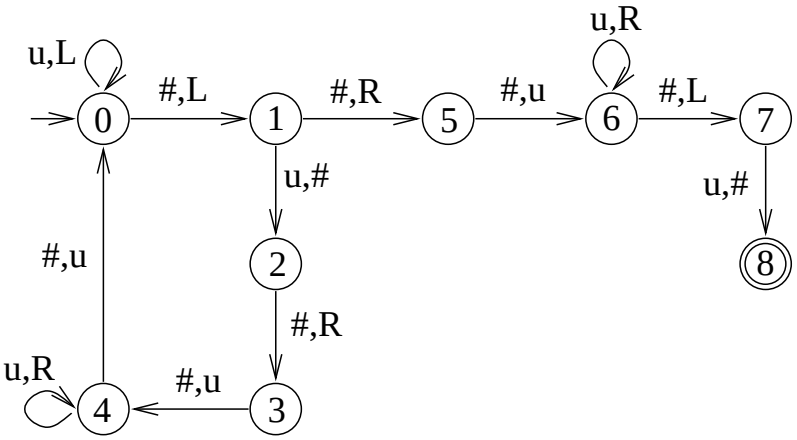
Université de Liège

Faculté des sciences

Algorithmique et Calculabilité



3èmes Bacheliers en sciences mathématiques

Année académique 2009–2010

Michel Rigo

Table des matières

Chapitre I. Fonctions récursives 1
1. Fonctions primitives récursives 1
2. Prédicats primitifs récursifs 6
3. Enumération de p-uples 11
4. La fonction d'Ackermann 14
5. Caractère non dénombrable 22
6. Fonctions récursives 24

Chapitre II. Machines de Turing 27
1. Quelques définitions 28
2. Fonctions calculables 31
3. Composition de machines de Turing 34
4. Fonctions calculables et récursives 40
5. Langages acceptable et décidable 52
6. Extension des machines de Turing 53
7. Fonctions non calculables 60
8. Machines de Turing universelles 62
9. Langages récursifs et récursivement énumérables 68
10. Le problème de l'arrêt 79

Chapitre III. Complexité 89
1. Premières définitions 90
2. Transformations polynomiales 93
3. Les classes P, NP et NPC 95
4. Le théorème de Cook 100
5. D'autres cas 105
6. Quelques réductions 106
7. Complexité et cryptographie 114
8. D'autres classes de complexité 117

Bibliographie 119

Liste des figures 121

Index 123

CHAPITRE I

Fonctions récursives

Une question fondamentale de l'informatique théorique est de déterminer si un problème donné peut ou non être résolu au moyen d'un ordinateur (i.e., par une procédure effective). En particulier se pose la question de déterminer si une fonction donnée (à arguments et valeurs entiers) peut être calculée au moyen d'un algorithme. La question ainsi formulée est indépendante de la machine utilisée, de ses éventuelles performances, de la mémoire disponible ou encore du langage de programmation employé.

Avant de donner un modèle formalisant, d'une certaine manière, la notion de "procédure effective" (nous introduirons le moment voulu le concept de machine de Turing), nous présentons dans ce chapitre[1] une classe de fonctions de $\mathbb{N}^p$ à valeurs dans $\mathbb{N}$ pour lesquelles une telle procédure de calcul existe toujours. Il s'agit des *fonctions primitives récursives*. Nous montrerons que de nombreuses fonctions usuelles sont de ce type.

Ensuite, on verra grâce à la *fonction d'Ackermann* qu'il existe des fonctions non primitives récursives pour lesquelles on dispose néanmoins d'une procédure effective de calcul. L'ensemble des fonctions primitives récursives est donc trop restreint pour capturer exactement la notion de fonction calculable par "procédure effective" et c'est pour cette raison que nous introduirons alors la classe des *fonctions récursives*. Cette dernière contient donc strictement l'ensemble des fonctions primitives récursives et, comme nous pourrons nous en convaincre, correspondra exactement aux fonctions calculables (on entend par là, calculables par machine de Turing).

1. Fonctions primitives récursives

Soit p un entier. On note $\mathfrak{F}_p$ l'ensemble des applications de $\mathbb{N}^p$ dans $\mathbb{N}$. Dans le cas où $p = 0$, les éléments de $\mathfrak{F}_0$ sont identifiés avec les éléments de $\mathbb{N}$ (puisqu'on a une fonction sans argument, on associe à la suite vide un naturel). On pose de plus

$$\mathfrak{F} = \bigcup_{p \in \mathbb{N}} \mathfrak{F}_p.$$

Nous commençons par introduire des fonctions qui vont nous servir de composants élémentaires dans l'élaboration de nouvelles fonctions. Nous les appellerons dès lors *fonctions initiales* (on parle aussi parfois de *fonctions primitives récursives de base*). Elles sont de trois types.

[1] Ce premier chapitre s'inspire principalement de R. Cori, D. Lascar, *Logique Mathématique*, Tome 2, Ch. 5 : *Récursivité*, Dunod, Paris, 1993.

Définition I.1.1. Soient i et p deux entiers tels que $1 \leq i \leq p$. La *fonction de projection* $\mathcal{P}_{i,p}$ appartenant à $\mathfrak{F}_p$ est définie par

$$\mathcal{P}_{i,p} : \mathbb{N}^p \to \mathbb{N} : (x_1, \ldots, x_p) \mapsto x_i.$$

Par exemple, $\mathcal{P}_{1,1} : \mathbb{N} \to \mathbb{N}$ est la fonction identité.

Définition I.1.2. La *fonction successeur* σ appartenant à $\mathfrak{F}_1$ est définie par

$$\sigma : \mathbb{N} \to \mathbb{N} : x \mapsto x + 1.$$

Définition I.1.3. Enfin, la *fonction zéro* $\mathbf{0}$ appartenant à $\mathfrak{F}_0$ est simplement la constante 0. C'est une fonction sans argument qui a toujours la valeur 0.

Nous introduisons à présent deux moyens de construction de nouvelles fonctions à partir de fonctions déjà définies. Ces schémas sont classiques et ne devraient aucunement surprendre le lecteur.

Définition I.1.4. Soient $f_1, \ldots, f_n$ des fonctions de $\mathfrak{F}_p$ et g une fonction de $\mathfrak{F}_n$. La *fonction composée* $h = g(f_1, \ldots, f_n)$ est la fonction de $\mathfrak{F}_p$ définie par

$$h(x_1, \ldots, x_p) = g(f_1(x_1, \ldots, x_p), \ldots, f_n(x_1, \ldots, x_p)).$$

Dans la suite, on s'autorisera à noter indifféremment la composée $g \circ f$ ou $g(f)$.

Définition I.1.5. Soient $g \in \mathfrak{F}_p$ et $h \in \mathfrak{F}_{p+2}$. On définit une fonction $f \in \mathfrak{F}_{p+1}$ telle que pour tous $x_1, \ldots, x_p, n \in \mathbb{N}$

- $f(x_1, \ldots, x_p, 0) = g(x_1, \ldots, x_p)$
- $f(x_1, \ldots, x_p, n+1) = h(x_1, \ldots, x_p, n, f(x_1, \ldots, x_p, n))$.

On dit naturellement que f est définie par *récursion primitive* à partir de g et de h.

Remarque I.1.6. Dans l'introduction, nous avons émis le souhait de disposer de fonctions *effectivement calculables*, c'est-à-dire de fonctions pour lesquelles on dispose d'un algorithme de calcul (au sens naïf du terme).

Le lecteur pourra se convaincre aisément que disposant d'algorithmes $F_1, \ldots, F_n$ pour calculer des fonctions $f_1, \ldots, f_n$ et d'un algorithme G pour calculer une fonction g, alors on dispose aussi d'un algorithme pour en calculer la composée $g(f_1, \ldots, f_n)$.

De même, si on dispose d'un algorithme G (resp. H) pour calculer $g \in \mathfrak{F}_p$ (resp. $h \in \mathfrak{F}_{p+2}$), alors on dispose d'un algorithme pour calculer une fonction f définie par récursion primitive à partir de g et de h. Pour rechercher la valeur de $f(x_1, \ldots, x_p, n)$, on calcule d'abord $f(x_1, \ldots, x_p, 0) = g(x_1, \ldots, x_p)$ au moyen de G. Connaissant cette valeur, on peut alors calculer $f(x_1, \ldots, x_p, 1)$ au moyen de H. On continue à appliquer H jusqu'à obtenir la valeur recherchée.

Remarque I.1.7. La définition par récursion primitive d'une fonction est plus restrictive que la récursion générale employée dans les langages de programmation. En effet, il est toujours possible et ce, quel que soit le langage impératif de programmation employé, de définir des fonctions dont l'évaluation ne se termine pas. Par exemple en langage `C`, on pourrait définir une fonction `f` comme suit :

```
int f(int n)
 {
  if (n == 0)
    return 0;
  else
    return 1+f(n);
 }
```

Il est clair que l'évaluation de cette fonction pour un argument non nul ne se termine pas[2]. Noter que dans les définitions par récursion primitive, un tel cas de figure n'est pas envisageable.

Nous pouvons à présent définir l'ensemble $\mathcal{PR}$ des fonctions primitives récursives.

Définition I.1.8. L'ensemble $\mathcal{PR}$ des *fonctions primitives récursives* est le plus petit sous-ensemble de $\mathfrak{F}$ qui

- contient la fonction zéro $\mathbf{0}$,
- contient les fonctions de projection $\mathcal{P}_{i,p}$ quels que soient les entiers i et p tels que $1 \leq i \leq p$,
- contient la fonction successeur σ,
- est stable pour la composition, i.e., si n et p sont des entiers, $f_1, \ldots, f_n$ des fonctions de $\mathfrak{F}_p$ qui appartiennent à $\mathcal{PR}$ et g une fonction de $\mathfrak{F}_n$ qui appartient à $\mathcal{PR}$, alors la fonction composée $g(f_1, \ldots, f_n)$ appartient encore à $\mathcal{PR}$,
- est stable par récursion primitive, i.e., si p est un entier, g une fonction de $\mathfrak{F}_p$ appartenant à $\mathcal{PR}$ et h une fonction de $\mathfrak{F}_{p+2}$ appartenant à $\mathcal{PR}$, alors la fonction définie par récursion primitive à partir de g et de h appartient encore à $\mathcal{PR}$.

Dans la suite, pour simplifier l'exposé, on s'autorisera souvent à écrire "f est $\mathcal{PR}$" ou "f est de classe $\mathcal{PR}$" lorsque f est une fonction primitive récursive.

Passons en revue quelques exemples de fonctions primitives récursives. De manière générale, pour vérifier qu'une fonction est primitive récursive, il suffit de vérifier qu'elle s'obtient par composition ou récursion primitive à partir de fonctions $\mathcal{PR}$.

[2]Sur un véritable ordinateur, la pile utilisée pour stocker les appels récursifs étant finie, une telle définition provoquerait inévitablement une erreur de dépassement de pile et forcerait le programme à s'achever.

Exemple I.1.9. Les fonctions constantes de $\mathfrak{F}_0$ sont primitives récursives. En effet, la fonction constante[3] $\mathbf{1}$ s'obtient en composant la fonction zéro et la fonction successeur qui sont toutes les deux primitives récursives

$$\mathbf{1} = \sigma \circ \mathbf{0}.$$

D'une manière générale, la fonction constante $\mathbf{n+1}$ est la composée de la fonction successeur et de la fonction $\mathbf{n}$ qui, par hypothèse de récurrence, est primitive récursive.

Exemple I.1.10. Les fonctions constantes de $\mathfrak{F}_p$ sont primitives récursives. Soit n un naturel, on note $\mathbf{n}_p$ la fonction de $\mathfrak{F}_p$ définie par

$$\mathbf{n}_p : \mathbb{N}^p \to \mathbb{N} : (x_1, \ldots, x_p) \mapsto n.$$

Au vu de l'exemple précédent, nous savons déjà que pour tout $n \in \mathbb{N}$, la fonction $\mathbf{n}_0 \in \mathfrak{F}_0$ (notée précédemment $\mathbf{n}$) est primitive récursive. Procédons par récurrence sur p. Supposons que $\mathbf{n}_p$ est une fonction primitive récursive et montrons que $\mathbf{n}_{p+1}$ l'est encore. Il est clair que

$$\begin{cases} \mathbf{n}_{p+1}(x_1, \ldots, x_p, 0) = \mathbf{n}_p(x_1, \ldots, x_p) \\ \mathbf{n}_{p+1}(x_1, \ldots, x_p, m+1) = \mathcal{P}_{p+2,p+2}(x_1, \ldots, x_p, m, \mathbf{n}_{p+1}(x_1, \ldots, x_p, m)). \end{cases}$$

Ainsi, $\mathbf{n}_{p+1}$ s'obtient par récursion primitive à partir des fonctions $\mathbf{n}_p$ et $\mathcal{P}_{p+2,p+2}$ qui sont toutes deux primitives récursives. Par conséquent, $\mathbf{n}_{p+1}$ appartient aussi à $\mathcal{PR}$.

Exemple I.1.11. Pour tout entier $p \geq 1$, la *fonction d'addition*

$$\Sigma_p : (x_1, \ldots, x_p) \mapsto \sum_{i=1}^{p} x_i$$

appartenant à $\mathfrak{F}_p$ est primitive récursive. On procède par récurrence sur p. Pour $p = 1$, $\Sigma_1 = \mathcal{P}_{1,1}$ est une fonction primitive récursive. Supposons le résultat acquis pour p et vérifions-le pour $p+1$ en se ramenant au schéma de définition par récursion primitive :

$$\begin{cases} \Sigma_{p+1}(x_1, \ldots, x_p, 0) = \Sigma_p(x_1, \ldots, x_p) \\ \Sigma_{p+1}(x_1, \ldots, x_p, n+1) = \sigma(\mathcal{P}_{p+2,p+2}(x_1, \ldots, x_p, n, \Sigma_{p+1}(x_1, \ldots, x_p, n))). \end{cases}$$

La fonction $\Sigma_p \in \mathfrak{F}_p$ étant $\mathcal{PR}$ et $\sigma \circ \mathcal{P}_{p+2,p+2} \in \mathfrak{F}_{p+2}$ étant la composée de fonctions $\mathcal{PR}$ (donc $\mathcal{PR}$), on en conclut que Σ_{p+1} est $\mathcal{PR}$.

Exemple I.1.12. Pour tout entier $p \geq 1$, la *fonction produit*

$$\Pi_p : (x_1, \ldots, x_p) \mapsto \prod_{i=1}^{p} x_i$$

appartenant à $\mathfrak{F}_p$ est primitive récursive. On procède comme dans l'exemple I.1.11. Les détails sont laissés au lecteur, on notera simplement que

$$\begin{cases} \Pi_{p+1}(x_1, \ldots, x_p, 0) = \mathbf{0}_p(x_1, \ldots, x_p) \\ \Pi_{p+1}(x_1, \ldots, x_p, n+1) = h(x_1, \ldots, x_p, n, \Pi_{p+1}(x_1, \ldots, x_p, n)) \end{cases}$$

[3] Il s'agit d'une fonction de $\mathfrak{F}_0$ qui à la suite vide, associe le naturel 1.

où

$$h = \Sigma_2(\Pi_p(\mathcal{P}_{1,p+2}, \ldots, \mathcal{P}_{p,p+2}), \mathcal{P}_{p+2,p+2}).$$

Cet exemple nous montre que les vérifications rigoureuses peuvent conduire à des notations assez lourdes à manipuler. Nous nous autoriserons donc des simplifications permettant d'alléger les écritures. En particulier, on utilisera les notations usuelles pour les sommes, les produits et les constantes.

Exemple I.1.13. La *fonction puissance* de $\mathfrak{F}_2$ définie par $p : (x, n) \mapsto x^n$ est $\mathcal{PR}$. En effet, il suffit encore une fois d'appliquer le schéma de récursion primitive. On a

$$x^0 = \mathbf{1}_1(x) \quad \text{et} \quad x^{n+1} = \Pi_2(\mathcal{P}_{1,3}, \mathcal{P}_{3,3})(x, n, x^n)$$

ou de manière plus concise,

$$x^0 = 1 \quad \text{et} \quad x^{n+1} = x.x^n.$$

Exemple I.1.14. La *fonction prédécesseur* $\mathcal{P} \in \mathfrak{F}_1$ qui est définie par

$$\mathcal{P}(0) = 0 \quad \text{et} \quad \mathcal{P}(n+1) = n$$

est primitive récursive. C'est immédiat.

Exemple I.1.15. Plus généralement, la *fonction monus* $x \dot{-} y$ de $\mathfrak{F}_2$ définie par

$$x \dot{-} y = \begin{cases} x - y & \text{si } x \geq y, \\ 0 & \text{sinon,} \end{cases}$$

est $\mathcal{PR}$. En effet, en simplifiant les notations,

$$x \dot{-} 0 = x \quad \text{et} \quad x \dot{-} (y+1) = \mathcal{P}(x \dot{-} y).$$

Exemple I.1.16. La fonction

$$sign(x) = \begin{cases} 1 & \text{si } x > 0, \\ 0 & \text{si } x = 0 \end{cases}$$

est $\mathcal{PR}$ car $sign(x) = 1 \dot{-} (1 \dot{-} x)$.

Exercice I.1.17. Vérifier que les fonctions suivantes sont primitives récursives

- la fonction $n \uparrow\uparrow m$ définie par

$$\begin{cases} n \uparrow\uparrow 0 = 1 \\ n \uparrow\uparrow m+1 = n^{n \uparrow\uparrow m}, \end{cases}$$

$$n \uparrow\uparrow m = n^{n^{n^{n^{\cdot^{\cdot^{\cdot}}}}}} \Big\} m \text{ fois.}$$

- la fonction factorielle $n!$.

- Soit $f \in \mathfrak{F}_{p+1}$ une fonction $\mathcal{PR}$. Vérifier que les fonctions $g, h \in \mathfrak{F}_{p+1}$ de *somme bornée* et de *produit borné* définies par

$$g(x_1, \ldots, x_p, n) = \sum_{i=0}^{n} f(x_1, \ldots, x_p, i)$$

et

$$h(x_1, \ldots, x_p, n) = \prod_{i=0}^{n} f(x_1, \ldots, x_p, i)$$

sont encore $\mathcal{PR}$.

2. Prédicats primitifs récursifs

Définition I.2.1. Soit E une partie de $\mathbb{N}^p$. La *fonction caractéristique* de E est définie par

$$\chi_E(x_1, \ldots, x_p) = \begin{cases} 1 \text{ si } (x_1, \ldots, x_p) \in E \\ 0 \text{ sinon.} \end{cases}$$

Une partie $E \subseteq \mathbb{N}^p$ est *primitive récursive* si sa fonction caractéristique χ_E l'est.

Soit P une propriété[4] portant sur les p-uples d'entiers (on parle aussi de *prédicat d'arité* p). Cette propriété sera dite *primitive récursive* si son *ensemble caractéristique*

$$\delta_P = \{(x_1, \ldots, x_p) \mid P(x_1, \ldots, x_p) \text{ vrai}\} \subseteq \mathbb{N}^p$$

est primitif récursif. Si on définit la fonction caractéristique χ_P d'un prédicat P comme étant la fonction caractéristique de l'ensemble δ_P, i.e.,

$$\chi_P(x_1, \ldots, x_p) = 1 \text{ si et seulement si } (x_1, \ldots, x_p) \in \delta_P,$$

on dira encore qu'un prédicat est $\mathcal{PR}$ si sa fonction caractéristique l'est.

Remarque I.2.2. Dans la littérature, il arrive souvent d'identifier les prédicats d'arité p et les sous-ensembles de $\mathbb{N}^p$. En d'autres termes, on identifie la propriété P et son ensemble caractéristique δ_P. Et réciproquement, à un sous-ensemble $E \subset \mathbb{N}^p$ correspond un prédicat P_E défini par $P_E(x_1, \ldots, x_p)$ si et seulement si $(x_1, \ldots, x_p)$ appartient à E. Nous nous autoriserons également cette identification dans la suite de ces notes.

Exemple I.2.3. Soit $\mathfrak{P}$ l'ensemble des nombres premiers. On a par exemple,

$$\chi_{\mathfrak{P}}(2) = \chi_{\mathfrak{P}}(3) = \chi_{\mathfrak{P}}(5) = 1 \text{ et } \chi_{\mathfrak{P}}(4) = \chi_{\mathfrak{P}}(6) = 0.$$

Considérons la propriété $P(x, y) \equiv x < y$. Les couples suivants

$$(1,2), (1,3), (1,4), \ldots, (2,3), (2,4), \ldots$$

appartiennent à δ_P. Par contre, $(1,1)$ et $(3,2)$ n'y appartiennent pas. Ainsi, $\chi_P(1,1) = 0$ et $\chi_P(1,2) = 1$.

[4] On entend par *propriété* une fonction de $\mathbb{N}^p$ à valeurs dans l'ensemble $\{vrai, faux\}$.

Nous allons à présent voir que de nombreux prédicats usuels sont primitifs récursifs.

Lemme I.2.4. *Soient P, Q deux prédicats primitifs récursifs d'arité p. Les prédicats $P \wedge Q$, $P \vee Q$ et $\neg P$ sont encore primitifs récursifs.*

Démonstration. Soit $\overline{x} = (x_1, \ldots, x_p)$. On a

$$\chi_{P\wedge Q}(\overline{x}) = \chi_P(\overline{x}).\chi_Q(\overline{x}),$$
$$\chi_{P\vee Q}(\overline{x}) = sign(\chi_P(\overline{x}) + \chi_Q(\overline{x}))$$

et

$$\chi_{\neg P}(\overline{x}) = 1\dot{-}\chi_P(\overline{x})$$

Pour $\vee$, il s'agit du "ou non exclusif".

■

Remarque I.2.5. Le lemme précédent peut également s'énoncer comme suit. La classe des parties primitives récursives de $\mathbb{N}^p$ est stable pour les opérations booléennes (union, intersection et complémentation).

Proposition I.2.6. *Les prédicats binaires $<, \leq, =, \geq, >$ sont primitifs récursifs.*

Démonstration. L'ensemble caractéristique du prédicat $>$ est donné par $\delta_> = \{(x, y) \mid x > y\}$ et on vérifie que sa fonction caractéristique est alors égale à

$$\chi_>(x, y) = sign(x\dot{-}y).$$

Par conséquent, $>$ est un prédicat primitif récursif. Pour l'égalité, on a

$$\chi_=(x, y) = 1\dot{-}(sign(x\dot{-}y) + sign(y\dot{-}x)).$$

Les autres se déduisent directement du lemme I.2.4. Par exemple, $x < y \equiv \neg(x > y \vee x = y)$.

■

Proposition I.2.7 (Quantification bornée). *Soit P un prédicat $\mathcal{PR}$ d'arité $p + 1$. Les prédicats*

$$A(x_1, \ldots, x_p, m) \equiv \forall n \leq m\, P(x_1, \ldots, x_p, n)$$

et

$$B(x_1, \ldots, x_p, m) \equiv \exists n \leq m\, P(x_1, \ldots, x_p, n)$$

sont $\mathcal{PR}$.

Cela signifie que $A(x_1, \ldots, x_p, m)$ (resp. $B(x_1, \ldots, x_p, m)$) est *vrai* si pour tout entier $n \leq m$ (resp. s'il existe au moins un entier $n \leq m$), $P(x_1, \ldots, x_p, n)$ est *vrai*.

Démonstration. En posant $\overline{x} = (x_1, \ldots, x_p)$ et avec les notations de l'énoncé, il vient

$$\chi_A(\overline{x}, m) = \prod_{i=0}^{m} \chi_P(\overline{x}, i)$$

et

$$\chi_B(\overline{x}, m) = sign\left(\sum_{i=0}^{m} \chi_P(\overline{x}, i)\right).$$

On conclut en utilisant l'exercice I.1.17.

■

Dans les langages de programmation, il n'est pas rare de rencontrer des constructions faisant intervenir des instructions de branchement du type "`if then else`". Ce type d'instruction peut donc apparaître dans l'élaboration de procédures effectives de calcul et doit donc pouvoir s'adapter au cas des fonctions primitives récursives.

Proposition I.2.8 (Schéma de définition par cas). *Soient P un prédicat d'arité p primitif récursif et $g, h \in \mathfrak{F}_p$ deux fonctions de classe $\mathcal{PR}$. La fonction f définie par*

$$f(x_1, \ldots, x_p) = \begin{cases} g(x_1, \ldots, x_p) \text{ si } P(x_1, \ldots, x_p) \\ h(x_1, \ldots, x_p) \text{ sinon} \end{cases}$$

est primitive récursive.

Démonstration. En effet, pour tout $\overline{x} \in \mathbb{N}^p$,

$$f(\overline{x}) = g(\overline{x}).\chi_P(\overline{x}) + h(\overline{x}).\chi_{\neg P}(\overline{x}).$$

■

On dispose souvent de procédures plus élaborées permettant le choix multiple entre diverses situations. Par exemple, en `C`, on trouve l'instruction

```
switch (expression)
{
   case modele_1: instruction(s);
   case modele_2: instruction(s);
...
   case modele_n: instruction(s);
   default: instruction(s);
}
```

qui permet un choix multiple. On retrouve ce schéma de construction de fonctions primitives récursives comme suit.

Proposition I.2.9 (Schéma (généralisé) de définition par cas). *Soient des prédicats primitifs récursifs $P_1, \ldots, P_n$ d'arité p mutuellement exclusifs, (i.e., pour tous $i \neq j$, $\delta_{P_i} \cap \delta_{P_j} = \emptyset$) et $f_1, \ldots, f_n, f_{n+1} \in \mathfrak{F}_p$ des fonctions $\mathcal{PR}$. La fonction $g \in \mathfrak{F}_p$ définie par*

$$g(x_1, \ldots, x_p) = \begin{cases} f_1(x_1, \ldots, x_p) \text{ si } P_1(x_1, \ldots, x_p), \\ f_2(x_1, \ldots, x_p) \text{ si } P_2(x_1, \ldots, x_p), \\ \cdots \\ f_n(x_1, \ldots, x_p) \text{ si } P_n(x_1, \ldots, x_p), \\ f_{n+1}(x_1, \ldots, x_p) \text{ sinon} \end{cases}$$

est primitive récursive.

Démonstration. On procède comme dans la preuve de la proposition précédente.

■

Proposition I.2.10. *Soient un prédicat P d'arité n primitif récursif et des fonctions $f_1, \ldots, f_n \in \mathfrak{F}_p$ primitives récursives. L'ensemble*

$$E = \{(x_1, \ldots, x_p) \in \mathbb{N}^p \mid P(f_1(x_1, \ldots, x_p), \ldots, f_n(x_1, \ldots, x_p)) \text{ vrai}\}$$

est primitif récursif.

Démonstration. C'est immédiat, on a $\chi_E = \chi_P(f_1, \ldots, f_n)$ qui est une composée de fonctions $\mathcal{PR}$.

■

Corollaire I.2.11. *Soient f et g deux fonctions $\mathcal{PR}$ de $\mathfrak{F}_p$. L'ensemble*

$$\{(x_1, \ldots, x_p) \in \mathbb{N}^p \mid f(x_1, \ldots, x_p) \sim g(x_1, \ldots, x_p)\}$$

où $\sim$ peut prendre les valeurs $\{<, \leq, =, \geq, >\}$, est primitif récursif. En particulier, les ensembles

$$\{(x_1, \ldots, x_p) \mid f(x_1, \ldots, x_p) > 0\} \text{ et } \{(x_1, \ldots, x_p) \mid f(x_1, \ldots, x_p) = 0\}$$

sont aussi $\mathcal{PR}$.

Démonstration. C'est une conséquence immédiate de la proposition précédente et de la proposition I.2.6. Pour le cas particulier, il suffit par exemple de considérer la fonction $g = \mathbf{0}_p$.

■

Nous considérons à présent un dernier schéma de construction permettant, une fois encore, d'obtenir des fonctions primitives récursives. La définition donnée ici s'exprime en termes de partie de $\mathbb{N}^{p+1}$ mais peut bien évidemment être réexprimée en termes de prédicats[5] d'arité $n+1$.

Définition I.2.12 (Schéma μ borné ou minimisation bornée). Soit A une partie de $\mathbb{N}^{p+1}$. La fonction f de $\mathfrak{F}_{p+1}$ est définie par

$f(x_1, \ldots, x_p, n) = 0$ s'il n'existe aucun $t \leq n$ tel que $(x_1, \ldots, x_p, t) \in A$.

Dans le cas contraire, elle est définie par

$$f(x_1, \ldots, x_p, n) = \inf\{t \leq n \mid (x_1, \ldots, x_p, t) \in A\}.$$

On note cette fonction comme suit :

$$f(x_1, \ldots, x_p, n) = \mu_{t \leq n}((x_1, \ldots, x_p, t) \in A)$$

[5]Soit P un prédicat d'arité $n+1$. On définit une fonction f de $\mathfrak{F}_{p+1}$ par $f(x_1, \ldots, x_p, n) = 0$ s'il n'existe aucun $t \leq n$ tel que $P(x_1, \ldots, x_p, t)$. Dans le cas contraire, $f(x_1, \ldots, x_p, n) = \inf\{t \leq n \mid P(x_1, \ldots, x_p, t)\}$. On note encore cette fonction $f(x_1, \ldots, x_p, n) = \mu_{t \leq n}(P(x_1, \ldots, x_p, t))$.

Remarque I.2.13. Insistons sur le fait qu'il s'agit ici du schéma μ borné (on s'intéresse aux t bornés par n). En effet, nous verrons plus loin qu'il existe un schéma de construction μ non borné.

Le schéma de minimisation bornée permet d'obtenir des fonctions primitives récursives à partir d'ensembles $\mathcal{PR}$.

Proposition I.2.14. *Si $A \subset \mathbb{N}^{p+1}$ est $\mathcal{PR}$, alors*

$$f(x_1, \ldots, x_p, n) = \mu_{t \leq n}((x_1, \ldots, x_p, t) \in A)$$

est aussi une fonction primitive récursive.

Démonstration. Comme d'habitude, posons $\overline{x} = (x_1, \ldots, x_p)$. On montre par récurrence sur n que la fonction $\mu_{t \leq n}((\overline{x}, t) \in A)$ est primitive récursive. Pour $n = 0$, on a

$$\mu_{t \leq 0}((\overline{x}, t) \in A) = 0.$$

Si on suppose le résultat vérifié pour n, montrons qu'il l'est encore pour $n + 1$.

$$\mu_{t \leq n+1}((\overline{x}, t) \in A) = \begin{cases} 0 & \text{si} \quad \neg\exists t \leq n+1 : (\overline{x}, t) \in A \\ \mu_{t \leq n}((\overline{x}, t) \in A) & \text{si} \quad \exists t \leq n : (\overline{x}, t) \in A \\ n+1 & \text{si} \quad (\overline{x}, n+1) \in A \\ & \quad \wedge \neg\exists t \leq n : (\overline{x}, t) \in A \end{cases}$$

d'où la conclusion (on utilise le schéma de définition par cas en remarquant que les prédicats rencontrés sont $\mathcal{PR}$ et mutuellement exclusifs; on a aussi recours à la quantification bornée).

■

Remarque I.2.15. Soit $g \in \mathfrak{F}_{p+1}$ une fonction $\mathcal{PR}$. Au vu du corollaire I.2.11, l'ensemble $\{(\overline{x}, t) \in \mathbb{N}^{p+1} \mid g(\overline{x}, t) = 0\}$ est $\mathcal{PR}$. Ainsi, on utilisera souvent le résultat précédent pour montrer que la fonction

$$f(\overline{x}, n) = \mu_{t \leq n}(g(\overline{x}, t) = 0)$$

est $\mathcal{PR}$.

Voici encore quelques exemples de fonctions et prédicats $\mathcal{PR}$.

Exemple I.2.16. Les ensembles suivants sont primitifs récursifs.

- $\mathbb{N}$ est $\mathcal{PR}$. Sa fonction caractéristique est égale à $\mathbf{1}_1$.
- $2\mathbb{N}$ est $\mathcal{PR}$. Sa fonction caractéristique est définie par récursion primitive par

$$\chi_{2\mathbb{N}}(0) = 1 \quad \text{et} \quad \chi_{2\mathbb{N}}(n+1) = 1 \dot{-} \chi_{2\mathbb{N}}(n).$$

Exemple I.2.17. Les fonctions suivantes sont primitives récursives.

- La fonction $\tau \in \mathfrak{F}_2$ définie par

$$\tau(x, y) = \begin{cases} 0 & \text{si } y = 0 \\ \lfloor \frac{x}{y} \rfloor & \text{sinon} \end{cases}$$

est $\mathcal{PR}$. En effet, on se convainc aisément que

$$\tau(x,y) = \mu_{t\leq x}((t+1).y > x).$$

On peut remarquer que cette fonction correspond à la fonction `DIV` des langages de programmation qui fournit le quotient entier de la division de x par y. De même, la fonction `MOD` qui fournit le reste de la division est elle aussi primitive récursive. En effet,

$$x \,\texttt{MOD}\, y = x \dot{-} (x \,\texttt{DIV}\, y).y.$$

- En corollaire, l'ensemble $\mathcal{D} = \{(x,y) \mid y \text{ divise } x\}$ est $\mathcal{PR}$. En effet, sa fonction caractéristique est donnée par

$$\chi_{\mathcal{D}}(x,y) = 1 \dot{-} sign(x \,\texttt{MOD}\, y).$$

De là, on en tire que l'ensemble $\mathfrak{P}$ des nombres premiers est aussi $\mathcal{PR}$. En effet, p est premier si et seulement si

$$p > 1 \wedge \forall n \leq p\, (n \leq 1 \vee n = p \vee (p,n) \notin \mathcal{D}).$$

- La fonction $\nu \in \mathfrak{F}_1$ qui à n associe le $(n+1)$-ième nombre premier est de classe $\mathcal{PR}$. On le montre en utilisant les schémas de récursion primitive et de minimisation bornée.

$$\nu(0) = 2 \quad \text{et} \quad \nu(n+1) = \mu_{t\leq(\nu(n)!+1)}(t > \nu(n) \wedge t \in \mathfrak{P}).$$

Pour la borne $t \leq (\nu(n)!+1)$, nous avons simplement utilisé le fait que si $\nu(n)$ est un nombre premier, il existe toujours un nombre premier dans l'intervalle $[\nu(n)+1, \nu(n)!+1]$. En effet, considérons l'entier $\nu(0)\cdots\nu(n)+1$. Il appartient bien à l'intervalle proposé. S'il est premier, le résultat est démontré. Sinon, il est divisible par un nombre premier mais par aucun des nombres $\nu(0),\ldots,\nu(n)$ d'où la conclusion.

3. Enumération de p-uples

Dans cette section, nous montrons qu'on peut énumérer les p-uples d'entiers de manière primitive récursive. Cela permet donc de restreindre l'étude des fonctions $\mathcal{PR}$ de $\mathfrak{F}_p$ à celles de $\mathfrak{F}_1$. De plus, nous verrons que grâce à cette énumération, nous allons pouvoir définir un schéma généralisé de définition par récursion primitive.

Théorème I.3.1. *Pour tout $p \geq 2$, il existe des fonctions primitives récursives $\pi_p \in \mathfrak{F}_p$ et $\theta_1^{(p)},\ldots,\theta_p^{(p)} \in \mathfrak{F}_1$ telles que*

$$\theta_i^{(p)}(\pi_p(x_1,\ldots,x_p)) = x_i \quad \forall i \in \{1,\ldots,p\}, (x_1,\ldots,x_p) \in \mathbb{N}^p.$$

En d'autres termes, la fonction π_p fait correspondre à tout point de $\mathbb{N}^p$ un élément de $\mathbb{N}$ et réciproquement, on retrouve les composantes du p-uple initial grâce aux fonctions $\theta_i^{(p)}$.

Démonstration. Considérons tout d'abord le cas $p = 2$. On énumère les points de $\mathbb{N}^2$ comme sur la figure I.1. On considère les couples (i,j)

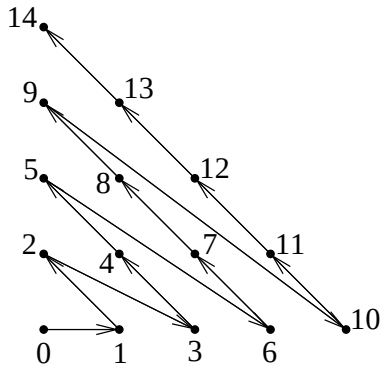


FIGURE I.1. Enumération de Peano.

se trouvant sur une diagonale de la forme $i + j = k$. Et dans l'ordre, on considère sur cette diagonale les couples

$$(k, 0),\ (k-1, 1),\ (k-2, 2), \ldots,\ (2, k-2),\ (1, k-1),\ (0, k).$$

Sur la diagonale $i + j = k$ on trouve exactement $k + 1$ points de $\mathbb{N}^2$. Ainsi, avant d'énumérer le point $(k, 0)$, on aura passé en revue les points des diagonales $i + j = 0, \ldots, i + j = k - 1$, c'est-à-dire

$$1 + 2 + \cdots + k = \frac{k(k+1)}{2}$$

points. Donc, $\pi_2(k, 0) = k(k + 1)/2$ et d'une manière générale,

$$\boxed{\pi_2(i, j) = \frac{1}{2}(i + j)(i + j + 1) + j.}$$

Pour se convaincre du caractère $\mathcal{PR}$ de π_2, on peut la réécrire comme $\pi_2(i, j) = (i + j)(i + j + 1)\,\mathtt{DIV}\,2 + j$. Il est de plus évident que π_2 est une bijection.

Soit $n = \pi_2(i, j)$. Notre but est, connaissant n, de retrouver i et j. En tirant parti du caractère bijectif[6] de π_2, il est clair que

$$\theta_1^{(2)}(n) = \mu_{t \le n}(\exists j \le n : \pi_2(t, j) = n)$$

et

$$\theta_2^{(2)}(n) = \mu_{t \le n}(\exists i \le n : \pi_2(i, t) = n).$$

Le caractère $\mathcal{PR}$ de ces deux fonctions est immédiat à vérifier (il résulte d'une quantification bornée et d'un schéma de minimisation bornée).

Pour $p = 3$, on peut définir π_3 comme suit :

$$\pi_3(x_1, x_2, x_3) = \pi_2(x_1, \pi_2(x_2, x_3))$$

et alors,

$$\theta_1^{(3)} = \theta_1^{(2)}, \quad \theta_2^{(3)} = \theta_1^{(2)} \circ \theta_2^{(2)} \quad \text{et} \quad \theta_2^{(3)} = \theta_2^{(2)} \circ \theta_2^{(2)}.$$

[6]Il existe un unique couple $(i, j) \in \mathbb{N}^2$ tel que $\pi_2(i, j) = n$.

Le caractère $\mathcal{PR}$ est encore une fois immédiat à vérifier. De manière générale, on procède par récurrence sur p. On peut définir les fonctions comme suit :

$$\pi_{p+1}(x_1,\dots,x_{p+1}) = \pi_p(x_1,\dots,x_{p-1},\pi_2(x_p,x_{p+1}))$$

et

$$\theta_1^{(p+1)} = \theta_1^{(p)},\dots,\theta_{p-1}^{(p+1)} = \theta_{p-1}^{(p)},$$
$$\theta_p^{(p+1)} = \theta_1^{(2)} \circ \theta_p^{(p)} \quad \text{et} \quad \theta_{p+1}^{(p+1)} = \theta_2^{(2)} \circ \theta_p^{(p)}.$$

On peut noter qu'on aurait très bien pu définir π_{p+1} par

$$\pi_{p+1}(x_1,\dots,x_{p+1}) = \pi_2(x_1,\pi_p(x_2,\dots,x_{p+1}))$$

et adapter les fonctions $\theta_i^{(p+1)}$ en conséquence.

■

Nous pouvons à présent définir un schéma de récursion primitive généralisé.

Proposition I.3.2. *Soit $k \geq 1$. Soient des fonctions $g_1,\dots,g_k \in \mathfrak{F}_p$ et $h_1,\dots,h_k \in \mathfrak{F}_{p+k+1}$ toutes primitives récursives. Les k fonctions $f_1,\dots,f_k$ de $\mathfrak{F}_{p+1}$ définies pour tout $i \in \{1,\dots,k\}$ par*

$$f_i(x_1,\dots,x_p,0) = g_i(x_1,\dots,x_p)$$

et

$$f_i(x_1,\dots,x_p,n+1) = h_i(x_1,\dots,x_p,n,f_1(x_1,\dots,x_p,n),\dots,f_k(x_1,\dots,x_p,n))$$

sont primitives récursives.

Remarque I.3.3. Dans le cas $k=1$, on retrouve exactement la définition I.1.5 de la récursion primitive. De plus, on peut observer qu'intuitivement, il est clair que si on dispose d'une méthode de calcul effective (d'un algorithme) pour calculer les fonctions g_i et h_i $(i=1,\dots,k)$, on sera alors en mesure de calculer les fonctions f_i.

Exemple I.3.4. Soient $g_1,g_2 \in \mathfrak{F}_p$ et $h_1,h_2 \in \mathfrak{F}_{p+3}$ quatre fonctions $\mathcal{PR}$. Les fonctions f_1 et f_2 de $\mathfrak{F}_{p+1}$ définies par récursion primitive généralisée sont données par

$$\begin{cases} f_1(x_1,\dots,x_p,0) = g_1(x_1,\dots,x_p), \\ f_2(x_1,\dots,x_p,0) = g_2(x_1,\dots,x_p), \\ f_1(x_1,\dots,x_p,n+1) = h_1(x_1,\dots,x_p,n,f_1(x_1,\dots,x_p,n),f_2(x_1,\dots,x_p,n)), \\ f_2(x_1,\dots,x_p,n+1) = h_2(x_1,\dots,x_p,n,f_1(x_1,\dots,x_p,n),f_2(x_1,\dots,x_p,n)). \end{cases}$$

Voici la preuve de la proposition I.3.2.

Démonstration. Comme d'habitude, $\overline{x} = (x_1,\dots,x_p)$. Posons

$$F(\overline{x},n) = \pi_k(f_1(\overline{x},n),\dots,f_k(\overline{x},n)).$$

Il nous suffit de montrer que F est $\mathcal{PR}$ car

$$f_i = \theta_i^{(k)} \circ F.$$

On a, par définition des f_i,

$$F(\overline{x},0) = \pi_k(g_1(\overline{x}),\dots,g_k(\overline{x}))$$

et

$$\begin{aligned} F(\overline{x},n+1) = \quad & \pi_k(h_1(\overline{x},n,\theta_1^{(k)}(F(\overline{x},n)),\dots,,\theta_k^{(k)}(F(\overline{x},n))),\dots, \\ & h_k(\overline{x},n,\theta_1^{(k)}(F(\overline{x},n)),\dots,\theta_k^{(k)}(F(\overline{x},n)))). \end{aligned}$$

On est donc bien en présence d'un schéma de récursion primitive appliqué à des fonctions $\mathcal{PR}$ ce qui suffit.

■

Exemple I.3.5. Voici quelques exemples. Les justifications sont laissées au lecteur.

- Soit A une matrice carrée de $\mathbb{N}_t^t$. Les éléments $a_{i,j}(n) = (A^n)_{i,j}$ sont des fonctions de classe $\mathcal{PR}$.
- La fonction $f \in \mathfrak{F}_1$ définie par
$$\begin{cases} f(0) = u,\ f(1) = v, \\ f(n+2) = a\,f(n+1) + b\,f(n) \end{cases}$$
où $a, b, u, v \in \mathbb{N}$, est $\mathcal{PR}$.
- La fonction qui à $(x_1,\dots,x_p)$ associe $\sup(x_1,\dots,x_p)$ (resp. $\inf(x_1,\dots,x_p)$) est une fonction de classe $\mathcal{PR}$.
- La fonction qui à n associe $\lfloor\sqrt{n}\rfloor$ est aussi de classe $\mathcal{PR}$.

4. La fonction d'Ackermann

Jusqu'à présent, nous avons rencontré de nombreuses fonctions primitives récursives et donné divers moyens d'obtention de nouvelles fonctions $\mathcal{PR}$ à partir d'anciennes. Nous sommes également convaincus que toute fonction $\mathcal{PR}$ est calculable effectivement. On serait dès lors tenter de conclure hâtivement que toute fonction calculable (par un algorithme) est primitive récursive. Nous allons voir dans cette section qu'une telle conclusion serait erronée[7].

Définition I.4.1. La *fonction d'Ackermann* $\mathcal{A} \in \mathfrak{F}_2$ est définie par

$$\begin{cases} \mathcal{A}(0,m) &= m+1, \\ \mathcal{A}(m+1,0) &= \mathcal{A}(m,1), \\ \mathcal{A}(m+1,n+1) &= \mathcal{A}(m,\mathcal{A}(m+1,n)). \end{cases}$$

On pose $\mathcal{A}_m \in \mathfrak{F}_1$ comme

$$\mathcal{A}_m : n \mapsto \mathcal{A}(m,n).$$

[7]Vers les années 1900, on pensait à tort que toute fonction calculable était primitive récursive

Inspectons les premières fonctions $\mathcal{A}_0, \ldots, \mathcal{A}_3$. On a

$$\mathcal{A}_0(n) = n + 1,$$

$$\mathcal{A}_1(0) = \mathcal{A}_0(1) = 2 \quad \text{et} \quad \mathcal{A}_1(n+1) = \mathcal{A}_0(\mathcal{A}_1(n)) = \mathcal{A}_1(n) + 1,$$

on en conclut que $\mathcal{A}_1(n) = n + 2$. Ensuite,

$$\mathcal{A}_2(0) = \mathcal{A}_1(1) = 3 \quad \text{et} \quad \mathcal{A}_2(n+1) = \mathcal{A}_1(\mathcal{A}_2(n)) = \mathcal{A}_2(n) + 2,$$

d'où $\mathcal{A}_2(n) = 2n + 3$. Pour $\mathcal{A}_3$, il vient[8]

$$\mathcal{A}_3(0) = \mathcal{A}_2(1) = 5 \quad \text{et} \quad \mathcal{A}_3(n+1) = \mathcal{A}_2(\mathcal{A}_3(n)) = 2\mathcal{A}_3(n) + 3,$$

d'où $\mathcal{A}_3(n) = 2^{n+3} - 3$.

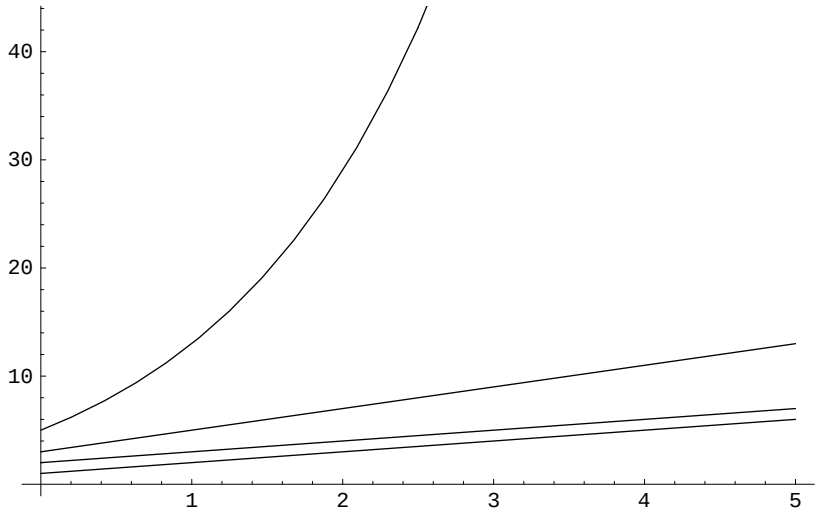


FIGURE I.2. Les fonctions $\mathcal{A}_0, \ldots, \mathcal{A}_3$.

Remarque I.4.2. Pour tout $m \in \mathbb{N}$, la fonction $\mathcal{A}_m \in \mathfrak{F}_1$ est primitive récursive. On procède par récurrence sur m. Le cas de base est immédiat puisque $\mathcal{A}_0(n) = n+1$. Supposons $\mathcal{A}_m$ de classe $\mathcal{PR}$ et montrons que $\mathcal{A}_{m+1}$ l'est encore. C'est immédiat, on dispose du schéma de récursion primitive suivant

$$\begin{cases} \mathcal{A}_{m+1}(0) = \mathcal{A}_m(1), \\ \mathcal{A}_{m+1}(n+1) = \mathcal{A}_m(\mathcal{A}_{m+1}(n)). \end{cases}$$

Insistons encore que $\mathcal{A}_m \in \mathcal{PR}$ n'implique en rien que $\mathcal{A}$ appartienne aussi à $\mathcal{PR}$.

Nous allons à présent nous convaincre que la fonction d'Ackermann $\mathcal{A}$ est calculable. Il suffit pour cela d'exhiber un algorithme de calcul.

Proposition I.4.3. *La fonction d'Ackermann $\mathcal{A}$ est calculable par une procédure effective.*

[8]Cela revient à résoudre la relation de récurrence $x_0 = 5$ et $x_{n+1} = 2x_n + 3$.

Démonstration. Il est clair qu'on doit être en mesure de calculer des expressions de la forme

$$\mathcal{A}(m_1, \mathcal{A}(m_2, \ldots, \mathcal{A}(m_{k-1}, m_k))) \quad \text{où } k \geq 2.$$

Nous coderons une telle expression par le k-uple $\mathtt{S} = (m_1, \ldots, m_k) \in \mathbb{N}^k$ et $|\mathtt{S}|$ en représente la longueur, i.e., $|\mathtt{S}| = k$. Nous notons $\mathtt{q}$ l'élément m_k le plus à droite de $\mathtt{S}$, $\mathtt{p}$ son avant-dernier élément m_{k-1} et $\mathtt{T}$ la suite des $k-2$ premiers éléments ($\mathtt{T}$ peut être vide si $k = 2$). Ainsi,

$$\mathtt{S} = (\underbrace{m_1, \ldots, m_{k-2}}_{\mathtt{T}}, \underbrace{m_{k-1}}_{\mathtt{p}}, \underbrace{m_k}_{\mathtt{q}}) = (\mathtt{T}, \mathtt{p}, \mathtt{q}).$$

L'algorithme suivant calcule $\mathcal{A}(m, n)$:

$\mathtt{T} \leftarrow \emptyset$, $\mathtt{p} \leftarrow m$, $\mathtt{q} \leftarrow n$ (on initialise $\mathtt{S}$ à $(\emptyset, m, n)$)
Tant que $|S| \geq 2$
  Si $\mathtt{p} = 0$ et $\mathtt{q} \geq 0$, alors $\mathtt{S} \leftarrow (\mathtt{T}, \mathtt{q} + 1)$
  Si $\mathtt{p} > 0$ et $\mathtt{q} = 0$, alors $\mathtt{S} \leftarrow (\mathtt{T}, \mathtt{p} - 1, 1)$
  Si $\mathtt{p} > 0$ et $\mathtt{q} > 0$, alors $\mathtt{S} \leftarrow (\mathtt{T}, \mathtt{p} - 1, \mathtt{p}, \mathtt{q} - 1)$
Sortir $\mathtt{S} = (\mathtt{q})$.

L'idée de cet algorithme est de toujours s'intéresser à la fonction $\mathcal{A}$ la plus imbriquée dans $\mathtt{S}$. Les trois règles de l'algorithme correspondent exactement à la définition de $\mathcal{A}$:

$$(\mathtt{T}, 0, \mathtt{q}) \text{ devient } (\mathtt{T}, \mathtt{q} + 1) \text{ car } \mathcal{A}(0, n) = n + 1,$$

$$(\mathtt{T}, \mathtt{p}, 0) \text{ devient } (\mathtt{T}, \mathtt{p} - 1, 1) \text{ car } \mathcal{A}(m + 1, 0) = \mathcal{A}(m, 1),$$

$(\mathtt{T}, \mathtt{p}, \mathtt{q})$ devient $(\mathtt{T}, \mathtt{p} - 1, \mathtt{p}, \mathtt{q} - 1)$ car $\mathcal{A}(m+1, n+1) = \mathcal{A}(m, \mathcal{A}(m+1, n))$.

Il est donc clair que *si l'algorithme s'achève*, ce sera avec le bon résultat. Il nous suffit alors de prouver (par récurrence sur m) qu'il s'achève pour toutes valeurs de m et n. Pour $m = 0$, $\mathtt{S}$ est initialisé à $(0, n)$, on sort de la boucle avec une suite $\mathtt{S} = (n + 1)$ de longueur 1. Si l'algorithme s'achève pour m, montrons qu'il s'achève encore pour $m + 1$. Si $\mathtt{S}$ est initialisé à $(m + 1, n)$, alors on a, en appliquant la troisième règle,

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

En appliquant la deuxième règle, on trouve

$$\to (m, m, \ldots, m, m, 1)$$

ce qui nous permet d'appliquer $n + 1$ fois l'hypothèse de récurrence pour conclure.

■

Exemple I.4.4. Pour calculer $\mathcal{A}(1,2)$, avec les notations précédentes, on trouve

$$(1,2) \to (0,1,1) \to (0,0,1,0) \to (0,0,0,1) \to (0,0,2) \to (0,3) \to (4).$$

Pour montrer que $\mathcal{A}$ n'appartient pas à $\mathcal{PR}$, quelques lemmes sont nécessaires. Ces lemmes sont pour la plupart aisés mais nombreux !

Lemme I.4.5. *Pour tout m et pour tout n,*

$$\mathcal{A}_m(n) > n.$$

On a une double récurrence.

Démonstration. On procède par récurrence sur m. Pour $m = 0$ et pour tout n, $\mathcal{A}_0(n) = n + 1 > n$. Supposons que, pour tout n, $\mathcal{A}_m(n) > n$ et vérifions que, pour tout n, $\mathcal{A}_{m+1}(n) > n$. On procède cette fois par récurrence sur n. Si $n = 0$, alors par hypothèse de récurrence sur m, on a bien $\mathcal{A}_{m+1}(0) = \mathcal{A}_m(1) > 1 > 0$. Supposons à présent que $\mathcal{A}_{m+1}(n) > n$ et vérifions que $\mathcal{A}_{m+1}(n+1) > n+1$. Il vient en utilisant successivement les hypothèses de récurrence sur m et sur n

$$\mathcal{A}_{m+1}(n+1) = \mathcal{A}_m(\mathcal{A}_{m+1}(n)) > \mathcal{A}_{m+1}(n) > n$$

d'où le résultat annoncé, $\mathcal{A}_{m+1}(n+1) > n+1$, puisque nous sommes en présence de deux inégalités strictes consécutives.

■

Lemme I.4.6. *Pour tout m, la fonction $\mathcal{A}_m$ est strictement croissante, i.e., $\mathcal{A}_m(n+1) > \mathcal{A}_m(n)$.*

Démonstration. Si $m = 0$, c'est immédiat puisque $\mathcal{A}_0(n) = n + 1$. Si $m > 0$, alors en utilisant le lemme précédent, il vient

$$\mathcal{A}_m(n+1) = \mathcal{A}_{m-1}(\mathcal{A}_m(n)) > \mathcal{A}_m(n).$$

■

Lemme I.4.7. *Pour tout m et pour tout n, on a*

$$\mathcal{A}_{m+1}(n) \geq \mathcal{A}_m(n).$$

Démonstration. Pour $n = 0$, on a $\mathcal{A}_{m+1}(0) = \mathcal{A}_m(1) > \mathcal{A}_m(0)$ car $\mathcal{A}_m$ est une fonction strictement croissante (cf. lemme I.4.6).

Pour $n > 0$, au vu du lemme I.4.5, $\mathcal{A}_{m+1}(n-1) \geq n$ et puisque $\mathcal{A}_m$ est une fonction strictement croissante, il vient

$$\mathcal{A}_m(\mathcal{A}_{m+1}(n-1)) \geq \mathcal{A}_m(n).$$

En appliquant la définition de la fonction d'Ackermann, on en conclut que

$$\mathcal{A}_{m+1}(n) = \mathcal{A}_m(\mathcal{A}_{m+1}(n-1)) \geq \mathcal{A}_m(n).$$

Ceci achève la preuve.

■

On note $\mathcal{A}_m^k$ la composition de k copies de $\mathcal{A}_m$, i.e.,

$$\mathcal{A}_m^k = \underbrace{\mathcal{A}_m \circ \cdots \circ \mathcal{A}_m}_{k \text{ fois}}.$$

En particulier, $\mathcal{A}_m^0$ est la fonction identité.

Lemme I.4.8. *Les fonctions $\mathcal{A}_m^k$ sont toutes strictement croissantes. De plus, pour tous m, n, k, ℓ, on a*

- $\mathcal{A}_m^{k+1}(n) > \mathcal{A}_m^k(n)$,
- $\mathcal{A}_m^k(n) \geq n$,
- *si $m \leq p$, alors $\mathcal{A}_m^k(n) \leq \mathcal{A}_p^k(n)$.*

Démonstration. La preuve est immédiate et découle principalement du fait que $\mathcal{A}_m$ est un fonction strictement croissante. La dernière assertion découle du lemme I.4.7.

■

Pour montrer que la fonction d'Ackermann n'est pas primitive récursive, nous avons besoin d'introduire la notion suivante.

Définition I.4.9. Soient $f \in \mathfrak{F}_1$ et $g \in \mathfrak{F}_p$. La fonction f *domine* g, ce que l'on notera $g \prec f$, s'il existe une constante C telle que

$$\forall (x_1, \ldots, x_p) \in \mathbb{N}^p : \; g(x_1, \ldots, x_p) \leq f(\sup(x_1, \ldots, x_p, C)).$$

Remarque I.4.10. Si $f \in \mathfrak{F}_1$ est une fonction strictement croissante, alors $g \prec f$ si et seulement si $g(x_1, \ldots, x_p) \leq f(\sup(x_1, \ldots, x_p))$ sauf pour un nombre fini de points. La vérification est immédiate.

La condition est nécessaire. Le nombre de p-uples $(x_1, \ldots, x_p) \in \mathbb{N}^p$ tels que $\sup(x_1, \ldots, x_p) < C$ est majoré par C^p et est donc fini.

Réciproquement, soient $\overline{a_1}, \ldots, \overline{a_t}$ les points (en nombre fini) tels que $g(\overline{a_i}) > f(\sup(\overline{a_i}))$. On pose $C = \sup\{g(\overline{a_1}), \ldots, g(\overline{a_t})\}$. Puisque f est strictement croissante, il existe D tel que pour tout $x \geq D$, $f(x) > C$. Ainsi, pour tout $\overline{x} \in \mathbb{N}^p$, $g(\overline{x}) \leq f(\sup(\overline{x}, D)$ et $g \prec f$.

Pour $m \geq 0$, on pose

$$\mathfrak{C}_m = \{g \in \mathfrak{F} \mid \exists k \in \mathbb{N} : g \prec \mathcal{A}_m^k\}$$

comme étant l'ensemble des fonctions dominées par un itéré de la fonction $\mathcal{A}_m$ et

$$\mathfrak{C} = \bigcup_{m \geq 0} \mathfrak{C}_m.$$

Remarquons que les fonctions primitives récursives de base appartiennent toutes à $\mathfrak{C}_0$. En outre, il est évident que si $f, g \in \mathfrak{F}_p$, si $g \in \mathfrak{C}_m$ et si

$$\forall (x_1, \ldots, x_p) \in \mathbb{N}^p : f(x_1, \ldots, x_p) \leq g(x_1, \ldots, x_p),$$

alors f appartient aussi à $\mathfrak{C}_m$.

Remarque I.4.11. Il est aisé de se convaincre que les fonctions

- $\sup : (x_1, \ldots, x_p) \mapsto \sup(x_1, \ldots, x_p)$,
- $\Sigma_2 : (m, n) \mapsto m + n$,
- pour $k \in \mathbb{N}$, $n \mapsto k.n$

appartiennent toutes à $\mathfrak{C}_2$. Cette observation nous sera utile dans la preuve du lemme I.4.14.

Notre but est de montrer que $\mathfrak{C}$ contient l'ensemble $\mathcal{PR}$. Puisque $\mathfrak{C}$ contient les fonctions primitives récursives de base, il nous suffit de vérifier que $\mathfrak{C}$ est stable par composition et récursion primitive.

Lemme I.4.12. *Pour tout $m \geq 0$, l'ensemble $\mathfrak{C}_m$ est stable par composition. En particulier, $\mathfrak{C}$ l'est aussi.*

Démonstration. Soient $f_1, \ldots, f_n$ des fonctions de $\mathfrak{F}_p \cap \mathfrak{C}_m$ et une fonction $g \in \mathfrak{F}_n \cap \mathfrak{C}_m$. Par définition de l'ensemble $\mathfrak{C}_m$, il existe des constantes $k, k_1, \ldots, k_n$ et $C, C_1, \ldots, C_n$ telles que

$$\forall (y_1, \ldots, y_n) \in \mathbb{N}^n : \; g(y_1, \ldots, y_n) \leq \mathcal{A}_m^k(\sup(y_1, \ldots, y_n, C))$$

et pour tout $i \in \{1, \ldots, n\}$, on a

$$\forall (x_1, \ldots, x_p) \in \mathbb{N}^p : \; f_i(x_1, \ldots, x_p) \leq \mathcal{A}_m^{k_i}(\sup(x_1, \ldots, x_p, C_i)).$$

Posons $D = \sup(C, C_1, \ldots, C_n)$ et $K = \sup(k_1, \ldots, k_n)$, il vient

$$\begin{array}{rl} & g(f_1(x_1, \ldots, x_p), \ldots, f_n(x_1, \ldots, x_p)) \\ \leq & \mathcal{A}_m^k(\sup(f_1(x_1, \ldots, x_p), \ldots, f_n(x_1, \ldots, x_p), C) \\ \leq & \mathcal{A}_m^k(\mathcal{A}_m^K \sup(x_1, \ldots, x_p, D)) \\ \leq & \mathcal{A}_m^{k+K}(\sup(x_1, \ldots, x_p, D)) \end{array}$$

où, à l'avant-dernière ligne, on a utilisé le lemme I.4.8.

■

Avant de passer au cas de la stabilisation par récursion primitive, un petit lemme est encore nécessaire.

Lemme I.4.13. *Pour tous m, n, k, on a*

$$\mathcal{A}_m^k(n) \leq \mathcal{A}_{m+1}(n + k).$$

Démonstration. On procède par récurrence sur k. Le cas de base $k = 0$ est immédiat. Supposons le résultat satisfait pour k et vérifions-le pour $k+1$. Il vient, en appliquant l'hypothèse de récurrence,

$$\mathcal{A}_m^{k+1}(n) = \mathcal{A}_m(\mathcal{A}_m^k(n)) \leq \mathcal{A}_m(\mathcal{A}_{m+1}(n + k)) = \mathcal{A}_{m+1}(n + k + 1),$$

la dernière égalité étant la définition même de la fonction d'Ackermann.

■

Lemme I.4.14. *Si $g \in \mathfrak{F}_p$ et $h \in \mathfrak{F}_{p+2}$ sont deux fonctions de $\mathfrak{C}_m$, alors la fonction f obtenue par récursion primitive à partir de g et de h appartient à $\mathfrak{C}_{m+1}$. En particulier, $\mathfrak{C}$ est stable par récursion primitive.*

Démonstration. Soient $g \in \mathfrak{F}_p$ et $h \in \mathfrak{F}_{p+2}$ deux fonctions de $\mathfrak{C}_m$. On considère la fonction f définie par

$$\begin{cases} f(x_1, \ldots, x_p, 0) = g(x_1, \ldots, x_p) \\ f(x_1, \ldots, x_p, n+1) = h(x_1, \ldots, x_p, n, f(x_1, \ldots, x_p, n)). \end{cases}$$

Par définition de l'ensemble $\mathfrak{C}_m$, il existe des constantes C_1, C_2, k_1, k_2 telles que

$$\forall (x_1, \ldots, x_p) \in \mathbb{N}^p : \; g(x_1, \ldots, x_p) \leq \mathcal{A}_m^{k_1}(\sup(x_1, \ldots, x_p, A_1))$$

et $\forall (x_1, \ldots, x_p, x_{p+1}, x_{p+2}) \in \mathbb{N}^{p+2}$:

$$h(x_1, \ldots, x_p, x_{p+1}, x_{p+2}) \leq \mathcal{A}_m^{k_2}(\sup(x_1, \ldots, x_p, x_{p+1}, x_{p+2}, A_2)).$$

On montre tout d'abord par récurrence sur n que

$$f(x_1, \ldots, x_p, n) \leq \mathcal{A}_m^{k_1+nk_2}(\sup(x_1, \ldots, x_p, n, A_1, A_2)). \tag{1}$$

Le résultat est vrai pour $n = 0$. Supposons-le satisfait pour n et vérifions-le pour $n+1$. Il vient

$$f(x_1, \ldots, x_p, n+1) \leq \mathcal{A}_m^{k_2}(\sup(x_1, \ldots, x_p, n, f(x_1, \ldots, x_p, n), A_2)).$$

En appliquant l'hypothèse de récurrence, on trouve

$$\begin{aligned} f(x_1, \ldots, x_p, n+1) &\leq \mathcal{A}_m^{k_2}(\mathcal{A}_m^{k_1+nk_2}(\sup(x_1, \ldots, x_p, n, A_1, A_2))) \\ &\leq \mathcal{A}_m^{k_1+(n+1)k_2}(\sup(x_1, \ldots, x_p, n, A_1, A_2)) \\ &\leq \mathcal{A}_m^{k_1+(n+1)k_2}(\sup(x_1, \ldots, x_p, n+1, A_1, A_2)) \end{aligned}$$

Nous pouvons à présent conclure. Par le lemme précédent appliqué à (1),

$$f(x_1, \ldots, x_p, n+1) \leq \mathcal{A}_{m+1}(\sup(x_1, \ldots, x_p, n, A_1, A_2) + k_1 + (n+1)k_2)$$

Au vu de la remarque I.4.11, il est clair[9] que la fonction du membre de droite appartient à $\mathfrak{C}_{m+1}$, d'où la même conclusion pour f.

■

Proposition I.4.15. *On a $\mathcal{PR} \subseteq \mathfrak{C}$.*

Démonstration. C'est une conséquence des lemmes I.4.12 et I.4.14.

■

Théorème I.4.16. *La fonction d'Ackermann n'est pas primitive récursive.*

Démonstration. On procède par l'absurde. Supposons $\mathcal{A}$ de classe $\mathcal{PR}$. La fonction $f : \mathbb{N} \to \mathbb{N} : n \mapsto \mathcal{A}(n, 2n)$ est alors elle aussi de classe $\mathcal{PR}$. Au vu de la proposition précédente ($f \in \mathfrak{C}$), il existe des constantes C, m et k telles que pour tout $n > C$, $\mathcal{A}(n, 2n) \leq \mathcal{A}_m^k(n)$. Ainsi, en appliquant le lemme I.4.13, pour tout $n > C$,

$$\mathcal{A}(n, 2n) \leq \mathcal{A}_m^k(n) \leq \mathcal{A}_{m+1}(n+k).$$

[9] C'est la composée d'une fonction appartenant à $\mathfrak{C}_{m+1}$ (à savoir $\mathcal{A}_{m+1}$) avec une fonction de $\mathfrak{C}_2$. Pour $m \geq 1$, elle appartient donc à $\mathfrak{C}_{m+1}$ au vu du lemme I.4.12. Pour $m = 0, 1$, elle appartient à $\mathfrak{C}_2$.

D'autre part, si $n > \sup(C, k, m+1)$, alors

$$\mathcal{A}_{m+1}(n+k) < \mathcal{A}_{m+1}(2n) < A_n(2n) = \mathcal{A}(n, 2n)$$

d'où une contradiction.

■

Nous avons donc trouvé en la fonction d'Ackermann une fonction calculable (au sens naïf du terme, i.e., pour laquelle on dispose d'une procédure de calcul) qui n'est pas primitive récursive. Une méthode de *diagonalisation* permet, elle aussi, d'assurer l'existence d'une fonction calculable non primitive récursive.

Commençons par un argument de comptage. Tout d'abord, il est clair que l'ensemble $\mathcal{PR}$ est dénombrable. En effet, une fonction $\mathcal{PR}$ peut être donnée par une chaîne finie de caractères[10] (à savoir sa définition en termes de fonctions initiales, de composition et de récursion primitive). Par contre, l'ensemble $\mathfrak{F}$ des fonctions n'est pas dénombrable. Pour s'en convaincre, il est facile de voir qu'il contient un sous-ensemble non dénombrable. L'ensemble des fonctions de $\mathfrak{F}_1$ à valeurs dans $\{0,1\}$ est non dénombrable puisqu'il peut être mis en bijection avec l'intervalle $[0,1]$ qui est non dénombrable (les détails sont rappelés au lecteur dans la section suivante : à une suite $(x_n)_{n\in\mathbb{N}} \in \{0,1\}^{\mathbb{N}}$, on associe le nombre réel $\sum_{i=0}^{\infty} x_i\, 2^{-i-1}$; il faut juste être soigneux en éliminant les suites se terminant par une infinité de 1).

Théorème I.4.17. *Il existe une fonction calculable (au sens naïf du terme) qui n'est pas primitive récursive.*

Démonstration. [11] L'ensemble $\mathcal{PR}$ étant en bijection avec $\mathbb{N}$, on peut énumérer ses éléments comme suit : $f_0, f_1, f_2, \ldots$. Puisque ces fonctions peuvent avoir un nombre arbitraire d'arguments, on notera $f_i(n)$ la valeur

$$f_i(\underbrace{n, \ldots, n}_{p \text{ fois}}),$$

[10] L'ensemble des mots sur un alphabet fini est dénombrable. Rappelons qu'un ensemble est dénombrable si et seulement si ses éléments peuvent être caractérisés par un nombre fini d'indices prenant leurs valeurs dans des ensembles dénombrables. En considérant un alphabet fini contenant les symboles $\circ$, σ, $\mathcal{P}$, ",", $0, \ldots, 9$, ";",), (, ... et si on emploie des conventions de codage pour, par exemple, écrire un schéma de récursion primitive sur une seule ligne en séparant les deux champs par un point-virgule, il est possible de décrire la définition de toute fonction $\mathcal{PR}$ au moyen d'une chaîne de caractères.

[11] Cette preuve est tirée de P. Wolper, *Introduction à la calculabilité*, InterEditions, Paris, 1991.

si $f_i \in \mathfrak{F}_p$. (On suppose que cette notation est encore valable lorsque f appartient à $\mathfrak{F}_0$.) Considérons le tableau A

A	0	1	2	$\cdots$	j	$\cdots$
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$\cdots$	$f_0(j)$	$\cdots$
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$\cdots$	$f_1(j)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	
f_i	$f_i(0)$	$f_i(1)$	$f_i(2)$	$\cdots$	$f_i(j)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\ddots$

d'où le terme de diagonalisation.

L'élément $A_{i,j}$ du tableau contient la valeur $f_i(j)$. On définit la fonction

$$g(n) := f_n(n) + 1 = A_{n,n} + 1.$$

Cette fonction est calculable mais n'est pas primitive récursive. En effet, si g était $\mathcal{PR}$, puisqu'on a énuméré les fonctions $\mathcal{PR}$, elle devrait être égale à f_k pour un certain k et dès lors, on devrait avoir $g(k) = f_k(k)$; or, par définition, $g(k) = f_k(k) + 1$. Ceci est impossible.

D'autre part, g est calculable.

- On peut énumérer l'ensemble $\mathcal{PR}$ comme suit. Il est possible d'énumérer toutes les chaînes de caractères sur un alphabet fini adéquatement choisi et de ne garder que celles qui correspondent à la définition d'une fonction $\mathcal{PR}$. (On se convainc facilement de l'existence d'un tel algorithme; on peut tester si une chaîne de caractères donne une définition valide de fonction. En effet, il suffit de satisfaire certaines règles de syntaxe facilement exprimables. Par exemple, aucun préfixe ne peut contenir plus de symboles ")" que "(", etc....)
- Une fois f_n obtenu, on l'évalue en n (ce qui est toujours possible car f_n étant $\mathcal{PR}$, f_n est calculable).
- Il ne reste plus qu'à calculer $f_n(n) + 1$ pour obtenir la valeur de $g(n)$.

■

5. Caractère non dénombrable

La propriété suivante est une propriété élémentaire rencontrée classiquement dans les cours d'analyse.

Lemme I.5.1. *Pour tout $n \in \mathbb{N}$, $I_n = [a_n, b_n]$ est un fermé de $\mathbb{R}$. Si $I_1 \supseteq I_2 \supseteq I_3 \supseteq \cdots$, alors l'intersection*

$$\bigcap_{i=0}^{\infty} I_n$$

est non vide.

Démonstration. La suite $(a_n)_{n\geq 0}$ est croissante et majorée. En effet, pour tout n_0 fixé, $a_n \leq b_{n_0}$ pour tout $n \geq 0$ (autrement dit, b_{n_0} est un majorant des éléments de la suite $(a_n)_{n\geq 0}$). Soit

$$\alpha = \sup\{a_n \mid n \geq 0\}$$

le plus petit majorant des éléments de la suite $(a_n)_{n\geq 0}$. En particulier, $a_n \leq \alpha$ pour tout $n \geq 0$. Puisque pour tout $n \geq 0$, b_n est un majorant de $\{a_n \mid n \geq 0\}$, on a aussi $\alpha \leq b_n$ pour tout n et donc $\alpha \in \bigcap_{i=0}^{\infty} I_n$.

■

Proposition I.5.2. *L'ensemble $\mathbb{R}$ est non dénombrable.*

Démonstration. Procédons par l'absurde et supposons qu'il existe une bijection f de $\mathbb{N}$ dans $\mathbb{R}$. Ainsi, on peut énumérer les éléments de $\mathbb{R}$, $x_1 = f(1)$, $x_2 = f(2)$, ... Soit I_1 un intervalle de $\mathbb{R}$ ne contenant pas x_1. Soit I_2 un intervalle de $\mathbb{R}$ inclus dans I_1 et ne contenant pas x_2. Ainsi, de proche en proche, on définit pour tout $n \geq 0$ un intervalle I_{n+1} inclus dans I_n et ne contenant pas x_{n+1}.

On remarque que l'intersection $\cap_{i=0}^{\infty} I_n$ ne peut, par construction, contenir aucun x_i car $x_i \notin I_i$ et donc $x_i \notin \cap_{i=0}^{\infty} I_n$. Autrement dit, $\cap_{i=0}^{\infty} I_n \cap \mathbb{R} = \emptyset$, alors que par le lemmme précédent, $\cap_{i=0}^{\infty} I_n$ doit au moins contenir un réel.

■

Proposition I.5.3. *L'intervalle $]0,1[$ est non dénombrable.*

Démonstration. Procédons par l'absurde et supposons qu'il existe une bijection f de $\mathbb{N}$ dans $]0,1[$. Ainsi, on peut énumérer les éléments de $]0,1[$, $x_1 = f(1)$, $x_2 = f(2)$, ... et considérer leur développement décimal[12],

$$f(k) = \sum_{i=1}^{+\infty} a_i^{(k)} 10^{-i} = .a_1^{(k)} a_2^{(k)} a_3^{(k)} a_4^{(k)} \cdots .$$

Appliquons à présent un argument de "diagonalisation" et définissons le nombre réel $\alpha \in]0,1[$ dont le développement décimal

$$\alpha = .b_1 b_2 b_3 b_4 \cdots$$

est défini par

$$b_i = \begin{cases} 2 & \text{si } a_i^{(i)} \neq 2 \\ 3 & \text{si } a_i^{(i)} = 2 \end{cases}$$

Puisque $\alpha \in]0,1[$, il existe n tel que $f(n) = \alpha$. Regardons l'élément $a_n^{(n)}$ du développement décimal de $f(n)$. Si $a_n^{(n)} = 2$, alors $b_n = 3$, une absurdité. De même, si $a_n^{(n)} \neq 2$, alors $b_n = 2$, ce qui conduit à la même absurdité.

■

Aurait-on pu adapter la preuve précédente à $\mathbb{Q}$, alors que cet ensemble est dénombrable ?

[12] Lorsqu'on dispose de deux développements possibles se terminant soit par une infinité de 9, soit par une infinité de 0, on choisit ce dernier développement.

6. Fonctions récursives

Nous introduisons à présent un nouveau schéma de construction de fonctions. Il s'agit du schéma μ non borné (ou de minimisation non bornée). Cela débouche sur la définition de l'ensemble $\mathcal{R}$ des fonctions récursives qui contient strictement l'ensemble $\mathcal{PR}$ (en effet, la fonction d'Ackermann est récursive). On remarquera une fois encore que toute fonction récursive est calculable par une procédure effective. La définition suivante est fort proche du schéma de minimisation borné.

Définition I.6.1 (Schéma de minimisation). Soit $A \subseteq \mathbb{N}^{p+1}$. On définit la fonction $f \in \mathfrak{F}_p$ comme[13]

$$f(x_1, \ldots, x_p) = \begin{cases} 0, \text{ si } \{t \mid (x_1, \ldots, x_p, t) \in A\} = \emptyset; \\ \inf\{t \mid (x_1, \ldots, x_p, t) \in A\}, \text{ sinon.} \end{cases}$$

On note cette fonction

$$f(x_1, \ldots, x_p) = \mu_t((x_1, \ldots, x_p, t) \in A).$$

Bien évidemment, on dispose d'une définition analogue en termes de prédicats d'arité p.

Nous avons vu que la minimisation bornée appliquée à des ensembles primitifs récursifs fournit des fonctions primitives récursives. La définition donnée ici est plus problématique. En effet, si on n'y prend garde, on peut même construire des fonctions non calculables. En effet, sans aucune précaution sur l'ensemble A, il se peut que l'on n'obtienne jamais la valeur de $f(x_1, \ldots, x_p)$. Imaginez vouloir calculer

$$f(x_1, \ldots, x_p) = \mu_t((x_1, \ldots, x_p, t) \in A).$$

Pour rechercher le plus petit entier t tel que $(x_1, \ldots, x_p, t)$ appartienne à A, on teste d'abord si $(x_1, \ldots, x_p, 0)$ appartient à A. S'il n'appartient pas à A, on considère alors $(x_1, \ldots, x_p, 1)$ et ainsi de suite. Si on trouve une valeur de t satisfaisant $(x_1, \ldots, x_p, t) \in A$, la procédure s'achève; sinon, on n'obtiendra jamais la valeur de $\mu_t((x_1, \ldots, x_p, t) \in A)$! En effet, puisqu'on ne fixe ici aucune borne à ne pas dépasser, on ne pourra jamais décider d'arrêter des recherches infructueuses.

C'est pour cette raison que nous introduisons la définition suivante.

Définition I.6.2. Une partie $A \subseteq \mathbb{N}^{p+1}$ est dite *sûre* si

$$\forall (x_1, \ldots, x_p) \in \mathbb{N}^p, \exists t \in \mathbb{N} : (x_1, \ldots, x_p, t) \in A.$$

On a bien évidemment une définition analogue pour un prédicat sûr.

Dans la suite, on considérera uniquement des parties ou des prédicats sûrs. De cette manière, le schéma de minimisation (non borné) fournira

[13] On pourrait simplifier cette définition en posant $\inf \emptyset = 0$.

toujours des fonctions calculables. Pour cette raison, on parle parfois du schéma μ *total*[14]

Définition I.6.3. L'ensemble $\mathcal{R}$ des *fonctions récursives* est le plus petit sous-ensemble de $\mathfrak{F}$ qui contient les fonctions primitives récursives de base et qui est stable pour la composition, la récursion primitive et la minimisation (non bornée de parties sûres).

Remarque I.6.4. Puisqu'on ne considère que des parties sûres, le lecteur aura déjà été convaincu par nos propos que toute fonction récursive est calculable par une procédure effective. Pour répondre à la question posée en début de chapitre de savoir quelles fonctions de $\mathfrak{F}$ sont calculables par un algorithme, un réponse possible est donnée par les fonctions de $\mathcal{R}$. Il nous faudra encore patienter un peu pour s'en convaincre. En fait, il nous faudra modéliser ce que l'on entend par procédure effective et montrer que l'ensemble des fonctions calculables dans ce formalisme coïncide avec $\mathcal{R}$.

On peut montrer que la fonction d'Ackermann est récursive[15]. Ainsi, on a

$$\mathcal{PR} \subsetneq \mathcal{R}.$$

[14]En fait, il est aussi possible d'introduire la notion de fonction partielle, i.e., dont le domaine de définition est un sous-ensemble strict de $\mathbb{N}^p$. On peut alors étudier la calculabilité des fonctions partielles. Nous avons décidé d'éviter ces raffinements dans ce cours introductif. Par exemple, pour pouvoir composer des fonctions partielles, il faut être soigneux dans la caractérisation de l'ensemble de définition de la fonction obtenue.

[15]Il s'agit par exemple d'un (long) exercice commenté dans Cori et Lascar, p.57, exercice 11.

CHAPITRE II

Machines de Turing

> "Living backwards!" Alice repeated in great astonishment. "I never heard of such a thing!"
>
> "—but there's one great advantage in it, that one's memory works both way."
>
> "I'm sure *mine* only works one way," Alice remarked. "I ca'n't remember things before they happen."
>
> "It's a poor sort of memory that only works backwards," the Queen remarked.
>
> Through the looking-glass, Lewis Carroll

Jusqu'à présent, nous n'avons jamais *formalisé* la notion d'algorithme (ou de procédure effective de calcul). Pour ce faire, on introduit dans ce chapitre les machines de Turing[1] comme modélisation des procédures effectives. Bien que nous n'ayons pas encore présenté et défini ce nouveau concept, nous pouvons d'ores et déjà faire quelques remarques qui seront explicitées tout au long du présent chapitre.

Il ne nous sera **pas** possible de démontrer rigoureusement que les machines de Turing modélisent exactement le concept de procédure effective. En effet, pour démontrer l'équivalence des deux notions, il faudrait disposer d'une formalisation de la notion de procédure effective et c'est bien cette formalisation qui nous fait défaut ! Il nous faudra donc justifier la formalisation adoptée et ces justifications permettront de convaincre le lecteur de se rallier à la thèse de Church-Turing.

Une première justification de ce choix est que toutes les autres tentatives employées jusqu'à présent pour formaliser la notion de procédure effective ont été démontrées équivalentes aux machines de Turing. En particulier, il est possible de considérer des machines de Turing plus générales que celles introduites dans ce chapitre (par exemple, utiliser plusieurs bandes mémoire, un ruban bi-infini ou encore des machines non déterministes). Ces machines *a priori* plus générales ont en fait la même puissance de calcul que les machines que nous introduirons (elles calculent exactement le même ensemble de fonctions).

[1] Alan Turing, mathématicien anglais, peut être considéré comme l'un des pères fondateurs de l'informatique, 1912–1954. Il s'intéressa notamment à la mécanique quantique, aux probabilités, à la biologie ou encore à la cryptographie.

En particulier, nous admettrons la thèse de Church[2]-Turing : *Les langages reconnus par une procédure effective sont ceux décidés par une machine de Turing.* Ainsi, cette thèse (qui n'est pas un théorème !) revient à choisir les machines de Turing comme modélisation du concept de procédure effective. Le lecteur peu convaincu pourra se prêter à l'exercice suivant : imaginer une autre définition de la notion de procédure effective puis démontrer que toute fonction calculable à l'aide de cette nouvelle définition l'est aussi en termes de machine de Turing.

Enfin, un deuxième argument de poids est que, si on considère la formalisation des fonctions calculables en termes de fonctions récursives, alors cette dernière coïncide exactement avec celle définie par machine de Turing (c'est un résultat important du chapitre). Ainsi, une formalisation *a priori* complètement différente du concept d'algorithme se ramène encore aux machines de Turing.

1. Quelques définitions

Définition II.1.1. Un *alphabet* est un ensemble fini. Ses éléments sont appelés *lettres* ou *symboles*. On aura le plus souvent recours aux lettres grecques majuscules pour dénoter un alphabet. Un *mot fini* sur l'alphabet Σ est une suite finie de lettres. Par exemple, si $\Sigma = \{a, b\}$, $abba$, bab et $bbbb$ sont des mots sur Σ. La longueur d'un mot w est dénotée par $|w|$. Ainsi, $|abba| = 4$, $|bab| = 3$ et $|bbbb| = 4$. Le seul mot de longueur 0 correspond à la suite vide et est noté ε. Il s'agit du *mot vide*. L'ensemble de tous les mots sur Σ est noté Σ^*. On définit sur Σ^* une opération binaire interne et partout définie, la *concaténation*, comme suit : si $u = u_1 \cdots u_k$ et $v = v_1 \cdots v_\ell$, $u_i, v_j \in \Sigma$, alors la concaténation de u et de v, notée simplement uv, est le mot w défini par

$$w = w_1 \cdots w_{k+\ell} \text{ où } w_i = \begin{cases} u_i \text{ si } i \leq k, \\ v_{i-k} \text{ si } k < i \leq k + \ell. \end{cases}$$

Par exemple, si $u = abba$ et $v = bb$, alors uv est simplement $abbabb$. En particulier, la concaténation de n copies du mot u se note u^n et on pose $u^0 = \varepsilon$. Par exemple, $(aba)^3 = abaabaaba$.

Remarque II.1.2. L'ensemble Σ^* muni de l'opération de concaténation est un monoïde non commutatif ayant ε comme neutre. L'application $|\cdot| : \Sigma^* \to \mathbb{N} : u \mapsto |u|$ est un homomorphisme de monoïdes (on considère $\mathbb{N}$ muni de l'addition d'entiers). Il s'agit d'un isomorphisme si et seulement si Σ est restreint à une seule lettre.

Définition II.1.3. Un *langage* sur Σ est une partie de Σ^*, i.e., un ensemble de mots sur Σ. Par exemple, si $\Sigma = \{a, b\}$, on peut considérer le langage L formé des mots ayant un nombre pair de a,

$$L = \{\varepsilon, b, bb, \ldots, aa, aab, aba, baa, aabb, abab, abba, \ldots, aabbbaab, \ldots\}.$$

[2]Alonzo Church, logicien américain et père du λ-calcul, 1903-1995.

Définition II.1.4. Un *mot infini* (à droite) sur Σ est simplement une suite $(x_n)_{n\in\mathbb{N}} \in \Sigma^{\mathbb{N}}$. Par exemple,

$$ababbab\cdots$$

est un mot infini. Si $u \in \Sigma^*$ est un mot fini, on note u^ω le mot infini obtenu en concaténant une infinité de copies de u. Ainsi,

$$(ab)^\omega = abababbab\cdots \text{ et } a^\omega = aaaaaa\cdots .$$

En particulier, l'ensemble des mots infinis sur Σ est noté Σ^ω. On peut bien sûr encore définir[3] la concaténation d'un mot fini $u \in \Sigma^*$ et d'un mot infini $v \in \Sigma^\omega$. Enfin, on note $\Sigma^\infty = \Sigma^\omega \cup \Sigma^*$, l'ensemble des mots finis ou infinis sur Σ.

Dans la littérature, il existe plusieurs variantes permettant de définir une machine de Turing et il est souvent aisé de montrer leur équivalence avec le modèle choisi ici.

Définition II.1.5. Nous définissons une *machine de Turing* comme suit. Il s'agit d'un 6-uple $\mathcal{M} = (Q, q_0, F, \Sigma, \Gamma, \delta)$ où Q est un ensemble fini d'*états*, $q_0 \in Q$ est un état privilégié, l'*état initial*, $F \subseteq Q$ est l'ensemble des *états accepteurs*, Σ est l'*alphabet d'entrée* et Γ est l'*alphabet de ruban* ($\Sigma \subset \Gamma$). On suppose qu'on dispose d'un symbole privilégié $\# \in \Gamma \setminus \Sigma$ appelé *symbole blanc* et de deux symboles L et R n'appartenant pas à Γ. Généralement, on prendra simplement $\Gamma = \Sigma \cup \{\#\}$. Enfin, la *fonction de transition* est

$$\delta : Q \times \Gamma \to Q \times (\Gamma \cup \{L, R\}).$$

Pour représenter une machine de Turing : les états sont représentés par des cercles, l'état initial est marqué d'une flèche entrante, les états accepteurs d'un double cercle et si $\delta(p, \gamma) = (q, x)$, $p, q \in Q$, $\gamma \in \Gamma$ et $x \in \Gamma \cup \{L, R\}$, alors on trace :

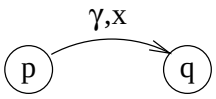


FIGURE II.1. Une transition.

La machine de Turing dispose en outre d'un ruban[4] mémoire $m \in \Gamma^\omega$ dont les cases sont numérotées à partir de 0. La j-ième case du ruban m sera simplement notée m_j. De plus, seul un nombre fini de symboles diffèrent de $\#$, i.e., on impose $m \in \Gamma^*\#^\omega$. A tout moment, un curseur $k \in \mathbb{N}$ référence une case du ruban, à savoir la case m_k.
Ainsi, une *configuration mémoire* est un couple $(m, k) \in \Gamma^*\#^\omega \times \mathbb{N}$. En général, on représente une configuration mémoire par sa *partie significative*. Ainsi, si $m = u\#^\omega$ avec $u = u_0 \cdots u_\ell \in \Gamma^*$ ne se terminant pas par $\#$ (i.e.,

[3] La définition rigoureuse est triviale et laissée au lecteur.
[4] Un ruban est donc simplement un mot infini.

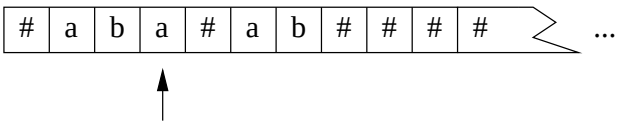


FIGURE II.2. Un ruban mémoire et son curseur.

$u = \varepsilon$ ou bien $u \neq \varepsilon$ et sa dernière lettre u_ℓ appartient à Σ), alors la partie significative de (m, k) est notée

$$u_0 \cdots u_{k-1}\underline{u_k}u_{k+1} \cdots u_\ell \quad \text{si } k \leq \ell$$

et

$$u_0 \cdots u_\ell \underbrace{\#\cdots\#}_{k-\ell-1}\underline{\#} \quad \text{si } k > \ell.$$

Le soulignement sert à repérer la cellule référencée par le curseur. Par exemple, la *partie significative* de $(aba\#^\omega, 1)$ (resp. $(aba\#^\omega, 5)$ est $a\underline{b}a$ (resp. $aba\#\#\underline{\#}$).

Enfin, une *configuration machine* est un triplet (q, m, k) où $q \in Q$ et (m, k) est une configuration mémoire. Si r est la partie significative de la configuration mémoire (m, k), on s'autorisera souvent à écrire $q.r$ au lieu de (q, m, k).

Il nous reste à expliciter les transitions définies par δ. La machine $\mathcal{M}$ *passe* d'une configuration (q, m, k) où $q \in Q \setminus F$ à une configuration (q', m', k'), ce que l'on note $(q, m, k) \vdash (q', m', k')$, de la manière suivante :

- Si $\delta(q, m_k) = (p, \sigma)$, $\sigma \in \Gamma$, alors $q' = p$, $k = k'$ et $m = m'$ sauf en position k où $m'_k = \sigma$; en d'autres termes, on remplace la case du ruban référencée par k,

$$q.u\underline{\tau}v \vdash p.u\underline{\sigma}v.$$

- Si $\delta(q, m_k) = (p, R)$, alors $q' = p$, $m = m'$ et $k' = k+1$; en d'autres termes, on ne modifie pas le ruban mais on déplace le curseur de référence d'une case vers la droite,

$$q.u\underline{\sigma}\tau v \vdash p.u\sigma\underline{\tau}v.$$

- Si $\delta(q, m_k) = (p, L)$, alors $q' = p$, $m = m'$ et $k' = k-1$; en d'autres termes, on ne modifie pas le ruban mais on déplace le curseur de référence d'une case vers la gauche,

$$q.u\sigma\underline{\tau}v \vdash p.u\underline{\sigma}\tau v.$$

Notons que ce passage n'est pas défini si $q \in F$. On décide que se trouvant dans un état accepteur, la machine $\mathcal{M}$ *s'arrête*. Le passage n'est pas non plus défini pour les triplets (q, m, k) tels que $k = 0$ et $\delta(q, m_k) = (p, L)$. Dans cette dernière situation, on parle de *configuration pendante*. On note $\vdash^*$, la clôture réflexive et transitive de $\vdash$. En d'autres termes, on notera $(q, m, k) \vdash^* (q', m', k')$ si

$$(q, m, k) \vdash (q_1, m_1, k_1) \vdash \cdots \vdash (q_r, m_r, k_r) \vdash (q', m', k').$$

Ainsi, à chaque étape, suivant l'état dans lequel se trouve la machine de Turing $\mathcal{M}$, on regarde le contenu de la cellule mémoire référencée et on agit en conséquence : la machine bascule dans un nouvel état et sur le ruban, soit on écrit un nouveau symbole dans la cellule référencée, soit on déplace le curseur de référence d'une case vers la gauche ou la droite. Les opérations sont donc de trois types : lecture, écriture et déplacement.

Exemple II.1.6. Considérons la machine définie par le diagramme donné à la figure II.3. Ses états sont $1, 2, 3$, l'état initial est 2, on a un seul état

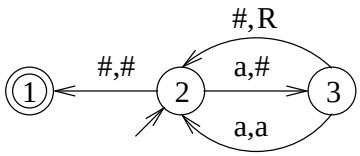


FIGURE II.3. Une machine de Turing.

accepteur 1, son alphabet d'entrée est $\Sigma = \{a\}$, l'alphabet de ruban est $\Gamma = \{a, \#\}$. Supposons disposer de la configuration mémoire $(aa\#a\#^\omega, 0) = \underline{a}a\#a$. Partant de l'état initial 2, on a la suite de configurations machine

$$\begin{aligned} 2.\underline{a}a\#a &\vdash 3.\underline{\#}a\#a \\ &\vdash 2.\#\underline{a}\#a \\ &\vdash 3.\#\underline{\#}\#a \\ &\vdash 2.\#\#\underline{\#}a \\ &\vdash 1.\#\#\underline{\#}a. \end{aligned}$$

On peut donc écrire $2.\underline{a}a\#a \vdash^* 1.\#\#\underline{\#}a$.

2. Fonctions calculables

Intuitivement, si on place sur le ruban mémoire d'une machine de Turing les paramètres d'une fonction f en respectant des conventions de codage, alors on dira que la machine calcule f si partant de l'état initial, on aboutira dans un état d'arrêt avec un ruban mémoire contenant exactement la valeur de f (à des conventions de codage près).

Définition II.2.1. Soit Σ et Δ deux alphabets finis. Une fonction $f : \Sigma^* \times \cdots \times \Sigma^* \to \Delta^* : (w_1, \ldots, w_p) \mapsto f(w_1, \ldots, w_p)$ est *calculable* (plus précisément, *calculable par machine de Turing*) s'il existe une machine de Turing $\mathcal{M} = (Q, q_0, F, \Sigma, \Gamma, \delta)$ telle que $\Sigma \cup \Delta \subset \Gamma$ et pour tout $(w_1, \ldots, w_p) \in (\Sigma^*)^p$, il existe $h \in F$ tel que

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

Le cas des fonctions numériques est particulièrement simple. On utilise l'isomorphisme de monoïdes entre $\mathbb{N}$ et un alphabet unaire $\{u\}$ pour coder les entiers sur un alphabet fini.

Définition II.2.2. En particulier, une fonction $f \in \mathfrak{F}_p$ est *calculable* (plus précisément, *calculable par machine de Turing*) s'il existe une machine de Turing $\mathcal{M} = (Q, q_0, F, \{u\}, \Gamma, \delta)$ telle que pour tout $(x_1, \ldots, x_p) \in \mathbb{N}^p$, il existe $h \in F$ tel que

$$q_0.\#u^{x_1}\# \cdots \#u^{x_p}\underline{\#} \vdash^* h.\#u^{f(x_1,\ldots,x_p)}\underline{\#}.$$

Exemple II.2.3. Considérons la machine de Turing représentée à la figure II.4.

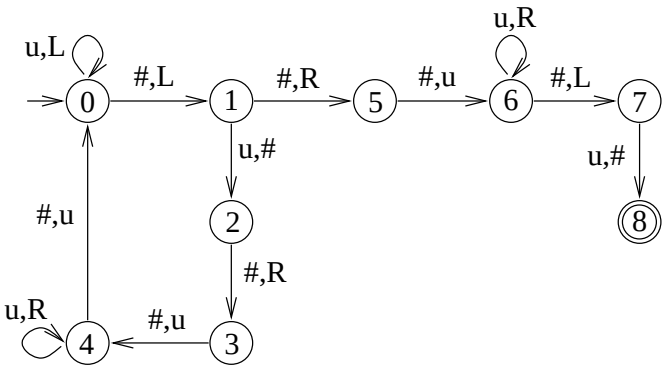


FIGURE II.4. Une machine de Turing calculant $n \mapsto 2n$.

Par convention, la machine démarre dans la configuration $0.\#u^n\underline{\#}$. Pour vérifier que cette machine calcule effectivement la fonction $n \mapsto 2n$, au vu de la définition précédente, nous devons nous convaincre que pour tout $n \geq 0$,

$$0.\#u^n\underline{\#} \vdash^* 8.\#u^{2n}\underline{\#}.$$

Pour le cas $n = 0$, il vient

$$0.\#\underline{\#} \vdash 1.\underline{\#}\# \vdash 5.\#\underline{\#} \vdash 6.\#\underline{u} \vdash 6.\#u\underline{\#} \vdash 7.\#\underline{u} \vdash 8.\#\underline{\#}.$$

Si $n > 0$, alors en démarrant d'une configuration $0.\#u^n\underline{\#}u^m$, pour m quelconque, on passe dans le cycle $(0, 1, 2, 3, 4, 0)$ pour obtenir

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

Il est donc clair que

$$0.\#u^n\underline{\#} \vdash^* 0.\#\underline{\#}u^{2n}$$

et pour conclure,

$$0.\#\underline{\#}u^{2n} \vdash 1.\underline{\#}\#u^{2n} \vdash 5.\#\underline{\#}u^{2n} \vdash 6.\#\underline{u}u^{2n} \vdash 6.\#u^{2n+1}\underline{\#} \vdash 7.\#u^{2n}\underline{u} \vdash 8.\#u^{2n}\underline{\#}.$$

La machine de Turing calcule donc bien (au sens de la définition II.2.2) la fonction $n \mapsto 2n$.

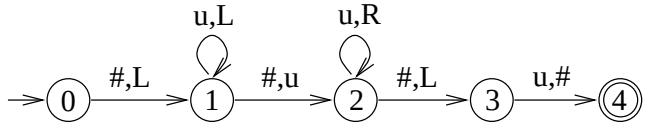


FIGURE II.5. Une machine de Turing calculant Σ_2 : $(m,n) \mapsto m+n$.

Exemple II.2.4. Considérons à présent la machine de la figure II.5. La fonction $\Sigma_2 : (m,n) \mapsto m+n$ est calculable par cette machine. Ainsi, nous devons vérifier que pour tous $m, n \in \mathbb{N}$,

$$0.\#u^m\#u^n\underline{\#} \vdash^* 4.\#u^{m+n}\underline{\#}.$$

Il vient (on suppose ici $n > 0$, le cas $n = 0$ est semblable)

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

Exemple II.2.5. La fonction caractéristique d'un langage $L \subset \Sigma^*$ n'est rien d'autre que la fonction caractéristique d'un sous-ensemble (non pas de $\mathbb{N}$, mais de Σ^*). Ainsi,

$$\chi_L : \Sigma^* \to \{0,1\} : w \mapsto \begin{cases} 1 \text{ si } w \in L, \\ 0 \text{ sinon.} \end{cases}$$

La machine représentée à la figure II.6 calcule la fonction caractéristique du langage sur $\{a,b\}$ formé des mots ayant un nombre pair de a. Cette machine

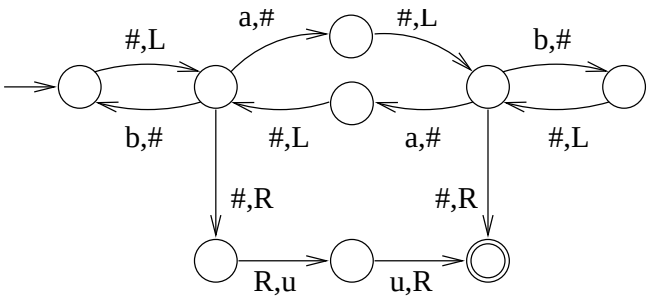


FIGURE II.6. Une machine de Turing calculant χ_L.

efface en partant de la gauche une à une les lettres du mot d'entrée et garde trace, grâce à l'état dans laquelle elle se trouve, de la parité du nombre de a rencontrés.

Ayant eu l'occasion de manipuler quelque peu les machines de Turing, nous pouvons faire quelques remarques quant aux suites de configurations qu'il est possible d'obtenir.

Remarque II.2.6. Lorsqu'on initialise une machine de Turing avec une configuration machine (q_0, m, k) où m est le mot se trouvant sur le ruban mémoire et k la position du curseur, trois situations peuvent apparaître.

1) On aboutit en un nombre fini de transitions dans une configuration (f, m', k') où f est un état accepteur. C'est le cas rencontré dans les exemples précédents lorsqu'on initialisait le ruban en suivant des conventions de formatage ad hoc.

2) On aboutit dans une configuration pendante ou dans une configuration à partir de laquelle la fonction de transition n'est pas définie. Par exemple, pour la machine de la figure II.5, la configuration $0.\underline{u}$ ne donne rien (depuis l'état 0, la fonction de transition précise uniquement ce que la machine peut effectuer lorsque la cellule référencée contient le caractère #). Un exemple de configuration pendante est trivialement obtenu en considérant la machine suivante et en considérant la configuration $0.\underline{\#}$.

FIGURE II.7. Obtention d'une configuration pendante.

3) On peut obtenir une suite infinie de configurations sans jamais atteindre un état accepteur. Dans ce cas, la machine ne s'arrête jamais. Un exemple trivial est de considérer la machine suivante encore une fois à partir

FIGURE II.8. Une machine de Turing ne s'arrêtant pas toujours.

de la configuration $0.\underline{\#}$.

3. Composition de machines de Turing

La construction de machines de Turing calculant une fonction donnée peut très rapidement s'avérer fastidieuse. Pour cette raison, nous allons nous autoriser quelques raccourcis dans l'élaboration de ces machines et nous permettre de construire de nouvelles machines à partir de machines déjà construites.

La première construction, la plus simple, consiste simplement à enchaîner deux machines un peu comme lorsqu'on compose deux fonctions f et g. Pour calculer la valeur $f(g(x))$, on calcule d'abord $g(x)$ puis à partir du résultat obtenu, on peut appliquer la fonction f.

Soient $\mathcal{M}$ et $\mathcal{N}$ deux machines de Turing (pour faire référence à la machine considérée, nous utiliserons les notations $\vdash_{\mathcal{M}}$ et $\vdash_{\mathcal{N}}$). Si l'état initial de $\mathcal{M}$ est q_0 et si $(q_0, m, k) \vdash^*_{\mathcal{M}} (f, m', k')$, pour un état accepteur f de $\mathcal{M}$, alors l'idée de la composition des machines $\mathcal{M}$ et $\mathcal{N}$ est de lancer l'exécution de la machine $\mathcal{N}$ à partir de la configuration mémoire (m', k') obtenue. Si q'_0 est l'état initial de $\mathcal{N}$ et si $(q'_0, m', k') \vdash^* (f', m'', k'')$ avec f' un état accepteur de $\mathcal{N}$, alors on dira que (m'', k'') est le résultat de l'*enchaînement* des machines $\mathcal{M}$ et $\mathcal{N}$ à partir de (m, k). La machine ainsi obtenue sera

simplement notée[5] $\mathcal{M}\mathcal{N}$ ou parfois $\mathcal{M} \longrightarrow \mathcal{N}$. On s'autorise naturellement à écrire $\mathcal{M}^n$ pour l'enchaînement de n copies de la machine de Turing $\mathcal{M}$, $n \geq 1$.

La deuxième construction est la *composition conditionnelle*. On peut décider de subordonner l'exécution de la deuxième machine $\mathcal{N}$ en fonction du contenu de la cellule référencée $m'_{k'}$ obtenu à la fin de l'exécution de $\mathcal{M}$. Ainsi, avec les mêmes notations que précédemment, on peut décider d'effectuer la composition $\mathcal{M}\mathcal{N}$ si et seulement si la cellule référence $m'_{k'}$ contient un certain symbole σ et dès lors, le résultat de cette composition sera

$$\begin{cases} (m'', k'') \text{ si } m'_{k'} = \sigma, \\ (m', k') \text{ si } m'_{k'} \neq \sigma. \end{cases}$$

On notera cette composition $\mathcal{M} \stackrel{\sigma}{\longrightarrow} \mathcal{N}$. On peut aussi considérer une composition conditionnelle où la condition prend la forme

$$\begin{cases} (m'', k'') \text{ si } m'_{k'} = \sigma_1, \ldots, \sigma_t \\ (m', k') \text{ sinon} \end{cases}$$

et dans ce cas, la notation est $\mathcal{M} \stackrel{\sigma_1,\ldots,\sigma_t}{\longrightarrow} \mathcal{N}$. Enfin, la négation de la première condition, c'est-à-dire

$$\begin{cases} (m'', k'') \text{ si } m'_{k'} \neq \sigma, \\ (m', k') \text{ si } m'_{k'} = \sigma \end{cases}$$

sera notée $\mathcal{M} \stackrel{\underline{\sigma}}{\longrightarrow} \mathcal{N}$.

Remarque II.3.1. Lors d'une composition conditionnelle, il se peut que l'on désire conserver le contenu de la cellule référencée pour un usage ultérieur. (Par exemple, pour copier le contenu γ de la cellule référencée deux cases à droite de celle-ci, il faut déplacer la tête de lecture/écriture puis écrire le caractère γ qu'il aura fallu d'une façon ou d'une autre conserver.) Pour ce faire, on notera par exemple une composition conditionnelle par

$$\mathcal{M} \stackrel{\sigma \neq a}{\longrightarrow} \mathcal{N}$$

pour signifier que si la cellule référencée contient un certain symbole σ différent de a, alors on exécute $\mathcal{N}$. Cette notation possède, contrairement à la notation $\mathcal{M} \stackrel{\underline{a}}{\longrightarrow} \mathcal{N}$, l'avantage de donner un nom explicite au contenu de la cellule référencée. Par la suite, on pourra donc faire appel à cette variable σ. Cette construction sera par exemple utile dans la construction du "shift-left k" (voir plus loin). On pourrait se passer de ce raffinement et considérer autant de compositions conditionnelles distinctes qu'il n'y a de lettres dans l'alphabet du ruban, mais cela ne ferait que surcharger les machines considérées.

[5]Remarquons que, pour des fonctions, la notation $f \circ g$ stipule qu'on applique d'abord g puis f tandis que pour des machines de Turing, la notation $\mathcal{M}\mathcal{N}$ signifie qu'on exécute dans l'ordre $\mathcal{M}$ puis $\mathcal{N}$.

Nous avons vu comment composer des machines de Turing par enchaînement ou par composition conditionnelle. Nous voudrions nous autoriser encore plus de libertés en permettant d'itérer et de combiner ces constructions.

Un organigramme est une "méga-machine". Avec beaucoup de patience, on pourrait en refaire une machine de Turing "classique".

Définition II.3.2. Soient $\mathcal{M}_1, \ldots, \mathcal{M}_n$ des machines de Turing. Un *organigramme* sur $\mathcal{M}_1, \ldots, \mathcal{M}_n$ est la donnée d'un graphe pointé[6] dont les sommets sont des mots finis sur $\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ et dont les arcs représentent des compositions conditionnelles. On impose en outre que, pour tout sommet, les arcs issus de ce sommet correspondent à des conditions mutuellement exclusives[7].

Un organigramme représente une machine de Turing dont l'exécution débute par la machine désignée par la racine du graphe et se poursuit conformément aux compositions conditionnelles décrites par les arcs. Chaque sommet qui est un mot sur $\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ correspond à un enchaînement de machines.

Exemple II.3.3. Soient des machines de Turing $\mathcal{M}$, $\mathcal{N}$ et $\mathcal{P}$. On considère l'organigramme représenté à la figure II.9. Pour une configuration mémoire

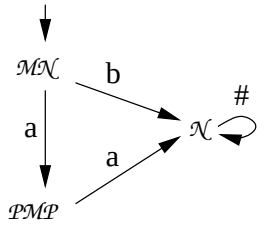


FIGURE II.9. Un organigramme.

donnée, l'exécution de l'organigramme débute par l'enchaînement des machines $\mathcal{M}$ et $\mathcal{N}$. Ensuite, suivant le contenu de la cellule mémoire référencée à la fin de l'exécution de $\mathcal{MN}$, on exécutera l'enchaînement $\mathcal{PMP}$ (si la cellule contient un a) ou $\mathcal{N}$ (si la cellule contient b). Si la cellule contient un autre caractère, l'exécution de l'organigramme s'achève avec l'exécution de $\mathcal{MN}$. De même, à la fin de l'exécution de $\mathcal{PMP}$, si la cellule référencée contient a, on débutera alors l'exécution de $\mathcal{N}$, et ainsi de suite.

Puisque les organigrammes constituent un moyen efficace de construire à moindre frais de nouvelles machines de Turing, nous allons à présent définir quelques *machines de Turing élémentaires* qui serviront de constituants de base à nos organigrammes.

Soit une machine de Turing $(\{q_0, f\}, q_0, \{f\}, \Sigma, \Gamma, \delta)$ possédant deux états. Si pour tout $\sigma \in \Gamma$, la fonction de transition est définie par

$$\delta(q_0, \sigma) = (f, L) \text{ (resp. } (f, R), (f, \gamma)),$$

[6]Cela signifie qu'on distingue un sommet particulier appelé *racine* et représenté par une flèche entrante.

[7]Ceci assure le caractère déterministe de la construction.

alors cette machine, notée $\mathcal{L}$ (resp. $\mathcal{R}$, γ), a simplement pour effet de déplacer la cellule référencée d'une case vers la gauche (resp. de déplacer la cellule référencée d'une case vers la droite, d'écrire $\gamma \in \Gamma$ dans la cellule référencée).

Remarque II.3.4. Nous avons déjà introduit la notation $\mathcal{R}$ pour désigner l'ensemble des fonctions récursives. Au vu du contexte, il n'y aura jamais d'ambiguïté entre la machine $\mathcal{R}$ que nous venons de définir et cet ensemble de fonctions.

Exemple II.3.5. La machine de Turing $\mathcal{R}a\mathcal{L}$ a pour effet de placer un caractère a à droite de la cellule mémoire référencée sans toucher à cette dernière.

Exemple II.3.6. Soit l'organigramme

$$\mathcal{R}^2\#\mathcal{L}^3 \xleftarrow{\#} \mathcal{R} \xrightarrow{\gamma\neq\#} \mathcal{R}^2\gamma\mathcal{L}^3 \quad .$$
$$\uparrow$$

Celui-ci a pour effet de recopier le contenu de la cellule à droite de la cellule référencée deux cases plus loin. Remarquez que nous avons utilisé ici la remarque II.3.1.

Voici à présent à la figure II.10, quatre organigrammes rudimentaires et pourtant fort utiles. Ainsi, $\mathcal{R}_\sigma$ ($\sigma \in \Sigma$) déplace le curseur de référence

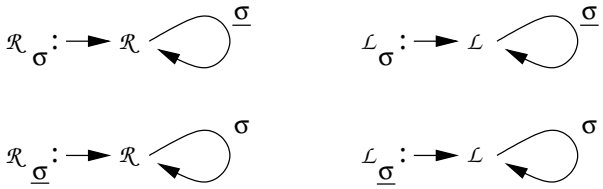


FIGURE II.10. Des machines élémentaires.

vers la droite jusqu'à rencontrer le caractère σ, i.e., on se positionne, si possible[8], sur la première cellule à droite de la cellule référencée qui contient le symbole σ. Par contre, $\mathcal{R}_{\underline{\sigma}}$ déplace le curseur de référence vers la droite jusqu'à rencontrer un caractère différent de σ. On comprend aisément le comportement des machines $\mathcal{L}_\sigma$ et $\mathcal{L}_{\underline{\sigma}}$. La restriction $\sigma \in \Sigma$ peut être levée en considérant $\sigma \in \Gamma$. Ainsi, on utilisera souvent des machines comme $\mathcal{R}_\#$ ou $\mathcal{L}_{\underline{\#}}$.

Les machines suivantes $\mathcal{S}_{L,k}$, $\mathcal{S}_{R,k}$, $\mathcal{C}_k$ et $\mathcal{E}_k$ sont définies pour tout entier $k \geq 1$ et permettent de réaliser des opérations de manipulation du ruban

[8]Si aucune cellule à droite de la cellule référencée ne contient a, alors la machine ne s'arrêtera jamais.

mémoire. Pour tous $m_1, \ldots, m_k \in \Sigma^*$, on a

$$\begin{array}{rcll} \mathcal{S}_{L,k} & : & q_0.\#m_1\#m_2\#\cdots\#m_k\underline{\#} & \vdash^* \quad f.m_1\#m_2\#\cdots\#m_k\underline{\#} \\ \mathcal{S}_{R,k} & : & q_0.\#m_1\#m_2\#\cdots\#m_k\underline{\#} & \vdash^* \quad f.\#\#m_1\#m_2\#\cdots\#m_k\underline{\#} \\ \mathcal{C}_k & : & q_0.\#m_1\#m_2\#\cdots\#m_k\underline{\#} & \vdash^* \quad f.\#m_1\#m_2\#\cdots\#m_k\#m_1\underline{\#} \\ \mathcal{E}_k & : & q_0.\#m_1\#m_2\#\cdots\#m_k\underline{\#} & \vdash^* \quad f.\#m_2\#\cdots\#m_k\underline{\#} \end{array}$$

En bon français, "décalage vers la gauche" ou "vers la droite".

On parle respectivement du "shift-left k", du "shift-right k", du "copieur k" et de "l'effaceur k". Le paramètre k permet de connaître le nombre d'arguments utilisés par ces fonctions. Il nous faut maintenant nous convaincre qu'il est possible de construire ces machines.

Remarque II.3.7. Une machine de Turing ne peut jamais accéder à une cellule plus à gauche que la première cellule du ruban. Tirant parti de cette constatation triviale, il est clair que pour tout mot $w \in \Gamma^*$, on a par exemple

$$\mathcal{S}_{L,k} : q_0.w\#m_1\#m_2\#\cdots\#m_k\underline{\#} \vdash^* f.w\, m_1\#m_2\#\cdots\#m_k\underline{\#}.$$

En effet, à aucun moment, la machine $\mathcal{S}_{L,k}$ n'ira altérer le contenu du ruban contenant le préfixe w. Sinon, cela signifierait que, partant d'une configuration $q_0.\#m_1\#m_2\#\cdots\#m_k\underline{\#}$, la machine aboutirait dans une configuration pendante, ce qui n'est pas le cas. Nous ferons très souvent usage de cette remarque que ce soit dans la construction de machines de Turing élémentaires ou dans des constructions plus élaborées.

Exemple II.3.8. Au vu de la remarque ci-dessus et si on suppose disposer de la machine $\mathcal{E}_k$ $(k = 1, 2, 3)$, alors on a

$$\begin{array}{rcll} \mathcal{E}_1 & : & q_0.\#m_1\#m_2\#m_3\underline{\#} & \vdash^* \quad f.\#m_1\#m_2\underline{\#} \\ \mathcal{E}_2 & : & q_0.\#m_1\#m_2\#m_3\underline{\#} & \vdash^* \quad f.\#m_1\#m_3\underline{\#} \\ \mathcal{E}_3 & : & q_0.\#m_1\#m_2\#m_3\underline{\#} & \vdash^* \quad f.\#m_2\#m_3\underline{\#} \end{array}.$$

Pour la construction de $\mathcal{S}_{L,k}$, on procède par récurrence sur k. Pour $k = 1$, il nous faut une machine réalisant

$$q_0.\#m_1\underline{\#} \vdash^* m_1\underline{\#}.$$

Il suffit de considérer l'organigramme donné à la figure II.11. (Le lecteur attentif observera que nous utilisons ici la remarque II.3.1.) Le lecteur se

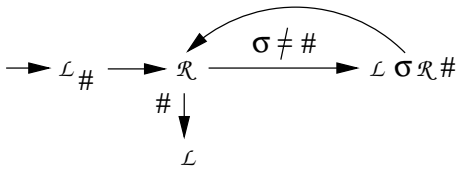


FIGURE II.11. Un organigramme pour $\mathcal{S}_{L,1}$.

convaincra aisément que l'organigramme proposé est le bon. Supposons à présent disposer de $\mathcal{S}_{L,k-1}$ et construisons $\mathcal{S}_{L,k}$. Une fois encore, on peut considérer l'organigramme de la figure II.12 (il suffit en fait de modifier légèrement l'organigramme précédent). Le fonctionnement de cet organigramme

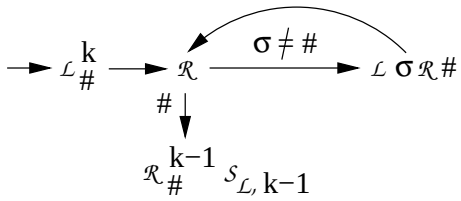


FIGURE II.12. Un organigramme pour $\mathcal{S}_{L,k}$.

est assez simple. Partant d'une configuration mémoire $\#m_1\#m_2\#\cdots\#m_k\underline{\#}$, l'application de $\mathcal{L}^k_\#$ donne la configuration $\underline{\#}m_1\#m_2\#\cdots\#m_k\#$ à laquelle on applique $\mathcal{R}$. Si $m_1 \in \Sigma^*$ est non vide[9], on entre alors dans la boucle de corps $\mathcal{L}\sigma\mathcal{R}\#$ et on obtient, après avoir épuisé les lettres de m_1, la configuration $m_1\underline{\#}\#m_2\#\cdots\#m_k\#$. A cette dernière, on applique à nouveau $\mathcal{R}$ pour obtenir la configuration $m_1\#\underline{\#}m_2\#\cdots\#m_k\#$. En appliquant à présent $\mathcal{R}^{k-1}_\#$, on obtient la configuration $m_1\#\#m_2\#\cdots\#m_k\underline{\#}$. Au vu de la remarque II.3.7, il ne reste plus qu'à appliquer $\mathcal{S}_{L,k-1}$ pour obtenir le résultat annoncé.

Les détails de construction de $\mathcal{S}_{R,k}$, $\mathcal{C}_k$ et $\mathcal{E}_k$ s'obtiennent de manière analogue et sont laissés au lecteur. Les figures II.13, II.14 et II.15 reprennent leur organigramme respectif.

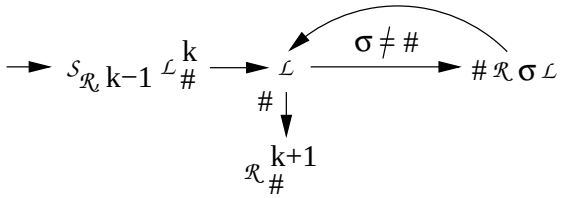


FIGURE II.13. Un organigramme pour $\mathcal{S}_{R,k}$.

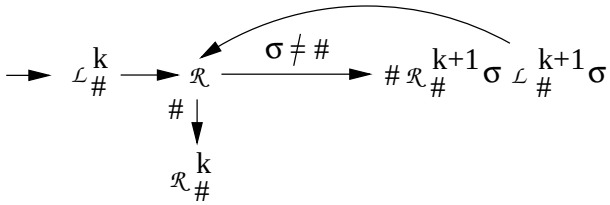


FIGURE II.14. Un organigramme pour $\mathcal{C}_k$.

Exemple II.3.9. Si nous désirons construire une machine permutant les deux champs de son ruban mémoire, i.e.,

$$q_0.\#m_1\#m_2\underline{\#} \vdash^* f.\#m_2\#m_1\underline{\#},$$

[9]Si m_1 est vide, on démarre avec une configuration $\#\#m_2\#\cdots\#m_k\underline{\#}$ et les justifications sont analogues.

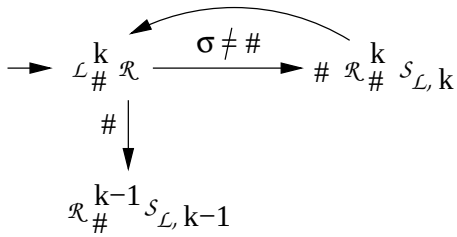


FIGURE II.15. Un organigramme pour $\mathcal{E}_k$.

il suffit de composer deux machines élémentaires comme suit,

$$\mathcal{C}_2\,\mathcal{E}_3.$$

Exemple II.3.10. La fonction $\Sigma_3 : \mathbb{N}^3 \to \mathbb{N} : (x, y, z) \mapsto x+y+z$ s'obtient très facilement. En effet, on désire construire une machine réalisant

$$q_0.\#u^x\#u^y\#u^z\underline{\#} \vdash^* f.\#u^{x+y+z}\underline{\#}.$$

Il suffit de considérer $\mathcal{S}_{L,1}\,\mathcal{S}_{L,1}$. En effet, une première application de $\mathcal{S}_{L,1}$ à la configuration mémoire $\#u^x\#u^y\#u^z\underline{\#}$ donne $\#u^x\#u^{y+z}\underline{\#}$ et la seconde application fournit le résultat attendu.

Exercice II.3.11. Utiliser les machines vues précédemment pour résoudre les exercices suivants.

- Soient $1 \leq i < j \leq k$. Construire une machine permutant les champs i et j lorsqu'on démarre avec un ruban mémoire ayant k champs, i.e.,

$$\begin{aligned} & q_0.\#m_1\#\cdots\#m_{i-1}\#m_i\#m_{i+1}\#\cdots\#m_{j-1}\#m_j\#m_{j+1}\#\cdots\#m_k\underline{\#} \\ \vdash^* \quad & f.\#m_1\#\cdots\#m_{i-1}\#m_j\#m_{i+1}\#\cdots\#m_{j-1}\#m_i\#m_{j+1}\#\cdots\#m_k\underline{\#}. \end{aligned}$$

- Construire une machine de Turing calculant la fonction produit $\pi_2 : \mathbb{N} \times \mathbb{N} \to \mathbb{N} : (m, n) \mapsto m.n$.

4. Fonctions calculables et récursives

Dans les sections précédentes, nous avons manipulé quelque peu les machines de Turing. Nous espérons qu'à présent, le lecteur sera satisfait de la définition de ces machines comme formalisation de la notion de procédure effective de calcul. De fait, supposons disposer d'un algorithme calculant les valeurs d'une fonction f. Dès lors, nous admettons qu'il est possible[10] de se ramener à une machine de Turing effectuant les mêmes opérations que l'algorithme et calculant f. (En effet, nous avons dans les paragraphes précédents réalisé au moyen de machines de Turing des enchaînements, des

[10]Rappelons que cette assertion ne peut être démontrée puisqu'on ne dispose pas d'une définition formelle de procédure effective. Il s'agit donc en quelque sorte d'un acte de foi traduit par la thèse de Church-Turing : *toute fonction pour laquelle on dispose d'une procédure effective de calcul est calculable par machine de Turing.* Cette thèse sera donnée en des termes plus précis à la section 5.

compositions conditionnelles, des boucles, des sommes, des produits, des permutations de variables, etc .... Par conséquent et même s'il s'agit d'une tâche fastidieuse, on doit être en mesure de traduire tout algorithme en une machine de Turing réalisant les mêmes opérations.)

Notons que la réciproque est quant à elle triviale. Si une fonction est calculable par une machine de Turing, alors elle l'est par une procédure effective.

En quelque sorte, on peut voir une machine de Turing comme un programme opérant au plus bas niveau. En effet, de manière la plus interne qu'il soit, un processeur n'effectue que des opérations de manipulation élémentaire de blocs mémoire. Les systèmes d'exploitation et les environnements de développement étant de nos jours évolués et conviviaux, le programmeur n'a plus à se soucier, dans un langage de haut niveau comme le `Pascal`, `C`, `Mathematica`, ..., de ces tâches rudimentaires et pénibles de manipulation de la mémoire. En effet, c'est le compilateur (ou l'interpréteur) qui s'acquittera de transformer les instructions du programmeur en instructions pour le processeur. La machine de Turing possède sur un processeur réel l'avantage théorique[11] d'être plus "simple" à manipuler car elle dispose d'un ruban mémoire infini (nous n'aurons donc pas de dépassement de registres, de saturation d'espace de stockage, etc...).

Rappelons que la question fondamentale énoncée au début de ce cours était de caractériser les fonctions calculables par une procédure effective. Il est donc naturel d'introduire l'ensemble suivant.

Définition II.4.1. On note $\mathcal{C}$ l'ensemble des fonctions de $\mathfrak{F}$ calculables par une machine de Turing. On dit simplement que $\mathcal{C}$ est l'*ensemble des fonctions calculables*.

Pour faire le lien avec le chapitre précédent, nous allons dans un premier temps montrer que toute fonction récursive est calculable, i.e., on a

$$\mathcal{PR} \subsetneq \mathcal{R} \subset \mathcal{C}.$$

En fait, on montrera qu'on a même $\mathcal{R} = \mathcal{C}$ (cf. corollaire II.4.14). L'inclusion proposée résulte de quelques lemmes faciles.

Lemme II.4.2. *Les fonctions initiales sont calculables.*

Démonstration. C'est immédiat. A titre d'exemple, si $1 \leq i \leq p$, alors la fonction de projection $\mathcal{P}_{i,p}$ est calculée par la machine

$$\mathcal{E}_p \, \mathcal{E}_{p-1} \cdots \mathcal{E}_{p-i+2} \, \mathcal{E}_1^{p-i}.$$

[11] La question posée depuis le départ était elle aussi théorique : "quelles fonctions sont calculables par un algorithme ?". Cette question ne précise en rien la facilité avec laquelle on peut calculer une fonction en termes de temps ou ressources nécessaires et est donc quelque peu éloignée de certains aspects pratiques importants (complexités temporelle et spatiale).

En effet, partant de la configuration mémoire $\#m_1\#m_2\#\cdots\#m_p\underline{\#}$, l'application de $\mathcal{E}_p$ donne $\#m_2\#\cdots\#m_p\underline{\#}$ puis l'application de $\mathcal{E}_{p-1}$ donne $\#m_3\#\cdots\#m_p\underline{\#}$, jusqu'à obtenir $\#m_i\#\cdots\#m_p\underline{\#}$. Etant en présence de $p-i+1$ champs, les $p-i$ applications successives de $\mathcal{E}_1$ suppriment dans l'ordre les champs $m_p, m_{p-1}, \ldots, m_{i+1}$. Ceci permet d'obtenir finalement $\#m_i\underline{\#}$. Remarquons que ce n'est pas la seule façon d'obtenir la fonction de projection $\mathcal{P}_{i,p}$. On aurait pu tout aussi bien considérer la machine

$$\mathcal{E}_1^{p-i}\,\mathcal{E}_2^{i-1}.$$

■

Lemme II.4.3. *L'ensemble $\mathcal{C}$ des fonctions calculables est stable par composition.*

Démonstration. Soient $f_1, \ldots, f_n$ des fonctions de $\mathfrak{F}_p$ calculables respectivement par des machines de Turing $\mathcal{F}_1, \ldots, \mathcal{F}_n$ et une fonction g de $\mathfrak{F}_n$ calculable par une machine de Turing $\mathcal{G}$. Il est clair que la machine

$$\mathcal{C}_p^p\,\mathcal{F}_1\,\mathcal{C}_{p+1}^p\,\mathcal{F}_2\,\cdots\mathcal{C}_{p+n-1}^p\,\mathcal{F}_n\,\mathcal{G}\,\mathcal{E}_2^p$$

calcule la composée $g(f_1, \ldots, f_n)$. Les vérifications sont immédiates. Pour simplifier les écritures, on s'autorise à écrire n au lieu de u^n, cela n'entraînant pas d'ambiguïté. On a dès lors la suite de configurations mémoire

$$\begin{array}{lll}
 & & \#n_1\#\cdots\#n_p\underline{\#} \\
\mathcal{C}_p^p & \vdash^* & \#n_1\#\cdots\#n_p\#n_1\#\cdots\#n_p\underline{\#} \\
\mathcal{F}_1 & \vdash^* & \#n_1\#\cdots\#n_p\#f_1(n_1,\ldots,n_p)\underline{\#} \\
\mathcal{C}_{p+1}^p & \vdash^* & \#n_1\#\cdots\#n_p\#f_1(n_1,\ldots,n_p)\#n_1\#\cdots\#n_p\underline{\#} \\
\vdots & & \vdots \\
\mathcal{F}_n & \vdash^* & \#n_1\#\cdots\#n_p\#f_1(n_1,\ldots,n_p)\#\cdots\#f_n(n_1,\ldots,n_p)\underline{\#} \\
\mathcal{G} & \vdash^* & \#n_1\#\cdots\#n_p\#g(f_1(n_1,\ldots,n_p),\ldots,f_n(n_1,\ldots,n_p))\underline{\#} \\
\mathcal{E}_2^p & \vdash^* & \#g(f_1(n_1,\ldots,n_p),\ldots,f_n(n_1,\ldots,n_p))\underline{\#}
\end{array}$$

■

Lemme II.4.4. *L'ensemble $\mathcal{C}$ des fonctions calculables est stable par récursion primitive.*

Démonstration. Soient $g \in \mathfrak{F}_p$ et $h \in \mathfrak{F}_{p+2}$ deux fonctions calculables respectivement par des machines $\mathcal{G}$ et $\mathcal{H}$. Nous allons montrer que la fonction $f \in \mathfrak{F}_{p+1}$ définie par $f(\overline{x}, 0) = g(\overline{x})$ et $f(\overline{x}, n+1) = h(\overline{x}, n, f(\overline{x}, n))$, pour tout $\overline{x} \in \mathbb{N}^p$, est également calculable. Tout d'abord, remarquons qu'un algorithme[12] naïf permet de calculer $f(x_1, \ldots, x_p, n)$:

$\mathtt{x} \leftarrow n$, $\mathtt{y} \leftarrow 0$, $\mathtt{z} \leftarrow g(x_1, \ldots, x_p)$
Tant que $\mathtt{x} > 0$, répéter
    $\mathtt{x} \leftarrow \mathtt{x} - 1$

[12]La démonstration de l'exactitude de cet algorithme est laissée au lecteur. On remarquera par exemple que $\mathtt{x} + \mathtt{y}$ est un invariant de boucle.

$\mathtt{z} \leftarrow h(x_1, \ldots, x_p, \mathtt{y}, \mathtt{z})$
$\mathtt{y} \leftarrow \mathtt{y} + 1$
Sortir $\mathtt{z}$.

Cet algorithme calcule dans l'ordre $f(\overline{x}, 0)$, $f(\overline{x}, 1)$, …, $f(\overline{x}, n)$. Il nous faut à présent simuler cet algorithme au moyen d'une machine de Turing. En utilisant une fois encore la notation n à la place de u^n, la configuration initiale du ruban de la machine doit être de la forme

$$\#x_1\#x_2\#\cdots\#x_p\#n\underline{\#}.$$

Notre stratégie va être, pour simuler l'algorithme, de considérer un ruban mémoire à $p+3$ champs

$$\#x_1\#x_2\#\cdots\#x_p\#\mathtt{x}\#\mathtt{y}\#\mathtt{z}\#$$

pour stocker le contenu des variables $\mathtt{x}$, $\mathtt{y}$ et $\mathtt{z}$ et également préserver les données initiales. En effet, ces dernières sont utilisées à chaque application de $\mathcal{H}$.

Ainsi, à partir de la configuration initiale donné ci-dessus, on va appliquer

$$\mathcal{R}\,\mathcal{C}^p_{p+2}\,\mathcal{G}$$

pour obtenir

$$\#x_1\#x_2\#\cdots\#x_p\#n\#\#g(x_1, \ldots, x_p)\underline{\#}$$

qui correspond bien à la phase d'initialisation de l'algorithme. Passons à présent à la boucle "tant que". Etant dans une configuration de la forme

$$\#x_1\#x_2\#\cdots\#x_p\#\mathtt{x}\#\mathtt{y}\#\mathtt{z}\underline{\#},$$

tester si $\mathtt{x} > 0$ peut être réalisé comme suit

$$\mathcal{L}^2_{\#}\mathcal{L}.$$

Si la cellule référencée contient u, alors $\mathtt{x} > 0$; sinon elle contient $\#$ et $\mathtt{x} = 0$.

Si $\mathtt{x} > 0$, on a donc une configuration de la forme

$$\#x_1\#x_2\#\cdots\#x_p\#u^{\mathtt{x}-1}\underline{u}\#\mathtt{y}\#\mathtt{z}\#.$$

L'application de $\#\,\mathcal{R}^3_{\#}\,\mathcal{S}_{L,3}$ donne

$$\#x_1\#x_2\#\cdots\#x_p\#\mathtt{x}-1\#\mathtt{y}\#\mathtt{z}\underline{\#}.$$

Ceci correspond à la première instruction de la boucle. Passons à la deuxième en appliquant à la configuration en cours

$$\mathcal{C}^p_{p+3}\,\mathcal{C}^2_{p+2}\,\mathcal{H}\,\mathcal{E}_2$$

pour obtenir

$$\#x_1\#x_2\#\cdots\#x_p\#\mathtt{x}-1\#\mathtt{y}\#h(x_1, \ldots, x_p, \mathtt{y}, \mathtt{z})\underline{\#}.$$

Il reste à appliquer la dernière instruction de la boucle avec

$$\mathcal{S}_{R,1}\,\mathcal{L}^2_{\#}\,u\,\mathcal{R}^2_{\#}.$$

Si $\mathtt{x} = 0$, on a, au sortir du test, une configuration de la forme

$$\#x_1\#x_2\#\cdots\#x_p\underline{\#}\#\mathtt{y}\#\mathtt{z}\#.$$

Il suffit alors d'appliquer $\mathcal{R}^3_{\#}\,\mathcal{E}^{p+2}_2$. Il est aisé d'obtenir l'organigramme complet simulant l'algorithme et calculant la fonction f.

$$\rightarrow \mathcal{R}\,\mathcal{C}^p_{p+2}\,\mathcal{G} \rightarrow \quad \begin{array}{c} \mathcal{L}^2_{\#}\mathcal{L} \\ \downarrow \# \\ \mathcal{R}^3_{\#}\,\mathcal{E}^{p+2}_2 \end{array} \quad \begin{array}{c} \longleftarrow \\ \overset{\neq\#}{\longrightarrow} \end{array} \quad \#\,\mathcal{R}^3_{\#}\,\mathcal{S}_{L,3}\,\mathcal{C}^p_{p+3}\,\mathcal{C}^2_{p+2}\,\mathcal{H}\,\mathcal{E}_2\,\mathcal{S}_{R,1}\,\mathcal{L}^2_{\#}\,u\,\mathcal{R}^2_{\#}$$

■

Lemme II.4.5. *L'ensemble $\mathcal{C}$ des fonctions calculables est stable par minimisation (non bornée).*

Démonstration. Soit une fonction $f(\overline{x}) = \mu_t((\overline{x}, t) \in A)$ où A est un sous-ensemble de $\mathbb{N}^{p+1}$ *calculable*, i.e., tel que sa fonction caractéristique $\chi_A : \mathbb{N}^{p+1} \rightarrow \{0, 1\}$ soit calculable par une machine de Turing $\mathcal{A}$. On procède comme dans le lemme précédent en exhibant un algorithme de calcul que l'on transformera ensuite en machine de Turing. Pour calculer $f(x_1, \ldots, x_p)$, on effectue

$\mathtt{x}\leftarrow 0$, $\mathtt{y}\leftarrow \chi_A(x_1, \ldots, x_p, 0)$
Tant que $\mathtt{y} = 0$, répéter
    $\mathtt{x} \leftarrow \mathtt{x} + 1$
    $\mathtt{y} \leftarrow \chi_A(x_1, \ldots, x_p, \mathtt{x})$
Sortir $\mathtt{x}$.

En utilisant une fois de plus la convention de noter indifféremment n ou u^n, la configuration initiale du ruban est

$$\#x_1\#x_2\#\cdots\#x_p\underline{\#}.$$

Notre stratégie est de considérer un ruban mémoire comprenant $p+2$ champs, les deux champs supplémentaires servant à coder les variables $\mathtt{x}$ et $\mathtt{y}$. Partant de la configuration initiale, la phase d'initialisation de l'algorithme s'obtient par

$$\mathcal{R}\,\mathcal{C}^p_{p+1}\,\mathcal{R}\,\mathcal{A}$$

pour avoir un ruban mémoire contenant

$$\#x_1\#x_2\#\cdots\#x_p\#\#\chi_A(x_1, \ldots, x_p, 0)\underline{\#}.$$

Pour tester si $\mathtt{y} = 0$ ou 1, il suffit d'appliquer la machine $\mathcal{L}$ et de regarder si la cellule référencée contient respectivement # ou u.

Si $\mathtt{y} = 0$, on est dans une configuration

$$\#x_1\#x_2\#\cdots\#x_p\#\mathtt{x}\underline{\#}\#.$$

On effectue alors $u\,\mathcal{R}\,\mathcal{C}_{p+1}^{p+1}\,\mathcal{A}$ qui a pour effet d'effectuer le corps de la boucle "tant que".

Si $\mathtt{y} = 1$, on est dans une configuration

$$\#x_1\#x_2\#\cdots\#x_p\#\mathtt{x}\#\underline{u}\#.$$

Il suffit alors pour conclure d'appliquer $\mathcal{R}\,\mathcal{E}_1\,\mathcal{E}_2^p$. L'organigramme complet se déduit aisément.

■

En conclusion des quatre lemmes ci-dessus, le résultat suivant est immédiat.

Théorème II.4.6. *Toute fonction récursive est calculable (par une machine de Turing).*

Nous allons maintenant prouver la réciproque de ce résultat et ainsi obtenir que l'ensemble des fonctions calculables par machine de Turing et celui des fonctions récursives coïncident. De cette façon, on obtient un argument supplémentaire en faveur de la thèse de Church-Turing. En effet, deux formalisations *a priori* totalement différentes du même concept de procédure effective fournissent le même résultat. Ainsi, on peut accepter encore un peu plus facilement le fait que les machines de Turing capturent exactement la notion d'algorithme. La réciproque du théorème II.4.6 s'énonce comme suit.

Théorème II.4.7. *Pour toute fonction $f \in \mathfrak{F}_p$ calculable (par machine de Turing), il existe des fonctions primitives récursives P et Q telles que*

$$f(\overline{x}) = P(\overline{x}, \mu_t(Q(\overline{x}, t) = 0)),\ \forall \overline{x} \in \mathbb{N}^p.$$

En particulier, toute fonction calculable (par machine de Turing) est récursive.

Définition II.4.8. Nous utiliserons dans la preuve l'opération *miroir* définie récursivement par[13] $\varepsilon^R = \varepsilon$ et $(w\gamma)^R = \gamma w^R$, pour tous $w \in \Gamma^*$, $\gamma \in \Gamma$. Ainsi, le miroir du mot $\gamma_1 \cdots \gamma_k$ est simplement le mot $(\gamma_1 \cdots \gamma_k)^R = \gamma_k \cdots \gamma_1$.

Démonstration. L'idée principale de la preuve est de coder une machine de Turing $\mathcal{M} = (Q, q_0, F, \Sigma, \Gamma, \delta)$ calculant f au moyen de fonctions de $\mathfrak{F}$ primitives récursives. On pourra en particulier coder la fonction de transition de la machine ainsi que le passage d'une configuration machine à une autre. Puisque $\mathcal{M}$ calcule f, pour tout $(x_1, \ldots, x_p) \in \mathbb{N}^p$, il existe $h \in F$ tel que

$$q_0.\#u^{x_1}\#\cdots u^{x_p}\underline{\#} \vdash^* h.\#u^{f(x_1,\ldots,x_p)}\underline{\#}.$$

[13] La notation $\cdot^R$ rappelle la nomenclature anglo-saxonne "*reversal*". Un mot égal à son miroir est un *palindrome*.

Sans aucune perte de généralité[14], on peut supposer que $F = \{h\}$.

i) *Codage d'une machine de Turing.* L'alphabet de ruban de la machine $\mathcal{M}$ est de la forme $\Gamma = \{\gamma_1 = \#, \gamma_2 = u, \gamma_3, \ldots, \gamma_t\}$ avec $t \geq 2$. Si on identifie les lettres $\gamma_1, \gamma_2, \ldots, \gamma_t$ de Γ aux chiffres $1, 2, \ldots, t$ de l'écriture[15] en base $t+1$, tout mot w de Γ^* correspond à un unique nombre entier $c(w)$ défini[16] par

Puisque $\mathcal{M}$ calcule une fonction de $\mathfrak{F}$, il est normal que $u \in \Gamma$.

"c" comme "codage".

$$c(\varepsilon) = 0 \text{ et } c(\gamma_{i_n} \cdots \gamma_{i_0}) = \sum_{j=0}^{n} i_j \, (t+1)^j. \tag{2}$$

En particulier, $c(\gamma_i) = i$. Il est clair que la fonction $c : \Gamma^* \to \mathbb{N}$ est injective et permet donc de **coder les mots** de Γ^* par des entiers.

Exemple II.4.9. Si $\Gamma = \{\#, u\}$, $t = 2$ et le mot $\#u\#u$ est codé par

$$c(\#u\#u) = 1.3^3 + 2.3^2 + 1.3^1 + 2.3^0 = 50.$$

Par contre, l'entier 20 ne représente aucun mot. En effet, la représentation en base 3 de $20 = 2.3^2 + 0.3 + 2$ est le mot "202" qui contient un zéro. En d'autres termes, c n'est pas une fonction surjective sur $\mathbb{N}$.

Nous pouvons à présent **coder les configurations mémoire** de la machine $\mathcal{M}$. Pour ce faire, nous codons uniquement la partie significative d'une configuration

$$x\underline{\tau}y, \;\; x, y \in \Gamma^*, \tau \in \Gamma$$

où, rappelons que y peut être vide, si les cellules à droite de la cellule référencée contiennent toutes le caractère blanc #. Ce codage est simplement réalisé au moyen d'un triplet

On code y^R et non pas y, pour des raisons pratiques qui s'expliqueront rapidement.

$$(c(x), c(\tau), c(y^R)) \in \mathbb{N}^3.$$

Remarque II.4.10. Tout comme chaque entier n'est pas nécessairement le code d'un mot de Γ^*, tout triplet de $\mathbb{N}^3$ n'est pas nécessairement le code d'une configuration mémoire.

[14] Il suffit de rassembler tous les états accepteurs en un seul et même état. Cela ne pose pas de problème car, par définition, la machine de Turing s'arrête une fois un tel état atteint et donc aucune transition n'est *a priori* définie à partir d'un état accepteur. D'ailleurs une définition alternative des machines de Turing consiste à prendre des machines ayant un unique état privilégié appelé *état d'arrêt*.

[15] Pour écrire les nombres en base $t+1$, on utilise les chiffres de 0 à t. Si on n'utilise pas le chiffre 0, on ne représente pas tous les entiers mais un sous-ensemble strict de $\mathbb{N}$. Par exemple en base 2, si on utilise uniquement le chiffre 1, on ne représente que les nombres de la forme $2^n - 1$.

[16] Le nombre $c(w)$ est appelé le *nombre de Gödel* du mot w (Kurt Gödel, 1906–1978, logicien autrichien). On parle parfois de "*gödélisation*". Remarquons qu'il est indispensable de ne pas utiliser le chiffre "0" car il conduit à un codage non univoque. En effet, imaginez que $\Gamma = \{\gamma_0 = \#, \gamma_1 = u, \gamma_2, \ldots, \gamma_t\}$. Dans ce cas, on aurait $c(\#u) = 0.(t+1) + 1 = 1$ et $c(\#\#u) = 0.(t+1)^2 + 0.t + 1 = 1$!

On convient en plus de poser

$$\gamma_{t+1} = L,\ c(L) = t+1,\ \gamma_{t+2} = R \text{ et } c(R) = t+2.$$

Pour pouvoir calculer $f(x_1, \ldots, x_p)$, il faut **coder les données** fournies à la machine $\mathcal{M}$. Ce codage est réalisé par la fonction $c_d : \mathbb{N}^p \to \mathbb{N}$ définie par

"c_d" comme "codage des données".
On ne considère que le premier tronçon de la partie significative.

$$c_d : \mathbb{N}^p \to \mathbb{N} : (x_1, \ldots, x_p) \mapsto c(\#u^{x_1}\#\cdots\#u^{x_p}).$$

Lemme II.4.11. *La fonction $c_d \in \mathfrak{F}_p$ est primitive récursive.*

Démonstration. On procède par récurrence sur p. Si $p = 1$, alors

$$\begin{aligned} c_d(x) &= c(\#u^x) = 1.(t+1)^x + 2.(t+1)^{x-1} + \cdots + 2.(t+1)^0 \\ &= (t+1)^x + 2\ \textstyle\sum_{n=0}^{x-1}(t+1)^n. \end{aligned}$$

Ceci se réexprime par

$$c_d(x) = (t+1)^x + 2\,(((t+1)^x \dot{-} 1)\,\mathtt{DIV}\,t).$$

On en conclut que c_d est de classe $\mathcal{PR}$.

Supposons le résultat acquis pour p et démontrons-le pour $p+1$. Il vient[17]

$$\begin{aligned} c_d(x_1, \ldots, x_p, x) &= c(\#u^{x_1}\#\cdots\#u^{x_p}\#u^x) \\ &= c_d(x_1, \ldots, x_p)\,(t+1)^{x+1} + c(\#u^x) \end{aligned}$$

et cette fonction est $\mathcal{PR}$. Il suffit d'appliquer l'hypothèse de récurrence et de remarquer qu'on ne fait qu'effectuer des sommes, des produits et des puissances de fonctions $\mathcal{PR}$.

■

Si nous parvenons à réaliser le codage des transitions de $\mathcal{M}$, il nous sera nécessaire de **décoder les résultats** obtenus. Ceci est réalisé par une fonction d_r. Une fois encore, nous nous intéressons uniquement à la partie significative d'une configuration mémoire.

"d_r" comme "décodage des résultats".

Lemme II.4.12. *On peut définir une fonction $d_r : \mathbb{N} \to \mathbb{N}$ de classe $\mathcal{PR}$ telle que*

$$d_r(c(\#u^n)) = n.$$

Démonstration. Il est clair que le nombre $c(\#u^n)$ écrit en base $t+1$ possède exactement $n+1$ chiffres[18]. Si x est un entier, il est bien connu que le nombre de chiffres dans la représentation de x en base $b \geq 2$ est donné par

$$1 + \lfloor \log_b x \rfloor = 1 + \mu_{s \leq x}(x < b^{s+1}).$$

[17] Pour effectuer un décalage vers la gauche de $x+1$ chiffres dans l'écriture en base $t+1$ d'un entier, il suffit de multiplier cet entier par $(t+1)^{x+1}$ pour placer $x+1$ zéros à la droite de la représentation.

[18] En base $t+1$, $c(\#u^n)$ s'écrit $1\underbrace{2\cdots2}_{n\times}$.

Par application de la minimisation bornée à un prédicat $\mathcal{PR}$, il est immédiat que cette dernière fonction est $\mathcal{PR}$ et par conséquent,

$$d_r(x) = \lfloor \log_{t+1} x \rfloor$$

est $\mathcal{PR}$.

■

Nous pouvons à présent **coder les configurations machine**. Ces dernières se composent d'un état et d'une configuration mémoire. Rappelons que nous avons ici $F = \{h\}$. Enumérons les états de Q en commençant par l'état accepteur,

$$Q = \{h = \theta_0, q_0 = \theta_1, \ldots, q_e = \theta_{e+1}\},$$

En particulier, une machine s'arrête lorsqu'elle atteint un état accepteur, i.e., si le code de l'état vaut 0.

$e \geq 0$. Le code d'un état est simplement donné par son indice,

$$c(\theta_i) = i.$$

Ainsi, un configuration machine $q.x\underline{\tau}y$ est codée par un 4-uple

$$(c(q), c(x), c(\tau), c(y^R)) \in \mathbb{N}^4.$$

Passons au **codage de la fonction de transition**

$$\delta : Q \times \Gamma \to Q \times (\Gamma \cup \{L, R\}).$$

On définit deux fonctions $D_1, D_2 \in \mathfrak{F}_2$ telles que si $\delta(p, \gamma) = (q, x)$, $p, q \in Q$, $\gamma \in \Gamma$, $x \in \Gamma \cup \{L, R\}$ alors

$$D_1(c(p), c(\gamma)) = c(q) \text{ et } D_2(c(p), c(\gamma)) = c(x).$$

Autrement dit, si $1 \leq i \leq e+1$ et $1 \leq j \leq t$

$$\delta(q_i, \gamma_j) = (\theta_{D_1(i,j)}, \gamma_{D_2(i,j)}).$$

D'où l'intérêt d'avoir posé $\gamma_{t+1} = L$ et $\gamma_{t+2} = R$.

Remarque II.4.13. Les fonctions D_1 et D_2 peuvent être prises $\mathcal{PR}$. En effet, il suffit de considérer n'importe quelle fonction $\mathcal{PR}$ de $\mathfrak{F}_2$, par exemple la fonction constante $\mathbf{0}_2$, et de modifier sa valeur en un nombre fini de couples (i, j) correspondant au codage des couples de $Q \times \Gamma$. Le nombre d'états et l'alphabet de $\mathcal{M}$ étant finis, ce nombre de couples est fini. On conclut en observant que modifier une fonction en un nombre fini de points n'altère pas son caractère $\mathcal{PR}$. (Il s'agit d'une application du schéma généralisé de définition par cas, cf. proposition I.3.2.)

ii) *Codage des transitions.* Nous devons expliciter comment réaliser, au moyen d'une fonction

$$F : \mathbb{N}^4 \to \mathbb{N}^4 : \overline{n} = (n_1, n_2, n_3, n_4) \mapsto (F_1(\overline{n}), F_2(\overline{n}), F_3(\overline{n}), F_4(\overline{n})),$$

le passage du code d'une configuration machine au code de la configuration machine résultant d'une transition dans $\mathcal{M}$. Soit (n_1, n_2, n_3, n_4), le codage d'une configuration machine. Il est évident, au vu des conventions prises ci-dessus, que le code de l'état atteint après transition est $D_1(n_1, n_3)$ et ainsi,

$F_1 = D_1(\mathcal{P}_{1,4}, \mathcal{P}_{3,4})$ est de classe $\mathcal{PR}$. Nous savons que $D_2(n_1, n_3)$ est le code du caractère de $\Gamma \cup \{L, R\}$ fourni par la fonction de transition. Trois situations sont à envisager. (On pourra une fois encore utiliser le schéma généralisé de définition par cas.)

a) $D_2(n_1, n_3) = t + 2$, i.e., le caractère obtenu est R. Dans ce cas, il est facile de se convaincre que les fonctions suivantes réalisent la transition

$$\begin{array}{rcl} F_2(n_1, n_2, n_3, n_4) & = & n_2\,(t+1) + n_3 \\ F_3(n_1, n_2, n_3, n_4) & = & \left\{ \begin{array}{ll} n_4 \,\mathtt{MOD}\, (t+1) & , \text{ si } n_4 > 0 \\ 1 = c(\#) & , \text{ si } n_4 = 0 \end{array} \right. \\ F_4(n_1, n_2, n_3, n_4) & = & n_4 \,\mathtt{DIV}\, (t+1). \end{array}$$

On vérifie aisément que toutes ces fonctions sont $\mathcal{PR}$.

On se rappelera que le codage de la dernière composante fait intervenir l'opération miroir.

b) $D_2(n_1, n_3) = t + 1$, i.e., le caractère obtenu est L. Dans ce cas, il vient

$$\begin{array}{rcl} F_2(n_1, n_2, n_3, n_4) & = & n_2 \,\mathtt{DIV}\, (t+1) \\ F_3(n_1, n_2, n_3, n_4) & = & n_2 \,\mathtt{MOD}\, (t+1) \\ F_4(n_1, n_2, n_3, n_4) & = & \left\{ \begin{array}{ll} 0 & , \text{ si } n_3 = 1 = c(\#) \text{ et } n_4 = 0 \\ (t+1)\, n_4 + n_3 & , \text{ sinon.} \end{array} \right. \end{array}$$

Ces fonctions sont une fois encore $\mathcal{PR}$.

c) $1 \leq D_2(n_1, n_3) \leq t$, i.e., le caractère obtenu appartient à Γ. Dans ce cas, on a

$$\begin{array}{rcl} F_2(n_1, n_2, n_3, n_4) & = & n_2 \\ F_3(n_1, n_2, n_3, n_4) & = & D_2(n_1, n_3) \\ F_4(n_1, n_2, n_3, n_4) & = & n_4. \end{array}$$

Ces fonctions sont une fois encore $\mathcal{PR}$.

Au vu de ce qui précède, on en conclut que F est de classe $\mathcal{PR}$ et que[19]

$$F \circ c = c \circ \delta.$$

Soit C une configuration machine. Nous voudrions à présent exprimer le fait qu'une configuration C' est obtenue à partir de C, après $s \geq 0$ applications de la fonction de transition (ce que l'on notera, par abus d'écriture, $\delta^s(C) = C'$). Pour ce faire, nous allons considérer des quintuples d'entiers plutôt que des 4-uples comme nous l'avions fait jusqu'à présent et des fonctions

$$F_i^* : \mathbb{N}^5 \to \mathbb{N},\ i = 1, 2, 3, 4,$$

telles que

$$(F_1^*, F_2^*, F_3^*, F_4^*)(c(C), s) = c(\delta^s(C)).$$

On peut définir ces fonctions comme suit : si $\overline{n} = (n_1, n_2, n_3, n_4)$, alors pour $i = 1, 2, 3, 4$,

$$\left\{ \begin{array}{rcl} F_i^*(\overline{n}, 0) & = & n_i \\ F_i^*(\overline{n}, s+1) & = & F_i(F_1^*(\overline{n}, s), F_2^*(\overline{n}, s), F_3^*(\overline{n}, s), F_4^*(\overline{n}, s)). \end{array} \right.$$

[19] Rigoureusement, δ ne s'applique pas à une configuration machine, mais par abus d'écriture, nous nous autorisons une telle extension.

En appliquant la proposition I.3.2 (schéma de récursion primitive généralisé), on en conclut que les F_i^* sont $\mathcal{PR}$.

iii) *Expression de f à l'aide du codage de $\mathcal{M}$.* Tous les ingrédients sont à présent en notre possession. La configuration initiale de la machine de Turing $\mathcal{M}$ pour le calcul de f sur les données $(x_1, \ldots, x_p)$ est

$$q_0.\#u^{x_1}\# \cdots \#u^{x_p}\underline{\#}$$

qui est codée, avec nos conventions, par

$$(1 = c(q_0), c_d(x_1, \ldots, x_p), 1 = c(\#), 0 = c(\varepsilon)).$$

Pour mener à bien le calcul de f sur ces données, la machine $\mathcal{M}$ effectue un certain nombre s de transitions pour aboutir finalement dans une configuration de la forme

$$h.\#u^n\underline{\#}$$

qui doit être codée par

$$(0 = c(h), c(\#u^n), 1 = c(\#), 0 = c(\varepsilon)).$$

En d'autres termes, il existe $s \geq 0$ tel que

$$f(x_1, \ldots, x_p) = d_r(F_2^*(1, c_d(x_1, \ldots, x_p), 1, 0, s)).$$

Mais, nous devons considérer **la plus petite valeur** de t permettant d'aboutir dans l'état accepteur. En effet, par définition, la machine de Turing s'arrête dès qu'un tel état est atteint. Par contre, les fonctions que nous venons de définir pourraient être appliquées un nombre arbitraire de fois et on ne maîtriserait dès lors plus le résultat obtenu si on en continuait l'application après avoir atteint l'état accepteur. Par conséquent,

$$f(x_1, \ldots, x_p) =$$
$$d_r[F_2^*(1, c_d(x_1, \ldots, x_p), 1, 0, \mu_t\{F_1^*(1, c_d(x_1, \ldots, x_p), 1, 0, s) = 0\})].$$

On obtient bien le résultat annoncé.

■

Des théorèmes II.4.6 et II.4.7, on en déduit immédiatement le résultat suivant.

Corollaire II.4.14. *On a $\mathcal{R} = \mathcal{C}$.*

Nous allons à présent tirer parti de la notion de codage introduite dans la preuve du théorème II.4.7, pour caractériser les fonctions calculables appartenant non pas à $\mathfrak{F}$ mais définies sur des alphabets quelconques.

Soient Σ et Δ deux alphabets finis et une fonction

$$f : \Sigma^* \times \cdots \times \Sigma^* \to \Delta^* : (w_1, \ldots, w_p) \mapsto f(w_1, \ldots, w_p).$$

Tout comme en (2), si $\#\Sigma = t$ (resp. $\#\Delta = s$), alors les mots de Σ^* (resp. Δ^*) peuvent être codés par des entiers; il suffit de considérer les représentations correspondantes en base $t + 1$ (resp. $s + 1$). Dans les deux

cas, on note c la fonction de codage qui à un mot associe un entier. Ceci n'entraînera pas d'ambiguïté[20].

A la fonction f, on associe la fonction $c_f \in \mathfrak{F}_p$ définie par

$$c_f(x_1,\ldots,x_p) = \begin{cases} 0 & \text{, si } \exists i \leq p : x_i \notin Im(c) \\ c(f(c^{-1}(x_1),\ldots,c^{-1}(x_p))) & \text{, sinon.} \end{cases}$$

$$\begin{array}{ccc} (\Sigma^*)^p & \xrightarrow{f} & \Delta^* \\ c\downarrow & & \downarrow c \\ \mathbb{N}^p & \xrightarrow{c_f} & \mathbb{N} \end{array}$$

Proposition II.4.15. *La fonction $f : \Sigma^* \times \cdots \times \Sigma^* \to \Delta^*$ est calculable (par une machine de Turing) si et seulement si la fonction $c_f \in \mathfrak{F}_p$ est calculable.*

Démonstration. Pour simplifier les notations, nous supposerons $f : \Sigma^* \to \Delta^*$. Soit θ un nouveau symbole n'appartenant pas à Σ. Définissons tout d'abord une fonction

$$d : n \in \mathbb{N} \mapsto \begin{cases} c^{-1}(n) & \text{, si } n \in Im(c) \\ \theta & \text{, sinon.} \end{cases}$$

Une fois encore, nous avons une fonction d pour chacun des deux alphabets, le choix de la fonction envisagée étant clairement fixé par le contexte. Il est facile de se convaincre que les fonctions c et d sont calculables[21]. On note $\mathcal{M}_c$ et $\mathcal{M}_d$, des machines de Turing calculant ces fonctions.

Montrons que la condition est nécessaire. Supposons f calculable par une machine $\mathcal{M}_f$ et montrons que c_f l'est. Partant d'une configuration mémoire $\#u^x\underline{\#}$, on applique $\mathcal{M}_d$ pour obtenir $\#d(x)\underline{\#}$. De cette configuration, on teste si $d(x) = \theta$. Si tel est le cas, il faut sortir $\#\underline{\#}$. Sinon, on peut appliquer

[20] On peut par exemple supposer les deux alphabets Σ et Δ disjoints. Dès lors, suivant les symboles rencontrés, la base $t+1$ ou $s+1$ choisie va de soi.

[21] Soient $u, v \in \Sigma^*$. Il est clair que

$$c(uv) = c(u)\,(t+1)^{|v|} + c(v).$$

Ainsi, pour calculer $c(w)$, il suffit de lire une à une les lettres de w de droite à gauche. A chaque fois, on multiplie la valeur correspondant à la lettre lue par une puissance de $t+1$ (à chaque lettre lue, l'exposant est incrémenté d'une unité) et on somme les résultats obtenus. On peut facilement construire une machine de Turing réalisant ces opérations en considérant par exemple trois champs contenant respectivement, la partie du mot restant à lire, la puissance de $t+1$ considérée et le résultat partiel déjà calculé. (Une lecture de gauche à droite est également envisageable et les opérations sont encore plus simples à réaliser.)

Pour la fonction d, si on se donne un entier n, alors le reste de la division de n par $t+1$ donne la valeur de la lettre la plus à droite de $d(n)$. On considère alors le quotient de cette division et on recommence la même procédure. En particulier, n n'est pas le code d'un mot, si, à une quelconque étape, on obtient un reste nul.

Enfin, pour passer de σ_i à u^i et réciproquement, on peut mémoriser linéairement une table de conversion dans le ruban d'une machine de Turing

$$u\#\sigma_1\#uu\#\sigma_2\#\cdots\#u^t\#\sigma_t$$

et effectuer les tests nécessaires de comparaison.

la machine $\mathcal{M}_f$ qui donne $\#f(c^{-1}(x))\underline{\#}$. Il ne reste plus qu'à appliquer $\mathcal{M}_c$ pour obtenir

$$\#c(f(c^{-1}(x)))\underline{\#}.$$

L'organigramme correspondant est donné à la figure II.16

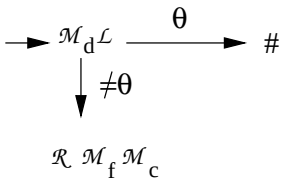


FIGURE II.16. La fonction c_f est calculable.

La condition est suffisante. Supposons c_f calculable par une machine $\mathcal{M}$. Soit w un mot de Σ^*. Partant de la configuration $\#w\underline{\#}$, on applique la machine $\mathcal{M}_c$ pour obtenir $\#c(w)\underline{\#}$. Il suffit alors d'appliquer $\mathcal{M}$ puis $\mathcal{M}_d$ pour obtenir $\#f(w)\underline{\#}$.

■

5. Langages acceptable et décidable

Comme nous l'avons vu à la remarque II.2.6, une machine de Turing ne s'arrête pas toujours à partir d'une configuration machine initiale donnée (on peut avoir une infinité de transitions et la machine ne s'arrête jamais ou encore obtenir une configuration pendante). Pour cette raison, nous devons faire la distinction entre le langage accepté par une machine de Turing et le langage décidé par cette même machine.

Définition II.5.1. Soit $\mathcal{M} = (Q, q_0, F, \Sigma, \Gamma, \delta)$. Un mot $w \in \Sigma^*$ est *accepté* par $\mathcal{M}$ s'il existe $f \in F$ et $x \in \Gamma^*$ tel que

Pour accepter un mot, l'important est que la machine s'arrête.

$$q_0.\#w\underline{\#} \vdash^* f.x.$$

En d'autres termes, w est accepté par $\mathcal{M}$ si, partant de la configuration machine $q_0.\#w\underline{\#}$, la machine $\mathcal{M}$ finit par s'arrêter en atteignant un état accepteur[22]. De manière naturelle, le *langage accepté* par $\mathcal{M}$ est l'ensemble des mots acceptés par $\mathcal{M}$. On le note $L(\mathcal{M})$. Un langage est *acceptable* s'il existe une machine de Turing qui l'accepte.

Définition II.5.2. Un langage $L \subset \Sigma^*$ est *décidable* si sa fonction caractéristique χ_L est calculable (par machine de Turing).

Remarque II.5.3. Si un langage $L \subset \Sigma^*$ est décidable, la machine de Turing $\mathcal{M}$ qui calcule χ_L accepte tout mot de Σ^*. En effet, pour tout $w \in \Sigma^*$, $q_0.\#w\#$ permettra toujours d'atteindre $f.\#\underline{\#}$ ou $f.\#u\underline{\#}$, pour un état accepteur f. De là, on en tire que *tout langage décidable est acceptable.*

Nous verrons que la réciproque est fausse.

[22] Si on anticipe l'utilisation des machines universelles, on peut même dans cette définition imposer l'arrêt en $f.\underline{\#}$ (cf. plus loin, la remarque II.10.7 et le théorème de Rice).

En effet, il suffit de modifier quelque peu $\mathcal{M}$ pour que cette machine ne s'arrête exclusivement que sur les mots de L. Ce n'est pas difficile[23]. Plutôt que de sortir $f.\#\underline{\#}$, on s'arrange dans ce cas pour obtenir une infinité de transitions (par exemple, en imposant à la machine d'aller toujours vers la droite quel que soit le caractère référencé).

Remarque II.5.4. On parle parfois de langage *récursivement énumérable* (resp. *récursif*) pour les langages acceptables (resp. décidables). La justification de ces dénominations viendra plus tard.

Nous sommes maintenant en mesure d'énoncer précisément la thèse de Church-Turing déjà si souvent évoquée : *les langages reconnus par une procédure effective sont ceux décidés par une machine de Turing.* Remarquez que nous restreignons notre propos (puisque nous avions plus tôt évoqués le cas des fonctions).

6. Extension des machines de Turing

En préambule à ce chapitre, nous avions annoncé que les généralisations éventuelles des machines de Turing n'apportent rien au point de vue de la puissance de calcul : on calcule exactement les mêmes fonctions. Dans cette section, nous présentons quelques-unes de ces généralisations et montrons qu'elles sont toutes équivalentes aux machines de Turing présentées jusqu'ici. Ce résultat donne une fois encore un argument supplémentaire en faveur de la thèse de Church-Turing : d'autres formalisations du concept d'algorithme n'apportent rien de neuf.

Notons que, dans cette section, on présente en particulier les machines de Turing non déterministes. Le non déterminisme est un concept très important que l'on retrouve souvent en informatique théorique et qui permet le cas échéant certaines simplifications[24]. En particulier, le non déterminisme interviendra dans l'étude de la complexité.

Définition II.6.1. Un *mot bi-infini* sur Σ est simplement une suite $(x_n)_{n\in\mathbb{Z}}$ indexée par les entiers relatifs. L'ensemble des mots bi-infinis est noté $\Sigma^{\mathbb{Z}}$.

On peut aisément définir une machine de Turing dont le ruban mémoire est un mot bi-infini. La seule différence avec une machine de Turing "classique" est qu'il est toujours possible d'effectuer un déplacement vers la gauche sans jamais obtenir de configuration pendante.

Proposition II.6.2. *Tout langage accepté ou décidé par une machine de Turing possédant un ruban bi-infini l'est aussi par une machine dont le ruban mémoire est un mot infini (à droite).*

[23] A ce stade, le lecteur n'en sera peut-être pas tout à fait convaincu. Le meilleur conseil est alors d'attendre la notion de machine de Turing universelle qui permet de simuler n'importe quelle machine de Turing.

[24] Pensez par exemple à la construction très simple d'un automate non déterministe correspondant à une expression régulière donnée, cf. le cours de théorie des automates.

La démonstration qui suit n'est pas difficile mais quelque peu fastidieuse. Lorsque nous aurons à notre disposition les machines de Turing universelles, on pourra obtenir une preuve bien plus élégante.

Démonstration. Soit $\mathcal{M} = (Q, q_0, F, \Sigma, \Gamma, \delta)$ une machine de Turing possédant un ruban mémoire bi-infini. L'idée est de remplacer le ruban bi-infini sur Γ

$$\cdots m_{-3}m_{-2}m_{-1}m_0m_1m_2\cdots$$

On "plie" le ruban bi-infini en deux.

par un ruban infini (à droite) sur un alphabet de couples[25]

$$\binom{m_0}{\$}\binom{m_1}{m_{-1}}\binom{m_2}{m_{-2}}\binom{m_3}{m_{-3}}\cdots$$

Pour nous simplifier la tâche, nous supposerons que la machine $\mathcal{M}$ est initialisée avec une configuration

$$q_0.\underline{\#}w\#,\ w = w_1\cdots w_k \in \Sigma^*, \tag{3}$$

où la cellule référencée porte l'indice 0. (Nous dérogeons quelque peu à nos conventions de débuter avec une configuration $q_0.\#w\underline{\#}$, mais cela permettra de simplifier les constructions envisagées et il n'est pas difficile de se convaincre que les résultats obtenus sont équivalents.) On définit une machine de Turing de ruban mémoire infini à droite

$$\mathcal{M}' = (Q', q_0', F', \Sigma', \Gamma', \delta')$$

comme suit :

Ce nouveau symbole \$ permet de savoir quand la machine se trouve sur la première case du ruban.

- l'alphabet de ruban de $\mathcal{M}'$ est $\Gamma' = \Gamma \times (\Gamma \cup \{\$\})$ où \$ est un nouveau symbole n'appartenant pas à Γ,
- l'alphabet d'entrée de $\mathcal{M}'$ est $\Sigma' = \Sigma \times \{\#\}$, i.e., les données fournies à la machine sont de la forme $\binom{\sigma_1}{\#}\cdots\binom{\sigma_r}{\#}$,
- le symbole blanc de $\mathcal{M}'$ est $\binom{\#}{\#}$,
- l'ensemble des états de $\mathcal{M}'$ est $Q' = (Q \times \{H, B\}) \cup \{q_0'\}$ où q_0' est un nouvel état qui est l'état initial de $\mathcal{M}'$. La seconde composante H ou B d'un état sera utilisée pour déterminer si le curseur de $\mathcal{M}$ référence une cellule d'indice négatif (on se placera alors dans le bas du ruban de $\mathcal{M}'$ rappelé par la composante B comme "Bas") ou, par contre, si la cellule référencée de $\mathcal{M}$ est d'indice positif ou nul (on se placera alors dans le haut du ruban de $\mathcal{M}'$ rappelé par la composante H comme "Haut").
- L'ensemble des états accepteurs est $F' = F \times \{H, B\}$.

[25] Un alphabet de couples est toujours un alphabet. La seule condition est que l'ensemble soit fini.

- Il nous reste à définir la fonction de transition δ'. Puisque $\mathcal{M}$ est initialisé avec (3), on convient naturellement que $\mathcal{M}'$ est initialisée avec une configuration

$$q'_0.\underline{\binom{\#}{\$}}\binom{w_1}{\#}\cdots\binom{w_k}{\#}\binom{\#}{\#}.$$

Soient $\gamma, \tau, \gamma_1, \gamma_2 \in \Gamma$, $q, r \in Q$. Commençons par définir les transitions depuis l'état initial q'_0.

$$\text{Si } \delta(q_0,\gamma) = (q,\tau), \text{ alors } \delta'(q'_0, \binom{\gamma}{\$}) = ((q,H), \binom{\tau}{\$}),$$

$$\text{si } \delta(q_0,\gamma) = (q,R), \text{ alors } \delta'(q'_0, \binom{\gamma}{\$}) = ((q,H),R),$$

$$\text{si } \delta(q_0,\gamma) = (q,L), \text{ alors } \delta'(q'_0, \binom{\gamma}{\$}) = ((q,B),R).$$

Si $\mathcal{M}'$ n'est pas au début du ruban, i.e., la seconde composante de la cellule référencée contient un symbole différent de \$, alors

$$\text{si } \delta(q,\gamma_1) = (r,\tau), \text{ alors } \delta'((q,H), \binom{\gamma_1}{\gamma_2}) = ((r,H), \binom{\tau}{\gamma_2}),$$

$$\text{si } \delta(q,\gamma_2) = (r,\tau), \text{ alors } \delta'((q,B), \binom{\gamma_1}{\gamma_2}) = ((r,B), \binom{\gamma_1}{\tau}),$$

$$\text{si } \delta(q,\gamma_1) = (r,x),\ x \in \{L,R\}, \text{ alors } \delta'((q,H), \binom{\gamma_1}{\gamma_2}) = ((r,H),x),$$

$$\text{si } \delta(q,\gamma_2) = (r,L), \text{ alors } \delta'((q,B), \binom{\gamma_1}{\gamma_2}) = ((r,B),R),$$

$$\text{si } \delta(q,\gamma_2) = (r,R), \text{ alors } \delta'((q,B), \binom{\gamma_1}{\gamma_2}) = ((r,B),L).$$

Enfin, si $\mathcal{M}'$ est au début du ruban,

$$\text{si } \delta(q,\gamma) = (r,\tau), \text{ alors } \delta'((q,x), \binom{\gamma}{\$}) = ((r,x), \binom{\tau}{\$}),\ x \in \{H,B\},$$

$$\text{si } \delta(q,\gamma) = (r,R), \text{ alors } \delta'((q,H), \binom{\gamma}{\$}) = ((r,H),R),$$

$$\text{si } \delta(q,\gamma) = (r,L), \text{ alors } \delta'((q,H), \binom{\gamma}{\$}) = ((r,B),R),$$

$$\text{si } \delta(q,\gamma) = (r,R), \text{ alors } \delta'((q,B), \binom{\gamma}{\$}) = ((r,H),R),$$

$$\text{si } \delta(q,\gamma) = (r,L), \text{ alors } \delta'((q,B), \binom{\gamma}{\$}) = ((r,B),R).$$

Soit $w = w_1 \cdots w_k \in \Sigma^*$. Il est clair que les comportements de $\mathcal{M}$ et $\mathcal{M}'$ à partir de $q_0.\underline{\#}w\#$ et

$$q'_0.\underline{\begin{pmatrix}\#\\ \$\end{pmatrix}} \begin{pmatrix}w_1\\ \#\end{pmatrix} \cdots \begin{pmatrix}w_k\\ \#\end{pmatrix} \begin{pmatrix}\#\\ \#\end{pmatrix}$$

sont comparables.

■

Il nous serait loisible de définir des machines de Turing possédant des rubans mémoire multiples. Ainsi, la fonction de transition dépendrait des cellules référencées sur chacun des rubans et à chaque étape, on pourrait déplacer le curseur de référence ou écrire sur chacun des rubans. Une définition rigoureuse d'une telle machine et la démonstration de l'équivalence avec une machine à une seule bande ne constituent que de fastidieux exercices que nous préférons éviter. En fait, ces variantes dans la définition des machines de Turing peuvent avoir comme avantage d'être parfois plus simples à "programmer", plus rapides dans leurs exécutions ou encore, comme nous le verrons dans la remarque suivante, de diminuer le nombre de transitions, la taille de l'alphabet ou le nombre d'états.

Le lecteur préférera aussi éviter ces exercices !

Remarque II.6.3. Il arrive parfois qu'on définisse une machine de Turing de manière telle que sa fonction de transition soit de la forme

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}.$$

Ainsi, avec cette définition alternative, à chaque transition, la machine effectue une écriture sur le ruban suivie d'un déplacement vers la droite ou vers la gauche de la tête de lecture. Cela a pour effet d'obtenir des machines plus compactes. En effet, avec notre définition initiale, deux transitions sont nécessaires pour remplacer une seule transition de cette nouvelle fonction δ. Si $\delta(p, \gamma) = (q, \tau, x)$, avec $p, q \in Q$, $\gamma, \tau \in \Gamma$, $x \in \{L, R\}$, on représentera cette transition par

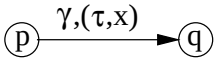


FIGURE II.17. Représentation sagitale "compacte".

On peut dès à présent noter que l'exemple qui suit, sera repris sous une forme altérée pour obtenir un exemple de fonction non calculable.

Exemple II.6.4. Avec la définition alternative donnée à la remarque précédente, on peut définir le problème du "*castor affairé*[26]". Dans ce problème, on considère une machine de Turing d'alphabet $\{\#, 1\}$, de ruban bi-infini et possédant n états[27]. La question qui est posée est de déterminer, partant d'un ruban ne contenant que des caractères #, le nombre maximum de 1

[26] En anglais, "Busy Beaver Problem".

[27] L'état accepteur n'est pas comptabilisé dans ces n états.

consécutifs[28] qu'il est possible d'écrire sur le ruban avec une machine à n états pour finalement aboutir dans un unique état accepteur (ce dernier état n'étant pas comptabilisé dans les n états de la machine). La fonction qui à n associe ce nombre maximum pour les machines à n états s'appelle la *fonction du castor affairé* et se note $BB(n)$. La machine représentée à la figure II.18 montre que $BB(2) \geq 4$.

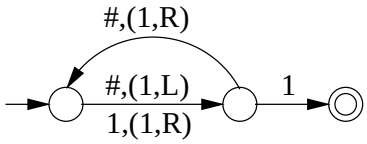


FIGURE II.18. $BB(2) \geq 4$.

De même, on rencontre parfois des définitions de machines de Turing possédant un seul ruban mémoire mais plusieurs têtes de lecture-écriture. Une fois encore, on peut montrer l'équivalence avec les machines classiques.

Nous en arrivons au concept important de machine de Turing non déterministe. La différence primordiale avec le cas déterministe est qu'on n'est plus en présence d'une fonction de transition mais bien d'une *relation de transition*.

Définition II.6.5. Une *machine de Turing non déterministe* est la donnée d'un 6-uple $\mathcal{M} = (Q, q_0, F, \Sigma, \Gamma, \Delta)$ où les 5 premières composantes sont définies comme dans le cas des machines déterministes et où

$$\Delta \subset Q \times \Gamma \times Q \times (\Gamma \cup \{L, R\})$$

est une *relation de transition*. Cette relation est un sous-ensemble fini car tous les facteurs du produit cartésien sont eux-mêmes des ensembles finis.

Cela signifie en particulier qu'à un couple $(q, \gamma) \in Q \times \Gamma$, il peut correspondre plus d'un couple de $Q \times (\Gamma \cup \{L, R\})$ voire aucun. Notons que puisque la relation Δ est finie, à un couple $(q, \gamma) \in Q \times \Gamma$, il correspond tout au plus un nombre fini de couples, ce nombre étant majoré par $\#Q.(\#\Gamma + 2)$. Si $(p, \gamma, q, \tau) \in \Delta$, on utilisera la même représentation sagitale que dans le cadre déterministe.

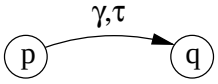


FIGURE II.19. Représentation sagitale.

L'exemple suivant illustre bien la notion de non déterminisme. A chaque étape, on a le choix entre plusieurs transitions.

[28] Analogie avec le castor qui empile des rondins, la machine empile des 1...

Exemple II.6.6 (Générateur aléatoire de nombres). Soit la machine de Turing non déterministe représentée à la figure II.20. Suivant les transitions

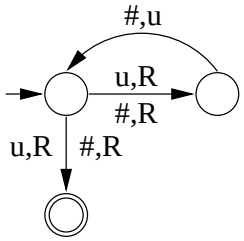


FIGURE II.20. Un générateur aléatoire de nombres.

choisies à chaque étape, partant de la configuration $q_0.\underline{\#}$, on aboutira dans une configuration $f.\#u^n\underline{\#}$ pour un certain $n \in \mathbb{N}$.

Les machines de Turing non déterministes peuvent uniquement être utilisées comme *accepteur*. En effet, on les voit mal intervenir dans le calcul d'une fonction puisque, d'une exécution à l'autre, le comportement de la machine pourrait changer et fournir des résultats différents (voire pas de résultat pour certaines exécutions). Il suffit de repenser à l'exemple du générateur aléatoire. Partant de $q_0.\underline{\#}$, la machine peut fournir "selon son humeur" n'importe quel entier.

Définition II.6.7. Soit $\mathcal{M} = (Q, q_0, F, \Sigma, \Gamma, \Delta)$ une machine de Turing non déterministe. Le mot $w \in \Sigma^*$ est *accepté* par $\mathcal{M}$ s'il existe une suite finie de transitions permettant de passer de la configuration $q_0.\#w\underline{\#}$ à une configuration $f.x$ où $f \in F$ et $x \in \Gamma^*$. On peut naturellement définir le *langage accepté* par $\mathcal{M}$ comme l'ensemble $L(\mathcal{M})$ des mots acceptés par $\mathcal{M}$.

Pour les connaisseurs, il s'agit simplement d'un automate fini non déterministe "déguisé" en machine de Turing.

Exemple II.6.8. Considérons la machine $\mathcal{M}$ représentée à la figure II.21. Par exemple, les mots

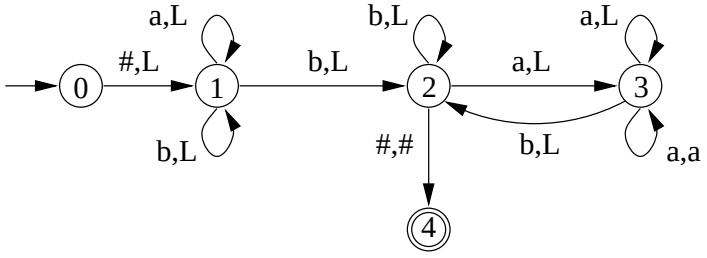


FIGURE II.21. Une machine de Turing non déterministe.

$$b, bb, bbb, \ldots, ba, baa, baaa, \ldots, baba, \ldots$$

sont acceptés par $\mathcal{M}$. Pour $baba$, on a la suite de configurations

$$0.\#baba\underline{\#},\ 1.\#bab\underline{a}\#,\ 1.\#ba\underline{b}a\#,\ 2.\#b\underline{a}ba\#,$$

$$3.\#\underline{b}aba\#, 2.\underline{\#}baba\#,\ 4.\underline{\#}baba\#.$$

Par exemple, $a, aa, aaa, \ldots$ ne sont pas acceptés. Le mot *baaba* est lui aussi accepté, mais il est possible à partir de ce mot d'obtenir une suite infinie de configurations

$$0.\#baaba\underline{\#},\ \ldots,\ 3.\#b\underline{a}aba\#,\ 3.\#b\underline{a}aba\#,\ \ldots$$

Insistons donc une fois encore sur le fait que, pour qu'un mot soit accepté, il suffit d'une suite finie de configurations menant à un état accepteur.

Il est clair qu'une machine de Turing déterministe est un cas particulier de machine non déterministe. La proposition suivante nous montre que les machines non déterministes ne permettent pas d'accepter de nouveaux langages.

Proposition II.6.9. *L'ensemble des langages acceptés par les machines de Turing non déterministes et celui des langages acceptés par les machines déterministes coïncident.*

Démonstration. Il suffit de se convaincre que le langage accepté par une machine non déterministe l'est aussi par une machine déterministe. Soit $w \in \Sigma^*$ un mot. Lorsqu'on dispose d'une machine déterministe la suite de transitions résultant d'une exécution est univoquement déterminée par la configuration initiale $q_0.\#w\underline{\#}$. Si la machine est non déterministe, on a éventuellement plusieurs possibilités que l'on va lister au moyen d'un arbre.

Exemple II.6.10. Pour la machine de la figure II.21 et le mot *bbb*, on a l'arbre de la figure II.22 décrivant toutes les possibilités d'exécutions, i.e., toutes les suites de transitions, que l'on peut obtenir à partir de $0.\#bbb\underline{\#}$. Au niveau k de l'arbre, se trouve l'ensemble des configurations accessibles en k transitions.

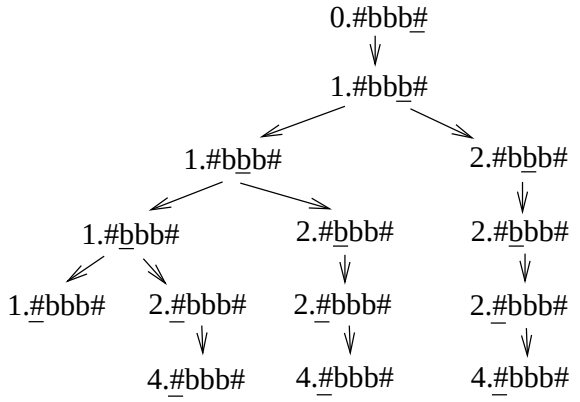


FIGURE II.22. L'arbre des configurations.

La relation de transition étant finie, un noeud a au plus un nombre fini de fils ($\#Q.(\#\Gamma + 2)$). Il suffit alors de vérifier si l'une des feuilles de l'arbre

Pensez à la thèse de Church-Turing.

correspond à un état accepteur. Il est clair que la construction de cet arbre peut être réalisée algorithmiquement. Par conséquent, le lecteur sera convaincu qu'on peut construire une machine déterministe construisant cet arbre niveau par niveau et testant l'apparition d'un état accepteur dans la machine non déterministe. Si tel est le cas, on décide que la machine déterministe bascule dans un état accepteur et s'arrête elle aussi. De cette façon, tout mot w accepté par la machine non déterministe le sera aussi par notre "algorithme" (donc par une machine déterministe[29]) et réciproquement.

■

Remarque II.6.11. Dans le cas d'une machine déterministe, les arbres considérés sont linéaires, i.e., chaque noeud qui n'est pas une feuille terminale a exactement un fils. Le déterminisme se traduit par une structure filiforme puisqu'aucun choix n'est possible. Par exemple, si on considère la machine de l'exemple II.2.4, on obtient, à partir de la configuration $0.\#u\#u\underline{\#}$, l'arbre suivant

$$
\begin{array}{c}
0.\#u\#u\underline{\#} \\
\downarrow \\
1.\#u\#\underline{u}\# \\
\downarrow \\
1.\#u\underline{\#}u\# \\
\downarrow \\
2.\#u\underline{u}u\# \\
\downarrow \\
2.\#uu\underline{u}\# \\
\downarrow \\
2.\#uuu\underline{\#} \\
\downarrow \\
3.\#uu\underline{u}\# \\
\downarrow \\
4.\#uu\underline{\#}
\end{array}
$$

Quand on parle, sans autre précision, de machine de Turing, il est sous-entendu qu'il s'agit d'une machine déterministe.

7. Fonctions non calculables

Un argument[30] de comptage permet de se convaincre de l'existence de fonctions non calculables. En effet, l'ensemble des fonctions calculables (ou récursives) est dénombrable[31] alors que l'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ ne l'est pas.

Pour exhiber une fonction non calculable, on peut tout comme dans le théorème I.4.17, utiliser un argument de *diagonalisation*. Puisque les fonctions calculables de $\mathfrak{F}_1$ sont dénombrables, on peut les énumérer : $f_0, f_1, f_2, \ldots$.

[29] Pensez une fois encore à la thèse de Church-Turing.

[30] Il s'agit du même argument que celui utilisé dans le théorème I.4.17.

[31] On peut décrire une machine de Turing à l'aide d'un mot sur un alphabet fini.

Soit g la fonction de $\mathfrak{F}_2$ définie par

$$g : \mathbb{N}^2 \to \mathbb{N} : (m, n) \mapsto f_m(n).$$

La fonction g n'est pas calculable. En effet, si elle l'était, alors la fonction $h \in \mathfrak{F}_1$ définie par

$$h(n) = g(n, n) + 1$$

le serait aussi. Puisque h est calculable, il existe un entier i tel que $h = f_i$. En d'autres termes,

$$h(n) = g(i, n), \ \forall n \in \mathbb{N}$$

et donc $h(i) = g(i, i)$ mais par définition, $h(i) = g(i, i) + 1$.

Remarque II.7.1. Le lecteur pourrait se demander pourquoi la fonction g obtenue ici n'est pas calculable, alors que la fonction h du théorème I.4.17 l'était (et qu'elle était obtenue essentiellement de manière identique). La raison est la suivante : bien qu'on puisse énumérer les machines de Turing, il est impossible de tester systématiquement si une machine donnée calcule effectivement une fonction (problème de l'arrêt). De manière équivalente, en générant toutes les chaînes de caractères devant donner lieu aux fonctions récursives, rien ne permet de garantir la présence de partie/prédicat sûr. Ceci empêche un calcul effectif.

Considérons un second exemple de fonction non calculable.

Définition II.7.2. Un entier n est *produit* par une machine de Turing $\mathcal{M}$ (déterministe) si

$$q_0.\#\underline{\#} \vdash^* f.\#u^n\underline{\#}.$$

La fonction β présentée ci-après est analogue à la fonction BB du castor affairé, à ceci près qu'elle est définie pour des machines de Turing "classiques".

Proposition II.7.3. *Soit la fonction*[32] $\beta \in \mathfrak{F}_1$ *qui à un entier* n *associe le plus grand entier* $\beta(n)$ *produit par une machine de Turing déterministe d'alphabet* $\{\#, u\}$ *et possédant* $n+1$ *états. La fonction* β *n'est pas calculable.*

Remarque II.7.4. Remarquons tout d'abord que pour un entier n fixé, le nombre de machines de Turing déterministes d'alphabet $\{\#, u\}$ et possédant $n + 1$ états est fini. Par conséquent, la fonction β est bien définie[33] pour tout n.

Démonstration. Remarquons que $\beta(n)$ vaut au moins n. Pour le voir, il suffit de considérer la machine $\mathcal{M}_n$ représentée à la figure II.23 et qui produit n. Soit $\mathcal{B}_n$ une machine de Turing ayant $n + 1$ états qui produit $\beta(n)$.

[32] Remarquons que les fonctions non calculables seront toujours définies par des périphrases. En effet, si une fonction est définie algorithmiquement, alors, en vertu de la thèse de Church-Turing, elle est calculable.

[33] On prend la borne supérieure d'un ensemble fini.

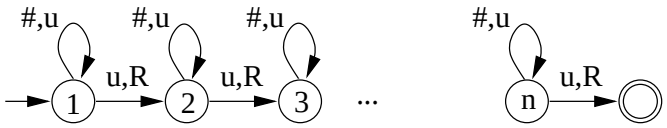


FIGURE II.23. La machine $\mathcal{M}_n$ produisant n.

Considérons l'enchaînement $\mathcal{B}_n\mathcal{M}_1$. La machine correspondante possède $n+2$ états car l'état accepteur de $\mathcal{B}_n$ (que l'on peut supposer unique) est identifié avec l'état initial de $\mathcal{M}_1$ (ainsi, $\mathcal{B}_n\mathcal{M}_1$ ne possède pas $n+3$ états mais bien $n+2$). Il est clair que $\mathcal{B}_n\mathcal{M}_1$ produit $\beta(n)+1$ et donc

$$\beta(n)+1 \leq \beta(n+1) \text{ ou encore } \beta(n) < \beta(n+1).$$

En d'autres termes, la fonction β est strictement croissante.

Procédons par l'absurde et supposons β calculable. Si tel est le cas, la fonction

$$n \mapsto \beta(2\,n)$$

est elle aussi calculable par une machine $\mathcal{T}$ possédant k états. La machine $\mathcal{M}_n\mathcal{T}$ produit $\beta(2\,n)$ et possède $k+n$ états. De là, on en tire que

$$\beta(2\,n) \leq \beta(k+n-1).$$

Or, β étant strictement croissante, il vient

$$2\,n \leq k+n-1, \quad \forall n \in \mathbb{N}$$

et donc

$$n \leq k-1, \quad \forall n \in \mathbb{N}$$

ce qui est impossible puisque k est une constante indépendante de n.

■

8. Machines de Turing universelles

En fin de comptes, on peut dire qu'une machine de Turing correspond à un programme fixé. En informatique, on rencontre des machines "programmables". Ainsi, un interpréteur est un programme permettant d'exécuter d'autres programmes. Ce concept est ici traduit par les machines de Turing universelles.

Une telle machine est une machine de Turing "classique" capable, grâce à des codages convenables, de simuler le comportement de n'importe quelle machine de Turing. Plus précisément, on fournit à la machine "universelle" un mot codant une machine $\mathcal{M}$ ainsi qu'un mot w. La machine universelle est utilisée pour simuler le comportement de $\mathcal{M}$ sur l'entrée w. Autrement dit, une machine universelle joue un peu le rôle d'interpréteur. Un interpréteur n'est autre qu'un programme informatique exécuté par le processeur et qui exécute le code formaté d'un autre programme. L'utilisation d'un interpréteur permet notamment d'éviter les "plantages" de programmes mal

conçus. En effet, des instructions qui seraient erronées ne sont pas directement transmises au processeur mais sont d'abord évaluées par l'interpréteur. Ce dernier peut alors décider de stopper si nécessaire le programme coupable. De même, un programme contenant une boucle infinie peut être stoppé lorsqu'il est interprété. Dans le contexte théorique abordé ici, on verra, qu'avec une machine de Turing universelle, on peut par exemple éliminer les configurations pendantes. En effet, ces dernières peuvent être détectées par la machine universelle qui simule le comportement d'une autre machine.

Exemple II.8.1. Voici un petit programme `Mathematica` permettant de simuler une machine de Turing. Cette dernière est encodée dans une liste `m` reprenant l'ensemble des états, l'état initial, l'ensemble des états accepteurs, l'alphabet de la machine (ne comprenant pas les symboles particuliers `"L"` et `"R"`) et la table encodant fonction de transition. La fonction `trans[m,c]` effectue une transition de la machine `m` à partir d'une configuration donnée `c` (elle permet en particulier de détecter les éventuelles configurations pendantes). Une configuration machine $q.u\underline{v}w$ est encodée par une liste à quatre éléments `{q,u,v,w}`. La machine encodée ci-dessous est celle de la figure II.3.

```
(* Une machine donnée par {Q, q0, F, A, d} *)

m={{1, 2, 3}, 1, {3}, {"a", "#"},
   {{1,"a","#",2}, {1,"#","#",3}, {2,"a","a",1}, {2,"#","R",1}}};

getQ[m_] := m[[1]];
getq0[m_]:= m[[2]];
getF[m_] := m[[3]];
getA[m_] := m[[4]];
getd[m_] := m[[5]];

(* Une configuration : etat, debut du ruban, tete de lecture, fin du ruban *)

trans[m_, c_] := Module[
    {t = Select[getd[m], #[[1]]==c[[1]] && #[[2]]==c[[3]]&][[1]],
     out = Table[0, {i,1,4}]},

    out[[1]] = t[[4]];

    If[t[[3]] == "R",
      out[[2]] = c[[2]] <> c[[3]];
      If[StringLength[c[[4]]] > 0,
        out[[3]] = StringTake[c[[4]], 1];
        out[[4]] = StringDrop[c[[4]], 1],
        out[[3]] = "#"; out[[4]] = "";
        ];
      ];
```

```
    If[t[[3]] == "L",
      If[StringLength[c[[2]]] == 0, Print["configuration pendante !"];
        out = 0,
        out[[2]] = StringDrop[c[[2]], -1];
        out[[3]] = StringTake[c[[2]], -1];
        out[[4]] = c[[3]] <> c[[4]];
        ]];

    If[MemberQ[getA[m], t[[3]]],
      out[[2]] = c[[2]];
      out[[3]] = t[[3]];
      out[[4]] = c[[4]];];

    Print["config:", c, " trans:", t, " -> ", out];
    out
    ]

(* un exemple *)

c = {1, "", "a", "a#a"};
trans[m, c]
Out := config:{1, , a, a#a} trans:{1, a, #, 2} -> {2, , #, a#a}
       {2, , #, a#a}

(* on continue, jusqu'a une configuration pendante ou un etat accepteur *)

run[m_, c_] :=
  NestWhile[trans[m,#] &, c,
    Length[#] == 4 && Not[MemberQ[getF[m], #[[1]]]] &]

run[m, {1, "", "a", "a#a"}]
Out := config:{1, , a, a#a} trans:{1, a, #, 2} -> {2, , #, a#a}
       config:{2, , #, a#a} trans:{2, #, R, 1} -> {1, #, a, #a}
       config:{1, #, a, #a} trans:{1, a, #, 2} -> {2, #, #, #a}
       config:{2, #, #, #a} trans:{2, #, R, 1} -> {1, ##, #, a}
       config:{1, ##, #, a} trans:{1, #, #, 3} -> {3, ##, #, a}
       {3, ##, #, a}
```

Il nous faut coder la machine de Turing $\mathcal{M}$ considérée. On place en tête du ruban de la machine universelle $\mathcal{U}$, le code $\rho(\mathcal{M})$ de $\mathcal{M}$ suivi du code $\rho(w)$ du mot w que la machine $\mathcal{M}$ doit traiter (w est un mot correspondant à une configuration mémoire de $\mathcal{M}$, il est donc écrit sur l'alphabet de ruban de $\mathcal{M}$). La machine de Turing universelle disposant dès lors de ces informations pourra facilement simuler sur son ruban l'exécution de $\mathcal{M}$ à partir de la configuration mémoire w. L'exemple II.8.2 donné plus bas est vraiment révélateur des codages utilisés.

Soit la machine $\mathcal{M} = (Q, q_0, F, \Sigma, \Gamma, \delta)$. Si $Q = \{q_0, q_1, \ldots, q_\alpha\}$ où nous supposons sans aucune restriction que $F = \{q_\alpha\}$, alors on code un état de $\mathcal{M}$ par

$$\rho(q_i) = u^{i+1}, \quad \forall i = 0, \ldots, \alpha.$$

Si $\Gamma = \{\gamma_1, \ldots, \gamma_\beta\}$, on pose

$$\rho(\gamma_i) = u^{i+2}, \quad \forall i = 1, \ldots, \beta$$

et

$$\rho(L) = u, \quad \rho(R) = u^2.$$

Il nous faut à présent coder un mot quelconque $m = \gamma_{i_1} \cdots \gamma_{i_k} \in \Gamma^*$. Puisqu'une lettre est un mot de longueur 1, il pourrait exister une ambiguïté entre le codage des lettres et celui des mots. Pour cette raison, on introduit la fonction ρ' définie par

$$\rho'(m) = *\rho(\gamma_{i_1}) * \cdots * \rho(\gamma_{i_k}) * .$$

Le symbole $*$ est un nouveau symbole de l'alphabet de ruban de $\mathcal{U}$. En particulier, $\rho'(\varepsilon) = **$ et $\rho'(\gamma_i) = *\rho(\gamma_i)* = *u^{i+2}*$. Passons au codage de la fonction de transition δ de $\mathcal{M}$. Si $\delta(q_i, \gamma_j) = (q, x)$ avec $q_i, q \in Q$, $\gamma \in \Gamma$ et $x \in \Gamma \cup \{L, R\}$, alors on code la fonction de transition par

$$\rho_{ij} = *\rho(q_i) * \rho(\gamma_j) * \rho(q) * \rho(x)*$$

pour $0 \leq i \leq \alpha$ et $1 \leq j \leq \beta$ (si $i = \alpha$, on se trouve dans l'état accepteur et la machine de Turing s'arrête; on ne trouvera donc pas de code de la forme $\rho_{\alpha j}$). Il suffit alors de concaténer les différents ρ_{ij} pour construire la table de transition de la machine.

Une machine de Turing universelle $\mathcal{U}$ étant elle-même une machine de Turing, elle a également son propre ensemble d'états, ses alphabets d'entrée et de ruban (ce dernier contient au moins les symboles #, $*$ et u pour permettre les codages) et sa propre fonction de transition. Nous supposons que ces ensembles sont disjoints de ceux de $\mathcal{M}$. (En particulier, les machines auront des caractères blancs distincts. S'il n'y a aucune ambiguïté, ils seront tous deux dénotés par #, le contexte permettant de choisir.) Nous allons construire une machine universelle sans cependant en donner les détails les plus élémentaires. Ceci ne ferait qu'alourdir l'exposé de manière inutile et n'apporterait pas d'argument supplémentaire au lecteur. Sur le ruban de $\mathcal{U}$, nous allons placer

- le code de l'état initial de $\mathcal{M}$, $\rho(q_0)$,
- le code de la fonction de transition de $\mathcal{M}$ obtenu comme la concaténation des ρ_{ij},
- le code de la configuration mémoire de $\mathcal{M}$ calculé grâce à ρ' et enfin,
- l'extrémité droite de la configuration mémoire de $\mathcal{U}$ qui contient une zone de travail dans laquelle les calculs de $\mathcal{U}$ permettant de simuler $\mathcal{M}$ sont réalisés.

Le code de la configuration mémoire de $\mathcal{M}$ est lui-même divisé en trois parties

- le code de la partie de la configuration mémoire de $\mathcal{M}$ situé à gauche de la cellule référencée,
- le code de la cellule référencée et enfin,
- le code de la partie significative situé à droite de la cellule référencée.

Remarquez les séparations entre les champs donnés par $*$.

Ainsi, le ruban mémoire de $\mathcal{U}$ possède un préfixe de la forme

$$\#\text{ETAT} * \text{CODE}\delta * \text{CODE GAUCHE} * \text{CODE CAR} * \text{CODE DROIT}$$

où $ETAT$ est initialisé à $\rho(q_0)$ et où $CODE\delta$ est composé comme suit

$$\rho_{01}\rho_{02}\cdots\rho_{0\beta}\rho_{11}\cdots\rho_{\alpha-1\beta}.$$

L'exemple suivant va nous permettre d'illustrer ce codage.

Exemple II.8.2. Soit la machine simpliste donnée à la figure II.24. Cette

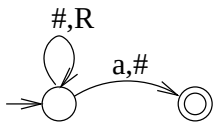


FIGURE II.24. Une machine à coder.

machine a deux états $Q = \{q_0, q_1\}$, avec $F = \{q_1\}$, et l'alphabet de ruban est $\Gamma = \{\gamma_1 = \#, \gamma_2 = a\}$. Si on code cette machine à l'aide d'une machine universelle, on a tout d'abord

$$\rho_{01} = *u * u^3 * u * u^2 * \text{ car } \delta(q_0, \gamma_1) = (q_0, R)$$

et

$$\rho_{02} = *u * u^4 * u^2 * u^3 * \text{ car } \delta(q_0, \gamma_2) = (q_1, \gamma_1).$$

Si la configuration mémoire de $\mathcal{M}$ est de la forme

$$\#a\underline{a}a,$$

on place dès lors en début de ruban

$$\#\underbrace{u}_{\rho(q_0)} * \underbrace{*u * u^3 * u * u^2*}_{\rho_{01}} \underbrace{*u * u^4 * u^2 * u^3*}_{\rho_{02}} * \underbrace{*u^3 * u^4*}_{\rho'(\#a)} * \underbrace{*u^4*}_{\rho'(a)} * \underbrace{*u^4*}_{\rho'(a)}.$$

Il est facile de distinguer les différents champs du codage. En effet, les ρ_{ij} sont séparés par $**$. Le codage de la fonction de transition est séparé du code de la mémoire par $***$ et les trois composantes de ce dernier sont également séparés par $***$. Réciproquement, si on connaît les conventions de codage employées, il est très facile de reconstruire une machine de Turing et le contenu de son ruban mémoire à partir d'un code syntaxiquement valide. Les constatations réalisées sur cet exemple sont générales.

La proposition suivante est alors immédiate.

Proposition II.8.3. *Les langages suivants sont décidables*

- $\{w \mid w$ *est le code* $\rho(\mathcal{M})$ *d'une machine de Turing* $\mathcal{M}\}$,
- $\{w \mid w$ *est le code* $\rho'(m)$ *d'un mot* $m\}$,
- $\{w \mid w$ *est le code* $\rho(\mathcal{M})\rho'(m)$ *d'une machine* $\mathcal{M}$ *et d'un mot* $m\}$.

Si $\mathcal{M} = (Q, q_0, F, \Sigma, \Gamma, \delta)$ *est une machine de Turing et* m *un mot appartenant à* Γ^**, il existe des machines de Turing réalisant respectivement*

- $q_0.\underline{\#} \vdash^* f.\#\rho(\mathcal{M})\underline{\#}$,
- $q_0.\underline{\#} \vdash^* f.\#\rho'(m)\underline{\#}$,
- $q_0.\underline{\#} \vdash^* f.\#\rho(\mathcal{M})\rho'(m)\underline{\#}$,
- $q_0.\#\rho'(m)\underline{\#} \vdash^* f.\#m\underline{\#}$

Démonstration. En effet, nous venons de voir comment réaliser de tels codages et le lecteur sera convaincu qu'ils peuvent être testés et réalisés algorithmiquement. Il s'agit de simples tests de nature syntaxique. Pour qu'un mot écrit sur le ruban soit un code valide, il y a un certain nombre de règles à respecter. Par exemple, pour le codage de la fonction de transition, on trouve des 4-uples

$$\cdots * u * u^3 * u * u^2 * *u * u^4 * u^2 * u^3 * \cdots$$

dont les composantes sont séparées par exactement un symbole $*$ et ces différents 4-uples sont eux-mêmes séparés par un symbole $*$, etc.... Il est donc possible de définir avec précisément un nombre fini de règles de syntaxe à respecter pour qu'un mot soit un code valide.

■

Il est à présent clair que lorsque le ruban d'une machine $\mathcal{U}$ contient le code d'une machine $\mathcal{M}$ et le code d'une configuration mémoire w, il est possible de simuler le comportement de $\mathcal{M}$ ayant w sur son ruban. En effet, on dispose de toutes les informations nécessaires sur le ruban. Si la machine $\mathcal{M}$ est initialisée avec la configuration mémoire $\#m\underline{\#}$, pour la machine $\mathcal{U}$, on initialise CODE GAUCHE avec $\rho'(\#m)$, CODE CAR avec $\rho'(\#)$ et CODE DROIT est vide[34], i.e., il contient uniquement $\rho'(\varepsilon) = **$. Le champ ETAT est initialisé avec le code de l'état initial de $\mathcal{M}$. A tout moment, connaissant le code de la cellule référencée (grâce à CODE CAR) et l'état dans lequel se trouve $\mathcal{M}$ (grâce à ETAT), une simple lecture dans CODEδ permet de déterminer le comportement de $\mathcal{M}$ lorsqu'on effectue une transition (on recherche un ρ_{ij} dont les deux premiers champs sont égaux respectivement à ETAT et CODE CAR). Il suffit alors d'agir en conséquence et de modifier les champs correspondants sur le ruban de $\mathcal{U}$. Il faut modifier le champ ETAT ainsi que les champs CODE GAUCHE, CODE CAR et CODE DROIT. Comme annoncé, nous ne précisons pas les détails de construction.

[34] On ne code que la partie significative de la configuration mémoire.

Ainsi, nous avons immédiatement le résultat suivant[35].

Théorème II.8.4. *Il existe une machine de Turing $\mathcal{U}$ telle que, pour toute machine de Turing $\mathcal{M}$ et toute donnée m, l'exécution de $\mathcal{U}$ à partir de la configuration $\#\rho(\mathcal{M})\rho(m)\underline{\#}$ présente les mêmes caractéristiques que l'éxécution de $\mathcal{M}$ à partir de $\#m\underline{\#}$.*

Démonstration. Cela résulte de la discussion effectuée précédemment.

■

Remarque II.8.5 (Elimination des configurations pendantes). La machine universelle $\mathcal{U}$ simulant le comportement de $\mathcal{M}$, il est possible grâce à $\mathcal{U}$ d'éliminer les configurations pendantes que l'on rencontrerait dans $\mathcal{M}$. En effet, dans la machine $\mathcal{M}$, il se peut que la cellule référencée soit en tête de ruban et que l'action prescrite par la fonction de transition corresponde à un déplacement de la tête de lecture vers la gauche. On est donc en présence d'une configuration pendante dans $\mathcal{M}$. Cependant, lorsqu'on utilise la machine $\mathcal{U}$ qui simule $\mathcal{M}$, on sait déterminer si on se trouve au début du ruban de $\mathcal{M}$ (il suffit de vérifier si CODE GAUCHE est vide) et, ayant accès au code de la fonction de transition de $\mathcal{M}$, on pourra déterminer si $\mathcal{M}$ est censé basculer dans une configuration pendante sans pour autant que $\mathcal{U}$ y bascule. On pourra par exemple décider que dans un tel cas de figure, la machine $\mathcal{U}$ boucle indéfiniment (on peut, par exemple, imposer un déplacement inconditionnel vers la droite de la tête de lecture de $\mathcal{U}$). L'utilisation d'une machine universelle permet donc un contrôle accru du comportement des machines de Turing "classiques" et nous pourrons donc supposer qu'une machine de Turing ne bascule jamais dans une configuration pendante lorsqu'on démarre d'une configuration machine $q_0.\underline{\#}$ ou $q_0.\#m\underline{\#}$. L'introduction faite en début de section doit à présent être tout à fait claire au lecteur les machines universelles jouent le rôle de compilateur ou d'interpréteur auxquels on fournit des codes de programmes. Lorsqu'un programme est erroné, son interprétation ne bloque pas l'ordinateur. L'interpréteur détecte l'erreur et la signale.

9. Langages récursifs et récursivement énumérables

Un *problème de décision* est un problème pour lequel la réponse est oui ou non. Ses données sont appelées *instances* de celui-ci. Une instance pour laquelle la réponse est oui (resp. non) est dite *positive* (resp. *négative*).

Exemple II.9.1. On peut considérer le problème simpliste de "déterminer si un nombre est premier". Les instances possibles du problème sont les nombres entiers. Ainsi, 7 est une instance positive du problème. Par contre 16 en est une instance négative.

[35] Y. Rogozhin a construit en 1982 une machine universelle sur 4 symboles et à 6 états simulant n'importe quelle machine, cf. Y. Rogozhin, Seven universal Turing machines. *Systems and theoretical programming. Mat. Issled.* **69** (1982), 76–90.

Pour tout problème, il est clair qu'on peut coder chaque instance par un mot sur un alphabet fini. En guise de première illustration, pour l'exemple précédent, on peut coder tous les entiers en les représentant en base 2, i.e., chaque entier sera représenté par un mot sur $\{0,1\}$, ou encore, comme de coutume dans le cadre des machines de Turing, en codant l'entier n par le mot $u^n \in \{u\}^*$. Ainsi, si on utilise un encodage binaire, des instances positives du problème donné dans l'exemple II.9.1 seront

$$\texttt{PRIMES} = \{10, 11, 101, 111, 1011, 1101, 10001, \ldots\}$$

En considérant de manière équivalente l'autre encodage, on a

$$u^2, u^3, u^5, u^7, u^{11}, u^{13}, u^{17}, \ldots$$

On considérera comme instance négative du problème, non seulement les codages des données pour lesquelles la réponse au problème est non, mais aussi les mots représentant des codages non valides. Ainsi, pour le problème II.9.1, on considérera comme des instances négatives les mots

$$100, 110, 1000, 123, aa, a1b, \ldots$$

Les trois premiers sont des codes valides de nombres composés et les trois derniers ne sont pas des représentations binaires valides.

Par conséquent, on peut dire qu'un problème P est complètement caractérisé par le langage L_P formé des codages de ses instances positives. Nous nous autoriserons donc à ne pas dissocier le problème P du langage L_P.

Définition II.9.2. Un problème P est *décidable* si le langage L_P est décidable. Dans le cas contraire, on dit que P est *indécidable*. Il est clair que le problème de l'exemple II.9.1 est décidable[36].

Jusqu'à présent, notre motivation première a toujours été de caractériser les problèmes pour lesquels on dispose d'une procédure effective de calcul. Si un problème est indécidable, cela signifie donc qu'il n'existe pas de procédure effective pour le résoudre. Un argument de comptage nous convainc facilement de l'existence de problèmes indécidables. En effet, les machines de Turing pouvant être codées par un mot sur un alphabet fini, l'ensemble des machines de Turing est donc dénombrable. Par contre, l'ensemble de tous les langages sur un alphabet fini n'est pas dénombrable[37]. Il existe donc plus de langages, c'est-à-dire de problèmes, que de langages décidables.

[36]Pour savoir si n est premier, il suffit de tester tous les diviseurs potentiels compris entre 2 et $\sqrt{n}$.

[37]Pour ce faire, montrons que l'ensemble des sous-ensembles d'un ensemble infini dénombrable n'est pas dénombrable. Ce résultat est une fois encore une simple application de la technique de diagonalisation. Soient $A = \{a_1, a_2, \ldots\}$ un ensemble dénombrable infini et $\mathcal{P}(A)$ l'ensemble des parties de A. Procédons par l'absurde et supposons $\mathcal{P}(A) = \{S_1, S_2, \ldots\}$ dénombrable. Soit

$$D = \{a_i \mid a_i \notin S_i\}.$$

Il existe donc k tel que $D = S_k$. Par définition de D, $a_k \in D \Leftrightarrow a_k \notin S_k$ d'où une contradiction.

Dans la suite, on ne précisera pas forcément le codage employé pour encoder les instances d'un problème. Pour convaincre le lecteur qu'un tel codage est possible, nous allons présenter quelques problèmes classiques que nous étudierons plus en détail plus tard[38].

Exemple II.9.3. Le problème de la couverture des sommets ou *vertex cover* (VC) est défini comme suit. Etant donné un graphe (simple) non orienté G et un entier n, la question posée est de savoir s'il existe une couverture de G formée de n sommets. Une instance du problème est donc formée d'un graphe et d'un entier et la réponse attendue est oui ou non.

Quelques rappels de théorie des graphes sont nécessaires. Un *graphe* $G = (V, E)$ est la donnée d'un ensemble fini V de *sommets* (vertex, vertices) et d'une partie E de $V \times V$, l'ensemble d'*arêtes* (edges) ou *arcs*. Dans le cas d'un graphe non orienté, on suppose que si (v_1, v_2) appartient à E, alors (v_2, v_1) aussi – autrement dit, E est une relation binaire symétrique. Si (v_1, v_2) appartient à E, on appelle v_1 et v_2 les *extrémités* de l'arête. Enfin, une *couverture* d'un graphe $G = (V, E)$ est un sous-ensemble $W \subset V$ tel que toute arête de E possède au moins une de ses extrémités dans W.

Soit le graphe G représenté à la figure II.25 où $V = \{a, b, c, d, e\}$ et

$$E = \{(a, c), (a, b), (b, c), (c, d), (c, e), (d, e)\}.$$

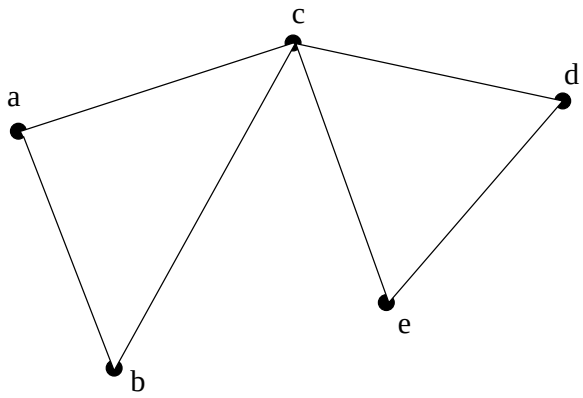


FIGURE II.25. Un graphe $G = (V, E)$.

Il est clair que $(G, 2)$ est une instance négative du problème. Par contre $(G, 3)$ en est une instance positive. On peut par exemple considérer la couverture $W = \{a, c, d\}$. Pour coder les instances d'un tel problème, on peut par exemple procéder comme suit. Considérer un premier champ contenant la donnée n du problème, suivi par le codage du graphe où l'on procède comme suit. Tout d'abord, on encode la taille du graphe, à savoir $\#V$. Ensuite, on peut considérer la *matrice d'incidence* A du graphe définie par

[38]Dans le chapitre suivant, on s'intéressera à leur complexité temporelle.

$A_{ij} = 1$ si et seulement si le graphe G possède une arête entre les sommets i et j. Ici, cette matrice est de la forme

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}.$$

Ainsi, l'instance $(G, 3)$ peut être codée par

$$\underbrace{3}_{n}\#\underbrace{5}_{\#V}\#\underbrace{0\#1\#1\#0\#0}_{\text{ligne 1}}\#\underbrace{1\#0\#1\#0\#0}_{\text{ligne 2}}\#\underbrace{1\#1\#0\#1\#1}_{\text{ligne 3}}\#$$
$$\underbrace{0\#0\#1\#0\#1}_{\text{ligne 4}}\#\underbrace{0\#0\#1\#1\#0}_{\text{ligne 5}}.$$

D'autres codages sont envisageables. En effet, puisque la matrice d'incidence est symétrique et possède des 0 sur la diagonale principale, on peut se contenter d'encoder les éléments de la matrice se trouvant au-dessus de la diagonale principale sans aucune perte d'information. On obtient dès lors le codage plus compact suivant,

$$3\#5\#\underbrace{1\#1\#0\#0}\#\underbrace{1\#0\#0}\#\underbrace{1\#1}\#\underbrace{1}.$$

Ce problème est clairement décidable. En effet, il suffit de passer en revue tous les sous-ensembles de V de taille n et de vérifier si l'un d'eux forme une couverture de G. Ces constructions peuvent être réalisées aisément de manière algorithmique. On peut déjà noter que si la taille du graphe est importante, ce problème peut très vite nécessiter de nombreux tests (le nombre de sous-ensembles de cardinal k d'un ensemble de taille ℓ vaut C_ℓ^k; à titre indicatif, si un graphe possède 30 sommets et qu'on recherche une couverture de taille 12, $\mathrm{C}_{30}^{12} = 8\,6493\,225$).

Exemple II.9.4. Le problème du voyageur de commerce (*travel-salesman problem* TS). Dans ce problème (ou plus exactement dans la variante considérée ici), on considère un graphe étiqueté $G = (V, E)$ où E est, cette fois, une partie de $V \times \mathbb{N} \times V$. On considère n villes, représentées par les sommets d'un graphe. Si une route existe entre deux villes, alors une arête existe dans le graphe correspondant. L'étiquette de cette arête représente le coût de transport pour se déplacer d'une ville à l'autre[39]. Etant donnés un entier ℓ, n villes et la matrice des coûts de transport, le problème est de savoir s'il est possible de parcourir l'ensemble des villes pour un coût total inférieur à ℓ. Considérons cinq villes a, b, c, d, e. On a la situation représentée à la

[39]Nous supposons ici le graphe non orienté : si on peut se déplacer d'une ville A à une ville B, alors on peut aussi se déplacer de B en A au même coût. On est donc en présence d'une relation symétrique. On pourrait considérer une variante de ce problème dans lequel des "sens uniques" seraient présents ou encore pour lequel les coûts de transport seraient différents suivant le sens de parcours employé. Dans ce dernier cas, on considérerait des graphes orientés.

figure II.26. Il est tout d'abord clair que le problème n'a de solution que

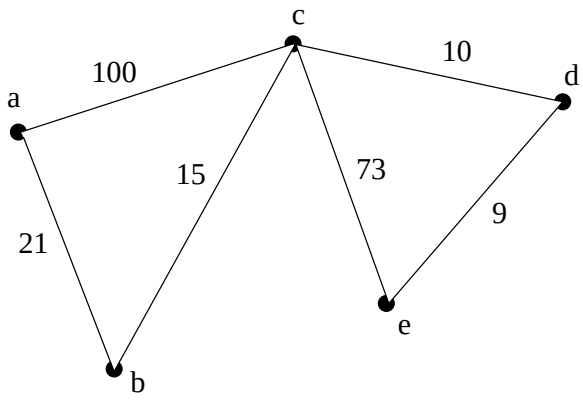


FIGURE II.26. Villes et coûts de transport.

si le graphe est connexe. On peut bien évidemment considérer la matrice associée,

$$\begin{pmatrix} 0 & 21 & 100 & 0 & 0 \\ 21 & 0 & 15 & 0 & 0 \\ 100 & 15 & 0 & 10 & 73 \\ 0 & 0 & 10 & 0 & 9 \\ 0 & 0 & 73 & 9 & 0 \end{pmatrix}.$$

Si $\ell = 40$, on peut donc coder cette instance en procédant comme précédemment par

$$\underbrace{40}_{\ell}\#\underbrace{5}_{\#V}\#\underbrace{0\#21\#100\#0\#0}\#\underbrace{21\#0\#15\#0\#0}\#\underbrace{100\#15\#0\#10\#73}\#$$

$$\underbrace{0\#0\#10\#0\#9}\#\underbrace{0\#0\#73\#9\#0}.$$

On peut vérifier que cette instance est négative. En effet, le chemin de coût minimum est donné par le chemin (e, d, c, b, a) de coût 55.

Ce problème est lui aussi décidable. En effet, puisque le graphe est fini, il n'existe qu'un nombre fini de chemins passant par tous les sommets du graphe[40]. Pour chacun de ces chemins, on calcule le coût total correspsondant et on vérifie s'il est ou non inférieur à ℓ. Une fois encore, si le nombre de villes considéré est important, le nombre de chemins à considérer augmente rapidement.

[40]Par exemple, on peut borner supérieurement la longueur des chemins passant par l'ensemble des sommets du graphe par $(n-1)^2$. En effet, si $a_1, \ldots, a_n$ sont les sommets du graphe et ν une permutation de l'ensemble $\{1, \ldots, n\}$, alors, pour aller de a_{ν_1} à a_{ν_2}, on emprunte un chemin de longueur au plus $n-1$. Il en va de même pour aller de a_{ν_2} à a_{ν_3} et ainsi de suite jusqu'au chemin joignant $a_{\nu_{n-1}}$ à a_{ν_n}. On aura donc au maximum $n-1$ chemins de longueur $n-1$ pour parcourir l'ensemble des sommets. Cette borne n'est bien sûr pas optimale mais est suffisante pour nos considérations.

Exemple II.9.5. Le problème de l'arrêt : étant donné une machine de Turing $\mathcal{M}$ et un mot w, la question qui est posée est de savoir si la machine $\mathcal{M}$ accepte ou non w. Au vu de la section précédente, il est clair que les instances de ce problème peuvent être codées (se rappeler les codages $\rho(\mathcal{M})$ et $\rho(w)$). Nous verrons dans la section suivante que ce problème est indécidable. Il n'y a donc pas de procédure algorithmique permettant de décider, d'une manière générale, si une machine quelconque s'arrête à partir d'un mot donné.

Exemple II.9.6. Le problème de satisfaisabilité de formules (*satisfaisability problem* SAT). Il s'agit d'un problème issu de la logique propositionnelle[41]. Soient des variables propositionnelles $x_1, \ldots, x_n$. Une *clause* est une disjonction de variables propositionnelles ou de négations de variables propositionnelles; par exemple

$$\neg x_1 \vee x_2 \vee \neg x_3.$$

Une formule sous *forme normale conjonctive* est une conjonction de clauses. Par exemple,

$$\begin{aligned} \phi_1 &\equiv (\neg x_1 \vee x_2 \vee \neg x_3) \wedge x_1 \wedge (\neg x_1 \vee x_4) \\ \phi_2 &\equiv (x_1 \vee x_2) \wedge x_3 \\ \phi_3 &\equiv x_1 \wedge \neg x_1 \end{aligned}$$

Une formule est *satisfaisable* s'il existe une distribution de valeurs de vérité des variables propositionnelles qui la rende vraie[42]. Il est commode de poser $x = 1$ (resp. $x = 0$) si la valeur de vérité de x est vraie (resp. fausse). La formule ϕ_1 est satisfaisable, il suffit de considérer la distribution des valeurs de vérité

$$x_1 = 1,\ x_2 = 1,\ x_3 = 1,\ x_4 = 1.$$

[41] On y construit des formules propositionnelles ou propositions. Leur syntaxe est régie par certaines règles de construction à partir d'un ensemble A de variables propositionnelles et des connecteurs propositionnels $\wedge$, $\vee$, $\neg$, $\rightarrow$ et $\leftrightarrow$. Lorsqu'on s'intéresse non seulement à l'aspect syntaxique mais aussi sémantique, on introduit la notion de *distribution de valeurs de vérité* qui n'est autre qu'une fonction de A à valeurs dans $\{0, 1\}$.

[42] Rappelons les tables de vérité des cinq connecteurs logiques,

x	$\neg x$
0	1
1	0

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

x	y	$x \leftrightarrow y$
0	0	1
0	1	0
1	0	0
1	1	1

.

On vérifie facilement que $x \rightarrow y$ est logiquement équivalent à $\neg x \vee y$ et que $x \leftrightarrow y$ est équivalent à $(x \rightarrow y) \wedge (y \rightarrow x)$.

Ce n'est pas la seule distribution qui rende ϕ_1 vraie, les valeurs de vérité $x_1 = 1$, $x_2 = 0$, $x_3 = 0$ et $x_4 = 1$ conviennent tout aussi bien. On vérifie que ϕ_2 est aussi satisfaisable mais ϕ_3 ne l'est pas[43]. Une instance du problème SAT est une formule de la logique propositionnelle mise sous forme normale conjonctive et la question posée est de savoir s'il existe une distribution des valeurs de vérité qui satisfait cette formule. Ce problème est encore une fois clairement décidable. Si la formule contient n variables, les distributions possibles des valeurs de vérité sont en nombre 2^n. Il suffit de passer en revue l'ensemble de ces possibilités pour résoudre le problème.

Voici encore quelques exemples de problèmes indécidables[44] classiques. Les deux premiers font référence aux grammaires algébriques présentées dans le cours de théorie des automates et langages formels. Les deux derniers sont des problèmes célèbres qui ont été largement étudiés.

- Etant donné une grammaire algébrique G générant un langage sur l'alphabet Σ, décider si le langage $L(G)$ généré par G est égal à Σ^*. On appelle souvent ce problème, le problème de l'universalité.
- Etant données deux grammaires algébriques G et H, décider si l'intersection $L(G) \cap L(H)$ est vide.
- Citons également le célèbre *dixième problème de Hilbert* qui consiste à déterminer si l'équation polynômiale $P(X_1, \ldots, X_n) = 0$ où $P \in \mathbb{Z}[X_1, \ldots, X_n]$ possède ou non une solution entière[45].
- Signalons enfin le *problème de correspondance de Post* (Post Correspondence Problem, PCP) classique en combinatoire sur les mots. Il s'énonce comme suit. Une instance du problème est donnée par deux morphismes $f, g : \Sigma^* \to \Delta^*$ (il faut comprendre deux homomorphismes de monoïdes) et on demande s'il existe un mot non vide $w \in \Sigma^*$ tel que $f(w) = g(w)$. Par exemple, on peut considérer les morphismes f et g définis par

	a	b	c	d
f	a	$bbabb$	ab	a
g	ab	b	bba	aa.

  Ces deux morphismes forment une instance positive du problème car on peut vérifier que

$$f(dacbbccbaccaadd) = g(dacbbccbaccaadd).$$

  En toute généralité, le problème ainsi posé est indécidable. Cependant, si on impose certaines restrictions sur les instances admissibles, on peut obtenir des problèmes décidables. Ainsi, si $\#\Sigma = 2$,

[43] ϕ_3 est la négation de la tautologie $x_1 \vee \neg x_1$.

[44] Nous ne démontrerons pas leur indécidabilité.

[45] Yuri Matiyasevič, The diophantineness of enumerable sets, *Dokl. Akad. Nauk SSSR* **191** (1970), 279–282.

alors PCP est décidable. On obtient un résultat analogue[46] lorsque f est périodique, i.e., il existe $x \in \Delta^*$ tel que pour tout $y \in \Sigma^*$, $f(y)$ appartient à x^*.

Après ces quelques exemples, il paraît clair qu'une étude détaillée des langages décidables s'impose pour tenter de cerner les problèmes susceptibles d'être résolus algorithmiquement. Soient R, l'ensemble des langages décidables (encore appelés langages récursifs[47]) et RE, l'ensemble des langages acceptables (encore appelés langages récursivement énumérables).

Proposition II.9.7. *L'ensemble R des langages récursifs (i.e., décidables) est stable pour l'union, l'intersection, la concaténation, la complémentation et l'étoile de Kleene.*

Démonstration. Soient A et B deux langages récursifs. En d'autres termes, leurs fonctions caractéristiques χ_A et χ_B sont récursives. Il est clair que

$$\chi_{A\cup B} = \chi_A + \chi_B - \chi_A\chi_B,$$
$$\chi_{A\cap B} = \chi_A\,\chi_B \quad \text{et} \quad \chi_{\Sigma^*\setminus A} = 1 - \chi_A$$

sont encore des fonctions récursives.

Passons à la concaténation de A et de B. Soit $m = m_1 \cdots m_k$ un mot. On considère les $k+1$ factorisations $x_i y_i$ de m où $x_1 = \varepsilon$, $y_1 = m$, $x_2 = m_1$, $y_2 = m_2 \cdots m_k$,..., $x_k = m_1 \cdots m_{k-1}$, $y_k = m_k$, $x_{k+1} = m$, $y_{k+1} = \varepsilon$. Pour $i \in \{1, \ldots, k+1\}$, puisque A (resp. B) est décidable, on peut décider si x_i (resp. y_i) appartient à A (resp. B). Si on répond affirmativement aux deux questions pour une valeur de i, alors m appartient à AB qui est donc décidable.

Pour l'étoile de Kleene A^*, on procède de manière analogue. Pour un mot $m = m_1 \cdots m_k$, on considère toutes les factorisations possibles de m comprenant de 1 à k facteurs. Puisque A est décidable, on peut tester si ces facteurs appartiennent ou non à A.

■

Rappelons que tout langage décidable est acceptable, i.e., $R \subset RE$ (cf. remarque II.5.3). Si un langage est récursivement énumérable, il n'est pas nécessairement décidable et on pourrait donc le considérer comme peu intéressant à nos yeux. Cependant, si un langage L est acceptable, il est d'une certain façon *proche* d'un langage décidable. En effet, on dispose d'une machine de Turing qui fournit une réponse positive pour les mots de L et si un mot n'est pas dans L, la machine fournira une réponse négative ou ne s'arrêtera jamais (boucle infinie). Ainsi, lorsqu'un langage est acceptable, on parle parfois de problème *partiellement récursif* ou *partiellement décidable*.

[46] voir par exemple, T. Harju, J. Karhumäki, *Morphisms*, Handbook of Formal Languages, vol. 1, Springer, (1997) pour un exposé sur les restrictions décidables de PCP.

[47] Cette dénomination est naturelle puisque, si un langage est décidable, cela signifie que sa fonction caractéristique est calculable, c'est-à-dire *récursive*.

Il faudra attendre la section suivante pour avoir un exemple de langage acceptable mais indécidable et ainsi montrer que R est un sous-ensemble strict de RE.

Proposition II.9.8. *Un langage L est décidable si et seulement si L et son complémentaire sont acceptables.*

Démonstration. La condition est nécessaire. Soit $\mathcal{M}$, une machine de Turing calculant la fonction caractéristique δ_L de L. Considérons les machines $\mathcal{P}$ et $\mathcal{Q}$ représentées à la figure II.27. Il est clair que $\mathcal{MP}$ (resp. $\mathcal{MQ}$)

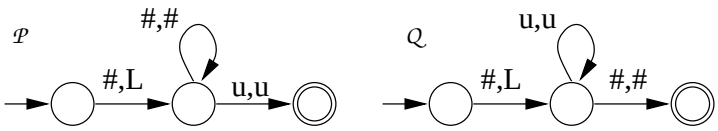


FIGURE II.27. L et son complémentaire sont acceptables.

accepte exactement L (resp. le complémentaire de L).

La condition est suffisante. Nous allons utiliser les codages introduits dans la preuve montrant que toute fonction calculable est récursive (cf. théorème II.4.7). Dans cette dernière, les configurations des machines de Turing étaient codées par des 4-uples de $\mathbb{N}^4$ et les transitions permettant de passer d'une configuration machine à la suivante par une fonction $F = (f_1, f_2, f_3, f_4) : \mathbb{N}^4 \to \mathbb{N}^4$. En particulier, avec les conventions employées, une machine atteint un état d'arrêt lorsque la première composante d'un 4-uple (qui contient le code de l'état dans lequel se trouve la machine) vaut 0. Il est clair que L et son complémentaire forment une partition de Σ^*. Par hypothèse, L et son complémentaire sont acceptés respectivement par des machines $\mathcal{M}$ et $\mathcal{M}'$. On note F et F' les fonctions codant les transitions de $\mathcal{M}$ et $\mathcal{M}'$ respectivement. Soient $m \in \Sigma^*$ et $\overline{x}$ (resp. $\overline{x}'$) un élément de $\mathbb{N}^4$ codant la configuration initiale de la machine $\mathcal{M}$ (resp. $\mathcal{M}'$) correspondant à l'entrée $\#m\#$. On applique à $\overline{x} = \overline{x_0}$ la fonction F pour obtenir $\overline{x_1}$ et à $\overline{x}' = \overline{x_0}'$ la fonction F' pour obtenir $\overline{x_1}'$. Si la première composante de $\overline{x_1}$ (resp. $\overline{x_1}'$) vaut 0, on en conclut que le mot appartient à L (resp. L'). Sinon, on applique une nouvelle fois F et F' pour obtenir $\overline{x_2}$ et $\overline{x_2}'$. On continue de la sorte jusqu'à obtenir i tel que $\overline{x_i}$ ou $\overline{x_i}'$ ait une première composante nulle. Ce sera toujours le cas, puisqu'on a une partition de Σ^*.

■

Remarque II.9.9. Dans cette preuve, il est nécessaire de travailler simultanément sur les deux machines $\mathcal{M}$ et $\mathcal{M}'$. En effet, imaginons qu'un mot

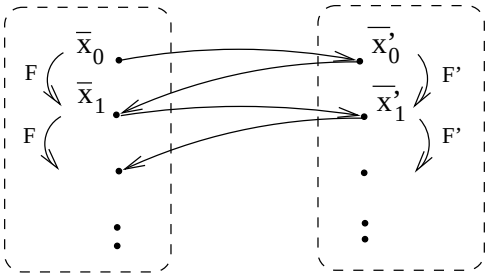


FIGURE II.28. Simulation en parallèle de $\mathcal{M}$ et $\mathcal{M}'$.

w appartienne au complémentaire de L. Dans ce cas, si on lançait exclusivement la machine $\mathcal{M}$ sur w, cette dernière ne s'arrêterait jamais[48] et on ne saurait pas décider[49] si w appartient ou non à L.

Voici maintenant l'explication de la terminologie *récursivement énumérable* pour les langages acceptables. On va retrouver l'idée de parallélisme développée dans la preuve précédente.

Proposition II.9.10. *Si $L \subseteq \Sigma^*$ est acceptable, alors on peut énumérer de manière effective les mots de L.*

Démonstration. Il nous faut exhiber un algorithme produisant exactement les mots acceptés par une machine de Turing $\mathcal{M}$ donnée. Tout d'abord, remarquons qu'il est aisé de produire un algorithme générant par ordre généalogique[50] croissant les mots de Σ^* dans son entièreté[51]. Notons

$$m_1, m_2, m_3, \ldots$$

les mots de Σ^* ainsi énumérés. Une méthode naïve mais erronnée consisterait à fournir, dans l'ordre, m_1 puis m_2, etc... à la machine $\mathcal{M}$ (on ne fournit m_2 à $\mathcal{M}$ que lorsque m_1 a été accepté et ainsi de suite) et d'énumérer ainsi les mots qu'elle accepterait. Dès qu'un premier mot m_i ne serait pas accepté par $\mathcal{M}$, la machine $\mathcal{M}$ ne s'arrêterait jamais et cette méthode ne permettrait donc pas d'énumérer les mots acceptés par $\mathcal{M}$. On serait irrémédiablement bloqué avec l'énumération de $m_1, \ldots, m_{i-1}$.

Il ne faut donc pas procéder séquentiellement mais bien **parallèlement** à l'énumération. A l'étape k, on produit les mots $m_1, \ldots, m_k$ (on peut

[48]Rappelons que nous avons supposé pouvoir éliminer les configurations pendantes.

[49]Le problème de l'arrêt étant indécidable, on ne peut pas prédire si $\mathcal{M}$ s'arrêtera ou non. Ce n'est pas parce qu'un nombre important de transitions a déjà été effectué qu'il ne faudra pas encore un nombre encore plus important de transitions pour atteindre l'état d'arrêt. Il n'est donc pas possible d'imposer une borne supérieure sur le nombre de transitions à effectuer.

[50]On classe d'abord les mots par longueur et au sein d'une même longueur, on utilise l'ordre du dictionnaire.

[51]Nous laissons au lecteur le soin d'imaginer une procédure permettant d'énumérer tous les mots de longueur n que l'on peut écrire sur un alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$. Par exemple, si $\Sigma = \{a, b\}$, alors $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, \ldots\}$.

omettre de cette liste les mots qui auraient déjà été acceptés lors d'une étape antérieure). Pour chacun de ces mots, on applique à partir de la configuration $q_0.\#m_i\#$ un maximum de k transitions au sein de $\mathcal{M}$. Deux cas de figures peuvent se présenter[52] :

- $q_0.\#m_i\#$ permet d'atteindre une configuration d'arrêt en au plus k transitions, le mot m_i est donc accepté par $\mathcal{M}$ et il est énuméré par l'algorithme (on pourra donc l'omettre lors des étapes ultérieures de l'algorithme),
- partant de $q_0.\#m_i\#$, la machine ne s'arrête pas après k transitions. Dans ce cas, on conserve le mot m_i pour l'étape $k+1$ de l'algorithme.

A l'étape $k+1$ de l'algorithme, on considère les mots $m_1, \ldots, m_k, m_{k+1}$ desquels on peut retirer ceux où l'appartenance à $L(\mathcal{M})$ a été prouvée. Pour les mots restants, on applique une transition de plus à la machine $\mathcal{M}$ et pour m_{k+1}, on lui applique un maximum de $k+1$ transitions.

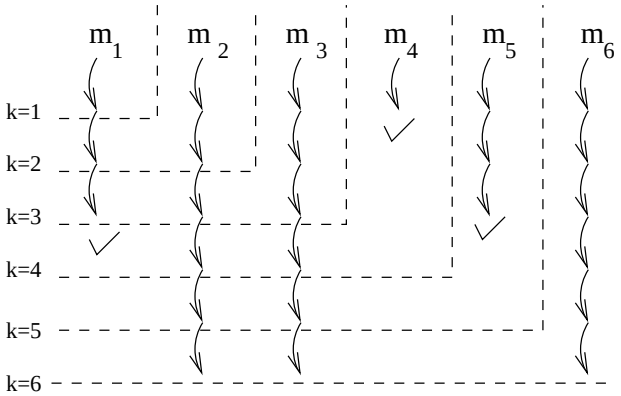


FIGURE II.29. A l'étape 6, les mots m_1, m_4, m_5 sont déjà énumérés.

Il est clair que cette procédure[53] permet d'énumérer les mots acceptés par $\mathcal{M}$ et évite l'écueil d'une suite infinie de transitions bloquant la génération des mots de $L(\mathcal{M})$.

■

Pour conclure cette section, la proposition suivante fournit un lien évident entre fonction calculable (sur un alphabet arbitraire) et le caractère décidable de son graphe.

Proposition II.9.11. *Soient Σ, Δ deux alphabets et μ un symbole n'appartenant pas à $\Sigma \cup \Delta$. Une fonction $f : \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{p \text{ arguments}} \to \Delta^*$ est calculable si*

[52]On suppose une fois encore que $\mathcal{M}$ n'a pas de configuration pendante, sinon il faudrait envisager un troisième cas.

[53]Même si cette procédure nécessite de nombreuses ressources mémoire, tout mot de L sera généré en un temps fini.

et seulement si le langage

$$L_f = \{x_1\mu\cdots\mu x_p\mu f(x_1,\ldots,x_p) \mid x_1,\ldots,x_p \in \Sigma^*\}$$

codant le graphe de f est décidable.

Démonstration. La condition est nécessaire. Pour le prouver, nous devrions exhiber une machine de Turing calculant la fonction caractéristique du langage L_f. Nous allons nous contenter de montrer qu'un algorithme testant si un mot m de $(\Sigma \cup \Delta \cup \{\mu\})^*$ appartient ou non à L_f peut être facilement obtenu. Il suffit de vérifier que

- m contient exactement p symboles μ,
- les p premiers champs contiennent des mots sur Σ,
- enfin, puisque f est calculable, on peut calculer au moyen d'une machine de Turing le mot $f(x_1,\ldots,x_p)$ et vérifier si celui-ci correspond au dernier champ de m.

Si on répond par l'affirmative à ces trois questions, alors on décide que m appartient à L_f.

La condition est suffisante. Etant donnés les mots $x_1,\ldots,x_p \in \Sigma^*$, nous devons exhiber une méthode pour calculer $f(x_1,\ldots,x_p)$. On procède comme suit. On énumère par ordre généalogique croissant les mots de Δ^* : $m_1, m_2, \ldots$. Pour le i-ème mot généré, on construit le mot

$$y = x_1\mu\cdots\mu x_p\mu m_i.$$

Puisque L_f est décidable, on sait décider si y appartient à L_f. Si tel est le cas, alors la valeur de $f(x_1,\ldots,x_p)$ est m_i.

■

10. Le problème de l'arrêt

Dans cette section, on présente un problème primordial pour l'informatique classique et la vérification de programmes : le problème de l'arrêt ainsi que ses variantes. Celui-ci a déjà été énoncé dans l'exemple II.9.5 et utilisé implicitement dans les preuves des propositions II.9.8 et II.9.10. Précisons-en la définition. Une instance du problème est donnée par une machine de Turing $\mathcal{M}$ et un mot m écrit sur son alphabet d'entrée. La question qui est posée est de déterminer si $\mathcal{M}$ s'arrête à partir de la configuration $q_0.\#m\underline{\#}$. En nous rappelant le codage des machines de Turing introduit dans le cadre des machines de Turing universelles, nous considérons le codage suivant du problème,

$$A = \{\rho(\mathcal{M})\rho'(m) \mid m \text{ est accepté par la machine } \mathcal{M}\}.$$

Théorème II.10.1. *Le problème de l'arrêt est indécidable.*

Démonstration. Procédons par l'absurde et supposons que le langage

$$A = \{\rho(\mathcal{M})\rho'(m) \mid m \text{ est accepté par la machine } \mathcal{M}\}$$

soit décidable, i.e., que χ_A soit calculé par une machine $\mathcal{M}_A$. Nous allons en déduire que la fonction β introduite à la proposition II.7.3 est alors calculable. Pour rappel, la fonction β associe à n le plus grand entier produit[54] par une machine d'alphabet $\{\#, u\}$ et ayant $n+1$ états (état accepteur compris).

Soit n un entier. Nous allons mettre en évidence une méthode permettant de calculer $\beta(n)$. Une machine de Turing à $n+1$ états étant principalement définie par sa fonction de transition, il est clair qu'on peut énumérer effectivement toutes les machines à $n+1$ états : $\mathcal{M}_1, \mathcal{M}_2, \ldots$. Pour la i-ème machine énumérée, puisque A est décidable, on peut décider si $\mathcal{M}_i$ s'arrête à partir de la configuration $q_0.\#\underline{\#}$. En effet, il suffit de tester si $\rho(\mathcal{M}_i)\rho'(\varepsilon)$ appartient à A. Si tel est le cas, on considère une machine de Turing universelle $\mathcal{U}$ simulant le comportement de $\mathcal{M}_i$ sur $\#\underline{\#}$. Si la simulation fournit une configuration de la forme $\#u^t\underline{\#}$, alors on compare le nombre obtenu au plus grand nombre produit par les machines considérées précédemment pour ne conserver que le plus grand. Si la configuration n'est pas de cette forme, alors $\mathcal{M}_i$ ne produit pas de nombre. Dans les deux cas, une fois ces opérations effectuées, on passe à la machine suivante $\mathcal{M}_{i+1}$. Enfin, si $\rho(\mathcal{M}_i)\rho'(\varepsilon)$ n'appartient pas à A, on passe directement à $\mathcal{M}_{i+1}$. En passant toutes les machines en revue, on est donc en mesure de calculer $\beta(n)$ d'où la contradiction annoncée.

■

Remarque II.10.2. En fait, dans la preuve précédente, on montre que le langage $\{\rho(\mathcal{M})\rho'(\varepsilon) \mid \mathcal{M}$ machine de Turing$\}$ qui est un sous-ensemble de A, est indécidable.

Ce théorème a une conséquence importante : il n'existe pas de procédé systématique qui, pour un algorithme arbitraire[55], permet de savoir s'il s'achève pour des données fixées. C'est la raison pour laquelle, lorsqu'on veut démontrer l'exactitude d'un algorithme, on procède en général en deux phases. On montre que

- 1. **si** l'algorithme s'achève, c'est avec le bon résultat,
- 2. l'algorithme s'achève.

Remarque II.10.3. Lorsqu'on a établi que $\mathcal{C} \subseteq \mathcal{R}$, une minimisation est apparue dans l'énoncé du théorème II.4.7. En effet, nous comprenons mieux à présent pourquoi cette minimisation non bornée est inévitable. Il est impossible de prédire de manière effective quand une machine de Turing arbitraire s'arrête à partir d'un mot donné.

La méthode de démonstration employée dans la preuve du théorème II.10.1 est une simple contraposition ($P \Rightarrow Q \equiv \neg Q \Rightarrow \neg P$ avec ici $P =$

[54] Un entier t est produit par $\mathcal{M}$ si $q_0.\#\underline{\#} \vdash^* f.\#u^t\underline{\#}$.

[55] Bien entendu, pour un algorithme particulier, il est probable que vous puissiez déterminer sur quelles données celui s'achève. Cependant, le problème de l'arrêt ne s'intéresse pas à un algorithme donné, mais bien à l'ensemble de tous les algorithmes.

"β non calculable" et Q = "A indécidable"). Dans un tel contexte, on parle parfois d'une méthode par *réduction*. La difficulté rencontrée par certains dans ce genre de raisonnement est qu'on utilise une double négation. En effet, on va nier le fait que β n'est pas calculable. Cela ne devrait cependant pas nous poser de problème !

Voici quelques exemples de réduction.

Exemple II.10.4. Démontrez que les langages suivants sont indécidables.

- $A_2 = \{\rho(\mathcal{M}) \mid \mathcal{M}$ s'arrête à partir de $q_0.\underline{\#}\}$,
- $A_3 = \{\rho(\mathcal{M}) \mid L(\mathcal{M}) \neq \emptyset\}$,
- $A_4 = \{\rho(\mathcal{M}) \mid L(\mathcal{M}) = \Sigma^*\}$,
- $A_5 = \{\rho(\mathcal{M})\rho(\mathcal{N}) \mid L(\mathcal{M}) = L(\mathcal{N})\}$,
- $A_6 = \{\rho(\mathcal{M})\rho(\mathcal{N}) \mid L(\mathcal{M}) \cap L(\mathcal{N}) \neq \emptyset\}$.

Théorème II.10.5. *Le langage associé au problème de l'arrêt est acceptable mais indécidable. Autrement dit, R est un sous-ensemble propre de RE.*

Démonstration. Soit

$$A = \{\rho(\mathcal{M})\rho'(m) \mid m \text{ est accepté par la machine } \mathcal{M}\}$$

le langage codant le problème de l'arrêt. Considérons une machine de Turing universelle $\mathcal{U}$ à laquelle on fournit une donnée w. Cette machine peut tester si w est de la forme $\rho(M)\rho'(m)$. Si tel n'est pas le cas, on convient que la machine $\mathcal{U}$ ne s'arrête jamais. Sinon, on peut simuler le comportement de $\mathcal{M}$ sur $\#m\underline{\#}$. Il suffit alors, comme de coutume, de convenir que $\mathcal{U}$ ne s'arrête pas si $\mathcal{M}$ atteint une configuration pendante. Bien évidemment, si $\mathcal{M}$ s'arrête (resp. ne s'arrête jamais) à partir de $\#m\underline{\#}$, alors $\mathcal{U}$ s'arrête aussi (resp. ne s'arrête pas non plus) (cf. théorème II.8.4). De cette manière, la machine $\mathcal{U}$ accepte exactement le langage A. Ceci termine la preuve.

■

La première partie du théorème suivant stipule que toute propriété non triviale des langages récursivement énumérables est indécidable.

Théorème II.10.6 (Théorème de Rice). *Pour tout sous-ensemble propre C de l'ensemble des langages acceptables, le problème de savoir si le langage accepté par une machine de Turing donnée appartient ou non à C est indécidable.*

Pour tout sous-ensemble C non vide de fonctions calculables, le problème de savoir si une machine de Turing donnée calcule une fonction de C est indécidable.

Remarque II.10.7. Au vu de la définition II.6.7, rappelons qu'un mot m est *accepté* par une machine de Turing $\mathcal{M}$ si, partant de la configuration $q_0.\#m\underline{\#}$, on peut atteindre un état accepteur (et ce, quel que soit le contenu du ruban mémoire finalement obtenu). Il est clair qu'en utilisant une

machine de Turing universelle, on peut imposer que le mot m soit *accepté* si

$$q_0.\#m\underline{\#} \vdash^* f.\underline{\#}.$$

En effet, grâce à la simulation de $\mathcal{M}$, la machine de Turing universelle employée peut encore effacer le contenu du ruban après que l'état d'arrêt ait été atteint dans $\mathcal{M}$. Dans la preuve qui suit, nous considérons l'acceptation au sens défini ci-dessus.

Démonstration. Commençons par quelques remarques préliminaires. La preuve proprement dite ne fera qu'adapter un raisonnement général à des cas particuliers.

A est un langage acceptable arbitraire.

Soient $\mathcal{M}$ une machine de Turing dont le langage accepté $L(\mathcal{M})$ est non vide et $\mathcal{A}$, une machine de Turing acceptant un langage $L(\mathcal{A}) = A$.

$L(\mathcal{M}) = L(\mathcal{RW}_m\mathcal{AM})$ $\Leftrightarrow m \in A$.

Remarquons que la machine $\mathcal{RW}_m\mathcal{AM}$, où $\mathcal{W}_m$ a pour fonction d'écrire m sur le ruban : $\#\underline{\#} \vdash^* \#m\underline{\#}$, accepte exactement les mêmes mots que $\mathcal{M}$ si et seulement si $m \in A$. En effet, partant d'une configuration $\#y\underline{\#}$, l'application de $\mathcal{RW}_m$ donne tout d'abord $\#y\#m\underline{\#}$. Si m est accepté par $\mathcal{A}$, en utilisant la remarque précédente, l'application de $\mathcal{A}$ donne $\#y\underline{\#}$ et il ne reste plus qu'à appliquer $\mathcal{M}$ (la machine $\mathcal{RW}_m\mathcal{AM}$ a donc bien exactement le même comportement que $\mathcal{M}$ à partir de $\#y\underline{\#}$). Par contre, si $m \notin L(\mathcal{A})$, alors $\mathcal{RW}_m\mathcal{AM}$ ne s'arrêtera jamais à partir de $\#y\#m\underline{\#}$ et ce, quel que soit y (et même si $y \in L(\mathcal{M})$).

D'où l'intérêt de l'hypothèse, $L(\mathcal{M}) \neq \emptyset$ car ici, on a $L(\mathcal{RW}_m\mathcal{AM}) = \emptyset$.

Enfin, la section relative aux machines de Turing universelles nous a convaincus qu'il est toujours possible de construire une machine $\mathcal{T}$ qui à un mot m associe le code $\rho(\mathcal{RW}_m\mathcal{AM})$.

On dispose de la propriété suivante. *Si le langage acceptable A est indécidable, alors le langage*

$$B = \{\rho(\mathcal{M}') \mid L(\mathcal{M}') = L(\mathcal{M})\}$$

est lui aussi indécidable. Pour ce faire, nous allons montrer la contraposée. Supposons que B est décidé par une machine $\mathcal{N}$. Soit m, un mot arbitraire dont nous voudrions décider l'appartenance à A. Partant de $\#m\underline{\#}$, l'application de $\mathcal{T}$ fournit le code d'une machine $\mathcal{M}' = \mathcal{RW}_m\mathcal{AM}$ telle que $L(\mathcal{M}) = L(\mathcal{M}')$ si et seulement si $m \in A$. Par conséquent, l'application de $\mathcal{N}$ à $\#\rho(\mathcal{M}')\underline{\#}$ fournit $\#u\underline{\#}$ si $m \in A$ et $\#\underline{\#}$ sinon. Autrement dit, A est décidé par $\mathcal{TN}$.

Schématiquement, on peut aussi l'expliquer comme suit :

$$\begin{array}{lll} \text{si } m \in A & L(\mathcal{M}) = L(\mathcal{RW}_m\mathcal{AM}) & \text{donc } \rho(\mathcal{RW}_m\mathcal{AM}) \in B \\ \text{si } m \notin A & L(\mathcal{RW}_m\mathcal{AM}) = \emptyset \neq L(\mathcal{M}) & \text{donc } \rho(\mathcal{RW}_m\mathcal{AM}) \notin B \end{array}$$

Or $\mathcal{N}$ décide de B donc $\mathcal{TN}$ décide de A, une absurdité.

Passons à la preuve de la première partie du théorème. On y applique un raisonnement identique. Nous utilisons en particulier les mêmes notations. Soit $C \subsetneq RE$ un ensemble non vide de langages acceptables. Supposons dans un premier temps que le langage vide n'appartient pas à C. Autrement dit,

C contient un langage L non vide accepté par une machine $\mathcal{M}$. Nous allons montrer que

On a la même situation que précédemment : un langage non vide accepté par une machine $\mathcal{M}$.

$$B_C = \{\rho(\mathcal{M}') \mid L(\mathcal{M}') \in C\}$$

est indécidable. En fait, on montre que *si A est un langage acceptable mais indécidable, alors B_C est indécidable* (et on pourra prendre pour A le langage associé au problème de l'arrêt). Pour ce faire, on considère une fois encore la contraposée. Supposons B_C décidé par une machine $\mathcal{N}$. Soit m un mot dont on voudrait décider l'appartenance à A. Partant de $\#m\underline{\#}$, l'application de $\mathcal{T}$ fournit le code d'une machine $\mathcal{M}' = \mathcal{RW}_m\mathcal{AM}$ telle que $L(\mathcal{M}') = L \in C$ si et seulement si $m \in A$. De plus, si $m \notin A$, alors $L(\mathcal{M}') = \emptyset \notin C$. Il ne reste plus qu'à appliquer $\mathcal{N}$ pour décider si m appartient ou non à A.

Schématiquement, on peut aussi l'expliquer comme suit :

si $m \in A$	$L(\mathcal{M}) = L(\mathcal{RW}_m\mathcal{AM}), L(\mathcal{M}) \in C$	donc $\rho(\mathcal{RW}_m\mathcal{AM}) \in B$
si $m \notin A$	$L(\mathcal{RW}_m\mathcal{AM}) = \emptyset, \emptyset \notin C$	donc $\rho(\mathcal{RW}_m\mathcal{AM}) \notin B$

Or $\mathcal{N}$ décide de B_C donc $\mathcal{TN}$ décide de A, une absurdité.

Si C est un ensemble propre de RE contenant le langage vide, alors on peut appliquer le raisonnement effectué ci-dessus à son complémentaire $RE \setminus C$. Ainsi, le langage $B_{RE\setminus C}$ est indécidable et il en va de même de son complémentaire[56] qui n'est autre que B_C.

Passons à la seconde partie et considérons un sous-ensemble non vide $C \subset R$ de fonctions calculables. Soient

$$B_C = \{\rho(\mathcal{M}') \mid \mathcal{M}' \text{ calcule un élément de } C\}$$

et f une fonction de C calculée par une machine de Turing $\mathcal{M}$. Comme d'habitude, pour montrer que B_C est indécidable, nous allons montrer que *si A est un langage acceptable mais indécidable, alors B_C est indécidable* en procédant par contraposition. On procède comme dans la première partie. On remarque simplement que si $m \in A$, les machines $\mathcal{M}$ et $\mathcal{M}' = \mathcal{RW}_m\mathcal{AM}$ ont exactement le même comportement, c'est-à-dire qu'elles calculent la même fonction $f \in C$. Par contre, si $m \notin A$, alors $\mathcal{M}'$ ne calcule pas de fonction et $\rho(\mathcal{M}') \notin B_C$.

car C est non vide.

■

Nous présentons maintenant un autre exemple classique de problème indécidable : déterminer s'il est possible, au moyen d'un ensemble fini de modèles de pavés et de règles d'assemblage, de paver le premier quadrant du plan. Bien que ce problème puisse paraître quelque peu artificiel, la technique utilisée ici se rencontre fréquemment dans diverses preuves plus élaborées[57]. Voici la formalisation de ce problème de décision. Une instance du problème est la donnée de

[56] Il est clair qu'un langage est indécidable *si et seulement si* son complémentaire est également indécidable

[57] On pense par exemple aux automates sur les arbres et aux langages réguliers d'arbres qui sont une généralisation naturelle des mots qui ont une structure purement linéaire.

- T un ensemble fini de *pavés* ou *tuiles*,
- $i \in T$, un *pavé initial*,
- $H \subset T \times T$, l'ensemble des règles de juxtapositions horizontales et
- $V \subset T \times T$, l'ensemble des règles de juxtapositions verticales.

La question posée est de déterminer si les modèles de pavés et les règles de construction proposés permettent ou non de paver le premier quadrant. De manière formelle, étant donné un problème de pavage (T, i, H, V), une *solution au problème* est une fonction $f : \mathbb{N}^2 \to T$ telle que

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

Exemple II.10.8. Soit le problème de pavage (T, i, H, V) où $T = \{1,2,3,4\}$, $i = 1$,

$$H = \{(1,1),(1,2),(2,1),(2,2),(2,3),(3,4)\}$$

et

$$V = \{(1,2),(2,1),(2,2),(3,2),(4,1)\}.$$

Une solution périodique est représentée à la figure II.30,

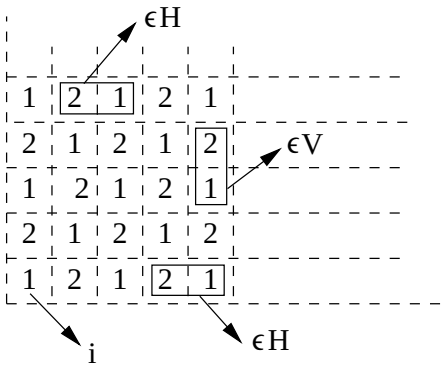


FIGURE II.30. Un pavage du premier quadrant.

$$f(i,j) = \begin{cases} 1 & \text{si } i+j \equiv 0 \mod 2 \\ 2 & \text{si } i+j \equiv 1 \mod 2. \end{cases}$$

Rappelons une fois encore que ce n'est pas parce qu'on peut déterminer une solution particulière pour des instances spécifiques du problème qu'en général (i.e., pour des instances arbitraires), on pourra en donner une solution algorithmique. Comme le montre le résultat suivant, il n'en est rien.

Théorème II.10.9. *Le problème de pavage défini précédemment est indécidable.*

Démonstration. L'idée de la démonstration est de montrer que si le problème de pavage était décidable, alors il en serait de même du problème de l'arrêt (en fait, sa variante A_2 considérée dans l'exemple II.10.4). C'est encore une "réduction".

A toute machine de Turing $\mathcal{M}$, on va associer un problème de pavage $P_{\mathcal{M}}$ de manière telle que le problème $P_{\mathcal{M}}$ admette une solution si et seulement si la machine $\mathcal{M}$ s'arrête à partir de $q_0.\underline{\#}$. Comme d'habitude, nous supposons que $\mathcal{M} = (Q, q_0, F, \Sigma, \Gamma, \delta)$ n'atteint pas de configuration pendante et que $F = \{h\}$.

Les pavés de $P_{\mathcal{M}}$ sont de cinq types et permettent de décrire exactement les transitions effectuées au sein de $\mathcal{M}$. Pour tout symbole $\gamma \in \Gamma$, on dispose d'un premier type de pavés qui seront représentés comme à la figure II.31.

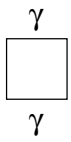


FIGURE II.31. Pavé de type 1.

Pour tous $p, q \in Q \setminus \{h\}$, $\gamma, \gamma' \in \Gamma$ tels que $\delta(p, \gamma) = (q, \gamma')$, on a des pavés représentés à la figure II.32.

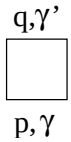


FIGURE II.32. Pavé de type 2.

Pour tous $p, q \in Q \setminus \{h\}$, $\gamma \in \Gamma$ tels que $\delta(p, \gamma) = (q, R)$, on a des pavés représentés à la figure II.33.

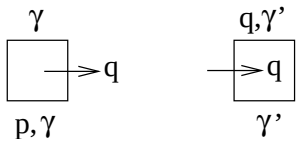


FIGURE II.33. Deux pavés complémentaires de type 3.

Pour tous $p, q \in Q \setminus \{h\}$, $\gamma \in \Gamma$ tels que $\delta(p, \gamma) = (q, L)$, on a des pavés représentés à la figure II.34. Enfin, on dispose de deux pavés représentés à la figure II.35. Le premier est le pavé initial.

Les règles de juxtaposition sont simples. On ne peut juxtaposer deux pavés que si leurs côtés contigus (gauche/droit, haut/bas) possèdent la même

FIGURE II.34. Deux pavés complémentaires de type 4.

FIGURE II.35. Le pavé initial et un pavé "blanc".

information. On pourra donc, par exemple, juxtaposer les pavés repris à la figure II.36. A droite de la figure, nous avons représenté un raccourci d'écriture que nous nous autoriserons à employer par la suite. Ainsi, en

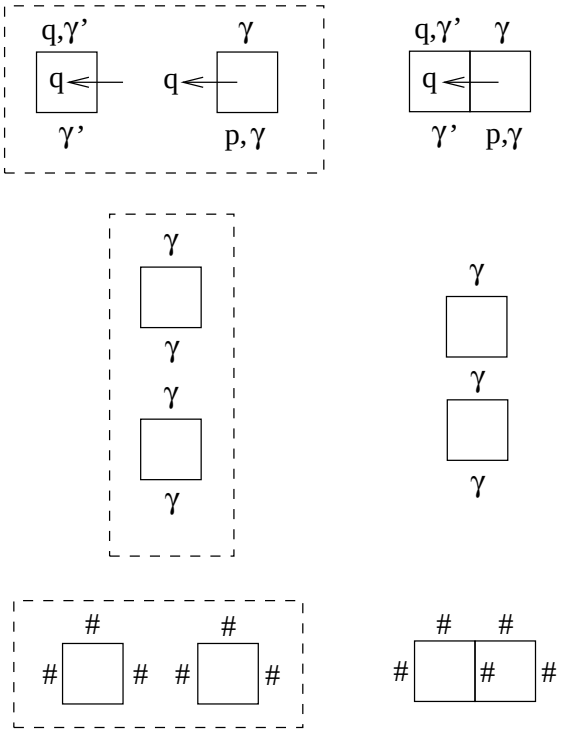


FIGURE II.36. Quelques juxtapositions admissibles.

suivant les directives imposées, la première ligne du pavage $P_{\mathcal{M}}$ est nécessairement de la forme donnée à la figure II.37.

Pour conclure, il suffit de remarquer que le bord supérieur d'une ligne du pavage décrit complètement une configuration machine de $\mathcal{M}$. En effet, elle contient l'information relative à l'état atteint, le contenu du ruban mémoire ainsi que le contenu de la cellule référencée (en effet, un seul pavé

FIGURE II.37. Première ligne du pavage.

par ligne contient une double information "p, γ" ce qui permet de repérer une cellule particulière). De plus, les pavés ont été définis de manière telle qu'une transition au sein de $\mathcal{M}$ corresponde exactement à ajouter une ligne dans le pavage. La première ligne du pavage correspond à la configuration initiale $q_0.\underline{\#}$ et la j-ième ligne correspond à la configuration atteinte après j transitions. Ainsi, si $\mathcal{M}$ s'arrête à partir de la configuration $q_0.\underline{\#}$, le problème de pavage $P_{\mathcal{M}}$ ne possède pas de solution et inversement. En effet, aucun pavé de $P_{\mathcal{M}}$ ne contient d'information correspondant à l'état d'arrêt h.

■

Exemple II.10.10. Considérons la machine de Turing simpliste donnée à la figure II.38. A cette machine $\mathcal{M}$ correspond un problème de pavage P_M

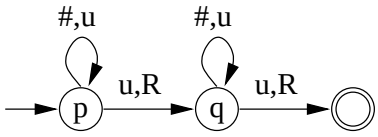


FIGURE II.38. Une machine de Turing $\mathcal{M}$.

et les 9 pavés repris à la figure II.39.

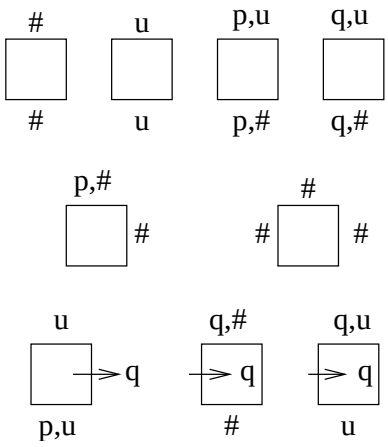


FIGURE II.39. Les pavés de $P_{\mathcal{M}}$ associés à $\mathcal{M}$.

Partant de la configuration $p.\underline{\#}$, on a la suite de configurations

$$p.\underline{\#} \vdash p.\underline{u} \vdash q.u\underline{\#} \vdash q.u\underline{u}$$

qui mène à l'état accepteur. Comme le montre la figure II.40, le problème de pavage $P_{\mathcal{M}}$ n'a pas de solution car aucun pavé ne fait référence à l'état d'arrêt.

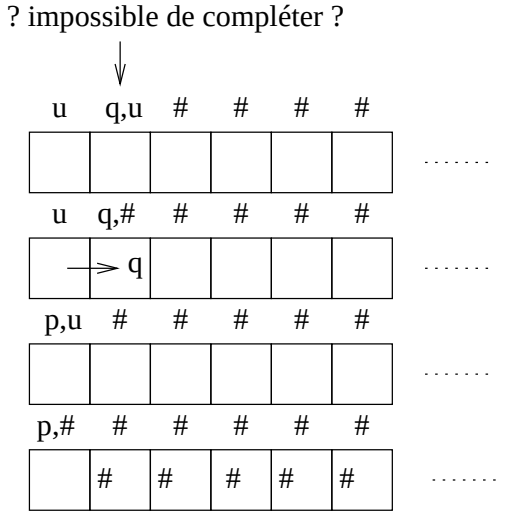


FIGURE II.40. Le pavage P_M.

CHAPITRE III

Complexité

A la fin du chapitre précédent, nous nous sommes intéressés à des problèmes de décision et nous avons montré que certains d'entre eux étaient *indécidables* (comme le problème de l'arrêt et ses variantes ou encore le problème de pavage).

Lorsqu'un problème est par contre *décidable* (par exemple, tester si un nombre est premier, le problème du voyage de commerce, ...), on dispose dès lors d'un algorithme pour résoudre ce problème de décision et la question naturelle qui vient à l'esprit est d'estimer les ressources nécessaires à mettre en oeuvre pour en obtenir la solution. Par ressource, on entend non seulement la durée d'exécution de l'algorithme et on parle alors de *complexité temporelle* mais aussi l'espace mémoire requis et on parle alors de *complexité spatiale*.

Nous nous intéressons ici principalement à la complexité temporelle. Pour la mesurer, nous estimerons le nombre de transitions effectuées par une machine de Turing. Ce choix est raisonnable puisqu'il traduit le nombre de manipulations élémentaires de la mémoire qu'il convient de réaliser et on peut supposer que de telles opérations s'effectuent en temps constant. Ainsi, nous verrons que la complexité (temporelle) d'un algorithme peut être vue comme une fonction qui à $n \in \mathbb{N}$ associe le temps maximal nécessaire pour traiter des données de taille n. Généralement, on considère comme limite acceptable[1], les algorithmes ayant une complexité polynomiale (i.e., bornée par un polynôme). A l'opposé, les algorithmes ayant une complexité exponentielle sont considérés comme impraticables. Imaginez deux algorithmes, l'un ayant une complexité de la forme n^2 et l'autre de la forme 2^n. Pour fixer les idées, on suppose que réaliser un calcul sur une donnée de taille n nécessite respectivement n^2 et 2^n secondes. Sur une donnée de taille 50, le premier s'exécuterait, dans le pire cas, en un peu plus de 40 minutes alors que le second nécessiterait plus de 35 millions d'années !

Bien évidemment, plusieurs algorithmes différents peuvent résoudre un même problème et il n'est pas aisé de démontrer qu'un algorithme donné a la "meilleure" complexité. Par conséquent, ce n'est pas parce qu'on dispose uniquement d'une solution exponentielle qu'il n'existe pas de solution polynomiale. En pratique, on essayera souvent de montrer qu'un problème est

[1]En pratique, on considère comme praticables des algorithmes dont la complexité ne dépasse pas n^4.

"difficile" (i.e., non soluble de manière efficace) s'il n'existe pas[2] d'algorithme polynomial pour le résoudre. Pour un problème de ce type, il faudra alors, suivant les cas, recourir à des heuristiques, à des approximations ou encore à des spécifications plus restrictives des instances.

1. Premières définitions

Bien évidemment, pour parler d'algorithme, nous allons conserver ici notre formalisme donné par les machines de Turing. Il ne serait d'ailleurs pas raisonnable de s'intéresser à une architecture informatique particulière. En effet, un même programme peut avoir des temps d'exécution différents sur deux machines différentes (un Pentium 50 Mhz, un Pentium IV 3 Ghz ou encore un MacIntosh G4, ...) sans parler du compilateur employé, de la version du système d'exploitation, etc.... En effet, de telles mesures de performance n'auraient aucun sens et ne reflèteraient pas les qualités de l'algorithme.

Commençons par quelques définitions. Pour des raisons qui seront claires par la suite, nous utiliserons dans ce chapitre tantôt des machines de Turing déterministes, tantôt des machines non déterministes. (Dans ce dernier cas, rappelons qu'une même configuration machine initiale peut donner lieu à plusieurs suites de transitions).

Définition III.1.1. Soient $\mathcal{M}$ une machine de Turing (déterministe ou non) et m un mot. La *durée d'exécution* de $\mathcal{M}$ sur m, notée $d_{\mathcal{M}}(m)$ est

- 0 si m n'est pas accepté par $\mathcal{M}$,
- le nombre minimum de transitions permettant d'atteindre un état accepteur depuis la configuration $q_0.\#m\underline{\#}$.

Compter le nombre de transitions est tout à fait raisonnable pour estimer le temps de calcul d'un algorithme puisqu'il s'agit du nombre d'opérations (écriture et déplacement) que l'on va effectuer sur le ruban mémoire.

Définition III.1.2. La *fonction de complexité* d'une machine $\mathcal{M}$ (déterministe ou non) est la fonction $T_{\mathcal{M}} : \mathbb{N} \to \mathbb{N}$ définie par

$$T_{\mathcal{M}}(n) = \sup\{d_{\mathcal{M}}(m) \mid |m| = n\}.$$

Il s'agit d'une étude de la complexité temporelle dans le *pire cas* (worst case analysis). En effet, on regarde parmi toutes les données de taille n, le mot donnant lieu à l'exécution la plus longue. Il s'agit donc d'une mesure particulière du comportement de $\mathcal{M}$ par rapport aux données qu'on lui fournit.

Remarque III.1.3. Nous n'aborderons pas ce thème dans ce qui suit. Mais sachez qu'il existe également une analyse plus fine de la complexité pour laquelle on s'intéresse alors à la *complexité moyenne* (average case

[2] A la condition que $P \neq NP$.

analysis). Dans ce dernier cas[3], il faut disposer d'un plus grand nombre d'informations, en particulier, la distribution des données en fonction de leur durée d'exécution.

On pourrait par exemple imaginer que dans la majorité des cas, un certain algorithme se comporte en un temps polynomial mais que pour quelques rares exceptions, il se comporte de manière exponentielle. Notre analyse dans le pire cas classerait par conséquent cet algorithme comme impraticable, alors que statistiquement, il pourrait presque toujours fournir la solution en un temps raisonnable. La complexité en moyenne est étudiée de manière intensive dans ce qu'on appelle aujourd'hui l'analyse d'algorithmes dont les fondements peuvent être attribués à D. E. Knuth. La détermination de la distribution des données est une question souvent très difficile qui utilise de manière intensive de nombreuses branches des mathématiques (combinatoire, analyse asymptotique, ...).

Les deux approches "en moyenne" et "dans le pire cas" sont donc deux approches complémentaires d'un même problème.

Voici quelques rappels de notations.

Définition III.1.4. On note $f \in \mathcal{O}(g)$ (ou parfois $f = \mathcal{O}(g)$ et l'on prononce "grand O de g") s'il existe un entier[4] N et une constante C telle que

$$\forall n \geq N, f(n) \leq Cg(n).$$

Exemple III.1.5. Par exemple, les fonctions $\ln x$ et $x|\sin x|$ sont en $\mathcal{O}(n)$.

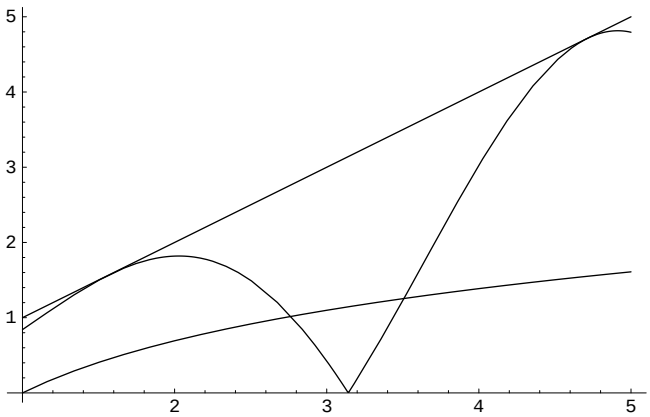


FIGURE III.1. x, $\ln x$, $x|\sin x|$.

[3]Voir par exemple, P. Flajolet, R. Sedgewick, *An introduction to the analysis of algorithms*, Addison-Wesley, (1996).

[4]On pourrait considérer un nombre réel N et pas un entier, mais les fonctions que nous rencontrerons par la suite seront toutes définies sur $\mathbb{N}$.

Remarque III.1.6. On trouve aussi d'autres notations qui peuvent s'avérer utiles pour décrire de manière plus fine la complexité d'un algorithme. Ainsi, $f \in \Omega(g)$ si et seulement si $g \in \mathcal{O}(f)$, ce qui signifie donc que $f(n)$ est minoré pour n suffisamment grand par $C\,g(n)$. De plus, $f \in \Theta(g)$ si $f \in \Omega(g) \cap \mathcal{O}(g)$. Enfin, $f = o(g)$ si $f/g \to 0$ et $f \sim g$ si $f/g \to 1$. Par exemple, $x^2 + \sin 6x$ appartient à $\Theta(x^2)$, comme illustré à la figure III.2.

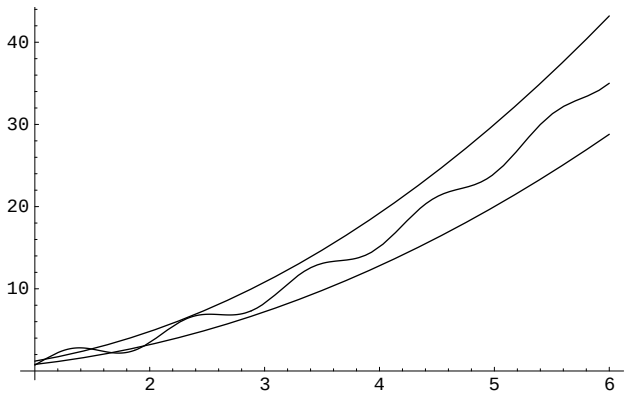


FIGURE III.2. $x^2 + \sin 6x$, $\frac{4}{5}x^2$ et $\frac{6}{5}x^2$.

Définition III.1.7. Une machine de Turing $\mathcal{M}$ (déterministe ou non) est qualifiée de *polynomiale* s'il existe $k \in \mathbb{N}$ tel que

$$T_{\mathcal{M}}(n) \in \mathcal{O}(n^k),$$

ou de manière équivalente[5], s'il existe un polynôme Q tel que

$$\forall n \in \mathbb{N} : T_{\mathcal{M}}(n) \leq Q(n),$$

cette seconde forme étant parfois plus aisée à manipuler. Autrement dit, une machine de Turing est polynomiale si sa fonction de complexité est majorée par un polynôme. En particulier, une fonction est *calculable en un temps polynomial* ou *P-calculable* s'il existe une machine de Turing (déterministe[6]) polynomiale qui la calcule.

Exemple III.1.8. Reprenons la machine de Turing considérée à la figure II.4 et recopiée par facilité à la figure III.3. Nous allons montrer que cette machine est polynomiale. Remarquons[7] tout d'abord que pour effectuer la transition

$$0.\#u^a\underline{\#}u^b \vdash^* 0.\#u^{a-1}\underline{\#}u^{b+2}, \quad a \geq 1, b \geq 0,$$

$2b + 8$ transitions élémentaires sont nécessaires. Ainsi, pour obtenir

$$0.\#u^k\underline{\#} \vdash^* 0.\#u^{k-1}\underline{\#}u^2 \vdash^* 0.\#u^{k-2}\underline{\#}u^4 \vdash^* \cdots \vdash^* 0.\#u\underline{\#}u^{2(k-1)} \vdash^* 0.\#\underline{\#}u^{2k},$$

[5] Démontrer cette équivalence

[6] Une machine de Turing non déterministe ne calcule pas de fonction, pensez à l'exemple II.6.6.

[7] Il suffit de passer en revue l'ensemble des transitions effectuées.

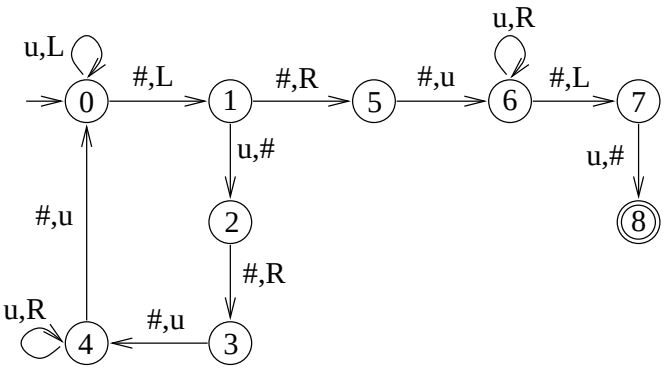


FIGURE III.3. Une machine de Turing calculant $n \mapsto 2n$.

le nombre de transitions nécessaires est

$$\sum_{i=0}^{k-1}(2.2i+8) = 4\frac{k\,(k-1)}{2} + 8k.$$

Enfin, pour passer de $0.\#\underline{\#}u^{2k}$ à $8.\#u^{2k}\underline{\#}$, il est aisé de vérifier que $2k+6$ transitions élémentaires sont nécessaires. Par conséquent, la complexité de la machine est donnée par

$$2k^2 + 8k + 6$$

et en particulier la fonction $n \mapsto 2n$ est P-calculable.

2. Transformations polynomiales

Si un problème est décidable mais qu'on ne connaît aucun algorithme polynomial pour le résoudre, cela ne signifie pas qu'un tel algorithme n'existe pas. Cependant, nous aimerions pouvoir déterminer si un tel algorithme existe ou non. Pour ce faire, nous voudrions ici pouvoir "classer" les problèmes suivant leur complexité temporelle et formaliser d'une certaine manière le fait "qu'un problème est plus simple qu'un autre". Nous introduisons la notion de transformation polynomiale qui permet de définir rigoureusement ces concepts.

Rappelez-vous qu'il est équivalent de considérer un problème de décision ou un langage codant ses instances positives. C'est pour cette raison que la définition suivante s'exprime en termes de langages et non de problèmes.

Définition III.2.1. Soient $L_i \subset \Sigma_i^*$, $i = 1, 2$, deux langages. Une fonction $f : \Sigma_1^* \to \Sigma_2^*$ est une *transformation polynomiale* de L_1 vers L_2 si

- f est P-calculable et
- $f(x) \in L_2 \Leftrightarrow x \in L_1$.

S'il existe une transformation polynomiale de L_1 vers L_2, alors on notera $L_1 \leq_P L_2$ ou plus simplement $L_1 \leq L_2$. (Certains auteurs préfèrent la notation $L_1 \propto L_2$.)

En quelque sorte, si $L_1 \leq L_2$, L_1 est algorithmiquement plus "simple" que L_2 car tout algorithme permettant de décider l'appartenance à L_2 peut être transformé en un algorithme testant l'appartenance à L_1. On parlera donc encore dans ce contexte de *réduction* et on dira parfois que L_1 est "*polynomialement réductible à*" L_2.

Remarque III.2.2. La relation $\leq$ est réflexive et transitive. Les vérifications sont immédiates. Soient $L_i \subset \Sigma_i^*$, $i = 1, 2, 3$ tels que $L_1 \leq L_2$ et $L_2 \leq L_3$. Il existe donc des transformations polynomiales $g : \Sigma_1^* \to \Sigma_2^*$ et $h : \Sigma_2^* \to \Sigma_3^*$ de L_1 vers L_2 et de L_2 vers L_3 respectivement. Il est clair que $f = h \circ g : \Sigma_1^* \to \Sigma_3^*$ est une transformation polynomiale de L_1 vers L_3.

Remarque III.2.3. Remarquons d'ores et déjà que si L_2 correspond à un problème soluble en un temps polynomial et si $L_1 \leq L_2$, alors L_1 correspond lui aussi à un problème soluble en un temps polynomial. En effet, si on considère une instance $x \in \Sigma_1^*$ de taille n, on peut la transformer en une instance $f(x)$ de L_2 en un temps proportionnel à $Q(n)$ pour un certain polynôme Q. En particulier, la taille de $f(x)$ ne saurait dépasser[8] l'ordre de $Q(n)$. Enfin, puisque L_2 est soluble en un temps polynomial, il existe un polynôme Q' tel qu'on sait décider si une instance de L_2 est positive en un temps borné par $Q'(\ell)$ où ℓ est la taille de l'instance. Ainsi, on sait décider si $f(x)$ est une instance positive de L_2 (et donc si x est une instance positive de L_1) en un temps borné par $Q'(Q(n))$. On conclut car la composée de polynômes est encore un polynôme.

Dans l'exemple suivant, on considère une transformation polynomiale entre deux problèmes classiques : le problème du circuit hamiltonien (HC) et une variante du problème du voyageur de commerce introduit dans l'exemple II.9.4.

Exemple III.2.4. Considérons le problème du voyageur de commerce introduit dans l'exemple II.9.4 mais avec la condition supplémentaire qu'il est interdit de passer plus d'une fois par une même ville. On suppose, de plus, qu'un coût de transport $c_{ij} > 0$ existe toujours entre deux villes distinctes $v_i \neq v_j$. Ainsi, si les villes sont $v_1, \ldots, v_k$ et si la borne à ne pas dépasser est ℓ, une solution au problème est une permutation ν de $\{1, \ldots, k\}$ telle que

$$\sum_{i=1}^{k-1} c_{\nu_i \nu_{i+1}} + c_{\nu_k \nu_1} \leq \ell.$$

On note TS' ce problème.

Nous allons à présent introduire le problème du circuit hamiltonien (Hamiltonian Circuit HC) et montrer que $HC \leq TS'$. Une instance de HC est un graphe $G = (V, E)$ où $V = \{v_1, \ldots, v_k\}$. Un *circuit hamiltonien* est un cycle $(u_1, u_2, \ldots, u_{k+1})$ passant une et une seule fois par chaque sommet du

[8] En effet, si le nombre d'opérations est proportionnel à $Q(n)$, alors c'est en particulier le cas du nombre d'écritures et de déplacements sur le ruban.

graphe, i.e., $u_1 = u_{k+1}$, $\forall i, j \in \{1, \ldots, k\}$, $(u_i, u_{i+1}) \in E$ et $u_i \neq u_j$, si $i \neq j$. La question qui est posée est donc de déterminer si le graphe $G = (V, E)$

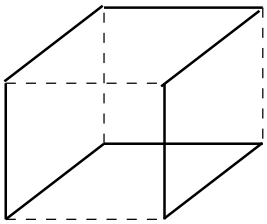


FIGURE III.4. Un circuit hamiltonien dans un cube.

possède ou non un circuit hamiltonien. Autrement dit, s'il existe une permutation ν de $\{1, \ldots, n\}$ telle que si $V = \{1, \ldots, n\}$, alors $(\nu_i, \nu_{i+1}) \in E$ pour tout $i < n$ et $(\nu_n, \nu_1) \in E$.

Nous allons montrer qu'il existe une transformation polynomiale permettant d'associer à toute instance x de HC (i.e., un graphe) une instance $f(x)$ de TS' (i.e., un graphe, une matrice de coûts et une borne) de manière telle que $f(x)$ admette une solution si et seulement si x en admet une.

Une instance de HC est codée par un couple (n, A) où n représente le nombre de sommets du graphe et A sa matrice d'incidence. Une instance de TS' est codée par un triplet (n, C, ℓ) où n représente le nombre de villes, C la matrice des coûts et ℓ la borne. On considère la fonction

$$f : (n, A) \mapsto (n, D^A, n)$$

où

$$D^A_{ij} = \begin{cases} 1 & \text{si } A_{ij} = 1, \\ 2 & \text{si } A_{ij} = 0. \end{cases}$$

Si l'instance $f(n, A)$ de TS' est positive, il faut passer par les n villes avec un coût total ne dépassant pas n. Par conséquent, on doit uniquement emprunter des arêtes de coût 1. Par définition de la matrice D^A, on dispose alors d'une solution au problème initial (n, A) de HC. Inversement, si (n, A) est une instance positive de HC, elle correspond à une instance positive $f(n, A) = (n, D^A, n)$ de TS'.

3. Les classes P, NP et NPC

On définit une relation $\equiv_P$ (simplement notée $\equiv$, si le contexte est clair) sur l'ensemble des langages, par

$$L_1 \equiv L_2 \Leftrightarrow L_1 \leq L_2 \text{ et } L_2 \leq L_1.$$

Remarque III.3.1. Il est facile de vérifier que la relation $\equiv$ ainsi définie est une relation d'équivalence.

Définition III.3.2. La classe de complexité P est formée des langages décidés par une machine de Turing (déterministe) polynomiale.

Remarque III.3.3. Les considérations de la remarque III.2.3 se réexpriment comme suit. Soient deux langages L_1 et L_2 tels que $L_1 \leq L_2$.

- Si $L_2 \in P$, alors $L_1 \in P$.
- Si $L_1 \notin P$, alors $L_2 \notin P$

Les problèmes pour lesquels on connaît l'appartenance à P sont jugés solubles de manière efficace[9]. En anglais, on leur consacre le vocable "tractable". Par contre, si un problème n'appartient pas à P, il sera jugé impraticable et on rencontre le vocable anglais "intractable".

Définition III.3.4. La classe de complexité NP est formée des langages acceptés par une machine de Turing non déterministe polynomiale.

Remarque III.3.5. Il est erroné de parler, comme en l'entend parfois, de langage "non polynomial" pour "NP" ! De plus, si un langage L appartient à NP et si on dispose d'un mot m appartenant à L, alors la vérification que $m \in L$ peut se faire en un temps polynomial (on parle parfois de *certification succincte*).

Par exemple, si on considère le problème "l'entier n est-il un nombre composé ?", alors si on dispose de deux facteurs x et y (obtenus "miraculeusement" de manière non déterministe), vérifier que n est composé est aisé : il suffit de réaliser le produit de x et de y (en temps polynomial) et de vérifier que le résultat vaut bien n.

Remarque III.3.6. Il est clair[10] que

$$P \subset NP.$$

Cependant, il s'agit d'une question ouverte de savoir si $P = NP$. Un prix d'un million de dollars offert par le "Clay Mathematics Institute" est offert[11] à quiconque pourrait démontrer l'égalité ou l'inclusion stricte entre P et NP. Néanmoins, pour beaucoup, il est généralement admis que

$$''P \neq NP''.$$

Voici deux classes d'équivalence pour la relation $\equiv_P$.

Exemple III.3.7. L'ensemble

$$\{\Sigma^* \mid \Sigma \text{ alphabet fini}\}$$

est une classe d'équivalence pour $\equiv$. En effet, tout d'abord, si Σ et Γ sont deux alphabets finis, alors la fonction $f : \Sigma^* \to \Gamma^* : w \mapsto \varepsilon$ est une transformation polynomiale de Σ^* vers Γ^*. Il est clair qu'elle est calculable en un temps polynomial et bien évidemment, $w \in \Sigma^* \Leftrightarrow f(w) = \varepsilon \in \Gamma^*$ (il n'y a aucune véritable condition à satisfaire).

[9] Avec les précautions d'usage sur le degré du polynôme majorant la fonction de complexité.

[10] Toute machine déterministe est un cas particulier de machine non déterministe et décider est une propriété plus forte qu'accepter.

[11] `http://www.claymath.org/index.php` : "P versus NP challenge"

Ensuite, il faut encore montrer que si $L \subset \Sigma^*$ et si $L \equiv \Gamma^*$, alors $L = \Sigma^*$. En particulier, $L \leq \Gamma^*$, il existe donc une transformation polynomiale $f : \Sigma^* \to \Gamma^*$ telle que $w \in L \Leftrightarrow f(w) \in \Gamma^*$. On en conclut que w appartient à L pour tout mot $w \in \Sigma^*$, ce qui suffit.

Exemple III.3.8. L'ensemble des langages[12] dans P, écrits sur un alphabet[13] quelconque Σ, mais différents de Σ^*,

$$P' = \{L \subsetneq \Sigma^* \mid L \in P\} = P \setminus \{\Sigma^* \mid \Sigma \text{ alphabet fini}\},$$

est aussi une classe d'équivalence pour $\equiv$. Montrons tout d'abord que si $M \in P'$ et si $L \equiv M$, alors $L \in P'$. Tout d'abord, il est clair que $L \in P$ car $L \leq M$ et $M \in P$. Si $L \subseteq \Sigma^*$ et $M \subsetneq \Gamma^*$, il nous faut encore vérifier que $L \neq \Sigma^*$. Nous savons que $M \leq L$. Dès lors, si $L = \Sigma^*$, on en conclurait, comme dans l'exemple précédent que $M = \Gamma^*$ ce qui est impossible.

Il nous reste à vérifier que si $L \subsetneq \Sigma^*$ et $M \subsetneq \Gamma^*$ appartiennent à P', alors $L \equiv M$. Soient $m \in M$ et $x \in \Gamma^* \setminus M$ (ce qui est toujours possible car $M \subsetneq \Gamma^*$). La fonction

$$f : \Sigma^* \to \Gamma^* : w \mapsto \begin{cases} x & \text{si } w \notin L, \\ m & \text{si } w \in L, \end{cases}$$

est telle que $w \in L \Leftrightarrow f(w) \in M$. Pour avoir $L \leq M$, il suffit de se convaincre que f est polynomiale. Par symétrie, on a aussi $M \leq L$ et donc $L \equiv M$. Puisque $L \in P$, il existe une machine de Turing déterministe polynomiale $\mathcal{M}$ qui décide L et qui fournit donc pour toute entrée $w \in \Sigma^*$, une sortie de la forme $\#\underline{\#}$ ou $\#u\underline{\#}$ (suivant que $w \notin L$ ou $w \in L$). Il suffit de composer cette machine avec une machine "de conversion"[14], remplaçant la première sortie par $\#x\underline{\#}$ et la seconde par $\#m\underline{\#}$. Il est clair que la machine composée ainsi obtenue est encore polynomiale et calcule f.

Définition III.3.9. La classe de complexité NPC est un sous-ensemble de NP défini comme suit,

$$NPC = \{L \in NP \mid \forall L' \in NP,\ L' \leq L\}.$$

Si L appartient à NPC, on parle de langage NP-*complet*. Il est clair que NPC est une classe d'équivalence pour $\equiv$. Au vu de cette définition, on peut dire en quelque sorte que les problèmes NP-complets sont "les plus difficiles" du point de vue algorithmique.

Remarque III.3.10. Il est clair que $P = NP$ si et seulement si il existe un langage NP-complet appartenant à P. Cela résulte de la remarque III.3.3 et de la définition même de NPC. Ainsi, si nous supposons $P \neq NP$, alors $NPC \cap P = \emptyset$ et on a le schéma repris à la figure III.5.

[12] supposés non vides

[13] Chaque langage possède son propre alphabet. Ce dernier peut différer d'un langage à l'autre.

[14] La construction d'une telle machine est laissée au lecteur.

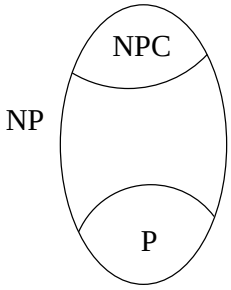


FIGURE III.5. $P \subset NP$ et $NPC \subset NP$.

Dans la littérature, on rencontre parfois la terminologie "NP-dur", sa signification étant la suivante.

Définition III.3.11. Un langage L est NP-*dur* ou NP-*difficile* (NP-hard language) si

$$\forall L' \in NP,\ L' \leq L.$$

Ainsi, pour démontrer qu'un langage L est NP-complet, on montre en général qu'il appartient à NP et qu'il est NP-dur. (Suivant les problèmes, c'est l'une ou l'autre partie qui est facile à démontrer.) Enfin, pour montrer que L est NP-dur, il suffit en général de procéder par réduction, i.e., trouver un autre langage L' NP-dur (ou NP-complet) tel que $L' \leq L$.

Si un langage est décidé par une machine de Turing (déterministe) polynomiale, alors son complémentaire l'est aussi[15]. Par contre, si un langage appartient à NP, il est accepté par une machine de Turing non déterministe polynomiale et rien ne permet d'affirmer qu'il en soit de même pour son complémentaire[16] (le problème de décision correspondant est obtenu en niant la question du problème initial[17]). On dira qu'un langage appartient à co-NP (resp. à co-NPC) si son complémentaire appartient à NP (resp. à NPC), i.e.,

$$\text{co-}NP = \{L \subset \Sigma^* \mid \Sigma^* \setminus L \in NP\}$$

et

$$\text{co-}NPC = \{L \subset \Sigma^* \mid \Sigma^* \setminus L \in NPC\}.$$

Remarque III.3.12. Tout comme il est généralement admis que $P \neq NP$, il est supposé que

$$''NP \neq \text{co-}NP''.$$

De plus, tout comme à la remarque III.3.5, si un langage L appartient à co-NP et si on dispose d'un mot m n'appartenant à L, alors la vérification que $m \notin L$ peut se faire en un temps polynomial (on parle parfois de *discrédit succinct*). Par exemple, le complémentaire de l'ensemble des

[15] Il suffit d'inverser les sorties $\#u\#$ et $\#\#$.

[16] L'ensemble des langages acceptables n'est pas stable pour le passage au complémentaire. Il n'est donc *a priori* pas clair que NP soit égal à co-NP.

[17] La question "l'entier n est-il premier ?" devient "l'entier n est-il composé ?"

nombres premiers est formé des nombres composés pour lesquels on dispose d'une certification succinte, cf. remarque III.3.5. Ainsi, `Primes` appartient à co-NP.

Proposition III.3.13. *S'il existe un langage appartenant à NP dont le complémentaire est NP-complet (ou encore, s'il existe un langage NP-complet dont le complémentaire est dans NP), alors $NP =$ co-NP.*

Démonstration. Remarquons tout d'abord que $L \in \text{co-}NP \cap NPC$ si et seulement si $\Sigma^* \setminus L \in NP \cap \text{co-}NPC$. Il suffit donc de démontrer l'une des deux situations évoquées ci-dessus.

Soit L un langage appartenant à NP et à $\text{co} - NPC$. En particulier, L est accepté par une machine de Turing non déterministe polynomiale $\mathcal{M}$. Nous allons montrer que co-$NP \subset NP$. (On obtient l'autre inclusion par symétrie en passant au complémentaire[18].) Soit $L' \in \text{co-}NP$. Puisque $L \in \text{co}-NPC$, il existe une transformation polynomiale f telle[19] que $L' \leq L$. Pour se convaincre que $L' \in NP$, on remarque que, pour accepter les mots de L' en un temps polynomial, il suffit de transformer tout mot d'entrée x en un temps polynomial au moyen de f puis de fournir $f(x)$ à $\mathcal{M}$ qui travaille également en temps polynomial.

■

Corollaire III.3.14. *Soit L un langage appartenant à $NP \cap$ co-NP. Si $NP \neq$ co-NP, alors ce langage ne peut être NP-complet (pas plus que co-NP-complet).*

Démonstration. C'est immédiat, si un langage appartient à NPC et co-NP, alors par le résultat précédent, on en tire que $NP =$ co-NP.

■

En supposant que $P \neq NP$ et $NP \neq$ co-NP, on a la représentation schématique donnée à la figure III.6 ($P \cap NPC = \emptyset$, $P \cap \text{co-}NPC = \emptyset$, $NPC \cap \text{co-}NP = \emptyset$ et co-$NPC \cap NP = \emptyset$). Notez que nous n'avons donné aucune indication sur les relations liant $NP \cap \text{co-}NP$ et P.

Si un langage est dans NP (i.e., accepté par une machine de Turing non déterministe polynomiale), il est aussi décidé par une machine de Turing déterministe, mais on ne peut garantir une exécution en un temps polynomial. On dispose en fait du résultat plus précis suivant.

Théorème III.3.15. *Si L est un langage appartenant à NP, alors il existe une machine de Turing (déterministe) qui décide L avec une complexité bornée par 2^Q où Q est un polynôme.*

[18] Si $L' \in NP$, alors $\Sigma^* \setminus L' = M \in \text{co-}NP$. Si l'on démontre que co-$NP \subset NP$, alors on aura aussi $M \in NP$ et donc $\Sigma^* \setminus M = L' \in \text{co-}NP$. Ceci montre que $NP \subset \text{co-}NP$ et il suffit donc de démontrer une seule inclusion.

[19] *stricto sensu*, $\Sigma^* \setminus L' \leq \Sigma^* \setminus L$ et il existe une transformation polynomiale f telle que $x \in \Sigma^* \setminus L' \Leftrightarrow f(x) \in \Sigma^* \setminus L$. Mais bien évidemment, on aussi $x \in L' \Leftrightarrow f(x) \in L$ et donc bien $L' \leq L$.

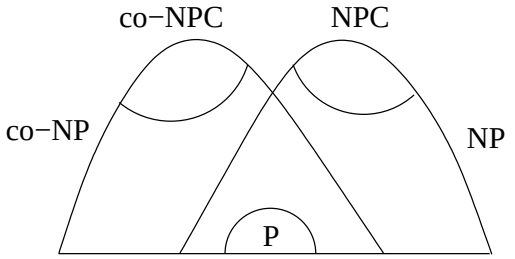


FIGURE III.6. Les différentes classes.

Démonstration. Soit $\mathcal{M}$ une machine de Turing non déterministe acceptant $L \in NP$ et telle que $T_{\mathcal{M}}(n) \leq Q(n)$, avec Q un polynôme. Comme dans la preuve de la proposition II.6.9, on construit une machine déterministe $\mathcal{M}'$ qui décide L. Il suffit de construire un arbre décrivant toutes les exécutions possibles de $\mathcal{M}$ à partir d'une configuration initiale $q_0.\#w\underline{\#}$ donnée. Pour rappel, le j-ième niveau de l'arbre contient exactement les configurations accessibles par $\mathcal{M}$ en j étapes. Il est clair qu'il est inutile, pour un mot w dont on voudrait décider l'appartenance à L, d'aller explorer un niveau de l'arbre supérieur à $Q(|w|)$. En effet, par hypothèse, $T_{\mathcal{M}}(n) \leq Q(n)$ et donc $\mathcal{M}$ accepte w en au plus $Q(|w|)$ transitions. Il suffit dès lors de concevoir $\mathcal{M}'$ de manière telle qu'elle s'arrête après au plus $Q(|w|)$ transitions. Pour conclure, il est clair que la simulation d'une transition de $\mathcal{M}$ par $\mathcal{M}'$ s'effectue en un temps borné par une constante a (il s'agit de parcourir un tableau fini codant la table de transition fixée une fois pour toutes). De plus, un noeud de l'arbre a un nombre de fils borné par une constante[20] r (ainsi, le niveau j de l'arbre contient au maximum r^j noeuds). Ainsi, la complexité temporelle de $\mathcal{M}'$ est bornée par

$$a(r + r^2 + r^3 + \cdots + r^{Q(|w|)}) = a\frac{r^{Q(|w|)+1} - r}{r - 1}.$$

D'où le résultat annoncé[21].

■

4. Le théorème de Cook

Nous avons établi une première classification de la complexité temporelle, mais existe-t-il un problème NP-complet ? Stephen Cook fut le premier à répondre à cette question par l'affirmative. En 1971, il montra que le problème SAT présenté dans l'exemple II.9.6 est NP-complet.

Théorème III.4.1. *SAT est NP-complet.*

Démonstration. **(1)** Il est clair que SAT appartient à NP. Il est facile de concevoir une machine de Turing non déterministe polynomiale qui génère une distribution des valeurs de vérité des variables puis qui vérifie que cette

[20] Si $r = 1$, cela signifierait que la machine est déterministe.
[21] Trivialement, $r^Q = 2^{Q \log_2 r}$ et $\alpha r^Q = 2^{Q \log_2 r + \log_2 \alpha}$.

distribution satisfait la formule donnée. Autrement dit, au vu de la remarque III.3.5, on dispose d'un certificat succinct : on peut vérifier en temps polynomial qu'une distribution proposée satisfait la formule donnée.

(2) Pour montrer que SAT appartient à NPC, nous devons montrer que pour tout langage $L \subseteq \Sigma^*$ appartenant à NP, on a $L \leq SAT$. Soit $\mathcal{M} = (Q, q_0, \Sigma, \Gamma, F, \delta)$, une machine de Turing non déterministe qui accepte L et dont la complexité est bornée par un polynôme T. Ainsi, lorsque $\mathcal{M}$ accepte un mot w, il existe une suite de configurations pour laquelle $\mathcal{M}$ subit au plus $T(|w|)$ transitions et passe donc par $T(|w|) + 1$ configurations (de $q_0.\#w\underline{\#}$ à une configuration d'acceptation). Puisque $\mathcal{M}$ est non déterministe, de chaque configuration, $\mathcal{M}$ peut subir au plus r transitions différentes (pour une constante r dépendant uniquement de $\mathcal{M}$).

On désire construire une réduction polynomiale f de L vers SAT. Il nous faut donc transformer toute instance x de L en une instance de SAT (i.e., en une formule mise sous forme normale conjonctive) de manière telle que $x \in L \Leftrightarrow f(x) \in SAT$. Notre but va être de "*décrire une formule qui simule le comportement de $\mathcal{M}$ sur un chemin d'acceptation*" de manière telle que x soit accepté par $\mathcal{M}$ si et seulement si il existe une distribution des valeurs de vérité qui rende la formule construite vraie. Il faudra de plus veiller à ce que la réalisation de cette formule puisse se faire en un temps polynomial.

Remarque III.4.2. La machine $\mathcal{M}$ accepte le mot w en au plus $T(|w|)$ transitions. Dans le pire cas, i.e., chaque transition est un déplacement vers la droite, le nombre de cellules du ruban utilisées est $|w| + 2 + T(|w|)$. En effet, on initialise le ruban avec $\#w\underline{\#}$ et on utilise donc $|w| + 2$ cases puis, chaque déplacement vers la droite consomme encore une case supplémentaire. Quitte à remplacer le polynôme $T(n)$ par le polynôme $T(n) + n + 2$, nous supposerons dans la suite que $\mathcal{M}$ accepte un mot w en au plus $T(|w|)$ transitions et en utilisant au plus $T(|w|)$ cases mémoire.

Les variables utilisées sont de plusieurs types. Nous allons de plus préciser les conditions qu'elles doivent satisfaire pour représenter un fonctionnement correct de $\mathcal{M}$.

- $C_{ij\gamma}$ reçoit la valeur vrai lorsque dans la configuration i, la cellule j contient le caractère γ. Au vu de la remarque III.4.2, il est clair que
$$0 \leq i, j \leq T(|w|).$$
Ainsi, le nombre de variables de ce type est majoré par $(T(|w|))^2.\#\Gamma$.
- Pour chaque configuration, une cellule est référencée par le curseur. Ainsi, L_{ij} reçoit la valeur vrai lorsque dans la configuration i, la cellule référencée est j. On a $(T(|w|))^2$ variables de ce type.

- Pour chaque configuration, nous devons connaître l'état dans lequel $\mathcal{M}$ se trouve. Ainsi, P_{ip} reçoit la valeur vrai lorsque dans la configuration i, la machine $\mathcal{M}$ se trouve dans l'état p. On a $\#Q.T(|w|)$ variables de ce type.
- Puisque la machine est non déterministe, on a dans chaque configuration, le choix entre au plus r transitions possibles supposées numérotées. Ainsi, A_{ij} reçoit la valeur vrai si le choix j a été opéré à la configuration i, $1 \leq j \leq r$. Si dans la configuration i, il y a $s < r$ transitions possibles, on pose $A_{ij} = 0$ pour $j = s+1, \ldots, r$. Le nombre de variables de ce type est majoré par $r.T(|w|)$.

Il est facile de vérifier que le nombre de variables utilisées est majoré par un polynôme en $|w|$. Il nous faut à présent associer une formule au comportement de $\mathcal{M}$. Le résultat intermédiaire suivant est immédiat.

Lemme III.4.3. *Soient $x_1, \ldots, x_n$ des variables propositionnelles. Exprimer le fait que dans une distribution des valeurs de vérité, exactement une des variables recoive la valeur vrai est équivalent à ce que l'évaluation de*

$$\left(\bigvee_{i=1}^{n} x_i\right) \wedge \left(\bigwedge_{1 \leq i < j \leq n} (\neg x_i \vee \neg x_j)\right)$$

soit vraie.

En particulier, nous allons l'utiliser avec plusieurs types de variables introduits ci-dessus. Pour chaque configuration de $\mathcal{M}$, il y a une seule cellule référencée (on applique le lemme aux variables L_{ij}) et il y a un seul état actif (les variables P_{ip}). Idem aussi pour les variables $C_{i,j,\sigma}$. Ensuite, pour passer de la configuration i à la confiuguration $i+1$, on choisit une seule transition (les variables A_{ij}). Observons d'ores et déjà que la longueur de la formule correspondante ainsi construite est polynomiale en le nombre de variables.

• Nous devons aussi exprimer que la configuration initiale est $q_0.\#w\underline{\#}$. Si $w = w_1 \cdots w_k$, alors la formule correspondante est de la forme

$$P_{0,q_0} \wedge C_{0,0,\#} \wedge C_{0,1,w_1} \wedge \cdots \wedge C_{0,k,w_k} \wedge C_{0,k+1,\#} \wedge L_{0,k+1}$$
$$\wedge C_{0,k+2,\#} \wedge \cdots \wedge C_{0,T(k),\#}.$$

• Il faut ensuite exprimer que l'on passe d'une configuration à la suivante en respectant la relation de transition. Tout d'abord, si une cellule n'est pas référencée à l'étape i, son contenu reste inchangé à l'étape $i+1$. Ceci se traduit par[22]

$$\bigwedge_{i,j,\gamma} (\neg L_{i,j} \rightarrow (C_{i,j,\gamma} \rightarrow C_{i+1,j,\gamma}))\,.$$

[22]Si à la configuration i, la cellule j n'est pas référencée, alors si cette cellule j contient le caractère γ à l'étape i, elle le contient encore à l'étape $i+1$.

Bien qu'ayant l'avantage d'être claire, cette formule n'est pas sous forme normale conjonctive. Pour y remédier, il suffit de se rappeler que $p \to q$ est logiquement équivalent à $\neg p \vee q$. Ainsi, la formule précédente se réécrit

$$\bigwedge_{i,j,\gamma} (L_{i,j} \vee \neg C_{i,j,\gamma} \vee C_{i+1,j,\gamma}) .$$

Passons à la cellule référencée. Lorsqu'on se trouve en l'état p à l'étape i et qu'un symbole γ est lu, on choisit, de manière non déterministe, une certaine transition de numéro u pour passer à la configuration $i+1$. Une fois ce choix effectué (et symbolisé par A_{iu}), on connaît le nouvel état q dans lequel va se trouver la machine $\mathcal{M}$, le nouveau symbole σ se trouvant dans la cellule qui était référencée à l'étape i ainsi que le déplacement de la tête de lecture. Ce dernier est codé par d qui peut prendre les valeurs -1, 0 et 1 suivant qu'un déplacement vers la gauche, qu'aucun déplacement ou qu'un déplacement vers la droite de la tête de lecture est effectué. Tout ceci se traduit simplement par

$$\bigwedge_{i,j,u,\gamma} (P_{i,p} \wedge C_{i,j,\gamma} \wedge L_{i,j} \wedge A_{i,u} \to P_{i+1,q} \wedge C_{i+1,j,\sigma} \wedge L_{i+1,j+d}) .$$

Il suffit encore une fois de mettre cette formule sous forme normale conjonctive.

• L'état d'acceptation (que nous supposons comme de coutume unique) est atteint en au plus $T(|w|)$ transitions. Si cet état était rencontré avant $T(|w|)$ étapes, on suppose que $\mathcal{M}$ subit des transitions supplémentaires pour un total de $T(|w|)$ sans modifier la configuration obtenue. (Une telle simulation pourrait une fois encore être réalisée par une machine de Turing universelle.) Ainsi, on a la formule

$$\bigvee_{i=1}^{T(|w|)} P_{i,h}$$

où h est l'état accepteur.

La formule finale est la conjonction des formules partielles obtenues au cours de la preuve. Il est clair qu'elle est de longueur polynomiale par rapport à $|w|$ et qu'elle admet une distribution des valeurs de vérité qui la rend vraie si et seulement si w est accepté par $\mathcal{M}$. On a donc bien une réduction polynomiale de L vers SAT.

■

Remarque III.4.4. Comme le fait par exemple observer P. Wolper, le fait que SAT soit NP-complet peut certes être vu comme un résultat négatif. Cependant, dans de nombreux cas pratiques pour lesquels on est amené à résoudre un problème de type SAT, on peut très bien réussir à obtenir une distribution des valeurs de vérité qui rende vraie une formule donnée en un temps raisonnable. En effet, la mesure de complexité considérée ici fait référence au *pire cas*. Alors qu'en pratique, il se peut que la majorité des

formules soient aisément satisfaisables ou facilement vérifiées comme n'étant jamais satisfaisables. Il ne reste alors qu'une "portion réduite" de formules devant conduire à parcourir l'espace des distributions possibles (cette constatation rejoint dès lors l'idée d'étudier la complexité moyenne). Ainsi, en pratique, la situation n'est pas toujours aussi désespérée qu'il n'y paraît puisqu'on peut très bien être en présence d'une formule "facile" et pas en présence des cas extrêmes.

Un premier exemple de réduction consiste à montrer que le problème $3SAT$ est lui aussi NP-complet. Ce problème est un cas particulier de SAT. On s'intéresse encore une fois à des formules sous forme normale conjonctive, mais chaque clause est la disjonction d'exactement trois termes (qui sont chacun des variables ou la négation de variables propositionnelles). Par exemple,

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee x_4 \vee \neg x_5)$$

est une instance de $3SAT$ (trivialement positive, il suffit par exemple que la valeur de vérité de x_1 soit fixée à vrai).

Proposition III.4.5. *$3SAT$ appartient à NPC.*

Démonstration. Il est clair que $3SAT$ appartient à NP (on dispose d'un certificat succint). Pour prouver que $3SAT$ est NP-complet, il suffit de vérifier que $SAT \leq 3SAT$. Autrement dit, nous devons exhiber une transformation polynomiale f, remplaçant toute instance φ de SAT par une instance $f(\varphi)$ de $3SAT$, de manière telle que l'instance φ puisse être satisfaite si et seulement si $f(\varphi)$ peut l'être. Une formule quelconque φ de SAT étant une conjonction de clauses, nous allons étudier séparément chaque clause de la forme

$$x_1 \vee x_2 \vee \cdots \vee x_n \tag{4}$$

où les x_i sont des variables ou des négations de variables. Nous allons montrer qu'une telle clause peut être remplacée par une disconjonction de clauses à 3 termes dont la taille est majorée par un polynôme en n. En effet, pour qu'une formule sous forme normale conjonctive soit satisfaite, il faut que chaque clause le soit séparément.

La clause

$$x_1$$

peut être remplacée en introduisant deux nouvelles variables y_1 et y_2 par

$$(x_1 \vee y_1 \vee y_2) \wedge (x_1 \vee \neg y_1 \vee y_2) \wedge (x_1 \vee y_1 \vee \neg y_2) \wedge (x_1 \vee \neg y_1 \vee \neg y_2).$$

On vérifie aisément que cette dernière formule est satisfaite si et seulement si la valeur de vérité de x_1 est vrai.

La clause

$$x_1 \vee x_2$$

peut être remplacée par

$$(x_1 \vee x_2 \vee y) \wedge (x_1 \vee x_2 \vee \neg y).$$

Une fois encore, la variable y est une nouvelle variable et la première clause est satisfaite si et seulement si la seconde l'est.

Enfin, si $n \geq 4$, la clause (4) peut être remplacée par

$$(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \cdots$$
$$\wedge(\neg y_{n-4} \vee x_{n-2} \vee y_{n-3}) \wedge (\neg y_{n-3} \vee x_{n-1} \vee x_n).$$

Si on dispose d'une distribution d des valeurs de vérité des variables apparaissant dans la formule initiale φ de SAT qui rende φ vrai, alors la distribution suivante permet de satisfaire la formule $f(\varphi)$. Soit j le plus grand indice tel que x_j soit vrai. On considère la distribution d' suivante des valeurs de vérité des variables de $f(\varphi)$. Pour toute variable y de φ, $d'(y) = d(y)$ et pour les nouvelles variables : $d'(y_1) = \cdots = d'(y_{j-2}) = 1$, $d'(y_{j-1}) = \cdots = d'(y_{n-3}) = 0$.

Nous devons encore montrer que si $f(\varphi)$ peut être satisfaite, il en est de même pour φ. Soit d' une distribution des valeurs de vérité des variables de $f(\varphi)$ qui la rende vraie. La restriction de d' aux variables de φ rend φ vrai. Procédons par l'absurde et supposons qu'avec une telle distribution, la clause (4) soit fausse. Autrement dit, pour tout $i \in \{1, \ldots, n\}$, $d(x_i) = d'(x_i) = 0$. Dans ce cas, on a nécessairement $d'(y_{n-3}) = 0$, puis $d'(y_{n-4}) = 0$ et de proche en proche, $d'(y_1) = 0$. Ceci rend la nouvelle clause de $f(\varphi)$ fausse, d'où la contradiction !

Ainsi, nous avons montré que si la formule initiale φ dispose d'une distribution des valeurs de vérité des variables qui la rende vraie, alors il en est de même pour la formule $f(\varphi)$ et inversement.

■

Remarque III.4.6. Le problème 2-SAT appartient à P (il s'agit d'un exercice que de le montrer). Par contre, avec des arguments semblables à ceux développés ci-dessus, k-SAT appartient à NPC pour tout $k > 3$.

5. D'autres cas

Un exemple de problème décidable et pourtant ne se trouvant ni dans NP, ni dans $co-NP$ est donné par la décidabilité de l'arithmétique de Presburger. Il s'agit de la théorie du premier ordre des nombres naturels muni de l'addition. On considère donc l'ensemble des formules closes (en anglais, "sentence", i.e., toute variable est liée par un quantificateur existentiel ou universel) et pour lesquelles on quantifie uniquement les variables (qui sont des naturels), comme par exemple,

$$(\forall x)(\forall y)(\exists z)(x + y = z),$$

ou

$$(\forall x)(\exists y)(x = y + y).$$

La première formule est vraie, alors que la seconde ne l'est pas (il suffit de prendre x impair). Pour passer au complémentaire, il suffit de regarder les formules fausses plutôt que les vraies.

Voici un extrait d'un article[23] de Fisher et Rabin :

Theorem 1: There exists a constant $c > 0$ such that for every decision procedure (algorithm) AL for Presburger arithmetic PA, there exists an integer n_0 so that for every $n > n_0$ there exists a sentence F of length n for which AL requires more than $2^{2^{cn}}$ computational steps to decide whether $F \in PA$.

The previous theorem applies also in the case of non-deterministic algorithms. This implies that not only algorithms require a super-exponential number of computational steps, but also proofs of true statements concerning addition of natural numbers are super-exponentially long.

La remarque suivante traite d'un langage NP-dur qui n'est pas dans NP.

Remarque III.5.1. Le langage H associé au problème de l'arrêt est NP-difficile sans pour autant être NP-complet[24]. En effet, il est facile de se convaincre que $SAT \leq H$. A toute instance de SAT, on associe une machine de Turing testant toutes les distributions des valeurs de vérité des variables. Si une distribution satisfait l'instance, alors la machine s'arrête. Sinon, on construit la machine de telle façon qu'elle entre d'une suite infinie de transitions (par exemple, déplacer la tête de lecture indéfiniment vers la droite). La machine ainsi construite s'arrête si et seulement si l'instance initiale de SAT est satisfaisable.

6. Quelques réductions

Après le théorème de Cook, on a démontré que des centaines d'autres problèmes classiques[25] sont eux aussi NP-complets. Dans cette section, nous allons en présenter quelques-uns. En général, on procède par réduction en montrant qu'un problème déjà démontré NP-complet se réduit au problème considéré (comme on l'a fait à la section précédente en montrant que $SAT \leq 3SAT$).

Pour débuter, montrons que le problème de la couverture de sommets (VC) introduit dans l'exemple II.9.3 est lui aussi NP-complet.

Proposition III.6.1. *VC est NP-complet.*

Démonstration. Il est clair que VC appartient à NP. Etant donné un graphe G et un entier n, on peut construire une machine de Turing qui

[23] cf. http://publications.csail.mit.edu/lcs/pubs/ps/MIT-LCS-TM-043.ps

[24] Si H appartenait à NP, au vu du théorème III.3.15, le langage serait décidable et nous savons qu'il n'en est rien

[25] Voir par exemple, sur internet la liste
http://www.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html
ou bien le classique M. Garey, D. Johnson, *Computers and Intractability – A Guide to the Theory of NP-Completeness*, Freeman, (1979).

choisit de manière non déterministe n sommets et vérifie s'il s'agit d'une couverture de G. Autrement dit, au vu de la remarque III.3.5, on dispose d'un certificat succinct : on sait tester de manière polynomiale qu'un sous-ensemble de sommets est bien une couverture de G.

Il nous suffit alors de montrer que $3SAT \leq VC$. Ainsi, nous recherchons une transformation polynomiale f qui, à toute formule φ de $3SAT$, associe un graphe G_φ et un entier n_φ (calculés en un temps polynomial) de manière telle que la formule admette une distribution de valeurs de vérité des variables qui la rende vraie si et seulement si l'instance $f(\varphi)$ correspondante est positive, i.e., le graphe G_φ admet une couverture de taille n_φ.

Soit φ une formule de la forme

$$C_1 \wedge \cdots \wedge C_k$$

où C_i est de la forme $x_{i1} \vee x_{i2} \vee x_{i3}$. On dénote par $y_1, \ldots, y_\ell$ les variables propositionnelles intervenant[26] dans φ. A chaque variable y_i, on associe le graphe élémentaire $\mathcal{G}_i$ représenté à la figure III.7 et à chaque clause C_i, on

FIGURE III.7. Graphe $\mathcal{G}_i$ associé à la variable y_i.

associe le graphe élémentaire $\mathcal{H}_i$ représenté à la figure III.8 .

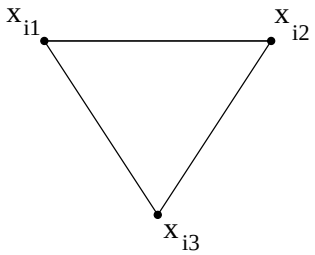


FIGURE III.8. Graphe $\mathcal{H}_i$ associé à la clause $C_i = x_{i1} \vee x_{i2} \vee x_{i3}$.

Enfin, on considère le graphe obtenu en réalisant l'union disjointe de ces graphes élémentaires à laquelle on ajoute des arcs joignant chaque sommet de $\mathcal{H}_i$ correspondant au terme x_{ij} ($1 \leq i \leq k$, $1 \leq j \leq 3$) au sommet associé à la variable qu'il représente.

Exemple III.6.2. La formule $\varphi = (y_1 \vee y_2 \vee \neg y_3) \wedge (\neg y_1 \vee y_2 \vee y_3)$ correspond, par la construction décrite ci-dessus, au graphe G_φ donné à la figure III.9.

[26] Il se peut qu'une même variable y_i ou que sa négation $\neg y_i$ apparaisse comme terme x_{mn} de plusieurs clauses.

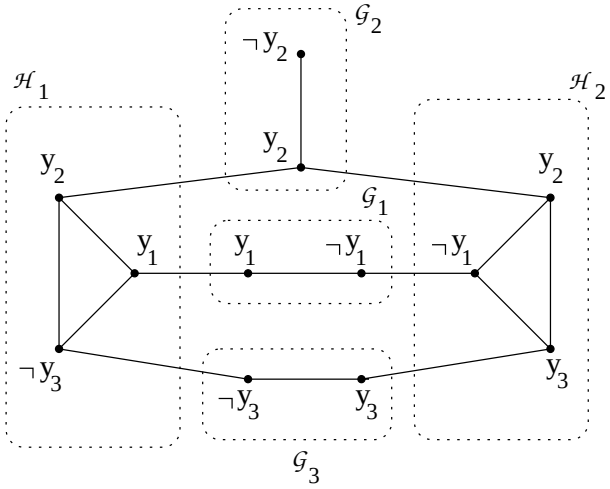


FIGURE III.9. Graphe correspondant à une instance de $3SAT$.

La transformation proposée est polynomiale car le nombre de sommets (resp. arêtes) obtenus est exactement $3k + 2\ell$ (resp. $6k + \ell$). Le nombre associé à l'instance de VC est fixé[27] à $2k + \ell$.

Si la formule φ admet une distribution τ des valeurs de vérité qui la rend vraie, alors le graphe G_φ admet une couverture : dans chaque graphe élémentaire $\mathcal{G}_i$, on place dans la couverture le sommet y_i ou $\neg y_i$ dont la valeur de vérité attribuée par τ est vrai. Dans chaque graphe $\mathcal{H}_i$, on place dans la couverture deux sommets de telle sorte que τ affecte la valeur vrai au sommet non choisi.

Exemple III.6.3. Si on poursuit l'exemple précédent, la distribution

$$\tau(y_1) = 1,\ \tau(y_2) = 0 \text{ et } \tau(y_3) = 1$$

est telle que la formule φ soit satisfaite. Il lui correspond la couverture donnée à la figure III.10.

Réciproquement, si le graphe G_φ possède une couverture de taille $2k + \ell$, celle-ci définit nécessairement une fonction d'évaluation qui permet de satisfaire la formule φ. Puisque $2k+\ell$ est la taille minimale de la couverture, pour tous i, j, exactement un sommet de chaque $\mathcal{G}_i$ et deux sommets de chacun des $\mathcal{H}_j$ sont dans la couverture. On considère la distribution des valeurs de vérité associant la valeur vrai au sommet de $\mathcal{G}_i$ présent dans la couverture. Pour chaque $\mathcal{H}_j$, associé à la clause $x_{j1} \vee x_{j2} \vee x_{j3}$, un des sommets x_{jk} n'est pas dans la couverture. Or puisque le graphe tout entier est couvert, cela signifie que x_{jk} est couvert dans le graphe $\mathcal{G}_i$ correspondant et donc que la clause est satisfaite par la distribution des valeurs de vérité considérée.

[27] C'est un minimum. En effet, pour chaque graphe élémentaire $\mathcal{G}_i$ correspondant à une variable y_i, la couverture doit en contenir un sommet et pour chaque graphe élémentaire $\mathcal{H}_i$ correspondant à une clause C_i, la couverture doit en contenir deux.

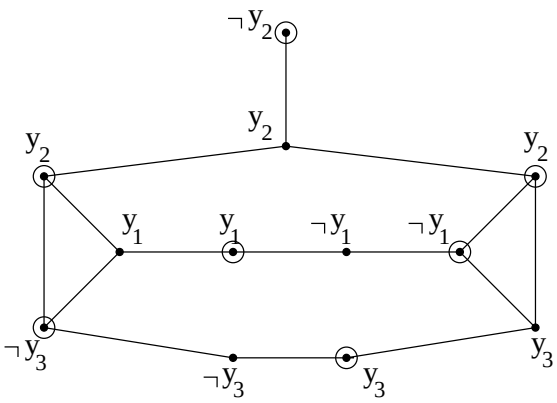


FIGURE III.10. Une couverture de G_φ correspondant à τ.

■

Remarque III.6.4. Si φ est une formule admettant une distribution des valeurs de vérité des variables qui la rende vraie, alors il n'y a pas nécessairement unicité de la couverture du graphe G_φ associé. Prenez par exemple la formule $(x_1 \vee x_2 \vee x_3)$.

Le problème du circuit hamiltonien HC introduit dans l'exemple III.2.4 est lui aussi NP-complet.

Proposition III.6.5. *HC est NP-complet.*

Démonstration. On se convainc encore une fois aisément que HC appartient à NP (étant donné un circuit, on vérifie de manière polynomiale s'il est hamiltonien). Nous allons montrer[28] que $VC \leq HC$. Nous devons donc définir une transformation polynomiale f qui, étant donné une instance x de VC (un graphe et une borne), fournit une instance $f(x)$ de HC (un graphe) de manière telle que x possède une couverture si et seulement si $f(x)$ possède un circuit hamiltonien.

Soit un graphe $G = (V, E)$ (non orienté) et une borne k. Avec ces données, on construit un nouveau graphe G' de telle manière que si $(x, y) \in V$, alors les 6 sommets et 14 arêtes en "double croix" représentés à la figure III.11 appartiennent à G'. Ainsi, à chaque sommet d'une arête de G, correspondent 6 sommets de G'. De plus, si n arêtes de G ont une extrémité en commun ($n \geq 2$), on joint les suites correspondantes de 6 sommets dans G' à l'aide de $n-1$ nouveaux arcs. Par exemple, si (x, y), (x, z) et (x, t) appartiennent à V, on a la situation représentée à la figure III.12 dans G' (l'ordre n'a pas d'importance). Enfin, à la borne k, correspondent k nouveaux sommets $\alpha_1, \ldots, \alpha_k$ de G'. Chacun d'eux est relié aux (deux) extrémités libres des chemins correspondant à un même sommet de G. Pour fixer les idées, considérons l'exemple suivant (où $k = 2$) représenté à la figure III.13. Il est

[28] Il est intéressant de remarquer que $SAT \leq 3SAT \leq VC \leq HC$.

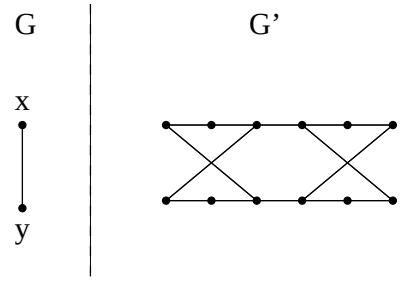


FIGURE III.11. Construction de G'.

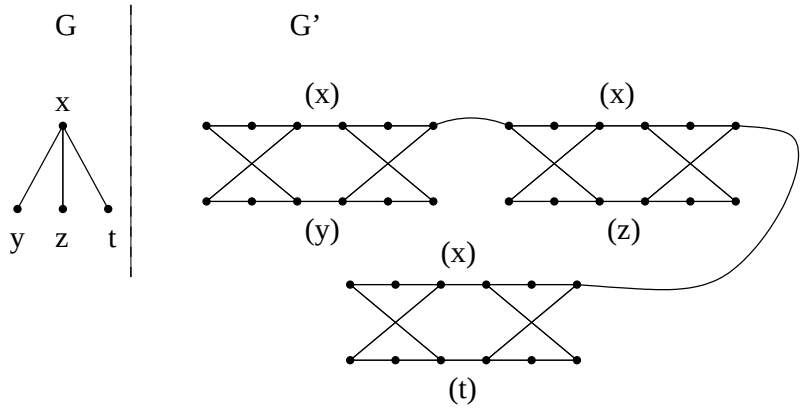


FIGURE III.12. Construction de G' (deuxième partie).

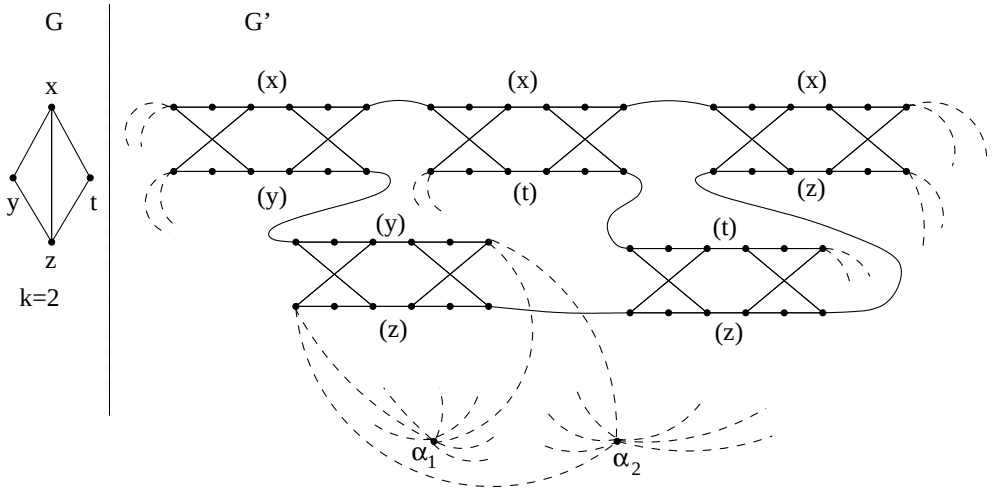


FIGURE III.13. Construction de G' (fin).

clair que la construction de G' ne requiert qu'un temps polynomial en la donnée G, k.

(a) Nous allons nous convaincre que si le graphe G' possède un circuit hamiltonien, alors G possède une couverture de taille k. En particulier, le circuit hamiltonien passe une et une seule fois par les différents sommets

$\alpha_1, \ldots, \alpha_k$ dans un ordre déterminé par une permutation ν. Entre α_{ν_i} et $\alpha_{\nu_{i+1}}$, le circuit passe uniquement par des sommets de G' placés en "double croix" et construits à partir d'arcs de G. Remarquons que pour passer une et une seule fois par chaque sommet de G', les possibilités de parcours d'une "double croix" sont réduites. En fait, elles sont complètement décrites à la figure III.14. En particulier, lorsqu'on entre dans une "double croix", on en ressort nécessairement le long de la même extrémité. Le sens de

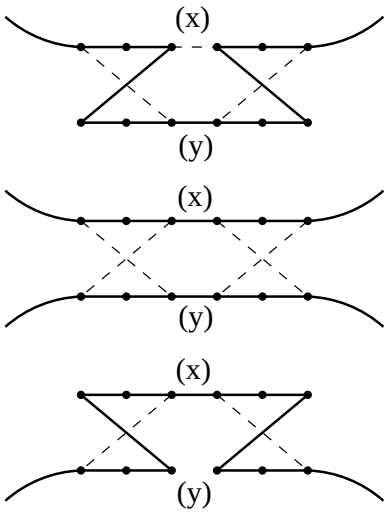


FIGURE III.14. Les seules possibilités de parcours.

parcours défini par le circuit hamiltonien induit le choix d'une couverture. Entre α_{ν_i} et $\alpha_{\nu_{i+1}}$, on passe par des "doubles croix" et on va nécessairement parcourir d'une seule traite toutes les "doubles croix" correspondant aux arcs de G ayant une même extrémité en commun[29]. On va donc placer ce sommet dans la couverture de G. Remarquons qu'entre α_{ν_1} et α_{ν_2}, on sélectionne un sommet de G, ...; entre α_{ν_k} et α_{ν_1}, on sélectionne encore un sommet de G. Donc au total, on aura sélectionné k sommets de G et de par la construction, on a clairement[30] une couverture de G (chaque arc de G possède au moins une extrémité dans la couverture puisque chaque "double croix" a été parcourue dans G'). Pour bien comprendre, reprenons la figure III.13 dans laquelle on ne représente que les arcs d'un circuit hamiltonien.

[29]Lorsqu'on entre dans une "double croix", on en ressort nécessairement le long de la même extrémité et cela nous conduit donc à entrer à l'extrémité la "double croix" voisine qui, par construction, correspond au même sommet de G et ainsi de suite jusqu'à avoir épuisé toutes les "doubles croix" construites sur un même sommet de G.

[30]Dit autrement, puisqu'on a un circuit hamiltonien, chaque "double croix" de G' est entièrement parcourue. Chacune de ces croix correspond exactement à une arête du graphe de départ G. De plus, suivant le point d'entrée du circuit hamiltonien dans la "double croix", on sélectionne un sommet dans la couverture de G. Par conséquent, chaque arête de G possède au moins une extrémité dans la couverture construite.

La figure correspondante est donnée à la figure III.15 et un exemple pour $k = 3$ est donné à la figure III.16.

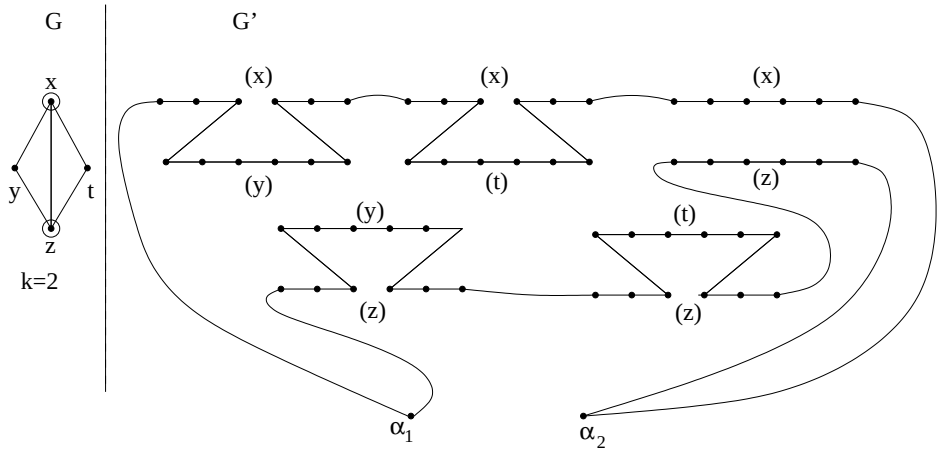


FIGURE III.15. Un circuit hamiltonien et la couverture correspondante ($k = 2$).

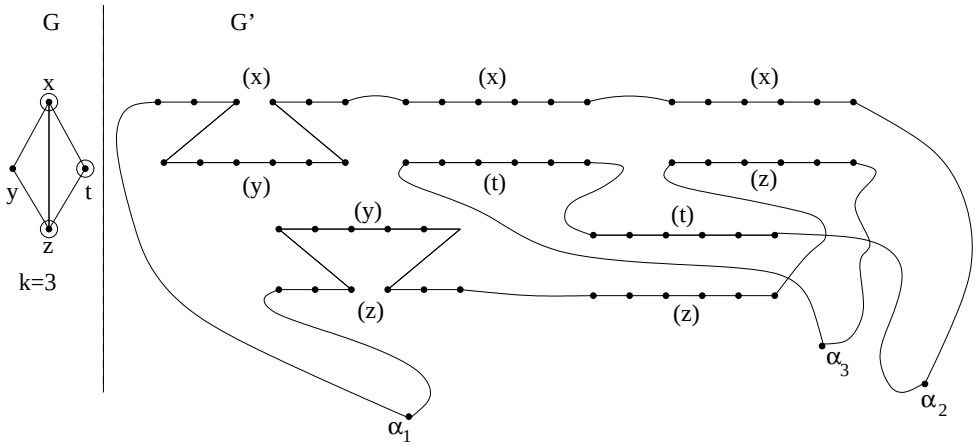


FIGURE III.16. Un circuit hamiltonien et la couverture correspondante ($k = 3$).

(b) Inversement, si G possède une couverture de taille k, alors le graphe correspondant G' possède un circuit hamiltonien. Si un arc de G a ses deux extrémités dans la couverture, alors on parcourt dans G' la "double croix" correspondante comme à la position centrale de la figure III.14. Sinon, une seule extrémité appartient à la couverture et on parcourt entièrement la "double croix" correspondante en débutant le long de l'extrémité appartenant à la couverture.

■

On pourrait multiplier les exemples astucieux de réduction. Pour terminer cette section, signalons quelques résultats amusants.

Exemple III.6.6 (Tetris). Considérons le célèbre jeu Tetris joué *hors ligne*, i.e., le joueur connaît d'avance la suite (finie) de tetrominos qui va lui être proposée. Il dispose donc de plus d'informations que dans le véritable jeu. De plus, on suppose le joueur parfaitement habile : il a toujours le temps de placer une pièce comme il le désire. On peut montrer que les problèmes suivants sont NP-complets :

- maximiser le nombre de lignes effacées en une partie,
- maximiser le nombre de "tetris", i.e., quadruplets de lignes remplies et effacées en un coup,
- minimiser la hauteur maximum occupée par les pièces,
- maximiser le nombre de pièces jouées avant la fin de la partie.

On transforme facilement ces problèmes en problèmes de décision. On se donne une suite de tetrominos et un entier k et on pose par exemple la question de savoir s'il est possible ou non d'obtenir ou non k "tetris" (ou encore d'effacer k lignes, etc...).

Le lecteur curieux pourra consulter les articles : E. D. Demaine, S. Hohenberger, D. Liben-Nowell, *Tetris is Hard, Even to Approximate* et R. Breukelaar, H. J. Hoogeboom, W. A. Kosters, *Tetris is Hard, Made Easy* compilés dans International Journal of Computational Geometry & Applications **14** (2004), 41–68.

Une version électronique du premier papier se trouve aussi sur `http://lanl.arXiv.org/abs/cs.CC/0210020`.

Exemple III.6.7 (Démineur). Dans la même optique, le célèbre jeu du démineur est lui aussi NP-complet. Plus précisément, il s'agit du problème de "consistance de grilles". Une instance du problème est une configuration du jeu[31] et la question qui est posée est de déterminer si une telle configuration peut réellement apparaître. Par exemple, la grille

	1	1	1	
	1	0	1	
	1	1	1	
				0

est une instance négative du problème[32]. Voir l'article : R. Kaye, Minesweeper is NP-complete, *Math. Intelligencer* **22** (2000), 9–15.

Au vu de la remarque III.3.10, pour démontrer que $P = NP$ (et ainsi gagner un prix d'un million de dollars), il vous suffirait de trouver un algorithme polynomial résolvant ce problème...

[31] Nous supposons que vous avez tous déjà au moins une fois joué au démineur !

[32] Démonstration ?

7. Complexité et cryptographie

De nombreux problèmes de nature cryptographique comme le problème du logarithme discret ou la factorisation de grands entiers reposent sur des problèmes réputés difficiles et sur la notion de fonction à sens unique[33]. Dans cette courte section, nous allons voir ce qu'il en est réellement.

En guise de préambule, on pourrait naïvement penser que le problème voulant décider si un entier n est premier (problème dont le langage codant les instances positives sera noté `PRIMES` par la suite) est polynomial en la *taille des données*. En effet, si n est composé, il possède un diviseur inférieur ou égal à $\sqrt{n}$. Il suffit[34] alors de passer en revue les nombres entiers $x \leq \sqrt{n}$ et d'effectuer pour chacun d'eux la division euclidienne de n par x. On peut de plus montrer que cette dernière opération nécessite de l'ordre de $\log^2 n$ opérations élémentaires. Ainsi, cet algorithme naïf requiert $\mathcal{O}(n \log^2 n)$ opérations et on serait tenté de conclure trop hâtivement qu'il s'agit d'un algorithme polynomial (où il faut sous-entendre polynomial en la *taille des données*). En effet, si la donnée n est écrite dans une base quelconque $b \geq 2$, nous savons que la longueur de la représentation de n est proportionnelle à $\log_b n$. L'algorithme décrit ici n'est donc pas polynomial mais bien exponentiel par rapport à la taille $\log_b n$ de la donnée n !

Historiquement, `PRIMES` est resté un des seuls problèmes dont on connaissait l'appartenance à $NP \cap \text{co-}NP$ mais dont on ne connaissait pas l'appartenance à P. Plus précisément, on pouvait uniquement démontrer son appartenance à P en supposant l'hypothèse de Riemann étendue satisfaite.

Proposition III.7.1. `PRIMES` $\in NP \cap$ *co-*NP*,*

Démonstration. Pour montrer que `PRIMES` appartient à NP, on peut par exemple considérer le critère connu sous le nom de *test de Lucas*. Ce dernier va nous permettre de construire un algorithme non déterministe polynomial vérifiant l'appartenance à `PRIMES`.

Lemme III.7.2. *Un entier n est premier si et seulement si il existe $g \in \mathbb{Z}_n^*$ tel que*

- $g^{n-1} \equiv 1 \mod n$ *et*
- $g^{\frac{n-1}{p}} \not\equiv 1 \mod n$ *pour tout p facteur premier de $n-1$.*

Démonstration. La condition est nécessaire. Si n est premier, $\mathbb{Z}_n$ est un champ et il existe donc g engendrant[35] $\mathbb{Z}_n^*$ (on dit que g est un élément

[33] cf. le cours de structures discrètes. Pour comprendre la notion de fonction à sens unique, considérons l'exemple suivant : il est facile de faire le produit de deux grands nombres entiers, par contre, connaissant le résultat, il peut être beaucoup plus long d'en retrouver les facteurs.

[34] Méthode connue sous le nom du crible d'Erathostène.

[35] Si $(A, +, .)$ est un anneau, A^* dénote le groupe (multiplicatif) des éléments inversibles de A. On peut montrer que le nombre de générateurs de $\mathbb{Z}_n^*$ est $\varphi(n-1)$ où φ est la fonction d'Euler.

primitif modulo n, son ordre est égal à l'ordre de $\mathbb{Z}_n^*$ qui est $n-1$). Un tel élément g satisfait les deux conditions. Réciproquement, s'il existe $g \in \mathbb{Z}_n^*$ satisfaisant les deux conditions, alors l'ordre de g est nécessairement $n-1$. En effet, la première condition nous montre que l'ordre de g divise $n-1$ et vu la seconde condition, il ne peut être inférieur. Puisque $\mathbb{Z}_n^*$ contient un élément d'ordre $n-1$, $\mathbb{Z}_n^*$ contient lui-même $n-1$ éléments et on en conclut que $\mathbb{Z}_n$ est un champ (car tout élément non nul est inversible). Par conséquent, n est premier.

■

Au vu de ce résultat, on peut considérer l'algorithme suivant, connu sous le nom d'*algorithme de Pratt*[36]. Une instance du problème est un entier n. Si n est premier, on en obtient une certification succincte, c'est-à-dire qu'on peut vérifier, grâce à certains choix non déterministes, en un temps polynomial en la taille de la donnée que n est premier.

(1) Si $n = 2$, l'instance est positive (n est premier).
(2) Si $n > 2$ est pair, l'instance est négative (n n'est pas premier).
(3) Choisir de manière non déterministe une factorisation de $n-1$,

$$n - 1 = \prod_{i=1}^{m} p_i^{\alpha_i}.$$

On peut vérifier polynomialement si la factorisation proposée est correcte (il suffit d'effectuer les produits et vérifier qu'on retrouve bien $n-1$).
(4) Choisir $g \in \{2, \ldots, n-1\}$ de manière non déterministe, puis vérifier que $g^{n-1} \equiv 1 \mod n$.
(5) Vérifier que pour tout $i \leq m$, $g^{\frac{n-1}{p_i}} \not\equiv 1 \mod n$.
(6) Vérifier récursivement que les p_i sont premiers.

Il faudrait à présent étudier la complexité de l'algorithme proposé. On peut vérifier que les multiplications du point 3 peuvent être réalisées[37] en $\mathcal{O}(\log n)$ opérations. Les points 4 et 5 peuvent être calculés par exponentiation modulaire (élévations successives au carré et réduction $\mod n$) en $\mathcal{O}(\log n)$ opérations. Enfin, puisque l'algorithme est résursif, on en déduit une relation de récurrence sur le temps de calcul. Lorsqu'on résout cette dernière, on peut montrer que la complexité totale est en $\mathcal{O}(\log^5 n)$. Ceci montre bien que $\texttt{PRIMES} \in NP$.

Le fait que $\texttt{PRIMES} \in$ co-NP est évident. Si n est composé, on en devine deux facteurs de manière non déterministe et pour obtenir un certificat succinct, il suffit de les multiplier pour vérifier qu'on retrouve bien n (la multiplication étant de complexité polynomiale).

[36] V. R. Pratt, Every prime has a succinct certificate, *SIAM J. Comput.* **4** (1975), 214–220.

[37] Considérer des opérations sur les bits plutôt que des opérations élémentaires en termes de machines de Turing, peut éventuellement modifier les degrés des polynômes mis en cause mais ne changera en aucun cas l'appartenance à P ou NP.

■

Remarque III.7.3. En particulier, au vu du corollaire III.3.14, si on admet que $NP \neq \text{co-}NP$, alors PRIMES n'est pas NP-complet.

Remarque III.7.4. Soit le problème de décision $FACT$ dont une instance est donnée par un entier n et par un nombre $M \leq n$ et qui consiste à déterminer si n possède un diviseur inférieur à M. Ce problème de décision est "aussi difficile" que de déterminer les facteurs de n. Plus précisement, si $FACT \in P$, alors on peut factoriser n en un temps polynomial (la réciproque est quant à elle triviale). Il suffit de procéder par dichotomie. On sait répondre polynomialement à la question : n possède un diviseur inférieur à $\lfloor\sqrt{n}\rfloor$. Si la réponse est non, on en conclut que n est premier. Sinon, n est composé et on sait décider polynomialement si n possède un diviseur inférieur à $\lfloor\sqrt{\frac{n}{2}}\rfloor$. Si la réponse est oui (resp. non), n possède un diviseur dans l'intervalle $[1, \sqrt{\frac{n}{2}}]$ (resp. $[\sqrt{\frac{n}{2}}, \sqrt{n}]$). On continue en divisant à chaque étape par 2 la longueur de l'intervalle considéré.

En fait, en 2002, M. Agrawal et deux de ses étudiants N. Saxena et N. Kayal ont obtenu bien mieux en démontrant le résultat suivant[38] et ce, de manière inconditionnelle (i.e., indépendamment de la validité d'une "quelconque" conjecture en théorie des nombres, comme par exemple l'hypothèse de Riemann).

Théorème III.7.5. PRIMES $\in P$.

Rassurez-vous ! Ce n'est pas parce que PRIMES est dans P que la sécurité des systèmes cryptographiques est remise en cause. En effet, le cryptosystème RSA repose par exemple sur la difficulté calculatoire de factoriser un grand nombre composé. Savoir en un temps polynomial qu'un nombre est composé n'apporte pas nécessairement d'information sur la nature de ses facteurs. Il faut cependant rester vigilant sur les avancées de la recherche qui pourraient remettre en cause la sécurité de tels systèmes cryptographiques.

Terminons ces quelques remarques en considérant le problème du logarithme discret. Si p est un nombre premier et γ un élément primitif modulo p, alors

$$\text{dlog}_\gamma(x)$$

est l'unique $d \in \{1, \ldots, p-1\}$ tel que $\gamma^d = x$. Si p n'est pas premier ou si γ n'est pas primitif, on convient ici de poser $\text{dlog}_\gamma(x) = 0$.

Tout comme à la remarque III.7.4, on peut transformer le problème consistant à calculer $\text{dlog}_\gamma(x)$ en un problème de décision équivalent[39]. Ce

[38] L'article se trouve sur http://www.cse.iitk.ac.in/news/primality.html. On pourra aussi consulter F. Bornemann, PRIMES is in P, une avancée accessible à "l'homme ordinaire", Gazette des Mathématiciens **98** (2003), 14–29, traduction de l'article, PRIMES is in P: a breakthrough for "Everyman", Notices Amer. Math. Soc. **50** (2003), no. 5, 545–552.

[39] Si le problème de décision $PLOG$ appartient à P, alors, en procédant par dichotomie, on pourrait calculer le logarithme discret en un temps polynomial.

dernier problème, noté $PLOG$, a pour instance un quadruplet d'entiers p, γ, x, t et la question qui est posée est : a-t-on $\mathrm{dlog}_\gamma(x) > t$?

Proposition III.7.6. *$PLOG \in NP \cap co\text{-}NP$ et en particulier, $PLOG$ n'est pas NP-complet.*

Démonstration. La démonstration de ce résultat sort du cadre introductif de ce cours.

■

8. D'autres classes de complexité

Il n'est pas rare de rencontrer d'autres classes de complexité temporelle faisant intervenir une génération aléatoire de "bits". Dans ce cas de figure, on imagine disposer d'une instruction du type : étant donné un nombre positif b, générer un échantillon aléatoire ayant une distribution uniforme sur $\{0, \ldots, b-1\}$. Un algorithme utilisant ce type d'instruction sera dit probabiliste (*randomized algorithm*) et on suppose en outre qu'un élément de $\{0, \ldots, b-1\}$ peut être obtenu aléatoirement en $\log b$ opérations.

Définition III.8.1. Un langage L appartient à RP (*Randomized Polynomial time*) s'il existe un algorithme probabiliste acceptant, avec une probabilité élevée, les mots de L et refusant systématiquement les mots n'appartenant pas à L. On demande que l'algorithme fonctionne en temps polynomial et ce, quels que soient les tirages aléatoires qu'il effectue durant son exécution.

Remarque III.8.2. Si L appartient à RP, l'algorithme probabiliste mis en oeuvre pour décider de L est tel que

$$\mathbb{P}(\text{réponse : } w \in L | w \in L) = 1 - \epsilon, \quad \mathbb{P}(\text{réponse : } w \notin L | w \in L) = \epsilon$$
$$\mathbb{P}(\text{réponse : } w \notin L | w \notin L) = 1$$

On parle parfois de "*one-sided error*".

Exemple III.8.3. En utilisant le test de Miller-Rabin, on peut montrer que `PRIMES` $\in$ co-RP. Nous ne présenterons pas ce test ici[40], mais on peut rappeler que si on applique k fois consécutivement le test de Miller-Rabin à un nombre composé m, la probabilité de ne pas découvrir son caractère composé est inférieure à 4^{-k}.

Définition III.8.4. Un langage L appartient à BPP (*Bounded Probabilistic Polynomial-time*) s'il existe un algorithme probabiliste acceptant, avec une probabilité élevée, les mots de L et refusant, avec une probabilité élevée, les mots n'appartenant pas à L (on parle de "*two-sided error*". On demande, tout comme pour RP, que l'algorithme fonctionne en temps polynomial et ce, quels que soient les tirages aléatoires qu'il effectue durant son exécution.

[40] cf. une fois encore le cours de structures discrètes.

Définition III.8.5. Un langage L appartient à ZPP (*Zero-error Probabilistic Polynomial-time*) s'il existe un algorithme acceptant tous les mots de L, refusant tous les mots de son complémentaire et dont la durée d'exécution *attendue* est polynomiale. Cela signifie qu'avec une probabilité faible, l'algorithme peut ne pas se comporter polynomialement.

On peut montrer que

$$P \subseteq ZPP \subseteq RP \subseteq NP \cap BPP.$$

Signalons que pour la complexité spatiale, on peut procéder comme précédemment et définir diverses classes de complexité.

Définition III.8.6. Soient $\mathcal{M}$ une machine de Turing (déterministe ou non) et m un mot. La *consommation mémoire* de $\mathcal{M}$ sur m, notée $e_{\mathcal{M}}(m)$ est

- 0 si m n'est pas accepté par $\mathcal{M}$,
- la borne inférieure du nombre maximum de cases mémoire utilisées dans une suite de transition permettant d'atteindre un état accepteur depuis la configuration $q_0.\#m\underline{\#}$.

Définition III.8.7. La *fonction de complexité spatiale* d'une machine $\mathcal{M}$ (déterministe ou non) est la fonction $U_{\mathcal{M}} : \mathbb{N} \to \mathbb{N}$ définie par

$$U_{\mathcal{M}}(n) = \sup\{e_{\mathcal{M}}(m) \mid |m| = n\}.$$

De manière analogue à ce qui a été fait précédemment, on définit la classe $P - SPACE$ (resp. $NP - SPACE$) des langages décidés par une machine de Turing déterministe (resp. acceptés par une machine de Turing non déterministe) consommant des ressources mémoire majorées par un polynôme en la taille des données.

Il est clair qu'une classe de complexité temporelle est incluse dans la classe de complexité spatiale correspondante car une unité de temps est nécessaire pour se déplacer d'une case et considérer un nouveau bloc mémoire. Il est à noter que pour les classes de complexité spatiale, on a

$$P - SPACE = NP - SPACE$$

et donc,

$$P \subseteq NP \subseteq P - SPACE = NP - SPACE.$$

Bibliographie

[1] E. Bach, J. Shallit, *Algorithmic Number Theory*, vol. 1, efficient algorithms, MIT Press, (1997).

[2] R. Breukelaar, E. D. Demaine, S. Hohenberger, H. J. Hoogeboom, W. A. Kosters, D. Liben-Nowell, Tetris is Hard, Even to Approximate, *Internat. J. Comput. Geom. Appl.* **14** (2004), 41–68.

[3] R. Cori, D. Lascar, *Logique Mathématique*, Dunod, Paris, (1993).

[4] P. Flajolet, R. Sedgewick, *An introduction to the analysis of algorithms*, Addison-Wesley, (1996).

[5] M. J. Fischer, M. O. Rabin, *Super-Exponential Complexity of Presburger Arithmetic*, Proceedings of the SIAM-AMS Symposium in Applied Mathematics Vol. **7** (1974) 27–41.

[6] M. R. Garey, D. S. Johnson, *Computers and Intractability, A guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, (1979).

[7] V. Halava, *The Post Correspondence Problem for Marked Morphisms*, TUCS Dissertations **37**, Turku Centre for Computer Science.

[8] T. Harju, J. Karhumäki, Morphisms, in *Handbook of Formal Languages*, vol. 1, Springer, (1997).

[9] A. Hodges, The Alan Turing home page, `http://www.turing.org.uk/turing/`

[10] R. Kaye, Minesweeper is NP-complete, *Math. Intelligencer* **22** (2000), 9–15.

[11] P. Lecomte, *Algorithmique et calculabilité*, Notes de cours, Université de Liège, (1994).

[12] H. R. Lewis, C. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, (1981).

[13] B. Martin, *Codage, cryptologie et applications*, Presses polytechniques et universitaires romandes, (2004).

[14] C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley, (1994).

[15] *Handbook of discrete and combinatorial mathematics*, K. H. Rosen et al. Ed., CRC Press, Boca Raton, (2000).

[16] A. Salomaa, *Public-Key Cryptography*, second Ed., Texts in Theoretical Computer Science, Springer, (1996).

[17] A. Turing, On Computable Numbers, with an Application to the Entscheidungsproblem, *Proc.London Math. Society. Second Series* **42**, 230–265, (1936).

[18] D. Wood, *Theory of Computation*, Harper and Row, (1987).

[19] P. Wolper, *Introduction à la calculabilité*, InterEditions, Paris, (1991).

Liste des figures

I.1	Enumération de Peano.	12
I.2	Les fonctions $\mathcal{A}_0, \ldots, \mathcal{A}_3$.	15
II.1	Une transition.	29
II.2	Un ruban mémoire et son curseur.	30
II.3	Une machine de Turing.	31
II.4	Une machine de Turing calculant $n \mapsto 2n$.	32
II.5	Une machine de Turing calculant $\Sigma_2 : (m, n) \mapsto m + n$.	33
II.6	Une machine de Turing calculant χ_L.	33
II.7	Obtention d'une configuration pendante.	34
II.8	Une machine de Turing ne s'arrêtant pas toujours.	34
II.9	Un organigramme.	36
II.10	Des machines élémentaires.	37
II.11	Un organigramme pour $\mathcal{S}_{L,1}$.	38
II.12	Un organigramme pour $\mathcal{S}_{L,k}$.	39
II.13	Un organigramme pour $\mathcal{S}_{R,k}$.	39
II.14	Un organigramme pour $\mathcal{C}_k$.	39
II.15	Un organigramme pour $\mathcal{E}_k$.	40
II.16	La fonction c_f est calculable.	52
II.17	Représentation sagitale "compacte".	56
II.18	$BB(2) \geq 4$.	57
II.19	Représentation sagitale.	57
II.20	Un générateur aléatoire de nombres.	58
II.21	Une machine de Turing non déterministe.	58
II.22	L'arbre des configurations.	59
II.23	La machine $\mathcal{M}_n$ produisant n.	62
II.24	Une machine à coder.	66
II.25	Un graphe $G = (V, E)$.	70
II.26	Villes et coûts de transport.	72
II.27	L et son complémentaire sont acceptables.	76

II.28	Simulation en parallèle de $\mathcal{M}$ et $\mathcal{M}'$.	77
II.29	A l'étape 6, les mots m_1, m_4, m_5 sont déjà énumérés.	78
II.30	Un pavage du premier quadrant.	84
II.31	Pavé de type 1.	85
II.32	Pavé de type 2.	85
II.33	Deux pavés complémentaires de type 3.	85
II.34	Deux pavés complémentaires de type 4.	86
II.35	Le pavé initial et un pavé "blanc".	86
II.36	Quelques juxtapositions admissibles.	86
II.37	Première ligne du pavage.	87
II.38	Une machine de Turing $\mathcal{M}$.	87
II.39	Les pavés de $P_{\mathcal{M}}$ associés à $\mathcal{M}$.	87
II.40	Le pavage P_M.	88
III.1	x, $\ln x$, $x\lvert\sin x\rvert$.	91
III.2	$x^2 + \sin 6x$, $\frac{4}{5}x^2$ et $\frac{6}{5}x^2$.	92
III.3	Une machine de Turing calculant $n \mapsto 2n$.	93
III.4	Un circuit hamiltonien dans un cube.	95
III.5	$P \subset NP$ et $NPC \subset NP$.	98
III.6	Les différentes classes.	100
III.7	Graphe $\mathcal{G}_i$ associé à la variable y_i.	107
III.8	Graphe $\mathcal{H}_i$ associé à la clause $C_i = x_{i1} \vee x_{i2} \vee x_{i3}$.	107
III.9	Graphe correspondant à une instance de $3SAT$.	108
III.10	Une couverture de G_φ correspondant à τ.	109
III.11	Construction de G'.	110
III.12	Construction de G' (deuxième partie).	110
III.13	Construction de G' (fin).	110
III.14	Les seules possibilités de parcours.	111
III.15	Un circuit hamiltonien et la couverture correspondante ($k = 2$).	112
III.16	Un circuit hamiltonien et la couverture correspondante ($k = 3$).	112

Index

Notations

3SAT *104*

A

Ackermann (fonction d') *14*
addition *4*
alphabet *28*
  entrée (d') *29*
  ruban (de) *29*
arête *70*

C

castor affairé *56*
castor affairé
  fonction *57*
certification succincte *96*
Church Alonzo *28*
circuit hamiltonien *94*
clause *73*
complexité *90*
complexité spatiale *116*
composition *2*
concaténation *28*
configuration
  mémoire *29*
  machine *30*
  pendante *30*
couverture *70*

D

discrédit succinct *98*
durée d'exécution *90, 116*

E

etat *29*
  accepteur *29*
  initial *29*
Euler (fonction φ) *113*

F

fonction
  Ackermann *14*
  addition *4*
  calculable *31*
  caractéristique *6*
  castor affairé *57*
  composée *2*
  initiale *1*
  prédécesseur *5*
  primitive récursive *3*
  produit *4*
  produit borné *6*
  projection *2*
  puissance *5*
  récursion généralisée *13*
  récursion primitive *2*
  récursive *25*
  somme bornée *6*
  successeur *2*
  transition *29*
  zéro *2*
formule
  forme normale *73*
  satisfaisable *73*

G

gödélisation *46*
Gödel *46*
générateur aléatoire *58*
graphe *70*
  matrice d'incidence *70*

H

HC 94

I

incidence
  matrice 70
instance
  négative 68
  positive 68

L

langage 28
  accepté 52, 58
  acceptable 52
  co-NP 98
  co-NPC 98
  décidable 52
  NP-complet 97
  NP-difficile 98
  NP-dur 98
  récursif 53
  récursivement énumérable . . . 53, 77
lettre 28
Lucas (test de) 113

M

machine de Turing 29
  non déterministe 57
  polynomiale 92
matrice d'incidence 70
minimisation 24
  bornée 9
miroir 45
mot
  accepté 52, 58
  bi-infini 53
  fini 28
  infini 29
  vide 28

N

nombre (de Gödel) 46
NP 96
NPC 97

O

organigramme 36

P

P 95
P-calculable 92
palindrome 45
partie
  primitive récursive 6
  sûre 24
pavé 84
pavage 83
  solution 84
PCP 74
Post correspondence problem 74
prédicat 6
Pratt (algorithme de) 113
primitif (élément) 113
problème
  décidable 69
  décision 68
  instance 68
produit 4

R

récursion primitive 2
  généralisée 13
récursivement énumérable
  langage 77
réduction 81
relation de transition 57

S

SAT 73
satisfaisabilité 73
schéma
  composition 2
  définition par cas 8
  minimisation 24
  minimisation bornée 9
  récursion primitive 2
    généralisée 13
sommet 70
symbole 28
  blanc 29

T

tetris 111
transformation polynomiale 93
travel-salesman problem 71
TS 71
TS' 94
tuile 84

Turing
Alan 27
machine 29

V

valeur de vérité 73
VC 70, 105
vertex cover 70
voyageur de commerce 71