. Name C one of the intersection of the two circles. The triangle ABC is the desired solution.*

We will see in the following chapters that not all the problems can be solved by algorithms, and even for problems very easy to formulate: For example,

- there is no algorithm to determine if a given closed formula of predicate calculus is valid in the general case;

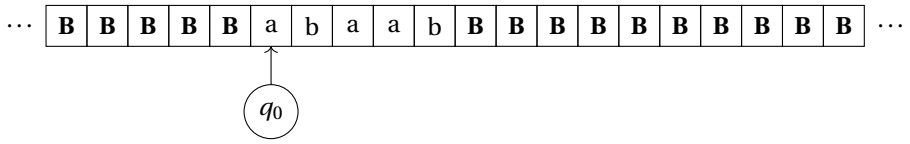


Figure 7.1: Turing machine. The machine is on the initial state of a computation on word $abaab$.

- there is no algorithm to determine if a multivariate polynomial (i.e. with several variables) with integer coefficients has an integer root (*Hilbert* $10th$ *problem*).

Historically, it was actually the formalization of the notion of proofs in mathematics and the question of limits of formal proof systems that led to the models that we will discuss. It was later understood that the notion captured by these models was more than simply the formalization of the notion of proof and that it was actually also a formalization of everything that can be computed by a digital machine. This remains true today since today's computers are digital.

Furthermore, several formalizations have been proposed, in an independent way, using at first sight very different notions: In particular, Alonzo Church in 1936, using the formalism of λ*-calculus*,, Alan Turing in 1936, using what is now called the *Turing machines*, or Emil Post in 1936, using systems based on very simple rules, called *Post systems*. Later, it was shown that these formalisms are all equivalent.

The objective of this chapter is to describe the Turing machine model. In the next chapter, we will define a few other models of computation and will show that they are equivalent to the Turing machine model.

We will then by talk about the *Church-Turing thesis*.

The models that we are going to describe can all be considered as very abstract, and might at first sight give the impression to be very limited, and far from being able to cover everything that can be programmed using today's languages such as Python, CAML or JAVA. The main objective of this chapter and of the next one is to convince the reader that this is NOT the case: Everything that can be programmed can actually be programmed using these "basic" models.

7.1 Turing machines

7.1.1 Ingredients

A (deterministic) Turing machine (See Figure 7.1) is composed of the following elements:

1. An infinite memory of the form of a *tape*. The tape is divided in cells. Each cell can contain an element of a set Σ (i.e. of some alphabet). We assume that the alphabet Σ is some finite set.

2. a (reading/writing) head that can move along the tape.
3. A program given as a *transition function* that, for every internal state q of the machine, among a finite number possible internal states Q, gives according to the symbol under the reading head:
   (a) the next internal state $q' \in Q$;
   (b) the new element of Σ to write in place of the element of M currently in front of the head;
   (c) a moving direction for the reading head.

The execution of a Turing machine on some word $w \in \Sigma^*$ can then be described as follows: initially, the input w is on the tape, and the head is positioned in front of the first letter of the word. The cells not corresponding to the input all contain the element **B** (blank symbol), that is a special letter. The machine is in its initial internal state q_0: See Figure 7.1.

At every execution step, the machine, reads the symbol in front of the head, and, according to its internal state and this symbol, following its program, it:

- replaces the symbol in front of the head by the one given by the transition function;
- (possibly) moves its head to the left or to the right, according to the direction given by the transition function;
- changes the internal state to the internal state given by the transition function.

The word w is said to be accepted when the execution of the Turing machine eventually reaches the accepting internal state, in which case the execution of the machine stops.

In the next section, we will give a formal definition of Turing machines and their execution.

7.1.2 Description

Definition 7.1 (Turing machine) A Turing machine *is an 8-uple*

$$M = (Q, \Sigma, \Gamma, \mathbf{B}, \delta, q_0, q_a, q_r)$$

where:

1. Q *is the finite set of internal states;*
2. Σ *is a* finite *alphabet;*
3. Γ *is the* finite *tape alphabet with* $\Sigma \subset \Gamma$*;*
4. $\mathbf{B} \in \Gamma$ *is the blank symbol;*
5. $q_0 \in Q$ *is the initial state;*

6. *$q_a \in Q$ is the accepting state;*
7. *$q_r \in Q$ is the rejecting state;*
8. *δ is the transition function: δ is a function (possibly partial) from $Q \times \Gamma$ to $Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}$. The symbol $\leftarrow$ is used to mean a move to the left, $|$ no move, $\rightarrow$ a move to the right.*

The language accepted by a Turing machine is defined using the notions of *configurations* and of successor relation between configurations of a Turing machine. A configuration consists of all the information required to describe the state of the machine at a given moment, and to determine the future states of the machine, namely:

- the internal state;
- the content of the tape;
- the position of the head.

We give a more formal definition.

Definition 7.2 (Configuration) A configuration *is given by the description of the tape, the position of the head, and the internal state.*

To write a configuration, one difficulty is that the tape is infinite, and thus consists of an infinite sequence of symbols of the tape alphabet Γ of the machine. However, we focus on finite executions, and consequently, at any moment of an execution, only a finite part of the tape has been visited by the machine. Indeed, initially the tape contains an input of finite length, and at every step the machine moves its head at most of one cell. As a consequence, after t steps, the head has moved at most t cells to the left or to the right from its initial position. Consequently, the content of the tape can be defined at any moment by a fixed sequence of symbols, the rest containing only the blank symbol **B**.

To denote the position of the head, we could use some integer $n \in \mathbb{Z}$. We will actually use the following trick that has the advantage of simplifying the coming definitions: Instead of seeing the tape as a finite sequence, we will represent it by two finite sequences: The content of what is on the right and what is on the left of the head. We will write the right prefix as usual from left to right. In contrast, we will write the prefix corresponding to what is on the left of the head from right to left: The interest is that the first letter of the left prefix is the letter immediately at the left of the head. A *configuration* will hence be an element of $Q \times \Gamma^* \times \Gamma^*$.

Formally:

Definition 7.3 (Denoting a configuration) *A configuration is denoted by $C = (q, u, v)$, with $u, v \in \Gamma^*$, $q \in Q$: u and v respectively denote the content of the tape respectively to the left and to the right of the head which is in front of the first letter of v. We assume that the letters of u and of v are not containing the blank symbol* **B**.

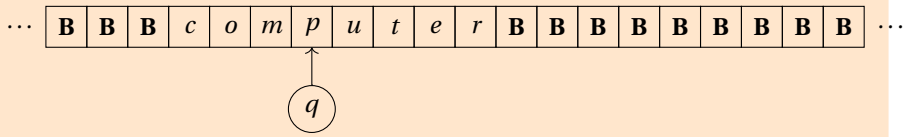
We make the convention that the word v is written from left to right (letter number $i+1$ of v corresponds to the cell at the right of the letter/cell number i) while the word u is written from right to left (the letter number $i+1$ of u corresponds to the content of tape at the left of letter/cell number i, the first letter of u being the cell immediately at the left of the head).

Example 7.2 *The configuration of the machine represented on Figure 7.1 is* $(q_0, \epsilon, abaab)$.

We will sometimes write configurations in an other way.

Definition 7.4 (Alternative notation) *The configuration (q,u,v) will also been seen/denoted in some sections or chapters by $u\boldsymbol{q}v$, keeping u and v written from left to right.*

Example 7.3 *A configuration such as*



is encoded by configuration $(q, moc, puter)$, or sometimes by $com\boldsymbol{q}puter$.

A configuration is said to be *accepting* if $q = q_a$, *rejecting* if $q = q_r$.

For $w \in \Sigma^*$, the initial configuration corresponding to w is the configuration $C[w] = (q_0, \epsilon, w)$.

We write: $C \vdash C'$ if the configuration C' is the immediate successor of configuration C by the program (given by δ) of the Turing machine.

Formally, if $C = (q,u,v)$ and if a denotes the first letter [1] of v, and if $\delta(q,a) = (q',a',m')$ then $C \vdash C'$ if $C' = (q',u',v')$, and

- if $m' = |$, then $u' = u$, and v' is obtained by replacing the first letter a of v by a';
- if $m' = \leftarrow$, u' is obtained by deleting the first letter a'' of u, v' is obtained by concatenating a'' and the result of replacing the first letter a of v by a';
- if $m' = \rightarrow$, $u' = a'u$, and v' is obtained by deleting the first letter a of v.

[1] With the convention that the first letter of the empty word is the blank symbol **B**.

Remark 7.1 *The above rules are formalizing the changing of the cell in front of the head from a to a' and the corresponding potential shift of the tape to the right or to the left.*

Definition 7.5 (Accepted word) *A word $w \in \Sigma^*$ is said to be* accepted *(in time t) by the Turing machine, if there exists a sequence of configurations $C_1, \cdots, C_t$ with:*

1. *$C_0 = C[w]$;*
2. *$C_i \vdash C_{i+1}$ for all $i < t$;*
3. *none of the configurations C_i for $i < t$ is accepting or rejecting*
4. *C_t is accepting.*

Definition 7.6 (Rejected word) *A word $w \in \Sigma^*$ is said to be* rejected *(in time t) by the Turing machine, if there exists a sequence of configurations $C_1, \cdots, C_t$ with:*

1. *$C_0 = C[w]$;*
2. *$C_i \vdash C_{i+1}$ for all $i < t$;*
3. *none of the configurations C_i for $i < t$ is accepting or rejecting.*
4. *C_t is rejecting.*

Definition 7.7 (Machine that loops on a word) *We say that a Turing machine* loops *on a word w, if w is neither accepted nor rejected.*

Remark 7.2 *For every word w we thus have exactly one of the following three exclusive cases:*

1. *it is accepted by the Turing machine;*
2. *it is rejected by the Turing machine;*
3. *the machine loops on this word.*

Remark 7.3 *The terminology* loops *means simply that the machine is not halting on this word: This does not necessarily mean that one repeats for ever the same instructions. The machine can loop for several reasons. For example, since it reaches a configuration that has no successor configuration that is defined,*

or because it enters a complex behaviours that produces an infinite sequence of configurations not accepting nor rejecting.

More generally, one calls *computation of M on a word $w \in \Sigma^*$*, a (finite or infinite) sequence of configurations $(C_i)_{i\in\mathbb{N}}$ such that $C_0 = C[w]$ and for all i, $C_i \vdash C_{i+1}$, with the convention that an accepting or rejecting configuration has no successor.

Definition 7.8 (Language accepted by a machine) *The language $L \subset \Sigma^*$* accepted by Turing machine *M is the set of words w that are accepted by the machine. It is denoted by L(M). We also call $L(M)$ as the* language recognized *by M.*

A machine that does not halt is usually undesirable. Thus, we generally try to guarantee a stronger property:

Definition 7.9 (Langage decided by a machine) *One says that a language $L \subset \Sigma^*$ is* decided *by the machine M if:*

- *for every $w \in L$, w is accepted by M;*
- *for every $w \notin L$ (=otherwise), w is rejected by M.*

In other words, the machine accepts L and terminates on every input (i.e. it never loops).

One says in that case that the machine *M decides L.*

7.1.3 Programming with Turing machines

Programming with Turing machines is extremely low level. We will however see that one can really program many things with this model. The first step is to get convinced that we can program solutions to many problems with Turing machines. To say the truth, the only way to get convinced is to try to program by oneself with Turing machines, for example by trying to solve the following exercises.

Exercise 7.1 *Construct a Turing machine that accepts exactly the words w on alphabet $\Sigma = \{0,1\}$ of the form $0^n 1^n$, $n \in \mathbb{N}$.*

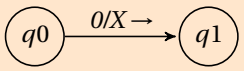
Here is a solution. Consider a machine $Q = \{q0, q1, q2, q3, q4\}$, $\Gamma = \{0, 1, X, Y, B\}$, the accepting state $q4$ and the transition function δ such that:

- $\delta(q0, 0) = (q1, X, \rightarrow)$;
- $\delta(q0, Y) = (q3, Y, \rightarrow)$;
- $\delta(q1, 0) = (q1, 0, \rightarrow)$;
- $\delta(q1, 1) = (q2, Y, \leftarrow)$;
- $\delta(q1, Y) = (q1, Y, \rightarrow)$;

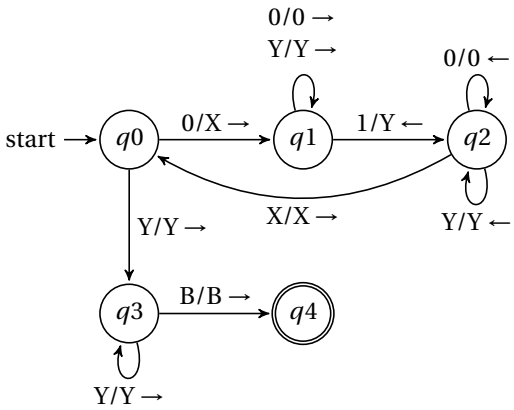
- $\delta(q2,0) = (q2,0,\leftarrow)$;
- $\delta(q2,X) = (q0,X,\rightarrow)$;
- $\delta(q2,Y) = (q2,Y,\leftarrow)$;
- $\delta(q3,Y) = (q3,Y,\rightarrow)$;
- $\delta(q3,B) = (q4,B,\rightarrow)$.

As one can see, a description of a Turing machine in this way is essentially unreadable. We thus prefer a representation of the program of a machine (the function δ) in the form of a graph: The vertices of the graph represent the states of the machine. Every transition $\delta(q,a) = (q',a',m)$ is represented with an arc from the state q to the state q' labeled by a/a' m. The initial state is marked with an incoming arc. The accepting state is marked with a double circle.

Example 7.4 *For example, the transition $\delta(q0,0) = (q1,X,\rightarrow)$ is represented graphically by:*



With these conventions, the previous program can be represented by:



How does this program work? During a computation, the part of the tape that the machine has visited will be of the form $X^*0^*Y^*1^*$. Every time that a 0 is read, it is replaced by X and one goes to state $q1$ which starts the following sub-procedure: One moves right as long as a 00 or a Y is read. As soon as a 1 is reached, it is transformed into a Y, and one goes back to the left until one reaches a X (the X that has been written) and then one does a one cell right shift.

Doing so, for each 0 that is erased (i.e. marked by a X), we will have erased a 1 (i.e. marked a Y). If all the 0 are marked and one reaches a Y, one goes to state $q3$, which has the effect of checking that what is on the right is indeed only consists of

Y's. Once everything has been read, i.e. a **B** is reached, one accepts, i.e. one goes to state $q4$.

Of course, a true proof of the correctness of this algorithm would consist in proving that if a word is accepted, it is necessarily of type 0^n1^n. We leave to the reader to get convinced of this.

Example 7.5 *Here is an example of accepting computation for M:* $q_0 0011 \vdash Xq_1011 \vdash X0q_111 \vdash Xq_20Y1 \vdash q_2X0Y1 \vdash Xq_00Y1 \vdash XXq_1Y1 \vdash XXYq_11 \vdash XXq_2YY \vdash Xq_2XYY \vdash XXq_0YY \vdash XXYq_3Y \vdash XXYYq_3\mathbf{B} \vdash XXYY\mathbf{B}q_4\mathbf{B}$.

Definition 7.10 (Space-time diagram) *One often represents a sequence of configurations line by line: The line number i represents the ith configuration of the computation, using the encoding of Definition 7.4. This representation is called a* space-time diagram *of a machine.*

Example 7.6 *Here is the space-time diagram corresponding to the previous computation on* 0011.

$\cdots$	**B**	**B**	**B**	**B**	q_0	0	0	1	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	q_1	0	1	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	0	q_1	1	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	q_2	0	Y	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	q_2	X	0	Y	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	q_0	0	Y	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	q_1	Y	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	Y	q_1	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	q_2	Y	Y	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	q_2	X	Y	Y	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	q_0	Y	Y	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	Y	q_3	Y	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	Y	Y	q_3	B	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	Y	Y	B	q_4	B	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$

Example 7.7 *Here is the space-time diagram of the computation of the machine on* 0010*:*

⋯	**B**	**B**	**B**	**B**	q_0	*0*	*0*	*1*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	*X*	q_1	*0*	*1*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	*X*	*0*	q_1	*1*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	*X*	q_2	*0*	*Y*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	q_2	*X*	*0*	*Y*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	*X*	q_0	*0*	*Y*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	*X*	*X*	q_1	*Y*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	*X*	*X*	*Y*	q_1	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	*X*	*X*	*Y*	*0*	q_1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯

Observe that in the last configuration no continuation is possible, and hence that there is no accepting computation starting from 0010.

Exercise 7.2 *(solution on page 234) [Subtraction in unary] Construct a Turing machine program that realizes a subtraction in unary: Starting from a word of the form* 0^m10^n*, the machine stops with* $0^{m\ominus n}$ *on its tape surrounded by blanks (where* $m\ominus n$ *is* $\max(0, m-n)$*).*

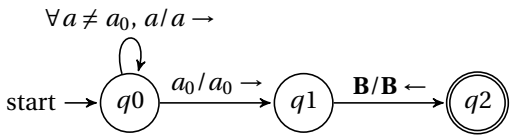
7.1.4 Some programming techniques

Here are a few programming techniques that are used for programming Turing machines.

The first consists in encoding some finite information in the internal state of the machine. We will illustrate this on an example, where we will store the first read character in the internal state. As long as the information to store is finite, this is possible.

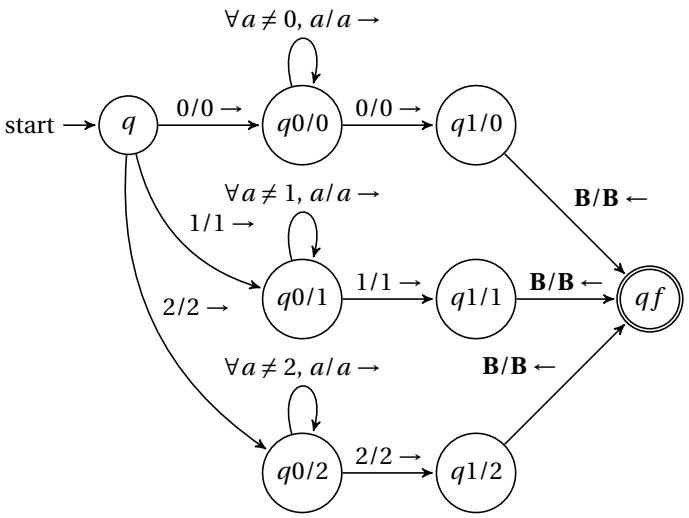
Exercise 7.3 *Construct a Turing machine that reads the symbol in front of the head and checks that this character letter is also the last letter of the input word but does not appear anywhere else in the input.*

If one fixes the symbol $a_0 \in \Sigma$ of alphabet Σ, it is easy to construct a program that checks that the symbol a_0 is appearing nowhere but as the last letter on the right. Indeed, consider:



where $\forall a \neq a_0$ means that one repeats the transition $a/a, \rightarrow$ for all symbols $a \neq a_0$.

Now, to solve our problem, it is sufficient to read the first letter a_0 and to copy this program as many times as the number of letters in alphabet Σ. If $\Sigma = \{0,1,2\}$ for example:



We are hence using the fact that this program is working on some states that can be ordered pairs: here we use ordered pair q_i/j with $i \in \{1,2\}$, and $j \in \Sigma$.

A second technique is using subprocedures. Once again, we will illustrate this with an example.

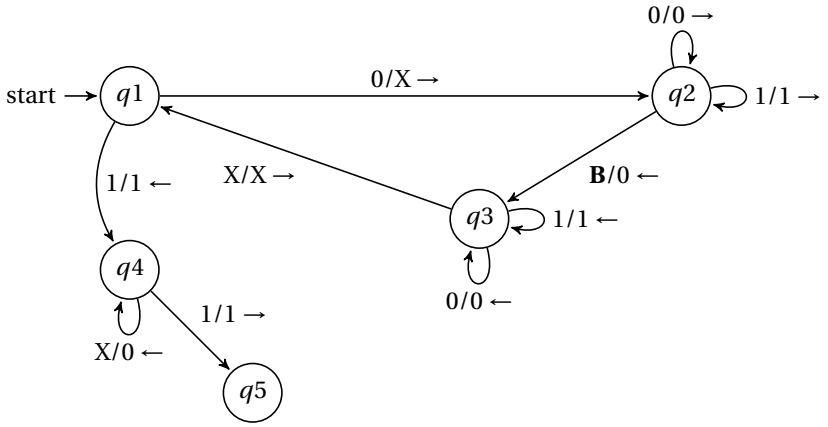
Exercise 7.4 *[Multiplication in unary] Construct a Turing machine that performs multiplication in unary: Starting from a word of the form* 0^m10^n*, the machine stops with* $0^{m\cdot n}$ *on its tape.*

A possible strategy is the following:

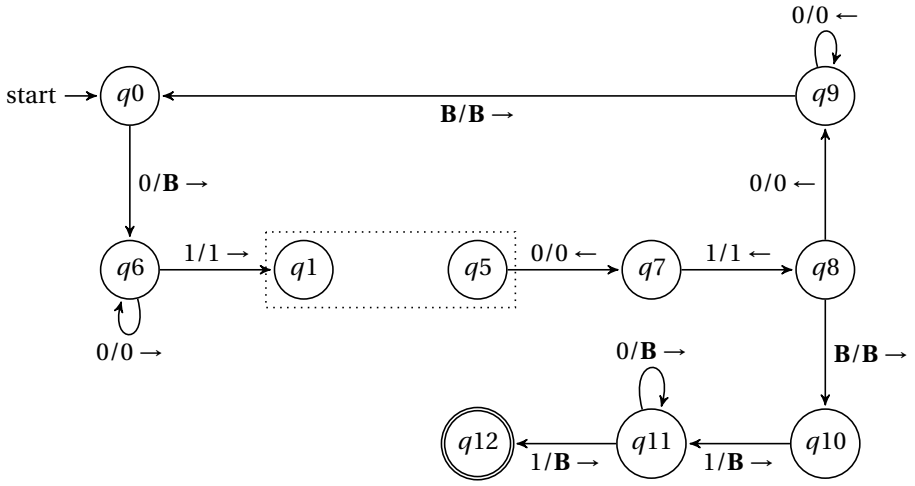
1. the tape will contain a word of the form $0^i10^n10^{kn}$ for an integer k;
2. at every step, one will change a 0 of the first group into a blank, and one will add n 0's to the first group, to obtain the word $0^{i-1}10^n10^{(k+1)n}$;
3. by doing so, we will copy the group of n 0's m times, once for every symbol of the first group set to blank. When there is no blank left in the first group of 0's, there will consequently be $m \cdot n$ 0's in the last group;
4. the last step is to overwrite the prefix 10^n1 with blanks.

The heart of the method is hence the subprocedure, that we will call $Copy$ that implements step 2: It transforms a configuration $0^{m-k}1q_10^n1^{(k-1)n}$ into $0^{m-k}1q_50^n1^{kn}$.

Here is a way to program this: If one starts in state q_1 with such an input, we will eventually go to state q_5 with the correct result.



Once we have this subprocedure, one can construct the global algorithm.



where the dashed rectangle means "paste the program just describe for the subprocedure here".

In this example we see that it is possible to program with Turing machines in a modular way, by using subprocedures. In practise, this means pasting pieces of program inside a program of a machine as in the example.

7.1.5 Applications

As we said before, the only way to understand all what can be programmed with a Turing machine is trying to program them.

So here are a few exercises.

Exercise 7.5 *Construct a Turing machine that adds* 1 *to the number written in binary (hence with* **0** *and* **1***'s) on its tape.*

Exercise 7.6 *Construct a Turing machine that subtracts* 1 *to the number written in binary (hence with* **0** *and* **1***'s) on its tape.*

Exercise 7.7 *Construct a Turing machine that accepts the words with the same number of* **0***'s and* **1***'s.*

7.1.6 Variants of the notion of Turing machine

The Turing machine model is extremely robust.

Indeed, there are many possible variations of the model, but they do not change what can be computed with these machines.

In this section, we will illustrate this by discussing several of these modifications.

Restriction to a binary alphabet

Proposition 7.1 *Every Turing machine working over some arbitrary alphabet* Σ *can be simulated by a Turing machine working over alphabet* $\Sigma = \Gamma$ *with only two letters (without counting the blank symbol).*

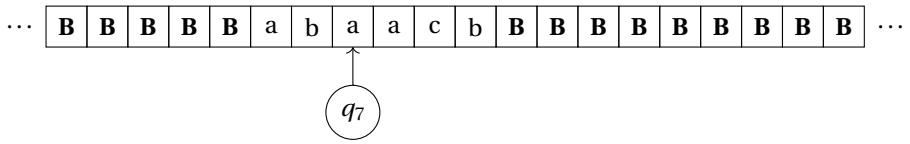
Sketch of proof: The idea is that one can always encode the letters of the alphabet by using a binary encoding. For example, if the alphabet Σ has 3 letters a, b, and c, one can decide to encode a by 00, b by 01 and c by 10: See Figure 7.2. In the general case, one just needs to use possibly more than 2 letters.

Of course, this may require first to initially transform the input in order to rewrite it using this encoding.

One can then transform the program of a Turing machine M that works over alphabet Σ into a program M' that works over this encoding.

For example, if the program of M contains an instruction that says that if M is in state q and that the head reads a one must write c and move to the right, the program of M' will consist in stating that if one is in state q and one reads 0 in front of the head, and 0 on its right (and so what is on the right of the head starts by 00, the encoding of a), then one must replace these two 0's by 10 (i.e. the encoding of c) and go to state q'. By doing so, every time that a computation of M produces a tape corresponding to some word w, then the machine M' will produce a tape corresponding to the encoding of w in binary letter by letter. □

Machine M on alphabet $\{a, b, c\}$



Machine M' simulating M on alphabet $\{0, 1\}$.

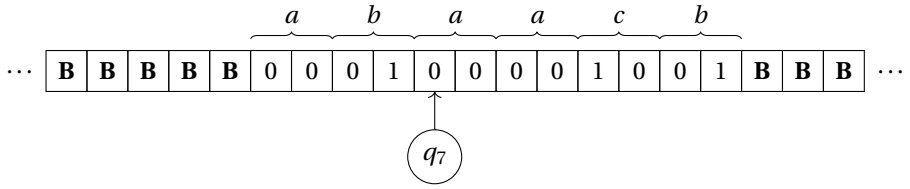


Figure 7.2: Illustration of the proof of Proposition 7.1.

Turing machines with several tapes

One can also consider some Turing machines with several tapes, say k, where k is some integer. Each of these k tapes has its own reading head. The machine still have a finite number of internal states Q. Simply, now the transition function δ is not any more a function of $Q \times \Gamma$ to $Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}$. but from $Q \times \Gamma^k$ to $Q \times \Gamma^k \times \{\leftarrow, |, \rightarrow\}^k$: Depending of the the internal state of the machine and the symbols read by each of the k heads, the transition function gives the new symbols to be written on each of the tapes, and the movements for each of the heads.

It is possible to formalize this model, but we will not do so as this does not really bring a new difficulty.

We will sketch the following result.

Proposition 7.2 *Every Turing machine with k tapes can be simulated by a Turing machine with a unique tape.*

Sketch of proof: The idea is that if a machine M works with k tapes on the alphabet Γ, one can simulate M by a machine M' with a unique tape that works on alphabet $(\Gamma \times \{0, 1\} \cup \{\#\})$ (which is still a finite alphabet).

The tape of M' contains the concatenation of the contents of all the tapes of M, separated by some marker $\#$. We use $(\Gamma \times \{0, 1\} \cup \{\#\})$ instead of $(\Gamma \cup \{\#\})$ in order to use 1 more bit of information for every cell to store the information "the read is in front this cell".

M' will simulate step by step the transitions of M: To simulate a transition of M, M' will scan from left to right its tape to determine the position of each of the

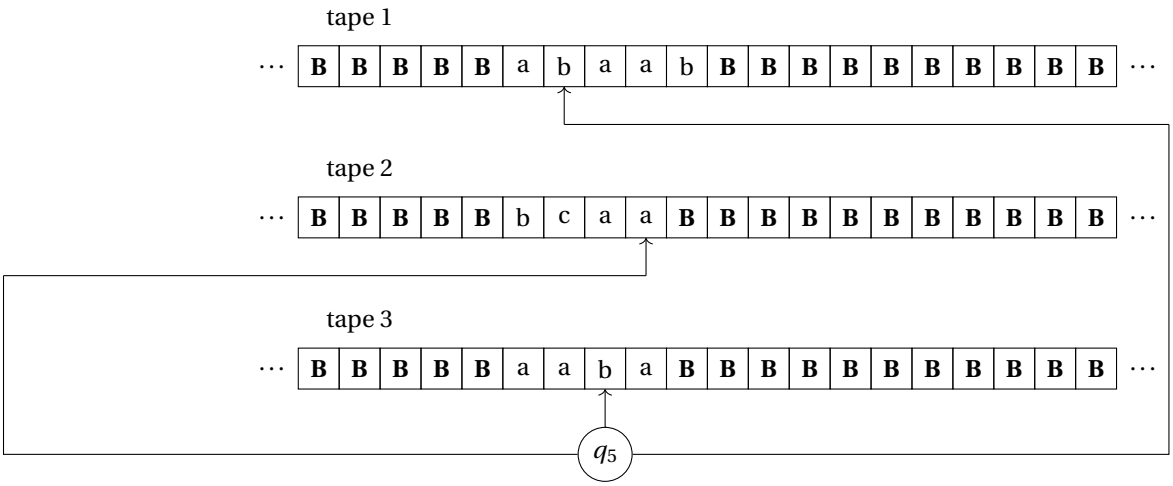


Figure 7.3: A Turing machine with 3 tapes

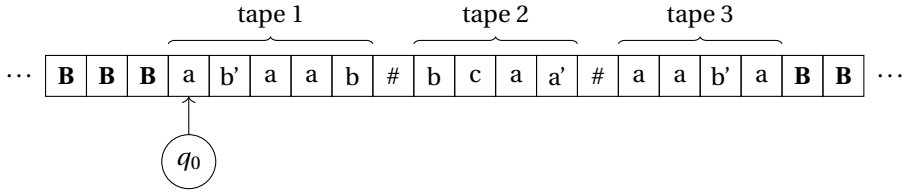


Figure 7.4: Illustration of the proof of Proposition 7.2: Graphical representation of a Turing machine with 1 tape simulating the machine with 3 tapes of Figure 7.3. On this graphical representation, we write a primed letter when the bit "the head is in front of this cell" is set to 1.

reading heads, and the symbol in front of each of the heads (by memorizing these symbols in its internal state). Once all the symbols in front of all the head known, M' knows the symbols to be written and the movements to be done for each of the heads: M' will scan again its tape from left to right to update its encoding of the of the configuration of M. By doing so systematically transition after transition, M' will perfectly simulate the evolution of M with unique tape: See Figure 7.1.6 □

Non-deterministic Turing machines

We can also introduce non-determinism in Turing machines: The definition of a Turing machine is exactly as the notion of (deterministic) Turing machine except for one point. δ is not a function from $Q \times \Gamma$ to $Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}$ anymore, but a relation of the form

$$\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}).$$

In other words, to a given internal state and a letter read in front of the head, δ is not defining a unique triple of $Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}$, but possibly a set of triples. Intuitively, during an execution, the machine has the possibility to choose any triple.

Formally, this is expressed by the fact that one can go from configuration C to successor configuration C' if and only one can go from C to C' (we will write this $C \vdash C'$) with previous definitions, but replacing $\delta(q,a) = (q', a', m')$ by $((q,a),(q',a',m')) \in \delta$. The other definitions are unchanged, and the same as for the deterministic Turing machines.

The difference is that a non-deterministic Turing machine does not have a unique execution on some input w but possibly several: Actually, the executions of the machine on a word w give rise to a tree of possibilities, and the idea is that one accepts a word if one of the branches contains an accepting configuration.

The notion of word w accepted is (still) given by Definition 7.5.

The language $L \subset \Sigma^*$ *accepted by* M is (still) the set of words w that are accepted by the machine. We (still) denote it by $L(M)$. We (still) also call $L(M)$ *the language recognized* by M.

One avoids in general in this context to talk of *rejected word*.

We will however say that a language $L \subset \Sigma^*$ is *decided by* M if it is accepted by a machine that halts on every input: That is to say, for every $w \in L$, the machine has (at least) **ONE** accepting computation that leads to some accepting configuration as in Definition 7.5, and for every $w \notin L$, **ALL** the computations of the machine lead to some rejecting configuration.

One can prove the following result (we will do so in a following chapter).

Proposition 7.3 *Any non-deterministic Turing machine can be simulated by a deterministic Turing machine: A language L is accepted by a non-deterministic Turing machine if and only if it is accepted by some (deterministic) Turing machine.*

Obviously, one can consider a Turing machine as a particular non-deterministic Turing machine. The non-trivial direction of the proposition is that one can always simulate a non-deterministic machine by some deterministic one.

In other words, allowing non-determinism does not extend the power of the model, as long as one talks about *computability*, that is to say about the problems that one can solve. We will see that this is not so direct when talking about efficiency, i.e. *complexity*.

7.1.7 Locality of the notion of computation

We now give a fundamental property of the notion of computation that we will use at several occasions, and that we invite the reader to meditate on:

Proposition 7.4 (Locality of the notion of computation) *Consider the space-time diagram of a machine M. We consider the possible contents of a subrectangle of width 3 and height 2 in this diagram. For every machine M there is a finite number of possible contents that one can find in these rectangles. We call the possible contents for the machine M the* legal window *: See Figure 7.6 for an illustration.*

This even provides a characterization of the space-time diagrams of a given machine: An array is a space-time diagram of M for some initial configuration C_0 if and only if its first line corresponds to C_0, and furthermore in this array, the content of all the subrectangles of width 3 and height 2 are among the legal windows.

Proof: It is sufficient to look at all possible cases. This is very tedious, but not difficult. □

We will come back to this. Forget this for now, and let us come back for now to other models in next chapter.

Remark 7.4 *Similar results can also be shown for other models. However, they are often harder to formulate for them.*

7.2 Bibliographic notes

Suggested readings To go further with all the mentioned notions in this chapter, we suggest to read [Sipser, 1997], [Hopcroft & Ullman, 2000].

Bibliography This chapter has been written using the presentation of Turing machines in [Wolper, 2001], and discussions in [Hopcroft & Ullman, 2000] for the part about their programming. The part on the (S)RAM is inspired by [Papadimitriou, 1994] and [Jones, 1997].

(*a*)

⋯	**B**	**B**	**B**	**B**	q_0	0	0	1	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	X	q_1	0	1	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	X	0	q_1	1	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	X	q_2	0	Y	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	q_2	X	0	Y	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	X	q_0	0	Y	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	X	X	q_1	Y	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	X	X	Y	q_1	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	X	X	Y	0	q_1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯

(*b*)

1	0	B
Y	0	B

Figure 7.5: (*a*). The space-time diagram of Example 7.7. We show one 3×2 subrectangle with gray background. (*b*) The corresponding (legal) window.

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

Figure 7.6: Some legal windows for a Turing machine M: each of them can be observed on a 3×2 subrectangle of the space-time diagram of M .

(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_1	a	a

(c)

b	q_1	b
q_2	b	q_2

Figure 7.7: Some illegal windows for a certain Turing machine M with $\delta(q_1, b) = (q_1, c, \leftarrow)$. One cannot observe these contents in a 3×2 subrectangle of the space-time diagram of M: Indeed, in (*a*), the middle symbol cannot be changed without the head being next to it. In (*b*), the symbol at the bottom right should be c but not a, according to the transition function. In (*c*), there cannot be two heads on the tape.

Chapter 8

A few other models of computation

8.1 RAM

The Turing machine model may seem extremely rudimentary. However, it is in fact extremely powerful, and it is able to capture the notion of *computability* in computer science.

The objective of this chapter is to argue the following Church-Turing thesis: Every computation that can be programmed by a digital computational device, such as a modern computer, can be simulated by a Turing machine. In order to make this plausible, we will introduce first a model very close (actually the closest that I know) to the way that today's computers work: The *RAM model*.

8.1.1 RAM model

The model of the *RAM (Random Access Machine)* is a computational model that is close to today's machine languages, and the way the processors of today work.

A RAM has registers that each contain a natural number (null when initialized, i.e. if not yet used). The machine is assumed to have infinitely many registers, indexed by integers. The authorized instructions depend of the processor that one wants to model[1] but in general they include the following:

1. copying the content of a register into another;

2. doing indirect addressing: Get/Write the content of a register whose index is given by the value of some other register;

3. doing some very basic elementary operation on some particular register such as for example adding 1, subtract 1 or test equality to 0 of a register;

[1] And actually, also of the reference book that one takes to formally describe the model.

4. doing some other operation on a or several register(s), for example addition, subtraction, multiplication, division, binary shifts or bitwise binary operations.

In the following, we will limit the discussion to the *SRAM (Successor Random Access Machine)* model that only has instructions of type 1., 2. and 3. We will see later that this does not really change things, as long as each of the allowed operations of type 4. can be simulated by a Turing machine (and this is the case for all operations mentioned above).

8.1.2 Simulation of a RISC machine by a Turing machine

We are going to show that any (S)RAM can be simulated by a Turing machine.

To help the understanding of the proof, we will reduce the set of instructions of the RAM to a reduced set of instructions (*RISC reduced instruction set*) by using a unique register x_0 as an accumulator.

Definition 8.1 *A RISC machine is a (S)RAM whose instructions are (only) of the form:*

1. $x_0 \leftarrow 0$;
2. $x_0 \leftarrow x_0 + 1$;
3. $x_0 \leftarrow x_0 \ominus 1$;
4. ***if*** $x_0 = 0$ ***then*** *go* ***to*** *instruction number* j ;
5. $x_0 \leftarrow x_i$;
6. $x_i \leftarrow x_0$;
7. $x_0 \leftarrow x_{x_i}$;
8. $x_{x_0} \leftarrow x_i$.

Clearly, every SRAM program with instructions of type 1., 2. and 3. can be converted to an equivalent RISC program, by replacing every instruction by instructions that do the equivalent operation by systematically using the accumulator x_0 (if needed).

Example 8.1 *For example: the instruction $x_i \leftarrow x_j$ can be replaced by the two instructions $x_0 \leftarrow x_j$ and then $x_i \leftarrow x_0$.*

We will start with the following simulation:

Theorem 8.1 *Every RISC machine can be simulated by a Turing machine.*

Proof: We describe how to construct a Turing machine that simulates the RISC machine. The Turing machine has 4 tapes. The first two tapes are encoding the pairs (i, x_i) for x_i non null. The third tape encodes the accumulator x_0 and the fourth tape is used as a scratch tape.

More concretely, for every integer i, denote by $\langle i \rangle$ its binary representation. The first tape encodes a word of the form

$$\ldots \mathbf{BB}\langle i_o \rangle \mathbf{B}\langle i_1 \rangle \cdots \mathbf{B} \ldots \langle i_k \rangle \mathbf{BB} \cdots .$$

The second tape encodes a word of the form

$$\ldots \mathbf{BB}\langle x_{i_o} \rangle \mathbf{B}\langle x_{i_1} \rangle \cdots \mathbf{B} \ldots \langle x_{i_k} \rangle \mathbf{BB} \cdots$$

The heads of the first two tapes are on the second **B**. The third tape encodes $\langle x_0 \rangle$, the head being at the left. We call this position of the heads the *standard position*.

The simulation is described for three examples. The reader will find it easy to complete the other cases:

1. $x_0 \leftarrow x_0 + 1$: We increment the content of the third tape which by convention contains the binary encoding of x_0. To do so, the head of tape 3 moves right until it reads a **B** symbol. It then moves left once. It then replaces the **1**'s by by **0**'s, while moving to the left as long as as it reads **1**'s. When a **0** or a **B** is found, it is changed into a **1** and the head moves left until it comes back to the standard position.

2. $x_{23} \leftarrow x_0$: The heads scan the tapes 1 and 2 to the right, block by block (we call block a word delimited by two **B**'s), in parallel (i.e if head of tape 1 reads block number i, then this is true for head of tape 2 and conversely), until the head of tape 1 (and also of tape 2) reaches the end of tape 1, or until a block **B10111B** (**10111** is 23 in binary) is found on tape 1.

   If the end of tape 1 is reached, this means that memory position 23 has never been seen previously. One adds it by writing **10111** at the end of tape 1, and one copies the content of tape 3 (the value of x_0) onto tape 2. Afterwards, all heads are moved back to the standard position.

   Otherwise we have found **B10111B** on tape 1. By construction, the head of tape 2 then points at the **B** after $\langle x_{23} \rangle$. In that case, we must modify the part of tape 3 containing $\langle x_{23} \rangle$ which is done in the following way:

   (a) One copies the content at the right of the head of tape 2 onto tape 4.

   (b) One overwrites the content of x_{23} on tape 2 by the the content of tape 3 (the value of x_0).

   (c) One writes **B**, and one copies the content of tape 4 at the right of the head of tape 2, in order to restore the rest of tape 2.

   (d) One returns to the standard position.

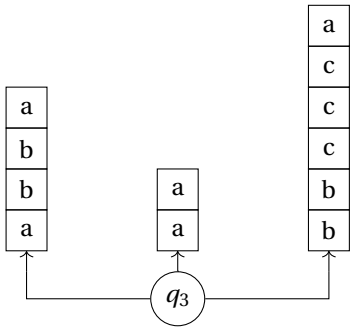


Figure 8.1: A machine with 3 stacks.

3. $x_0 \leftarrow x_{x_{23}}$: Starting from the left of tapes 1 and 2, one scans the tapes 1 and 2 going to the right, block by block, in parallel, until one reaches the end of tape 1, or a block **B10111B** (**10111** is 23 in binary) is read.

   If the end of tape 1 has been reached, one does nothing, since x_{23} values 0 and tape 3 already contains $\langle x_0 \rangle$.

   Otherwise, this means that we found **B10111B** on tape 1. One then reads $\langle x_{23} \rangle$ on tape 2, that one copies on tape 4. As above, one scans tapes 1 and 2 in parallel until one finds **B**$\langle x_{23} \rangle$**B** or the end of tape 1 is reached. If the end of tape 1 is reached, then one writes **0** on tape 3, since $x_{x_{23}} = x_0$. Otherwise, one copies the block corresponding to tape 2 on tape 3, since the block on tape 2 contains $x_{x_{23}}$, and one returns to the standard position.

□

8.1.3 Simulation of a RAM by a Turing machine

Let us come back to the fact that we have reduced the set of possible operations of an *SRAM* to instructions of type 1., 2. and 3. In fact, it is easy to see that one can deal with all instructions of type 4., as long as the underlying operation can be computed by a Turing machine: Every operation $x_0 \leftarrow x_0$ ''operation'' x_i can be simulated as above, as soon as *"operation"* corresponds to some computable operation.

8.2 Rudimentary models

The Turing machine model is extremely rudimentary. It turns out that one can consider models that are even more rudimentary, and that are still able to simulate them.

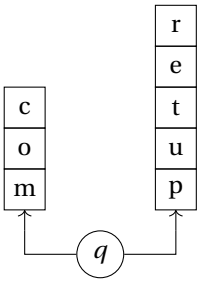


Figure 8.2: The Turing machine from Example 7.3 seen as a 2-stacks machine.

8.2.1 Machines with $k \geq 2$ stacks

A k-stack machine, has k stacks $r_1, r_2, \ldots, r_k$. Each of this stacks corresponds to some stack of elements of the finite alphabet Σ. The instructions of the machine permit only to push a symbol on one of the stacks, read the symbol at the top of a stack, or pop the symbol at the top of stack.

If one prefers, one can see each stack r_i of elements of the finite alphabet Σ as a word w_i over alphabet Σ. Pushing (written $push(i,a)$) a symbol $a \in \Sigma$ on this stack consists in replacing w_i by aw_i. Reading (written $a \leftarrow top(r,i)$) the symbol at the top of this stack consists in reading the first letter of w_i. Doing a pop on (written $pop(i)$) this stack consists in deleting the first letter of w_i.

Theorem 8.2 *Every Turing machine can be simulated by a* 2 *stacks machine.*

Proof: According to the formalization of Page 104, a configuration of a Turing machine corresponds to $C = (q,u,v)$, where u and v denote the content respectively on left and on right of the head of tape i. One can see u and v as stacks: See Figure 8.2. If one reads the formalization Page 104 carefully, one sees that the operations done by the program of a Turing machine to go from configuration C to its successor configuration C' coincide with operations that can be trivially coded by $push$, pop, and top: One can construct a machine with 2 stacks, each stack encoding u or v (the content on the right and on the left of the tape) and that simulate the Turing machine step by step. For example, moving the head to the right, consists in reading the top of the second stack ($a \leftarrow top(2)$), pushing this symbol a on the first stack ($push(1,a)$), and doing a pop on the second stack ($pop(2)$). Moving the head to the left, consists in reading the top of the first stack ($a \leftarrow top(1)$), pushing this symbol a on the second stack ($push(2,a)$), and doing a pop on the first stack $pop(1)$. Changing the symbol in front of the head into symbol a consists in doing a pop on the second stack ($pop(2)$), and then pushing symbol a on second stack ($push(2,a)$). □

8.2.2 Counter machines

We introduce now a model even more rudimentary: A *counter machine* has a finite number k of counters $r_1, r_2, \cdots, r_k$, which contain natural numbers. The instructions of a counter machine allow only to test equality of a given counter to 0, increment a given counter or decrement a given counter. Initially all the counters are set to 0, except for the one encoding the input.

Remark 8.1 *The machine is usually considered as computing functions over the integers, or as recognizing languages defined as subsets of the integers. If one wants to compute over words, say words over the alphabet* Σ *is* $\{0,1\}$*, this requires to encode words into integers, for example by considering binary expansions.*

Remark 8.2 *We consider in this document that machines either halt or loop. Rejection is encoded in the coming simulation by the fact that the machine does not halt. It would be possible also to consider counter machines with* Accept *and* Reject *instructions, to simulate in a fine way acceptance and rejection of Turing machines.*

Remark 8.3 *This is hence a (S)RAM, but with a extremely reduced set of instructions, and with furthermore a finite number of registers.*

More formally, all the instructions of a counter machine are of the following 4 types:

- Inc(c, j): counter c is incremented and then one goes to instruction j;
- Decr(c, j): counter c is decremented (if non null, unchanged otherwise) and then one goes to instruction j;
- IsZero(c, j, k): one tests whether counter c is 0 and one goes to instruction j if this is the case, or to instruction k otherwise;
- Halt: the computation is halted.

Example 8.2 *For example, the following program with* 3 *counters*

1. IsZero$(1,5,2)$
2. Decr$(1,3)$
3. Inc$(3,4)$
4. Inc$(3,1)$
5. Halt

transforms $(n,0,0)$ *into* $(0,0,2n)$*: If one starts with* $r_1 = n$*,* $r_2 = r_3 = 0$*, then when instruction* Halt *is reached, we have* $r_3 = 2n$*, and all other counters set to* 0*.*

Exercise 8.1 *For every of the following conditions, describe some counters machine that reaches instruction* Halt *if and only if the following condition is true on the initial condition:*

1. $r_1 \geq r_2 \geq 1$*;*
2. $r_1 = r_2$ *or* $r_1 = r_3$*;*
3. $r_1 = r_2$ *or* $r_1 = r_3$ *or* $r_2 = r_3$*.*

Theorem 8.3 *Every machine with* k*-stacks can be simulated by a machine with* $k+1$ *counters.*

Proof: The idea is to see a stack $w = a_1 a_2 \cdots a_n$ on an alphabet Σ of size $r-1$ as an integer i in basis (radix) r: Without loss of generality, we can consider Σ to be $\Sigma = \{0,1,\cdots,r-1\}$. The word w can be interpreted as the integer $i = a_n r^{n-1} + a_{n-1} r^{n-2} + \cdots a_2 r + a_1$.

One uses a counter i for every stack. A $k+1$th counter, that we will call *additional counter*, is used to adjust the value of the counters and simulate every operation (push, pop, reading the top) of one of the stacks.

Popping is replacing i by i div r, where div denotes integer division: Starting with the additional counter set to 0, one iteratively in a loop decrements the counter i by r (in r steps) and increments the additional counter by 1. This operation is repeated until counter i reaches value 0. One then copies the additional counter to counter i: one iteratively decrements the additional counter by 1 while incrementing counter i by 1 until the additional counter reaches 0. At this moment, counter i contains the correct result.

Pushing symbol a is replacing i by $i * r + a$: First, one multiplies the counter i by r: Starting with the additional counter set to 0, one decrements the counter i by 1 and one increments the additional counter by r (in r steps) until counter i reaches 0. One then decrements the additional counter by 1 while incrementing counter i until the former reaches 0. At this moment, one reads counter i contains $i * r$. One then increments counter i by a (using a incrementation operations).

Reading the top of stack i is computing $i \ mod \ r$, where $i \ mod \ r$ denotes the remainder of the Euclidean division of i by r: We start by moving the content of i to the additional counter as follows. Starting with additional counter set to 0, one decrements counter i by 1 and one increments the additional counter by 1. When counter i reaches 0 one stops. One then decrements the additional counter by 1 while incrementing counter i until the former reaches 0. While doing this, one memorizes in parallel the number of operations incrementations to counter i performed modulo r in the internal state. When the loop is done, the internal state thus contains i modulo r. □

Theorem 8.4 *Every machine with $k \geq 3$ counters can be simulated by a machine with 2 counters.*

Proof: Suppose first that $k = 3$.

The idea is to encode the three counters i, j and k by integer $m = 2^i 3^j 5^k$. One of the counters stores this integer. The other counter is used to do multiplications, divisions, and compute remainders modulo 2, 3, and 5.

To increment i, j or k by 1, it is sufficient to multiply m by 2, 3 or 5 by using the techniques of the previous proof.

To test whether i, j or $k = 0$, it is sufficient to test whether m is divisible by 2, 3 or 5, by using the techniques of the previous proof.

To decrement i, j or k of 1, it is sufficient to divide m by 2, 3 or 5 using the techniques of the previous proof.

For $k > 3$, we use the same approach, but with the first k prime numbers instead of simply 2, 3, and 5. □

Exercise 8.2 *Reconsider the previous exercise but by using systematically at most 2 counters.*

By combining the previous results, we obtain:

Corollary 8.1 *Every Turing machine can be simulated by a 2 counters machine.*

Remark 8.4 *Observe that the simulation is particularly inefficient: The simulation of a time t of the Turing machine requires an exponential time by a 2 counter machine.*

8.3 Church-Turing thesis

8.3.1 Equivalence of all considered models

In this chapter, we have introduced various models, and we have shown that they can all be simulated by Turing machines, or simulate Turing machines.

Actually, all these models are equivalent in terms of what they are able to compute: We already proved that Turing machines can simulate RAMs. We could also easily prove the contrary: One can simulate a Turing machine using a RAM.

We have also shown that the counter machines and the stack machines with 2 or more stacks can simulate Turing machines. It is easy to see that the contrary holds: One can simulate the evolution of a stack machine or of counter machine by a Turing machine.

Consequently, all these models are equivalent at the level of what they can compute.

8.3.2 Church-Turing thesis

The equivalences we observed before and many others led to the Church-Turing thesis, expressed historically by Alonzo Church, Alan Turing and later also formalized by Stephen Kleene. This thesis states that "what is effectively *calculable* is computable by a Turing machine".

In this formulation, the first notion of "*calculable*" makes reference to a intuitively given notion, while the second notion of "computable" means "computable by a Turing machine", i.e., a formal notion.

Since it is not possible to formally capture the first notion, this is a thesis in the *philosophical* sense of this term: It is not possible to prove it.

However, given two (sufficiently expressive) models, we can mathematically prove, as we did, that everything that can be computed by the first can be simulated, and hence computed, by the second (and conversely). This gives evidence that the notion of "computable" as defined with Turing machines is matching the intuitive notion of "computable" (or calculable).

While it cannot be proved, the Church-Turing thesis is widely assumed to be true.

8.4 Bibliographic notes

Suggested readings To go further with all the mentioned notions in this chapter, we suggest to read [Sipser, 1997],[Hopcroft & Ullman, 2000].

Other formalisms equivalent to Turing machines exist, in particular the notion of recursive functions that is presented for example in [Dowek, 2008], [Stern, 1994] or in [Cori & Lascar, 1993b].

Bibliography (S)RAM is inspired by [Papadimitriou, 1994] and [Jones, 1997].

Chapter 9

Computability

This chapter presents some main results of computability theory. In other words, this chapter is devoted to understanding the power of (modern today's) computers. We will prove that some problems cannot be solved using a computer: The objective is to explore the limits of computer programming.

Remark 9.1 *In practice, one could say that in computer science, one aims to solve problems by implementing algorithms as programs and that discussing the problems that cannot be solved by programs has only little interest. But, actually, it is very important to understand that we will not focus on problems for which no solution is known, but on something much stronger: We will focus on problems for which it is (provably) impossible to produce any algorithmic solution.*

Why focusing on understanding the problems that cannot be solved? First because understanding that a problem cannot be solved is useful: This means in particular that the problem must be simplified or modified in order to be solved. Second, because all these results are culturally very interesting and provide a perspective on programming, and on limits of computational devices, or of the automation of some tasks, such as the verification of programs.

9.1 Universal machines

9.1.1 Interpreters

A certain number of these results is the consequence of a simple fact, that has many consequences: One can program *interpreters*, that is to say programs that takes as input the description of some other program and that then simulate this program.

Example 9.1 *A language such as Java or Python is*[a] *interprete d: A Java program for example is compiled into some encoding that is called a* bytecode. *When one wants to start this program, the Java interpreter simulates this bytecode on the*

machine on which it is executed. This principle of interpretation allows a Java program to work on numerous platforms and directly on various machines: Only the interpreters depends on the machine on which the program is executed. This portability is partly what historically led to the success of the Java language (and remains true for interpreted languages such as Python).

[a] The discussion here is true for early versions of Java. Now, just-in-time compilation is often used, and this discussion is only partially true.

The possibility of programming interpreters is thus practically extremely positive.

However, it also leads to mathematical proofs of numerous negative results or to paradoxical results about the impossibility of solving certain problems with a computer, even for very simple problems, as we will see shortly.

Programming an interpreter is possible in all usual programming languages, in particular it can be programmed using Turing machines.

Remark 9.2 *We will not talk about Java or Python in what follows, but about programs for Turing machines. Reasoning on Java (or any other language) would only complicate the discussion without changing the heart of the arguments.*

Let us start by getting convinced that one can construct an interpreter for Turing machines. In that context, an interpreter is called a *universal!Turing machine*.

9.1.2 Encoding Turing machines

We first need to fix a representation of programs of Turing machines. Here is a way to do so.

Remark 9.3 *The following encoding is only a convention. Any other encoding that would guarantee the possibility of decoding (for example in the spirit of Lemma 9.1) would work and would be a valid alternative.*

Definition 9.1 (Encoding of a Turing machine) *Let M be a Turing machine on alphabet $\Sigma = \{0,1\}$.*

According to Definition 7.1, M is given by a 8-tuple

$$M = (Q, \Sigma, \Gamma, \mathbf{B}, \delta, q_0, q_a, q_r):$$

- *Q is a finite set, whose elements are $Q = \{q_0, q_1, \cdots, q_{r-1}\}$, with the convention that q_0 is the initial state, and that $q_1 = q_a$, $q_2 = q_r$;*
- *Γ is a finite set, whose elements are $\Gamma = \{X_1, X_2, \cdots, X_s\}$, with the convention that X_s is the blank $\mathbf{B}$ symbol, and that X_1 is the symbol 0 of Σ, and that X_2 is the symbol 1 of Σ.*

For $m \in \{\leftarrow, |, \rightarrow\}$, define $\langle m \rangle$ as follows: $\langle \leftarrow \rangle = 1$, $\langle | \rangle = 2$, $\langle \rightarrow \rangle = 3$.

We can encode the transition function δ as follows: Suppose that the transition rule is $\delta(q_i, X_j) = (q_k, X_l, m)$: The encoding of this rule is the word $0^i 10^j 10^k 10^l 10^{\langle m \rangle}$ on alphabet $\{0,1\}$. Observe that for any non-null integers i, j, k, l, there is no consecutive 1 in this word.

An encoding, denoted $\langle M \rangle$, of Turing machine M is a word on alphabet $\{0,1\}$ of the form

$$C_1 11 C_2 11 C_3 \cdots C_{n-1} 11 C_n,$$

where each C_i is the encoding of one of the transition rules of δ.

Remark 9.4 *To a given Turing machine M one can associate several encodings $\langle M \rangle$: In particular, one can permute the $(C_i)_i$, or the states, etc.*

The only interest of this particular encoding is that it can be easily decoded: One can easily find all the ingredients of the description of a Turing machine from the encoding of the machine.

For example, if one wants to determine the movement m for a given transition:

Lemma 9.1 (Decoding the encoding) *One can construct a Turing machine M with four tapes, such that if the encoding $\langle M' \rangle$ of a machine M' is put on its first tape, 0^i is put on its second tape, and 0^j is put on its third tape, M produces on its fourth tape the encoding $\langle m \rangle$ of the movement $m \in \{\leftarrow, |, \rightarrow\}$ such that $\delta(q_i, X_j) = (q_k, X_l, m)$ where δ is the transition function of the machine M'.*

Sketch of proof: Construct a machine that scans the encoding of M' until it finds the encoding of the associated transition, and that then reads in this encoding of this transition the desired value of m. □

We will need to encode pairs made of the encoding of a Turing machine M and of a word w on the alphabet $\{0,1\}$. One way to do this is to define the encoding of this pair, denoted $\langle \langle M \rangle, w \rangle$, by

$$\langle \langle M \rangle, w \rangle = \langle M \rangle 111 w,$$

that is to say the word obtained by concatenating the encoding of the Turing machine M, three times the symbol 1, and then the word w. Since our encoding of Turing machines never produces three consecutive 1's, the idea is that one can find in the word $\langle M \rangle 111 w$ the part which is $\langle M \rangle$ and the part which is w: In short: this encoding guarantees that one can decode unambiguously $\langle M \rangle$ and w from $\langle \langle M \rangle, w \rangle$.

9.1.3 Encoding pairs, triplets, etc...

We have just fixed an encoding that works for a Turing machine and a word. We now more generally fix a way to encode two words w_1 and w_2 into a unique word w. In other words, we give a way to encode an (ordered) pair of words, i.e., an element of $\Sigma^* \times \Sigma^*$, by a unique element (i.e. word) of Σ^*, that we will denote $\langle w_1, w_2 \rangle$.

How to do this?

A first idea is to encode two words of Σ using a bigger alphabet, in such a way that it is possible to reconstruct the initial words.

For example, one could encode the words $w_1 \in \Sigma^*$ and $w_2 \in \Sigma^*$ by the word $w_1 \# w_2$ on alphabet $\Sigma \cup \{\#\}$. A Turing machine can then easily determine both w_1 and w_2 from the word $w_1 \# w_2$.

One can also re-encode the obtained word one letter after the other to obtain a way to encode two words on $\Sigma = \{0,1\}$ by a unique word on $\Sigma = \{0,1\}$: For example, if $w_1 \# w_2$ is written $a_1 a_2 \cdots a_n$ on alphabet $\Sigma \cup \{\#\}$, we define $\langle w_1, w_2 \rangle$ as the word $e(a_1)e(a_2)\dots e(a_n)$ where $e(0) = 00$, $e(1) = 01$ and $e(\#) = 10$. This encoding is still decodable: From $\langle w_1, w_2 \rangle$, a Turing machine can decode w_1 and w_2.

From now, we will denote by $\langle w_1, w_2 \rangle$ the encoding of the pair consisting of the word w_1 and of the word w_2.

Observe that the coming results do not depend on the concrete encoding used for pairs: One can hence encode a pair made of a Turing machine and of a word as in previous section, or consider it as $\langle \langle M \rangle, w \rangle$, that is to say the encoding of a pair made of the encoding of the machine and of the word, indifferently.

9.1.4 Existence of a universal Turing machine

After these preliminary discussions, we now show that one can construct an interpreter, that is to say what is called a *universal!Turing machine* in the context of Turing machines.

Its existence is given by the following theorem:

Theorem 9.1 (Existence of a universal Turing machine) *There exists a Turing machine M_{univ} such that, on input $\langle \langle A \rangle, w \rangle$ where*

1. *$\langle A \rangle$ is the encoding of a Turing machine A;*
2. *$w \in \{0,1\}^*$;*

M_{univ} simulates the Turing machine A on input w.

Proof: One can easily see that there exists a Turing machine M_{univ} with three tapes such that if one puts:

- the encoding $\langle A \rangle$ of a Turing machine A on the first tape;
- a word w on alphabet $\Sigma = \{0,1\}$ on the second;

then M_{univ} simulates the Turing machine A on input w by using its third tape.

Indeed, the machine M_{univ} simulates transition after transition the machine A on input w on its second tape: M_{univ} uses the third tape to store 0^q where q is encoding the state of the Turing machine A at the transition that one is currently simulating: Initially, this tape contains 0, the encoding of q_0.

To simulate each transition of A, M_{univ} reads the letter X_j in front of its head on the second tape, then reads in the encoding $\langle A \rangle$ on the first tape the value of q_k, X_l and m, for the transition $\delta(q_i, X_j) = (q_k, X_l, m)$ of A, where 0^i is the content of the

third tape. Then M_{univ} writes then X_l on its second tape, writes q_k on its third tape, and moves the head of the second tape according to the movement m.

To prove the result, it is then sufficient to use a Turing machine with only one tape that simulates the previous machine with three tapes, by first decoding $\langle A \rangle$ and w from the input. □

9.1.5 First consequences

Here is a first consequence of the existence of interpreters: The proof of Proposition 7.3, that is to say the proof that a non-deterministic Turing machine M can be simulated by a deterministic Turing machine.

Proof:[Of Propositon 7.3] The transition relation of the non-deterministic machine M is necessarily of *bounded degree of non-determinism*: That is to say, the number

$$r = max_{q \in Q, a \in \Gamma} |\{((q,a),(q',a',m)) \in \delta\}|$$

is finite (|.| denotes the size).

Suppose that for each pair (q,a), we number the choices among the transition relation of the non-deterministic machine M from 1 to (at most) r. At this moment, to describe the non-deterministic choices done by the machine up to time t, it is sufficient to give a sequence of t numbers between 1 and (at most) r.

We construct a (deterministic) Turing machine that simulates the machine M in the following way: For $t = 1, 2, \cdots$, it enumerates all the sequences of length t of integers between 1 and r. For each of these sequences, it simulates t steps of the machine M by making the choices given by the sequence. Doing this for all sequences simulates all potential non-deterministic choices of M. The machine stops and accepts as soon as it finds some t and a sequence such that M reaches an accepting configuration. □

9.2 Languages and decidable problems

Having established the existence of a universal Turing machines (interpreter), we will present a few additional definitions in this section.

In the rest of this chapter, we will only focus on problems whose answer is either *true* or *false*, which will allow us to simplify the discussion, see Figure 9.1.

9.2.1 Decision problems

Definition 9.2 *A* decision problem *is given by a set E, called the set of* instances, *and by a subset $E^+ \subseteq E$, called the set of the* positive instances.

The question on which we will focus is development of an algorithm (when this is possible) that decides whether a given instance is positive or not, i.e. belongs to E^+. We will formulate the decision problems systematically in the following form:

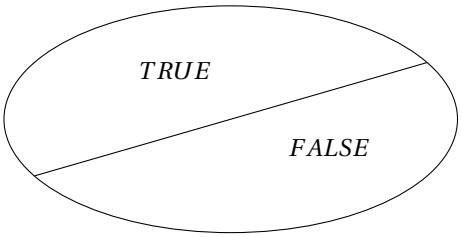


Figure 9.1: Decision problems: In a decision problem, we have a property that is either true or false for each instance. The objective is to distinguish the positive instances E^+ (were the answer is true) from negative instances $E \backslash E^+$ (where the property is false).

Definition 9.3 (*Name of the problem*)

Input: *An instance (that is to say an element of E).*

Answer: *Decide if a given property holds (that is to say if this element belongs to E^+).*

For example, we can consider the following problems:

Definition 9.4 (PRIME NUMBER)

Input: *An integer n.*

Answer: *Decide if n is prime.*

Definition 9.5 (ENCODING)

Input: *A word w.*

Answer: *Decide if w is the encoding $\langle M \rangle$ of some Turing machine M.*

Definition 9.6 (REACH)

Input: *A triple consisting of a graph G, a vertex u and a vertex v of the graph.*

Answer: *Decide whether there exists a path between u and v in G.*

9.2.2 Problems versus Languages

We interchangably use the terminology *problem* or *language* in the upcoming discussions and chapters.

Remark 9.5 (Problems vs Languages) *This is due to the following considerations: To a decision problem we can associate a language and conversely.*

Indeed, to a decision problem we generally implicitly associate an encoding function (for example for graphs, a way to encode the graphs) that allows encoding the instances, that is to say the elements of E, by a given word on a certain alphabet Σ. One can then see E as a subset of Σ^*, where Σ is this alphabet: With a decision problem $\mathscr{P}$, we associate the language $L(\mathscr{P})$ defined as the set of words which encode instances of E which belong to E^+:

$$L(\mathscr{P}) = \{w \mid w \in E^+\}.$$

Conversely, we can see any language L as a decision problem, by formulating it as follows:

***Definition 9.7** (Problem associated to the language **L**)*

***Input**: A word w.*

***Answer**: Decide if $w \in L$.*

9.2.3 Decidable languages

We recall the notion of decidable language.

Definition 9.8 (Decidable language) *A language $L \subset \Sigma^*$ is said to be* decidable *if it is decided by some Turing machine.*

A language or a problem that is decidable is also said to be *recursive*. A language that is not decidable is said to be *undecidable*.

We write D for the class of languages or of problems that are *decidable*.

For example:

Proposition 9.1 *The decision problems* PRIME NUMBER, ENCODING *and* REACH *are* decidable.

The proofs consists in constructing a Turing machine that decides if its input is a prime number (respectively: the encoding of a Turing machine, or a positive instance of REACH). We leave this as an exercise in elementary programming to our readers.

Exercise 9.1 *(solution on page 234) Let A be the language consisting of the only string s where*

$$s = \begin{cases} 0 & \text{if God does not exist} \\ 1 & \text{if God exists} \end{cases}$$

Is A decidable? Why? (Hint: The answer does not depend on the religious convictions the reader).

9.3 Undecidability

In this section, we will prove one of the philosophically most important result in the theory of programming: The existence of problems that cannot be decided, i.e. that are *undecidable*.

9.3.1 First considerations

Observe first that this can be established easily.

Theorem 9.2 *There exist decision problems which are not* decidable.

Proof: We have seen that one can encode a Turing machine by a finite word on alphabet $\Sigma = \{0,1\}$, see Definition 9.1. There is consequently a countable number of Turing machines.

By contrast, there is an uncountable number of languages over the alphabet $\Sigma = \{0,1\}$: Indeed, we saw in Chapter 1 that the power set of $\mathbb{N}$ is uncountable, using Cantor diagonalization argument. Now, this must also be the case for the set of languages, as there is an easy bijection this latter set and power set of $\mathbb{N}$: just take the characteristic functions of the languages.

There are consequently more problems than those that can be solved by any Turing machine. There must thus be decision problems that are not solved by any Turing machine (and there is even an uncountable number of such problems). □

In general, a proof as the one above does not say anything about examples of undecidable problems. Are they esoteric? Are they only of interest for theoreticians?

Unfortunately, this is not the case as even some simple and natural problems turn out not be solvable by any algorithm.

9.3.2 Is this problematic?

For example, in one important undecidable problem, we are given a program and a specification of what this program is supposed to do (for example sorting some numbers) and one wants to check if the program matches its specification (i.e. is a correct sorting algorithm).

We could hope that this process of *verification* could be automatized, that is to say that we could design an algorithm that would test if a given program satisfies its *specification*. Unfortunately, this is impossible: The general problem of *verification* is undecidable, and can thus not be solved on a computer.

We will meet some other *undecidable* problems in this chapter. Our objective will be to make our reader feel what types of problems are undecidable, and to understand the techniques that permit to prove that a given problem cannot be solved algorithmically, i.e. cannot be solved by a computer.

9.3.3 A first undecidable problem

We will use a *diagonalisation argument* that is to say an argument close to Cantor's diagonalisation. Recall that Cantor's diagonalisation is used to prove that the set of subsets of $\mathbb{N}$ is uncountable, see the first chapter.

Remark 9.6 *Behind the previous argument on the fact that there is an uncountable number of languages on the alphabet $\Sigma = \{0,1\}$ is already a diagonalization argument. Here, we will do a more explicit, and more constructive diagonalization.*

We call the following decision problem *universal!language*.

Definition 9.9 (L_{univ})

Input: *The encoding $\langle M \rangle$ of a Turing machine M and a word w.*

Answer: *Decide if the machine M accepts the word w.*

Remark 9.7 *One can also see this problem in the following way: We are given a pair $\langle \langle M \rangle, w \rangle$, where $\langle M \rangle$ is the encoding of a Turing machine M, and w a word, and one wants to decide if the machine M accepts the word w.*

Theorem 9.3 *The problem L_{univ} is not decidable.*

Proof: We prove the result by contradiction. Suppose that L_{univ} is decided by some Turing machine A.

We are then able to construct a machine B working as follows:

- B takes as input a word $\langle C \rangle$ representing the encoding of a Turing machine C;
- B calls the Turing machine A on the pair $\langle \langle C \rangle, \langle C \rangle \rangle$ (that is to say on the input consisting of the encoding of the Turing machine C and the word w equal to this encoding);
- If the Turing machine A:

- accepts this word, B rejects;
- rejects this word, B accepts.

By construction, and from our hypothesis, B halts on every input.

We prove now that there is a contradiction, by applying the Turing machine B on the word $\langle B \rangle$, that is on the word encoding the Turing machine B.

- If B accepts $\langle B \rangle$, then if follows by definition of $\mathrm{L}_{\mathrm{univ}}$ and of A, that A accepts $\langle \langle B \rangle, \langle B \rangle \rangle$. But if A accepts this word, B is constructed such that it rejects the input $\langle B \rangle$. Contradiction.
- If B rejects $\langle B \rangle$, then it follows by definition of $\mathrm{L}_{\mathrm{univ}}$ and of A, that A rejects $\langle \langle B \rangle, \langle B \rangle \rangle$. But if A rejects this word, B accept the input $\langle B \rangle$ by construction. Contradiction.

□

9.3.4 Semi-decidable problems

While the problem $\mathrm{L}_{\mathrm{univ}}$ is undecidable, it is semi-decidable in the following sense:

Definition 9.10 (Semi-decidable language) *A language $L \subset M^*$ is said to be* semi-decidable *if it is the set of words accepted by some Turing machine M.*

Corollary 9.1 *The universal language* $\mathrm{L}_{\mathrm{univ}}$ *is semi-decidable.*

Proof: To know if one must accept some input that is the encoding $\langle M \rangle$ of a Turing machine M and of a word w, it is sufficient to simulate the Turing machine M on input w. One stops the simulation and one accepts if one detects in this simulation that the Turing machine M reaches some accepting state. Otherwise, one simulates M for ever. □

A language *semi-decidable* is also called *computably enumerable* (sometimes also called *recursively enumerable*).

We write CE for the class of languages and decision problems semi-decidable: See Figure 9.2.

Corollary 9.2 $\mathrm{D} \subsetneq \mathrm{CE}$.

Proof: The inclusion follows directly from the definitions. Since $\mathrm{L}_{\mathrm{univ}}$ is in CE and is not in D, the inclusion is strict. □

9.3.5 A problem that is not semi-decidable

Let us start by establish the following fundamental result:

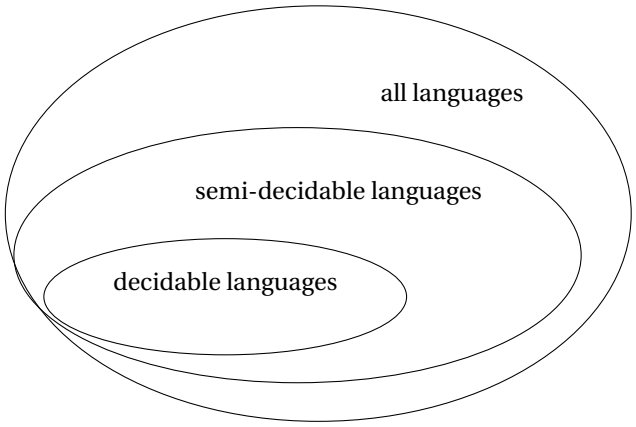


Figure 9.2: Inclusions between classes of languages.

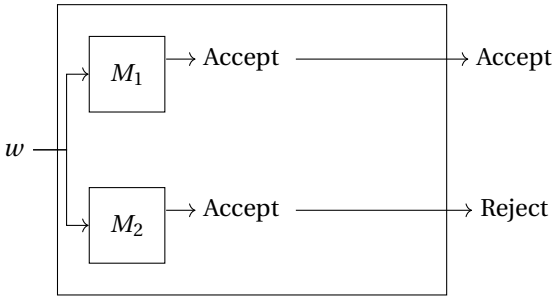


Figure 9.3: Illustration of the proof of Theorem 9.4.

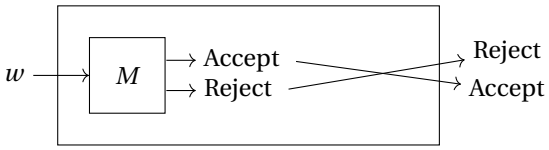


Figure 9.4: Construction of a Turing machine that accepts the complement of a decidable language.

Theorem 9.4 *A language is* decidable *if and only it it is* semi-decidable *and its complement is also* semi-decidable.

Remark 9.8 *Theorem 9.4 explains the terminology* semi-decidable*: A language that is semi-decidable and whose complement as well, is decidable. So when a language is semi-decidable, one half of the requirements to be decidable is in a sense satisfied.*

Proof: Direction $\Leftarrow$. Suppose that L is semi-decidable as well as its complement. There exists a Turing machine M_1 which halts and accepts words of L, and a Turing machine M_2 which halts and accepts words of its complement. We construct a Turing machine M that, on a given input w, simulates in parallel[1] M_1 and M_2, (that is to say it simulates t steps of M_1 on w, and then t steps of M_2 on w, for $t = 1, 2, \cdots,$) until one of the two machines halts. See Figure 9.3. If M_1 halts and accepts, the Turing machine M accepts. If M_2 does so, the machine M rejects. Obviously, the Turing machine that we just described decides L.

Direction $\Rightarrow$. By definition, a decidable language is semi-decidable. By exachanging the accepting state and the rejecting state in the Turing machine, its complement is also decidable (See Figure 9.4) and hence is also semi-decidable. □

We now consider then the complement of the problem $\mathrm{L}_{\mathrm{univ}}$, that we will denote $\overline{\mathrm{L}_{\mathrm{univ}}}$.

Remark 9.9 *In other words, by definition, a word w is in $\overline{\mathrm{L}_{\mathrm{univ}}}$ if and only w is not in $\mathrm{L}_{\mathrm{univ}}$, that is to say*

- *not of the form $\langle\langle M\rangle, w\rangle$, for some Turing machine M,*
- *or of the form $\langle\langle M\rangle, w\rangle$ but with Turing machine M that does not accept input w.*

Corollary 9.3 *The problem $\overline{\mathrm{L}_{\mathrm{univ}}}$ is not semi-decidable.*

Proof: Otherwise, by the Theorem 9.4, its complement, the problem $\mathrm{L}_{\mathrm{univ}}$, would be decidable. □

[1] One alternative is to consider that M is a non-deterministic Turing machine that simulates in a non-deterministic way either M_1 or M_2.

9.3.6 On the terminology

A decidable language is also called a *recursive language*: The terminology is a reference to the notion of recursive functions, see for example the course [Dowek, 2008] which present computability through recursive functions, or Section 9.7.2.

The notion of *enumerable* in *computably enumerable* is explained by the following result.

Theorem 9.5 *A language $L \subset M^*$ is* computably enumerable *if and only if one can construct some Turing machine that outputs one after the other all the words of language L.*

Proof: Direction $\Rightarrow$. Suppose that L is computably enumerable. There exists some Turing machine A which halts and which accepts the words of L.

The set of pairs (t, w), where t is some integer, and where w is a word is countable. One can even get convinced easily that is *effectively* countable: One can construct some Turing machine that produces the encoding $\langle t, w\rangle$ of all the pairs (t, w). For example, one considers a loop that for $t = 1, 2, \cdots$ for ever, considers all the words w of length or equal to t, and produces for each pair the word $\langle t, w\rangle$.

Consider a Turing machine B that for each produced pair (t, w), simulates t steps of the machine A. If the machine halts and accepts in exactly t steps, B outputs the word w. Otherwise B does not print anything for this pair.

A word of language L, is accepted by A at some particular time t. It will then be printed by B when it considers the pair (t, w). By assumption, any word w produced by B is accepted by A, and hence is a word of L.

Direction $\Leftarrow$. If there is a Turing machine B which enumerates all the words of the language L, then one can construct a Turing machine A, which, given some word w, simulates B, and every time that B produces a word, compares this word to the word w. If there are equal, then A stops and accepts. Otherwise, A continues.

By construction, on some input w, A halts and accepts if w is among the words enumerated by B, that is to say if $w \in L$. If w is not among these words, by hypothesis, $w \notin L$, and hence by construction, A will not accept w. □

9.3.7 Closure properties

Theorem 9.6 *The set of semi-decidable languages is closed by union and by intersection: In other words, if L_1 and L_2 are semi-decidable, then $L_1 \cup L_2$ and $L_1 \cap L_2$ are.*

Proof: Suppose that L_1 is $L(A_1)$ for some Turing machine A_1 and L_2 is $L(A_2)$ for some Turing machine A_2. Then $L_1 \cup L_2$ is $L(A)$ for the Turing machine A which simulates in parallel A_1 and A_2 and which halts and accepts as soon as one of the Turing machine A_1 or A_2 halts and accepts: See Figure 9.5.

$L_1 \cap L_2$ is $L(A)$ for the Turing machine A which simulates in parallel A_1 and A_2 and which halts and accepts as soon as both Turing machines A_1 and A_2 halt and accept.

□

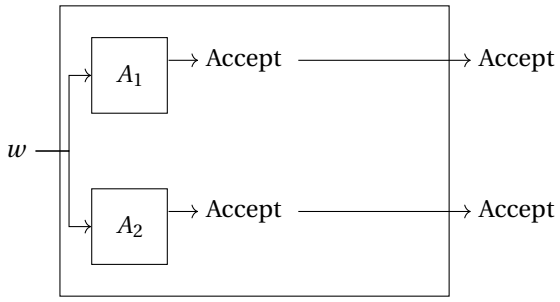


Figure 9.5: Construction of a Turing machine accepting $L_1 \cup L_2$.

Theorem 9.7 *The set of decidable languages is closed by union, intersection and complement: In other words, if L_1 and L_2 are decidable, then $L_1 \cup L_2$, $L_1 \cap L_2$, and L_1^c are.*

Proof: We have already used the closure by complement. Indeed, by exchanging the accepting state and the rejecting state of the Turing machine, the complement of a decidable set is also decidable: See Figure 9.4.

It remains to prove that with the hypotheses, the languages $L_1 \cup L_2$ and $L_1 \cap L_2$ are decidable. But this is clear from the previous theorem and the fact that a set is decidable if and only if it is semi-decidable as well as its complement, by using Morgan's law (the complement of a union is the intersection of the complement, and symmetrically) and the fact that complements of decidable languages are decidable. □

In particular, we deduce:

Definition 9.11 ($\overline{\text{L}_{\text{univ}}}'$)

Input: *The encoding $\langle M \rangle$ of a Turing machine M and a word w.*

Answer: *Decide if the machine M does not accept the word w.*

Corollary 9.4 *The problem $\overline{\text{L}_{\text{univ}}}'$ is undecidable.*

Proof: $\overline{\text{L}_{\text{univ}}}$ is the union of $\overline{\text{L}_{\text{univ}}}'$ and of the complement of ENCODING. If $\overline{\text{L}_{\text{univ}}}'$ were decidable, then $\overline{\text{L}_{\text{univ}}}$ would be decidable. □

9.4 Other undecidable problems

Having shown a first result to be undecidable, we will now obtain other undecidable languages.

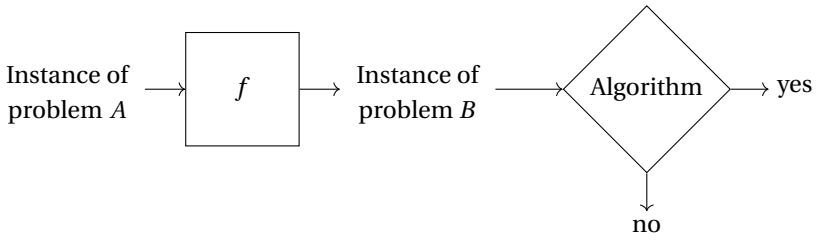


Figure 9.6: Reduction of problem A to problem B. If one can solve the problem B, then one can solve the problem A. The problem A is consequently at least as easy as problem B, denoted by $A \leq_m B$.

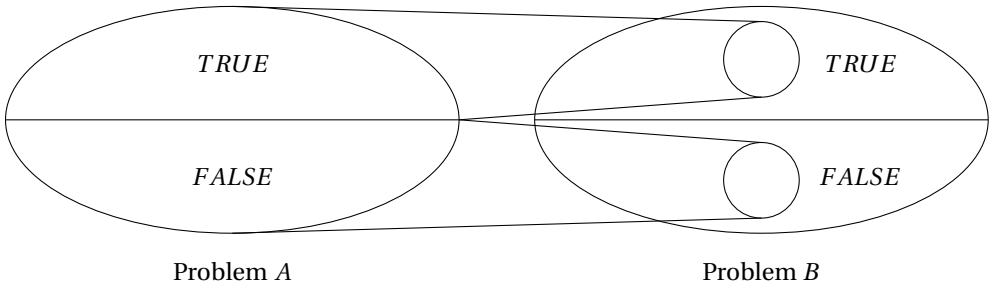


Figure 9.7: A reduction transforms the positive instances to positive instance,s and negative instances to negative instances.

9.4.1 Reductions

We know two undecidable problems, L_{univ} and its complement. Our aim is now to obtain some more. We will also show how to compare problems. To this end, we will introduce the notion of *reduction*.

First, we generalize the notion of *computable* from languages and decision problems to functions.

Definition 9.12 (Computable function) *Let Σ and Σ' be two alphabets. A (total) function $f : \Sigma^* \to \Sigma'^*$ is* computable *if there exists a Turing machine A working on alphabet $\Sigma \cup \Sigma'$, such that for all words w, A on input w, halts and accepts, with $f(w)$ written on its tape at the moment it halts.*

One can see that the composition of two computable functions is computable.

This provides a way to introduce a notion of reduction between problems: The idea is that if A reduces to B, then problem A is as least as easy as problem B, or, if one prefers, the problem B is at least as hard than problem A. See Figure 9.6 and Figure 9.7.

Definition 9.13 (Reduction) *Let A and B two problems of respective alphabet Σ_A and Σ_B. A* reduction from A to B *is a computable function $f : \Sigma_A^* \to \Sigma_B^*$ such that $w \in A$ if and only if $f(w) \in B$. We write $A \leq_m B$ when A reduces to B, i.e., there is a reduction from A to B.*

Reductions, as defined above, behave as we would like: A problem is at least as easy (and hard) as itself, and the relation "is at least as easy as" is transitive. In other words:

Theorem 9.8 *$\leq_m$ is a preorder:*

1. *$L \leq_m L$;*
2. *$L_1 \leq_m L_2$, $L_2 \leq_m L_3$ implies $L_1 \leq_m L_3$.*

Proof: Consider the identity function as function f for the first point.

For the second, suppose that $L_1 \leq_m L_2$ via the reduction f, and that $L_2 \leq_m L_3$ via the reduction g. We have $x \in L_1$ if and only if $g(f(x)) \in L_2$. It is then sufficient to see that $g \circ f$, being the composition of two computable functions is computable. □

Remark 9.10 *The reduction relation $\leq_m$ is not an order, since $L_1 \leq_m L_2$, $L_2 \leq_m L_1$ does not imply $L_1 = L_2$. It is actually rather natural to introduce the following concept: Two problems L_1 and L_2 are equivalent, denoted $L_1 \equiv L_2$, if $L_1 \leq_m L_2$ and $L_2 \leq_m L_1$.*

Intuitively, if a problem is at least as easy as a decidable problem, then it is decidable. We show this here formally:

Proposition 9.2 (Reduction) *If $A \leq_m B$, and if B is decidable then A is decidable .*

Proof: A is decided by the Turing machine that, on a given input w, computes $f(w)$, and then simulate the Turing machine that decides B on input $f(w)$. Since we have $w \in A$ if and only if $f(w) \in B$, the Turing machine behaves correctly. □

Proposition 9.3 (Reduction) *If $A \leq_m B$, and if A is undecidable, then B is undecidable.*

Proof: This is the contrapositive of the previous proposition. □

9.4.2 Some other undecidable problems

Proposition 9.3 provides a way to obtain the undecidability proof for many other problems.

As a first example, it is not possible to determine algorithmically if a given Turing machine halts.

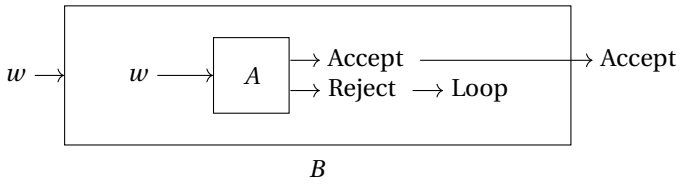


Figure 9.8: Illustration of the Turing machine used in the proof of Proposition 9.4.

Definition 9.14 (*Halting Problem*)

Input: *The encoding $\langle M \rangle$ of a Turing machine M and some input w.*

Answer: *Decide if M halts on input w.*

Proposition 9.4 *The problem Halting Problem is undecidable.*

Proof: We construct a reduction from $\mathrm{L}_{\mathrm{univ}}$ to the halting problem: For every pair $\langle \langle A \rangle, w \rangle$, we consider the Turing machine B defined in the following way (see Figure 9.8):

- B takes as input a word w;
- B simulates A on w;
- If A accepts w, then B accepts. If A rejects w, then B loops (possibly B simulates A forever, if A never halts).

The function f that maps $\langle \langle A \rangle, w \rangle$ to $\langle \langle B \rangle, w \rangle$ is computable. Furthermore, we have $\langle \langle A \rangle, w \rangle \in \mathrm{L}_{\mathrm{univ}}$ if and only if B halts on input w, that is to say $\langle \langle B \rangle, w \rangle \in Halting\ Problem$. □

As another example, it is not possible to decide algorithmically (that is to say by a Turing machine, i.e. a program) if a Turing machine accepts at least one input:

Definition 9.15 ($L_\emptyset$)

Input: *The encoding $\langle M \rangle$ of a Turing machine M.*

Answer: *Decide if $L(M) \neq \emptyset$.*

Proposition 9.5 *The problem $L_\emptyset$ is undecidable.*

Proof: We design a reduction from $\mathrm{L}_{\mathrm{univ}}$ to $L_\emptyset$: For any pair $\langle \langle A \rangle, w \rangle$, we consider the Turing machine A_w defined as follows (see Figure 9.9):

- A_w takes as input a word u;

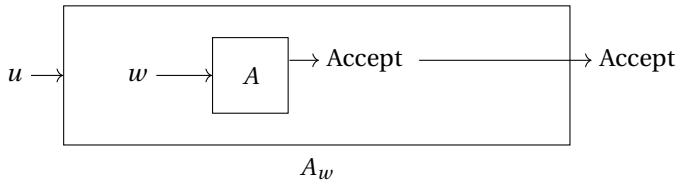


Figure 9.9: Illustration of the Turing machine used in the proof of Proposition 9.5.

- A_w simulates A on w;
- If A accepts w, then A_w accepts.

The function f that maps $\langle\langle A\rangle, w\rangle$ to $\langle A_w\rangle$ is indeed computable. Furthermore, we have $\langle\langle A\rangle, w\rangle \in \mathrm{L}_{\mathrm{univ}}$ if and only if $L(A_w) \neq \emptyset$, that is to say $\langle A_w\rangle \in L_\emptyset$: Indeed, A_w accepts either all the words (and hence the associated language is not empty) if A accepts w, or accepts no word otherwise (and hence the associated language is empty). □

Definition 9.16 ($L_{\neq}$)

Input: *The encoding $\langle A\rangle$ of a Turing machine A and the encoding of $\langle A'\rangle$ of a Turing machine A'.*

Answer: *Determine if $L(A) \neq L(A')$.*

Proposition 9.6 *The problem $L_{\neq}$ is undecidable.*

Proof:
We design a reduction from $L_\emptyset$ to $L_{\neq}$. We consider a Turing machine B that accepts the empty language: Take for example a Turing machine B that enters immediately in a non-terminating loop. The function f that maps $\langle A\rangle$ to the pair $\langle A, B\rangle$ is indeed computable. Furthermore, we have $\langle A\rangle \in L_\emptyset$ if and only if $L(A) \neq \emptyset$ if and only if $\langle A, B\rangle \in L_{\neq}$. □

9.4.3 Rice's theorem

The two previous results can be seen as the consequence of a very general statement that asserts that any non-trivial property of algorithms is undecidable.

A property of semi-decidable languages is said to be non-trivial if it is not always true or always false for Turing machines: That is to say, there is at least a Turing machine M_1 such that $L(M_1)$ satisfies P and a Turing machine M_2 such that $L(M_2)$ does not satisfy P.

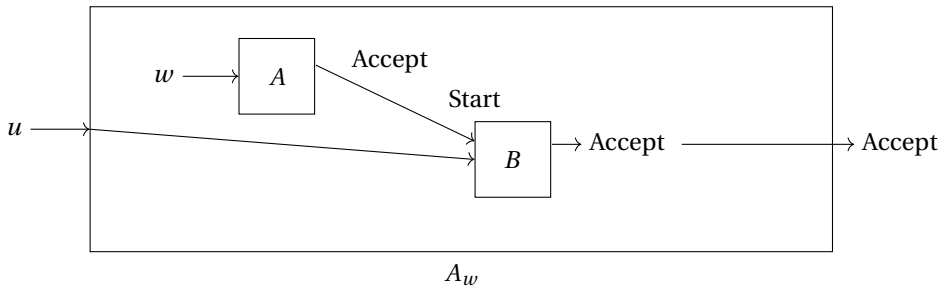


Figure 9.10: Illustration of the Turing machine used in the proof of Rice Theorem.

Theorem 9.9 (Rice's theorem) *Any non-trivial property of semi-decidable languages is undecidable.*

Then the following decision problem L_P:

***Input**: The encoding $\langle M \rangle$ of a Turing machine M;*

***Answer**: Decide if $L(M)$ satisfies property P;*

is undecidable.

Remark 9.11 *Observe that if a property P is trivial in the above sense, L_P is trivially decidable: Construct a Turing machine that does not even read its input and accepts (respectively: rejects).*

Proof: We need to prove that the decision problem L_P is undecidable.

Replacing P by its negation if needed, one can assume that the empty word does not satisfy the property P (proving the undecidability of L_p is equivalent to proving the undecidability of its complement). Since P is non-trivial, there exists at least one Turing machine B whose accepted language $L(B)$ satisfies P.

We design a reduction from L_{univ} to the language L_P. Given a pair $\langle \langle A \rangle, w \rangle$, we consider a Turing machine A_w defined as follows (see Figure 9.10):

- A_w takes as input a word u;
- On word u, A_w simulates A on word w;
- If A accepts w, then A_w simulates B on the word u: A_w accepts if and only if B accepts u.

In other words, A_w accepts, if and only if A accepts w and if B accepts u. If w is accepted by A, then $L(A_w)$ equals $L(B)$, and hence satisfies property P. If w is not accepted by A, then $L(A_w) = \emptyset$, and hence does not satisfy property P.

The function f that maps $\langle \langle A \rangle, w \rangle$ to $\langle A_w \rangle$ is obviously computable. □

Exercise 9.2 *(solution on page 235) Prove that the set of encodings of Turing machines which accepts all the words which are palindromes (possibly accepting other words) is undecidable.*

9.4.4 The drama of verification

From Rice's theorem, we directly get the following:

Corollary 9.5 *It is not possible to design an algorithm that takes as input a program, its specification, and that determines if the program satisfies the specification.*

The above is true, even if the specification is fixed to a property P (as soon as the property P is not trivial) by Rice's theorem.

From what we have seen before, this turns out to be true even for very rudimentary systems. For example:

Corollary 9.6 *It is not possible to design an algorithm that takes as input the description of a system, its specification, and that determines if the system satisfies its specification.*

The above is true, even if the specification is fixed to a property P (as soon as the property P is not trivial), and even for systems as simple as two counter machines by Rice's theorem, and the simulation results of the previous chapter.

9.4.5 Notion of completeness

We now introduce a notion of completeness.

Definition 9.17 (CE-completeness) *A problem A is called* CE*-complete, if:*

1. *it is computably enumerable;*
2. *for every other computably enumerable problem B we have $B \leq_m A$.*

In other words, a CE-complete problem is maximal for $\leq_m$ among the problems of class CE.

Theorem 9.10 *The problem* $\mathrm{L}_{\mathrm{univ}}$ *is* CE*-complete.*

Proof: $\mathrm{L}_{\mathrm{univ}}$ is semi-decidable. Now, let L be a semi-decidable language. By definition, there exists a Turing machine A which recognizes L. Consider the function f which maps w to the word $\langle\langle A\rangle, w\rangle$. We have that $w \in L$ if and only if $f(w) \in \mathrm{L}_{\mathrm{univ}}$, and hence we obtain $L \leq_m \mathrm{L}_{\mathrm{univ}}$. □

9.5 Natural undecidable problems

One can object that the previous problems, relative to algorithms are "artificial" in the sense that they are talking about properties of algorithms, the algorithms have been in turn defined by the theory of computability.

It is difficult to define formally what one would like to call a "*natural problem*", but one can say that a problem that has been discussed before the invention of computability theory is (more) natural.

9.5.1 Hilbert's tenth problem

This is clearly the case of Hilbert's tenth problem, identified by David Hilbert as one of the most interesting problems for the 20th century in 1900: Can we determine if a given polynomial equation with integer coefficients has an integer solution?

Definition 9.18 ($Hilbert's\ 10th\ problem$)

Input: *A polynomial* $P \in \mathbb{N}[X_1, \cdots, X_n]$ *with integer coefficients.*

Answer: *Decide if* P *has an integer root*

Theorem 9.11 *The problem* $Hilbert's\ 10th\ problem$ *is undecidable.*

The proof of this result, due to Matiyasevich [Matiyasevich, 1970] (extending statements from Davis, Putnam and Robinson) is beyond the ambition of this document.

9.5.2 The Post correspondence problem

The proof of the undecidability of Post correspondence problem is easier, even if we will not give it here. One can consider this problem as a *natural problem*, in the sense that it is not making a (direct) reference to the notion of algorithms, or to Turing machines.

Definition 9.19 (Post correspondence problem)

Input: *A sequence* $(u_1, v_1), \cdots, (u_n, v_n)$ *of pairs of words on alphabet* Σ.

Answer: *Decide if this sequence admits a* correspondence*, that is to say a sequence of indexes* $i_1, i_2, \cdots, i_m$ *of* $\{1, 2, \cdots, n\}$ *such that*

$$u_{i_1} u_{i_2} \cdots u_{i_m} = v_{i_1} v_{i_2} \cdots v_{i_m}.$$

Theorem 9.12 *The problem* Post correspondence problem *is undecidable.*

9.5.3 Decidability/Undecidability of theories in logic

We have already presented in Chapter 6, the axioms of Robinson arithmetic and the axioms of Peano: One expects these axioms to be true on the integers, that is to say in the *standard model of the integers* where the base set is the integers, and where + is interpreted by addition, $*$ by multiplication, and $s(x)$ is interpreted by the function that maps x to $x+1$.

Given some closed formula F on the signature that contains the symbols of arithmetic, F is either true or false on the the integers (that is to say in the standard model of the integers). Let's call *theory!of the arithmetic* the set $Th(\mathbb{N})$ of closed formulas F which are true on the integers.

The constructions of the previous chapter proves the following result:

Theorem 9.13 *$Th(\mathbb{N})$ is not decidable.*

Proof: We prove in the following chapter that $Th(\mathbb{N})$ is not computably enumerable. It is then sufficient to observe that a decidable set is computably enumerable to obtain a contradiction with assuming $Th(\mathbb{N})$ decidable. □

We can prove (we will not do it) that if one considers the set of formulas written without using the multiplication symbol, then the associated theory is decidable.

Theorem 9.14 *One can decide if a closed formula F on signature $(\mathbf{0}, s, +, =)$ (i.e. the one from Peano without the multiplication symbol) is satisfied on the integers.*

We also obtain the following results:

Theorem 9.15 *Let F be a closed formula on the signature of Peano axioms. The decision problem that consists, given F, to determine whether it can be proved from the axioms of Peano is undecidable..*

Proof: Given some pair $\langle\langle M\rangle, w\rangle$, where M is a machine and w a word, we show in the next chapter how to produce some closed formula γ on the signature of arithmetic such that

$$\langle\langle M\rangle, w\rangle \in \overline{\mathrm{HP}} \Leftrightarrow \gamma \in Th(\mathbb{N}),$$

where $\overline{\mathrm{HP}}$ is the complement of the halting problem of Turing machines.

But, doing so, one can check that the reasoning that is done for that can be formalized with Peano arithmetic and can be deduced from Peano axioms.

We consequently have actually $\langle\langle M\rangle, w\rangle \in \overline{\mathrm{HP}}$ if and only if γ can be proved from Peano axioms.

This provides a reduction from the complement of the universal problem of Turing machines to our problem: The transformation that maps $\langle\langle M\rangle, w\rangle$ to γ is indeed easily computable. Our problem is hence undecidable, since the first is. □

One can prove.

Theorem 9.16 *One can decide if some closed formula F on the signature $(\mathbf{0}, s, +, =)$ (i.e. the one from Peano without the multiplication symbol) is provable from Peano axioms.*

9.6 Fixpoint problems

The results of this section are very subtle, but extremely powerful.

Let us start by a simple version, that will help understanding the proofs.

Proposition 9.7 *There exists a Turing machine A^* that produces its own algorithm: It outputs $\langle A^* \rangle$.*

In other words, it is possible for a program to write its own code.

This is true in any programming language which is equivalent to Turing machines.

In *UNIX* shell for example, the following program

```
x='y=‘echo . | tr . "\47" ‘; echo "x=$y$x$y;$x"'; y=‘echo . | tr .
 "\47"‘; echo "x=$y$x$y;$x"
```

produces

```
x='y=‘echo . | tr . "\47" ‘; echo "x=$y$x$y;$x"'; y=‘echo . | tr .
 "\47"‘; echo "x=$y$x$y;$x"
```

which is a command, once executed, prints its own code.

Such programs are sometimes called *quines*, in honor of philosopher Willard van Orman Quine, who discussed the existence of self-reproducing programs.

Proof: We consider Turing machines which halt on every input. For two such machines A and A', we write AA' for the Turing machine that is obtained by composing in a sequential manner A and A'. Formally, AA' is the Turing machine that runs first the program of A and then when A halts with its tape set to w, runs the program of A' on the input w.

We construct the following machines:

1. Given a word w, the Turing machine $Print_w$ halts with the result w;
2. For a given input w of the form $w = \langle X \rangle$, where X is a Turing machine, the Turing machine B produces as output the encoding of the Turing machine $Print_w X$, that is to say the encoding of the Turing machine obtained by composing $Print_w$ and X.

We consider then the Turing machine A^* given by $Print_{\langle B \rangle} B$, that is to say the sequential composition of the machines $Print_{\langle B \rangle}$ and B.

Let us unfold the result of this machine: The Turing machine $Print_{\langle B \rangle}$ produces as output $\langle B \rangle$. The sequential composition with B produces then the encoding of $Print_{\langle B \rangle} B$, which is indeed the encoding of the Turing machine A^*. □

The recursion theorem allows self references in programing languages. Its proof consists of extending the ideas behind the proof of the previous result.

Theorem 9.17 (Recursion theorem) *Let $t : M^* \times M^* \to M^*$ be a computable function. Then there exists a Turing machine R which computes a function $r : M^* \to M^*$ such that for all words w*

$$r(w) = t(\langle R\rangle, w).$$

Tje statement of the Recursion Theorem is rather technical, but its use is simple. To obtain a Turing machine that obtains its own description, and use it to compute, we simply need a Turing machine T which computes some function t as in the statement, that takes as input some supplementary entry which contains the description of the Turing machine. Then the recursion theorem produces a new machine R which operates as T but with the description of $\langle R\rangle$ encoded in its code.

Proof: We use basically the same idea as before. Let T be a Turing machine that computes a function t: T takes as input a pair $\langle u, w\rangle$ and produces as output $t(u, w)$.

We then consider the following machines:

1. Given a word w, the Turing machine $Print_w$ takes as input a word u and halts with the result $\langle w, u\rangle$;
2. For a given input w' of the form $\langle\langle X\rangle, w\rangle$, the Turing machine B:
   (a) computes $\langle\langle Print_{\langle X\rangle} X\rangle, w\rangle$, where $Print_{\langle X\rangle} X$ denotes the Turing machine which composes $Print_{\langle X\rangle}$ with X;
   (b) and then gives the control to the Turing machine T.

We consider then the Turing machine R given by $Print_{\langle B\rangle} B$, that is to say the Turing machine obtained by composing $Print_{\langle B\rangle}$ with B.

Let us unfold the result $r(w)$ of this machine R on an input w: On the input w, the Turing machine $Print_{\langle B\rangle}$ produces as output $\langle\langle B\rangle, w\rangle$. The composition with B produces then the encoding of $\langle\langle Print_{\langle B\rangle} B\rangle, w\rangle$, and gives the control to T. The latter produces then $t(\langle Print_{\langle B\rangle} B\rangle, w) = t(\langle R\rangle, w) = r(w)$. □

We directly obtain the followign result:

Theorem 9.18 (Kleene fixed point theorem) *Let f be a computable function that to every word $\langle A\rangle$ encoding a Turing machine associates a word $\langle A'\rangle = f(\langle A\rangle)$ encoding a Turing machine. For conciseness, write $A' = f(A)$ in that case.*
Then there exists a Turing machine A^* such that $L(A^*) = L(f(A^*))$.

Proof: Consider a function $t : M^* \times M^* \to M^*$ such that $t(\langle A\rangle, x)$ is the result of the simulation of the Turing machine $f(A)$ on input x. By the previous theorem, there exists a Turing machine R which computes a function r such that $r(w) = t(\langle R\rangle, w)$. By construction $A^* = R$ and $f(A^*) = f(R)$ have hence the same value on w for all w. □

Remark 9.12 *One can interpret the previous results in relation with computer viruses. Indeed a virus is a program that aims at propagating, that is to say at self-reproducing, without being detected. The principle of the previous proof of*

the recursion theorem is a way to self-reproduce, by duplicating its own code.

9.7 A few remarks

9.7.1 Computing on other domains

We have introduced the notion of decidable sets, computably enumerable, for the subsets of Σ^*, and the notion of (total) computable function $f : \Sigma^* \to \Sigma^*$.

One may want to work on other domains than words over some given alphabet, for example on the integers. In that case, on can simply consider encodings of integers by their binary expansion to come back to the case of a word on the alphabet $\{0,1\}$.

Remark 9.13 *One could also encode an integer for example in unary, i.e., n by a^n for a letter a on some alphabet Σ with $a \in \Sigma$. This would not change the notion of computable function.*

In the general case, to work on some domain E, one fixes some encoding of the elements of E in some alphabet Σ: One then says for example that a subset $S \subset E$ is computably enumerable (respectively decidable) if the subset of encodings of elements of E is computably enumerable (resp. decidable).

Similarly, a (total) function $f : E \to F$ is called computable if the function from the encoding $e \in E$ to the encoding of $f(e) \in F$ is computable.

Example 9.2 *We can encode $\vec{n} = (n_1, \ldots, n_k) \in \mathbb{N}^k$ by*

$$\langle \vec{n} \rangle = a^{n_1+1} b a^{n_2+1} b \cdots a^{n_k+1}$$

on the alphabet $\Sigma = \{a, b\}$. A function $f : \mathbb{N}^k \to \mathbb{N}$ is called computable if it is computable with respect to this encoding.

The obtained notions of computable functions, semi-decidability, etc. do not depend on the encodings, for the usual encodings (actually technically as soon as one can go from one encoding to the other encoding in a computational way, a property that holds for all "natural" encodings of objects, and in particular for all encodings considered in this document).

9.7.2 Algebraic vision of computability

The notions of computability are sometimes introduced in an algebraic manner, by talking about functions over the integers.

In particular, one can introduce the notion of computable partial function, which extends the notion of computable functions to the case of non-total functions.

Definition 9.20 (Partial computable function) *Let $f : E \to F$ be a function, possibly partial.*

The function $f : E \to F$ is computable *if there exists a Turing machine A such that for any word w encoding an element $e \in E$ in the domain of f, the machine A on the input w, halts and accepts with the encoding of $f(e)$ written on its tape at the moment when it halts.*

Of course, this notion matches to the previous notion for the case of total functions.

One can characterize in a purely algebraic way the notion of computable functions:

Definition 9.21 (Recursive functions) *A (possibly partial) function $f : \mathbb{N}^n \to \mathbb{N}$ is* recursive *if it is either the constant* 0*, or one of the functions:*

- $\mathsf{Zero} : x \mapsto 0$ *the function 0;*
- $\mathsf{Succ} : x \mapsto x+1$ *the successor function;*
- $\mathsf{Proj}_n^i : (x_1, \ldots, x_n) \mapsto x_i$ *the projection functions for* $1 \leq i \leq n$*;*
- $\mathsf{Comp}_m(g, h_1, \ldots, h_m) : (x_1, \ldots, x_n) \mapsto g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$ *the composition of the recursive functions $g, h_1, \ldots, h_m$;*
- $\mathsf{Rec}(g, h)$ *the function defined by recurrence as*

$$\begin{cases} f(0, x_2, \ldots, x_n) = g(x_2, \ldots, x_n), \\ f(x_1 + 1, x_2, \ldots, x_n) = h(f(x_1, \ldots, x_n), x_1, \ldots, x_n), \end{cases}$$

  where g and h are recursive.

- $\mathsf{Min}(g)$ *the function that to $(x_2, \ldots, x_n)$ associates the least $y \in \mathbb{N}$ such that*

$$g(y, x_2, \ldots, x_n) = 1$$

  if there is one (and which is not defined otherwise) where g is recursive.

A primitive recursive *function is a function that can be defined without using the schema* Min.

One can prove the following result:

Theorem 9.19 *A function $f : \mathbb{N}^n \to \mathbb{N}$ is recursive if and only if it is computable by some Turing machine.*

The notion of decidability or semi-decidability can then also be defined in an algebraic way:

Theorem 9.20 *A subset $S \subset \mathbb{N}$ is decidable if the characteristic function of S, i.e., the (total) function $\chi : n \mapsto \begin{cases} 1 & \text{if } n \in S \\ 0 & \text{if } n \notin S \end{cases}$ is recursive.*

Theorem 9.21 *A subset $S \subset \mathbb{N}$ is semi-decidable if the (partial) function $n \mapsto \begin{cases} 1 & \text{if } n \in S \\ \text{undefined} & \text{if } n \notin S \end{cases}$ is recursive.*

Exercise 9.3 *(solution on page 235) Prove these theorems.*

9.8 Exercises

***Exercise 9.1** *(solution on page 235) [Generalized halting problem] Let A be a decidable subset of the set of encodings of Turing machines, such that all machines of A always halt.*

Then A is incomplete: There exists some (unary) total function $f : \mathbb{N} \to \mathbb{N}$ computable that is not represented by any Turing machine of A.

Explain why this result implies the halting problem.

Exercise 9.4 *(solution on page 235) Let $E \subset \mathbb{N}$ be a computably enumerable set enumerated by some computable function f strictly increasing. Prove that E is decidable.*

Exercise 9.5 *(solution on page 236) Deduce that any infinite computably enumerable set of $\mathbb{N}$ contains some infinite decidable subset.*

Exercise 9.6 *(solution on page 236) Let $E \subset \mathbb{N}$ be a decidable set. Prove that it can be enumerated by some computable function f strictly increasing.*

Exercise 9.7 *(solution on page 236) Let $A \subset \mathbb{N}^2$ be a decidable set of pairs of integers.*

Write $\exists A$ for the (first) projection of A, that is to say the subset of $\mathbb{N}$ defined by

$$\exists A = \{x | \exists y \in \mathbb{N} \text{ such that } (x,y) \in A\}.$$

1. *Prove that the projection of a decidable set is computably enumerable.*
2. *Prove that any computably enumerable set is the projection of some decidable set.*

Exercise 9.8 *A real number a is called* computable *if there exist computable functions F and G from $\mathbb{N}$ to $\mathbb{N}$ such that for any $n > 0$ we have $G(n) > 0$ and*

$$\left| |a| - \frac{F(n)}{G(n)} \right| \leq \frac{1}{n}.$$

1. *Prove that any rational number is computable.*
2. *Prove that $\sqrt{2}$, π, e are computable.*
3. *Prove that any real number $0 < a < 1$ is computable if and only there exists a computable radix 10 expansion of a, that is to say a computable function $H : \mathbb{N} \to \mathbb{N}$ such that for all $n > 0$ we have $0 \leq H(n) \leq 9$ and*
$$|a| = \sum_{n=0}^{\infty} \frac{H(n)}{10^n}.$$
4. *Prove that the set of computable reals is a countable sub-field of $\mathbb{R}$, such that any polynomial of odd degree has a root.*
5. *Give an example of a non-computable real.*

Exercise 9.9 *A "useless" internal state of a Turing machine is a state $q \in Q$ in which the machine never enters on any input. Formulate the problem to decide whether a given Turing machine has a useless state as a decision problem, and prove that it is undecidable.*

Exercise 9.10 *(solution on page 236) Consider the following decision problem: A Turing machine A is given, and one wants to determine*

1. *if $L(A)$ contains at least two distinct words*
2. *if $L(A)$ is empty*

Is the problem decidable? semi-decidable?

9.9 Bibliographic notes

Suggested readings To go further on the notions of this chapter, we suggest reading the books [Sipser, 1997] and[Hopcroft & Ullman, 2000] in English, or [Wolper, 2001], [Stern, 1994] [Carton, 2008] in French.

The book [Sipser, 1997] is very pedagogical.

Bibliography This chapter contains some standard results from computability. We used essentially their presentation in [Wolper, 2001], [Carton, 2008], [Jones, 1997], [Kozen, 1997], [Hopcroft & Ullman, 2000], as well as in [Sipser, 1997].

Chapter 10

Incompleteness of arithmetic

In 1930, Kurt Gödel proved a result whose philosophical consequences in science started a revolution: He proved that any sufficiently expressive theory to capture the arithmetic reasoning's is necessarily incomplete, that is to say that there exists some statements that cannot be proved, and whose negation can nor be proved.

This theorem is largely considered as one of the greatest achievements of the logic in the 20th century.

With all the previous ingredients, we are in position to understand this theorem, and to provide a full proof. This is the objective of this chapter. Actually, we will propose a proof due to Turing. We will only mention the proof from Gödel, that allows to say more.

10.1 Theory of Arithmetic

10.1.1 Peano axioms

The question we are focusing on now is to try to axiomatise the arithmetic, that is to say the properties of the inters.

We have already presented in Chapter 6, the axioms of the arithmetic from Robinson and the axioms from Peano: One expects that all these axioms are satisfied on the integers, that is to say in the *standard model of the integers* where the base set is the integers, and where + is interpreted by addition, $*$ by multiplication and $s(x)$ by successor function $x \mapsto x+1$.

In other words, one expects that these axioms have at least one model: *The standard model of the integers*.

Given some closed formula F on the signature containing these symbols, F is either true or false on the integers (that is to say in the standard model of the integers). Call *theory!of the arithmetic* the set $Th(\mathbb{N})$ of closed formula F that are true over the integers.

10.1.2 Some concepts from arithmetic

It is possible to prove that numerous concepts from number theory can be defined perfectly from these axioms.

For example, we can express the following concepts:

- INTDIV(x, y, q, r) defined as "q is the quotient and r the remainder of the euclidean division of x by y".

  Indeed, this can be written as formula:

  $$(x = q * y + r \wedge r < y).$$

- DIV(y, x) defined as "y divides x".

  Indeed, this can be written:

  $$\exists q \ \text{INTDIV}(x, y, q, 0).$$

- EVEN(x) defined as "x is even". Indeed, this can be written:

  $$\text{DIV}(2, x).$$

- ODD(x) defined as "x is odd. Indeed, this can be written:

  $$\neg \text{EVEN}(x).$$

- PRIME(x) defined as "x is prime". Indeed, this can be written:

  $$(x \geq 2 \wedge \forall y(\text{DIV}(y, x) \Rightarrow (y = 1 \vee y = x))).$$

- POWER$_2(x)$ defined as "x is a power of 2". Indeed, this can be written:

  $$\forall y((\text{DIV}(y, x) \wedge \text{PRIME}(y)) \Rightarrow y = 2).$$

10.1.3 The possibility of talking of bits of an integer

One can also write formulas like BIT(x, y) that means that "y is a power of 2, say 2^k, and the kth bit of the binary representation of integer x is 1"'.

This is more subtle, but possible. Indeed, this can be written:

$$(\text{POWER}_2(y) \wedge \forall q \forall r(\text{INTDIV}(x, y, q, r) \Rightarrow \text{ODD}(q))).$$

The idea is that if y satisfies the formula, then y is a power of 2, and hence in binary is written 2^k for some integer k. By dividing x by y, the remainder of the division r will be the k less significant bits of x and the quotient q the other bits of x since we have $x = q * y + r$. By testing if q is odd, one "reads" the $k+1$th bit of x, hence the bit corresponding to the bit set to 1 in the integer y encoding this position.

10.1.4 Principle of the proof from Gödel

Kurt Gödel proved the incompleteness theorem by building in any reasonable proof system a formula ϕ from arithmetic that states its own non-provability in the system:

$$\phi \text{ is true } \Leftrightarrow \phi \text{ is not provable.} \tag{10.1}$$

Every reasonable proof system is valid, and hence one must have.

$$\psi \text{ provable } \Rightarrow \psi \text{ is true.} \tag{10.2}$$

Then ϕ must be true, since otherwise.

$$\begin{array}{lll} \phi \text{ is wrong} & \Rightarrow \phi \text{ is provable.} & \text{(par (10.1))} \\ & \Rightarrow \phi \text{ is true.} & \text{(by (10.2))} \end{array}$$

The construction of ϕ by itself is instructive, as it captures the notion of self-reference.

We will come back to the construction from Gödel.

10.2 Incompleteness theorem

10.2.1 Principle of the proof from Turing

We will prove the incompleteness theorem by using an approach that allows to get the main consequences of the theorem, and which is due to Alan Turing.

This approach is simpler, and mainly, we have now all the ingredients to do full formal proof, by using the arguments from computability theory.

The idea is to convince ourselves that in Peano arithmetic, as well as in any "reasonable" proof system for the theory of arithmetic:

Theorem 10.1 *1. The set of theorems (closed formula that can be proved from the Peano axioms (or any "reasonable" axiomatisation of integers)) is computably enumerable.*

2. The set $Th(\mathbb{N})$ of closed formal F that are rue on the integers is not computably enumerable.

Consequently, the two sets cannot be the same, and the proof system cannot be complete. In other words:

Corollary 10.1 *There consequently exist some closed formula of $Th(\mathbb{N})$ that cannot be proved, or whose negation cannot be proved from Peano axioms or from any "reasonable" axiomatisation of the integers.*

This is the first incompleteness theorem from Kurt Gödel.

Exercise 10.1 *(solution on page 236) How to conciliate the previous incompleteness result (Gödel incompleteness theorem) with the completeness theorem (Gödel completeness theorem)?*

10.2.2 The easy direction

The set of theorems (closed formula provable from Peano axioms) is certainly computably enumerable: Whatever the proof method is (see for example those of Chapter 6), one can enumerate the theorems by enumeration the axioms and by applying in a systematic way all the reduction rules in all the possible manners, and produce as an output all the closed formula that can be derived.

This remains true as soon as we suppose that one can enumerate the axioms of the axiomatisation that one starts from. This is why, one can state that the set of theorems from any reasonable axiomatisation of the integers is recursively enumerable.

Remark 10.1 *In other words, if one wants a formal definition of "reasonable", one can take "computably enumerable".*

10.2.3 Crucial lemma

The crucial point is then to prove the following lemma.

Lemma 10.1 *The set $Th(\mathbb{N})$ is not computably enumerable.*

We prove this by reducing the complementary $\overline{\text{HP}}$ of the halting problem of Turing machines to this problem, i.e. by proving that $\overline{\text{HP}} \leq_m Th(\mathbb{N})$.

The theorem then follows from:

- $\overline{\text{HP}}$ is not recursively enumerable;
- and from the fact that $A \leq_m B$ and that A is not recursively enumerable, then consequently neither B.

Remember that the halting problem HP is the following problem: Given $\langle\langle M\rangle, w\rangle$, one must determine if the Turing machine M halts on input w.

Given $\langle\langle M\rangle, w\rangle$, we show how to produce a closed formula γ on the signature of arithmetic such that

$$\langle\langle M\rangle, w\rangle \in \overline{\text{HP}} \Leftrightarrow \gamma \in Th(\mathbb{N}).$$

In other words, given M and w, we must produce a closed formula γ on the signature of arithmetic that states that "the Turing machine M is not halting on input w".

This turns out to be possible since the language of arithmetic is sufficiently powerful to talk about Turing machines and the fact that they halt.

By using the principle of the previous formula BIT(y,x), we will construct a sequence of formula whose culminating point will be a formula VALCOMP$_{M,w}(y)$ that asserts that y is some integer that represents a sequence of configurations of M on input w: In other words, y represents a sequence of configurations $C_0, C_1, \cdots, C_t$ of M, encoded on a given alphabet Σ such that:

- C_0 is the initial configuration $C[w]$ of M on w;
- C_{i+1} is the successor configuration o f C_i, according to the transition function δ of the Turing machine M, for $i < t$;
- C_t is some accepting configuration

Once we will succeed to write the formula VALCOMP$_{M,w}(y)$, it will be easy to write that M is not halting on input x: The formula γ can be written as

$$\neg \exists y \text{ VALCOMP}_{M,w}(y).$$

This proves the reduction and will terminate the proof of previous lemma, and hence the proof of the theorem, recalling that $\overline{\text{HP}}$ is not recursively enumerable.

10.2.4 Construction of the formula

There only remain to provide the tedious details of the construction of the formula γ from M and w. Let us go.

Suppose that we encode the configurations of M on some finite alphabet Σ, that we will suppose without loss of generality of size p, with p some prime integer.

Every number has a unique representation in radix p: We will use this representation in radix p instead of the the binary representation to simplify the discussion.

Suppose that the initial configuration of M on $w = a_1 a_2 \cdots a_n$ is encoded by the integer whose digits in radix p are respectively $q_0 a_1 a_2 \cdots a_n$: We use the representation of the Definition 7.4 to represent configurations.

Consider that the blank symbol **B** is coded by digit k in radix p.

Let LEGAL the set of 6-tuples (a,b,c,d,e,f) of numbers in radix p that correspond to some legal windows for machine M: See the notion of legal window of Chapter 7. If one prefers, LEGAL is the set of 6-tuples (a,b,c,d,e,f) such that these three elements of Σ represented respectively by a,b and c appear consecutively in a configuration C_i, and if d,e,f appear consecutively in same locations in configuration C_{i+1}, then this is coherent with the transition function δ of Turing machine M.

We now define a few formulas:

- POWER$_p(x)$: "The number x is a power of p": Here p is a fixed primed number. This can be written:

$$\forall y((\text{DIV}(y,x) \wedge \text{PRIME}(y)) \Rightarrow y = p).$$

- LENGTH$_p(v,d)$: "The number d is a power of p that provides (an upper bound of) the length of v seen as a word on alphabet Σ with p letters. This can be written:
$$(\text{POWER}_p(d) \wedge v < d \wedge p * v \geq d).$$

- DIGIT$_p(v,K,b)$: "The 'kth digit of v written in radix p is b (where $K = p^k$)". This can be written:
$$\exists u \exists a (v = a + b * K + u * p * K \wedge a < K \wedge b < p).$$

- 3DIGIT$_p(v,K,b,c,d)$: "The 3 consecutive digits of v at position k are b, c and d (where $K = p^k$)". This can be written
$$\exists u \exists a (v = a + b * K + c * p * K + d * p * p * K + u * p * p * p * K \wedge a < K \wedge b < p \wedge c < p \wedge d < p).$$

- MATCH$_p(v,L,M)$: "The 3 digits of v at the position ℓ are respectively a, b and c and correspond to the 3 digits of v at the position m according to the transition function δ of the Turing machine (where $L = p^\ell$ and $M = p^m$). This can be written
$$\bigvee_{(a,b,c,d,e,f) \in \text{LEGAL}} 3\text{DIGIT}_p(v,L,a,b,c) \wedge 3\text{DIGIT}_p(v,M,d,e,f).$$

Remark 10.2 *We write obviously here,* $\bigwedge_{(a,b,c,d,e,f)\in\text{LEGAL}}$ *for the conjunction for each of the* 6*-tuples of* LEGAL.

- MOVE$_p(v,C,D)$: "The sequence v describe[1] a sequence of successive configurations of M of length c until d (where $C = p^c$ and $D = p^d$): All the pairs of sequences of 3-digits separated by exactly c positions in v are corresponding according to δ". This can be written as:
$$\forall y ((\text{POWER}_p(y) \wedge y * p * p * C < D) \Rightarrow \text{MATCH}_p(v,y,y * C)).$$

- START$_p(v,C)$: "The sequence v starts with the initial configuration of M on input $w = a_1 a_2 \cdots a_n$ with the addition of some blanks B until length c ($C = p^c$; $n, p^i, 0 \leq i \leq n$ are some fixed constants that are not depending of w)". This can be written:
$$\bigwedge_{i=0}^{n} \text{DIGIT}_p(v,p^i,a_i) \wedge p^n < C \wedge \forall y (\text{POWER}_p(y) \wedge p^n < y < C \Rightarrow \text{DIGIT}_p(v,y,B)).$$

- HALT$_p(v,D)$: "The sequence v has some accepting state somewhere". This can be written as:
$$\exists y (\text{POWER}_p(y) \wedge y < D \wedge \text{DIGIT}_p(v,y,q_a)).$$

[1] We see here a two-dimensional array as a unique word by putting the lines one after the other.

- VALCOMP$_{M,w}(v)$: "The sequence v is a valid computation of M on w". This can be written as:

  $$\exists c \exists d \ (\text{POWER}_p(c) \wedge c < d \wedge \text{LENGTH}_p(v,d) \wedge \text{START}_p(v,c) \wedge \text{MOVE}_p(v,c,d) \wedge \text{HALT}_p(v,d)).$$

- $\gamma_{M,w}$: "The machine M is not halting on w". This can be written as:

  $$\neg \exists v \ \text{VALCOMP}_{M,w}(v).$$

Our proof is over.

***Exercise 10.1** *(solution on page 236) The default of the previous constructions is that they allow to claim that there exists some true formulas which are not probable, but without providing any example of such a closed formula.*

Use the fix point theorem of computability (previous chapter) to provide explicitly a formula ψ which is not provable.

We will see later that the second theorem from Kurt Gödel allows to go further, and to prove that one can take ψ as the formula that asserts that the theory is not consistent.

(The solution of the previous formula produces a formula ψ whose practical interpretation is not clear).

10.3 The proof from Gödel

Kurt Gödel proved his incompleteness theorem in another manner, by constructing a closed formula that states its own non-provability. Write $\vdash$ for provable and $\models$ for true over the integers.

Suppose that we fix an encoding of formulas by the integers in any reasonable manner: If ϕ is a formula, then $\langle \phi \rangle$ denotes its encoding (an integer).

10.3.1 Fixpoint lemma

Here is a lemma that has been proved by Gödel, and that reads similar to the fixed point theorems already mentioned in previous chapter.

Lemma 10.2 (Gödel's fixpoint theorem) *For any formula $\psi(x)$ with free variable x, there is a closed formula τ such that*

$$\vdash \tau \Leftrightarrow \psi(\langle \tau \rangle),$$

i.e. the closed formula τ and $\psi(\langle \tau \rangle)$ are provably equivalent in Peano arithmetic.

Proof: Let x_0 be a fixed variable. One can certainly construct a formula SUBST(x,y,z) with free variables x,y,z which claims "the number z is the encoding of a formula

obtained by substituting the constant whose value is x in any occurrence of the free variable x_0 in the formula whose encoding is y".

For example, if $\phi(x_0)$ is a formula that contains a free occurrence of x_0, but no other free variable, the formula SUBST$(7, \langle\phi(x_0)\rangle, 312)$ is true if $312 = \langle\phi(7)\rangle$.

We will not provide the details of the construction of the formula SUBST, but the idea is to observe that this is indeed possible, by using for example the idea of relation BIT(x, y).

One considers now $\sigma(x)$ defined by $\forall y$ (SUBST$(x, x, y) \Rightarrow \psi(y)$), and τ defined by $\sigma(\langle\sigma(x_0)\rangle)$.

Then τ is the desired solution, since

$$\begin{array}{rcl} \tau & = & \sigma(\langle\sigma(x_0)\rangle) \\ & = & \forall y\ (\text{SUBST}(\langle\sigma(x_0)\rangle, \langle\sigma(x_0)\rangle, y) \Rightarrow \psi(y)) \\ & \Leftrightarrow & \forall y\ y = \langle\sigma(\langle\sigma(x_0)\rangle)\rangle \Rightarrow \psi(y) \\ & \Leftrightarrow & \forall y\ y = \langle\tau\rangle \Rightarrow \psi(y) \\ & \Leftrightarrow & \psi(\langle\tau\rangle) \end{array}$$

Of course, we have used here some informal equivalences, but the argument can indeed be fully formalized in Peano arithmetic. □

10.3.2 Arguments from Gödel

We observe now that the language of arithmetic is sufficiently expressive to talk about provability in Peano arithmetic. In particular, it is possible to code a sequence of formulas by an integer and to write a formula PROOF(x, y) that means that the sequence of formulas whose encoding is x is a proof of the formula whose encoding is y.

In other words, $\vdash$ PROOF$(\langle\pi\rangle, \langle\psi\rangle) \Leftrightarrow \pi$ is a proof of ψ in Peano arithmetic.

The provability in Peano arithmetic can hence be coded by the formula PROVABLE(y) defined by $\exists x$ PROOF(x, y).

Then for any closed formula ϕ,

$$\vdash \phi \Leftrightarrow \ \models PROVABLE(\langle\phi\rangle). \tag{10.3}$$

We then have

$$\vdash \phi \Leftrightarrow \ \vdash PROVABLE(\langle\phi\rangle). \tag{10.4}$$

The direction $\Rightarrow$ is true since if ϕ is provable then there is a proof π of ϕ. The arithmetic of Peano and the proof system allow to use this proof to prove ϕ (i.e. that PROOF$(\langle\pi\rangle, \langle\phi\rangle)$). The direction $\Leftarrow$ follows from 10.3 and from the validity of proof in Peano arithmetic.

Let us then use the point fix lemma to the closed formula $\neg$PROVABLE(x). We then obtain a closed formula ρ that states its own non-provability:

$$\vdash \rho \Leftrightarrow \neg \text{ PROVABLE}(\langle\rho\rangle),$$

in other words, ρ is true if and only if it is not provable in Peano arithmetic.

From the validity of proof in Peano arithmetic, we have

$$\models \rho \Leftrightarrow \neg \text{ PROVABLE}(\langle\rho\rangle). \tag{10.5}$$

Then formula ρ must be true, since otherwise then

$$\begin{array}{rcll} \models \neg\rho & \Rightarrow & \text{PROVABLE}(\langle\rho\rangle) & \text{(by 10.5)} \\ & \Rightarrow & \vdash \rho & \text{(by 10.3)} \\ & \Rightarrow & \models \rho & \text{(by validity of Peano arithmetic)} \end{array}$$

a contradiction.

So $\models \rho$. But now,

$$\begin{array}{rcll} \models \rho & \Rightarrow & \neg\,\text{PROVABLE}(\langle\rho\rangle) & \text{(by 10.5)} \\ & \Rightarrow & \not\models \rho & \text{(by definition of truth)} \\ & \Rightarrow & \not\vdash \rho & \text{(by 10.3)} \end{array}$$

Hence ρ is true, but cannot be proved.

10.3.3 Second incompleteness theorem from Kurt Gödel

The default of the previous proof is of course that it does not really make sense to formula ρ.

The second incompleteness theorem from Kurt Gödel provides an explicit example of a formula that cannot be proved.

One can express a formula CONSIST that expresses the fact that the theory is consistent. Basically, one writes that its is not possible to prove a formula F and its negations: It is "sufficient"' to write $\neg\exists x(\text{PROVABLE}(x) \wedge \text{PROVABLE}(y) \wedge NEG(x,y))$, where $NEG(x,y)$ means that y is the encoding of the negation of the formula encoded by x.

The second incompleteness theorem from Kurt Gödel allows to prove that this precise formula cannot be proved.

In other words:

Theorem 10.2 (Second incompleteness theorem from Kurt Gödel) *No deduction system can prove its own consistency.*

We will not go into further details.

10.4 Bibliographic notes

Suggested readings To go further with the notions of this chapter, we suggest the reading of the last chapters of the book [Kozen, 1997], which remain short and direct, or of the book [Cori & Lascar, 1993b] for a complete proof.

Bibliography This chapter is taken from one of the last three chapters of the excellent book [Kozen, 1997].

Chapter 11

Basic of complexity analysis of algorithms

The previous discussions have been concerned with the existence or non-existence of algorithms for solving a given problem, but ignoring an essential practical aspect: the resources needed for its execution, that is to say for example the *computation time* or the *memory* that is required on the machine for its execution.

The objective of the next chapter is to focus on one resource, the *computation time*. In the later chapters, we will evoke other resources such as memory space. We could also talk about parallel time, that is to say the time required on a parallel machine.

Let us however start by better understanding the difference between previous chapters and the following chapters. In previous chapters, we were talking about *computability*, that is to say we asked about the existence of an algorithmic solution to a given problem. We will now focus on *complexity*: That is to say, we focus now on decidable problems, i.e., problems for which an algorithm is known. The question is to decide whether there is an *efficient* algorithm.

This leads first to the question what one calls *efficient*, and how this efficiency can be measured. First of all, we think it is important that our reader has clear ideas on what is called the complexity of an algorithm and the complexity of a problem, which are not the same concept.

Remark 11.1 *Even if we will talk about average case complexity in this chapter, we will not need this in the coming chapters : We introduce it here mainly to explain why average case complexity is not used much in practice (at least in complexity theory).*

11.1 Complexity of algorithm

In this chapter, we mostly consider a (decision or general) problem $\mathscr{P}$ for which one knows an algorithm $\mathscr{A}$: This algorithm is known to be correct, and is terminating. It

takes as input some data d, and it produces as its output a result $\mathscr{A}(d)$ by using some resources (so we will only talk about decidable problems for decision problems).

Example 11.1 *The problem $\mathscr{P}$ could for example consist in determining if a given number v is among a list of nb numbers.*

It is clear that one can come up with an algorithm $\mathscr{A}$ to solve this problem. For example:

- *one uses a variable res initially set to* 0*;*
- *one scans the list, and for each element:*
  - *one checks if this element is the number v:*
    - *it this is the case, one sets the variable res to* 1*;*
- *at the end of the loop, on returns res.*

This algorithm certainly is not the most efficient that one can think of. First, we could stop as soon as one sets res to 1*, since the answer is known. Furthermore, one can clearly do something different, such as a* dichotomic search *(a recursive algorithm) if one knows that the list is sorted.*

11.1.1 First considerations

One always measures the efficiency, that is to say the complexity of an algorithm in terms of an *elementary measure* with integer value: This can be the number of instructions executed, the size of the memory that is used, the number of comparisons made, or any other measure.

One just needs that, given an input d, one knows how to associate the value of this measure, denoted by $\mu(\mathscr{A},d)$, to the algorithm $\mathscr{A}$ on input d. For example, for a sorting algorithm working with comparisons, if the elementary measure μ is he number of comparisons done, $\mu(\mathscr{A},d)$ is the number of comparison performed on the input d (a sequence of integers) by an algorithm $\mathscr{A}$ to produce the result $\mathscr{A}(d)$ (the sorted list).

The function $\mu(\mathscr{A},d)$ depends of $\mathscr{A}$, but also of the input d. The quality of an algorithm $\mathscr{A}$ is hence not an absolute criterion, but a quantitative function $\mu(\mathscr{A},.)$ from the inputs to the integers.

11.1.2 Worst case complexity of an algorithm

In practice, to understand the function $\mu(\mathscr{A},.)$, one often aims to evaluate the complexity for the inputs of a given *size*: There is often a function size that maps to every input data d, an integer size(d), that corresponds to some natural parameter. For example, this function can be the number of elements for a sorting algorithm, the size of a matrix for computing the determinant, or the sum of the lengths of the strings for a concatenation algorithm.

To go from a function from the inputs to the integers to a function from the integers (the sizes) to the integers, one then considers the *worst case complexity*: The complexity $\mu(\mathscr{A}, n)$ of algorithm $\mathscr{A}$ on inputs of size n is defined as

$$\mu(\mathscr{A}, n) = \max_{d \text{ input with } \text{size}(d)=n} \mu(\mathscr{A}, d).$$

In words, the complexity $\mu(\mathscr{A}, n)$ is the worst complexity observed on inputs of size n.

By default, when one talks about the *complexity of an algorithm*, one considers the worst case complexity as above.

If one does not know more on the inputs, there is no real hope to do better than this pessimistic view of life, and evaluating the complexity in the worst case (the best case has no particular practical meaning, and in this context, pessimism is far more significant).

11.1.3 Average case complexity of some algorithm

In order to say more, one must know more about the inputs. For example, that they are distributed according to some probabilistic distribution.

In that case, we can then talk about *average case complexity*: The average case complexity $\mu(\mathscr{A}, n)$ of algorithm $\mathscr{A}$ of inputs of size n is defined as

$$\mu(\mathscr{A}, n) = \mathbb{E}[\mu(\mathscr{A}, d) | d \text{ inputs with } \text{size}(d) = n],$$

where $\mathbb{E}$ denotes expectation (the average).

This is equivalent to

$$\mu(\mathscr{A}, n) = \sum_{d \text{ inputs with } \text{size}(d)=n} \pi(d)\mu(\mathscr{A}, d),$$

where $\pi(d)$ denotes the probability of having this input of size n.

In practice, if the worst case might be rare, and the average case analysis may seem more appealing.

But first, it is important to know that one cannot talk about expectation/average without a probability distribution on the inputs. This implies on the one hand that the distribution of the data given as input must be known, something which is very delicate to predict or estimate in practice. How to anticipate for example the lists that will be given to a sorting algorithm?

One sometimes makes the hypothesis that the inputs have same probability (when this makes sense, as in the case where one wants to sort n numbers between 1 and n) but this is often very arbitrary, and not totally justifiable.

On the other hand, as we will see, computing the average case complexity is often more delicate to deal with than worst case analysis.

11.2 Complexity of a problem

One can also talk about the *complexity of a problem* which provides a way to talk about the optimality of an algorithm to solve a given problem.

One fixes a problem $\mathscr{P}$, for example, the problem of sorting a list of integers. Let $Alg(\mathscr{P})$ be the class of all algorithms that solves $\mathscr{P}$: an algorithm $\mathscr{A}$ of $Alg(\mathscr{P})$ is an algorithm that answers to the specification of the problem $\mathscr{P}$: For every input d, it produces a correct answer $\mathscr{A}(d)$.

The complexity of the problem $\mathscr{P}$ is defined as the infinimum[1] of the complexity of the algorithms of $Alg(\mathscr{P})$. Consequently, an algorithm $\mathscr{A}$ is *optimal* if its complexity is equal to the optimal complexity of $Alg(\mathscr{P})$: That is to say, there is no other algorithm $B \in Alg(\mathscr{P})$ with a smaller complexity. We write $\mu(\mathscr{P}, n)$ for the complexity of some optimal algorithm[2] on inputs of size n .

In other words, we do not only make the inputs of size n vary, but also the algorithm. One considers the best algorithm that solves the problem. The best being the one with the best complexity in terms of previous definition, and hence in the worst case. This is hence the complexity of the best algorithm in the worst case.

The interest of this definition is to be able to state that some algorithm is optimal: That is to say, that an algorithm is such that any other correct algorithm would be less efficient by definition.

11.3 Example : Computing the maximum

We will illustrate the previous discussion with an example: The problem of computing the maximum. The problem is the following: We are given a list of non-negative integers $e_1, e_2, \cdots, e_n$, with $n \geq 1$, and we want to output $M = \max_{1 \leq i \leq n} e_i$, that is the maximum of these integers.

11.3.1 Complexity of a first algorithm

Assuming that the input is in an array, the following Java function solves the problem:

```
static int max(int T[]) {
    int l = T.length-1;
    int M =  T[l];
    l = l-1;
    while (l ≥ 0) {
     if (M < T[l])  M = T[l];
     l = l-1;
    }
    return M;
  }
```

[1] If it exists.

[2] Assuming it exists. It may not exist.

Assume that our elementary measure is the number of comparisons. We make 2 comparisons per iteration of the loop, which is executed $n-1$ times, plus 1 last of type $l \geq 0$ when l has the value 0. We therefore make $\mu(\mathscr{A}, n) = 2n-1$ comparisons for this algorithm $\mathscr{A}$, where n is the size of the input, that is to say the number of integers in the list $e_1, e_2, \cdots, e_n$. This number is independent of the input d, and hence $\mu(\mathscr{A}, n) = 2n-1$.

In contrast, if our elementary measure μ is the number of assignments, we analyze the complexity as follows: we make 3 assignments before the **while** loop. Each iteration of the loop does either 1 or 2 assignments according to the result of the test M<T[l]. We hence have for an input d of size n, $n+2 \leq \mu(\mathscr{A}, d) \leq 2n+1$: The minimal value is reached for a list that has its maximum in its last element, and the maximum value for a list of n different numbers sorted in decreasing order. So here $\mu(\mathscr{A}, d)$ depends on the input d. The worst case complexity is hence $\mu(\mathscr{A}, n) = 2n$.

11.3.2 Complexity of a second algorithm

If the input is in an array, defined for example by:

```
class List {
  int val ;      // The element
  List next ;    // La suite

  List (int  val, List next) {
    this.val = val ; this.next = next ;
  }
}
```

the following function solves the problem.

```
static int max(List a) {
  int M = a.val;
  for (a = a.next; a ≠ null;  a = a.next) {
    if (a.val > M)
      M = a.val;}
  return M;
}
```

Assume that our elementary measure is the number of comparisons between integers (we are not counting the comparisons between variables of type "reference" on the type List). We make one comparison per iteration of the loop, that is executed $n-1$-times, so $n-1$ comparisons in total.

The complexity $\mu(\mathscr{A}, n)$ of this algorithm $\mathscr{A}$ on the inputs of size n is hence $n-1$.

11.3.3 Complexity of the problem

One can wonder if it is possible to do better, and solve the problems with less than $n-1$ comparisons: The answer is no, under the condition that one restricts to algo-

rithms that work only with comparisons[3]. Indeed, this algorithm is optimal in terms of number of comparisons.

Consider the class $\mathscr{C}$ of algorithms that solve the problem of finding the maximum of n elements by using as decision criteria the comparisons between elements, with the above hypothesis.

Let us start by stating the following property:

Lemma 11.1 *Any algorithm $\mathscr{A}$ of $\mathscr{C}$ is such that any element distinct from the maximum is compared at least once to an element greater then itself.*

Proof: Indeed, let i_0 be the index of the maximum M returned by the algorithm on a list $L = e_1 e_2 \cdots e_n$, that is $e_{i_0} = M = \max_{1 \leq i \leq n} e_i$. We reason by contradiction: Let $j_0 \neq i_0$ such that e_{j_0} is not compared to any element greater than itself. Then the element e_{j_0} has then not been compared to the maximum e_{i_0} .

Consider the list $L' = e_1 e_2 \cdots e_{j_0-1} M+1 e_{j_0+1} \cdots e_n$ obtained from L by replacing the element of index j_0 by $M+1$.

The algorithm $\mathscr{A}$ will do exactly the same comparisons on L and on L', without comparing $L'[j_0]$ with $L'[i_0]$ and hence will return $L'[i_0]$, so an incorrect result. We reach a contradiction which proves the property. □

It follows from this lemma that it is not possible to find the maximum of n elements with less than $n-1$ comparisons between integers. In other words, the complexity of the problem $\mathscr{P}$ of computing the maximum on the inputs of size n is $\mu(\mathscr{P}, n) = n-1$.

The previous algorithm works with $n-1$ such comparisons and is thus optimal for this measures of complexity.

11.3.4 Average case complexity of the algorithm

If our elementary measure μ is the number of assignments inside the **for** loop, one sees that the complexity depends on the input.

To evaluate its average case complexity, one needs to make some hypothesis on the distribution of inputs. Suppose that the lists given as inputs are permutations of $\{1, 2, \cdots, n\}$, and that the $n!$ permutations all have same probability.

One can prove [Sedgewick & Flajolet, 1996, Froidevaux et al., 1993] that the average case complexity on inputs of size n for this elementary measure μ is then H_n, the nth harmonic number: $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$. The number H_n is of order $\log n$ when n tends to infinity.

However, the computation is rather technical and would be laborious in the framework of this course.

Let us simplify the discussion, and let us focus on an even simpler problem: Instead of finding the maximum in the list $e_1, e_2, \ldots, e_n$, with $n \geq 1$, suppose we are given a list of integers of $\{1, 2, \ldots, k\}$ and some integer $1 \leq v \leq k$, and we want to determine if there is some index $1 \leq i \leq n$ with $e_i = v$.

The following algorithm solves the problem:

[3] If the inputs are integers, and arithmetic is authorized, it may be possible to decrease the number of comparisons. We will not discuss this type of algorithms here.

```
static boolean find(int[] T, int v) {
    for (int i = 0; i < T.length; i++)
        if (T[i] == v)
            return true;
    return false;
}
```

Its worst case complexity in terms of elementary instructions is linear in n, since the loop is executed n times in the worst case.

Observe that the lists given as inputs are functions from $\{1,2,\ldots,n\}$ to $\{1,2,\ldots,k\}$, that we will call *array*. Suppose that each of these arrays has the same probability to be the input.

Observe that there are k^n arrays. Among those, $(k-1)^n$ do not contain the element v and in that case, the algorithm performs exactly n iterations. In the contrary case, the integer is in the array, and its first occurrence is then i with probability

$$\frac{(k-1)^{i-1}}{k^i}$$

and the algorithms stops after i iterations.

In total, we have a average case complexity of

$$C = \frac{(k-1)^n}{k^n} \times n + \sum_{i=1}^{n} \frac{(k-1)^{i-1}}{k^i} \times i$$

But for all x we have

$$\sum_{i=1}^{n} i x^{i-1} = \frac{1 + x^n(nx - n - 1)}{(1-x)^2}$$

(to establish this result, it is sufficient to take derivative of $\sum_{i=1}^{n} x^i = \frac{1-x^{n+1}}{1-x}$) and hence

$$C = n\frac{(k-1)^n}{k^n} + k\left(1 - \frac{(k-1)^n}{k^n}(1 + \frac{n}{k})\right) = k\left(1 - \left(1 - \frac{1}{k}\right)^n\right)$$

11.4 Asymptotics

11.4.1 Asymptotic complexity

As we have seen in the previous example, a precise and complete study of the complexity of a problem can be very fastidious, and often hard. This is why the focus in computer science is often on the order of growth of the (asymptotic) complexity when the size n of the inputs becomes very big. Such an analysis is often quite representative of the performance of the algorithm, even if of course, talking about asymptotics up to some constants has its limits.

11.4.2 Landau notations

As it is the custom in computer science, one often reasons on the order of growth using the $\mathcal{O}(.)$ notation. We recall the following notations:

Definition 11.1 (Notation $\mathcal{O}(.)$) *Let f and g be two functions $f, g : \mathbb{N} \to \mathbb{R}^{>0}$. We write $f(n) = \mathcal{O}\left(g(n)\right)$ if there exist integers c and n_0 such that for all $n \geq n_0$,*

$$f(n) \leq cg(n).$$

Intuitively, this means that f is lower than g up to some multiplicative constant, for sufficiently big input instances.

In a similar way, one defines:

Definition 11.2 (Notations o, Ω, Θ) *Let f and g be two functions $f, g : \mathbb{N} \to \mathbb{R}^{>0}$.*

- *We write $f(n) = o(g(n))$ if for all positive real numbers c there exists an integer n_0 such that for all $n \geq n_0$,*

$$f(n) \leq cg(n).$$

- *We write $f(n) = \Omega(g(n))$ if there exist integers c and n_0 such that for all $n \geq n_0$,*

$$cg(n) \leq f(n).$$

(we have in this case $g = \mathcal{O}\left((f)\right)$)

- *We write $f(n) = \Theta(g(n))$ when $f(n) = \mathcal{O}\left(g(n)\right)$ and $f(n) = \Omega(g(n))$ hold.*

Exercise 11.1 *(solution on page 237) Let f and g be two functions such that*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

exists and is a number $c > 0$. Prove that $f(n) = \Theta(g(n))$.

Exercise 11.2 *Prove:*

- *If $f = \mathcal{O}\left(g\right)$ and $g = \mathcal{O}(h)$ then $f = \mathcal{O}(h)$.*
- *If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.*
- *If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.*

Exercise 11.3 *(solution on page 237) Give some examples of algorithms of respective complexity (in terms of number of instructions):*

- *linear, that is* $O(n)$
- $O(n \log n)$
- *cubic, that is* $O(n^3)$
- *non-polynomial*

Exercise 11.4 *(solution on page 238) Suppose that one has algorithms with the complexities listed below (assuming that this corresponds to some exact times). How much are these algorithms slowed down when* (*a*) *the size of the input is doubled* (*b*) *the size of the input is increased by* 1.

1. n^2
2. n^3
3. $100n^2$
4. $n \log n$
5. 2^n

11.5 Bibliographic notes

Suggested readings To go further on the notions of this chapter, we suggest to read the first chapters of [Kleinberg & Tardos, 2006], or of the course INF421 (old version) of École Polytechnique.

Bibliography The text of this chapter is taken from a text that we wrote for the lecture notes of INF412. It is inspired by the introduction of the textbook [Kleinberg & Tardos, 2006]. The analysis of the computation of the maximum and its variations is based on the book [Froidevaux et al., 1993].

Chapter 12

Time complexity

This chapter is focusing on some particular resource of an algorithm: The times it takes to be executed.

The previous chapter applies in particular to this measure: The computation time of some algorithm is defined as the time it takes to be executed.

To illustrate the importance of this measure of complexity, let us focus on the time corresponding to algorithms of complexity n, $n\log_2 n$, n^2, n^3, 1.5^n, 2^n and $n!$ for input of size n, for increasing n, on a processor able to execute one million of elementary instructions by second. We write ∞ in the array as soon as values more than 10^{25} years (this figure is repeated from [Kleinberg & Tardos, 2006]).

Complexity	n	$n\log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	4 s
$n = 30$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	18 min	10^{25} years
$n = 50$	< 1 s	< 1 s	< 1 s	< 1 s	11 min	36 years	t|
$n = 100$	< 1 s	< 1 s	< 1 s	1s	12,9 years	10^{17} years	t|
$n = 1000$	< 1 s	< 1 s	1s	18 min	t|	t|	t|
$n = 10000$	< 1 s	< 1 s	2 min	12 days	t|	t|	t|
$n = 100000$	< 1 s	2 s	3 hours	32 years	t|	t|	t|
$n = 1000000$	1s	20s	12 days	31,710 years	t|	t|	t|

As one can see, an algorithm of exponential complexity is very rapidly useless, and is hence *not reasonable*. The main subject of this chapter is to understand what is called a *reasonable* algorithm in theoretical computer science, and to understand the theory of NP-completeness that allows to discuss the frontier between reasonable and non-reasonable algorithms.

12.1 The notion of reasonable time

12.1.1 Convention

For several reasons, the following convention has been adopted in Computer Science:

Definition 12.1 (Efficient algorithm) *An algorithm is* efficient *if its time complexity is polynomial, that is to say in $\mathcal{O}\left(n^k\right)$ for some integer k.*

This is a convention (and others could have been chosen[1]) that has been widely been accepted since the 70's.

Remark 12.1 *One can argue that an algorithm of complexity $\mathcal{O}\left(n^{1794}\right)$ is not* very reasonable. *Yes, but one must indeed fix a convention, and this is indeed considered as reasonable in theory of complexity.*

Remark 12.2 *Why don't taking a linear time, or a quadratic time as the notion of "reasonable": because this is not working so well. See Remark 12.3 below.*

12.1.2 First reason: To abstract from coding issues

One of the reasons for this convention is the following remark: Most of the computer science objects can be represented in various manners, but transforming one representation into the other is doable in a time that remains polynomial in the size of the encoding.

The class of polynomial being stable by composition, this implies that an algorithm that is polynomial with respect to a given representation can be transformed into a polynomial algorithm with respect to another representation.

One can then talk about *efficient* algorithm on these objects without having to go to the details on how these objects are actually represented.

Example 12.1 *A graph can be represented by a matrix, its* adjacency!matrix*: if the graph has n vertices, one considers an array T of size n by n, whose element $T[i][j] \in \{0,1\}$ values 1 if and only if there is an edge between the vertex i and the vertex j.*

One can also represent a graph by an adjacency!list*: To represent a graph with n vertices, one considers n lists. The list number i encodes the neighbours of vertex number i.*

One can go from one representation to the other in a time that is polynomial in the size of each: This is left to the reader to get convinced of this fact in her or his preferred programming language.

An efficient *algorithm for one of the representation can always be transformed into an* efficient *algorithm for the other representation: One just needs to start*

[1] And actually, there were others previously.

by possibly converting the representation to the representation on which the algorithm works.

Furthermore, for the same reasons, as all of these graph representations remain polynomial in n, by using the fact that a graph with n vertices has at most n^2 vertices, an algorithm polynomial in n is nothing but an efficient algorithm on graphs, that is to say on any of the previous representations, or any usual representations of graphs.

12.1.3 Second reason: To abstract from the computational model

Another deep reason is the following: Let's come back to Chapter 7. We have proved that all computational models considered in this latter chapter can simulate one the other: RAM machines, Turing machines, 2 stacks machines, counters machines.

If we put aside the counters machines whose simulation is particularly inefficient, and whose interest is perhaps mainly only theoretical, we can observe that a number t of instructions for one model can be simulated using a number polynomial in t of instructions for the other. The class of polynomial being stable by composition, this implies that, possibly by simulating one model by the other, an algorithm that is polynomial in one model of computation can be transformed into a polynomial algorithm for any of the other models of computation.

We can then talk about *efficient* algorithms on an object without having to precise if the program is considered in one model of computation or the other[2]

In particular the notion of efficient algorithm is independent of the chosen programming language: An efficient algorithm in CAML is an efficient algorithm in JAVA, or an efficient algorithm in C.

Remark 12.3 *We come back to Remark 12.2. Why don't taking a linear time, or a quadratic time as the notion of "reasonable": In particular, because these notions of linear and quadratic time would not satisfy the above property. Indeed, the notion of linear time or of quadratic time is depending on the chosen model of computation, contrary to the notion of polynomial time computability, and/or are not closed by so nice closure properties.*

For example, for linear time, a time T for a Turing machine with two tapes is not clearly simulated in a time linear in T (This is $\mathcal{O}\left(T^2\right)$, that is quadratic with the obvious technique detailed in previous chapters, hence not linear. Notice that if this is possible to prove that $\mathcal{O}\left(T\log(T)\right)$ is possible if using a smart divide and conquer technique).

For example, for quadratic time: As $(T^2)^2 = T^4$, quadratic time is not closed by composition, hence composing a quadratic time "reasonable" algorithm with a quadratic time "reasonable" algorithm would not be "reasonable".

[2]Most purist will observe a problem with the RAM model of Chapter 7: One must take into account in the complexity measure the size of the integers involved in the executed elementary operations and not only the number of instructions. But this is only details, and what is written above remains totally true, is one forbids to RAM machines to manipulate integers of arbitrary size. Notice that this would anyway not be reasonable with respect to the processors that they intend to model that work in practise on words with a finite number of bits (typically 32 or 64 bits for example).

Since the notion of efficiency is not depending of the model, one will use the Turing machine model in all what follows: When w is a word, remember that we write length(w) for its length.

Definition 12.2 (TIME(t(n))) *Let $t : \mathbb{N} \to \mathbb{N}$ be a function. We define the class* TIME(t(n)) *as the class of problems (languages) decided by a Turing machine in time $\mathcal{O}(t(n))$, where $n =$ length(w) is the size of the input.*

If one prefers, $L \in$ TIME(t(n)) if there exists a Turing machine M such that:

- M decides L: for any word w, M accepts w if and only if $w \in L$, and M rejects w if and only if $w \notin L$;
- M takes a time bounded by $\mathcal{O}(t(n))$:
  - if one prefers: There are integers n_0, c, and k such that for every word w, if w is of sufficiently big size, that is to say if length$(w) \geq n_0$, then M accepts or rejects using at most $c * t(n)$ steps, where $n =$ length(w) denotes the length of w.

Remember that we focus in this chapter and in the following (and more generally in complexity theory) uniquely on decidable problems.

12.1.4 Class P

The class of problems which admit a reasonable algorithm corresponds then to the following class:

Definition 12.3 (Class P) *The class* P *is the class of problems (languages) defined by:*

$$\mathrm{P} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME}(\mathrm{n}^{\mathrm{k}}).$$

In other words P is exactly the class of problems that admit a polynomial algorithm.

Here a several examples of problems in P.

Example 12.2 (Testing the colouring of a graph)

Input: *A graph $G = (V, E)$, a finite set C of colours, and colour $c(v) \in C$ for every vertex $v \in V$.*

Answer: *Decide if G is* coloured *with these colours: that is to say if there are no edge of G with two extremities of the same colour.*

This problem is in class P. *Indeed, it is sufficient to go through the edges of the*

graph and to test for each of these edges if the colour of its two extremities are the same.

Example 12.3 (Evaluation in propositionnal calculus)

Input: *A formula $F(x_1, x_2, \cdots, x_n)$ of propositional calculus, some values $x_1, \cdots, x_n \in \{0,1\}$ for each of the variables of the formula.*

Answer: *Decide whether the formula F evaluates to true for this value of these variables.*

This problem is in class P. *Indeed, given some formula of propositional calculus $F(x_1, \cdots, x_n)$ and some values $x_1, x_2, \cdots, x_n \in \{0,1\}$, it is easy to compute the truth value of $F(x_1, \cdots, x_n)$. This is done in a time that one can easily check to be polynomial in the size of the input: Basically, one evaluates the formula inductively by propagating the truth value of variables and constants through logical operators (and, or, and negations) of the formula.*

Many other problems are in P.

12.2 Comparing problems

12.2.1 Motivation

It turns out however that there is a whole class of problems for which we did not succeed up to today to prove formally that this is not possible.

This is historically what leaded to consider the class of problems that we call NP, that we will consider in the following sections.

Some example of problems in this class are the following:

Example 12.4 (k-COLORABILITY)

Input: *A graph $G = (V, E)$ and some integer k.*

Answer: *Decide if there exists a colouring of the graph using at most k colours: that is to say decide if there exists a way to colour the vertices of G with at most k colours to obtain a colouring of G.*

Example 12.5 (SAT**, Satisfaction in propositionnal calculus)**

Input: *A formula $F(x_1, \cdots, x_n)$ of propositional calculus.*

Answer: *Decide if F is satisfiable: that is to say if there exists $x_1, \cdots, x_n \in \{0,1\}^n$ such*

that F evaluates to true with this value of the variables $x_1, \cdots, x_n$.

Example 12.6 (HAMILTONIAN CIRCUIT)

Input: *A graph $G = (V, E)$ (non-oriented).*

Answer: *Decide if there exists a* Hamiltonian circuit *in G, that is to say decide if there exists a path that goes through, once and exactly once, every vertex and that comes back to its starting point.*

For the three problems, some exponential time algorithm is known: test all the ways to colour the vertices for the first, or all the values of $\{0,1\}^n$ for the second, or all the paths for the last one. For the three problems, one does not know any efficient algorithm, and one has not succeeded to prove that there is none at this date.

As we will see, one can however prove that these three problems are equivalent with respect to their level of difficulty, and this will lead to consider the notion of reduction, that is to say to compare the hardness of problems.

Before, let's precise a few things.

12.2.2 Remarks

In this chapter and in the next chapter, we will essentially only talk about decision problems, that is to say about problems whose answer is either "true" or "false": See Definition 9.2.

Example 12.7 *"Sort n numbers" is not a decision problem: The output is a list of sorted numbers.*

Example 12.8 *"Given a graph $G = (V, E)$, determine the number of colours to colour G" is not a decision problem, as the output is some integer. One can however formulate this problem as a decision problem, of type "Given a graph $G = (V, E)$, and some integer k, determine if the graph G admits a colouring with less than k colours": This is the problem* k-COLORABILITY.

Before talking about reductions, we must talk about functions computable in polynomial time: This is the expected notion, even if we are force to provide the details as we have not done it yet.

Definition 12.4 (Function computable in polynomial time) *Let Σ and Σ' be two alphabets. A (total) function $f : \Sigma^* \to \Sigma'^*$ is* computable in polynomial time *if there exists a Turing machine Turing A, working over alphabet $\Sigma \cup \Sigma'$, and some integer k, such that for every word w, A with input w terminates in time $\mathcal{O}\left(n^k\right)$ with, at the moment it stops, $f(w)$ written on its tape, where $n = \text{length}(w)$.*

The following result is easy to establish:

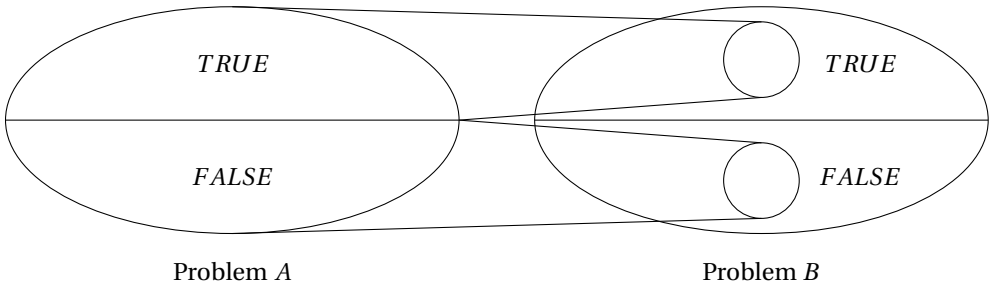


Figure 12.1: The reductions transform positive instances to positive instances and negative instances to negative instances.

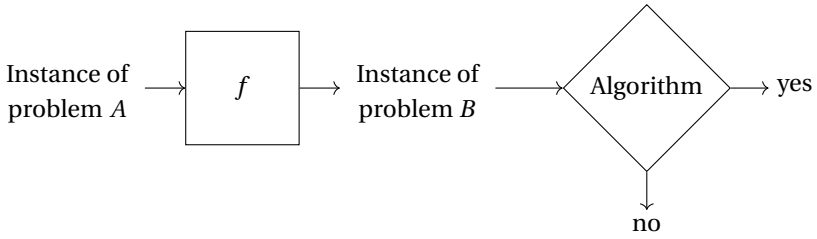


Figure 12.2: Reduction from problem A to problem B. If one can solve problem B in polynomial time, then one can solve problem A in polynomial time. The problem A is hence at least as easy as problem B, denoted by $A \leq B$.

Proposition 12.1 (Stability by composition) *The composition of two functions computable in polynomial time is computable in polynomial time.*

12.2.3 The notion of reduction

This permits to introduce the notion of reduction between problems (similar to the one of chapter 9, except that we are talking about computability in polynomial time instead of just computability): the idea is that if A reduces to B, then problem A is at least as easy as problem B, or if one prefers, the problem B is at least as hard as problem A: See Figure 12.2 and Figure 12.1.

Definition 12.5 (Reduction) *Let A and B two problems of respective alphabets Σ_A and Σ_B. A* reduction from *A to B is a function $f : \Sigma_A^* \to \Sigma_B^*$ computable in polynomial time such that $w \in A$ if and only if $f(w) \in B$. We write $A \leq B$ when A reduces to B.*

This behaves as expected: A problem is at least as easy (and hard) as itself, an the relation "being at least as easy as" is transitive. In other words:

Theorem 12.1 *$\leq$ is a preorder:*

1. *$L \leq L$;*
2. *$L_1 \leq L_2$, $L_2 \leq L_3$ implies $L_1 \leq L_3$.*

Proof: Consider the identity function as function f for the first point.

For the second point, suppose $L_1 \leq L_2$ via the reduction f, and $L_2 \leq L_3$ via the reduction g. We have $x \in L_1$ if and only if $g(f(x)) \in L_2$. It is then sufficient to see that $g \circ f$, provides the reduction, and is computable in polynomial time, since it is the composition of two functions computable in polynomial time. □

Remark 12.4 *It is not an order, since $L_1 \leq L_2$, $L_2 \leq L_1$ does not imply $L_1 = L_2$.*

It is then natural to introduce:

Definition 12.6 *Two problems L_1 and L_2 are equivalent, denoted by $L_1 \equiv L_2$, if $L_1 \leq L_2$ and if $L_2 \leq L_1$.*

We have then $L_1 \leq L_2$, $L_2 \leq L_1$ implies $L_1 \equiv L_2$.

12.2.4 Applications to comparison of hardness

Intuitively, if a problem is at least as easy as a polynomial problem, then it is polynomial. Formally:

Proposition 12.2 (Reduction) *If $A \leq B$, and if $B \in$ P then $A \in$ P.*

Proof: Let f be a reduction from A to B. A is decided by the Turing machine that, on some input w, compute $f(w)$ and then simulates the Turing machine that decides B on input $f(w)$. Since we have $w \in A$ and only if $f(w) \in B$, the algorithm is correct, and it works in polynomial time. □

By considering the contrapositive of previous proposition, we obtain the following formulation that says that if a problem has no polynomial algorithm, and it is at least as easy as another one, then this latter has also no polynomial algorithm.

Proposition 12.3 (Reduction) *If $A \leq B$, and if $A \notin$ P then $B \notin$ P.*

Example 12.9 *We will see that the problems* k-COLORABILITY, SAT *and* HAMILTONIAN CIRCUIT *are equivalent (and are equivalent to all the* NP*-complete problems). There is hence an efficient algorithm for one of them if and only if there is one for the other(s).*

12.2.5 Hardest problems

If one considers a class of problems, we can introduce the notion of hardest problem for the class, i.e. maximum for $\leq$. This is the notion of *completeness*:

Definition 12.7 ($\mathscr{C}$-completness) *Let $\mathscr{C}$ be class of decision problems. A problem A is said to be $\mathscr{C}$-complete, if*

1. *it is in class $\mathscr{C}$;*
2. *every problem B of $\mathscr{C}$ is such that $B \leq A$.*

We say that a problem A if *$\mathscr{C}$-hard* it it satisfies condition 2. of Definition 12.7. A problem A is hence $\mathscr{C}$-completeness if it is $\mathscr{C}$-hard and in class $\mathscr{C}$.

A $\mathscr{C}$-complete problem is hence the most difficult, or one of the most difficult problems of the class $\mathscr{C}$. Clearly, if there are several, they are all equivalent:

Corollary 12.1 *Let $\mathscr{C}$ be a class of languages. All the $\mathscr{C}$-complete problems are equivalent.*

Proof: Let A and B two $\mathscr{C}$-complete problems. Apply condition 2 of Definition 12.7 to A with respect to $B \in \mathscr{C}$, and to B with respect to $A \in \mathscr{C}$. □

12.3 The class NP

12.3.1 The notion of verifier

The problems k-COLORABILITY, SAT and HAMILTONIAN CIRCUIT mentioned previously have a common point: While it is not clear that they admit a polynomial algorithm, it is very clear that they admit a polynomial *verifier*.

Definition 12.8 (Verifier) *A* verifier *for a problem A is an algorithm (i.e. Turing machine) V such that*

$$A = \{w | V \text{ accepts } \langle w, u\rangle \text{ for some word } u\}.$$

The verifier is polynomial if algorithm V decides its answer in a time polynomial in the length of w. One say that a language is polynomially verifiable *if it admits a polynomial verifier.*

The word u is called a *certificate* (sometimes also a *proof*, or a *witness*) for w. In other words, a verifier is using one more information, namely u, to check that w is in A.

Remark 12.5 *Observe that one can always restrict to certificates of length polynomial in the length of w, since in a time polynomial in the length of w the*

algorithm V will not read more than a polynomial number of symbols of the certificate.

Example 12.10 *A certificate for the problem* k-COLORABILITY *is given by some colours for all the vertices.*

We will not always provide so many details, but here is the justification: Indeed, a graph G is in k-COLORABILITY is and only if one can find some word u that encodes the colours for all the vertices and all these colours provide a correct colouring: The algorithm V, i.e. the verifier, given $\langle G, u\rangle$, is only checking that the colouring corresponding to u is correct. This can be done in a time polynomial in the size of the graph: See discussion of Example 12.2.

Example 12.11 *A certificate for the problem* SAT *is constituted of a value $x_1, \ldots, x_n \in \{0,1\}$ for each of the variables of the formula F: The verifier needs only to check that these values satisfy the formula F. This can be done in a time polynomial in the size of the formula: See example 12.3.*

Example 12.12 *A certificate for the problem* HAMILTONIAN CIRCUIT *is constituted by a circuit. The verifier needs only to check that the circuit is Hamiltonian. This can be done in a time polynomial in the size of the graph.*

This leads to the following definition:

Definition 12.9 NP *is the class of problems (languages) that have a polynomial verifier.*

This class is important that it turns out that it contains an incredible number of problems of practical interest. It contains k-COLORABILITY, SAT and HAMILTONIAN CIRCUIT but also many other problems: See for example all the examples of the next chapter.

By construction, we have (as the empty word is a valid certificate for any problem of P):

Proposition 12.4 $\mathrm{P} \subseteq \mathrm{NP}$.

12.3.2 The question P = NP?

Clearly, we have either P = NP or $\mathrm{P} \subsetneq \mathrm{NP}$: See Figure 12.3.

The question to know if these two classes are equal or distinct is an impressive challenge. First because it is one of the unsolved questions among the most (maybe the) famous of Theoretical Computer Science that have been challenging research for the last 50 years: It has been selected in the list of the most important questions

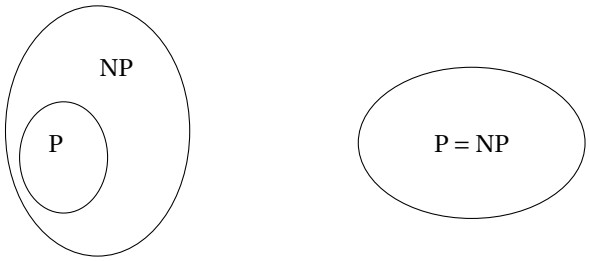


Figure 12.3: One of the two possibilities is correct.

for Mathematics and Computer Science in 2000. The *Clay Mathematics Institute* offers 1 000 000 dollars to the person that will determine the answer to this question.

But mainly, if P = NP, then all the problems polynomially verifiable would be decidable in polynomial time. Most of the people think that the two classes are distinct since there are a very huge number of problems for which nobody have succeeded to provide a polynomial algorithm for more than 40 years.

It has also an impressive economical impact, since many systems, including today's cryptographic systems are based on the hypothesis that these two classes are distinct. If it is not the case, many considerations about these systems would collapse, and numerous cryptographic techniques would have been to be revisited.

12.3.3 Non-deterministic polynomial time

Let's first do a small parenthesis on terminology: The "N" in NP comes from *non deterministic* (and not from *not* as many often believe), because of the following result:

Theorem 12.2 *A problem is in* NP *if and only if it is decided by a non-deterministic Turing machine in polynomial time.*

Remember that we have introduced the non-deterministic Turing machines in Chapter 7. We say that a language $L \subset \Sigma^*$ is *decided by the non-deterministic machine M in polynomial time* if M decides L and M takes a time bounded by $\mathcal{O}(t(n))$: There are integers n_0, c, and k such that for all words w of sufficiently big size, i.e. $n = \text{length}(w) \geq n_0$, M admits a computations that accepts in less than $c * n^k$ steps, and for $w \notin L$, all the computations of M lead to a rejecting configuration in less than $c * n^k$ steps.

Proof: Consider a problem A of NP. Let V be the associated verifier, that runs in polynomial time $p(n)$. We build a non-deterministic Turing machine M, that, on some word w, will produce in a non-deterministic way a word u of length $p(n)$, and then will simulate V on $\langle w, u \rangle$: If V accepts, then M accepts. If V rejects, then M rejects. The machine M decides A.

Conversely, let A be a problem decided by a non-deterministic Turing machine M in polynomial time $p(n)$. As in the proof of Proposition 7.3 in Chapter 9, we can

state that the non-deterministic degree of the machine is bounded by some integer r, and that the sequence of the non-deterministic choices made by the machine M up to time t can be encoded by a sequence of length t of integers between 1 and (at most) r.

Consequently, a sequence of integers of length $p(n)$ between 1 and r is a valid certificate for a word w: Given w and a word u encoding such a sequence, a verifier V can easily check in polynomial time if the machine M accepts w with this sequence of non-deterministic choices. □

More generally, we define:

Definition 12.10 (NTIME(t(n))) *Let $t : \mathbb{N} \to \mathbb{N}$ be a function. We define the class* NTIME(t(n)) *as the class of problems (languages) decided by a non-deterministic Turing machine in time $\mathcal{O}(t(n))$, where n is the size of the input.*

Corollary 12.2

$$\text{NP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}(\text{n}^k).$$

12.3.4 NP-completeness

It turns out that the class NP contains a very huge number of complete problems. The following chapter presents a whole list of such problems.

The difficulty is to succeed to produce a first such problem. This is the object of the Cook and Levin's theorem.

Theorem 12.3 (Cook-Levin) *The problem* SAT *is* NP*-complete.*

We will prove this theorem in the next section.

Let's first start by reformulating what this means.

Corollary 12.3 P = NP *if and only if* SAT ∈ P.

Proof: Since SAT is in NP, if P = NP, then SAT ∈ P.

Conversely, since SAT is NP-complete, for any problem $B \in$ NP, $B \leq$ SAT and so $B \in$ P if SAT ∈ P by Proposition 12.2. □

What we have just done is true for any NP-complete problem.

Theorem 12.4 *Let A be a* NP*-complete problem.*
P = NP *if and only if $A \in$* P.

Proof: Since A is complete it is in NP, and hence if P = NP, then $A \in$ P. Conversely, since A is NP-hard, for any problem $B \in$ NP, $B \leq A$ and hence $B \in$ P if $A \in$ P by Proposition 12.2. □

Remark 12.6 *We hence see the importance of producing* NP*-complete problems for proving* $P \neq NP$*: Producing a problem for which one could succeed to prove that there is no polynomial time algorithm. At this day, none of the thousand of known* NP*-complete problems have provided a way to prove that* $P \neq NP$.

Remark 12.7 *Remember that all the complete problems are equivalent by Corollary 12.1.*

12.3.5 A method to prove NP-completeness

The NP-completeness of a problem is established in the quasi-totality of the cases as follows:

In order to prove the NP*-completeness of a problem A, it is sufficient:*

1. *to prove that it is in* NP*;*
2. *and to prove that $B \leq A$ for some problem B that is known to be* NP*-complete.*

Indeed, the point 1. guarantees that $B \in NP$, and point 2. guarantees that for any problem $C \in NP$ we have $C \leq A$: Indeed by the NP-completeness of B we have $C \leq B$, and since $B \leq A$, we obtain $C \leq A$.

Remark 12.8 *Be careful, the* NP*-completeness of a problem A is obtained by proving that is is at least as hard as another* NP*-complete, and not the contrary. This is a frequent error.*

The following chapter is devoted to many applications of this strategy on various problems.

12.4 Two examples of proofs of NP-completeness

We apply the above strategy to prove the that 3-SAT is NP-complete.

12.4.1 Proof of the NP-completeness of 3-SAT

Definition 12.11 (3-SAT)

Input: *A set of variables $\{x_1, \cdots, x_n\}$ and a formula $F = C_1 \wedge C_2 \cdots \wedge C_\ell$ with $C_i = y_{i,1} \vee y_{i,2} \vee y_{i,3}$, where for every i, j, $y_{i,j}$ is either x_k, or $\neg x_k$ for one of the x_k.*

Answer: *Decide whether F is satisfiable, that is, decide if there exist $x_1, \cdots, x_n \in \{0,1\}^n$ such that F evaluates to true with this value of its variables $x_1, \cdots, x_n$.*

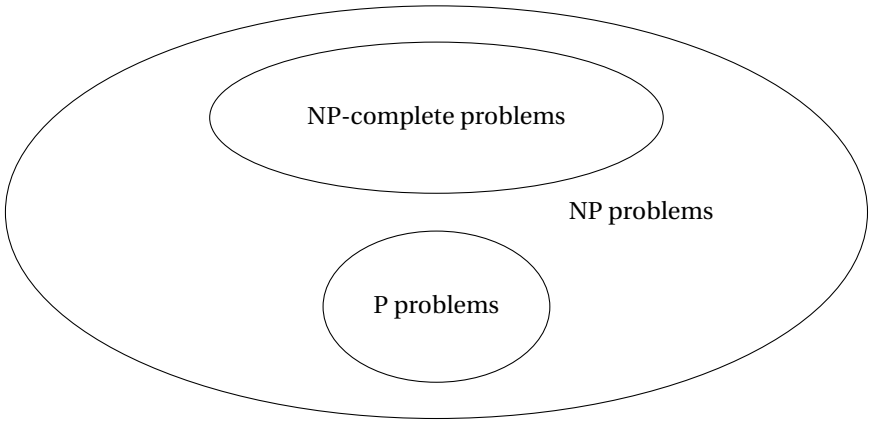


Figure 12.4: Situation with hypothesis P ≠ NP.

Theorem 12.5 *The problem* 3-SAT *is* NP*-complete.*

Proof: First note that 3-SAT is in NP. Indeed, given an assignment of the truth value of the variables, it is easy to check in polynomial time that the formula is true with these values of the variables.

We will reduce SAT to 3-SAT. Let F be a CNF-formula. Let C be a clause of F, say $C = x \vee y \vee z \vee u \vee v \vee w \vee t$. We introduce new variables a, b, c, d associated to this clause, and we replace C by the formula

$$(x \vee y \vee a) \wedge (\neg a \vee z \vee b) \wedge (\neg b \vee u \vee c) \wedge (\neg c \vee v \vee d) \wedge (\neg d \vee w \vee t).$$

It is easy to check that an assignment of x, y, z can be completed to an assignment of a, b, c, d that satisfies this formula if and only if C is true. By applying this construction to every clause of F, and by taking the conjunction of the formulas constructed in that way, we obtain a CNF-formula F' in which every clause has at most 3 literals whose satisfiability is equivalent to that of F.

The computation time reduces to writing the clauses, whose length is polynomial. Consequently, the whole reduction can be computed in polynomial time, and we proved, starting from SAT that 3-SAT is NP-complete. □

12.4.2 Proof of the NP-completeness of 3-COLORABILITY

Remember that a *colouring* of a graph is an assignment of colours to vertices of the graph such that no edge has its extremities of the same colour.

Definition 12.12 (3-COLORABILITY)

Input: *An undirected graph* $G = (V, E)$.

Answer: *Decide if there exists a colouring of the graph that uses at most* 3 *colours.*

Theorem 12.6 *The problem* 3-COLORABILITY *is* NP*-complete.*

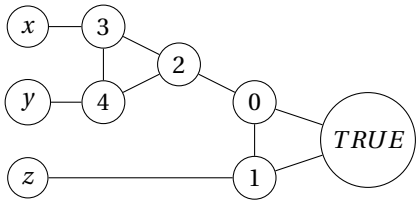
Proof: 3-COLORABILITY is in NP, since given a colour for each of the vertices, it is easy to check in polynomial time if this is a (valid) colouring with at most 3 colours.

We reduce 3-SAT to 3-COLORABILITY. We hence suppose that a conjunction of m clauses with 3 literals is given, over n variables, and we have to produce a graph (with the expected properties to get a reduction). As in all reductions from 3-SAT, we have to express two constraints: first, that a variable can only take either the value 0 or 1, and second, the evaluation rules of clauses.

We construct a graph with $3+2n+5m$ vertices, the first three (called distinguished vertices in what follows) are denoted by $TRUE$, $FALSE$, $DONTKNOW$. These three vertices are linked two by two in a triangle Thus, in a colouring these three vertices must all have different colours.

We associate a vertex to every variable and to the negation of every variable. To make sure that a variable takes the value $TRUE$ or $FALSE$, for every variable x_i, we add a triangle whose vertices are x_i, $\neg x_i$, and $DONTKNOW$. This makes sure that in a colouring we must have $colour(x_i) = colour(TRUE)$ and $colour(\neg x_i) = colour(FALSE)$, or $colour(x_i) = colour(FALSE)$ and $colour(\neg x_i) = colour(TRUE)$, where of course, $colour(v)$ denotes the colour of vertex v.

It remains to encode the evaluation rules of the clauses. To do so, we introduce the following subgraph, for every clause $x \vee y \vee z$:



It can be checked that if this pattern (where the three distinguished vertices with above mentioned triangles are implicit) is 3-colourable, then the vertices 0 and 1 are $colour(FALSE)$ and $colour(DONTKNOW)$. IF 1 is $colour(FALSE)$ since a vertex corresponding to a variable must be $TRUE$ or $FALSE$, we have $colour(z) = colour(TRUE)$. If 0 is $colour(FALSE)$, then 2 cannot be $colour(FALSE)$, so 3 or 4 is, and the corresponding variable is coloured $colour(TRUE)$.

Conversely, if one of the variables is true, one can then easily construct a 3-colouration of the pattern.

Consider then the graph formed of the three distinguished vertices, of the triangles formed on these variables, and the given patterns. If this graph is 3-colourable, then in particular every subgraph is colourable. The triangles of variables are in particular colourable. From a 3-colouring of the graph, one constructs a truth assignment by setting to 1 all variables coloured with $colour(TRUE)$. This assignment is

coherent (a variable and its negation have opposite values) and at least one literal for each clause is set to 1, according to the properties of the pattern above. Conversely, given an assignment of truth values, it is easy to deduce a 3-colouring of the graph.

The existence of a 3-colouring of the graph is hence equivalent to the satisfiability of the initial formula.

The reduction is clearly polynomial; hence, we have shown that 3-SAT reduces to 3-COLORABILITY. The latter is hence NP-complete. □

12.4.3 Proof of the Cook-Levin theorem

We cannot use the above method to prove the NP-completeness of SAT, as we do not now at this moment any NP-complete problem to reduce from.

We need to do the proof in another way, by coming back to the definition of NP-completeness: one must first prove that SAT is in NP, and second that any other problem A from NP satisfies $A \leq \text{SAT}$.

The fact that SAT is in NP has already been established, see example 12.11.

Consider a problem A of NP, and an associated verifier V. The idea (which has similarities with the constructions of Chapter 10) is given a word w, to construct a formula of propositional calculus $\gamma = \gamma(u)$ which encodes the existence of an accepting computation of V on $\langle w, u\rangle$ for a certificate u.

We will build a series of formulas whose culminating point will be formula $\gamma = \gamma(u)$ that will code the existence of a sequence of configurations $C_0, C_1, \cdots, C_t$ of M such that:

- C_0 is the initial configuration of V on $\langle w, u\rangle$;
- C_{i+1} is the successor configuration of C_i, according to the transition function δ of Turing machine V, for $i < t$;
- C_t is an accepting configuration.

In other words, the existence of a valid space-time diagram corresponding to a computation of V on $\langle w, u\rangle$.

By observing that the obtained propositional formula γ remains of size polynomial in the size of w, and can indeed be obtained by a polynomial algorithm from w, we will have shown the theorem: Indeed, we will get $w \in L$ if and only if there exists u that satisfies $\gamma(u)$, that is to say $A \leq \text{SAT}$ via the function f that to w associates $\gamma(u)$.

It only remains to provide the tedious details of the construction of formula $\gamma(u)$. By hypothesis, V runs in time $p(n)$ polynomial in the size n of w. In that time, V cannot move its head more than $p(n)$ cells to the left or $p(n)$ cells to the right. We can hence restrict to a sub-rectangle of size $(2 * p(n) + 1) \times p(n)$ from the space-time diagram of the computation of V on $\langle w, u\rangle$, see Figure 12.5.

The cells of array $T[i, j]$ corresponding to the space-time diagram are elements of finite set $C = \Gamma \cup Q$. For every $1 \leq i \leq p(n)$ and $1 \leq j \leq 2 * p(n) + 1$ and for every $s \in C$, we define a propositional variable $x_{i,j,s}$. If $x_{i,j,s}$ has the value 1, that means that the cell $T[i, j]$ contains s.

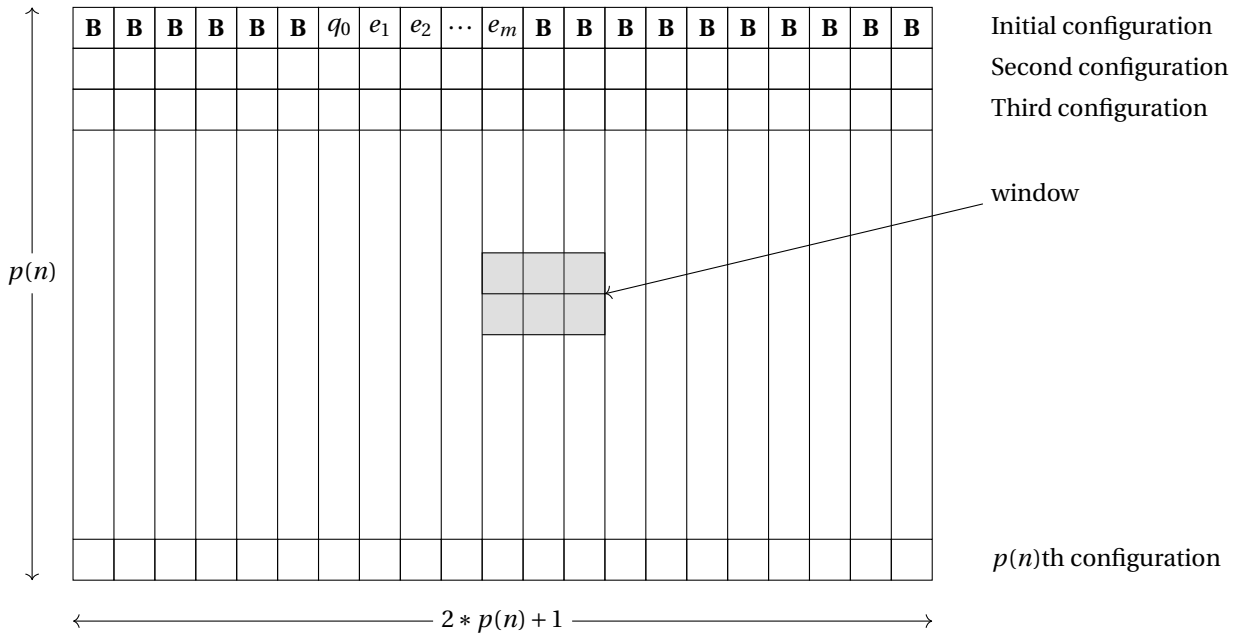


Figure 12.5: A $(2p(n)+1) \times p(n)$ array that codes the space-time diagram of the computation of V on $\langle w, u \rangle$.

The formula Γ is the conjunction of 4 formulas CELL $\wedge$ START $\wedge$ MOVE $\wedge$ HALT.

The formula CELL is there to guarantee that there is exactly one symbol in every cell.

$$\text{CELL} = \bigwedge_{1\le i\le p(n), 1\le j\le 2p(n)+1} \left[\left(\bigvee_{s\in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{s,t\in C, s\neq t} (\neg x_{i,j,s} \vee \neg x_{i,j,t}) \right) \right].$$

The symbols $\bigwedge$ and $\bigvee$ denote the iteration of corresponding symbols $\wedge$ and $\vee$. For example, $\bigvee_{s\in C} x_{i,j,s}$ is a shortcut for formula $x_{i,j,s_1} \vee \cdots \vee x_{i,j,s_l}$ if $C = \{s_1, \cdots, s_l\}$.

If we write the word $e_1 e_2 \cdots e_m$ for the word $\langle w, u\rangle$, the formula START guarantees that the first line corresponds to the initial configuration of V on $\langle w, u\rangle$.

$$\text{START} = x_{1,1,\mathbf{B}} \vee x_{1,2,\mathbf{B}} \vee \cdots \vee x_{1,p(n)+1,q_0} \vee x_{1,p(n)+2,e_1} \vee \cdots \vee x_{1,p(n)+m+1,e_m}$$

$$\vee x_{1,p(n)+m+2,\mathbf{B}} \vee \cdots \vee x_{1,2p(n)+1,\mathbf{B}}.$$

The formula HALT guarantees that one line corresponds to an accepting configuration.

$$\text{HALT} = \bigvee_{1\le i\le p(n), 1\le j\le 2p(n)+1} x_{i,j,q_a}.$$

Finally, the formula MOVE expresses that all 3×2 sub-rectangles from array T are legal windows: see the notion of legal window from Chapter 7.

$$\text{MOVE} = \bigwedge_{1\le i\le p(n), 1\le j\le 2p(n)+1} \text{LEGAL}_{i,j},$$

where $\text{LEGAL}_{i,j}$ is a positional formula that expresses that the 3×2 subformula at position i, j is a legal window:

$$\text{LEGAL}_{i,j} = \bigwedge_{(a,b,c,d,e,f)\in \text{WINDOW}} (x_{i,j-1,a} \wedge x_{i,j,b} \wedge x_{i,j+1,c} \wedge x_{i+1,j-1,d} \wedge x_{i+1,j,e} \wedge x_{i,j+1,f}),$$

where WINDOW is the set of 6-tuple (a, b, c, d, e, f) such that if the three elements of Σ represented respectively by a, b and c appear consecutively in a configuration C_i, and if d, e, f appear consecutively at the same position C_{i+1}, then this is coherent with transition function δ of Turing machine M.

This completes the proof, noting that each of the formulas can be written easily (and hence can be produced in polynomial time from w) and that they remain of size polynomial in the size of w.

12.5 Some other results from complexity theory

In this section, we give several additional important results on time complexity.

12.5.1 Decision vs. Construction

Let us start by a remark about the hypothesis that we did on the choice of restricting to decision problems

We have talked up to now only about *decision* problems, that is problems whose answer is either true or false (for example: "given a formula F decide if the formula F is satisfiable") in contrast to problems that consist in *producing an object with a property* (for example: given a formula F, produce an assignment of the variables that makes it true if there exists one).

Clearly, producing a solution is at least as hard as deciding if there exists one, and hence if P $\neq$ NP, none of the two problems has a solution in polynomial time, and the same holds for any NP-complete problem.

However, if P = NP, it turns out that we can also produce a solution:

Theorem 12.7 *Assume that* P = NP. *Let L be a problem of* NP *and V the associated verifier. One can construct a Turing machine that on any input $w \in L$ produces in polynomial time a certificate u for w for verifier V.*

Proof: Let us start by proving the theorem for L being the satisfaction problem of propositional formula (so the problem SAT). Assume P = NP. Then one can then test if a propositional formula F with n variables is satisfiable or not in polynomial time. If it is satisfiable, then one can fix its first variable to 0 and test if the obtained formula F_0 is satisfiable. If it is, then we write 0 and then restart recursively with this formula F_0 with $n-1$ variables. Otherwise, necessarily any certificate must have its first variable set to 1. Write 1 and start recursively with formula F_1 whose first variable is fixed to 1, and that has $n-1$ variables. Since it is easy to check if a formula without any variable is satisfiable, by this method, a correct certificate will be produced.

Now if L is an arbitrary language of NP, we can use the fact that the reduction produced by the proof of the Cook-Levin theorem is a *Levin* reduction: Not only do we have $w \in L$ if and only if $f(w)$ is a satisfiable formula, but one can find a certificate for w from a certificate of the satisfiability of formula $f(w)$. One can then use the previous algorithm to find the certificate for L. □

What we used in the above proof is the fact that the satisfiability problem of a CNF-formula is *self-reducible* to instances of lower size.

12.5.2 Hierarchy theorems

We say that a function $f(n) \geq n\log(n)$ is *time constructible*, if the function that sends 1^n to the binary representation of $1^{f(n)}$ is computable in time $\mathcal{O}\left(f(n)\right)$.

Most of the usual functions are time constructible, and in practice this is not really a restriction.

Remark 12.9 *For example, $n\sqrt{n}$ is time constructible: On input 1^n, one starts by counting the number of 1 in binary. One can use for that a counter, which remains of size $\log(n)$, that one increments: This is hence done in time $\mathcal{O}\left(n\log(n)\right)$*

since one uses at most $\mathcal{O}\big(\log(n)\big)$ steps for every letter of the input word. One can then compute $\lfloor n\sqrt{n}\rfloor$ in binary from the representation of n. Any standard method for doing so runs in time $\mathcal{O}\big(n\log(n)\big)$, since the size of the involved numbers is $\mathcal{O}\big(\log(n)\big)$.

Theorem 12.8 (Time Hierarchy theorem) *For every time constructible function $f : \mathbb{N} \to \mathbb{N}$, there exists a language L that is decidable in time $\mathcal{O}\big(f(n)\big)$ but not in time $o(f(n)/\log f(n))$.*

Proof: The proof is a generalization of the idea of the proof of Theorem 14.11 of next chapter: We invite our reader to wait and start by this latter proof.

We prove a version weaker than the statement above. Let $f : \mathbb{N} \to \mathbb{N}$ be a time constructible function.

One considers the (very artificial) language L that is decided by the following Turing machine B:

- on an input w of size n, B computes $f(n)$ and memorize $\langle f(n)\rangle$ the binary encoding of $f(n)$ in a binary counter c;
- If w is not of the form $\langle A\rangle 10^*$, for some Turing machine A, then Turing machine B rejects.
- Otherwise, B simulates A on the word w for $f(n)$ steps to determine whether A accepts in a time less than $f(n)$:
  - If A accepts in this time, then B rejects;
  - otherwise B accepts.

In other words, B simulates A on w, step by step, and decrements the counter c at each step. If this counter reaches 0 or if A rejects, then B accepts. Otherwise, B rejects.

By the existence of a universal Turing machine, there exist integers k and d such that L is decided in time $d \times f(n)^k$.

Suppose that L is decided by a Turing machine A in time $g(n)$ with $g(n)^k = o(f(n))$. There must exists an integer n_0 such that for $n \geq n_0$, we have $d \times g(n)^k < f(n)$.

As a consequence, the simulation of A by B will indeed be complete on some input of size n_0 or more.

Consider what happens when B is run on the input $\langle A\rangle 10^{n_0}$. Since this input is of size greater than n_0, B answers the opposite of Turing machine A on the same input. Hence B and A are not deciding the same language, and hence Turing machine A is not deciding L, which leads to a contradiction.

As a consequence L is not decidable in time $g(n)$ for any function $g(n)$ with $g(n)^k = o(f(n))$.

The theorem is a generalization of this idea. The (inverse) factor $\log(f(n))$ comes from the construction of a universal Turing machine that is more efficient than the one considered in this document, introducing only a logarithmic time slow-down.

□

Formulating the above theorem differently, we get:

Theorem 12.9 (Time Hiearchy theorem) *Let $f, f' : \mathbb{N} \to \mathbb{N}$ be time constructible functions such that $f(n)\log(f(n)) = o(f'(n))$. Then the inclusion* TIME(f(n)) $\subsetneq$ TIME(f'(n)) *is strict.*

We obtain for example:

Corollary 12.4 $\text{TIME}(n^2) \subsetneq \text{TIME}(n^{\log n}) \subsetneq \text{TIME}(2^n)$.

We define:

Definition 12.13 *Let*

$$\text{EXPTIME} = \bigcup_{c \in \mathbb{N}} \text{TIME}(2^{n^c}).$$

We obtain:

Corollary 12.5 P $\subsetneq$ EXPTIME.

Proof: Any polynomial becomes eventually negligible smaller than 2^n, and hence P is a subset of $\text{TIME}(2^n)$. Now, $\text{TIME}(2^n)$, that contains all P is strict subset of, for example, $\text{TIME}(2^{n^3})$, that is included in EXPTIME. □

12.5.3 EXPTIME and NEXPTIME

Consider

$$\text{EXPTIME} = \bigcup_{c \geq 1} \text{TIME}(2^{n^c})$$

and

$$\text{NEXPTIME} = \bigcup_{c \geq 1} \text{NTIME}(2^{n^c}).$$

We can prove the following result (and this is not hard):

Theorem 12.10 *If* EXPTIME $\neq$ NEXPTIME *then* P $\neq$ NP.

12.6 One the meaning of the $P = NP$ question

We make a digression about the meaning of the P vs. NP question in relation to proof theory and other chapters of this document.

One can see NP as the class of languages such that testing the containment to it is equivalent to determining if there is a *short* (polynomial size) certificate. This can

be related to the existence of a proof in mathematics. Indeed, in its own principle, mathematical deduction consists in proving theorems starting from axioms.

One expects that the validity of a proof is easy to check: one only needs to check that each line of the proof is a consequence of the previous lines, in the proof system. Actually, in most of the axiomatic proof systems (for example in all the proof systems we have seen) this verification can be done in a time that remains polynomial in the length of the proof.

Consequently, the following decision problem is NP for all the particular axiomatic usual proof systems $\mathscr{A}$, and in particular for the one $\mathscr{A}$ that we have seen for the predicate calculus.

$$\text{THEOREMS} = \{\langle \phi, \mathbf{1}^n \rangle | \phi \text{ has a proof of length } \leq n \text{ in system } \mathscr{A}\}.$$

We leave to our reader the following exercise:

Exercise 12.1 *The set theory of* Zermelo-Fraenkel *is one of the axiomatic systems that allows axiomatizing mathematics with a finite description. (Even without knowing all the details of the set theory of* Zermelo-Fraenkel*) argue at a high level that the problem* THEOREMS *is* NP*-complete for the set theory of* Zermelo-Fraenkel.

Hint: the satisfiability of a Boolean circuit is particular statement.

In other words, in virtue of Theorem 12.7, the P = NP question is the one (that has been asked for the first time by Kurt Gödel) to know whether there exists a Turing machine that is able to produce a mathematical proof of all statements ϕ in a time polynomial in the length of its proof.

Does this seem reasonable?

What is the meaning of the NP = coNP **question?** Remember that coNP is the class of languages whose complement is in NP. The question NP = coNP, is related to the existence of short proofs (of certificates) for statements that do not seem to have one: for example, it is easy to prove that propositional formula is satisfiable (one produces a valuation of its inputs, that one can encode in a proof that says that by propagation of the inputs towards the outputs, that the circuit outputs 1). On the other hand, in the general case, it is not clear how to write a short proof that a given propositional formula is not satisfiable. If NP = coNP, there must always exist one: The question is related to the existence of way proving the non-satisfiability of a propositional formula different from usual methods.

One can formulate equivalent statements for all the mentioned NP-complete problems.

12.7 Exercises

Exercise 12.2 *Prove that class* P *is closed under union, concatenation and complement.*

Exercise 12.3 *Prove that class* NP *is closed under union and concatenation.*

Exercise 12.4 *(solution on page 238) Prove that if* NP *is different from its complement then* $P \neq NP$.

Exercise 12.5 *Prove that if* $P = NP$ *then all languages* $A \in P$ *except for* $A = \emptyset$ *and* $A = \Sigma^*$ *are* NP*-complete.*

12.8 Bibliographic notes

Suggested readings To go further with the notions of this chapter, we suggest to read the books [Sipser, 1997], [Papadimitriou, 1994] [Lassaigne & de Rougemont, 2004].

A reference book that contains the last results of the domain is [Arora & Barak, 2009].

Bibliography This chapter contains some standard results in complexity. We essentially used their presentation in [Sipser, 1997], [Poizat, 1995], [Papadimitriou, 1994]. The last part "discussion" is taken from [Arora & Barak, 2009].

A *clique of a graph* $G = (V, E)$ is a subset $V' \subset V$ such that any pair of vertices of V' are linked by some edge of G.

Theorem 12.11 *Given a graph* G*, and some integer* k*, the problem to decide whether it admits a clique of size* $\geq k$ *is* NP*-complete.*

A *covering subset* of a graph $G = (V, E)$ is a subset $V' \subset V$ such that any pair of edges of E has at least one of its extremity in V'.

Theorem 12.12 *Given a graph* G*, and some integer* k*, the problem to know whether it admits a covering subset of size* $\leq k$ *is* NP*-complete.*

An *Hamiltonian path of a graph* G is a path that goes once and only once through every vertex of G. The problem of Hamiltonian path is to determine if a graph has some Hamiltonian path.

Theorem 12.13 *The Hamiltonian problem is* NP*-complete.*

The subset sum problem consists, given a finite set S of integers and some integer s, determining whether there exists a subset $X \subset S$ such that $\sum_{i \in X} i = s$.

Theorem 12.14 *The subset sum problem is* NP*-complete.*

Chapter 13

Some NP-complete problems

Now that we have established the NP-completeness of a few problems, we are going to prove that a very huge number of problems are NP-complete.

The book [Garey & Johnson, 1979] listed more than 300 NP-complete problems in 1972. We do not have the ambition of presenting so many problems, but we will list some famous NP-complete problems: Our main purpose is actually to provide some examples of proofs of NP-completeness.

13.1 Some NP-complete problems

13.1.1 Around SAT

Recall the following theorem proved in previous chapter.

Definition 13.1 (3-SAT)

Input: *A set of variables* $\{x_1, \cdots, x_n\}$ *and a formula* $F = C_1 \wedge C_2 \cdots \wedge C_\ell$ *with* $C_i = y_{i,1} \vee y_{i,2} \vee y_{i,3}$, *where for every* i, j, $y_{i,j}$ *is either* x_k, *or* $\neg x_k$ *for one of the* x_k.

Answer: *Decide whether* F *is satisfiable: that is, decide if there exist* $x_1, \cdots, x_n \in \{0,1\}^n$ *such that* F *evaluates to true with this value of its variables* $x_1, \cdots, x_n$.

Theorem 13.1 *The problem* 3-SAT *is* NP*-complete.*

Remark 13.1 *The problem* $2-$SAT, *where we would consider clauses with two literals, is in* P.

Definition 13.2 (NAESAT)

Input: *A set of variables* $\{x_1, \cdots, x_n\}$ *and a set of clauses* $y_{i,1} \vee \cdots \vee y_{i,k_i}$, *where for all* i, j, $y_{i,j}$ *is either* x_k, *or* $\neg x_k$ *for some of the* x_k.

Answer: *Decide if there is some assignment of the variables* $x_i \in \{0,1\}$ *in such a way that every clause contains at least one true literal and at least one false literal (that is, for every* i, *there is a* j *and a* k *with* $y_{i,j} = 1$ *and* $y_{i,k} = 0$*).*

Theorem 13.2 *The problem* NAESAT *is* NP*-complete.*

Proof: The problem is in NP since, given some assignment of the variables, it is easy to check in polynomial time that the property holds for these values of the variables.

We will reduce SAT to NAESAT. Let F a formula of SAT on the variables $\{x_1, \cdots, x_n\}$. We add a unique distinct variable z and we construct the clauses for NAESAT by replacing each clause $C_i = y_{i,1} \vee \cdots \vee y_{i,k}$ of F by $C'_i = y_{i,1} \vee \cdots \vee y_{i,k} \vee z$.

This transformation can be realized in polynomial time.

If the given instance of SAT is satisfiable, the same assignment of variables fixing for z the value 0 is a valid assignment for NAESAT.

Conversely, suppose that the constructed instance of NAESAT is satisfiable. If the truth value of z in the corresponding assignment is 0, then the values of the variables x_i in the assignment give some valid assignment for the original formula F (for the instance of SAT). If, on the contrary z values 1, then change all the values of all the variables in the assignment. The assignment remains valid for NAESAT since at least one literal by clause in the initial assignment values 0, and hence values now 1, whereas z values 0. We have built an assignment in which z values 0, and by previous case the initial instance of SAT is satisfiable.

We have indeed proved the equivalence between satisfiability of F and the corresponding instance of NAESAT. Hence NAESAT is NP-complete. □

By using the same reduction for instances 3SAT, we get that NAE4SAT is NP-complete: NAE4SAT is NAESAT restricted to formulas with 4 literals in each clause. We can actually prove, that this holds for the version with three literals.

Corollary 13.1 NAE3SAT *is* NP*-complete.*

Proof: We will reduce NAE4SAT to NAE3SAT. Let $C = x \vee y \vee z \vee t$ be a clause with 4 literals. We introduce a new variable u_C, and we construct the two clauses $C_1 = x \vee y \vee \neg u_C$ and $C_2 = z \vee t \vee u_C$. Doing so for all the clauses, we clearly construct an instance F' of NAE3SAT in polynomial time.

Suppose that F' is some positive instance of NAE3SAT, and consider the assignment of the corresponding truth value. If $u_C = 0$, then x or y is 0, and z or t is 1, so $x \vee y \vee z \vee t$ has at least a literal 1 and at least one literal 0; Similarly, if $u_C = 1$; So F is a positive instance of NAE4SAT.

Conversely, if F is a positive instance of NAE4SAT, consider the corresponding truth assignment. In $x \vee y \vee z \vee t$, if x and y are both set to 1, set u_C to 1; otherwise if

x and y are both set to 0, set u_C to 0; otherwise, set u_C to the suitable truth value for the clause $u_C \vee z \vee t$. That produces an assignment that proves that F' is a positive instance of NAE3SAT.

Once again, NAE3SAT is in NP from definition, as a value for the variable is clearly a certificate that can be checked in polynomial time. □

13.1.2 Around INDEPENDANT SET

Definition 13.3 (INDEPENDANT SET**)**

Input: *An (undirected) graph $G = (V, E)$ and some integer k.*

Answer: *Decide whether there exists $V' \subset V$, with $|V'| = k$, such that $u, v \in V' \Rightarrow (u, v) \notin E$.*

Theorem 13.3 *The problem* INDEPENDANT SET *is* NP*-complete.*

Remark 13.2 *Such an independent set V' is sometimes also called a* stable set *(or* stable*).*

Proof: INDEPENDANT SET is indeed in NP, since giving V' provides a certificate that can be easily checked in polynomial time.

We reduce the problem 3-SAT to INDEPENDANT SET, that is to say, given some formula F of type 3-SAT, we construct in polynomial time a graph G in such a way that the existence of a stable set in G is equivalent to the existence of a truth assignment that satisfies F.

Let $F = \bigwedge_{1 \leq j \leq k} (x_{1j} \vee x_{2j} \vee x_{3j})$. We construct a graph G with $3k$ vertices, one for each occurrence of a literal in a clause.

- For every variable x_i of 3-SAT, G has an edge between every vertex associated to a literal x_i and every vertex associated to a literal $\neg x_i$ (and so an independent set of G corresponds to a truth assignment of a subset of variables);
- For every clause C, we associate a triangle: for example for a clause of F of the form $C = (x_1 \vee \neg x_2 \vee x_3)$, then G has the edges $(x_1, \neg x_2)$, $(\neg x_2, x_3)$, (x_3, x_1) (by doing, an independent set of G must contain at most one of the vertices associated to clause C).

Let k be the number of clauses in F. One proves that F is satisfiable if and only if G has an independent set of size k.

Indeed, if F is satisfiable, consider an assignment of the variables that satisfies F. For every clause C of F, select y_C a literal of C that is set to true by the assignment: that defines k vertices defining an independent set of G.

Conversely, if G has an independent set of size k, then it has necessarily a vertex in every triangle. This vertex corresponds to a literal that makes the associated

clause true, and this forms an assignment of the variables that is consistent by construction of the edges.

The reduction is clearly polynomial. □

Two classical problems are related to INDEPENDANT SET.

Definition 13.4 (CLIQUE)

Input: *An (undirected) graph $G = (V, E)$ and some integer k.*

Answer: *Decide if there exists $V' \subset V$, with $|V'| = k$, such that $u, v \in V' \Rightarrow (u, v) \in E$.*

Theorem 13.4 *The problem* CLIQUE *is* NP*-complete.*

Proof: The reduction from INDEPENDANT SET consists in going to the complementary on the edges. Indeed, it is sufficient to observe that a graph $G = (V, E)$ has an independent set of size k if and only if complementary graph $\overline{G} = (V, \overline{E})$ (where $\overline{E} = \{(u, v) | (u, v) \notin E\}$) has a clique of size k. □

Definition 13.5 (VERTEX COVER)

Input: *An (undirected) graph $G = (V, E)$ and some integer k.*

Answer: *Decide if there exists $V' \subset V$, with $|V'| = k$, such that every edge of G has at least one of its extremity in V'.*

Theorem 13.5 *The problem* VERTEX COVER *is* NP*-complete.*

Proof: The reduction from INDEPENDANT SET consists in considering the complementary on the vertices. □

Definition 13.6 (MAXIMAL CUT)

Input: *An (undirected) graph $G = (V, E)$ and some integer k.*

Answer: *Decide if there exists a partition $V = V_1 \cup V_2$ such that the number of edges between V_1 and V_2 is at least k.*

Theorem 13.6 *The problem* MAXIMAL CUT *is* NP*-complete.*

Proof: We reduce NAE3SAT to MAXIMAL CUT. Suppose an instance of NAE3SAT is given, in which we can suppose without loss of generality that every clause does not contain simultaneously a variable and its complementary. Replacing $u \vee v$ by $((u \vee v \vee w) \wedge (u \vee v \vee \neg w))$ if needed, we can suppose that every clause contains exactly 3 literals. Furthermore, if we have $(u \vee v \vee w)$ and $(u \vee v \vee z)$, we can, by introducing two variables t_1 and t_2 and by proceeding as we did to reduce NAE4SAT

to NAE3SAT, rewrite these two clauses as $(u \vee t_1 \vee t_2) \wedge (v \vee w \vee \neg t_1) \wedge (v \vee z \vee \neg t_2)$. In other words, we can suppose that two given clauses have at most one variable in common.

We denote by $x_1, \cdots, x_n$ the variables of the formula F.

We will construct a graph $G = (V, E)$ in the following way: G has $2n$ vertices and every variable u of F is corresponding to two vertices u and $\neg u$. G has an edge between every couple of vertices (u, v) such that u and v appear in the same clause, and an edge between the vertices u and $\neg u$ for every variable u.

The reductions in the first paragraph of the proof permit to state that to every clause corresponds a triangle and that two of these triangles have distinct edges.

If we denote by n the number of variables and by m the number of clauses, the graph G has $2n$ vertices and $3m + n$ edges. It is easy to see that the number of edges in a cut corresponding to a valid NAE3SAT assignment is $2m + n$: The edge between u and $\neg u$ for every variable u, and two edges of the triangle uvw for every clause $u \vee v \vee w$.

Conversely, every cut of G has at most $2m + n$ edges, since a cut can not include more than the number of edges for a given triangle associated to a clause. Consequently, a cut of value $2m + n$ provides immediately a valid NAE3SAT assignment.

In other words, solving an instance of NAE3SAT is equivalent to solve MAXIMAL CUT on $(G, 2m + n)$.

The reduction is polynomial. Now MAXIMAL CUT is in NP since giving V_1 is a valid certificate that can be checked in polynomial time. □

13.1.3 Around HAMILTONIAN CIRCUIT

The problem HAMILTONIAN CIRCUIT is often at the source of the proof of NP-completeness of properties related to paths in graphs.

Definition 13.7 (HAMILTONIAN CIRCUIT**)**

Input: *An (undirected) graph $G = (V, E)$.*

Answer: *Decide if there exists a* Hamiltonian circuit *that is to say a path of G that goes exactly once through every vertex and that comes back to its starting point.*

Theorem 13.7 *The problem* HAMILTONIAN CIRCUIT *is* NP*-complete.*

Proof: We will prove that fact by reducing VERTEX COVER to this problem.

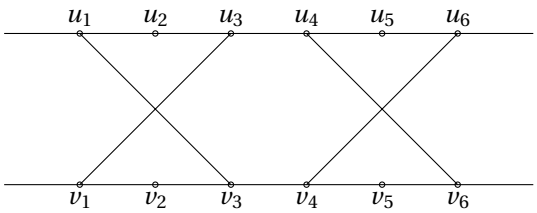
The idea consists, starting from an instance of VERTEX COVER, in constructing a graph in which every initial edge will be replaced by a "pattern" that admits exactly two Hamiltonian paths, that is to say paths that visit exactly once every vertex. One of this path will correspond to the case where the corresponding vertex belongs to the cover, the other where it does not belong.

There will remain to tell how to glue all these patterns so that a global Hamiltonian circuit corresponds exactly to a union of Hamiltonian paths through every

pattern, and to ensure that one obtains the good result to problem VERTEX COVER by solving HAMILTONIAN CIRCUIT in this graph.

We now go to the details: Denote $(G = (V, E), k)$ the considered instance of VERTEX COVER.

The pattern that we will use is the following:



To obtain a traversal of this pattern that goes exactly once through every vertex, only two solutions are possible: either a traversal in two steps, first $u_1u_2u_3u_4u_5u_6$ then later $v_1v_2v_3v_4v_5v_6$ (in one direction or the other); or a traversal in only one step $u_1u_2u_3v_1v_2v_3v_4v_5v_6u_4u_5u_6$ (or the same but switching the role of the u's and of the v's).

To every edge (u, v) of the graph G, we associate a pattern of this type, by putting in correspondence the vertex u_i's at side u and the vertex v_i's at side v. We then link together all the sides of all the patterns corresponding to a same vertex. We construct consequently a chain associated to a given vertex, with still two free "outputs". We then link again these two "outputs" to k new vertices $s_1, \cdots, s_k$. We denote by H the graph constructed in that way in the remaining part of this proof.

Suppose now that a cover of the initial graph of size k, whose vertices are $\{g_1, \cdots, g_k\}$ is given. One can then construct a Hamiltonian circuit of H in the following way:

- starts from s_1;
- follow the chain g_1 in the following way. When one traverses an edge (g_1, h), if h is also in the cover, simply cross the side g_1; otherwise, cross the two sides simultaneously;
- once the chain g_1 is finished, come back to s_2 and restart by g_2 and so on.

It is clear that every vertex s_k is traversed exactly once.

Consider a vertex h of the pattern corresponding to an edge (u, v). One can always suppose that u is in the cover, say $u = g_1$. It follows that if h is on the same side as u, h will be reached at least once. We see, according to second item, that it will not be reached later on. If h is on the same side as v, and if v is not in the cover, h is visited along the traversal of u. If $v = g_i$, h is crossed by the traversal of the chain corresponding to g_i and at this moment only. We hence have indeed a Hamiltonian circuit.

Conversely, suppose that we have a Hamiltonian circuit of H. The construction of our pattern implies that, coming from vertex s_i, one traverses completely a chain u and then one goes to a vertex s_j. One then traverses k chains; the k corresponding vertices form a cover. Indeed, if (u, v) is an edge, the corresponding pattern is

traversed by the Hamiltonian path; But it can be only a traversal of a loop corresponding to one of the extremities of the edge.

Finally, the reduction is trivially polynomial. HAMILTONIAN CIRCUIT is hence NP-complete. □

As in previous section, we can then deduce the NP-completeness of many variants.

Definition 13.8 (TRAVELING SALESMAN**)**

Input: *A couple (n, M), where M is a matrix $n \times n$ of integers and some integer k.*

Answer: *Decide if there exists some permutation π of $[1, 2, \cdots, n]$ such that*

$$\sum_{1 \leq i \leq n} M_{\pi(i)\pi(i+1)} \leq k.$$

(here indices are taken modulo n so that it makes sense)

Corollary 13.2 *The problem* TRAVELING SALESMAN *is* NP*-complete.*

Remark 13.3 *This problems has its name, since it can be seen as establishing the planning of visits of a traveling salesman that must visit n town, whose distances are given by matrix M, in less than k kilometers.*

Proof:

We reduce HAMILTONIAN CIRCUIT to TRAVELING SALESMAN. To do so, given a graph $G = (V, E)$, consider $V = \{x_1, \cdots, x_n\}$. We consider then the matrix M $n \times n$ of integers such that

$$M_{i,j} = \begin{cases} 1 & \text{if } (i, j) \in E; \\ 2 & \text{otherwise.} \end{cases}$$

We then claim that HAMILTONIAN CIRCUIT(V, E) is true if and only if TRAVELING SALESMAN(n, M, n). Indeed:

If there exists a Hamiltonian circuit in G, one can then indeed construct the permutation π as description the order of traversal of the vertices of graph G: By construction, the sum of the distances of the edges on this circuit will value n.

Conversely, given some permutation with this property, the fact that the n terms of the sum value at least 1 implies that they are all equal to 1, and so that the edges $(\pi(i), \pi(i+1))$ exist in the graph G: we hence have a Hamiltonian circuit. □

Definition 13.9 (LONGEST CIRCUIT**)**

Input: *An (undirected) graph $G = (V, E)$, with a distance on each edge, and some integer r.*

Answer: *Decide if there exists a circuit G that does not visit twice the same vertex*

whose length is $\geq r$.

Corollary 13.3 *The problem* LONGEST CIRCUIT *is* NP*-complete.*

Proof: We construct a reduction from HAMILTONIAN CIRCUIT: to do so, to every graph for HAMILTONIAN CIRCUIT, we associate the length 1 to every edge. Finding a Hamiltonian cycle is then trivially identical to finding a circuit of length $\geq n$ in the graph. □

13.1.4 Around 3-COLORABILITY

We have defined what is called a *(well) colouring* of a graph in the previous chapter. The smallest integer k such that a graph can be (well) colored is called the *chromatic number* of the graph.

It is known that planar graphs are always colorable with 4 colors.

We recall the following result established in previous chapter.

Definition 13.10 (3-COLORABILITY)

Input: *An (undirected) graph* $G = (V, E)$.

Answer: *Decide if there exists a colouring of the graph that uses at most* 3 *colors.*

Theorem 13.8 *The problem* 3-COLORABILITY *is* NP*-complete.*

13.1.5 Around SUBSET SUM

Definition 13.11 (SUBSET SUM)

Input: *A finite sequence of integers* $x_1, x_2, \ldots, x_n$ *and some integer* t.

Answer: *Decide if there exists* $E \subset \{1, 2, \ldots, n\}$ *such that* $\sum_{i \in E} x_i = t$.

Theorem 13.9 *The problem* SUBSET SUM *is* NP*-complete.*

Proof: SUBSET SUM is in NP since giving E is a certificate that can be checked in polynomial time.

We use a reduction from VERTEX COVER.

Suppose that a graph $G = (V, E)$ is given in which one wants to determine if there exist a vertex cover of size k. Number the vertices and the edges. Let $B = (b_{ij})$ the incident matrix vertex-edges, that is to say $b_{ij} = 1$ if edge i is incident to vertex j, $b_{ij} = 0$ otherwise.

Consider $b \geq 4$. We will construct a sequence F of integers: For every edge i, we add integer b^i to F. For every vertex j we add integer a_j to F, where $a_j = b^m + \sum_{i=0}^{m-1} b_{i,j} b^i$.

We consider then

$$t = kb^m + \sum_{i=0}^{m-1} 2b^i. \tag{13.1}$$

Consider a cover S of G of cardinality k. Construct the subsequence made of the a_j's such that $j \in S$, and of the b^i's such that exactly one of the two extremities of edge i is in S. Then the sum of the elements of this subsequence is t: Indeed, we sum k times the term b^m and each edge i with two extremities in S contributes to b^i for each of its two extremities by the part of the sum related to the a_j's, and each edge i with one extremity in S contributes to b^i once by the part of the sum related to the a_j's, and once by the integer b^i.

Conversely, suppose that we have a subsequence of sum t. We split the subsequence into X_1 made of the elements $x_i \geq m$, and into X_2 made of the elements $x_i < b^m$. To every element of X_1, is associated some vertex j: We take S to be the set of such vertices associated to elements of X_1. Since $kb^m \leq t < (k+1)b^m$, necessarily the size of S is k.

It remains to show that S is a cover. As in every sum of elements of F, there are at most three terms b^i for $i < m$, and as $b \geq 4$, no carry can be produced in the addition (except possibly for the coefficient of b^m that can exceed $b-1$). Consequently, the number of occurrences of term b^i can be read on equation (13.1), and it must necessarily be 2 for $i < m$.

That means that each edge i must necessarily have at least one of its extremity in S, since otherwise the number of occurrences of term b^i would be 0 or 1.

We have indeed reduced VERTEX COVER to SUBSET SUM.

Indeed, it is easy to see that the reduction is done in polynomial time, and hence we have proved the theorem. □

We can deduce:

Definition 13.12 (KNAPSACK)

Input: *A set of weights $a_1, \cdots, a_n$, a set of values $v_1, \cdots, v_n$, a weight limit A, and some integer V.*

Answer: *Decide if there exists $E' \subset \{1, 2, \cdots, n\}$ such that $\sum_{i \in E'} a_i \leq A$ and $\sum_{i \in E'} v_i \geq V$.*

Corollary 13.4 *The problem* KNAPSACK *is* NP*-complete.*

Proof: From SUBSET SUM: Given $E = \{e_1, \cdots, e_n\}$ and t an instance of SUBSET SUM, one considers $v_i = a_i = e_i$, and $V = A = t$. □

Definition 13.13 (PARTITION)

Input: *A finite sequence of integers $x_1 x_2 \ldots x_n$.*

Answer: *Decide if there exists $E \subset \{1, 2, \ldots, n\}$ such that $\sum_{i \in E} x_i = \sum_{i \notin E} x_i$.*

Theorem 13.10 *The problem* PARTITION *is* NP*-complete.*

Proof: We will reduce SUBSET SUM to PARTITION. Let $(x_1 x_2 \dots x_n, t)$ be an instance of SUBSET SUM. We set $S = \sum_{1 \leq i \leq n} x_i$. Changing t in $S - t$ if needed (which is equivalent to change the obtained set in its complement), we can suppose that $2t \leq S$.

A first natural idea would consist in adding the element $u = S - 2t$ to the sequence. The result of a partition would then be two subsequences of sum $S - t$. One of the two would contain the element $S - 2t$, and hence by suppressing the latter, we would find a subsequence of sum t. Unfortunately, this reasoning fails if $S - 2t$ is already in the sequence.

Instead of doing so, we take the number $X = 2S$ and $X' = S + 2t$, and we apply PARTITION to the sequence $x_1, x_2, \dots, x_n, X, X'$. There exists a partition of this sequence if and only if there exists a subsequence of $x_1, x_2, \dots, x_n$ of sum t. Indeed, if there exists a partition of $x_1, x_2, \dots, x_n, X, X'$, there exists two subsets complementary of sum $2S + t$. Each of this two subsets must contain either X, or X', since otherwise its sum would exceed $2S + t$; So one of the two subsets contains X and not X'. By suppressing X in it, we must then obtain a subsequence F of $x_1, x_2, \dots, x_n$ of total sum t. Conversely, given such a subsequence F of total sum t, then the subsequence made of F and X, and its complement, constitutes a partition of $x_1, x_2, \dots x_n, X, X'$. We have indeed reduced SUBSET SUM to PARTITION.

It remains to justify that the reduction is indeed polynomial. The main part of the reduction is the computation of A and A', and this is indeed polynomial in the size of the inputs (the addition of k numbers of n bits can be done in time $\mathcal{O}(k \log n)$). □

13.2 Exercises

13.2.1 Polynomial variants

Exercise 13.1 *A graph $G = (V, E)$ is said* Eulerian *if there exists a cycle that goes through every edge of G exactly once.*

Prove that a connected graph is Eulerian if and only if all its vertices are of degree two.

Propose a polynomial algorithm to determine if a graph is Eulerian.

Exercise 13.2 *We consider* 2SAT.

1. *Propose an evaluation t that satisfies $\phi_1(u_1,u_2,u_3) = (u_2 \vee \neg u_3) \wedge (\neg u_2 \vee u_3) \wedge (u_2 \vee \neg u_1)$.*
2. *What happens for $\phi_2(u_1,u_2) = (u_2 \vee u_3) \wedge (\neg u_2 \vee u_3) \wedge (\neg u_3 \vee u_1) \wedge (\neg u_3 \vee \neg u_1)$?*
3. *Prove that $u \vee v = (\neg u \Rightarrow v) \wedge (\neg v \Rightarrow u)$.*

   Starting from an instance of 2SAT *we construct a directed graph $G_\phi = (V,E)$ with*

   - *a vertex for every literal;*
   - *and an arc for every implication (by transforming every clause into two implications).*
4. *Draw the graphs G_{ϕ_1} and G_{ϕ_2}.*
5. *Prove that there exists a variable u such that G_ϕ contains a cycle between u towards $\neg u$ in G, if and only if ϕ is not satisfiable.*
6. *Prove that* 2-SAT *can be solved in polynomial time.*

Exercise 13.3 *(solution on page 239) [Knights of the round table] Given n knights, and knowing all the pairs of fierce enemies among them, is it possible to position them in polynomial time around some round table in such a way that no pair of fierce enemy are sitting near each other.*

13.2.2 NP-completeness

Exercise 13.4 *(solution on page 239) A* Hamiltonian path *is a path that goes through every vertex of the graph exactly once.*

Prove that the following problem is NP*-complete:*

Input: *A (undirected) graph G of n vertices, two distinct vertices u and v of G.*

Answer: *Decide if G contains a Hamiltonian path whose extremities are vertices u and v.*

Exercise 13.5 *(solution on page 240)* *Prove that the following problem is* NP*-complete.*

Input: *An (undirected) graph G with n vertices, two distinct vertices u and v of G.*

Answer: *Decide if G contains a path of length $n/2$ between u and v.*

Exercise 13.6 *(solution on page 241)* *Prove that the following problem is* NP*-complete.*

Input: *A graph $G = (V, E)$, and some integer k.*

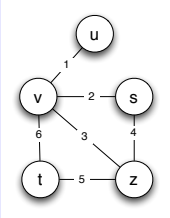
Answer: *Decide if there exists a tree covering all the vertices of G with at least k leaves.*

Exercise 13.7 *(solution on page 241) Let $G = (V, E)$ be a graph.*

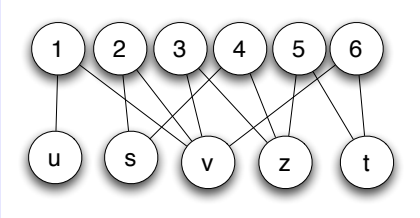
- *A* vertex cover *S of graph G is a subset of vertices such that all edges of G are incident to at least a vertex of S.*
- *A* dominating set *C of graph G is a subset of vertices such that every vertex is either in C or neighbour of a vertex of C.*

Let $G = (V, E)$ be a connected graph. We are going to construct a graph $G' = (V', E')$ from G such that

- $V' = V \cup E$;
- $E' = E \cup \{(v, a) | v \in V, a \in E, v$ *is a extremity of a in* $G\}$



Graph G



Graph G' (without the edges of G)

1. *Prove that if S is a vertex cover of G, then S is a dominating set of G'.*
2. *Prove that if S' is a dominating set of G', then there exists some vertex cover $S \subseteq V$ of graph G of cardinal less or equal to S'.*
3. *Express the problem of minimization of the dominating set as a decision problem.*
4. *Prove that this problem is in* NP.
5. *Prove that the problem is* NP*-complete.*

Exercise 13.8 *(solution on page 241) We will focus on the problem of k-centre: Given a set of towns whose distances are given, select k towns in order to put some warehouses in order to minimize the maximal distance from a town to the closest warehouse. Such a set of k town is called a k-center.*

The associated decision problem is the following:

Input: *A complete graph $K = (V, E)$ having a weight function w on the edges, and some strictly positive integers k and b.*

Answer: *Decide if exists a set S of vertices such that $|S| = k$ and such that every vertex v of V satisfies the following condition*

$$min\{w(v,u) : u \in S\} \leq b.$$

1. *Prove that k-CENTRE is in* NP.
2. *Prove that k-CENTRE is* NP*-complete, knowing that* DOMINANT *is* NP*-complete.*

13.3 Bibliographic Notes

Suggested readings The book [Garey & Johnson, 1979] provides a list of more than 300 NP-complete problems, and is rather pleasant to read. One can find updates of lists of complete problems on the web.

Bibliography This chapter is reproduced, and adapted from course book [Cori et al., 2010] from course INF550 of École Polytechnique.

Chapter 14

Space complexity

In this chapter, we will focus on another critical resource of algorithms: Memory. In complexity theory, when talking about memory as a resource, it is more commonly called *memory space* or simply *space*.

We will start by showing how one can measure the memory used by an algorithm. We will then introduce the main complexity classes considered in complexity theory.

14.1 Polynomial space

In this section, we will state a set of definitions and results without proofs. The proofs will be given in next section.

We introduce the analog of TIME(t(n)) for memory:

Definition 14.1 (SPACE(t(n))**)** *Let $t : \mathbb{N} \to \mathbb{N}$ be a function. We define* SPACE(t(n)) *as the class of problems (languages) that are decided by a Turing machine using $\mathcal{O}(t(n))$ cells of the tape, where n is the size of the input.*

14.1.1 Class PSPACE

We first consider the class of problems decided using polynomial space.

Definition 14.2 PSPACE *is the class of problems (languages) decided in polynomial space. In other words,*

$$\text{PSPACE} = \bigcup_{k \in \mathbb{N}} \text{SPACE}(n^k).$$

Remark 14.1 *As in Chapter 12, we observe that the notion of space is independent of the computational model we use. Consequently the use of the Turing*

machine model as the basis model for measuring space complexity is rather arbitrary in what follows.

We can also introduce the non-deterministic analog:

Definition 14.3 (NSPACE(t(n))) *Let $t : \mathbb{N} \to \mathbb{N}$ be a function. We define* NSPACE(t(n)) *as the class of problems (languages) that are accepted by a non-deterministic Turing machine using $\mathcal{O}(t(n))$ cells of the tape on every branch of the computation tree, where n is the size of the input.*

It would then natural to define:

$$\text{NPSPACE} = \bigcup_{k \in \mathbb{N}} \text{NSPACE}(\text{n}^{\text{k}}),$$

but it turns out that the complexity class NPSPACE is nothing but PSPACE.

Theorem 14.1 (Savitch theorem) NPSPACE = PSPACE.

The proof of this result can be found in Section 14.10.

14.1.2 PSPACE-complete problems

The class PSPACE has some complete problems: The problem QBF (sometimes also called QSAT) consists, given some propositional calculus formula in conjunctive normal form ϕ with the variables $x_1, x_2, \cdots, x_n$ (that is to say given an instance similar to an instance of SAT), to determine whether $\exists x_1 \forall x_2 \exists x_3 \cdots \phi(x_1, \cdots, x_n)$?

Theorem 14.2 *The* QBF *problem is* PSPACE*-complete.*

We will not prove this result in this document.

Strategic games on graphs lead natural birth to PSPACE-complete problems.

For example, the game GEOGRAPHY consists in taking as input a finite oriented graph $G = (V, E)$. Player 1 selects a node u_1 of the graph. The player 2 must then select a node v_1 such that there is some arc from u_1 to v_1. This is then the turn of player 1 to select another node u_2 such that there is an arc from v_1 to u_2, and so one. One does not have the right and so on. We don't have the right to come back twice to the same node. The first player that cannot continue the path $u_1 v_1 u_2 v_2 \cdots$ loses. The problem GEOGRAPHY consists, given some graph G and a node for player 1, in determining if there exists some winning strategy for player 1.

Theorem 14.3 *The problem* GEOGRAPHY *is* PSPACE*-complete.*

14.2 Logarithmic space

It turns out that the class PSPACE is huge and contains all of P and also all NP: Imposing a polynomial memory space is practice often little restrictive.

This is why one often considers more restictive space bounds, in particular logarithmic space. But this introduces some difficulties which show a problem in the definitions. Indeed, a Turing machine uses at least the cells that contain its input, and hence Definition 14.1 is not able to talk about functions $t(n) < n$.

This is why one changes this definition with the following convention: When one measures the memory space, by convention one does not count the cells containing the input.

To do so, properly, one must replace Definition 14.1 by the following.

Definition 14.4 (SPACE(t(n))) *Let $t : \mathbb{N} \to \mathbb{N}$ be a function. We define* SPACE(t(n)) *as the class of problems (languages) that are decided by a two tapes Turing machine:*

- *the first tape contains the input and is read-only: it can be read, but it cannot be written;*
- *the second is initially empty and read-write: it can be read and written, i.e., it is a usual tape;*

using $\mathcal{O}(t(n))$ cells of the second tape, where n is the size of the input.
We define NSPACE(t(n)) *with the analogous convention.*

Remark 14.2 *This new definition does not change anything to all for the previously introduced complexity classes. However, it the the following definitions meaningful.*

Definition 14.5 (LOGSPACE) *The class* LOGSPACE *is the class of languages (problems) decided by a Turing machine in logarithmic space. In other words,*

$$\text{LOGSPACE} = \text{SPACE}(\log(n)).$$

Definition 14.6 (NLOGSPACE) *The class* NLOGSPACE *is the class of languages (problems) decided by a non-deterministic Turing machine in logarithmic space. In other words.*

$$\text{NLOGSPACE} = \text{NSPACE}(\log(n)).$$

It turns out that.

Theorem 14.4 $\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$.

We furthermore know that $\text{NLOGSPACE} \subsetneq \text{PSPACE}$ but we do not know which of the intermediary inclusions are strict.

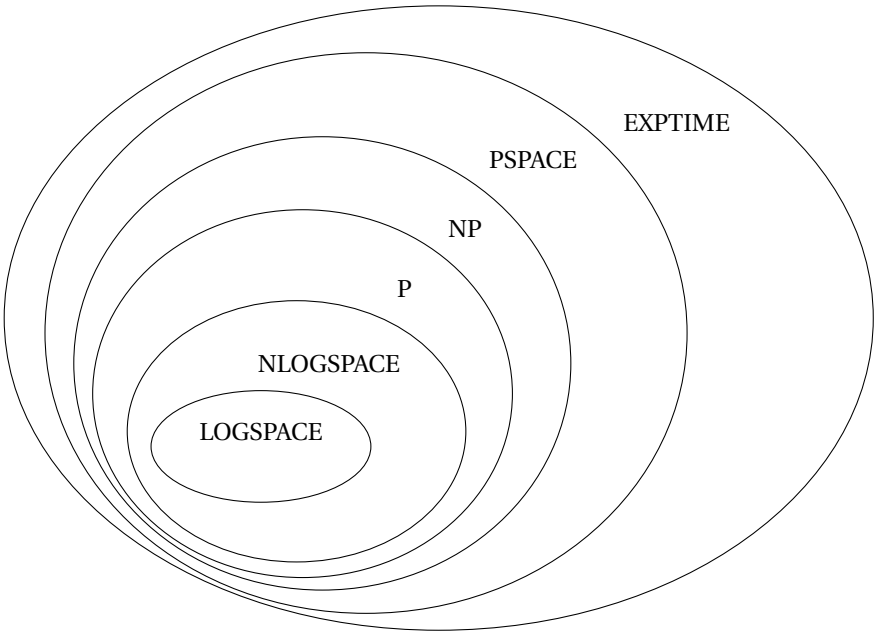


Figure 14.1: Inclusions between complexity classes

14.3 Some results and their proof

This section is devoted to proving some basic results of complexity theory, in particular to the relations between time and space. Observe that Theorem 14.4 follows from the results below.

14.3.1 Preliminaries

In order not to uselessly complicate some of the proofs, we will restrict to functions $f(n)$ *of proper complexity*: We assume that the function $f(n)$ is non-decreasing, that is to say $f(n+1) \geq f(n)$, and such that there exists a Turing machine that takes as input w and that outputs $\mathbf{1}^{f(n)}$ in time $\mathcal{O}\big(n+f(n)\big)$ and in space $\mathcal{O}\big(f(n)\big)$, where $n = \text{length}(w)$.

Remark 14.3 *This is not really restrictive, since all the non-decreasing usual functions, such as* $\log(n)$, n, $n^2, \cdots, n\log n$, $n!$ *satisfy these properties.*

Remark 14.4 *We need this hypothesis, since function* $f(n)$ *could be not computable, and it could be impossible for example to write a word of length* $f(n)$ *in the coming proofs and algorithms.*

Remark 14.5 *In most of the following statements, one can avoid this hypothesis, but at the price of complications in the proofs that we will not discuss.*

14.3.2 Trivial relations

Since a deterministic Turing machine is a particular non-deterministic Turing machine, we have:

Theorem 14.5 SPACE(f(n)) ⊆ NSPACE(f(n)).

Furthermore.

Theorem 14.6 TIME(f(n)) ⊆ SPACE(f(n)).

Proof: A Turing machine writes at most one new cell at every step. The used memory space hence remains linear in the used time. Remember that the space taken by the input is not taken into account in the memory space. □

14.3.3 Non deterministic vs deterministic time

The following result if more interesting.

Theorem 14.7 *For every language in* NTIME(f(n))*, there exists an integer c such that this language is in* TIME($c^{f(n)}$)*. If one prefers:*

$$\text{NTIME(f(n))} \subseteq \bigcup_{c\in\mathbb{N}} \text{TIME}(c^{f(n)}).$$

Proof: Let L be a problem in NTIME(f(n)). By using the principle that we used in previous chapter, we know that there exists a problem A such that to determine if a word w of length n is in L, it is sufficient to determine whether there exists a word $u \in \Sigma^*$ with $\langle w,u\rangle \in A$. This last test can be done in time $f(n)$, where $n = \text{length}(w)$. Since in time $f(n)$ one cannot read more than $f(n)$ letters from u, we can restrict to words u of length $f(n)$. Testing if $\langle w,u\rangle \in A$ for all the words $u \in \Sigma^*$ of length $f(n)$ is easily done in time $\mathcal{O}\left(c^{f(n)}\right) * \mathcal{O}\left(f(n)\right) = \mathcal{O}\left(c^{f(n)}\right)$, where $c > 1$ is the size of the alphabet Σ of the machine: Generating all words u of a given length, here $f(n)$), can be done for example by counting in base c. □

Remark 14.6 *To write the first u to be tested of length $f(n)$, we implicitly use the fact that this is feasible: This is the case if we assume $f(n)$ to be of proper complexity. We see here the interest of this (implicit) hypothesis. We will avoid discussing these type of problems in what follows, because they do not arise for usual functions $f(n)$.*

14.3.4 Non-deterministic time vs space

Theorem 14.8 NTIME(f(n)) ⊆ SPACE(f(n)).

Proof: We use exactly the same principle as in the previous proof, with the only difference that we are talking about space. Let L be a problem in NTIME(f(n)). By using the same idea as before, we know that a problem A such that to determine whether a word w of length n is in L, it is sufficient to know whether there exists $u \in \Sigma^*$ of length $f(n)$ with $\langle w, u\rangle \in A$: We use space $\mathcal{O}\left(f(n)\right)$ to generate one after the other the words $u \in \Sigma^*$ of length $f(n)$ (for example by counting in base c) and then test for each of them if $\langle w, u\rangle \in A$. This last test can be done in time $f(n)$, hence space $f(n)$. The same space can be used for each of the words u. The space use is $\mathcal{O}\left(f(n)\right)$ for writing the u's plus $\mathcal{O}\left(f(n)\right)$ for the tests. So this takes space $\mathcal{O}\left(f(n)\right)$ in total. □

14.3.5 Non-deterministic space vs time

The decision problem REACH will play an important role: Given a directed graph $G = (V, E)$, two vertices u and v, one wants to decide if there exists a path between u and v in G. It is easy to see that REACH is in P.

To every (deterministic or not) Turing machine we associate a directed graph, its *configuration graph*, where the vertices correspond to configurations and whose arcs correspond to the one-step evolution function of the machine M, that is to say to relation $\vdash$ between configurations.

Every configuration X can be described by a word $[X]$ on the alphabet of the machine M: If the input w of length n is fixed, for a computation in space $f(n)$, there are less than $\mathcal{O}\left(c^{f(n)}\right)$ vertices in this graph G_w, where $c > 1$ is the size of the alphabet of the machine.

A word w is accepted by the machine M if and only if there is a path in this graph G_w between the initial configuration $X[w]$ encoding the input w, and an accepting configuration. We may assume without loss of generality that there is a unique accepting configuration X^*. Deciding the containment of a word w in the language recognized by M is consequently solving the problem REACH on $\langle G_w, X[w], X^*\rangle$.

We will translate in various forms all that is done on problem REACH. First, it is clear that the problem REACH can be solved in time and space $\mathcal{O}\left(n^2\right)$, where n is the number of vertices, by for example a depth-first search traversal.

We deduce:

Theorem 14.9 *If $f(n) \geq \log n$, then*

$$\text{NSPACE(f(n))} \subseteq \bigcup_{c\in\mathbb{N}} \text{TIME}(\text{c}^{\text{f(n)}}).$$

Proof: Let L be a problem of NSPACE(f(n)) recognized by a non-deterministic Turing machine M. By the previous discussion, one can determine whether $w \in L$ by solving decision problem REACH on $\langle G_w, X[w], X^*\rangle$: We said this can be done in

time polynomial (quadratic) time on the number of vertices, hence in time $\mathcal{O}\left(c^{2\mathcal{O}(f(n))}\right)$, where $c > 1$ is the size of alphabet of the machine.

□

14.3.6 Non-deterministic space vs deterministic space

We will now show that REACH can be solved in space $\log^2(n)$.

Proposition 14.1 REACH ∈ SPACE($\log^2$n).

Proof: Let $G = (V, E)$ be the directed graph given as input. Given two vertices x and y of this graph and an integer i, we write $PATH(x, y, i)$ if and only if there is a path of length than 2^i between x and y. We have $\langle G, u, v\rangle \in$ REACH if and only if $PATH(u, v, \log(n))$, where n is the number of vertices. It is hence sufficient to know how to decide the relation $PATH$ in order to decide REACH.

The trick is to compute $PATH(x, y, i)$ recursively by observing that we have $PATH(x, y, i)$ if and only if there is an intermediate vertex z such that $PATH(x, z, i-1)$ and $PATH(z, y, i-1)$. One tests then at each level of the recursion every possible vertex z.

To represent every vertex, $\mathcal{O}\left(\log(n)\right)$ bits are sufficient. To represent x, y, and i, one hence uses $\mathcal{O}\left(\log(n)\right)$ bits. The algorithm has a recursion of depth $\log(n)$, every level of the recursion requiring only to store a triple x, y, i and to test every z of length $\mathcal{O}\left(\log(n)\right)$. In total, we hence use space $\mathcal{O}\left(\log(n)\right) * \mathcal{O}\left(\log(n)\right) = \mathcal{O}\left(\log^2(n)\right)$. □

Theorem 14.10 (Savitch) *If $f(n) \geq \log(n)$, then*

$$\text{NSPACE(f(n))} \subseteq \text{SPACE(f(n)}^2\text{)}.$$

Proof: We use the previous algorithm to determine if there is a path in graph G_w between $X[w]$ and X^*.

Observe that there is no need to explicitly construct the graph G_w but that one can use the previous algorithm *on-line*: Instead of writing down the graph G_w completely, and then reading in this encoding of the graph if there is an arc between a vertex X and a vertex X', one can in a lazy way, by recompute this information, determine every time that a test is needed whether $X \vdash X'$. □

Corollary 14.1 PSPACE = NPSPACE.

Proof: We have $\bigcup_{c\in\mathbb{N}} \text{SPACE(n}^c\text{)} \subseteq \bigcup_{c\in\mathbb{N}} \text{NSPACE(n}^c\text{)}$ by Theorem 14.5, and $\bigcup_{c\in\mathbb{N}} \text{NSPACE(n}^c\text{)} \subseteq \bigcup_{c\in\mathbb{N}} \text{SPACE(n}^{2c}\text{)} \subseteq \bigcup_{c\in\mathbb{N}} \text{SPACE(n}^c\text{)}$ by the previous theorem. □

14.4 Separation results

14.4.1 Hierarchy theorems

We say that a function $f(n) \geq \log(n)$ is *space constructible*, if the function that maps $\mathbf{1}^n$ to $\mathbf{1}^{f(n)}$ is computable in space $\mathcal{O}\left(f(n)\right)$.

Most of the usual functions are space constructible. For example, n^2 is space constructible since a Turing machine can obtain n in binary by counting the number of $\mathbf{1}$s, and writing n^2 in binary by using any method to multiply n with itself. The space used for this is certainly at most $\mathcal{O}\left(n^2\right)$.

Theorem 14.11 (Space Hierarchy theorem) *For every space constructible function $f : \mathbb{N} \to \mathbb{N}$, there exists a language L that is decided in space $\mathcal{O}\left(f(n)\right)$ but not in space $o(f(n))$.*

Remark 14.7 *We will prove only a version weaker than the statement above. The precise above theorem is a generalization of the following idea. The factor* log *comes from the construction of a univeral Turing machine really more efficience that the one considered in this document, introducing only a logarithmic time speeddown.*

Proof:

One considers the (very artificial) language L that is decided by the following Turing machine B:

- on an input w of size n, B computes $f(n)$ and reserves (marks) a space $f(n)$ for the coming simulation;
- If w is not of the form $\langle A \rangle 10^*$, for a Turing machine A, then the Turing machine B rejects.
- Otherwise, B simulates A on the word w for $c^{f(n)}$ steps to determine whether A accepts in space at most $f(n)$:
  - If A accepts in this time, then B rejects;
  - otherwise B accepts.

In other words, B simulates A on w, step by step, and decrements a counter c at each step. If this counter reaches 0 or if A rejects, then B accepts. Otherwise, B rejects.

By the existence of a universal Turing machine, there exist integers k and d such that L is decided in space $d \times f(n)^k$.

Suppose that L is decided by a Turing machine A in space $g(n)$ with $g(n)^k = o(f(n))$. There must exists an integer n_0 such that for $n \geq n_0$, we have $d \times g(n)^k < f(n)$.

As a consequence, the simulation of A by B will indeed be complete on every input of size n_0 or more.

Consider what happens when B is run on the input $\langle A \rangle 10^{n_0}$. Since this input is of size greater than n_0, B answers the opposite of Turing machine A on the same input. Hence B and A do not decide the same language, and hence Turing machine A does not decide L, which is a contradiction.

As a consequence L is not decidable in space $g(n)$ for any function $g(n)$ with $g(n)^k = o(f(n))$.

□

In other words:

Theorem 14.12 (Space Hierarchy theorem) *Let $f, f' : \mathbb{N} \to \mathbb{N}$ be two space constructible functions such that $f(n) = o(f'(n))$.*
Then the inclusion SPACE(f) $\subsetneq$ SPACE(f') *is strict.*

Using the same principle, one can prove.

Theorem 14.13 (Nondeterminstic Hierarchy theorem) *Let $f, f' : \mathbb{N} \to \mathbb{N}$ be two space constructible functions such that $f(n) = o(f'(n))$.*
Then the inclusion NSPACE(f) $\subsetneq$ NSPACE(f') *is strict.*

14.4.2 Applications

We deduce.

Theorem 14.14 NLOGSPACE $\subsetneq$ PSPACE.

Proof: The class NLOGSPACE is completely included in SPACE($\log^2$ n) by Savitch's theorem. But the latter is a strict subclass of SPACE(n), which is included in PSPACE. □

Analogously, we obtain.

Definition 14.7 *Let*

$$\text{EXPSPACE} = \bigcup_{c \in \mathbb{N}} \text{SPACE}(2^{n^c}).$$

Theorem 14.15 PSPACE $\subsetneq$ EXPSPACE.

Proof: The class PSPACE is completely included in, say, SPACE($n^{\log(n)}$). The latter is a strict subset of SPACE(2^n), that is in turn included in EXPSPACE. □

14.5 Exercices

Exercise 14.1 *(solution on page 242) Prove that if every* NP*-hard language is* PSPACE*-hard, then* PSPACE = NP.

14.6 Bibliographic notes

Suggested readings To go further on the notions in this chapter, we suggest [Sipser, 1997], [Papadimitriou, 1994] and [Lassaigne & de Rougemont, 2004].

A reference book on the last results of the field is [Arora & Barak, 2009].

Bibliography This chapter contains some standard results in complexity theory. We mostly used their presentation in the books [Sipser, 1997] and [Papadimitriou, 1994].

Chapter 15

Solutions of some exercises

We will find in this chapter the correction of some of the exercises. The reader is invited to sent to bournez@lix.polytechnique.fr a redaction of the solution [1] of any exercise that is not corrected in these pages, or any more elegant solution to the presented solutions.

Chapter 1

Exercice 1.3 (page 10). We can write $A \cap B^c = (A \cap A^c) \cup (A \cap B^c) = A \cap (A^c \cup B^c) = A \cap (A \cap B)^c$. Similarly $A \cap C^c = A \cap (A \cap C)^c$. So $A \cap B = A \cap C$ implies $A \cap B^c = A \cap C^c$.

Since $(X^c)^c = X$ for any subset X of E, we deduce that $A \cap B^c = A \cap C^c$ implies $A \cap (B^c)^c = A \cap (C^c)^c$ implies $A \cap B = A \cap C$.

Exercice 1.5 (page 15). The function $f(x,y) = y + (0+1+2+\cdots+(x+y))$ enumerates the elements of $\mathbb{N}^2$ as in the figure.

The function f is indeed bijective: Let $n \in \mathbb{N}$. There exists a unique $a \in \mathbb{N}$ such that $0+1+\cdots+a \leq n < 0+1+\cdots+(a+1)$. The unique antecedent n by f is given by (x,y) with $y = n - (0+1+\cdots+a)$ and $x = a - y$.

Another bijection between $\mathbb{N}^2$ and $\mathbb{N}$ is given by the function $g : \mathbb{N}^2 \to \mathbb{N}$ defined by $g(x,y) = 2^x(2y+1) - 1$. The fact that this is indeed a bijection comes from the fact that any strictly positive integer is the produce of a power of two and some odd number.

Chapter 2

Exercice 2.3 (page 18). The proof is by contradiction. Consider $X = \{k \in \mathbb{N} | P(k) \text{ is false}\}$. If X is non empty, it admits a least element n.

[1] If possible using LaTeX.

We cannot have $n \neq 0$. Indeed, one knows that for $n = 0$, by supposing for any integer $k < 0$ the property $P(k)$ we deduce $P(0)$, since there is no $k < 0$, that means that we can deduce $P(0)$ without any hypothesis.

This states: $P(n-1), P(n-2), \ldots, P(0)$ must be true from definition of X. We obtain a contradiction with the property applied in n.

Exercice 2.4 (page 19). We must first be convinced that $L^*.M$ is solution of equation $X = L.X \cup M$.

For $M = \{\epsilon\}$, this is equivalent to state that $L^* = L.L^* \cup \{\epsilon\}$. This follows from

$$L^* = \bigcup_{n\in\mathbb{N}} L^n = L^0 \cup (\bigcup_{n\geq 1} L^n) = \{\epsilon\} \cup (\bigcup_{n\geq 0} L.L^n) = \{\epsilon\} \cup L.L^*.$$

We deduce for general M that $L^*.M = L.L^*.M \cup \{\epsilon\}.M$, and so that $L^*.M$ is indeed solution of equation $X = L.X \cup M$.

There remains to prove that this is the unique solution. Let $X \subset \Sigma^*$ satisfying $X = L.X \cup M$.

To prove that $L^*.M \subset X$, it suffices to prove that for any integer n we have $L^n.M \subset X$, since $L^*.M = \bigcup_{n\geq 0} L^n.M$. We prove by recurrence on n the property $P(n)$: $L^n.M \subset X$.

$P(0)$ is true since $L^0.M = \{\epsilon\}.M = M \subset M \cup L.X = X$.

Suppose $P(n)$ true. We have $L^{n+1}.M = L.L^n.M \subset L.X \subset L.X \cup M = X$. So $P(n+1)$ is true.

Conversely, we prove by recurrence the property $Q(n)$: Every word w of X of length n belongs to $L^*.M$. This clearly provides the other inclusion.

For this, we use the second induction principle. Suppose that for all $k < n$, $Q(k)$ is true. Let $w \in X$ be a word of length n. Since $X = L.X \cup M$, two cases must be considered. Let $w \in M$ and we have directly $w \in L^*.M$ since $M \subset L^*.M$.

Let $w \in L.X$ and we can write $w = u.v$ with $u \in L$, and $v \in X$. Since $\epsilon \notin L$, the length of u is non-null and hence the length of v is strictly less than the one of w. By induction hypothesis, we have $v \in L^*.M$. So $w = u.v \in L.L^*.M \subset L^*.M$, which proves $Q(n)$, and terminates the demonstration.

Exercice 2.5 (page 22). The language L of the well parentheses expressions formed with identifiers taken in a set A and from the operators + and × is the subset of $E = (A \cup \{+, \times\} \cup \{(,)\})^*$ defined inductively by

(*B*) $A \subset L$;

(*I*) $e, f \in L \Rightarrow (e + f) \in L$;

(*I*) $e, f \in L \Rightarrow (e \times f) \in L$.

Exercice 2.6 (page 24). The proof is similar to the proof of Theorem 2.4 (which is a generalisation of this phenomenon).

Exercice 2.7 (page 27). Let $P(x)$ be the property: "x is non-empty and without any vertex with a unique non-empty son". Clearly $P(x)$ is true for $x = (\emptyset, a, \emptyset)$. If we

suppose $P(g)$ and $P(d)$, for $g,d \in ABS$, clearly $P(x)$ is also true for $x = (g,a,d)$. The first property is hence satisfied.

Let $P(x)$ be the property $n(x) = 2f(x) - 1$. We have $P(x)$ for $x = (\emptyset, a, \emptyset)$, since $n(x) = 1$, and $f(x) = 1$, and $1 = 2 * 1 - 1$.

Suppose $P(g)$ and $P(d)$ for $g,d \in ABS$. Consider $x = (g,a,d)$. We have $n(x) = 1 + n(g) + n(d) = 1 + 2 * f(g) - 1 + 2 * f(d) - 1 = 2 * (f(g) + f(d)) - 1 = 2 * f(x) - 1$.

The property is hence true for every $x \in ABS$.

Chapter 3

Exercice 3.17 (page 47). It is clear that the second property implies the first: For any truth value distribution v, we have $\overline{v}(G) = 1$ if G is a tautology, and $\overline{v}(F) = 0$ if $\neg F$ is one. In both cases $\overline{v}((F \Rightarrow G)) = 1$.

Suppose now that the second property is wrong. We can choose some distribution of truth values v such that $\overline{v}(\neg F) = 0$, and some truth value distribution v' such that $\overline{v}'(G) = 0$.

We define a distribution of truth value v'', by letting for each propositional variable x, $v''(x) = v(x)$ if x has at least one occurrence in F and $v''(x) = v'(x)$ if x has no occurrence in F. By construction, this distribution of truth values coincide with v on F and v' over G. We deduce that $\overline{v}''(F) = \overline{v}(F) = 0$ and $\overline{v}''(G) = \overline{v}'(G) = 0$. And hence $\overline{v}''((F \Rightarrow G)) = 0$. The first property is hence necessarily wrong.

Exercice 3.18 (page 47). It is clear that if a graph is colorable with k colors, each of its subgraph is colorable with k colors (the same). The difficulty is in proving the converse direction.

We introduce for each pair $(u,i) \in V \times \{1,2,\ldots,k\}$ some propositional variable $A_{u,i}$. We construct a set Γ of formulas of propositional calculus over the set of variables $A_{u,i}$ that is satisfiable if and only if G is k-colorable. The idea is that $A_{u,i}$ encodes the fact that the vertex u is colored with the color i. The set Γ is defined as $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3$, where each of the Γ_i expresses a particular constraint.

- For Γ_1: Every vertex has a color:

$$\Gamma_1 = \{A_{u,1} \vee \cdots \vee A_{u,k} | u \in V\}.$$

- For Γ_2: Every vertex has at most one color:

$$\Gamma_2 = \{\neg(A_{u,i} \wedge A_{u,j}) | u \in V, 1 \leq i,j \leq k, i \neq j\}.$$

- For Γ_3: Each edge has not its extremities of the same color:

$$\Gamma_3 = \{\neg(A_{u,i} \wedge A_{v,i}) | u \in V, 1 \leq i \leq k, (u,v) \in E\}.$$

Doing so, a graph is colorable with k colors if and only if one can satisfy all the formulas of $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3$.

We will then use the compactness theorem: Let Γ_0 be some finite subset of Γ. Let $V_0 = \{u_1, \cdots, u_n\}$ be the vertices u such that $A_{u,i}$ appears among the formulas of Γ_0. Let $G_0 = (V_0, E_0)$ be the subgraph determined by V_0.

If we we suppose that we have a graph such all the subgraphs are colorable with k colors, in particular this must hold for G_0, and so Γ_0, that is a subset of the constraints expression the fact that Γ_0 is k-colorable, is satisfiable.

Since Γ has all its finite subsets satisfiable, by the compactness theorem, Γ is hence satisfiable. This means that G is hence satisfiable. This means that G is k-colorable, since G is k-colorable if and only if Γ is satisfiable.

Chapter 4

Exercice 4.1 (page 52). We write

- $F_1 : ((F \Rightarrow ((F \Rightarrow F) \Rightarrow F)) \Rightarrow (((F \Rightarrow (F \Rightarrow F)) \Rightarrow (F \Rightarrow F)))$ (instance of axiom 2.)
- $F_2 : ((F \Rightarrow ((F \Rightarrow F) \Rightarrow F))$ (instance of axiom 1.);
- $F_3 : ((F \Rightarrow (F \Rightarrow F)) \Rightarrow (F \Rightarrow F))$ (modus ponens from F_1 and F_2);
- $F_4 : (F \Rightarrow (F \Rightarrow F))$ (instance of axiom 1.);
- $F_5 : (F \Rightarrow F)$ (modus ponens from F_3 and F_4).

Exercice 4.2 (page 52). One direction is easy: If $F_1, F_2, \ldots, F_n$ is a proof of $F \Rightarrow G$ from T, then $F_1, F_2, \ldots, F_n, F, G$ is a proof from G from $T \cup \{F\}$.

Conversely, we prove by recurrence on n that if there exists a proof of length n of G from $T \cup \{F\}$, then there exists a proof of $(F \Rightarrow G)$ from T.

By recurrence hypothesis, there exists a proof from T of each of the formulas $(F \Rightarrow F_1)$, $\ldots$, $(F \Rightarrow F_n)$, this applying by default to the case $n = 1$. Consider the formula F_n that is to say G. Three cases are possible:

1. The formula G is an axiom or a formula of T. We then have $T \vdash G$. We know that $(G \Rightarrow (F \Rightarrow G))$ is an instance of axiom 1., hence we have $\vdash (G \Rightarrow (F \Rightarrow G))$, and hence also $T \vdash (G \Rightarrow (F \Rightarrow G))$. By modus ponens, we have $T \vdash (F \Rightarrow G)$.

2. G is the formula F. The correction of the previous exercise provides a proof of $(F \Rightarrow F)$.

3. There exists $i, j < n$ such that H_j is a formula $(H_i \Rightarrow G)$. By recurrence hypothesis, we have $T \vdash (F \Rightarrow H_i)$ and $T \vdash (F \Rightarrow (H_i \Rightarrow G))$. Then the sequence:
   - $((F \Rightarrow (H_i \Rightarrow G)) \Rightarrow ((F \Rightarrow H_i) \Rightarrow (F \Rightarrow G)))$ (instance of axiom 2.);
   - $((F \Rightarrow H_i) \Rightarrow (F \Rightarrow G))$ (modus ponens)
   - $(F \Rightarrow G)$ (modus ponens)

is a proof of $(F \Rightarrow G)$ from $(F \Rightarrow H_i)$ and $(F \Rightarrow (H_i \Rightarrow G))$. By concatenating this proof of $(F \Rightarrow H_i)$ and of $(F \Rightarrow (H_i \Rightarrow G))$ from T, we obtain a proof of $F \Rightarrow G$ from T.

Exercice 4.3 (page 52). For the first assertion: Suppose that $T \cup \{F\} \vdash G$. By the deduction theorem (Exercise 4.2), we have $T \vdash (F \Rightarrow G)$. But the formula $((F \Rightarrow G) \Rightarrow (\neg G \Rightarrow \neg F))$ is an instance of axiom 5, so by modus ponens, we obtain $T \vdash (\neg G \Rightarrow \neg F)$, so by the deduction theorem again $T \cup \{\neg G\} \vdash \neg F$.

Suppose now $T \cup \{\neg G\} \vdash \neg F$. By what is above, we obtain $T \cup \{\neg\neg F\} \vdash \neg\neg G$. Since $\neg\neg G \Rightarrow G$ is an instance of axiom 4., by modus ponens, we deduce $T \cup \{\neg\neg F\} \vdash G$. But $(F \Rightarrow \neg\neg F)$ is an instance of axiom 3. From this, we deduce from any proof of a formula from $T \cup \{\neg\neg F\}$ a proof of the same formula from $T \cup \{F\}$, and we finally obtain $T \cup \{F\} \vdash G$.

For the second assertion: We have $\{\neg F, \neg G\} \vdash \neg F$ by definition, so by the first assertion, $\{\neg F, F\} \vdash G$, and from there $T \vdash G$ if both F and $\neg F$ are provable from T.

Exercice 4.4 (page 52). We have $\{\neg G, \neg G \Rightarrow G\} \vdash G$. By using Exercice 4.3, item 1, this is equivalent to say $\{\neg G, \neg G\} \vdash \neg(\neg G \Rightarrow G)$. Hence $\{\neg G\} \vdash \neg(\neg G \Rightarrow G)$. By using Exercice 4.3, item 1 in the reverse direction, $\{\neg G \Rightarrow G\} \vdash G$.

Exercice 4.5 (page 52). Suppose $T \cup \{F\} \vdash G$ and $T \cup \{\neg F\} \vdash G$. By applying the first assertion of Exercise 4.3, we obtain $T \cup \{\neg G\} \vdash \neg F$ and $T \cup \{\neg G\} \vdash F$. By the second assertion of Exercise 4.3, we obtain that any formula is provable from $T \cup \{\neg G\}$, and in particular $T \cup \{\neg G\} \vdash G$. By the deduction theorem (Exercise 4.2), $T \vdash (\neg G \Rightarrow G)$. It suffices then to use Exercise 4.4 in order to deduce $T \vdash G$.

Chapter 5

Exercice 5.1 (page 70). Only the last word corresponds to a formula: In the first and in the second, the arity of R_2 is not respected. In the third the quantification $\exists R$ is not on a variable but on a relation symbol. This is what is called a second order formula, and this is not considered as valid formula in the previous definition (i.e. in this document).

Exercice 5.2 (page 71). There is no free variable nor free occurrence in the first formula. In the second formula, the variable x is free: Its first occurrence is bound, its second occurrence is free.

Exercice 5.4 (page 75). To avoid too heavy notations, we will write xRy pour $R(x, y)$, and we authorize ourselves not to write all the parentheses. It suffices to consider

$$(\forall x xRx) \wedge (\forall x \forall y (xRy \wedge yRx \Rightarrow x = y)) \wedge (\forall x \forall y \forall z (xRy \wedge yRz \Rightarrow xRz)).$$

Exercice 5.7 (page 78). We will only indicate here by some arrow the implications. For example $\Rightarrow$ means that the left member implies the right member.

1. $\Leftrightarrow$
2. $\Leftrightarrow$
3. $\Rightarrow$
4. $\Leftrightarrow$
5. $\Rightarrow$
6. $\Rightarrow$

Exercice 5.8 (page 80). Here are some equivalent prenex forms.

$$\exists x \forall x' \forall y (P(x) \wedge (Q(y) \Rightarrow R(x')))$$

$$\forall x \forall y \exists x' (P(x') \wedge (Q(y) \Rightarrow R(x)))$$

$$\forall x \exists x' \forall y (P(x') \wedge (Q(y) \Rightarrow R(x)))$$

Chapter 6

Exercice 6.1 (page 86). The third item of Definition 6.3 is true for any relation symbol R and in particular for the symbol $=$ of arity 2: In particular, we have $\forall x_1 \forall x_1' \forall x_2 (x_1 = x_1' \Rightarrow (x_1 = x_2 \Rightarrow x_1' = x_2)$. Since we have $x = x$ by the first item of Definition 6.3, if we have $x = y$ then we have $y = x$ by applying the case where x_1, x_2, x_1' are respectively x, x and y.

Exercice 6.6 (page 90). Clearly the last assertion follows from the second, since the first produces a model that cannot be the standard model of the integers that satisfies the axioms of Robinson. Indeed, in the standard model of the integers (in the integers) the addition is commutative.

For the first assertion, it suffices to prove the property by recurrence over n. It is true for $n = 0$ by axiom $\forall x\, \mathbf{0} + x = x$, applied in $x = s^m(\mathbf{0})$. Suppose the property true at rank $n-1 \geq 0$: We have $s^n(\mathbf{0}) + s^m(\mathbf{0}) = s(s^{n-1}(\mathbf{0})) + s^m(\mathbf{0})$, which according to axiom $\forall x \forall y\, s(x) + y = s(x+y)$ applied for $x = s^{n-1}(\mathbf{0})$ and $y = s^m(\mathbf{0})$ values $s(s^{n-1}(\mathbf{0}) + s^m(\mathbf{0}))$ so $s(s^{n+m-1}(\mathbf{0}))$ by recurrence hypothesis, in other words, $s^{n+m-1+1}(\mathbf{0}) = s^{n+m}(\mathbf{0})$.

For the second assertion, we must consequently construct a model whose base set contains something else than (only) the elements $s^{(n)}(\mathbf{0})$ for some integer n. Here is a way to proceed. One considers a set X with at least two elements.

We consider the structure $\mathfrak{M}$ whose base set is

$$M = \mathbb{N} \cup (X \times \mathbb{Z}),$$

and where the symbols $s, +, *, =$ are interpreted by the following conditions:

- $=$ is interpreted by equality. $s, +, *$ extends the corresponding functions over $\mathbb{N}$;
- for $a = (x, n)$:
  - $s(a) = (x, n+1)$;
  - $a + m = m + a = (x, n+m)$;
  - $a * m = (x, n * m)$ if $m \neq 0$, and $(x, n) * 0 = 0$;
  - $m * a = (x, m * n)$;
- for $a = (x, n)$ and $b = (y, m)$:
  - $(x, n) + (y, m) = (x, n+m)$;
  - $(x, n) * (y, m) = (x, n * m)$.

(in these definitions, $\mathbb{N}$ and $\mathbb{Z}$ denotes the usual (standard) sets). One checks that this structure satisfies all the axioms of Robinson, and that the addition is not commutative.

Exercice 6.9 (page 91). We will only provide here a scheme of the proof . We prove successively that

- $\forall v(v + \mathbf{0} = v)$
- $\forall v \forall v' v + s(v') = s(v + v')$
- $\forall v(v + \mathbf{1} = s(v))$ where 1 denotes $s(\mathbf{0})$
- $\forall v \forall v'(v + v' = v' + v)$

For example, $\forall v(v + \mathbf{0} = v)$ is proved by observing that $\mathbf{0} + \mathbf{0} = \mathbf{0}$ and that $\forall v((v + \mathbf{0} = v) \Rightarrow (s(v) + \mathbf{0} = s(v))$. We use the scheme of Peano axioms in the case where the formula F is the formula $v + \mathbf{0} = v$ to deduce $\forall v(v + \mathbf{0} = \mathbf{0})$.

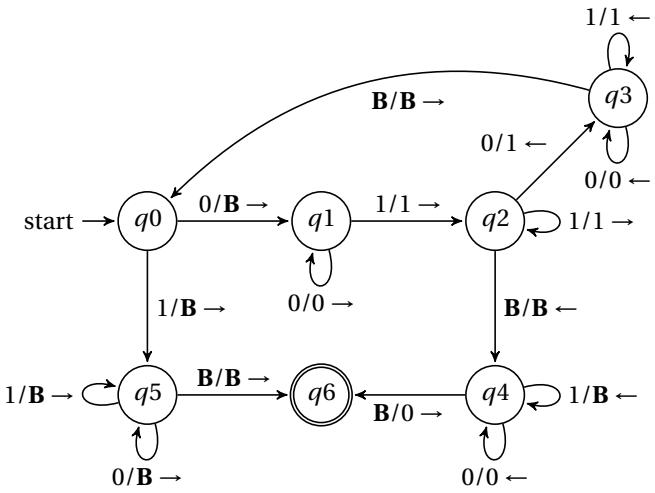
Exercice 6.12 (page 99). Consider a new constant symbol c. Add this constant symbol to the signature of Peano axioms. Consider $\mathscr{T}$ defined as the union of Peano axioms of the formulas $\neg c = s^n(\mathbf{0})$, for some integer n. Every finite subset of $\mathscr{T}$ has a model as it is included in the union of Peano axioms, and of the formulas $\neg c = s^n(\mathbf{0})$ for $1 \leq n \leq N$ for some integer N: It suffices to observe that if one interprets c by $N+1$, one then obtain a model of this finite subset.

By compactness theorem, $\mathscr{T}$ has a model. This model must satisfy $\neg c = s^n(\mathbf{0})$ for every integer n. The interpretation of c is hence not a standard integer. The model is hence non-standard: It contains some "integers" that are not standard.

Exercice 6.13 (page 100). Suppose that $\mathscr{T}$ is a theory at most countable that has a model. It is coherent: Corollary 6.3 applies. Observe that the proof of Corollary 6.3 (actually the proof of completeness theorem) consists at the end in constructing a model $\mathfrak{M}$ of $\mathscr{T}$ whose domain is made of terms over the signature. Since the set of terms over a countable signature is countable, the model $\mathfrak{M}$ that is built is indeed a model of $\mathscr{T}$.

Chapter 7

Exercice 7.2 (page 110). Here is a solution: One considers a machine on the set of states $Q = \{q0, q1, q2, \cdots, q6\}$ with $\Gamma = \{0, 1, \mathbf{B}\}$.



The machine is built in order to do the following work: It seeks the 0 at the left-most position, and it replaces it by a blank. It searches then on the right a 1, and when it finds one, it continues on the right until it finds a 0, that it replaces by a 1. The machine then returns left in order to find the 0 at the most left position that it identifies by finding the first blank on the left, and then moving one cell right from this blank.

The process is repeated until:

- either when searching on right a 0, one meets a blank. Then the n 0's in 0^m10^n have been changed into 1's and $n+1$ of the m symbols 0 have been changed into **B**. In that case, the machine replaces the $n+1$ symbols 1 by a 0 and n blanks, which remains $m-n$ symbols 0 on the tape. Since in this case, $m \geq n$, $m \ominus n = m - n$.

- or in restarting the cycle, the machine does not succeed to find a 0 to be changed in a blank, since the m first 0's have already been changed in **B**. Then $n \geq m$, and so $m \ominus n = 0$. The machine then replaces every 1 and 0 remaining by some blanks and halts with a tape completely blank.

Chapter 9

Exercice 9.1 (page 136). A is decidable.

Indeed: Suppose that s values 0. In that case, A is recognized by the Turing machine that compares the letter in front of the head to 0 and accepts if it values 0 and rejects otherwise.

Suppose that s values 1. In that case, A is recognized by the Turing machine that compares the letter in front of the head to 0 and accepts if it values 1 and rejects otherwise.

In any case, A is decided by a Turing machine.

Exercice 9.2 (page 148). This is a direct application of Rice theorem.

Exercice 9.3 (page 155). Suppose that $S \subset \mathbb{N}$ is decidable. The function χ is computable, since it is sufficient on input n to determine if $n \in S$ and to return 1 if this is the case, 0 otherwise.

Conversely, if χ is computable, then S is decidable: On some input n, one computes χ and one accepts (respectively: rejects) if $\chi(n) = 1$ (resp. $\chi(n) = 0$).

Suppose that $S \subset \mathbb{N}$ is semi-decidable. The function is computable, since it is sufficient on input n to simulate the machine that computes the function, and accepts if this simulation halts.

Conversely, if the function is computable, S is semi-decidable: On some input n, one simulates the computation of the function, and one accepts if the simulation accepts.

Exercice 9.1 (page 155). Let H be the computable function defined by: If t is some unary program (a Turing machine) of A, then $H(\langle t \rangle, n)$ provides the result of t on n, otherwise $H(\langle t \rangle, n) = 0$. By construction, H is some interpreter for all the unary programs of A. The function H is total.

We prove that the total computable function $H'(n) = H(n,n) + 1$ is not in A: Indeed, otherwise there would be a t in A that computes H'. We would have for all n, $H(\langle t \rangle n)$ that would be the result of t on n, so $H'(n) = H(n,n) + 1$. In particular, $H(\langle t \rangle, \langle t \rangle) = H(\langle t \rangle, \langle t \rangle) + 1$. Contradiction.

This result implies the undecidability of the halting problem.

- Indeed, suppose that $H(A,n)$ decides of the halting of the Turing machine A on input n. For all unary program f, let f' be the unary program that always halts defined by $f'(n) = f(n)$ if $Ha(f,n)$, 0 otherwise.
- Let A defined as the set of coding of Turing machines of the form if $H(f,n)$ then $f(n)$ otherwise 0.
- For any unary program f that always halt, f' and f computes the same function.
- Any total unary function is represented by some Turing machine A. This is in contradiction with the previous result.

Exercice 9.4 (page 155). To test whether $x \in E$, one enumerates the elements of E until one finds x, in which case one accepts, or an element greater than x, in which case one rejects.

Exercice 9.5 (page 155). To extract an infinite decidable set from an infinite computably enumerable set, it is sufficient to extract a subsequence strictly increasing from the sequence of the $f(n)$: one starts with $y = max = 0$. For $n = 0, 1, \ldots,$

- One computes $y = f(n)$.
- If $y > max$ the one sets $max := y$, and one prints $f(n)$

Exercice 9.6 (page 155). For $n = 0, 1, \ldots,$ one tests if n is in the set, and if so, one prints it.

Exercice 9.7 (page 156). To test if $x \in \exists A$, it is sufficient to test for $y = 0, 1, \ldots$ if $(x, y) \in A$. One halts as soon as one find such a y.

Every computably enumerable language is enumerated by some computable function f. It is then to the projection of the set $A = \{(f(n), n) | n \in \mathbb{N}\}$. This set A is indeed decidable.

Exercice 9.10 (page 157). Their uncomputability comes from Rice theorem.

The first is computably enumerable: One enumerates the triples (a, b, t) with a and b two distinct words, t some integer, and for each of them one tests if A accepts a and b in time t. If yest, one accepts.

The second is not computably enumerable since its complement is computably enumerable: One enumerates the pairs (a, t) and one tests if a is accepted by A in time t. If so, one accepts.

Chapter 10

Exercice 10.1 (page 162). Let $\mathbb{N}$ be the standard model of the integers. $Th(\mathbb{N})$ corresponds to the closed formula F true in $\mathbb{N}$.

The incompleteness theorem states that exist some closed formula true of $Th(\mathbb{N})$ which are not provable, nor their negation from Peano axioms, or from any "reasonable" axiomatisation of the integers.

Let F be such a closed formula. Suppose without loss of generality that F is satisfied on $\mathbb{N}$.

The incompleteness theorem states that non-provable formulas are exactly those that are true in all the models. This simply means that there must exist some other models than $\mathbb{N}$ to Peano axioms: In particular, there must exist a model where F is not satisfied.

Repeated in another way, there is a model of Peano axiom with F satisfied (for example $\mathbb{N}$) and another mode where F is not satisfied.

The completeness theorem remains compatible since F is not satisfied in all the models.

Exercice 10.1 (page 165). We consider the Turing machine S that does the following:

- on every input w
  - obtain by the recursion theorem its own description $\langle S \rangle$.
  - construct the formula $\psi = \gamma_{S,\epsilon}$ (the formula γ of the proof of the course for S)
  - enumerate the provable formulas as long as $\gamma_{S,\epsilon}$ is not produced.
  - if previous steps eventual terminates, then accepts.

The formula $\psi = \gamma_{S,\epsilon}$ of the second step is not provable: Indeed ψ is true if an only if S does not accept the empty word, because:

- If S finds a proof of ψ, then S accepts the empty word, and hence formula ψ is wrong:
- (If arithmetic is coherent one cannot prove some wrong formula, and hence) This case cannot happen.

Now observe that if S does not find a proof of ψ, then S is not accepting the empty word: Consequently, ψ is true, but is not provable!

In short: ψ is true, but is not provable.

Chapter 11

Exercice 11.1 (page 176). We will use the fact that the limit exists and is positive to prove that $f(n) = \mathcal{O}\left(g(n)\right)$ and $f(n) = \Omega(g(n))$, according to the definition of Θ.

Since

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0,$$

from the definition of a limit, there is a rank n_0 from which the ratio is between $\frac{1}{2}c$ and $2c$. So $f(n) \leq 2cg(n)$ and $f(n) \geq \frac{1}{2}cg(n)$ for all $n \geq n_0$, which proves exactly what is required.

Exercice 11.3 (page 177). The computation of the maximum of n numbers can be done in linear time (See course).

A sorting algorithm (one takes n numbers in input and must produce as output the same numbers but in increasing order) such as the *merge sort* works in time $\emptyset(n \log n)$: The merge sort consists, in order to sort n numbers, in splitting the set in two subsets of the same time (up to 1), sort recursively each subset, and then merge the two results. The merging of two sorted list can be solved in a time linear in the sum of the lengths of the two lists. Indeed, merging in that case consists in repeating the following operation as long as it is possible: Write the least element of the two lists„ and then delete this element. This gives a global complexity for merge sort that satisfies a recursive equation of type $T_n = \mathcal{O}(2 * T_{n/2}) + \mathcal{O}(n)$ which can be proved to providing a complexity of $On \log n$.

Suppose that n sets S_1, S_2, ..., S_n are given, each of them being a subset of $\{1,2,\ldots,n\}$, and that one wants to know if there is a disjoint pair among these sets. This can be solved in cubic time: It is sufficient to go through all the pairs S_i and S_j, and for each pair to scan the elements of S_i to know if there are in S_j.

The following chapters contain mainly problems for which no polynomial solutions is found: The presented algorithms are not polynomial.

Exercice 11.4 (page 177). The time is respectively for (a),

- multiplied by 4;
- multiplied by 8;
- multiplied by 400;
- multiplied by2 + 1;
- squared.

And for (b):

- increases of $2n+1$;
- increases of $3n^2+3n+1$;
- increases of $200n+100$;
- transformed into $(n+1)\log(n+1)$, hence essentially increased of $\mathcal{O}\left(n\log n\right)$;
- multiplied by 2.

Chapter 12

Exercice 12.4 (page 201). If P = NP then, since the complement of P is P (it is sufficient to inverse the accepting state and the rejecting state of a machine), we must have NP equal to its complement. Hence, we cannot have NP which is not equal to its complement.

Exercice ?? (page ??). This exercise is among the corrected exercises in [Kozen, 2006].

Concerning first item, this is incorrect because we have not shown how to produce x deterministically when it exists.

For second item: The data of x corresponds to a valid certificate that can be checked in polynomial time.

For the third item. Suppose that P = NP. Hence B is in P. Using this fact, given y of length n we can do a binary search on strings of length n to find x such that $f(x) = y$. Indeed, first ask whether $(y,\epsilon) \in B$?. If not, then no such x exists; halt an report failure. If so, ask, whether $(y,0) \in B$. If yes, there is an x with $f(x) = y$ whose first bit is 0, and if no, all such x have first bit 1. Now, depending on the previous

answer, ask whether $(y,00) \in B$ or $(y,10) \in B$ as appropriate. The answer determines the second bit of x. Continue in this fashion until all the bits of some x with $f(x) = y$ have been determined.

For the fourth and fifth item: By definition.

Concerning sixth item: First determine whether the input is of the form $\phi\#t$, and if so, evaluate ϕ on t. If f is invertible, then P = NP, because ϕ is satisfiable if and only if there exists x such that $f(x) = \phi\#1^{|t|}$. We already proved the reverse direction.

The last assertion is then immediate.

Chapter 13

Exercice 13.3 (page 213).

- The problem is in NP since given some seating plan, one can check in polynomial time if every knight is not sitting close to one of its enemies.
- We will do the reduction from the problem HAMILTONIAN CIRCUIT.

  Let $\mathscr{I} =< G = (V,E) >$ be an instance of HAMILTONIAN CIRCUIT.

  We transform this instance into an instance $\mathscr{I}_2$ of the problem KNIGHTS OF THE ROUND TABLE in the following way:

  - Each vertex of the graph is a knight;
  - Two knights are enemies if and only there is no edge in G involving the two vertices represented by these two knights.

  This transformation can be done in polynomial time (we have constructed actually the complementary of graph G).

  It is easy to prove that:

  - If there exists some Hamiltonian cycle in G, then there is a seating plan. It suffices to see that an edge in G encodes the fact that two knights are not enemies. So the seating plan corresponds to a Hamiltonian cycle.
  - If there exists a seating plan, then there exists some Hamiltonian cycle in G.

  So the problem KNIGHTS OF THE ROUND TABLE is at least as hard as the Hamiltonian cycle problem.

Hence, the problem KNIGHTS OF THE ROUND TABLE is NP-complete.

Exercice 13.4 (page 213).

- The problem HAMILTONIAN CIRCUIT is in NP since given a path, one can check in in polynomial time if it crosses exactly once every vertex of the graph and if it has u and v as extremities.

- We are going now to do a reduction from the problem HAMILTONIAN CYCLE. Let $\mathscr{I} =< G = (V,E) >$ be an instance of problem HAMILTONIAN CYCLE. We are going now to transform this instance into some instance of the problem HAMILTONIAN PATH in the following way: We will construct a graph $G' = (V',E')$ such that.

  - Let u be some arbitrary vertex of V;
  - $V' := V \cup \{v\}$ such that v is a vertex not belonging to V;
  - $E' := E \cup \{(v,\ell) : \ell \text{ is a neighborhood of } u \text{ in } G\}$.

  This transformation can be done in polynomial time (we just need to copy the graph G by adding a vertex and some edges).

  It is easy to prove that:

  - If there exists some Hamiltonian cycle in G, then there exists a Hamiltonian path in G'.
    Let $C = (u,\ell_1,\ldots,\ell_{n-1},u)$ be a Hamiltonian cycle in G. We construct the path $\mathscr{P} = (u,\ell_1,\ldots,\ell_{n-1},v)$ in the graph G'. This path is Hamiltonian: It goes exactly once through v and through each vertex of G since C is some Hamiltonian cycle.
  - If there exists some Hamiltonian path in G', then there exists some Hamiltonian cycle in G.
    Let $\mathscr{P} = (u,\ell_1,\ldots,\ell_{n-1},v)$ be a path in the graph G'. The cycle $C = (u,\ell_1,\ldots,\ell_{n-1},u)$ is Hamiltonian for the graph G.

  So the problem HAMILTONIAN CYCLE reduces to problem HAMILTONIAN PATH in polynomial time.

So the problem HAMILTONIAN PATH is NP-complete.

Exercice 13.5 (page 214).

- The problem PATH is in NP since given some path, on can check in polynomial time if it is of length $n/2$ and that it as u and v as extremities.
- We are going now to do a reduction from the problem HAMILTONIAN PATH. Let $\mathscr{I} =< G = (V,E), u, v >$ be an instance of problem HAMILTONIAN PATH.

  We transform this instance in an instance $\mathscr{I}'$ of problem CHAIN in the following way. We construct a graph $G' = (V',E')$ such that

  > G' is a copy of graph G plus a chain of $|V|$ vertices whose a unique vertex of this chain is neighbour of u.

  This transformation is done in polynomial time (we have just copied the graph G by adding a vertex and some edges).

  It is easy to prove that there exists some Hamiltonian chain in G if and only if there exists a Hamiltonian chain in G' of length $\frac{|V'|}{2}$.

So the problem HAMILTONIAN CHAIN reduces to the problem CHAIN; and the latter is hence NP-complete.

Exercice 13.6 (page 214). The problem TREE is NP-complete. It suffices to observe that an Hamiltonian path is a covering tree with two leaves.

Exercice 13.7 (page 215).

Point 1.: Let S be a vertex cover of graph G. One needs to prove that all the vertices of the graph are either neighbours of S or in S in the graph G'.

1. As all the edges of G have at least one of their extremities in S, all the vertices of G' corresponding to some edge of G are neighbours of S.
2. Let v be a vertex of V. If v does not belong to S, then v has at least an extremity of some edge (since G is connected). Since S is a vertex cover of graph G this implies that the other extremity of this edge is in S. So v is a neighbour of a vertex of S. Consequently, all the vertices of G are either in S or are neighbour of S.

So S is a dominating set of G'.

Point 2.: Suppose that $S' \subseteq V$. S' dominates the graph G'. This means that all the edges of G have one of its extremities in S'. So S' is a vertex cover of graph G.

Suppose that S' is not a subset of V. There exists a vertex s not belonging to V in S'. Then s is a vertex representing an edge (u, v) in graph G.

- If the two vertices u and v are in S', then $S' \backslash \{s\}$ is a dominant set of G' of cardinality smaller than S'.
- If one of the two vertices u and v are in S', then $S' \backslash \{s\} \cup \{u, v\}$is a dominant set of G' of same cardinality as S'.

We repeat the same reasoning on this set until all the vertices not belonging to V are suppressed.

Point 3: *Inputs* : a non-oriented graph G, and an integer k
Question : Does there exist a dominant set S of G such that $|S| \leq k$?

Point 4: One can check in polynomial time if a set of vertices is a dominant set and it is of cardinality less than k.

Point 5. It suffices to combine previous questions.

Exercice 13.8 (page 216). Point 1. Given a subset of vertices of K, one can check in polynomial time

1. if S of of cardinality less than k;
2. if, for every vertex v of V, its distance to one of the vertices of S is less than b.

Point 2. We will do the reduction from the problem DOMINANT. Let I be an instance of the minimum dominant: $G = (V, E)$ and some integer k'.

We construct the instance of problem k-CENTRES : $k' = k$ and the complete graph $K = (V, E')$ with the following weight function on the edges:

$$w(u,v) = \begin{cases} 1 & if \quad (u,v) \in E; \\ 2 & if \quad (u,v) \notin E. \end{cases}$$

This reduction can be done in polynomial time and it satisfies the following properties:

- If there exists a dominant set of size less than k in G then K admits a k-center of cost 1.
- If there exists a dominant set of size more than k in G then K admits a k-center of cost 2.

We can deduce that there exists a dominant set of size less than k in G if and only if K admits a k-center of cost 1.

Chapter 14

Exercice 14.1 (page 225). We now that NP $\subseteq$ PSPACE. We must prove the reverse inclusion. One considers SAT in PSPACE, that is NP-complete. It is hence NP-hard, and hence also PSPACE-hard. So for every language A PSPACE, A reduces to SAT, and since SAT $\in$ NP, we have A in NP.

15.1 Bibliographic notes

Several corrections are shamefully taken from [Arnold & Guessarian, 2005] for the first chapter. Some others are inspired from results proved in book [Cori & Lascar, 1993a] or from book [Kleinberg & Tardos, 2006], or exercises from [Sipser, 1997].

%chapitreSolutions de certains exercicescorrectionschapquatorze

List of Figures

Index

Bibliography

[Arnold & Guessarian, 2005] Arnold, A. & Guessarian, I. (2005). *Mathématiques pour l'informatique*. Ediscience International.

[Arora & Barak, 2009] Arora, S. & Barak, B. (2009). *Computational Complexity: A Modern Approach*. Cambridge University Press. `https://doi.org/10.1017/cbo9780511804090`

[Carton, 2008] Carton, O. (2008). *Langages formels, calculabilité et complexité*.

[Cori et al., 2010] Cori, R., Hanrot, G., Kenyon, C., & Steyaert, J.-M. (2010). Conception et analyse d'algorithmes. Cours de l'Ecole Polytechnique, Cours de l'Ecole Polytechnique.

[Cori & Lascar, 1993a] Cori, R. & Lascar, D. (1993a). *Logique mathématique. Volume I*. Masson.

[Cori & Lascar, 1993b] Cori, R. & Lascar, D. (1993b). *Logique Mathématique, volume II*. Masson.

[Davis, 1989] Davis, R. E. (1989). *Truth, deduction, and computation - logic and semantics for computer science*. Principles of computer science series. Computer Science Press.

[Dehornoy, 2006] Dehornoy, P. (2006). *Logique et théorie des ensembles*. Notes de cours.

[Dowek, 2008] Dowek, G. (2008). *Les démonstrations et les algorithmes*. Polycopié du cours de l'Ecole Polytechnique.

[Froidevaux et al., 1993] Froidevaux, C., Gaudel, M., & Soria, M. (1993). *Types de donnees et algorithmes*. Ediscience International.

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

[Hopcroft & Ullman, 2000] Hopcroft, J. E. & Ullman, J. D. (2000). *Introduction to Automata Theory, Languages and Computation, Second Edition* (2nd ed.). Addison-Wesley.

[Jones, 1997] Jones, N. D. (1997). *Computability and complexity - from a programming perspective*. Foundations of computing series. MIT Press. https://doi.org/10.7551/mitpress/2003.001.0001

[Kleinberg & Tardos, 2006] Kleinberg, J. M. & Tardos, É. (2006). *Algorithm design*. Addison-Wesley.

[Kozen, 1997] Kozen, D. (1997). *Automata and computability*. Springer Verlag. https://doi.org/10.1007/978-1-4612-1844-9

[Kozen, 2006] Kozen, D. (2006). *Theory of computation*. Springer-Verlag New York Inc.

[Kreisel & Krivine, 1967] Kreisel, G. & Krivine, J., editors (1967). *Eléments de logique mathématique. Théorie des modèles*. Monographique de la société mathématique de France. Dunod Paris.

[Lassaigne & de Rougemont, 2004] Lassaigne, R. & de Rougemont, M. (2004). *Logic and complexity*. Discrete Mathematics and Theoretical Computer Science. Springer. https://doi.org/10.1007/978-0-85729-392-3

[Matiyasevich, 1970] Matiyasevich, Y. (1970). Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR*, 191(2), 279–282.

[Mendelson, 1987] Mendelson, E. (1987). *Introduction to mathematical logic (3. ed.)*. Chapman and Hall.

[Nerode & Shore, 1997] Nerode, A. & Shore, R. A. (1997). *Logic for Applications, Second Edition*. Graduate Texts in Computer Science. Springer. https://doi.org/10.1007/978-1-4612-0649-1

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

[Poizat, 1995] Poizat, B. (1995). *Les petits cailloux: Une approche modèle-théorique de l'Algorithmie*. Aléas Editeur.

[Sedgewick & Flajolet, 1996] Sedgewick, R. & Flajolet, P. (1996). *Introduction à l'analyse d'algorithmes*. International Thomson Publishing, FRANCE.

[Sipser, 1997] Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing Company.

[Stern, 1994] Stern, J. (1994). Fondements mathématiques de l'informatique. *Ediscience International, Paris*.

[Steyaert,] Steyaert, J.-M. Théorie des automates, langages formels, calculabilité. Cours de l'Ecole Polytechnique, Cours de l'Ecole Polytechnique.

[Wolper, 2001] Wolper, P. (2001). *Introduction à la calculabilité: cours et exercices corrigés*. Dunod.