LOGIQUE, THÉORIE DES MODÈLES, COMPLEXITÉ

Cours à l'Université de Rennes 1 (2011–2012)

Antoine Chambert-Loir

Antoine Chambert-Loir
IRMAR, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex.
E-mail : antoine.chambert-loir@univ-rennes1.fr
Url : http://perso.univ-rennes1.fr/antoine.chambert-loir

Pour l'essentiel, ce cours et ses exercices sont tirés du livre de Sipser (2006), Introduction to the Theory of Computation.
Version du 18 janvier 2012
La version la plus à jour est disponible sur le Web à l'adresse http://perso.univ-rennes.fr/antoine.chambert-loir/2011-12/ltmc/

SOMMAIRE

CHAPITRE 1

AUTOMATES FINIS

§1.1. Automates finis

Un automate fini est le modèle le plus simple d'ordinateur : un digicode, une calculatrice rudimentaire, un programmateur de machine à laver.

Il s'agit d'une machine dont le comportement est modélisé par un ensemble d'*états*, les données — une suite de symboles — qui lui fournies lui faisant passer d'un état à un autre suivant des règles précises. Il y a un état initial ; certains des états sont appelés *finaux* : si la machine s'y trouve lorsque les données ont été utilisées, elle accepte la suite de symbole donnée, sinon, elle la refuse.

On représente les états par des cercles, doublés pour les états finaux ; les transitions sont indiquées par des flèches reliant un état à un autre et dont une étiquette précise quels symboles provoquent cette transition ; l'état initial est le but d'une flèche qui ne vient de nulle part.

Pour donner un exemple, considérons l'automate à trois états de la figure 1. Les symboles que lit l'automate sont 0 et 1 ; L'état initial est l'état E_1, l'automate y reste s'il reçoit un 0 et passe à l'état E_2 avec un 1 ; de l'état E_2, l'automate part à l'état E_3 avec un 0, et reste en E_2 avec un 1 ; enfin, tant un 0 qu'un 1 envoie l'automate de l'état E_3 à l'état E_2. Seul l'état E_2 est un état final. Sous l'effet de la commande 010110, l'automate parcourt

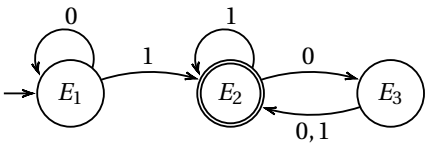


FIGURE 1.

donc successivement les états E_1 (initial), E_1, E_2, E_3, E_2, E_2, E_3. L'état E_3 n'étant pas un état final, le message est refusé. En revanche, la commande 1001 met l'automate dans les états successifs E_1, E_2, E_3, E_2, E_2 et la commande est acceptée.

Passons maintenant à une définition formelle.

DÉFINITION 1.1.1. — *Un* automate fini *est la donnée d'un quintuplet* (E, Σ, τ, e, F)*, où* E, Σ *sont des ensembles finis,* e *est un élément de* E*,* F *une partie de* E *et* τ *une application de* $E \times \Sigma$ *dans* E*.*

Les éléments de E *sont appelés* états*,* e *est l'état initial,* F *est l'ensemble des* états finaux *;* Σ *est appelé* alphabet *et ses éléments* symboles *; la fonction* τ *est la* fonction de transition.

Soit Σ un alphabet. On appelle mot sur Σ une suite finie (éventuellement vide) d'éléments de Σ ; on note Σ^* l'ensemble des mots sur Σ : c'est la réunion de Σ^0 (ensemble réduit à un élément, le mot vide noté ε), Σ^1 (mots d'un symbole), Σ^2, etc.

Soit $\mathscr{A} = (E, \Sigma, \tau, e, F)$ un automate fini et soit $m = (m_1, \ldots, m_n)$ un mot sur Σ. Lorsque l'automate $\mathscr{A}$ reçoit le mot m, il part de l'état initial $e_0 = e$, puis passe successivement dans les états $e_1 = \tau(e_0, m_1)$, $e_2 = \tau(e_1, m_2)$, …, $e_n = \tau(e_{n-1}, m_n)$; si l'état e_n appartient à F, le mot m est accepté, ou reconnu, sinon il est refusé.

Voici un autre automate, toujours à trois états, mais dont l'alphabet est $\{0, 1, 2\}$. Lorsqu'il reçoit un mot, il additionne modulo 3 les symboles de ce mot et se termine

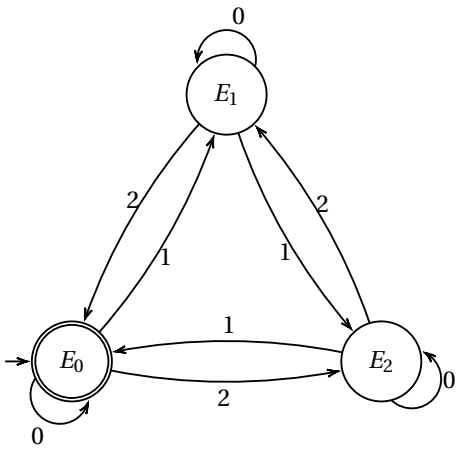


FIGURE 2.

dans l'état E_i si la somme est i ; les mots acceptés sont donc ceux dont la somme des chiffres est multiple de 3.

§1.2. Langages

Un *langage* est un ensemble de mots ; c'est donc une partie de Σ^*.

Si $\mathscr{A}$ est un automate fini, le langage $\mathscr{L}(\mathscr{A})$ est l'ensemble des mots qu'il reconnaît. On dit qu'un langage est *régulier* si c'est l'ensemble des mots reconnus par un automate $\mathscr{A}$.

Une partie de l'étude des automates et des langages consiste ainsi à pouvoir décider si un langage donné est régulier ou non.

Pour prouver qu'un langage est irrégulier, c'est-à-dire pour démontrer qu'il n'existe pas d'automate le reconnaisant, nous aurons besoin de résultats généraux sur les langages réguliers.

Pour prouver qu'il est régulier, une méthode évidente consiste à construire explicitement un automate ; des questions subsidiaires apparaissent alors, comme la construction d'un automate ayant le plus petit nombre d'états possibles. Pour simplifier la construction des automates, il est utile de savoir que les langages réguliers sont stables par un certain nombre d'opérations.

On peut en effet faire plusieurs opérations sur les langages :

– *complémentaire* : si L est un langage, $\bar{L}$ est le langage $\Sigma^* \setminus L$;

– *intersection* : si L_1 et L_2 sont deux langages, $L_1 \cap L_2$ est le langage dont les mots sont ceux qui appartiennent à la fois à L_1 et à L_2 ;

– *réunion* : si L_1 et L_2 sont deux langages, $L_1 \cup L_2$ est le langage dont les mots sont ceux qui appartiennent à L_1 ou à L_2 ;

– *concaténation* : si L_1 et L_2 sont deux langages, les mots du langage $L_1 \circ L_2$ sont ceux de la forme $m_1 m_2$, avec $m_1 \in L_1$ et $m_2 \in L_2$;

– *répétition* : si L est un langage, les mots du langage L^* sont ceux de la forme $m_1 \dots m_n$, avec $n \in \mathbf{N}$ et $m_i \in L$ pour tout i.

PROPOSITION 1.2.1. — *Le langage complémentaire d'un langage régulier est régulier.*

Démonstration. — Soit $\mathscr{A}$ un automate reconnaissant un langage L. Soit $\bar{\mathscr{A}}$ l'automate possédant même alphabet, mêmes états, même fonction de transition, même état initial, mais dont l'ensemble des états terminaux est le complémentaire de celui de $\mathscr{A}$. Un mot est reconnu par $\bar{\mathscr{A}}$ si et seulement si il n'est pas reconnu par $\mathscr{A}$; autrement dit, le langage reconnu par $\bar{\mathscr{A}}$ est le langage $\bar{L}$. □

THÉORÈME 1.2.2. — *Si L_1 et L_2 sont deux langages réguliers, $L_1 \cap L_2$ et $L_1 \cup L_2$ sont encore des langages réguliers.*

Démonstration. — Par hypothèse, nous disposons de deux automates $\mathscr{A}_1 = (E_1, \Sigma, \tau_1, e_1, F_1)$ et $\mathscr{A}_2 = (E_2, \Sigma, \tau_2, e_2, F_2)$ qui reconnaisent respectivement les langages L_1 et L_2, et il s'agit de construire un automate $\mathscr{A}$ qui reconnaisse exactement les mots appartenant à L_1 et à L_2.

Étant donné un mot m dont on veut savoir s'il appartient à $L_1 \cup L_2$, l'idéal serait de pouvoir exécuter simultanément les deux automates avec le mot m en entrée et d'accepter le mot si, à la fin, les deux automates sont dans un état final. À chaque étape, on connaîtrait alors à la fois l'état de chaque automate, ce qui suggère de poser $E = E_1 \times E_2$, $e = (e_1, e_2)$, $F = E_1 \times E_2$ et pour fonction de transition, la fonction $\tau : E \times \Sigma \to E$ définie par

$$\tau((x_1, x_2), a) = (\tau_1(x_1, a), \tau_2(x_2, a)).$$

Cet « automate-produit » reconnaît le langage $L_1 \cap L_2$.

Le langage $L_1 \cup L_2$ est reconnu par une variante de l'automate précédent où l'on modifie l'ensemble des états finaux en posant $F = F_1 \times E_2 \cup E_1 \times F_2$. □

§1.3. Automates non déterministes

Il se trouve que les langages réguliers sont aussi stables par les opérations de concaténation et d'itération. Une preuve comme celle ci-dessus n'est cependant pas très aisée. Une façon élégante — et profonde — de s'en tirer consiste à introduire une deuxième sorte d'automates, appelés *non déterministes*. Dans un tel automate, plusieurs états peuvent découler d'une même entrée. Le diagramme de la figure 3 représente un automate non déterministe ayant quatre états $E_1, \ldots, E_4$, un état initial E_1, un état final E_4 ; son alphabet est $\{0, 1\}$. On voit plusieurs différences avec un automate fini

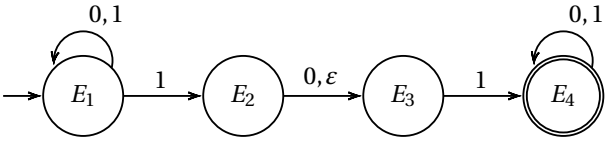


FIGURE 3.

(déterministe) :

– deux flèches étiquetées 1 partent de E_1 ;

– aucune flèche étiquetée 1 ne part de E_2 ; de même, aucune flèche étiquetée 0 ne part de E_3 ;

– en revanche, il part de E_2 une flèche étiquetée ε.

L'automate non déterministe fonctionne de la façon suivante :

– s'il a plusieurs choix face à un symbole, il se démultiplie automatiquement de sorte à les traiter tous en parallèle ;

– si une flèche est étiquetée ε, l'automate se dédouble et un de ses avatars change d'état sans lire de symbole ;

– si une instance de l'automate ne peut plus avancer, parce qu'aucune flèche issue de l'état courant ne comporte le symbole lu, cette instance est supprimée.

L'automate accepte un mot en entrée si et seulement s'il parvient à le lire entièrement et qu'une au moins des instances de l'automate est dans un état final. Sinon, il le refuse.

Du point de vue formel, cette possibilité de supprimer les instances est peu pratique. On la contourne en rajoutant un état « poubelle » dont l'automate ne sort plus et qui ne font pas partie de l'ensemble des états finaux. Dans l'exemple de l'automate de la figure 3, cela revient à rajouter un état P recevant une flèche étiquetée 1 de l'état E_2, une flèche étiquetée 0 de l'état E_3, et deux flèches étiquetées 0 et 1 issues de P lui-même. On aboutit ainsi à l'automate représenté dans la figure 4.

L'intérêt des automates non déterministes est qu'ils sont plus faciles à construire que les automates finis déterministes tout en leur étant pourtant équivalents. Pour en

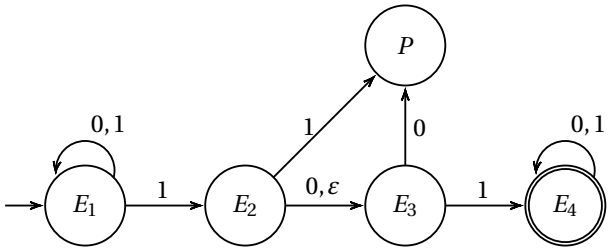


FIGURE 4.

donner une définition formelle, on s'inspire de celle des automates finis en modifiant la fonction de transition : pour chaque état e et chaque symbole a, on doit connaître l'ensemble des états qui sont possibles lorsque l'automate reçoit le symbole a en étant dans l'état e : c'est une partie de l'ensemble des états ; on doit aussi connaître l'ensemble des états possibles sans utiliser de symbole. Si Σ est un alphabet (ne contenant pas le symbole ε), on note Σ' l'alphabet $\Sigma \cup \{\varepsilon\}$.

DÉFINITION 1.3.1. — *Un automate fini non déterministe est un quintuplet* $(E, \Sigma, \tau, e_0, F)$ *où* E, Σ *sont des ensemble finis (états, alphabet),* $e_0 \in E$ *(état initial),* F *est une partie de* E *(états finaux) et* τ *est une application (de transition) de* $E \times \Sigma'$ *dans* $\mathfrak{P}(E)$.

Étant donné un tel automate fini non déterministe, un mot $m = (m_1, \ldots, m_n)$ est reconnu si et seulement si il existe une suite $(e_1, \ldots, e_N)$ d'états et une suite $\mu = (\mu_1, \ldots, \mu_N)$ d'éléments de Σ' telles que :

– si l'on enlève les symboles ε de la suite μ, on obtient le mot m (formellement : il existe une application strictement croissante $f : \{1, \ldots, n\} \to \{1, \ldots, N\}$ telle que $m_i = \mu_{f(i)}$ pour tout i, et $\mu_j = \varepsilon$ si j n'est pas dans l'image de f) ;
– pour tout $i \in \{1, \ldots, N\}$, l'état e_i appartient à $\tau(e_{i-1}, \mu_i)$;
– l'état e_N appartient à F.

THÉORÈME 1.3.2. — *Pour tout automate fini non déterministe, il existe un automate fini qui reconnaît le même langage.*

Démonstration. — Soit $\mathscr{A} = (E, \Sigma, \tau, e_0, F)$ un automate non déterministe. Imaginons son fonctionnement lorsqu'on lui fournit un mot en entrée. À tout instant, un certain nombre d'instances de l'automate $\mathscr{A}$ sont actives ; si on considère que l'automate exécute instantanément les flèches étiquetées ε, ces instances ont alors lu exactement les mêmes caractères du mot en entrée. Par suite, si deux instances sont dans le même état, on peut sans dommage en supprimer l'une des deux : seul compte l'ensemble des états actifs à un instant donné, et cet ensemble est une partie de E.

On introduit alors un automate fini $\widetilde{\mathscr{A}}$ dont l'ensemble des états est $\mathfrak{P}(E)$, l'alphabet est Σ et l'état initial est le singleton $\{e_0\}$. Négligeant pour l'instant les flèches ε, on peut

définir sa fonction de transition par la formule

$$\tilde{\tau}(S,a) = \bigcup_{e\in S} \tau(e,a).$$

En effet, si l'automate $\mathscr{A}'$ est dans l'état S (une partie de E), les états des diverses instances de l'automate $\mathscr{A}$ sont les éléments de S ; à la lecture du symbole a, l'instance de l'automate $\mathscr{A}$ qui est dans un état $e \in S$ se dédouble et se place dans les états appartenant à $\tau(e,a)$; l'ensemble des états des instances de l'automate $\mathscr{A}$ est bien l'ensemble $\tilde{\tau}(S,a)$ indiqué.

Un mot est accepté par l'automate $\mathscr{A}$ si une de ses instances au moins se termine dans un état final. On définit donc les états finaux de $\widetilde{\mathscr{A}}$ comme les parties de E qui contiennent un état final de $\mathscr{A}$ au moins.

Cet automate convient si $\mathscr{A}$ n'utilise pas de transitions ε, c'est-à-dire si $\tau(s,\varepsilon) = \varnothing$ pour tout $s \in E$.

Pour tenir compte des flèches étiquetées ε et de la possibilité qu'a un automate non déterministe de changer d'état sans lire de caractère, il faut modifier cette définition de $\tilde{\tau}$, ainsi que l'état initial de l'automate $\widetilde{\mathscr{A}}$.

Le principe est d'effectuer les transitions ε lors de l'installation dans l'état initial, et après lecture de chaque caractère.

Pour $S \subset E$, posons ainsi $S^\varepsilon = \bigcup_{e\in S} \tau(e,\varepsilon)$ et $S' = S \cup S^\varepsilon \cup S^{\varepsilon\varepsilon} \cup \ldots$. L'ensemble S' est l'ensemble des états accessibles depuis S en n'effectuant que des transitions ε.

On définit alors un automate fini $\mathscr{A}'$: son ensemble d'états est l'ensemble $E' = \mathfrak{P}(E)$, son état initial est l'ensemble $\{e_0\}'$ des états accessibles depuis e_0 à l'aide de transitions ε ; ses états finaux sont les parties S de E qui contiennent un élément de F, c'est-à-dire telles que $S \cap F \neq \varnothing$. Enfin, la fonction de transition τ' est définie par

$$\tau'(S,a) = \tilde{\tau}(S,a)' = \left(\bigcup_{e\in S} \tau(e,a)\right)'.$$

Il est équivalent à l'automate $\mathscr{A}$. □

COROLLAIRE 1.3.3. — *Pour qu'un langage soit régulier, il faut et il suffit qu'il soit reconnu par un automate fini non déterministe.*

COROLLAIRE 1.3.4. — *Si L_1 et L_2 sont des langages réguliers, leur concaténation $L_1 \circ L_2$ est un langage régulier.*

Démonstration. — Il suffit de construire un automate non déterministe reconnaissant le langage $L_1 \circ L_2$. Soit $\mathscr{A}_1 = (E_1, \Sigma, \tau_1, e_1, F_1)$ et $\mathscr{A}_2 = (E_2, \Sigma, \tau_2, e_2, F_2)$ des automates (déterministes ou non, cela n'a pas d'importance) qui reconnaissent respectivement L_1 et L_2. On définit un automate dont l'ensemble des états est $E_1 \cup E_2$; l'état initial est celui de e_1 et les états finaux sont ceux de E_2. On ne change pas les flèches dans E_1, ni dans E_2, en revanche, on rajoute des flèches ε de chaque état final de $\mathscr{A}_1$ à l'état initial de E_2. L'automate ainsi construit reconnaît exactement $L_1 \circ L_2$. □

COROLLAIRE 1.3.5. — *Si L est un langage régulier, le langage L^* est régulier.*

Démonstration. — Soit $\mathscr{A} = (E, \Sigma, \tau, e, F)$ un automate reconnaissant le langage L. On définit un automate $\mathscr{A}^*$ comme suit : ses états sont E auquel on rajoute un état e', son alphabet est Σ, son état initial est e', ses états finaux sont $F \cup \{e'\}$. On modifie aussi la fonction de transition en rajoutant des flèches ε de chaque état final à l'état e. L'automate ainsi construit reconnaît le langage L^*. □

Montrer sur un exemple l'intérêt de rajouter cet état e' (pourquoi ne peut-on pas utiliser e et décréter que e est un état final ?).

§1.4. Lemme de pompage et langages irréguliers

Le théorème suivant fournit une propriété importante des langages réguliers : elle permet de démontrer que de nombreux langages ne sont pas réguliers.

THÉORÈME 1.4.1 (Lemme de pompage). — *Soit L un langage régulier dans un alphabet Σ. Il existe un entier naturel p tel que pour tout mot m de L de longueur au moins p, il existe trois mots $x, y, z \in \Sigma^*$ tels que :*

- *on a $m = xyz$;*
- *le mot y n'est pas vide ;*
- *la longueur du mot xy est inférieure ou égale à p ;*
- *pour tout entier $k \geqslant 0$, le mot xy^kz (obtenu en concaténant le mot x, k fois le mot y et le mot z) appartient à L.*

L'entier p est appelé longueur de pompage.

Démonstration. — Soit $\mathscr{A} = (E, \Sigma, \tau, e, F)$ un automate fini (déterministe) qui reconnaît le langage L. Soit p le nombre d'états de $\mathscr{A}$, c'est-à-dire le cardinal de E. Soit m un mot de longueur au moins p. Notons $m = (m_1, \ldots, m_n)$ et $(e_0, e_1, \ldots, e_n)$ la suite des états de l'automate lorsqu'il lit le mot m ; on a donc $e_0 = e$ et $e_i = \tau(e_{i-1}, m_i)$ pour $i \in \{1, \ldots, n\}$.

Par hypothèse, on a $n \geqslant p$. D'après le principe des tiroirs, il y a parmi les $p+1$ états $e_0, \ldots, e_p$ deux moments i et j tels que $i < j$ et $e_i = e_j$. Soit x le mot $(m_1, \ldots, m_i)$, y le mot $(m_{i+1}, \ldots, m_j)$ et z le mot $(m_{j+1}, \ldots, m_n)$. Les trois premières conditions sont satisfaites puisque la longueur de y est $j - i > 0$ et celle de xy est $j \leqslant p$. Démontrons la dernière. Lorsqu'on fournit le mot xy^kz à l'automate, il commence par lire x et se retrouve dans l'état e_i. Il lit alors k fois le mot y ce qui le ramène à chaque fois dans l'état $e_i = e_j$. Pour finir, il lit z ce qui le fait passer dans les états $e_{j+1}, \ldots, e_n$. L'état e_n est un état terminal, car le mot m appartient au langage L. Par suite, le mot xy^kz est reconnu par $\mathscr{A}$. Il appartient donc aussi au langage L. □

Exemple 1.4.2. — Le langage $L = \{0^n 1^n\,;\, n \geqslant 0\}$, dont les mots sont formés d'une suite de 0 suivie d'autant de 1, n'est pas régulier.

Considérons sinon, par l'absurde, sa longueur de pompage p. Soit m le mot $0^p 1^p$; on peut l'écrire xyz, où y est non vide, xy est de longueur $\leqslant p$, de sorte que pour tout entier $k \geqslant 0$, $xy^k z$ soit dans L, donc de la forme $0^n 1^n$, pour un certain $n \in \mathbf{N}$. Le mot xy est le début de m, et est de longueur au plus p ; il ne contient donc que des 0, si bien qu'il existe des entiers a et $b \geqslant 0$ tels que $x = 0^a$ et $y = 0^b$; par suite, $z = 0^{p-a-b}1^p$. Comme $b > 0$, on constate alors que $xz = 0^{p-b}1^p$ n'appartient pas à L, contradiction.

Exemple 1.4.3. — Le langage L' formé des mots qui ont autant de 0 que de 1 n'est pas régulier.

On observe que le langage L de l'exemple précédent est l'intersection du langage L' et du langage L'' dont les mots sont de la forme 0^*1^*. Le langage L'' est régulier. Si le langage L' était régulier, le langage L le serait aussi.

On peut le démontrer directement à l'aide du lemme de pompage ; supposons par l'absurde que L' soit régulier et soit p sa longueur de pompage. Soit m le mot $0^p 1^p$; écrivons-le xyz, où y n'est pas vide, xy est de longueur au plus p et $xy^k z$ est dans L' pour tout $k \geqslant 0$. Comme dans l'exemple précédent, il existe des entiers $a \geqslant 0$ et $b \geqslant 0$ tels que $x = 0^a$, $y = 0^b$, $z = 0^{p-a-b}1^p$. Alors, $xy = 0^{p-b}1^p$ n'appartient pas à L puisque ce mot a plus de 1 que de 0, contradiction !

§1.5. Expressions régulières et langages réguliers

Les *expressions régulières* sont un moyen très commode de définir des langages. Elles sont aussi très utilisées en informatique, par exemple dans la commande `grep` d'Unix.

DÉFINITION 1.5.1. — *Soit Σ un alphabet. On dit que R est une expression régulière sur Σ si elle est de la forme*

a) $\varnothing$;

b) ε ;

c) a, *pour un élément* $a \in \Sigma$;

d) $(R_1 \cup R_2)$, *où R_1 et R_2 sont des expressions régulières ;*

e) $(R_1 \circ R_2)$, *où R_1 et R_2 sont des expressions régulières ;*

f) (R_1^*), *où R_1 est une expression régulière.*

Pour être formels, on devrait dire que les expressions régulières sont elles-mêmes des mots sur l'alphabet $\Sigma \cup \{\varepsilon, \varnothing, (,), \cup, \circ, ^*\}$ (où l'on suppose que ces derniers symboles n'appartiennent pas à Σ). La définition ci-dessus est correcte car elle définit si R est une expression régulière par récurrence sur la longueur du mot R.

Toujours par récurrence sur la longueur du mot R définissant une expression régulière, on définit le langage représenté par R.

DÉFINITION 1.5.2. — *Soit Σ un alphabet et soit R une expression régulière sur Σ. Le* langage décrit *par R, noté $\mathscr{L}(R)$, est le langage sur l'alphabet Σ défini par récurrence comme suit :*

a) $\mathscr{L}(R) = \varnothing$ *si* $R = \varnothing$;

b) $\mathscr{L}(R) = \{\varepsilon\}$ *si* $R = \varepsilon$;

c) $\mathscr{L}(R) = \{a\}$ *si* $R = a$, *pour* $a \in \Sigma$;

d) $\mathscr{L}(R) = \mathscr{L}(R_1) \cup \mathscr{L}(R_2)$ *(réunion) si* $R = (R_1 \cup R_2)$;

e) $\mathscr{L}(R) = \mathscr{L}(R_1)\mathscr{L}(R_2)$ *(concaténation) si* $R = (R_1 \circ R_2)$;

f) $\mathscr{L}(R) = \mathscr{L}(R_1)^*$ *(répétition) si* $R = (R_1^*)$.

Par exemple $(0 \cup 1) \circ (0^*)$ est une expression régulière, concaténation de l'expression $(0 \cup 1)$ et de l'expression (0^*). Le langage qu'elle décrit est l'ensemble des mots formé d'un 0 ou d'un 1, et suivi d'un nombre quelconque (fini, éventuellement nul) de 0.

L'expression $(0 \cup 1)^*$ est aussi une expression régulière, répétition de l'expression régulière $(0 \cup 1)$. Le langage qu'elle décrit n'est autre que Σ^*.

Pour alléger l'écriture, on omet le symbole $\circ$ de concaténation. On peut aussi omettre les parenthèses ; les opérations sont évaluées dans l'ordre suivant : répétition, concaténation, réunion. L'expression $(0 \cup 1) \circ (0^*)$ est ainsi notée $(0 \cup 1)0^*$. L'expression $(0 \cup 1)^*$ ne doit pas être confondue avec $0 \cup 1^*$ dont le langage décrit est $\{\varepsilon, 0, 1, 11, 111, \ldots\}$.

Toujours pour alléger l'écriture, Σ étant un alphabet fini $\{c_1, \ldots, c_s\}$, on note Σ l'expression régulière $c_1 \cup c_2 \cdots \cup c_s$. Le lange qu'elle décrit est précisément Σ, ensemble des mots de longueur 1.

THÉORÈME 1.5.3. — *Un langage est régulier si et seulement s'il existe une expression régulière qui le décrit.*

Démonstration. — Soit R une expression régulière. Démontrons que le langage $\mathscr{L}(R)$ qu'elle décrit est régulier.

Cela résulte, par récurrence, de ce que les langages $\varnothing$, $\{\varepsilon\}$, $\{a\}$ (pour $a \in \Sigma$) sont réguliers, et de ce que l'ensemble des langages réguliers est stable par les opérations de réunion, concaténation et répétition.

Soit A un automate fini. Démontrons que le langage L reconnu par A peut être décrit par une expression régulière.

La démonstration se fait en deux temps : conversion de l'automate A (sur l'alphabet Σ) en une sorte d'automate non déterministe A' dont les étiquettes sont des expressions régulières ; puis conversion de l'automate A' en expression régulière.

L'automate A' est déduit de A par adjonction d'un état initial et d'un état final et en faisant quelques modifications :

– l'état initial I de A' est simplement muni d'une flèche étiquetée ε vers l'état initial de A ;

– l'état final F de A' est simplement muni d'une flèche étiquetée ε depuis l'état initial de A ;

– si deux états de A sont reliés par plusieurs flèches, celles-ci sont réunies en une seule, dont l'étiquette est l'expression régulière réunion des étiquettes des flèches ;

– on ajoute une flèche étiquetée $\varnothing$ entre les états qui ne sont pas reliés.

On obtient ainsi une sorte d'automate fini, non déterministe, sur l'alphabet (infini) $\mathscr{R}$ des expressions régulières sur Σ, avec deux états particuliers i et f. Appelons un tel automate *automate (non déterministe) à expressions régulières*. Comme la concaténation d'expressions régulières en est une, le langage reconnu par cet automate est un ensemble d'expressions régulières. On dira qu'un tel automate reconnaît le langage L si L est la réunion des langages décrits par les expressions régulières qu'il reconnaît. Ainsi, l'automate construit reconnaît le langage L.

En outre, chaque couple d'états est relié par exactement une flèche — dans la description formelle de l'automate A', cela signifie que l'application de transition $\tau_{A'}$ est telle que pour tout état e et toute expression régulière R, $\tau_{A'}(e,R)$ est de cardinal $\leqslant 1$, et que la réunion de ces parties est égale à l'ensemble des états standard de A'.

Si e et e' sont des états de A', on notera $\delta(e,e')$ l'étiquette de la flèche qui va de e à e' ; c'est l'unique expression régulière R telle que $\tau_{A'}(e,R) = \{e'\}$. On observe aussi que $\delta(f,i) = \varnothing$.

Dans la seconde phase, on va supprimer un par un tous les états autres que i et f (appelons-les états standard), tout préservant et le langage reconnu par l'automate.

Montrons donc comment enlever un état standard, disons e. Pour préserver le langage reconnu par A', nous devons modifier les étiquettes. Pour chaque couple (e_1,e_2) d'états de A', et distincts de e on remplace l'étiquette $\delta_{A'}(e_1,e_2)$ de la flèche reliant e_1 à e_2 dans A' par l'étiquette construite de la façon suivante : Soit R_1, R_2, R_3, R_4 les étiquettes des flèches $e_1 \to e$, $e \to e$, $e \to e_2$ et $e_1 \to e_2$ respectivement. On remplace l'étiquette R_4 de $e_1 \to e_2$ par l'expression régulière

$$(R_1)(R_2)^*(R_3) \cup (R_4).$$

L'automate A'' qu'on obtient reconnaît le même langage que A' et est encore un automate à expressions régulières : chaque couple d'états est relié par exactement une flèche étiquetée par une expression régulière sur Σ, et l'étiquette $\delta_{A''}(f,i)$ est égale à $\varnothing$.

Au bout d'autant d'étapes que l'automate A avait d'états, on obtient un automate non déterministe, sur l'alphabet $\mathscr{R}$, ayant deux états, l'état initial et l'état final. On a $\delta(f,i) = \varnothing$, et l'étiquette $\delta(i,f)$ est une expression régulière qui décrit le langage initial L. □

§1.6. Exercices

Exercice 1.6.1. — Construire des automates finis reconnaissant les langages suivants sur l'alphabet $\{0,1\}$:

a) mots commençant par un 1 et finissant par un 0 ;

b) mots contenant au moins trois 1 ;

c) mots contenant le sous-mot 0101 ;

d) mots de longueur au moins 3 dont le troisième symbole est un 0 ;

e) mots commençant par un 0 et de longueur paire, ou commençant par un 1 et de longueur impaire ;

f) $\{\varepsilon, 0\}$;

g) le langage vide.

Exercice 1.6.2. — Construire des automates finis reconnaissant les langages suivants sur l'alphabet $\{a,b\}$:

a) mots ayant au moins trois a et deux b ;

b) mots ayant exactement deux a et au moins deux b ;

c) mots ayant un nombre pair de a et un ou deux b ;

d) mots ayant un nombre pair de a, chacun d'eux étant suivi par au moins un b ;

e) mots ayant un nombre impair de a et de longueur paire.

Dans chaque cas, le langage est l'intersection de langages plus simples ; on demande de construire d'abord des automates pour ces langages plus simples et de les combiner pour obtenir un automate fini qui reconnaît le langage donné.

Exercice 1.6.3. — Construire des automates non déterministes reconnaissant les langages suivants et ayant le nombre d'états indiqué (l'alphabet est $\{0,1\}$) :

a) mots finissant par 00, avec trois états ;

b) mots contenant le sous-mot 0101, avec cinq états ;

c) mots contenant un nombre pair de 0, ou exactement deux 1, avec six états ;

d) $\{0\}$, avec deux états ;

e) $\{\varepsilon\}$, avec un état.

Exercice 1.6.4. — Démontrer qu'un automate non déterministe peut être converti en un automate équivalent qui n'a qu'un seul état final.

Exercice 1.6.5. — Soit L le langage sur l'alphabet $\{0,1\}$ formé des mots qui ne contiennent pas une paire de 1 séparés par un nombre impair de symboles. Construire un automate non déterministe avec quatre états qui reconnaît le complémentaire de L. Construire alors un automate déterministe à cinq états qui reconnaît L.

Exercice 1.6.6. — Démontrer à l'aide du lemme de pompage que les langages suivants ne sont pas réguliers :

a) $L_1 = \{0^{n^2} ; n \geqslant 0\}$;

b) $L_2 = \{0^{2^n} ; n \geqslant 0\}$;

c) $L_3 = \{0^n 1^n 2^n ; n \geqslant 0\}$;

d) $L_4 = \{0^p ; p \text{ premier}\}$.

Exercice 1.6.7. — Soit n un entier.

a) Démontrer que le langage $L_n = \{0^k ; n \text{ divise } k\}$ est régulier.

b) Démontrer que le langage L'_n sur l'alphabet $\{0,1\}$ formé des développements binaires de multiples de n est régulier.

Exercice 1.6.8. — Soit L et M des langages ; on note L/M le langage formé des mots m tels qu'il existe $m' \in M$ avec $mm' \in L$ (mots tels qu'en leur rajoutant un mot de M, on obtienne un mot de L). Si L est régulier, démontrer que L/M l'est aussi.

Exercice 1.6.9. — Démontrer que le langage L formé des mots qui ont autant d'occurrences du sous-mot 01 que du sous-mot 10 est un langage régulier. (Par exemple, 101 appartient à L, mais pas 1010.)

Exercice 1.6.10. — a) Démontrer que le langage L dont les mots sont de la forme $1^k x$, où $k \geqslant 1$ et où x est un mot sur $\{0,1\}$ comportant au moins k symboles 1 est régulier.

b) Démontrer que le langage L dont les mots sont de la forme $1^k x$, où $k \geqslant 1$ et où x est un mot sur $\{0,1\}$ comportant au plus k symboles 1 n'est pas régulier.

Exercice 1.6.11. — Quelle est la longueur de pompage des langages suivants :

a) mots de la forme 0001^* ;

b) mots de la forme 0^*1^* ;

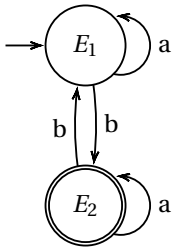
c) $\{\varepsilon\}$;

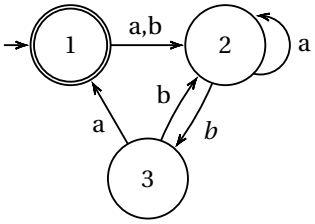
d) mots de la forme $(01)^*$;

e) Σ^*.

Exercice 1.6.12. — Expliciter des expressions régulières qui décrivent les langages de l'exercice 1.6.1.

Exercice 1.6.13. — Déterminer des expressions régulières qui décrivent le langage reconnu par les automates suivants :





Exercice 1.6.14. — Soit Σ un alphabet. Si $w = c_1 \dots c_n$ est un mot sur Σ, on note w' le mot $c_n \dots c_1$ obtenu en écrivant w dans l'autre sens. Soit L un langage sur un alphabet Σ ; soit L' l'ensemble des mots w', pour $w \in L$.

Si L est reconnaissable, démontrer qu'il en est de même de L'.

Exercice 1.6.15 (Théorème de Myhill-Nerode). — Soit L un langage sur un alphabet Σ. On dit que deux mots x et y dans Σ^* sont distingables par L s'il existe un mot $z \in \Sigma^*$ tel qu'exactement un des deux mots xz et yz appartienne à L ; on dit qu'ils sont indistingables sinon.

a) Démontrer que la relation $\sim_L$ définie par « $x \sim_L y$ si x et y sont des mots de Σ^* indistingables par L » est une relation d'équivalence dans Σ^*.

b) On suppose que L est reconnaissable par un automate fini (déterministe) ayant k états. Pour toute partie X de Σ^* de cardinal $> k$, démontrer qu'il existe des mots x et $y \in X$ qui sont indistingables par L. De manière équivalente, démontrer que la relation $\sim_L$ a au plus k classes d'équivalences.

c) On suppose que la relation $\sim_L$ possède exactement k classes d'équivalence ; démontrer que L est reconnaissable par un automate fini ayant k états.

Exercice 1.6.16. — Soit L le langage sur l'alphabet $\{a, b, c\}$ dont les mots sont de la forme $a^i b^j c^k$, avec $i, j, k \geqslant 0$, et $j = k$ si $i = 1$.

a) Démontrer que L n'est pas régulier.

b) Démontrer que L est « pompable »

Exercice 1.6.17. — Soit k un entier $\geqslant 1$ et Σ l'alphabet $\{a, b\}$. Soit L le langage dont les mots possèdent un a en k-ième position en partant de la droite (on a $L = \Sigma^* a \Sigma^{k-1}$).

a) Construire un automate non déterministe ayant $k+1$ états qui reconnaît le langage L.

b) Démontrer qu'un automate déterministe qui reconnaît le langage L possède au moins 2^k états.

CHAPITRE 2

MACHINES DE TURING

§2.1. Le 10^{e} problème de Hilbert

Il s'agit du problème suivant, posé en 1900 par le grand mathématicien David HILBERT à l'occasion du Congrès international des mathématiciens de Paris :

> *Entscheidung der Lösbarkeit einer diophantischen Gleichung.* — Eine diophanthe Gleichung mit irgendwelchen Unbekannten und mit ganzen rationalen Zahlkoefficienten sei vorgelegt : man soll ein Verfahren angeben, nach welchen sich mittels einer endlichen Anzahl von Operationen entscheiden läßt, ob die Gleichung in ganzen rationalen Zahlen lösbar ist. [(1)]

Les équations diophantiennes dont il s'agit sont les équations de la forme

$$P(x, y, z, \dots) = 0,$$

où P est un polynôme à coefficients entiers en des indéterminées $x, y, z, \dots$; on en cherche les solutions en nombres entiers. Plus généralement, on peut s'intéresser à des systèmes de telles équations ; ceux-ci se ramènent en fait à une équation, celle donnée par la somme des carrés des polynômes définissant le système. L'adjectif qualificatif « diophantienne » leur est attribué en hommage au mathématicien Diophante d'Alexandrie (III^{e} siècle av. J.-C.) qui étudia ce type d'équations dans son *Arithmétique,* livre qui eut une grande influence sur la mathématique arabe, puis celle de la Renaissance.

Un exemple ancien et fameux est l'équation de Pythagore $x^2 + y^2 - z^2 = 0$ que l'on peut reformuler ainsi : quels sont les triangles rectangles dont les trois côtés sont de longueur entière ? Par exemple $3^2 + 4^2 = 5^2$ signifie qu'il existe un triangle rectangle de côtés 3 cm, 4 cm, 5 cm. La tablette babylonienne *Plimpton 322* datant d'environ 1800 av. J.-C. donne des listes de nombres qu'on a longtemps interprété comme étant

1. *Décidabilité de la résolubilité d'une équation diophantienne.* — Étant donnée une équation diophantienne à un nombre arbitraire d'inconnues et dont les coefficients sont des nombres entiers rationnels, donner un procédé qui au moyen d'un nombre fini d'opérations permet de décider si l'équation possède une solution en entiers rationnels.

des solutions de cette équation. Cependant, une étude récente laisse penser que ce ne sont que des exercices scolaires. Quoi qu'il en soit, la liste des solutions est décrite au Livre X des *Éléments* d'EUCLIDE (300 av. J.-C.) : si $(x,y,z) \in \mathbf{Z}^3$ est solution, alors il existe $(a,b,d) \in \mathbf{Z}^3$, a et b étant premiers entre eux, tel que $x = d(a-b)^2$, $y = 2abd$, $z = d(a+b)^2$ ou la solution analogue en échangeant x et y.

Un exemple moins ancien mais plus fameux est l'équation de FERMAT $x^n + y^n - z^n = 0$, où n est un entier naturel tel que $n \geqslant 3$. Pierre de Fermat était un magistrat toulousain, féru de mathématiques et de physique — il découvra le calcul différentiel (voir la statue le figurant en « pleine action » qui se trouve au Capitole [2]), des principes d'optique, ainsi que de très beaux théorèmes d'arithmétique. C'est vers 1640, en lisant le passage de l'*Arithmétique* de DIOPHANTE consacré à l'équation de PYTHAGORE (Livre II, problème 8) qu'il inscrivit, dans la marge, l'observation suivante [3] :

OBSERVATIO DOMINI PETRI DE FERMAT.

Cubum autem in duos cubos, aut quadratoquadratum in duos quadratoquadratos & generaliter nullam in infinitum vltra quadratum potestatem in duos eiusdem nominis fas est diuidere cuius rei demonstrationem mirabilem sane detexi. Hanc marginis exiguitas non caperet.

Le destin de cette remarque est bien connu : malgré des progrès permanents pendant 350 ans dûs aux plus grands mathématiciens (FERMAT lui-même, EULER, GAUSS, KUMMER,...), ce n'est qu'en 1995 qu'Andrew WILES (aidé de Richard TAYLOR) apporta la première démonstration de cette proposition ; sa méthode utilise tout l'arsenal des mathématiques découvertes depuis 350 ans et ne tiendrait pas dans ce cours.

Revenons au problème général de HILBERT. Sous l'hypothèse supplémentaire que l'équation n'a qu'une variable, il est assez facile de décider de l'existence de solutions entières. On a en effet le lemme suivant :

LEMME 2.1.1. — *Soit $P = a_0 x^n + a_1 x^{n-1} + \cdots + a_n$ un polynôme de degré n à coefficients complexes. Toute racine complexe z de P vérifie l'inégalité*

$$|z| \leqslant \max\left(1, \frac{|a_1| + \cdots + |a_n|}{|a_0|}\right).$$

Ce lemme fournit une majoration *a priori* des racines complexes de P, donc aussi de ses racines entières. Pour en faire la liste, il ne reste ainsi plus qu'à calculer $P(0), P(\pm 1), \ldots, P(m)$, où m est le plus grand entier vérifiant l'inégalité du lemme.

2. *http://www.jacobins.mairie-toulouse.fr/patrhist/edifices/textes/capitole/Nord_22.htm*

3. En français : « Il n'est pas possible de diviser un cube en la somme deux cubes, un bicarré en deux bicarrés, et plus généralement une puissance supérieure en deux puissances identiques : j'ai trouvé une démonstration merveilleuse de cette proposition. Mais la marge est trop étroite pour la contenir. »

Démonstration. — Si $|z| \leqslant 1$, cette inégalité est satisfaite. Supposons $|z| > 1$. De l'égalité $P(z) = 0$, c'est-à-dire

$$a_0 z^n = -a_1 z^{n-1} - \cdots - a_n,$$

on déduit que

$$|a_0||z|^n \leqslant |a_1||z|^{n-1} + \cdots + |a_n| \leqslant (|a_1| + \cdots + |a_n|)\,|z|^{n-1}$$

où l'on a utilisé l'inégalité $|z|^k \leqslant |z|^{n-1}$ pour $k \in \{0, \ldots, n-1\}$, laquelle découle de l'hypothèse $|z| \geqslant 1$. En divisant les deux membres par $|a_0||z|^{n-1}$, on en déduit l'inégalité voulue. □

En deux variables, la situation est déjà beaucoup plus subtile. Il n'y a en effet pas de majoration des racines complexes d'une équation de la forme $P(x, y) = 0$ — prendre par exemple $P(x, y) = xy - 1$. Un théorème d'Alan BAKER (1968), qui lui valut une médaille Fields en 1970, fournit cependant une majoration des solutions entières (x, y) de certaines équation polynomiale en deux variables, permettant donc de répondre par l'affirmative au problème de HILBERT dans ces cas particuliers.

En 1970, à la suite d'importants travaux de Julia ROBINSON, Martin DAVIS et Hilary PUTNAM, Yuri MATIYASEVICH démontra finalement que le 10e problème de Hilbert n'admet pas de solution : il n'existe pas d'algorithme permettant de décider si une équation diophantienne arbitraire possède une solution en nombres entiers ou non.

En 1928, HILBERT généralisa deux de ses problèmes de 1900 et posa l'*Entscheidungsproblem* ; il s'agissait de démontrer :

– que toute assertion mathématique vraie peut être démontrée (*complétude* des mathématiques) ;

– que seules des assertions mathématiques vraies peuvent être démontrées (*consistance* des mathématiques) ;

– que les mathématiques sont décidables, qu'il existe un algorithme permettant de décider de la vérité (ou non) de toute assertion mathématique.

À la surprise de HILBERT lui-même, Kurt GÖDEL démontra en 1930 que les deux premiers problèmes ont une réponse négative.

Le troisième problème nécessite la définition mathématique du mot « algorithme ». Cela semble ne mériter aucun commentaire aujourd'hui, alors que les ordinateurs participent à peu près de chaque activité humaine. Ce n'était pas le cas en 1900, pas plus en 1928, et ce n'est qu'à la fin des années 30 que différentes axiomatiques ont été proposées, dont le λ-calcul d'Alonzo CHURCH, les fonctions récursives de Kurt GÖDEL, et les *machines* d'Alan TURING.

Toutes ces axiomatiques se sont d'ailleurs révélées équivalentes. La *thèse de Church–Turing* est qu'elles représentent convenablement la notion informelle d'algorithme, c'est-à-dire de procédé calculatoire qui peut effectivement être mené et conduit à la solution d'un problème. Autrement dit, elle postule l'équivalence entre deux notions :

l'une, de nature intuitive, est qu'un problème donné possède une solution algorithmique, l'autre, mathématique, est que ce problème peut être résolu par une machine de Turing.

La réponse négative au problème de la décision fut apportée indépendamment par CHURCH et TURING en 1936–37. Le théorème de MATIYASEVICH en est une confirmation.

§2.2. Machines de Turing

La machine de Turing est le modèle d'ordinateur inventé par Alan TURING en 1936. En fait, par la mémoire infinie dont elle dispose, une telle machine modélise plus le calcul qu'un calculateur. D'apparence très rudimentaire, elle est en fait aussi performante qu'un ordinateur contemporain, au sens où elle peut effectuer les mêmes calculs.

Dans une machine de Turing, il y a trois parties :

– un « ruban », suite infinie de cases susceptibles de contenir un symbole de l'alphabet, ou d'être blanches ; c'est sa mémoire. Au début, ce ruban contient le mot fourni en entrée à la machine, suivi de caractères blancs ;

– une « tête de lecture/écriture », qui à tout moment est placée sur une des cases du ruban, est capable de lire le symbole sur la case, et d'en écrire un ;

– un « programme », automate possédant un nombre fini d'états. En fonction du symbole lu par la tête de lecture/écriture et de l'état dans lequel est l'automate, ce programme écrit un symbole dans la case de la tête de lecture/écriture, change d'état, et décale la tête de lecture d'une case vers la droite ou vers la gauche.

Le programme a deux états particuliers, *accepte* et *refuse* dont l'effet est immédiat et aboutit à l'arrêt de la machine. Le mot entré initialement est reconnu par la machine de Turing si la machine s'arrête dans l'état *accepte*. Il est aussi possible que la machine s'arrête dans l'état *refuse* mais rien ne garantit que la machine s'arrête ; dans ce cas, l'utilisateur ne saura jamais à quoi s'en tenir...

DÉFINITION 2.2.1. — *Une machine de Turing est un septuplet* $\mathscr{M} = (E, \Sigma, \Gamma, \tau, e, e_a, e_r)$, *où* E, Σ *et* Γ *sont des ensembles finis (états, alphabet d'entrée et alphabet du ruban),* τ *est une application de* $Q \times \Gamma$ *dans* $Q \times \Gamma \times \{L, R\}$ *est la fonction de transition,* e, e_a *et* e_r *sont trois états (état initial, état accepte et état refuse), avec* $e_a \neq e_r$.

On exige que l'alphabet Σ *soit contenu dans l'alphabet* Γ *et que* $\Gamma \setminus \Sigma$ *contienne un symbole particulier, noté* _.

Une telle machine calcule comme suit. Son ruban est une suite (infinie) R d'éléments de Γ (indexée par $\mathbf{N}$), la position t de la tête de lecture est un entier $t \in \mathbf{N}$. Initialement, on a $t = 0$, la suite R est de la forme $(c_1, \ldots, c_n, \_, \_, \ldots)$, où $(c_1, \ldots, c_n)$ est le mot fourni à la machine, et la machine est dans l'état e.

A chaque pas, la machine, supposée dans l'état q, calcule $\tau(q, R_t) = (q', c, d)$, où $q' \in Q$, $c \in \Gamma$ et $d \in \{L, R\}$ puis :

– remplace R_t par c (écriture)

– remplace t par $t+1$ si $d = R$ et par $\max(t-1, 0)$ si $d = L$ — elle ne se déplace pas à gauche si elle ne peut pas (déplacement)

– passe dans l'état q'. Si $q' = e_a$ ou $q' = e_r$, elle accepte ou refuse respectivement, sinon elle continue (changement d'état).

Exemple 2.2.2. — Soit $\mathscr{A} = (E, \Sigma, \tau, e_0, F)$ un automate fini. Nous allons construire une machine de Turing $\mathscr{T}$ qui accepte un mot s'il est reconnu par $\mathscr{A}$ et le refuse sinon. Le principe est simple : lire un caractère sur le ruban ; si c'est un caratère de l'alphabet Σ, changer d'état comme le ferait $\mathscr{A}$, conformément à τ, puis décaler la tête vers la droite ; si c'est un caractère blanc, passer dans l'état « accepte » si l'on est dans un des états finaux de $\mathscr{A}$, et dans l'état « refuse » sinon.

Formellement, l'alphabet du ruban de $\mathscr{T}$ est $\Sigma \cup \{\_\}$; ses états sont ceux de $\mathscr{A}$ auquel on adjoint les états e_a et e_r (« accepte » et « refuse ») ; l'état initial de $\mathscr{T}$ est le même que celui de $\mathscr{A}$. La fonction de transition $\tilde{\tau}$ de $\mathscr{T}$ est définie par $\tilde{\tau}(e, \alpha) = (\tau(e, \alpha), \alpha, R)$, $\tilde{\tau}(e, \_) = (e_a, \_, L)$ si $e \in F$, et $\tilde{\tau}(e, \_) = (e_r, \_, L)$ sinon.

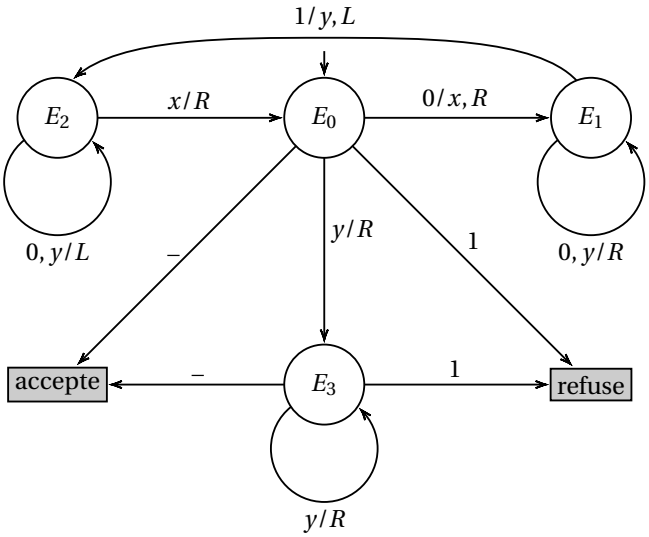
Cet exemple montre que les machines de Turing sont plus puissantes que les automates finis. D'après le suivant, elles le sont strictement plus.

Exemple 2.2.3. — Donnons un exemple de machine de Turing sur l'alphabet $\Sigma = \{0, 1\}$ qui accepte tous les mots de la forme $0^n 1^n$, où n est un entier naturel, et refuse tous les autres. L'alphabet du ruban contiendra deux autres symboles x et y. Le principe consiste à lire le premier caractère ; à accepter s'il est blanc, et à refuser si c'est un 1. Sinon, c'est un 0. Boucle : [on le marque d'un x, et on va à droite jusqu'à trouver un 1, et en refusant si on trouve au passage autre chose que des 0 (ou des y), ou si on n'en trouve pas. On marque le 1 trouvé d'un y et on revient en arrière jusqu'au premier x, et on va à droite. Si c'est un y (on a lu tous les 0), on vérifie que la fin du ruban n'est formée que de y puis un blanc ; sinon, c'est un 0.]

Détaillons sur un exemple ce qui se passe. Cela permet au passage d'introduire une notation pour décrire l'état d'une machine de Turing. Le principe consiste à écrire la chaîne de caractères représentée par le ruban, en insérant l'état de la machine juste *avant* le caractère que la tête peut lire.

Si l'on fournit la chaîne 000111 à la machine, son fonctionnement est alors le suivant :

⓪000111	x00①111	②x00y11	xx0①y11
x①00111	x0②0y11	x⓪00y11	xx0y①11
x0①0111	x②00y11	xx①0y11	xx0②yy1



xx②0yy1	xxxy①y1	xx②xyyy	xxxyyy③
x②x0yy1	xxxyy①1	xxx⓪yyy	accepte
xx⓪0yy1	xxxy②yy	xxxy③yy	
xxx①yy1	xxx②yyy	xxxyy③y	

Exemple 2.2.4. — Utiliser un alphabet étendu, dans lequel, par exemple chaque symbole peut être « marqué » est une technique importante de programmation. Comment demander à la machine de revenir au début du ruban ? Rappelons qu'une machine de Turing à qui on demande d'aller à gauche, alors qu'elle est au début du ruban, ne bouge pas. Le principe consiste à marquer le caractère sous la tête de lecture avant d'aller à gauche : si le caractère lu est encore marqué, c'est qu'on n'a pas bougé et qu'on était au début de la tête. Dans le cas contraire, on retourne à droite, on supprime la marque et on fait un pas vers la gauche.

Une autre technique consiste à marquer le premier caractère du ruban dès le début du calcul et à détecter cette marque.

§2.3. Variantes

On peut définir de nombreuses variantes des machines de Turing. Nous en présentons trois, par degré de sophistication croissant. En fait, chacune de ces variantes est *équivalente* à la définition initiale, au sens où l'ensemble des mots accepté par une machine modifiée est égal à l'ensemble des mots accepté par une machine de Turing adéquate, et inversement.

A. Machine dont la tête peut rester fixe

Pour une telle machine, le troisième paramètre de la fonction de transition (celui de direction) est à valeurs dans $\{L, R, S\}$, où S signifie que la tête ne bouge pas. *Une telle machine est équivalente à une machine de Turing.* En effet, il suffit de remplacer chaque transition S par une transition vers un état intermédiaire et le déplacement vers la droite, suivi d'une transition vers l'état voulu en déplaçant la tête vers la gauche.

B. Machines à plusieurs rubans

Une machine de Turing à plusieurs rubans est un automate qui dispose, comme son nom l'indique, d'un nombre fini, k, de rubans et d'une tête de lecture/écriture par ruban. À chaque étape, chacune des têtes peut, ou non, se déplacer vers la droite ou vers la gauche.

Du point de vue formel, cela remplace la fonction de transition par une fonction

$$\tau : E \times \Gamma^k \to E \times \Gamma^k \times \{L, R, S\}^k$$

décrivant le changement d'état en fonction des valeurs lues par la tête de lecture de chaque ruban, ce qu'écrivent ces têtes de lecture et leurs déplacements (gauche, droite, immobile).

Une telle machine est aussi équivalente à une machine de Turing.

Le principe consiste à simuler les k rubans à l'aide d'un seul qui contient le contenu des rubans, caractère par caractère, en signifiant d'un # que l'on revient au premier ruban. Si le premier ruban contenait abc, le second $01a$, le troisième $ab\_$, le ruban unique contient $\#a0a\#b10\#ca\_\#$ suivi de $k+1$ caractères # marquant la fin du ruban.

Par ailleurs, on utilise un alphabet étendu de sorte à marquer les caractères correspondant à la position de la tête de lecture. Si la première tête était sur a, la seconde aussi et la troisième sur le caractère _, le ruban contiendra en fait $\#\bar{a}0a\#b10\#c\bar{a}\bar{\_}\#$.

Pour lire le caractère de la i-ième tête, la méthode est la suivante : revenir au début du ruban (aller à gauche jusqu'à un #, puis encore une fois : si on lit un # c'est qu'on est au début ; sinon, recommencer) puis à parcourir i caractères à droite ; si le symbole est marqué, on y est, sinon, on va jusqu'au # suivant, puis i caractères encore.

Pour écrire, le mécanisme est le même.

Pour déplacer une tête vers la droite ou la gauche, on démarque le symbole lu par la tête, on fait le déplacement, puis on marque le nouveau symbole.

Il faut faire attention au début et à la fin du ruban. On a déjà expliqué comment traiter le début du ruban. Pour aller à gauche, il faut aller à gauche jusqu'à rencontrer un symbole #, puis encore une fois jusqu'au # suivant, puis aller à droite de i symboles.

Pour aller à droite, on va à droite jusqu'au premier # rencontré, puis on va à droite i fois. Si le symbole lu est un #, c'est qu'on était au bout du ruban : il faut alors reprendre les k symboles # indiquant la fin de la bande, les remplacer par _ et remettre les # après. On peut alors se mettre à la position voulue.

C. Machines de Turing non déterministes

On introduit la notion de machine de Turing non déterministe en modifiant la définition, de façon analogue à ce qu'on avait fait pour les automates finis. Une telle machine dispose d'un ruban infini, d'une tête de lecture, d'un ensemble d'états E, mais sa fonction de transition τ est une application $E \times \Gamma \to \mathfrak{P}(E \times \Gamma \times \{L, R\})$: l'image $\tau(e, a)$ d'un couple état–symbole (e, a) est l'ensemble des triplets état–symbole écrit–direction (q, s, δ) possibles lorsque la machine, dans l'état e, lit le symbole a. Par convention, $\tau(e, a) = \varnothing$ si $e = e_\mathrm{r}$ ou $e = e_\mathrm{a}$ — cela forcera la machine à s'arrêter si elle tombe dans un de ces deux états.

Comment calcule une telle machine ? Son ruban est une suite (infinie) $R \in \Sigma^{\mathbf{N}^*}$ d'éléments de l'alphabet Γ, la position de la tête est un entier $t \in \mathbf{N}^*$. Initialement, R est de la forme $(c_1, \ldots, c_n, \_, \ldots)$ et la machine est dans l'état initial e.

Le mot $(c_1, \ldots, c_n)$ est accepté par la machine non déterministe s'il existe une suite d'états $(e_0, \ldots, e_N)$, une suite $(R_0, \ldots, R_N)$ de « rubans », une suite $(t_0, \ldots, t_N)$ de positions de la tête, vérifiant les contraintes suivantes :

- (état initial) $e_0 = e$, $t_0 = 1$, $R_0 = (c_1, \ldots, c_n, \_, \ldots)$;
- (état final) $e_N = e_\mathrm{a}$;
- (transitions) pour tout $i \in \{0, \ldots, N-1\}$, il existe $(q, s, \delta) \in \tau(e_i, R_{i,t_i})$ tel que $e_{i+1} = q$, $R_{i+1,t_i} = s$, $R_{i+1,t} = R_{i,t}$ si $t \neq s$, $t_{i+1} = t_i + 1$ si $\delta = R$, $t_{i+1} = \max(0, t_i - 1)$ si $\delta = L$.

Une machine de Turing non déterministe est équivalente à une machine classique.

Pour démontrer cette assertion, nous devons, à partir d'une machine non déterministe, construire une machine déterministe qui reconnaît et refuse exactement le même langage.

À chaque étape, la machine a un certain nombre de choix ; le principe consistera à les simuler tous en les plaçant comme sur un arbre. Cependant, comme une machine de Turing peut ne pas s'arrêter, il faut prendre garde à la façon dont on explore cet arbre de sorte à simuler tous ces choix. En effet, si l'on commence par prendre systématiquement le premier, on risque de s'enferrer dans une branche du calcul de la machine de Turing qui ne s'arrête pas et l'on ne saura jamais ce qu'il en est des autres branches. Pour cela, on explore l'arbre, non pas en profondeur, mais en largeur : on teste toutes les possibilités $((e_0, \ldots, e_N), (R_0, \ldots, R_N), (t_0, \ldots, t_N))$ de longueur $N = 1$, puis toutes celles de longueur 2, etc.

L'idée consiste à utiliser trois rubans : le premier ne sera jamais altéré et contient le ruban initial ; le second contient la liste des choix qu'il faut faire, le troisième, le ruban de calcul, contient l'état qu'aurait le ruban compte tenu des choix du second ruban.

Pour tout couple $(e, a) \in E \times \Gamma$, on fixe une fois pour toutes un ordre sur l'ensemble $\tau(e, a)$. Le ruban de choix contient un mot de la forme $(n_1, \ldots, n_N, \_, \ldots)$, où $n_1, \ldots, n_N$ sont des entiers naturels tels que $1 \leqslant n_i \leqslant m$, où m est le plus grand cardinal des ensembles $\tau(e, a)$. Cela signifie que l'on fait le calcul en choisissant à l'étape i, le n_i-ième

élément de l'ensemble $\tau(e_{i-1}, a_{i-1})$, ou l'arrêter si n_i est strictement supérieur au cardinal de cet ensemble. Si ce calcul aboutit à un état e_a, la machine accepte le mot initial. Sinon, on écrit sur le ruban de choix le mot de choix suivant par la méthode suivante : on va au dernier symbole différent de _ ; c'est un entier entre 1 et m. S'il est $< m$, on l'augmente de 1 ; sinon, on le remplace par 1 et on recule d'une case qu'on augmente de 1, etc. Si l'on tente de dépasser le début du ruban, c'est que le mot était $(m, \ldots, m, \_, \ldots)$ et qu'on a exploré tous les choix possibles pour un calcul en N étapes. Par l'algorithme ci-dessus, le mot de choix est devenu $(1, \ldots, 1, \_, \ldots)$; on remplace le premier _ par un 1, et on réexplore le calcul en $N+1$ étapes.

S'il existe une suite d'états aboutissant à l'état e_a pour la machine non déterministe initiale, on voit que la machine construite finira par le détecter et le mot sera accepté. Sinon, cette machine calcule indéfiniment.

§2.4. Langages reconnaissables

DÉFINITION 2.4.1. — *Soit Σ un alphabet. Le langage $\mathscr{L}(T)$ reconnu par une machine de Turing T d'alphabet Σ est l'ensemble des mots de Σ^* qui sont acceptés par T.*

Deux machines T et T' sont équivalentes si elles reconnaissent le même langage, c'est-à-dire si $\mathscr{L}(T) = \mathscr{L}(T')$.

On dit qu'un langage $L \subset \Sigma^*$ est reconnaissable s'il existe une machine de Turing T tel que $L = \mathscr{L}(T)$.

Les variantes de machines de Turing introduites dans le paragraphe précédent donnent lieu à la même notion de langage reconnaissable.

PROPOSITION 2.4.2. — *Les langages réguliers sont reconnaissables.*

Démonstration. — Il faut simuler un automate fini à l'aide d'une machine de Turing. Il suffit de prendre les mêmes états (plus les états e_a et e_r) et de déplacer la tête vers la droite après chaque lecture, sans modifier le symbole sous la tête. Lorsque le caractère lu est _, c'est que l'on a parcouru tout le mot, on passe dans l'état e_a si l'on était dans un état terminal, et dans l'état e_r sinon. □

PROPOSITION 2.4.3. — *La réunion, l'intersection, la concaténation, la répétition de langages reconnaissables est un langage reconnaissable.*

Démonstration. — *Réunion et intersection.* Soit L et L' deux langages reconnus par des machines de Turing T et T'. Pour reconnaître le langage $L \cup L'$, on construit une machine de Turing à deux rubans dont les états sont les couples formé d'un état de T et d'un état de T'. La machine commence par recopier le premier ruban (ruban d'entrée) sur le second, de sorte que chacune des deux machines simulées disposent de la même entrée. Elle simule ensuite le fonctionnement des deux machines en effectuant sur le

premier ruban ce qu'aurait fait T et sur le second ce qu'aurait fait T'. Les états spéciaux e_a et e_r sont traités à part, comme si la fonction de transition associait à (e_r, σ) le triplet (e_r, S, σ) signifiant « ne pas changer d'état, ne pas bouger la tête, ne rien écrire », bref, « ne rien faire ! ». En revanche, si l'une des machines est dans l'état e_a, la nouvelle machine accepte le mot initial de sorte que le langage reconnu est bien la réunion $L \cup L'$ de ceux reconnus par T et T'.

Pour reconnaître le langage $L \cap L'$, la nouvelle machine est analogue mais attend que les deux machines simulées soient dans l'état e_a pour accepter le mot.

Comme dans le cas des automates finis, la stabilité de l'ensemble des langages reconnaissables par *concaténation* et *répétition* fait usage de non-déterminisme. Commençons par la concaténation. On utilise en fait une machine non déterministe à trois rubans : si le ruban initial contient un mot w, on le découpe en deux morceaux $w_1 w_2$ de façon non déterministe, on écrit w_1 sur le deuxième ruban, et w_2 sur le troisième. On exécute la machine T sur le mot w_1, en même temps que la machine T' sur le mot w_2, chacune des machines restant dans leur état e_a ou e_r s'ils se produisent. La machine construite accepte dès que les machines T et T' sont toutes deux dans l'état e_a, elle refuse si elles sont toutes deux dans l'état e_r.

Pour prouver que la répétition du langage reconnaissable L est encore reconnaissable, on coupe un mot w en mots $w_1 \dots w_n$, de façon non déterministe, et on exécute la machine T simultanément sur chacun des mots w_i. □

Le lecteur observera que nous n'avons rien dit du *complémentaire* d'un langage reconnaissable. La démonstration qui fonctionnait pour les automates finis ne marche plus : si la machine de Turing T reconnaît le langage L, on peut échanger les états e_a et e_r mais la nouvelle machine T' n'acceptera pas les mots qui conduisent T à calculer indéfiniment. De fait, nous verrons qu'il existe des langages reconnaissables dont le complémentaire ne l'est pas.

Cela conduira à la notion importante de langage décidable.

Terminons ce paragraphe en discutant de l'existence de langages non reconnaissables. Plus précisément, l'ensemble des langages reconnaissables est dénombrable puisque l'ensemble des machines de Turing est lui-même dénombrable. En revanche, comme l'alphabet Σ n'est pas vide, l'ensemble Σ^* des mots sur Σ est infini dénombrable et l'ensemble des parties de Σ^* a donc la puissance du continu (est équipotent à $\mathbf{R}$), donc n'est pas dénombrable. Rappelons la démonstration de ce résultat de théorie des ensembles.

LEMME 2.4.4. — *Soit A un ensemble ; il n'existe pas de bijection entre A et $\mathfrak{P}(A)$. Il n'existe même pas de surjection de A sur $\mathfrak{P}(A)$.*

Démonstration. — Considérons, par l'absurde, une telle surjection $T\colon A \to \mathfrak{P}(A)$. Soit B la partie de A, ensemble des $a \in A$ tels que $a \notin T(a)$. Puisque T est surjective, il existe $b \in A$ tel que $B = T(b)$.

La contradiction survient lorsque l'on étudie si b appartient à B ou non. En effet, si $b \in B$, on a $b \notin T(b)$, c'est-à-dire $b \notin B$; inversement, si b n'appartient pas à B, $b \in T(b)$, c'est-à-dire $b \in B$. Cette contradiction prouve que T n'existe pas. □

Cet argument ne donne cependant pas de langage explicite qui ne soit pas reconnaissable. En voici un. Soit Σ un ensemble fini, non vide. Soit $(w_1, w_2, \dots)$ la liste des mots de Σ^*, énumérés d'abord par ordre croissant de longueur, puis suivant l'ordre lexicographique. De même, soit $(M_1, M_2, \dots)$ la liste des machines de Turing, énumérées suivant la longueur puis l'ordre lexicographique du mot qui décrit le septuplet correspondant $(E, \Sigma, \Gamma, \dots)$ dans un alphabet convenable.

PROPOSITION 2.4.5. — *Soit L le langage formé des mots w pour lesquels il existe un entier i tel que $w = w_i$ et que la machine M_i n'accepte pas le mot w_i. Le langage L n'est pas reconnaissable.*

Démonstration. — C'est une variante de l'argument précédent. Soit, par l'absurde, M une machine de Turing qui reconnaît L et soit i un entier tel que $M = M_i$.

La contradiction est obtenue en analysant si le mot w_i est reconnnu, ou non, par la machine M_i. Si w_i est accepté par M_i, alors $w_i \notin L$, donc w_i n'est pas reconnu par M, ce qui contredit l'hypothèse que $M_i = M$ reconnaît w_i. À l'inverse, si w_i n'est pas accepté par M_i, la définition du langage L entraîne que $w_i \in L$, donc w_i est accepté par M, ce qui contredit l'égalité $M = M_i$. □

§2.5. Langages décidables

DÉFINITION 2.5.1. — *On dit qu'une machine de Turing est un* décideur *si elle s'arrête dans l'un des états e_a et e_r pour toute entrée qui lui est fournie.*

On dit qu'une machine de Turing non déterministe est un décideur *si chacune des branches du calcul s'arrête.*

On dit qu'un langage est décidable *si c'est le langage reconnu par une machine de Turing qui est un décideur.*

Les résultats que nous avons expliqués concernant les langages reconnaissables s'étendent aux langages décidables.

PROPOSITION 2.5.2. — *Un langage est décidable si et seulement s'il est reconnu par une machine de Turing non déterministe qui est un décideur.*

Démonstration. — Reprenons la démonstration de l'équivalence des machines de Turing déterministes et non déterministes et l'appliquant à une machine de Turing non

déterministe T qui décide le langage L. La machine de Turing (déterministe) obtenue reconnaît le langage L. Il suffit de vérifier que c'est un décideur. Telle que nous l'avons décrite, ce n'est pas tout à fait vrai car nous avons été imprécis et l'avons éventuellement laissée calculer indéfiniment.

Faisons donc la modification suivante, qui ne change rien à ce qu'elle reconnaît : arrêter le calcul si toutes les branches ont été explorées et ont abouti à l'état e_r.

Considérons l'arbre d'exploration : il a pour racine l'état initial (état, ruban, position) de la machine de Turing et ses feuilles sont des états de la machine de Turing ; d'une feuille sont issues tous les états que permet la fonction de transition. Les branches du calcul non déterministe sont précisément des branches de l'arbre issues de la racine ; ils s'arrêtent lorsque la branche s'arrête, soit qu'il n'y a pas d'issue, soit que l'on aboutit sur l'un des états e_r ou e_a.

Pour démontrer que l'on a bien construit un décideur, nous devons démontrer que si, pour un mot donné en entrée toutes les branches du calcul non déterministe de T s'arrêtent, cet arbre est fini. La proposition découle donc du le lemme de théorie des graphes suivant. □

LEMME 2.5.3. — *Si chaque sommet d'un arbre n'a qu'un nombre fini de branches, et si toutes les branches sont finies, alors l'arbre n'a qu'un nombre fini de sommets.*

Démonstration. — Soit A un arbre, soit α_0 sa racine. Supposons qu'il ait une infinité de sommets. Comme α_0 n'a qu'un nombre fini de fils, par hypothèse, le sous-arbre issu d'au moins un d'entre eux, disons α_1, possède une infinité de sommets. Reprenons l'argument à partir de ce fils ; on trouve un sommet α_2 issu de α_1 tel que le sous-arbre issu de α_2 soit infini. Continuons ainsi par récurrence. On constate alors que la suite $(\alpha_0, \alpha_1, \alpha_2, \ldots)$ de sommets est une branche infinie de l'arbre A, ce qui est une contradiction. □

PROPOSITION 2.5.4. — *La réunion, l'intersection, le complémentaire, la concaténation, la répétition de langages décidables est un langage décidable.*

Démonstration. — Traitons le cas du complémentaire d'un langage L décidé par une machine de Turing T : ainsi qu'on l'a sous-entendu avant de définir les langages décidables, il suffit juste d'échanger les états e_a et e_r de la machine T !

La démonstration des autres assertions de la proposition est analogue à celle faite pour les langages reconnaissables. □

PROPOSITION 2.5.5. — *Un langage L est décidable si et seulement s'il est reconnaissable, ainsi que son complémentaire.*

Démonstration. — Si L est décidable, il est reconnaissable ; son complémentaire est aussi décidable, donc reconnaissable.

Supposons maintenant que L soit reconnaissable ainsi que son complémentaire. Soit T une machine de Turing qui reconnaît L et T' une machine de Turing qui reconnaît $\bar{L}$. On construit une machine de Turing qui décide le langage L de la façon suivante : c'est une machine a deux rubans qui simule T sur le premier ruban et T' sur le second. Si T accepte le mot w entré, on a $w \in \mathscr{L}(T) = L$, donc la machine accepte ; si T' accepte le mot w entré, c'est que $w \in \mathscr{L}(T') = \complement L$, donc $w \notin L$ et la machine refuse. Par définition d'un langage reconnaissable, l'une des deux alternatives finit par se produire, donc L est décidable. □

Nous avons déjà démontré qu'il existe des langages qui ne sont pas reconnaissables. La question de l'existence de langages reconnaissables qui ne sont pas décidables est plus subtile et fera l'objet du paragraphe suivant.

Terminons ce paragraphe-ci en démontrant que la théorie des automates finis est décidable. Pour cela, il faut préciser le langage dont on va décider. Rappelons qu'un automate fini est défini comme un quintuplet (E, Σ, τ, e, F), où E est l'ensemble fini des états, Σ l'alphabet, $\tau : E \times \Sigma \to E$ la fonction de transition, $e \in E$ l'état initial et F l'état final. Les états et les symboles de l'alphabet peuvent être identifiés à des entiers, eux-mêmes codés par exemple en binaire ; seul compte alors le nombre d'états et le nombre de symboles. L'application de transition se résume alors en la liste des valeurs $\tau(e, \sigma)$, de même que F est codé la liste des états terminaux. On peut représenter ainsi décrire de façon systématique un automate fini sous la forme d'une suite de caractères $0, 1$ et d'un (ou plusieurs) séparateurs, disons le symbole #, de sorte à indiquer $\mathrm{Card}(E)$, $\mathrm{Card}(\Sigma)$, la liste des codes des $\tau(e, \sigma)$, pour $(e, \sigma) \in E \times \Sigma$ ordonnés dans l'ordre lexicographique, le numéro de l'état initial e et la liste des états terminaux. De même, le mot donné à l'automate est alors codé comme la suite de ses symboles, séparés par le symbole #. Si $A = (E, \Sigma, \tau, e, F)$ est un automate fini et $w \in \Sigma^*$, on notera $\langle A, w \rangle$ le code ainsi produit permettant de décrire A et le mot w en entrée.

PROPOSITION 2.5.6. — *Le langage*

$$A_{\mathrm{AF}} = \{\langle A, w \rangle \,;\, w \textit{ est un mot accepté par l'automate } A\}$$

est décidable.

Démonstration. — La preuve est aisée à concevoir si on a quelque habitude de la programmation. Elle consiste à construire une machine de Turing qui simule l'automate A avec le mot w en entrée et accepte $\langle A, w \rangle$ si l'automate aurait accepté, et à refuser sinon. La machine commence par examiner le mot entré et vérifie s'il correspond bien au codage d'un automate fini suivi d'un mot de son alphabet (dans le cas contraire, la machine passe dans l'état e_{r} et refuse le mot $\langle A, w \rangle$). La simulation de l'automate peut alors commencer. La machine utilise le ruban pour écrire l'état dans lequel l'automate se trouve et marque le symbole du mot w à lire ; elle cherche alors le nouvel état dans la partie du codage de A qui correspond à la fonction τ et passe au symbole suivant.

Lorsque le mot w est entièrement lu, la machine compare l'état dans lequel elle est aux état terminaux de A et passe dans l'état e_a si elle est dans l'un d'entre eux, dans l'état e_r sinon. □

§2.6. Indécidabilité du problème d'arrêt

Nous venons d'expliquer qu'il existait un procédé algorithmique pour décider si un mot est accepté ou non par un automate fini, et nous avons décrit une machine de Turing pour ce faire. La question qui vient immédiatement après est celle d'un procédé algorithmique pour décider si un mot est accepté par une machine de Turing. On introduit ainsi le langage

$$A_{\mathrm{MT}} = \{\, \langle T, w\rangle \,;\, \text{le mot } w \text{ est accepté par la machine de Turing } T\}.$$

On a note $\langle T, w\rangle$ un codage d'une machine de Turing T et du mot w qui lui est donné en entrée. Le codage choisi importe peu, la généralisation évidente de celui utilisé pour les automates finis convient tout à fait.

Observons que ce langage est reconnaissable. Il suffit en effet de construire une machine de Turing qui simule la machine T avec le mot w en entrée et qui accepte $\langle T, w\rangle$ si la machine simulée T accepte w. Une telle machine est ainsi appelée *machine de Turing universelle*.

La découverte importante d'A. TURING est l'*indécidabilité* de ce langage.

THÉORÈME 2.6.1. — *Le langage A_{MT} n'est pas décidable.*

Démonstration. — Supposons par l'absurde qu'il le soit et imaginons une machine de Turing M qui décide de ce langage. Autrement dit, lorsque M accepte $\langle T, w\rangle$ si T accepte w et refuse sinon. Construisons alors une machine de Turing N qui fonctionne de la façon suivante : si son ruban contient initialement la description $\langle T\rangle$ d'une machine de Turing T, elle utilise M comme sous-routine, avec l'entrée $\langle T, \langle T\rangle\rangle$ formée de la description de T suivie de la description de cette description. La machine N renvoie alors l'opposé de ce qu'a renvoyé M : elle accepte (s'arrête dans l'état e_a) si M refuse $\langle T, \langle T\rangle\rangle$ et refuse (s'arrête dans l'état e_r) si M accepte ce mot.

La contradiction apparaît dès que l'on se demande ce que fait N lorsqu'elle reçoit en entrée sa propre description $\langle N\rangle$. Conformément à la construction de N, elle commence par exécuter la machine M avec l'entrée $\langle N, \langle N\rangle\rangle$. Par définition, la machine M accepte cette entrée si N accepte $\langle N\rangle$ et refuse sinon. Par suite, N refuse $\langle N\rangle$ si N l'accepte et l'accepte si elle le refuse. Cette contradiction prouve qu'il ne peut exister de machine M qui décide du langage A_{MT}. □

Rappelons qu'un langage est décidable si et seulement s'il est reconnaissable ainsi que son complémentaire. Puisque le langage A_{MT} est reconnaissable, son complémentaire ne peut l'être.

COROLLAIRE 2.6.2. — *Le langage* $\complement A_{\mathrm{MT}}$ *n'est pas reconnaissable.*

Une autre conséquence de ce théorème est l'indécidabilité de nombreux autres langages liés aux machines de Turing. Le plus fameux d'entre eux est le problème de l'arrêt d'une machine de Turing : il s'agit du langage

$$H_{\mathrm{MT}} = \{\langle T, w\rangle \,;\ \text{la machine } T \text{ s'arrête avec l'entrée } w\,\}.$$

COROLLAIRE 2.6.3. — *Le langage* H_{MT} *n'est pas décidable.*

Démonstration. — Supposons par l'absurde que ce langage soit décidable et soit H une machine de Turing qui le décide. Construisons alors une machine de Turing M qui, lorsqu'elle reçoit la description $\langle T, w\rangle$ d'une machine de Turing T et d'un mot w fait les deux choses suivantes :

– elle exécute $\langle T, w\rangle$ sur la machine H et refuse si H refuse, c'est-à-dire si, soumise à l'entrée w, la machine T ne s'arrête jamais ;

– si H a accepté, elle simule T sur l'entrée w et accepte si T accepte, refuse si T refuse.

Cette machine est bien un décideur puisqu'elle ne simule T qu'après avoir eu l'assurance (via la machine H) que T s'arrêterait. En outre, cette machine M accepte $\langle T, w\rangle$ si T accepte w, et refuse ce mot si T ne s'arrête pas ou si T refuse w. À l'aide de la machine H, nous avons construit une machine de Turing qui décide du langage A_{MT}, ce qui est absurde, d'où le corollaire. □

§2.7. Exercices

Exercice 2.7.1. — Pour chacun des langages suivants, décrire une machine de Turing qui accepte un mot dans l'alphabet $\{0,1\}$ si et seulement s'il lui appartient :

– $L_1 = \{w | w \text{ contient autant de 0 que de 1}\}$;

– $L_2 = \{w | w \text{ contient exactement deux fois plus de 0 que de 1}\}$;

– $L_3 = \{w | w \text{ contient au moins deux fois plus de 0 que de 1}\}$.

Exercice 2.7.2. — Décrire une machine de Turing qui accepte un mot dans l'alphabet $\{a,b,c\}$ si et seulement s'il est de la forme $a^i b^j c^k$, où i, j, k sont des entiers tels que $k = i + j$.

Même question en remplaçant la condition $k = i + j$ par la condition $k = ij$.

Exercice 2.7.3. — Soit A_{REX} le langage formé des couples (R, w), où R est une expression régulière et w un mot décrit par R. Démontrer que ce langage est décidable.

Exercice 2.7.4. — Soit V_{AF} le langage formé des $\langle A\rangle$ où A est un automate fini dont le langage $\mathscr{L}(A)$ est vide. Démontrer que ce langage est décidable.

Exercice 2.7.5. — Soit I_{AF} le langage formé des $\langle A \rangle$ où A est un automate fini dont le langage $\mathscr{L}(A)$ est infini. Démontrer que ce langage est décidable.

Exercice 2.7.6. — Soit L un langage. Démontrer que L est reconnaissable si et seulement s'il existe un langage décidable L' tel que L soit l'ensemble des mots x pour lesquels il existe un mot y tel que le mot $x\#y$ appartient à L',

Exercice 2.7.7. — Soit L un langage reconnaissable formé de descriptions $\langle M_i \rangle$ de machines de Turing M_i ($i \geqslant 1$) qui sont des décideurs. Démontrer qu'il existe un langage décidable D qui n'est décidé par aucune des machines M_i.

Exercice 2.7.8. — Soit V_{MT} le langage formé des descriptions $\langle T \rangle$ de machines de Turing T dont le langage $\mathscr{L}(T)$ est vide. Démontrer que ce langage est indécidable.

Exercice 2.7.9. — Soit R_{MT} le langage formé des descriptions $\langle T \rangle$ de machines de Turing T dont le langage $\mathscr{L}(T)$ est un langage régulier.

a) Étant donné une machine de Turing M et un mot w, construire une machine de Turing M' telle que $\mathscr{L}(M') = \Sigma^*$ si M reconnaît w, et $\mathscr{L}(M') = \{0^n 1^n \,;\, n \geqslant 0\}$ sinon.

b) Démontrer que le langage $\mathscr{L}(M') = \{0^n 1^n \,;\, n \geqslant 0\}$ n'est pas régulier.

c) Démontrer que le langage R_{MT} n'est pas décidable.

Exercice 2.7.10 (Théorème de Rice). — Soit P un langage formé de descriptions $\langle M \rangle$ de machines de Turing. On fait les hypothèses suivantes :

- P n'est pas vide ;
- il existe au moins une description $\langle M \rangle$ qui n'appartient pas à P ;
- si deux machines M et M' reconnaissent le même langage et que $\langle M \rangle$ appartient à P, alors $\langle M' \rangle$ aussi.

Dit informellement, P est une propriété non-triviale des langages reconnus par les machines de Turing. Démontrer que ce langage P n'est pas décidable.

Exercice 2.7.11. — Déduire du théorème de Rice que les langages suivants ne sont pas décidables :

a) le langage I_{MT} formé des descriptions $\langle M \rangle$ de machines de Turing telles que $\mathscr{L}(M)$ soit infini ;

b) le langage formé des descriptions $\langle M \rangle$ de machines de Turing telles que $\mathscr{L}(M)$ contienne 1011 ;

c) le langage T_{MT} formé des descriptions $\langle M \rangle$ de machines de Turing telles que $\mathscr{L}(M) = \Sigma^*$.

CHAPITRE 3

COMPLEXITÉ

Dans ce troisième chapitre, on va quantifier la complexité de problèmes mathématiques ou de langages en mesurant le temps et/ou l'espace que requiert une machine de Turing pour les décider.

§3.1. Complexité d'une machine de Turing ; complexité d'un langage

A. Définitions

Soit M une machine de Turing qui est un décideur. La complexité en temps et la complexité en espace de M sont les deux fonctions $\tau_M : \mathbf{N} \to \mathbf{N}$ et $\sigma_M : \mathbf{N} \to \mathbf{N}$ telles que $\tau_M(n)$ soit le nombre maximal de pas qu'effectue M, et $\sigma_M(n)$ le nombre maximal de cases du ruban lues par M lorsqu'on lui soumet un ruban initial de taille n.

Si M est une machine de Turing non déterministe, la définition est analogue, si ce n'est qu'on demande que chaque branche du calcul requière au plus $\tau_M(n)$ étapes, resp. la lecture d'au plus $\sigma_M(n)$ cases du ruban lorsqu'on soumet à M un ruban de taille n.

Si $f : \mathbf{N} \to \mathbf{N}$ est une fonction, on dit que la machine M est en temps (en espace) $f(n)$ si $\tau_M(n) \leqslant f(n)$ ($\sigma_M(n) \leqslant f(n)$) pour tout n.

On s'intéressera en fait à la complexité asymptotique, c'est-à-dire pour n grand, et en se contentant des ordres de grandeur et de la croissance de la fonction f. On parlera de machine en temps $\mathrm{O}(f(n))$, ou en espace $\mathrm{O}(f(n))$. Cela signifie qu'il existe une constante c et un entier n_0 tels que $\tau_M(n) \leqslant cf(n)$ (resp. $\sigma_M(n) \leqslant cf(n)$) pour $n \geqslant n_0$.

Soit $f : \mathbf{N} \to \mathbf{N}$. Les classes de complexité $\mathrm{TIME}(f(n))$ (resp. $\mathrm{SPACE}(f(n))$) sont les langages qui sont décidés par une machine de Turing en temps $\mathrm{O}(f(n))$ (resp. en espace $\mathrm{O}(f(n))$). Les classes de complexité $\mathrm{NTIME}(f(n))$ (resp. $\mathrm{NSPACE}(f(n))$) sont les langages qui sont décidés par une machine de Turing non déterministe en temps $\mathrm{O}(f(n))$ (resp. en espace $\mathrm{O}(f(n))$).

B. Dépendance du modèle de machine de Turing

Nous avons vu plusieurs modèles de machines de Turing et démontré qu'ils décidaient les mêmes langages. Qu'en est-il pour les notions de complexité qu'ils permettent de définir ?

Une première inégalité évidente est $\sigma_M(n) \leqslant \tau_M(n)$: chaque accès à une nouvelle case du ruban requiert une étape du calcul. Cette inégalité permettra de majorer la longueur des déplacements qu'effectue une machine de Turing lorsqu'elle doit parcourir tout le ruban, par exemple pour rechercher un symbole particulier, ou revenir en tête du ruban.

LEMME 3.1.1. — *Soit $f : \mathbf{N} \to \mathbf{N}$ une fonction telle que $f(n) \geqslant n$ pour tout n. Alors toute machine de Turing à plusieurs rubans qui décide d'un langage en temps $f(n)$ possède un équivalent à un seul ruban qui décide de ce langage en temps $\mathrm{O}(f(n)^2)$.*

Démonstration. — On reprend la machine N qui simulait les k rubans de la machine M en écrivant sur un seul ruban les premiers caractères, suivi d'un séparateur, suivi des seconds caractères, etc. La préparation du ruban requiert un temps $\mathrm{O}(n) \leqslant \mathrm{O}(\tau_M(n))$. En outre, lorsque la nouvelle machine simule la lecture des k rubans, elle doit promener sa tête tout le long du ruban, lisant ainsi au plus $\mathrm{O}(\sigma_M(n)) \leqslant \mathrm{O}(\tau_M(n))$ cases du ruban, ce qu'elle fait au plus $\tau_M(n)$ fois, d'où une complexité en temps $\mathrm{O}(\tau_M(n)^2)$. □

LEMME 3.1.2. — *Soit $f : \mathbf{N} \to \mathbf{N}$ une fonction telle que $f(n) \geqslant n$ pour tout n. Soit M une machine de Turing non déterministe qui décide un langage L en temps $f(n)$; il existe une machine de Turing déterministe qui décide L en temps $2^{\mathrm{O}(f(n))}$.*

Démonstration. — Nous avions construit une machine de Turing déterministe à trois rubans qui décide L ; analysons sa complexité en temps. Soit m un majorant du nombre de choix à chaque transition ; il y a donc au plus m calculs de longueur 1, m^2 calculs de longueur 2, etc. L'exploration de ces calculs, jusqu'à leur longueur maximale $f(n)$, requiert ainsi la simulation d'au plus $m + m^2 + \cdots + m^{f(n)}$ calculs par la machine M, requérant au total $m + 2m^2 + \cdots + f(n)m^{f(n)}$ étapes. La préparation de chaque calcul (remise à zéro du ruban, etc.) est en $\mathrm{O}(nf(n))$. Au total, tout cela requiert au plus $\mathrm{O}(f(n)m^{f(n)}) \leqslant 2^{\mathrm{O}(f(n))}$ étapes.

Comme cette estimation est faite pour une machine à trois rubans, il reste à l'élever au carré, ce qui fournit encore une machine de Turing en temps $2^{\mathrm{O}(f(n))}$. □

C. Les classes P et NP

La classe P est la réunion des classes $\mathrm{TIME}(n^k)$, pour $k \geqslant 1$. De même, la classe NP est la réunion des classes $\mathrm{NTIME}(n^k)$, pour $k \geqslant 1$. L'intérêt de ces classes est qu'elles sont insensibles au modèle spécifique de machine de Turing qui a été choisi, et qu'elles correspondent bien à l'idée intuitive de ce qui est réalisable à l'aide d'un ordinateur. Le

tableau suivant, tiré de Garey & Johnson (1979), montre bien le temps que met une machine de Turing effectuant un million d'opérations par seconde suivant sa complexité :

	taille n des données					
complexité	10	20	30	40	50	60
n	10^{-5} s	$2\cdot10^{-5}$ s	$3\cdot10^{-5}$ s	$4\cdot10^{-5}$ s	$5\cdot10^{-5}$ s	$6\cdot10^{-5}$ s
n^2	10^{-4} s	$4\cdot10^{-4}$ s	$9\cdot10^{-4}$ s	$16\cdot10^{-4}$ s	$25\cdot10^{-4}$ s	$36\cdot10^{-4}$ s
n^3	10^{-3} s	$8\cdot10^{-3}$ s	$27\cdot10^{-3}$ s	$64\cdot10^{-3}$ s	0,125 s	0,216 s
n^5	0,11 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	10^{-3} s	1,0 s	17,9 min	12,7 j	35,7 années	366 siècles
3^n	0,059 s	58 min	6,5 années	3855 siècles	$2\cdot10^8$ siècles	$1{,}3\cdot10^{13}$ siècles

Cela ne signifie pas que l'exposant, voire la constante implicite dans le O, n'ait pas d'importance, mais l'appartenance à la classe P est dans de nombreux cas une propriété fondamentale des langages considérés.

Donnons-en quelques exemples.

Exemple 3.1.3 (Le problème *PATH*). — Un graphe est la donnée d'un ensemble S de sommets, d'un ensemble A d'arêtes et de deux applications origine et terme, $o, t\colon A \to S$: une arête $a \in A$ relie le sommet $o(a)$ au sommet $t(a)$. On peut coder un graphe par sa matrice d'adjacence : c'est une matrice $S \times S$, l'entrée $a_{s,s'}$ étant le nombre d'arêtes reliant s à s'. Le problème *PATH* consiste à déterminer si, étant donné un graphe et deux sommets s et s', il existe un chemin reliant s à s', c'est-à-dire une suite d'arêtes $(a_1,\dots,a_n)$ telles que $s = o(a_1)$, $t(a_1) = o(a_2)$, …, $t(a_{n-1}) = o(a_n)$, $t(a_n) = s'$.

Le langage correspondant est donc l'ensemble des triplets $(\langle G\rangle, s, s')$ tels que $\langle G\rangle$ soit la description d'un graphe G, s et s' des sommets de G pour lesquels il existe un chemin reliant s à s'.

Ce langage appartient à la classe NP. En effet, une machine non déterministe peut construire un chemin reliant s à s' en partant de s et en se déplaçant dans G en choisissant de façon non déterministe une arête issue du sommet courant. S'il y a des boucles dans le graphe G, cette machine peut calculer indéfiniment. Toutefois, s'il existe un chemin reliant s à s', il en existe un qui ne revienne jamais par un sommet déjà visité, et sa longueur est au plus égale au nombre de sommets. Si on marque les sommets visités et qu'on exige de la machine de ne jamais retourner en un sommet visité, on obtient une machine de Turing non déterministe qui décide le langage *PATH* temps $\mathrm{O}(n^2)$: chacun des $\leqslant n$ choix d'arêtes requiert une recherche dans G qui est effectuable en temps $\mathrm{O}(n)$.

Si l'on rend naïvement déterministe cet algorithme, on obtient que le langage *PATH* appartient à la classe $\mathrm{TIME}(2^{\mathrm{O}(n^2)})$. Montrons qu'en fait *PATH* appartient à P. L'idée

consiste à marquer le sommet initial s puis, à chaque étape, à marquer tous les sommets non encore marqués qui sont le terme d'une arête dont l'origine est un sommet marqué. La machine s'arrête (et accepte) lorsqu'elle marque le sommet s', s'arrête (et refuse) lorsqu'à une étape elle n'a pu marquer aucun nouveau sommet. Il y a au plus n étapes (on marque au moins un sommet à chaque fois), et chacune de ces étapes requérant la lecture du graphe est effectuable en temps $O(n^2)$. Au total, on obtient bien que *PATH* appartient à P.

Exemple 3.1.4 (Coprimalité). — Il s'agit, étant donné deux entiers m et n, de décider s'ils sont premiers entre eux ou non. Le langage *COPRIME* est ainsi formé des mots de la forme $\langle m\rangle\#\langle n\rangle$ où $\langle m\rangle$ et $\langle n\rangle$ sont les développements en base 2 d'entiers m et n. Puisque le nombre de chiffres binaires d'un entier est le plus petit entier inférieur ou égal à son logarithme en base 2, la taille du mot en entrée est donc égale à $\lfloor \log_2(m)\rfloor + \lfloor \log_2(n)\rfloor + 1$.

La méthode naïve consistant, pour chaque entier d de 2 à m (voire à $\min(m,n)$), à décider si d divise m et n, fournit un algorithme de complexité exponentielle.

L'algorithme d'Euclide (IV$^{\text{e}}$ s. av. J.-C) est en revanche dans la classe P. Il consiste, tant que $n \neq 0$, à calculer $m' = m \pmod n$ (en temps linéaire) et à recommencer avec n et m' ; le dernier entier non nul obtenu est le pgcd de m et n. Chaque étape du calcul est effectuée en temps linéaire en la longueur de m. En effet, si n et m sont des entiers, avec $m \neq 0$, le reste de la division de n par m se calcule facilement en temps polynomial : si $n = 2^k n_k + 2^{k-1} n_{k-1} + \cdots + n_0$, on calcule successivement $n_k \pmod m$, $2n_k + n_{k-1} \pmod m$, etc. Le nombre d'étapes est le nombre de chiffres binaires de n ; chaque étape requiert une multiplication par 2, une comparaison avec m et, éventuellement, une soustraction de m (on utilise que si $0 \leqslant \xi < m$ et $\nu \in \{0,1\}$, alors $1 \leqslant 2\xi + \nu < 2m$, donc $2\xi + \nu \pmod m$ vaut $2\xi + \nu$ ou $2\xi + \nu - m$), donc s'effectue en temps linéaire en la longueur de $\max(m,n)$. Il reste à voir que le nombre d'étapes est lui-même linéaire en la longueur de $\max(m,n)$. Pour cela, on remarque que chaque étape remplace un couple (n,m), où $n \geqslant m$, par le couple $(m, n \mod m)$ et que le plus grand entier du couple est au moins divisé par deux : si $1 \leqslant m \leqslant n$, alors $n \pmod m < m$; si de plus $n \geqslant 2m$, alors $n \pmod m < m \leqslant \frac{1}{2}n$; sinon, $n < 2m$, donc $n \pmod m = n - m < \frac{1}{2}n$.

La complexité de cet algorithme est ainsi quadratique en la longueur de m et n, ce qui prouve que le langage *COPRIME* appartient à P.

Exemple 3.1.5 (Le problème *PRIME* ; celui de la factorisation).

Le langage *COMPOSITE* formé des développements binaires d'entiers qui sont des nombres premiers est dans la classe NP : vérifier qu'un entier d tel que $2 \leqslant d < n$ divise n est faisable en temps polynomial ; et tester tous ces entiers de façon non déterministe (voire seulement ceux jusqu'à $\leqslant \sqrt{n}$) étant accessible à une machine de Turing non déterministe, on voit que ce langage appartient à NP. On dit que le langage

complémentaire, *PRIME*, formé des développements binaires d'entiers qui sont des nombres premiers est dans la classe coNP.

Nous verrons plus tard qu'il est dans la classe NP, en appliquant des résultats classiques en théorie de la primalité. Il est important de comprendre pourquoi l'argument précédent ne le prouve pas. Rappelons que pour qu'une machine de Turing non déterministe accepte un mot, il faut et il suffit qu'*une* des branches du calcul l'accepte. Pour le problème *COMPOSITE*, une des branches du calcul détecte un diviseur strict de n, dès que n est premier ; se contenter d'inverser la réponse de la machine de Turing fournirait le langage pour lesquels un des entiers entre 2 et $n-1$ ne divise pas n, ce que vérifie tout entier >2 !

C'est en revanche un théorème très récent, prouvé en 2002 par les informaticiens indiens AGRAWAL, KAYAL et SAXENA, que *PRIME* appartient à P, pour ces deux résultats, voir le paragraphe 3.3 de ce cours.

En conséquence, *PRIME*, tout comme *COMPOSITE*, appartient à PSPACE, ce qu'il est en fait très facile de voir directement : la division successive par chaque entier entre 2 et $\sqrt{n}$ ne requiert qu'un espace polynomial si l'on réutilise la même partie du ruban pour chaque division euclidienne.

La complexité du problème de la factorisation est, en revanche, inconnue. Ce problème (plus qu'un langage) consiste à calculer la décomposition en facteurs premiers d'un entier donné. S'il appartient à la classe NP pour les mêmes raisons que les algorithmes précédents, on ne sait pas (mais on ne le pense pas) s'il appartient à la classe P.

§3.2. NP-complétude

Commençons par un nouvel exemple de langage appartenant à la classe NP.

Exemple 3.2.1 (Le langage *SAT*). — Une formule booléenne est une expression algébrique construite à l'aide de variables et des opérateurs $\vee$ (OU), $\wedge$ (ET) et $\neg$ (NON) ; par exemple

$$\varphi = (\neg x \vee y) \wedge (x \vee \neg z).$$

Si $x, y, z, \ldots$ sont des valeurs booléennes (0, 1 ; ou VRAI, FAUX), la valeur de la formule se calcule à l'aide des tables de vérité standard ; par exemple, si $x = 0$, $y = 1$, $z = 1$, on a

$$\varphi = (\neg 0 \vee 1) \wedge (0 \vee \neg 1) = (1 \vee 1) \wedge (0 \vee 0) = 1 \wedge 0 = 0.$$

Le langage *SAT* est formé des formules booléennes en des variables $x, y, z, \ldots$ qui sont satisfiables, c'est-à-dire pour lesquelles il existe des valeurs de $x, y, z, \ldots$ en lesquelles la formule s'évalue en 1.

Comme l'évaluation d'une formule est effectuable en temps polynomial, ce langage appartient à NP : il suffit de choisir, de façon non déterministe, des valeurs pour les variables.

THÉORÈME 3.2.2 (Cook, 1971). — *Si* SAT *appartient à P, alors* P = NP.

Autrement dit, le problème *SAT* est résoluble en temps polynomial, n'importe quel problème de la classe NP l'est aussi !

Démonstration. — Supposons que *SAT* appartienne à P et soit M une machine de Turing non déterministe décidant en temps $p(n)$, de l'appartenance d'un mot de longueur n à un langage L, où p est un polynôme dont on suppose, pour simplifier que $p(n) \geqslant n$ pour tout $n \geqslant 0$. Nous devons démontrer qu'il existe une machine de Turing déterministe de la classe P qui décide le langage L. Pour cela, nous allons construire une formule booléenne φ_M dont les variables représentent l'historique du calcul qu'effectuerait M, avec un ruban initial donné, un choix de valeurs des variables fournissant une branche de calcul.

Notons Γ l'alphabet du ruban et E l'ensemble des états. Les variables sont les suivantes :

- pour $1 \leqslant i \leqslant p(n)$, $j \in \Gamma$, $0 \leqslant k \leqslant p(n)$, une variable $R_{i,j,k}$ signifiant que la case R_i du ruban contient le symbole j à l'étape k du calcul ;
- pour $1 \leqslant i \leqslant p(n)$ et $0 \leqslant k \leqslant p(n)$, une variable $T_{i,k}$ signifiant que la tête de lecture est à la case i du ruban à l'étape k du calcul ;
- pour $q \in E$ et $0 \leqslant k \leqslant p(n)$, une variable $Q_{q,k}$ signifiant que M est dans l'état q à l'étape k du calcul.

Le nombre de ces variables est $O(p(n)^2)$.

La formule construite est la conjonction d'une longue liste de clauses qui expriment que chaque étape du calcul obéit aux règles de la machine M :

- pour $1 \leqslant i \leqslant p(n)$ et j l'état initial de la la i-ième case du ruban initial, la clause $R_{i,j,0}$;
- pour $q \in E$, la clause $E_{q,0}$ si l'état initial de la machine est q, la clause $\neg E_{q,0}$ sinon ;
- $T_{1,1}$: à l'état initial, la tête est au début du ruban ;
- pour $1 \leqslant i \leqslant p(n)$, $j \neq j' \in \Gamma$, $0 \leqslant k \leqslant p(n)$, la clause $\neg R_{i,j,k} \vee \neg R_{i,j',k}$ exprimant que les cases du ruban ne contiennent qu'un symbole ;
- pour $1 \leqslant i < i' \leqslant p(n)$ et $0 \leqslant k \leqslant p(n)$, la clause $\neg T_{i,k} \vee \neg T_{i',k}$ exprimant qu'à l'étape k, la tête du ruban ne peut être simultanément en i et i' ;
- pour $q \neq q' \in Q$ et $0 \leqslant k \leqslant p(n)$, la clause $\neg E_{q,k} \vee \neq E_{q',k}$ exprimant que la machine ne peut être dans deux états à la fois ;
- pour $1 \leqslant i \leqslant p(n)$ et $0 \leqslant k < p(n)$, la clause $(R_{i,j,k} = R_{i,j,k+1}) \vee T_{i,k}$ qui signifie que le ruban n'est modifié qu'à la position de la tête de lecture, où $x = y$ est une abbréviation pour $(\neg x \wedge \neg y) \vee (x \wedge y)$;
- pour tout couple (q, j) et tout triplet (q', j', ε) tels que $q, q' \in Q$, $j, j' \in \Gamma$, $\varepsilon \in \{-1, 0, 1\}$ tel que (q', j', ε) est une transition possible du calcul à partir de la lecture

du symbole j dans l'état q, pour tout $1 \leqslant i \leqslant p(n)$, $0 \leqslant k < p(n)$, une clause

$$(R_{i,k} \wedge E_{q,k} \wedge T_{i,j,k}) \to (R_{\max(i+\varepsilon,1),k+1} \wedge E_{q',k+1} \wedge T_{i,j',k+1})$$

exprimant la possibilité de cette transition. Pour simplifier, on exige que la fonction de transition permette de ne rien faire c'est-à-dire que $(q, j, 0)$ est une transition possible à partir de (q, j) ;

– la clause $Q_{e_a,p(n)}$ exprimant que l'on est dans l'état qui accepte à l'étape finale $p(n)$ du calcul ; c'est pour cette clause que l'on a fait l'hypothèse précédente sur la fonction de transition.

Le nombre de ces clauses est $\mathrm{O}(p(n)^3)$.

Pour indiquer les variables, on utilise un langage qui les représente par leur développement binaire ; ainsi N variables requièrent en gros $\log_2 N$ cases du ruban pour être spécifiées, chacune de ces clauses est de longueur $\mathrm{O}(\log p(n))$, si bien que la formule booléenne donnée par la conjonction de toutes ces clauses est de longueur majorée par $\mathrm{O}(\log p(n)\, p(n)^3)$, elle-même majorée par un polynôme $q(n)$ en n. En outre, sa satisfabilité est, par construction même, équivalente à l'existence d'une branche du calcul qui accepte le mot du ruban initial. (On rappelle la remarque déjà utilisée que seules les $p(n)$ premières cases du ruban pourront être modifiées par le calcul ; ce n'est pas la peine de se préoccuper des autres.)

Puisque *SAT* est supposé appartenir à P, il reste à tester en temps polynomial en $q(n)$ la satifiabilité de cette formule, d'où le théorème. □

§3.3. Primalité en temps polynomial

A. Quelques propriétés des anneaux $\mathbf{Z}/n\mathbf{Z}$

Soit p un nombre premier et F l'anneau $\mathbf{Z}/p\mathbf{Z}$. Nous aurons à faire usage des résultats suivants :

– Pour tout $a \in F$, $a^p = a$; de manière équivalente, $n^p \equiv n \pmod p$ pour tout entier $n \in \mathbf{Z}$.

– L'anneau F est un corps, son groupe multiplicatif $F^* = F \setminus \{0\}$ est cyclique d'ordre $p-1$.

– Inversement, si $n \geqslant 2$ n'est pas un nombre premier, alors $(\mathbf{Z}/n\mathbf{Z})^*$ est de cardinal $\varphi(n) < n-1$ donc n'est pas cyclique d'ordre $n-1$.

B. Générateurs de $(\mathbf{Z}/p\mathbf{Z})^*$

Notons G le groupe $(\mathbf{Z}/p\mathbf{Z})^*$; rappelons qu'il est cyclique donc il existe un élément de G dont l'ordre est $p-1$. Inversement, comment vérifier qu'un tel élément, disons a, est effectivement d'ordre $p-1$? Calculer ses puissances successives résulterait en un algorithme exponentiel. La remarque essentielle est la suivante. Soit $p-1 = \prod_i \ell_i^{m_i}$ la

décomposition en facteurs premiers de $p-1$, où $\ell_1, \dots$ sont des nombres premiers distincts. Si a est d'ordre $p-1$, alors $a^{(p-1)/\ell_i} \neq 1$ pour tout i ; inversement, si ces inégalités sont satisfaites, l'ordre de a (qui est un diviseur de $p-1$) est nécessairement égal à $p-1$, car un diviseur strict de $p-1$ divise l'un des entiers $(p-1)/\ell_i$.

La donnée de l'entier a et de la factorisation de $p-1$ fournit donc un certificat de primalité de l'entier p. Étant donné ce certificat, les calculs à effectuer pour vérifier la primalité de p ne requièrent qu'un temps polynomial.

En effet, élever un entier a modulo p à une puissance k requiert $O(\log k \log^2 p)$ opérations (par l'algorithme d'exponentiation binaire), soit $O(\log^3 p)$ pour chacune des puissances $(p-1)/\ell_i$. Par ailleurs, comme $p-1 = \prod \ell_i^{m_i} \geqslant 2^r$, où r est le nombre de facteurs premiers distincts, il y a au plus $O(\log p)$ tests à faire, d'où $O(\log^4 p)$ au total.

Pour que ces calculs fournissent une preuve complète de la primalité de p, il faut en outre s'assurer de la primalité des entiers ℓ_i. ce qui est fait par récurrence. La nombre $N(p)$ de nombres premiers à vérifier pour établir la primalité de p vérifie ainsi l'inégalité

$$N(p) \leqslant \sum_{i=1}^{r} N(\ell_i), \quad \text{où } \ell_1, \dots, \ell_r \text{ sont les facteurs premiers de } p-1,$$

avec l'initialisation $N(2) = 0$. Par suite, $N(p) \leqslant \log p$ pour tout p, car, si $N(\ell) < \ell$ pour tout nombre premier $\ell < p$, alors

$$N(p) \leqslant \sum_{i=1}^{r} \log \ell_i \leqslant \log \prod \ell_i \leqslant \log(p-1) < \log p.$$

Chacun des calculs requiert au plus $O(\log^4 p)$ opérations, d'où un certificat de primalité dont la vérification requiert au plus $O(\log^5 p)$.

THÉORÈME 3.3.1 (Pratt (1975)). — *Le langage* PRIMES *appartient à NP et à coNP.*

Démonstration. — Une machine de Turing non déterministe peut construire un certificat de primalité en temps polynomial, ce qui démontre que *PRIMES* est dans NP. Par ailleurs, nous avions vu que le langage complémentaire, *COMPOSITES*, appartient à NP, en devinant une factorisation. D'où le théorème de Pratt. □

C. Petit théorème de Fermat dans les polynômes

On commence par un critère de primalité simple, mais absolument inefficace.

LEMME 3.3.2. — *Soit n un entier naturel tel que $n > 2$. Pour que n soit un nombre premier, il faut et il suffit que tous les coefficients binomiaux $\binom{n}{k}$, pour $1 \leqslant k \leqslant n-1$, soient multiples de n.*

Démonstration. — Supposons que n soit premier. Alors, pour tout entier k tel que $1 \leqslant k \leqslant n-1$, $k!\binom{n}{k} = n(n-1)\dots(n-k+1)$ est multiple de n. Comme n est premier, il ne divise pas $k!$; il divise donc $\binom{n}{k}$.

Inversement, supposons que n ne soit pas premier et soit p un facteur premier de n, soit e l'exposant de p dans la décomposition en facteurs premiers de n. D'après ce qui précède et la formule du binôme de Newton, on a $(1+X)^p \equiv 1+X^p \pmod{p}$, et, par récurrence, $(1+X)^{p^e} \equiv 1+X^{p^e} \pmod{p}$. Mettons ceci à la puissance $m = n/p^e$; on obtient

$$\begin{aligned}(1+X)^n &\equiv (1+X^{p^e})^m \pmod{p}\\ &\equiv 1+mX^{p^e}+\binom{m}{2}X^{2p^e}+\cdots+X^{mp^e} \pmod{p}\\ &\equiv 1+mX^{p^e} \pmod{(p, X^{2p^e})}.\end{aligned}$$

En particulier, $\binom{n}{p^e} \equiv m \pmod{p}$. Comme m est premier à p, cela entraîne que p ne divise pas $\binom{n}{p^e}$ et, a fortiori, n ne divise pas $\binom{n}{p^e}$. □

L'inefficacité de ce test vient qu'il faut a priori calculer $n-1$ coefficients binomiaux pour décider la non primalité d'un entier n, c'est-à-dire une complexité au moins exponentielle en la longueur du développement binaire de n.

L'idée fondamentale de Agrawal *et al.* (2004) est qu'un affaiblissement convenable de ce test persiste à décider la primalité de n, mais devient de complexité polynomiale. On commence par combiner la congruence des coefficients binomiaux avec le petit théorème de Fermat en écrivant que la congruence $(X-a)^n \equiv X^n - a \pmod{n}$ pour tout $a \in \mathbf{Z}/n\mathbf{Z}$ est une condition nécessaire et suffisante pour que n soit premier. En fait, ces auteurs prouvent le théorème suivant.

THÉORÈME 3.3.3. — *Soit n un entier naturel tel que $n \geqslant 2$. Soit r un nombre premier tel que l'on ait :*

a) *aucun nombre premier $\leqslant r$ ne divise n ;*

b) *l'ordre de n modulo r est supérieur ou égal à $1+4\log_2(n)^2$.*

Si, pour tout entier a tel que $1 \leqslant a \leqslant r-1$, on a

$$(X-a)^n \equiv X^n - a \pmod{(n, X^r-1)},$$

alors n est une puissance d'un nombre premier ; sinon, n nest pas premier.

En outre, il existe un nombre premier r vérifiant les deux conditions a) *et* b) *et majoré par $c\log(n)^5$, où c est un nombre réel indépendant de n.*

Avant de démontrer ce théorème, déduisons-en la très belle conséquence :

COROLLAIRE 3.3.4. — *Le langage* PRIMES *appartient à la classe P.*

Démonstration. — Soit n un nombre entier tel que $n \geqslant 2$.

On commence par vérifier si n est une puissance d'un autre entier, c'est-à-dire s'il s'écrit a^b, pour des entiers a et $b \geqslant 2$. Si c'est le cas, on a $b\log a = \log n$, donc

$b \log\log_2 n$; pour chacune de ces valeurs, l'entier $\lfloor n^{1/b} \rfloor$ est calculable en temps polynomial en $\log n$, par exemple en cherchant un à un les chiffres de son développement binaire. Les détails sont un peu désagréables, mais quiconque a appris à calculer à la main une racine carrée devrait être convaincu. Une façon plus moderne de faire le calcul consiste aussi à déterminer l'unique entier a tel que $a^b \equiv n$ modulo la plus petite puissance de 2 supérieure à n ; si l'on a n'a pas égalité, alors n n'est pas une puissance b-ième. Autrement dit, on calcule non pas une approximation réelle de $n^{1/b}$, mais une approximation 2-adique.

Dans la suite, on suppose que n n'est pas une puissance d'un autre entier ; ce n'est en particulier pas une puissance $\geqslant 2$ d'un nombre premier.

On détermine ensuite le plus petit nombre premier r vérifiant les trois conditions de l'énoncé du théorème. Pour cela, on essaye successivement $r = 2, 3, \ldots$ et on effectue les tests suivants :

– vérifier si r est premier, c'est-à-dire n'est divisible par aucun des nombres premiers plus petits (et qu'on a pris soin de conserver sur le ruban) ;
– vérifiquer que r ne divise pas n, sinon répondre que n n'est pas premier ;
– vérifier que r est premier avec $n^b - 1$ pour tout entier b tel que $1 \leqslant b \leqslant B$, où $B = 1 + \lfloor 2\log_2(n)^2 \rfloor$.

Pour chaque entier r, la complexité des calculs est polynomiale en $\log n$; d'après le théorème, le plus petit entier r qui convient est inférieur à un multiple de $\log(n)^5$; cette étape est effectuable en temps polynomial en $\log(n)$.

Il reste à tester la dernière condition. Pour cela, il faut calculer $r-1$ puissances n-ièmes dans l'anneau $A = (\mathbf{Z}/n\mathbf{Z})[X]/(x^r - 1)$. Par l'algorithme d'exponentiation binaire, chacun de ces calculs est effectuable à l'aide de $\log_2(n)$ multiplications et élévations au carré dans l'anneau A, d'où une complexité totale en temps majorée par $\mathrm{O}(r^3 \log_2(n)^3) = \mathrm{O}(\log(n)^{18})$. Si l'une de ces vérifications échoue, n n'est pas premiers ; sinon, n est une puissance d'un nombre premier, donc un nombre premier car nous avons vérifier que n n'est pas une puissance $\geqslant 2$ d'un autre entier.

La démonstration que *PRIMES* appartient à la classe de complexité P est ainsi terminée. □

D. Démonstration du théorème d'Agrawal, Kayal et Saxena

Commençons la démonstration par un lemme combinatoire dont nous aurons à faire usage.

LEMME 3.3.5. — *Soit L et D des entiers naturels. Le nombre de suites $(m_1, \ldots, m_L)$ d'entiers naturels tels que $m_1 + \cdots + m_L \leqslant D$ est égal à $\binom{D}{L}$; il est au moins égal à $2^{\min(D,L)}$, et même à $2^{\min(D,L)+1}$ si $\min(D,L) \geqslant 3$.*

Démonstration. — L'application qui envoie une suite $(m_1, \ldots, m_L)$ d'entiers naturels de somme $\leqslant D$ sur la suite $(m_1 + 1, m_1 + m_2 + 2, \ldots, m_1 + \cdots + m_L + L)$ est une bijection

sur l'ensemble des suites strictement croissantes d'entiers de $\{1,\dots,D+L\}$. Une telle suite étant déterminée par ses valeurs, la première assertion du lemme en découle.

Pour la seconde, on suppose pour fixer les idées que $D \leqslant L$ et on écrit

$$\binom{D+L}{L} = \frac{(D+L)!}{D!L!} = \frac{(L+1)\dots(L+D)}{1\dots D} = \frac{L+1}{1}\frac{L+2}{2}\cdots\frac{L+D}{D}.$$

Chaque facteur étant au moins égal à 2, le produit est supérieur ou égal à 2^D. En outre, si $D \geqslant 1$ et $L \geqslant 3$, le premier facteur est au moins 4, si bien que le produit est minoré par 2^{D+1}, ce qu'il fallait démontrer. □

La proposition suivante recèle le cœur de la preuve du théorème 3.3.3.

PROPOSITION 3.3.6. — *Soit n un entier, r un entier premier à n ; on note M l'ordre de n modulo r. Soit S une partie de $\mathbf{Z}/n\mathbf{Z}$ telle que pour $s \neq s' \in S$, $(s-s', n) = 1$ et telle que $(X-s)^n \equiv X^n - s \pmod{(n, X^r-1)}$ pour tout $s \in S$.*

Si $M > 1 + 4\log_2(n)^2$ et $|S| > 2\sqrt{r}\log_2(n)$, alors n est une puissance d'un nombre premier.

Démonstration. — Soit p un facteur premier de n et soit A l'anneau $\mathbf{F}_p[X]/(X^r-1)$. On observe que pour tout entier m, il existe un unique endomorphisme σ_m de A qui applique X sur X^m ; de plus $\sigma_m \circ \sigma_{m'} = \sigma_{mm'}$ et $\sigma_m = \mathrm{id}$ si $m \equiv 1 \pmod{r}$. En particulier, σ_m est un automorphisme de A si $(m, r) = 1$.

En outre, on remarque que si $P \in A$, $m, m' \in (\mathbf{Z}/r\mathbf{Z})^*$ vérifient $\sigma_m(P) = P^m$ et $\sigma_{m'}(P) = P^{m'}$, alors $\sigma_{mm'}(P) = \sigma_m(\sigma_{m'}(P)) = \sigma_m(P^{m'}) = \sigma_m(P)^{m'} = P^{mm'}$. Comme $\sigma_1(P) = P$, l'ensemble des entiers m tels que $\sigma_m(P) = P^m$ est donc un sous-groupe de $(\mathbf{Z}/r\mathbf{Z})^*$.

Pour tout $P \in \mathbf{F}_p[X]$, on a $P(X^p) = P(X)^p$, donc $\sigma_p(P) = P^p$ pour tout $P \in A$.

Par hypothèse, pour tout $s \in S$, on a $\sigma_n(X-s) = X^n - s = (X-s)^n$ dans A. Soit $P \in \mathbf{F}_p[X]$ un polynôme de la forme $\prod_{s\in S}(X-s)^{e_s}$. Par multiplicativité, il vient $\sigma_n(P) = P^n$, donc aussi $\sigma_m(P) = P^m$ pour tout m appartenant au sous-groupe E de $(\mathbf{Z}/r\mathbf{Z})^*$ engendré par n et p.

Comme l'anneau A est un peu compliqué, on va regarder ces relations dans un de ses quotients qui est un corps. Soit ainsi Φ un facteur irréductible du polynôme $(X^r-1)/(X-1)$ dans $\mathbf{F}_p[X]$ et soit K le corps fini $\mathbf{F}_p[X]/(\Phi)$. Notons x la classe de X dans K ; comme p ne divise pas n, c'est une racine primitive r-ième de l'unité. Par construction, on a $P(x^m) = P(x)^m$ dans K, pour tout polynôme $P \in \mathbf{F}_p[X]$ de la forme $\prod_{s\in S}(X-s)^{e_s}$.

LEMME 3.3.7. — *Pour tout $s \in S$, $x - s \in K^*$. De plus, le sous-groupe de K^* engendré par ces éléments est de cardinal au moins égal à $2^{\min(|E|,|S|)}$.*

Démonstration. — Supposons $x - s = 0$ dans K. Alors, $x^m - s = (x-s)^m = 0$ pour tout élément m du sous-groupe E. En particulier, $x = x^m$ pour tout $m \in E$. Comme x est

d'ordre r, on a $m = 1 \pmod r$ pour tout $m \in E$, en particulier $n = 1 \pmod r$, d'où $M = 1$, ce qui est absurde.

Nous allons montrer que les éléments de la forme $\prod_{s\in S}(x-s)^{e_s}$ de K^*, où $(e_s)_{s\in S}$ est une famille d'entiers naturels de somme $< |E|$ sont deux à deux distincts. Dans le cas contraire, il existe deux polynômes $P = \prod(X-s)^{e_s}$ et $Q = \prod(X-s)^{f_s}$, où $\sum e_s < |E|$, $\sum f_s < |E|$, tels que $P(x) = Q(x)$. Pour tout $m \in E$, on en déduit $P(x^m) = P(x)^m = Q(x)^m = Q(x^m)$, si bien que le polynôme $P - Q$ s'annule en chacun des éléments x^m de K, pour $0 \leqslant m < |E|$. Comme $P - Q$ est de degré $< |E|$, cela entraîne $P = Q$. Comme $(s - s', n) = 1$ pour s et s' distincts dans S, on a $s \neq s' \pmod p$; l'égalité des polynômes P et Q entraîne $e_s = f_s$ pour tout $s \in S$.

D'après le lemme combinatoire 3.3.5, l'ensemble des telles familles (e_s) est de cardinal $\binom{|E|+|S|-1}{|S|} > 2^{\min(|S|,|E|)}$. □

Soit G le sous-groupe de K^* engendré par les $x - s$ pour $s \in S$. Comme K^* est un groupe cyclique, il existe un polynôme $P \in \mathbf{F}_p[X]$ de la forme $\prod(X-s)^{e_s}$ tel que $P(x)$ soit un générateur de G.

L'ensemble des couples (i, j) tels que $0 \leqslant i, j \leqslant \sqrt{|E|}$ est de cardinal $(1 + \lfloor |E| \rfloor^{1/2})^2 > |E|$. Par conséquent, il existe deux couples distincts (i, j) et (k, ℓ) tels que les éléments $m = n^i p^j$ et $m' = n^k p^\ell$ du groupe E soient égaux, c'est-à-dire $m \equiv m' \pmod r$. Si $m = m'$, alors $n^{i-k} = p^{\ell-j}$ est une puissance de p ; nécessairement, n est aussi une puissance de p.

Supposons donc $m \neq m'$. Comme $P(x)^m = P(x^m) = P(x^{m'}) = P(x)^{m'}$, il vient $P(x)^{m'-m} = 1$ si bien que $m' - m$ est multiple de l'ordre de $P(x)$. Puisque $P(x)$ engendre G, on en déduit l'inégalité $\left|m' - m\right| \geqslant |G|$. On a alors

$$2^{\min(|S|,|E|)} \leqslant |G| \leqslant \left|m' - m\right| \leqslant n^{2\sqrt{|E|}},$$

soit encore

$$\min(|S|, |E|) \leqslant 2\sqrt{|E|}\log_2(n).$$

Si $|E| \leqslant |S|$, on a alors $|E| \leqslant 4\log_2(n)^2$, ce qui contredit l'inégalité $M > 4\log_2(n)^2$, étant donné que $M \leqslant |E|$. On a donc $|E| \geqslant |S|$, donc $|S| \leqslant 2\sqrt{|E|}\log_2(n)$. Mais E est un sous-groupe de $(\mathbf{Z}/r\mathbf{Z})^*$, d'où $|E| < r$ et l'inégalité $|S| < 2\sqrt{r}\log_2(n)$, qui est aussi absurde. □

Le théorème 3.3.3 se déduit de la proposition 3.3.6, appliquée à $S = \{0, 1, \ldots, r-1\}$. Comme aucun nombre premier $\leqslant r$ n'est supposé diviser n, les différences d'entiers de S sont premières à n. Par ailleurs, l'ordre de n modulo r est bien supérieur à $4\log_2(n)^2$. Enfin, puisque $|S| = r$, l'inégalité $|S| > 2\sqrt{r}\log_2(n)$ équivaut à l'inégalité $r \geqslant 4\log_2(n)^2$, laquelle est satisfaite car l'ordre de n modulo r divisant $r - 1$, $r - 1 \geqslant 4\log_2(n)^2$.

Pour terminer la démonstration du théorème 3.3.3, reste à démontrer l'existence d'un nombre premier r supérieur ou égal au plus petit entier vérifiant les conditions

a) et *b*) de ce théorème, ce qui fait l'objet de la proposition suivante appliquée avec $B = 4\log_2(n)^2$.

LEMME 3.3.8. — *Soit n un entier tel que $n \geqslant 2$. Soit B un entier > 1. Il existe un nombre premier p vérifiant les trois propriétés suivantes :*

a) *p ne divise pas n ;*

b) *l'ordre de n modulo p est au moins égal à B ;*

c) $p \leqslant \mathrm{O}(B^2 \log n)$.

Démonstration. — Posons $A = n\prod_{i=1}^{B}(n^i - 1)$ et soit p le plus petit nombre premier qui ne divise pas A. Il s'ensuit que p ne divise pas n et que $n^i \not\equiv 1 \pmod{p}$ pour $i \in \{1, \ldots, B\}$; en particulier, la classe de n dans $(\mathbf{Z}/p\mathbf{Z})$ est inversible et est d'ordre $> B$. D'autre part, tout nombre premier $q < p$ divise A, par hypothèse, donc $\prod_{q<p} q \leqslant A$, ce qui entraîne

$$\sum_{q<p} \log q \leqslant \log A \leqslant \left(1 + \frac{B(B+1)}{2}\right)\log n.$$

D'après le théorème des nombres premiers, ou plutôt l'inégalité de Tchébitcheff, plus élémentaire, (prop. 3.3.10), il existe un nombre réel $c > 0$ tel que le membre de gauche soit minoré par cp (l'inégalité stricte n'a pas d'incidence, car $\log p = \mathrm{o}(p)$). Par suite, $p \leqslant \mathrm{O}(B^2 \log n)$. □

Il nous reste à démontrer l'inégalité de Tchébitcheff (proposition 3.3.10). La démonstration suivante, tirée de Nair (1982), passe par l'étude du plus petit multiple commun des entiers de 1 à n, mais est un peu plus simple que la preuve habituelle, telle qu'on la trouve par exemple dans Hardy & M.Wright (1960).

LEMME 3.3.9. — *Pour tout entier m tel que $m \geqslant 1$, le plus petit multiple commun des entiers $1, \ldots, m$ est supérieur ou égal à 2^{m-1}.*

Démonstration. — Pour tout $m \in \mathbf{N}$, notons A_m le ppcm des entiers de 1 à m. Pour $m, n \in \mathbf{N}$, on pose $B(m,n) = \int_0^1 x^m(1-x)^n\,dx$; par intégration par parties successives, on obtient, si $n \geqslant 1$,

$$\begin{aligned} B(m,n) &= \frac{n}{m+1}B(m+1,n-1) = \cdots = \frac{n!}{(m+1)\ldots(m+n)}B(m+n,0) \\ &= \frac{n!m!}{(m+n+1)!} = \left((m+1)\binom{m+n+1}{n}\right)^{-1} = \left((m+n+1)\binom{m+n}{n}\right)^{-1}. \end{aligned}$$

Par ailleurs, en développant $(1-x)^n$ par la formule du binôme, on obtient

$$B(m,n) = \sum_{k=0}^{n}(-1)^k\binom{n}{k}\int_0^1 x^{m+k}\,dx = \sum_{k=0}^{n}(-1)^k\binom{n}{k}\frac{1}{m+k+1},$$

si bien que le dénominateur de $B(m,n)$ est divisible par A_{m+n+1}. Autrement dit, $(m+1)\binom{m+n+1}{n} = (m+n+1)\binom{m+n}{n}$ divise A_{m+n+1}.

En prenant $m = n$, on observe que A_{2n+1} est multiple de $(2n+1)\binom{2n}{n} = 1/B(n,n)$. Puisque $x(1-x) < 1/4$ pour $0 \leqslant x \leqslant 1$ et $x \neq 1/2$, on a $B(n,n) < 4^{-n}$, d'où $A_{2n+1} > 2^{2n}$, ce qui démontre l'inégalité $A_m > 2^{m-1}$ pour $m \geqslant 3$ impair. En prenant $m = n+1$, on voit alors que que A_{2n+2} est multiple de $(2n+2)\binom{2n+1}{n} > (1+1)^{2n+1} = 2^{2n+1}$, d'où l'inégalité $A_m > 2^{m-1}$ pour $m \geqslant 4$ pair. Au vu des égalités $A_1 = 1 = 2^0$ et $A_2 = 2 \geqslant 2^1$, on en déduit que $A_m \geqslant 2^{m-1}$ pour tout entier $m \geqslant 1$. □

PROPOSITION 3.3.10. — *Il existe un nombre réel $c > 0$ tel que l'on ait, pour tout entier $n \geqslant 2$, on ait l'inégalité*

$$\sum_{p \leqslant n} \log(p) \geqslant cn,$$

la somme étant restreinte aux nombres premiers p inférieurs ou égaux à n.

Démonstration. — La décomposition en facteurs premiers du ppcm A_n des entiers de 1 à n est de la forme $A_n = \prod_{p \leqslant n} p^{a_p}$, où a_p est la plus grande puissance de p inférieure ou égale à n : $a_p = \lfloor \log(n)/\log(p) \rfloor$; Elle vaut 1 si $n < p^2$, 2 si $p^2 \leqslant n < p^3$, etc. Par conséquent,

$$\begin{aligned}\log(A_n) &= \sum_{n^{1/2} < p \leqslant n} \log p + 2 \sum_{n^{1/3} < p \leqslant n^{1/2}} \log p + \dots \\ &= \sum_{k \geqslant 1} \left(\sum_{p \leqslant n^{1/k}} \log(p) \right) = \sum_{k \geqslant 1} \theta(n^{1/k}),\end{aligned}$$

où, pour tout nombre réel x, on a posé $\theta(x) = \sum_{p \leqslant x} \log p$. Le premier terme de cette somme est $\theta(n)$, les autres sont inférieurs ou égaux à $\theta(\sqrt{n}) \leqslant \sqrt{n}\log\sqrt{n} \leqslant \frac{1}{2}\sqrt{n}\log n$. Comme $n^{1/k} < 2$ pour $k > \log(n)/\log(2)$, il y a au plus $\log(n)/\log(2)$ termes, ce qui entraîne

$$\theta(n) \geqslant \log(A_n) - \frac{\log(n)}{2\log(2)}\sqrt{n}\log(n).$$

D'où, d'après le lemme précédent

$$\theta(n) \geqslant n\log 2 + \mathrm{o}(n).$$

□

§3.4. Exercices

Exercice 3.4.1. — a) Démontrer que la classe P est stable par réunion, intersection et passage au complémentaire.

b) Démontrer que la classe NP est stable par réunion et intersection.

Exercice 3.4.2. — Démontrer que les classes P et NP sont stables par répétition.

Exercice 3.4.3. — On note PSPACE la réunion des classes SPACE(n^k), pour $k \geqslant 1$.

a) Démontrer que la classe PSPACE est stable par union, intersection et répétition.

b) Démontrer que PSPACE est contenu dans la réunion EXP des classes SPACE(e^{kn}), pour $k \geqslant 1$.

c) Démontrer que NP est contenu dans PSPACE

On note NPSPACE la réunion des classes NSPACE(n^k), pour $k \geqslant 1$.

d) Démontrer que NPSPACE=PSPACE.

Exercice 3.4.4. — Pour qu'un langage $\mathscr{L}$ appartienne à NP, il vaut et il suffit qu'il existe une machine de Turing M de complexité polynomiale et un polynôme p tels que pour tout $x \in \Sigma^*$, $x \in \mathscr{L}$ si et seulement s'il existe un mot v de longueur $\leqslant p(\ell(x))$ tel que xv soit reconnu par M.

Exercice 3.4.5. — Montrer que le langage *CONNECTED* formé des descriptions de graphes connexes appartient à la classe P.

Exercice 3.4.6. — Dans un graphe orienté G, un *chemin* reliant deux sommets a et b de G est une suite d'arêtes $(e_1, \ldots, e_n)$ telles que a soit l'origine de e_1, e_2 relie l'extrémité de e_1 à l'origine de e_3, etc., et b soit l'extrémité de e_n. Un tel chemin est dit *hamiltonien* si chaque sommet de G autre que b est l'origine d'une flèche e_i et d'une seule.

Le langage *HAMPATH* est formé des descriptions de triplets (G, a, b) où G est un graphe orienté et a, b sont deux sommets de G pour lesquels il existe un chemin hamiltonien reliant a à b.

a) Démontrer que ce langage appartient à la classe NP.

b) Prouver qu'il est NP-complet.

Exercice 3.4.7. — Soit M une machine de Turing (à un seul ruban). Soit $x \in \Sigma^*$ un mot et soit m un entier naturel. La suite des états en m est la suite des états dans lesquels se trouve la machine M lorsque la tête vient de passer de la case m à la case $m+1$ ou inversement.

a) Soit w_1 et w_1' deux mots de même longueur m qui ont même suite d'états en m. Démontrer que pour tout mot w_2, M reconnaît $w_1 w_2$ si et seulement si elle reconnaît $w_1' w_2$.

Soit $n \in \mathbf{N}$. Pour $m \in \{1, \ldots, n\}$, on note $e(m)$ la longueur moyenne des suites d'états en m, pour des mots de longueur n.

b) Soit x un mot et soit n sa longueur. Remarquer que la somme, pour $m \in \{1, \ldots, n\}$, des longueurs des suites d'états en m est égale au temps que met la machine M avant de s'arrêter. En déduire que $\tau_M(n) \geqslant \sum_{m \leqslant n} e(m)$.

Exercice 3.4.8. — On suppose que M décide du langage $\mathscr{L}$ formé des mots de la forme $w\overline{w}$ (palindromes — mots dont la seconde moitié est la première mais écrite dans l'autre sens). Soit s le nombre d'états de M.

a) Pour $n \in \mathbf{N}$, combien $\mathscr{L}$ contient-il de mots de longueur n ?

b) On note $e(m)$ la moyenne des longueurs des suites d'états en m, lorsque x parcourt l'ensemble des mots de longueur n. Démontrer qu'au moins la moitié des mots x de longueur n, ont une suite d'états en m de longueur $\leqslant 2e(m)$. En déduire qu'au moins $2^{m-1}/s^{2e(m)+1}$ mots de longueur n ont la même suite d'états en m.

c) Prouver par l'absurde que $2^{m-1} \leqslant s^{2e(m)+1}$, d'où $e(m) \geqslant \frac{(m-1)\log 2}{2\log s} - \frac{1}{2}$.

d) Conclure que $\tau_M(n) \gg n^2$.

Exercice 3.4.9. — Construire une machine de Turing à deux rubans qui décide du langage des palindromes en temps $O(n)$.

Exercice 3.4.10. — Soit M une machine de Turing qui est un décideur. Démontrer que si la fonction τ_M n'est pas bornée, alors $\tau_M(n) \geqslant n$ pour une infinité d'entiers n.

Exercice 3.4.11. — Soit M une machine de Turing qui est un décideur. On suppose que $\tau_M(n)$ n'est pas $O(n)$.

a) Pour un mot $x \in \Sigma^*$ en entrée de M, notons $s(x)$ le temps maximal dans lequel la machine visite une case donnée du ruban. Démontrer que pour tout entier n, il existe un mot x_n de plus petite longueur tel que $s(x_n) \geqslant n$.

b) Démontrer par l'absurde qu'il n'existe pas trois entiers m_1, m_2, m_3 tels que $1 \leqslant m_1 < m_2 < m_3 \leqslant \ell(x_n)$ en lesquels les suites des états coïncident. (Sinon, écrire $x_n = w_1w_2w_3w_4$, où w_1 est de longueur m_1, x_2 de longueur $w_2 - w_1$, w_3 de longueur $m_3 - m_2$ et démontrer que pour l'un des deux mots $x = w_1w_3w_4$ ou $x = w_1w_2w_4$, on a $s(x) = s(x_n)$.)

c) Conclure que le nombre de suites d'états de M, lorsqu'elle reçoit en entrée x_n, est au plus égal à $(\ell(x_n) - 1)/2$.

d) Soit s le nombre d'états de M. Démontrer que pour tout entier $m \geqslant 1$, au plus m suites d'états distinctes ont une longueur $\leqslant \log(m)/\log(s)$.

e) Démontrer que $\tau(x_n) \geqslant \frac{\ell(x_n)-1}{4}\log_m(\frac{\ell(x_n)+1}{4})$. En conclure que $\tau_M(n) \gg n\log(n)$.

Exercice 3.4.12. — Soit M une machine de Turing (à un seul ruban) telle que $\tau_M(n) = o(n\log(n))$. Soit s le nombre d'états de M.

On suppose par l'absurde que les longueurs des suites d'états de M ne sont pas bornées. Soit A un entier soit g la fonction définie par $g(n) = \sqrt{n/\log(n)\tau_M(n)}$.

a) Démontrer que $g(n) = o(\log(n))$, mais que $ng(n)/\tau_M(n)$ tend vers l'infini.

Soit x un mot de longueur minimale n possédant une suite d'états (en m) de longueur $\geqslant A$ et soit h le nombre de positions en lesquelles les suites d'états sont de longueur $< g(n)$.

b) Démontrer par l'absurde que $h > 0$: sinon, on aurait $\tau_M(n) \geqslant ng(n)$. En déduire que $\tau_M(n) \geqslant A + (n-h)g(n)$ puis que $h > n + (A - \tau_M(n))/g(n)$.

b) Observer que pour A bien choisi, on a $h > 3s^{g(n)}$. En déduire qu'il existe quatre positions m_1, m_2, m_3, m_4 en lesquelles les suites d'états coïncident. Au moins deux d'entre elles, disons m_1 et m_2, sont toutes deux ou bien $> m$, ou bien $< m$.

c) Soit x' le mot obtenu en enlevant de x la partie située entre les positions $m_1 + 1$ et m_2. Démontrer que x' possède une suite d'états de longeur $\geqslant A$.

d) En déduire que toute suite d'états de M est de longueur au plus A.

e) Soit $\sim$ la relation d'équivalence sur Σ^* pour laquelle $x \sim y$ si pour tout $z \in \Sigma^*$, M accepte xz si et seulement si elle accepte yz.

Pour $x \in \Sigma^*$, soit $S(x)$ l'ensemble des suites d'états en $\ell(x)$. Si x et y ont même longueur et satisfont $S(x) = S(y)$, prouver que $x \sim y$.

f) Déduire du théorème de Myhill-Nerode que le langage $\mathscr{L}(M)$ est régulier.

Exercice 3.4.13. — On dit qu'une fonction $f\colon \mathbf{N} \to \mathbf{N}$ est t-constructible s'il existe une machine de Turing M à deux rubans qui, si le ruban en entrée est le mot 1^n (pour $n \in \mathbf{N}$), écrit le développement binaire de $f(n)$ sur le second ruban en temps $\mathrm{O}(f(n))$.

Démontrer que les fonctions $n \mapsto n^2$, $n \mapsto 2^n$ sont t-constructibles, de même que la fonction $n \mapsto n\lfloor \log(n) \rfloor$.

Exercice 3.4.14. — Soit f une fonction t-constructible. Le but de cet exercice est de construire un langage qui appartient à $\mathrm{TIME}(f(n))$, mais qui n'est pas décidable en temps $\mathrm{o}(f(n)/\log(f(n)))$.

Soit D la machine suivante. Si le mot en entrée w n'est pas de la forme $\langle M \rangle 01^n$, où $\langle M \rangle$ est la description d'une machine de Turing et n est un entier naturel, D refuse w. Si n est la longueur de w, la machine simule M sur le mot w pendant $f(n)/\log f(n)$ étapes. Si cela a suffi pour rejeter w, alors D accepte w, sinon D refuse w.

Démontrer que le langage décidé par D n'est pas décidable en temps $\mathrm{o}(f(n)/\log(f(n)))$ mais qu'il l'est en temps $\mathrm{O}(f(n))$.

Exercice 3.4.15. — Déduire de l'exercice précédent les énoncés suivants.

a) Si a et b sont des nombres réels tels que $b > a \geqslant 1$, alors $\mathrm{TIME}(n^a) \subsetneq \mathrm{TIME}(n^b)$.

b) La classe P n'est pas égale à la classe EXP.

Exercice 3.4.16. — En s'inspirant de l'exercice 3.4.14, définir la notion de fonction s-constructible (s est pour *space*) et démontrer que si f et g sont des fonctions s-constructibles telles que $f(n) = \mathrm{o}(g(n))$, alors $\mathrm{SPACE}(f(n)) \neq \mathrm{SPACE}(g(n))$.

BIBLIOGRAPHIE

M. AGRAWAL, N. KAYAL & N. SAXENA (2004), « PRIMES is in P ». *Ann. of Math.*, **169** (2), p. 781–793.

M. R. GAREY & D. S. JOHNSON (1979), *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman.

G. H. HARDY & E. M.WRIGHT (1960), *An introduction to the theory of numbers*. Oxford University Press, 4th édition.

M. NAIR (1982), « On Chebyshev inequalities for primes ». *Amer. Math. Monthly*, **89** (2), p. 126–129.

V. R. PRATT (1975), « Every prime has a succinct certificate ». *SIAM J. Comput.*, **4** (3), p. 214–220.

M. SIPSER (2006), *Introduction to the Theory of Computation*. Thomson Course Technology, 2^{e} édition.

A. TURING (1936), « On computable numbers, with an application to the Entscheidungsproblem. » *Proc. Lond. Math. Soc., II. Ser.*, **42**, p. 230–265.

A. TURING & J.-Y. GIRARD (1995), *La machine de Turing*. Numéro 131 in Points Sciences, Seuil.

TABLE DES MATIÈRES