Algorithmique et calculabilité

Émilie Charlier

Année académique 2025-2026

Informations générales

— **Code cours :** INFO0212
— **Titulaire :** Émilie Charlier
— **Assistant :** Bastian Espinoza
— **Contact :** Campus du Sart-Tilman - zone Polytech 1
Institut de Mathématique B37, bureau 1/28
— **Horaire du cours :** Q1, lundi de 10h15 à 12h45 et jeudi de 13h à 15h.
— **Locaux :** Voir sur Celcat
— **Évaluation :** Examen oral portant sur la théorie et applications directes de celle-ci. Un exercice sera également posé sur feuille.
— **Modulation :** 8 crédits

Chapitre 1

Introduction

Lors du 2^{e} congrès international des mathématiques en 1900 à Paris, David Hilbert présente sa célèbre liste de 23 problèmes qu'il considère comme un programme de recherche à destination des mathématiciens du 20^{e} siècle. Ces problèmes sont de difficultés et de natures différentes. Certains ont été résolus rapidement, certains ont demandé un travail plus conséquent, d'autres ont été jugés mal posés, et d'autres encore sont toujours ouverts ! Parmi ces 23 problèmes, le dixième d'entre eux concerne, avant l'heure, une question de décidabilité. Il s'énonce comme suit : *Étant donné un polynôme multivarié P à coefficients entiers, c'est-à-dire $P \in \mathbb{Z}[X_1, \ldots, X_n]$ pour un certain n, déterminer si l'équation $P(x_1, \ldots, x_n) = 0$ possède une solution entière.* Une telle équation s'appelle une équation diophantienne et les ensembles de la forme

$$\{(a_1, \ldots, a_m) \in \mathbb{N}^m : \exists (b_1, \ldots, b_n) \in \mathbb{Z}^n,\ P(a_1, \ldots, a_m, b_1, \ldots, b_n) = 0\}$$

pour un certain polynôme $P \in \mathbb{Z}[X_1, \ldots, X_{m+n}]$ sont appelés les ensembles diophantiens.

Le mot « déterminer » de la formulation de Hilbert renvoie à une notion intuitive de « procédure de décision » ou d'« algorithme », notion qui faisait défaut en 1900. C'est grâce aux travaux d'Alonzo Church et d'Alan Turing des années 1930, initiateurs de la théorie de la calculabilité, que le dixième problème de Hilbert a pu être reformulé de façon rigoureuse. Et ce n'est qu'en 1970 que Youri Matiiassevitch, à 23 ans à peine, a répondu au dixième problème de Hilbert par la négative en démontrant qu'il s'agissait là d'un problème *indécidable*, un concept inconnu des mathématiciens de 1900. La preuve de Matiiassevitch s'appuie sur les travaux précédents de Julia Robinson, c'est pourquoi on parle en général du théorème de Matiiassevitch-Robinson. Celui-ci s'énonce comme suit : *Les ensembles diophantiens coïncident avec les ensembles d'entiers récursivement énumérables.* Il est alors une conséquence immédiate des travaux de Turing que le dixième problème de Hilbert est indécidable.

Dans ce cours, nous allons formaliser les concepts de procédure effective et de fonctions calculables, et nous comprendrons ce que signifient les termes « ensembles d'entiers récursivement énumérables » et « problème indécidable » évoqués précédemment.

Dans un deuxième temps, nous étudierons la théorie de la complexité des algorithmes. Parmi les problèmes décidables, nous distinguerons les problèmes dits « faciles » des problèmes dits « difficiles ». Nous serons alors à même de comprendre un autre célèbre problème des mathématiques, à savoir le problème « P vs NP ». Il s'agit cette fois d'un des 7 problèmes du prix du millénaire posés en 2000, soit 100 ans après Hilbert, et pour chacun desquels l'Institut de mathématiques de Clay à Boston offre un prix d'un million de dollars à quiconque y apportera une solution !

Chapitre 2

Calculabilité

La thèse de Church-Turing affirme que les fonctions calculables par une procédure effective, quelle que soit la définition qu'on puisse donner de ce concept, sont calculables par une machine de Turing. Cette thèse ne peut être un théorème puisque, précisément, une définition de la notion de procédure effective nous fait défaut. Il faut donc plutôt la voir comme la proposition de définir une procédure effective, ou un algorithme, par une machine de Turing. Cette thèse sera étayée de plusieurs manières au fur et à mesure de l'avancement de ce chapitre.

2.1 Rappels de théorie des langages

Un alphabet est un ensemble fini non vide. Les éléments d'un alphabet sont appelés les lettres de cet alphabet. Un mot fini (resp. infini) sur un alphabet est une suite finie (resp. infinie) de lettres de cet alphabet. On note $|w|$ la longueur d'un mot fini w. On note la k^{e} lettre d'un mot w par $w[k]$ et on note $w[k,\ell]$ le facteur $w[k]w[k+1]\dots w[\ell]$ (avec la convention que $w[k,\ell] = \varepsilon$ lorsque $\ell < k$). Pour un mot fini w et $n \in \mathbb{N}$, on note w^n la concaténation de n fois le mot w et on note w^ω le mot infini formé d'une infinité de répétition de w. L'ensemble des mots fini écrits sur un alphabet A est noté A^*. Le mot vide est noté ε. L'opération de concaténation des mots finis munit A^* d'une structure de monoïde avec ε pour neutre.

2.2 Machines de Turing

Tout au long du cours, nous considérerons # comme un symbole particulier, appelé le *symbole blanc*. De même, les symboles `L` et `R`, utilisés pour « left » et « right », joueront également un rôle spécifique.

Définition 2.2.1. Une *machine de Turing* est la donnée d'un quintuple $\mathcal{M} = (Q, q_0, h, A, \delta)$ où

- Q est un ensemble fini non vide, appelé ensemble des *états* ;
- q_0 et h sont des éléments privilégiés de Q, appelé *état initial* et *état final* respectivement ;
- A est un alphabet contenant le symbole blanc # mais ne contenant pas les symboles `L` et `R` ;
- $\delta\colon Q\backslash\{h\} \times A \to Q \times (A \cup \{\texttt{L}, \texttt{R}\})$ est une fonction partielle, appelée *fonction de transition*.

Une machine de Turing $\mathcal{M} = (Q, q_0, h, A, \delta)$ peut être représentée par un graphe orienté dont les sommets sont les états et où pour tous $p, q \in Q$, $a \in A$ et $x \in A \cup \{\texttt{L}, \texttt{R}\}$ tels que

$\delta(p, a) = (q, x)$, on dessine un arc de p vers q étiqueté par a, x. De plus, l'état initial est désigné par une flèche entrante et l'état final par un double cercle.

Exemple 2.2.2. Le graphe de la Figure 2.1 représente la machine de Turing

$$(\{0, 1, 2, 3, 4, 5, 6, 7, 8\}, 0, 8, \{\#, \mathtt{u}\}, \delta)$$

dont la fonction de transition δ est donnée par la Table 2.1.

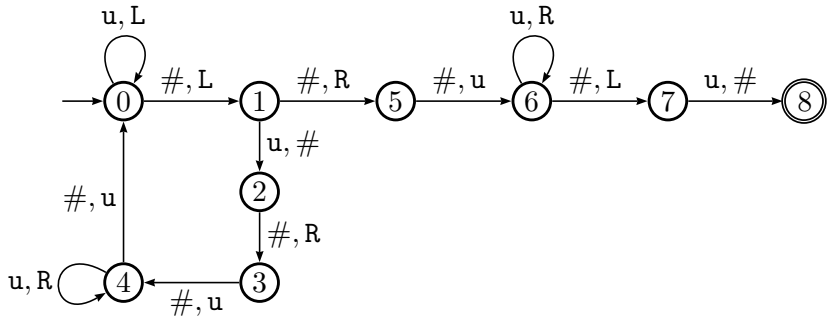


FIGURE 2.1 – Machine de Turing calculant la multiplication par deux en représentation unaire

	#	u
0	(1, L)	(0, L)
1	(5, R)	(2, #)
2	(3, R)	/
3	(4, u)	/
4	(0, u)	(4, R)
5	(6, u)	/
6	(7, L)	(6, R)
7	/	(8, #)

TABLE 2.1 – Table de transition de δ.

Définition 2.2.3. Une *configuration mémoire* est un couple $(w, k) \in A^{\mathbb{N}_0} \times \mathbb{N}_0$, où le mot infini w ne contient qu'un nombre fini de fois le symbole blanc #. Le mot infini w est appelé le ruban mémoire et l'entier k est un pointeur qui pointe sur la k^{e} lettre de w. On dit aussi que k désigne une cellule référencée qui contient le symbole $w[k]$. Pour simplifier les écritures, on renseigne souvent uniquement la partie significative d'une configuration mémoire. La *partie significative* r *d'une configuration mémoire* (w, k) telle que $w[\ell] \neq \#$ et $w[n] = \#$ pour tout $n > \ell$ est définie par

$$r = \begin{cases} w[1, k-1]\,\underline{w[k]}\,w[k+1, \ell] & \text{si } \ell \geqslant k \\ w[1, \ell]\#^{k-\ell-1}\underline{\#} & \text{si } \ell < k. \end{cases}$$

Une *configuration machine* est un triplet (p, w, k) où p est un état et (w, k) est une configuration mémoire. Une configuration machine (p, w, k) est souvent notée $p.r$ où r est la partie significative de la configuration mémoire (w, k).

Par la suite, lorsque le contexte permet de lever toute ambigüité, on s'autorisera à parler simplement de configuration machine et de configuration mémoire plutôt que de partie significative de celles-ci.

Une machine de Turing $\mathcal{M}$ agit sur les configurations machine de la manière suivante.

— Si on se trouve dans la configuration machine (p, w, k) et que $\delta(p, w[k]) = (q, a)$ avec $a \in A$, alors on bascule dans la configuration machine (q, w', k) où

$$w'[n] = \begin{cases} w[n] & \text{si } n \neq k \\ a & \text{si } n = k. \end{cases}$$

— Si on se trouve dans la configuration machine (p, w, k) et que $\delta(p, w[k]) = (q, \mathtt{R})$, alors on bascule dans la configuration machine $(q, w, k+1)$.
— Si on se trouve dans la configuration machine (p, w, k) avec $k \geqslant 2$ et que $\delta(p, w[k]) = (q, \mathtt{L})$, alors on bascule dans la configuration machine $(q, w, k-1)$.

La notation $C \vdash C'$ signifie que C' est une configuration machine atteignable depuis C. On note $C \vdash^* C'$ s'il existe $j \geqslant 0$, des configurations machine $C_0, \ldots, C_j$ telles que l'on ait

— $C = C_0$;
— $C' = C_j$;
— $C_i \vdash C_{i+1}$ for all $i \in \{0, \ldots, j-1\}$.

Une *configuration pendante* est une configuration machine (p, w, k), avec $p \neq h$, depuis laquelle aucune configuration machine n'est atteignable. Cela peut se produire soit lorsque la fonction de transition δ n'est pas définie en $(p, w[k])$ [1], soit lorsque $k = 1$ et $\delta(p, w[1]) = (q, \mathtt{L})$. Enfin, si $d = h$, on parle de configuration d'arrêt.

Trois situations peuvent se produire lorsqu'on lance une machine de Turing à partir d'une configuration machine donnée.

1. La machine de Turing aboutit dans une configuration d'arrêt en un nombre fini de transitions, auquel cas on dit que la machine de Turing s'arrête.
2. La machine de Turing atteint une configuration pendante.
3. Une infinité de transitions successives sont possibles, auquel cas on dit que la machine de Turing ne s'arrête pas.

2.3 Fonctions calculables par machines de Turing

Dans ce chapitre, d sera toujours un naturel.

Définition 2.3.1. Soient $A_1, \ldots, A_{d+1}$ des alphabets ne contenant pas le symbole blanc #. Une fonction $f \colon A_1^* \times \cdots \times A_d^* \to A_{d+1}^*$ est *calculable* s'il existe une machine de Turing $\mathcal{M} = (Q, q_0, h, B, \delta)$ telle que $\bigcup_{i=1}^{d+1} A_i \subseteq B$ et telle que pour tout $(w_1, \ldots, w_d) \in A_1^* \times \cdots \times A_d^*$, on ait

$$q_0.\#w_1\#\cdots\#w_d\underline{\#} \vdash^* h.\#f(w_1, \ldots, w_d)\underline{\#}.$$

Dans la suite, nous allons principalement considérer des fonctions numériques, c'est-à-dire, à arguments et valeurs naturels. On note $\mathcal{F}_d$ l'ensemble des fonctions de $\mathbb{N}^d$ dans $\mathbb{N}$ et on note $\mathcal{F} = \bigcup_{d \in \mathbb{N}} \mathcal{F}_d$. Afin d'introduire la notion d'une fonction numérique calculable comme un cas particulier de la définition précédente, nous devons choisir un codage des entiers par des mots finis. Dans ce cours, nous travaillerons principalement avec le codage unaire des naturels : on considère un symbole spécial $\mathtt{u}$, et on code tout naturel n par le mot $\mathtt{u}^n$.

Définition 2.3.2. Une fonction f de $\mathcal{F}_d$ est *calculable* s'il existe une machine de Turing $\mathcal{M} = (Q, q_0, h, A, \delta)$ telle que $\mathtt{u} \in A$ et pour tout $(n_1, \ldots, n_d) \in \mathbb{N}^d$, on ait

$$q_0.\#\,\mathtt{u}^{n_1}\,\#\cdots\#\,\mathtt{u}^{n_d}\,\underline{\#} \vdash^* h.\#\,\mathtt{u}^{f(n_1,\ldots,n_d)}\,\underline{\#}.$$

On note $\mathcal{C}$ l'ensemble des fonctions de $\mathcal{F}$ qui sont calculables.

1. N'oubliez pas que δ est une fonction partielle.

Exemple 2.3.3. Montrons que la machine de Turing de la figure 2.1 calcule la fonction $f\colon \mathbb{N} \to \mathbb{N},\ m \mapsto 2m$. En effet, pour tout $m, n \in \mathbb{N}$ avec $m \geqslant 1$, on a

$$\begin{aligned} 0.\#\,\mathtt{u}^m\,\underline{\#}\,\mathtt{u}^n &\vdash 1.\#\,\mathtt{u}^{m-1}\,\underline{\mathtt{u}}\#\,\mathtt{u}^n \\ &\vdash 2.\#\,\mathtt{u}^{m-1}\,\underline{\#}\#\,\mathtt{u}^n \\ &\vdash 3.\#\,\mathtt{u}^{m-1}\,\#\underline{\#}\,\mathtt{u}^n \\ &\vdash 4.\#\,\mathtt{u}^{m-1}\,\#\underline{\mathtt{u}}\,\mathtt{u}^n \\ &\vdash^* 4.\#\,\mathtt{u}^{m-1}\,\#\,\mathtt{u}^{n+1}\,\underline{\#} \\ &\vdash 0.\#\,\mathtt{u}^{m-1}\,\#\,\mathtt{u}^{n+1}\,\underline{\mathtt{u}} \\ &\vdash^* 0.\#\,\mathtt{u}^{m-1}\,\underline{\#}\,\mathtt{u}^{n+2}\,. \end{aligned}$$

En itérant cet argument, on obtient que pour tout $m \in \mathbb{N}$, on a

$$0.\#\,\mathtt{u}^m\,\underline{\#} \vdash 0.\#\underline{\#}\,\mathtt{u}^{2m}\,.$$

De plus, on a

$$\begin{aligned} 0.\#\underline{\#}\,\mathtt{u}^{2m} &\vdash 1.\underline{\#}\#\,\mathtt{u}^{2m} \\ &\vdash 5.\#\underline{\#}\,\mathtt{u}^{2m} \\ &\vdash 6.\#\underline{\mathtt{u}}\,\mathtt{u}^{2m} \\ &\vdash^* 6.\#\,\mathtt{u}^{2m+1}\,\underline{\#} \\ &\vdash^* 7.\#\,\mathtt{u}^{2m}\,\underline{\mathtt{u}} \\ &\vdash^* 8.\#\,\mathtt{u}^{2m}\,\underline{\#}. \end{aligned}$$

D'où la conclusion.

2.4 Composition de machines de Turing

Écrire une machine de Turing réalisant une tâche donnée, même simple, peut rapidement s'avérer fastidieux. C'est la raison pour laquelle on introduit les organigrammes, c'est-à-dire des graphes représentant des compositions conditionnées de machines de Turing.

Définition 2.4.1. Soient $\mathcal{M} = (Q, q_0, h, A, \delta)$ et $\mathcal{M}' = (Q', q_0', h', A, \delta')$ deux machines de Turing ayant le même alphabet mais des ensembles d'états disjoints, et soit C un sous-ensemble de A. On définit la *composition de $\mathcal{M}$ et $\mathcal{M}'$ conditionnellement à C* la machine de Turing

$$\mathcal{M} \xrightarrow{C} \mathcal{M}' = (Q \cup Q', q_0, h', A, \delta'')$$

où la fonction de transition δ'' est définie par

- $\delta''_{|Q\setminus\{h\}\times A} = \delta$
- $\delta''_{|Q'\setminus\{h'\}\times A} = \delta'$
- $\delta''(h, a) = (q_0', a)$ pour tout $a \in C$
- $\delta''(h, a) = (h', a)$ pour tout $a \in A\setminus C$.

Si le sous-ensemble C est $\{a_1, \ldots, a_n\}$, on note $\xrightarrow{a_1,\ldots,a_n}$, et si le sous-ensemble C est $A\setminus\{a_1, \ldots, a_n\}$, on écrit $\xrightarrow{\underline{a_1},\ldots,\underline{a_n}}$. Lorsque $C = A$, on écrit $\mathcal{M} \to \mathcal{M}'$, voire même simplement $\mathcal{M}\,\mathcal{M}'$.

Dit de façon informelle, l'idée est de lancer d'abord la machine de Turing $\mathcal{M}$, et si celle-ci atteint une configuration d'arrêt (h, w, k) (où h est l'état terminal de $\mathcal{M}$) et que $w[k] \in C$, alors on lance la machine $\mathcal{M}'$ sur la configuration initiale (q'_0, w, k) (où q'_0 est l'état initial de $\mathcal{M}'$). Dans le cas où $\mathcal{M}$ atteint une configuration d'arrêt avec (w, k) comme configuration mémoire mais que $w[k] \notin C$, la machine de Turing $\mathcal{M} \xrightarrow{C} \mathcal{M}'$ s'arrête avec la même configuration mémoire (w, k).

Si l'on souhaite composer une machine de Turing avec elle-même un certain nombre n de fois, on utilisera la définition précédente avec n copies de la machine. En effet, dans cette définition, il est demandé que les ensembles d'états soient disjoints. On peut aussi vouloir composer une machine de Turing avec elle-même aussi longtemps qu'une condition est satisfaite. Dans ce cas, il nous faut agir différemment.

Définition 2.4.2. Soit $\mathcal{M} = (Q, q_0, h, A, \delta)$ une machine de Turing et soit C un sous-ensemble de A. On définit la *composition répétée de $\mathcal{M}$ conditionnellement à C* la machine de Turing

$$\longrightarrow \mathcal{M} \circlearrowright C \ = (Q \cup \{h'\}, q_0, h', A, \delta')$$

où $h' \notin Q$ et la fonction de transition δ' est définie par

- $\delta'_{|Q\backslash\{h\}\times A} = \delta$
- $\delta'(h, a) = (q_0, a)$ pour tout $a \in C$
- $\delta''(h, a) = (h', a)$ pour tout $a \in A\backslash C$.

Si le sous-ensemble C est $\{a_1, \ldots, a_n\}$, on note

$$\longrightarrow \mathcal{M} \circlearrowright a_1, \ldots, a_n$$

et si le sous-ensemble C est $A\backslash\{a_1, \ldots, a_n\}$, on écrit

$$\longrightarrow \mathcal{M} \circlearrowright \underline{a_1}, \ldots, \underline{a_n}$$

Définition 2.4.3. Un *organigramme* est un graphe orienté avec une racine dont les sommets sont des machines de Turing et les arcs représentent les compositions conditionnées entre ces machines, avec la contrainte que les conditions des arcs sortant d'un même sommet soient mutuellement exclusives. La racine de l'organigramme est représentée par une flèche entrante.

Étant donné un organigramme, on obtient en combinant les définitions 2.4.1 et 2.4.2 une machine de Turing exécutant les enchaînements de machines de Turing prescrits par l'organigramme. Par exemple, considérons l'organigramme de la figure 2.2, dont les sommets $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4$ sont des machines de Turing d'alphabet $\{\#, a, b\}$.

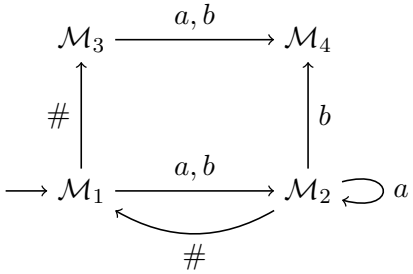


FIGURE 2.2 – Un organigramme.

Un schéma de la construction de la machine de Turing décrite par cet organigramme est représenté aux figures 2.3 et 2.4, où on a ajouté un état final unique tenant compte de toutes les possibilités de mener à une configuration d'arrêt.

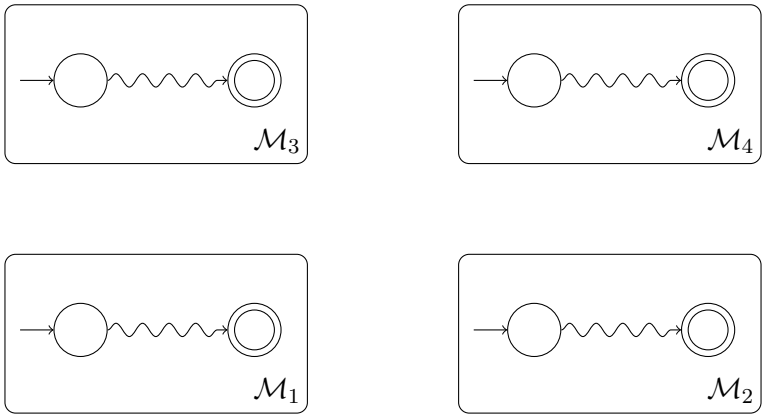


FIGURE 2.3 – Machines correspondant aux sommets de l'organigramme. Pour chacune d'elle, uniquement leurs états initial et final sont dessinés.

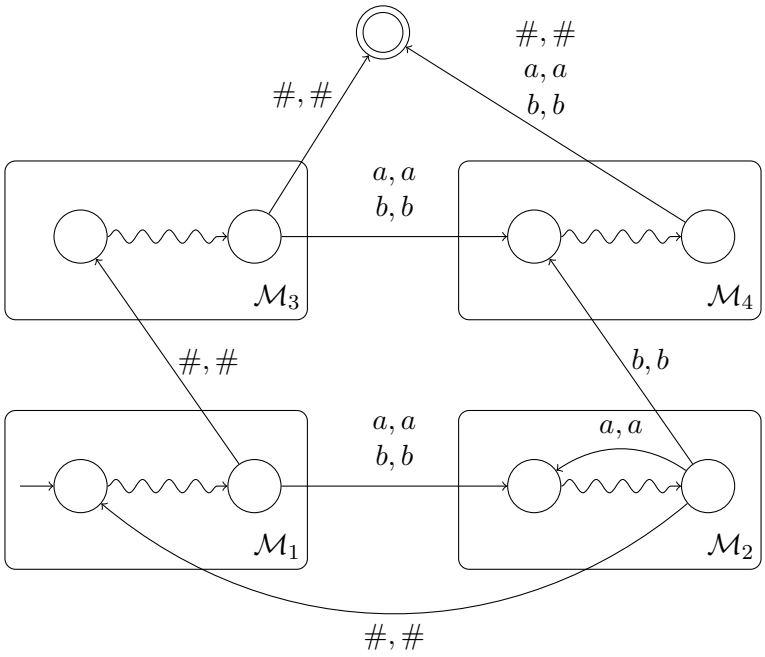


FIGURE 2.4 – Modification de la fonction de transition en accord avec les instructions prescrites par l'organigramme. L'état initial est celui de $\mathcal{M}_1$ et l'état final est un état nouvellement créé.

Toujours dans le but de faciliter la construction de machines de Turing, nous distinguons quelques machines de Turing de base.

1. $\mathcal{L}$ est une machine de Turing qui va une fois à gauche, inconditionnellement.
   Une machine de Turing $\mathcal{L}$ est représentée à la figure 2.5, où il y a autant d'arcs de label $\sigma, \mathtt{L}$ que de lettres σ dans l'alphabet.

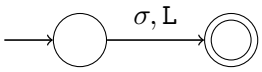


FIGURE 2.5 – Machine de Turing $\mathcal{L}$.

2. $\mathcal{R}$ est une machine de Turing qui va une fois à droite, inconditionnellement.
   Une machine de Turing $\mathcal{R}$ est représentée à la figure 2.6, où il y a autant d'arcs de label $\sigma, \mathtt{R}$ que de lettres σ dans l'alphabet.

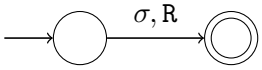


FIGURE 2.6 – Machine de Turing $\mathcal{R}$.

3. Pour une lettre a, a est une machine de Turing qui remplace le contenu de la cellule référence par a, inconditionnellement.
   Une machine de Turing a est représentée à la figure 2.7, où il y a autant d'arcs de label σ, a que de lettres σ dans l'alphabet.

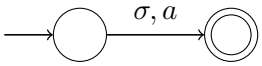


FIGURE 2.7 – Machine de Turing a.

4. Pour une lettre a, $\mathcal{L}_a$ est une machine de Turing qui déplace la tête de lecture sur la première cellule à gauche contenant a.
   Un organigramme pour $\mathcal{L}_a$ est représenté à la figure 2.8.

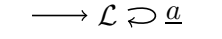


FIGURE 2.8 – Machine de Turing $\mathcal{L}_a$.

5. Pour une lettre a, $\mathcal{R}_a$ est une machine de Turing qui déplace la tête de lecture sur la première cellule à droite contenant a.
   Un organigramme pour $\mathcal{R}_a$ est représenté à la figure 2.9.

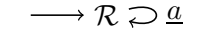


FIGURE 2.9 – Machine de Turing $\mathcal{R}_a$.

6. Pour une lettre a, $\mathcal{L}_{\underline{a}}$ est une machine de Turing qui va à gauche tant que la cellule référence contient a.
   Un organigramme pour $\mathcal{L}_{\underline{a}}$ est représenté à la figure 2.10.

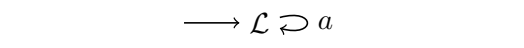


FIGURE 2.10 – Machine de Turing $\mathcal{L}_{\underline{a}}$.

7. Pour une lettre a, $\mathcal{R}_{\underline{a}}$ est une machine de Turing qui va à droite tant que la cellule référence contient a.
   Un organigramme pour $\mathcal{R}_{\underline{a}}$ est représenté à la figure 2.11.

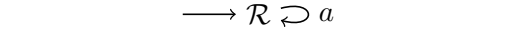


FIGURE 2.11 – Machine de Turing $\mathcal{R}_{\underline{a}}$.

8. Pour $d \geqslant 1$, $\mathcal{S}_{L,d}$ est une machine de Turing qui réalise l'action suivante :

$$q_0.x\#w_1\#w_2\#\cdots\#w_d\underline{\#} \vdash^* h.xw_1\#w_2\#\cdots\#w_d\underline{\#}$$

   où $w_1, \ldots, w_d$ sont des mots finis ne contenant pas le symbole blanc $\#$ et x est un mot fini quelconque.
   Détaillons une construction de $\mathcal{S}_{L,d}$. On procède par récurrence sur d. L'organigramme de la figure 2.12 convient pour $\mathcal{S}_{L,1}$.

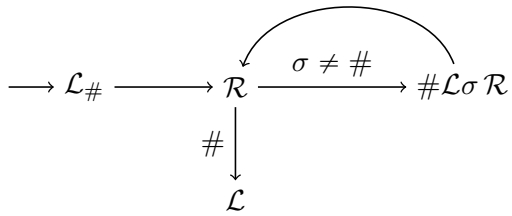


FIGURE 2.12 – $\mathcal{S}_{L,1}$.

   Supposons maintenant disposer d'une machine de Turing $\mathcal{S}_{L,d-1}$ pour $d \geqslant 2$. Alors l'organigramme de la figure 2.13 convient pour $\mathcal{S}_{L,d}$.

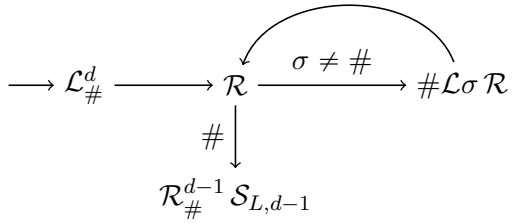


FIGURE 2.13 – $\mathcal{S}_{L,d}$ à partir de $\mathcal{S}_{L,d-1}$.

9. Pour $d \geqslant 1$, $\mathcal{S}_{R,d}$ est une machine de Turing qui réalise l'action suivante :

$$q_0.x\#w_1\#w_2\#\cdots\#w_d\underline{\#} \vdash^* h.x\#\#w_1\#w_2\#\cdots\#w_d\underline{\#}$$

   où $w_1, \ldots, w_k$ sont des mots finis ne contenant pas le symbole blanc $\#$ et x est un mot fini quelconque.
10. Pour $d \geqslant 1$, $\mathcal{C}_d$ est une machine de Turing qui réalise l'action suivante :

$$q_0.x\#w_1\#w_2\#\cdots\#w_d\underline{\#} \vdash^* h.x\#w_1\#w_2\#\cdots\#w_d\#w_1\underline{\#}$$

où $w_1, \dots, w_d$ sont des mots finis ne contenant pas le symbole blanc # et x est un mot fini quelconque.

11. Pour $d \geqslant 1$, $\mathcal{E}_d$ est une machine de Turing qui réalise l'action suivante :

$$q_0.x\#w_1\#w_2\#\cdots\#w_d\underline{\#} \vdash^* h.x\#w_2\#\cdots\#w_d\underline{\#}$$

où $w_1, \dots, w_d$ sont des mots finis ne contenant pas le symbole blanc # et x est un mot fini quelconque.

Les constructions des machines $\mathcal{S}_{R,d}, \mathcal{C}_d, \mathcal{E}_d$ sont laissées en exercices.

2.5 Fonctions récursives

Nous allons maintenant présenter une deuxième famille de fonctions, celle des fonctions récursives. Nous verrons ensuite que cette famille de fonctions se révèle en fait être identique à celle des fonctions calculables par machine de Turing. Ce premier résultat constitue notre premier argument en faveur de la thèse de Church-Turing.

2.5.1 Fonctions récursives primitives

Nous commençons par définir la sous-famille des fonctions récursives primitives.

Appelons *fonctions initiales* les fonctions suivantes.

— La fonction $\underline{0}$ de $\mathcal{F}_0$. Cette fonction ne prend pas d'argument et rend la valeur 0. On peut l'identifier au naturel 0.

— La fonction $\sigma\colon \mathbb{N} \to \mathbb{N},\ n \mapsto n+1$. Cette fonction est appelée la *fonction successeur*.

— Pour tout entier $d \geqslant 1$ et tout $i \in \{1, \dots, d\}$, la fonction $P_{d,i}\colon \mathbb{N}^d \to \mathbb{N},\ (n_1, \dots, n_d) \mapsto n_i$. Ces fonctions sont appelées les *projections*.

Ensuite, nous considérons deux règles de formation de nouvelles fonctions à partir d'autres fonctions.

— *Composition.*
Soient $k \in \mathbb{N}_0$, $h_1, \dots, h_k \in \mathcal{F}_d$ et $g \in \mathcal{F}_k$. La fonction composée $g(h_1, \dots, h_k) \in \mathcal{F}_d$ est la fonction $g(h_1, \dots, h_k)\colon \mathbb{N}^d \to \mathbb{N},\ \mathbf{m} \mapsto g(h_1(\mathbf{m}), \dots, h_k(\mathbf{m}))$. Si $k = 1$, on note $g \circ h$ au lieu de $g(h)$.

— *Récursion primitive.*
Soient $g \in \mathcal{F}_d$ et $h \in \mathcal{F}_{d+2}$. La fonction $f \in \mathcal{F}_{d+1}$ définie par récursion primitive à partir de g et h est définie comme suit : pour tous $\mathbf{m} \in \mathbb{N}^d$ et $n \in \mathbb{N}$, $f(\mathbf{m}, 0) = g(\mathbf{m})$ et $f(\mathbf{m}, n+1) = h(\mathbf{m}, n, f(\mathbf{m}, n))$.

Définition 2.5.1. Une fonction $f \in \mathcal{F}$ est *récursive primitive* si elle peut être obtenue à partir des fonctions initiales en appliquant un nombre fini de fois la composition et la récursion primitive. On note $\mathcal{PR}$ l'ensemble des fonctions récursives primitives.

Exemple 2.5.2. Les fonctions suivantes sont récursives primitives :

— les fonctions constantes de $\mathcal{F}_d$

— l'addition $\mathbb{N}^d \to \mathbb{N},\ (n_1, \dots, n_d) \mapsto \sum_{i=1}^d n_i$

— la multiplication $\mathbb{N}^d \to \mathbb{N},\ (n_1, \dots, n_d) \mapsto \prod_{i=1}^d n_i$

— la puissance $\mathbb{N}^2 \to \mathbb{N},\ (m, n) \mapsto m^n$

— la fonction prédécesseur

$$\mathbb{N} \to \mathbb{N},\ m \mapsto \begin{cases} m-1 & \text{si } m \geqslant 1 \\ 0 & \text{si } m = 0 \end{cases}$$

— la fonction monus

$$\dot{-}\colon \mathbb{N}^2 \to \mathbb{N},\ (m,n) \mapsto \begin{cases} m-n & \text{si } m \geqslant n \\ 0 & \text{si } m < n \end{cases}$$

— la fonction signe

$$\operatorname{sign}\colon \mathbb{N} \to \mathbb{N},\ m \mapsto \begin{cases} 1 & \text{si } m \geqslant 1 \\ 0 & \text{si } m = 0. \end{cases}$$

Pour tous $k \in \mathbb{N}$, on note $\underline{k}_d$ la fonction constante de $\mathcal{F}_d$ correspondant à k. Lorsque $d = 0$, on s'autorise à écrire simplement $\underline{k}$ au lieu de $\underline{k}_0$.

2.5.2 Fonctions récursives

Pour définir les fonctions récursives, on a besoin d'une troisième règle de formation de nouvelles fonctions à partir d'anciennes. Il s'agit de la minimisation.

Définition 2.5.3. Un ensemble $A \subseteq \mathbb{N}^{d+1}$ est dit *sûr* si pour tout $\mathbf{m} \in \mathbb{N}^d$, il existe $n \in \mathbb{N}$ tel que $(\mathbf{m}, n) \in A$. Si $A \subseteq \mathbb{N}^{d+1}$ est un ensemble sûr, alors la fonction obtenue par minimisation de A est la fonction

$$\mathbb{N}^d \to \mathbb{N},\ \mathbf{m} \mapsto \inf\{n \in \mathbb{N} : (\mathbf{m}, n) \in A\}.$$

On écrit $\mu_n((\mathbf{m}, n) \in A)$ pour désigner la valeur $\inf\{n \in \mathbb{N} : (\mathbf{m}, n) \in A\}$.

Définition 2.5.4. Une fonction $f \in \mathcal{F}$ est *récursive* si elle peut être obtenue à partir des fonctions initiales en appliquant un nombre fini de fois la composition, la récursion primitive et la minimisation d'ensembles sûrs. On note $\mathcal{R}$ l'ensemble des fonctions récursives.

Remarque 2.5.5. Une fonction $g \in \mathcal{F}_{d+1}$ est dite *sûre* si pour tout $\mathbf{m} \in \mathbb{N}^d$, il existe $n \in \mathbb{N}$ tel que $g(\mathbf{m}, n) = 0$. Si $g \in \mathcal{F}_{d+1}$ est une fonction sûre, alors la fonction obtenue par *minimisation* de g est la fonction

$$\mathbb{N}^d \to \mathbb{N},\ \mathbf{m} \mapsto \inf\{n \in \mathbb{N} : g(\mathbf{m}, n) = 0\}.$$

On écrit $\mu_n(g(\mathbf{m}, n) = 0)$ pour désigner la valeur $\inf\{n \in \mathbb{N} : g(\mathbf{m}, n) = 0\}$. Il est laissé en exercice de montrer que la classe des fonctions obtenues par minimisation de fonctions sûres coïncide avec la classe des fonctions obtenues par minimisation d'ensembles sûrs.

2.5.3 Les fonctions calculables et récursives coïncident

Dans cette section, notre but est de montrer que $\mathcal{R} = \mathcal{C}$, résultat attribué au mathématicien américain Stephen Kleene.

Proposition 2.5.6. *Les fonctions récursives sont calculables.*

Preuve. Les fonctions initiales sont calculables. En effet, la machine de Turing $\mathcal{R}$ calcule la fonction zéro, la machine de Turing $\mathtt{u}\mathcal{R}$ calcule la fonction σ et pour tout $i \in \{1, \ldots, d\}$, la machine de Turing $\mathcal{E}_1^{d-i}\mathcal{E}_2^{i-1}$ calcule la projection $P_{d,i}$.

Montrons maintenant que l'ensemble $\mathcal{C}$ est stable par composition, par récursion primitive et par minimisation de fonctions sûres. Par définition de l'ensemble $\mathcal{R}$, nous aurons alors obtenu l'inclusion $\mathcal{R} \subseteq \mathcal{C}$ comme souhaité.

Soient $k \in \mathbb{N}_0$, $h_1, \ldots, h_k \in \mathcal{F}_d$ des fonctions calculées par des machines de Turing $\mathcal{H}_1, \ldots, \mathcal{H}_k$ respectivement et soit $g \in \mathcal{F}_k$ une fonction calculée par une machine de Turing

$\mathcal{G}$. Alors la machine de Turing $\mathcal{C}_d^d\mathcal{H}_1\mathcal{C}_{d+1}^d\mathcal{H}_2\mathcal{C}_{d+2}^d\mathcal{H}_3\cdots\mathcal{C}_{d+k-1}^d\mathcal{H}_k\mathcal{G}\mathcal{E}_2^d$ calcule la fonction composée $g(h_1,\ldots,h_k)$. Ceci prouve la stabilité de $\mathcal{C}$ par composition.

Soient $g\in\mathcal{F}_d$ une fonction calculée par une machine de Turing $\mathcal{G}$ et $h\in\mathcal{F}_{d+2}$ fonction calculée par une machine de Turing $\mathcal{H}$. L'algorithme 1 donné ci-après calcule la fonction $f\in\mathcal{F}^{d+1}$ définie par récursion primitive à partir de g et h. Montrons qu'après k passages dans

Algorithm 1 Calcul de la récursion primitive.

Require: $(\mathbf{m},n)\in\mathbb{N}^{d+1}$
Ensure: À la sortie, r vaut $f(\mathbf{m},n)$.
  $p\leftarrow n,\ q\leftarrow 0,\ r\leftarrow g(\mathbf{m})$
  while $p>0$ **do**
    $p\leftarrow p-1$
    $r\leftarrow h(\mathbf{m},q,r)$
    $q\leftarrow q+1$
  end while
  return r

la boucle, la variable q vaut k et la variable r vaut $f(\mathbf{m},k)$. On procède par récurrence sur k. Si $k=0$, on ne passe pas dans la boucle. Les variables k et r sont initialisées à 0 et $g(\mathbf{m})$ respectivement. Puisque $g(\mathbf{m})=f(\mathbf{m},0)$, le cas de base est réglé. Supposons qu'après k passages dans la boucle, la variable q vaut k et la variable r vaut $f(\mathbf{m},k)$. Après un passage supplémentaire, la variable r est actualisée à $h(\mathbf{m},q,r)=h(\mathbf{m},k,f(\mathbf{m},k))=f(\mathbf{m},k+1)$ et la variable q est actualisée à $q+1=k+1$. Ensuite, nous construisons une machine de Turing qui calcule la fonction f en suivant les étapes de cet algorithme. Une telle machine est décrite par l'organigramme de la figure 2.14. Nous laissons au lecteur le soin de vérifier les détails. Ceci prouve la stabilité de $\mathcal{C}$ par récursion primitive.

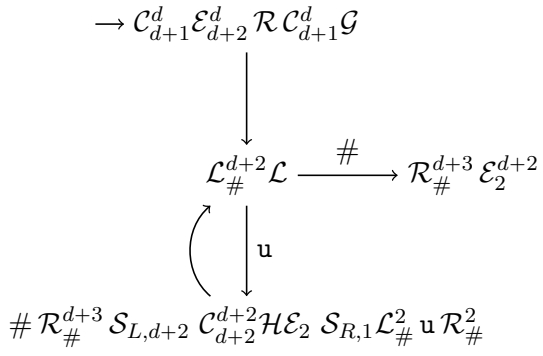


FIGURE 2.14 – Organigramme pour la récursion primitive.

Passons enfin à la stabilité de $\mathcal{C}$ par minimisation d'ensembles sûrs. Soit $A\subset\mathcal{F}^{d+1}$ un ensemble sûr et soit $\mathcal{A}$ une machine de Turing calculant χ_A. L'algorithme 2 donné ci-après calcule la fonction obtenue par minimisation de A et la figure 2.15 est un organigramme de machines de Turing basé sur cet algorithme. Les détails sont laissés aux soins du lecteur. □

Afin de montrer la réciproque du résultat précédent, nous avons besoin d'une série de notions et résultats préparatoires.

Définition 2.5.7. Un *prédicat d'arité* $d\geqslant 1$ est une partie de $\mathbb{N}^d$. Un prédicat P d'arité d

Algorithm 2 Calcul de la minimisation.

Require: $\mathbf{m} \in \mathbb{N}^d$
Ensure: À la sortie, q vaut $\mu_n((\mathbf{m}, n) \in A)$.
  $q \leftarrow 0,\ r \leftarrow \chi_A(\mathbf{m}, 0)$
  while $r = 0$ **do**
    $q \leftarrow q + 1$
    $r \leftarrow \chi_A(\mathbf{m}, q)$
  end while
  return q

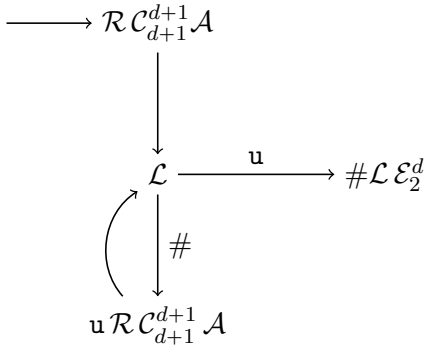


FIGURE 2.15 – Organigramme pour la minimisation.

est dit *récursif primitif* si sa fonction caractéristique

$$\chi_P \colon \mathbb{N}^d \to \mathbb{N},\ \mathbf{m} \mapsto \begin{cases} 1 & \text{si } \mathbf{m} \in P \\ 0 & \text{si } \mathbf{m} \notin P \end{cases}$$

est récursive primitive.

L'ensemble des prédicats récursifs primitifs est stable pour les opérations booléennes.

Proposition 2.5.8. *Soient P et Q des prédicats récursifs primitifs de même arité d. Alors les prédicats $P \cap Q$, $P \cup Q$, $\mathbb{N}^d \setminus P$ sont récursifs primitifs.*

Preuve. On a $\chi_{P\cap Q} = \chi_P \cdot \chi_Q$, $\chi_{P\cup Q} = \operatorname{sign} \circ (\chi_P + \chi_Q)$ et $\chi_{\mathbb{N}^d\setminus P} = \underline{1}_1 \dot{-} \chi_P$. D'où la conclusion. □

La proposition suivante nous fournit quelques premiers prédicats récursifs primitifs.

Proposition 2.5.9. *Pour tout $\sim \in \{\leqslant, <, =, \geqslant, >\}$, le prédicat binaire $P_\sim = \{(m, n) \in \mathbb{N}^2 : m \sim n\}$ est récursif primitif.*

Preuve. On a $\chi_{P_>} = \operatorname{sign} \circ (P_{2,1} \dot{-} P_{2,2})$ et $\chi_{P_<} = \operatorname{sign} \circ (P_{2,2} \dot{-} P_{2,1})$. D'où $P_>$ et $P_<$ sont récursifs primitifs. Ensuite, on a $P_\leqslant = \mathbb{N}^2 \setminus P_>$, $P_\geqslant = \mathbb{N}^2 \setminus P_<$ et $P_= = P_\leqslant \cap P_\geqslant$. Grâce à la proposition 2.5.8, on obtient que $P_\leqslant$, $P_\geqslant$ et $P_=$ sont récursifs primitifs également. □

Proposition 2.5.10. *Soit P un prédicat récursif primitif d'arité k et soient $f_1, \ldots, f_k$ des fonctions récursives primitives de $\mathcal{F}_d$. Alors le prédicat*

$$\{\mathbf{m} \in \mathbb{N}^d : (f_1(\mathbf{m}), \ldots, f_k(\mathbf{m})) \in P\}$$

est récursif primitif.

Preuve. Il suffit de noter que la fonction caractéristique de ce prédicat est égale à la fonction composée $\chi_P(f_1, \ldots, f_k)$. □

Corollaire 2.5.11. *Soient $f, g \in \mathcal{F}_d \cap \mathcal{PR}$. Pour tout $\sim \in \{<, \leqslant, =, \geqslant, >\}$, le prédicat $\{\mathbf{m} \in \mathbb{N}^d : f(\mathbf{m}) \sim g(\mathbf{m})\}$ est récursif primitif. En particulier, les prédicats $\{\mathbf{m} \in \mathbb{N}^d : f(\mathbf{m}) > 0\}$ et $\{\mathbf{m} \in \mathbb{N}^d : f(\mathbf{m}) = 0\}$ sont aussi récursifs primitifs.*

Preuve. C'est une conséquence des propositions 2.5.9 et 2.5.10. □

Proposition 2.5.12 (Définition par cas)**.** *Soient $P_1, \ldots, P_k$ des prédicats récursifs primitifs d'arité d deux à deux disjoints et soient $f_1, \ldots, f_{k+1}$ des fonctions récursives primitives de $\mathcal{F}_d$. Alors la fonction*

$$\mathbb{N}^d \to \mathbb{N},\ \mathbf{m} \mapsto \begin{cases} f_1(\mathbf{m}) & \text{si } \mathbf{m} \in P_1 \\ \quad \vdots & \\ f_k(\mathbf{m}) & \text{si } \mathbf{m} \in P_k \\ f_{k+1}(\mathbf{m}) & \text{sinon.} \end{cases}$$

est récursive primitive.

Preuve. Cette fonction définie par cas est égale à

$$f_1 \cdot \chi_{P_1} + \cdots + f_k \cdot \chi_{P_k} + f_{k+1} \cdot \chi_{\mathbb{N}^d \setminus (P_1 \cup \cdots \cup P_k)}.$$

On conclut en utilisant la proposition 2.5.8. □

Corollaire 2.5.13.

- *Si on modifie un nombre fini de valeurs d'une fonction récursive primitive, on obtient encore une fonction récursive primitive.*
- *Tout sous-ensemble fini de $\mathbb{N}^d$ est un prédicat récursif primitif.*

Proposition 2.5.14. *Le produit cartésien de prédicats récursifs primitifs est récursif primitif.*

Preuve. Soient $P \subseteq \mathbb{N}^p$ et $Q \subseteq \mathbb{N}^q$ deux prédicats récursifs primitifs. On a $\chi_{P \times Q} = \chi_P(P_{p+q,1}, \ldots, P_{p+q,p}) \cdot \chi_Q(P_{p+q,p+1}, \ldots, P_{p+q,p+q})$. □

Proposition 2.5.15 (Quantification bornée)**.** *Si P est un prédicat récursif primitif d'arité $d+1$, alors les prédicats*

$$\{(\mathbf{m}, n) \in \mathbb{N}^{d+1} : \forall i \leqslant n,\ (\mathbf{m}, i) \in P\}$$

et

$$\{(\mathbf{m}, n) \in \mathbb{N}^{d+1} : \exists i \leqslant n,\ (\mathbf{m}, i) \in P\}$$

sont récursifs primitifs.

Preuve. Notons A le premier prédicat et B le second. Les fonctions caractéristiques χ_A et χ_B s'obtiennent par récursion primitive de fonctions récursives primitives. En effet, pour tous $\mathbf{m} \in \mathbb{N}^d$ et $n \in \mathbb{N}$, on a

$$\chi_A(\mathbf{m}, 0) = \chi_B(\mathbf{m}, 0) = \chi_P(P_{d,1}, \ldots, P_{d,d}, \underline{0}_d)(\mathbf{m}),$$

$$\chi_A(\mathbf{m}, n+1) = \chi_A(\mathbf{m}, n) \cdot \chi_P(\mathbf{m}, n+1)$$

$$= (P_{d+2,d+2} \cdot \chi_P(P_{d+2,1}, \dots, P_{d+2,d}, \sigma \circ P_{d+2,d+1}))(\mathbf{m}, n, \chi_A(\mathbf{m}, n))$$

et

$$\begin{aligned}\chi_B(\mathbf{m}, n+1) &= \operatorname{sign}(\chi_B(\mathbf{m}, n) + \chi_P(\mathbf{m}, n+1)) \\ &= \operatorname{sign}(P_{d+2,d+2} + \chi_P(P_{d+2,1}, \dots, P_{d+2,d}, \sigma \circ P_{d+2,d+1}))(\mathbf{m}, n, \chi_B(\mathbf{m}, n)).\end{aligned}$$

□

Définition 2.5.16. La fonction obtenue par *minimisation bornée d'un ensemble* $A \subseteq \mathbb{N}^{d+1}$ est la fonction

$$\mathbb{N}^{d+1} \to \mathbb{N},\ (\mathbf{m}, n) \mapsto \begin{cases} \inf\{t \leqslant n : (\mathbf{m}, t) \in A\} & \text{si } \exists t \leqslant n,\ (\mathbf{m}, t) \in A \\ 0 & \text{sinon.} \end{cases}$$

On écrit $\mu_{t \leqslant n}((\mathbf{m}, t) \in A)$ pour désigner la valeur de cette fonction en $(\mathbf{m}, n)$.

Remarque. La fonction obtenue par *minimisation bornée d'une fonction* $g \in \mathcal{F}_{d+1}$ est la fonction

$$\mathbb{N}^{d+1} \to \mathbb{N}, (\mathbf{m}, n) \mapsto \begin{cases} \inf\{t \leqslant n : g(\mathbf{m}, t) = 0\} & \text{si } \exists t \leqslant n,\ g(\mathbf{m}, t) = 0 \\ 0 & \text{sinon.} \end{cases}$$

On note $\mu_{t \leqslant n}(g(\mathbf{m}, t) = 0)$ la valeur de cette fonction en $(\mathbf{m}, n)$. Il est laissé en exercice de montrer que la classe des fonctions obtenues par minimisation bornée de fonctions coïncide avec la classe des fonctions obtenues par minimisation bornée d'ensembles.

Proposition 2.5.17 (Minimisation bornée)**.**

- *Une fonction obtenue par minimisation bornée d'une fonction récursive primitive est récursive primitive.*
- *Une fonction obtenue par minimisation bornée d'un prédicat récursif primitif est récursive primitive.*

Preuve. Nous montrons le deuxième item, le premier se montrant de façon similaire. Soit A un prédicat récursif primitif d'arité $d+1$. Pour tous $\mathbf{m} \in \mathbb{N}^d$ et $n \in \mathbb{N}$, on a

$$\mu_{t \leqslant 0}((\mathbf{m}, t) \in A) = \underline{0}_d(\mathbf{m})$$

et

$$\begin{aligned}\mu_{t \leqslant n+1}((\mathbf{m}, t) \in A) &= \begin{cases} \mu_{t \leqslant n}((\mathbf{m}, t) \in A) & \text{si } \exists t \leqslant n, (\mathbf{m}, t) \in A \\ 0 & \text{si } \forall t \leqslant n+1, (\mathbf{m}, t) \notin A \\ n+1 & \text{sinon} \end{cases} \\ &= \begin{cases} P_{d+2,d+2}(\mathbf{m}, n, \mu_{t \leqslant n}((\mathbf{m}, t) \in A)) & \text{si } (\mathbf{m}, n, \mu_{t \leqslant n}((\mathbf{m}, t) \in A)) \in B_1 \\ \underline{0}_{d+2}(\mathbf{m}, n, \mu_{t \leqslant n}((\mathbf{m}, t) \in A)) & \text{si } (\mathbf{m}, n, \mu_{t \leqslant n}((\mathbf{m}, t) \in A)) \in B_2 \\ \sigma \circ P_{d+2,d+1}(\mathbf{m}, n, \mu_{t \leqslant n}((\mathbf{m}, t) \in A)) & \text{sinon} \end{cases}\end{aligned}$$

où

$$B_1 = \{(\mathbf{m}, n, s) \in \mathbb{N}^{d+2} : \exists t \leqslant n, (\mathbf{m}, t) \in A\}$$

et

$$B_2 = \{(\mathbf{m}, n, s) \in \mathbb{N}^{d+2} : \forall t \leqslant n+1, (\mathbf{m}, t) \in \mathbb{N}^{d+1} \setminus A\}.$$

Par les propositions 2.5.8 et 2.5.14, les prédicats

$$C_1 = \{(\mathbf{m}, n) \in \mathbb{N}^{d+1} : \exists t \leqslant n, (\mathbf{m}, t) \in A\}$$

et

$$C_2 = \{(\mathbf{m}, n) \in \mathbb{N}^{d+1} : \forall t \leqslant n, (\mathbf{m}, t) \in \mathbb{N}^{d+1} \setminus A\}$$

sont récursifs primitifs. Comme

$$\chi_{B_1} = \chi_{C_1}(P_{d+2,1}, \ldots, P_{d+2,d+1})$$

et

$$\chi_{B_2} = \chi_{C_2}(P_{d+2,1}, \ldots, P_{d+2,d}, \sigma \circ P_{d+2,d+1}),$$

on obtient que B_1 et B_2 sont récursifs primitifs. Les fonctions $P_{d+2,d+2}$, $\underline{0}_{d+2}$ et $\sigma \circ P_{d+2,d+1}$ étant également récursives primitives, on conclut en utilisant la proposition 2.5.12. □

Nous donnons à présent quelques nouveaux exemples de fonctions récursives primitives qui seront utiles dans la preuve du théorème 2.5.22.

Lemme 2.5.18.

— *La fonction*

$$\mathrm{DIV}\colon \mathbb{N}^2 \to \mathbb{N},\ (m,n) \mapsto \begin{cases} \lfloor \frac{m}{n} \rfloor & \text{si } n \geqslant 1 \\ 0 & \text{si } n = 0 \end{cases}$$

est récursive primitive.

— *La fonction*

$$\mathrm{MOD}\colon \mathbb{N}^2 \to \mathbb{N},\ (m,n) \mapsto \begin{cases} m \bmod n & \text{si } n \geqslant 1 \\ m & \text{si } n = 0 \end{cases}$$

est récursive primitive.

— *La fonction*

$$\mathrm{LOGD}\colon \mathbb{N}^2 \to \mathbb{N},\ (m,n) \mapsto \begin{cases} \lfloor \log_m(n) \rfloor & \text{si } m \geqslant 2 \text{ et } n \geqslant 1 \\ 0 & \text{sinon} \end{cases}$$

est récursive primitive.

— *Le prédicat $D = \{(m,n) \in \mathbb{N}^2 : n \text{ divise } m\}$ est récursif primitif.*

Preuve. Le prédicat $P = \{(m,n,s) \in \mathbb{N}^3 : (s+1)n > m\}$ est récursif primitif par les propositions 2.5.9 et 2.5.10. Ainsi, au vu de la proposition 2.5.17, la fonction $f\colon \mathbb{N}^3 \mapsto \mathbb{N},\ (m,n,s) \mapsto \mu_{t \leqslant s}((m,n,t) \in P)$ est récursive primitive. Puisque

$$\mathrm{DIV}\colon \mathbb{N}^2 \to \mathbb{N},\ (m,n) \mapsto \begin{cases} f(m,n,m) & \text{si } n \geqslant 1 \\ 0 & \text{si } n = 0, \end{cases}$$

la fonction DIV est récursive primitive par la proposition 2.5.12. La fonction MOD est alors elle aussi récursive primitive puisque pour tout $(m,n) \in \mathbb{N}$, on a $\mathrm{MOD}(m,n) = m \dot{-} n \cdot \mathrm{DIV}(m,n)$.

Montrons maintenant que la fonction LOGD est récursive primitive. Le prédicat $Q = \{(m,n,s) \in \mathbb{N}^3 : m^{s+1} > n\}$ est récursif primitif par les propositions 2.5.9 et 2.5.10. Ainsi, au vu de la proposition 2.5.17, la fonction $g\colon \mathbb{N}^3 \mapsto \mathbb{N},\ (m,n,s) \mapsto \mu_{t \leqslant s}((m,n,t) \in Q)$ est récursive primitive. Puisque

$$\mathrm{LOGD}\colon \mathbb{N}^2 \to \mathbb{N},\ (m,n) \mapsto \begin{cases} g(m,n,n) & \text{si } m \geqslant 2 \text{ et } n \geqslant 1 \\ 0 & \text{sinon,} \end{cases}$$

la fonction LOGD est récursive primitive par la proposition 2.5.12.

Enfin, D est récursif primitif car

$$\chi_D\colon \mathbb{N}^2 \to \mathbb{N},\ (m,n) \mapsto \begin{cases} 1 & \text{si MOD}(m,n) = 0 \text{ et } n \neq 0 \\ 0 & \text{sinon.} \end{cases}$$

□

Voici le dernier résultat préparatoire à la preuve de $\mathcal{C} \subseteq \mathcal{R}$. Il s'agit de donner une nouvelle règle de construction de fonctions généralisant la récursion primitive. Nous avons d'abord besoin de deux lemmes. Leur but est de montrer qu'on peut énumérer les éléments de $\mathbb{N}^d$ de manière récursive primitive. Nous commençons par le cas de $\mathbb{N}^2$.

Lemme 2.5.19. *La fonction $E_2\colon \mathbb{N}^2 \to \mathbb{N}, (m,n) \mapsto 2^m(2n+1) - 1$ est une bijection récursive primitive. De plus, il existe des fonctions $\beta_1, \beta_2 \in \mathcal{F}_1 \cap \mathcal{PR}$ telles que $\beta_1 \circ E_2 = P_{2,1}$ et $\beta_2 \circ E_2 = P_{2,2}$.*

Preuve. Il est facile de vérifier que E_2 est une bijection récursive primitive. La fonction

$$\beta_1\colon \mathbb{N} \to \mathbb{N},\ k \mapsto \inf\{t \in \mathbb{N} : 2^{t+1} \text{ ne divise pas } k+1\}$$

est telle que $\beta_1 \circ E_2 = P_{2,1}$. Par le lemme 2.5.18, le prédicat D est récursif primitif et donc $\mathbb{N}^2 \setminus D$ aussi. Ainsi, la fonction $f\colon \mathbb{N}^2 \to \mathbb{N},\ (k,s) \mapsto \mu_{t \leqslant s}((k+1, 2^{t+1}) \in \mathbb{N}^2 \setminus D)$ est récursive primitive. Puisque $\beta_1(k) = f(k,k)$, on obtient que la fonction β_1 est récursive primitive. La fonction

$$\beta_2\colon \mathbb{N} \to \mathbb{N},\ k \mapsto \frac{1}{2}\left(\frac{k+1}{2^{\beta_1(k)}} - 1\right)$$

est telle que $\beta_2 \circ E_2 = P_{2,2}$. Puisque la fonction β_1 est récursive primitive, la fonction β_2 est elle aussi récursive primitive. □

Lemme 2.5.20. *Pour tout $d \geqslant 1$, il existe une bijection $E_d \in \mathcal{F}_d \cap \mathcal{PR}$ et des fonctions récursives primitives $\beta_{d,1}, \ldots, \beta_{d,d} \in \mathcal{F}_1 \cap \mathcal{PR}$ telles que pour tout $i \in \{1, \ldots, d\}$, on ait $\beta_{d,i} \circ E_d = P_{d,i}$.*

Preuve. Nous procédons par récurrence sur d. Le résultat est trivialement vrai pour $d = 1$ et le cas $d = 2$ est réglé par le lemme 2.5.19. Soit à présent $d \geqslant 2$ et supposons que de telles fonctions $E_d, \beta_{d,1}, \ldots, \beta_{d,2}$ existent. Alors les fonctions

$$\begin{aligned} E_{d+1} &= E_2(E_d(P_{d+1,1}, \ldots, P_{d+1,d}), P_{d+1,d+1}) \\ \beta_{d+1,i} &= \beta_{d,i} \circ \beta_{2,1} \quad \text{pour } 1 \leqslant i \leqslant d \\ \beta_{d+1,d+1} &= \beta_{2,2} \end{aligned}$$

conviennent pour la thèse. □

Remarquons que les fonctions obtenues dans les lemmes 2.5.19 et 2.5.20 ne sont pas uniques. Par exemple, la fonction de Peano $\mathbb{N}^2 \to \mathbb{N},\ (m,n) \mapsto \frac{(m+n)(m+n+1)}{2} + n$ est aussi une bijection récursive primitive. [2]

Proposition 2.5.21 (Récursion primitive généralisée)**.** *Soient $g_1, \ldots, g_k \in \mathcal{F}_d$ et $h_1, \ldots, h_k \in \mathcal{F}_{d+k+1}$ des fonctions récursives primitives. Alors les fonctions $f_1, \ldots, f_k \in \mathcal{F}_{d+1}$ définie comme suit sont récursives primitives : pour tous $\mathbf{m} \in \mathbb{N}^d$, $n \in \mathbb{N}$ et $i \in \{1, \ldots, k\}$, on a*

$$f_i(\mathbf{m}, 0) = g_i(\mathbf{m}) \quad \textit{et} \quad f_i(\mathbf{m}, n+1) = h_i(\mathbf{m}, n, f_1(\mathbf{m}, n), \ldots, f_k(\mathbf{m}, n)).$$

2. Que deviennent les fonctions β_1 et β_2 dans ce cas ?

Preuve. Soient $E_k, \beta_{k,1}, \ldots, \beta_{k,k}$ des fonctions comme dans le lemme 2.5.20. Posons $F = E_k(f_1, \ldots, f_k)$. Puisque $f_i = \beta_{k,i} \circ F$ pour tout $i \in \{1, \ldots, k\}$, il suffit de montrer que la fonction F est récursive primitive. Pour tous $\mathbf{m} \in \mathbb{N}^d$ et $n \in \mathbb{N}$, on a $F(\mathbf{m}, 0) = E_k(g_1, \ldots, g_k)(\mathbf{m})$ et $F(\mathbf{m}, n+1) = h(\mathbf{m}, n, F(\mathbf{m}, n))$ avec

$$h = E_k(\ldots, h_i(P_{d+2,1}, \ldots, P_{d+2,d+1}, \beta_{k,1} \circ P_{d+2,d+2}, \ldots, \beta_{k,k} \circ P_{d+2,d+2}), \ldots).$$

D'où la conclusion. □

Exercice. Montrer que les nombres de Fibonacci sont récursifs primitifs en utilisant la proposition 2.5.21. Autrement dit, montrer que la fonction $F\colon \mathbb{N} \to \mathbb{N}$ définie récursivement par $F(0) = 0$, $F(1) = 1$ et $F(n+2) = F(n+1) + F(n)$ pour tout entier $n \geqslant 0$ est récursive primitive.

Nous sommes enfin prêts pour démontrer que toute fonction calculable par machine de Turing est récursive. Nous montrons même un résultat plus précis : toute fonction calculable par machine de Turing peut s'obtenir en appliquant aux fonctions initiales la composition, la récursion primitive et une unique fois la minimisation d'une fonction sûre.

Théorème 2.5.22. *Pour toute fonction $f \in \mathcal{F}_d \cap \mathcal{C}$, il existe une fonction $g \in \mathcal{F}_{d+1}\,\mathcal{PR}$ et une fonction sûre $h \in \mathcal{F}_{d+1}\,\mathcal{PR}$ telles que $f(\mathbf{m}) = g(\mathbf{m}, \mu_n(h(\mathbf{m}, n) = 0))$ pour tout $\mathbf{m} \in \mathbb{N}^d$. En particulier, les fonctions calculables sont récursives.*

Preuve. Soit $f \in \mathcal{F}_d \cap \mathcal{C}$. Comme $\mathcal{F}_0 \subseteq \mathcal{R}$, on peut supposer que $d \geqslant 1$. Soit $\mathcal{M} = (Q, q_0, h, A, \delta)$ une machine de Turing calculant f.

L'idée générale de la preuve est d'encoder le comportement de la machine de Turing $\mathcal{M}$ à l'aide de fonctions récursives. En fait, nous n'utiliserons partout que des fonctions récursives primitives, à l'exception de la toute dernière étape où une minimisation (non bornée) interviendra.

Nous encodons successivement les configurations machines, les transitions et enfin la fonction f elle-même.

1. Codage de $\mathcal{M}$.

   (a) Codage des mots de A^*.

   Notons $A = \{a_1, \ldots, a_k\}$ avec $k \geqslant 2$. Sans perte de généralité, on peut supposer que $a_1 = \#$ et $a_2 = \mathtt{u}$. On définit un codage

   $$c\colon A^* \to \mathbb{N},\ a_{i_{n-1}} \cdots a_{i_0} \mapsto \sum_{j=0}^{n-1} i_j (k+1)^j.$$

   Il s'agit de la fonction valeur en base $k+1$. Puisque le chiffre 0 n'est pas utilisé, la fonction c est injective. En particulier, on a $c(\varepsilon) = 0$ et $c(a_i) = i$ pour tout $i \in \{1, \ldots, k\}$.

   (b) Codage des états.

   Notons $Q = \{p_0, \ldots, p_\ell\}$ avec $\ell \geqslant 1$. Sans perte de généralité, on peut supposer que $p_0 = h$ et $p_1 = q_0$. On définit un codage

   $$c\colon Q \to \mathbb{N},\ p_i \mapsto i.$$

   (c) Codage de la fonction de transition.

   On pose $a_{k+1} = \mathtt{L}$ et $a_{k+2} = \mathtt{R}$. Considérons des fonctions $D_1, D_2 \in \mathcal{F}_2 \cap \mathcal{PR}$ telles que pour tous $i \in \{1, \ldots, \ell\}$ et $j \in \{1, \ldots, k\}$ tels que $\delta(p_i, a_j)$ est défini, on ait

   $$\delta(p_i, a_j) = (p_{D_1(i,j)}, a_{D_2(i,j)}).$$

Remarquons que seulement un nombre fini de valeurs de D_1 et D_2 sont fixées par ces conditions.

(d) Codage des configurations machine.
Une configuration machine $p.x\underline{a}y$ est codée par le quadruplet de $\mathbb{N}^4$

$$(c(q), c(x), c(a), c(y^R)).$$

2. Codage des transitions opérant sur les configurations machines.

(a) On cherche une fonction $F\colon \mathbb{N}^4 \to \mathbb{N}^4$ telle que si $\mathbf{c} = (c_1, c_2, c_3, c_4)$ est le code d'une configuration machine à partir de laquelle une nouvelle configuration machine est atteignable en une transition, alors $F(\mathbf{c})$ est le code de cette nouvelle configuration machine. Pour ce faire, nous allons définir des fonctions $F_1, F_2, F_3, F_4 \in \mathcal{F}_4 \cap \mathcal{PR}$ telles que pour tout $\mathbf{c} \in \mathbb{N}^4$, on ait

$$F(\mathbf{c}) = (F_1(\mathbf{c}), F_2(\mathbf{c}), F_3(\mathbf{c}), F_4(\mathbf{c})).$$

La table 2.2 indique les cas où les définitions de F_1, F_2, F_3, F_4 sont précisées. On

	$1 \leqslant D_2(c_1,c_3) \leqslant k$	$D_2(c_1,c_3) = k{+}1$	$D_2(c_1,c_3) = k{+}2$
$F_1(\mathbf{c})$	$D_1(c_1,c_3)$	$D_1(c_1,c_3)$	$D_1(c_1,c_3)$
$F_2(\mathbf{c})$	c_2	$\mathrm{DIV}(c_2,k{+}1)$	$(k{+}1)c_2{+}c_3$
$F_3(\mathbf{c})$	$D_2(c_1,c_3)$	$\mathrm{MOD}(c_2,k{+}1)$	$\begin{cases}\mathrm{MOD}(c_4,k{+}1) & \text{si } c_4 \neq 0\\ 1 & \text{si } c_4 = 0\end{cases}$
$F_4(\mathbf{c})$	c_4	$\begin{cases}(k{+}1)c_4{+}c_3 & \text{si } c_3 \neq 1 \text{ ou } c_4 \neq 0\\ 0 & \text{sinon}\end{cases}$	$\mathrm{DIV}(c_4,k{+}1)$

TABLE 2.2 – Définition des fonctions F_1, F_2, F_3, F_4.

prolonge ensuite les définitions de ces fonctions à $\mathbb{N}^4$ par des fonctions récursives primitives quelconques (par exemple, la fonction constante $\underline{0}_4$). En utilisant la proposition 2.5.12, on obtient que $F_1, F_2, F_3, F_4 \in \mathcal{PR}$.

(b) On cherche une fonction $F^*\colon \mathbb{N}^5 \to \mathbb{N}^4$ telle que si $\mathbf{c}$ est le code d'une configuration machine à partir de laquelle une nouvelle configuration machine est atteignable en n transitions, alors $F^*(\mathbf{c}, n)$ est le code de cette nouvelle configuration machine. Pour $i \in \{1, 2, 3, 4\}$, on définit $F_i^* \in \mathcal{F}_5$ comme suit : pour tout $\mathbf{c} = (c_1, c_2, c_3, c_4) \in \mathbb{N}^4$ et tout $n \in \mathbb{N}$, on a

$$F_i^*(\mathbf{c}, 0) = c_i$$

et

$$F_i^*(\mathbf{c}, n+1) = F_i(F_1^*(\mathbf{c}, n), F_2^*(\mathbf{c}, n), F_3^*(\mathbf{c}, n), F_4^*(\mathbf{c}, n)).$$

Les fonctions $F_1^*, F_2^*, F_3^*, F_4^*$ sont récursives primitives par la proposition 2.5.21. La fonction

$$F^*\colon \mathbb{N}^5 \to \mathbb{N}^4,\ (\mathbf{c}, n) \mapsto (F_1^*(\mathbf{c}, n), F_2^*(\mathbf{c}, n), F_3^*(\mathbf{c}, n), F_4^*(\mathbf{c}, n))$$

convient.

3. Codage des données et décodage du résultat.
Soit

$$\mathrm{cod}\colon \mathbb{N}^d \to \mathbb{N},\ (m_1, \ldots, m_d) \mapsto c(\#\,\mathtt{u}^{m_1}\,\#\cdots\#\,\mathtt{u}^{m_d})$$

et soit

$$\mathrm{decod}\colon \mathbb{N} \to \mathbb{N},\ n \mapsto \mathrm{LOGD}(k+1, n).$$

La fonction decod est récursive primitive et pour tout $n \in \mathbb{N}$, on a $\mathrm{decod}(c(\#\,\mathtt{u}^m)) = m$. Montrons que la fonctions cod est aussi récursive primitive. Nous montrons ceci par récurrence sur d et nous notons $\mathrm{cod} = \mathrm{cod}_d$ pour cette preuve. Pour $d = 1$, on a

$$\begin{aligned}\mathrm{cod}_1(m) &= c(\#\,\mathtt{u}^m)\\ &= (k+1)^m + 2((k+1)^{m-1} + \cdots + (k+1) + 1)\\ &= (k+1)^m + 2\frac{((k+1)^m - 1)}{k}\\ &= (k+1)^m + 2\,\mathrm{DIV}((k+1)^m - 1), k).\end{aligned}$$

Il s'agit d'une composition de fonctions récursives primitives, donc $\mathrm{cod}_1 \in \mathcal{PR}$. Supposons maintenant que $d \geqslant 1$ et que $\mathrm{cod}_d \in \mathcal{PR}$. On a

$$\begin{aligned}\mathrm{cod}_{d+1}(m_1, \ldots, m_{d+1}) &= c(\#\,\mathtt{u}^{m_1} \#\cdots\#\,\mathtt{u}^{m_d} \#\,\mathtt{u}^{m_{d+1}})\\ &= c(\#\,\mathtt{u}^{m_1} \#\cdots\#\,\mathtt{u}^{m_d})(k+1)^{m_{d+1}+1} + c(\#\,\mathtt{u}^{m_{d+1}})\\ &= \mathrm{cod}_d(m_1, \ldots, m_d)(k+1)^{m_{d+1}+1} + \mathrm{cod}_1(m_{d+1}).\end{aligned}$$

Puisque $\mathrm{cod}_d \in \mathcal{PR}$ par hypothèse de récurrence et que $\mathrm{cod}_1 \in \mathcal{PR}$, on obtient que $\mathrm{cod}_{d+1} \in \mathcal{PR}$.

4. Expression de f à l'aide des codages de $\mathcal{M}$ et des transitions.
   La configuration machine initiale est codée par $(1, \mathrm{cod}(\mathbf{m}), 1, 0)$ et la configuration d'arrêt est codée par $(0, c(\#\,\mathtt{u}^{f(\mathbf{m})}), 1, 0)$. Par hypothèse, il existe $n \in \mathbb{N}$ tel que $f(\mathbf{m}) = \mathrm{decod}(c(\#\,\mathtt{u}^{f(\mathbf{m})})) = \mathrm{decod}(F_2^*(1, \mathrm{cod}(\mathbf{m}), 1, 0, n))$. Cet entier n est donné par $\mu_t(F_1^*(1, \mathrm{cod}(\mathbf{m}), 1, 0, t) = 0)$. D'où les fonctions récursives primitives

$$g\colon\ \mathbb{N}^{d+1} \to \mathbb{N},\ (\mathbf{m}, n) \mapsto \mathrm{decod}(F_2^*(1, \mathrm{cod}(\mathbf{m}), 1, 0, n))$$

et

$$h\colon\ \mathbb{N}^{d+1} \to \mathbb{N},\ (\mathbf{m}, n) \mapsto F_1^*(1, \mathrm{cod}(\mathbf{m}), 1, 0, n))$$

conviennent pour la thèse.

□

2.6 La fonction d'Ackermann

Le but de cette section est de montrer qu'on a $\mathcal{PR} \subsetneq \mathcal{R}$. Clairement, on a $\mathcal{PR} \subseteq \mathcal{R}$ par définition de ces deux familles de fonctions. Puisque nous savons déjà que $\mathcal{R} = \mathcal{C}$, la question revient donc à montrer qu'il existe une fonction calculable mais non récursive primitive. Nous procédons de la manière la plus directe qui soit en exhibant explicitement une telle fonction.

Définition 2.6.1. La *fonction d'Ackermann*[3] est la fonction $\mathcal{A}\colon\ \mathbb{N}^2 \to \mathbb{N}$ définie par les conditions suivantes : pour tous $m, n \in \mathbb{N}$,

1. $\mathcal{A}(0, n) = n + 1$
2. $\mathcal{A}(m + 1, 0) = \mathcal{A}(m, 1)$
3. $\mathcal{A}(m + 1, n + 1) = \mathcal{A}(m, \mathcal{A}(m + 1, n))$.

Exercice. Calculer $\mathcal{A}(2, 3)$.

3. Ackermann avait d'abord défini une fonction plus complexe. Cette forme simplifiée à deux arguments a été proposée par Rózsa Péter, une mathématicienne hongroise. C'est pourquoi on parle aussi de la fonction d'Ackermann-Péter

Définition 2.6.2. Pour tout $m \in \mathbb{N}$, posons $\mathcal{A}_m \colon \mathbb{N} \to \mathbb{N},\ n \mapsto \mathcal{A}(m,n)$.

Proposition 2.6.3. *Pour tout $n \in \mathbb{N}$, on a*

1. $\mathcal{A}_0(n) = n+1$
2. $\mathcal{A}_1(n) = n+2$
3. $\mathcal{A}_2(n) = 2n+3$
4. $\mathcal{A}_3(n) = 2^{n+3} - 3$.

Preuve. Il suffit de procéder par récurrence sur n. □

Exercice. Trouver une expression de la fonction $\mathcal{A}_4$. Essayer de vous imaginer la vitesse à laquelle croît $\mathcal{A}_5$.

Au vu de la proposition précédente, on remarque que les fonctions $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2$ et $\mathcal{A}_3$ sont récursives primitives. C'est en fait le cas de toutes les fonctions $\mathcal{A}_m$.

Proposition 2.6.4. *Pour tout $m \in \mathbb{N}$, la fonction $\mathcal{A}_m$ est récursive primitive.*

Preuve. On procède par récurrence sur m. Pour $m = 0$, on a $\mathcal{A}_0 = \sigma$, qui est une fonction initiale. Soit à présent $m \in \mathbb{N}$ tel que $\mathcal{A}_m \in \mathcal{PR}$. On a $\mathcal{A}_{m+1}(0) = \mathcal{A}_m(1)$ et pour tout $n \in \mathbb{N}$, on a $\mathcal{A}_{m+1}(n+1) = \mathcal{A}_m(\mathcal{A}_{m+1}(n))$ par définition de la fonction d'Ackermann. Ainsi, la fonction $\mathcal{A}_{m+1}$ est définie par récursion primitive à partir de $g = \mathcal{A}_m \circ \underline{1}_0 \in \mathcal{F}_0$ et $h = \mathcal{A}_m \circ P_{2,2} \in \mathcal{F}_2$, qui sont des fonctions récursives primitives par hypothèse de récurrence. Ceci montre que $\mathcal{A}_{m+1} \in \mathcal{PR}$. □

Proposition 2.6.5. *La fonction d'Ackermann est calculable.*

Preuve. Nous allons montrer que la fonction d'Ackermann est calculable en décrivant un algorithme de calcul de celle-ci. Il est laissé en exercice de construire une machine de Turing calculant la fonction d'Ackermann à partir de cet algorithme.

On souhaite calculer des expressions de la forme

$$\mathcal{A}(m_1, \mathcal{A}(m_2, \ldots, \mathcal{A}(m_{k-1}, m_k)\ldots))$$

où $k \geqslant 1$. On code une telle expression par le k-uple

$$S = (m_1, \ldots, m_k).$$

Montrons que l'algorithme 3 calcule $\mathcal{A}(m,n)$. La notation $|S|$ désigne le nombre k d'éléments de la liste S. On écrit $S[i]$ pour désigner le i^{e} élément de S depuis la gauche et $S[-i]$ pour désigner le i^{e} élément de S depuis la droite. La notation $\text{Drop}[S, -i]$ signifie qu'on supprime les i derniers éléments de la liste S et la notation $\text{Join}[S,T]$ indique qu'on crée une liste composée des éléments de la liste S suivis des éléments de la liste T.

On procède par récurrence sur m. Pour une entrée de la forme $(0,n)$, on passe une unique fois dans la boucle et on sort $n+1$, qui est bien égal à $\mathcal{A}(0,n)$. Supposons à présent que $m \geqslant 0$ et que pour toute entrée de la forme (m,n), l'algorithme s'arrête et sort la valeur $\mathcal{A}(m,n)$. À partir d'une entrée de la forme $(m+1,n)$, on a successivement les mises à jour suivantes de la liste S :

$$\begin{aligned}(m+1,n) &\to (m, m+1, n-1)\\ &\to (m, m, m+1, n-2)\\ &\ \ \vdots\end{aligned}$$

Algorithm 3 Calcul de la fonction d'Ackermann.

Require: $(m, n) \in \mathbb{N}^2$
Ensure: $S[1]$ is $\mathcal{A}(m, n)$
  $S \leftarrow (m, n),\ T,\ p,\ q$
  while $|S| \geqslant 2$ **do**
    $T \leftarrow \mathrm{Drop}[S, -2],\ p \leftarrow S[-2],\ q \leftarrow S[-1]$
    if $p = 0$ **then**
    $S \leftarrow \mathrm{Join}[T, (q + 1)]$
    else if $p > 0$ and $q = 0$ **then**
    $S \leftarrow \mathrm{Join}[T, (p - 1, 1)]$
    else
    $S \leftarrow \mathrm{Join}[T, (p - 1, p, q - 1)]$
    end if
  end while
  return $S[1]$

$$
\begin{aligned}
&\to (\underbrace{m, \ldots, m}_{n \text{ fois}}, m + 1, 0)\\
&\to (\underbrace{m, \ldots, m}_{n \text{ fois}}, m, 1)\\
&\to^* (\underbrace{m, \ldots, m}_{n \text{ fois}}, \underbrace{\mathcal{A}(m, 1)}_{\mathcal{A}(m+1,0)})\\
&\to^* (\underbrace{m, \ldots, m}_{n-1 \text{ fois}}, \underbrace{\mathcal{A}(m, \mathcal{A}(m + 1, 0))}_{\mathcal{A}(m+1,1)})\\
&\to^* (\underbrace{m, \ldots, m}_{n-2 \text{ fois}}, \underbrace{\mathcal{A}(m, \mathcal{A}(m + 1, 1))}_{\mathcal{A}(m+1,2)})\\
&\vdots\\
&\to^* (\underbrace{\mathcal{A}(m, \mathcal{A}(m + 1, n - 1))}_{\mathcal{A}(m+1,n)})
\end{aligned}
$$

où $\to$ représente un passage dans la boucle et $\to^*$ désigne la clôture réflexive et transitive de $\to$, et où on a utilisé l'hypothèse de récurrence aux étapes impliquant $\to^*$. L'algorithme se termine donc bien sur toutes les entrées (m, n) de $\mathbb{N}^2$, et ce, avec la bonne valeur, c'est-à-dire $\mathcal{A}(m, n)$. □

Lemme 2.6.6. *Pour tout $m, n \in \mathbb{N}$, on a*

1. $n < \mathcal{A}(m, n)$
2. $\mathcal{A}(m, n) < \mathcal{A}(m, n + 1)$
3. $\mathcal{A}(m, n + 1) \leqslant \mathcal{A}(m + 1, n)$
4. $\mathcal{A}(m, n) < \mathcal{A}(m + 1, n)$.

Preuve. Montrons le point 1 par une double récurrence, sur m d'abord et ensuite sur n. Pour $m = 0$ et pour tout $n \in \mathbb{N}$, on a $\mathcal{A}(0, n) = n + 1 > n$. Considérons à présent un naturel m fixé et supposons que pour tout $n \in \mathbb{N}$, on a $n < \mathcal{A}(m, n)$. Nous devons montrer que pour tout $n \in \mathbb{N}$, on a $n < \mathcal{A}(m+1, n)$ également. On procède à présent par récurrence sur n. Pour $n = 0$, on a $A(m + 1, 0) = \mathcal{A}(m, 1) > 1 > 0$ où on a utilisé l'hypothèse de récurrence sur m. Supposons maintenant que n est un naturel tel que $n < \mathcal{A}(m+1, n)$. Nous devons montrer que $n + 1 < \mathcal{A}(m + 1, n + 1)$. En appliquant successivement l'hypothèse

de récurrence sur m et l'hypothèse de récurrence sur n, on obtient $\mathcal{A}(m+1,n+1) = \mathcal{A}(m,A(m+1,n)) > \mathcal{A}(m+1,n) \geqslant n+1$, comme souhaité.

Au vu du point 1, si $m \in \mathbb{N}_0$ et $n \in \mathbb{N}$, on a $\mathcal{A}(m,n+1) = \mathcal{A}(m-1,\mathcal{A}(m,n)) > \mathcal{A}(m,n)$. De plus, pour tout $n \in \mathbb{N}$, on a $\mathcal{A}(0,n) = n+1$ et $\mathcal{A}(0,n+1) = n+2$. Le point 2 est donc démontré.

Nous montrons le point 3. Considérons un naturel m fixé et procédons par récurrence sur n. Pour $n = 0$, on a $\mathcal{A}(m+1,0) = \mathcal{A}(m,1)$ par définition de la fonction d'Ackermann. Supposons maintenant que n est un naturel tel que $\mathcal{A}(m,n+1) \leqslant \mathcal{A}(m+1,n)$. Nous devons montrer que $\mathcal{A}(m,n+2) \leqslant \mathcal{A}(m+1,n+1)$. En appliquant successivement l'hypothèses de récurrence et le point 1, on obtient $\mathcal{A}(m+1,n) > n+1$. On en déduit que $\mathcal{A}(m+1,n) \geqslant n+2$. En appliquant la définition, l'inégalité obtenue précédemment et le point 2, on obtient que

$$\mathcal{A}(m+1,n+1) = \mathcal{A}(m,\underbrace{\mathcal{A}(m+1,n)}_{\geqslant n+2}) \geqslant \mathcal{A}(m,n+2)$$

comme souhaité.

Enfin, en combinant les points 2 et 3, on obtient que pour tout $m,n \in \mathbb{N}$, on a $\mathcal{A}(m,n) < \mathcal{A}(m,n+1) \leqslant \mathcal{A}(m+1,n)$. Le point 4 est donc démontré également. □

Proposition 2.6.7. *Pour toute fonction $f \in \mathcal{F}_d \cap \mathcal{PR}$, il existe $M \in \mathbb{N}$ tel que pour tout $\mathbf{m} \in \mathbb{N}^d$, on a $f(\mathbf{m}) < \mathcal{A}(M,\sup(\mathbf{m}))$.*

Preuve. Pour montrons que toute fonction récursive primitive a la propriété souhaitée, nous montrons que cette propriété est vraie pour les fonctions initiales et que l'ensemble des fonctions satisfaisant cette propriété est stable par composition et par récursion primitive.

Intéressons-nous d'abord au cas des fonctions initiales. On cherche un naturel M tel que $\underline{0} < \mathcal{A}(M,\sup()) = \mathcal{A}(M,0)$. Remarquons que la borne supérieure de l'ensemble vide dans $\mathbb{N}$ vaut 0 puisque le plus petit élément de $\mathbb{N}$ est 0. Le choix de $M = 0$ convient puisque $\mathcal{A}(0,0) = 1$. Ensuite, on cherche M tel que pour tout $m \in \mathbb{N}$, on a $\sigma(m) < \mathcal{A}(M,m)$. Le choix de $M = 1$ convient puisque $\mathcal{A}(1,m) = m+2$. Enfin, pour tout $d \in \mathbb{N}_0$ et tout $i \in \{1,\dots,d\}$, on cherche M tel que pour tout $m \in \mathbb{N}$, on a $P_{d,i}(\mathbf{m}) < \mathcal{A}(M,\sup(\mathbf{m}))$. Le choix de $M = 0$ convient pour tous d,i car on a $m_i < \sup(\mathbf{m}) + 1$.

Montrons la stabilité par composition. Soient $h_1,\dots,h_n \in \mathcal{F}_d$ et $g \in \mathcal{F}_n$, et supposons qu'il existe $H_1,\dots,H_n,G \in \mathbb{N}$ tels que

— pour tout $\mathbf{m} \in \mathbb{N}^d$, on a $h_i(\mathbf{m}) < \mathcal{A}(H_i,\sup(\mathbf{m}))$, et
— pour tout $\mathbf{m} \in \mathbb{N}^n$, on a $g(\mathbf{m}) < \mathcal{A}(G,\sup(\mathbf{m}))$.

On cherche $M \in \mathbb{N}$ tel que pour tout $\mathbf{m} \in \mathbb{N}^d$, on a

$$g(h_1(\mathbf{m}),\dots,h_n(\mathbf{m})) < \mathcal{A}(M,\sup(\mathbf{m})).$$

En posant $H = \sup(H_1,\dots,H_n)$, par le point 4 du lemme, pour tout $i \in \{1,\dots,n\}$ et pour tout $\mathbf{m} \in \mathbb{N}^d$, on a

$$h_i(\mathbf{m}) < \mathcal{A}(H,\sup(\mathbf{m})),$$

et donc

$$\sup(h_1(\mathbf{m}),\dots,h_n(\mathbf{m})) < \mathcal{A}(H,\sup(\mathbf{m})).$$

En posant $P = \sup(G,H)$, par les points 2, 3 et 4 du lemme, on obtient que pour tout $\mathbf{m} \in \mathbb{N}^d$, on a

$$\begin{aligned} g(h_1(\mathbf{m}),\dots,h_n(\mathbf{m})) &< \mathcal{A}(G,\sup(h_1(\mathbf{m}),\dots,h_n(\mathbf{m}))) \\ &< \mathcal{A}(G,\mathcal{A}(H,\sup(\mathbf{m}))) \\ &< \mathcal{A}(P,\mathcal{A}(P+1,\sup(\mathbf{m}))) \end{aligned}$$

$$\begin{aligned} &= \mathcal{A}(P+1, \sup(\mathbf{m})+1) \\ &\leqslant \mathcal{A}(P+2, \sup(\mathbf{m})). \end{aligned}$$

Ainsi, $M = P + 2$ convient.

Enfin, montrons la stabilité par récursion primitive. Soient $g \in \mathcal{F}_d$ et $h \in \mathcal{F}_{d+2}$, et supposons qu'il existe $G, H \in \mathbb{N}$ tels que

— pour tout $\mathbf{m} \in \mathbb{N}^d$, on a $g(\mathbf{m}) < \mathcal{A}(G, \sup(\mathbf{m}))$, et
— pour tout $\mathbf{m} \in \mathbb{N}^{d+2}$, on a $h(\mathbf{m}) < \mathcal{A}(H, \sup(\mathbf{m}))$.

Soit $f \in \mathcal{F}_{d+1}$ la fonction définie par récursion primitive à partir de g et h. On cherche $M \in \mathbb{N}$ tel que pour tout $\mathbf{m} \in \mathbb{N}^{d+1}$, on a

$$f(\mathbf{m}) < \mathcal{A}(M, \sup(\mathbf{m})).$$

D'abord, remarquons que par hypothèse sur h et par le point 1 du lemme, pour tout $\mathbf{m} \in \mathbb{N}^d$ et tout $n \in \mathbb{N}_0$, on a

$$\sup(\mathbf{m}, n, f(\mathbf{m}, n)) \leqslant \mathcal{A}(H, \sup(\mathbf{m}, n-1, f(\mathbf{m}, n-1))). \tag{2.1}$$

Soient $\mathbf{m} \in \mathbb{N}^d$ et $n \in \mathbb{N}$ fixés. En itérant n fois l'inégalité (2.1) et en utilisant point 2 du lemme, on obtient que

$$\begin{aligned} f(\mathbf{m}, n) &\leqslant \mathcal{A}(H, \sup(\mathbf{m}, n-1, f(\mathbf{m}, n-1))) \\ &\leqslant \mathcal{A}(H, \mathcal{A}(H, \sup(\mathbf{m}, n-2, f(\mathbf{m}, n-2)))) \\ &\vdots \\ &\leqslant \underbrace{\mathcal{A}(H, \mathcal{A}(H, \ldots, A(H,}_{n \text{ fois}} \sup(\mathbf{m}, 0, f(\mathbf{m}, 0))) \ldots)) \\ &= \underbrace{\mathcal{A}(H, \mathcal{A}(H, \ldots, \mathcal{A}(H,}_{n \text{ fois}} \sup(\mathbf{m}, g(\mathbf{m}))) \ldots)). \end{aligned}$$

Par hypothèse sur g et par le point 1 du lemme, on a

$$\sup(\mathbf{m}, g(\mathbf{m})) < \mathcal{A}(G, \sup(\mathbf{m})).$$

En posant $P = \sup(G, H)$ et en utilisant les points 2 et 4 du lemme, on obtient que

$$\begin{aligned} f(\mathbf{m}, n) &< \underbrace{\mathcal{A}(P, \mathcal{A}(P, \ldots, \mathcal{A}(P,}_{n \text{ fois}} \mathcal{A}(P+1, \sup(\mathbf{m}))) \ldots)) \\ &= \underbrace{\mathcal{A}(P, \mathcal{A}(P, \ldots, \mathcal{A}(P,}_{n-1 \text{ fois}} \mathcal{A}(P+1, \sup(\mathbf{m})+1)) \ldots)) \\ &\vdots \\ &= \mathcal{A}(P+1, \sup(\mathbf{m}) + n). \end{aligned}$$

Par la proposition 2.6.3, pour tout $k, \ell \in \mathbb{N}$, on a $k + \ell < \mathcal{A}(2, \sup(k, \ell))$. En particulier, on a $\sup(\mathbf{m}) + n < \mathcal{A}(2, \sup(\mathbf{m}, n))$. En combinant ces inégalités et en utilisant les points 2, 3 et 4 du lemme, on obtient que

$$\begin{aligned} f(\mathbf{m}, n) &< \mathcal{A}(P+1, \mathcal{A}(2, \sup(\mathbf{m}, n))) \\ &\leqslant \mathcal{A}(P+1, \mathcal{A}(P+2, \sup(\mathbf{m}, n))) \\ &= \mathcal{A}(P+2, \sup(\mathbf{m}, n)+1) \\ &\leqslant \mathcal{A}(P+3, \sup(\mathbf{m}, n)). \end{aligned}$$

Ainsi, le choix de $M = P + 3$ convient. □

Corollaire 2.6.8. *La fonction d'Ackermann n'est pas récursive primitive.*

Preuve. Procédons par l'absurde en supposant que la fonction d'Ackermann soit récursive primitive. Par la proposition précédente, on peut alors trouver un naturel M tel que pour tout $(m,n) \in \mathbb{N}^2$, on ait $\mathcal{A}(m,n) < \mathcal{A}(M, \sup(m,n))$. Cette inégalité évaluée en $(m,n) = (M,M)$ donne lieu à une contradiction. □

2.7 Fonctions non calculables

Tout d'abord, nous montrons que l'ensemble $\mathcal{F}_1 \setminus \mathcal{C}$ est non vide. En particulier, il existe des fonctions non calculables.

Proposition 2.7.1. *Il existe des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ non calculables.*

Preuve. L'ensemble $\mathcal{F}_1$ des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ est non dénombrable [4]. L'ensemble $\mathcal{C}$ des fonctions calculables, lui, est dénombrable. En effet, il suffit de remarquer que les machines de Turing elles-mêmes sont dénombrables. L'ensemble $\mathcal{F}_1 \cap \mathcal{C}$ des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ calculables est donc dénombrable lui aussi. D'où la conclusion. □

Remarquons qu'on a en fait montré bien plus qu'annoncé puisque nous avons montré que l'ensemble $\mathcal{F}_1 \setminus \mathcal{C}$ était non dénombrable !

Nous allons à présent exhiber une fonction non calculable. Puisque l'ensemble des $\mathcal{F}_1 \cap \mathcal{C}$ est dénombrable, il existe une énumération de ses éléments (c'est-à-dire une bijection de $\mathbb{N}$ dans $\mathcal{F}_1 \cap \mathcal{C}$). Nous notons $f_0, f_1, f_2, \ldots$ la liste des fonctions de $\mathcal{F}_1 \cap \mathcal{C}$. Si une telle énumération existe, elle n'est pas calculable, au sens suivant : la fonction $u\colon \mathbb{N}^2 \to \mathbb{N},\ (m,n) \mapsto f_m(n)$ n'est pas calculable.

Proposition 2.7.2. *La fonction u n'est pas calculable.*

Preuve. Supposons au contraire qu'il existe une machine de Turing $\mathcal{M}$ calculant u. Alors la machine de Turing $\mathcal{C}_1\mathcal{M}\mathbf{u}\mathcal{R}$ calcule la fonction $v\colon \mathbb{N} \to \mathbb{N},\ n \mapsto u(n,n)+1$. Comme $v \in \mathcal{F}_1 \cap \mathcal{C}$, il existe $k \in \mathbb{N}$ tel que $v = f_k$. C'est impossible car ces fonctions prennent des valeurs différentes en k. □

Une autre fonction non calculable est donnée par la fonction β définie comme suit. Cette fonction nous sera utile dans la suite lorsque nous parlerons du problème de l'arrêt.

Définition 2.7.3. On dit qu'un naturel m est *produit par une machine de Turing* lorsque $q_0.\#\underline{\#} \vdash^* h.\#\,\mathtt{u}^m\,\underline{\#}$. Pour tout $m \in \mathbb{N}$, on note $\mathcal{M}_m$ la machine de Turing dessinée à la Figure 2.16. Il s'agit d'une machine de Turing d'alphabet $\{\#, \mathtt{u}\}$, possédant $m+1$ états et produisant l'entier m. Pour tout $m \in \mathbb{N}$, on note $\beta(m)$ le plus grand entier produit par une machine de Turing d'alphabet $\{\#, \mathtt{u}\}$ et possédant $m+1$ états. La *fonction β* est la fonction $\mathbb{N} \to \mathbb{N},\ m \mapsto \beta(m)$.

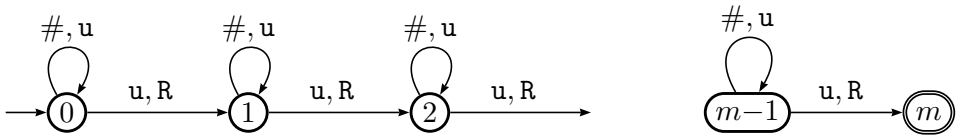


FIGURE 2.16 – La machine de Turing $\mathcal{M}_m$ produit l'entier m.

4. Pourquoi ?

Proposition 2.7.4. *Toute fonction numérique calculable est calculable par une machine de Turing sur l'alphabet* $\{\#, \mathtt{u}\}$.

Idée de la preuve. Puisque les fonctions calculables et récursives coïncident, il suffit de montrer que les fonctions initiales sont calculables par une machine de Turing sur l'alphabet $\{\#, \mathtt{u}\}$, et que l'ensemble des fonctions calculables par une machine de Turing sur l'alphabet $\{\#, \mathtt{u}\}$ est stable par composition, récursion primitive et minimisation de fonctions sûres. Il s'agit donc d'un raffinement de la preuve de la proposition 2.5.6 : à chaque étape, il faut vérifier que les machines de Turing construites sont d'alphabet $\{\#, \mathtt{u}\}$. Les détails sont laissés en exercices. Il faudra en particulier justifier que les machines $\mathcal{S}_{L,d}, \mathcal{S}_{R,d}, \mathcal{C}_d, \mathcal{E}_d$ utilisées sont également d'alphabet $\{\#, \mathtt{u}\}$.

Proposition 2.7.5. *La fonction* β *n'est pas calculable.*

Preuve. Montrons d'abord que β est une fonction strictement croissante. Soit $m \in \mathbb{N}$. Soit B_m une machine de Turing d'alphabet $\{\#, \mathtt{u}\}$ ayant $m+1$ états et produisant $\beta(m)$ (une telle machine existe par définition de $\beta(m)$). Alors la machine de Turing $B_m M_1$ est d'alphabet $\{\#, \mathtt{u}\}$, a $m+2$ états et produit $\beta(m)+1$. On en déduit que $\beta(m+1) \geqslant \beta(m)+1$. Procédons maintenant par l'absurde et supposons que β est calculable. Alors la fonction $\mathbb{N} \to \mathbb{N},\ m \mapsto \beta(2m)$ est calculable aussi. Soit une machine de Turing $\mathcal{N}$ la calculant et soit k le nombre d'états de $\mathcal{N}$. Au vu de la proposition précédente, on peut supposer que $\mathcal{N}$ est d'alphabet $\{\#, \mathtt{u}\}$. Pour tout $m \in \mathbb{N}$, la machine de Turing $\mathcal{M}_m\mathcal{N}$ produit $\beta(2m)$, a $m+k$ états et est d'alphabet $\{\#, \mathtt{u}\}$. D'où, pour tout $m \in \mathbb{N}$, on a $\beta(m+k-1) \geqslant \beta(2m)$. Puisque β est une fonction strictement croissante, on obtient que pour tout $m \in \mathbb{N}$, on a $m+k-1 \geqslant 2m$, c'est-à-dire $m \leqslant k-1$, une contradiction. □

2.8 Langages décidables

Définition 2.8.1. Un langage L sur un alphabet A ne contenant pas le symbole blanc $\#$ est dit *décidable* si sa fonction caractéristique $\chi_L \colon A^* \to \mathbb{N}$ est calculable.

Autrement dit, un langage $L \subseteq A^*$ (avec $\# \notin A$) est calculable s'il existe une machine de Turing $\mathcal{M} = (Q, q_0, h, B, \delta)$ avec $A \subseteq B$ telle que, pour tout $w \in A^*$, lorsqu'elle est exécutée à partir de la configuration initiale $q_0.\#w\underline{\#}$, atteint la configuration d'arrêt $h.\#\,\mathtt{u}\,\underline{\#}$ si w appartient au langage L et atteint la configuration d'arrêt $h.\#\underline{\#}$ sinon. De manière informelle, dans le premier cas, on dira que la machine répond « oui » et dans le deuxième cas, on dira que la machine répond « non ».

Puisque les fonctions calculables et récursives coïncident, les langages décidables sont également appelés les langages récursifs et la famille des langages décidables est notée R.

Si A est un alphabet totalement ordonné, l'ordre radiciel sur A^* est l'ordre défini comme suit : les mots de A^* sont ordonnés longueur par longueur, et pour les mots de même longueur, ils sont ordonnés en suivant l'ordre lexicographique[5]. Par exemple, si $A = \{a, b, c\}$ avec $a < b < c$, les premiers mots de A^* dans l'ordre radiciel sont donné à la table 2.3.

Lemme 2.8.2. *Soit* A *un alphabet totalement ordonné. La fonction de* A^* *dans* A^* *qui à un mot de* A^* *associe le mot suivant de* A^* *dans l'ordre radiciel est calculable.*

Preuve. Supposons que $A = \{a_1, \ldots, a_k\}$ avec $a_1 < \cdots a_k$. On vérifie aisément que l'organigramme de la figure 2.17 calcule la fonction souhaitée. □

5. C'est-à-dire, l'ordre du dictionnaire.

0	ε	6	ac	12	cc	18	abc
1	a	7	ba	13	aaa	19	aca
2	b	8	bb	14	aab	20	acb
3	c	9	bc	15	aac	21	acc
4	aa	10	ca	16	aba	22	baa
5	ab	11	cb	17	abb	23	bab

TABLE 2.3 – Ordre radiciel sur $\{a, b, c\}^*$.

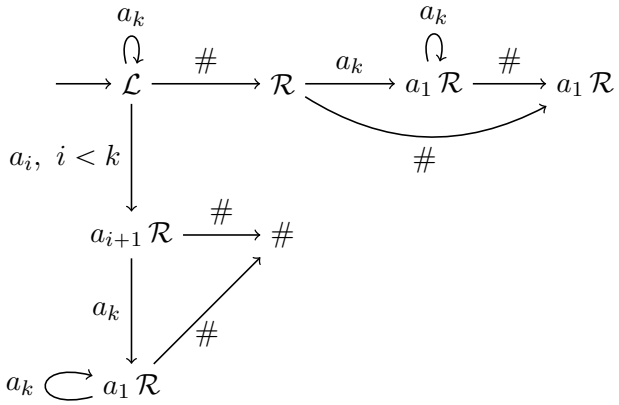


FIGURE 2.17 – Organigramme pour la fonction successeur dans l'ordre radiciel.

Proposition 2.8.3. *Soient $A_1, \ldots, A_{d+1}$ des alphabets ne contenant pas le symbole blanc $\#$ et soit μ un symbole n'appartenant pas à $\bigcup_{i=1}^{d+1} A_i \cup \{\#\}$. Une fonction*

$$f\colon A_1^* \times \cdots \times A_d^* \to A_{d+1}^*$$

est calculable si et seulement si le langage

$$L_f = \{w_1\mu w_2\mu \ldots \mu w_d\mu f(w_1, \ldots, w_d) : w_1 \in A_1^*, \ldots, w_d \in A_d^*\}$$

est décidable.

Preuve. Soit $f\colon A_1^* \times \cdots \times A_d^* \to A_{d+1}^*$. Supposons d'abord que L_f est décidable. Soit $\mathcal{M}_{L_f}$ une machine de Turing calculant la fonction caractéristique de L_f. Notons $\mathcal{S}$ la machine de Turing décrite par l'organigramme de la figure 2.17. Alors l'organigramme de la figure 2.18 calcule la fonction f. En effet, si on démarre avec la configuration $q_0.\#w_1\#w_2\#\ldots\#w_d\underline{\#}$, cette machine de Turing énumère les mots de A_{d+1}^* dans l'ordre radiciel en utilisant la machine $\mathcal{S}$, et pour chaque $i \in \mathbb{N}$, si x_i est le i^{e} mot énuméré, elle vérifie que $w_1\mu w_2\mu \ldots \mu w_d\mu x_i \in L_f$. Si c'est le cas, alors $x_i = f(w_1, \ldots, w_d)$ et la machine de Turing sort la valeur obtenue. Sinon, on passe au mot x_{i+1} suivant. Comme f est une fonction totale, cette procédure va nécessairement s'arrêter, et ce, avec la bonne image.

Inversement, supposons que f est fonction calculable. Soit $\mathcal{M}_f$ une machine de Turing calculant f. On va expliquer les étapes de construction d'une machine de Turing calculant la fonction caractéristique de L_f. On démarre avec la configuration $q_0.\#w\underline{\#}$ où $w \in \bigcup_{i=1}^{d+1} A_i \cup \{\mu\}$. On commence par vérifier que w est de la forme $w_1\mu w_2\mu \ldots \mu w_d\mu w_{d+1}$ avec $w_1 \in A_1^*, \ldots, w_{d+1} \in A_{d+1}^*$. Si ce n'est pas le cas, la machine de Turing répond « non ». Sinon, on calcule $f(w_1, \ldots, w_d)$ au moyen de $\mathcal{M}_f$. Ensuite, on vérifie que $w_{d+1} = f(w_1, \ldots, w_d)$. Si ce n'est pas le cas, la machine de Turing répond « non » et sinon, elle répond « oui ». Les détails de construction sont laissés au lecteur. □

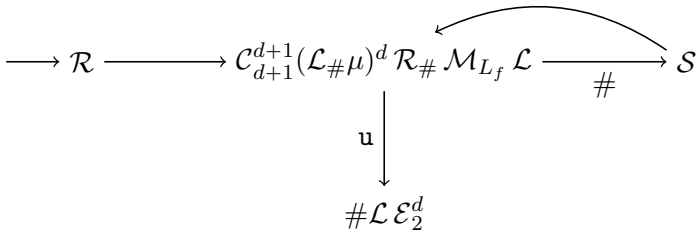


FIGURE 2.18 – Organigramme pour le calcul de f en utilisant $\mathcal{M}_{L_f}$.

Proposition 2.8.4. *La famille des langages décidables est stable pour l'union, l'intersection, la complémentation, la concaténation, et l'étoile de Kleene.*

Preuve. Soient K, L des langages sur un alphabet A. On a

— $\chi_{K\cap L} = \chi_K \cdot \chi_L$
— $\chi_{K\cup L} = \chi_K + \chi_L - \chi_K \cdot \chi_L$
— $\chi_{A^*\setminus K} = 1 - \chi_K$.

Ceci montre la stabilité pour les opérations booléennes.

Montrons à présent la stabilité par concaténation. Supposons que K et L soient décidables. Soient $\mathcal{M}_K$ et $\mathcal{M}_L$ des machines de Turing calculant χ_K et χ_L respectivement. On décrit une machine de Turing calculant χ_{KL}. On démarre avec une configuration $q_0.\#w\underline{\#}$ où $w \in A^*$. Il y a $|w|+1$ factorisations de w de la forme $w = w_1w_2$. La machine de Turing crée ces factorisations l'une après l'autre, et pour chacune d'elles, teste si $w_1 \in K$ et $w_2 \in L$ au moyen de $\mathcal{M}_K$ et $\mathcal{M}_L$. Si on trouve une factorisation telle que les deux réponses sont positives, alors la machine de Turing répond « oui » ; sinon, elle répond « non ».

Enfin, montrons à présent la stabilité par étoile de Kleene. Supposons que K est décidable. Soit $\mathcal{M}_K$ une machine de Turing calculant χ_K. On décrit une machine de Turing calculant χ_{K^*}. On démarre avec une configuration $q_0.\#w\underline{\#}$ où $w \in A^*$. Il y a un nombre fini de factorisations de w de la forme $w = w_1 \cdots w_n$ avec $n \geqslant 0$ et $w_i \neq \varepsilon$ pour tout $i \in \{1, \ldots, n\}$. La machine de Turing crée ces factorisations l'une après l'autre, et pour chacune d'elle, teste si $w_i \in K$ pour tout $i \in \{1, \ldots, n\}$ grâce à $\mathcal{M}_K$. Si on trouve une factorisation telle que tous les facteurs appartiennent à K, alors la machine de Turing répond « oui » et elle répond « non » sinon. □

2.9 Langages acceptables, machines de Turing universelles et élimination des configurations pendantes

Une machine de Turing ne s'arrête pas toujours à partir d'une configuration initiale donnée. D'où la définition suivante.

Définition 2.9.1. Soit une machine de Turing $\mathcal{M} = (Q, q_0, h, A, \delta)$. Un mot w sur $A\setminus\{\#\}$ est *accepté* par $\mathcal{M}$ si, en partant de la configuration $q_0.\#w\underline{\#}$, on atteint une configuration d'arrêt en suivant les transitions de $\mathcal{M}$. Le *langage accepté par une machine de Turing* $\mathcal{M}$, noté $L(\mathcal{M})$, est l'ensemble des mots qu'elle accepte. Un langage est dit *acceptable* s'il existe une machine de Turing qui l'accepte.

Afin d'étudier les propriétés des langages acceptables, nous allons montrer comment construire une machine de Turing, dite universelle, capable de simuler le comportement de n'importe quelle machine de Turing. Cette idée fondamentale est déjà présente dans l'article original de Turing [5], même si le codage présenté ici diffère de l'original.

Nous commençons par encoder les machines de Turing et les configurations machine par des mots finis sur un alphabet de deux symboles $\mathtt{u}$ et $\star$. Sans perte de généralité, nous supposerons toujours qu'une machine de Turing possède un ensemble d'états Q inclus dans $Q_\infty = \{p_0, p_1, p_2, \ldots)$ avec $p_0 = h, p_1 = q_0$, et un alphabet A inclus dans $A_\infty = \{a_1, a_2, a_3, \ldots\}$ avec $a_1 = \#$ et $\star \notin A_\infty$. On considère le codage ρ donné à la table 2.4.

x	$\rho(x)$
p_i	$\mathtt{u}^{i+1}$
L	$\mathtt{u}$
R	$\mathtt{u}^2$
a_i	$\mathtt{u}^{i+2}$

TABLE 2.4 – Codage des états et des instructions de la machine de Turing universelle.

Pour chaque i, j tels que la transition $\delta(p_i, a_j)$ est définie, on définit

$$\rho_{ij} = \star\rho(p_i) \star \rho(a_j) \star \rho(p) \star \rho(x)\star$$

où $(p, x) = \delta(p_i, a_j)$. Si la transition $\delta(p_i, a_j)$ n'est pas définie, on pose $\rho_{ij} = \varepsilon$. Si $Q = \{p_0, \ldots, p_k\}$ et $A = \{a_1, \ldots, a_\ell\}$, on note

$$c(\mathcal{M}) = \rho_{11} \cdots \rho_{1\ell} \cdots \rho_{k1} \cdots \rho_{k\ell}.$$

Montrons maintenant comment coder les configurations machine. Pour un mot $w \in A^*$ de longueur k, on définit

$$\rho'(w) = \star\rho(w[1]) \star \rho(w[2]) \star \cdots \star \rho(w[k]) \star .$$

Remarquons que pour une lettre $a \in A$, on a $\rho'(a) = \star\rho(a)\star \neq \rho(a)$. Remarquons aussi que $\rho'(\varepsilon) = \star$. Ensuite, on définit le code de la partie significative d'une configuration machine $q.u\underline{a}v$ par

$$\rho(q.u\underline{a}v) = \star\rho(q) \star \rho'(u)\rho'(a)\rho'(v).$$

Enfin, on définit

$$c(w) = \rho(q_0.\#w\underline{\#})$$

pour un mot fini w ne contenant pas les symboles $\#$ et $\star$.

La configuration mémoire initiale de la machine de Turing universelle est de la forme

$$\#c(\mathcal{M})c(w)\underline{\#}.$$

L'idée est que la machine de Turing universelle simule le comportement de $\mathcal{M}$ sur w, c'est à dire qu'elle est programmée de telle sorte que si $\mathcal{M}$ passe de la configuration machine $q.r$ à $q'.r'$, la machine de Turing universelle met à jour sa mémoire en passant de $\#c(\mathcal{M})\rho(q.r)\underline{\#}$ à $\#c(\mathcal{M})\rho(q.r')\underline{\#}$. Pour effectuer une telle mise à jour, la partie à droite de la cellule référencée pourra être utilisée comme zone de travail par la machine de Turing universelle. Cette partie ne contient jamais le symbole $\star$.

Exemple 2.9.2. Considérons la machine de Turing de la figure 2.19. Comme convenu, on note $a_1 = \#$ et $a_2 = a$, l'état initial est noté p_1 et l'état final p_0. On obtient

$$\rho_{11} = \star\rho(p_1) \star \rho(a_1) \star \rho(p_1) \star \rho(\mathtt{R})\star = \star\,\mathtt{u}^2 \star \mathtt{u}^3 \star \mathtt{u}^2 \star \mathtt{u}^2 \star$$
$$\rho_{12} = \star\rho(p_1) \star \rho(a_2) \star \rho(p_0) \star \rho(\#)\star = \star\,\mathtt{u}^2 \star \mathtt{u}^4 \star \mathtt{u} \star \mathtt{u}^3 \star.$$

En considérant la configuration machine $p_1.\#a\underline{a}a\#a$, la partie significative de la mémoire de la machine de Turing universelle sera donnée par $\#c(\mathcal{M})\rho(p_1.\#a\underline{a}a\#a)\underline{\#}$, c'est-à-dire

$$\#\underbrace{\star\,\mathtt{u}^2 \star \mathtt{u}^3 \star \mathtt{u}^2 \star \mathtt{u}^2 \star \star\, \mathtt{u}^2 \star \mathtt{u}^4 \star \mathtt{u} \star \mathtt{u}^3 \star}_{c(\mathcal{M})} \star \underbrace{\mathtt{u}^2}_{\rho(p_1)} \star \underbrace{\star\,\mathtt{u}^3 \star \mathtt{u}^4 \star}_{\rho'(\#a)} \underbrace{\star\,\mathtt{u}^4 \star}_{\rho'(a)} \underbrace{\star\,\mathtt{u}^4 \star \mathtt{u}^3 \star \mathtt{u}^4 \star}_{\rho'(a\#a)} \underline{\#}.$$

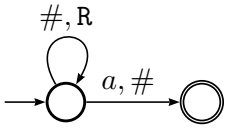


FIGURE 2.19 – Une machine de Turing à deux états.

Nous synthétisons la discussion qui précède dans le résultat suivant.

Le codage présenté permet d'obtenir les résultat suivants [6]. En effet, il ne s'agit que de vérifications et constructions syntaxiques.

Proposition 2.9.3.

- *Les langages suivants sont décidables.*
  - $\{c(w) : w$ *un mot fini*$\}$
  - $\{c(\mathcal{M}) : \mathcal{M}$ *une machine de Turing*$\}$
  - $\{c(\mathcal{M})c(w) : \mathcal{M}$ *une machine de Turing,* w *un mot fini*$\}$.
- *Pour toute machine de Turing* $\mathcal{M}$*, on peut construire des machines de Turing réalisant les actions suivantes, quels que soient la configuration mémoire* r *et l'état* q *:*
  - $q_0.r \vdash^* h.\#c(\mathcal{M})\rho(q_0.r)\underline{\#}$
  - $q_0.\#c(\mathcal{M})\rho(q.r)\underline{\#} \vdash^* h.r$.

En fait, le choix du codage importe peu pour la théorie. Ce qui compte est d'avoir un codage c satisfaisant les propositions précédentes. De nombreux auteurs ont construit des machines de Turing universelles. Les nombres s d'états et t de lettres des plus petites d'entre elles sont donnés par les couples (s, t) suivants : $(15, 2)$, $(9, 3)$, $(6, 4)$, $(5, 5)$, $(4, 6)$, $(3, 9)$, $(2, 18)$. En comparaison, notre codage est construit sur 3 lettres, mais le nombre d'états d'une machine universelle correspondant à ce codage est probablement plus grand que 9. Nous ne détaillerons pas ici la construction d'une telle machine de Turing, mais nous espérons que le lecteur sera convaincu qu'il pourrait, avec de la patience, mener à bien cette construction s'il le souhaitait. Des constructions impressionnantes de machines de Turing universelles sont celles réalisées entièrement en LEGO® ! Le lecteur est invité à regarder plusieurs des nombreuses vidéos en ligne à ce sujet. Il existe même des parodies de certaines d'entre elles !

Une machine de Turing universelle peut être vue comme un super-programme capable d'exécuter d'autres programmes. L'intérêt de cette approche est immense. Nous avons maintenant prise sur les programmes eux-mêmes ! Nous sommes par exemple capables de détecter certaines situations problématiques lors de l'exécution de ceux-ci, et aussi d'en modifier l'action si de tels problèmes sont rencontrés. Un deuxième atout de taille est que nous pouvons également simuler plusieurs programmes en parallèle. Nous verrons que ces deux idées nous seront utiles. Tout d'abord, nous tirons profit de la première afin d'éliminer le problème des configurations pendantes.

Théorème 2.9.4. *Il existe une machine de Turing* $\mathcal{U}$ *telle que pour toute machine de Turing* $\mathcal{M}$ *et toute configuration mémoire* r *de* $\mathcal{M}$*,*

- $\mathcal{U}$ *atteint la configuration d'arrêt* $h.\#c(\mathcal{M})\rho(h.s)\underline{\#}$ *à partir de* $q_0.\#c(\mathcal{M})\rho(q_0.r)\underline{\#}$ *si* $\mathcal{M}$ *atteint la configuration d'arrêt* $h.s$ *à partir de* $q_0.r$ *;*
- $\mathcal{U}$ *ne s'arrête pas à partir de* $q_0.\#c(\mathcal{M})\rho(q_0.r)\underline{\#}$ *si* $\mathcal{M}$ *ne s'arrête pas ou atteint une configuration pendante à partir de* $q_0.r$.

6. Avec la convention que les mots sont écrits sur l'alphabet A_∞ et que les états des machines de Turing appartiennent à Q_∞

Dans le théorème précédent, chacune des machines de Turing sont lancées à partir de leurs états initiaux respectifs (appelés tous deux q_0) et s'arrêtent sur leurs états finaux respectifs (appelés tous deux h). Cela ne cause aucune ambiguïté puisqu'il n'y a jamais d'interférence entre ces états.

Corollaire 2.9.5. *Pour toute machine de Turing $\mathcal{M}$, on peut construire une machine de Turing $\mathcal{M}'$ qui à partir d'une configuration initiale donnée, atteint la même configuration d'arrêt que $\mathcal{M}$ lorsque $\mathcal{M}$ en en atteint une et ne s'arrête pas lorsque $\mathcal{M}$ ne s'arrête pas ou que $\mathcal{M}$ atteint une configuration pendante.*

Preuve. Supposons que $\mathcal{M}$ soit une machine de Turing. Notons $\mathcal{A}$ une machine de Turing qui réalise l'action $q_0.r \vdash^* h.\#c(\mathcal{M})\rho(q_0.r)\underline{\#}$. Soit $\mathcal{U}$ une machine de Turing universelle comme dans le théorème 2.9.4. Notons $\mathcal{B}$ une machine de Turing qui réalise l'action $q_0.\#c(\mathcal{M})\rho(q.r)\underline{\#} \vdash^* h.r$. La machine de Turing $\mathcal{A}\,\mathcal{U}\,\mathcal{B}$ convient pour la thèse. □

Passons à présent aux propriétés des langages acceptables. Nous allons exploiter l'idée d'une machine de Turing universelle capable de simuler plusieurs machines de Turing en parallèle.

Proposition 2.9.6. *Un langage est décidable si et seulement si lui et son complémentaire sont acceptables.*

Preuve. Si L est un langage décidable et si $\mathcal{M}$ est une machine de Turing calculant χ_L, alors la machine de Turing $\mathcal{M}\,\mathcal{P}$ accepte L, où $\mathcal{P}$ est la machine de Turing de la figure 2.20. Ainsi, tout langage décidable est acceptable. Le complémentaire d'un langage décidable étant encore décidable, on obtient la condition nécessaire.

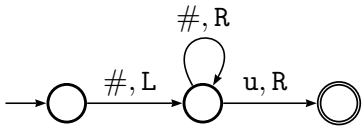


FIGURE 2.20 – Machine de Turing $\mathcal{P}$.

Montrons à présent la condition suffisante. Soit L un langage sur un alphabet A et supposons que $\mathcal{M}$ et $\mathcal{M}'$ sont des machines de Turing acceptant L et $A^*\backslash L$ respectivement. On va décrire une machine de Turing $\mathcal{N}$ décidant L. En partant de $q_0.\#w\underline{\#}$ avec $w \in A^*$, on va, au moyen d'une machine de Turing universelle, simuler une transition de $\mathcal{M}$ sur $q_0.\#w\underline{\#}$ et tester si $\mathcal{M}$ a atteint une configuration d'arrêt. Si oui, on sait que $w \in L$ et la machine de Turing $\mathcal{N}$ répond « oui ». Si non, on va simuler une transition de $\mathcal{M}'$ sur $q_0.\#w\underline{\#}$ et tester si $\mathcal{M}'$ a atteint une configuration d'arrêt. Si oui, on sait que $w \notin L$ et la machine de Turing $\mathcal{N}$ répond « non ». Si non, on recommence avec une nouvelle transition de $\mathcal{M}$. La machine $\mathcal{N}$ va donc alterner les simulations de $\mathcal{M}$ et $\mathcal{M}'$, à chaque étape en considérant une transition supplémentaire. Cette procédure va s'arrêter puisque le mot w appartient forcément à L ou à $A^*\backslash L$. □

Définition 2.9.7. On dit qu'un langage infini L est *récursivement énumérable* s'il existe une bijection de $\mathbb{N}$ dans L calculable.

Proposition 2.9.8. *Un langage infini est acceptable si et seulement s'il est récursivement énumérable.*

Preuve. Soit $\mathcal{M}$ une machine de Turing acceptant un langage infini L. Sans perte de généralité, on peut supposer que $\mathcal{M}$ n'atteint jamais de configuration pendante. On va

montrer comment obtenir une machine de Turing qui calcule une bijection de $\mathbb{N}$ dans L. On énumère les mots de A^* dans l'ordre radiciel : $w_0, w_1, w_2, \ldots$. On travaille avec deux listes : une liste L_1 qui contiendra les éléments énumérés de L et une liste L_2 qui contiendra les éléments de A^* dont on n'a pas encore prouvé l'appartenance à L. Pour commencer, L_1 est la liste vide et la liste L_2 contient uniquement le mot w_0. À l'étape k, pour chaque mot w_i de L_2, au moyen d'une machine de Turing universelle, on applique un maximum de k transitions de $\mathcal{M}$ à la configuration initiale $q_0.\#w_i\underline{\#}$. Si $\mathcal{M}$ atteint une configuration d'arrêt en moins de k transitions, on déplace le mot w_i de la liste L_2 vers la liste L_1. Sinon, le mot w_i est conservé dans la liste L_2 pour l'étape $k+1$. Si on continuait cette procédure indéfiniment, la liste L_1 contiendrait exactement les mots de L (puisque nous avons pris soin de d'abord éliminer les configurations pendantes). Pour chaque entrée $n \in \mathbb{N}$, la machine de Turing s'arrête dès que la liste L_1 possède $n+1$ éléments et sort le $(n+1)^{\text{e}}$ mot de cette liste.

Supposons à présent que L est un langage infini récursivement énumérable. Soit une machine de Turing $\mathcal{M}$ calculant une bijection $\mathbb{N} \to L,\ n \mapsto w_n$. Décrivons une machine de Turing qui accepte L. Si w est un mot en entrée, la machine réalise une boucle parcourant successivement les naturels n, pour chacun d'entre eux produit le mot w_n au moyen de $\mathcal{M}$ et ensuite teste si $w = w_n$. Dès que la condition est vérifiée, la machine s'arrête. Cette machine s'arrête donc exactement sur les mots de L. □

Au vu de la proposition précédente, les langages acceptables sont également appelés les langages récursivement énumérables et la famille des langages acceptables est notée RE.

2.10 Le problème de l'arrêt

Théorème 2.10.1. *Le langage*

$$\mathcal{A} = \{c(\mathcal{M})c(w) : w \text{ est accepté par la machine de Turing } \mathcal{M}\}$$

est indécidable.

Preuve. On procède par l'absurde. Supposons que $\mathcal{A}$ est décidé par une machine de Turing $\mathcal{M}_{\mathcal{A}}$. Nous allons en déduire que la fonction β est calculable, une contradiction. Soit $m \in \mathbb{N}$. Il existe un nombre fini k de machines de Turing d'alphabet $\{\#, \mathtt{u}\}$ ayant $m+1$ états, numérotés $0, \ldots, m$. On choisit une énumération de ces machines de Turing : $\mathcal{M}_1, \ldots, \mathcal{M}_k$. Pour chaque $i \in \{1, \ldots, k\}$, on peut décider à l'aide de $\mathcal{M}_{\mathcal{A}}$ si $c(\mathcal{M}_i)c(\varepsilon) \in \mathcal{A}$. On fait une boucle sur i. À l'étape i, si $c(M_i)c(\varepsilon) \notin \mathcal{A}$, on passe à $i+1$. Sinon, c'est-à-dire si $\mathcal{M}_i$ s'arrête à partir de $q_0.\#\underline{\#}$, alors on simule l'exécution de $\mathcal{M}_i$ sur ε au moyen d'une machine de Turing universelle et on teste si la configuration d'arrêt est de la forme $h.\#u^t\underline{\#}$ avec $t \in \mathbb{N}$. Si ce n'est pas le cas, on passe à $i+1$. Sinon, on compare t au plus grand entier produit jusqu'à présent. On conserve la plus grande des deux valeurs et on passe à $i+1$. Comme la boucle est finie, la dernière valeur conservée est $\beta(m)$. □

Le langage $\mathcal{A}$ du théorème précédent est appelé le langage du problème de l'arrêt. Ceci est justifié par le fait, simple mais important, que le codage c choisi n'influence pas le résultat obtenu. On s'autorise donc à dire que le problème de l'arrêt lui-même est indécidable.

Remarquons qu'on a en fait montré le résultat suivant.

Théorème 2.10.2. *Le langage*

$$\mathcal{A}_\varepsilon = \{c(\mathcal{M})c(\varepsilon) : \varepsilon \text{ est accepté par la machine de Turing } \mathcal{M}\}$$

est indécidable.

Exercice. Montrer que les langages suivants sont indécidables.

1. $\{c(\mathcal{M})c(w) : \mathcal{M}$ ne s'arrête pas à partir de $\#w\underline{\#}\}$
2. $\{c(\mathcal{M}) : \mathcal{M}$ ne s'arrête pas à partir de $\#\underline{\#}\}$
3. $\{c(\mathcal{M}) : \mathcal{M}$ ne s'arrête pas à partir de $\underline{\#}\}$
4. $\{c(\mathcal{M}) : L(\mathcal{M}) \neq \varnothing\}$
5. $\{c(\mathcal{M}) : L(\mathcal{M}) = \mathrm{alph}(L(\mathcal{M}))^*\}$, où $\mathrm{alph}(L)$ est l'alphabet minimal de L
6. $\{c(\mathcal{M})c(\mathcal{N}) : L(\mathcal{M}) \cap L(\mathcal{N}) \neq \varnothing\}$
7. $\{c(\mathcal{M})c(\mathcal{N}) : L(\mathcal{M}) = L(\mathcal{N})\}$.

Théorème 2.10.3. *Le langage $\mathcal{A}$ est acceptable.*

Preuve. Montrons comment construire une machine de Turing acceptant $\mathcal{A}$. On démarre avec une configuration machine $q_0.\#y\underline{\#}$ où $y \in \{\star, \mathtt{u}\}^*$. On teste d'abord si $y = c(\mathcal{M})c(w)$ pour une machine de Turing $\mathcal{M}$ et w un mot fini sur l'alphabet de $\mathcal{M}$ ne contenant pas le symbole #. Si c'est le cas, on simule l'exécution de $\mathcal{M}$ sur w au moyen d'une machine de Turing universelle. Sinon, notre machine entre dans une boucle infinie. □

Théorème 2.10.4. *Le langage $\{\star, \mathtt{u}\}^* \setminus \mathcal{A}$ n'est pas acceptable.*

Preuve. Sinon, par le théorème 2.10.3, les langages $\mathcal{A}$ et $\{\star, \mathtt{u}\}^* \setminus \mathcal{A}$ seraient tous les deux acceptables et par la proposition 2.9.6, le langage $\mathcal{A}$ serait décidable, contredisant le théorème 2.10.1. □

2.11 Le théorème de Rice

Le théorème suivant nous dit que le problème de l'arrêt est loin d'être un cas isolé : en effet, n'importe quelle propriété non triviale des langages acceptables est non décidable ! L'idée de la démonstration est de montrer que si une telle propriété était décidable, alors tout langage acceptable serait décidable, ce qu'on sait ne pas être le cas.

Dans ce qui suit, on qualifie une partie A d'un ensemble B de propre lorsque A n'est ni vide ni égale à B.

Théorème 2.11.1 (Rice)**.** *Pour toute partie propre S de* RE*, le langage*

$$\mathcal{A}_S = \{c(M) : M \textit{ est une machine de Turing telle que } L(M) \in S\}$$

est indécidable.

Preuve. Par souci de clarté, nous découpons la preuve en plusieurs parties.

1. Tout d'abord, remarquons que quitte à travailler avec $\mathrm{RE} \setminus S$ plutôt qu'avec S, on peut supposer que $\varnothing \notin S$. En effet, sinon on aurait $\varnothing \notin \mathrm{RE} \setminus S$. Or, d'une part, $\mathrm{RE} \setminus S$ est une partie propre de RE et, d'autre part, $\mathcal{A}_{\mathrm{RE}\setminus S}$ est décidable si et seulement si $\mathcal{A}_S$ est décidable puisque

$$\begin{aligned}\mathcal{A}_{\mathrm{RE}\setminus S} &= \{c(M) : M \text{ est une machine de Turing telle que } L(M) \in \mathrm{RE} \setminus S\} \\ &= \{c(M) : M \text{ est une machine de Turing}\} \setminus \mathcal{A}_S.\end{aligned}$$

2. Soit L un langage acceptable écrit sur un alphabet A. Il existe une machine de Turing $\mathcal{M}_L$ qui, pour tout mot $w \in A^*$, atteint la configuration d'arrêt $h.\underline{\#}$ à partir de $q_0.\#w\underline{\#}$ lorsque $w \in L$ et boucle indéfiniment sinon.

3. Puisque S est non vide par hypothèse, il existe une machine de Turing $\mathcal{M}$ telle que $L(\mathcal{M}) \in S$. Pour tout mot $w \in A^*$, on considère une machine de Turing $\mathcal{B}_w$ qui à partir de

la configuration $q_0.\underline{\#}$ atteint la configuration d'arrêt $h.\#w\underline{\#}$. Soit maintenant une machine de Turing $\mathcal{B}$ qui, pour tout mot $w \in A^*$, atteint la configuration d'arrêt $h.\#c(\mathcal{B}_w\, \mathcal{M}_L\, \mathcal{M})\underline{\#}$ à partir de la configuration $q_0.\#w\underline{\#}$.

4. On a

$$L(\mathcal{B}_w\, \mathcal{M}_L\, \mathcal{M}) = \begin{cases} L(M) & \text{si } w \in L \\ \varnothing & \text{sinon.} \end{cases}$$

Puisque $\varnothing \notin S$ et $L(M) \in S$, on obtient que $L(\mathcal{B}_w\, \mathcal{M}_L\, \mathcal{M}) \in S$ si et seulement $w \in L$.

5. Montrons à présent que si le langage $\mathcal{A}_S$ est décidable, alors le langage L aussi. En effet, au vu des deux points précédents, si $\mathcal{A}_S$ est décidé par une machine de Turing $\mathcal{D}$, alors la machine de Turing $\mathcal{BD}$ décide le langage L.

6. Nous déduisons du point précédent que $\mathcal{A}_S$ est indécidable puisque le langage L est un langage acceptable arbitraire et qu'il existe des langages acceptables non décidables (par exemple le langage du problème de l'arrêt). □

Voici quelques exemples d'applications du théorème de Rice qu'on pourra réaliser en exercice.

Exercice.

1. Adapter la preuve du théorème de Rice pour montrer que si S est une partie propre de l'ensemble $\mathcal{C}$ des fonctions numériques calculables, le problème de savoir si une machine de Turing donnée calcule une fonction de S est indécidable. Autrement dit, le langage

$$\{c(\mathcal{M}) : \mathcal{M} \text{ est une machine de Turing calculant une fonction de } S\}$$

   est indécidable.
2. Le problème de savoir si le langage accepté par une machine de Turing est non vide est indécidable. Autrement dit, le langage

$$\{c(\mathcal{M}) : \mathcal{M} \text{ est une machine de Turing telle que } L(\mathcal{M}) \neq \varnothing\}$$

   est indécidable.
3. Le problème de savoir si deux machines de Turing acceptent le même langage est indécidable. Autrement dit, le langage

$$\{c(\mathcal{M})c(\mathcal{N}) : \mathcal{M} \text{ et } \mathcal{N} \text{ sont des machines de Turing telles que } L(\mathcal{M}) = L(\mathcal{N})\}$$

   est indécidable.

2.12 Variantes des machines de Turing

Dans cette section, nous présentons quelques variations de la définition de machine de Turing. Nous allons voir que ces définitions alternatives apparemment plus générales ne permettent en fait pas de calculer plus de fonctions que les machines de Turing dites standards. Il s'agit là d'un nouvel argument en faveur de la thèse de Church-Turing.

2.12.1 Machines de Turing à ruban bi-infini

La définition d'une *machine de Turing à ruban bi-infini* est identique à celle d'une machine de Turing standard, à ceci près que le ruban mémoire est maintenant un mot bi-infini sur l'alphabet de la machine.

Formellement, une *configuration mémoire* est un couple $(w, k) \in A^{\mathbb{Z}} \times \mathbb{Z}$, où le mot bi-infini w ne contient qu'un nombre fini de fois le symbole blanc #. La *partie significative de la mémoire* est de la forme $u\underline{a}v$ où a est la cellule référencée, la première lettre de u n'est pas # et la dernière lettre de v n'est pas #.

Soient $A_1, \ldots, A_{d+1}$ des alphabets ne contenant pas #. Une fonction $f\colon A_1^* \times \cdots \times A_d^* \to A_{d+1}^*$ est *calculable par une machine de Turing à ruban bi-infini* $\mathcal{M} = (Q, q_0, h, B, \delta)$ si $\bigcup_{i=1}^{d+1} A_i \subseteq B$ et si pour tout $(w_1, \ldots, w_d) \in A_1^* \times \cdots \times A_d^*$, on a

$$q_0.w_1\# \cdots \#w_d\underline{\#} \vdash^* h.f(w_1, \ldots, w_d)\underline{\#}.$$

Les définitions de langages décidables et acceptables sont adaptées de manière évidente.

Théorème 2.12.1. *Les machines de Turing à ruban bi-infini calculent les mêmes fonctions que les machines de Turing standards. Les langages décidés/acceptés par machines de Turing à ruban bi-infini coïncident avec les langages décidés/acceptés par machines de Turing standards.*

Preuve. Tout d'abord, observons qu'une machine de Turing standard est une machine de Turing à ruban bi-infini particulière : il s'agit d'une machine de Turing à ruban bi-infini qui n'utilise que la partie droite de sa mémoire.

Intéressons-nous à l'autre direction. Soit une machine de Turing à ruban bi-infini $\mathcal{B}$. On construira d'abord une machine de Turing standard $\mathcal{S}$ réalisant l'action

$$q_0.\#w\underline{\#} \vdash^* h.\#c(\mathcal{B})c(w)\underline{\#}.$$

Ensuite, on construira une machine de Turing universelle $\mathcal{U}$ (standard) comme dans le théorème 2.9.4. On construira finalement une machine de Turing standard $\mathcal{S}'$ réalisant l'action

$$q_0.\#c(\mathcal{B})\rho(q.u\underline{a}v)\underline{\#} \vdash^* h.\#u\underline{a}v.$$

Ainsi, si $\mathcal{B}$ calcule une fonction de $\mathcal{F}_1$, la machine de Turing standard $\mathcal{S}\,\mathcal{U}\,\mathcal{S}'$ calcule la même fonction. (L'encodage devra être adapté pour le calcul des fonctions de $\mathcal{F}_d$ où $d \geqslant 0$ est arbitraire.) En particulier, si $\mathcal{B}$ décide un langage L, la machine $\mathcal{S}\,\mathcal{U}\,\mathcal{S}'$ décide le même langage. De plus, on a $L(\mathcal{S}\,\mathcal{U}) = L(\mathcal{B})$. □

Une autre variante est la suivante. On considère cette fois que la fonction de transition est de la forme $\delta\colon Q\backslash\{h\} \times A \to Q \times A \times \{\mathtt{L}, \mathtt{R}, \mathtt{S}\}$. Autrement dit, si la machine de Turing se trouve dans un état q et si la cellule référencée contient la lettre a, la machine de Turing peut écrire un nouveau symbole dans la cellule référencée et effectuer un déplacement à gauche (instruction `L`) ou à droite (instruction `S`), ou encore ne pas effectuer de déplacement (instruction `S`, pour « stationnaire »). Il n'est pas difficile de voir qu'une telle machine ne calcule pas plus de fonctions qu'une machine de Turing standard (nous ne détaillerons pas).

Exemple 2.12.2 (Castor affairé)**.** La *fonction du castor affairé* est la fonction $\mathrm{BB}\colon \mathbb{N} \to \mathbb{N}$, $n \mapsto \mathrm{BB}(n)$ où $\mathrm{BB}(n)$ est le nombre maximum de `u` consécutifs qu'on peut écrire sur le ruban mémoire d'une machine de Turing à ruban bi-infini, d'alphabet $\{\#, \mathtt{u}\}$ et ayant $n + 1$ états en partant de la configuration $q_0.\underline{\#}$ et dont la fonction de transition est du type $\delta\colon Q\backslash\{h\} \times A \to Q \times A \times \{\mathtt{L}, \mathtt{R}\}$. Une telle machine ne peut donc rester stationnaire. La notation BB vient de l'anglais « Busy Beaver ». On peut montrer que la fonction BB n'est pas calculable. En fait, il s'agit d'un des premiers exemples de fonction non calculable. La fonction β que nous avons vue précédemment est une variante de la fonction BB pour s'adapter au machines de Turing standards. Une machine de Turing réalisant la valeur $\mathrm{BB}(n)$ est appelé un castor affairé. Des castors affairés pour $n = 2$ et $n = 3$ sont représentées aux figures 2.21 et 2.22. On sait aujourd'hui que $(\mathrm{BB}(1), \mathrm{BB}(2), \mathrm{BB}(3), \mathrm{BB}(4)) = (1, 4, 6, 13)$ mais on ne connaît pas les valeurs de $\mathrm{BB}(n)$ pour $n \geqslant 5$. On peut démontrer que $\mathrm{BB}(5) \geqslant 4098$ et que $\mathrm{BB}(6) \geqslant 10^{1439}$.

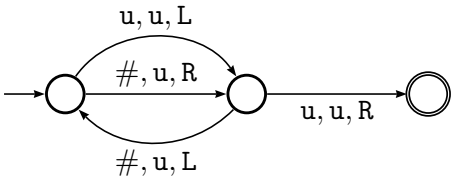


FIGURE 2.21 – Castor affairé atteignant la valeur BB(2) = 4.

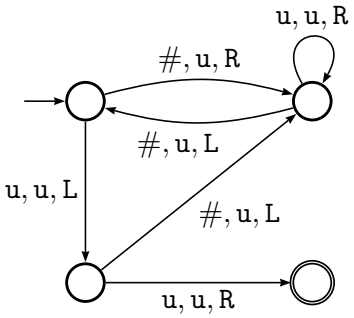


FIGURE 2.22 – Castor affairé atteignant la valeur BB(3) = 6.

2.12.2 Machines de Turing à plusieurs bandes

Une *machine de Turing à r bandes* est définie comme une machine de Turing standard à la différence près que la mémoire est constituée de r rubans mémoire. Une *configuration mémoire* est un $2r$-uple $(w_1, k_1, w_2, k_2, \ldots, w_r, k_r) \in (A^*\#^\omega \times \mathbb{N})^r$. La *partie significative de la mémoire* est de la forme $u_1\underline{a_1}v_1 \,.\, u_2\underline{a_2}v_2 \,.\, \cdots \,.\, u_r\underline{a_r}v_r$. La fonction de transition d'une telle machine est de la forme $\delta\colon Q\backslash\{h\} \times A^r \to Q \times (A \cup \{\mathtt{L}, \mathtt{R}\})^r$.

Seule la première bande contient les informations nécessaires au calcul d'une fonction à la configuration initiale et la configuration d'arrêt. Les bandes supplémentaires ne servent qu'à aider dans la réalisation du calcul. Formellement, une fonction $f\colon A_1^* \times \cdots \times A_d^* \to A_{d+1}^*$ (avec les conventions habituelles sur les alphabets) est *calculable par une machine de Turing à r bandes* si pour tout $(w_1, \ldots, w_d) \in A_1^* \times \cdots \times A_d^*$, on a

$$q_0.\#w_1\#\cdots\#w_d\underline{\#} \,.\, \underbrace{\underline{\#} \,.\, \cdots \,.\, \underline{\#}}_{r-1 \text{ fois}} \vdash^* h.\#f(w_1, \ldots, w_d)\underline{\#} \,.\, \underbrace{\underline{\#} \,.\, \cdots \,.\, \underline{\#}}_{r-1 \text{ fois}}.$$

À nouveau, les définitions de langages décidables et acceptables s'adaptent de façon naturelle.

Le graphe d'une machine de Turing à r bandes est adapté comme suit : pour tous $p, q \in Q$, $a_1, \ldots, a_r \in A$ et $x_1, \ldots, x_r \in A \cup \{\mathtt{L}, \mathtt{R}\}$ tels que $\delta(p, a_1, \ldots, a_r) = (q, x_1, \ldots, x_r)$, on dessine un arc de p vers q étiqueté par $(a_1, \ldots, a_r), (x_1, \ldots, x_r)$.

Exemple 2.12.3. Notre but est de programmer $\mathcal{S}_{R,d}$ au moyen d'une machine de Turing à 2 bandes. On va réaliser l'action

$$q_0.x\#w_1\#\cdots\#w_d\underline{\#}.\underline{\#} \vdash^* h.x\#\#w_1\#\cdots\#w_d\underline{\#}.\underline{\#}.$$

On commence par construire une machine à 2 bandes $\mathcal{A}$ réalisant

$$q_0.\#w\#.\underline{\#} \vdash^* h.\underline{\#}.\#w^R\underline{\#}.$$

La machine de Turing de la figure 2.23 convient. Soient maintenant $\mathcal{R}^{(1)}$ une machine à 2

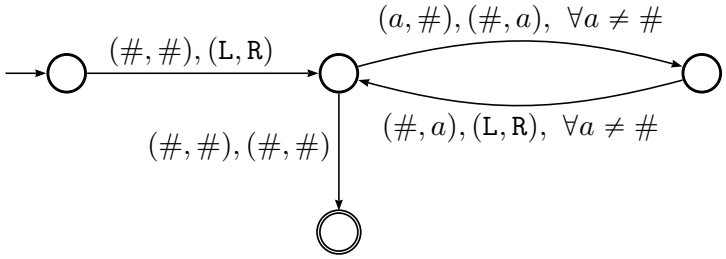


FIGURE 2.23 – Machine de Turing à 2 bandes qui au départ d'un mot w placé sur sa première bande, d'arrêt avec le mot miroir w^R sur sa deuxième bande.

bandes qui déplace sa tête de lecture à droite sur sa première bande et $\mathcal{B}$ une machine à 2 bandes qui réalise

$$q_0.\underline{\#}.\#w\#. \vdash^* h.\#w^R\underline{\#}.\underline{\#}.$$

(De telles machines sont faciles à construire.) Alors la machine de Turing à 2 bandes $\mathcal{A}^d\,\mathcal{R}^{(1)}\,\mathcal{B}^d$ convient.

Exercice.

1. Programmer $\mathcal{S}_{L,d}, \mathcal{C}_d, \mathcal{E}_d$ au moyen de machines de Turing à 2 bandes.
2. Programmer la fonction d'Ackermann au moyen d'une machine de Turing à r bandes, avec r au choix.

À nouveau, nous montrons que les machines de Turing à plusieurs bandes ne sont pas plus puissantes que les machines de Turing standard en termes de calculabilité.

Théorème 2.12.4. *Les machines de Turing à plusieurs bandes calculent les mêmes fonctions que les machines de Turing standards. Les langages décidés/acceptés par machines de Turing à plusieurs bandes coïncident avec les langages décidés/acceptés par machines de Turing standards.*

Preuve. Toute machine de Turing standard est une machine de Turing à plusieurs bandes qui n'utilise que la première bande de sa mémoire.

Intéressons-nous à l'autre direction. Soit une machine de Turing à r bande $\mathcal{R}$. Pour utiliser les machines universelles dans ce contexte, nous devons d'abord adapter les codages de ces machines. On peut par exemple coder la fonction de transition d'une machine à r bandes en concaténant les codes des $2(r+1)$-uples

$$(p, a_1, a_2, \ldots, a_r, q, x_1, x_2, \ldots, x_r)$$

tels que $\delta(p, a_1, a_2, \ldots, a_r) = (q, x_1, x_2, \ldots, x_r)$. On prendra soin de placer un nombre adéquat de séparateurs $\star$. Une fois que le choix d'un codage $c(\mathcal{M})$ des machines de Turing à r bandes est posé, on construira une machine de Turing standard $\mathcal{S}$ réalisant l'action

$$q_0.\#w\underline{\#} \vdash^* h.\#c(\mathcal{R})\rho(q_0.\#w\underline{\#}) \star \underbrace{\rho(\underline{\#}) \star \cdots \star \rho(\underline{\#})}_{r-1 \text{ fois}} \underline{\#}$$

et une machine de Turing universelle standard $\mathcal{U}$ simulant l'exécution d'une machine de Turing à r bandes (il faut adapter l'énoncé du théorème 2.9.4 aux machines à plusieurs bandes). On conclut comme pour le théorème 2.12.1. □

2.12.3 Machines de Turing non déterministes

Une machine de Turing non déterministe est définie comme une machine de Turing standard à ceci près que δ est maintenant une relation de transition.

Afin de se représenter les possibles actions d'une machine de Turing non déterministe au départ d'une configuration mémoire donnée, on construit l'arbre des transitions de la machine dont le sommet est la configuration d'intérêt.

Exemple 2.12.5 (Générateur aléatoire de nombre)**.** La machine de Turing de la figure 2.24 engendre aléatoirement un naturel n : partant de $q_0.\underline{\#}$, on aboutit à une configuration d'arrêt $h.\#\,\mathtt{u}^n\,\underline{\#}$ pour un certain $n \in \mathbb{N}$.

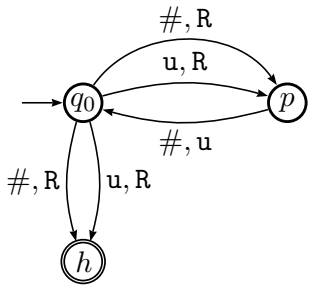


FIGURE 2.24 – Générateur aléatoire de nombres.

La relation de transition est donnée par l'ensemble de quadruplets

$$\delta = \{(q_0, \mathtt{u}, p, \mathtt{R}), (q_0, \#, p, \mathtt{R}), (q_0, \mathtt{u}, h, \mathtt{R}), (q_0, \#, h, \mathtt{R}), (p, \#, a_0, \mathtt{u})\}.$$

L'arbre des transitions du générateur aléatoire de nombres de la figure 2.24 au départ de la configuration $q_0.\underline{\#}$ est représenté à la figure 2.25.

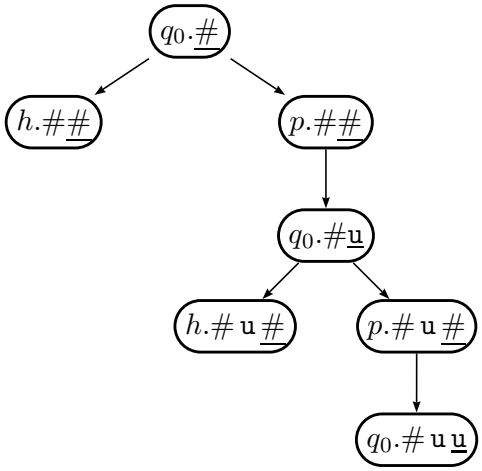


FIGURE 2.25 – Premiers niveaux de l'arbre des transitions du générateur aléatoire de nombres au départ de la configuration $q_0.\underline{\#}$.

Dans l'arbre des transitions, certaines branches sont finies, d'autres pas. Une branche finie mène soit dans une configuration d'arrêt soit dans une configuration pendante. Un mot w est *accepté par une machine de Turing non déterministe* s'il existe une branche dans l'arbre des transitions de sommet $q_0.\#w\underline{\#}$ qui se termine dans une configuration d'arrêt.

Remarquons qu'on ne peut pas utiliser les machines de Turing non déterministes pour calculer des fonctions ou pour décider des langages. En effet, au départ d'une configuration donnée, la machine peut atteindre plusieurs configurations d'arrêt différentes. Les machines de Turing non déterministes sont utilisées uniquement comme des accepteurs.

Théorème 2.12.6. *Les machines de Turing non déterministes acceptent les mêmes langages que les machines de Turing standards.*

Preuve. Les machines de Turing standards sont des machines de Turing non déterministes particulières : celles dont la relation de transition est en fait une fonction.

Intéressons-nous à l'autre direction. Soit une machine de Turing non déterministe $\mathcal{N}$. Nous décrivons une procédure qui s'arrête au départ d'une entrée w si et seulement si $\mathcal{N}$ accepte w. L'idée est de passer en revue l'arbre des transitions de $\mathcal{N}$ de sommet $q_0.\#w\underline{\#}$ niveau par niveau. À chaque étape, on teste si $\mathcal{N}$ a atteint une configuration d'arrêt. Si oui, on s'arrête. Si non, on continue. Dans le cas où toutes les branches sont finies sans atteindre de configuration d'arrêt (c'est-à-dire mènent à des configurations pendantes), on boucle indéfiniment. De cette manière, on ne s'arrête pas si et seulement si aucune branche de l'arbre ne mène à une configuration d'arrêt. □

Remarquons que dans la preuve précédente, on aurait pu tout aussi bien commencer par éliminer les configurations pendantes. Ceci peut se faire de la même manière que pour les machines de Turing standards, encore une fois en utilisant des codages adéquats.

Exercice. Décrire une machine de Turing universelle standard exécutant l'algorithme de la preuve.

Chapitre 3

Complexité

3.1 Complexité temporelle des machines de Turing

Définition 3.1.1. Soient $\mathcal{M}$ une machine de Turing non déterministe (potentiellement à plusieurs bandes) d'alphabet A et w un mot sur $A\backslash\{\#\}$. La *durée d'exécution de $\mathcal{M}$ sur w*, notée $d_{\mathcal{M}}(w)$, vaut 0 si $w \notin L(\mathcal{M})$ et vaut le nombre minimum de transitions permettant d'atteindre une configuration d'arrêt à partir de $q_0.\#w\underline{\#}$ en respectant les transitions de $\mathcal{M}$. La *fonction de complexité (temporelle) de $\mathcal{M}$* est la fonction

$$T_{\mathcal{M}} : \mathbb{N} \to \mathbb{N},\ n \mapsto \sup\{d_{\mathcal{M}}(w) : w \in (A\backslash\{\#\})^n\}$$

Remarque. Il s'agit de la complexité temporelle dans le pire cas. Il est aussi souvent pertinent d'étudier d'autres complexités temporelles, par exemple la complexité temporelle moyenne. Dans ce cas, on a besoin de connaître la distribution des données en fonction de leur durée d'exécution. Cette question peut s'avérer très difficile en pratique. De façon générale, les deux approches (pire cas et moyenne) sont complémentaires.

Exemple 3.1.2. La complexité temporelle de la machine de Turing représentée à la Figure 2.1 est $2n^2 + 8n + 6$.

Définition 3.1.3. Une machine de Turing est *polynomiale* si sa fonction de complexité est majorée par un polynôme, ou de manière équivalente, si $T_{\mathcal{M}}(n) \in O(n^k)$ pour un certain $k \in \mathbb{N}$. Une fonction est *P-calculable* s'il existe une machine de Turing déterministe polynomiale qui la calcule.

Exercice.

1. Montrer que tout polynôme encodé en unaire est P-calculable.
2. Montrer que la fonction $n \mapsto 2^n$ n'est pas P-calculable. Envisager le codage unaire et le codage binaire.
3. Montrer que l'égalité de deux entiers encodés en unaire est calculable par une machine de Turing standard en temps $O(n^2)$.
4. Montrer que l'égalité de deux entiers encodés en unaire est calculable par une machine de Turing à 2 bandes en temps $O(n)$.
5. Montrer que la composition de fonctions P-calculables est P-calculable.
6. Montrer que toute fonction $(A^*)^d \to B^*$ calculable par une machine de Turing à 2 bandes de complexité $T(n) \geqslant n$ est calculable par une machine de Turing standard de complexité en $O((T(n))^2)$.

3.2 Transformations polynomiales

Définition 3.2.1. Soient $K \subseteq A^*$ et $L \subseteq B^*$. Une application $f\colon A^* \to B^*$ est une *transformation polynomiale* de K vers L si les deux conditions suivantes sont satisfaites :

1. f est P-calculable
2. $\forall w \in A^*,\ w \in K \iff f(w) \in L$.

On écrit $K \leqslant L$ pour exprimer qu'il existe une transformation polynomiale de K vers L.

Remarque 3.2.2.

— La deuxième condition de la définition peut se réexprimer par $K = f^{-1}(L)$.
— La relation $\leqslant$ est transitive : si $K \leqslant L$ et $L \leqslant M$, alors $K \leqslant M$. Elle est aussi réflexive puisque l'identité est une transformation polynomiale d'un langage vers lui-même.
— Pour toute transformation polynomiale f de K vers L, on a $\chi_K = \chi_L \circ f$.

Proposition 3.2.3. *Lorsque $K \leqslant L$, on a que K est décidable/acceptable si L l'est.*

Preuve. Supposons que f est une transformation polynomiale de K vers L. En particulier, il existe une machine de Turing $\mathcal{F}$ qui calcule f. Si $\mathcal{M}$ est une machine de Turing qui calcule χ_L (resp. accepte L), alors la machine de Turing $\mathcal{F}\mathcal{M}$ calcule χ_K (resp. accepte K). □

3.3 Problèmes de décision

Problème du voyageur de commerce

Les instances du problème sont n villes, numérotées $1, \dots, n$, une matrice des distances entre ces villes $D = (D_{ij})_{1 \leqslant i,j \leqslant n} \in \mathbb{N}^{n \times n}$ avec comme condition que $D_{ij} > 0$ si $i \neq j$, et enfin un nombre b représentant la distance maximale autorisée. Une instance du problème est donc la donnée du couple (D, b). Le problème est alors le suivant : existe-t-il un circuit passant exactement une fois par chaque ville dont la longueur totale ne dépasse par b ? Autrement dit, le problème est de déterminer s'il existe une permutation $\nu \in \mathcal{S}_n$ telle que

$$D_{\nu_1\nu_2} + D_{\nu_2\nu_3} + \cdots + D_{\nu_{n-1}\nu_n} + D_{\nu_n\nu_1} \leqslant b.$$

Ce problème est noté TS (pour travelling saleman).

Afin de se ramener à la définition des langages décidables, nous devons convenir d'un codage des instances (ou les données) du problème considéré par des mots. Par exemple, dans le cas du problème du voyageur de commerce, on peut coder les instances du problème par le mot

$$\mathrm{rep}_2(D_{11}) \star \cdots \star \mathrm{rep}_2(D_{1n}) \star \cdots \star \mathrm{rep}_2(D_{n1}) \star \cdots \star \mathrm{rep}_2(D_{nn}) \star \mathrm{rep}_2(b).$$

Définition 3.3.1. Une *instance positive* (resp. *négative*) est une instance pour laquelle la réponse au problème est « oui » (resp. « non »).

À chaque codage du problème est associé un langage, qui est le langage des instances positives. Ce langage est écrit sur l'alphabet utilisé pour le codage des instances. Remarquons qu'un mot quelconque sur cet alphabet n'est pas nécessairement le codage d'une instance du problème. Dans le cas où il l'est, il est soit le codage d'une instance positive, soit le codage d'une instance négative. Ces trois cas de figure sont à considérer séparément.

Définition 3.3.2. Un problème est *décidable* si le langage des instances positives associé l'est.

On peut s'étonner d'une telle définition puisqu'elle dépend a priori du codage choisi. Il convient donc de prendre quelques précautions sur les codages autorisés afin d'assurer l'indépendance de la définition par rapport au choix du codage. Il suffit de s'assurer que le codage de notre problème soit effectif (tout comme nous en avions déjà eu le souci dans la proposition 2.9.3.

Proposition 3.3.3. TS *est décidable.*

Preuve. Il existe un nombre fini de permutations de n villes. Il suffit donc de les tester toutes l'une après l'autre, et de vérifier si oui ou non la somme des distances correspondantes vaut au plus b. □

Problème du circuit hamiltonien

Une instance du problème est un graphe non orienté $G = (V, E)$, où on convient que $V = \{1, \ldots, n\}$. Le problème est de déterminer si G possède un circuit hamiltonien. Si on convient qu'une instance du problème est donnée par la matrice $A = (A_{ij})_{1 \leqslant i,j \leqslant n} \in \mathbb{N}^{n \times n}$, où n est le nombre de sommets du graphe, définie par

$$A_{ij} = \begin{cases} 1 & \text{si } (i,j) \in E \\ 0 & \text{sinon,} \end{cases}$$

alors le problème revient à déterminer s'il existe une permutation $\nu \in \mathcal{S}_n$ telle que

$$A_{\nu_1\nu_2} = A_{\nu_2\nu_3} = \cdots = A_{\nu_{n-1}\nu_n} = A_{\nu_n\nu_1} = 1.$$

Ce problème est noté HC (pour Hamitonian circuit).

Proposition 3.3.4. HC *est décidable.*

Preuve. Il existe un nombre fini de permutations des sommets d'un graphe. Pour chacune d'entre elle, on peut tester si la permutation définit un circuit du graphe. □

Proposition 3.3.5. HC $\leqslant$ TS.

Preuve. Soit f la fonction $A \mapsto (D^A, n)$ où

$$(D^A)_{ij} = \begin{cases} 1 & \text{si } A_{ij} = 1 \\ 2 & \text{si } A_{ij} = 0. \end{cases}$$

et n est la dimension de la matrice A. Cette fonction est clairement P-calculable. Montrons que si A est une instance positive de HC, alors $f(A) = (D^A, n)$ est une instance positive de TS. Soit A une instance positive de HC. Alors il existe une permutation $\nu \in \mathcal{S}_n$ telle que

$$A_{\nu_1\nu_2} = A_{\nu_2\nu_3} = \cdots = A_{\nu_{n-1}\nu_n} = A_{\nu_n\nu_1} = 1.$$

Par définition de D^A, on a

$$(D^A)_{\nu_1\nu_2} + (D^A)_{\nu_2\nu_3} + \cdots + (D^A)_{\nu_{n-1}\nu_n} + (D^A)_{\nu_n\nu_1} = n$$

et (D^A, n) est une instance positive de TS.

Il nous reste à montrer que réciproquement, si $f(A)$ est une instance positive de TS, alors A est une instance positive de HC. Supposons que (D^A, n) soit une instance positive de TS. Alors il existe une permutation $\nu \in \mathcal{S}_n$ telle que

$$(D^A)_{\nu_1\nu_2} + (D^A)_{\nu_2\nu_3} + \cdots + (D^A)_{\nu_{n-1}\nu_n} + (D^A)_{\nu_n\nu_1} \leqslant n.$$

Par définition de D^A, tous les termes de la sommes sont égaux à 1, et donc on a

$$A_{\nu_1\nu_2} = A_{\nu_2\nu_3} = \cdots = A_{\nu_{n-1}\nu_n} = A_{\nu_n\nu_1} = 1.$$

Ainsi, A est une instance positive de HC. □

Problème du pavage

Une instance du problème du pavage est la donnée

- d'un ensemble fini de tuiles T
- d'une tuile initiale $i \in T$
- d'un ensemble de règles de juxtaposition horizontales $H \subseteq T \times T$
- et d'un ensemble de règles de juxtaposition verticales $V \subseteq T \times T$.

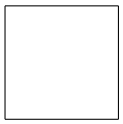
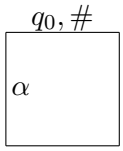
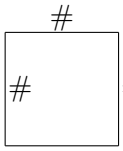
La question est de déterminer s'il existe une façon de disposer les tuiles de T partout dans $\mathbb{N}^2$ en respectant H et V et telle que la tuile posée en $(0,0)$ soit la tuile initiale i. Autrement dit, on demande de déterminer s'il existe une fonction $f\colon \mathbb{N}^2 \to T$ telle que

- $f(0,0) = i$
- $\forall m, n \in \mathbb{N}$, $(f(m,n), f(m+1,n)) \in H$
- $\forall m, n \in \mathbb{N}$, $(f(m,n), f(m,n+1)) \in V$.

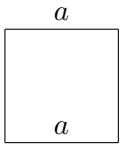
Théorème 3.3.6. *Le problème du pavage est indécidable.*

Preuve. L'idée de la preuve est de se ramener à l'indécidabilité du problème de l'arrêt. Pour ce faire, à toute machine de Turing $\mathcal{M}$, on associe une instance du problème du pavage $(T_\mathcal{M}, i_\mathcal{M}, H_\mathcal{M}, V_\mathcal{M})$ de la façon suivante. Si $\mathcal{M} = (Q, q_0, h, A, \delta)$ est une machine de Turing sans configuration pendante [1], alors l'ensemble $T_\mathcal{M}$ est constitué des tuiles suivantes :

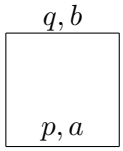
- en considérant un nouveau symbole spécial α :

$i_\alpha =$  α $\quad i_\mathcal{M} =$  $\#$ $\quad$ et $\quad t_B =$  $\#$

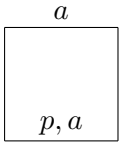
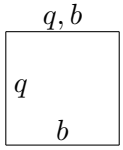
- pour chaque $a \in A \cup \{\alpha\}$:

$t_a =$ 

- pour chaque $p, q \in Q\backslash\{h\}$ et $a, b \in A$ tels que $\delta(p,a) = (q,b)$:

$t_{p,a,q,b} =$ 

- pour chaque $p, q \in Q\backslash\{h\}$ et $a \in A$ tels que $\delta(p,a) = (q,R)$ et pour chaque $b \in A$:

$t_{p,a,q,R} =$  q $\quad$ et $\quad R_{q,b} =$ 

1. Ce n'est pas une vraie restriction au vu de la section 2.9.

- pour chaque $p, q \in Q\backslash\{h\}$ et $a \in A$ tel que $\delta(p, a) = (q, L)$ et chaque $b \in A$:

$$t_{p,a,q,L} = \begin{array}{|c|}\hline a \\ q \quad\quad \\ p, a \\ \hline\end{array} \quad \text{et} \quad L_{q,b} = \begin{array}{|c|}\hline q, b \\ \quad\quad q \\ b \\ \hline\end{array}$$

La tuile initiale est la tuile i_α. L'ensemble $H_\mathcal{M} \subseteq T_\mathcal{M} \times T_\mathcal{M}$ est formé par les couples de tuiles (t_1, t_2) telles que le bord droit de t_1 contient la même information que le bord gauche de t_2. L'ensemble $V_\mathcal{M} \subseteq T_\mathcal{M} \times T_\mathcal{M}$ est formé par les couples de tuiles (t_1, t_2) telles que le bord supérieur de t_1 contient la même information que le bord inférieur de t_2.

Supposons qu'on puisse décider du problème du pavage. Alors on pourrait décider s'il existe une façon de disposer les tuiles de $T_\mathcal{M}$ en respectant les règles imposées par le choix de $i_\mathcal{M}$, $H_\mathcal{M}$ et $V_\mathcal{M}$. Or, un tel pavage de $\mathbb{N}^2$ existe si et seulement si la machine de Turing $\mathcal{M}$ ne s'arrête pas à partir de $q_0.\underline{\#}$. On pourrait donc décider si une machine de Turing s'arrête à partir de $q_0.\underline{\#}$, ce que nous savons être impossible. □

Exemple 3.3.7. Considérons la machine de Turing $\mathcal{M}$ donnée à la figure 3.1. Afin d'illus-

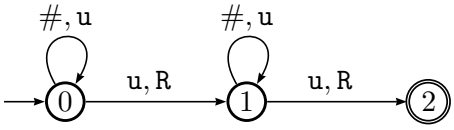


FIGURE 3.1 – Une machine de Turing $\mathcal{M}$.

trer la preuve du théorème 3.3.6, nous donnons les tuiles constituant l'ensemble $T_\mathcal{M}$ correspondant. Il s'agit des onze tuiles suivantes.

$$i_\alpha = \begin{array}{|c|c|}\hline \alpha & \\ \hline \quad\quad & \alpha \\ \hline\end{array} \qquad i_\mathcal{M} = \begin{array}{|c|c|}\hline 0, \# & \\ \hline \alpha \quad\quad & \# \\ \hline\end{array} \qquad t_B = \begin{array}{|c|c|}\hline \# & \\ \hline \# \quad\quad & \# \\ \hline\end{array} \qquad t_\alpha = \begin{array}{|c|}\hline \alpha \\ \quad\quad \\ \alpha \\ \hline\end{array}$$

$$t_\# = \begin{array}{|c|}\hline \# \\ \quad\quad \\ \# \\ \hline\end{array} \qquad t_\mathtt{u} = \begin{array}{|c|}\hline \mathtt{u} \\ \quad\quad \\ \mathtt{u} \\ \hline\end{array} \qquad t_{0,\#,0,\mathtt{u}} = \begin{array}{|c|}\hline 0, \mathtt{u} \\ \quad\quad \\ 0, \# \\ \hline\end{array} \qquad t_{1,\#,1,\mathtt{u}} = \begin{array}{|c|}\hline 1, \mathtt{u} \\ \quad\quad \\ 1, \# \\ \hline\end{array}$$

$$t_{0,\mathtt{u},1,R} = \begin{array}{|c|c|}\hline \mathtt{u} & \\ \hline \quad\quad & 1 \\ \hline 0, \mathtt{u} & \\ \hline\end{array} \qquad R_{1,\#} = \begin{array}{|c|c|}\hline 1, \# & \\ \hline 1 \quad\quad & \\ \hline \# & \\ \hline\end{array} \qquad R_{1,\mathtt{u}} = \begin{array}{|c|c|}\hline 1, \mathtt{u} & \\ \hline 1 \quad\quad & \\ \hline \mathtt{u} & \\ \hline\end{array}$$

On commence à paver le quart de plan en plaçant les tuiles ligne par ligne. Toutes les tuiles placées sont forcées par les règles de juxtaposition et il n'est possible de placer aucune tuile en position $(1, 4)$. Ce pavage partiel est représenté à la figure 3.2

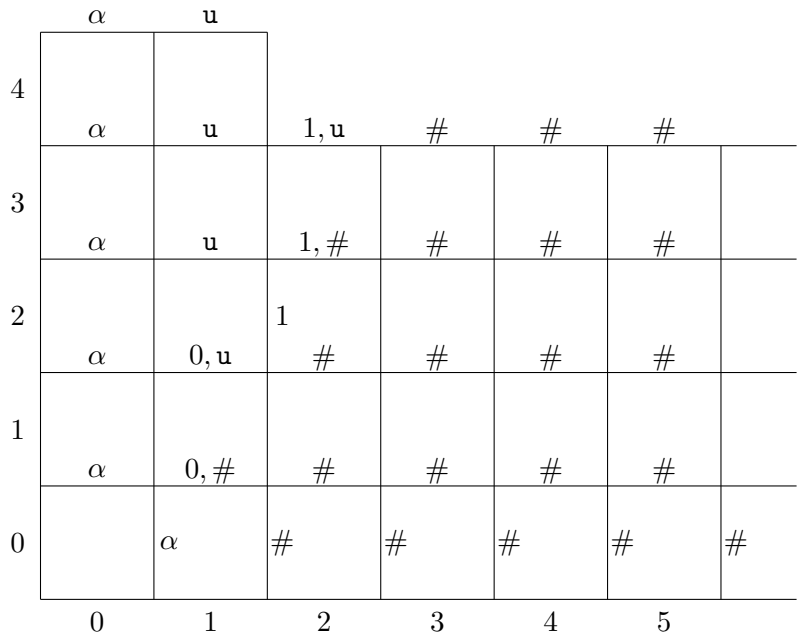


FIGURE 3.2 – Pavage partiel correspondant à l'exécution de $\mathcal{M}$ au départ de $0.\underline{\#}$.

3.4 Les classes P, NP et NPC

Définition 3.4.1. On note P la classe des langages décidés par une machine de Turing déterministe polynomiale. On note NP la classe des langages acceptés par une machine de Turing non déterministe polynomiale.

Remarque. Clairement, on a $\mathrm{P} \subseteq \mathrm{NP}$. Le problème de savoir si $\mathrm{P} = \mathrm{NP}$ est l'un des sept problèmes mathématiques du millénaire ![2] Ce problème a été posé par Stephen Cook.

Proposition 3.4.2. *Si $L \in \mathrm{NP}$, alors L est décidable.*

Preuve. Soit $L \in \mathrm{NP}$ et soit $\mathcal{M}$ une machine de Turing non déterministe polynomiale qui accepte L. Soit Q un polynôme tel que pour tout $n \in \mathbb{N}$, on ait $T_{\mathcal{M}}(n) \leqslant Q(n)$. Si $w \in L$, alors il existe un chemin d'acceptation de w dans $\mathcal{M}$ de longueur au plus $Q(|w|)$. Pour décider de l'appartenance d'un mot w à L, une machine de Turing déterministe D va calculer $Q(|w|)$ et simuler l'exécution de M sur $q_0.\#w\underline{\#}$ en parcourant les $Q(|w|)$ premiers niveaux de l'arbre des transitions de M. La machine répond « oui » si elle a trouvé une configuration d'arrêt parmi ces niveaux et répond « non » sinon. □

Remarquez que la machine de Turing $\mathcal{D}$ de la preuve précédente n'est pas en général pas polynomiale !

Proposition 3.4.3. *Lorsque $K \leqslant L$, on a que $K \in \mathrm{P}$ (resp. NP) si $L \in \mathrm{P}$ (resp. NP).*

Preuve. La preuve est similaire à celle de la proposition 3.2.3. Soit $\mathcal{F}$ une machine de Turing déterministe polynomiale qui calcule une transformation polynomiale de K vers L. Si $\mathcal{M}$ est une machine de Turing déterministe (resp. non déterministe) polynomiale qui calcule χ_L (resp. qui accepte L), alors la machine de Turing $\mathcal{F}\mathcal{M}$ est déterministe (resp. non déterministe) polynomiale et calcule χ_K (resp. accepte K). □

Définition 3.4.4. Soient $K \subseteq A^*$ et $L \subseteq B^*$. On note $K \equiv L$ si $K \leqslant L$ et $L \leqslant K$. Dans ce cas, on dit que K et L sont *P-équivalents*.

2. https://en.wikipedia.org/wiki/Millennium_drize_droblems

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

Proposition 3.4.5. *Les ensembles suivants sont des classes d'équivalence pour la relation $\equiv$.*

1. $\{\varnothing\}$
2. $\{A^* : A \text{ est un alphabet}\}$
3. $\mathrm{P}\backslash\big(\{A^* : A \text{ est un alphabet}\} \cup \{\varnothing\}\big)$
4. $\{L \in \mathrm{NP} : \forall K \in \mathrm{NP},\ K \leqslant L\}$.

Preuve.

1. Il suffit de montrer que $L \leqslant \varnothing \implies L = \varnothing$. Soient A et B des alphabets et soit $L \subseteq A^*$, le langage vide étant vu comme un sous-ensemble de B^*. Supposons que $L \leqslant \varnothing$, c'est-à-dire qu'il existe une transformation polynomiale $f\colon A^* \to B^*$ de L vers $\varnothing$. Par définition, pour tout $w \in A^*$, on a $w \in L \iff f(w) \in \varnothing$. D'où $L = \varnothing$.
2. Montrons que pour tous alphabets A et B, on a $A^* \leqslant B^*$. Soit $f\colon A^* \to B^*,\ w \mapsto \varepsilon$. Cette fonction est P-calculable et pour tout $w \in A^*$, on a $w \in A^* \iff \varepsilon \in B^*$. Il s'agit donc d'une transformation polynomiale de A^* vers B^*.

   De plus, si $f\colon A^* \to B^*$ est une transformation polynomiale de L vers B^*, alors $L = A^*$. En effet, pour tout $w \in A^*$, on a $w \in L \iff f(w) \in B^*$. On en déduit que $L = A^*$.
3. Soient $K, L \in \mathrm{P}\backslash\big(\{A^* : A \text{ est un alphabet}\} \cup \{\varnothing\}\big)$. Montrons que $K \leqslant L$. Soient A et B des alphabets tels que $K \subseteq A^*$ et $L \subseteq B^*$ et soient $u \in L$ et $v \in B^*\backslash L$. Alors la fonction
$$f\colon A^* \to B^*,\ w \mapsto \begin{cases} u & \text{si } w \in K \\ v & \text{si } w \notin K \end{cases}$$
   est une transformation polynomiale de K dans L.

   Soit à présent $L \in \mathrm{P}\backslash\big(\{A^* : A \text{ est un alphabet}\} \cup \{\varnothing\}\big)$ et soit $K \equiv L$. Par les points 1 et 2, nous savons que $K \notin \{A^* : A \text{ est un alphabet}\} \cup \{\varnothing\}$. De plus, $K \in \mathrm{P}$ par la Proposition 3.4.3.
4. Pour tous $L, M \in \mathrm{NP}$ tels que pour tout $K \in \mathrm{NP}$, on a $K \leqslant L$ et $K \leqslant M$, on a clairement $L \equiv M$.

   Soit à présent $L \in \mathrm{NP}$ tel que $K \leqslant L$ pour tout $K \in \mathrm{NP}$ et soit $M \equiv L$. Comme $L \leqslant M$, on obtient par transitivité de $\leqslant$ que $K \leqslant M$ pour tout $K \in \mathrm{NP}$. Comme $M \leqslant L$, on obtient que $M \in \mathrm{NP}$ par la Proposition 3.4.3.

□

Définition 3.4.6. La dernière classe d'équivalence de la proposition 3.4.5 est notée NPC :
$$\mathrm{NPC} = \{L \in \mathrm{NP} : \forall K \in \mathrm{NP}, K \leqslant L\}.$$
Les langages appartenant à NPC sont appelés les *langages* NP-*complets*. Un langage L est dit NP-*difficile* si pour tout $K \in \mathrm{NP}$ on a $K \leqslant L$.

Ainsi, pour montrer qu'un langage L est NP-complet, on doit montrer que $L \in \mathrm{NP}$ et que L est NP-difficile. Ces deux étapes sont indépendantes. En général, la première étape consiste à trouver un certificat succinct de L (voir définition ci-après) et la deuxième étape en une réduction d'un problème qu'on sait déjà être NP-difficile vers L.

Définition 3.4.7. Soit L un langage sur un alphabet A. Un *certificat succinct* de L est la donnée d'un langage $D \in \mathrm{P}$, écrit sur un alphabet B contenant A et un symbole spécial $\star \notin A$, et d'un polynôme Q tels que
$$L = \{w \in A^* : \exists x \in (B\backslash\{\star\})^{\leqslant Q(|w|)} \text{ tel que } w \star x \in D\}.$$

L'idée du certificat succinct est que pour chaque instance positive, il existe une preuve « courte » que cette instance est en effet positive. Par exemple, considérons le problème de déterminer si un nombre naturel donné est composé. Si Pierre se trouve devant l'instance 476677, pour décider que ce nombre est composé, Pierre va passer en revue les diviseurs possibles, ce qui lui prendra tout de même un peu de temps. Par contre, si Laura affirme à Pierre que 476677 est un nombre composé, et que pour le lui prouver, elle lui fournit ses diviseurs premiers 179 et 2663, il suffit à Pierre de vérifier que le produit de ces nombres vaut effectivement 476677 pour être d'accord avec Laura.

Théorème 3.4.8. *La classe* NP *est la famille des langages pour lesquels il existe un certificat succinct.*

Preuve. Soit un langage $L \subseteq A^*$ et supposons que celui-ci possède un certificat succinct donné par D et Q (nous utilisons les notations de la définition 3.4.7). On construit une machine de Turing non déterministe $\mathcal{N}$ qui à partir d'une entrée $w \in A^*$ se comporte de la façon suivante : $\mathcal{N}$ produit un mot $x \in (B\backslash\{\star\})^*$ de façon non déterministe, ensuite elle vérifie au moyen d'une machine de Turing déterministe polynomiale si la longueur de x est au plus $Q(|w|)$ et si le mot $w \star x$ appartient à D. Si c'est le cas, elle s'arrête, et dans le cas contraire, elle entre dans une boucle infinie. Ainsi, la machine de Turing $\mathcal{N}$ est polynomiale et telle que $L = L(\mathcal{N})$.

Inversement, soit $L \in \mathrm{NP} \cap A^*$. Soit une machine de Turing non déterministe polynomiale $\mathcal{N}$ qui accepte L. Notons k le nombre maximum de transitions possibles à partir d'un état de $\mathcal{N}$. En supposant que les sommets atteints depuis chaque sommet des arbres des transitions sont ordonnées de 1 à k, on considère qu'un mot x sur l'alphabet $\{1, 2, \ldots, k\}$ décrit une branche de l'arbre des transitions de $\mathcal{N}$ (on peut supposer que tout sommet a k fils, quitte à ajouter des transitions inutiles). Soit à présent D le langage des mots de la forme $w \star x$, avec $w \in A^*$, $x \in \{1, 2, \ldots, k\}^*$ et $\star \notin A \cup \{1, 2, \ldots, k\}$, où la suite de transitions décrite par x correspond à un chemin d'acceptation de w dans $\mathcal{N}$. Clairement, on a $D \in \mathrm{P}$ (détaillez). Comme la machine de Turing $\mathcal{N}$ est polynomiale, on peut trouver un polynôme Q tel que $T_{\mathcal{N}} \leqslant Q$. Ainsi, si $w \in L$, alors il est accepté par $\mathcal{N}$ en au plus $Q(|w|)$ transitions. Un mot w appartient à L si et seulement s'il existe un mot x de longueur au plus $Q(|w|)$ tel que $w \star x \in D$. La donnée de D et de Q constitue donc un certificat succinct de L. □

Pour illustrer la preuve précédente, une machine de Turing non déterministe produisant les mots de $\{0, 1\}^*$ en temps linéaire est représentée à la figure 3.3 et les premiers niveaux de son arbre des transitions à partir de la configuration $0.\underline{\#}$ sont représentés à la figure 3.4.

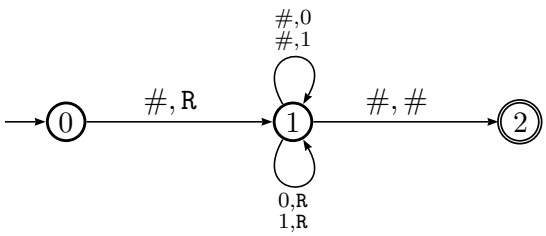


FIGURE 3.3 – Générateur aléatoire de mots sur $\{0, 1\}$.

Tout comme nous avons parlé de *problèmes décidables*, nous souhaiterions également pouvoir parler de *problèmes* dans P et *problèmes* dans NP, sans avoir à se soucier du codage utilisé. En termes de complexité, nous devons être un peu précautionneux et opter pour des codages dis *raisonnables*. On respectera les trois principes suivants :

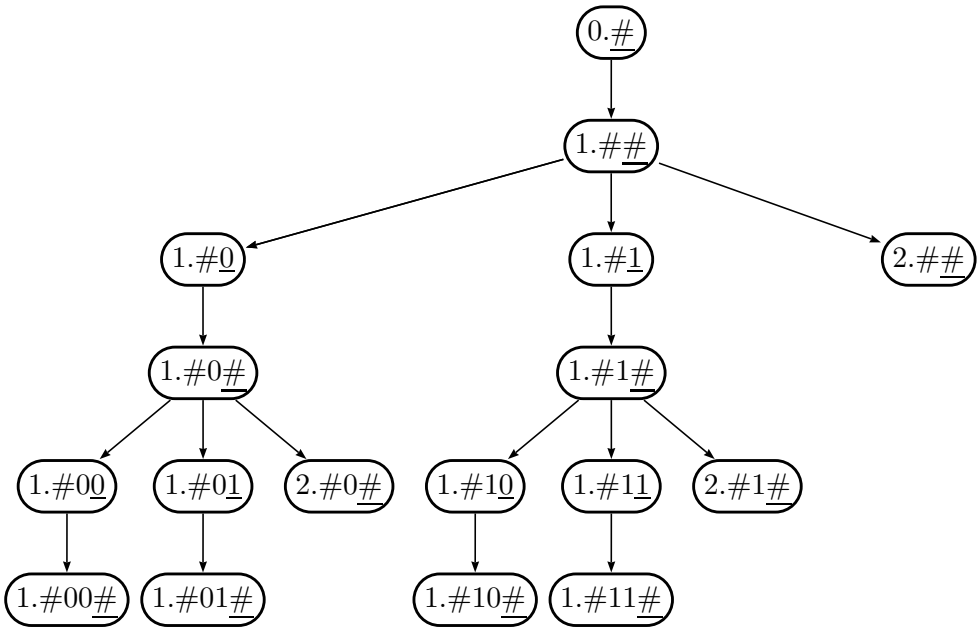


FIGURE 3.4 – Premiers niveaux de l'arbre des transitions du générateur aléatoire de mots sur $\{0,1\}$ au départ de la configuration $0.\underline{\#}$.

1. Le codage choisi ne peut pas contenir de bourrage, c'est-à-dire un grand nombre de symboles inutiles.
2. Le codage des entiers est fait en base entière $b \geqslant 2$ (donc, jamais en unaire).
3. Le décodage doit pouvoir s'effectuer en temps polynomial.

On admettra qu'en respectant ces principes, le passage d'un codage à un autre se fait toujours en temps polynomial, de sorte que le choix du codage n'a pas d'importance pour définir un problème dans P, NP ou NPC.

3.5 Le théorème de Cook

Définition 3.5.1. Une formule de la logique propositionnelle est dite *satisfaisable* s'il existe une distribution des valeurs de vérité des variables qui la composent qui la rend vraie. Une *clause* est une disjonction de variables propositionnelles ou de négation de variables propositionnelles. Une *forme normale conjonctive (FNC)* est une conjonction de clauses. La *longueur d'une clause* est le nombre de variables ou négations de variables propositionnelles qui la composent. La *longueur d'une FNC* est la somme des longueurs des clauses qui la composent.

Exemple 3.5.2. La formule $x \vee y \vee \neg z$ est une clause de longueur 3. La formule $(x \vee y \vee \neg z) \wedge (x \vee z \vee t \vee \neg u)$ est une FNC. Sa longueur est la somme des longueur de ses clauses, c'est-à-dire 7.

Le problème SAT est le suivant : étant donné une FNC, décider si elle est satisfaisable.

Théorème 3.5.3 (Cook)**.** SAT $\in$ NPC.

Preuve. Il est facile de voir que SAT possède un certificat succinct. En effet, étant donné une FNC et une distribution des valeurs de vérités des variables, on peut vérifier en temps linéaire si la FNC est vraie ou fausse pour cette distribution.

Montrons que SAT est NP-difficile. Soit donc $L \in \mathrm{NP} \cap A^*$ et soit une machine de Turing $\mathcal{M} = (Q, q_0, h, A', \delta)$ non déterministe polynomiale qui accepte L, où l'alphabet A' de la machine contient l'alphabet A du langage L. Sans perte de généralité, on peut supposer que $\mathcal{M}$ n'atteint pas de configuration pendante. Notre but est de montrer que $L \leqslant \mathrm{SAT}$. Nous allons donc construire une transformation polynomiale de L vers SAT. À tout mot $w \in A^*$, nous allons associer une FNC $\phi(w)$ de longueur majorée par un polynôme en $|w|$ et telle que $w \in L$ si et seulement si $\phi(w)$ est satisfaisable.

Soit P un polynôme majorant la complexité de $\mathcal{M}$. Si un mot w est accepté par $\mathcal{M}$, alors il l'est en au plus $P(|w|)$ transitions. Donc pour accepter un mot w, la machine passe par au plus $P(|w|) + 1$ configurations en utilisant au plus $P(|w|) + |w| + 2$ cellules de sa mémoire. Par commodité, quitte à remplacer P par $P(n) + n + 2$, on supposera que pour accepter un mot w, la machine passe par au plus $P(|w|)$ configurations en utilisant au plus $P(|w|)$ cellules de sa mémoire. On supposera même que tout chemin d'acceptation d'un mot w passe par exactement $P(|w|)$ configurations, qu'on convient de numéroter par $i = 1, \ldots, P(|w|)$. Il suffit de considérer que si une configuration d'arrêt est rencontrée en moins de $P(|w|)$ configurations successives, $\mathcal{M}$ subit des transitions supplémentaires de la forme (h, a, h, a) pour un total de $P(|w|)$ configurations.

De même, si r est le nombre maximal de transitions définies dans $\mathcal{M}$ à partir de tout couple $(p, a) \in Q \times A'$, on supposera qu'il existe exactement r transitions depuis chaque couple $(p, a) \in Q \times A'$. S'il y a j transitions au départ d'un couple (p, a) avec $j < r$, il suffit de considérer que $\mathcal{M}$ possède $r - j$ transitions (p, a, p, a) supplémentaires. Cela revient à compléter l'arbre des transitions de $\mathcal{M}$ au départ de $q_0.\#w\underline{\#}$ avec des branches inutiles pour avoir un arbre complet de $P(|w|)$ niveaux.

Avec ces conventions, nous pouvons maintenant définir les variables propositionnelles de notre formule $\phi(w)$. Pour chaque couple (p, a), on convient d'une numérotation des transitions sortantes de 1 à r. Nous distinguons 4 types de variables :

- $C_{i,j,a}$ pour $1 \leqslant i, j \leqslant P(|w|)$ et $a \in A'$. La valeur de vérité 1 pour $C_{i,j,a}$ traduit le fait que à la configuration i, le contenu de la cellule j est a.
- $R_{i,j}$ pour $1 \leqslant i, j \leqslant P(|w|)$. La valeur de vérité 1 pour $R_{i,j}$ traduit le fait que à la configuration i, la cellule référencée est la cellule j.
- $S_{i,q}$ pour $1 \leqslant i \leqslant P(|w|)$ et $q \in Q$. La valeur de vérité 1 pour $S_{i,q}$ traduit le fait que à la configuration i, on se trouve dans l'état q.
- $T_{i,k}$ pour $1 \leqslant i < P(|w|)$ et $1 \leqslant k \leqslant r$. La valeur de vérité 1 pour $T_{i,k}$ traduit le fait que depuis la configuration i, on opte pour la transition k parmi les r transitions possibles.

Nous définissons plusieurs FNC.

1. La FNC

   $$S_{1,q_0} \wedge R_{1,|w|+2} \wedge C_{1,1,\#} \wedge C_{1,2,w[1]} \wedge \cdots \wedge C_{1,|w|+1,w[|w|]} \wedge C_{1,|w|+2,\#} \wedge \cdots \wedge C_{1,P(|w|),\#}$$

   traduit le fait que la configuration initiale est $q_0.\#w\underline{\#}$. Sa longueur est en $O\big(P(|w|)\big)$.
2. La FNC

   $$\bigwedge_{1 \leqslant i \leqslant P(|w|)} \left(\left(\bigvee_{1 \leqslant j \leqslant P(|w|)} R_{i,j} \right) \wedge \left(\bigwedge_{1 \leqslant j < j' \leqslant P(|w|)} (\neg R_{i,j} \vee \neg R_{i,j'}) \right) \right)$$

   traduit le fait que pour toute configuration i, exactement une cellule j est référencée. Sa longueur est en $O\big((P(|w|))^3\big)$. En effet, de façon générale, la FNC

   $$(x_1 \vee \cdots \vee x_d) \ \wedge \ \left(\bigwedge_{1 \leqslant i < j \leqslant d} (\neg x_i \vee \neg x_j) \right)$$

est de longueur d^2 et est vraie si et seulement exactement une des variables x_i est vraie.

3. En convenant d'un ordre sur l'alphabet A, la FNC

$$\bigwedge_{1\leqslant i,j\leqslant P(|w|)} \left(\left(\bigvee_{a\in A'} C_{i,j,a} \right) \wedge \left(\bigwedge_{\substack{a,a'\in A \\ a<a'}} (\neg C_{i,j,a} \vee \neg C_{i,j,a'}) \right) \right)$$

traduit le fait que pour toute configuration i, toute cellule j contient exactement une lettre de A'. Sa longueur est en $O\big((P(|w|))^2\big)$.

4. En convenant d'un ordre sur l'ensemble des états Q, la FNC

$$\bigwedge_{1\leqslant i\leqslant P(|w|)} \left(\left(\bigvee_{q\in Q} S_{i,q} \right) \wedge \left(\bigwedge_{\substack{q,q'\in Q \\ q<q'}} (\neg S_{i,q} \vee \neg S_{i,q'}) \right) \right)$$

traduit le fait que pour toute configuration i, la machine de Turing $\mathcal{M}$ se trouve dans exactement un état de Q. Sa longueur est en $O\big(P(|w|)\big)$.

5. La FNC

$$\bigwedge_{1\leqslant i<P(|w|)} \left(\left(\bigvee_{1\leqslant k\leqslant r} T_{i,k} \right) \wedge \left(\bigwedge_{1\leqslant k<k'\leqslant r} (\neg T_{i,k} \vee \neg T_{i,k'}) \right) \right)$$

traduit le fait que pour toute configuration i, la machine de Turing $\mathcal{M}$ opte pour exactement une transition parmi les r transitions possibles. Sa longueur est en $O\big(P(|w|)\big)$.

6. La FNC

$$\bigwedge_{\substack{1\leqslant i<P(|w|) \\ 1\leqslant j\leqslant P(|w|) \\ a\in A'}} (R_{i,j} \vee \neg C_{i,j,a} \vee C_{i+1,j,a})$$

traduit le fait que pour tout i, toute cellule non référencée à la configuration i ne change pas de contenu à la configuration $i+1$. En effet, les propositions

$$(\neg R_{i,j} \wedge C_{i,j,a}) \implies C_{i+1,j,a} \quad \text{et} \quad R_{i,j} \vee \neg C_{i,j,a} \vee C_{i+1,j,a}$$

sont logiquement équivalentes. Sa longueur est en $O\big((P(|w|))^2\big)$.

7. Considérons maintenant la formule

$$\bigwedge_{\substack{1\leqslant i<P(|w|) \\ 1\leqslant j\leqslant P(|w|) \\ p\in Q \\ a\in A' \\ 1\leqslant k\leqslant r}} \Big((S_{i,p} \wedge R_{i,j} \wedge C_{i,j,a} \wedge T_{i,k}) \implies (S_{i+1,q} \wedge R_{i+1,j+d} \wedge C_{i+1,j,b}) \Big)$$

où les données p, a, k déterminent les valeurs de q, b, d de la façon suivante : si (p, a, q, x) est la k^{e} transition possible à partir de (p, a), alors on pose

$$b = \begin{cases} x & \text{si } x \in A' \\ a & \text{sinon} \end{cases} \quad \text{et} \quad d = \begin{cases} -1 & \text{si } x = \texttt{L} \\ 1 & \text{si } x = \texttt{R} \\ 0 & \text{sinon.} \end{cases}$$

Cette formule traduit que les transitions de la machine de Turing $\mathcal{M}$ sont respectées lors du passage d'une configuration à la suivante. La formule est équivalente à la FNC

$$\bigwedge_{\substack{1\leqslant i<P(|w|)\\ 1\leqslant j\leqslant P(|w|)\\ 1\leqslant k\leqslant r\\ p\in Q\\ a\in A'}} \Big((\neg S_{i,p} \vee \neg R_{i,j} \vee \neg C_{i,j,a} \vee \neg T_{i,k} \vee S_{i+1,q}) \\ \wedge (\neg S_{i,p} \vee \neg R_{i,j} \vee \neg C_{i,j,a} \vee \neg T_{i,k} \vee R_{i+1,j+d}) \\ \wedge (\neg S_{i,p} \vee \neg R_{i,j} \vee \neg C_{i,j,a} \vee \neg T_{i,k} \vee C_{i+1,j,b}) \Big)$$

dont la longueur est en $O\big((P(|w|))^2\big)$.

8. Enfin, la FNC

$$S_{P(|w|),h}$$

traduit que la dernière configuration est une configuration d'arrêt. Sa longueur est en $O(1)$.

La FNC $\phi(w)$ est la conjonction de toutes les FNC précédentes. Sa longueur est en $O\big((P(|w|))^3\big)$. Par construction, à chaque chemin d'acceptation de w correspond une distribution des valeurs de vérité de la formule $\phi(w)$ qui la rend vraie, et réciproquement, à chaque distribution des valeurs de vérité de la formule $\phi(w)$ qui la rend vraie correspond un chemin d'acceptation de w dans $\mathcal{M}$ (détaillez). Ainsi, w est accepté par $\mathcal{M}$ si et seulement si la formule $\phi(w)$ est satisfaisable. Nous pouvons donc conclure que $L \leqslant \mathrm{SAT}$ puisque la fonction $w \mapsto \phi(w)$ est une transformation polynomiale de L vers SAT. □

Corollaire 3.5.4. $\mathrm{P} = \mathrm{NP} \iff \mathrm{NPC} \cap \mathrm{P} \neq \varnothing$.

Preuve. Supposons d'abord qu'il existe $K \in \mathrm{NPC} \cap \mathrm{P}$. Montrons qu'alors $\mathrm{NP} \subseteq \mathrm{P}$. Soit $L \in \mathrm{NP}$. Puisque $K \in \mathrm{NPC}$, on a $L \leqslant K$. Mais puisque $K \in \mathrm{P}$, la Proposition 3.4.3 implique que $L \in \mathrm{P}$.

Supposons à présent que $\mathrm{P} = \mathrm{NP}$. Soit $L \in \mathrm{NPC}$ (un tel L existe vu le théorème de Cook). En particulier $L \in \mathrm{NP}$, donc $L \in \mathrm{P}$ vu notre hypothèse. D'où $\mathrm{NPC} \cap \mathrm{P} \neq \varnothing$. □

3.6 Catalogue de problèmes NP-complets

Définition 3.6.1 (3 SAT)**.** Étant donné une FNC comportant uniquement des clauses de longueur 3, décider si elle est satisfaisable.

Théorème 3.6.2. $3\,\mathrm{SAT} \in \mathrm{NPC}$.

Preuve. Il suffit de montrer que $3\,\mathrm{SAT} \equiv \mathrm{SAT}$. On a clairement que $3\,\mathrm{SAT} \leqslant \mathrm{SAT}$. Montrons qu'on a aussi $\mathrm{SAT} \leqslant 3\,\mathrm{SAT}$. À toute FNC ϕ, nous allons associer une FNC $f(\phi)$ comportant uniquement des clauses de longueur 3, dont la longueur est proportionnelle à celle de ϕ et telle que ϕ est satisfaisable si et seulement si $f(\phi)$ l'est aussi.

Considérons d'abord le cas où ϕ est une clause. Dans les formules suivantes, les notations $\alpha, \beta, \gamma, \alpha_1, \ldots, \alpha_m$ représentent des variables propositionnelles ou des négations de variables propositionnelles.

- Si $\phi \equiv \alpha$, alors on définit $f(\phi) \equiv \alpha \vee \alpha \vee \alpha$.
- Si $\phi \equiv \alpha \vee \beta$, alors on définit $f(\phi) \equiv \alpha \vee \beta \vee \alpha$.
- Si $\phi \equiv \alpha \vee \beta \vee \gamma$, alors on définit $f(\phi) \equiv \phi$.
- Si $\phi \equiv \alpha_1 \vee \alpha_2 \vee \cdots \vee \alpha_m$ avec $m \geqslant 4$, alors on définit

$$f(\phi) \;\equiv\; (\alpha_1 \vee \alpha_2 \vee y_1) \wedge \left(\bigwedge_{i=3}^{m-2} (\neg y_{i-2} \vee \alpha_i \vee y_{i-1}) \right) \wedge (\neg y_{m-3} \vee \alpha_{m-1} \vee \alpha_m)$$

où $y_1, \ldots, y_{m-3}$ sont des nouvelles variables propositionnelles.

Dans les trois premiers cas, il est clair que ϕ est vrai pour une distribution des valeurs de vérité de ses variables si et seulement si $f(\phi)$ est vrai pour la même distribution.

Dans le quatrième cas, montrons que pour toute distribution des valeurs de vérité des variables de ϕ, on a que ϕ est vrai si et seulement s'il existe une distribution des valeurs de vérité de $y_1, \ldots, y_{m-3}$ telle que $f(\phi)$ est vrai.

Considérons une distribution des valeurs de vérité des variables de ϕ rendant ϕ vrai. Alors il existe $i \in \{1, \ldots, m\}$ tel que α_i soit vrai pour cette distribution. Nous distinguons trois cas. Si $i \in \{1, 2\}$, alors la distribution $y_1 = \cdots = y_{m-3} = 0$ convient. Si $i \in \{m-1, m\}$, alors la distribution $y_1 = \cdots = y_{m-3} = 1$ convient. Si $i \in \{3, \ldots, m-2\}$, alors la distribution donnée par $y_1 = \cdots = y_{i-2} = 1$ et $y_{i-1} = \cdots = y_{m-3} = 0$ convient.

Inversement, soit une distribution des valeurs de vérité des variables de ϕ et de $y_1, \ldots, y_{m-3}$ telle que $f(\phi)$ soit vrai Nous devons montrer que ϕ est vrai pour cette même distribution. Supposons le contraire. Dans ce cas, tous les α_i sont faux pour la distribution choisie. En parcourant les clauses de $f(\phi)$ de gauche à droite jusqu'à l'avant-dernière, on obtient successivement que $y_1 = \cdots = y_{m-3} = 1$. Mais alors la dernière clause de $f(\phi)$ est fausse, et la conjonction $f(\phi)$ est fausse elle aussi, une contradiction.

Considérons maintenant le cas où ϕ est une conjonction de clauses.

- Si $\phi \equiv c_1 \wedge \cdots \wedge c_k$ où $k \geqslant 1$ et $c_1, \ldots, c_k$ sont des clauses, alors on définit $f(\phi) \equiv f(c_1) \wedge \cdots \wedge f(c_k)$, où les variables potentiellement introduites sont différentes pour chaque clause.

Montrons que ϕ est satisfaisable si et seulement si $f(\phi)$ l'est. Notons $x_1, \ldots, x_m$ les variables de ϕ et $x_1, \ldots, x_m, y_1 \ldots, y_n$ les variables de $f(\phi)$. Au vu de ce qui précède, nous obtenons successivement

$f(\phi)$ est satisfaisable
$\iff \exists$ distribution de $x_1, \ldots, x_m, y_1 \ldots, y_n$ telle que $f(\phi) = 1$
$\iff \exists$ distribution de $x_1, \ldots, x_m, y_1 \ldots, y_n$ telle que $f(c_1) = 1$ et $\ldots$ et $f(c_k) = 1$
$\iff \exists$ distribution de $x_1, \ldots, x_m$ telle que $c_1 = 1$ et $\ldots$ et $c_k = 1$
$\iff \exists$ distribution de $x_1, \ldots, x_m$ telle que $\phi = 1$
$\iff \phi$ est satisfaisable.

La fonction f que nous avons définie est une transformation polynomiale puisque pour toute FNC ϕ, on a $|f(\phi)| \leqslant 3|\phi|$ et $\phi \in \mathrm{SAT} \iff f(\phi) \in 3\,\mathrm{SAT}$. □

Définition 3.6.3 (VC)**.** Une couverture d'un graphe G est un ensemble S de sommets de G tel que toute arête de G a au moins une extrémité dans S. On suppose qu'une couverture ne contient pas de sommet isolé. Le problème de couverture de sommets, abrévié VC, est le suivant : étant donné un graphe G et un entier n, existe-t-il une couverture de G de taille n ?

Exemple 3.6.4. Considérons le graphe représenté figure 3.5. L'ensemble de sommets $\{1, 4, 5, 6\}$ en est une couverture de taille 4 et ce graphe ne possède pas de couverture avec moins de 4 sommets.

Théorème 3.6.5. VC $\in$ NPC.

Preuve. Il est facile de voir que VC $\in$ NP. En effet, étant donné un ensemble S de sommets d'un graphe G, on peut vérifier si S est une couverture de G en un temps linéaire par rapport aux nombres d'arêtes de G.

Au vu du théorème précédent, il suffit de montrer que $3\,\mathrm{SAT} \leqslant \mathrm{VC}$ pour obtenir que VC est NP-difficile. À chaque FNC ϕ dont toutes les clauses sont de longueur 3, nous allons

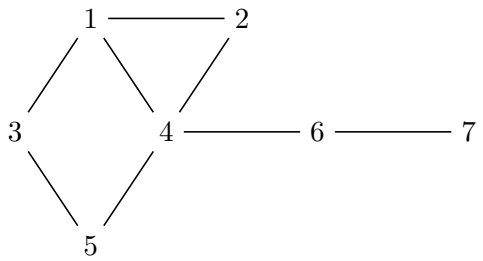


FIGURE 3.5 – Graphe A_i associé à la clause c_i.

associer un couple (G_ϕ, n_ϕ) tel que ϕ est satisfaisable si et seulement si le graphe G_ϕ possède une couverture à n_ϕ sommets. De plus, les tailles de G_ϕ et n_ϕ seront proportionnelles à la longueur de ϕ.

Soit $\phi \equiv c_1 \wedge \cdots \wedge c_k$ où chaque c_i est une clause de longueur 3. Soit ℓ le nombre de variables propositionnelles de ϕ. On définit le graphe G_ϕ de la façon suivante. À chaque clause $c_i \equiv \alpha_{i1} \vee \alpha_{i2} \vee \alpha_{i3}$ (où $\alpha_{i1}, \alpha_{i2}, \alpha_{i3}$ sont des variables propositionnelles ou des négations de variables propositionnelles), on associe le graphe A_i représenté à la figure 3.6. À chaque variable x_j de ϕ, on associe le graphe B_j représenté à la figure 3.7. Pour obtenir le

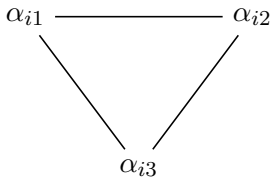


FIGURE 3.6 – Graphe associé A_i à la clause c_i.

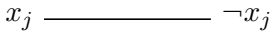


FIGURE 3.7 – Graphe associé B_j à la variable propositionnelle y_j.

graphe G_ϕ, on relie chacun des sommets des graphes A_i au seul sommet lui correspondant parmi les graphes B_j. Enfin, on pose $n_\phi = 2k + \ell$. Remarquons que $\ell \leqslant 3k$. On a donc $n_\phi \leqslant 5k = \frac{5}{3}|\phi|$. Si la taille d'un graphe $G = (V, E)$ est définie comme $|G| = |V| + |E|$, on a $|G_\phi| = (3k + 2\ell) + (6k + \ell) = 9k + 3\ell \leqslant 18k = 6|\phi|$.

Supposons d'abord que ϕ admet une distribution des valeurs de vérité de ses ℓ variables propositionnelles qui la rend vraie. Dans chaque graphe B_j, on sélectionne le sommet x_j si sa valeur est 1 et le sommet $\neg x_j$ sinon. Dans chaque graphe A_i, on sélectionne deux sommets de telle sorte que le sommet non sélectionné soit vrai. On a ainsi sélectionné $2k+\ell$ de G_ϕ. Montrons que l'ensemble de ces sommets est une couverture de G_ϕ. D'une part, les arêtes des graphes A_i et B_j sont trivialement couvertes. D'autre part, si un sommet d'un graphe A_i est non sélectionné, alors il est vrai et il est donc relié à un sommet d'un graphe B_j vrai également, qui doit donc avoir été sélectionné.

Inversement, supposons que S est une couverture de G_ϕ de taille $2k+\ell$. Toute couverture de G_ϕ doit contenir deux sommets de chaque graphe A_i et un sommet de chaque graphe B_j. Ainsi S possède exactement deux sommets de chaque graphe A_i et un sommet de chaque graphe B_j. À chaque variable x_j de ϕ, on attribue la valeur de vérité 1 si le sommet correspondant des graphes B_j appartient à S et 0 sinon. Montrons que cette distribution des valeurs de vérité de $x_1, \ldots, x_\ell$ rend ϕ vrai. Il suffit de montrer que chaque clause c_i de ϕ doit être vraie. Il suffit de montrer que pour chaque i, le sommet de A_i qui n'est pas

dans S est vrai. C'est bien le cas car l'arête qui le relie au sommet correspondant parmi les graphes B_j est couverte par S, donc le sommet correspondant doit être dans S, et donc être vrai par construction. □

Exemple 3.6.6. Nous illustrons la construction de G_ϕ de la preuve du théorème 3.6.5. Si

$$\phi \equiv (x \vee y \vee z) \wedge (\neg x \vee y \vee z) \tag{3.1}$$

alors le graphe G_ϕ correspondant est le graphe représenté à la figure 3.8. La distribution

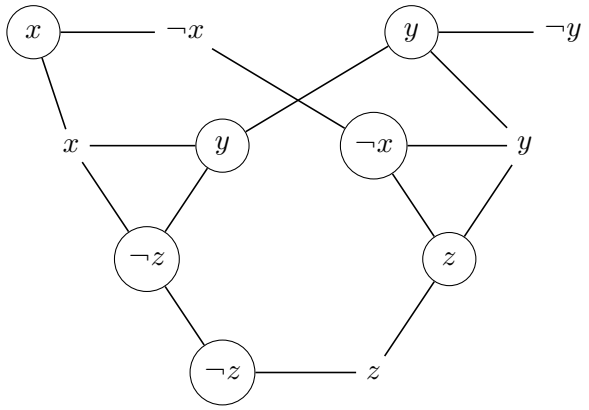


FIGURE 3.8 – Graphe associé à la FNC (3.1).

$(1, 1, 0)$ pour (x, y, z) rend ϕ vrai et la couverture de sommets donnée par les sommets entourés correspond à cette distribution. Remarquons que le choix des deux sommets dans le graphe A_2 correspondant à la clause $c_2 = (\neg x \vee y \vee z)$ est unique mais que tous les choix de deux sommets parmi les trois sommets du graphe A_1 correspondant à la clause $c_1 = (\neg x \vee y \vee z)$ étaient possibles. Il n'y a donc pas une unique couverture à $2k + \ell$ sommets associée à une distribution des valeurs de vérité des variables.

Théorème 3.6.7. HC ∈ NPC.

Preuve. Il est facile de voir que HC ∈ NP. En effet, étant donné une suite finie de sommets d'un graphe G, on peut vérifier si cette suite définit un circuit hamiltonien de G en un temps polynomial par rapport aux nombres d'arêtes de G.

Au vu du théorème précédent, il suffit de montrer que VC $\leqslant$ HC pour obtenir que HC est NP-difficile. À chaque couple (G, n), nous allons associer un graphe G' tel que G possède une couverture à n sommets si et seulement si G' possède un circuit hamiltonien. De plus, $|G'|$ sera proportionnel à $|G| + n$.

Voici les étapes de construction de G'.

1. À chaque arête $\{x, y\}$ de G, on associe le sous-graphe $A_{x,y}$ de G' à 12 sommets et 14 arêtes représenté à la figure 3.9. Dans un graphe $A_{x,y}$, on considère que 6 sommets d'une ligne correspondent au sommet x et que les 6 sommets de l'autre ligne correspondent au sommet y.
2. Pour chaque sommet x de G, s'il y a r arêtes adjacentes en x dans G, on convient d'un ordre $\{x, y_1\}, \ldots, \{x, y_r\}$ sur ces arêtes et on ajoute $r - 1$ arêtes reliant les sous-graphes $A_{x,y_1}, \ldots, A_{x,y_r}$ dans cet ordre : pour chaque $i \in \{1, \ldots, r - 1\}$, on relie une extrémité des 6 sommets associés à x dans A_{x,y_i} à une extrémité des 6 sommets associés à x dans $A_{x,y_{i+1}}$. Ceci est illustré à la figure 3.10.
3. Enfin, on ajoute n sommets $\alpha_1, \ldots, \alpha_n$ et pour chacun d'eux et chaque sommet x de G, on ajoute 2 arêtes reliant les extrémités des 6 sommets associés à x dans A_{x,y_1} et A_{x,y_r}.

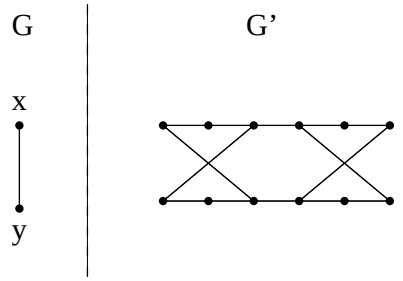


FIGURE 3.9 – Graphe $A_{x,y}$.

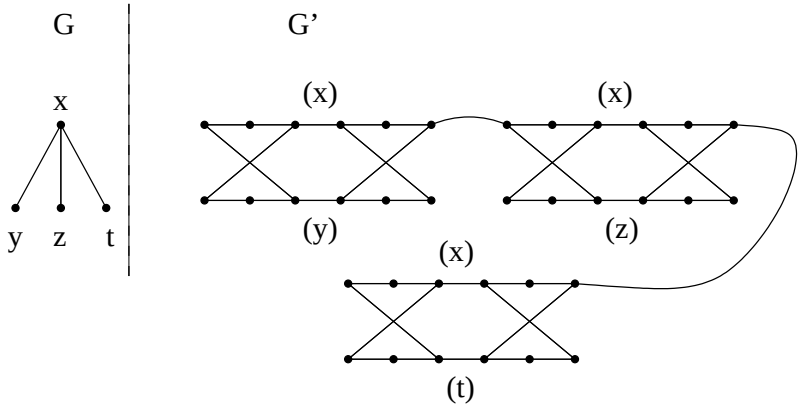


FIGURE 3.10 – Ajout de $r-1$ arêtes correspondant aux r arêtes adjacentes en un sommet.

Le graphe G' ainsi construit possède $12|E| + n$ sommets. Puisque chaque sommet de G possède au plus $|V| - 1$ arêtes adjacentes, le graphe G' possède au plus $14|E| + 2n|V| + |V|(|V| - 1)$ arêtes. Ainsi, la taille de G' est polynomiale en la taille $|V| + |E| + n$ des entrées.

Supposons d'abord que G possède une couverture S de n sommets. Décrivons un circuit hamiltonien de G'. On démarre de α_1. On choisit un sommet x de S et on se rend dans A_{x,y_1} via l'extrémité correspondant à x (celle-ci est reliée à tous les α_j par construction). On parcourt ce graphe en zigzag si $y_1 \notin S$ (premier cas de la figure 3.11) et en ligne droite sinon (deuxième cas de la figure 3.11). Dans les deux cas, on ressort du graphe A_{x,y_1} par l'autre extrémité correspondant au sommet x. On continue notre chemin en entrant dans A_{x,y_2} et on choisit les mêmes règles de parcours. De cette manière, on parcourt tous les graphes $A_{x,y_1}, \ldots, A_{x,y_r}$. On arrive donc à l'extrémité de cette suite de graphes correspondants aux arêtes adjacentes au sommet x de S et on se rend ensuite dans le sommet α_2. On recommence la même procédure : on choisit un nouveau sommet de la couverture et on parcourt d'une traite tous les graphes correspondants aux arêtes adjacentes en ce sommet avant d'entrer dans le sommet α_3. Quand tous les sommets de la couverture sont épuisés, on retourne dans le sommet α_1. On a ainsi défini un circuit de G'. Chaque graphe $A_{x,y}$ de G' aura été visité puisque, S étant une couverture de G, x ou y doit appartenir à S et les n sommets de la couverture ont été utilisés. De plus, vu nos choix de parcours de ces graphes, un graphe $A_{x,y}$ est visité exactement une fois si un seul des deux sommets x et y est dans S et un graphe $A_{x,y}$ est visité exactement deux fois si les deux sommets x, y sont dans S. Dans les deux cas, à nouveau grâce à nos choix de parcours, tous les sommets de $A_{x,y}$ auront été visités une et une seule fois. Les sommets $\alpha_1, \ldots, \alpha_n$ étant eux aussi visités une et une seule fois, le circuit décrit est hamiltonien.

Inversement, supposons que G' possède un circuit hamiltonien. Il passe donc par tous les sommets $\alpha_1, \ldots, \alpha_n$ une et une seule fois. Quitte à les renuméroter, on peut suppose qu'ils

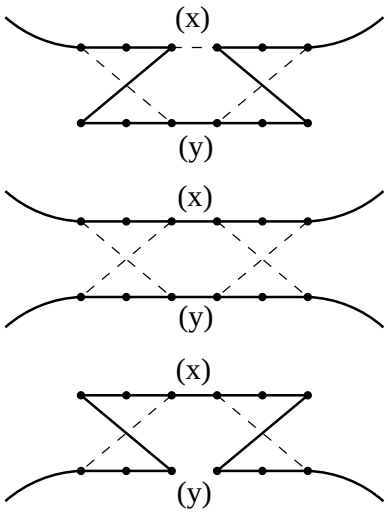


FIGURE 3.11 – Parcours d'un graphe $A_{x,y}$.

sont visités dans cet ordre. Par construction de G', entre deux sommets α_j et α_{j+1}, le circuit doit passer par au moins un graphe $A_{x,y}$. De plus, pour faire partie d'un circuit hamiltonien, le parcours d'un graphe $A_{x,y}$ est obligatoirement un de ceux représentés à la figure 3.11. En particulier, si le circuit entre dans tel graphe par une extrémité correspondant à un sommet, il doit en ressortir par l'autre extrémité correspondant à ce même sommet. Cela implique que le circuit doit nécessairement parcourir tous les graphes $A_{x,y}$ correspondant aux arêtes adjacentes en un sommet pour ensuite arriver dans le sommet α_{j+1}. Construisons à présent une couverture de sommets S de G à n sommets. En suivant le circuit hamiltonien, on place dans S les sommets correspondants aux extrémités des graphes $A_{x,y}$ connectées aux sommets $\alpha_1, \ldots, \alpha_n$. L'ensemble S possède ainsi n sommets. Pour chaque arête $\{x, y\}$ de G, le graphe $A_{x,y}$ de G' est visité par le circuit hamiltonien, donc au moins un des deux sommets x ou y a été placé dans S. Ceci montre que S est une couverture de G. □

Théorème 3.6.8. TS $\in$ NPC.

Preuve. Au vu de la proposition 3.3.5 et du théorème 3.6.7, nous savons déjà que TS est NP-difficile. On a aussi que TS $\in$ NP parce que, étant donné un graphe G, une borne b et une suite finie de sommets de G, on peut vérifier si cette suite définit un circuit hamiltonien de G dont la somme des distances est inférieur ou égale b en un temps polynomial par rapport à $|G| + b$. □

3.7 Autres classes de complexité

Dans cette dernière section, nous donnons les définitions de nouvelles classes de complexité et nous les comparons entre elles et avec les classes vues jusqu'ici.

Définition 3.7.1. On note

$$\text{co-NP} = \{L : L \subseteq A^*,\ A \text{ alphabet et } A^* \backslash L \in \text{NP}\}$$

et

$$\text{co-NPC} = \{L : L \subseteq A^*,\ A \text{ alphabet et } A^* \backslash L \in \text{NPC}\}.$$

Remarquons que les définitions des classes précédentes sont indépendantes de l'alphabet choisi : si $A \subseteq B$ et $L \subseteq A^*$, alors on a $A^* \backslash L \in \text{NP} \iff B^* \backslash L \in \text{NP}$ (resp. $A^* \backslash L \in$

NPC $\iff$ $B^*\backslash L \in$ NPC). Ceci est dû au fait [3] que $B^*\backslash L = (A^*\backslash L) \sqcup (B^*\backslash A^*)$ et que $B^*\backslash A^* \in$ P.

Tout comme il est généralement admis que $P \neq$ NP, il est supposé que NP $\neq$ co-NP. Nous avons le résultat suivant, analogue du Corollaire 3.5.4.

Proposition 3.7.2. NP = co-NP $\iff$ NP $\cap$ co-NPC $\neq \varnothing$.

Preuve. Supposons d'abord que NP = co-NP. Soit $L \in$ co-NPC (un tel L existe, par exemple le complémentaire de SAT). Pour montrer que NP $\cap$ co-NPC $\neq \varnothing$, il suffit de montrer que $L \in$ NP. Soit A un alphabet tel que $L \subseteq A^*$. Par définition, $A^*\backslash L \in$ NPC. En particulier, $A^*\backslash L \in$ NP. Donc $A^*\backslash L \in$ co-NP vu notre hypothèse. On obtient que $L \in$ NP.

Montrons à présent la réciproque. Supposons qu'il existe $L \in$ NP $\cap$ co-NPC. Soit A un alphabet tel que $L \subseteq A^*$. Par symétrie, il suffit de montrer que NP $\subseteq$ co-NP. Soit $K \in$ NP et soit B un alphabet tel que $K \subseteq B^*$. Par hypothèse, $A^*\backslash L \in$ NPC. On a donc $K \leqslant A^*\backslash L$. Soit $f\colon B^* \to A^*$ une transformation polynomiale de K vers $A^*\backslash L$. Il est facile de voir que f est aussi une transformation polynomiale de $B^*\backslash K$ vers L. Ainsi, on a $B^*\backslash K \leqslant L$. Comme $L \in$ NP, on obtient que par la Proposition 3.4.3 que $B^*\backslash K \in$ NP, c'est-à-dire que $K \in$ co-NP. □

Définition 3.7.3. On note EXPTIME la classe des langages décidés par une machine de Turing (déterministe) dont la complexité est en $O(2^{P(n)})$ pour un polynôme P, c'est-à-dire bornée par une fonction exponentielle.

Il est connu que non seulement la classe EXPTIME est non vide, mais on sait même qu'elle contient des langages non polynomiaux. Nous donnons le résultat suivant sans démonstration.

Théorème 3.7.4. P $\subsetneq$ EXPTIME *et* NP $\subseteq$ EXPTIME.

De la même façon que nous avons défini les problèmes NP-complets, on peut définir les problèmes EXPTIME-complets.

Définition 3.7.5. On note

$$\text{EXPTIME-complet} = \{L \in \text{EXPTIME} : \forall K \in \text{EXPTIME},\ K \leqslant L\}.$$

De ce théorème admis, nous pouvons déduire le résultat suivant.

Corollaire 3.7.6. EXPTIME-complet $\cap$P $= \varnothing$.

Preuve. Supposons au contraire qu'il existe un langage $L \in$ EXPTIME-complet $\cap$P. Montrons qu'alors on aurait EXPTIME = P, ce qu'on sait être faux par le théorème 3.7.4. En effet, considérons $M \in$ EXPTIME. On a donc $M \leqslant L$. Mais puisque $L \in$ P, on obtient que $M \in$ P par la proposition 3.4.3. □

La complexité spatiale est définie de manière similaire à la complexité temporelle en tenant compte du nombre de cases du ruban mémoire utilisées (en plus de celui utilisé pour stocker le mot en entrée) au cours d'une exécution d'une machine de Turing plutôt que du nombre de transitions effectuées [4].

Définition 3.7.7. On note PSPACE la classe des langages décidés par une machine de Turing déterministe dont la complexité spatiale est majorée par un polynôme et NPSPACE la classe des langages acceptés par une machine de Turing non déterministe dont la complexité spatiale est majorée par un polynôme.

3. Le symbole $\sqcup$ désigne l'union disjointe.
4. Adaptez la définition 3.1.1 à ce contexte.

Contrairement à la complexité temporelle, on peut montrer que ces deux classes coïncident. Nous admettons ce résultat.

Théorème 3.7.8. PSPACE = NPSPACE.

Nous pouvons en déduire le résultat suivant.

Corollaire 3.7.9. NP $\subseteq$ PSPACE *et* co-NP $\subseteq$ PSPACE.

Preuve. Il suffit de remarquer que la complexité spatiale est toujours inférieure à la complexité temporelle. □

Nous avons le résultat suivant, admis également.

Théorème 3.7.10. PSPACE $\subseteq$ EXPTIME.

Comme d'habitude, on peut définir les problèmes PSPACE-complets, mais il n'est pas connu si ces problèmes sont tous EXPTIME-complets ou non.

Enfin, considérons une dernière classe de complexité.

Définition 3.7.11. On note LOGSPACE la classe des langages décidés par une machine de Turing déterministe utilisant un espace mémoire logarithmique en plus de celui utilisé pour stocker le mot en entrée.

Théorème 3.7.12. LOGSPACE $\subseteq$ P.

On pense que cette inclusion est stricte, mais cette affirmation n'a pas encore été démontrée à ce jour. Il s'agit d'une conjecture majeure en théorie de la complexité.

3.8 Deux problèmes indécidables célèbres

Nous terminons ce document par mentionner deux importants problèmes en théorie de la décidabilité. Le premier est le dixième problème de Hilbert. En 1900, au deuxième congrès international des mathématiciens, David Hilbert expose 23 problèmes qu'il considère comme les problèmes mathématiques de l'époque. Ces problèmes sont de natures différentes. Le dixième d'entre eux concerne en fait, avant l'heure, une question de décidabilité.

Définition 3.8.1. Le dixième problème de Hilbert est le suivant. Étant donné un polynôme multivarié P à coefficients entiers, c'est-à-dire $P \in \mathbb{Z}[X_1, \ldots, X_n]$ pour un certain n, déterminer si l'équation $P = 0$ possède une solution entière.

Une équation du $P = 0$, où P est un polynôme multivarié, pour laquelle on recherche des solutions entières est ce qu'on appelle une équation diophantienne. Grâce aux travaux de Church et Turing des années 1930, initiateurs de la théorie de la calculabilité, le dixième problème de Hilbert a pu être formulé de façon rigoureuse (la formulation originale de Hilbert reposait sur une notion intuitive de procédure effective). Ce n'est qu'en 1970 que Youri Matiiassevitch (à 23 ans à peine) a démontré qu'il s'agissait d'un problème indécidable. La preuve de Matiiassevitch s'appuie sur les travaux précédents de Julia Robinson, c'est pourquoi on parle en général du théorème de Matiiassevitch-Robinson.

Pour énoncer le théorème de Matiiassevitch-Robinson, nous donnons d'abord les définitions suivantes.

Définition 3.8.2. Un *ensemble diophantien* est un ensemble de la forme

$$\{(a_1, \ldots, a_m) \in \mathbb{N}^m : \exists (b_1, \ldots, b_n) \in \mathbb{Z}^n,\ P(a_1, \ldots, a_m, b_1, \ldots, b_n) = 0\}$$

pour un certain polynôme $P \in \mathbb{Z}[X_1, \ldots, X_{m+n}]$.

Par exemple, le sous-ensemble de $\mathbb{N}^2$ formés des couples de naturels premiers entre eux est diophantien puisque, par le théorème de Bézout, cet ensemble est donné par

$$\{(a, b) \in \mathbb{N}^2 : \exists (m, n) \in \mathbb{Z}^2,\ ma + nb - 1 = 0\}.$$

Définition 3.8.3. On qualifie un sous-ensemble A de $\mathbb{N}^m$ de *récursif* (resp. *récursivement énumérable*) lorsque le langage $\{\mathtt{u}^{a_1}\, \mu\, \mathtt{u}^{a_2} \cdots \mu\, \mathtt{u}^{a_m} : (a_1, \ldots, a_m) \in A\}$, où μ est un symbole différent de $\mathtt{u}$, est décidable (resp. acceptable).

Le théorème de Matiiassevitch-Robinson nous dit que ces deux notions coïncident.

Théorème 3.8.4 (Matiiassevitch-Robinson)**.** *La classe des ensembles diophantiens coïncide avec la classe des ensembles d'entiers récursivement énumérables.*

Puisqu'il existe des langages acceptables indécidables, le théorème de Matiiassevitch-Robinson a pour conséquence le résultat suivant.

Corollaire 3.8.5. *Il existe un ensemble diophantien non récursif.*

En conséquence, on obtient l'indécidabilité du dixième problème de Hilbert.

Corollaire 3.8.6. *Le dixième problème de Hilbert est indécidable.*

Preuve. Supposons au contraire que le dixième problème de Hilbert soit décidable. Soient $P \in \mathbb{Z}[X_1, \ldots, X_{m+n}]$ et $(a_1, \ldots, a_m) \in \mathbb{N}^m$. Considérons le polynôme

$$Q = P(a_1, \ldots, a_m, X_{m+1}, \ldots, X_{m+n})$$

de $\mathbb{Z}[X_{m+1}, \ldots, X_{m+n}]$. Par hypothèse, nous pouvons décider si l'équation $Q = 0$ possède une solution entière. L'ensemble diophantien correspondant

$$\{(a_1, \ldots, a_m) \in \mathbb{N}^m : \exists (b_1, \ldots, b_n) \in \mathbb{Z}^n,\ P(a_1, \ldots, a_m, b_1, \ldots, b_n) = 0\}$$

est donc récursif. Puisque m, n, P sont arbitraires, ceci montre que tous les ensembles diophantiens sont récursifs, ce qui est en contradiction avec le corollaire précédent. □

En fait, on peut même montrer ce qu'on appelle la version forte de l'indécidabilité du dixième problème de Hilbert.

Théorème 3.8.7 (Version forte de l'indécidabilité du dixième problème de Hilbert)**.** *Il existe un polynôme $P \in \mathbb{Z}[X_1, \ldots, X_m]$ tel que l'ensemble*

$$\{a \in \mathbb{N} : \exists b_2, \ldots, b_m \in \mathbb{Z},\ P(a, b_2, \ldots, b_m) = 0\}$$

est non récursif.

Le deuxième problème indécidable que nous mentionnons est le problème de correspondance de Post.

Définition 3.8.8. Le problème de correspondance de Post (PCP) est le suivant. Étant donné deux morphismes (de monoïdes) $f, g\colon A^* \to B^*$, déterminer s'il existe un mot non vide w sur l'alphabet A tel que $f(w) = g(w)$.

Exemple 3.8.9. Soient les alphabets $A = \{0, 1\}$ et $B = \{a, b\}$, et les morphismes $f, g\colon A^* \to B^*$ définis par $f(0) = ab$, $f(1) = a$, $g(0) = a$, $g(0) = ba$. On a $aba = f(01) = g(01)$. Ceci montre que le couple (f, g) est une instance positive du problème de correspondance de Post.

On peut montrer que PCP est indécidable en exhibant une réduction directe du problème de l'arrêt à PCP. Nous ne montrons pas ce résultat dans ces notes.

Théorème 3.8.10. PCP *est indécidable.*

Néanmoins, certaines restrictions de PCP sont décidables. Par exemple, si on impose que A soit un alphabet binaire, le problème devient décidable. Si la taille de l'alphabet A est fixée et supérieure ou égale à 5, le problème reste indécidable. Le statut du problème restreint à un alphabet A de taille 3 ou 4 est inconnu.

Table des matières

Bibliographie

[1] Stephen Cook. The complexity of theorem-proving procedures automata. In *Proceedings of the third annual ACM symposium on Theory of computing*, page 151–158, 1971.

[2] Pierre Lecomte. Algorithmique et calculabilité. Notes de cours, Université de Liège, 1994-1995.

[3] Michel Rigo. Algorithmique et calculabilité. Notes de cours, Université de Liège, 2009-2010.

[4] Michael Sipser. *Introduction to the theory of computation.* Course Technology, Boston, MA, Third edition, 2013.

[5] Alan Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. London Math. Soc. (2)*, 42(3) :230–265, 1936.

[6] Pierre Wolper. *Introduction à la calculabilité.* Dunod, Troisème édition, 2006.