Fondements de l'Informatique: Logique, Modèles, Calculs

Olivier Bournez

Fondements de l'informatique
Logique, modèles, et calculs

Cours INF423
de l'Ecole Polytechnique

Olivier Bournez

Table des matières

Chapitre 1

Introduction

Ce cours est un cours sur les fondements de l'informatique : il se focalise sur trois domaines centraux en informatique : la logique, les modèles de calculs et la complexité.

Tous ces domaines sont reliés par la question suivante : quelles sont les capacités et les limites des ordinateurs ?

Même un téléphone est maintenant capable de résoudre très rapidement certains problèmes, comme trier un répertoire de plus d'un million d'entrées. Par contre, certains problèmes s'avèrent beaucoup plus lents et difficiles à résoudre : par exemple, résoudre un problème d'emploi du temps, ou affecter les choix d'affectations des élèves de l'école polytechnique en fonction de leurs préférences ordonnées.

Au cœur de ce cours est la compréhension de ce qui fait qu'un problème, comme le tri, est simple à résoudre informatiquement, alors qu'un problème comme un problème d'emploi du temps peut prendre des siècles à résoudre avec seulement un millier de données en entrées.

Autrement dit, au cœur de nos interrogations est aussi la question suivante : qu'est-ce qui rend certains problèmes difficiles, et d'autres faciles ?

C'est la question centrale de la complexité, et de la calculabilité.

Pourquoi s'intéresser à comprendre les problèmes difficiles, plutôt que d'essayer de résoudre des problèmes très concrets ? Premièrement, parce que des problèmes très simples et concrets, et aux enjeux économiques considérables, s'avèrent faire partie des problèmes difficiles.

Deuxièmement, parce que comprendre qu'un problème ne peut pas être résolu facilement est utile parce que cela signifie que le problème doit être simplifié ou modifié pour pouvoir être résolu. Ce cours permet réellement de comprendre les pistes pour éviter les problèmes difficiles à résoudre informatiquement.

Enfin et surtout parce que les problèmes difficiles ont réellement des implications dans la conception de nombreux systèmes actuels.

Par exemple, pour la vérification, l'analyse, et la conception de systèmes : lorsqu'on conçoit un système, on souhaite en général qu'il se comporte au minimum selon la spécification avec laquelle on l'a conçu. On aimerait que le pro-

cessus de vérification puisse s'automatiser, c'est-à-dire que l'on puisse garantir informatiquement qu'un système donné vérifie une/sa spécification. Surtout, lorsque le système en question est d'une complexité énorme, comme les processeurs actuels, et qu'un unique être humain n'est plus capable d'en comprendre seul tous les composants.

Les résultats de ce cours montrent précisément que le processus de vérification ne peut pas s'automatiser facilement. Tout l'art de la vérification de systèmes, et donc de la conception de systèmes, est de tenter d'éviter ces difficultés pour la rendre praticable, ce qui nécessite d'avoir compris ces difficultés.

D'autres domaines sont fortement impactés par la complexité. Un des premiers qui l'a été historiquement est la cryptographie : dans la plupart des domaines, on cherche plutôt à privilégier les problèmes faciles à résoudre aux problèmes difficiles. La cryptographie est originale de ce point de vue, car elle cherche plutôt à comprendre des problèmes difficiles à résoudre plutôt que simples, car un code secret doit être dur à casser sans la clé secrète.

On peut aussi se demander : pourquoi autant de logique dans un cours sur les fondements de l'informatique ?

Une première raison est parce que les programmes informatiques et les langages informatiques sont essentiellement basés sur la logique. Les processeurs sont d'ailleurs essentiellement composés de portes logiques. Les programmes sont essentiellement faits d'instructions logiques, et de tests logiques. Comprendre cette logique permet de bien comprendre ce que font les programmes informatiques.

Plus fondamentalement, les programmes et les systèmes informatiques obligent bien souvent à décrire très précisément les objets sur lesquels ils travaillent : pour résoudre un problème d'emploi du temps, le système va obliger à décrire toutes les contraintes. Pour faire une requête sur une base de données, le système va obliger à formuler très précisément cette requête. Il s'avère que la logique mathématique est un outil très naturel pour décrire le monde qui nous entoure, et à vrai dire, le modèle le plus naturel que nous connaissons pour le faire. Comprendre les concepts de la logique, permet de bien comprendre nombre de concepts informatiques. Par exemple, pour décrire le système d'information d'une entreprise, ou tout système complexe, le meilleur outil reste bien souvent la logique mathématique.

Une troisième raison est historique, et au cœur en fait de la naissance de l'informatique. Durant la première moitié du siècle dernier, des mathématiciens comme Kurt Gödel, Alonzo Church ou Alan Turing ont découvert que certains problèmes ne pouvaient pas se résoudre par des dispositifs informatiques ou automatiques comme les ordinateurs. Par exemple, le problème de déterminer si un énoncé mathématique est vrai ou non. Cette tâche, qui est le quotidien du mathématicien, ne peut pas être résolue par aucun ordinateur, quelle que soit sa puissance.

Les conséquences de ces résultats profonds ont permis la naissance d'idées sur des modèles d'ordinateurs qui ont mené à la conception des ordinateurs actuels.

Au cœur de ces découvertes sont des liens très forts qui unissent algorithmes

et démonstrations : une démonstration logique correspond à un algorithme. A partir d'hypothèses, on déduit des nouvelles assertions à l'aide de règles logiques. Réciproquement, un programme correspond à une démonstration dans un certain sens.

C'est ces liens forts entre algorithmes et démonstrations qui ont fait naître l'informatique et ses concepts et ce bien avant l'existence même de machines aussi puissantes que celles que nous connaissons actuellement. Historiquement, la première question était : "qu'est-ce qu'une démonstration" ? Elle est maintenant devenue : "qu'est-ce qu'un ordinateur" ?

Ils ont par ailleurs révolutionné notre conception des mathématiques, de l'informatique, et plus généralement du monde qui nous entoure.

En mathématiques, ils ont mené à une crise des fondements, avec le retour sur des questions aussi fondamentales que celle-ci : qu'est-ce qu'un ensemble, qu'est-ce qu'une démonstration ? Que peut-on prouver ?

L'ambition de ce document est plutôt de se focaliser sur l'informatique. Nous estimerons que ce document aura atteint son but si à sa lecture notre lecteur change au final la réponse qu'il aurait pu faire à priori sur des questions aussi simples que celle-ci :

- qu'est-ce qu'un ordinateur ?
- qu'est-ce qu'une preuve ?
- qu'est-ce qu'un algorithme ?
- qu'est-ce qu'un bon algorithme ?

Si tel est le cas, sa façon de programmer, ou d'appréhender un problème informatique ne devrait plus être la même.

Remerciements L'auteur de ce document souhaite remercier vivement Johanne Cohen, Bruno Salvy et David Monniaux pour leurs retours sur des versions préliminaires de ce document.

Tous les commentaires (mêmes typographiques, orthographiques, etc.) sur ce document sont les bienvenus et à adresser à bournez@lix.polytechnique.fr.

1.1 Concepts mathématiques

1.1.1 Ensembles, Fonctions

Soit E un ensemble, et e un élément. On note $e \in E$ pour signifier que e est un élément de l'ensemble E. Si A et B sont deux ensembles, on note $A \subset B$ pour signifier que tout élément de A est un élément de B. On dit dans ce cas que A est une partie de B. Lorsque E est un ensemble, les parties de E constituent un ensemble que l'on note $\mathcal{P}(E)$. On notera $A \cup B$, $A \cap B$ pour respectivement l'union et l'intersection des ensembles A et B. Lorsque A est une partie de E, on notera A^c pour le complémentaire de A dans E.

On appelle *produit cartésien* de deux ensembles E et F l'ensemble des couples formés d'un élément de E et d'un élément de F :

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

Pour $n \geq 1$ un entier, on note $E^n = E \times \cdots \times E$ le produit cartésien de E par lui même n fois. E^n peut aussi se définir[1] récursivement par $E^1 = E$, et $E^{n+1} = E \times E^n$.

Intuitivement, une *application* f d'un ensemble E vers un ensemble F est un objet qui associe à chaque élément e d'un ensemble E un unique élément $f(e)$ de F. Formellement, une fonction f d'un ensemble E vers un ensemble F est une partie Γ de $E \times F$. Son *domaine* est l'ensemble des $x \in E$ tel que $(x, y) \in \Gamma$ pour un certain $y \in E$. Son *image* est l'ensemble des $y \in F$ tel que $(x, y) \in \Gamma$ pour un certain $x \in E$. Une *application* f d'un ensemble E vers un ensemble F est une fonction dont le domaine est E.

Une *famille* $(x_i)_{i \in I}$ *d'éléments d'un ensemble* X est une application d'un ensemble I dans X. I est appelé l'ensemble des indices, et l'image par cette application de l'élément $i \in I$ est notée x_i.

Le produit cartésien se généralise à une famille d'ensembles :

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

L'union et l'intersection se généralisent à une famille quelconque de parties d'un ensemble E. Soit $(A_i)_{i \in I}$ une famille de parties de E.

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

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

On notera $\mathbb{N}$ l'ensemble des entiers naturels, $\mathbb{R}$ l'ensemble des réels, et $\mathbb{C}$ l'ensemble des complexes. $\mathbb{Z}$ est un anneau. $\mathbb{R}$ et $\mathbb{C}$ sont des corps. On notera $\mathbb{R}^{\geq 0}$ l'ensemble des réels positifs ou nuls.

1.1.2 Alphabets, Mots, Langages

Nous rappelons maintenant quelques définitions élémentaires sur les mots et les langages. La terminologie, empruntée à la linguistique, rappelle que les premiers travaux sont issus de la modélisation de la langue naturelle.

On fixe un ensemble Σ que l'on appelle *alphabet*. Les éléments de Σ sont appelés des *lettres* ou des symboles. Un *mot* w sur l'alphabet Σ est une suite finie $w_1 w_2 \cdots w_n$ de lettres de Σ. L'entier n est appelé la *longueur* du mot w. Il est noté $|w|$.

Le mot vide ϵ est le seul mot de longueur 0. Un *langage sur* Σ est un ensemble de mots sur Σ. L'ensemble de tous les mots sur l'alphabet Σ est noté Σ^*.

Exemple 1.1 $\{0,1\}^*$ *désigne l'ensemble des mots sur l'alphabet* $\Sigma = \{0,1\}$. *Par exemple,* $00001101 \in \{0,1\}^*$.

[1] Il y a une bijection entre les objets définis par les deux définitions.

On définit une opération de *concaténation* sur les mots : la concaténation du mot $u = u_1u_2\cdots u_n$ et du mot $v = v_1v_2\cdots v_m$ est le mot noté $u.v$ défini par $u_1u_2\cdots u_nv_1v_2\cdots v_m$, c'est-à-dire le mot dont les lettres sont obtenues en juxtaposant les lettres de v à la fin de celles de u. L'opération de concaténation notée . est associative, mais non-commutative. Le mot vide est un élément neutre à droite et à gauche de cette opération. On appelle aussi Σ^* le *monoïde* (libre) sur l'alphabet Σ (car l'opération de concaténation lui donne une structure de monoïde).

On note aussi uv pour la concaténation $u.v$. En fait, tout mot $w_1w_2\cdots w_n$ peut se voir comme $w_1.w_2\cdots .w_n$, où w_i représente le mot de longueur 1 réduit à la lettre w_i. Cette confusion entre les lettres et les mots de longueur 1 est souvent très pratique.

Exemple 1.2 *Par exemple, si Σ est l'ensemble $\Sigma = \{a, b\}$, aaab est le mot de longueur 4 dont les trois premières lettres sont a, et la dernière est b.*

Lorsque i est un entier, on écrit w^i pour le mot obtenu en concaténant i fois le mot w : w^0 est le mot vide ϵ, w^1 est le mot w, et w^{i+1} est $w.w.\cdots, w$ où il y a i fois le mot w. Autrement dit, $w^{i+1} = w^iw = ww^i$ pour tout entier i.

Exemple 1.3 *En utilisant la confusion précédente entre lettres et mots de longueur 1, aaabbc peut aussi s'écrire a^3b^2c.*

Un mot u est un *préfixe* d'un mot w, s'il existe un mot z tel que $w = u.z$. C'est un *préfixe propre* si $u \neq w$. Un mot u est un *suffixe* d'un mot w s'il existe un mot z tel que $w = z.u$.

1.1.3 Changement d'alphabet

Il est souvent utile de pouvoir réécrire un mot sur un alphabet en un mot sur un autre alphabet. Par exemple, on a souvent besoin en informatique de coder en binaire, c'est-à-dire avec l'alphabet $\Sigma = \{0, 1\}$.

Une façon de faire pour changer d'alphabet est de procèder par réécriture lettre par lettre.

Exemple 1.4 *Par exemple, si Σ est l'alphabet $\Sigma = \{a, b, c\}$, et $\Gamma = \{0, 1\}$, on peut coder les mots de Σ^* sur Γ^* par la fonction h telle que $h(a) = 01$, $h(b) = 10$, $h(c) = 11$. Le mot abab se code alors par $h(abab) = 01100110$, c'est-à-dire par le mot obtenu en codant lettre par lettre.*

Très formellement, étant donnés deux alphabets Σ et Γ, un *homomorphisme* est une application de Σ^* dans Γ^* telle que

- $h(\epsilon) = \epsilon$
- $h(u.v) = h(u).h(v)$ pour tous mots u et v.

En fait, tout homomorphisme est parfaitement déterminé par son image sur les lettres de Σ. Il s'étend alors aux mots de Σ^* par

$$h(w_1w_2\cdots w_n) = h(w_1).h(w_2).\ldots.h(w_n)$$

pour tout mot $w = w_1 w_2 \cdots w_n$: c'est en fait une conséquence du théorème 2.5 du chapitre 2.

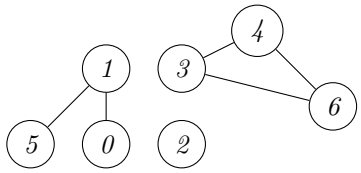
1.1.4 Graphes

Un *graphe* $G = (V, E)$ est donné par un ensemble V, dont les éléments sont appelés *sommets*, ou *nœuds*, et d'une partie de $E \subset V \times V$, dont les éléments sont appelés des *arcs*. Un *chemin* de s à t est une suite $(s = s_0, \ldots, s_n = t)$ de nœuds tels que, pour $1 \leq i \leq n$, (s_{i-1}, s_i) soit un arc. Un *chemin simple* est un chemin qui ne passe pas deux fois par le même sommet. Un *circuit* est un chemin de longueur non nulle dont l'origine coïncide avec l'extrémité.

Si les arcs ne sont pas orientés, c'est-à-dire, si l'on considère qu'à chaque fois qu'il y a l'arc (u, v) il y a aussi l'arc (v, u) (et réciproquement), on dit que le graphe G est *non orienté*, et les éléments de E sont appelés des *arêtes*. Lorsqu'il y a une arête entre u et v, c'est-à-dire lorsque $(u, v) \in E$, on dit que u et v sont voisins. Le degré d'un sommet u est le nombre de ses voisins.

Exemple 1.5 *Le graphe (non-orienté)* $G = (V, E)$ *avec*
- $V = \{0, 1, \ldots, 6\}$
- $E = \{(0, 1), (3, 4), (5, 1), (6, 3), (6, 4)\}$.

est représenté ci-dessous.



Un graphe est dit *connexe* si deux quelconques de ses nœuds sont reliés par un chemin.

Exemple 1.6 *Le graphe de l'exemple 1.5 n'est pas connexe.*

1.1.5 Arbres

Les arbres sont omniprésents en informatique. En fait, plusieurs notions distinctes se cachent sous cette terminologie : arbres libres, arbres enracinés, arbres ordonnés, etc.

Il y a par ailleurs plusieurs façons de présenter les arbres, et ces différentes notions. Essentiellement, on peut le faire en partant de la théorie des graphes, c'est-à-dire de la notion de graphe, ou alors en partant de définitions inductives (récursives).

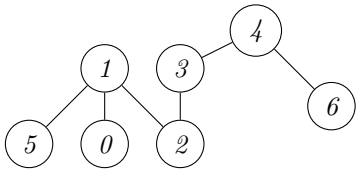
Puisque nous reviendrons sur les définitions inductives de plusieurs de ces classes d'arbres dans le chapitre 2, présentons ici les arbres en partant de la notion de graphe.

On va présenter dans ce qui suit des familles d'arbres de plus en plus contraints : dans l'ordre, on va présenter les arbres libres, puis les arbres enracinés, les arbres ordonnés.

Arbres libres

Un *arbre libre* est un graphe non-orienté connexe et sans circuit. On appelle *feuille* un nœud de l'arbre qui ne possède qu'un seul voisin. Un sommet qui n'est pas une feuille est appelé *un sommet interne*.

Exemple 1.7 (Un arbre libre) *Représentation graphique d'un arbre libre, dont les feuilles sont les nœuds* $5, 0$ *et* 6*.*

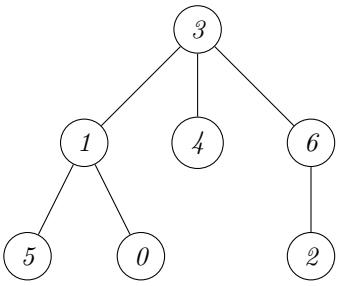


Presque tous nos arbres seront étiquetés : soit A un ensemble dont les éléments sont appelés des *étiquettes*. Un *arbre étiqueté par* A est la donnée d'un arbre $G = (V, E)$ et d'une application qui associe à chaque sommet de V un élément de A.

Arbre enraciné

Un *arbre enraciné* ou *arbre* est un arbre libre muni d'un sommet distingué, appelé sa *racine*. Soit T un arbre de racine r.

Exemple 1.8 (Un arbre) *Représentation graphique d'un arbre, dont les feuilles sont les nœuds* $5, 0, 4$ *et* 2*. On représente la racine* 3 *en haut.*



Pour tout sommet x, il existe un chemin simple unique de r à x. Tout sommet y sur ce chemin est un *ancêtre* de x, et x est un *descendant* de y. Le *sous-arbre* de racine x est l'arbre contenant tous les descendants de x. L'avant-dernier sommet y sur l'unique chemin reliant r à x est le *parent* (ou le *père* ou la *mère*) de x, et x est un *enfant* (ou un *fils* ou une *fille*) de y.

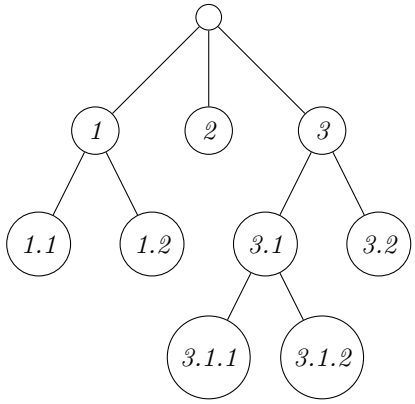
L'*arité* d'un sommet est le nombre de ses enfants. Un sommet sans enfant est une *feuille*, un sommet d'arité strictement positive est appelé *sommet interne*. La *hauteur* d'un arbre T est la longueur maximale d'un chemin reliant sa racine à une feuille. Un arbre réduit à un seul nœud est de hauteur 0.

Arbres ordonnés

Un *arbre ordonné* (on dit aussi *arbre plan*) est un arbre dans lequel l'ensemble des enfants de chaque nœud est totalement ordonné. Autrement dit, pour chaque sommet interne d'arité k, on la notion de 1ier fils, 2ième fils, ..., $k$ème fils.

Par exemple, un livre structuré en chapitres, sections, etc se présente comme un arbre ordonné.

Exemple 1.9 (Arbre ordonné de la table des matières d'un livre)



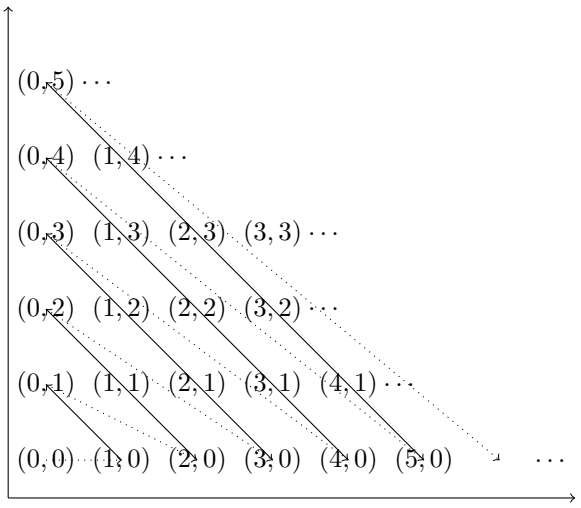
Autres notions d'arbre

Le concept *d'arbre binaire* est assez différent des définitions d'arbre libre, arbre enraciné et arbre ordonné. Il est présenté dans la section 2.4.2 du chapitre 2.

Les *termes* sont des arbres ordonnés étiquetés particuliers. Ils sont présentés dans la section 2.4.3 du chapitre 2.

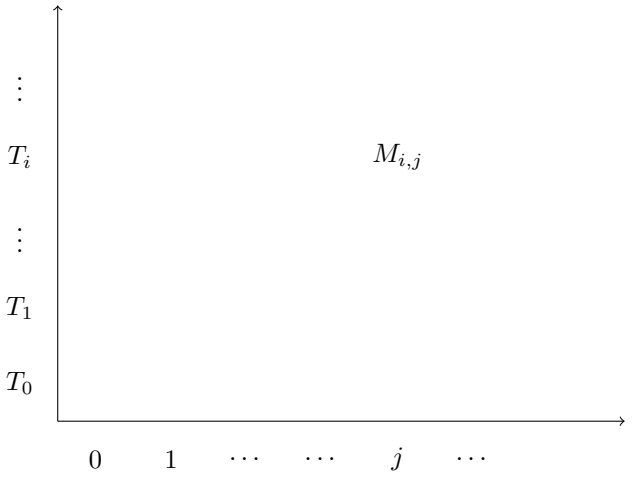
1.2 La méthode de diagonalisation

Rappelons que $\mathbb{N}^2 = \mathbb{N} \times \mathbb{N}$ est dénombrable : il est possible de mettre en correspondance $\mathbb{N}$ avec $\mathbb{N}^2$. Nous allons illustrer graphiquement une façon de parcourir les couples d'entiers.



Par contre, les parties de $\mathbb{N}$ ne sont pas dénombrables : cela peut se voir par la *méthode de diagonalisation* dûe à Cantor.

Illustrons-le graphiquement. Supposons que l'on puisse énumérer les parties de $\mathbb{N}$, et notons les T_1, T_2, $\ldots T_n$ $\ldots$Chaque partie T_i de $\mathbb{N}$ peut se voir comme la ligne i du tableau $M = (M_{i,j})_{i,j}$ à entrées dans $\{0,1\}$ dont l'élément $M_{i,j}$ est 1 si et seulement si l'élément j est dans la $i$ème partie de $\mathbb{N}$.



On considère alors la partie T^* obtenue en "inversant la diagonale de M" : formellement, on considère $T^* = \{j | M_{j,j} = 0\}$. Cette partie de $\mathbb{N}$ n'est pas dans l'énumération, car sinon elle devrait avoir un numéro j_0 : si $j_0 \in T^*$, alors on devrait avoir $M_{j_0,j_0} = 1$ par définition de M, et $M_{j_0,j_0} = 0$ par définition de T^* : impossible. Si $j_0 \notin T^*$, alors on devrait avoir $M_{j_0,j_0} = 0$ par définition de M, et $M_{j_0,j_0} = 1$ par définition de T^* : impossible.

Cet argument est à la base de certains raisonnements en calculabilité, comme nous le verrons.

1.3 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture de [Arnold and Guessarian, 2005] ou du polycopié du cours INF421, ou des polycopiés des cours de l'école polytechnique de première année et des cours de classes préparatoires.

Bibliographie La partie sur les arbres est essentiellement reprise du polycopié de INF421. Le reste du chapitre est inspiré de différentes sources dont le polycopié de INF561, [Hopcroft et al., 2001] et [Arnold and Guessarian, 2005]. L'introduction est essentiellement reprise de [Sipser, 1997].

Chapitre 2

Récursivité et induction

2.1 Motivation

Les définitions récursives sont omniprésentes en informatique. Elles sont présentes à la fois dans les langages de programmation, mais aussi présentes dans de nombreux concepts que l'on manipule.

Exemple 2.1 (Listes en JAVA) *Par exemple, en JAVA, lorsqu'on définit*

```
class Liste {
  int contenu;
  Liste suivant;
  }}
Liste lst;
```

on définit la classe Liste de façon récursive (inductive) : en utilisant dans la définition de la classe, le champ "suivant" du type de la classe Liste elle même.

Exemple 2.2 (Arbres ordonnés) *Nous avons défini les arbres ordonnés dans le chapitre précédent en passant par la notion de graphe. Une alternative naturelle serait de présenter les arbres ordonnés par une définition récursive : un arbre ordonné est soit vide, soit réduit à un sommet (une racine), soit constitué d'un sommet (une racine) et une liste (ordonnée) d'arbres ordonnés (ses fils).*

Dans ce chapitre, nous nous attardons sur les définitions inductives d'ensembles et de fonctions, qui permettent de donner un sens à des définitions récursives.

Nous discutons, par ailleurs comment il est possible de faire des preuves sur des structures définies inductivement, en introduisant les preuves par induction structurelle.

2.2 Raisonnement par récurrence sur l'ensemble $\mathbb{N}$

L'induction structurelle est une généralisation de la preuve par récurrence : revenons sur cette dernière pour avoir les idées au clair.

Lorsque l'on raisonne sur les entiers, le *premier principe d'induction* aussi appelé *principe de récurrence mathématique* est un mode de raisonnement particulièrement utile.

Théorème 2.1 *Soit $P(n)$ un prédicat (une propriété) dépendant de l'entier n. Si les deux conditions suivantes sont vérifiées :*

(B) $P(0)$ est vrai ;

(I) $P(n)$ implique $P(n+1)$ pour tout n ;

alors pour tout entier n, $P(n)$ est vrai.

Démonstration: Le raisonnement se fait par l'absurde. Considérons $X = \{k \in \mathbb{N} | P(k) \text{ est faux}\}$. Si X est non vide, il admet un plus petit élément n. D'après la condition (B), $n \neq 0$, et donc $n-1$ est un entier, et $P(n-1)$ est vrai par définition de X. On obtient une contradiction avec la propriété (I) appliquée pour l'entier $n-1$. □

Pour faire une preuve par récurrence, on établit donc une propriété en 0 (cas de base), et on établit que la propriété est *héréditaire*, ou *inductive* : $P(n)$ implique $P(n+1)$ pour tout n.

Le concept de preuve inductive généralise cette idée à d'autres ensembles que les entiers, à savoir aux ensembles qui se définissent inductivement.

2.3 Définitions inductives

Les définitions inductives visent à définir des parties d'un ensemble E.

Remarque 2.1 *Cette remarque, pour les puristes, peut être évitée dans une première lecture.*

Nous nous restreignons dans ce document au cadre où l'on souhaite définir par induction des objets qui correspondent à des parties d'un ensemble déjà connu E. Nous faisons cela pour éviter les subtilités et paradoxes de la théorie des ensembles.

Le lecteur très attentif pourra observer que l'on considérera dans la suite très souvent l'écriture syntaxique des objets plutôt que les objets eux-mêmes. En effet, en faisant ainsi, on garantit que l'on se place sur l'ensemble $E = \Sigma^*$ pour un certain alphabet Σ, et on évite de se poser la question de l'existence de l'ensemble E sous-jacent dans les raisonnements qui suivent.

Par exemple, pour formaliser complètement l'exemple 2.1 plus haut, on chercherait plutôt à définir une représentation syntaxique des listes plutôt que les listes.

Lorsque l'on veut définir un ensemble, ou une partie d'un ensemble, une façon de faire est de le définir de façon *explicite*, c'est-à-dire en décrivant précisément quels sont ses éléments.

Exemple 2.3 *Les entiers pairs peuvent se définir par* $P = \{n | \exists k \in \mathbb{N}\ n = 2*k\}$.

Malheureusement, ce n'est pas toujours aussi facile, et il est souvent beaucoup plus commode de définir un ensemble par une définition *inductive*. Un exemple typique de définition inductive est une définition comme celle-ci :

Exemple 2.4 *Les entiers pairs correspondent aussi au plus petit ensemble qui contient* 0 *et tel que si* n *est pair, alors* $n + 2$ *est pair.*

Remarque 2.2 *Observons que l'ensemble des entiers tout entier vérifie bien que* 0 *est un entier, et que si* n *est un entier* $n + 2$ *aussi. Il y a donc besoin de dire que c'est le plus petit ensemble avec cette propriété.*

2.3.1 Principe général d'une définition inductive

Intuitivement, une partie X se définit inductivement si on peut la définir avec la donnée explicite de certains éléments de X et de moyens de construire de nouveaux éléments de X à partir d'éléments de X.

De façon générique, dans une définition inductive,

- certains éléments de l'ensemble X sont donnés explicitement (c'est-à-dire on se donne un ensemble B d'éléments de X, qui constitue l'ensemble de base de la définition inductive) ;
- les autres éléments de l'ensemble X sont définis en fonction d'éléments appartenant déjà à l'ensemble X selon certaines règles (c'est-à-dire on se donne des règles R de formation d'éléments, qui constituent les étapes inductives de la définition récursive).

On considère alors le plus petit ensemble qui contient B et qui est *clos* (on dit aussi *stable* ou *fermé*) par les règles R de formation.

2.3.2 Formalisation : Premier théorème du point fixe

Formellement, tout cela se justifie par le théorème qui suit.

Définition 2.1 (Définition inductive) *Soit* E *un ensemble. Une* définition inductive *d'une partie* X *de* E *consiste à se donner*

- *un sous ensemble non vide* B *de* E *(appelé* ensemble de base*)*
- *et d'un ensemble de* règles R *: chaque règle* $r_i \in R$ *est une fonction (qui peut être partielle)* r_i *de* $E^{n_i} \to E$ *pour un certain entier* n_i.

Théorème 2.2 (Théorème du point fixe) *A une définition inductive correspond un plus petit ensemble qui vérifie les propriétés suivantes :*

(B) *il contient* B *:* $B \subset X$ *;*

(I) *il est stable par les règles de R : pour chaque règle $r_i \in R$, pour tout $x_1, \cdots, x_{n_i} \in X$, on a $r_i(x_1, \cdots, x_{n_i}) \in X$.*

On dit que cet ensemble est alors défini inductivement.

Démonstration: Soit $\mathcal{F}$ l'ensemble des parties de E vérifiant (B) et (I). L'ensemble $\mathcal{F}$ est non vide car il contient au moins un élément : en effet, l'ensemble E vérifie les conditions (B) et (I) et donc $E \in \mathcal{F}$.

On peut alors considérer X défini comme l'intersection de tous les éléments de $\mathcal{F}$. Formellement,

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

Puisque B est inclus dans chaque $Y \in \mathcal{F}$, B est inclus dans X. Donc X vérifie la condition (B).

L'ensemble obtenu vérifie aussi (I). En effet, considérons une règle $r_i \in R$, et des $x_1, \cdots, x_{n_i} \in X$. On a $x_1, \cdots, x_{n_i} \in Y$ pour chaque $Y \in \mathcal{F}$. Pour chaque tel Y, puisque Y est stable par la règle r_i, on doit avoir $r(x_1, \cdots, x_{n_i}) \in Y$. Puisque cela est vrai pour tout $Y \in \mathcal{F}$, on a aussi $r(x_1, \cdots, x_{n_i}) \in X$, ce qui prouve que X est stable par la règle r_i.

X est le plus petit ensemble qui vérifie les conditions (B) et (I), car il est par définition inclus dans tout autre ensemble vérifiant les conditions (B) et (I). □

2.3.3 Différentes notations d'une définition inductive

Notation 2.1 *On note souvent une définition inductive sous la forme*

(B) $x \in X$
avec une telle ligne pour chaque $x \in B$
(ou éventuellement on écrit $B \subset X$ pour éviter de préciser chaque x) ;

(I) $x_1, \cdots x_{n_i} \in X \Rightarrow r_i(x_1, \cdots, x_{n_i}) \in X$
avec une telle ligne pour chaque règle $r_i \in R$.

Exemple 2.5 *Selon cette convention, la définition inductive des entiers pairs (Exemple 2.4) se note*

(B) $0 \in P$;
(I) $n \in P \Rightarrow n + 2 \in P$.

Exemple 2.6 *Soit $\Sigma = \{(,)\}$ l'alphabet constitué de la parenthèse ouvrante et de la parenthèse fermante. L'ensemble $D \subset \Sigma^*$ des parenthésages bien formés, appelé langage de Dyck, est défini inductivement par*

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

Notation 2.2 *On préfère parfois écrire une définition inductive sous la forme de* règles de déduction *:*

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

Le principe de cette notation est qu'un trait horizontal signifie une règle de déduction. Ce qui est écrit au dessus est une hypothèse. Ce qui est écrit en dessous est une conclusion. Si ce qui est au dessus est vide, alors c'est que la conclusion est valide sans hypothèse.

Notation 2.3 *On écrit aussi parfois directement*

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

2.4 Applications

2.4.1 Quelques exemples

Exemple 2.7 ($\mathbb{N}$) *La partie X de $\mathbb{N}$ définie inductivement par*

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

n'est autre que $\mathbb{N}$ tout entier.

Exemple 2.8 (Σ^*) *La partie X du monoïde Σ^*, où Σ est un alphabet, définie inductivement par*
(B) $\epsilon \in X$;
(I) $w \in X \Rightarrow wa \in X$, pour chaque $a \in \Sigma$;
n'est autre que Σ^* tout entier.

Exemple 2.9 (Langage $\{a^n bc^n\}$) *Le langage L sur l'alphabet $\Sigma = \{a, b\}$ des mots de la forme $a^n bc^n$, $n \in \mathbb{N}$, se définit inductivement par*
(B) $b \in L$;
(I) $w \in L \Rightarrow awc \in L$.

2.4.2 Arbres binaires étiquetés

Reprenons le texte suivant du polycopié de INF421 : "la notion d'arbre binaire est assez différente des définitions d'arbre libre, arbre enraciné et arbre ordonné. Un *arbre binaire* sur un ensemble fini de sommets est soit vide, soit l'union disjointe d'un sommet appelé sa *racine*, d'un arbre binaire appelé *sous-arbre gauche*, et d'un arbre binaire appelé *sous-arbre droit*. Il est utile de représenter un arbre binaire non vide sous la forme d'un triplet $A = (A_g, r, A_d)$."

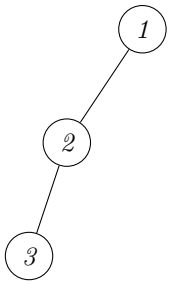
On obtient immédiatement une définition inductive de l'écriture des arbres binaires étiquetés à partir de ce texte.

Exemple 2.10 (Arbres binaires étiquetés) *L'ensemble AB des arbres binaires étiquetés par l'ensemble A est la partie de Σ^*, où $\Sigma = A \cup \{\emptyset, (,), ,\}$, définie inductivement par*

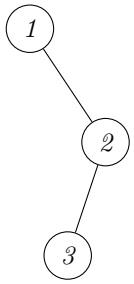
(B) $\emptyset \in AB$
(I) $g, d \in AB \Rightarrow (g, a, d) \in AB$, *pour chaque* $a \in A$.

Remarque 2.3 *Attention : un arbre binaire n'est pas un arbre ordonné dont tous les nœuds sont d'arité au plus* 2.

Exemple 2.11 *Par exemple, l'arbre binaire étiqueté*



et l'arbre binaire étiqueté



ne sont pas les mêmes, car le premier correspond à $(((\emptyset, 3, \emptyset), 2, \emptyset), 1, \emptyset)$ *et le second à* $(\emptyset, 1, ((\emptyset, 3, \emptyset), 2, \emptyset))$. *Pourtant si on considère ces arbres comme des arbres ordonnés, ce sont les mêmes.*

2.4.3 Termes

Les termes sont des arbres ordonnés étiquetés particuliers. Ils jouent un rôle essentiel dans beaucoup de structures en informatique.

Soit $F = \{f_0, f_1, \cdots, f_n, \cdots\}$ un ensemble de symboles, appelés *symboles de fonctions*. A chaque symbole f est associé un entier $a(f) \in F$, que l'on appelle *arité de* f et qui représente le nombre d'arguments du symbole de fonction f. On note F_i pour le sous-ensemble des symboles de fonctions d'arité i. Les symboles de fonctions d'arité 0 sont appelés des *constantes*.

Soit Σ l'alphabet $\Sigma = F \cup \{(,), , \}$ constitué de F et de la parenthèse ouvrante, de la parenthèse fermante, et de la virgule.

Définition 2.2 (Termes sur F) *L'ensemble* T *des termes construit sur* F *est la partie de* Σ^* *définie inductivement par :*

(B) $F_0 \subset T$
(c'est-à-dire : les constantes sont des termes)
(I) $t_1, t_2, \cdots, t_n \in T \Rightarrow f(t_1, t_2, \cdots, t_n) \in T$
pour chaque entier n, pour chaque symbole $f \in F_n$ d'arité n.

Remarque 2.4 *Dans la définition plus haute, on parle bien de mots sur l'alphabet Σ : $f(t_1, t_2, \cdots, t_n)$ désigne le mot dont la première lettre est f, la seconde $($, les suivantes celle de t_1, etc.*

Exemple 2.12 *Par exemple, on peut fixer $F = \{0, 1, f, g\}$, avec 0 et 1 d'arité 0 (ce sont des constantes), f d'arité 2 et g d'arité 1.*

$f(0, g(1))$ est un terme sur F. $f(g(g(0)), f(1, 0))$ est un terme sur F. $f(1)$ n'est pas un terme sur F.

Les termes sur F sont des arbres ordonnés étiquetés particuliers : les sommets sont étiquetés par les symboles de fonctions de F, et un sommet étiqueté par un symbole d'arité k possède exactement k fils.

2.5 Preuves par induction

On va devoir régulièrement prouver des propriétés sur les éléments d'un ensemble X défini implicitement. Cela s'avère possible en utilisant ce que l'on appelle la *preuve par induction*, parfois appelée *preuve par induction structurelle*, qui généralise le principe de la preuve par récurrence.

Théorème 2.3 (Preuve par induction) *Soit $X \subset E$ un ensemble défini inductivement à partir d'un ensemble de base B et de règles R. Soit $\mathcal{P}$ un* prédicat *exprimant une propriété d'un élément $x \in E$: c'est-à-dire une propriété $\mathcal{P}(x)$ qui est soit vraie soit fausse en un élément $x \in E$.*

Si les conditions suivantes sont vérifiées :

(B) *$\mathcal{P}(x)$ est vérifiée pour chaque élément $x \in B$;*

(I) *$\mathcal{P}$ est* héréditaire*, c'est-à-dire stable par les règles de R : Formellement, pour chaque règle $r_i \in R$, pour chaque $x_1, \cdots, x_{n_i} \in E$, $\mathcal{P}(x_1), \cdots, \mathcal{P}(x_{n_i})$ vraies impliquent $\mathcal{P}(x)$ vraie en $x = r(x_1, \cdots, x_{n_i})$.*

Alors $\mathcal{P}(x)$ est vraie pour chaque élément $x \in X$.

Démonstration: On considère l'ensemble Y des éléments $x \in E$ qui vérifient le prédicat $\mathcal{P}(x)$. Y contient B par la propriété (B). Y est stable par les règles de R par propriété (I). L'ensemble X, qui est le plus petit ensemble contenant B et stable par les règles de R, est donc inclus dans Y. □

Remarque 2.5 *La preuve par induction généralise bien la preuve par récurrence. En effet, $\mathbb{N}$ se définit inductivement comme dans l'exemple 2.7. Une preuve par induction sur cette définition inductive de $\mathbb{N}$ correspond à une preuve par récurrence, c'est-à-dire aux hypothèses du théorème 2.1.*

Exemple 2.13 *Pour prouver par induction que tous les mots du langage défini implicitement dans l'exemple 2.9 possèdent autant de a que de c, il suffit de constater que c'est vrai pour le mot réduit à une lettre b, qui possède* 0 *fois la lettre a et la lettre c, et que si cela est vrai pour le mot w alors le mot awb possède aussi le même nombre de fois la lettre a que la lettre c, à savoir exactement une fois de plus que dans w.*

Exercice 2.1 *Montrer que dans un arbre binaire, le nombre de sommets n vérifie $n \leq 2f - 1$, où f est le nombre de feuilles.*

Exercice 2.2 *Montrer que tout mot du langage de Dyck a autant de parenthèses fermantes qu'ouvrantes.*

2.6 Dérivations

2.6.1 Écriture explicite des éléments : Second théorème du point fixe

Nous avons vu jusque-là plusieurs exemples d'ensembles X définis inductivement. L'existence de chaque ensemble X découle du théorème 2.2, et en fait de l'équation (2.1) utilisée dans la preuve de celui-ci.

On parle de *définition de X par le haut*, puisque l'équation (2.1) définit X à partir de sur-ensembles de celui-ci. Cela a clairement l'avantage de montrer facilement l'existence d'ensembles définis inductivement, ce que nous avons abondamment utilisé jusque-là.

Cependant, cela a le défaut de ne pas dire quels sont exactement les éléments des ensembles X obtenus.

Il est en fait aussi possible de définir chaque ensemble X défini inductivement par *le bas*. On obtient alors une définition explicite des éléments de X, avec en prime une façon de les obtenir explicitement.

C'est ce que dit le résultat suivant :

Théorème 2.4 (Définition explicite d'un ensemble défini implicitement) *Chaque ensemble X défini implicitement à partir de l'ensemble de base B et des règles R s'écrit aussi*

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

où $(X_n)_{n \in \mathbb{N}}$ est la famille de parties de E définie par récurrence par

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

Autrement dit, tout élément de X est obtenu en partant d'éléments de B et en appliquant un nombre fini de fois les règles de R pour obtenir des nouveaux éléments.

Démonstration: Il suffit de prouver que cet ensemble est le plus petit ensemble qui contient B et qu'il est stable par les règles de R.

D'une part, puisque $X_0 = B$, B est bien dans l'union des X_n. D'autre part, si l'on prend une règle $r_i \in R$, et des éléments $x_1, \cdots, x_{n_i}$ dans l'union des X_n, par définition chaque x_j est dans un X_{k_j} pour un entier k_j. Puisque les ensembles X_i sont croissants (i.e. $X_i \subset X_{i+k}$ pour tout k, ce qui se prouve trivialement par récurrence sur k), tous les $x_1, \cdots, x_{n_i}$ sont dans X_{n_0} pour $n_0 = \max(k_1, \cdots, k_{n_i})$. On obtient immédiatement que $r(x_1, \cdots, x_{n_i})$ est dans X_{n_0+1}, ce qui prouve qu'il est bien dans l'union des X_n.

Enfin, c'est le plus petit ensemble, car tout ensemble qui contient B et qui est stable par les règles de R doit contenir chacun des X_n. Cela se prouve par récurrence sur n. C'est vrai au rang $n = 0$, car un tel ensemble doit contenir X_0 puisqu'il contient B. Supposons l'hypothèse au rang n, c'est-à-dire X contient X_n. Puisque les éléments de X_{n+1} sont obtenus à partir d'éléments de $X_n \subset X$ en appliquant une règle $r_i \in R$, X contient chacun de ces éléments. □

2.6.2 Arbres de dérivation

La définition *par le bas* de X du théorème précédent invite à chercher à garder la trace de comment chaque élément est obtenu, en partant de X et en appliquant les règles de R.

La notion naturelle est celle de terme, sur un ensemble de symboles F bien choisi : on considère que chaque élément b de la base B est un symbole d'arité 0. A chaque règle $r_i \in R$ on associe un symbole d'arité n_i. Un terme t sur cet ensemble de symboles est appelé une *dérivation*.

A chaque dérivation t est associé un élément $h(t)$ comme on s'y attend : si t est d'arité 0, on lui associe l'élément b de B correspondant. Sinon, t est de la forme $r_i(t_1, \cdots, t_{n_i})$, pour une règle $r_i \in R$ et pour des termes $t_1, \cdots, t_{n_i}$, et on associe à t le résultat de la règle r_i appliquée aux éléments $h(t_1), \cdots, h(t_{n_i})$.

On peut alors reformuler le théorème précédent de la façon suivante.

Proposition 2.1 *Soit un ensemble X défini implicitement à partir de l'ensemble de base B et des règles R. Soit D l'ensemble des dérivations correspondant à B et à R. Alors*

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

Autrement dit, X est précisément l'ensemble des éléments de E qui possèdent une dérivation.

Définition 2.3 *On dit qu'une définition inductive de X est* non ambiguë *si la fonction h précédente est injective.*

Intuitivement, cela signifie qu'il n'existe qu'une unique façon de construire chaque élément de X. C'est le cas pour toutes les définitions inductives précédentes.

Exemple 2.14 *La définition suivante de $\mathbb{N}^2$ est ambigüe.*
(B) $(0,0) \in \mathbb{N}^2$;
(I) $(n,m) \in \mathbb{N}^2 \Rightarrow (n+1,m) \in \mathbb{N}^2$;

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

En effet, on peut par exemple obtenir $(1, 1)$ *en partant de* $(0, 0)$ *et en appliquant la deuxième puis la troisième règle, mais aussi en appliquant la troisième règle puis la deuxième règle.*

2.7 Fonctions définies inductivement

Nous aurons parfois besoin de définir des fonctions sur des ensembles X définis inductivement. Cela peut se faire facilement lorsque X admet une définition non ambigüe.

Théorème 2.5 (Fonction définie inductivement) *Soit $X \subset E$ un ensemble défini inductivement de façon non ambigüe à partir de l'ensemble de base B et des règles R. Soit Y un ensemble.*

Pour qu'une application f de X dans Y soit parfaitement définie, il suffit de se donner :

(B) *la valeur de $f(x)$ pour chacun des éléments $x \in B$;*

(I) *pour chaque règle $r_i \in R$, la valeur de $f(x)$ pour $x = r_i(x_1, \cdots, x_{n_i})$ en fonction de la valeur de $f(x_1)$, ..., et $f(x_{n_i})$.*

Autrement dit, informellement, si l'on sait "programmer récursivement", c'est-à-dire "décrire de façon récursive la fonction", alors la fonction est parfaitement définie sur l'ensemble inductif X.

Démonstration: On entend par l'énoncé, qu'il existe alors une unique application f de X dans Y qui satisfait ces contraintes. Il suffit de prouver que pour chaque $x \in X$, la valeur de f en x est définie de façon unique. Cela se prouve facilement par induction : c'est vrai pour les éléments $x \in B$. Si cela est vrai en $x_1, \cdots, x_{n_i}$, cela est vrai en $x = r_i(x_1, \cdots, x_{n_i})$: la définition de X étant non ambigüe, x ne peut être obtenu que par la règle r_i à partir de $x_1, \cdots, x_{n_i}$. Sa valeur est donc parfaitement définie par la contrainte pour la règle r_i. □

Exemple 2.15 *La fonction factorielle $Fact$ de $\mathbb{N}$ dans $\mathbb{N}$ se définit inductivement par*

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

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

Exemple 2.16 *La hauteur h d'un arbre binaire étiqueté se définit inductivement par*

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

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

On observera que ces définitions ne sont essentiellement que la traduction de comment on peut les programmer de façon récursive.

2.8 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture de [Arnold and Guessarian, 2005]. Pour une présentation plus générale des définitions inductives et des théorèmes de points fixes, nous suggérons la lecture de [Dowek, 2008].

Bibliographie Ce chapitre a été rédigé en s'inspirant essentiellement de [Dowek, 2008] et [Arnold and Guessarian, 2005].

Chapitre 3

Calcul propositionnel

La *logique propositionnelle* permet essentiellement de discuter des connecteurs grammaticaux comme la négation, la conjonction et la disjonction, en composant des propositions à partir de propositions données. Ces connecteurs sont parfois appelés *aristotéliciens* , car ils ont été mis en évidence par Aristote.

Le *calcul propositionnel* permet essentiellement de parler de *fonctions booléennes*, c'est-à-dire de fonctions de $\{0,1\}^n \to \{0,1\}$. En effet, les variables, c'est-à-dire *les propositions*, ne peuvent prendre que deux valeurs, *vrai* ou *faux*.

Le calcul propositionnel tient une grande place en informatique : ne serait-ce parce que nos ordinateurs actuels sont digitaux, et travaillent en binaire. Ce qui fait que nos processeurs sont essentiellement constitués de portes binaires du type de celles que l'on va étudier dans ce chapitre.

D'un point de vue expressivité logique, le calcul propositionnel reste très limité : par exemple, on ne peut pas écrire en calcul propositionnel l'existence d'un objet ayant une propriété donnée. Le calcul des prédicats, plus général, que nous étudierons dans le chapitre 5, permet lui d'exprimer des propriétés d'objets et des relations entre objets, et plus généralement de formaliser le raisonnement mathématique.

Puisque le calcul propositionnel forme toutefois la base commune de nombreux systèmes logiques, et nous allons nous y attarder dans ce chapitre.

3.1 Syntaxe

Pour définir formellement et proprement ce langage, nous devons distinguer la syntaxe de la sémantique : la syntaxe décrit comment on écrit les formules. La sémantique décrit leur sens.

Fixons un ensemble fini ou dénombrable $\mathcal{P} = \{p_0, p_1, \cdots\}$ de symboles que l'on appelle *variables propositionnelles*.

Définition 3.1 (Formules propositionnelles) *L'ensemble des formules propositionnelles $\mathcal{F}$ sur $\mathcal{P}$ est le langage sur l'alphabet $\mathcal{P} \cup \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, (,)\}$ défini inductivement par les règles suivantes :*

(B) *il contient* $\mathcal{P}$ *: toute variable propositionnelle est une formule propositionnelle ;*
(I) *si* $F \in \mathcal{F}$ *alors* $\neg F \in \mathcal{F}$ *;*
(I) *si* $F, G \in \mathcal{F}$ *alors* $(F \wedge G) \in \mathcal{F}$, $(F \vee G) \in \mathcal{F}$, $(F \Rightarrow G) \in \mathcal{F}$, *et* $(F \Leftrightarrow G) \in \mathcal{F}$.

Il s'agit d'une définition inductive qui est légitime par les considérations du chapitre précédent. Il s'agit d'une définition inductive non ambiguë : on peut reformuler ce résultat par la proposition suivante, parfois appelé *théorème de lecture unique*.

Proposition 3.1 (Décomposition / Lecture unique) *Soit* F *une formule propositionnelle. Alors* F *est d'une, et exactement d'une, des formes suivantes*

1. *une variable propositionnelle* $p \in \mathcal{P}$ *;*
2. $\neg G$, *où* G *est une formule propositionnelle ;*
3. $(G \wedge H)$ *où* G *et* H *sont des formules propositionnelles ;*
4. $(G \vee H)$ *où* G *et* H *sont des formules propositionnelles ;*
5. $(G \Rightarrow H)$ *où* G *et* H *sont des formules propositionnelles ;*
6. $(G \Leftrightarrow H)$ *où* G *et* H *sont des formules propositionnelles.*

De plus dans les cas 2., 3., 4., 5. et 6., il y a unicité de la formule G *et de la formule* H *avec cette propriété.*

Le fait qu'une formule se décompose toujours dans un des 6 cas plus hauts est facile à établir inductivement. L'unicité de la décomposition découle de l'exercice suivant :

Exercice 3.1 *Montrer que la définition inductive précédente est non-ambiguë, c'est-à-dire que* G *et* H *sont uniquement définis dans chacun des cas plus haut.*
On pourra procéder de la façon suivante.
- *Montrer que dans toute formule* F *le nombre de parenthèses ouvrantes est égal au nombre de parenthèses fermantes.*
- *Montrer que dans tout mot* M *préfixe de* F, *on a* $o(M) \geq f(M)$, *où* $o(M)$ *est le nombre de parenthèses ouvrantes, et* $f(M)$ *le nombre de parenthèses fermantes.*
- *Montrer que pour toute formule* F *dont le premier symbole est une parenthèse ouvrante, et pour tout mot* M *préfixe propre de* F, *on a* $o(M) > f(M)$.
- *Montrer que tout mot* M *préfixe propre de* F *n'est pas une formule.*
- *En déduire le résultat.*

On appelle *sous-formule* de F une formule qui apparaît dans la décomposition récursive de F.

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

FIG. 3.1 – Tableau de vérité.

3.2 Sémantique

Nous allons maintenant définir la *sémantique* d'une formule propositionnelle, c'est-à-dire le sens qu'on lui donne.

La *valeur de vérité* d'une formule se définit comme l'interprétation de cette formule, une fois que l'on s'est fixé la valeur de vérité des variables propositionnelles : le principe est d'interpréter les symboles $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$ par la négation logique, le *ou* logique, le *et* logique, l'implication et la double implication logique.

Formellement, une *valuation* est une distribution de valeurs de vérité aux variables propositionnelles, c'est-à-dire une fonction de $\mathcal{P}$ vers $\{0, 1\}$. Dans tout ce qui suit, 0 représente faux, et 1 représente vrai.

Proposition 3.2 *Soit v une valuation.*

Par le théorème 2.5, il existe une unique fonction $\overline{v}$ définie sur tout $\mathcal{F}$ qui vérifie les conditions suivantes

(B) $\overline{v}$ étend v : pour toute variable propositionnelle $p \in \mathcal{P}$, $\overline{v}(p) = v(p)$;

(I) la négation s'interprète par la négation logique :
si F est de la forme $\neg G$, alors $\overline{v}(F) = 1$ ssi $\overline{v}(G) = 0$;

(I) $\wedge$ s'interprète comme le et logique :
si F est de la forme $G \wedge H$, alors $\overline{v}(F) = 1$ ssi $\overline{v}(G) = 1$ et $\overline{v}(H) = 1$;

(I) $\vee$ s'interprète comme le ou logique :
si F est de la forme $G \vee H$, alors $\overline{v}(F) = 1$ ssi $\overline{v}(G) = 1$ ou $\overline{v}(H) = 1$;

(I) $\Rightarrow$ s'interprète comme l'implication logique :
si F est de la forme $G \Rightarrow H$, alors $\overline{v}(F) = 1$ ssi $\overline{v}(H) = 1$ ou $\overline{v}(G) = 0$;

(I) $\Leftrightarrow$ s'interprète comme l'équivalence logique :
si F est de la forme $G \Leftrightarrow H$, alors $\overline{v}(F) = 1$ ssi $\overline{v}(G) = \overline{v}(H)$.

On écrit $v \models F$ pour $\overline{v}(F) = 1$, et on dit que v est *un modèle* de F, ou que v satisfait F. On note $v \not\models F$ dans le cas contraire. La valeur de $\overline{v}(F)$ pour la valuation v est appelée *la valeur de vérité de F sur v*.

On représente souvent les conditions de la définition précédente sous la forme d'un *tableau de vérité* : voir la figure 3.1.

3.3 Tautologies, formules équivalentes

On souhaite classer les formules selon leur interprétation. Une classe particulière de formules est celles qui sont toujours vraies que l'on appelle les *tautologies*.

Définition 3.2 (Tautologie) *Une* tautologie *est une formule F qui est satisfaite par toute valuation.*

Définition 3.3 (Equivalence) *Deux formules F et G sont dites* équivalentes *si pour toute valuation v, $\overline{v}(F) = \overline{v}(G)$. On écrit dans ce cas $F \equiv G$.*

Exemple 3.1 *La formule $p \vee \neg p$ est une tautologie. Les formules p et $\neg\neg p$ sont équivalentes.*

Remarque 3.1 *Il est important de bien comprendre que $\equiv$ est un symbole que l'on utilise pour écrire une relation entre deux formules, mais que $F \equiv G$ n'est pas une formule propositionnelle.*

Exercice 3.2 *Montrer que $\equiv$ est une relation d'équivalence sur les formules.*

3.4 Quelques faits élémentaires

Exercice 3.3 *Montrer que pour toutes formules F et G, les formules suivantes sont des tautologies :*

$$(F \Rightarrow F),$$

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

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

Exercice 3.4 (Idempotence) *Montrer que pour toute formule F on a les équivalences :*

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

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

Exercice 3.5 (Associativité) *Montrer que pour toutes formules F, G, H on a les équivalences :*

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

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

En raison de l'associativité, on note souvent $F_1 \vee F_2 \vee \cdots \vee F_k$ pour $(((F_1 \vee F_2) \vee F_3) \cdots \vee F_k)$, et $F_1 \wedge F_2 \wedge \cdots \wedge F_k$ pour $(((F_1 \wedge F_2) \wedge F_3) \cdots \wedge F_k)$.

Exercice 3.6 (Commutativité) *Montrer que pour toutes formules F et G on a les équivalences :*

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

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

Exercice 3.7 (Distributivité) *Montrer que pour toutes formules F, G, H on a les équivalences :*

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

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

Exercice 3.8 (Lois de Morgan) *Montrer que pour toutes formules F et G on a les équivalences :*

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

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

Exercice 3.9 (Absorption) *Montrer que pour toutes formules F et G on a les équivalences :*

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

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

3.5 Remplacements d'une formule par une autre équivalente

Nous connaissons maintenant quelques équivalences entre formules, mais nous allons maintenant nous convaincre qu'on peut utiliser ces équivalences de façon compositionnelle : si l'on remplace dans une formule une sous-formule par une formule équivalente, on obtient une formule équivalente.

3.5.1 Une remarque simple

Observons tout d'abord que la valeur de vérité d'une formule ne dépend que des variables propositionnelles présentes dans la formule : lorsque F est une formule, on notera $F(p_1, \cdots, p_n)$ pour dire que la formule F s'écrit avec les variables propositionnelles $p_1, \cdots, p_n$ seulement.

Proposition 3.3 *Soit $F(p_1, \cdots, p_n)$ une formule. Soit v une valuation. La valeur de vérité de F sur v ne dépend que de la valeur de v sur $\{p_1, p_2, \cdots, p_n\}$.*

Démonstration: La propriété s'établit facilement par induction structurelle. □

3.5.2 Substitutions

Il nous faut définir ce que signifie remplacer p par G dans une formule F, noté $F(G/p)$. Cela donne la définition un peut pédante qui suit, mais nous devons en passer par là :

Définition 3.4 (Substitution de p par G dans F) *La formule $F(G/p)$ est définie par induction sur la formule F :*

(*B*) *Si F est la variable propositionnelle p, alors $F(G/p)$ est la formule G ;*
(*B*) *Si F est une variable propositionnelle q, avec $q \neq p$, alors $F(G/p)$ est la formule F ;*
(*I*) *Si F est de la forme $\neg H$, alors $F(G/p)$ est la formule $\neg H(G/p)$;*
(*I*) *Si F est de la forme $(F_1 \vee F_2)$, alors $F(G/p)$ est la formule $(F_1(G/p) \vee F_2(G/p))$;*
(*I*) *Si F est de la forme $(F_1 \wedge F_2)$, alors $F(G/p)$ est la formule $(F_1(G/p) \wedge F_2(G/p))$;*
(*I*) *Si F est de la forme $(F_1 \Rightarrow F_2)$, alors $F(G/p)$ est la formule $(F_1(G/p) \Rightarrow F_2(G/p))$;*
(*I*) *Si F est de la forme $(F_1 \Leftrightarrow F_2)$, alors $F(G/p)$ est la formule $(F_1(G/p) \Leftrightarrow F_2(G/p))$.*

3.5.3 Compositionnalité de l'équivalence

On obtient le résultat promis : si l'on remplace dans une formule une sous-formule par une formule équivalente, on obtient une formule équivalente.

Proposition 3.4 *Soient F, F', G et G' des formules. Soit p une variable propositionnelle.*
- *Si F est une tautologie, alors $F(G/p)$ aussi.*
- *Si $F \equiv F'$, alors $F(G/p) \equiv F'(G/p)$.*
- *Si $G \equiv G'$ alors $F(G/p) \equiv F(G'/p)$.*

Exercice 3.10 *Prouver le résultat par induction structurelle.*

3.6 Système complet de connecteurs

Proposition 3.5 *Toute formule propositionnelle est équivalente à une formule qui est construite uniquement avec les connecteurs $\neg$ et $\wedge$.*

Démonstration: Cela résulte d'une preuve par induction sur la formule. C'est vrai pour les formules qui correspondent à des variables propositionnelles. Supposons la propriété vraie pour les formules G et H, c'est-à-dire supposons que G (respectivement H) est équivalente à une formule G' (respectivement H') construite uniquement avec les connecteurs $\neg$ et $\wedge$.

Si F est de la forme $\neg G$, alors F est équivalente à $\neg G'$ et l'hypothèse d'induction est préservée.

Si F est de la forme $(G \wedge H)$, alors F est équivalente à $(F' \wedge H')$ et la propriété d'induction est préservée.

Si F est de la forme $(G \vee H)$, en utilisant la deuxième loi de Morgan et le fait que $K \equiv \neg\neg K$ pour éliminer les doubles négations, on obtient que $F \equiv \neg(\neg G' \wedge \neg H')$ qui est bien construite en utilisant uniquement les connecteurs $\neg$ et $\wedge$.

Si F est de la forme $(G \Rightarrow H)$, alors F est équivalente à $(\neg G' \vee H')$ qui est équivalente à une formule construite uniquement avec les connecteurs $\neg$ et $\wedge$ par les cas précédents.

Si F est de la forme $(G \Leftrightarrow H)$, alors F est équivalente à $(G' \Rightarrow H') \wedge (H' \Rightarrow G')$ qui est équivalente à une formule construite uniquement avec les connecteurs $\neg$ et $\wedge$ par les cas précédents. □

Un ensemble de connecteurs qui a la propriété plus haut pour $\{\neg, \wedge\}$ est appelé un *système complet de connecteurs*.

Exercice 3.11 *Montrer que $\{\neg, \vee\}$ est aussi un système complet de connecteurs.*

Exercice 3.12 *Donner un connecteur logique binaire tel qu'à lui seul il constitue un système complet de connecteurs.*

3.7 Complétude fonctionnelle

Supposons $\mathcal{P} = \{p_1, p_2, \cdots, p_n\}$ fini. Soit V l'ensemble des valuations sur $\mathcal{P}$. Puisqu'une valuation est une fonction de $\{0,1\}^n$ dans $\{0,1\}$, V contient 2^n éléments.

Chaque formule F sur $\mathcal{P}$ peut être vue comme une fonction de V dans $\{0,1\}$, que l'on appelle *valeur de vérité de F* : cette fonction est la fonction qui à une valuation v associe la valeur de vérité de la formule sur cette valuation.

Il y a 2^{2^n} fonctions de V dans $\{0,1\}$. La question qui se pose est de savoir si toutes les fonctions peuvent s'écrire comme des formules. La réponse est positive :

Théorème 3.1 (Complétude fonctionnelle) *Supposons $\mathcal{P} = \{p_1, p_2, \cdots, p_n\}$ fini. Soit V l'ensemble des valuations sur $\mathcal{P}$. Toute fonction f de V dans $\{0,1\}$ est la valeur de vérité d'une formule F sur $\mathcal{P}$.*

Démonstration: La preuve se fait par récurrence sur le nombre de variables propositionnelles n.

Pour $n = 1$, il y a quatre fonctions de $\{0,1\}^1$ dans $\{0,1\}$, qui se représentent par les formules p, $\neg p$, $p \vee \neg p$, $p \wedge \neg p$.

Supposons la propriété vraie pour $n-1$ variables propositionnelles. Considérons $\mathcal{P} = \{p_1, \cdots, p_n\}$ et soit f une fonction de $\{0,1\}^n$ dans $\{0,1\}$. Chaque valuation v' sur $\{p_1, p_2, \cdots, p_{n-1}\}$ peut se voir comme la restriction d'une valuation sur $\{p_1, \cdots, p_n\}$. Soit f_0 (respectivement f_1) la restriction de f à la valuation v telle que $v(p_n) = 0$ (resp. $v(p_n) = 1$). Les fonctions f_0 et f_1 sont des fonctions définies des valuations sur $\{p_1, \cdots, p_{n-1}\}$ dans $\{0,1\}$ et se représentent par des formules $G(p_1, \cdots, p_{n-1})$ et $H(p_1, \cdots, p_{n-1})$ respectivement par hypothèse de récurrence. La fonction f peut alors se représenter par la formule

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

ce qui prouve l'hypothèse de récurrence au rang n. □

3.8 Formes normales

On cherche souvent à ramener les formules sous une forme équivalente la plus simple possible.

Définition 3.5 *Un* littéral *est une variable propositionnelle ou sa négation, i.e. de la forme p, ou $\neg p$, pour $p \in \mathcal{P}$.*

Définition 3.6 *Une* forme normale disjonctive *est une disjonction $F_1 \vee F_2 \cdots \vee F_k$ de k formules, $k \geq 1$ où chaque formule F_i, $1 \leq i \leq k$ est une conjonction $G_1 \wedge G_2 \cdots \wedge G_\ell$ de ℓ littéraux (ℓ pouvant dépendre de i).*

Exemple 3.2 *$((p \wedge q \wedge \neg r) \vee (q \wedge \neg p))$ est une forme normale disjonctive.*

Définition 3.7 *Une* forme normale conjonctive *est une conjonction $F_1 \wedge F_2 \cdots \wedge F_k$ de k formules, $k \geq 1$ où chaque formule F_i, $1 \leq i \leq k$ est une disjonction $G_1 \vee G_2 \cdots \vee G_\ell$ de ℓ littéraux (ℓ pouvant dépendre de i).*

Exemple 3.3 *$\neg p \vee q$ et $(\neg p \vee q) \wedge \neg r$ sont des formes normales conjonctives.*

Théorème 3.2 *Toute formule sur un nombre fini de variables propositionnelles est équivalente à une formule en forme normale conjonctive.*

Théorème 3.3 *Toute formule sur un nombre fini de variables propositionnelles est équivalente à une formule en forme normale disjonctive.*

Démonstration: Comme dans la dernière preuve, ces deux théorèmes se prouvent par récurrence sur le nombre n de variables propositionnelles.

Dans le cas $n = 1$, on a déjà considéré dans la preuve précédente des formules qui couvrent tous les cas possibles et qui sont en fait à la fois en forme normale conjonctive et disjonctive.

On suppose la propriété vraie pour $n-1$ variables propositionnelles. Soit f la fonction valeur de vérité associée à la formule $F(p_1, \cdots, p_n)$. Comme dans la dernière preuve, on peut construire une formule qui représente f, en écrivant une formule de la forme

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

Par hypothèse de récurrence, G et H sont équivalentes à des formules en forme normale disjonctive

$$G \equiv (G_1 \vee G_2 \vee \cdots \vee G_k)$$
$$H \equiv (H_1 \vee H_2 \vee \cdots \vee H_\ell)$$

On peut alors écrire

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

qui est en forme normale disjonctive et

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

qui est aussi en forme normale disjonctive. La fonction f est donc représentée par la disjonction de ces deux formules, et donc par une formule en forme normale disjonctive.

Si l'on veut obtenir F en forme normale conjonctive, l'hypothèse d'induction produit deux formes normales conjonctives G et H. L'équivalence que l'on utilise est alors

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

□

3.9 Théorème de compacité

3.9.1 Satisfaction d'un ensemble de formules

On se donne cette fois un ensemble Σ de formules. On cherche à savoir quand est-ce qu'on peut satisfaire toutes les formules de Σ.

Commençons par fixer la terminologie.

Définition 3.8 *Soit Σ un ensemble de formules.*
- *Une valuation* satisfait *Σ si elle satisfait chaque formule de Σ. On dit aussi dans ce cas que cette valuation est* un modèle *de Σ.*
- *Σ est dit* satisfiable*, ou* consistant*, s'il existe une valuation qui satisfait Σ.*
- *Σ est dit* inconsistant*, ou* contradictoire*, si Σ n'est pas satisfiable.*

Définition 3.9 (Conséquence) *Soit F une formule. La formule F est dite* une conséquence *de Σ si tout modèle de Σ est un modèle de F. On note alors $\Sigma \vdash F$.*

Exemple 3.4 *La formule q est une conséquence de l'ensemble de formules $\{p, p \Rightarrow q\}$. L'ensemble de formules $\{p, p \Rightarrow q, \neg q\}$ est inconsistant.*

On peut déjà se convaincre du résultat suivant, qui relève d'un jeu sur les définitions.

Proposition 3.6 *Toute formule F est une conséquence d'un ensemble Σ de formules si et seulement si $\Sigma \cup \{\neg F\}$ est inconsistant.*

Démonstration: Si toute valuation qui satisfait Σ satisfait F, alors il n'y a pas de valuation qui satisfait $\Sigma \cup \{\neg F\}$. Réciproquement, par l'absurde : s'il y a une valuation qui satisfait Σ et qui ne satisfait pas F, alors cette valuation satisfait Σ et $\neg F$. □

Exercice 3.13 *Montrer que pour toutes formules F et F', $\{F\} \vdash F'$ si et seulement si $F \Rightarrow F'$ est une tautologie.*

Plus fondamentalement, on a le résultat surprenant et fondamental suivant.

Théorème 3.4 (Théorème de compacité) *Soit Σ un ensemble de formules construites sur un ensemble dénombrable $\mathcal{P}$ de variables propositionnelles.*

Alors Σ est satisfiable si et seulement si toute partie finie de Σ est satisfiable.

Remarque 3.2 *Remarquons que l'hypothèse $\mathcal{P}$ dénombrable n'est pas nécessaire, si l'on accepte d'utiliser l'hypothèse de Zorn (l'axiome du choix). On se limitera au cas $\mathcal{P}$ dénombrable dans ce qui suit.*

En fait, ce théorème peut se reformuler sous la forme suivante

Théorème 3.5 (Théorème de compacité (2ième version)) *Soit Σ un ensemble de formules construites sur un ensemble dénombrable $\mathcal{P}$ de variables propositionnelles.*

Alors Σ est inconsistant si et seulement si Σ possède une partie finie inconsistante.

Ou encore sous la forme suivante :

Théorème 3.6 (Théorème de compacité (3ième version)) *Pour tout ensemble Σ de formules propositionnelles, et pour toute formule propositionnelle F construites sur un ensemble dénombrable $\mathcal{P}$ de variables propositionnelles, F est une conséquence de Σ si et seulement si F est une conséquence d'une partie finie de Σ.*

L'équivalence des trois formulations n'est qu'un simple exercice de manipulations de définitions. Nous allons prouver la première version du théorème.

Une des implications est triviale : si Σ est satisfiable, alors toute partie de Σ est satisfiable, et en particulier les parties finies.

Nous allons donner deux preuves de l'autre implication.

Une première preuve qui fait référence à des notions de topologie, en particulier de compacité, et qui s'adresse à ceux qui connaissent ces notions, et qui sont amateurs de topologie.

Démonstration:[Preuve topologique] L'espace topologique $\{0,1\}^{\mathcal{P}}$ (muni de la topologie produit) est un espace compact, car il s'obtient comme un produit de compacts (Théorème de Tychonoff).

Pour chaque formule propositionnelle $F \in \mathcal{F}$, l'ensemble $\overline{F}$ des valuations qui la satisfont est un ouvert dans $\{0,1\}^{\mathcal{P}}$, car la valeur de vérité d'une formule ne dépend que d'un nombre fini de variables, celles qui apparaissent dans la formule. Il est également fermé puisque celles qui ne satisfont pas F sont celles qui satisfont $\neg F$.

L'hypothèse du théorème entraîne que toute intersection finie de $\overline{F}$ pour $F \in \Sigma$ est non-vide. Comme $\{0,1\}^{\mathcal{P}}$ est compact, l'intersection de tous les $\overline{F}$ pour $F \in \Sigma$ est donc non-vide. □

Voici une preuve qui évite la topologie.

Démonstration:[Preuve directe] Considérons $\mathcal{P} = \{p_1, p_2, \cdots, p_k, \cdots\}$ une énumération de $\mathcal{P}$.

Nous allons prouver le lemme suivant : supposons qu'il existe une application v de $\{p_1, p_2, \cdots, p_n\}$ dans $\{0, 1\}$ telle que tout sous-ensemble fini de Σ ait un modèle dans lequel $p_1, \cdots, p_n$ prennent les valeurs $v(p_1)$, ..., $v(p_n)$. Alors on peut étendre v à $\{p_1, p_2, \cdots, p_{n+1}\}$ avec la même propriété.

En effet, si $v(p_{n+1}) = 0$ ne convient pas, alors il existe un ensemble fini U_0 de Σ qui ne peut pas être satisfait quand $p_1, \cdots, p_n, p_{n+1}$ prennent les valeurs respectives $v(p_1)$, ..., $v(p_n)$ et 0. Si U est un sous-ensemble fini quelconque de Σ, alors d'après l'hypothèse faite sur v, $U_0 \cup U$ a un modèle dans lequel $p_1, \cdots, p_n$ prennent les valeurs $v(p_1), \cdots, v(p_n)$. Dans ce modèle, la proposition p_{n+1} prend donc la valeur 1. Autrement dit, tout sous-ensemble fini U de Σ a un modèle dans lequel $p_1, \cdots, p_n, p_{n+1}$ prennent les valeurs respectives $v(p_1)$, ..., $v(p_n)$ et 1. Dit encore autrement, soit $v(p_{n+1}) = 0$ convient auquel cas on peut fixer $v(p_{n+1}) = 0$, soit $v(p_{n+1}) = 0$ ne convient pas auquel cas on peut fixer $v(p_{n+1}) = 1$ qui convient.

En utilisant ce lemme, on définit ainsi une valuation v telle que, par récurrence sur n, pour chaque n, tout sous-ensemble fini de Σ a un modèle dans lequel $p_1, \cdots, p_n$ prennent les valeurs $v(p_1)$, ..., $v(p_n)$.

Il en résulte que v satisfait Σ : en effet, soit F une formule de Σ. F ne dépend que d'un ensemble fini $p_{i_1}, p_{i_2}, \cdots, p_{i_k}$ de variables propositionnelles (celles qui apparaissent dans F). En considérant $n = \max(i_1, i_2, \cdots, i_k)$, chacune de ces variables p_{i_j} est parmi $\{p_1, \cdots, p_n\}$. Nous savons alors que le sous ensemble fini $\{F\}$ réduit à la formule F admet un modèle dans lequel $p_1, \cdots, p_n$ prennent les valeurs $v(p_1)$, ..., $v(p_n)$, i.e. F est satisfaite par v.

□

3.10 Exercices

Exercice 3.14 (Théorème d'interpolation) *Soient F et F' telle que $F \Rightarrow F'$ soit une tautologie. Montrer qu'il existe une formule propositionnelle C, dont les variables propositionnelles apparaissent dans F et F', telle que $F \Rightarrow C$ et $C \Rightarrow F'$ soient deux tautologies.*

Exercice 3.15 (Application de la compacité au coloriage de graphes) *Montrer qu'un graphe est coloriable avec k couleurs si et seulement si chacun de ses sous-graphes finis est coloriable avec k couleurs.*

Exercice 3.16 (Applications de la compacité à la théorie des groupes) *Un groupe G est dit totalement ordonné si on a sur G une relation d'ordre total telle que $a \leq b$ implique $ac \leq bc$ et $ca \leq cb$ pour tous $a, b, c \in G$. Montrer que pour qu'un groupe G puisse être ordonné, il faut et il suffit que tout sous-groupe de G engendré par un ensemble fini d'éléments de G puisse être ordonné.*

3.11 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture de l'ouvrage [Cori and Lascar, 1993a] et de

[Lassaigne and de Rougemont, 2004].

Bibliographie Ce chapitre a été rédigé en s'inspirant essentiellement de ces deux ouvrages : [Cori and Lascar, 1993a] et [Lassaigne and de Rougemont, 2004].

Chapitre 4

Démonstrations

L'objectif de ce chapitre est de commencer à aborder la question fondamentale suivante : qu'est-ce qu'une démonstration ?

Pour cela, plus précisément, on va se focaliser dans ce chapitre sur le problème suivant : on se donne une formule propositionnelle F, et on veut déterminer si F est une tautologie. Une tautologie est aussi appelée un *théorème*, et on dit aussi que F est *valide*.

Cela va nous amener à décrire des algorithmes particuliers...

4.1 Introduction

Une première méthode pour résoudre ce problème, qui est celle que nous avons utilisée dans le chapitre précédent, est la suivante : si F est de la forme $F(p_1, \cdots, p_n)$, on teste pour chacune des 2^n valuations v, c'est-à-dire pour les 2^n fonctions de $\{0,1\}^n$ dans $\{0,1\}$, si v est bien un modèle de F. Si c'est le cas, alors F est une tautologie. Dans tout autre cas, F n'est pas une tautologie. Il est facile de programmer une telle méthode dans son langage de programmation favori.

La bonne nouvelle est que cette méthode existe : le problème de déterminer si une formule est une tautologie, est *décidable*, en utilisant la terminologie que nous verrons dans les chapitres suivants, c'est-à-dire peut bien se résoudre par un programme informatique.

Remarque 4.1 *Cette observation peut sembler étrange et quelque part se contenter de peu, mais nous verrons que lorsque l'on considère des logiques plus générales, même très simples, même cela devient problématique : il n'existe pas toujours d'algorithme pour déterminer si une formule F est une tautologie.*

Cependant, cette méthode est particulièrement inefficace. Elle a l'inconvénient majeur de garantir que lorsque F est une tautologie, on fera 2^n fois un test du type "la valuation v est-elle un modèle de F ?" Lorsque n est grand, 2^n

explose très vite : si l'on peut effectivement programmer la méthode, le programme obtenu sera en pratique inutilisable, car prenant un temps énorme, dès que l'on considèrera des formules F avec un grand nombre de variables.

Revenons alors à notre problème : on peut se dire que dans le raisonnement classique en mathématique, la méthode usuelle pour prouver qu'une assertion est un théorème est de la *démontrer*.

Si l'on veut faire mieux que la méthode exhaustive précédente, il y a essentiellement deux angles d'attaque. Le premier angle d'attaque est de chercher à s'approcher de la notion de *démonstration* dans le raisonnement usuel : des méthodes de preuve dans l'esprit de la section suivante apparaissent. Le deuxième angle d'attaque est de chercher à produire des algorithmes le plus efficace possible : des méthodes comme les *preuves par résolution* ou *méthodes par tableaux* plus algorithmiques apparaissent alors.

En général, on s'attend à ce qu'une méthode de preuve soit toujours *valide* : elle ne produit que des déductions correctes. Dans tous les cas se pose la question de la *complétude* : est-ce que tous les théorèmes (tautologies) sont prouvables ?

Dans ce qui suit, on présentera trois systèmes de déduction valides et complets : les méthodes *à la Hilbert*, la méthode par résolution et la *méthode des tableaux*. On ne prouvera la validité et la complétude que de la méthode des tableaux.

4.2 Démonstrations à la Frege et Hilbert

Dans ce système de déduction, on part d'un ensemble d'axiomes, qui sont des tautologies, et on utilise une unique règle de déduction, le *modus ponens*, aussi appelé *coupure*, qui vise à capturer un type de raisonnement tout à fait naturel en mathématique.

La règle du modus ponens dit qu'à partir de la formule F et d'une formule $F \Rightarrow G$, on déduit G.

Graphiquement :

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

Exemple 4.1 *Par exemple, à partir de* $(A \wedge B)$ *et de* $(A \wedge B) \Rightarrow C$ *on déduit* C*.*

On considère alors un ensemble d'axiomes, qui sont en fait des instances d'un nombre fini d'axiomes.

Définition 4.1 (Instance) *On dit qu'une formule* F *est une instance d'une formule* G *si* F *s'obtient en substituant certaines variables propositionnelles de* G *par des formules* F_i*.*

Exemple 4.2 *La formule* $((C \Rightarrow D) \Rightarrow (\neg A \Rightarrow (C \Rightarrow D)))$ *est une instance de* $(A \Rightarrow (B \Rightarrow A))$*, en prenant* $(C \Rightarrow D)$ *pour* A*, et* $\neg A$ *pour* B*.*

Définition 4.2 (Axiomes de la logique booléenne) *Un* axiome de la logique booléenne *est n'importe quelle instance d'une des formules suivantes :*

1. $(X_1 \Rightarrow (X_2 \Rightarrow X_1))$ *(axiome 1 pour l'implication) ;*
2. $((X_1 \Rightarrow (X_2 \Rightarrow X_3)) \Rightarrow ((X_1 \Rightarrow X_2) \Rightarrow (X_1 \Rightarrow X_3)))$ *(axiome 2 pour l'implication) ;*
3. $(X_1 \Rightarrow \neg\neg X_1)$ *(axiome 1 pour la négation) ;*
4. $(\neg\neg X_1 \Rightarrow X_1)$ *(axiome 2 pour la négation) ;*
5. $((X_1 \Rightarrow X_2) \Rightarrow (\neg X_2 \Rightarrow \neg X_1))$ *(axiome 3 pour la négation) ;*
6. $(X_1 \Rightarrow (X_2 \Rightarrow (X_1 \wedge X_2)))$ *(axiome 1 pour la conjonction) ;*
7. $((X_1 \wedge X_2) \Rightarrow X_1)$ *(axiome 2 pour la conjonction) ;*
8. $((X_1 \wedge X_2) \Rightarrow X_2)$ *(axiome 3 pour la conjonction) ;*
9. $(X_1 \Rightarrow (X_1 \vee X_2))$ *(axiome 1 pour la disjonction) ;*
10. $(X_2 \Rightarrow (X_1 \vee X_2))$ *(axiome 2 pour la disjonction) ;*
11. $(\neg X_1 \Rightarrow ((X_1 \vee X_2) \Rightarrow X_2))$ *(axiome 3 pour la disjonction).*

On obtient une notion de démonstration.

Définition 4.3 (Démonstration par modus ponens) *Soit T un ensemble de formules propositionnelles, et F une formule propositionnelle. Une* preuve de F à partir de T *est une suite finie $F_1, F_2, \cdots, F_n$ de formules propositionnelles telle que F_n est égale à F, et pour tout i, ou bien F_i est dans T, ou bien F_i est un axiome de la logique booléenne, ou bien F_i s'obtient par modus ponens à partir de deux formules F_j, F_k avec $j < i$ et $k < i$.*

On note $T \vdash F$ si F est prouvable à partir de T. On note $\vdash F$ si $\emptyset \vdash F$, et on dit que F est *prouvable (par modus ponens).*

Exemple 4.3 *Soit F, G, H trois formules propositionnelles. Voici une preuve de $(F \Rightarrow H)$ à partir de $\{(F \Rightarrow G), (G \Rightarrow H)\}$:*

- $F_1 : (G \Rightarrow H)$ *(hypothèse) ;*
- $F_2 : ((G \Rightarrow H) \Rightarrow (F \Rightarrow (G \Rightarrow H)))$ *(instance de l'axiome 1.) ;*
- $F_3 : (F \Rightarrow (G \Rightarrow H))$ *(modus ponens à partir de F_1 et F_2) ;*
- $F_4 : ((F \Rightarrow (G \Rightarrow H)) \Rightarrow ((F \Rightarrow G) \Rightarrow (F \Rightarrow H)))$ *(instance de l'axiome 2.) ;*
- $F_5 : ((F \Rightarrow G) \Rightarrow (F \Rightarrow H))$ *(modus ponens à partir de F_3 et F_4) ;*
- $F_6 : (F \Rightarrow G)$ *(hypothèse) ;*
- $F_7 : (F \Rightarrow H)$ *(modus ponens à partir de F_6 et F_5).*

Cette méthode de preuve est valide : en vérifiant que les axiomes sont des tautologies, il est facile de se convaincre par récurrence sur la longueur n de la preuve du résultat suivant :

Théorème 4.1 (Validité) *Toute formule propositionnelle prouvable est une tautologie.*

Ce qui est moins trivial, et plus intéressant est la réciproque : toute tautologie admet une preuve de ce type.

Théorème 4.2 (Complétude) *Toute tautologie est prouvable (par modus ponens).*

Nous ne donnerons pas la preuve de ce résultat.

On vient de décrire un système de déduction qui est très proche de la notion usuelle de preuve en mathématique. Cependant, ce système n'est pas facilement exploitable pour construire un algorithme qui déterminerait si une formule F est une tautologie.

4.3 Démonstrations par résolution

Nous présentons brièvement la notion de preuve par résolution. Cette méthode de preuve est peut-être moins naturelle, mais est plus simple à implémenter informatiquement.

La résolution s'applique à un ensemble de formules qui sont en forme normale conjonctive. Puisque toute formule propositionnelle peut se mettre sous forme normale conjonctive cela n'est pas restrictif.

Remarque 4.2 *Tout du moins en apparence. En effet, il faut être plus astucieux que dans le chapitre précédent pour transformer une formule en forme normale conjonctive, si l'on ne veut pas faire exploser la taille des formules et implémenter pratiquement la méthode.*

On appelle *clause* une disjonction de littéraux. Rappelons qu'un *littéral* est une variable propositionnelle ou sa négation. On convient de représenter une clause c par l'ensemble des littéraux sur lesquels porte la disjonction.

Exemple 4.4 *On écrit donc* $\{p, \neg q, r\}$ *plutôt que* $p \vee \neg q \vee r$.

Étant donné un littéral u, on note $\overline{u}$ pour le littéral équivalent à $\neg u$: autrement dit, si u est la variable propositionnelle p, $\overline{u}$ vaut $\neg p$, et si u est $\neg p$, $\overline{u}$ est p. Enfin on introduit une clause vide, notée $\square$, dont la valeur est 0 pour toute valuation.

Définition 4.4 (Résolvante) *Soient* C_1, C_2 *deux clauses. On dit que la clause* C *est une résolvante de* C_1 *et* C_2 *s'il existe un littéral* u *tel que :*

- $u \in C_1$;
- $\overline{u} \in C_2$;
- C *est donné par* $(C_1 \backslash \{u\}) \cup (C_2 \backslash \{\overline{u}\})$.

Exemple 4.5 *Les clauses* $\{p, q, r\}$ *et* $\{\neg r, s\}$ *admettent la résolvante* $\{p, q, s\}$.

Exemple 4.6 *Les clauses* $\{p, q\}$ *et* $\{\neg p, \neg q\}$ *admettent deux résolvantes, à savoir* $\{q, \neg q\}$ *et* $\{p, \neg p\}$. *Les clauses* $\{p\}$ *et* $\{\neg p\}$ *admettent la résolvante* $\square$.

Cela donne une notion de preuve.

Définition 4.5 (Preuve par résolution) *Soit T un ensemble de clauses. Une* preuve par résolution de T *est une suite finie $F_1, F_2, \cdots, F_n$ de clauses telle que F_n est égale à $\square$, et pour tout i, ou bien F_i est une clause dans T, ou bien F_i est une résolvante de deux clauses F_j, F_k avec $j < i$ et $k < i$.*

Remarque 4.3 *Le modus ponens, au coeur du système de preuve précédent, consiste à dire qu'à partir de la formule F et d'une formule $(F \Rightarrow G)$, on déduit G. Si l'on considère que la formule $(F \Rightarrow G)$ est équivalente à la formule $(\neg F \vee G)$, le modus ponens peut aussi se voir comme dire qu'à partir de la formule F et d'une formule $(\neg F \vee G)$, on déduit G, ce qui ressemble au concept de résolvante : la résolvante de $\{f\}$ et de $\{\neg f, g\}$ est $\{g\}$.*

D'une certaine façon, la résolvante est un modus ponens généralisé, même si cette analogie n'est qu'une analogie et une preuve dans un système de preuve ne peut pas se traduire directement dans l'autre.

Cette méthode de preuve est valide (sens facile).

Théorème 4.3 (Validité) *Toute clause apparaissant dans une preuve par résolution de T est une conséquence de T.*

En fait, pour prouver une formule, on raisonne en général dans cette méthode de preuve plutôt sur sa négation, et on cherche à prouver que sa négation est contradictoire avec les hypothèses. La validité se formule alors en général plutôt de cette façon.

Corollaire 4.1 (Validité) *Si un ensemble de clauses T admet une preuve par résolution, alors T est contradictoire.*

Elle s'avère complète (sens plus difficile).

Théorème 4.4 (Complétude) *Soit T un ensemble de clauses contradictoires. Il admet une preuve par résolution.*

4.4 Démonstrations par la méthode des tableaux

Nous avons jusque-là uniquement évoqué des systèmes de déduction valides et complets, sans fournir aucune preuve de ce fait. Nous allons étudier plus complètement la méthode des tableaux. Nous avons choisi de développer cette méthode, car elle est très algorithmique et basée sur la notion d'arbre, ce qui appuiera notre argumentaire récurrent sur le fait que la notion d'arbre est partout en informatique.

4.4.1 Principe

Pour simplifier la discussion, on considèrera que les formules propositionnelles ne sont écrites qu'avec les connecteurs $\neg, \wedge, \vee, \Rightarrow$. La formule $(F \Leftrightarrow G)$ sera considérée comme une abréviation de la formule $((F \Rightarrow G) \wedge (G \Rightarrow F))$.

Supposons que l'on veuille prouver qu'une formule F est une tautologie. Si la formule F est de la forme $(F_1 \wedge F_2)$ on peut chercher à prouver F_1 et F_2, et écrire F_1, F_2. Si la formule est de la forme $(F_1 \vee F_2)$, on peut explorer deux possibilités, l'une pour explorer le cas de F_1 et l'autre pour le cas de F_2.

On va ramener toutes les autres possibilités à ces deux configurations, en utilisant les lois de Morgan, et quelques équivalences : si la formule F est de la forme $(F_1 \Rightarrow F_2)$, on va la voir comme $(F_2 \vee \neg F_1)$ et appliquer la règle du $\vee$, et si F est de la forme $\neg(F_1 \Rightarrow F_2)$, comme $(F_1 \wedge \neg F_2)$ et appliquer la règle du $\wedge$. On traite chacun des autres cas à l'aide des lois de Morgan.

En faisant cela de façon systématique, on va construire un arbre, dont la racine est étiquetée par la négation de la formule F : autrement dit, pour prouver une formule F, la méthode commence par la négation de la formule F.

Partons par exemple de la formule F suivante que nous cherchons à prouver :

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

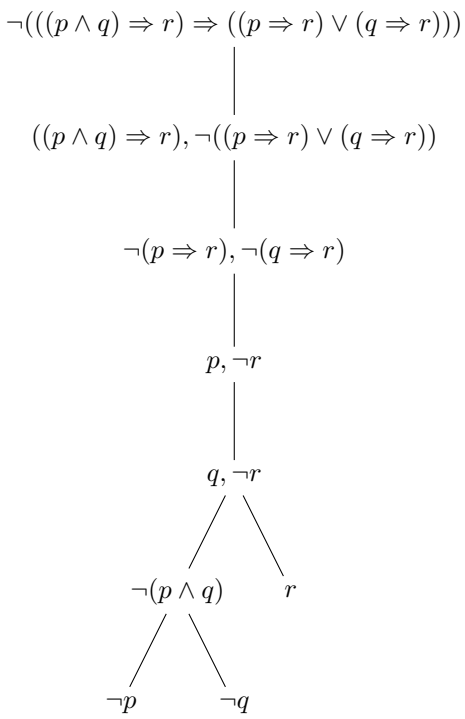
On part donc de la formule $\neg F$, , c'est-à-dire de

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

- en transformant l'implication $\neg(F_1 \Rightarrow F_2)$ en la formule équivalente $(F_1 \wedge \neg F_2)$, on obtient $((p \wedge q) \Rightarrow r) \wedge \neg((p \Rightarrow r) \vee (q \Rightarrow r))$ pour se ramener à la règle du $\wedge$.
- On applique alors la règle du $\wedge$: on considère les formules $((p \wedge q) \Rightarrow r)$ et $\neg((p \Rightarrow r) \vee (q \Rightarrow r))$.
- On considère à l'instant d'après la dernière formule, que l'on peut voir par les lois de Morgan comme $(\neg(p \Rightarrow r) \wedge \neg(q \Rightarrow r))$: on lui associe $\neg(p \Rightarrow r)$ et $\neg(q \Rightarrow r)$ par la règle du $\wedge$.
- On considère alors $\neg(p \Rightarrow r)$, ce qui donne les formules p et $\neg r$.
- On obtient alors q et $\neg r$ à partir de $\neg(q \Rightarrow r)$.
- Si l'on considère maintenant $((p \wedge q) \Rightarrow r)$, que l'on peut voir comme $(r \vee \neg(p \wedge q))$, on a le choix par la règle du $\vee$ entre la formule $\neg(p \wedge q)$ ou r.
- Le cas de r est exclus par l'étape précédente, où l'on avait $\neg r$.
- Dans le premier cas, on a encore le choix entre $\neg p$ ou $\neg q$. Les deux cas sont exclus parce que l'on avait avant p et q.

Puisque toutes les branches mènent à une contradiction, il n'y a aucune situation dans laquelle F pourrait être fausse : on sait alors que F est une tautologie.

Le calcul que l'on vient de faire se représente naturellement par un arbre.

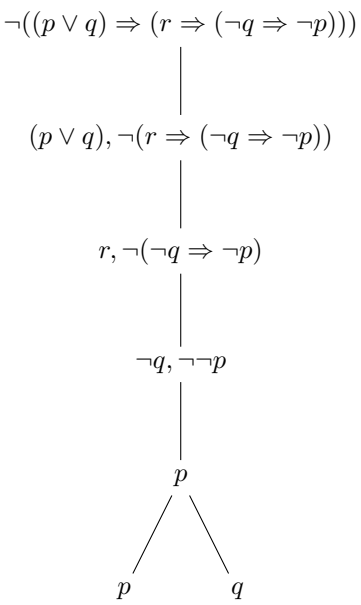


Chaque branche correspond à un scénario possible. Si une branche possède un sommet étiqueté par une formule A telle que $\neg A$ apparaît plus haut sur la même branche (ou l'opposé), alors on cesse de développer cette branche, et on dit que la branche est *close* : cela veut dire qu'il y a une contradiction. Si toutes les branches sont closes, on dit que *l'arbre est clos*, et on sait que tous les scénarios sont exclus.

Considérons maintenant l'exemple de la formule G donnée par

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

Avec la même méthode, on développe un arbre avec comme racine $\neg G$.



Cette fois l'arbre a deux branches. La branche de droite est close. La branche de gauche n'est pas close, et les variables propositionnelles sur cette branche sont $r, \neg q$ et p. Si l'on prend une valuation v telle que $v(r) = 1$, $v(q) = 0$, $v(p) = 1$, la valuation donne la valeur 1 à $\neg G$, et donc 0 à G. Autrement dit, G n'est pas une tautologie. On dit que l'arbre est *ouvert*.

4.4.2 Description de la méthode

Un *tableau* est un arbre binaire étiqueté dont les sommets sont des ensembles de formules propositionnelles et qui est construit récursivement à partir de la racine sommet par sommet en utilisant un nombre fini de fois deux types de règles : les règles α et les règles β.

On rappelle que pour simplifier la discussion, on considère que les formules propositionnelles écrites qu'avec les connecteurs $\neg, \wedge, \vee, \Rightarrow$. La formule $(F \Leftrightarrow G)$ est considérée comme une abréviation de la formule $((F \Rightarrow G) \wedge (G \Rightarrow F))$.

Les formules sont découpées en deux classes, la classe α et la classe β, et à chaque formule on associe inductivement deux autres formules par les règles suivantes :

- Les formules de type α sont les formules du type :
  1. $\alpha = (A \wedge B)$: on lui associe $\alpha_1 = A$ et $\alpha_2 = B$.
  2. $\alpha = \neg(A \vee B)$: on lui associe $\alpha_1 = \neg A$ et $\alpha_2 = \neg B$.
  3. $\alpha = \neg(A \Rightarrow B)$: on lui associe $\alpha_1 = A$ et $\alpha_2 = \neg B$.
  4. $\neg\neg A$: on lui associe $\alpha_1 = \alpha_2 = A$.
- Les formules du type β sont les formules du type :

1. $\beta = \neg(A \wedge B)$: on lui associe $\beta_1 = \neg A$, $\beta_2 = \neg B$.
2. $\beta = (A \vee B)$: on lui associe $\beta_1 = A$, $\beta_2 = B$.
3. $\beta = (A \Rightarrow B)$: on lui associe $\beta_1 = \neg A$, $\beta_2 = B$.

Si B est une branche d'un tableau, on note $\bigcup B$ pour l'ensemble des formules qui apparaissent sur un sommet de B.

Les deux règles de constructions d'un tableau sont les suivantes :

1. une règle α consiste à prolonger une branche finie d'un tableau T par le sommet étiqueté $\{\alpha_1, \alpha_2\}$, où α est une formule de type α qui apparaît sur un sommet de B.
2. une règle β consiste à prolonger une branche finie d'un tableau T par deux fils étiquetés respectivement par $\{\beta_1\}$ et $\{\beta_2\}$, où β est une formule de type β qui apparaît sur un sommet de B.

Remarque 4.4 *Observons que ce n'est pas nécessairement le dernier sommet d'une branche B qui est développé à chaque étape, mais une formule quelque part sur la branche.*

On dit qu'une branche B est *close* s'il existe une formule A telle que A et $\neg A$ apparaissent sur la branche B. Dans le cas contraire la branche est dite *ouverte*.

Une branche B est *développée* si

1. pour toute formule de type α de $\bigcup B$, $\alpha_1 \in \bigcup B$ et $\alpha_2 \in \bigcup B$.
2. pour toute formule de type β de $\bigcup B$, $\beta_1 \in \bigcup B$ ou $\beta_2 \in \bigcup B$.

Un tableau est *développé* si toutes ses branches sont soit closes, soit développées. Un tableau est *clos* si toutes ses branches sont closes. Un tableau est *ouvert* s'il possède une branche ouverte.

Finalement, un tableau pour une formule A (respectivement pour un ensemble de formules Σ) est un tableau dont la racine est étiquetée par $\{A\}$ (respectivement par $\{A | A \in \Sigma\}$).

4.4.3 Terminaison de la méthode

Observons tout d'abord que l'on peut toujours appliquer des règles α ou β jusqu'à obtenir un tableau développé.

Proposition 4.1 *Si Σ est un ensemble fini de formules, alors il existe un tableau développé (et fini) pour Σ.*

Démonstration: Cela se prouve par récurrence sur le nombre n d'éléments de Σ.

Pour le cas $n = 1$, observons que la longueur des formules α_1, α_2, β_1, et β_2 est toujours inférieure strictement à la longueur de α et β. Le processus d'extension des branches qui ne sont pas closes se termine donc nécessairement après un nombre fini d'étapes. Le tableau obtenu au final est développé, sinon il serait possible de l'étendre.

Pour le cas $n > 1$, écrivons $\Sigma = \{F_1, \cdots, F_n\}$. Considérons par hypothèse de récurrence un tableau développé pour $\Sigma = \{F_1, \cdots, F_{n-1}\}$. Si ce tableau est clos ou si F_n est une variable propositionnelle, alors ce tableau est un tableau développé pour Σ. Sinon, on peut étendre toutes les branches ouvertes en appliquant les règles correspondantes à la formule F_n, et en développant les branches obtenues : le processus termine pour la même raison que pour $n = 1$. □

Bien entendu, à partir d'une racine donnée, il y a de nombreuses façons de construire un tableau développé.

4.4.4 Validité et complétude

La méthode précédente donne une méthode de preuve.

Définition 4.6 *On dira qu'une formule F est* prouvable par tableau *s'il existe un tableau clos avec la racine $\{\neg A\}$.*

La méthode est valide.

Théorème 4.5 (Validité) *Toute formule prouvable est une tautologie.*

Démonstration: On dira qu'une branche B d'un tableau est *réalisable* s'il existe une valuation v telle que $v(A) = 1$ pour toute formule $A \in \bigcup B$ et $v(A) = 0$ si $\neg A \in \bigcup A$. Un tableau est dit *réalisable* s'il a une branche réalisable.

Il suffit de prouver le résultat suivant :

Lemme 4.1 *Soit T' une extension immédiate du tableau T : c'est-à-dire le tableau obtenu en appliquant une règle α ou une règle β à T. Si T est réalisable alors T' aussi.*

Ce lemme suffit à prouver le théorème : si F est prouvable, il y a un tableau clos avec $\neg F$ comme racine. Cela veut dire que pour toute branche, il y a une formule A telle que A et $\neg A$ apparaissent sur cette branche, et donc aucune des branches de T n'est réalisable. Par le lemme, cela veut dire que l'on est parti initialement d'un arbre réduit au sommet étiqueté par $\neg F$ qui n'était pas réalisable. Autrement dit, F est une tautologie.

Il reste à prouver le lemme. Soit B une branche réalisable de T, et soit B' la branche de T qui est étendue dans T'. Si $B \neq B'$, alors B reste une branche réalisable de T. Si $B = B'$, alors B est étendue dans T',

1. soit en une branche B_α par l'application d'une règle α ;
2. soit en deux branches B_{β_1} et B_{β_2} par l'application d'une règle β.

Dans le premier cas, soit α la formule utilisée par la règle, et v une valuation réalisant B : de $v(\alpha) = 1$, on déduit $v(\alpha_1) = 1$ et $v(\alpha_2) = 1$: donc v est une valuation qui réalise B_α et le tableau T' est réalisable.

Dans le second cas, soit β la formule utilisée par la règle : de $v(\beta) = 1$, on déduit qu'au moins une des valeurs $v(\beta_1)$ et $v(\beta_2)$ vaut 1 : donc v réalise l'une des branches B_{β_1} et B_{β_2}, et le tableau T' est réalisable. □

4.4.5 Complétude

La méthode est complète : en d'autres termes, la réciproque du théorème précédent est vraie.

Théorème 4.6 (Complétude) *Toute tautologie est prouvable.*

Corollaire 4.2 *Soit F une formule propositionnelle.*
F est une tautologie si et seulement si elle est prouvable.

Le reste de cette sous-section est consacré à prouver ce théorème.

Remarquons tout d'abord que si B est une branche à la fois développée et ouverte d'un tableau T, alors l'ensemble $\bigcup B$ de formules qui apparaissent dans B a les propriétés suivantes :

1. il n'y a aucune variable propositionnelle p telle que $p \in \bigcup B$ et telle que $\neg p \in \bigcup B$;
2. pour toute formule $\alpha \in \bigcup B$, $\alpha_1 \in \bigcup B$ et $\alpha_2 \in \bigcup B$;
3. pour toute formule $\beta \in \bigcup B$, $\beta_1 \in \bigcup B$ ou $\beta_2 \in \bigcup B$.

Lemme 4.2 *Toute branche développée et ouverte d'un tableau est réalisable.*

Démonstration: Soit B une branche développée et ouverte d'un tableau T. On définit une valuation v par :

1. si $p \in \bigcup B$, $v(p) = 1$;
2. si $\neg p \in \bigcup B$, $v(p) = 0$;
3. si $p \notin \bigcup B$ et $\neg p \notin \bigcup B$, on pose (arbitrairement) $v(p) = 1$.

On montre par induction structurelle sur A que : si $A \in \bigcup B$, alors $v(A) = 1$, et si $\neg A \in \bigcup B$, alors $v(A) = 0$.

En effet, c'est vrai pour les variables propositionnelles.

Si A est une formule α, alors $\alpha_1 \in \bigcup B$ et $\alpha_2 \in \bigcup B$: par hypothèse d'induction, $v(\alpha_1) = 1$, $v(\alpha_2) = 1$, et donc $v(\alpha) = 1$.

Si A est une formule β, alors $\beta_1 \in \bigcup B$ ou $\beta_2 \in \bigcup B$: par hypothèse d'induction, $v(\beta_1) = 1$ ou $v(\beta_2) = 1$, et donc $v(\beta) = 1$. □

Proposition 4.2 *S'il existe un tableau clos avec $\neg A$ comme racine, alors tout tableau développé avec la racine $\neg A$ est clos.*

Démonstration: Par l'absurde : soit T un tableau ouvert et développé avec la racine $\neg A$, et B une branche ouverte de T : par le lemme précédent, B est réalisable, et puisque $\neg A$ est dans B, $\neg A$ est satisfiable. A n'est donc pas une tautologie, et donc n'est pas prouvable par tableau : il n'y a donc pas de tableau clos avec la racine $\neg A$. □

On a enfin tous les ingrédients pour prouver le théorème 4.6.

Supposons que A ne soit pas prouvable par tableau, et soit T un tableau développé avec la racine $\neg A$: T n'est pas clos. Comme dans la preuve précédente, si B est une branche ouverte de T, alors B est réalisable, et donc $\neg A$ est satisfiable : autrement dit, A n'est pas une tautologie.

4.4.6 Une conséquence du théorème de compacité

Définition 4.7 *On dira qu'un ensemble* Σ *de formules est* réfutable par tableau *s'il existe un tableau clos avec la racine* Σ.

Corollaire 4.3 *Tout ensemble* Σ *de formules qui n'est pas satisfiable est réfutable par tableau.*

Démonstration: Par le théorème de compacité, un ensemble de formules Σ qui n'est pas satisfiable possède un sous-ensemble fini Σ_0 qui n'est pas satisfiable. Cet ensemble fini de formules a une réfutation par tableau, i.e. il y a un tableau clos avec la racine Σ_0. Ce tableau donne aussi un tableau clos avec la racine Σ. □

4.5 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture de [Cori and Lascar, 1993a], de [Mendelson, 1997] pour les méthodes de preuve basées sur le modus ponens, de [Stern, 1994] pour une présentation simple des méthodes de preuve basées sur la résolution, et de [Lassaigne and de Rougemont, 2004] et [Nerode and Shore, 1997] pour la méthode des tableaux.

Bibliographie Ce chapitre a été rédigé en s'inspirant essentiellement de l'ouvrage [Cori and Lascar, 1993a], de [Dehornoy, 2006] pour la partie sur les méthodes preuves basées sur le modus ponens, de [Stern, 1994] pour la présentation de la preuve par résolution. La section sur la méthode des tableaux est reprise de l'ouvrage [Lassaigne and de Rougemont, 2004].

Chapitre 5

Calcul des prédicats

Le calcul propositionnel reste très limité, et ne permet essentiellement que d'exprimer des opérations booléennes sur des propositions.

Si l'on veut pouvoir raisonner sur des assertions mathématiques, il nous faut autoriser des constructions plus riches. Par exemple, on peut vouloir écrire l'énoncé

$$\forall x((Premier(x) \wedge x > 2) \Rightarrow Impair(x)). \tag{5.1}$$

Un tel énoncé n'est pas capturé par la logique propositionnelle. Tout d'abord par ce qu'il utilise des prédicats comme $Premier(x)$ dont la valeur de vérité dépend d'une variable x, ce qui n'est pas possible en logique propositionnelle. Par ailleurs, on utilise ici des quantificateurs comme $\exists$, $\forall$ qui ne sont pas présents non plus en logique propositionnelle.

L'énoncé précédent est un exemple de formule du calcul des prédicats du premier ordre. Dans ce document, on ne parlera que de logique du premier ordre. La terminologie *premier ordre* fait référence au fait que les quantifications existentielles et universelles ne sont autorisées que sur les variables.

Un énoncé *du second ordre*, on parle plus généralement *d'ordre supérieur*, serait un énoncé où l'on autoriserait les quantifications sur les fonctions ou des relations : par exemple, on peut écrire $\neg\exists f(\forall x(f(x) < f(x+1)))$ pour signifier qu'il n'existe pas de suite infiniment décroissante. On ne cherchera pas à comprendre la théorie derrière ce type d'énoncé, car on le verra, les problèmes et difficultés avec le premier ordre sont déjà suffisamment nombreux.

L'objectif de ce chapitre est alors de définir la logique du premier ordre. Comme pour la logique propositionnelle, on va le faire en parlant d'abord de la syntaxe, c'est-à-dire comment on écrit les formules, puis de leur *sémantique*.

Le calcul des prédicats, et en particulier la logique du premier ordre, reste le formalisme le plus courant pour exprimer des propriétés mathématiques. C'est aussi un formalisme très utilisé en informatique pour décrire les objets : par exemple, les langages de requêtes à des bases de données sont essentiellement basés sur ce formalisme, appliqué à des objets finis, qui représentent des données.

5.1 Syntaxe

Pour écrire une formule d'un langage du premier ordre, on utilise certains symboles qui sont communs à tous les langages, et certains symboles qui varient d'un langage à l'autre. Les symboles communs à tous les langages sont :

- les connecteurs $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$;
- les parenthèses (et) et la virgule , ;
- le quantificateur universel $\forall$ et le quantificateur existentiel $\exists$;
- un ensemble infini dénombrable de symboles $\mathcal{V}$ de variables.

Les symboles qui peuvent varier d'un langage à l'autre sont capturés par la notion de *signature*. Une signature fixe les symboles de constantes, les symboles de fonctions et les symboles de relations qui sont autorisés.

Formellement :

Définition 5.1 (Signature d'un langage du premier ordre) *La signature*

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

d'un langage du premier ordre est la donnée :

- *d'un premier ensemble $\mathcal{C}$ de symboles, appelés* symboles de constantes *;*
- *d'un second ensemble $\mathcal{F}$ de symboles, appelés* symboles de fonctions*. A chaque symbole de cet ensemble est associé un entier strictement positif, que l'on appelle* son arité *;*
- *d'un troisième ensemble $\mathcal{R}$ de symboles, appelés* symboles de relations*. A chaque symbole de cet ensemble est associé un entier strictement positif, que l'on appelle* son arité*.*

On suppose que $\mathcal{V}, \mathcal{C}, \mathcal{F}, \mathcal{R}$ sont des ensembles disjoints deux à deux.

Une formule du premier ordre sera alors un mot sur l'alphabet

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

Remarque 5.1 *On utilisera dans ce qui suit les conventions suivantes : On convient que x, y, z, u et v désignent des variables, c'est-à-dire des éléments de $\mathcal{V}$. a, b, c, d désigneront des constantes, c'est-à-dire des éléments de $\mathcal{C}$.*

L'intuition est que les symboles de constantes, fonctions et relations auront vocation ensuite à être interprétés (dans ce que l'on appellera des *structures*) ; l'arité d'un symbole de fonction ou de relation aura pour vocation à correspondre au nombre d'arguments de la fonction ou de la relation.

Exemple 5.1 *Par exemple, on peut considérer la signature*

$$\Sigma = (\{2\}, \{Impair, Premier\}, \{=, <\})$$

qui possède l'unique symbole de constante 2*, les symboles de fonctions Impairs et Premier d'arité* 1*, les symboles de relations $=$ et $<$ d'arité* 2*.*

On va définir par étapes : d'abord les *termes*, qui visent à représenter des objets, puis les *formules atomiques*, qui visent à représenter des relations entre objets, et enfin les formules.

5.1.1 Termes

Nous avons déjà défini les termes dans le chapitre 2 : ce que nous appelons ici *terme sur une signature* Σ, est un terme construit sur l'union de l'ensemble des symboles de fonctions de la signature, des symboles de constantes de la signature, et des variables.

Pour être plus clair, réexprimons notre définition :

Définition 5.2 (Termes sur une signature) *Soit $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{R})$ une signature.*

L'ensemble T des termes *sur la signature Σ est le langage sur l'alphabet $\mathcal{A}(\Sigma)$ défini inductivement par :*

(B) toute variable est un terme : $\mathcal{V} \subset T$;

(B) toute constante est un terme : $\mathcal{C} \subset T$;

(I) si f est un symbole de fonction d'arité n et si $t_1, t_2, \cdots, t_n$ sont des termes, alors $f(t_1, \cdots, t_n)$ est un terme.

Définition 5.3 *Un* terme clos *est un terme sans variable.*

Exemple 5.2 *$Premier(x)$ est un terme sur la signature de l'exemple 5.1 qui n'est pas clos. $Impair(2)$ est un terme clos.*

5.1.2 Formules atomiques

Définition 5.4 (Formules atomiques) *Soit $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{R})$ une signature.*

Une formule atomique sur la signature Σ est un mot de $\mathcal{A}(\Sigma)^*$ de la forme $R(t_1, t_2, \cdots, t_n)$, où $R \in \mathcal{R}$ est un symbole de relation d'arité n, et où $t_1, t_2, \cdots, t_n$ sont des termes sur Σ.

Exemple 5.3 *$> (x, 2)$ est une formule atomique sur la signature de l'exemple 5.1. $= (x, y)$ aussi.*

Remarque 5.2 *On va convenir parfois d'écrire $t_1 R t_2$ pour certains symboles binaires, comme $=$, $<$, $+$ pour éviter des notations trop lourdes : par exemple, on écrira $x > 2$ pour $> (x, 2)$.*

5.1.3 Formules

Définition 5.5 (Formules) *Soit $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{R})$ une signature.*

L'ensemble des formules sur la signature Σ est le langage sur l'alphabet $\mathcal{A}(\Sigma)$ défini inductivement par :

(B) toute formule atomique est une formule ;

(I) si F est une formule, alors $\neg F$ est une formule ;

(I) si F et G sont des formules, alors $(F \wedge G)$, $(F \vee G)$, $(F \Rightarrow G)$, et $(F \Leftrightarrow G)$ sont des formules ;

(I) si F est une formule, et si $x \in \mathcal{V}$ est une variable, alors $\forall x F$ est une formule, et $\exists x F$ aussi.

Exemple 5.4 *L'énoncé* (5.1) *est une formule sur la signature de l'exemple 5.1.*

5.2 Premières propriétés et définitions

5.2.1 Décomposition / Lecture unique

Comme pour les formules du calcul propositionnel, on peut toujours décomposer une formule, et ce de façon unique.

Proposition 5.1 (Décomposition / Lecture unique) *Soit F une formule. Alors F est d'une, et exactement d'une, des formes suivantes :*

1. *une formule atomique ;*
2. *$\neg G$, où G est une formule ;*
3. *$(G \wedge H)$ où G et H sont des formules ;*
4. *$(G \vee H)$ où G et H sont des formules ;*
5. *$(G \Rightarrow H)$ où G et H sont des formules ;*
6. *$(G \Leftrightarrow H)$ où G et H sont des formules ;*
7. *$\forall x G$ où G est une formule et x une variable ;*
8. *$\exists x G$ où G est une formule et x une variable.*

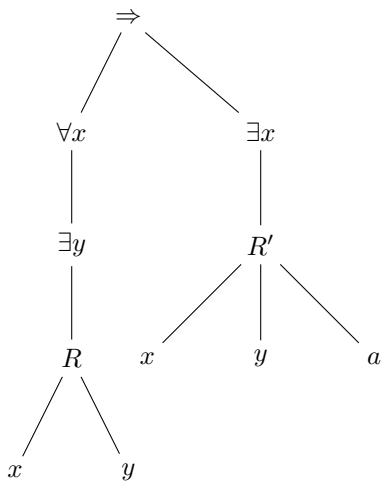
De plus dans le premier cas, il y a une unique façon de "lire" la formule atomique. Dans chacun des autres cas, il y a unicité de la formule G et de la formule H avec cette propriété.

On peut alors naturellement représenter chaque formule par un arbre (son arbre de décomposition, qui est en fait en correspondance avec son arbre de dérivation au sens du chapitre 2) : chaque sommet est étiqueté par un symbole de constante, de fonction, de relation, ou par les symboles $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ ou un quantificateur existentiel ou universel.

Exemple 5.5 *Par exemple, la formule*

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

se représente par l'arbre suivant



Chaque sous-arbre d'un tel arbre représente une *sous-formule* de F. Si l'on préfère :

Définition 5.6 (Sous-formule) *Une formule G est une sous-formule d'une formule F si elle apparaît dans la décomposition de F.*

5.2.2 Variables libres, variables liées

L'intuition de ce qui va suivre est de distinguer les variables *liées* des variables qui ne le sont pas : tout cela est en fait à propos des "$\forall x$" et "$\exists x$" qui sont des *lieurs* : lorsqu'on écrit $\forall xF$ ou $\exists xF$, x devient une variable liée dans F. En d'autres termes, x est une variable muette dans le sens où la valeur de vérité de $\forall xF$ ou $\exists xF$ aura vocation, lorsqu'on parlera de la sémantique des formules, à ne pas dépendre de x : on pourrait tout aussi bien écrire $\forall yF(y/x)$ (respectivement : $\exists yF(y/x)$) où $F(y/x)$ désigne intuitivement la formule que l'on obtient en remplaçant x par y dans F.

Remarque 5.3 *On a le même phénomène dans des symboles comme le symbole intégrale en mathématique : dans l'expression $\int_a^b f(t)dt$, la variable t est une variable muette (liée). En particulier $\int_a^b f(u)du$ est exactement la même intégrale.*

Faisons cela toutefois très proprement. Une même variable peut apparaître plusieurs fois dans une formule : nous avons besoin de savoir repérer chaque occurrence, en faisant attention aux $\exists$ et $\forall$.

Définition 5.7 (Occurence) *Une* occurrence *d'une variable x dans une formule F est un entier n tel que le $n$ème symbole du mot F est x et tel que le $(n-1)$ème symbole ne soit pas $\forall$ ni $\exists$.*

Exemple 5.6 8 *et* 17 *sont des occurrences de* x *dans la formule* (5.2). 7 *et* 14 *n'en sont pas :* 7 *parce que le* 7*ème symbole de* F *n'est pas un* x *(c'est une parenthèse ouvrante) et* 14 *parce que le* 14*ème symbole de* F *qui est bien un* x *est quantifié par* x.

Définition 5.8 (Variable libre, variable liée)
- *Une occurrence d'une variable* x *dans une formule* F *est une* occurrence liée *si cette occurrence apparaît dans une sous-formule de* F *qui commence par un quantificateur* $\forall x$ *ou* $\exists x$. *Sinon, on dit que l'occurrence est* libre.
- *Une variable est* libre *dans une formule si elle possède au moins une occurrence libre dans la formule.*
- *Une formule* F *est close si elle ne possède pas de variables libres.*

Exemple 5.7 *Dans la formule* (5.2)*, les occurrences* 8, 17 *et* 10 *de* x *sont liées. L'occurrence* 19 *de* y *est libre.*

Exemple 5.8 *Dans la formule* $(R(x,z) \Rightarrow \forall z(R(y,z) \vee y = z))$*, la seule occurrence de* x *est libre, les deux occurrences de* y *sont libres. La première (plus petite) occurrence de* z *est libre, et les autres sont liées. La formule* $\forall x \forall z(R(x,z) \Rightarrow \exists y(R(y,z) \vee y = z))$ *est close.*

La notation $F(x_1, \cdots, x_k)$ signifie que les variables libres de F sont parmi $x_1, \cdots, x_k$.

Exercice 5.1 *Montrer que les formules libres* $\ell(F)$ *d'une formule* F *s'obtiennent par la définition inductive suivante :*
- $\ell(R(t_1, \cdots, t_n)) = \{x_i | x_i \in \mathcal{V}$ *et* x_i *apparaît dans* $R(t_1, \cdots, t_n)\}$;
- $\ell(\neg G) = \ell(G)$;
- $\ell(G \vee H) = l(G \wedge H) = \ell(G \Rightarrow H) = \ell(G \Leftrightarrow H) = \ell(G) \cup \ell(H)$;
- $\ell(\forall x F) = \ell(\exists x F) = \ell(F) \backslash \{x\}$.

5.3 Sémantique

Nous pouvons maintenant parler du sens que l'on donne aux formules. En fait, pour donner un sens aux formules, il faut fixer un sens aux symboles de la signature, et c'est l'objet de la notion de structure.

Définition 5.9 (Structure) *Soit* $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{R})$ *une signature.*
Une structure $\mathfrak{M}$ de signature Σ *est la donnée :*
- *d'un ensemble non-vide* M*, appelé* ensemble de base*, ou* domaine *de la structure ;*
- *d'un élément, noté* $c^{\mathfrak{M}}$*, pour chaque symbole de constante* $c \in \mathcal{C}$;
- *d'une fonction, notée* $f^{\mathfrak{M}}$*, de* $M^n \to M$ *pour chaque symbole de fonction* $f \in \mathcal{F}$ *d'arité* n ;
- *d'un sous-ensemble, noté* $R^{\mathfrak{M}}$*, de* M^n *pour chaque symbole de relation* $R \in \mathcal{F}$ *d'arité* n.

On dit que la constante c (respectivement la fonction f, la relation R) est *interprétée par* $c^{\mathfrak{M}}$ (resp. $f^{\mathfrak{M}}$, $R^{\mathfrak{M}}$). Une structure est parfois aussi appelée une *réalisation*.

Exemple 5.9 *Une réalisation de la signature* $\Sigma = (\{\mathbf{0}, \mathbf{1}\}, \{+, -\}, \{=\})$ *correspond à l'ensemble de base* $\mathbb{N}$ *des entiers naturels, avec* $\mathbf{0}$ *interprété par l'entier* 0, $\mathbf{1}$ *par* 1, $+$ *par l'addition,* $-$ *par la soustraction, et* $=$ *par l'égalité sur les entiers : c'est-à-dire par le sous-ensemble* $\{(x, x) | x \in \mathbb{N}\}$.

Exemple 5.10 *Une autre réalisation de cette signature correspond à l'ensemble de base* $\mathbb{R}$ *des réels, où* $\mathbf{0}$ *est interprété par le réel* 0, $\mathbf{1}$ *est interprété par le réel* 1, $+$ *par l'addition,* $-$ *la soustraction, et* $=$ *par l'égalité sur les réels.*

On va ensuite utiliser la notion de structure pour interpréter les termes, les formules atomiques, puis inductivement les formules, comme on peut s'y attendre.

5.3.1 Interprétation des termes

Définition 5.10 (Interprétation des termes) *Soit* $\mathfrak{M}$ *une structure de signature* $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{R})$.

Soit $t = t(x_1, \cdots, x_k)$ *un terme sur* Σ *de variables libres* $x_1, \cdots, x_k$.

Soit $s = (a_1, \cdots, a_k)$ *une suite de* k *éléments de l'ensemble de base* M.

*L'*interprétation $t^{\mathfrak{M}}$ du terme t pour la suite s, *aussi notée* $t^{\mathfrak{M}}[a_1, \cdots, a_k]$ *ou* $t^{\mathfrak{M}}[s]$, *est définie inductivement de la façon suivante :*

(B) *toute variable est interprétée par sa valeur dans la suite : si* t *est la variable* $x_i \in \mathcal{V}$, *alors* $t^{\mathfrak{M}}$ *est* a_i ;

(B) *toute constante est interprétée par son interprétation dans la structure : si* t *est la constante* $c \in \mathcal{C}$, *alors* $t^{\mathfrak{M}}$ *est* $c^{\mathfrak{M}}$;

(I) *chaque symbole de fonction est interprété par son interprétation dans la structure : si* t *est le terme* $f(t_1, \cdots, t_n)$, *alors* $t^{\mathfrak{M}}$ *est* $f^{\mathfrak{M}}(t_1^{\mathfrak{M}}, \cdots, t_n^{\mathfrak{M}})$, *où* $t_1^{\mathfrak{M}}, \cdots, t_n^{\mathfrak{M}}$ *sont les interprétations respectives des termes* $t_1, \cdots, t_n$.

Remarque 5.4 *L'interprétation d'un terme est un élément de* M, *où* M *est l'ensemble de base de la structure* $\mathfrak{M}$ *: les termes désignent donc des éléments de la structure.*

5.3.2 Interprétations des formules atomiques

Une formule atomique $F = F(x_1, \cdots, x_k)$ est un objet qui s'interprète soit par *vrai* soit par *faux* en une suite $s = (a_1, \cdots, a_k)$. Lorsque F s'interprète par vrai, on dit que *la suite* s *satisfait* F, et on note ce fait $s \models F$. On note $s \not\models F$ dans le cas contraire.

Il nous reste plus qu'à définir formellement cette notion :

Définition 5.11 (Interpétation d'une formule atomique) *Soit* $\mathfrak{M}$ *une structure de signature* $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{R})$.

La suite $s = (a_1, \cdots, a_k)$ d'éléments de M satisfait la formule atomique $R(t_1, t_2, \cdots, t_n)$ de variables libres $x_1, \cdots, x_k$ si $(t_1^{\mathfrak{M}}[s], t_2^{\mathfrak{M}}[s], \cdots, t_n^{\mathfrak{M}}[s]) \in R^{\mathfrak{M}}$, où $R^{\mathfrak{M}}$ est l'interprétation du symbole R dans la structure.

Exemple 5.11 *Par exemple, sur la structure de l'exemple 5.9, $x > 2$ s'interprète par 1 (vrai) en la suite $s = (5)$ (pour $x = 5$), et par 0 (faux) en la suite $s = (0)$ (pour $x = 0$). La formule atomique $\mathbf{0} = \mathbf{1}$ s'interprète par 0 (faux).*

5.3.3 Interprétation des formules

Plus généralement, une formule $F = F(x_1, \cdots, x_k)$ est un objet qui s'interprète soit par *vrai* soit par *faux* en une suite $s = (a_1, \cdots, a_k)$. Lorsque F s'interprète par vrai, on dit toujours que *la suite s satisfait F*, et on note toujours ce fait $s \models F$, et $s \not\models F$ pour le cas contraire.

Définition 5.12 (Inteprétation d'une formule) *Soit $\mathfrak{M}$ une structure de signature $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{R})$.*

L'expression "la suite $s = (a_1, \cdots, a_k)$ d'éléments de M satisfait la formule atomique $F = F(x_1, \cdots, x_k)$", notée $s \models F$, se définit inductivement de la façon suivante :

(B) elle a déjà été définie pour une formule atomique ;

$\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow$ sont interprétés exactement comme dans le calcul propositionnel :

(I) la négation s'interprète par la négation logique :
si F est de la forme $\neg G$, alors $s \models F$ ssi $s \not\models G$;

(I) $\wedge$ s'interprète comme une conjonction logique :
si F est de la forme $(G \wedge H)$, alors $s \models F$ ssi $s \models G$ et $s \models H$;

(I) $\vee$ s'interprète comme le ou logique :
si F est de la forme $(G \vee H)$, alors $s \models F$ ssi $s \models G$ ou $s \models H$;

(I) $\Rightarrow$ s'interprète comme l'implication logique :
si F est de la forme $(G \Rightarrow H)$, alors $s \models F$ ssi $s \models H$ ou $s \not\models G$;

(I) $\Leftrightarrow$ s'interprète comme l'équivalence logique :
si F est de la forme $(G \Leftrightarrow H)$, alors $s \models F$ ssi ($s \models G$ et $s \models H$) ou ($s \not\models G$ et $s \not\models H$).

$\exists x$ et $\forall x$ sont interprétés comme des quantifications existentielles et universelles :

(I) si F est de la forme $\forall x_0 G[x_0, x_1, \cdots, x_k]$, alors $s \models F$ ssi pour tout élément $a_0 \in M$ la suite $s' = (a_0, a_1, \cdots, a_k)$ satisfait G ;

(I) si F est de la forme $\exists x_0 G[x_0, x_1, \cdots, x_k]$, alors $s \models F$ ssi pour un certain élément $a_0 \in M$ la suite $s' = (a_0, a_1, \cdots, a_k)$ satisfait G.

Dans le cas où la suite s satisfait la formule F, on dit aussi que F est *vraie* en s. Dans le cas contraire, on dit que F est *fausse* en s.

Pour une formule F close, la satisfaction de F dans une structure $\mathfrak{M}$ ne dépend pas de la suite s. Dans le cas où la formule F est vraie, on dit que la structure $\mathfrak{M}$ est un modèle de F, ce que l'on note $\mathfrak{M} \models F$.

5.4 Équivalence. Formes normales

5.4.1 Formules équivalentes

Définition 5.13 *Soit $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{R})$ une signature.*

- *Une structure $\mathfrak{M}$ satisfait la formule $F(x_1, \cdots, x_k)$ si elle satisfait la formule close $\forall x_1 \cdots \forall x_k F(x_1, \cdots, x_k)$. Cette dernière formule est appelée* la clôture universelle de F.
- *Une formule close F est dite* valide *si elle est satisfaite par toute structure $\mathfrak{M}$.*
- *Une formule F est dite* valide *si sa clôture universelle est valide.*
- *Deux formules F et G sont équivalentes si pour toute structure, et pour toute suite s d'éléments interprétant les variables de F et G elle prennent la même valeur de vérité. On note $F \equiv G$ dans ce cas.*

Exercice 5.2 *Montrer que la relation $\equiv$ est une relation d'équivalence.*

Proposition 5.2 *Soit F une formule. On a les équivalences suivantes :*

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

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

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

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

Proposition 5.3 *Supposons que la variable x n'est pas libre dans la formule G. Soit F une formule. On a alors les équivalences suivantes :*

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

Chacune des équivalences étant en fait assez simple à établir, mais fastidieuse, nous laissons les preuves en exercice.

Exercice 5.3 *Prouver la proposition 5.3.*

5.4.2 Forme normale prénexe

Définition 5.14 (Forme prénexe) *Une formule F est dite* prénexe *si elle est de la forme*

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

où chacun des Q_i est soit un quantificateur $\forall$, soit un quantificateur $\exists$, et F' est une formule qui ne contient aucun quantificateur.

Proposition 5.4 *Toute formule F est équivalente à une formule prénexe G.*

Démonstration: Par induction structurelle sur F.

Cas de base. Si F est de la forme $R(t_1, \cdots, t_n)$, pour un symbole de relation R, alors F est sous forme prénexe.

Cas inductif :

- si F est de la forme $\forall x G$ ou $\exists x G$, par hypothèse d'induction G est équivalente à G' prénexe et donc F est équivalent à $\forall x G'$ ou $\exists x G'$ qui est prénexe.
- si F est de la forme $\neg G$, par hypothèse d'induction G est équivalente à G' prénexe de la forme $Q_1 x_1 Q_2 x_2 \cdots Q_n x_n G''$. En utilisant les équivalences de la proposition 5.2, F est équivalente à $Q'_1 x_1 Q'_2 x_2 \cdots Q'_n x_n \neg G''$, en prenant $Q'_i = \forall$ si $Q_i = \exists$ et $Q'_i = \exists$ si $Q_i = \forall$.
- Si F est de la forme $(G \wedge H)$ par hypothèse d'induction G et H sont équivalentes à des formules G' et H' en forme prénexe. En appliquant les équivalences (5.4) à (5.11), on peut faire "remonter" les quantificateurs en tête de formule : on doit toutefois procéder avec soin, car si par exemple $F = (F_1 \wedge F_2) = ((\forall x F'_1) \wedge F'_2)$ avec x libre dans F'_2, nous devons d'abord renommer la variable x dans F_1 en remplaçant x par une nouvelle variable z n'apparaissant ni dans F_1 ni dans F'_2, de façon à bien pouvoir utiliser l'équivalence.
- Les autres cas se traitent selon le même principe, en utilisant les équations des deux propositions précédentes.

□

En utilisant les équivalences similaires à celles écrites dans le calcul propositionnel, on peut même aller plus loin :

Proposition 5.5 *Toute formule F est équivalente à une formule prénexe G, où F' est en forme normale conjonctive.*

Proposition 5.6 *Toute formule F est équivalente à une formule prénexe G, où F' est en forme normale disjonctive.*

5.5 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture de [Cori and Lascar, 1993a], [Dowek, 2008] et [Lassaigne and de Rougemont, 2004].

Bibliographie Ce chapitre a été rédigé en s'inspirant essentiellement des ouvrages [Cori and Lascar, 1993a] et [Lassaigne and de Rougemont, 2004].

Chapitre 6

Modèles. Complétude.

Nous pouvons maintenant décrire différents objets, et parler de leurs propriétés. Nous avons en effet tous les ingrédients pour parler de modèles et de théories. Nous nous intéresserons ensuite au théorème de complétude.

Le concept de base est celui de théorie.

Définition 6.1 (Théorie)
- *Une* théorie $\mathcal{T}$ *est un ensemble de formules closes sur une signature donnée. Les formules d'une théorie sont appelées des* axiomes *de cette théorie.*
- *Une structure* $\mathfrak{M}$ *est un* modèle de la théorie $\mathcal{T}$ *si* $\mathfrak{M}$ *est un modèle de chacune des formules de la théorie.*

Définition 6.2 (Théorie consistante) *Une théorie est dite* consistante *si elle possède un modèle. Elle est dite* inconsistante *dans le cas contraire.*

Bien entendu, les théories inconsistantes sont de peu d'intérêt.

Remarque 6.1 *D'un point de vue informatique, on peut voir une théorie comme la spécification d'un objet : on décrit l'objet à l'aide de la logique du premier ordre, i.e. à l'aide des axiomes qui le décrivent.*

Une spécification (théorie) consistante est donc ni plus ni moins qu'une théorie qui spécifie au moins un objet.

Remarque 6.2 *Dans ce contexte, la question de la complétude est de savoir si l'on décrit bien l'objet en question, ou la classe des objets en question : le théorème de complétude permet de dire que oui pour une spécification consistante, tant que l'on s'intéresse à la classe de tous les modèles de ces spécifications.*

Nous allons commencer par donner différents exemples de théories, pour rendre notre discussion beaucoup moins abstraite.

6.1 Exemples de théories

6.1.1 Graphe

Un graphe orienté peut se voir comme un modèle de la théorie sans axiome sur la signature $\Sigma = (\emptyset, \emptyset, \{E\})$, où le symbole de relation E est d'arité 2 : $E(x, y)$ signifie qu'il y a un arc entre x et y.

Un graphe non-orienté peut se voir comme un modèle de la théorie avec l'unique axiome

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

sur la même signature. Cet axiome signifie que s'il y a un arc entre x et y, alors il y en a aussi un de y vers x et réciproquement.

6.1.2 Égalité

Soit $\mathcal{R}$ un ensemble de symboles de relations contenant au moins le symbole d'égalité $=$.

Les axiomes de l'égalité pour une signature $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{R})$ sont

– les 3 axiomes :

$$\forall x \ x = x, \tag{6.2}$$

$$\forall x \forall y \ (x = y \Rightarrow y = x), \tag{6.3}$$

$$\forall x \forall y \forall z \ ((x = y \wedge y = z) \Rightarrow x = z), \tag{6.4}$$

exprimant que $=$ est une relation d'équivalence,

– ainsi que les n axiomes

$$\forall x_1 \cdots \forall x_i \forall x'_i \cdots \forall x_n (x_i = x'_i \Rightarrow f(x_1, \cdots, x_i, \cdots, x_n) = f(x_1, \cdots, x'_i, \cdots, x_n)) \tag{6.5}$$

pour chaque symbole de fonction $f \in \mathcal{F}$ d'arité n,

– ainsi que les n axiomes

$$\forall x_1 \cdots \forall x_i \forall x'_i \cdots \forall x_n (x_i = x'_i \Rightarrow (R(x_1, \cdots, x_i, \cdots, x_n) \Leftrightarrow R(x_1, \cdots, x'_i, \cdots, x_n))) \tag{6.6}$$

pour chaque symbole de relation $R \in \mathcal{R}$ d'arité n.

Tous ces axiomes spécifient que l'égalité est préservée par composition par les symboles de fonction et de relation.

6.1.3 Petite parenthèse

Définition 6.3 *Un modèle $\mathfrak{M}$ d'une théorie $\mathcal{T}$ sur une signature avec le symbole de relation $=$ est dit* égalitaire*, si l'interprétation de $=$ dans $\mathfrak{M}$ est l'égalité.*

En d'autres termes, l'interprétation du symbole $=$ dans $\mathfrak{M}$ est le sous-ensemble $\{(x, x) | x \in M\}$ où M est l'ensemble de base de $\mathfrak{M}$.

Il se trouve que si ce n'est pas le cas, et si on a les axiomes de l'égalité, alors on peut s'y ramener.

Proposition 6.1 *Soit $\mathcal{T}$ une théorie sur une signature Σ, avec au moins le symbole $=$ comme symbole de relation, qui contient tous les axiomes de l'égalité pour Σ.*

Si $\mathcal{T}$ possède un modèle, alors $\mathcal{T}$ possède un modèle égalitaire.

Démonstration: On peut quotienter le domaine M de tout modèle $\mathfrak{M}$ de $\mathcal{T}$ par la relation d'équivalence qui place dans la même classe d'équivalence x et y lorsque l'interprétation de $x = y$ est vraie dans $\mathfrak{M}$. Le modèle quotient, c'est-à-dire celui dont les éléments sont les classes d'équivalence de cette relation d'équivalence, est par définition égalitaire. □

Du coup, une théorie $\mathcal{T}$ possède un modèle égalitaire si et seulement si tous les axiomes de l'égalité (pour la signature correspondante) possèdent un modèle.

6.1.4 Groupes

Commençons par parler des groupes, en théorie des groupes.

Exemple 6.1 (Groupe) *Un groupe est un modèle de la théorie constituée des deux formules :*

$$\forall x \forall y \forall z \ x * (y * z) = (x * y) * z \tag{6.7}$$

$$\exists e \forall x \ (x * e = e * x = x \wedge \exists y (x * y = y * x = e)) \tag{6.8}$$

sur la signature $\Sigma = (\emptyset, \{*\}, \{=\})$, où $*$ et $=$ sont d'arité 2.

Remarque 6.3 *C'est en fait une définition d'un groupe, si on ajoute les axiomes de l'égalité. Ou, si l'on préfère, par ce qui précède, si on remplace ci-dessus "modèle" par "modèle égalitaire", sans rajouter aucun axiome, cela reste une définition, quitte à raisonner à isomorphisme près.*

La première propriété exprime le fait que la loi du groupe $*$ est associative, et la seconde qu'il existe un élément neutre, e, tel que tout élément possède un inverse.

Exemple 6.2 (Groupe commutatif) *Un groupe commutatif est un modèle de la théorie constituée des trois formules :*

$$\forall x \forall y \forall z \ x * (y * z) = (x * y) * z \tag{6.9}$$

$$\exists e \forall x \ (x * e = e * x = x \wedge \exists y (x * y = y * x = e)) \tag{6.10}$$

$$\forall x \forall y \ x * y = y * x \tag{6.11}$$

sur la même signature.

Remarque 6.4 *Cela correspond à une définition d'un groupe commutatif, si on ajoute les axiomes de l'égalité*[1].

[1] Même commentaire que pour l'exemple précédent si l'on ne veut pas ajouter d'axiome.

6.1.5 Corps

Exemple 6.3 (Corps commutatif) *Un* corps commutatif *est un modèle de la théorie constituée des formules*

$$\forall x \forall y \forall z \ (x + (y + z) = (x + y) + z) \tag{6.12}$$
$$\forall x \forall y (x + y = y + x) \tag{6.13}$$
$$\forall x (x + \mathbf{0} = x) \tag{6.14}$$
$$\forall x \exists y (x + y = \mathbf{0}) \tag{6.15}$$
$$\forall x \forall y \forall z \ x * (y + z) = x * y + x * z \tag{6.16}$$
$$\forall x \forall y \forall z \ ((x * y) * z) = (x * (y * z)) \tag{6.17}$$
$$\forall x \forall y \ (x * y = y * x) \tag{6.18}$$
$$\forall x \ (x * \mathbf{1} = x) \tag{6.19}$$
$$\forall x \forall y (x = \mathbf{0} \vee x * y = \mathbf{1}) \tag{6.20}$$
$$\neg \mathbf{1} = \mathbf{0} \tag{6.21}$$

sur une signature avec deux symboles de constantes $\mathbf{0}$ *et* $\mathbf{1}$*, deux symboles de fonctions* $+$ *et* $*$ *d'arité* 2*, et le symbole de relation* $=$ *d'arité 2.*

Remarque 6.5 *C'est même en fait une définition d'un corps commutatif, si on ajoute les axiomes de l'égalité*[2]*.*

Par exemple $\mathbb{R}$ ou $\mathbb{C}$ avec l'interprétation standard sont des modèles de ces théories.

Si l'on ajoute à la théorie la formule F_p définie par $\mathbf{1} + \cdots + \mathbf{1} = \mathbf{0}$, où $\mathbf{1}$ est répété p fois, les modèles sont les corps de caractéristique p : par exemple $\mathbb{Z}_p$, lorsque p est premier.

Si l'on veut décrire un corps de caractéristique nulle, il faut considérer la théorie constituée des axiomes précédents et de l'union des axiomes F_p, pour p un nombre premier.

Exemple 6.4 (Corps algébriquement clos) *Pour chaque entier* n*, on considère la formule* G_n

$$\forall x_0 \forall x_1 \cdots \forall x_{n-1} \exists x (x_0 + x_1 * x + x_2 * x^2 + \cdots + x_{n-1} * x^{n-1} + x^n)$$

où le lecteur aura deviné que x^k *est* $x * \cdots * x$ *avec* x *répété* k *fois.*

Un corps commutatif algébriquement clos est un modèle de la théorie constituée des axiomes des corps commutatifs et de l'union des formules G_n *pour* $n \in \mathbb{N}$*.*

Remarque 6.6 *C'est même en fait une définition d'un corps commutatif algébriquement clos, si on ajoute les axiomes de l'égalité*[3]*.*

Par exemple, $\mathbb{C}$ est algébriquement clos. $\mathbb{R}$ n'est pas algébriquement clos, car $x^2 + 1$ ne possède pas de racine réelle.

[2] Même commentaire que pour les exemples précédents si l'on ne veut pas ajouter d'axiome.
[3] Même commentaire que pour les exemples précédents si l'on ne veut pas ajouter d'axiome.

6.1.6 Arithmétique de Robinson

On peut aussi chercher à axiomatiser les entiers. Voici une première tentative.

Exemple 6.5 (Arithmétique de Robinson) *Considérons la signature constituée du symbole de constante* $\mathbf{0}$*, d'une fonction unaire* s*, et de deux fonctions binaires* $+$ *et* $*$*, et des relations binaires* $<$ *et* $=$*.*

Les axiomes de l'arithmétique de Robinson sont

$$\forall x\ \neg s(x) = \mathbf{0} \tag{6.22}$$
$$\forall x \forall y\ (s(x) = s(y) \Rightarrow x = y) \tag{6.23}$$
$$\forall x\ \neg x < \mathbf{0} \tag{6.24}$$
$$\forall x \forall y\ (x < s(y) \Rightarrow (x = y \vee x < y)) \tag{6.25}$$
$$\forall x \forall y\ (x < y \vee x = y \vee y < x) \tag{6.26}$$
$$\forall x\ x + \mathbf{0} = x \tag{6.27}$$
$$\forall x \forall y\ x + s(y) = s(x + y) \tag{6.28}$$
$$\forall x\ x * \mathbf{0} = \mathbf{0} \tag{6.29}$$
$$\forall x \forall y\ (x * s(y)) = (x * y) + x \tag{6.30}$$

La structure dont l'ensemble de base est les entiers, et où l'on interprète $+$ par l'addition, $*$ par la multiplication, et $s(x)$ par $x + 1$ est un modèle de cette théorie. On appelle ce modèle *le modèle standard des entiers*.

6.1.7 Arithmétique de Peano

En voici une seconde, constituée en fait d'une famille d'axiomes.

Exemple 6.6 (Arithmétique de Peano) *Considérons une signature constituée du symbole de constante* $\mathbf{0}$*, d'une fonction unaire* s*, et de deux fonctions binaires* $+$ *et* $*$*, et de la relation binaire* $=$*.*

Les axiomes de l'arithmétique de Peano sont les axiomes

$$\forall x\ \neg s(x) = \mathbf{0} \tag{6.31}$$
$$\forall x \exists y(\neg x = \mathbf{0} \Rightarrow s(y) = x) \tag{6.32}$$
$$\forall x \forall y\ (s(x) = s(y) \Rightarrow x = y) \tag{6.33}$$
$$\forall x\ x + \mathbf{0} = x \tag{6.34}$$
$$\forall x \forall y\ x + s(y) = s(x + y) \tag{6.35}$$
$$\forall x\ x * \mathbf{0} = \mathbf{0} \tag{6.36}$$
$$\forall x \forall y\ (x * s(y)) = (x * y) + x \tag{6.37}$$

et l'ensemble de toutes les formules de la forme

$$\forall x_1 \cdots \forall x_n((F(\mathbf{0}, x_1, \cdots, x_n) \wedge \forall x_0(F(x_0, x_1, \cdots, x_n) \Rightarrow F(s(x_0), x_1, \cdots, x_n)))$$

$$\Rightarrow \forall x_0 F(x_0, x_1, \cdots, x_n)) \tag{6.38}$$

où n est n'importe quel entier et $F(x_0, \cdots, x_n)$ est n'importe quelle formule de variables libres $x_0, \cdots, x_n$

Il y a donc en fait une infinité d'axiomes. Les derniers axiomes visent à capturer le raisonnement par récurrence que l'on fait régulièrement sur les entiers.

Là encore, le modèle standard des entiers est un modèle de ces axiomes.

6.2 Complétude

Le *théorème de complétude*, dû à Kurt Gödel, et parfois appelé *premier théorème de Gödel*, relie la notion de conséquence à la notion de prouvabilité, en montrant que ces deux notions coïncident.

6.2.1 Conséquence

La notion de conséquence est facile à définir.

Définition 6.4 (Conséquence) *Soit F une formule. La formule F est dite* une conséquence (sémantique) *de la théorie $\mathcal{T}$ si tout modèle de la théorie $\mathcal{T}$ est un modèle de F. On note dans ce cas $\mathcal{T} \models F$.*

Exemple 6.7 *Par exemple, la formule $\forall x \forall y \; x * y = y * x$, qui exprime la commutativité, n'est pas une conséquence de la théorie des groupes (définition 6.1), car il y a des groupes qui ne sont pas commutatifs.*

Exemple 6.8 *On peut montrer que la formule $\forall x \; \mathbf{0}+x = x$ est une conséquence des axiomes de Peano.*

6.2.2 Démonstration

Il faut aussi fixer la notion de démonstration, ce que nous allons faire, mais disons dans un premier temps que nous avons une notion de démonstration telle que l'on note $\mathcal{T} \vdash F$ si l'on peut prouver la formule close F à partir des axiomes de la théorie $\mathcal{T}$.

On espère au minimum de cette notion de preuve d'être valide : c'est-à-dire de dériver uniquement des conséquences : si F est une formule close, et si $\mathcal{T} \vdash F$, alors F est une conséquence de $\mathcal{T}$.

6.2.3 Énoncé du théorème de complétude

Le théorème de complétude dit en fait qu'on arrive à atteindre toutes les conséquences : les relations $\models$ et $\vdash$ sont les mêmes.

Théorème 6.1 (Théorème de complétude) *Soit $\mathcal{T}$ une théorie. Soit F une formule close. F est une conséquence de $\mathcal{T}$ si et seulement si F se prouve à partir de $\mathcal{T}$.*

6.2.4 Signification de ce théorème

Arrêtons-nous sur ce que cela signifie : autrement dit, **les énoncés prouvables sont exactement ceux qui sont vrais dans tous les modèles de la théorie.**

Cela signifie en particulier que

- si une formule close F n'est pas prouvable alors c'est qu'il existe un modèle qui n'est pas un modèle de F.
- si une formule close F est vraie dans tous les modèles des axiomes de la théorie, alors F est prouvable.

Exemple 6.9 *Par exemple, la formule $\forall x \forall y\ x * y = y * x$, qui exprime la commutativité, n'est pas prouvable à partir des axiomes de la théorie des groupes.*

Exemple 6.10 *La formule $\forall x\ \mathbf{0} + x = x$ est prouvable à partir des axiomes de Peano.*

6.2.5 Autre formulation du théorème

On dit qu'une théorie $\mathcal{T}$ est *cohérente* s'il n'existe pas de formule F telle que $\mathcal{T} \vdash F$ et $\mathcal{T} \vdash \neg F$.

On verra au détour de la preuve que cela revient aussi à dire :

Théorème 6.2 (Théorème de complétude) *Soit $\mathcal{T}$ une théorie. $\mathcal{T}$ possède un modèle si et seulement si $\mathcal{T}$ est cohérente.*

6.3 Preuve du théorème de complétude

6.3.1 Un système de déduction

Il nous faut définir une notion de démonstration. Nous choisissons de considérer une notion basée sur la notion de preuve à la Frege et Hilbert.

Par rapport au calcul propositionnel, on n'utilise plus seulement la règle de modus ponens, mais aussi une règle de *généralisation* : si F est une formule et x une variable, la règle de généralisation déduit $\forall x F$ de F.

Cette règle peut-être considérée comme troublante, mais c'est ce que l'on fait dans le raisonnement courant régulièrement : si on arrive à prouver $F(x)$ sans hypothèse particulière sur x, alors on saura que $\forall x F(x)$.

On considère alors un certain nombre d'axiomes :

Définition 6.5 (Axiomes logiques du calcul des prédicats) *Les* axiomes logiques du calcul des prédicats *sont :*

1. *toutes les instances des tautologies du calcul propositionnel ;*
2. *les axiomes des quantificateurs, c'est-à-dire*
   (a) *les formules de la forme $(\exists x F \Leftrightarrow \neg \forall x \neg F)$, où F est une formule quelconque et x une variable quelconque ;*

(b) *les formules de la forme* $(\forall x(F \Rightarrow G) \Rightarrow (F \Rightarrow \forall xG))$ *où* F *et* G *sont des formules quelconques et* x *une variable qui n'a pas d'occurrence libre dans* F *;*

(c) *les formules de la forme* $(\forall xF \Rightarrow F(t/x))$ *où* F *est une formule,* t *un terme et aucune occurrence libre de* x *dans* F *ne se trouve dans le champ d'un quantificateur liant une variable de* t*, où* $F(t/x)$ *désigne la substitution de* x *par* t*.*

Exercice 6.1 *Montrer que les axiomes logiques sont valides.*

Remarque 6.7 *On pourrait ne pas mettre toutes les tautologies du calcul propositionnel, et comme pour le calcul propositionnel se limiter à certains axiomes, essentiellement les axiomes de la logique booléenne. Nous le faisons ici pour simplifier les preuves.*

On obtient la notion de démonstration.

Définition 6.6 (Démonstration par modus ponens et généralisation) *Soit* $\mathcal{T}$ *une théorie et* F *une formule. Une* preuve de F à partir de $\mathcal{T}$ *est une suite finie* $F_1, F_2, \cdots, F_n$ *de formules telle que* F_n *est égale à* F*, et pour tout* i*, ou bien* F_i *est dans* $\mathcal{T}$*, ou bien* F_i *est un axiome logique, ou bien* F_i *s'obtient par modus ponens à partir de deux formules* F_j, F_k *avec* $j < i$ *et* $k < i$*, ou bien* F_i *s'obtient à partir d'un formule* F_j *avec* $j < i$ *par généralisation.*

On note $\mathcal{T} \vdash F$ si F est prouvable à partir de $\mathcal{T}$.

6.3.2 Théorème de finitude

On obtient d'abord facilement au passage le théorème de finitude.

Théorème 6.3 (Théorème de finitude) *Pour toute théorie* $\mathcal{T}$*, et pour toute formule* F*, si* $\mathcal{T} \vdash F$*, alors il existe un sous-ensemble fini* $\mathcal{T}_0$ *de* $\mathcal{T}$ *tel que* $\mathcal{T}_0 \vdash F$*.*

Démonstration: Une démonstration est une suite finie de formules F_1, F_2, $\cdots, F_n$. Elle ne fait donc appel qu'à un nombre fini $\mathcal{T}_0$ de formules de $\mathcal{T}$. Cette démonstration est aussi une démonstration de F dans la théorie $\mathcal{T}_0$. □

Corollaire 6.1 *Si* $\mathcal{T}$ *est une théorie dont toutes les parties finies sont cohérentes, alors* $\mathcal{T}$ *est cohérente.*

Démonstration: Sinon $\mathcal{T}$ prouve $(F \wedge \neg F)$, pour une certaine formule F, et par le théorème de finitude on en déduit qu'il existe un sous-ensemble fini $\mathcal{T}_0$ de $\mathcal{T}$ qui prouve aussi $(F \wedge \neg F)$. □

6.3.3 Deux résultats techniques

On aura besoin des deux résultats suivants, dont les démonstrations relèvent d'un jeu de réécriture sur les démonstrations.

Lemme 6.1 (Lemme de déduction) *Supposons que $\mathcal{T} \cup \{F\} \vdash G$, avec F une formule close. Alors $\mathcal{T} \vdash (F \Rightarrow G)$.*

Démonstration: A partir d'une démonstration $G_0 G_1 \cdots G_n$ de G dans $\mathcal{T} \cup \{F\}$ on construit une démonstration de $(F \Rightarrow G)$ dans $\mathcal{T}$ en faisant des insertions dans la suite $(F \Rightarrow G_0)(F \Rightarrow G_1) \cdots (F \Rightarrow G_n)$.

Si G_i est une tautologie, alors il n'y a rien à faire car $(F \Rightarrow G_i)$ en est une aussi.

Si G_i est F, alors il n'y a rien à faire car $(F \Rightarrow G_i)$ est une tautologie.

Si G_i est un axiome des quantificateurs ou encore un élément de $\mathcal{T}$, alors il suffit d'insérer[4] entre $(F \Rightarrow G_{i-1})$ et $(F \Rightarrow G_i)$ les formules G_i et $(G_i \Rightarrow (F \Rightarrow G_i))$ (qui est une tautologie).

Supposons maintenant que G_i soit obtenue par modus ponens : il y a des entiers $j, k < i$ tels que G_k soit $(G_j \Rightarrow G_i)$. On insère alors entre $(F \Rightarrow G_{i-1})$ et $(F \Rightarrow G_i)$ les formules ;

1. $((F \Rightarrow G_j) \Rightarrow ((F \Rightarrow (G_j \Rightarrow G_i)) \Rightarrow (F \Rightarrow G_i)))$ (une tautologie) ;
2. $(F \Rightarrow (G_j \Rightarrow G_i)) \Rightarrow (F \Rightarrow G_i))$ qui s'obtient par modus ponens à partir de la précédente à l'aide de $(F \Rightarrow G_j)$ qui est déjà apparue ;
3. $(F \Rightarrow G_i)$ se déduit alors par modus ponens de cette dernière formule et de $(F \Rightarrow (G_j \Rightarrow G_i))$, qui est déjà apparue puisque c'est $(F \Rightarrow G_k)$.

Supposons enfin que G_i soit obtenue par généralisation à partir de G_j avec $j < i$. On insère dans ce cas entre $(F \Rightarrow G_{i-1})$ et $(F \Rightarrow G_i)$ les formules :

1. $\forall x(F \Rightarrow G_j)$ obtenue par généralisation en partant de $(F \Rightarrow G_j)$;
2. $(\forall x(F \Rightarrow G_j) \Rightarrow (F \Rightarrow \forall x G_j))$ (un axiome des quantificateurs). F étant une formule close, x n'y est pas libre ;
3. $(F \Rightarrow G_i)$ se déduit alors par modus ponens à partir des deux précédentes.

□

Lemme 6.2 *Soit $\mathcal{T}$ une théorie, et $F(x)$ une formule dont la seule variable libre est x . Soit c un symbole de constante qui n'apparaît ni dans F ni dans $\mathcal{T}$. Si $\mathcal{T} \vdash F(c/x)$ alors $\mathcal{T} \vdash \forall x F(x)$.*

Démonstration: Considérons une démonstration $F_1 F_2 \cdots F_n$ de $F(c/x)$ dans $\mathcal{T}$. On considère une variable w qui n'est dans aucune formule F_i et on appelle K_i la formule obtenue en remplaçant dans F_i le symbole c par w.

Il s'avère que cela donne une preuve de $F(w/x)$: si F_i est un axiome logique, K_i aussi ; si F_i se déduit par modus ponens, alors K_i se déduit des mêmes formules, et si $F_i \in \mathcal{T}$ alors K_i est F_i.

Par généralisation, on obtient donc une preuve de $\forall w F(w/x)$, et par la remarque qui suit, on peut alors obtenir une preuve de $\forall x F(x)$. □

[4] Pour $i = 0$, il suffit de placer ces formules au début.

Remarque 6.8 *Si w est une variable qui n'a aucune occurence dans F (ni libre ni liée) alors on peut prouver $\forall w F(w/x) \Rightarrow \forall x F$: en effet, puisque w n'a pas d'occurence dans F, on peut donc prouver $\forall w F(w/x) \Rightarrow F$, (axiome (c) des quantificateurs, en observant que $(F(w/x))(x/w) = F$ avec ces hypothèses). Par généralisation, on obtient $\forall x(\forall w F(w/x) \Rightarrow F)$, et puisque x n'est pas libre dans $\forall w F(w/x)$, la formule $\forall x(\forall w F(w/x) \Rightarrow F) \Rightarrow (\forall w F(w/x) \Rightarrow \forall x F)$ fait partie des axiomes (b) des quantificateurs, ce qui permet d'obtenir $\forall w F(w/x) \Rightarrow \forall x F$ par modus ponens.*

6.3.4 Validité du système de déduction

La validité de la méthode de preuve utilisée est facile à établir.

Théorème 6.4 (Validité) *Soit $\mathcal{T}$ une théorie. Soit F une formule.*

Si $\mathcal{T} \vdash F$, alors tout modèle de $\mathcal{T}$ est un modèle de la clôture universelle de F.

Démonstration: Il suffit de se convaincre que les axiomes logiques sont valides, et que le modus ponens et la généralisation ne font qu'inférer des faits qui sont valides dans tout modèle de $\mathcal{T}$. □

C'était le sens facile du théorème de complétude.

6.3.5 Complétude du système de déduction

L'autre sens consiste à montrer que si F est une conséquence de $\mathcal{T}$ alors F peut se prouver par notre méthode de preuve.

Définition 6.7

On dit qu'une théorie $\mathcal{T}$ est complète *si pour toute formule close F on a $\mathcal{T} \vdash F$ ou $\mathcal{T} \vdash \neg F$.*

On dit qu'une théorie $\mathcal{T}$ admet des témoins de Henkin *si pour toute formule $F(x)$ avec une variable libre x, il existe un symbole de constante c dans la signature tel que $(\exists x F(x) \Rightarrow F(c))$ soit une formule de la théorie $\mathcal{T}$.*

Le preuve du théorème de complétude due à *Henkin* que nous présentons ici fonctionne en deux étapes.

1. On montre qu'une théorie cohérente, complète, et avec des témoins de Henkin admet un modèle.
2. On montre que toute théorie consistante admet une extension avec ces trois propriétés.

Lemme 6.3 *Si $\mathcal{T}$ est une théorie cohérente, complète, et avec des témoins de Henkin, alors $\mathcal{T}$ possède un modèle.*

Démonstration: L'astuce est de construire de toutes pièces un modèle, dont l'ensemble de base (le domaine) est l'ensemble M des termes clos sur la signature de la théorie : ce domaine n'est pas vide, car la signature a au moins les constantes.

La structure $\mathfrak{M}$ est définie de la façon suivante :

1. si c est une constante, l'interprétation $c^{\mathfrak{M}}$ de c est la constante c elle-même.
2. si f est un symbole de fonction d'arité n, son interprétation $f^{\mathfrak{M}}$ est la fonction qui aux termes clos $t_1, \cdots, t_n$ associe le terme clos $f(t_1, \cdots, t_n)$.
3. si R est un symbole de relation d'arité n, son interprétation $R^{\mathfrak{M}}$ est le sous-ensemble de M^n constitué des $(t_1, \cdots, t_n)$ tels que $\mathcal{T} \vdash R(t_1, \cdots, t_n)$.

On observe que la structure obtenue vérifie la propriété suivante : pour toute formule close F, $\mathcal{T} \vdash F$ si et seulement si $\mathfrak{M}$ est un modèle de F. Cela se prouve par induction structurelle sur F.

La propriété est vraie pour les formules atomiques.

En raison des propriétés des quantificateurs et connecteurs, et de la possibilité d'utiliser des occurrences des tautologies du calcul propositionnel dans notre méthode de preuve, il suffit de se convaincre de ce fait inductivement sur les formules du type $\neg G$, $(G \vee H)$ et $\forall x G$.

1. Cas $\neg G$: puisque $\mathcal{T}$ est complète, $\mathcal{T} \vdash \neg G$ si et seulement si $\mathcal{T} \nvdash G$, ce qui signifie inductivement $\mathfrak{M} \not\models G$, ou encore $\mathfrak{M} \models \neg G$.
2. Cas $(G \vee H)$: supposons $\mathfrak{M} \models (G \vee H)$, et donc $\mathfrak{M} \models G$ ou $\mathfrak{M} \models H$. Dans le premier cas par exemple, par hypothèse d'induction on a $\mathcal{T} \vdash G$, et puisque $(G \Rightarrow (G \vee H))$ est une tautologie, on a $\mathcal{T} \vdash (G \vee H)$. Réciproquement supposons que $\mathcal{T} \vdash (G \vee H)$. Si $\mathcal{T} \vdash G$ alors par hypothèse d'induction $\mathfrak{M} \models G$ et donc $\mathfrak{M} \models (G \vee H)$. Sinon, c'est que $\mathcal{T} \nvdash G$, et parce que la théorie est complète, on a $\mathcal{T} \vdash \neg G$. Or puisque $(G \vee H \Rightarrow (\neg G \Rightarrow H))$ est une tautologie, on obtient que $\mathcal{T} \vdash H$ et par hypothèse d'induction, $\mathfrak{M} \models H$ et donc $\mathfrak{M} \models (G \vee H)$.
3. Cas $\exists x G(x)$: si $\mathfrak{M} \models \exists x G(x)$ c'est qu'il existe un terme clos t tel que $\mathfrak{M} \models G(t/x)$. Par hypothèse d'induction, $\mathcal{T} \vdash G(t/x)$. Or il est facile de trouver une démonstration formelle de $\exists x G(x)$ à partir d'une de $G(t/x)$. Réciproquement, supposons que $\mathcal{T} \vdash \exists x G(x)$. Grâce aux témoins de Henkin, on en déduit qu'il existe une constante c telle que $\mathcal{T} \vdash G(c/x)$, et par hypothèse d'induction $\mathfrak{M} \models G(c/x)$, et donc $\mathfrak{M} \models \exists x G(x)$.

□

Il reste la seconde étape. Une *extension d'une théorie* $\mathcal{T}$ est une théorie $\mathcal{T}'$ qui contient $\mathcal{T}$.

Proposition 6.2 *Toute théorie cohérente $\mathcal{T}$ sur une signature Σ possède une extension $\mathcal{T}'$ sur une signature Σ' (avec Σ' qui contient Σ) qui est cohérente, complète et avec des témoins de Henkin.*

Avant de prouver cette proposition, discutons de ce que nous obtenons : puisqu'un modèle de $\mathcal{T}'$ est un modèle de $\mathcal{T}$, le lemme précédent et la proposition précédente permettent tout d'abord d'obtenir :

Corollaire 6.2 *Une théorie cohérente possède un modèle.*

La remarque suivante relève d'un jeu sur les définitions :

Proposition 6.3 *Pour toute théorie $\mathcal{T}$ et pour toute formule close F, F est une conséquence de T si et seulement si $\mathcal{T} \cup \{\neg F\}$ n'a pas de modèle.*

Démonstration: Si F est une conséquence de $\mathcal{T}$, alors par définition tout modèle de $\mathcal{T}$ est un modèle de F, autrement dit, il n'y pas de modèle de $\mathcal{T} \cup \{\neg F\}$. La réciproque est triviale. □

On obtient avec cette remarque exactement le théorème de complétude (ou le sens manquant de ce que nous avons appelé le théorème de complétude).

Théorème 6.5 *Soit F une formule close. Si F est une conséquence de la théorie $\mathcal{T}$, alors $\mathcal{T} \vdash F$.*

Démonstration: Si $\mathcal{T}$ ne prouve pas F, alors $\mathcal{T} \cup \{\neg F\}$ est cohérente : par le corollaire précédente, $\mathcal{T} \cup \{\neg F\}$ possède donc un modèle. Cela veut donc dire que F n'est pas une conséquence de la théorie $\mathcal{T}$. □

Il ne reste plus qu'à prouver la proposition 6.2.

Démonstration: La signature Σ' est obtenue en ajoutant un nombre dénombrable de nouvelles constantes à la signature Σ. La signature Σ' obtenue reste dénombrable et on peut énumérer les formules closes $(F_n)_{n\in\mathbb{N}}$ de Σ'. La théorie $\mathcal{T}'$ est obtenue comme l'union d'une suite croissante de théories $\mathcal{T}_n$, définie par récurrence, en partant de $\mathcal{T}_0 = \mathcal{T}$. Supposons $\mathcal{T}_n$ cohérente construite. Pour construire $\mathcal{T}_{n+1}$ on considère la formule F_{n+1} dans l'énumération des formules closes de Σ'. Si $\mathcal{T}_n \cup F_{n+1}$ est cohérente, alors on pose $G_n = F_{n+1}$, sinon on pose $G_n = \neg F_{n+1}$. Dans les deux cas $\mathcal{T}_n \cup \{G_n\}$ est cohérente.

La théorie $\mathcal{T}_{n+1}$ est définie par :

1. $\mathcal{T}_{n+1} = \mathcal{T}_n \cup \{G_n\}$ si G_n n'est pas de la forme $\exists x H$.
2. sinon : $\mathcal{T}_{n+1} = \mathcal{T}_n \cup \{G_n, H(c/x)\}$ où c est un nouveau symbole de constante qui n'apparaît dans aucune formule de $T_n \cup \{G_n\}$: il y a toujours un tel symbole, car il y a un nombre fini de symboles de constantes dans $T_n \cup \{G_n\}$.

La théorie $\mathcal{T}_{n+1}$ est cohérente : en effet, si elle ne l'était pas, alors cela voudrait dire que G_n serait de la forme $\exists x H$, et que $T_n \cup \{\exists x H\} \vdash \neg H(c/x)$. Par le choix de la constante c, et par le lemme 6.2, on obtient que $T_n \cup \{\exists x H\} \vdash \forall x \neg H(x)$, ce qui est impossible car sinon $\mathcal{T}_n$ ne serait pas cohérente.

La théorie $\mathcal{T}' = \bigcup_{n\in\mathbb{N}} \mathcal{T}_n$ définie comme l'union des théories $\mathcal{T}_n$ est cohérente, puisque tout sous-ensemble fini de celle-ci est contenu dans l'une des théories $\mathcal{T}_n$, et donc est cohérent.

La théorie $\mathcal{T}'$ est aussi complète : si F est une formule close de Σ', elle apparaît à un moment dans l'énumération des formules F_n, et par construction, soit $F_n \in \mathcal{T}_n$ soit $\neg F_n \in \mathcal{T}_n$.

Enfin la théorie $\mathcal{T}'$ a des témoins de Henkin : si $H(x)$ est une formule avec la variable libre x, alors la formule $\exists x H$ apparaît comme une formule dans

l'énumération des formules F_n. Il y a alors deux cas, soit $\neg F_n \in T_{n+1}$ ou il y a une constante c telle que $H(c/x) \in T_{n+1}$. Dans les deux cas, $\mathcal{T}_{n+1} \vdash \exists x H(x) \Rightarrow H(c/x)$, ce qui prouve que $(\exists x H(x) \Rightarrow F(c))$ est dans $\mathcal{T}'$ (sinon sa négation y serait, et $\mathcal{T}'$ ne serait pas cohérente). □

6.4 Compacité

Observons que l'on a fait établi d'autres faits.

Théorème 6.6 (Théorème de compacité) *Soit $\mathcal{T}$ une théorie telle que toute partie finie de $\mathcal{T}$ possède un modèle. Alors $\mathcal{T}$ possède un modèle.*

Démonstration: Considérons un sous-ensemble fini d'une telle théorie $\mathcal{T}$. Ce sous-ensemble est cohérent puisqu'il a un modèle. $\mathcal{T}$ est donc une théorie telle que toute partie finie soit cohérente. Par le théorème de finitude, cela veut dire que la théorie elle-même est cohérente.

Par le corollaire 6.2, cela veut dire que $\mathcal{T}$ possède un modèle. □

6.5 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture de [Cori and Lascar, 1993a], [Dowek, 2008] et [Lassaigne and de Rougemont, 2004].

Bibliographie Ce chapitre a été rédigé en s'inspirant essentiellement des ouvrages [Cori and Lascar, 1993a] et [Lassaigne and de Rougemont, 2004].

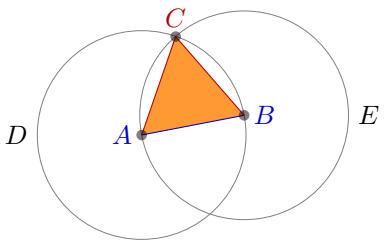
Chapitre 7

Modèles de calculs

Nous avons utilisé jusque-là à de multiples reprises la notion d'algorithme, sans en avoir donné une définition formelle. Intuitivement, on peut se dire qu'un algorithme est une méthode automatique pour résoudre un problème donné, qui peut s'implémenter par un programme informatique.

Par exemple, la technique familière pour réaliser une addition, une multiplication, ou une division sur des nombres enseignée à l'école primaire correspond à un algorithme. Les techniques discutées pour évaluer la valeur de vérité d'une formule propositionnelle à partir de la valeur de ses propositions est un algorithme. Plus généralement, nous avons décrit des méthodes de démonstration pour le calcul propositionnel ou le calcul des prédicats qui peuvent se voir comme des algorithmes.

Exemple 7.1 *L'exemple suivant est repris du manuel du paquetage* TikZ-PGF *version 2.0, lui-même inspiré des* Éléments *d'*Euclide.



Algorithme:
Pour construire un triangle équilatéral ayant pour coté AB: tracer le cercle de centre A de rayon AB; tracer le cercle de centre B de rayon AB. Nommer C l'une des intersections de ces deux cercles. Le triangle ABC est la solution recherchée.

Nous allons voir dans les chapitres suivants que tous les problèmes ne peuvent pas être résolus par algorithme, et ce même pour des problèmes très simples à formuler : par exemple,

– il n'y a pas d'algorithme pour déterminer si une formule du calcul des prédicats est valide dans le cas général ;

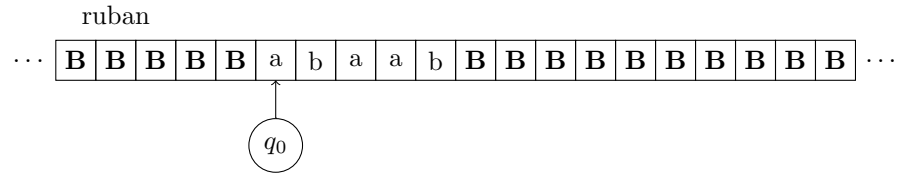


FIG. 7.1 – Machine de Turing. La machine est sur l'état initial d'un calcul sur le mot *abaab*.

– il n'y a pas d'algorithme pour déterminer si un polynôme multivarié (à plusieurs variables) à coefficients entiers possède une racine entière (10*ème problème de Hilbert*).

Historiquement, c'est en fait la formalisation de ce que l'on appelle une démonstration, et des limites des systèmes de preuve formelle qui a mené aux modèles que l'on va discuter, avant que l'on ne comprenne que la notion qui avait été capturée par ces modèles était beaucoup plus large que simplement la formalisation de la notion de démonstration, et couvrait en fait une formalisation de tout ce qui est calculable par dispositif informatique digital. Cela reste parfaitement d'actualité, puisque les ordinateurs actuels sont digitaux.

Par ailleurs, plusieurs formalisations ont été proposées de ces notions de façon indépendante, en utilisant des notions à priori très différentes, en particulier par Alonzo Church en 1936, à l'aide du formalisme du λ-*calcul*, par Turing en 1936, à l'aide de ce que l'on appelle les *machines de Turing*, ou par Post en 1936, à l'aide de systèmes de règles très simples, appelés *systèmes de Post*. Ultérieurement, on s'est convaincu que de nombreux formalismes étaient équivalents à tous ces modèles.

L'objet de ce chapitre est de définir quelques modèles de calculs et de montrer qu'ils sont équivalents à celui de la machine de Turing. Nous finirons en évoquant ce que l'on appelle la *thèse de Church-Turing*, qui a en fait été exprimée pour la première fois par Stephen Kleene, étudiant en thèse d'Alonzo Church.

Les modèles que l'on va décrire peuvent être considérés comme très abstraits, mais aussi et surtout très limités et loin de couvrir tout ce que l'on peut programmer avec les langages de programmation évolués actuels comme CAML ou JAVA. Tout l'objet du chapitre est de se convaincre qu'il n'en est rien : tout ce qui programmable est programmable dans ces modèles.

7.1 Machines de Turing

7.1.1 Ingrédients

Une machine de Turing (déterministe) (voir la figure 7.1) est composée des éléments suivants :

1. Une mémoire infinie sous forme de ruban. Le ruban est divisé en cases. Chaque case peut contenir un élément d'un ensemble Σ (qui se veut un

alphabet). On suppose que l'alphabet Σ est un ensemble fini.

2. une tête de lecture : la tête de lecture se déplace sur le ruban.
3. Un programme donné par une *fonction de transition* qui pour chaque état de la machine q, parmi un nombre fini d'états possibles Q, précise selon le symbole sous la tête de lecture :
   (a) l'état suivant $q' \in Q$;
   (b) le nouvel élément de Σ à écrire à la place de l'élément de Σ sous la tête de lecture ;
   (c) un sens de déplacement pour la tête de lecture.

L'exécution d'une machine de Turing sur un mot $w \in \Sigma^*$ peut alors se décrire comme suit : initialement, l'entrée se trouve sur le ruban, et la tête de lecture est positionnée sur la première lettre du mot. Les cases des rubans qui ne correspondent pas à l'entrée contiennent toutes l'élément **B** (symbole de blanc), qui est un élément particulier de Σ. La machine est positionnée dans son état initial q_0 : voir la figure 7.1.

A chaque étape de l'exécution, la machine, selon son état, lit le symbole se trouvant sous la tête de lecture, et selon ce symbole :
- remplace le symbole sous la tête de lecture par celui précisé par sa fonction transition ;
- déplace (ou non) cette tête de lecture d'une case vers la droite ou vers la gauche suivant le sens précisé par la fonction de transition ;
- change d'état vers l'état suivant.

Le mot w est dit accepté lorsque l'exécution de la machine finit par atteindre l'état d'acceptation.

7.1.2 Description

La notion de machine de Turing se formalise de la façon suivante :

Définition 7.1 (Machine de Turing) Une machine de Turing *est un 8-uplet*

$$M = (Q, \Sigma, \Gamma, \mathbf{B}, \delta, q_0, q_a, q_r)$$

où :

1. Q *est l'ensemble fini des états ;*
2. Σ *est un alphabet* fini *;*
3. Γ *est l'alphabet de travail* fini *:* $\Sigma \subset \Gamma$ *;*
4. $\mathbf{B} \in \Gamma$ *est le caractère blanc ;*
5. $q_0 \in Q$ *est l'état initial ;*
6. $q_a \in Q$ *est l'état d'acceptation ;*
7. $q_r \in Q$ *est l'état de refus (ou d'arrêt) ;*

8. *δ est la fonction de transition : δ est une fonction (possiblement partielle) de $Q \times \Gamma$ dans $Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}$. Le symbole $\leftarrow$ est utilisé pour signifier un déplacement vers la gauche, $|$ aucun déplacement, $\rightarrow$ un déplacement vers la droite.*

Le langage accepté par une machine de Turing se définit à l'aide des notions de *configurations* et de *dérivations* entre configurations. Une configuration correspond à toute l'information nécessaire pour décrire l'état de la machine à un instant donné, et pouvoir déterminer les états ultérieurs de la machine. A savoir :

- l'état ;
- le contenu du ruban ;
- la position de la tête de lecture.

Plus formellement,

Définition 7.2 (Configuration) Une configuration *est donnée par la description du ruban, par la position de la tête de lecture/écriture, et par l'état interne.*

Pour écrire une configuration, une difficulté est que le ruban est infini : le ruban correspond donc à une suite infinie de symboles de l'alphabet Γ de travail de la machine. Toutefois, à tout moment d'une exécution seule une partie finie du ruban a pu être utilisée par la machine. En effet, initialement la machine contient un mot en entrée de longueur finie et fixée, et à chaque étape la machine déplace la tête de lecture au plus d'une seule case. Par conséquent, après t étapes, la tête de lecture a au plus parcouru t cases vers la droite ou vers la gauche à partir de sa position initiale. Par conséquent, le contenu du ruban peut à tout moment se définir par le contenu d'un préfixe fini, le reste ne contenant que le symbole blanc **B**. Pour noter la position de la tête de lecture, nous pourrions utiliser un entier $n \in \mathbb{Z}$.

Nous allons en fait utiliser plutôt l'astuce suivante qui a le seul mérite de simplifier nos définitions et nos preuves ultérieures : au lieu de voir le ruban comme un préfixe fini, nous allons le représenter par deux préfixes finis : le contenu de ce qui est à droite de la tête de lecture, et le contenu de ce qui est à gauche de la tête de lecture. On écrira le préfixe correspondant au contenu à droite comme habituellement de gauche à droite. Par contre, on écrira le préfixe correspondant au contenu à gauche de la tête de lecture de droite à gauche : l'intérêt est que la première lettre du préfixe gauche est la case immédiatement à gauche de la tête de lecture. Une *configuration* sera donc un élément de $Q \times \Gamma^* \times \Gamma^*$.

Formellement :

Définition 7.3 (Notation d'une configuration) *Une configuration peut se noter $C = (q, u, v)$, avec $u, v \in \Gamma^*$, $q \in Q$: u et v désignent le contenu respectivement à gauche et à droite de la tête de lecture du ruban i, la tête de lecture du ruban i étant sur la première lettre de v. On suppose que les dernières lettres de u et de v ne sont pas* **B**.

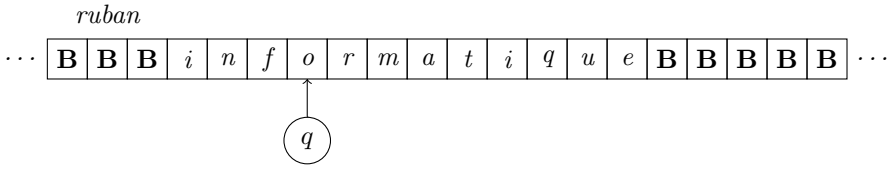
On fixe la convention que le mot v est écrit de gauche à droite (la lettre numéro $i+1$ de v correspond au contenu de la case à droite de celle de numéro i) alors que le mot u est écrit de droite à gauche (la lettre numéro $i+1$ de u correspond au contenu de la case à gauche de celle de numéro i, la première lettre de u étant à gauche de la tête de lecture).

Exemple 7.2 *La configuration de la machine représentée sur la figure 7.1 est* $(q_0, abaab, \epsilon)$.

On notera parfois autrement les configurations :

Définition 7.4 (Notation alternative) *La configuration (q, u, v) sera aussi vue/notée dans certaines sections ou chapitres comme/par $u\mathbf{q}v$, en gardant u et v écrit de gauche à droite.*

Exemple 7.3 *Une configuration comme*



se code par la configuration $(q, fni, ormatique)$, ou parfois par $inf\mathbf{q}ormatique$.

Une configuration est dite *acceptante* si $q = q_a$, *refusante* si $q = q_r$.

Pour $w \in \Sigma^*$, la configuration initiale correspondante à w est la configuration $C[w] = (q_0, \epsilon, w)$.

On note : $C \vdash C'$ si la configuration C' est le successeur direct de la configuration C par le programme (donné par δ) de la machine de Turing. Formellement, si $C = (q, u, v)$ et si a désigne la première lettre[1] de v, et si $\delta(q, a) = (q', a', m')$ alors $C \vdash C'$ si

- $C' = (q', u', v')$, et
  - si $m' = |$, alors $u' = u$, et v' est obtenu en remplaçant la première lettre a de v par a'.
  - si $m' = \leftarrow$, $v' = a'v$, et u' est obtenu en supprimant la première lettre de u.
  - si $m' = \rightarrow$, $u' = ua'$, et v' est obtenu en supprimant la première lettre a de v.

Remarque 7.1 *Ces règles traduisent simplement la notion de réécriture de la lettre a par la lettre a' et le déplacement correspondant à droite ou à gauche de la tête de lecture.*

[1] Avec la convention que la première lettre du mot vide est le blanc **B**.

Définition 7.5 (Mot accepté) *Un mot $w \in \Sigma^*$ est dit* accepté *(en temps t) par la machine de Turing, s'il existe une suite de configurations $C_1, \cdots, C_t$ avec :*

1. *$C_0 = C[w]$;*
2. *$C_i \vdash C_{i+1}$ pour tout $i < t$;*
3. *aucune configuration C_i pour $i < t$ n'est acceptante ou refusante.*
4. *C_t est acceptante.*

Définition 7.6 (Mot refusé) *Un mot $w \in \Sigma^*$ est dit* refusé *(en temps t) par la machine de Turing, s'il existe une suite de configurations $C_1, \cdots, C_t$ avec :*

1. *$C_0 = C[w]$;*
2. *$C_i \vdash C_{i+1}$ pour tout $i < t$;*
3. *aucune configuration C_i pour $i < t$ n'est acceptante ou refusante.*
4. *C_t est refusante.*

Définition 7.7 (Machine qui boucle sur un mot) *On dit que la machine de Turing* boucle *sur un mot w, si w n'est ni accepté, ni refusé.*

Remarque 7.2 *Chaque mot w est donc dans l'un des trois cas exclusifs suivants :*

1. *il est accepté par la machine de Turing ;*
2. *il est refusé par la machine de Turing ;*
3. *la machine de Turing boucle sur ce mot.*

Remarque 7.3 *La terminologie* boucle *signifie simplement que la machine ne s'arrête pas sur ce mot : cela ne veut pas dire nécessairement que l'on répète à l'infini les mêmes instructions. La machine peut boucler pour plusieurs raisons. Par exemple, parce qu'elle atteint une configuration qui n'a pas de successeur défini, ou parce qu'elle rentre dans un comportement complexe qui produit une suite infinie de configurations ni acceptante ni refusante.*

Plus généralement, on appelle *calcul de Σ sur un mot $w \in \Sigma^*$*, une suite (finie ou infinie) de configurations $(C_i)_{i \in \mathbb{N}}$ telle que $C_0 = C[w]$ et pour tout i, $C_i \vdash C_{i+1}$, avec la convention qu'un état acceptant ou refusant n'a pas de successeur.

Définition 7.8 (Langage accepté par une machine) *Le langage $L \subset \Sigma^*$* accepté par *M est l'ensemble des mots w qui sont acceptés par la machine. On le note $L(M)$. On l'appelle $L(M)$ aussi* le langage reconnu *par M.*

On n'aime pas en général les machines qui ne s'arrêtent pas. On cherche donc en général à garantir une propriété plus forte :

Définition 7.9 (Langage décidé par une machine) *On dit qu'un langage $L \subset \Sigma^*$ est* dit décidé par *Σ par la machine si :*

– *pour $w \in L$, w est accepté par la machine ;*
– *pour $w \notin L$ (=sinon), w est refusé par la machine.*

Autrement dit, la machine accepte L et termine sur toute entrée.

On dit dans ce cas que la machine *décide* L.

7.1.3 Programmer avec des machines de Turing

La programmation avec des machines de Turing est extrêmement bas niveau. Nous allons voir que l'on peut toutefois programmer réellement beaucoup de choses avec ce modèle. La première étape est de se convaincre que plein de problèmes simples peuvent se programmer. A vrai dire, la seule façon de s'en convaincre est d'essayer soit même de programmer avec des machines de Turing, c'est-à-dire de faire les exercices qui suivent.

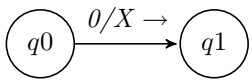
Exercice 7.1 *Construire une machine de Turing qui accepte exactement les mots w sur l'alphabet $\Sigma = \{0, 1\}$ de la forme $0^n 1^n$, $n \in \mathbb{N}$.*

Voici une solution. On considère une machine avec $Q = \{q0, q1, q2, q3, q4\}$, $\Gamma = \{0, 1, X, Y, B\}$, l'état d'acceptation $q4$ et une fonction de transition δ telle que :

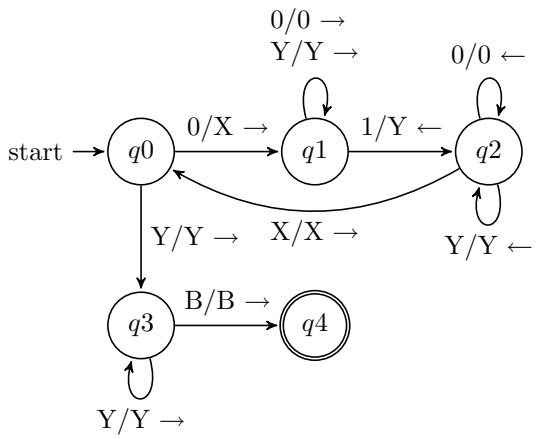
– $\delta(q0, 0) = (q1, X, \rightarrow)$;
– $\delta(q0, Y) = (q3, Y, \rightarrow)$;
– $\delta(q1, 0) = (q1, 0, \rightarrow)$;
– $\delta(q1, 1) = (q2, Y, \leftarrow)$;
– $\delta(q1, Y) = (q1, Y, \rightarrow)$;
– $\delta(q2, 0) = (q2, 0, \leftarrow)$;
– $\delta(q2, X) = (q0, X, \rightarrow)$;
– $\delta(q2, Y) = (q2, Y, \leftarrow)$;
– $\delta(q3, Y) = (q3, Y, \rightarrow)$;
– $\delta(q3, B) = (q4, B, \rightarrow)$.

On le voit, décrire de cette façon une machine de Turing est particulièrement peu lisible. On préfère représenter le programme d'une machine (la fonction δ) sous la forme d'un graphe : les sommets du graphe représentent les états de la machine. On représente chaque transition $\delta(q, a) = (q', a', m)$ par un arc de l'état q vers l'état q' étiqueté par a/a' m. L'état initial est marqué par une flèche entrante. L'état d'acceptation est marqué par un double cercle.

Exemple 7.4 *Par exemple, la transition $\delta(q0, 0) = (q1, X, \rightarrow)$ se représente graphiquement par :*



Selon ce principe, le programme précédent se représente donc par :



Comment fonctionne ce programme : lors d'un calcul, la partie du ruban que la machine aura visité sera de la forme $X^*0^*Y^*1^*$. A chaque fois que l'on lit un 0, on le remplace par un X, et on rentre dans l'état $q1$ ce qui correspond à lancer la sous-procédure suivante : on se déplace à droite tant que l'on lit un 0 ou un Y. Dès qu'on a atteint un 1, on le transforme en un Y, et on revient à gauche jusqu'à revenir sur un X (le X qu'on avait écrit) et s'être déplacé d'une case vers la droite.

En faisant ainsi, pour chaque 0 effacé (i.e. marqué X), on aura effacé un 1 (i.e. marqué un Y). Si on a marqué tous les 0 et que l'on atteint un Y, on rentre dans l'état $q3$, ce qui a pour objet de vérifier que ce qui est à droite est bien constitué uniquement de Y. Lorsqu'on a tout lu, i.e. atteint un B, on accepte, i.e. on va dans l'état $q4$.

Bien entendu, une vraie preuve de la correction de cet algorithme consisterait à montrer que si un mot est accepté, c'est que nécessairement il est du type 0^n1^n. Nous laissons le lecteur s'en convaincre.

Exemple 7.5 *Voici un exemple de calcul acceptant pour M : $q_0 0011 \vdash Xq_1011 \vdash X0q_111 \vdash Xq_20Y1 \vdash q_2X0Y1 \vdash Xq_00Y1 \vdash XXq_1Y1 \vdash XXYq_11 \vdash XXq_2YY \vdash Xq_2XYY \vdash XXq_0YY \vdash XXYq_3Y \vdash XXYYq_3\mathbf{B} \vdash XXYY\mathbf{B}q_4\mathbf{B}$.*

Définition 7.10 (Diagramme espace-temps) *On représente souvent une suite de configurations ligne par ligne : la ligne numéro i représente la i ème configuration du calcul, avec le codage de la définition 7.4. Cette représentation est appelée un* diagramme espace-temps *de la machine.*

Exemple 7.6 *Voici le diagramme espace-temps correspondant au calcul précédent sur* 0011.

$\cdots$	**B**	**B**	**B**	**B**	q_0	*0*	*0*	*1*	*1*	**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**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	*0*	q_1	*1*	*1*	**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**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	q_2	X	*0*	Y	*1*	**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**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	q_1	Y	*1*	**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**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	q_2	Y	Y	**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**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	q_0	Y	Y	**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**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	Y	Y	q_3	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**	$\cdots$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

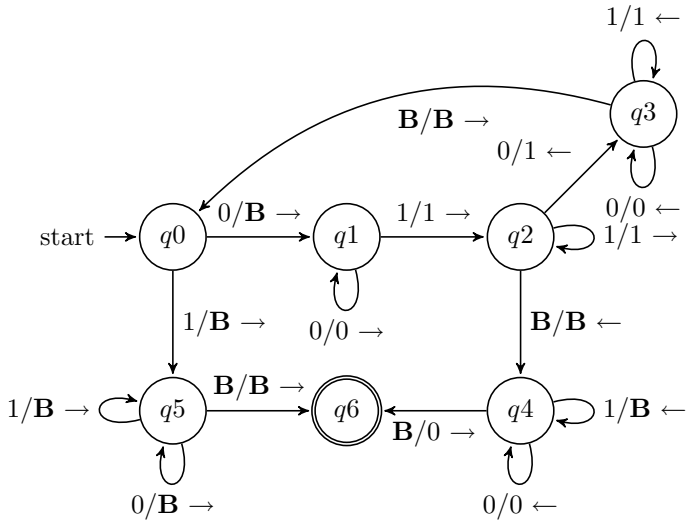
Exemple 7.7 *Voici le diagramme espace-temps du calcul de la machine sur* 0010 *:*

$\cdots$	**B**	**B**	**B**	**B**	q_0	*0*	*0*	*1*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	q_1	*0*	*1*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	*0*	q_1	*1*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	q_2	*0*	Y	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	q_2	X	*0*	Y	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	q_0	*0*	Y	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	q_1	Y	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	Y	q_1	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	Y	*0*	q_1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Observons que sur la dernière configuration plus aucune évolution n'est possible, et donc il n'y a pas de calcul accepté partant de 0010*.*

Exercice 7.2 (Soustraction en unaire) *Construire un programme de machine de Turing qui réalise une soustraction en unaire : partant d'un mot de la forme* 0^m10^n*, la machine s'arrête avec* $0^{m\ominus n}$ *sur son ruban (entouré de blancs), où* $m \ominus n$ *est* $\max(0, m-n)$*.*

Voici une solution : on considère une machine sur l'ensemble d'états $Q = \{q0, q1, q2, \cdots, q6\}$ avec $\Gamma = \{0, 1, \mathbf{B}\}$.



La machine est construite pour effectuer le travail suivant : elle recherche le 0 le plus à gauche et le remplace par un blanc. Elle cherche alors à droite un 1, quand elle en trouve un elle continue à droite jusqu'à trouver un 0 qu'elle remplace par un 1. La machine retourne alors à gauche pour trouver le 0 le plus à gauche qu'elle identifie en trouvant le premier blanc en se déplaçant à gauche et en se déplaçant depuis ce blanc d'une case vers la droite.

On répète le processus jusqu'à ce que :

- soit en cherchant à droite un 0, on rencontre un blanc. Alors les n 0 dans 0^m10^n ont été changés en 1 et $n+1$ des m 0 ont été changés en **B**. Dans ce cas, la machine remplace les $n+1$ 1 par un 0 et n blancs, ce qui laisse $m-n$ 0 sur le ruban. Puisque dans ce cas, $m \geq n$, $m \ominus n = m - n$.
- ou en recommençant le cycle, la machine n'arrive pas à trouver un 0 à changer en blanc, puisque les m premiers 0 ont déjà été changés en **B**. Alors $n \geq m$, et donc $m \ominus n = 0$. La machine remplace alors tous les 1 et 0 restants par des blancs, et termine avec un ruban complètement blanc.

7.1.4 Techniques de programmation

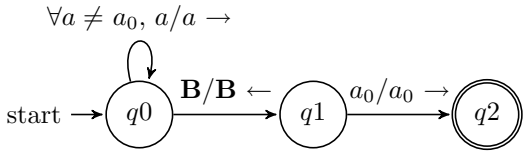
Voici quelques techniques utilisées couramment dans la programmation des machines de Turing.

La première consiste à coder une information finie dans l'état de la machine. On va l'illustrer sur un exemple, où l'on va stocker le premier caractère lu dans l'état. Tant que l'information à stocker est finie, cela reste possible.

Exercice 7.3 *Construire un programme de machine de Turing qui lit le symbole en face de la tête de lecture et vérifie que ce dernier n'apparaît nul part ailleurs*

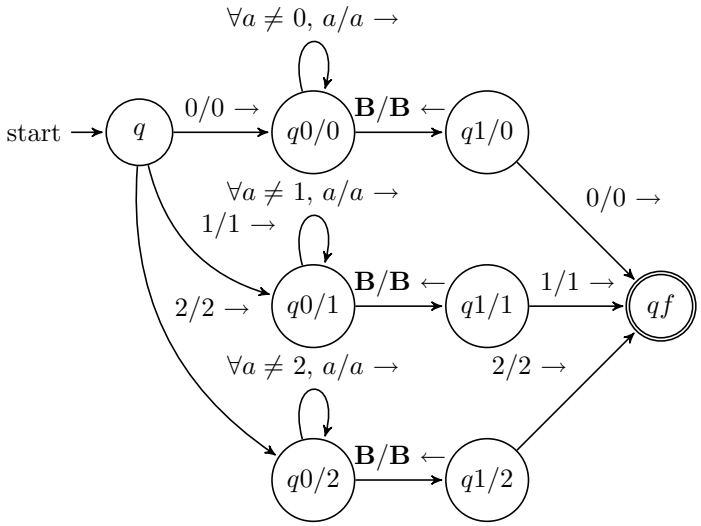
à droite.

Si l'on fixe un symbole $a_0 \in \Sigma$ de l'alphabet Σ, il est facile de construire un programme qui vérifie que le symbole a_0 n'apparaît nul part sauf sur la toute dernière lettre à droite.



où $\forall a \neq a_0$ désigne le fait que l'on doit répéter la transition $a/a, \rightarrow$ pour tout symbole $a \neq a_0$.

Maintenant pour résoudre notre problème, il suffit de lire la première lettre a_0 et de recopier ce programme autant de fois qu'il y a de lettres dans Σ. Si $\Sigma = \{0, 1, 2\}$ par exemple :



On utilise donc le fait dans cet automate que l'on travaille sur des états qui peuvent être des couples : ici on utilise des couples q_i/j avec $i \in \{1, 2\}$, et $j \in \Sigma$.

Une second technique consiste en l'utilisation de sous-procédures. Là encore, on va l'illustrer sur un exemple.

Exercice 7.4 (Multiplication en unaire) *Construire un programme de machine de Turing qui réalise une multiplication en unaire : partant d'un mot de la forme 0^m10^n, la machine s'arrête avec 0^{m*n} sur son ruban.*

Une stratégie possible est la suivante :

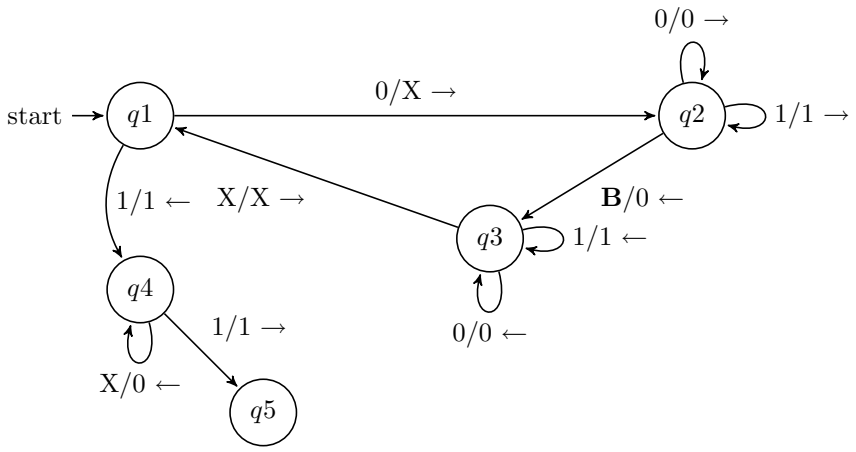
1. le ruban contiendra un mot de la forme $0^i10^n10^{kn}$ pour un certain entier k ;

2. dans chaque étape, on change un 0 du premier groupe en un blanc, et on ajoute n 0 au dernier groupe, pour obtenir une chaîne de la forme $0^{i-1}10^n10^{(k+1)n}$;

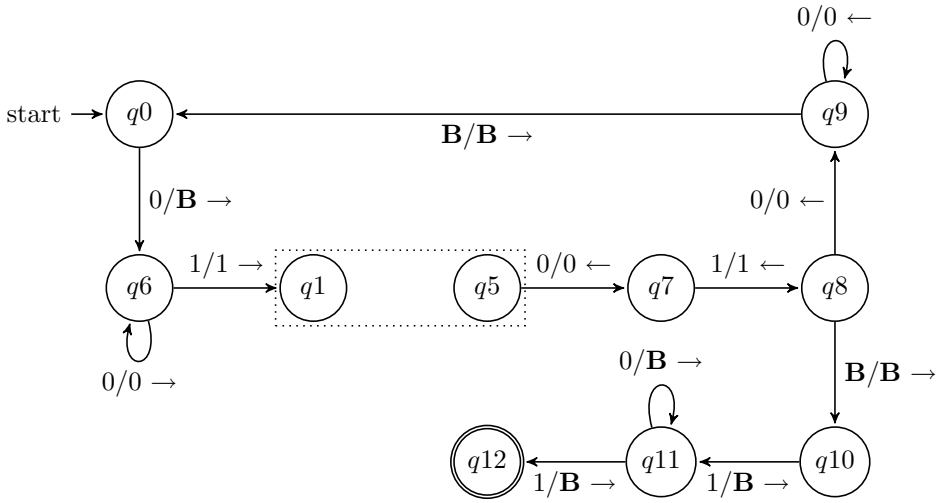
3. en faisant ainsi, on copie le groupe de n 0 m fois, une fois pour chaque symbole du premier groupe mis à blanc. Quand il ne reste plus de blanc dans le premier groupe de 0, il y aura donc $m * n$ groupes dans le dernier groupe ;

4. la dernière étape est de changer le préfixe 10^n1 en des blancs, et cela sera terminé.

Le cœur de la méthode est donc la sous-procédure, que l'on appellera *Copy* qui implémente l'étape 2 : elle transforme une configuration $0^{m-k}10^n1^{(k-1)n}$ en $0^{m-k}10^n1^{kn}$. Voici une façon de la programmer : si l'on part dans l'état q_1 avec une telle entrée, on se retrouve dans l'état q_5 avec le résultat correct.



Une fois que l'on a cette sous-procédure, on peut concevoir l'algorithme global.



où le rectangle en pointillé signifie “coller ici le programme décrit avant pour la sous-procédure”.

On le voit sur cet exemple, il est possible de programmer les machines de Turing de façon modulaire, en utilisant des notions de sous-procédure, qui correspondent en fait à des collages de morceaux de programme au sein du programme d'une machine, comme sur cet exemple.

7.1.5 Applications

Répétons-le : la seule façon de comprendre tout ce que l'on peut programmer avec une machine de Turing consiste à essayer de les programmer.

Voici quelques exercices.

Exercice 7.5 *Construire une machine de Turing qui ajoute* 1 *au nombre écrit en binaire (donc avec des* **0** *et* **1***) sur son ruban.*

Exercice 7.6 *Construire une machine de Turing qui soustrait* 1 *au nombre écrit en binaire (donc avec des* **0** *et* **1***) sur son ruban.*

Exercice 7.7 *Construire une machine de Turing qui accepte les chaînes de caractère avec le même nombre de* **0** *et de* **1***.*

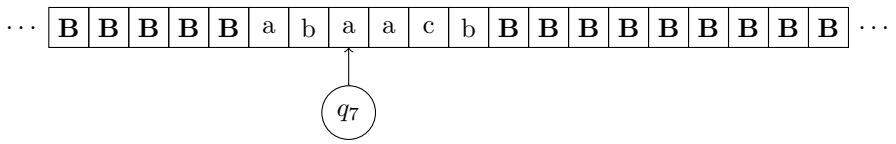
7.1.6 Variantes de la notion de machine de Turing

Le modèle de la machine de Turing est extrêmement robuste.

En effet, existe de nombreuses variantes possibles autour du concept de machine de Turing, qui ne changent rien en fait à ce que l'on arrive à programmer avec ces machines.

On peut en effet assez facilement se persuader des faits suivants.

Machine M sur l'alphabet $\{a, b, c\}$



Machine M' simulant M sur l'alphabet $\{0, 1\}$.

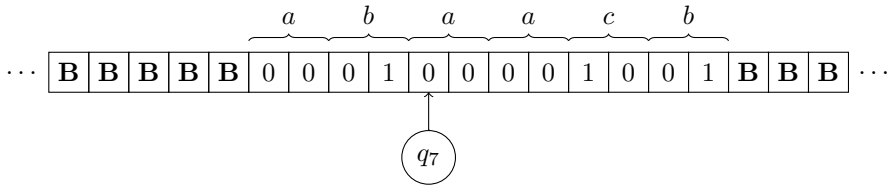


FIG. 7.2 – Illustration de la preuve de la proposition 7.1.

Restriction à un alphabet binaire

Proposition 7.1 *Toute machine de Turing qui travaille sur un alphabet* Σ *quelconque peut être simulée par une machine de Turing qui travaille sur un alphabet* $\Sigma = \Gamma$ *avec uniquement deux lettres.*

Démonstration (principe): L'idée est que l'on peut toujours coder les lettres de l'alphabet en utilisant un codage en binaire. Par exemple, si l'alphabet Σ possède 3 lettres a, b, et c, on peut décider de coder a par 00, b par 01 et c par 10 : voir la figure 7.2. Dans le cas plus général, il faut simplement utiliser éventuellement plus que 2 lettres.

On peut alors transformer le programme d'une machine de Turing M qui travaille sur l'alphabet Σ en un programme M' qui travaille sur ce codage.

Par exemple, si le programme de M contient une instruction qui dit que si M est dans l'état q, et que la tête de lecture lit un a il faut écrire un c et se déplacer à droite, le programme de M' consistera à dire que si l'on est dans l'état q et que l'on lit 0 en face de la tête de lecture, et 0 à sa droite (donc ce qui est à droite de la tête de lecture commence par 00, i.e. le codage de a), alors il faut remplacer ces deux 0 par 10 (i.e. le codage de c) et se rendre dans l'état q'. En faisant ainsi, à chaque fois qu'un calcul de M produit un ruban correspondant à un mot w, alors M' produira un un ruban correspondant au codage de w en binaire lettre par lettre. □

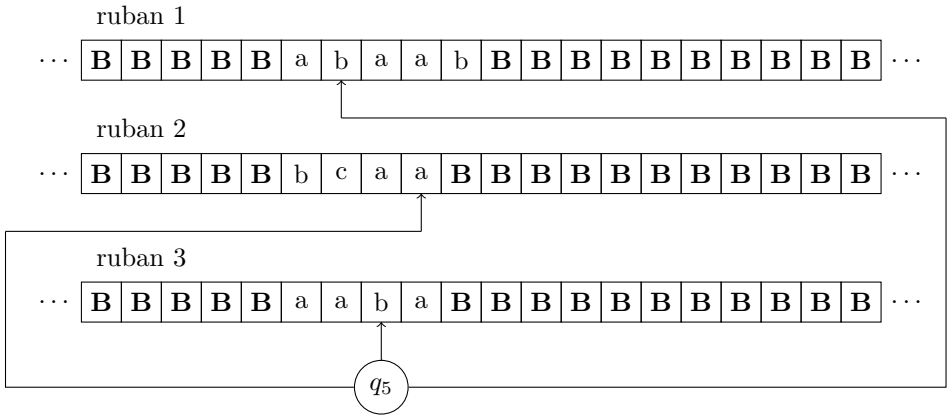


FIG. 7.3 – Une machine de Turing à 3 rubans

Machines de Turing à plusieurs rubans

On peut aussi considérer des machines de Turing qui auraient plusieurs rubans, disons k rubans, où k est un entier. Chacun des k rubans possède sa propre tête de lecture. La machine possède toujours un nombre fini d'états Q. Simplement, maintenant la fonction de transition δ n'est plus une fonction de $Q \times \Gamma$ dans $Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}$. mais de $Q \times \Gamma^k$ dans $Q \times \Gamma^k \times \{\leftarrow, |, \rightarrow\}^k$: en fonction de l'état de la machine et de ce qui est lu en face des têtes de lecture de chaque ruban, la fonction de transition donne les nouveaux symboles à écrire sur chacun des rubans, et les déplacements à effectuer sur chacun des rubans.

Il est possible de formaliser ce modèle, ce que nous ne ferons pas car cela n'apporte pas de réelle nouvelle difficulté.

On pourra se persuader du résultat suivant :

Proposition 7.2 *Toute machine de Turing qui travaille avec k-rubans peut être simulée par une machine de Turing avec un unique ruban.*

Démonstration (principe): L'idée est que si une machine M travaille avec k rubans sur l'alphabet Γ, on peut simuler M par une machine M' avec un unique ruban qui travaille sur l'alphabet $(\Gamma \times \{0,1\} \cup \{\#\})$ (qui est toujours un alphabet fini).

Le ruban de M' contient la concaténation des contenus des rubans de M, séparés par un marqueur $\#$. On utilise $(\Gamma \times \{0,1\} \cup \{\#\})$ au lieu de $(\Gamma \cup \{\#\})$ de façon à utiliser 1 bit d'information de plus par case qui stocke l'information "la tête de lecture est en face de cette case".

M' va simuler étape par étape les transitions de M : pour simuler une transition de M, M' va parcourir de gauche à droite son ruban pour déterminer la position de chacune des têtes de lecture, et le symbole en face de chacune des

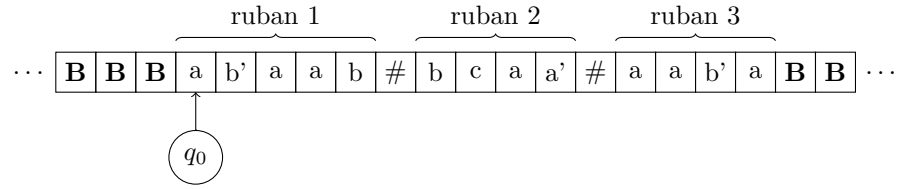


FIG. 7.4 – Illustration de la preuve de la proposition 7.2 : représentation graphique d'une machine de Turing à 1 ruban simulant la machine à 3 rubans de la figure 7.3. Sur cette représentation graphique, on écrit une lettre primée lorsque le bit "la tête de lecture est en face de cette case" est à 1.

têtes de lecture (en mémorisant ces symboles dans son état interne). Une fois connu tous les symboles en face de chacune des têtes de lecture, M' connaît les symboles à écrire et les déplacements à effectuer pour chacune des têtes : M' va parcourir son ruban à nouveau de gauche à droite pour mettre à jour son codage de l'état de M. En faisant ainsi systématiquement transition par transition, M' va parfaitement simuler l'évolution de M avec son unique ruban : voir la figure 7.1.6 □

Machines de Turing non-déterministes

On peut aussi introduire le concept de machine de Turing non-déterministe : la définition d'une machine de Turing non-déterministe est exactement comme celle de la notion de machine de Turing (déterministe) sauf sur un point. δ n'est plus une fonction de $Q \times \Gamma$ dans $Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}$, mais une relation de la forme

$$\delta \subset (Q \times \Gamma) \times (Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}).$$

En d'autres termes, pour un état et une lettre lue en face de la tête de lecture donnée, δ ne définit pas un seul triplet de $Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}$, mais un ensemble de triplets. Intuitivement, lors d'une exécution la machine a la possibilité de choisir n'importe quel triplet.

Formellement, cela s'exprime par le fait que l'on peut passer de la configuration C à la configuration successeur C' si et seulement si on peut passer de C à C' (ce que nous notons $C \vdash C'$) avec les définitions précédentes, mais en remplaçant $\delta(q, a) = (q', a', m')$ par $((q, a), (q', a', m')) \in \delta$. Les autres définitions sont alors essentiellement inchangées, et comme pour les machines de Turing déterministes.

La différence est qu'une machine de Turing non-déterministe n'a pas une exécution unique sur une entrée w, mais éventuellement plusieurs : en fait, les exécutions de la machine sur un mot w donnent lieu à un arbre de possibilité, et l'idée est qu'on accepte (respectivement : refuse) un mot si l'une des branches contient une configuration acceptante (resp. de refus).

La notion de mot w accepté est (toujours) donnée par définition 7.5.

Le langage $L \subset \Sigma^*$ *accepté par* M est (toujours) l'ensemble des mots w qui sont acceptés par la machine. On le note (toujours) $L(M)$. On appelle (toujours) $L(M)$ aussi *le langage reconnu* par M.

On évite dans ce contexte de parler en général de mot *refusé*.

On dira cependant qu'un langage $L \subset \Sigma^*$ est *décidé par* M si il est accepté par une machine qui termine sur toute entrée : c'est-à-dire telle que pour $w \in L$, la machine possède **un** calcul qui mène à une configuration acceptante comme dans la définition 7.5, et pour $w \notin L$, **tous** les calculs de la machine mènent à une configuration refusante.

On peut prouver le résultat suivant (ce que nous ferons dans un chapitre ultérieur).

Proposition 7.3 *Une machine de Turing non-déterministe peut être simulée par une machine de Turing déterministe : un langage L est accepté par une machine de Turing non-déterministe si et seulement si il est accepté par une machine de Turing (déterministe).*

Évidemment, on peut considérer une machine de Turing comme une machine de Turing non-déterministe particulière. Le sens moins trivial de la proposition est que l'on peut simuler une machine de Turing non-déterministe par une machine de Turing (déterministe).

Autrement dit, autoriser du non-déterministe n'étend pas le modèle, tant que l'on parle de *calculabilité*, c'est-à-dire de ce que l'on peut résoudre. Nous verrons qu'en ce qui concerne la *complexité*, cela est une autre paire de manches.

7.1.7 Localité de la notion de calcul

Voici une propriété fondamentale de la notion de calcul, que nous utiliserons à de plusieurs reprises, et que nous invitons notre lecteur à méditer :

Proposition 7.4 (Localité de la notion de calcul) *Considérons un diagramme espace-temps d'une machine M. Regardons les contenus possibles des sous-rectangles de largeur 3 et de hauteur 2 dans ce diagramme. Pour chaque machine M, il y a un nombre fini possible de contenus que l'on peut trouver dans ces rectangles. Appelons* fenêtres légales*, les contenus possibles pour la machine M : voir la figure 7.6.*

Par ailleurs, cela fournit même une caractérisation des diagrammes espace-temps d'une machine donnée : un tableau est un diagramme espace-temps de M sur une certaine configuration initiale C_0 si et seulement si d'une part sa première ligne correspond à C_0, et d'autre part dans ce tableau, le contenu de tous les rectangles de largeur 3 et de hauteur 2 possible sont parmi les fenêtres légales.

Démonstration: Il suffit de regarder chacun des cas possibles et de s'en convaincre, ce qui est fastidieux, mais sans aucune difficulté particulière. □

Nous y reviendrons. Oublions-la pour l'instant, et revenons à d'autres modèles.

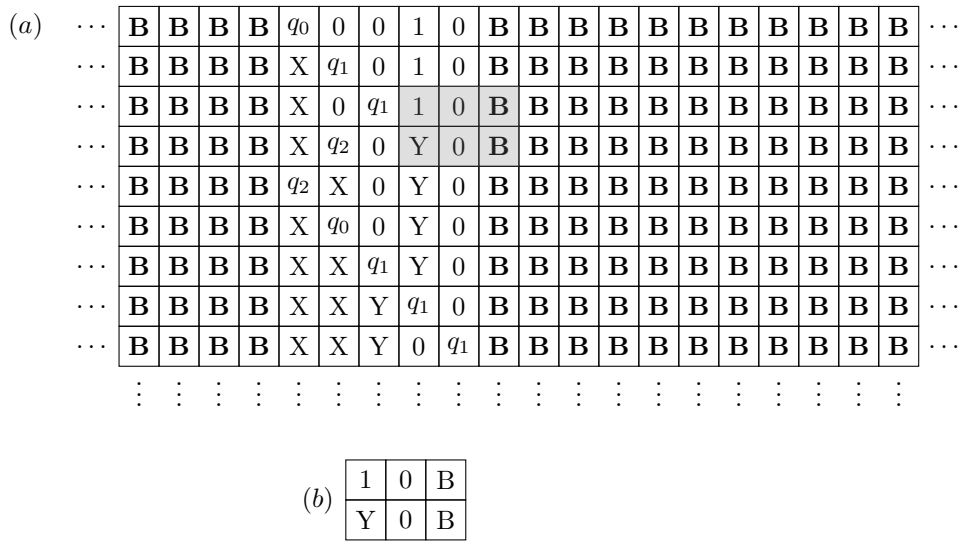


FIG. 7.5 – (a). Le diagramme espace-temps de l'exemple 7.7, sur lequel est grisé un sous-rectangle 3×2. (b) La fenêtre (légale) correspondante.

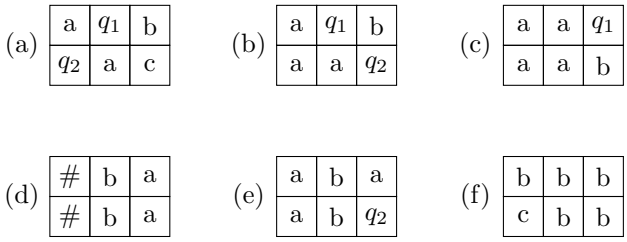


FIG. 7.6 – Quelques fenêtres légales pour une autre machine de Turing M : on peut rencontrer chacun de ces contenus dans un sous-rectangle 3×2 du diagramme espace-temps de M.

(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

FIG. 7.7 – Quelques fenêtres illégales pour une certaine machine M avec $\delta(q_1, b) = (q_1, c, \leftarrow)$. On ne peut pas rencontrer ces contenus dans un sous-rectangle 3×2 du diagramme espace-temps de M : En effet, dans (a), le symbole central ne peut pas changer sans que la tête lui soit adjacente. Dans (b), le symbole en bas à droite devrait être un c mais pas un a, selon la fonction de transition. Dans (c), il ne peut pas y avoir deux têtes de lecture sur le ruban.

Remarque 7.4 *C'est aussi vrai dans les autres modèles dans un certain sens. Toutefois, cela y est toutefois beaucoup plus difficile à formuler.*

7.2 Machines RAM

Le modèle de la machine de Turing peut paraître extrêmement rudimentaire. Il n'en demeure pas extrêmement puissant, et capable de capturer la notion de calculable en informatique.

L'objectif de cette section est de se persuader de ce fait : tout ce qui est programmable par un dispositif de calcul informatique digital actuel peut se simuler par une machine de Turing. Pour cela, on va introduire un modèle très proche (le plus proche en fait que je connaisse) de la façon dont fonctionnent les processeurs actuels : le *modèle des machines RAM*.

7.2.1 Modèle des machines RAM

Le modèle des *machines RAM (Random Access Machine)* est un modèle de calcul qui ressemble beaucoup plus aux langages machine actuels, et à la façon dont fonctionnent les processeurs actuels.

Une machine RAM possède des registres qui contiennent des entiers naturels (nuls si pas encore initialisés). Les instructions autorisées dépendent du processeur que l'on veut modéliser (ou de l'ouvrage que l'on consulte qui décrit ce modèle), mais elles incluent en général la possibilité de :

1. copier le contenu d'un registre dans un autre ;
2. l'adressage indirect : récupérer/écrire le contenu d'un registre dont le numéro est donné par la valeur d'un autre registre ;
3. effectuer des opérations élémentaires sur un ou des registres, par exemple additionner 1, soustraire 1 ou tester l'égalité à 0 ;
4. effectuer d'autres opérations sur un ou des registres, par exemple l'addition, la soustraction, la multiplication, division, les décalages binaires, les opérations binaires bit à bit.

Dans ce qui suit, nous réduirons tout d'abord la discussion aux *SRAM (Successor Random Access Machine)* qui ne possèdent que des instructions des types 1., 2. et 3. Nous verrons qu'en fait cela ne change pas grand chose, du moment que chacune des opérations du type 4. se simule par machine de Turing (et c'est le cas de tout ce qui est évoqué plus haut).

7.2.2 Simulation d'une machine RISC par une machine de Turing

Nous allons montrer que toute machine RAM peut être simulée par une machine de Turing.

Pour aider à la compréhension de la preuve, nous allons réduire le nombre d'instructions des machines RAM à un ensemble réduit d'instructions (*RISC reduced instruction set* en anglais) en utilisant un unique registre x_0 comme accumulateur.

Définition 7.11 *Une machine RISC est une machine SRAM dont les instructions sont uniquement de la forme :*

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*** *aller à l'instruction numéro* j ;
5. $x_0 \leftarrow x_i$;
6. $x_i \leftarrow x_0$;
7. $x_0 \leftarrow x_{x_i}$;
8. $x_{x_0} \leftarrow x_i$.

Clairement, tout programme RAM avec des instructions du type 1., 2. et 3. peut être converti en un programme RISC équivalent, en remplaçant chaque instruction par des instructions qui passent systématiquement par l'accumulateur x_0.

Théorème 7.1 *Toute machine RISC peut être simulée par une machine de Turing.*

Démonstration: La machine de Turing qui simule la machine RISC possède 4 rubans. Les deux premiers rubans codent les couples (i, x_i) pour x_i non nul. Le troisième ruban code l'accumulateur x_0 et le quatrième est un ruban de travail.

Plus concrètement, pour un entier i, notons $\langle i \rangle$ son écriture en binaire. Le premier ruban code un mot de la forme

$$\ldots \mathbf{BB}\langle i_o \rangle \mathbf{B}\langle i_1 \rangle \cdots \mathbf{B} \ldots \langle i_k \rangle \mathbf{BB} \cdots .$$

Le second ruban code un mot de la forme

$$\ldots \mathbf{BB}\langle x_{i_o} \rangle \mathbf{B}\langle x_{i_1} \rangle \cdots \mathbf{B} \ldots \langle x_{i_k} \rangle \mathbf{BB} \cdots$$

Les têtes de lecture des deux premiers rubans sont sur le deuxième **B**. Le troisième ruban code $\langle x_0 \rangle$, la tête de lecture étant tout à gauche. On appelle *position standard* une telle position des têtes de lecture.

La simulation est décrite pour trois exemples. Notre lecteur pourra compléter le reste.

1. $x_0 \leftarrow x_0 + 1$: on déplace la tête de lecture du ruban 3 tout à droite jusqu'à atteindre un symbole **B**. On se déplace alors d'une case vers la gauche, et on remplace les **1** par des **0**, en se déplaçant vers la gauche tant que possible. Lorsqu'un **0** ou un **B** est trouvé, on le change en **1** et on se déplace à gauche pour revenir en position standard.

2. $x_{23} \leftarrow x_0$: on parcourt les rubans 1 et 2 vers la droite, bloc délimité par **B** par bloc, jusqu'à atteindre la fin du ruban 1, ou ce que l'on lise un bloc **B10111B** (**10111** correspond à 23 en binaire).

   Si la fin du ruban 1 a été atteinte, alors l'emplacement 23 n'a jamais été vu auparavant. On l'ajoute en écrivant **10111** à la fin du ruban 1, et on recopie le ruban 3 (la valeur de x_0) sur le ruban 2. On retourne alors en position standard.

   Sinon, c'est que l'on a trouvé **B10111B** sur le ruban 1. On lit alors $\langle x_{23} \rangle$ sur le ruban 2. Dans ce cas, il doit être modifié. On fait cela de la façon suivante :

   (a) On copie le contenu à droite de la tête de lecture numéro 1 sur le ruban 4.

   (b) On copie le contenu du ruban 3 (la valeur de x_0) à la place de x_{23} sur le ruban 2.

   (c) On écrit **B**, et on recopie le contenu du ruban 4 à droite de la tête de lecture du ruban 2, de façon à restaurer le reste du ruban 2.

   (d) On retourne en position standard.

3. $x_0 \leftarrow x_{x_{23}}$: En partant de la gauche des rubans 1 et 2, on parcourt les rubans 1 et 2 vers la droite, bloc délimité par **B** par bloc, jusqu'à atteindre la fin du ruban 1, ou ce que l'on lise un bloc **B10111B** (**10111** correspond à 23 en binaire).

   Si la fin du ruban 1 a été atteinte, on ne fait rien, puisque x_{23} vaut 0 et le ruban 3 contient déjà $\langle x_0 \rangle$.

   Sinon, c'est que l'on a trouvé **B10111B** sur le ruban 1. On lit alors $\langle x_{23} \rangle$ sur le ruban 2, que l'on recopie sur le ruban 4. Comme ci-dessus, on parcourt les rubans 1 et 2 en parallèle jusqu'à trouver $\mathbf{B}\langle x_{23} \rangle\mathbf{B}$ où atteindre la fin du ruban 1. Si la fin du ruban 1 est atteinte, alors on écrit **0** sur le ruban 3, puisque $x_{x_{23}} = x_0$. Sinon, on copie le bloc correspondant sur le ruban 1 sur le ruban 3, puisque le bloc sur le ruban 2 contient $x_{x_{23}}$, et on retourne en position standard.

□

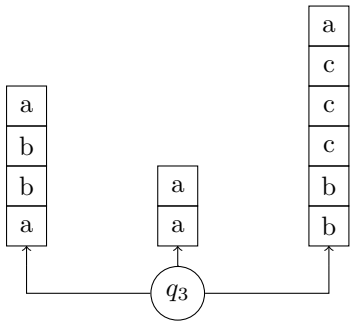


FIG. 7.8 – Une machine à 3 piles.

7.2.3 Simulation d'une machine RAM par une machine de Turing

Revenons sur le fait que nous avons réduit l'ensemble des opérations autorisées sur une machine *RAM* aux instructions du type 1., 2. et 3. En fait, on observera que l'on peut bien gérer toutes instructions du type 4., du moment que l'opération sous-jacente peut bien se calculer par machine de Turing : toute opération $x_0 \leftarrow x_0$ ''opération'' x_i peut être simulée comme plus haut, dès que '' opération'' correspond à une opération calculable.

7.3 Modèles rudimentaires

Le modèle des machine des Turing est extrêmement rudimentaire. On peut toutefois considérer des modèles qui le sont encore plus, et qui sont toutefois capable de les simuler.

7.3.1 Machines à $k \geq 2$ piles

Une *machine à k piles*, possède un nombre fini k de piles $r_1, r_2, \cdots, r_k$, qui correspondent à des piles d'éléments de Σ. Les instructions d'une machine à piles permettent seulement d'empiler un symbole sur l'une des piles, tester la valeur du sommet d'une pile, ou dépiler le symbole au sommet d'une pile.

Si l'on préfère, on peut voir une pile d'éléments de Σ comme un mot w sur l'alphabet Σ. Empiler (*push*) le symbole $a \in M$ correspond à remplacer w par aw. Tester la valeur du sommet d'une pile (*top*) correspond à tester la première lettre du mot w. Dépiler (*pop*) le symbole au sommet de la pile correspond à supprimer la première lettre de w.

Théorème 7.2 *Toute machine de Turing peut être simulée par une machine à* 2 *piles.*

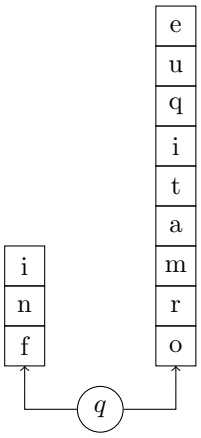


FIG. 7.9 – La machine de Turing de l'exemple 7.3 vue comme une machine à 2-piles.

Démonstration: Selon la formalisation page 84, une configuration d'une machine de Turing correspond à $C = (q, u, v)$, où u et v désignent le contenu respectivement à gauche et à droite de la tête de lecture du ruban i. On peut voir u et v comme des piles : voir la figure 7.9. Si l'on relit attentivement la formalisation page 84, on observe que les opérations effectuées par le programme de la machine de Turing pour passer de la configuration C a sa configuration successeur C' correspondent à des opérations qui se codent trivialement par des *push*, *pop*, et *top* : on peut donc construire une machine à 2 piles, chaque pile codant u ou v (le contenu à droite et à gauche de chacune des têtes de lecture), et qui simule étape par étape la machine de Turing. □

7.3.2 Machines à compteurs

Nous introduisons maintenant un modèle encore plus rudimentaire : une *machine à compteurs* possède un nombre fini k de compteurs $r_1, r_2, \cdots, r_k$, qui contiennent des entiers naturels. Les instructions d'une machine à compteur permettent seulement de tester l'égalité d'un des compteurs à 0, d'incrémenter un compteur ou de décrémenter un compteur. Tous les compteurs sont initialement nuls, sauf celui codant l'entrée.

Remarque 7.5 *C'est donc une machine RAM, mais avec un nombre très réduit d'instructions, et un nombre fini de registres.*

Théorème 7.3 *Toute machine à k-piles peut être simulée par une machine à $k+1$ compteurs.*

Démonstration: L'idée est de voir une pile $w = a_1 a_2 \cdots a_n$ sur l'alphabet Σ de cardinalité $r-1$ comme un entier i en base r : sans perte de généralité,

on peut voir Σ comme $\Sigma = \{0, 1, \cdots, r-1\}$. Le mot w correspond à l'entier $i = a_n r^{n-1} + a_{n-1} r^{n-2} + \cdots a_2 r + a_1$.

On utilise ainsi un compteur i pour chaque pile. Un $k+1$ème compteur, que l'on appellera *compteur supplémentaire*, est utilisé pour ajuster les compteurs et simuler chaque opération (empilement, dépilement, lecture du sommet) sur l'une des piles.

Dépiler correspond à remplacer i par i *div* r, où *div* désigne la division entière : en partant avec le compteur supplémentaire à 0, on décrémente le compteur i de r (en r étapes) et on incrémente le compteur supplémentaire de 1. On répète cette opération jusqu'à ce que le compteur i atteigne 0. On décrémente alors le compteur supplémentaire de 1 en incrémentant le compteur i de 1 jusqu'à ce que le premier soit 0. A ce moment, on lit bien le résultat correct dans le compteur i.

Empiler le symbole a correspond à remplacer i par $i * r + a$: on multiplie d'abord par r : en partant avec le compteur supplémentaire à 0, on décrémente le compteur i de 1 et on incrémente le compteur supplémentaire de r (en r étapes) jusqu'à ce que le compteur i soit à 0. On décrémente alors le compteur supplémentaire de 1 en incrémentant le compteur i de 1 jusqu'à ce que le premier soit 0. A ce moment, on lit $i * r$ dans le compteur i. On incrémente alors le compteur i de a (en a incrémentations).

Lire le sommet d'une pile i correspond à calculer i *mod* r, où i *mod* r désigne le reste de la division euclidienne de i par r : en partant avec le compteur supplémentaire à 0, on décrémente le compteur i de 1 et on incrémente le compteur supplémentaire de 1. Lorsque le compteur i atteint 0 on s'arrête. On décrémente alors le compteur supplémentaire de 1 en incrémentant le compteur i de 1 jusqu'à ce que le premier soit 0. On fait chacune de ces opérations en comptant en parallèle modulo r dans l'état interne de la machine. □

Théorème 7.4 *Toute machine à $k \geq 3$ compteurs se simule par une machine à 2 compteurs.*

Démonstration: Supposons d'abord $k = 3$. L'idée est coder trois compteurs i, j et k par l'entier $m = 2^i 3^j 5^k$. L'un des compteurs stocke cet entier. L'autre compteur est utilisé pour faire des multiplications, divisions, calculs modulo m, pour m valant 2, 3, ou 5.

Pour incrémenter i, j ou k de 1, il suffit de multiplier m par 2, 3 ou 5 en utilisant le principe de la preuve précédente.

Pour tester si i, j ou $k = 0$, il suffit de savoir si m est divisible par 2, 3 ou 5, en utilisant le principe de la preuve précédente.

Pour décrémenter i, j ou k de 1, il suffit de diviser m par 2, 3 ou 5 en utilisant le principe de la preuve précédente.

Pour $k > 3$, on utilise le même principe, mais avec les k premiers nombres premiers au lieu de simplement 2, 3, et 5. □

En combinant les résultats précédents, on obtient :

Corollaire 7.1 *Toute machine de Turing se simule par une machine à 2 compteurs.*

Observons que la simulation est particulièrement inefficace : la simulation d'un temps t de la machine de Turing nécessite un temps exponentiel pour la machine à 2 compteurs.

7.4 Thèse de Church-Turing

7.4.1 Équivalence de tous les modèles considérés

Nous avons jusque-là introduit différents modèles, et montré qu'ils pouvaient tous être simulés par des machines de Turing, ou simuler les machines de Turing.

En fait, tous ces modèles sont équivalents au niveau de ce qu'ils calculent : on a déjà montré que les machines RAM pouvaient simuler les machines de Turing. On peut prouver le contraire. On a montré que les machines à compteurs, et les machines à piles simulaient les machines de Turing. Il est facile de voir que le contraire est vrai : on peut simuler l'évolution d'une machine à piles ou d'une machine à compteurs par machine de Turing. Tous les modèles sont donc bien équivalents, au niveau de ce qu'ils sont capables de calculer.

7.4.2 Thèse de Church-Turing

C'est l'ensemble de ces considérations qui ont donné naissance à la thèse de Church-Turing, exprimée pour la première fois par Stephen Kleene, étudiant de Alonzo Church. Cette thèse affirme que "ce qui est effectivement *calculable* est calculable par une machine de Turing."

Dans cette formulation, la première notion de *calculable* fait référence à une notion donnée intuitive, alors que la seconde notion de calculable signifie "calculable par une machine de Turing", i.e. une notion formelle.

Puisqu'il n'est pas possible de définir formellement la première notion, cette thèse est une thèse au sens philosophique du terme. Il n'est pas possible de la prouver.

La thèse de Church est très largement admise.

7.5 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture de [Hopcroft et al., 2001] en anglais, ou de [Wolper, 2001], [Stern, 1994] [Carton, 2008] en français.

D'autres formalismes équivalents aux machines de Turing existent. En particulier, la notion de fonctions récursives, qui est présentée par exemple dans [Dowek, 2008], [Stern, 1994] ou dans [Cori and Lascar, 1993b].

Bibliographie Ce chapitre a été rédigé à partir de la présentation des machines de Turing dans [Wolper, 2001], et des discussions dans [Hopcroft et al., 2001] pour la partie sur leur programmation. La partie sur les machines RAM est inspirée de [Papadimitriou, 1994] et de [Jones, 1997].

Chapitre 8

Calculabilité

Ce chapitre présente les principaux résultats de la calculabilité. En d'autres termes, nous cherchons à comprendre la puissance des programmes informatiques. On y prouve que certains problèmes peuvent se résoudre informatiquement et que certains problèmes ne peuvent pas l'être. L'objectif est d'explorer les limites de la résolution informatique. En programmation, on s'intéresse généralement à résoudre des problèmes par algorithmes. Le fait que certains problèmes ne peuvent pas être résolus peut paraître surprenant.

Pourquoi s'intéresser à comprendre les problèmes qui ne peuvent pas être résolus ? Premièrement, parce que comprendre qu'un problème ne peut pas être résolu est utile parce que cela signifie que le problème doit être simplifié ou modifié pour pouvoir être résolu. Deuxièmement, parce que tous ces résultats sont culturellement très intéressants et permettent de bien mettre en perspective la programmation, et les limites des dispositifs de calculs, ou de l'automatisation de certaines tâches, comme par exemple la vérification.

8.1 Machines universelles

8.1.1 Interpréteurs

Un certain nombre de ces résultats est la conséquence d'un fait simple, mais qui mène à de nombreuses conséquences : on peut programmer *des interpréteurs*, c'est-à-dire des programmes qui prennent en entrée la description d'un autre programme et qui simulent ce programme.

Exemple 8.1 *Un langage comme JAVA est interprèté : un programme JAVA est compilé en une description dans un codage que l'on appelle* bytecode. *Lorsqu'on cherche à lancer ce programme, l'interpréteur JAVA simule ce bytecode. Ce principe d'interprétation permet à un programme JAVA de fonctionner sur de nombreuses plates-formes et machines directement : seulement l'interpréteur dépend de la machine sur laquelle on exécute le programme. Le même bytecode*

peut être lancé sur toutes les machines. C'est en partie ce qui a fait le succès de JAVA : sa portabilité.

La possibilité de programmer des interpréteurs est donc quelque chose d'extrêmement positif.

Cependant, cela mène aussi à de nombreux résultats négatifs ou paradoxaux à propos de l'impossibilité de résoudre informatiquement certains problèmes, même très simples, comme nous allons le voir dans la suite.

Programmer des interpréteurs est possible dans tous les langages de programmation usuels, en particulier même pour les langages aussi rudimentaires que ceux des machines de Turing.

Remarque 8.1 *Nous n'allons pas parler de JAVA dans ce qui suit, mais plutôt de programmes de machines de Turing. Raisonner sur JAVA (ou tout autre langage) ne ferait que compliquer la discussion, sans changer le fond des arguments.*

Commençons par nous persuader que l'on peut réaliser des interpréteurs pour les machines de Turing : dans ce contexte, on appelle cela des *machines de Turing universelles*.

8.1.2 Codage d'une machine de Turing

Il nous faut fixer une représentation des programmes des machines de Turing. Le codage qui suit n'est qu'une convention. Tout autre codage qui garantit la possibilité de décoder (par exemple dans l'esprit du lemme 8.1 pour le déplacement) ferait l'affaire.

Définition 8.1 (Codage d'une machine de Turing) *Soit M une machine de Turing sur l'alphabet $\Sigma = \{0, 1\}$.*

Selon la définition 7.1, M correspond à un 8-uplet

$$M = (Q, \Sigma, \Gamma, \mathbf{B}, \delta, q_0, q_a, q_r) :$$

- *Q est un ensemble fini, dont on peut renommer les états $Q = \{q_1, q_2, \cdots, q_z\}$, avec la convention que $q_1 = q_0$, $q_2 = q_a$, $q_3 = q_r$;*
- *Γ est un ensemble fini, dont on peut renommer les états $\Gamma = \{X_1, X_2, \cdots, X_s\}$, avec la convention que X_s est le symbole $\mathbf{B}$, et que X_1 est le symbole 0 de Σ, et que X_2 est le symbole 1 de Σ.*

Pour $m \in \{\leftarrow, |, \rightarrow\}$, définissons $\langle m \rangle$ de la façon suivante : $\langle \leftarrow \rangle = 1$, $\langle | \rangle = 2$, $\langle \rightarrow \rangle = 3$.

On peut alors coder la fonction de transition δ de la façon suivante : supposons que l'une des règles de transition soit $\delta(q_i, X_j) = (q_k, X_l, m)$: le codage de cette règle est le mot $0^i 1 0^j 1 0^k 1 0^l 1 1^{\langle m \rangle}$ sur l'alphabet $\{0, 1\}$. Observons que puisque tous les entiers i, j, k, l sont non nuls, il n'y pas de 1 consécutif dans ce mot.

Un codage, noté $\langle M \rangle$, de la machine de Turing M est un mot de la forme

$$C_1 11 C_2 11 C_3 \cdots C_{n-1} 11 C_n,$$

où chaque C_i est le codage d'une des règles de transition de δ.

Remarque 8.2 *A une machine de Turing peuvent correspondre plusieurs codages : on peut en particulier permuter les $(C_i)_i$, ou les états, etc....*

Le seul intérêt de ce codage est qu'il est décodable : on peut retrouver chacun des ingrédients de la description d'une machine de Turing à partir du codage de la machine.

Par exemple, si l'on veut retrouver le déplacement m pour une transition donnée :

Lemme 8.1 (Décodage du codage) *On peut construire une machine de Turing M à quatre rubans, telle que si l'on met un codage $\langle M' \rangle$ d'une machine M' sur son premier ruban, 0^i sur son second, et 0^j sur son troisième, M produit sur son quatrième ruban le codage $\langle m \rangle$ du déplacement $m \in \{\leftarrow, |, \rightarrow\}$ tel que $\delta(q_i, X_j) = (q_k, X_l, m)$ où δ est la fonction de transition de la machine de Turing M'.*

Démonstration (principe): On construit une machine qui parcourt le codage de M' jusqu'à trouver le codage de la transition correspondante, et qui lit dans le codage de cette transition la valeur de m désirée. □

8.1.3 Existence d'une machine de Turing universelle

On peut se convaincre que l'on peut réaliser un interpréteur, c'est-à-dire ce que l'on appelle *une machine de Turing universelle* dans le contexte des machines de Turing.

Théorème 8.1 (Existence d'une machine de Turing universelle) *Il existe une machine de Turing M_{univ} à trois rubans telle que si l'on place :*

- *le codage $\langle A \rangle$ d'une machine de Turing A sur le premier ruban ;*
- *un mot w sur l'alphabet $\Sigma = \{0, 1\}$ sur le second ;*

alors M_{univ} simule la machine de Turing A sur l'entrée w en utilisant son troisième ruban.

Démonstration: La machine M_{univ} simule transition par transition la machine A sur l'entrée w sur son second ruban : M_{univ} utilise le troisième ruban pour y stocker 0^q où q code l'état de la machine A à la transition que l'on est en train de simuler : initialement, ce ruban contient 0, le codage de q_0.

Pour simuler chaque transition de A, M_{univ} lit la lettre X_j en face de sa tête de lecture sur le second ruban, va lire dans le codage $\langle A \rangle$ sur le premier ruban la valeur de q_k, X_l et m, pour la transition $\delta(q_i, X_j) = (q_k, X_l, m)$ de A, où 0^i est le contenu du troisième ruban. M_{univ} écrit alors X_l sur son second ruban, écrit

q_k sur son troisième ruban, et déplace la tête de lecture selon le déplacement m sur le second ruban. □

Dans la preuve nous avons utilisé une machine à plusieurs rubans, mais cela était uniquement pour simplifier la description. On peut fixer une façon de coder deux mots sur Σ par un unique mot sur un alphabet plus grand, de telle sorte que l'on soit capable de retrouver ces mots.

Par exemple, on peut décider de coder les mots $w_1 \in \Sigma^*$ et $w_2 \in \Sigma^*$ par le mot $w_1 \# w_2$ sur l'alphabet $\Sigma \cup \{\#\}$. Une machine de Turing peut alors retrouver à partir de ce mot à la fois w_1 et w_2.

On peut même recoder le mot obtenu en binaire lettre par lettre pour obtenir une façon de coder deux mots sur $\Sigma = \{0, 1\}$ par un unique mot sur $\Sigma = \{0, 1\}$: par exemple, si $w_1 \# w_2$ s'écrit lettre à lettre $a_1 a_2 \cdots a_n$ sur l'alphabet $\Sigma \cup \{\#\}$, on définit $\langle w1, w_2 \rangle$ comme le mot $e(a_1)e(a_2) \ldots e(a_n)$ où $e(0) = 00$, $e(1) = 01$ et $e(\#) = 10$. Ce codage est encore décodable : à partir de $\langle w1, w_2 \rangle$, une machine de Turing peut reconstruire w_1 et w_2.

Avec ces considérations, on peut reformuler le théorème précédent en parlant de machines avec un unique ruban :

Théorème 8.2 (Existence d'une machine de Turing universelle) *Il existe une machine de Turing M_{univ} telle, que sur l'entrée $\langle \langle A \rangle, w \rangle$ où :*

1. *$\langle A \rangle$ est le codage d'une machine de Turing A ;*
2. *$w \in \{0, 1\}^*$;*

M_{univ} simule la machine de Turing A sur l'entrée w.

Démonstration: Utiliser une machine de Turing avec un unique ruban qui simule la machine à plusieurs rubans de la preuve précédente. □

8.1.4 Premières conséquences

Voici une première utilisation de l'existence d'interpréteurs : la preuve de la proposition 7.3, c'est-à-dire la preuve qu'une machine de Turing non déterministe M peut être simulée par une machine de Turing déterministe.

La preuve est la suivante : la relation de transition de la machine non déterministe M est nécessairement de *degré de non determinisme* borné : c'est-à-dire, le nombre

$$r = max_{q \in Q, a \in \Gamma} |\{((q, a), (q', a', m)) \in \delta\}|$$

est fini ($|.|$ désigne le cardinal).

Supposons que pour chaque paire (q, a) on numérote les choix parmi la relation de transition de la machine non déterministe M de 1 à (au plus) r. A ce moment-là, pour décrire les choix non déterministes effectués dans un calcul jusqu'au temps t, il suffit de donner une suite de t nombres entre 1 et (au plus) r.

On peut alors construire une machine de Turing (déterministe) qui simule la machine M de la façon suivante : pour $t = 1, 2, \cdots$, elle énumère toutes les suites de longueur t d'entiers entre 1 et r. Pour chacune de ces suites, elle simule

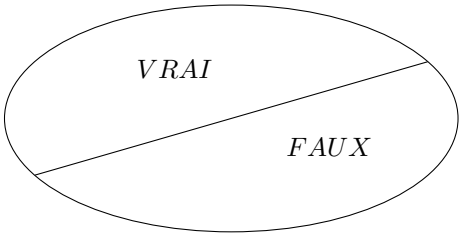


FIG. 8.1 – Problèmes de décision : dans un problème de décision, on a une propriété qui est soit vraie soit fausse pour chaque instance. L'objectif est de distinguer les instances positives E^+ (où la propriété est vraie) des instances négatives $E \backslash E^+$ (où la propriété est fausse).

t étapes de la machine M en utilisant les choix donnés par la suite pour résoudre chaque choix non déterministe de M. La machine s'arrête dès qu'elle trouve t et une suite telle que M atteint une configuration acceptante.

8.2 Langages et problèmes décidables

Une fois ces résultats établis sur l'existence de machines universelles (interpréteurs), nous allons dans cette section essentiellement présenter quelques définitions.

Tout d'abord, il nous faut préciser que l'on s'intéresse dorénavant dans ce chapitre uniquement aux problèmes dont la réponse est soit *vraie* soit *fausse*, et ce, essentiellement pour simplifier la discussion : voir la figure 8.1.

8.2.1 Problèmes de décision

Définition 8.2 *Un* problème de décision $\mathcal{P}$ *est la donnée d'un ensemble* E*,* que l'on appelle l'ensemble des instances*, et d'un sous-ensemble* E^+ *de* E*, que l'on appelle* l'ensemble des instances positives*.*

La question à laquelle on s'intéresse est de construire (si cela est possible) un algorithme qui décide si une instance donnée est positive ou non. On va formuler les problèmes de décision systématiquement sous la forme :

Définition 8.3 (Nom du problème)
Donnée*: Une instance (i.e. un élément de* E*).*
Réponse*: Décider une certaine propriété (i.e. si cet élément est dans* E^+*).*

On peut par exemple considérer les problèmes suivants :

Exemple 8.2 ***Définition 8.4*** **(**NOMBRE PREMIER**)**
Donnée*: Un entier naturel* n*.*
Réponse*: Décider si* n *est premier.*

Exemple 8.3 ***Définition 8.5 (***CODAGE***)***
Donnée*: Un mot w.*
Réponse*: Décider si w correspond au codage $\langle M \rangle$ d'une certaine machine de Turing M.*

Exemple 8.4 ***Définition 8.6 (***REACH***)***
Donnée*: Un triplet constitué d'un graphe G, d'un sommet u et d'un sommet v du graphe.*
Réponse*: Décider s'il existe un chemin entre u et v dans le graphe G.*

8.2.2 Problèmes versus Langages

On utilise indifféremment la terminologie problème ou langage dans tout ce qui suit.

Remarque 8.3 (Problèmes vs langages) *Cela découle des considérations suivantes : à un problème (de décision) est associé un langage et réciproquement.*

En effet, à un problème est associé généralement implicitement une fonction de codage (par exemple pour les graphes, une façon de coder les graphes) qui permet de coder les instances, c'est-à-dire les éléments de E, par un mot sur un certain alphabet Σ. On peut donc voir E comme un sous-ensemble de Σ^*, où Σ est un certain alphabet. On peut alors voir un problème comme un langage : au problème de décision $\mathcal{P}$, on associe le langage $L(\mathcal{P})$ correspondant à l'ensemble des mots codant un instance de E, qui appartient à E^+ :

$$L(\mathcal{P}) = \{w | w \in E^+\}.$$

Réciproquement, on peut voir tout langage L comme un problème de décision : en le formulant de cette façon.

Définition 8.7 (Problème associé au langage** L**)
Donnée*: Un mot w.*
Réponse*: Décider si $w \in L$.*

8.2.3 Langages décidables

On rappelle la définition suivante :

Définition 8.8 (Langage décidable) *Un langage $L \subset M^*$ est dit* décidable *s'il est décidé par une certaine machine de Turing.*

Un langage ou un problème décidable est aussi dit *récursif*. Un langage qui n'est pas décidable est dit *indécidable*.

On note R pour la classe des langages et des problèmes *décidables*.

Par exemple :

Proposition 8.1 *Les problèmes de décision* NOMBRE PREMIER, CODAGE *et* REACH *sont décidables.*

La preuve consiste à construire une machine de Turing qui reconnaît respectivement si son entrée est un nombre premier, le codage d'une machine de Turing, où une instance positive de REACH, ce que nous laissons en exercice de programmation élémentaire à notre lecteur.

8.3 Indécidabilité

Dans cette section, nous prouvons un des théorèmes les plus importants philosophiquement dans la théorie de la programmation : l'existence de problèmes qui ne sont pas décidables.

8.3.1 Premières considérations

Observons tout d'abord, que cela peut s'établir simplement.

Théorème 8.3 *Il existe des problèmes de décision qui ne sont pas décidables.*

Démonstration: On a vu que l'on pouvait coder une machine de Turing par un mot fini sur l'alphabet $\Sigma = \{0, 1\}$: voir la définition 8.1. Il y a donc un nombre dénombrable de machines de Turing.

Par contre, il y a un nombre non dénombrable de langages sur l'alphabet $\Sigma = \{0, 1\}$: cela peut se prouver en utilisant un argument de diagonalisation comme dans le premier chapitre.

Il y a donc des problèmes qui ne correspondent pas à aucune machine de Turing (et il y en a même un nombre non dénombrable). □

Le défaut d'une preuve comme celle-là est évidemment qu'elle ne dit rien sur les problèmes en question. Est-ce qu'ils sont ésotériques et d'aucun intérêt sauf pour le théoricien ?

Malheureusement, même des problèmes simples et naturels s'avèrent ne pas pouvoir se résoudre par algorithme.

8.3.2 Est-ce grave ?

Par exemple, dans l'un des problèmes indécidables, on se donne un programme et une spécification de ce que le programme est censé faire (par exemple trier des nombres), et l'on souhaite vérifier que le programme satisfait sa spécification.

On pourrait espérer que le processus de la vérification puisse s'automatiser, c'est-à-dire que l'on puisse construire un algorithme qui vérifie si un programme satisfait sa spécification. Malheureusement, cela est impossible : le problème général de la vérification ne peut pas se résoudre par ordinateur.

On rencontrera d'autres problèmes indécidables dans ce chapitre. Notre objectif est de faire sentir à notre lecteur le type de problèmes qui sont indécidables, et de comprendre les techniques permettant de prouver qu'un problème ne peut pas être résolu informatiquement.

8.3.3 Un premier problème indécidable

On va utiliser un argument de *diagonalisation*, c'est-à-dire un procédé analogue à la diagonalisation de Cantor qui permet de montrer que l'ensemble des parties de $\mathbb{N}$ n'est pas dénombrable : voir le premier chapitre.

Remarque 8.4 *Derrière l'argument précédent sur le fait qu'il y a un nombre non dénombrable de langages sur l'alphabet $\Sigma = \{0, 1\}$ se cachait déjà une diagonalisation. On fait ici une diagonalisation plus explicite, et constructive.*

On appelle *langage universel*, appelé encore *problème de l'arrêt des machines de Turing*, le problème de décision suivant :

Définition 8.9 (HALTING-PROBLEM)
Donnée: *Le codage $\langle M \rangle$ d'une machine de Turing M et un mot w.*
Réponse: *Décider si la machine M accepte le mot w.*

Remarque 8.5 *On peut aussi voir ce problème de la façon suivante : on se donne une paire $\langle \langle M \rangle, w \rangle$, où $\langle M \rangle$ est le codage d'une machine de Turing M, et w est un mot, et l'on souhaite décider si la machine M accepte le mot w.*

Théorème 8.4 *Le problème* HALTING − PROBLEM *n'est pas décidable.*

Démonstration: On prouve le résultat par un raisonnement par l'absurde. Supposons que HALTING − PROBLEM soit décidé par une machine de Turing A.

On construit alors une machine de Turing B qui fonctionne de la façon suivante :

- B prend en entrée un mot $\langle C \rangle$ codant une machine de Turing C ;
- B appelle la machine de Turing A sur la paire $\langle \langle C \rangle, \langle C \rangle \rangle$ (c'est-à-dire sur l'entrée constituée du codage de la machine de Turing C, et du mot w correspondant aussi à ce même codage) ;
- Si la machine de Turing A :
  - accepte ce mot, B refuse ;
  - refuse ce mot, B accepte.

On montre qu'il y a une contradiction, en appliquant la machine de Turing B sur le mot $\langle B \rangle$, c'est-à-dire sur le mot codant la machine de Turing B.

- Si B accepte $\langle B \rangle$, cela signifie, par définition de HALTING − PROBLEM et de A, que A accepte $\langle \langle B \rangle, \langle B \rangle \rangle$. Mais si A accepte ce mot, B est construit pour refuser son entrée $\langle B \rangle$. Contradiction.
- Si B refuse $\langle B \rangle$, cela signifie, par définition de HALTING − PROBLEM et de A, que A refuse $\langle \langle B \rangle, \langle B \rangle \rangle$. Mais si A refuse ce mot, B est construit pour accepter son entrée $\langle B \rangle$. Contradiction.

□

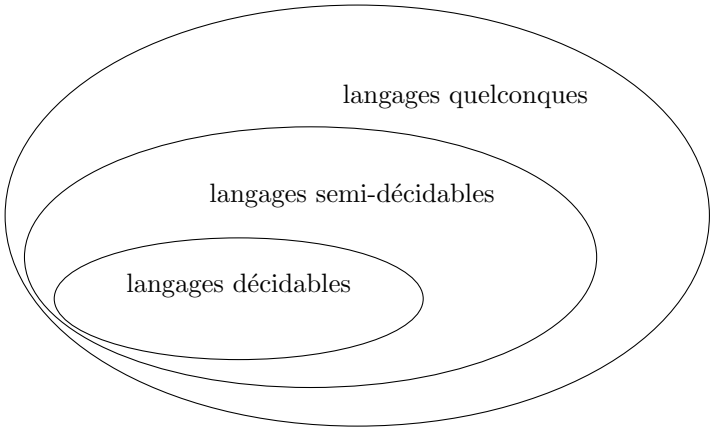


FIG. 8.2 – Inclusions entre classes de langages .

8.3.4 Problèmes semi-décidables

Le problème HALTING − PROBLEM est toutefois semi-décidable dans le sens suivant :

Définition 8.10 (Langage semi-décidable) *Un langage $L \subset M^*$ est dit* semi-décidable *s'il correspond à l'ensemble des mots acceptés par une machine de Turing M.*

Corollaire 8.1 *Le langage universel* HALTING − PROBLEM *est semi-décidable.*

Démonstration: Pour savoir si on doit accepter une entrée correspondante au codage $\langle M \rangle$ d'une machine de Turing M et au mot w, il suffit de simuler la machine de Turing M sur l'entrée w. On arrête la simulation et on accepte si l'on détecte dans cette simulation que la machine de Turing M atteint un état accepteur. Sinon, on simule M pour toujours. □

Un langage semi-décidable est aussi dit *récursivement énumérable*.

On note RE la classe des langages et des problèmes semi-décidables : voir la figure 8.2.

Corollaire 8.2 $\mathrm{R} \subsetneq \mathrm{RE}$.

Démonstration: L'inclusion est par définition. Puisque HALTING − PROBLEM est dans RE et n'est pas dans R, l'inclusion est stricte. □

8.3.5 Un problème qui n'est pas semi-décidable

Commençons par établir le résultat fondamental suivant :

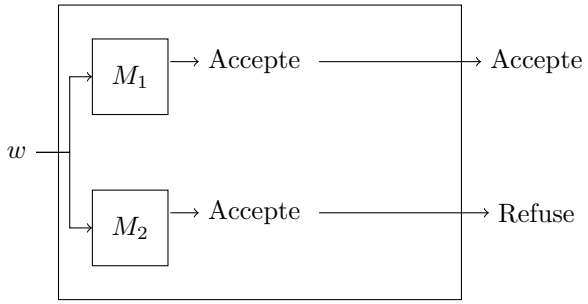


FIG. 8.3 – Illustration de la preuve du théorème 8.5.

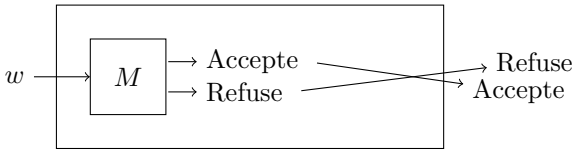


FIG. 8.4 – Construction d'une machine de Turing acceptant le complément d'un langage décidable.

Théorème 8.5 *Un langage est décidable si et seulement s'il est semi-décidable et son complémentaire aussi.*

Remarque 8.6 *Ce résultat justifie la terminologie de* semi-décidable*, puisqu'un langage qui est semi-décidable et dont le complémentaire l'est aussi est décidable.*

Démonstration: sens $\Leftarrow$. Supposons que L soit semi-décidable et son complémentaire aussi. Il existe une machine de Turing M_1 qui termine en acceptant sur L, et une machine de Turing M_2 qui termine en acceptant sur son complémentaire. On construit une machine de Turing M qui, sur une entrée w, simule en parallèle M_1 et M_2, (c'est-à-dire que l'on simule t étapes de M_1 sur w, puis on simule t étapes de M_2 sur w, pour $t = 1, 2, \cdots,$) jusqu'à ce que l'une des deux termine : voir la figure 8.3. Si M_1 termine, la machine de Turing M accepte. Si c'est M_2, la machine M refuse. La machine de Turing M que l'on vient de décrire décide L.

sens $\Rightarrow$. Par définition, un langage décidable est semi-décidable. En inversant dans la machine de Turing l'état d'acceptation et de refus, son complémentaire est aussi décidable (voir la figure 8.4) et donc aussi semi-décidable. □

On considère alors le complémentaire du problème HALTING − PROBLEM, que l'on va noter $\overline{\text{HALTING} - \text{PROBLEM}}$.

Définition 8.11 ($\overline{\text{HALTING} - \text{PROBLEM}}$)
Donnée*: Le codage $\langle M \rangle$ d'une machine de Turing M et un mot w.*

***Réponse**: Décider si la machine M n'accepte pas le mot w.*

Corollaire 8.3 *Le problème* $\overline{\text{HALTING} - \text{PROBLEM}}$ *n'est pas semi-décidable.*

Démonstration: Sinon, par le théorème précédent, le problème de décision HALTING – PROBLEM serait décidable. □

8.3.6 Sur la terminologie utilisée

Un langage décidable est aussi appelé un langage *récursif* : la terminologie de *récursif* fait référence à la notion de fonction récursive : voir par exemple le cours [Dowek, 2008] qui présente la calculabilité par l'intermédiaire des fonctions récursives.

La notion de *énumérable* dans *récursivement énumérable* s'explique par le résultat suivant.

Théorème 8.6 *Un langage $L \subset M^*$ est récursivement énumérable si et seulement si l'on peut produire une machine de Turing qui affiche un à un (énumère) tous les mots du langage L.*

Démonstration: sens $\Rightarrow$. Supposons que L soit récursivement énumérable. Il existe une machine de Turing A qui termine en acceptant sur les mots de L.

L'ensemble des couples (t, w), où t est un entier, w est un mot, est dénombrable. On peut se convaincre assez facilement qu'il est en fait effectivement dénombrable : on peut construire une machine de Turing qui produit le codage $\langle t, w\rangle$ de tous les couples (t, w). Par exemple, on considère une boucle qui pour $t = 1, 2, \cdots$ jusqu'à l'infini, considère tous les mots w de longueur inférieure ou égale à t, et produit pour chacun le couple $\langle t, w\rangle$.

Considérons une machine de Turing B qui en plus, pour chaque couple produit (t, w), simule en outre t étapes de la machine A. Si la machine A termine et accepte en exactement t étapes, B affiche alors le mot w. Sinon B n'affiche rien pour ce couple.

Un mot du langage L, est accepté par A en un certain temps t. Il sera alors écrit par B lorsque celui-ci considérera le couple (t, w). Clairement, tout mot w affiché par B est accepté par A, et donc est un mot de L.

sens $\Leftarrow$. Réciproquement, si l'on a une machine de Turing B qui énumère tous les mots du langage L, alors on peut construire une machine de Turing A, qui étant donné un mot w, simule B, et à chaque fois que B produit un mot compare ce mot au mot w. S'ils sont égaux, alors A s'arrête et accepte. Sinon, A continue à jamais.

Par construction, sur une entrée w, A termine et accepte si w se trouve parmi les mots énumérés par B, c'est-à-dire si $w \in L$. Si w ne se trouve pas parmi ces mots, par hypothèse, $w \notin L$, et donc par construction, A n'acceptera pas w. □

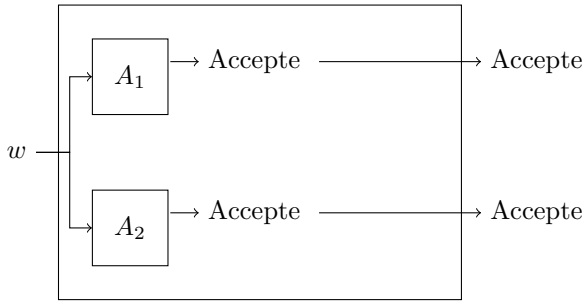


FIG. 8.5 – Construction d'une machine de Turing acceptant $L_1 \cup L_2$.

8.3.7 Propriétés de clôture

Théorème 8.7 *L'ensemble des langages semi-décidables est clos par union et par intersection : autrement dit, si L_1 et L_2 sont semi-décidables, alors $L_1 \cup L_2$ et $L_1 \cap L_2$ le sont.*

Démonstration: Supposons que L_1 corresponde à $L(A_1)$ pour une machine de Turing A_1 et L_2 à $L(A_2)$ pour une machine de Turing A_2. Alors $L_1 \cup L_2$ correspond à $L(A)$ pour la machine de Turing A qui simule en parallèle A_1 et A_2 et qui termine et accepte dès que l'une des deux machines de Turing A_1 et A_2 termine et accepte : voir la figure 8.5.

$L_1 \cap L_2$ correspond à $L(A)$ pour la machine de Turing A qui simule en parallèle A_1 et A_2 et qui termine dès que les deux machines de Turing A_1 et A_2 terminent et acceptent.

□

Théorème 8.8 *L'ensemble des langages décidables est clos par union, intersection, et complément : autrement dit, si L_1 et L_2 sont décidables, alors $L_1 \cup L_2$, $L_1 \cap L_2$, et L_1^c le sont.*

Démonstration: On a déjà utilisé la clôture par complémentaire : en inversant dans la machine de Turing l'état d'acceptation et de refus, le complémentaire d'un langage décidable est aussi décidable (voir la figure 8.4).

Il reste à montrer qu'avec les hypothèses, les langages $L_1 \cup L_2$ et $L_1 \cap L_2$ sont décidables. Mais cela est clair par le théorème précédent et le fait qu'un ensemble est décidable si et seulement si il est semi-décidable et son complémentaire aussi, en utilisant les lois de Morgan (le complémentaire d'une union est l'intersection des complémentaires, et inversement) et la stabilité par complémentaire des langages décidables. □

8.4 Autres problèmes indécidables

Une fois que nous avons obtenu un premier problème indécidable, nous allons en obtenir d'autres.

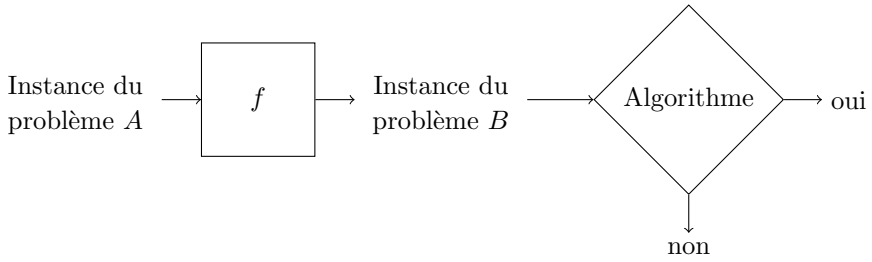


FIG. 8.6 – Réduction du problème A vers le problème B. Si l'on peut résoudre le problème B, alors on peut résoudre le problème A. Le problème A est donc plus facile que le problème B, noté $A \leq_m B$.

8.4.1 Réductions

Nous connaissons deux langages indécidables, HALTING – PROBLEM et son complémentaire. Notre but est maintenant d'en obtenir d'autres, et de savoir comparer les problèmes. Nous introduisons pour cela la notion de *réduction*.

Tout d'abord, nous pouvons généraliser la notion de *calculable* aux fonctions et pas seulement aux langages et problèmes de décision.

Définition 8.12 (Fonction calculable) *Soient Σ et Σ' deux alphabets. Une fonction $f : \Sigma^* \to \Sigma'^*$ est* calculable *s'il existe une machine de Turing A, qui travaille sur l'alphabet $\Sigma \cup \Sigma'$, telle que pour tout mot w, A avec l'entrée w écrit en sortie $f(w)$ (puis termine et accepte).*

On peut se convaincre facilement que la composée de deux fonctions calculables est calculable.

Cela nous permet d'introduire une notion de réduction entre problèmes : l'idée est que si A se réduit à B, alors le problème A est plus facile que le problème B, ou si l'on préfère, le problème B est plus difficile que le problème A : voir la figure 8.6 et la figure 8.7.

Définition 8.13 (Réduction) *Soient A et B deux problèmes d'alphabet respectifs M_A et M_B. Une* réduction de A vers B *est une fonction $f : M_A^* \to M_B^*$ calculable telle que $w \in A$ ssi $f(w) \in B$. On note $A \leq_m B$ lorsque A se réduit à B.*

Cela se comporte comme on le souhaite : un problème est aussi facile (et difficile) que lui-même, et la relation "être plus facile que" est transitive. En d'autres termes :

Théorème 8.9 *$\leq_m$ est un préordre :*

1. *$L \leq_m L$;*
2. *$L_1 \leq_m L_2$, $L_2 \leq_m L_3$ impliquent $L_1 \leq_m L_3$.*

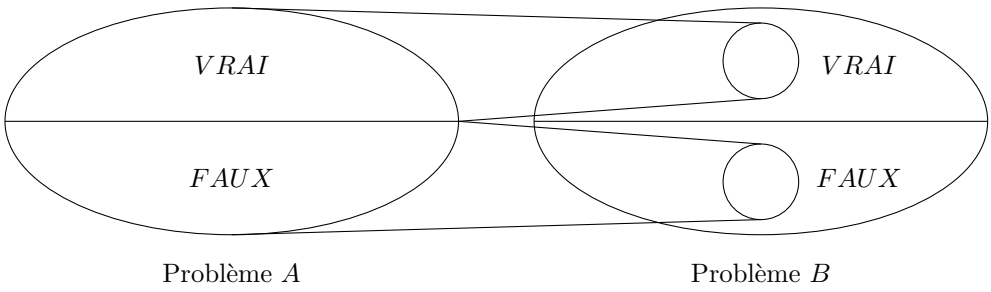


FIG. 8.7 – Les réductions transforment des instances positives en instances positives, et négatives en négatives.

Démonstration: Considérer la fonction identité comme fonction f pour le premier point.

Pour le second point, supposons $L_1 \leq_m L_2$ via la réduction f, et $L_2 \leq_m L_3$ via la réduction g. On a $x \in L_1$ ssi $g(f(x)) \in L_2$. Il suffit alors de voir que $g \circ f$, en temps que composée de deux fonctions calculables est calculable. □

Remarque 8.7 *Il ne s'agit pas d'un ordre, puisque $L_1 \leq_m L_2$, $L_2 \leq_m L_1$ n'implique pas $L_1 = L_2$. Il est en fait naturel d'introduire le concept suivant : deux problèmes L_1 et L_2 sont équivalents, noté $L_1 \equiv L_2$, si $L_1 \leq_m L_2$ et si $L_2 \leq_m L_1$. On a alors $L_1 \leq_m L_2$, $L_2 \leq_m L_1$ impliquent $L_1 \equiv L_2$.*

Intuitivement, si un problème est plus facile qu'un problème décidable, alors il est décidable. Formellement :

Proposition 8.2 (Réduction) *Si $A \leq_m B$, et si B est décidable alors A est décidable .*

Démonstration: A est décidé par la machine de Turing qui, sur une entrée w, calcule $f(w)$, puis simule la machine de Turing qui décide B sur l'entrée $f(w)$. Puisqu'on a $w \in A$ si et seulement si $f(w) \in B$, la machine de Turing est correcte. □

Proposition 8.3 (Réduction) *Si $A \leq_m B$, et si A est indécidable, alors B est indécidable.*

Démonstration: Il s'agit de la contraposée de la proposition précédente. □

8.4.2 Quelques autres problèmes indécidables

Cette idée va nous permettre d'obtenir immédiatement la preuve de l'indécidabilité de plein d'autres problèmes.

Par exemple, le fait qu'il n'est pas possible de déterminer algorithmiquement si une machine de Turing accepte au moins une entrée :

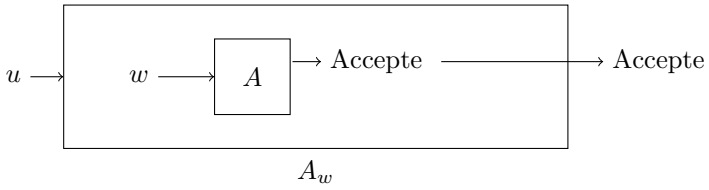


FIG. 8.8 – Illustration de la machine de Turing utilisée dans la preuve de la proposition 8.4.

Définition 8.14 (Problème $L_\emptyset$)
Donnée: *Le codage $\langle M \rangle$ d'une machine de Turing M.*
Réponse: *Décider si $L(M) \neq \emptyset$.*

Proposition 8.4 *Le problème Problème $L_\emptyset$ est indécidable.*

Démonstration: On construit une réduction de HALTING − PROBLEM vers $L_\emptyset$: pour toute paire $\langle \langle A \rangle, w \rangle$, on considère la machine de Turing A_w définie de la manière suivante (voir la figure 8.8) :
- A_w prend en entrée un mot u ;
- A_w simule A sur w ;
- Si A accepte w, alors A_w accepte.

La fonction f qui à $\langle \langle A \rangle, w \rangle$ associe $\langle A_w \rangle$ est bien calculable. De plus on a $\langle \langle A \rangle, w \rangle \in$ HALTING − PROBLEM si et seulement si $L(A_w) \neq \emptyset$, c'est-à-dire $\langle A_w \rangle \in L_\emptyset$: en effet, A_w accepte soit tous les mots (et donc le langage correspondant n'est pas vide) si A accepte w, soit accepte aucun mot (et donc le langage correspondant est vide) sinon. □

Définition 8.15 (Problème $L_{\neq}$)
Donnée: *Le codage $\langle A \rangle$ d'une machine de Turing A et le codage $\langle A' \rangle$ d'une machine de Turing A'.*
Réponse: *Déterminer si $L(A) \neq L(A')$.*

Proposition 8.5 *Le problème Problème $L_{\neq}$ est indécidable.*

Démonstration: On construit une réduction de $L_\emptyset$ vers $L_{\neq}$. On considère une machine de Turing fixe B qui accepte le langage vide : prendre par exemple une machine de Turing B qui rentre immédiatement dans une boucle sans fin. La fonction f qui à $\langle A \rangle$ associe la paire (A, B) est bien calculable. De plus on a $\langle A \rangle \in L_\emptyset$ si et seulement si $L(A) \neq \emptyset$ si et seulement si $\langle A, B \rangle \in L_{\neq}$. □

8.4.3 Théorème de Rice

Les deux résultats précédents peuvent être vus comme les conséquences d'un résultat très général qui affirme que toute propriété non triviale des algorithmes est indécidable.

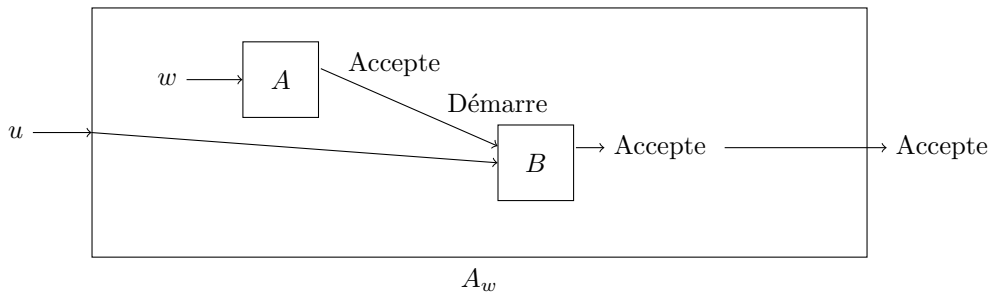


FIG. 8.9 – Illustration de la machine de Turing utilisée dans la preuve du théorème de Rice.

Théorème 8.10 (Théorème de Rice) *Toute propriété non triviale des langages semi-décidables est indécidable.*

Autrement dit, soit une propriété P des langages semi-décidables non triviale, c'est-à-dire telle qu'il y a au moins une machine de Turing M telle que $L(M)$ satisfait P et une machine de Turing M telle que $L(M)$ ne satisfait pas P.

Alors le problème de décision L_P :

***Donnée**: Le codage $\langle M \rangle$ d'une machine de Turing M ;*

***Réponse**: Décider si $L(M)$ vérifie la propriété P ;*

est indécidable.

Remarque 8.8 *Remarquons que si une propriété P est triviale, L_P est décidable trivialement : construire une machine de Turing qui ne lit même pas son entrée et qui accepte (respectivement : refuse).*

Démonstration: Il nous faut démontrer que le problème de décision L_P est indécidable.

Quitte à remplacer P par sa négation, on peut supposer que le langage vide ne vérifie pas la propriété P (prouver l'indécidabilité de L_p est équivalent à prouver l'indécidabilité de son complémentaire). Puisque P est non triviale, il existe un moins une machine de Turing B avec $L(B)$ qui vérifie P.

On construit une réduction de $\text{HALTING} - \text{PROBLEM}$ vers le langage L_P. Étant donnée une paire $\langle \langle A \rangle, w \rangle$, on considère la machine de Turing A_w définie de la façon suivante (voir la figure 8.9) :

- A_w prend en entrée un mot u ;
- Sur le mot u, A_w simule A sur le mot w ;
- Si A accepte w, alors A_w simule B sur le mot u : A_w accepte si seulement si B accepte u.

Autrement dit, A_w accepte, si et seulement si A accepte w et si B accepte u. Si w est accepté par A, alors $L(A_w)$ vaut $L(B)$, et donc vérifie la propriété P. Si

w n'est pas accepté par A, alors $L(A_w) = \emptyset$, et donc ne vérifie pas la propriété P.

La fonction f qui à $\langle\langle A\rangle, w\rangle$ associe $\langle A_w\rangle$ est bien calculable. □

8.4.4 Le drame de la vérification

Autrement dit :

Corollaire 8.4 *Il n'est pas possible de construire un algorithme qui prendrait en entrée un programme, et sa spécification, et qui déterminerait si le programme satisfait sa spécification.*

Et cela, en fait, même si l'on fixe la spécification à une propriété P (dès que la propriété P n'est pas triviale), par la théorème de Rice.

Par les discussions du chapitre précédent, cela s'avère vrai même pour des systèmes extrémements rudimentaires. Par exemple :

Corollaire 8.5 *Il n'est pas possible de construire un algorithme qui prendrait en entrée la description d'un système, et sa spécification, et qui déterminerait si le système satisfait sa spécification.*

Et cela, en fait, même si l'on fixe la spécification à une propriété P (dès que la propriété P n'est pas triviale), et même pour des systèmes aussi simples que les machines à 2-compteurs, par la théorème de Rice, et les résultats de simulation du chapitre précédent.

8.4.5 Notion de complétude

Notons que l'on peut aussi introduire une notion de complétude.

Définition 8.16 (RE**-complétude)** *Un problème A est dit* RE*-complet, si :*

1. *il est récursivement énumérable ;*
2. *tout autre problème récursivement énumérable B est tel que $B \leq_m A$.*

Autrement dit, un problème RE-complet est maximal pour $\leq_m$ parmi les problèmes de la classe RE.

Théorème 8.11 *Le problème* $\mathrm{HALTING} - \mathrm{PROBLEM}$ *est* RE*-complet.*

Démonstration: $\mathrm{HALTING} - \mathrm{PROBLEM}$ est semi-décidable. Maintenant, soit L un langage semi-décidable. Par définition, il existe une machine de Turing A qui accepte L. Considérons la fonction f qui à w associe le mot $\langle\langle A\rangle, w\rangle$. On a $w \in L$ si et seulement si $f(w) \in \mathrm{HALTING} - \mathrm{PROBLEM}$, et donc $L \leq_m \mathrm{HALTING} - \mathrm{PROBLEM}$. □

8.5 Problèmes indécidables naturels

On peut objecter que les problèmes précédents, relatifs aux algorithmes sont "artificiels", dans le sens où ils parlent de propriétés d'algorithmes, les algorithmes ayant eux-même été définis par la théorie de la calculabilité.

Il est difficile de définir formellement ce qu'est un problème *naturel*, mais on peut au moins dire qu'un problème qui a été discuté avant l'invention de la théorie de la calculabilité est (plus) naturel.

8.5.1 Le dixième problème de Hilbert

C'est clairement le cas du dixième problème identifié par Hilbert parmi les problèmes intéressants pour le 20ème siècle en 1900 : peut-on déterminer si une équation polynomiale à coefficients entiers possède une solution entière.

Définition 8.17 (Dixième problème de Hilbert)
Donnée: *Un polynôme $P \in \mathbb{N}[X_1, \cdots, X_n]$ à coefficients entiers.*
Réponse: *Décider s'il possède une racine entière.*

Théorème 8.12 *Le problème Dixième problème de Hilbert est indécidable.*

La preuve de ce résultat, due à Matiyasevich [Matiyasevich, 1970]est hors de l'ambition de ce document.

8.5.2 Le problème de la correspondance de Post

La preuve de l'indécidabilité du problème de la correspondance de Post est plus facile, même si nous ne la donnerons pas ici. On peut considérer ce problème comme "naturel", dans le sens où il ne fait pas référence directement à la notion d'algorithme, ou aux machines de Turing :

Définition 8.18 (Problème de la correspondance de Post**)**
Donnée: *Une suite $(u_1, v_1), \cdots, (u_n, v_n)$ de paires de mots sur l'alphabet Σ.*
Réponse: *Décider cette suite admet une* solution*, c'est-à-dire une suite d'indices $i_1, i_2, \cdots, i_m$ de $\{1, 2, \cdots, n\}$ telle que*

$$u_{i_1} u_{i_2} \cdots u_{i_m} = v_{i_1} v_{i_2} \cdots v_{i_m}.$$

Théorème 8.13 *Le problème* Problème de la correspondance de Post *est indécidable.*

8.6 Théorèmes du point fixe

Les résultats de cette section sont très subtils, mais extrêmement puissants.

Commençons par une version simple, qui nous aidera à comprendre les preuves.

Proposition 8.6 *Il existe une machine de Turing A^* qui écrit son propre algorithme : elle produit en sortie $\langle A^* \rangle$.*

Autrement dit, il est possible d'écrire un programme qui affiche son propre code.

C'est vrai dans tout langage de programmation qui est équivalent aux machines de Turing :

En shell *UNIX* par exemple, le programme suivant

```
x='y=‘echo . | tr . "\47" ‘; echo "x=$y$x$y;$x"'
y=‘echo . | tr . "\47"‘; echo "x=$y$x$y;$x"
```

produit

```
x='y=‘echo . | tr . "\47" ‘; echo "x=$y$x$y;$x"';y=‘echo .
| tr . "\47" ‘; echo "x=$y$x$y;$x"
```

qui est une commande, qui exécutée, affiche son propre code.

On appelle parfois de tels programme des *quines*, en l'honneur du philosophe Willard van Orman Quine, qui a discuté l'existence de programmes autoreproducteurs.

Démonstration: On considère des machines de Turing qui terminent sur tout entrée. Pour deux telles machines A et A', on note AA' la machine de Turing qui est obtenue en composant de façon séquentielle A et A'. Formellement, AA' est la machine de Turing qui effectue d'abord le programme de A, et puis lorsque A temine avec la sortie w, effectue le programme de A' sur l'entrée w.

On construit les machines suivantes :

1. Étant donné un mot w, la machine de Turing $Print_w$ termine avec le résultat w ;
2. Pour une entrée w de la forme $w = \langle X \rangle$, où X est une machine de Turing, la machine de Turing B produit en sortie le codage de la machine de Turing $Print_w X$, c'est-à-dire le codage de la machine de Turing obtenue en composant $Print_w$ et X.

On considère alors la machine de Turing A^* donnée par $Print_{\langle B \rangle} B$, c'est-à-dire la composition séquentielle des machines $Print_{\langle B \rangle}$ et B.

Déroulons le résultat de cette machine : la machine de Turing $Print_{\langle B \rangle}$ produit en sortie $\langle B \rangle$. La composition par B produit alors le codage de $Print_{\langle B \rangle} \langle B \rangle$, qui est bien le codage de la machine de Turing A^*. □

Le théorème de récursion permet des autoréférences dans un langage de programmation. Sa démonstration consiste à étendre l'idée derrière la preuve du résultat précédent.

Théorème 8.14 (Théorème de récursion) *Soit $t : M^* \times M^* \to M^*$ une fonction calculable. Alors il existe une machine de Turing R qui calcule une fonction $r : M^* \to M^*$ telle que pour tout mot w*

$$r(w) = t(\langle R \rangle, w).$$

Son énoncé peut paraître technique, mais son utilisation reste assez simple. Pour obtenir une machine de Turing qui obtient sa propre description et qui l'utilise pour calculer, on a besoin simplement d'une machine de Turing T qui calcule une fonction t comme dans l'énoncé, qui prend une entrée supplémentaire qui contient la description de la machine de Turing. Alors le théorème de récursion produit une nouvelle machine R qui opère comme T mais avec la description de $\langle R\rangle$ gravée dans son code.

Démonstration: Il s'agit de la même idée que précédemment. Soit T une machine de Turing calculant la fonction t : T prend en entrée une paire $\langle u, w\rangle$ et produit en sortie un mot $t(u, w)$.

On considère les machines suivantes :

1. Étant donné un mot w, la machine de Turing $Print_w$ prend en entrée un mot u et termine avec le résultat $\langle w, u\rangle$;
2. Pour une entrée w' de la forme $\langle\langle X\rangle, w\rangle$, la machine de Turing B :
   (a) calcule $\langle\langle Print_{\langle X\rangle}X\rangle, w\rangle$, où $Print_{\langle X\rangle}X$ désigne la machine de Turing qui compose $Print_{\langle X\rangle}$ avec X ;
   (b) puis passe le contrôle à la machine de Turing T.

On considère alors la machine de Turing R donnée par $Print_{\langle B\rangle}B$, c'est-à-dire la machine de Turing obtenue en composant $Print_{\langle B\rangle}$ avec B.

Déroulons le résultat $r(w)$ de cette machine R sur une entrée w : sur une entrée w, la machine de Turing $Print_{\langle B\rangle}$ produit en sortie $\langle\langle B\rangle, w\rangle$. La composition par B produit alors le codage de $\langle\langle Print_{\langle B\rangle}B\rangle, w\rangle$, et passe le contrôle à T. Ce dernier produit alors $t(\langle Print_{\langle B\rangle}B\rangle, w) = t(\langle R\rangle, w) = r(w)$. □

On obtient alors :

Théorème 8.15 (Théorème du point fixe de Kleene) *Soit une fonction calculable qui à chaque mot $\langle A\rangle$ codant une machine de Turing associe un mot $\langle A'\rangle$ codant une machine de Turing. Notons $A' = f(A)$.*

Alors il existe une machine de Turing A^* tel que $L(A^*) = L(f(A^*))$.

Démonstration: Considérons une fonction $t : M^* \times M^* \to M^*$ telle que $t(\langle A\rangle, x)$ soit le résultat de la simulation de la machine de Turing $f(A)$ sur l'entrée x. Par le théorème précédent, il existe une machine de Turing R qui calcule une fonction r telle que $r(w) = t(\langle R\rangle, w)$. Par construction $A^* = R$ et $f(A^*) = f(R)$ ont donc même valeur sur w pour tout w. □

Remarque 8.9 *On peut interpréter les résultat précédents en lien avec les virus informatiques. En effet, un virus est un programme qui vise à se diffuser, c'est-à-dire à s'autoreproduire, sans être détecté. Le principe de la preuve du théorème de récursion est un moyen de s'autoreproduire, en dupliquant son code.*

8.7 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture de [Sipser, 1997] et de [Hopcroft et al., 2001] en anglais, ou de [Wolper, 2001], [Stern, 1994] [Carton, 2008] en français.

Le livre [Sipser, 1997] est un excellent ouvrage très pédagogue.

Bibliographie Ce chapitre contient des résultats standards en calculabilité. Nous nous sommes inspirés ici essentiellement de leur présentation dans les livres [Wolper, 2001], [Carton, 2008], [Jones, 1997], [Kozen, 1997], [Hopcroft et al., 2001], ainsi que dans [Sipser, 1997].

Chapitre 9

Incomplétude de l'arithmétique

En 1931, Kurt Gödel a prouvé un résultat dont les conséquences philosophiques en science ont été révolutionnaires : il a ainsi prouvé que toute théorie suffisante pour capturer les raisonnements arithmétiques est nécessairement incomplète, c'est-à-dire telle qu'il existe des énoncés qui ne sont pas démontrables et dont la négation n'est pas non plus démontrable.

Ce théorème est largement considéré comme l'un des plus grands achèvements des mathématiques et de la logique du 20ème siècle.

Avec tous les ingrédients précédents, nous sommes en fait en position de comprendre ce théorème et d'en donner une preuve complète. C'est l'objet de ce chapitre : enfin, nous donnerons la preuve complète due à Turing. Nous ne ferons qu'évoquer la preuve de Gödel, qui permet d'en dire plus.

9.1 Théorie de l'arithmétique

9.1.1 Axiomes de Peano

La question à laquelle on s'intéresse est de tenter d'axiomatiser l'arithmétique, c'est-à-dire les propriétés des entiers.

Nous avons déjà présenté dans le chapitre 6, les axiomes de l'arithmétique de Robinson et les axiomes de Peano : on s'attend à ce que ces axiomes soient vérifiés sur les entiers, c'est-à-dire dans *le modèle standard des entiers* où l'ensemble de base est les entiers, et où l'on interprète $+$ par l'addition, $*$ par la multiplication, et $s(x)$ par $x + 1$.

Autrement dit, on s'attend à ce que ces axiomes possèdent au moins un modèle : *le modèle standard des entiers*.

Étant donnée une formule close sur une signature contenant ces symboles, F est soit vraie soit fausse sur les entiers (c'est-à-dire dans le modèle standard

des entiers). Appelons *théorie de l'arithmétique*, l'ensemble $Th(\mathbb{N})$ des formules closes F qui sont vraies sur les entiers.

9.1.2 Quelques concepts de l'arithmétique

Il est possible de prouver que de nombreux concepts de la théorie des nombres se définissent parfaitement à partir de ces axiomes.

Par exemple, on peut exprimer les concepts suivants :

- INTDIV(x, y, q, r) défini comme "q est le quotient et r le reste de la division euclidienne de x par y".
  En effet, cela peut s'écrire par la formule :
  $$(x = q * y + r \wedge r < y).$$
- DIV(y, x) défini comme "y divise x". En effet, cela peut s'écrire :
  $$\exists q \text{ INTDIV}(x, y, q, 0).$$
- EVEN(x) défini comme "x est pair". En effet, cela peut s'écrire :
  $$\text{DIV}(2, x).$$
- ODD(x) défini comme "x est impair". En effet, cela peut s'écrire :
  $$\neg\text{EVEN}(x).$$
- PRIME(x) défini comme "x est premier". En effet, cela peut s'écrire :
  $$(x \geq 2 \wedge \forall y(\text{DIV}(y, x) \Rightarrow (y = 1 \vee y = x))).$$
- POWER$_2(x)$ défini comme "x est une puissance de 2". En effet, cela peut s'écrire :
  $$\forall y((\text{DIV}(y, x) \wedge \text{PRIME}(y)) \Rightarrow y = 2).$$

9.1.3 La possibilité de parler des bits d'un entier

On peut aussi écrire des formules comme BIT(x, y) signifiant que "y est une puissance de 2, disons 2^k, et le $k$ème bit de la représentation binaire de l'entier x est 1".

Cela est plus subtile, mais possible. Cela s'écrit en effet :

$$(\text{POWER}_2(y) \wedge \forall q \forall r(\text{INTDIV}(x, y, q, r) \Rightarrow \text{ODD}(q))).$$

L'idée est que si y satisfait la formule, alors y est une puissance de 2, et donc en binaire il s'écrit 10^k pour un entier k. En divisant x par y, le reste de la division r sera les k bits de poids le plus faible de x, et le quotient q les autres bits de x, puisqu'on a $x = q * y + r$. En testant si q est impair, on "lit" le $k + 1$ème bit de x, soit le bit correspondant au bit à 1 dans l'entier y codant cette position.

9.1.4 Principe de la preuve de Gödel

Kurt Gödel a prouvé le théorème d'incomplétude en construisant, pour n'importe quel système de preuve raisonnable, une formule ϕ de l'arithmétique qui affirme sa propre non-prouvabilité dans le système :

$$\phi \text{ est vraie } \Leftrightarrow \phi \text{ n'est pas prouvable.} \tag{9.1}$$

Tout système de preuve raisonnable est valide, et donc on doit avoir

$$\psi \text{ prouvable } \Rightarrow \psi \text{ est vrai.} \tag{9.2}$$

Alors ϕ doit être vraie, car sinon

$$\begin{array}{lll} \phi \text{ est fausse} & \Rightarrow \phi \text{ est prouvable.} & \text{(par (9.1))} \\ & \Rightarrow \phi \text{ est vraie.} & \text{(par (9.2))} \end{array}$$

La construction de ϕ est intéressante par elle même, car elle capture la notion d'auto-référence.

On reviendra sur la construction de Gödel.

9.2 Théorème d'incomplétude

9.2.1 Principe de la preuve de Turing

On va prouver le théorème d'incomplétude en utilisant une approche qui permet d'obtenir les principales conséquences du théorème, et qui en fait due à Alan Turing.

Cette approche est plus simple, et surtout nous avons maintenant tous les ingrédients pour en faire une preuve complète, en utilisant les arguments de la calculabilité.

L'idée est de se convaincre que dans l'arithmétique de Peano, ainsi que dans tout système "raisonnable" de preuve pour la théorie de l'arithmétique :

Théorème 9.1 *1. L'ensemble des théorèmes (formules closes prouvables) à partir des axiomes de Peano (ou de toute axiomatisation "raisonnable" des entiers) est récursivement énumérable.*

2. L'ensemble $Th(\mathbb{N})$ des formules closes F vraies sur les entiers n'est pas récursivement énumérable.

Par conséquent, les deux ensembles ne peuvent pas être les mêmes, et le système de preuve ne peut pas être complet. En clair :

Corollaire 9.1 *Il existe donc des formules closes vraies de $Th(\mathbb{N})$ qui ne sont pas prouvables, ou dont la négation n'est pas prouvable à partir des axiomes de Peano, ou de toute axiomatisation "raisonnable" des entiers.*

C'est le premier théorème d'incomplétude de Kurt Gödel.

9.2.2 Le point facile

L'ensemble des théorèmes (formules closes prouvables à partir des axiomes de Peano) de l'arithmétique est certainement récursivement énumérable : quelle que soit la méthode de preuve (par exemple celle du chapitre 6), on peut énumérer les théorèmes en énumérant les axiomes et en appliquant systématiquement toutes les règles de déduction dans toutes les façons possibles, en émettant toutes les formules closes qui peuvent être dérivées.

Cela reste vrai en fait dès que l'on suppose que l'on peut énumérer les axiomes de l'axiomatisation dont on part. C'est pourquoi, on peut dire que l'ensemble des théorèmes de toute axiomatisation raisonnable des entiers est récursivement énumérable.

Remarque 9.1 *Autrement dit, plus haut, si on veut une définition de "raisonnable", il suffit de prendre "récursivement énumérable".*

9.2.3 Lemme crucial

Le point crucial est alors de prouver le résultat suivant.

Lemme 9.1 *L'ensemble $Th(\mathbb{N})$ n'est pas récursivement énumérable.*

On prouve cela en réduisant le complémentaire $\overline{\text{HALTING} - \text{PROBLEM}}$ du problème de l'arrêt $\text{HALTING} - \text{PROBLEM}$ des machines de Turing à ce problème, i.e. en montrant que $\overline{\text{HALTING} - \text{PROBLEM}} \leq_m Th(\mathbb{N})$.

Le théorème découle alors :

- du fait que $\overline{\text{HALTING} - \text{PROBLEM}}$ n'est pas récursivement énumérable ;
- et du fait que si $A \leq_m B$ et que si A n'est pas récursivement énumérable, alors B non plus.

Rappelons que L_{univ} est le problème suivant : on se donne $\langle\langle M\rangle, w\rangle$, et on veut déterminer si la machine de Turing M s'arrête sur l'entrée w.

Étant donné $\langle\langle M\rangle, w\rangle$, nous montrons comment produire une formule close γ sur la signature de l'arithmétique telle que

$$\langle\langle M\rangle, w\rangle \in \overline{\text{HALTING} - \text{PROBLEM}} \Leftrightarrow \gamma \in Th(\mathbb{N}).$$

En d'autres termes, étant donné M et w, on doit construire une formule close γ sur la signature de l'arithmétique qui affirme que "la machine de Turing M ne s'arrête pas sur l'entrée w".

Cela s'avère possible parce que le langage de l'arithmétique est suffisamment puissant pour parler des machines de Turing et du fait qu'elles s'arrêtent.

En utilisant le principe de la formule $\text{BIT}(y, x)$ précédent, on va construire une série de formules dont le point culminant sera une formule $\text{VALCOMP}_{M,w}(y)$ qui dit que y est un entier qui représente une suite de configurations de M sur l'entrée w : en d'autres termes, y représente une suite de configurations $C_0, C_1, \cdots, C_t$ de M, codée sur un certain alphabet Σ telle que :

- C_0 est la configuration initiale $C[w]$ de M sur w ;

– C_{i+1} est la configuration successeur de C_i, selon la fonction de transition δ de la machine de Turing M, pour $i < t$;
– C_t est une configuration acceptante.

Une fois que l'on a réussi à écrire la formule $\text{VALCOMP}_{M,w}(y)$, il est facile d'écrire que M ne s'arrête pas sur l'entrée x : la formule γ s'écrit

$$\neg\exists y \ \text{VALCOMP}_{M,w}(y).$$

Cela prouve la réduction, et donc termine la preuve du lemme, et donc prouve aussi le théorème, en rappellant que $\overline{\text{HALTING} - \text{PROBLEM}}$ n'est pas récursivement énumérable.

9.2.4 Construction de la formule

Il ne reste plus qu'à donner les détails fastidieux de la construction de la formule γ à partir de M et de w. Allons y.

Supposons que l'on encode les configurations de M sur un alphabet fini Σ, que l'on peut supposer sans perte de restriction de taille p, avec p premier.

Chaque nombre possède une unique représentation en base p : on va utiliser cette représentation en base p plutôt que la représentation binaire, pour simplifier la discussion.

Supposons que la configuration initiale de M sur $w = a_1 a_2 \cdots a_n$ soit codée par l'entier dont les chiffres de l'écriture en base p soient respectivement $q_0 a_1 a_2 \cdots a_n$: on utilise la représentation de la définition 7.4pour représenter les configurations.

Considérons que le symbole de blanc **B** est codé par le chiffre en base p k.

Soit LEGAL l'ensemble des 6-uplets (a, b, c, d, e, f) de nombres en base p qui correspondent aux fenêtres légales pour la machine M : voir la notion de fenêtre légale du chapitre 7. Si l'on préfère, LEGAL est l'ensemble des 6-uplets (a, b, c, d, e, f) tels que si trois éléments de Σ représentés respectivement par a, b et c apparaissent consécutivement dans une configuration C_i, et si d, e, f apparaissent consécutivement aux mêmes emplacements dans la configuration C_{i+1}, alors cela est cohérent avec la fonction de transition δ de la machine de Turing M.

On définit maintenant quelques formules :

– $\text{POWER}_p(x)$: "Le nombre x est une puissance de p" : ici p est un nombre premier fixé. Cela s'écrit :

$$\forall y((\text{DIV}(y, x) \wedge \text{PRIME}(y)) \Rightarrow y = p).$$

– $\text{LENGTH}_p(v, d)$: "Le nombre d est une puissance de p qui donne la longueur de v vu comme un mot sur l'alphabet Σ à p lettres". Cela s'écrit :

$$(\text{POWER}_p(d) \wedge v < d).$$

– $\text{DIGIT}_p(v, y, b)$: "Le $p$ème chiffre de v à la position y est b (y est donné par une puissance de p)". Cela s'écrit :

$$\exists u \exists a(v = a + b * y + u * p * y \wedge a < y \wedge b < p).$$

– 3DIGIT(v, y, b, c, d) : "Les 3 chiffres de v à la position y sont b, c et d (y est donné par une puissance de p)". Cela s'écrit :

$$\exists u \exists a (v = a + b*y + c*p*y + d*p*p*y + u*p*p*p*y \wedge a < y \wedge b < p \wedge c < p \wedge d < p).$$

– MATCH(v, y, z) : "Les 3 chiffres de v à la position y sont b, c et d et correspondent aux 3 chiffres de v à la position z selon la fonction de transition δ de la machine de Turing (y et z sont donnés par une puissance de p)". Cela s'écrit :

$$\bigwedge_{(a,b,c,d,e,f) \in \text{LEGAL}} \text{3DIGIT}(v, y, a, b, c) \wedge \text{3DIGIT}(v, z, d, e, f).$$

Remarque 9.2 *On note évidemment ici,* $\bigwedge_{(a,b,c,d,e,f) \in \text{LEGAL}}$ *pour la conjonction pour chacun des* 6*-uplets de* LEGAL.

– MOVE(v, c, d) : "la suite v représente une suite de configurations successives de M de longueur c jusqu'à d : toutes les paires de suites de 3-chiffres exactement écartés de c positions dans v se correspondent selon δ". Cela s'écrit :

$$\forall y((\text{POWER}_p(y) \wedge y * p * p * c < d) \Rightarrow \text{MATCH}(v, y, yc)).$$

– START(v, c) : "la suite v débute avec la configuration initiale de M sur l'entrée $w = a_1 a_2 \cdots a_n$ auxquelles on a ajouté des blancs B jusqu'à la longueur c (c est donné par une puissance de p ; $n, p^i, 0 \leq i \leq n$ sont des constantes fixées qui ne dépendent que de M)". Cela s'écrit :

$$\bigwedge_{i=0}^{n} \text{DIGIT}_p(v, p^i, a_i) \wedge p^n < c \wedge \forall y(\text{POWER}_p(y) \wedge p^n < y < c \Rightarrow \text{DIGIT}_p(v, y, B)).$$

– HALT(v, d) : "La suite v possède un état d'arrêt quelque part". Cela s'écrit :

$$\exists y(\text{POWER}_p(y) \wedge y < d \wedge \text{DIGIT}_p(v, y, q_a)).$$

– VALCOMP$_{M,w}(v)$: "La suite v est un calcul de M valide sur w". Cela s'écrit :

$$\exists c \exists d\, (\text{POWER}_p(c) \wedge c < d \wedge \text{LENGTH}_p(v, d) \wedge \text{START}(v, c) \wedge \text{MOVE}(v, c, d) \wedge \text{HALT}(v, d)).$$

– γ : "La machine M ne s'arrête pas sur w". Cela s'écrit :

$$\neg \exists v \ \text{VALCOMP}_{M,w}(v).$$

Notre preuve est terminée.

9.3 La preuve de Gödel

Kurt Gödel a en fait prouvé son théorème d'incomplétude d'une autre façon, en construisant une formule close qui affirme sa propre non-prouvabilité. Notons $\vdash$ pour prouvable, et $\models$ pour vrai respectivement sur les entiers.

Supposons que l'on ait fixé un codage des formules par les entiers d'une façon raisonnable : si ϕ est une formule, alors $\langle\phi\rangle$ désigne son codage (un entier).

9.3.1 Lemme de point fixe

Voici un lemme qui a été prouvé par Gödel, et qui ressemble fort aux théorèmes de point fixe déjà évoqués dans le chapitre précédent.

Lemme 9.2 (Lemme de point fixe de Gödel) *Pour toute formule $\psi(x)$ avec la variable libre x, il y a une formule close τ telle que*

$$\vdash \tau \Leftrightarrow \psi(\langle\tau\rangle),$$

i.e. les formules closes τ et $\psi(\langle\tau\rangle)$ sont prouvablement équivalente dans l'arithmétique de Peano.

Démonstration: Soit x_0 une variable fixée. On peut construire une formule $\mathrm{SUBST}(x, y, z)$ avec les variables libres x, y, z qui affirme "le nombre z est le codage d'une formule obtenue en substituant la constante dont la valeur est x dans toutes les occurrences de la variable libre x_0 dans la formule dont le codage est y".

Par exemple, si $\phi(x_0)$ est une formule qui contient une occurrence libre de x_0, mais aucune autre variable libre, la formule $\mathrm{SUBST}(7, \langle\phi(x_0)\rangle, 312)$ est vraie si $312 = \langle\phi(7)\rangle$.

On ne donnera pas les détails de la construction de la formule SUBST, mais l'idée est simplement d'observer que cela est bien possible, en utilisant par exemple l'idée de la relation $\mathrm{BIT}(x, y)$.

On considère maintenant $\sigma(x)$ définie par $\forall y\ (\mathrm{SUBST}(x, x, y) \Rightarrow \psi(y))$, et τ définie par $\sigma(\langle\sigma(x_0)\rangle)$.

Alors τ est la solution recherchée car

$$\begin{array}{rcl}
\tau & = & \sigma(\langle\sigma(x_0)\rangle) \\
& = & \forall y\ (\mathrm{SUBST}(\langle\sigma(x_0)\rangle, \langle\sigma(x_0)\rangle, y) \Rightarrow \psi(y)) \\
& \Leftrightarrow & \forall y\ y = \langle\sigma(\langle\sigma(x_0)\rangle)\rangle \Rightarrow \psi(y) \\
& \Leftrightarrow & \forall y\ y = \langle\tau\rangle \Rightarrow \psi(y) \\
& \Leftrightarrow & \psi(\langle\tau\rangle)
\end{array}$$

Bien entendu, on a utilisé ici des équivalences informelles, mais l'argument peut bien se formuler dans l'arithmétique de Peano. □

9.3.2 Arguments de Gödel

On observe maintenant que le langage de l'arithmétique est suffisamment puissant pour parler de prouvabilité en arithmétique de Peano. En particulier, il est possible de coder une suite de formules par des entiers et d'écrire une formule PROOF(x, y) qui signifie que la suite de formule dont le codage est x est une preuve de la formule dont le codage est y.

En d'autres termes, $\vdash$ PROOF$(\langle\pi\rangle, \langle\psi\rangle) \Leftrightarrow \pi$ est une preuve de ϕ dans l'arithmétique de Peano.

La prouvabilité dans l'arithmétique de Peano se code alors par la formule PROVABLE(y) définie par $\exists x$ PROOF(x, y).

Alors pour toute formule close ϕ,

$$\vdash \phi \Leftrightarrow \quad \models PROVABLE(\langle\phi\rangle). \tag{9.3}$$

On a alors :

$$\vdash \phi \Leftrightarrow \vdash PROVABLE(\langle\phi\rangle). \tag{9.4}$$

La direction $\Rightarrow$ est vraie car si ϕ est prouvable, alors il y a une preuve π de ϕ. L'arithmétique de Peano et le système de preuve permet d'utiliser cette preuve pour prouver ϕ (i.e. que PROOF$(\langle\pi\rangle, \langle\phi\rangle)$). La direction $\Leftarrow$ découle de 9.3 est de la validité de la preuve dans l'arithmétique de Peano.

Utilisons alors le lemme de point fixe à la formule close $\neg$PROVABLE(x). On obtient une formule close ρ qui affirme sa propre non-prouvabilité :

$$\vdash \rho \Leftrightarrow \neg \text{ PROVABLE}(\langle\rho\rangle),$$

en d'autres termes, ρ est vraie si et seulement si elle n'est pas prouvable dans l'arithmétique de Peano.

Par validité de la preuve dans l'arithmétique de Peano, on a

$$\models \rho \Leftrightarrow \neg \text{ PROVABLE}(\langle\rho\rangle). \tag{9.5}$$

Alors la formule ρ doit être vraie, puisque sinon, alors

$$\begin{array}{rcll} \models \neg\rho & \Rightarrow & \text{PROVABLE}(\langle\rho\rangle) & \text{(par 9.5)} \\ & \Rightarrow & \vdash \rho & \text{(par 9.3)} \\ & \Rightarrow & \models \rho & \text{(par validité de A. de Peano)} \end{array}$$

une contradiction.

Donc $\models \rho$. Mais maintenant,

$$\begin{array}{rcll} \models \rho & \Rightarrow & \neg\text{PROVABLE}(\langle\rho\rangle) & \text{(par 9.5)} \\ & \Rightarrow & \not\models \rho & \text{(par définition de la vérité)} \\ & \Rightarrow & \nvdash \rho & \text{(par 9.3)} \end{array}$$

Donc ρ est vraie, mais n'est pas prouvable.

9.3.3 Second théorème d'incomplétude de Gödel

Le défaut de la preuve précédente est bien entendu qu'elle ne donne pas un grand sens à la formule ρ.

Le second théorème d'incomplétude de Kurt Gödel donne un exemple explicite de formule non prouvable.

On peut exprimer une formule CONSIST qui exprime le fait que la théorie est cohérente. On écrit qu'il n'est pas possible de prouver une formule F et sa négation : il suffit d'écrire $\neg\exists x(\text{PROVABLE}(x) \wedge \text{PROVABLE}(y) \wedge NEG(x, y))$, où $NEG(x, y)$ signifie que y est le codage de la négation de la formule codée par x.

Le second théorème d'incomplétude de Gödel permet de prouver que cette formule précise n'est pas prouvable.

Autrement dit :

Théorème 9.2 (Second théorème d'incomplétude de Gödel) *Aucun système de déduction cohérent ne peut prouver sa propre cohérence.*

Nous ne rentrerons pas plus dans les détails.

9.4 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture des derniers chapitres de [Kozen, 1997], qui reste courts et directs, ou de l'ouvrage [Cori and Lascar, 1993b] pour une preuve complète.

Bibliographie Ce chapitre est repris des trois derniers chapitres de l'excellent livre [Kozen, 1997].

Chapitre 10

Bases de l'analyse de complexité d'algorithmes

Les discussions précédentes ont fait intervenir l'existence ou non d'algorithmes pour résoudre un problème donné, mais en ignorant un aspect pourtant essentiel en pratique : les ressources nécessaires à son exécution, c'est-à-dire par exemple le temps ou la mémoire nécessaire sur la machine pour l'exécuter.

L'objectif du chapitre suivant est de se focaliser sur l'une des ressources : le temps de calcul. Dans le chapitre qui suit, nous évoquerons d'autres ressources comme l'espace mémoire. On pourrait parler du temps parallèle, c'est-à-dire du temps sur une machine parallèle, etc.

Commençons toutefois par bien comprendre la différence entre le cadre des chapitres qui suivent et les chapitres précédents : dans les chapitres précédents, on parlait de *calculabilité*, c'est-à-dire que l'on se posait la question de l'existence (ou de la non-existence) de solutions algorithmiques à des problèmes donnés. On va maintenant parler de *complexité* : c'est-à-dire que l'on se focalise maintenant uniquement sur des problèmes *décidables*, pour lesquels on connaît un algorithme. La question que l'on se pose maintenant est de comprendre s'il existe un algorithme *efficace*.

Cela nous mène tout d'abord à préciser ce que l'on a envie d'appeler *efficace*, et comment on mesure cette *efficacité*. Toutefois, avant, il faut que notre lecteur ait les idées au clair sur ce que l'on appelle la complexité d'un algorithme, ou la complexité d'un problème, ce qui n'est pas la même chose.

Remarque 10.1 *Même si nous évoquons les complexités en moyenne dans ce chapitre, nous n'en aurons pas besoin à aucun moment dans les chapitres qui suivent : nous le faisons surtout pour expliquer pourquoi elles sont peu utilisées.*

10.1 Complexité d'un algorithme

On considère donc typiquement dans ce chapitre un problème $\mathcal{P}$ pour lequel on connaît un algorithme $\mathcal{A}$: cet algorithme, on sait qu'il est correct, et qu'il termine. Il prend en entrée une donnée d, et produit un résultat en sortie $\mathcal{A}(d)$ en utilisant certaines ressources (on ne parle donc plus que de problèmes décidables).

Exemple 10.1 *Le problème $\mathcal{P}$ peut par exemple être celui de déterminer si un nombre v est parmi une liste de n nombres.*

Il est clair que l'on peut bien construire un algorithme $\mathcal{A}$ pour résoudre ce problème : par exemple,
- on utilise une variable res que l'on met à 0 ;
- on parcourt la liste, et pour chaque élément :
  - on regarde si cet élément est le nombre v :
    - si c'est le cas, on met la variable res à 1
- A l'issu de cette boucle, on retourne res.

Cet algorithme n'est pas le plus efficace que l'on puisse envisager. D'une part, on pourrait s'arrêter dès que l'on a mis res à 1, puisqu'on connaît la réponse. D'autre part, on peut clairement faire tout autrement, et utiliser par exemple une *recherche par dichotomie* (un algorithme récursif).

10.1.1 Premières considérations

On mesure toujours l'efficacité, c'est-à-dire la complexité, d'un algorithme en terme d'une mesure élémentaire μ à valeur entière : cela peut être le nombre d'instructions effectuées, la taille de la mémoire utilisée, ou le nombre de comparaisons effectuées, ou toute autre mesure.

Il faut simplement qu'étant donnée une entrée d, on sache clairement associer à l'algorithme $\mathcal{A}$ sur l'entrée d, la valeur de cette mesure, notée $\mu(\mathcal{A}, d)$: par exemple, pour un algorithme de tri qui fonctionne avec des comparaisons, si la mesure élémentaire μ est le nombre de comparaisons effectuées, $\mu(\mathcal{A}, d)$ est le nombre de comparaisons effectuées sur l'entrée d (une liste d'entiers) par l'algorithme de tri $\mathcal{A}$ pour produire le résultat $\mathcal{A}(d)$ (cette liste d'entiers triée).

Il est clair que $\mu(\mathcal{A}, d)$ est une fonction de $\mathcal{A}$, mais aussi de l'entrée d. La qualité d'un algorithme $\mathcal{A}$ n'est donc pas un critère absolu, mais une fonction quantitative $\mu(\mathcal{A}, .)$ des données d'entrée vers les entiers.

10.1.2 Complexité d'un algorithme au pire cas

En pratique, pour pouvoir appréhender cette fonction, on cherche souvent à évaluer cette complexité pour les entrées d'une certaine *taille* : il y a souvent une fonction *taille* qui associe à chaque donnée d'entrée d, un entier $taille(d)$, qui correspond à un paramètre naturel. Par exemple, cette fonction peut être celle qui compte le nombre d'éléments dans la liste pour un algorithme de tri,

la taille d'une matrice pour le calcul du déterminant, la somme des longueurs des listes pour un algorithme de concaténation.

Pour passer d'une fonction des données vers les entiers, à une fonction des entiers (les tailles) vers les entiers, on considère alors la complexité *au pire cas* : la complexité $\mu(\mathcal{A}, n)$ de l'algorithme $\mathcal{A}$ sur les entrées de taille n est définie par

$$\mu(\mathcal{A}, n) = \max_{d \text{ entrée avec } taille(d)=n} \mu(\mathcal{A}, d).$$

Autrement dit, la complexité $\mu(\mathcal{A}, n)$ est la complexité la pire sur les données de taille n.

Par défaut, lorsqu'on parle de *complexité d'algorithme* en informatique, il s'agit de complexité au pire cas, comme ci-dessus.

Si l'on ne sait pas plus sur les données, on ne peut guère faire plus que d'avoir cette vision pessimiste des choses : cela revient à évaluer la complexité dans le pire des cas (le meilleur des cas n'a pas souvent un sens profond, et dans ce contexte le pessimisme est de loin plus significatif).

10.1.3 Complexité moyenne d'un algorithme

Pour pouvoir en dire plus, il faut en savoir plus sur les données. Par exemple, qu'elles sont distribuées selon une certaine loi de probabilité.

Dans ce cas, on peut alors parler de complexité en moyenne : la complexité moyenne $\mu(\mathcal{A}, n)$ de l'algorithme $\mathcal{A}$ sur les entrées de taille n est définie par

$$\mu(\mathcal{A}, n) = \mathbb{E}[\mu(\mathcal{A}, d) | d \text{ entrée avec } taille(d) = n],$$

où $\mathbb{E}$ désigne l'espérance (la moyenne).

Si l'on préfère,

$$\mu(\mathcal{A}, n) = \sum_{d \text{ entrée avec } taille(d)=n} \pi(d)\mu(\mathcal{A}, d),$$

où $\pi(d)$ désigne la probabilité d'avoir la donnée d parmi toutes les données de taille n.

En pratique, le pire cas est rarement atteint et l'analyse en moyenne peut sembler plus séduisante.

Mais, d'une part, il est important de comprendre que l'on ne peut pas parler de moyenne sans loi de probabilité (sans distribution) sur les entrées. Cela implique que l'on connaisse d'autre part la distribution des données en entrée, ce qui est très souvent délicat à estimer en pratique. Comment anticiper par exemple les matrices qui seront données à un algorithme de calcul de déterminant par exemple ?

On fait parfois l'hypothèse que les données sont équiprobables (lorsque cela a un sens, comme lorsqu'on trie n nombres entre 1 et n et où l'on peut supposer que les permutations en entrée sont équiprobables), mais cela est est bien souvent totalement arbitraire, et pas réellement justifiable.

Et enfin, comme nous allons le voir sur quelques exemples, les calculs de complexité en moyenne sont plus délicats à mettre en œuvre.

10.2 Complexité d'un problème

On peut aussi parler de la *complexité d'un problème* : cela permet de discuter de l'optimalité ou non d'un algorithme pour résoudre un problème donné.

On fixe un problème $\mathcal{P}$: par exemple celui de trier une liste d'entiers. Un algorithme $\mathcal{A}$ qui résout ce problème est un algorithme qui répond à la spécification du problème $\mathcal{P}$: pour chaque donnée d, il produit la réponse correcte $\mathcal{A}(d)$.

La complexité du problème $\mathcal{P}$ sur les entrées de taille n est définie par

$$\mu(\mathcal{P}, n) = \inf_{\mathcal{A} \text{ algorithme qui résout } \mathcal{P}} \max_{d \text{ entrée avec } taille(d)=n} \mu(\mathcal{A}, d).$$

Autrement dit, on ne fait plus seulement varier les entrées de taille n, mais aussi l'algorithme. On considère le meilleur algorithme qui résout le problème. Le meilleur étant celui avec la meilleure complexité au sens de la définition précédente, et donc au pire cas. C'est donc la complexité du meilleur algorithme au pire cas.

L'intérêt de cette définition est le suivant : si un algorithme $\mathcal{A}$ possède la complexité $\mu(\mathcal{P}, n)$, c'est-à-dire est tel que $\mu(\mathcal{A}, n) = \mu(\mathcal{P}, n)$ pour tout n, alors cet algorithme est clairement optimal. Tout autre algorithme est moins performant, par définition. Cela permet donc de prouver qu'un algorithme est optimal.

10.3 Exemple : Calcul du maximum

Nous allons illustrer la discussion précédente par un exemple : le problème du calcul du maximum. Ce problème est le suivant : on se donne en entrée une liste d'entiers naturels $e_1, e_2, \cdots, e_n$, avec $n \geq 1$, et on cherche à déterminer en sortie $M = \max_{1 \leq i \leq n} e_i$, c'est-à-dire le plus grand de ces entiers.

10.3.1 Complexité d'un premier algorithme

En considérant que l'entrée est rangée dans un tableau, la fonction Java suivante résout le problème.

```
static int max(int T[]) {
    int M =  T[0];
    int i = 1;
    while (i < T.length) {
     if (M < T[i])  M = T[i];
     i = i+1;
    }
    return M;
  }
```

Si notre mesure élémentaire μ correspond au nombre de comparaisons, nous en faisons 2 par itération de la boucle, qui est exécutée $n-1$ fois, plus 1 dernière

du type $i < T.length$ lorsque i vaut $T.length$. Nous avons donc $\mu(\mathcal{A}, d) = 2n-1$ pour cet algorithme $\mathcal{A}$. Ce nombre est indépendant de la donnée d, et donc $\mu(\mathcal{A}, n) = 2n - 1$.

Par contre, si notre mesure élémentaire μ correspond au nombre d'affectations, nous en faisons au minimum 2 avant la boucle **while**. Chaque itération de la boucle effectue soit 1 ou 2 affectations suivant le résultat du test M<T[i]. On a donc pour une entrée d de taille n, $n+1 \leq \mu(\mathcal{A}, d) \leq 2n$: la valeur minimum est atteinte pour une liste ayant son maximum en $T[0]$, et la valeur maximum pour une liste sans répétition triée dans l'ordre croissant. Cette fois $\mu(\mathcal{A}, d)$ dépend de l'entrée d. La complexité au pire cas est donnée par $\mu(\mathcal{A}, n) = 2n$.

10.3.2 Complexité d'un second algorithme

Si l'entrée est rangée dans une liste, définie par exemple par

```
class List {
  int val ;      // L'élément
  List next ;    // La suite

  List (int  val, List next) {
    this.val = val ; this.next = next ;
  }
}
```

la fonction suivante résout le problème.

```
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;
}
```

Si notre mesure élémentaire μ correspond au nombre de comparaisons entre entiers (nous ne comptons pas les comparaisons entre variables de type référence sur le type List) nous en faisons 1 par itération de la boucle, qui est exécutée $n-1$ fois, soit au total $n-1$.

La complexité $\mu(\mathcal{A}, n)$ de cet algorithme $\mathcal{A}$ sur les entrés de taille n est donc $n-1$.

10.3.3 Complexité du problème

On peut se poser la question de savoir s'il est possible de faire moins de $n-1$ telles comparaisons : la réponse est non. En effet, cet algorithme est optimal en nombre de comparaisons.

En effet, considérons la classe $\mathcal{C}$ des algorithmes qui résolvent le problème de la recherche du maximum de n éléments en utilisant comme critère de décision les comparaisons entre éléments.

Commençons par énoncer la propriété suivante : tout algorithme $\mathcal{A}$ de $\mathcal{C}$ est tel que tout élément autre que le maximum est comparé au moins une fois avec un élément qui lui est plus grand.

En effet, soit i_0 le rang du maximum M retourné par l'algorithme sur une liste $L = e_1.e_2.\cdots.e_n : e_{i_0} = M = \max_{1\leq i\leq n} e_i$. Raisonnons par l'absurde : soit $j_0 \neq i_0$ tel que e_{j_0} ne soit pas comparé avec un élément plus grand que lui. L'élément e_{j_0} n'a donc pas été comparé avec e_{i_0} le maximum.

Considérerons la liste $L' = e_1.e_2.\cdots.e_{j_0-1}.M+1.e_{j_0+1}.\cdots.e_n$ obtenue à partir de L en remplaçant l'élément d'indice j_0 par $M+1$.

L'algorithme $\mathcal{A}$ effectuera exactement les mêmes comparaisons sur L et L', sans comparer $L'[j_0]$ avec $L'[i_0]$ et donc retournera $L'[i_0]$, ce qui est incorrect. D'où une contradiction, qui prouve la propriété.

Il découle de la propriété qu'il n'est pas possible de déterminer le maximum de n-éléments en moins de $n-1$ comparaisons entre entiers. Autrement dit, la complexité du problème $\mathcal{P}$ du calcul du maximum sur les entrées de taille n est $\mu(\mathcal{P}, n) = n-1$.

L'algorithme précédent fonctionnant en $n-1$ telles comparaisons, il est optimal pour cette mesure de complexité.

10.3.4 Complexité de l'algorithme en moyenne

Si notre mesure élémentaire μ correspond au nombre d'affectations à l'intérieur de la boucle **for**, on voit que ce nombre dépend de la donnée.

On peut s'intéresser à sa complexité en moyenne : il faut faire une hypothèse sur la distribution des entrées. Supposons que les listes en entrées dont on cherche à calculer le maximum sont des permutations de $\{1, 2, \cdots, n\}$, et que les $n!$ permutations sont équiprobables.

On peut montrer [Sedgewick and Flajolet, 1996, Froidevaux et al., 1993] que la complexité moyenne sur les entrées de taille n pour cette mesure élémentaire μ est alors donnée par H_n, le $n$ième nombre harmonique : $H_n = 1+\frac{1}{2}+\frac{1}{3}+\cdots\frac{1}{n}$. H_n est de l'ordre de $\log n$ lorsque n tend vers l'infini.

Cependant, le calcul est laborieux, et pas si intéressant à développer dans ce polycopié, sans qu'il ne paraisse obscur.

Simplifions, en nous intéressons à un problème encore plus simple : plutôt que de rechercher le maximum dans la liste $e_1, e_2, \cdots, e_n$, avec $n \geq 1$, donnons nous cette liste et un entier v, et cherchons à déterminer s'il existe un indice $1 \leq i \leq n$ avec $e_i = v$.

L'algorithme suivant résout le problème.

```
static boolean trouve(int[] T, int v) {
    for (int i = 0; i < T.length; i++)
        if (T[i] == v)
            return true;
```

```
        return false;
    }
```

Sa complexité au pire cas en nombre d'instructions élémentaires est linéaire en n, puisque la boucle est effectuée n fois dans le pire cas.

Supposons que les listes en entrées sont des permutations de $\{1, 2, \cdots, n\}$, et que les $n!$ permutations sont équiprobables.

Remarquons qu'il y a k^n tableaux. Parmi ceux-ci, $(k-1)^n$ ne contiennent pas `v` et dans ce cas, l'algorithme procède à exactement n itérations. Dans le cas contraire, l'entier est dans le tableau et sa première occurrence est alors i avec une probabilité de

$$\frac{(k-1)^{i-1}}{k^i}$$

et il faut alors procéder à i itérations. Au total, nous avons une complexité moyenne de

$$C = \frac{(k-1)^n}{k^n} \times n + \sum_{i=1}^{n} \frac{(k-1)^{i-1}}{k^i} \times i$$

Or

$$\forall x, \sum_{i=1}^{n} i x^{i-1} = \frac{1 + x^n(nx - n - 1)}{(1-x)^2}$$

(il suffit pour établir ce résultat de dériver $\sum_{i=1}^{n} x^i = \frac{1-x^{n+1}}{1-x}$) et donc

$$C = n\frac{(k-1)^n}{k^n} + k\left(1 - \frac{(k-1)^n}{k^n}(1 + \frac{n}{k})\right) = k\left(1 - \left(1 - \frac{1}{k}\right)^n\right)$$

10.4 Asymptotiques

10.4.1 Complexités asymptotiques

On le voit sur l'exemple précédent, réaliser une étude précise et complète de complexité est souvent fastidieux, et parfois difficile. Aussi, on s'intéresse en informatique plutôt à l'ordre de grandeur (l'asymptotique) des complexités quand la taille n des entrées devient très grande.

10.4.2 Notations de Landau

Comme il en est l'habitude en informatique, on travaille souvent à un ordre de grandeur près, via les notations $\mathcal{O}$. Rappelons les définitions suivantes :

Définition 10.1 (Notation $\mathcal{O}$) *Soient f et g deux fonctions $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. On note $f(n) = \mathcal{O}(g(n))$ lorsqu'il existe des entiers c et n_0 tels que pour tout $n \geq n_0$,*

$$f(n) \leq cg(n).$$

Intuitivement, cela signifie que f est inférieur à g à une constante multiplicative près, pour les instances (données) de tailles suffisamment grandes.

De même on définit :

Définition 10.2 (Notations o, Ω, Θ) *Soient f et g deux fonctions $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$.*

- *On note $f(n) = o(g(n))$ lorsque pour tout réel c, il existe un entier n_0 tels que pour tout $n \geq n_0$,*
$$f(n) < cg(n).$$
- *On note $f(n) = \Omega(g(n))$ lorsqu'il existe des entiers c et n_0 tels que pour tout $n \geq n_0$,*
$$cg(n) \leq f(n).$$
- *On note $f(n) = \Theta(g(n))$ lorsque $f(n) = \mathcal{O}(g(n))$ et $f(n) = \Omega(g(n))$.*

10.5 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture du polycopié du cours INF421, ou de de l'ouvrage [Kleinberg and Tardos, 2005], et en particulier de ses premiers chapitres.

Bibliographie Le texte de ce chapitre est repris du texte que nous avions écrit dans le polycopié INF421, et inspiré de l'introduction de [Kleinberg and Tardos, 2005]. L'analyse du problème du calcul du maximum, et de ses variations est basée sur [Froidevaux et al., 1993].

Chapitre 11

Complexité en temps

Ce chapitre se focalise sur l'étude d'une ressource particulière élémentaire d'un algorithme : le temps qu'il prend pour s'exécuter.

Le chapitre précédent s'applique en particulier à cette mesure : le temps de calcul d'un algorithme est défini comme le temps qu'il prend pour s'exécuter.

Pour appuyer et illustrer l'idée de l'importance de la mesure de cette notion de complexité, intéressons nous au temps correspondant à la complexité des algorithmes n, $n \log_2 n$, n^2, n^3, 1.5^n, 2^n et $n!$ pour des entrées de taille n croissantes, sur un processeur capable d'exécuter un million d'instructions élémentaires par seconde. Nous notons ∞ dans le tableau suivant dès que le temps dépasse 10^{25} années (ce tableau est repris de [Kleinberg and Tardos, 2005]).

Complexité	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} ans
$n = 50$	< 1 s	< 1 s	< 1 s	< 1 s	11 min	36 ans	∞
$n = 100$	< 1 s	< 1 s	< 1 s	$1s$	$12,9$ ans	10^{17} ans	∞
$n = 1000$	< 1 s	< 1 s	1s	18 min	∞	∞	∞
$n = 10000$	< 1 s	< 1 s	2 min	12 jours	∞	∞	∞
$n = 100000$	< 1 s	2 s	3 heures	32 ans	∞	∞	∞
$n = 1000000$	1s	20s	12 jours	$31,710$ ans	∞	∞	∞

On le voit, un algorithme de complexité exponentielle est très rapidement inutilisable, et donc pas *très raisonnable*. Tout l'objet du chapitre est de comprendre ce que l'on appelle un algorithme *raisonnable* en informatique, et de comprendre la théorie de la NP-complétude qui permet de discuter la frontière entre le raisonnable et le non raisonnable.

11.1 La notion de temps raisonnable

11.1.1 Convention

Pour différentes raisons, la convention suivante s'est imposée en informatique :

Définition 11.1 (Algorithme efficace) *Un algorithme est* efficace *si sa complexité en temps est polynomiale, c'est-à-dire en $\mathcal{O}(n^k)$ pour un entier k.*

Il ne s'agit que d'une convention, et on aurait pu en choisir d'autres (et à vrai dire il y en a eu d'autres avant celle-là, qui s'est imposée dans les années 1970).

Remarque 11.1 *On peut argumenter qu'un algorithme de complexité $\mathcal{O}(n^{1794})$ n'est pas* très raisonnable*. Certes, mais il faut bien fixer une convention. On considère que c'est raisonnable en théorie de la complexité.*

Remarque 11.2 *Pourquoi ne pas prendre un temps linéaire, ou un temps quadratique comme notion de "raisonnable" : parce que cela ne fonctionne pas bien. En particulier, ces notions de temps linéaires et quadratiques ne vérifieraient pas la "Deuxième raison : s'affranchir du modèle de calcul" évoquée plus bas : la notion de temps linéaire ou de temps quadratique dépend du modèle de calcul utilisé, contrairement à la notion de calcul en temps polynomial.*

11.1.2 Première raison : s'affranchir du codage

Une des raisons de cette convention est la remarque suivante : la plupart des objets informatiques usuels peuvent se représenter de différentes façons, mais passer d'une façon de les représenter à l'autre est possible en un temps qui reste polynomial en la taille du codage.

La classe des polynômes étant stable par composition, cela implique qu'un algorithme qui est polynomial et qui travaille sur une représentation se transforme en un algorithme polynomial qui travaille sur toute autre représentation de l'objet.

On peut donc parler d'algorithme *efficace* sur ces objets sans avoir à rentrer dans les détails de comment on écrit ces objets.

Exemple 11.1 *Un graphe peut se représenter par une matrice, sa* matrice d'adjacence *: si le graphe à n sommets, on considère un tableau T de taille n par n, dont l'élément $T[i][j] \in \{0,1\}$ vaut 1 si et seulement s'il y a une arête entre le sommet i et le sommet j.*

On peut aussi représenter un graphe par des listes d'adjacence *: pour représenter un graphe à n sommets, on considère n listes. La liste de numéro i code les voisins du sommet numéro i.*

On peut passer d'une représentation à l'autre en un temps qui est polynomial en la taille de chacune : on laisse le lecteur se persuader de la chose dans son langage de programmation préféré.

Un algorithme efficace *pour l'une des représentations peut toujours se transformer en un algorithme* efficace *pour l'autre représentation : il suffit de commencer éventuellement par traduire la représentation en la représentation sur laquelle travaille l'algorithme.*

Remarque 11.3 *Par ailleurs, par les mêmes remarques, puisque chacune de ces représentations est de taille polynomiale en n, en utilisant le fait qu'un graphe à n sommet a au plus n^2 arêtes, un algorithme polynomial en n n'est rien d'autre qu'un algorithme efficace qui travaille sur les graphes, c'est-à-dire sur les représentations précédentes, ou toutes les représentations usuelles des graphes.*

11.1.3 Deuxième raison : s'affranchir du modèle de calcul

Une deuxième raison profonde est la suivante : revenons sur le chapitre 7. Nous avons montré que tous les modèles de calculs de ce chapitre se simulaient l'un et l'autre : machines RAM, machines de Turing, machines à piles, machines à compteurs.

Si l'on met de côté les machines à compteur dont la simulation est particulièrement inefficace, et dont l'intérêt n'est que théorique, on peut remarquer qu'un nombre t d'instructions pour l'un se simule en utilisant un nombre polynomial en t d'instructions pour l'autre. La classe des polynômes étant stable par composition, cela implique qu'un algorithme qui est polynomial dans un modèle de calcul se transforme en un algorithme polynomial en chacun des autres modèles de calcul (quitte à simuler l'un par l'autre).

On peut donc parler d'algorithme *efficace* sur un objet sans avoir à préciser si l'on programme l'algorithme dans un modèle de calcul ou dans un autre modèle de calcul[1].

En particulier, la notion d'algorithme efficace est indépendante du langage de programmation utilisé : un algorithme efficace en CAML est un algorithme efficace en JAVA, ou un algorithme efficace en C.

Puisque la notion d'efficacité ne dépend pas du modèle, on va donc utiliser celui de la machine de Turing dans tout ce qui suit : lorsque w est un mot, on note $|w|$ sa longueur.

Définition 11.2 (TIME(t(n))**)** *Soit $t : \mathbb{N} \to \mathbb{N}$ une fonction. On définit la classe* TIME(t(n)) *comme la classe des problèmes (langages) décidés par une machine de Turing en temps $\mathcal{O}(t(n))$, où n est la taille de l'entrée.*

Si on préfère, $L \in$ TIME(t(n)) s'il y a une machine de Turing M telle que

[1] Les plus puristes auront remarqué un problème avec le modèle des machines RAM du chapitre 7 : il faut prendre en compte dans la complexité la taille des entiers dans les opérations élémentaires effectuées et pas seulement le nombre d'instructions. Mais c'est de l'ergotage, et ce qui est écrit au dessus reste totalement vrai, si l'on interdit aux machines RAM de manipuler des entiers de taille arbitraire. De toute façon, ne pas le faire ne serait pas réaliste par rapport aux processeurs qu'ils entendent modéliser qui travaillent sur des entiers codés sur un nombre fini de bits (32 ou 64 par exemple).

- M décide L : pour tout mot w, M accepte w si et seulement si $w \in L$, et M refuse w si et seulement si $w \notin L$;
- M prend un temps borné par $\mathcal{O}(t(n))$:
  - si l'on préfère : il y a des entiers n_0, c, et k tels que pour tout mot w de taille suffisamment grande, i.e. $|w| \geq n_0$, et pour tout mot w, si on note $n = |w|$ sa longueur, M accepte ou refuse en utilisant au plus $c * n^k$ étapes.

Remarque 11.4 *On s'intéresse dans ce chapitre et dans le suivant (et en complexité) uniquement à des problèmes décidables.*

11.1.4 Classe P

La classe des problèmes qui admettent un algorithme raisonnable correspond alors à la classe suivante.

Définition 11.3 (Classe P) *La classe* P *est la classe des problèmes (langages) définie par :*

$$\mathrm{P} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME}(\mathrm{n}^{\mathrm{k}}).$$

Autrement-dit, P est exactement les problèmes qui admettent un algorithme polynomial.

Voici quelques exemples de problèmes de P.

Exemple 11.2 (Tester le coloriage d'un graphe) ***Donnée****: Un graphe $G = (V, E)$ avec la donnée pour chaque sommet $v \in V$ d'une couleur parmi un nombre fini de couleurs.*

Réponse*: Décider si G est* colorié *avec ces couleurs : c'est-à-dire si il n'y a pas d'arête de G avec deux extrémités de la même couleur.*

Ce problème est dans la classe P*. En effet, il suffit de parcourir les arêtes du graphe et de tester pour chacune si la couleur de ses extrémités est la même.*

Exemple 11.3 (Evaluation en calcul propositionnel) ***Donnée****: Une formule $F(x_1, x_2, \cdots, x_n)$ du calcul propositionnel, des valeurs $x_1, \cdots, x_n \in \{0, 1\}$ pour chacune des variables de la formule.*

Réponse*: Décider si la formule F s'évalue en vraie pour ces valeurs des variables.*

Ce problème est dans la classe P*. En effet, étant donnée une formule du calcul propositionnel $F(x_1, \cdots, x_n)$ et des valeurs pour $x_1, x_2, \cdots, x_n \in \{0, 1\}$, il est facile de calculer la valeur de vérité de $F(x_1, \cdots, x_n)$. Cela se fait en un temps que l'on vérifie facilement comme polynomial en la taille de l'entrée.*

Beaucoup d'autres problèmes sont dans P.

11.2 Comparer les problèmes

11.2.1 Motivation

Il s'avère toutefois qu'il y a toute une classe de problèmes pour lesquels à ce jour on n'arrive pas à construire d'algorithme polynomial, mais sans qu'on arrive à prouver formellement que cela ne soit pas possible.

C'est historiquement ce qui a mené à considérer la classe de problèmes que l'on appelle NP, que nous verrons dans la section suivante.

Des exemples de problèmes dans cette classe sont les suivants :

Exemple 11.4 (k-COLORABILITE**)** ***Donnée:*** *Un graphe $G = (V, E)$.*
Réponse: *Décider s'il existe un coloriage du graphe utilisant au plus k couleurs : c'est-à-dire s'il existe une façon de colorer les sommets de G (avec au plus k couleurs) tel que l'on obtienne un coloriage de G.*

Exemple 11.5 (SAT**, Satisfaction en calcul propositionnel)** ***Donnée:*** *Une formule $F = (x_1, \cdots, x_n)$ du calcul propositionnel.*
Réponse: *Décider si F est satisfiable : c'est-à-dire décider s'il existe $x_1, \cdots, x_n \in \{0, 1\}$ tel que F s'évalue en vraie pour cette valeur de ses variables $x_1, \cdots, x_n$.*

Exemple 11.6 (CIRCUIT HAMILTONIEN**)** ***Donnée:*** *Un graphe $G = (V, E)$ (non-orienté).*
Réponse: *Décider s'il existe un* circuit hamiltonien*, c'est-à-dire un chemin de G passant une fois et une seule par chacun des sommets et revenant à son point de départ.*

Pour les trois problèmes on connaît des algorithmes exponentiels : tester tous les coloriages, pour le premier, ou toutes les valeurs de $\{0, 1\}^n$ pour le second, ou tous les chemins pour le dernier. Pour les trois problèmes on ne connaît pas d'algorithme efficace (polynomial), et on n'arrive pas à prouver qu'il n'y en a pas.

Comme nous allons le voir, on sait toutefois montrer que ces trois problèmes sont équivalents au niveau de leur difficulté, et cela nous amène à la notion de réduction, c'est-à-dire à l'idée de comparer la difficulté des problèmes.

Avant, quelques précisions.

11.2.2 Remarques

Dans ce chapitre et le suivant, on va ne parler essentiellement que de problèmes de décisions, c'est-à-dire de problèmes dont la réponse est soit "vrai ou "faux" : voir la définition 8.2.

Exemple 11.7 *"Trier n nombres" n'est pas un problème de décision : la sortie est à priori une liste triée de nombres.*

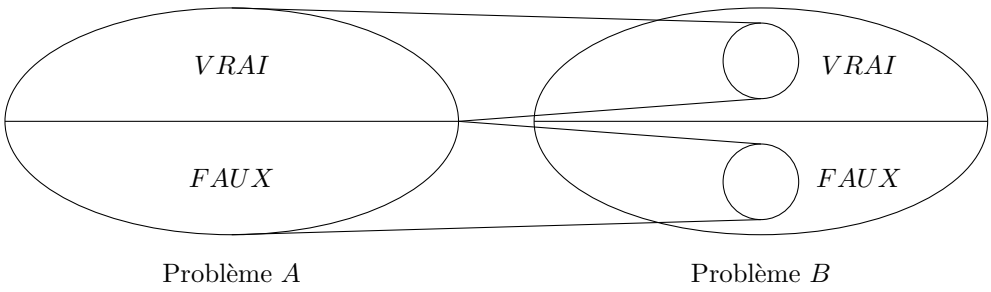


FIG. 11.1 – Les réductions transforment des instances positives en instances positives, et négatives en négatives.

Exemple 11.8 *"Étant donné un graphe $G = (V, E)$, déterminer le nombre de couleurs pour colorier G" n'est pas un un problème de décision, car la sortie est un entier. On peut toutefois formuler un problème de décision proche, du type "Étant donné un graphe $G = (V, E)$, et un entier k, déterminer si le graphe G admet un coloriage avec moins de k couleurs" : c'est le problème* k-COLORABILITE.

Avant de pouvoir parler de réduction, il faut aussi parler de fonctions calculables en temps polynomial : c'est la notion à laquelle on s'attend, même si on est obligé de l'écrire car on ne l'a encore jamais fait.

Définition 11.4 (Fonction calculable en temps polynomial) *Soient Σ et Σ' deux alphabets. Une fonction $f : \Sigma^* \to \Sigma'^*$ est* calculable en temps polynomial *s'il existe une machine de Turing A, qui travaille sur l'alphabet $\Sigma \cup \Sigma'$, et un entier k, telle que pour tout mot w, A avec l'entrée w termine en un temps $\mathcal{O}(n^k)$ avec le résultat $f(w)$, où $n = |w|$.*

Le résultat suivant est facile à établir :

Proposition 11.1 (Stabilité par composition) *La composée de deux fonctions calculables en temps polynomial est calculable en temps polynomial.*

11.2.3 Notion de réduction

Cela nous permet d'introduire une notion de réduction entre problèmes (similaire à celle du chapitre 8, si ce n'est que l'on parle de calculable en temps polynomial plutôt que de calculable) : l'idée est que si A se réduit à B, alors le problème A est plus facile que le problème B, ou si l'on préfère, le problème B est plus difficile que le problème A : voir la figure 11.2 et la figure 11.1.

Définition 11.5 (Réduction) *Soient A et B deux problèmes d'alphabet respectifs M_A et M_B. Une* réduction de A vers B *est une fonction $f : M_A^* \to M_B^*$*

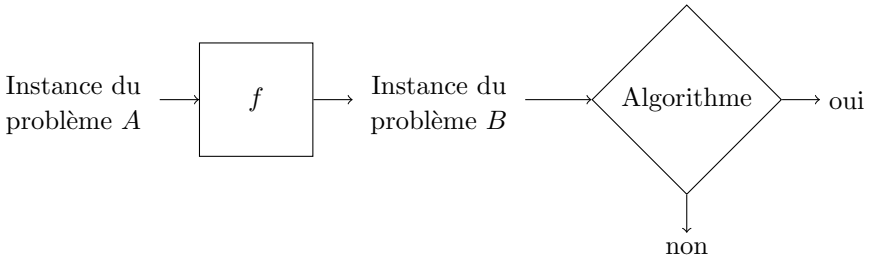


FIG. 11.2 – Réduction du problème A vers le problème B. Si l'on peut résoudre le problème B en temps polynomial, alors on peut résoudre le problème A en temps polynomial. Le problème A est donc plus facile que le problème B, noté $A \leq B$.

calculable en temps polynomial telle que $w \in A$ ssi $f(w) \in B$. On note $A \leq B$ lorsque A se réduit à B.

Cela se comporte comme on le souhaite : un problème est aussi facile (et difficile) que lui-même, et la relation "être plus facile que" est transitive. En d'autres termes :

Théorème 11.1 *$\leq$ est un préordre :*

1. *$L \leq L$;*
2. *$L_1 \leq L_2$, $L_2 \leq L_3$ impliquent $L_1 \leq L_3$.*

Démonstration: Considérer la fonction identité comme fonction f pour le premier point.

Pour le second point, supposons $L_1 \leq L_2$ via la réduction f, et $L_2 \leq L_3$ via la réduction g. On a $x \in L_1$ ssi $g(f(x)) \in L_2$. Il suffit alors de voir que $g \circ f$, en temps que composée de deux fonctions calculables en temps polynomial est calculable en temps polynomial. □

Remarque 11.5 *Il ne s'agit pas d'un ordre, puisque $L_1 \leq L_2$, $L_2 \leq L_1$ n'implique pas $L_1 = L_2$.*

Il est alors naturel d'introduire :

Définition 11.6 *Deux problèmes L_1 et L_2 sont équivalents, noté $L_1 \equiv L_2$, si $L_1 \leq L_2$ et si $L_2 \leq L_1$.*

On a alors $L_1 \leq L_2$, $L_2 \leq L_1$ impliquent $L_1 \equiv L_2$.

11.2.4 Application à la comparaison de difficulté

Intuitivement, si un problème est plus facile qu'un problème polynomial, alors il est polynomial. Formellement.

Proposition 11.2 (Réduction) *Si $A \leq B$, et si $B \in \mathrm{P}$ alors $A \in \mathrm{P}$.*

Démonstration: Soit f une réduction de A vers B. A est décidé par la machine de Turing qui, sur une entrée w, calcule $f(w)$, puis simule la machine de Turing qui décide B sur l'entrée $f(w)$. Puisqu'on a $w \in A$ si et seulement si $f(w) \in B$, l'algorithme est correct, et fonctionne bien en temps polynomial. □

En prenant la contraposée de la proposition précédente, on obtient la formulation suivante qui dit que si un problème n'a pas d'algorithme polynomial, et qu'il est plus facile qu'un autre, alors l'autre non plus.

Proposition 11.3 (Réduction) *Si $A \leq B$, et si $A \notin \mathrm{P}$ alors $B \notin \mathrm{P}$.*

Exemple 11.9 *Nous verrons que le problème d'un coloriage d'un graphe, de la satisfaction du calcul propositionnel, ou de l'existence d'un chemin hamiltonien sont équivalents (et équivalents à tous les problèmes* NP*-complets). Il y a donc un algorithme efficace pour l'un ssi il y en a un pour l'autre.*

11.2.5 Problèmes les plus durs

Si on considère une classe de problèmes, on peut introduire la notion de problème le plus difficile pour la classe, i.e. maximal pour $\leq$. C'est la notion de *complétude* :

Définition 11.7 ($\mathcal{C}$-complétude) *Soit $\mathcal{C}$ une classe de problèmes de décisions. Un problème A est dit $\mathcal{C}$-complet, si*

1. *il est dans $\mathcal{C}$;*
2. *tout autre problème B de $\mathcal{C}$ est tel que $B \leq A$.*

On dit qu'un problème A est *$\mathcal{C}$-dur* s'il vérifie la condition 2 de la définition 11.7. Un problème A est donc $\mathcal{C}$-complet s'il est $\mathcal{C}$-dur et dans la classe $\mathcal{C}$.

Un problème $\mathcal{C}$-complet est donc le plus difficile, ou un des plus difficiles, de la classe $\mathcal{C}$. Clairement, s'il y en a plusieurs, ils sont équivalents :

Corollaire 11.1 *Soit $\mathcal{C}$ une classe de langages. Tous les problèmes $\mathcal{C}$-complets sont équivalents.*

Démonstration: Soient A et B deux problèmes $\mathcal{C}$-complets. Appliquer la condition 2 de la définition 11.7 en A relativement à $B \in \mathcal{C}$, et en B relativement à $A \in \mathcal{C}$. □

11.3 La classe NP

11.3.1 La notion de vérificateur

Les problèmes k-COLORABILITE, SAT et CIRCUIT HAMILTONIEN évoqués précédemment ont un point commun : s'il n'est pas clair, qu'ils admettent un algorithme polynomial, il est clair qu'ils admettent un *vérificateur* polynomial.

Définition 11.8 (Vérificateur) *Un* vérificateur *pour un problème A est un algorithme V tel que*

$$A = \{w | V \text{ accepte } \langle w, u \rangle \text{ pour un certain mot } u\}.$$

Le vérificateur est polynomial si V se décide en temps polynomial en la longueur de w. On dit qu'un langage est polynomialement vérifiable *s'il admet un vérificateur polynomial.*

Remarque 11.6 *Remarquons que l'on peut toujours se restreindre aux certificats de longueur polynomiale en la longueur de w, puisqu'en temps polynomial en la longueur de w l'algorithme V ne pourra pas lire plus qu'un nombre polynomial de symboles du certificat.*

Le mot u est alors appelé un *certificat* (parfois aussi une *preuve*) pour w. Autrement dit, un vérificateur utilise une information en plus, à savoir u pour vérifier que w est dans A.

Exemple 11.10 *Un certificat pour le problème* k-COLORABILITE *est la donnée des couleurs pour chaque sommet.*

Nous ne donnerons pas toujours autant de détails, mais voici la justification : en effet, un graphe G est dans k-COLORABILITE si et seulement si on peut trouver un mot u qui code des couleurs pour chaque sommet tel que ces couleurs donnent un coloriage correct : l'algorithme V, i.e. le vérificateur, se contente, étant donné $\langle G, u \rangle$ de vérifier que le coloriage est correct, ce qui se fait bien en temps polynomial en la taille du graphe.

Exemple 11.11 *Un certificat pour le problème* SAT *est la donnée d'une valeur $x_1, \cdots, x_n \in \{0, 1\}$ pour chacune des variables de la formule F : le vérificateur, qui se contente de vérifier que ces valeurs satisfont la formule F, peut bien se réaliser en temps polynomial en la taille de la formule.*

Exemple 11.12 *Un certificat pour le problème* CIRCUIT HAMILTONIEN *est la donnée d'un circuit. Le vérificateur se contente de vérifier que le circuit est hamiltonien, ce qui se fait bien en temps polynomial.*

Cela nous amène à la définition suivante :

Définition 11.9 NP *est la classe des problèmes (langages) qui possèdent un vérificateur polynomial.*

Cette classe est importante car elle s'avère contenir un nombre incroyable de problèmes d'intérêt pratique. Elle contient k-COLORABILITE, SAT et CIRCUIT HAMILTONIEN mais aussi beaucoup d'autres problèmes : voir par exemple tout le chapitre qui suit.

Par construction, on a (car le mot vide est par exemple un certificat pour tout problème de P) :

Proposition 11.4 $\mathrm{P} \subseteq \mathrm{NP}$.

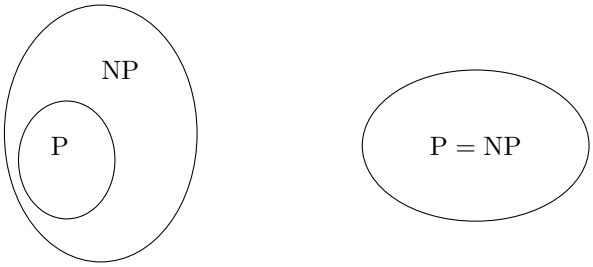


FIG. 11.3 – Une des deux possibilités est correcte.

11.3.2 La question P = NP ?

Clairement, on a soit P = NP soit P $\subsetneq$ NP : voir la figure 11.3.

La question de savoir si ces deux classes sont égales ou distinctes est d'un enjeu impressionnant. D'une part parce que c'est l'une des questions (voir la question) non résolue la plus célèbre de l'informatique théorique et des mathématiques qui défie les chercheurs depuis plus de 40 ans : elle a été placée parmi la liste des questions les plus importantes pour les mathématiques et l'informatique pour le millénaire en 2000. Le *Clay Mathematics Institute* offre $1,000,000$ de dollars à qui déterminerait la réponse à cette question.

Surtout, si P = NP, alors tous les problèmes vérifiables polynomialement seraient décidables en temps polynomial. La plupart des personnes pensent que ces deux classes sont distinctes car il y a un très grand nombre de problèmes pour lesquels on n'arrive pas à produire d'algorithme polynomiaux depuis plus de 40 ans.

Elle a aussi un enjeu économique impressionnant puisque de nombreux systèmes, comme les systèmes de cryptographie actuels sont basés sur l'hypothèse que ces deux classes sont distinctes : si ce n'était pas le cas, de nombreuses considérations sur ces systèmes s'effondreraient, et de nombreuses techniques de cryptage devraient être revues.

11.3.3 Temps non déterministe polynomial

Faisons avant une petite parenthèse sur la terminologie : le "N" dans NP vient de *non déterministe* (et pas de *non*, ou *not* comme souvent beaucoup le croient), en raison du résultat suivant :

Théorème 11.2 *Un problème est dans* NP *si et seulement s'il est décidé par une machine de Turing non déterministe en temps polynomial.*

Rappelons que nous avons introduit les machines de Turing non déterministes dans le chapitre 7. On dit qu'un langage $L \subset \Sigma^*$ est *décidé par la machine non déterministe M en temps polynomial* si M décide L et M prend un temps borné par $\mathcal{O}(t(n))$: il y a des entiers n_0, c, et k tels que pour tout mot w de taille suffisamment grande, i.e. $|w| \geq n_0$, et pour tout mot w, si on note $n = |w|$

sa longueur, pour $w \in L$, M admet un calcul qui accepte en utilisant moins de $c * n^k$ étapes, et pour $w \notin L$, tous les calculs de M mènent à une configuration de refus en moins de $c * n^k$ étapes.

Démonstration: Considérons un problème A de NP. Soit V le vérificateur associé, qui fonctionne en temps polynomial $p(n)$. On construit une machine de Turing M non déterministe qui, sur un mot w, va produire de façon non déterministe un mot u de longueur $p(n)$ puis simuler V sur $\langle w, u\rangle$: si V accepte, alors M accepte. Si V refuse, alors M refuse. La machine M décide A.

Réciproquement, soit A un problème décidé par une machine de Turing non déterministe M en temps polynomial $p(n)$. Comme dans la preuve de la proposition 7.3 dans le chapitre 8, on peut affirmer que le degré de non déterminisme de la machine est borné, et qu'il vaut un certain entier r, et que les suites des choix non déterministes réalisées par la machine M jusqu'au temps t se codent par une suite de longueur t d'entiers entre 1 à (au plus) r.

Par conséquent, une suite d'entiers de longueur $p(n)$ entre 1 et r est un certificat valide pour un mot w : étant donné w et un mot u codant une telle suite, un vérificateur V peut facilement vérifier en temps polynomial si la machine M accepte w avec ces choix non déterministes. □

Plus généralement, on définit :

Définition 11.10 (TIME(t(n))**)** *Soit $t : \mathbb{N} \to \mathbb{N}$ une fonction. On définit la classe* NTIME(t(n)) *comme la classe des problèmes (langages) décidés par une machine de Turing non déterministe en temps $\mathcal{O}(t(n))$, où n est la taille de l'entrée.*

Corollaire 11.2

$$\mathrm{NP} = \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}(\mathrm{n}^{\mathrm{k}}).$$

11.3.4 NP-complétude

Il s'avère que la classe NP possède une multitude de problèmes complets : le prochain chapitre en présente toute une liste.

La difficulté est d'arriver à en produire un premier. C'est l'objet du théorème de Cook et Levin.

Théorème 11.3 (Cook-Levin) *Le problème* SAT *est* NP*-complet.*

Nous prouverons ce théorème dans la section qui suit.

Commençons par reformuler ce que cela signifie.

Corollaire 11.3 P = NP *si et seulement si* SAT$\in$ P.

Démonstration: Puisque SAT est dans NP, si P = NP, alors SAT$\in$ P.

Réciproquement, puisque SAT est complet, pour tout problème $B \in$ NP, $B \leq$ SAT et donc $B \in$ P si SAT $\in$ P par la proposition 11.2. □

Ce que nous venons de faire est vrai pour n'importe quel problème NP-complet.

Théorème 11.4 *Soit A un problème* NP*-complet.*
$\mathrm{P} = \mathrm{NP}$ *si et seulement si* $A \in \mathrm{P}$.

Démonstration: Puisque A est complet il est dans NP, et donc si $\mathrm{P} = \mathrm{NP}$, alors $A \in \mathrm{P}$. Réciproquement, puisque A est NP-dur, pour tout problème $B \in \mathrm{NP}$, $B \leq A$ et donc $B \in \mathrm{P}$ si $A \in \mathrm{P}$ par la proposition 11.2. □

Remarque 11.7 *On voit donc tout l'enjeu de produire des problèmes* NP*-complets pour la question de prouver* $\mathrm{P} \neq \mathrm{NP}$ *: tenter de produire un problème pour lequel on arriverait à prouver qu'il n'existe pas d'algorithme polynomial. A ce jour, aucun des centaines, voir des milliers de problèmes* NP*-complets connus n'ont permis cependant de prouver que* $\mathrm{P} \neq \mathrm{NP}$.

Remarque 11.8 *Rappelons nous que tous les problèmes complets sont équivalents en difficulté par le corollaire 11.1.*

11.3.5 Méthode pour prouver la NP-complétude

La NP-complétude d'un problème s'obtient dans la quasi-totalité des cas de la façon suivante :

Théorème 11.5 *Pour prouver la NP-complétude d'un problème A, il suffit :*

1. *de prouver qu'il est dans* NP *;*
2. *et de prouver que $B \leq A$ pour un problème B que l'on sait déjà* NP*-complet.*

Démonstration: En effet, le point 1. permet de garantir que $B \in \mathrm{NP}$, et le point 2. de garantir que pour tout problème $C \in \mathrm{NP}$ on a $C \leq A$: en effet, par la NP-complétude de B on a $C \leq B$, et puisque $B \leq A$, on obtient $C \leq A$. □

Remarque 11.9 *Attention, la* NP*-complétude d'un problème A s'obtient en prouvant qu'il est plus difficile qu'un autre problème* NP*-complet, et pas le contraire. C'est une erreur fréquente dans les raisonnements.*

Le chapitre suivant est consacré à de multiples applications de cette stratégie pour différents problèmes.

11.3.6 Preuve du théorème de Cook-Levin

Nous ne pouvons pas appliquer la méthode précédente pour prouver la NP-complétude de SAT, car nous ne connaissons pas encore aucun problème NP-complet.

Il nous faut donc faire une preuve autrement, en revenant à la définition de la NP-complétude : il faut prouver d'une part que SAT est dans NP, et d'autre part que tout problème A de NP vérifie $A \leq \mathrm{SAT}$.

Le fait que SAT est dans NP a déjà été établi : voir l'exemple 11.11.

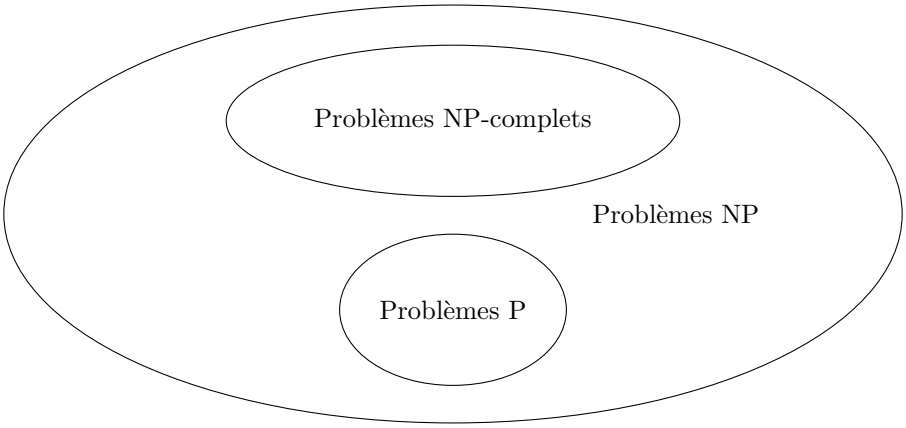


FIG. 11.4 – Situation avec l'hypothèse P $\neq$ NP.

Considérons un problème A de NP, et un vérificateur V associé. L'idée (qui a certaines similarités avec les constructions du chapitre 9) est, étant donné un mot w, de construire une formule propositionnelle $\gamma = \gamma(u)$ qui code l'existence d'un calcul accepteur de V sur $\langle w, u\rangle$ pour un certificat u.

On va en fait construire une série de formules dont le point culminant sera la formule $\gamma = \gamma(u)$ qui codera l'existence d'une suite de configurations $C_0, C_1, \cdots, C_t$ de M telle que :

- C_0 est la configuration initiale de V sur $\langle w, u\rangle$;
- C_{i+1} est la configuration successeur de C_i, selon la fonction de transition δ de la machine de Turing V, pour $i < t$;
- C_t est une configuration acceptante.

En d'autres termes, l'existence d'un diagramme espace-temps valide correspondant à un calcul de V sur $\langle w, u\rangle$.

En observant que la formule propositionnelle obtenue γ reste de taille polynomiale en la taille de w, et peut bien s'obtenir par un algorithme polynomial à partir de w, on aura prouvé le théorème : en effet, on aura $w \in L$ si et seulement s'il existe u qui satisfait $\gamma(u)$, c'est-à-dire $A \leq$ SAT via la fonction f qui à w associe $\gamma(u)$.

Il ne reste plus qu'à donner les détails fastidieux de la construction de la formule $\gamma(u)$. Par hypothèse, V fonctionne en temps polynomial $p(n)$ en la taille n de w. En ce temps là, V ne peut pas déplacer sa tête de lecture de plus de $p(n)$ cases vers la droite, ou de $p(n)$ cases vers la gauche. On peut donc se restreindre à considérer un sous-rectangle de taille $(2 * p(n) + 1) \times p(n)$ du diagramme espace-temps du calcul de V sur $\langle w, u\rangle$: voir la figure 11.5.

Les cases du tableau $T[i, j]$ correspondant au diagramme espace temps sont des éléments de l'ensemble fini $C = \Gamma \cup Q$. Pour chaque $1 \leq i \leq p(n)$ et $1 \leq j \leq 2*p(n)+1$ et pour chaque $s \in C$, on définit une variable propositionnelle $x_{i,j,s}$. Si $x_{i,j,s}$ vaut 1, cela signifie que la case $T[i, j]$ du tableau contient s.

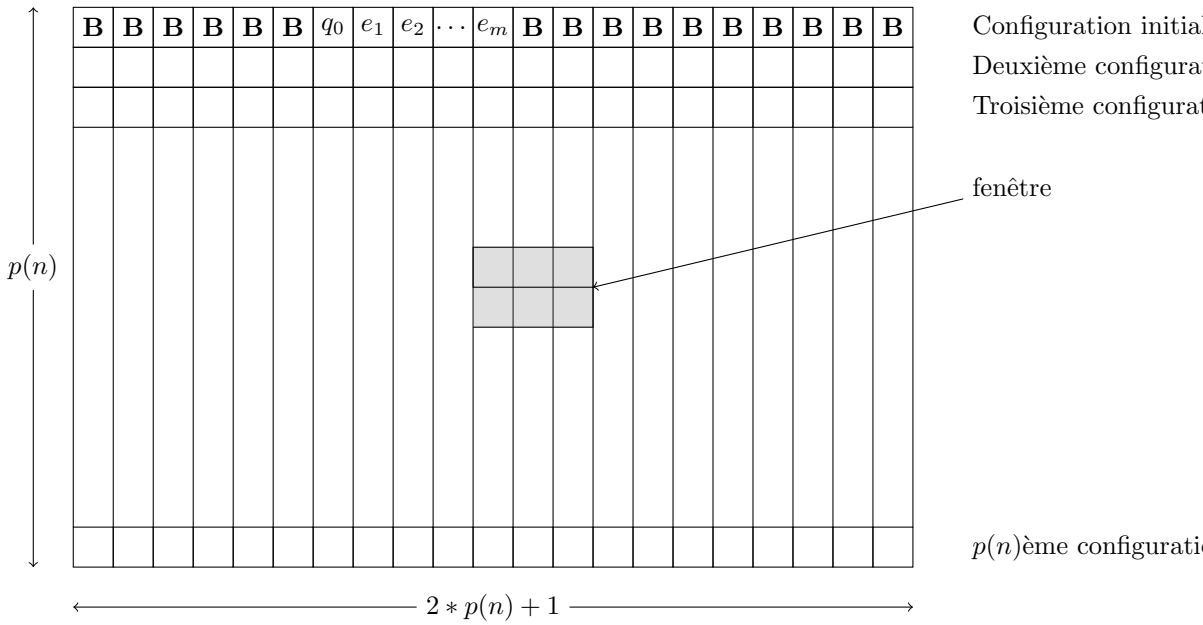


FIG. 11.5 – Un tableau $(2p(n) + 1) \times p(n)$ codant le diagramme espace-temps du calcul de V sur $\langle w, u \rangle$.

La formule Γ est la conjonction de 4 formules CELL $\wedge$ START $\wedge$ MOVE $\wedge$ HALT.

La formule CELL permet de garantir qu'il y a bien un symbole et un seul par case.

$$\text{CELL} = \bigwedge_{1\leq i\leq p(n), 1\leq j\leq 2p(n)+1} \left[\left(\bigvee_{s\in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{s,t\in C, s\neq t} (\neg x_{i,j,s} \vee \neg x_{i,j,t}) \right) \right].$$

Les symboles $\bigwedge$ et $\bigvee$ désigne l'itération des symboles $\wedge$ et $\vee$ correspondant. Par exemple, $\bigvee_{s\in C} x_{i,j,s}$ est un raccourci pour la formule $x_{i,j,s_1} \vee \cdots \vee x_{i,j,s_l}$ si $C = \{s_1, \cdots, s_l\}$.

Si on note le mot $e_1 e_2 \cdots e_m$ le mot $\langle w, u\rangle$, la formule START permet de garantir que la première ligne correspond bien à la configuration initiale du calcul de V sur $\langle w, u\rangle$.

$$\text{START} = x_{1,1,\mathbf{B}} \vee x_{1,2,\mathbf{B}} \vee \cdots \vee x_{1,p(n)+1,q_0} \vee x_{1,p(n)+2,e_1} \vee \cdots \vee x_{1,p(n)+m+1,e_m}$$

$$\vee x_{1,p(n)+m+2,\mathbf{B}} \vee \cdots \vee x_{1,2p(n)+1,\mathbf{B}}.$$

La formule HALT permet de garantir qu'une des lignes correspond bien à une configuration acceptante

$$\text{HALT} = \bigvee_{1\leq i\leq p(n), 1\leq j\leq 2p(n)+1} x_{i,j,q_a}.$$

Enfin la formule MOVE écrit que tous les sous-rectangles 3×2 du tableau T sont des fenêtres légales : voir la notion de fenêtre légale du chapitre 7.

$$\text{MOVE} = \bigwedge_{1\leq i\leq p(n), 1\leq j\leq 2p(n)+1} \text{LEGAL}_{i,j},$$

où $\text{LEGAL}_{i,j}$ est une formule propositionnelle qui exprime que le sous-rectangle 3×2 à la position i, j est une fenêtre légale :

$$\text{LEGAL}_{i,j} = \bigwedge_{(a,b,c,d,e,f)\in \text{WINDOW}} (x_{i,j-1,a} \wedge x_{i,j,b} \wedge x_{i,j+1,c} \wedge x_{i+1,j-1,d} \wedge x_{i+1,j,e} \wedge x_{i,j+1,f}),$$

où WINDOW est l'ensemble des 6-uplets (a, b, c, d, e, f) tels que si trois éléments de Σ représentés respectivement par a, b et c apparaissent consécutivement dans une configuration C_i, et si d, e, f apparaissent consécutivement aux mêmes emplacements dans la configuration C_{i+1}, alors cela est cohérent avec la fonction de transition δ de la machine de Turing M.

Ceci termine la preuve, si l'on ajoute que chacune de ces formules sont faciles à écrire (et donc produisibles facilement en temps polynomial à partir de w), et qu'elles restent bien de taille polynomiale en la taille de w.

11.4 Quelques autres résultats de la théorie de la complexité

Commençons par une remarque à propos de l'hypothèse que nous avons faite sur le fait de restreindre notre discussion aux problèmes de décision.

11.4.1 Décision vs Construction

Nous avons parlé jusque-là uniquement de problèmes de *décision*, c'est-à-dire dont la réponse est soit vraie ou soit fausse (par exemple : "étant donnée une formule F, décider si la formule F est satisfiable") en opposition aux problèmes qui consisterait à *produire un objet avec une propriété* (par exemple : étant donné une formule F, produire une affectation des variables qui la satisfait s'il en existe une).

Clairement, produire une solution est plus difficile que de savoir s'il en existe une, et donc si $\mathrm{P} \neq \mathrm{NP}$, aucun de ces deux problèmes n'admet une solution, et ce pour tout problème NP-complet.

Cependant, si $\mathrm{P} = \mathrm{NP}$, il s'avère que alors on sait aussi produire une solution :

Théorème 11.6 *Supposons que* $\mathrm{P} = \mathrm{NP}$. *Soit* L *un problème de* NP *et* V *un vérificateur associé. On peut construire une machine de Turing qui sur toute entrée* $w \in L$ *produit en temps polynomial un certificat* u *pour* w *pour le vérificateur* V.

Démonstration: Commençons par le prouver pour L correspondant au problème de la satisfaction de formules (problème SAT). Supposons $\mathrm{P} = \mathrm{NP}$: on peut donc tester si une formule propositionnelle F à n variables est satisfiable ou non en temps polynomial. Si elle est satisfiable, alors on peut fixer sa première variable à 0, et tester si la formule obtenue F_0 est satisfiable. Si elle l'est, alors on écrit 0 et on recommence récursivement avec cette formule F_0 à $n-1$ variables. Sinon, nécessairement tout certificat doit avoir sa première variable à 1, on écrit 1, et on recommence récursivement avec la formule F_1 dont la première variable est fixée à 1, qui possède $n-1$ variables. Puisqu'il est facile de vérifier si une formule sans variable est satisfiable, par cette méthode, on aura écrit un certificat.

Maintenant si L est un langage quelconque de NP, on peut utiliser le fait que la réduction produite par la preuve du théorème de Cook-Levin est en fait une réduction de *Levin* : non seulement on a $w \in L$ si et seulement si $f(w)$ est une formule satisfiable, mais on peut aussi retrouver un certificat pour w à partir d'un certificat de la satisfiabilité de la formule $f(w)$. On peut donc utiliser l'algorithme précédent pour retrouver un certificat pour L. □

En fait, on vient d'utiliser le fait que le problème de la satisfiabilité d'une formule est *auto-réductible* à des instances de tailles inférieures.

11.4.2 Théorèmes de hiérarchie

On dit qu'une fonction $f(n) \geq n\log(n)$ est *constructible en temps*, si la fonction qui envoie 1^n sur la représentation binaire de $1^{f(n)}$ est calculable en temps $\mathcal{O}(f(n))$.

La plupart des fonctions usuelles sont constructibles en temps, et en pratique ce n'est pas vraiment une restriction.

Remarque 11.10 *Par exemple, $n\sqrt{n}$ est constructible en temps : sur l'entrée 1^n, on commence par compter le nombre de 1 en binaire : on peut utiliser pour cela un compteur, qui reste de taille $\log(n)$, que l'on incrémente : cela se fait donc en temps $\mathcal{O}(n\log(n))$ puisqu'on utilise au plus $\mathcal{O}(\log(n))$ étapes pour chaque lettre du mot en entrée. On peut alors calculer $\lfloor n\sqrt{n}\rfloor$ en binaire à partir de la représentation de n. N'importe quelle méthode pour faire cela fonctionne en temps $\mathcal{O}(n\log(n))$, puisque la taille des nombres impliqués est $\mathcal{O}(\log(n))$.*

Théorème 11.7 (Théorème de hiérarchie) *Pour toute fonction $f : \mathbb{N} \to \mathbb{N}$ constructible en temps, il existe un langage L qui est décidable en temps $\mathcal{O}(f(n))$ mais pas en temps $o(f(n))$.*

Démonstration: Il s'agit d'une généralisation de l'idée de la preuve du théorème 13.15 du chapitre suivant : nous invitons notre lecteur à commencer par cette dernière preuve.

Nous prouverons une version plus faible que l'énoncé plus haut. Soit $f : \mathbb{N} \to \mathbb{N}$ une fonction constructible en temps.

On considère le langage (très artificiel) L qui est décidé par la machine de Turing B suivante :

- sur une entrée w de taille n, B calcule $f(n)$ et mémorise $\langle f(n)\rangle$ le codage en binaire de $f(n)$ dans un compteur binaire c ;
- Si w n'est pas de la forme $\langle A\rangle 10^*$, pour un certaine machine de Turing A, alors la machine de Turing B refuse.
- Sinon, B simule A sur le mot w pendant $f(n)$ étapes pour déterminer si A accepte en un temps inférieur à $f(n)$:
  - si A accepte en ce temps, alors B refuse ;
  - sinon B accepte.

Autrement dit B simule A sur w, étape par étape, et décrémente le compteur c à chaque étape. Si ce compteur c atteint 0 ou si A refuse, alors B accepte. Sinon, B refuse.

Par l'existence d'une machine de Turing universelle, il existe des entiers k et d tels que L soit décidé en temps $d \times f(n)^k$.

Supposons que L soit décidé par une machine de Turing A en temps $g(n)$ avec $g(n)^k = o(f(n))$. Il doit exister un entier n_0 tel que pour $n \geq n_0$, on ait $d \times g(n)^k < f(n)$.

Par conséquent, la simulation par B de A sera bien complète sur une entrée de longueur n_0 ou plus.

Considérons ce qui se passe lorsque B est lancé sur l'entrée $\langle A\rangle 10^{n_0}$. Puisque cette entrée est de taille plus grande que n_0, B répond l'inverse de la machine de

Turing A sur la même entrée. Donc B et A ne décident pas le même langage, et donc la machine de Turing A ne décide pas L, ce qui mène à une contradiction.

Par conséquent L n'est pas décidable en temps $g(n)$ pour toute fonction $g(n)$ avec $g(n)^k = o(f(n))$.

Le théorème est une généralisation de cette idée. Le facteur log vient de la construction d'une machine de Turing universelle nettement plus efficace que ceux considérés dans ce document, introduisant seulement un ralentissement logarithmique en temps. □

Autrement dit :

Théorème 11.8 (Théorème de hiérarchie) *Soient $f, f' : \mathbb{N} \to \mathbb{N}$ des fonctions constructibles en temps telles que $f(n)\log(f(n)) = o(f'(n))$. Alors l'inclusion* $\mathrm{TIME(f)} \subset \mathrm{TIME(f')}$ *est stricte.*

On obtient par exemple :

Corollaire 11.4 $\mathrm{TIME(n^2)} \subsetneq \mathrm{TIME(n^{\log n})} \subsetneq \mathrm{TIME}(2^{\mathrm{n}})$.

On définit :

Définition 11.11 *Soit*

$$\mathrm{EXPTIME} = \bigcup_{c \in \mathbb{N}} \mathrm{TIME}(2^{\mathrm{n}^c}).$$

On obtient :

Corollaire 11.5 $\mathrm{P} \subsetneq \mathrm{EXPTIME}$.

Démonstration: Tout polynôme devient ultimement négligeable devant 2^n, et donc P est un sous-ensemble de $\mathrm{TIME}(2^{\mathrm{n}})$. Maintenant $\mathrm{TIME}(2^{\mathrm{n}})$, qui contient tout P est un sous-ensemble strict de, par exemple, $\mathrm{TIME}(2^{\mathrm{n}^3})$, qui est inclus dans EXPTIME. □

11.4.3 EXPTIME and NEXPTIME

Considérons

$$\mathrm{EXPTIME} = \bigcup_{c \geq 1} \mathrm{TIME}(2^{\mathrm{n}^c})$$

et

$$\mathrm{NEXPTIME} = \bigcup_{c \geq 1} \mathrm{NTIME}(2^{\mathrm{n}^c}).$$

On sait prouver le résultat suivant (et ce n'est pas vraiment difficile) :

Théorème 11.9 *Si* $\mathrm{EXPTIME} \neq \mathrm{NEXPTIME}$ *alors* $\mathrm{P} \neq \mathrm{NP}$.

11.5 Que signifie la question $P = NP$?

Nous faisons une digression autour de la signification de cette question en lien avec la théorie de la preuve, et les autres chapitres du document.

On peut voir NP comme classe des langages tel qu'en tester l'appartenance revient à déterminer s'il existe un certificat **court** (polynomial). On peut relier cela à l'existence d'une preuve en mathématiques. En effet, dans son principe même, la déduction mathématique consiste à produire des théorèmes à partir d'axiomes.

On s'attend à ce que la validité d'une preuve soit facile à vérifier : il suffit de vérifier que chaque ligne de la preuve soit bien la conséquence des lignes précédentes, dans le système de preuve. En fait, dans la plupart des systèmes de preuves axiomatiques (par exemple dans tous les systèmes de preuve que nous avons vu) cette vérification se fait en un temps qui est polynomial en la longueur de la preuve.

Autrement dit, le problème de décision suivant est NP pour chacun des systèmes de preuve axiomatiques usuels $\mathcal{A}$, et en particulier pour celui $\mathcal{A}$ que nous avons vu pour le calcul des prédicats.

$$\text{THEOREMS} = \{\langle \phi, \mathbf{1}^n \rangle | \phi \text{ possède une preuve de longueur } \leq n$$

$$\text{dans le système } \mathcal{A}\}.$$

Nous laisserons à notre lecteur l'exercice suivant :

Exercice 11.1 *La théorie des ensembles de* Zermelo-Fraenkel *est un des systèmes axiomatiques permettant d'axiomatiser les mathématiques avec une description finie. (Même sans connaître tous les détails de la théorie des ensembles de* Zermelo-Fraenkel*) argumenter à un haut niveau que le problème* THEOREMS *est* NP*-complet pour la théorie des ensembles de* Zermelo-Fraenkel.

Indice : la satisfiabilité d'un circuit booléen est un énoncé.

Autrement dit, en vertu du théorème 11.6, la question P = NP est celle (qui a été posée par Kurt Gödel) de savoir s'il existe une machine de Turing qui soit capable de produire la preuve mathématique de tout énoncé ϕ en un temps polynomial en la longueur de la preuve.

Cela semble t'il raisonnable ?

Que signifie la question NP = coNP **?** Rappelons que coNP est la classe des langages dont le complémentaire est dans NP. La question NP = coNP, est reliée à l'existence de preuve courte (de certificats) pour des énoncés qui ne semblent pas en avoir : par exemple, il est facile de prouver qu'une formule propositionnelle est satisfiable (on produit une valuation de ses entrées, que l'on peut coder dans une preuve qui dit qu'en propageant les entrées vers les sorties, le circuit répond 1). Par contre, dans le cas général, il n'est pas facile d'écrire une preuve courte qu'une formule propositionnelle donnée est non satisfiable.

Si NP = coNP, il doit toujours en exister une : la question est donc reliée à l'existence d'un autre moyen de prouver la non satisfiabilité d'une formule propositionnelle, que les méthodes usuelles.

On peut reformuler l'équivalent pour chacun des problèmes NP-complets évoqués.

11.6 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture de [Sipser, 1997], [Papadimitriou, 1994] et de [Lassaigne and de Rougemont, 2004].

Un ouvrage de référence et contenant les derniers résultats du domaine est [Arora and Barak, 2009].

Bibliographie Ce chapitre contient des résultats standards en complexité. Nous nous sommes essentiellement inspirés de leur présentation dans les ouvrages [Sipser, 1997], [Poizat, 1995], [Papadimitriou, 1994]. La dernière partie discussion est reprise ici de sa formulation dans [Arora and Barak, 2009].

Chapitre 12

Quelques problèmes NP-complets

Maintenant que nous connaissons la NP-complétude d'au moins un problème (SAT), nous allons montrer qu'un très grand nombre de problèmes sont NP-complets.

Le livre [Garey and Johnson, 1979] en recensait plus de 300 en 1979. Nous n'avons pas l'ambition d'en présenter autant, mais d'en présenter quelques un, de façon à décrire quelques problèmes NP-complets célèbres, et à montrer quelques preuves de NP-complétude.

12.1 Quelques problèmes NP-complets

12.1.1 Autour de SAT

Définition 12.1 (3-SAT)

Donnée: *Un ensemble de variables* $\{x_1, \cdots, x_n\}$ *et une formule* $F = C_1 \wedge C_2 \cdots \wedge C_\ell$ *avec* $C_i = y_{i,1} \vee y_{i,2} \vee y_{i,3}$*, où pour tout* i*,* j*,* $y_{i,j}$ *est soit* x_k*, soit* $\neg x_k$ *pour l'un des* x_k*.*

Réponse: *Décider si* F *est satisfiable : c'est-à-dire décider s'il existe* $x_1, \cdots, x_n \in \{0,1\}$ *tel que* F *s'évalue en vraie pour cette valeur de ses variables* $x_1, \cdots, x_n$*.*

Théorème 12.1 *Le problème* 3-SAT *est* NP*-complet.*

Démonstration: Notons tout d'abord que $3-$SAT est bien dans NP. En effet, la donnée d'une affectation de valeurs de vérité aux variables constitue un certificat vérifiable en temps polynomial.

On va réduire SAT à $3-$SAT. Soit F une formule SAT. Soit C une clause de F, par exemple $C = x \vee y \vee z \vee u \vee v \vee w \vee t$. On introduit de nouvelles variables a, b, c, d associées à cette clause, et on remplace C par la formule

$$(x \vee y \vee a) \wedge (\neg a \vee z \vee b) \wedge (\neg b \vee u \vee c) \wedge (\neg c \vee v \vee d) \wedge (\neg d \vee w \vee t).$$

Il est facile de vérifier qu'une assignation de x, y, z peut être complétée par une assignation de a, b, c, d de façon à rendre cette formule vraie si et seulement si C est vraie. En appliquant cette construction à toutes les clauses de F et en prenant la conjonction des formules ainsi produites, on obtient ainsi une formule $3-\text{SAT}$ dont la satisfaction de F' équivalente à celle de F.

Le temps de calcul se réduit à écrire les clauses, dont la longueur est polynomiale. Par conséquent, l'ensemble du processus de réduction se réalise donc bien en temps polynomial, et on a prouvé, à partir de SAT que $3-\text{SAT}$ est NP-complet. □

Définition 12.2 (NAESAT)

Donnée*: Un ensemble de variables $\{x_1, \cdots, x_n\}$ et un ensemble de clauses $y_{i,1} \vee \cdots \vee y_{i,k_i}$, où pour tout i, j, $y_{i,j}$ est soit x_k, soit $\neg x_k$ pour l'un des x_k.*

Réponse*: Décider s'il existe une affectation des variables $x_i \in \{0, 1\}$ de telle sorte que chaque clause contienne au moins un littéral vrai et au moins un littéral faux (c'est-à-dire, pour tout i, il y a un j et un k avec $y_{i,j} = 1$ et $y_{i,k} = 0$).*

Théorème 12.2 *Le problème* NAESAT *est* NP*-complet.*

Démonstration: Le problème est dans NP car la donnée d'une affectation des variables est un certificat valide vérifiable aisément en temps polynomial.

On va réduire SAT à NAESAT. Soit F une formule de SAT sur les variables $\{x_1, \cdots, x_n\}$. On ajoute une unique variable distincte z et on forme les clauses pour NAESAT en remplaçant chaque clause $C_i = y_{i,1} \vee \cdots \vee y_{i,k}$ de F en $C'_i = y_{i,1} \vee \cdots \vee y_{i,k} \vee z$.

Cette transformation se réalise bien en temps polynomial.

Si l'instance donnée de SAT est satisfiable, la même affectation des variables tout en fixant pour z la valeur 0 fournit une affectation valide pour NAESAT.

Réciproquement, supposons que l'instance construite de NAESAT soit satisfaisable. Si la valeur de vérité de z dans l'affectation correspondante est 0, alors les valeurs des variables x_i dans l'affectation donnent une affectation valide pour la formule F d'origine (pour l'instance de SAT). Si au contraire z vaut 1, on change toutes les valeurs de toutes les variables dans l'affectation. L'affectation reste valide pour NAESAT car au moins un littéral par clause dans l'affectation initial valait 0 dans l'affectation initiale, et vaut donc maintenant 1, tandis que z vaut 0. On a donc construit une affectation dans laquelle z vaut 0, et en vertu du cas précédent l'instance de SAT initiale est satisfaisable.

On a donc bien prouvé l'équivalence entre satisfaisabilité de F et l'instance correspondante pour NAESAT. Donc NAESAT $<$ est NP-complet. □

En utilisant la même réduction sur 3SAT, on prouve que NAE4SAT est NP-complet. On va utiliser cette remarque pour prouver :

Corollaire 12.1 NAE3SAT*est* NP*-complet.*

Démonstration: On va réduire NAE4SAT à NAE3SAT. Soit $C = x \vee y \vee z \vee t$ une clause à 4 littéraux. On introduit une nouvelle variable u_C, et on forme les deux clauses $C_1 = x \vee y \vee \neg u_C$ et $C_2 = z \vee t \vee u_C$. En faisant ainsi pour toutes les clauses, on construit une instance F' de NAE3SAT en temps polynomial.

Supposons que F' soit une instance positive de NAE3SAT, et considérons l'affectation des valeurs de vérité correspondante. Si $u_C = 0$, alors x ou y est 0, et z ou t est 1, donc $x \vee y \vee z \vee t$ a au moins un littéral 1 et au moins un littéral 0 ; de même, si $u_C = 1$; donc F est une instance positive de de NAE4SAT.

Inversement, si F est une instance positive de NAE4SAT, considérons l'affectation de valeurs de vérité correspondante. Dans $x \vee y \vee z \vee t$, si x et y sont tous deux à 1, on affecte u_C à 1 ; sinon si x et y sont tous les deux à 0, on affecte u_C à 0 ; sinon, on affecte à u_C la valeur de vérité adéquate pour la clause $u_C \vee z \vee t$. Cela produit une assignation démontrant que F' est une instance positive de NAE3SAT.

Là encore la réduction est polynomiale, et NAE3SAT est dans NP de façon triviale. □

12.1.2 Autour de STABLE

Définition 12.3 (STABLE)
Donnée: *Un graphe $G = (V, E)$ non-orienté et un entier k.*
Réponse: *Décider s'il existe $V' \subset V$, avec $|V'| = k$, tel que $u, v \in V' \Rightarrow (u, v) \notin E$.*

Théorème 12.3 *Le problème* STABLE *est* NP*-complet.*

Démonstration: STABLE est bien dans NP, car la donnée de V' est un certificat facilement vérifiable en temps polynomial.

On va réduire le problème $3-$SAT à STABLE, c'est-à-dire, étant donné une formule F du type $3-$SAT, construire en temps polynomial un graphe G, de sorte que l'existence d'un stable dans G soit équivalente à l'existence d'une affectation de valeurs de vérité qui satisfait F.

Soit $F = \bigwedge_{1 \leq j \leq k} (x_{1j} \vee x_{2j} \vee x_{3j})$. On construit un graphe G avec $3k$ sommets, un pour chaque occurrence d'un littéral dans une clause.

- Pour chaque variable x_i de $3-$SAT, G possède une arête entre chaque sommet associé à un littéral x_i et chaque sommet associé à un littéral $\neg x_i$ (ainsi un stable de G correspond à une affectation de valeurs de vérité à une partie des variables) ;
- Pour chaque clause C, on associe un triangle : par exemple pour une clause de F de la forme $C = (x_1 \vee \neg x_2 \vee x_3)$, alors G possède les arêtes $(x_1, \neg x_2)$, $(\neg x_2, x_3)$, (x_3, x_1) (ainsi un stable de G contient au plus un des trois sommets associés à la clause C).

Soit k le nombre de clauses dans F. On démontre que F est satisfiable si et seulement si G possède un stable de taille k.

En effet, si F est satisfiable, on considère une assignation des variables satisfaisant F. Pour chaque clause C de F, on choisit y_C un littéral de C rendu vrai par l'assignation : cela définit k sommets formant un stable de G.

Réciproquement, si G a un stable de taille k, alors il a nécessairement un sommet dans chaque triangle. Ce sommet correspond à un littéral rendant la clause associée vraie, et forme une assignation des variables cohérente par construction des arêtes.

La réduction est clairement polynomiale. □

Deux problèmes classiques sont reliés à STABLE.

Définition 12.4 (CLIQUE)

Donnée: *Un graphe $G = (V, E)$ non-orienté et un entier k.*

Réponse: *Décider s'il existe $V' \subset V$, avec $|V'| = k$, tel que $u, v \in V' \Rightarrow (u, v) \in E$.*

Théorème 12.4 *Le problème CLIQUE est* NP*-complet.*

Démonstration: La réduction à partir de STABLE consiste à passer au complémentaire sur les arêtes. En effet, il suffit de prouver qu'un graphe $G = (V, E)$ a un stable de taille k si et seulement si son graphe complémentaire $\overline{G} = (V, \overline{E})$ (où $\overline{E} = \{(u, v) | (u, v) \notin E\}$) a une clique de taille k. □

Définition 12.5 (RECOUVREMENT DE SOMMETS)

Donnée: *Un graphe $G = (V, E)$ non-orienté et un entier k.*

Réponse: *Décider s'il existe $V' \subset V$, avec $|V'| \leq k$, tel que toute arête de G ait au moins une extrémité dans V'.*

Théorème 12.5 *Le problème RECOUVREMENT DE SOMMETS est* NP*-complet.*

Démonstration: La réduction à partir de STABLE consiste à passer au complémentaire sur les sommets. □

Définition 12.6 (COUPURE MAXIMALE)

Donnée: *Un graphe $G = (V, E)$ non-orienté et un entier k.*

Réponse: *Décider s'il existe une partition $V = V_1 \cup V_2$ telle que le nombre d'arêtes entre V_1 et V_2 soit au moins k.*

Théorème 12.6 *Le problème COUPURE MAXIMALE est* NP*-complet.*

Démonstration: On réduit NAE3SAT à COUPURE MAXIMALE. Supposons donc donnée une instance de NAE3SAT, dans laquelle on peut supposer sans perte de généralité qu'une clause ne contient pas simultanément une variable et son complémentaire. Quitte à remplacer $u \vee v$ par $((u \vee v \vee w) \wedge (u \vee v \vee \neg w))$, on peut aussi supposer que chaque clause contient exactement 3 littéraux (si une clause contient un seul littéral, la formule n'est pas satisfiable). Enfin, si l'on a deux clauses $(u \vee v \vee w)$ et $(u \vee v \vee z)$, on peut, en introduisant deux variables t_1 et t_2 et en procédant comme pour réduire NAE4SATà NAE3SAT, réécrire ces deux clauses comme $(u \vee t_1 \vee t_2) \wedge (v \vee w \vee \neg t_1) \wedge (v \vee z \vee \neg t_2)$. Bref, on peut donc supposer que deux clauses données ont au plus une variable en commun.

On note $x_1, \cdots, x_n$ les variables de la formule F.

On va construire un graphe $G = (V, E)$ de la façon suivante : G possède $2n$ sommets où pour chaque variable u de F, correspondent deux sommets u et $\neg u$. G possède une arête entre chaque couple de sommets (u, v) tels que u et v apparaissent dans la même clause, et une arête entre les sommets u et $\neg u$ pour toute variable u.

Les réductions dans le premier paragraphe de la preuve permettent de voir qu'à chaque clause correspond un triangle et que deux de ces triangles ont des arêtes distinctes.

Si on note n le nombre de variables, et m le nombre de clauses, le graphe G a donc $2n$ sommets et $3m + n$ arêtes. Il est alors facile de voir que le nombre d'arêtes dans une coupure correspondant à une affectation NAE3SAT valide est $2m + n$: l'arête entre u et $\neg u$ pour chaque variable u, et deux des arêtes du triangle uvw pour chaque clause $u \vee v \vee w$.

Inversement, toute coupure de G a au plus $(2m + n)$ arêtes, car une coupure ne peut inclure que deux arêtes par triangle associé à une clause. Par conséquent, une coupure de valeur $2m+n$ fournit immédiatement une affectation NAE3SAT valide.

En d'autres termes, résoudre l'instance de NAE3SAT revient à résoudre COUPURE MAXIMALE sur $(G, 2m + n)$.

La réduction étant polynomiale, et puisque COUPUREMAXIMALE est dans NP car la donnée de V_1 est un certificat valide vérifiable en temps polynomial. □

12.1.3 Autour de CIRCUIT HAMILTONIEN

Le problème CIRCUIT HAMILTONIEN est souvent à la base de propriétés liées aux chemins dans les graphes (nous reprenons la définition de ce problème, même s'il a déjà été défini dans le chapitre précédent).

Définition 12.7 (CIRCUIT HAMILTONIEN)
***Donnée**: Un graphe $G = (V, E)$ (non-orienté).*
***Réponse**: Décider s'il existe un* circuit hamiltonien*, c'est-à-dire un chemin de G passant une fois et une seule par chacun des sommets et revenant à son point de départ.*

Théorème 12.7 *Le problème CIRCUIT HAMILTONIEN est* NP*-complet.*

Démonstration: On va démontrer ce fait en réduisant RECOUVREMENT DE SOMMETS à ce problème.

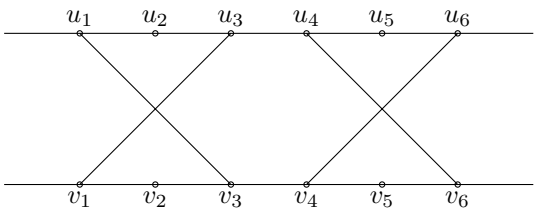
La technique consiste à construire, partant d'une instance de RECOUVREMENT DE SOMMETS, un graphe dans lequel chaque arête initiale sera remplacée par un "motif" admettant exactement deux chemins hamiltoniens, c'est-à-dire des chemins visitant une fois et une seule chaque sommet. L'un de ces deux chemins correspondra au cas où le sommet correspondant appartient à la couverture, l'autre au cas où il n'y appartient pas.

Il reste ensuite à préciser comment recoller ces différents motifs pour qu'un circuit hamiltonien global corresponde exactement à une réunion de chemins

hamiltoniens à travers chaque motif, et pour assurer qu'on obtient bien le bon résultat au problème RECOUVREMENT DE SOMMETS initial en résolvant CIRCUIT HAMILTONIEN dans ce graphe.

Notons $(G = (V, E), k)$ l'instance de RECOUVREMENT DE SOMMETS étudiée.

Le motif que nous allons utiliser est le suivant :



Pour obtenir un parcours de ce motif traversant une fois et une seule chaque sommet, seules deux solutions sont possibles : soit un passage en deux fois, une fois $u_1u_2u_3u_4u_5u_6$ puis ultérieurement $v_1v_2v_3v_4v_5v_6$ (dans un sens ou dans l'autre) ; ou alors un passage en une seule fois $u_1u_2u_3v_1v_2v_3v_4v_5v_6u_4u_5u_6$ (ou le même en inversant les u et les v).

À chaque arête (u, v) du graphe de départ, on associe un motif de ce type, en faisant correspondre le sommet u au côté u et le sommet v au côté v. On raccorde ensuite entre eux bout à bout tous les côtés de tous les motifs correspondant à un même sommet ; on forme donc une chaîne associée à un sommet donné, avec encore deux "sorties" libres. On raccorde alors chacune de ces deux sorties à k nouveaux sommets $s_1, \cdots, s_k$. On note le graphe ainsi construit H dans la suite.

Supposons maintenant donné un recouvrement du graphe initial de taille k, dont les sommets sont $\{g_1, \cdots, g_k\}$. On peut alors construire un circuit hamiltonien de H de la façon suivante :

- partir de s_1 ;
- parcourir la chaîne g_1 de la façon suivante. Quand on traverse une arête (g_1, h), si h est aussi dans la couverture, on traverse simplement le côté g_1, sinon, on parcourt les deux côtés simultanément ;
- une fois la chaîne g_1 finie, on revient en s_2 et on repart par g_2 et ainsi de suite.

Il est clair que tous les sommets s_k sont atteints une fois et une seule.

Considérons un sommet h du motif correspondant à une arête (u, v). On peut toujours supposer que u est dans la couverture, disons $u = g_1$. Il s'ensuit que si h est du côté de u, h sera atteint une fois au moins. On voit en vertu du second item qu'il ne sera plus atteint dans la suite. Si h est du côté de v, et que v n'est pas dans la couverture, h est parcouru lors du parcours de u. Si $v = g_i$, h est parcouru lors du parcours de la chaîne correspondant à g_i et à ce moment-là seulement. On a donc bien un circuit hamiltonien.

Réciproquement, supposons que l'on dispose d'un circuit hamiltonien de H. La construction de notre motif impose que venant d'un sommet s_i, on traverse

entièrement une chaîne u puis l'on passe à un autre des sommets s_j. On traverse ainsi k chaînes ; les k sommets correspondants forment alors une couverture. En effet, si (u, v) est une arête, le motif correspondant est parcouru par le chemin hamiltonien ; or il ne peut l'être que lors du parcours d'une boucle correspondant à une des deux extrémités de l'arête.

Enfin, la réduction est trivialement polynomiale. CIRCUIT HAMILTONIEN est donc bien NP-complet. □

Comme dans la section précédente, on peut alors en déduire la NP-complétude de nombreuses variantes.

Définition 12.8 (VOYAGEUR DE COMMERCE)

***Donnée**: Un couple (n, M), où M est une matrice $n \times n$ d'entiers et un entier k.*

***Réponse**: Décider s'il existe une permutation π de $[1, 2, \cdots, n]$ telle que*

$$\sum_{1 \leq i \leq n} M_{\pi(i)\pi(i+1)} \leq k.$$

Corollaire 12.2 *Le problème VOYAGEUR DE COMMERCE est* NP*-complet.*

Ce problème porte ce nom, car on peut voir cela comme l'établissement de la tournée d'un voyageur de commerce devant visiter n villes, dont les distances sont données par la matrice M de façon à faire moins de k kilomètres.

Démonstration:

On réduit CIRCUIT HAMILTONIEN à VOYAGEUR DE COMMERCE. Pour ce faire, étant donné un graphe $G = (V, E)$, on pose $V = \{x_1, \cdots, x_n\}$. On considère alors la matrice M $n \times n$ d'entiers telle que

$$M_{i,j} = \begin{cases} 1 & \text{si } (i, j) \in E; \\ 2 & \text{sinon.} \end{cases}$$

Montrons alors que CIRCUIT HAMILTONIEN(V, E) ssi VOYAGEUR DE COMMERCE(n, M, n).

S'il existe un circuit hamiltonien dans G on peut en effet construire la permutation π comme décrivant l'ordre de parcours des sommets du graphe G : par construction, la somme des distances des arêtes sur ce circuit vaudra n.

Inversement, étant donnée une permutation avec cette propriété, le fait que les $n-1$ termes de la somme soient au moins égaux à 1 implique qu'ils sont tous égaux à 1, et donc que les arêtes $(\pi(i), \pi(i+1))$ existent dans le graphe G : on a donc bien un circuit hamiltonien. □

Définition 12.9 (CIRCUIT LE PLUS LONG)

***Donnée**: Un graphe $G = (V, E)$ non-orienté, avec des distances sur chaque arête, un entier r.*

***Réponse**: Décider s'il existe un circuit de G ne passant pas deux fois par le même sommet dont la longueur est $\geq r$.*

Corollaire 12.3 *Le problème CIRCUIT LE PLUS LONG est* NP*-complet.*

Démonstration: On construit une réduction à partir de CIRCUIT HAMILTONIEN : pour cela, à un graphe G pour CIRCUIT HAMILTONIEN, on associe à chaque arête le poids 1. La recherche d'un cycle hamiltonien est alors trivialement identique à la recherche d'un circuit de longueur $\geq n$ dans le graphe. □

12.1.4 Autour de 3-COLORABILITE

Nous avons déjà définit ce que nous appelions un *bon coloriage* de graphes dans le chapitre précédent. Le plus petit entier k tel qu'un graphe soit (bien) coloriable est appelé le *nombre chromatique* du graphe.

Il est connu que les graphes planaires sont toujours coloriables avec 4 couleurs.

Définition 12.10 (3-COLORABILITE)
***Donnée**: Un graphe $G = (V, E)$ non-orienté.*
***Réponse**: Décider s'il existe un coloriage du graphe utilisant au plus 3 couleurs.*

Théorème 12.8 *Le problème* 3-COLORABILITE *est* NP*-complet.*

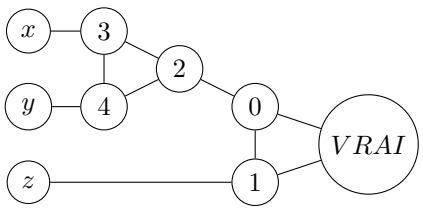
Démonstration: 3-COLORABILITE est dans NP, car la donnée des sommets colorés par chacune des 3 couleurs constitue un certificat vérifiable en temps polynomial.

On va réduire 3-SAT à 3-COLORABILITE. On se donne donc une conjonction de clauses à 3 littéraux, et il nous faut à partir de là construire un graphe. Comme dans les autres réductions de 3-SAT, il faut parvenir à traduire deux contraintes : une variable peut prendre la valeur 0 ou 1 d'une part, et les règles d'évaluation d'une clause d'autre part.

On construit un graphe ayant $3 + 2n + 5m$ sommets, les trois permiers sont notés $VRAI$, $FAUX$, NSP. Ces trois sommets sont reliés deux à deux en triangle, de sorte qu'ils doivent être tous trois de couleurs différentes. On appellera les couleurs correspondantes $VRAI$, $FAUX$, NSP.

On associe un sommet à chaque variable et au complémentaire de chaque variable. Pour assurer qu'une variable prenne la valeur $VRAI$ ou $FAUX$, pour chaque variable x_i on construit un triangle dont les sommets sont x_i, $\neg x_i$, et NSP. Cela impose que soit $couleur(x_i) = VRAI$ et $couleur(\neg x_i) = FAUX$, ou $couleur(x_i) = FAUX$ et $couleur(\neg x_i) = VRAI$.

Il nous reste donc à encoder les règles d'évaluation d'une clause. Pour ce faire, on introduit le sous-graphe suivant, qui correspond à une clause $x \vee y \vee z$:



Il est facile de voir que si ce motif (où les trois sommets distingués et les triangles construits sur les variables sont implicites) est 3-coloriable, alors les sommets 0 et 1 sont $FAUX$ et NSP. Si 1 est $FAUX$, comme un sommet correspondant à une variable doit être $VRAI$ ou $FAUX$, on a $couleur(z) = VRAI$. Si 0 est $FAUX$, alors 2 ne peut pas être $FAUX$, donc 3 ou 4 l'est, et la variable correspondante est coloriée $VRAI$.

Réciproquement, si l'une des variables est vraie, on peut facilement construire une 3-coloration du motif.

Considérons alors le graphe formé des trois sommets distingués, des triangles formés sur les variables, et des motifs donnés. Si ce graphe est 3-coloriable, alors en particulier tout sous-graphe est coloriable. Les triangles de variables sont en particulier coloriables. À partir d'une 3-coloration du graphe, on construit une affectation de valeurs de vérité en mettant à 1 toutes les variables coloriées par $VRAI$. Cette affectation est cohérente (une variable et son complémentaire ont bien une valeur opposée) et au moins une variable par clause est à 1, en vertu des propriétés du motif ci-dessus. Inversement, étant donné une affectation de valeurs de vérité, il est aisé de déduire une 3-coloration du graphe.

L'existence d'une 3-coloration du graphe est donc équivalente à la satisfaisabilité de la formule initiale.

La réduction est manifestement polynomiale ; on a donc bien prouvé que 3-SAT se réduisait à 3-COLORABILITE ; ce dernier est donc bien NP-complet. □

12.1.5 Autour de SOMME DE SOUS-ENSEMBLE

Définition 12.11 (SOMME DE SOUS-ENSEMBLE)
Donnée: *Un ensemble fini d'entiers E et un entier t.*
Réponse: *Décider s'il existe $E' \subset E$ tel que $\sum_{x \in E'} x = t$.*

Théorème 12.9 *Le problème SOMME DE SOUS-ENSEMBLE est* NP*-complet.*

Démonstration: Le fait que SOMME DE SOUS-ENSEMBLE est dans NP vient du fait que la donnée de E' constitue un certificat vérifiable en temps polynomial.

On va maintenant réduire une version de RECOUVREMENT DE SOMMETS dans laquelle on cherche à construire une couverture de taille exactement k à SOMME DE SOUS-ENSEMBLE. Il est facile de voir que cette version de RECOUVREMENT DE SOMMETS est essentiellement équivalente à la version classique, car il suffit de décider pour chaque $l \leq l_0$ s'il existe une couverture de taille l pour savoir s'il existe une couverture de taille au plus l_0.

On se donne un graphe $G = (V, E)$ dans lequel on souhaite construire un recouvrement de sommets de taille k. On numérote les sommets et arêtes ; soit $B = (b_{ij})$ la matrice d'incidence sommets-arêtes, c'est-à-dire que $b_{ij} = 1$ si l'arête i est incidente au sommet j, $b_{ij} = 0$ sinon.

On va construire un ensemble F d'entiers, tel que chaque sommet corresponde à un entier de F. On veut pouvoir lire sur la somme des éléments d'un

sous-ensemble les arêtes dont un des sommets au moins appartient au sous-ensemble, de façon à ce qu'on ait bien une couverture de sommets si et seulement si c'est le cas pour toutes les arêtes.

Cela peut se faire en exprimant les entiers dans une base b à fixer ultérieurement, et en faisant correspondre au sommet s_j l'entier $\sum b_{ij} b^i$. On veut aussi pouvoir compter le nombre de sommets présents dans la couverture, qui doit être égal à k ; pour ce faire, il suffit d'ajouter un terme b^n, si les arêtes sont numérotées de 0 à $n-1$.

Il subsiste une difficulté : pour qu'un sous-ensemble de somme $t = \sum_{i=0}^{n} a_i b^i$ corresponde à une couverture de sommets, le choix que l'on vient juste de faire impose que (noter que tous les a_i sauf a_n valent au plus 2, car une arête est incidente à deux sommets exactement) :

- $a_n = k$;
- $a_i \geq 1$.

Les spécifications du problème ne permettent pas de prendre en compte le second item. On peut contourner le problème en ajoutant à F les entiers b^i correspondant à l'arête i.

On peut alors choisir comme but

$$t = kb^n + \sum_{i=0}^{n-1} 2b^i. \tag{12.1}$$

À ce stade, on va imposer que la base b soit au moins égale à 4 ; comme dans toute somme d'éléments de F il y a au plus trois termes b^i pour $i < n$, cela entraîne qu'aucune retenue ne peut se produire dans l'addition (on permet toujours au coefficient de b^n d'excéder $b-1$), et donc que l'on peut lire sur l'équation (12.1) le nombre d'occurrences du terme b^i.

Pour tout sous-ensemble F' de somme t, on a donc :

- il y a k termes correspondant à des sommets dans F' (cf. le terme en b^n pour chaque arête i) ;
- F' contient au moins un terme correspondant à une des deux extrémités de i ; en effet, t contenant 2 termes b^i, il peut contenir soit les deux extrémités de l'arête, soit l'entier correspondant à l'arête et celui correspondant à l'une des deux extrémités.

Inversement, étant donné une couverture de sommets, on construit l'ensemble F' en prenant les sommets de la couverture, et les arêtes dont un seul des deux sommets est dans la couverture.

On a donc bien réduit RECOUVREMENT DE SOMMETS à SOMME DE SOUS-ENSEMBLE.

Il est facile de voir que la réduction s'effectue en temps polynomial, et donc que l'on a bien prouvé le théorème. □

On peut en déduire :

Définition 12.12 (SAC A DOS)

Donnée: *Un ensemble de poids $a_1, \cdots, a_n$, un ensemble de valeurs $v_1, \cdots, v_n$, un poids limite A, et un entier V.*

***Réponse**: Décider s'il existe $E' \subset \{1, 2, \cdots, n\}$ tel que $\sum_{i \in E'} a_i \leq A$ et $\sum_{i \in E'} v_i \geq V$.*

Corollaire 12.4 *Le problème SAC A DOS est* NP*-complet.*

Démonstration: A partir de SOMME DE SOUS-ENSEMBLE : étant donnée $E = \{e_1, \cdots, e_n\}$ et t une instance de SOMME DE SOUS-ENSEMBLE, on considère $v_i = a_i = e_i$, et $V = A = t$. □

Définition 12.13 (PARTITION)
***Donnée**: Un ensemble fini d'entiers E.*
***Réponse**: Décider s'il existe $E' \subset E$ tel que $\sum_{x \in E'} x = \sum_{x \notin E'} x$.*

Théorème 12.10 *Le problème PARTITION est* NP*-complet.*

Démonstration:
On va réduire SOMME DE SOUS-ENSEMBLEà PARTITION. Soit (E, t) une instance de SOMME DE SOUS-ENSEMBLE. On pose $S = \sum_{x \in E} x$. Quitte à changer t en $S - t$ (ce qui revient à changer l'ensemble obtenu en son complémentaire), on peut supposer que $2t \leq S$.

L'idée naturelle consisterait à ajouter l'élément $u = S - 2t$ à E ; le résultat de partition serait alors deux sous-ensembles (A' et son complémentaire) de somme $S - t$; l'un des deux contient l'élément $S - 2t$, donc en enlevant ce dernier, on trouve un ensemble de termes de E de somme t. Malheureusement, cette technique échoue si $S - 2t$ est déjà dans E.

Au lieu de cela, on prend le nombre $X = 2S$ et $X' = S + 2t$, et on applique PARTITION à $E' = E \cup \{X, X'\}$. Il existe une partition de E' si et seulement s'il existe un sous-ensemble de E de somme t. En effet, s'il existe une partition de E', il existe deux sous-ensembles complémentaires de somme $2S + t$. Chacun des deux sous-ensembles doit contenir soit X, soit X', car sinon sa somme ne peut excéder $2S + t$; donc un des deux ensembles contient X et non X', et on obtient en enlevant X un sous-ensemble F de E de taille t. Réciproquement, étant donné un tel F, $(F \cup \{X\}, E - F \cup \{X'\})$ constitue une partition de E'. On a donc bien réduit SOMME DE SOUS-ENSEMBLE à PARTITION.

Reste à justifier que la réduction est bien polynomiale. L'essentiel de la réduction est le calcul de A et A', qui est bien polynomial en la taille des entrées (l'addition de k nombres de n bits se fait en temps $\mathcal{O}(k \log n)$). □

12.2 Notes bibliographiques

Lectures conseillées Le livre [Garey and Johnson, 1979] recense plus de 300 problèmes NP-complets et est d'une lecture assez agréable. On peut trouver plusieurs mises à jour de cette liste sur le web.

Bibliographie Ce chapitre est reprise du polycopié [Cori et al., 2010] du cours INF550 de l'école polytechnique (avec quelques corrections mineures).

Chapitre 13

Complexité en espace mémoire

Dans ce chapitre, on s'intéresse à une autre ressource critique dans les algorithmes : la mémoire.

On va commencer par voir comment on mesure la mémoire utilisée par un algorithme. On introduira alors les principales classes considérées en théorie de la complexité.

En fait, en théorie de la complexité, on parle plutôt *d'espace*, ou *d'espace mémoire*, pour désigner la mémoire.

13.1 Espace polynomial

Dans toute cette section, nous énonçons un ensemble de définitions et de théorèmes sans preuve. Les preuves sont données dans la section suivante.

Introduisons l'analogue de TIME(t(n)) pour la mémoire :

Définition 13.1 (SPACE(t(n))**)** *Soit $t : \mathbb{N} \to \mathbb{N}$ une fonction. On définit la classe* SPACE(t(n)) *comme la classe des problèmes (langages) qui sont décidés par une machine de Turing en utilisant $\mathcal{O}(t(n))$ cases du ruban, où n est la taille de l'entrée.*

13.1.1 Classe PSPACE

On considère alors la classe des problèmes décidés en utilisant un espace mémoire polynomial.

Définition 13.2 PSPACE *est la classe des problèmes (langages) décidés en espace polynomial. En d'autres termes,*

$$\text{PSPACE} = \bigcup_{k \in \mathbb{N}} \text{SPACE}(\text{n}^{\text{k}}).$$

Remarque 13.1 *Comme dans le chapitre 11, on pourrait observer que cette notion ne dépend pas réellement du modèle de calcul utilisé, et que l'utilisation des machines de Turing comme modèle de base est relativement arbitraire.*

On peut aussi introduire l'analogue non déterministe :

Définition 13.3 (NSPACE(t(n))**)** *Soit $t : \mathbb{N} \to \mathbb{N}$ une fonction. On définit la classe* NSPACE(t(n)) *comme la classe des problèmes (langages) qui sont acceptés par une machine de Turing en utilisant $\mathcal{O}(t(n))$ cases du ruban sur chacune des branches du calcul, où n est la taille de l'entrée.*

Il serait alors naturel de définir

$$\text{NPSPACE} = \bigcup_{k \in \mathbb{N}} \text{SPACE}(\text{n}^{\text{k}}),$$

mais il s'avère que la classe NPSPACE n'est autre que PSPACE.

Théorème 13.1 (Théorème de Savitch) NPSPACE = PSPACE.

13.1.2 Problèmes $PSPACE$-complets

La classe PSPACE possède des problèmes complets : le problème QSAT (aussi appelé QBF) consiste étant donnée une formule du calcul propositionnel en forme normale conjonctive ϕ avec les variables $x_1, x_2, \cdots, x_n$ (c'est-à-dire une instance de SAT) à déterminer si $\exists x_1 \forall x_2 \exists x_3 \cdots\ \phi(x_1, \cdots, x_n)$?

Théorème 13.2 *Le problème* QSAT *est* PSPACE*-complet.*

Les jeux stratégiques sur les graphes donnent naturellement naissance à des problèmes PSPACE-complet.

Par exemple, le jeu GEOGRAPHY consiste à se donner un graphe orienté $G = (V, E)$. Le joueur 1 choisit un sommet u_1 du graphe. Le joueur 2 doit alors choisir un sommet v_1 tel qu'il y ait un arc de u_1 vers v_1. C'est alors au joueur 1 de choisir un autre sommet u_2 tel qu'il y ait un arc de v_1 vers u_2, et ainsi de suite. On n'a pas le droit de repasser deux fois par le même sommet. Le premier joueur qui ne peut pas continuer le chemin $u_1 v_1 u_2 v_2 \cdots$ perd. Le problème GEOGRAPHY consiste à déterminer étant donné un graphe G et un sommet de départ pour le joueur 1, s'il existe une stratégie gagnante pour le joueur 1.

Théorème 13.3 *Le problème* GEOGRAPHY *est* PSPACE*-complet.*

13.2 Espace logarithmique

Il s'avère que la classe PSPACE est énorme et contient tout P et aussi tout NP : imposer un espace mémoire polynomial est donc souvent peu restrictif.

C'est pourquoi on cherche souvent plutôt à parler d'un espace logarithmique : mais cela introduit une difficulté et un problème dans les définitions : en effet, une machine de Turing utilise au moins les cases qui contiennent son entrée, et donc la définition 13.1 ne permet pas de parler de fonctions $t(n) < n$.

C'est pour cela que l'on modifie cette définition avec la convention suivante : lorsque l'on mesure l'espace mémoire utilisé, par convention, on ne compte pas les cases de l'entrée.

Pour le faire, proprement, il faut donc remplacer la définition 13.1 par la suivante :

Définition 13.4 (SPACE(t(n))**)** *Soit $t : \mathbb{N} \to \mathbb{N}$ une fonction. On définit la classe* SPACE(t(n)) *comme la classe des problèmes (langages) qui sont décidés par une machine de Turing qui utilise deux rubans :*

- *le premier ruban contient l'entrée est accessible en lecture seulement : il peut être lu, mais il ne peut pas être écrit ;*
- *le second est lui initialement vide et est accessible en lecture et écriture ;*

en utilisant $\mathcal{O}(t(n))$ cases du second ruban, où n est la taille de l'entrée.

On définit NSPACE(t(n)) *avec la même convention.*

Remarque 13.2 *Cette nouvelle définition ne change rien aux classes précédentes. Elle permet simplement de donner un sens aux suivantes.*

Définition 13.5 (LOGSPACE**)** *La classe* LOGSPACE *est la classe des langages et des problèmes décidés par une machine de Turing en espace logarithmique. En d'autres termes,*

$$\text{LOGSPACE} = \text{SPACE}(\log(n)).$$

Définition 13.6 (NLOGSPACE**)** *La classe* NLOGSPACE *est la classe des langages et des problèmes décidés par une machine de Turing en espace non déterministe logarithmique. En d'autres termes,*

$$\text{NLOGSPACE} = \text{NSPACE}(\log(n)).$$

Il s'avère que l'on a :

Théorème 13.4 $\text{LOGSPACE} \subset \text{NLOGSPACE} \subset \text{P} \subset \text{NP} \subset \text{PSPACE}$.

On sait par ailleurs que $\text{NLOGSPACE} \subsetneq \text{PSPACE}$ mais on ne sait pas lesquelles des inclusions intermédiaires sont strictes.

13.3 Quelques résultats et démonstrations

Cette section est consacrée à prouver un certain nombre des principaux résultats de la théorie de la complexité, en particulier sur les liens entre temps et mémoire. Observons que le théorème 13.4 découle de chacun des résultats qui suivent.

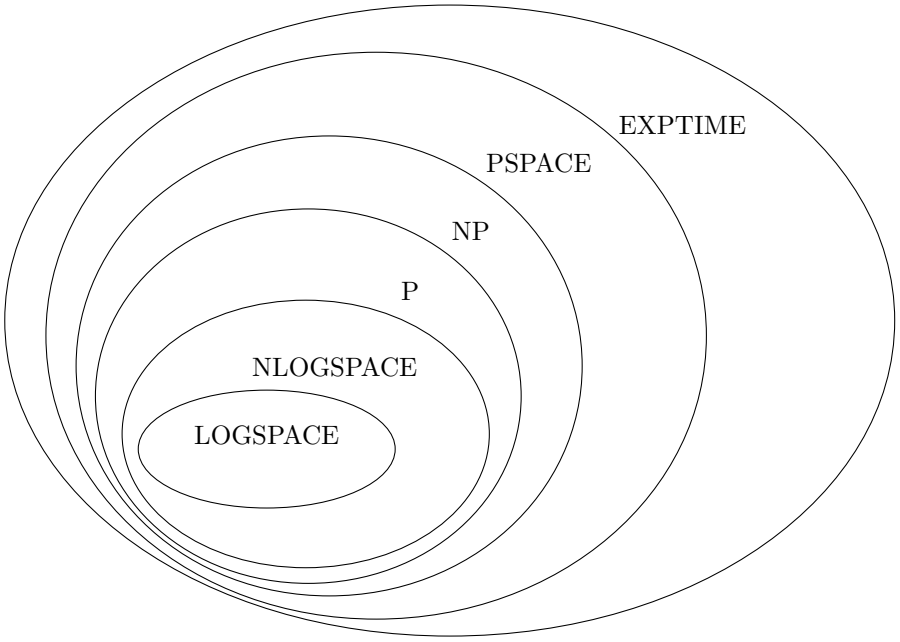


FIG. 13.1 – Inclusions entre les classes de complexité

13.3.1 Préliminaires

Pour éviter de compliquer inutilement certaines preuves, nous nous limiterons à des fonctions $f(n)$ *de complexité propre* : on suppose que la fonction $f(n)$ est non décroissante, c'est-à-dire que $f(n+1) \geq f(n)$, et qu'il existe une machine de Turing qui prend en entrée w et qui produit en sortie $\mathbf{1}^{f(n)}$ en temps $\mathcal{O}(n + f(n))$ et en espace $\mathcal{O}(f(n))$, où $n = |w|$.

Remarque 13.3 *Cela n'est pas vraiment restrictif, car toutes les fonctions usuelles non décroissantes, comme* $\log(n)$, n, n^2, $\cdots$, $n \log n$, $n!$ *vérifient ces propriétés. En outre, ces fonctions sont stables par somme, produit, et exponentielle.*

Remarque 13.4 *Nous avons besoin de cette hypothèse, car la fonction* $f(n)$ *pourrait ne pas être calculable, et donc il pourrait par exemple ne pas être possible d'écrire un mot de longueur* $f(n)$ *dans un des algorithmes qui suivent.*

Remarque 13.5 *Dans la plupart des assertions qui suivent, on peut se passer de cette hypothèse, au prix de quelques complications dans les preuves.*

13.3.2 Relations triviales

Un problème déterministe étant un problème non déterministe particulier, on a :

Théorème 13.5 SPACE(f(n)) ⊂ NSPACE(f(n)).

D'autre part :

Théorème 13.6 TIME(f(n)) ⊂ SPACE(f(n)).

Démonstration: Une machine de Turing écrit au plus une case à chaque étape. L'espace mémoire utilisé reste donc linéaire en le temps utilisé. Rappelons que l'on ne compte pas l'entrée dans l'espace mémoire. □

13.3.3 Temps non déterministe vs déterministe

De façon plus intéressante :

Théorème 13.7 *Pour tout langage de* NTIME(f(n))*, il existe un entier c tel que ce langage soit dans* TIME($c^{f(n)}$)*. Si l'on préfère :*

$$\text{NTIME(f(n))} \subset \bigcup_{c\in\mathbb{N}} \text{TIME}(c^{f(n)}).$$

Démonstration (principe): Soit L un problème de NTIME(f(n)). En utilisant le principe que l'on a utilisé dans le chapitre précédent, on sait qu'il existe un problème A tel que pour déterminer si un mot w de longueur n est dans L, il suffit de savoir s'il existe un mot $u \in \Sigma^*$ avec $\langle w, u\rangle \in A$, ce dernier test pouvant se faire en temps $f(n)$, où $n = |w|$. Puisqu'en temps $f(n)$ on ne peut pas lire plus que $f(n)$ lettres de u, on peut se limiter aux mots u de longueur $f(n)$. Tester si $\langle w, u\rangle \in A$ pour tous les mots $u \in \Sigma^*$ de longueur $f(n)$ se fait facilement en temps $\mathcal{O}(c^{f(n)}) + \mathcal{O}(f(n)) = \mathcal{O}(c^{f(n)})$, où $c > 1$ est le cardinal de l'alphabet Σ de la machine : tester tous les mots u peut par exemple se faire en comptant en base c. □

Remarque 13.6 *Pour écrire le premier u à tester de longueur $f(n)$, nous utilisons le fait que cela doit être possible : c'est le cas, si l'on suppose $f(n)$ de complexité propre. On voit donc l'intérêt ici de cette hypothèse implicite. Nous éviterons de discuter ce type de problèmes dans la suite, qui ne se posent pas de toute façon pour les fonctions $f(n)$ usuelles.*

13.3.4 Temps non déterministe vs espace

Théorème 13.8 NTIME(f(n)) ⊂ SPACE(f(n)).

Démonstration: On utilise exactement le même principe que dans la preuve précédente, si ce n'est que l'on parle d'espace. Soit L un problème de NTIME(f(n)). En utilisant la même idée que dans le chapitre précédent, on sait qu'il existe un problème A tel que pour déterminer si un mot w de longueur n est dans L, il suffit de savoir s'il existe $u \in \Sigma^*$ de longueur $f(n)$ avec $\langle w, u\rangle \in A$: on utilise un espace $\mathcal{O}(f(n))$ pour générer un à un les mots $u \in \Sigma^*$ de longueur $f(n)$ (par exemple en comptant en base c) puis on teste pour chacun si $\langle w, u\rangle \in A$, ce dernier test se faisant en temps $f(n)$, donc espace $f(n)$. Le même espace pouvant être utilisé pour chacun des mots u, au total cela se fait au total en espace $\mathcal{O}(f(n))$ pour générer les $u + \mathcal{O}(f(n))$ pour les tests, soit $\mathcal{O}(f(n))$. □

13.3.5 Espace non déterministe vs temps

Le problème de décision REACH suivant jouera un rôle important : on se donne un graphe orienté $G = (V, E)$, deux sommets u et v, et on cherche à décider s'il existe un chemin entre u et v dans G. On peut facilement se persuader que REACH est dans P.

A toute machine de Turing M (déterministe ou non) est associé un graphe orienté, son graphe des configurations, où les sommets correspondent aux configurations et les arcs correspondent à la fonction d'évolution en un pas de M, c'est-à-dire à la relation $\vdash$ entre configurations.

Chaque configuration X peut se décrire par un mot $[X]$ sur l'alphabet de la machine : si on fixe l'entrée w de longueur n, pour un calcul en espace $f(n)$, il y a moins de $\mathcal{O}(c^{f(n)})$ sommets dans ce graphe G_w, où $c > 1$ est le cardinal de l'alphabet de la machine.

Un mot w est accepté par la machine M si et seulement s'il y a un chemin dans ce graphe G_w entre l'état initial $X[w]$ codant l'entrée w, et un état acceptant. On peut supposer sans perte de généralité qu'il y a une unique configuration acceptante X^*. Décider l'appartenance d'un mot w au langage reconnu par M est donc résoudre le problème REACH sur $\langle G_w, X[w], X^* \rangle$.

On va alors traduire sous différentes formes tout ce que l'on sait sur le problème REACH. Tout d'abord, il est clair que le problème REACH se résout par exemple en temps et espace $\mathcal{O}(n^2)$, où n est le nombre de sommets, par un parcours en profondeur.

On en déduit :

Théorème 13.9 *Si $f(n) \geq \log n$, alors*

$$\mathrm{NSPACE(f(n))} \subset \bigcup_{c \in \mathbb{N}} \mathrm{TIME(c^{f(n)})}.$$

Démonstration: Soit L un problème de NSPACE(f(n)) reconnu par la machine de Turing non déterministe M. Par la discussion plus haut, on peut déterminer si $w \in L$ en résolvant le problème REACH sur $\langle G_w, X[w], X^* \rangle$: on a dit que cela pouvait se fait en temps quadratique en le nombre de sommets, soit en temps $\mathcal{O}(c^{2\mathcal{O}(f(n))})$, où $c > 1$ est le cardinal de l'alphabet de la machine.

□

13.3.6 Espace non déterministe vs espace déterministe

On va maintenant affirmer que REACH se résout en espace $\log^2(n)$.

Proposition 13.1 $\mathrm{REACH} \in \mathrm{SPACE(log^2 n)}$.

Démonstration: Soit $G = (V, E)$ le graphe orienté en entrée. Étant donnés deux sommets x et y de ce graphe, et i un entier, on note $CHEMIN(x, y, i)$ si et seulement s'il y a un chemin de longueur inférieure à 2^i entre x et y. On a $\langle G, u, v \rangle \in$ REACH si et seulement si $CHEMIN(u, v, \log(n))$, où n est le

nombre de sommets. Il suffit donc de savoir décider la relation $CHEMIN$ pour décider REACH.

L'astuce est de calculer $CHEMIN(x, y, i)$ récursivement en observant que l'on a $CHEMIN(x, y, i)$ si et seulement s'il existe un sommet intermédiaire z tel que $CHEMIN(x, z, i-1)$ et $CHEMIN(z, y, i-1)$. On teste alors à chaque niveau de la récursion chaque sommet possible z.

Pour représenter chaque sommet, il faut $\mathcal{O}(\log(n))$ bits. Pour représenter x, y, i, il faut donc $\mathcal{O}(\log(n))$ bits. Cela donne une récurrence de profondeur $\log(n)$, chaque étape de la récurrence nécessitant uniquement de stocker un triplet x, y, i et de tester chaque z de longueur $\mathcal{O}(\log(n))$. Au total, on utilise donc un espace $\mathcal{O}(\log(n)) * \mathcal{O}(\log(n)) = \mathcal{O}(\log^2(n))$. □

Théorème 13.10 (Savitch) *Si $f(n) \geq \log(n)$, alors*

$$\mathrm{NSPACE(f(n))} \subset \mathrm{SPACE(f(n)^2)}.$$

Démonstration: On utilise cette fois l'algorithme précédent pour déterminer s'il y a un chemin dans le graphe G_w entre $X[w]$ et X^*.

On remarquera que l'on a pas besoin de construire explicitement le graphe G_w mais que l'on peut utiliser l'algorithme précédent à la volée : plutôt que d'écrire complètement le graphe G_w, et ensuite de travailler en lisant dans cette écriture du graphe à chaque fois s'il y a un arc entre un sommet X et un sommet X', on peut de façon paresseuse, déterminer à chaque fois que l'on fait un test si $C_w(X, X') = 1$. □

Corollaire 13.1 PSPACE = NPSPACE.

Démonstration: On a $\bigcup_{c\in\mathbb{N}} \mathrm{SPACE(n^c)} \subset \bigcup_{c\in\mathbb{N}} \mathrm{NSPACE(n^c)}$, par le théorème 13.5, et $\bigcup_{c\in\mathbb{N}} \mathrm{NSPACE(n^c)} \subset \bigcup_{c\in\mathbb{N}} \mathrm{SPACE(n^{2c})} \subset \bigcup_{c\in\mathbb{N}} \mathrm{SPACE(n^c)}$ par le théorème précédent. □

13.3.7 Espace logarithmique non déterministe

En fait, on peut encore dire plus sur le problème REACH.

Théorème 13.11 REACH ∈ NLOGSPACE.

Démonstration: Pour déterminer s'il y a un chemin entre u et v dans un graphe G, on devine de façon non déterministe le chemin sommet par sommet. Cela nécessite uniquement de garder le sommet que l'on est en train de visiter en plus de u et de v. Chaque sommet se codant par $\mathcal{O}(\log(n))$ bits, l'algorithme est en espace $\mathcal{O}(\log(n))$. □

Définition 13.7 *Soit $\mathcal{C}$ une classe de complexité. On note* co-$\mathcal{C}$ *pour la classe des langages dont le complémentaire est dans la classe $\mathcal{C}$.*

On parle ainsi de problème coNP, coNLOGSPACE, etc...

En fait, on peut montrer :

Théorème 13.12 *Le problème* REACH *est aussi dans* coNLOGSPACE.

On en déduit :

Théorème 13.13 NLOGSPACE = coNLOGSPACE.

Démonstration: Il suffit de montrer que coNLOGSPACE ⊂ NLOGSPACE. L'inclusion inverse en découle car un langage L de NLOGSPACE aura son complémentaire dans coNLOGSPACE et donc aussi dans NLOGSPACE, d'où l'on déduit que L sera dans coNLOGSPACE.

Maintenant, pour décider si un mot w doit être accepté par un langage de coNLOGSPACE, on peut utiliser la machine non déterministe du théorème précédent qui utilise un espace logarithmique pour déterminer s'il existe un chemin entre $X[w]$ et X^* dans le graphe G_w. □

En fait, selon le même principe on peut montrer plus généralement.

Théorème 13.14 *Soit $f(n)$ une fonction telle que $f(n) \geq \log(n)$. Alors*

$$\text{NSPACE(f(n))} = \text{coNSPACE(f(n))}.$$

13.4 Résultats de séparation

13.4.1 Théorèmes de hiérarchie

On dit qu'une fonction $f(n) \geq \log(n)$ est *constructible en espace*, si la fonction qui envoie $\mathbf{1}^n$ sur la représentation binaire de $\mathbf{1}^{f(n)}$ est calculable en espace $\mathcal{O}(f(n))$.

La plupart des fonctions usuelles sont constructibles en espace. Par exemple, n^2 est constructible en espace puisqu'une machine de Turing peut obtenir n en binaire en comptant le nombre de $\mathbf{1}$, et écrire n^2 en binaire par n'importe quelle méthode pour multiplier n par lui-même. L'espace utilisé est certainement en $\mathcal{O}(n^2)$.

Théorème 13.15 (Théorème de hiérarchie) *Pour toute fonction $f : \mathbb{N} \to \mathbb{N}$ constructible en espace, il existe un langage L qui est décidable en espace $\mathcal{O}(f(n))$ mais qui ne l'est pas en espace $o(f(n))$.*

Démonstration: On considère le langage (très artificiel) L qui est décidé par la machine B suivante :

- sur une entrée w de taille n, B calcule $f(n)$ en utilisant la constructibilité en espace de f, et réserve un espace $f(n)$ pour la simulation qui va venir ;
- si w n'est pas de la forme $\langle A \rangle \mathbf{10}^*$, où $\langle A \rangle$ est le codage d'une machine A, alors la machine B refuse ;
- sinon, B simule la machine A sur le mot w pendant au plus $2^{f(n)}$ étapes pour déterminer si A accepte en ce temps avec un espace inférieur à $f(n)$:
  - si A accepte en ce temps et cet espace, alors B refuse ;
  - sinon B accepte.

Par construction, L est dans SPACE(f(n)), car la simulation n'introduit qu'un facteur constant dans l'espace nécessaire : plus concrètement, si A utilise un espace $g(n) \leq f(n)$, alors B utilise un espace au plus $dg(n)$ pour une constante d.

Supposons que L soit décidé par une machine de Turing A en espace $g(n)$ avec $g(n) = o(f(n))$. Il doit exister un entier n_0 tel que pour $n \geq n_0$, on ait $dg(n) < f(n)$. Par conséquent, la simulation par B de A sera bien complète sur une entrée de longueur n_0 ou plus.

Considérons ce qui se passe lorsque B est lancé sur l'entrée $\langle A \rangle \mathbf{10}^{n_0}$. Puisque cette entrée est de taille plus grande que n_0, B répond l'inverse de la machine A sur la même entrée. Donc B et A ne décident pas le même langage, et donc la machine A ne décide pas L, ce qui mène à une contradiction.

Par conséquent L n'est pas décidable en espace $o(f(n))$. □

Autrement dit :

Théorème 13.16 (Théorème de hiérarchie) *Soient $f, f' : \mathbb{N} \to \mathbb{N}$ des fonctions constructibles en espace telles que $f(n) = o(f'(n))$. Alors l'inclusion* SPACE(f) ⊂ SPACE(f′) *est stricte.*

Sur le même principe, on peut prouver :

Théorème 13.17 (Théorème de hiérarchie) *Soient $f, f' : \mathbb{N} \to \mathbb{N}$ des fonctions constructibles en espace telles que $f(n) = o(f'(n))$. Alors l'inclusion* NSPACE(f) ⊂ NSPACE(f′) *est stricte.*

13.4.2 Applications

On en déduit :

Théorème 13.18 NLOGSPACE ⊊ PSPACE.

Démonstration: La classe NLOGSPACE est complètement incluse dans SPACE($\log^2$ n) par le théorème de Savitch. Hors cette dernière est un sous-ensemble strict de SPACE(n), qui est inclus dans PSPACE. □

Sur le même principe, on obtient :

Définition 13.8 *Soit*

$$\text{EXPSPACE} = \bigcup_{c \in \mathbb{N}} \text{SPACE}(2^{n^c}).$$

Théorème 13.19 PSPACE ⊊ EXPSPACE.

Démonstration: La classe PSPACE est complètement incluse dans, par exemple, SPACE($n^{\log(n)}$). Hors cette dernière est un sous-ensemble strict de SPACE(2^n), qui est inclus dans EXPSPACE. □

13.5 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture de [Sipser, 1997], [Papadimitriou, 1994] et de [Lassaigne and de Rougemont, 2004].

Un ouvrage de référence et contenant les derniers résultats du domaine est [Arora and Barak, 2009].

Bibliographie Ce chapitre contient des résultats standards en complexité. Nous nous sommes essentiellement inspirés de leur présentation dans les ouvrages [Sipser, 1997] et [Papadimitriou, 1994].

Table des figures

Index

Bibliographie

[Arnold and Guessarian, 2005] Arnold, A. and Guessarian, I. (2005). *Mathématiques pour l'informatique*. Ediscience International.

[Arora and Barak, 2009] Arora, S. and Barak, B. (2009). *Computational Complexity : A Modern Approach*. Cambridge University Press.

[Carton, 2008] Carton, O. (2008). Langages formels, calculabilité et complexité.

[Cori et al., 2010] Cori, R., Hanrot, G., Kenyon, C., and Steyaert, J.-M. (2010). Conception et analyse d'algorithmes. Cours de l'Ecole Polytechnique.

[Cori and Lascar, 1993a] Cori, R. and Lascar, D. (1993a). *Logique mathématique. Volume I*. Mason.

[Cori and Lascar, 1993b] Cori, R. and Lascar, D. (1993b). *Logique Mathématique, volume II*. Mason.

[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., and Soria, M. (1993). *Types de donnees et algorithmes*. Ediscience International.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability*. W. H. Freeman and Co.

[Hopcroft et al., 2001] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition.

[Jones, 1997] Jones, N. (1997). *Computability and complexity, from a programming perspective*. MIT press.

[Kleinberg and Tardos, 2005] Kleinberg, J. and Tardos, E. (2005). *Algorithm design*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

[Kozen, 1997] Kozen, D. (1997). *Automata and computability*. Springer Verlag.

[Lassaigne and de Rougemont, 2004] Lassaigne, R. and de Rougemont, M. (2004). *Logic and complexity*. Discrete Mathematics and Theoretical Computer Science. Springer.

[Matiyasevich, 1970] Matiyasevich, Y. (1970). Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR*, 191(2) :279–282.

[Mendelson, 1997] Mendelson, E. (1997). *Introduction to mathematical logic*. Chapman & Hall/CRC.

[Nerode and Shore, 1997] Nerode, A. and Shore, R. (1997). *Logic for applications*. Springer Verlag.

[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 and Flajolet, 1996] Sedgewick, R. and 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*.

[Wolper, 2001] Wolper, P. (2001). *Introduction à la calculabilité : cours et exercices corrigés*. Dunod.