Langages formels, Calculabilité, Complexité

Année scolaire 2004–2005
premier semestre

Olivier Carton

Magistère de Mathématiques Fondamentales & Appliquées et d'Informatique

École Normale Supérieure

Rédacteurs

Gaëtan Bisson

François Garillot

Thierry Martinez

Sam Zoghaib

Table des matières

Première partie

Langages formels

Chapitre 1

Langages rationnels

1.1 Premières définitions

On consultera pour cette section [Per90].

Définition 1.1.1.
- Un *alphabet* est un ensemble fini (souvent noté Σ ou A) dont les éléments sont appelés *lettres*.
- Les *mots* sur l'alphabet A sont les suites finies d'éléments de A (on les notera : $u = u_1u_2 \dots u_n$ avec $\forall i, u_i \in A$).
- La *longueur* d'un mot $u = u_1u_2 \dots u_n$ est n, on la note $|u|$.
- Le *mot vide* noté ε ou 1 correspond à l'unique mot de longueur 0.
- Un *langage* sur l'alphabet A est un ensemble de mots sur A.
- On note A^* l'*ensemble de tous les mots* sur l'alphabet A.

Définition 1.1.2 (Concaténation).
La *concaténation* correspond à la mise bout à bout de deux mots. Il s'agit de la loi de composition :

$$u = u_1u_2 \dots u_n, v = v_1v_2 \dots v_m \mapsto uv = u \cdot v = u_1u_2 \cdots u_nv_1v_2 \dots v_m$$

Elle est associative ($(uv)w = u(vw) = uvw$) et il existe un élément neutre (ε).

Exemple.
$u = abb, v = bbbb, uv = ab^6$

Définition 1.1.3 (Monoîde libre).
L'ensemble de tous les mots sur A, muni de la concaténation, est appelé *monoîde libre* sur A.

Notation.
Le monoïide libre sur A est noté A^*. Cette notation sera justifiée par la suite.

1.2 Opérations rationelles

Définition 1.2.1 (Union, Produit).
- L'*union* est l'opération qui à deux langages L et L' associe le langage $L + L' = L \cup L'$.

– Le *produit* est l'opération qui à deux langages L et L' associe le langage $L \cdot L' = LL' = \{uv \mid u \in L \text{ et } v \in L'\}$.

Exemple.
Si on pose $L = \{u \in A^* \mid |u| \text{ pair }\}$ et $L' = \{u \in A^* \mid |u| \text{ impair }\}$, alors on a :

– $L + L' = A^*$ – $LL' = L' = L'L$
– $L'L' = L \setminus \{\varepsilon\}$ – $LL = L$ (L est un sous-monoïde de A^*)

Définition 1.2.2 (Étoile).
Soit $L \subseteq A^*$, on définit :

$$L^0 = \{\varepsilon\}, \qquad L^{i+1} = LL^i, \qquad L^* = \bigcup_{i \geq 0} L^i$$

Remarque.
En général $L^i \neq \{u^i \mid u \in L\}$.

Exemple.
Si $L = \{a, ba\}$, alors L^* est constitué de tous les mots dans lesquels chaque b est suivi d'un a.

Remarque.
Ceci justifie donc *a posteriori* la notation A^*.

1.3 Langages rationnels

Définition 1.3.1 (Langages rationnels).
La classe $\mathcal{R}$ des *langages rationnels* (sur A) est la plus petite famille de langages telle que :
– $\emptyset \in \mathcal{R}$ et $\forall a \in A, \{a\} \in \mathcal{R}$;
– $\mathcal{R}$ est close pour les opérations rationnelles (l'union, le produit et l'étoile).

Exemple.
– Le langage A est rationnel puisqu'il s'écrit $A = \bigcup_{a \in A} \{a\}$.
– $L = \{u \in A^* \mid |u| \text{ pair }\}$ est rationnel puisqu'il s'écrit $L = (AA)^* = (A^2)^*$.
– $L' = \{u \in A^* \mid |u| \text{ impair }\}$ est rationnel puisqu'il s'écrit $L' = AL$.

Définition 1.3.2 (Expressions rationnelles).
La classe $\mathcal{E}$ des *expressions rationnelles* est la plus petite famille d'expressions telles que :
– $\forall a \in A, \{a\} \in \mathcal{E}$;
– pour tout couple d'expressions (E, E') de $\mathcal{E}$, les expressions $E + E'$, $E \cdot E'$ et E^* sont encore dans $\mathcal{E}$

Remarque.
Notons qu'ici, $+$, $\cdot$ et $*$ sont des symboles inertes et non des opérations.

Notation.
On omettra les accolades lorsqu'on notera les singletons (dans des expressions rationelles) :
– $L = \{a\} + \{ba\} = a + ba$ et $L^* = (a + ba)^*$

– $A = a_1 + a_2 + \cdots + a_n$

Exemple.

A^*	tous les mots (sur A)
aA^*	mots qui commencent par a ;
A^*a	mots qui finissent par a ;
$(aa + b)^*$	mots dans lesquels les facteurs maximaux formés de a consécutifs sont de longueur paire ;
$(ab^*a + b)^*$	mots qui ont un nombre pair de a.

Définition 1.3.3 (Automate).

Un *automate* $\mathcal{A}$ sur l'alphabet A est un quintuplet (Q, A, E, I, F) où Q est fini, $I \subseteq Q$, $F \subseteq Q$ et $E \subseteq Q \times A \times Q$. On appelle les éléments de Q les états, ceux de I les états initiaux, ceux de F les états finaux et ceux de E les transitions.

Notation.

On notera les transitions : $p \xrightarrow{a} q$ (au lieu de $(p, a, q) \in E$).

Exemple.

$\mathcal{A} = (\{1, 2\}, \{a, b\}, \{(1, b, 1), (1, a, 2), (2, b, 2), (2, a, 1)\}, \{1\}, \{1\})$

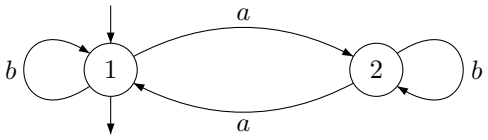


FIG. 1.1 – Exemple d'automate

Définition 1.3.4 (Chemin).

Un *chemin* (dans l'automate (Q, A, E, I, F)) est une suite finie de transitions consécutives : $p_i \xrightarrow{a_{i+1}} p_{i+1}$. On le note :

$$p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} p_{n-1} \xrightarrow{a_n} p_n$$

On dit que p_0 est l'*état de départ* et p_n, l'*état d'arrivée*.

Exemple.

La suite $2 \xrightarrow{a} 1 \xrightarrow{b} 1 \xrightarrow{b} 1 \xrightarrow{a} 2 \xrightarrow{b} 2$ est un chemin dans l'automate de l'exemple précédent.

Définition 1.3.5 (Étiquette d'un chemin).

Par définition, l'*étiquette* du chemin $p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} p_{n-1} \xrightarrow{a_n} p_n$ est le mot $a_1 a_2 \ldots a_{n-1} a_n$.

Notation.

On peut étendre la relation "$\rightarrow$" aux mots en notant (dans un automate fixé) $p \xrightarrow{w} q$ l'existence d'un chemin de p à q étiqueté par w.

Exemple.

L'étiquette du chemin de l'exemple précédent est *abbab*.

Définition 1.3.6 (Chemin acceptant).
Un chemin est dit *acceptant* ou *réussi* lorsque l'état de départ est initial et l'état d'arrivée est final.

Définition 1.3.7 (Mots acceptés).
Un mot est dit *accepté* par l'automate $\mathcal{A}$ s'il est l'étiquette d'un chemin acceptant de $\mathcal{A}$.

Notation.
On note $L(\mathcal{A})$, le langage des mots acceptés par $\mathcal{A}$.

Théorème 1.3.1 (Kleene, 1956).
Un langage L est rationnel si et seulement s'il existe un automate (fini) $\mathcal{A}$ tel que $L = L(\mathcal{A})$.

Preuve.
Montrons par induction (sur la longueur de l'expression rationelle) que si L est rationnel alors $L \setminus \{\varepsilon\}$ est accepté par un automate (ensuite, si besoin est, on ajoute l'état initial dans les états finaux pour que ε soit accepté, une fois l'automate normalisé).
– $L = a$:

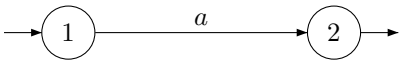


FIG. 1.2 – Preuve de Kleene : automate décrivant a

– $L = L' + L''$: voir FIG. 1.4, page 13
– $L = L'L''$: voir FIG. 1.5, page 13
– $L = (L')^*$: voir FIG. 1.6, page 13

Pour l'autre sens de l'équivalence, on renvoie le lecteur aux algorithmes ci-dessous. □

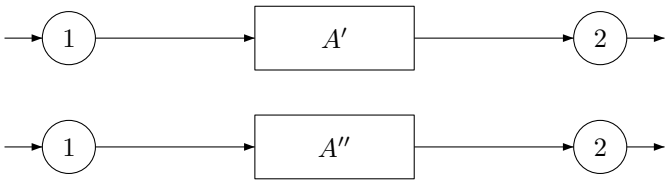


FIG. 1.3 – Preuve de Kleene : automates de base L' et L''

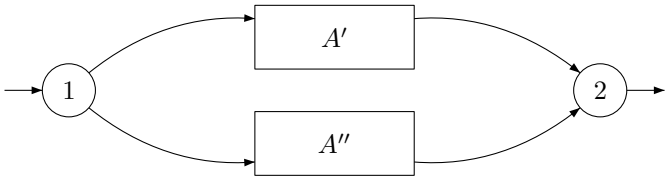


FIG. 1.4 – Preuve de Kleene : union de deux automates

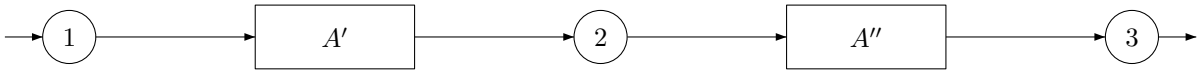


FIG. 1.5 – Preuve de Kleene : produit de deux automates

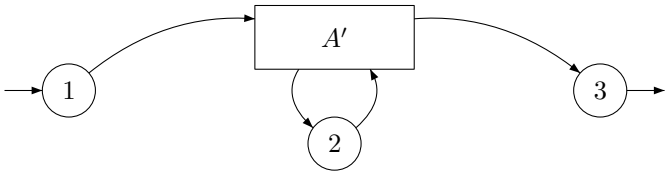


FIG. 1.6 – Preuve de Kleene : étoile d'un automate (l'état 2 s'obtient par la fusion des états finaux et initiaux de A', puisqu'on travaille sans ε-transitions)

Algorithme (McNaugton-Yamada).

On suppose $Q = \{1, \ldots, n\}$, et on pose :

$$L^k_{q,q'} = \{a_1 a_2 \ldots a_{n-1} a_n \mid n \in \mathbb{N} \text{ et } q \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q' \text{ avec } \forall i, p_i \in \{1, 2, \ldots, k\}\}$$

Ainsi, on a : $L(\mathcal{A}) = \bigcup_{i \in I, f \in F} L^n_{i,f}$. Or

$$L^0_{q,q'} = \{a \mid (q, a, q') \in E\}$$
$$L^{k+1}_{q,q'} = L^k_{q,q'} + L^k_{q,k+1}(L^k_{k+1,k+1})^* L^k_{k+1,q'}.$$

Ceci permet de conclure par récurrence sur k.

Algorithme (Méthode par élimination).

Voir FIG. 1.7, page 15 et suivantes (sur cet exemple, on trouve finalement $L = (a + ab^*a)^*$).

Lemme 1.3.2 (Lemme d'Arden).

Soit à résoudre l'équation en le langage X : $X = KX + L$. Alors :
- *si $\varepsilon \notin K$, l'unique solution est $X = K^*L$.*
- *si $\varepsilon \in K$, les solutions sont de la forme $X = K^*(B + L)$, où $B \subseteq A^*$.*

Preuve.

Notons que K^*L est solution de l'équation : en effet $K^*L = K(K^*L) + L$.

Soit maintenant X une solution de l'équation $X = KX + L$. On a $L \subseteq X$ et, par récurrence immédiate, $K^*L \subseteq X$.

Pour montrer l'autre inclusion, on distingue deux cas :
- si $\varepsilon \notin K$, montrons par l'absurde que $K^*L = X$ en supposant que $X \setminus K^*L$ est non vide.
  Soit donc u, un mot de $X \setminus K^*L$ de longueur minimale. Comme $u \notin L$, ce mot s'écrit $u = kx$ avec $k \in K$ et $x \in X$; alors $x \notin K^*L$ car si ce n'était pas le cas, on aurait $u \in K^*L$. Or $k \neq \varepsilon$; donc $|x| < |u|$ ce qui contredit la minimalité de $|u|$. La contradiction montre que $X \setminus K^*L$ est vide, ce qui est le résultat attendu.
- si $\varepsilon \in K$, remarquons d'abord que pour tout $B \subseteq A^*$, $K^*(B + L)$ est solution de l'équation.
  Montrons alors que si X est solution de l'équation on a $X = K^*(X+L)$. Le résultat $X \subseteq K^*(X + L)$ est immédiat. Mais on a aussi, X étant solution de l'équation, $L \subseteq X$. Ce qui donne $X = X + L$; et ainsi, par récurrence immédiate, $K^*(X + L) \subseteq X$.

□

Algorithme (Méthode de Gauss).

Pour $q \in Q = \{1, \ldots, n\}$, soit X_q l'ensemble des mots étiquettes d'un chemin de q à un état final. Alors, pour trouver $L(\mathcal{A}) = \bigcup_{i \in I} X_i$, on résout, grâce au lemme d'Arden, le système :

$$\forall i, X_i = \begin{cases} \sum_{(i,a,j) \in E} aX_j + \varepsilon \text{ si } i \text{ est final} \\ \sum_{(i,a,j) \in E} aX_j \text{ sinon} \end{cases}$$

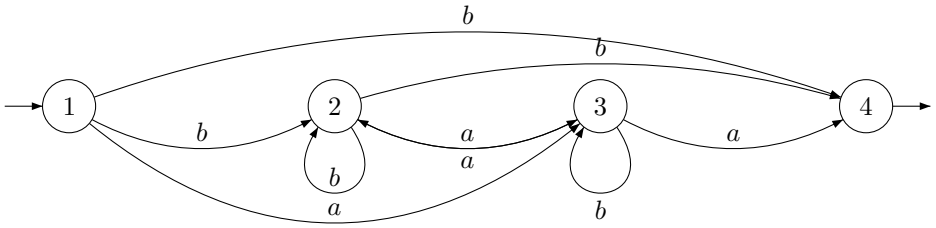


FIG. 1.7 – Méthode par élimination : on normalise.

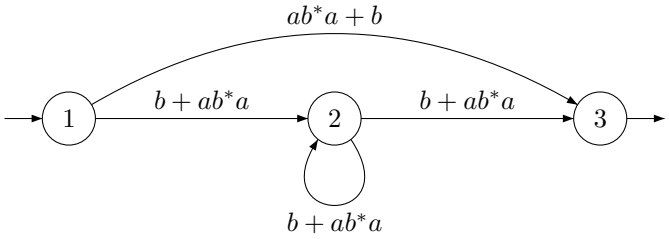


FIG. 1.8 – Méthode par élimination : nouvelle normalisation

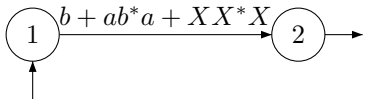


FIG. 1.9 – Méthode par élimination : on conclut (avec $X = b + ab^*a$).

1.4 Automates déterministes

Définition 1.4.1 (Automate déterministe).
Un automate $\mathcal{A} = (Q, A, E, I, F)$ est *déterministe* si :
– $|I| = 1$;
– si (p, a, q) et $(p, a, q') \in E$ alors $q = q'$.

Voir FIG. 1.10, page 16 et FIG. 1.11, page 16.

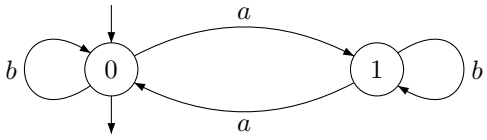


FIG. 1.10 – Automate déterministe

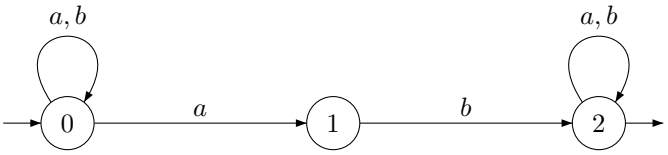


FIG. 1.11 – Automate non déterministe (langage reconnu : A^*abA^*)

Proposition 1.4.1.
Tout automate est équivalent à (accepte le même langage que) un automate déterministe.

Preuve.
Soit $\mathcal{A} = (Q, A, E, I, F)$. On pose

$$\hat{\mathcal{A}} = \big(\mathfrak{P}(Q), A, \hat{E}, \hat{I} = \{I\}, \hat{F} = \{P \subseteq Q \mid P \cap F \neq \emptyset\}\big)$$
$$\text{avec } \hat{E} = \big\{(P, a, P') \mid P' = \{q \mid \exists p \in P, p \xrightarrow{a} q\}\big\}.$$

Ainsi, $\hat{\mathcal{A}}$ est déterministe et accepte le même langage que $\mathcal{A}$ (comme le montre le lemme suivant).
Voir FIG. 1.12, page 18. □

Lemme 1.4.1.
Pour tout $w \in A^*$, il existe un chemin de I à P dans $\hat{\mathcal{A}}$ étiqueté par w si et seulement si $P = \{q \mid \exists i \in I, \xrightarrow{w} q$ dans $\mathcal{A}\}$

Définition 1.4.2 (Automate complet).
Un automate est dit *complet* lorsque $\forall p \in Q, \forall a \in A, \exists q \in Q, (p, a, q) \in E$.

Notation.

Dans un automate déterministe complet $\mathcal{A} = (Q, A, E, I, F)$, on notera (pour $q \in Q$ et $a \in A$) $q \cdot a$ l'unique état p tel que $(q, a, p) \in E$.

Proposition 1.4.2.

Tout automate est équivalent à un automate complet.

Preuve.

Il suffit d'introduire un état $p \notin Q$, le puit, et poser $\mathcal{A}' = (Q \cup \{p\}, A, E', I, F)$, avec $E' = E \cup \{(q, a, p) \mid q \in Q \cup \{p\} \text{ et } \not\exists q' \mid (q, a, q') \in E\}$. Notons que si $\mathcal{A}$ est déterministe, l'automate complété $\mathcal{A}'$ l'est également (voir FIG. 1.13, page 18). □

Corollaire 1.4.1 (Clôture par complémentation).

Si $L \subseteq A^*$ est rationnel, alors $A^* \setminus L$ est rationnel.

Preuve.

Si $\mathcal{A} = (Q, A, E, I, F)$ est déterministe et complet et $L(\mathcal{A}) = L$, alors, pour $\mathcal{A}' = (Q, A, E, I, Q \setminus F)$, $L(\mathcal{A}') = A^* \setminus L$. □

1.5 Quotients

Pour toute cette partie, on consultera [BBC93].

Définition 1.5.1 (Langages quotients).

Soit $L \subseteq A^*$. Le *quotient à gauche de L par $u \in A^*$* est $u^{-1}L = \{v \mid uv \in L\}$. Le *quotient à gauche de L par $K \subseteq A^*$* est $K^{-1}L = \bigcup_{k \in K} k^{-1}L$.

Proposition 1.5.1.

On a les relations suivantes :

- *pour $u \in A^*$, $u^{-1}(L + L') = u^{-1}L + u^{-1}L'$*
- *pour $a \in A$, $a^{-1}(LL') = (a^{-1}L)L' + \delta(L)a^{-1}L'$, avec $\delta(L) = \begin{cases} \varepsilon & \text{si } \varepsilon \in L \\ \emptyset & \text{sinon} \end{cases}$*
- *pour $a \in A$, $a^{-1}(L^*) = (a^{-1}L)L^*$*
- *pour $u, v \in A^*$, $(uv)^{-1}L = v^{-1}(u^{-1}L)$.*

Proposition 1.5.2.

Un langage L est rationnel si et seulement si il a un nombre fini de quotients à gauche.

On consultera également, pour ce critère de rationalité, [Sak03], P.127.

Preuve. Soit tout d'abord L, un langage rationnel accepté par un automate déterministe complet $\mathcal{A} = (Q, A, E, I, F)$. Pour $u \in A^*$, on note q_u le dernier état du chemin d'étiquette u partant de l'état initial. Alors

$$u^{-1}L = \{v \mid \text{il existe un chemin de } q_u \text{ à un état final étiqueté par } v\}$$

Ainsi $q_u = q_{u'} \Rightarrow u^{-1}L = u'^{-1}L$. Et donc $|\{u^{-1}L \mid u \in A^*\}| \leq |Q|$.

(Voir FIG. 1.14, page 18)

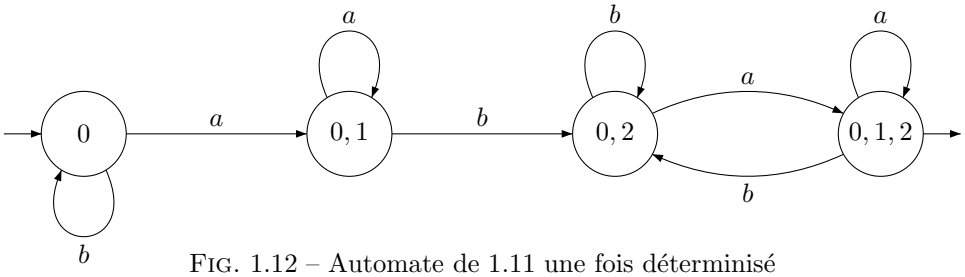


FIG. 1.12 – Automate de 1.11 une fois déterminisé

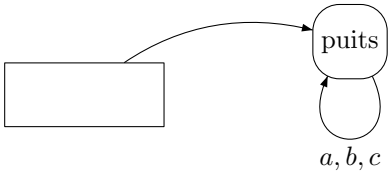


FIG. 1.13 – Ajout d'un état-puits sur l'alphabet $\{a, b, c\}$

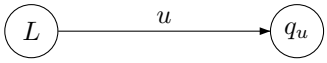


FIG. 1.14 – Quotientage à gauche

Soit à présent L un langage ayant un nombre fini de quotients à gauche. On définit un automate $\mathcal{A}_L = (Q, A, E, I, F)$ par :

$$\begin{aligned} Q &= \{u^{-1}L \mid u \in A^*\} \\ I &= \{L\} \\ F &= \{u^{-1}L \mid \varepsilon \in u^{-1}L\} = \{u^{-1}L \mid u \in L\} \\ E &= \{(u^{-1}L, a, (ua)^{-1}L = a^{-1}(u^{-1}L)) \mid u \in A^*, a \in A\}. \end{aligned}$$

On montre alors que $L \xrightarrow{u} u^{-1}L$, et, de là, que $\mathcal{A}_L$ accepte L. □

Exemple.

Soit $L = (ab)^*$. On a alors : $a^{-1}L = bL$, $b^{-1}L = \emptyset$, $a^{-1}(bL) = \emptyset$ et $b^{-1}(bL) = L$.

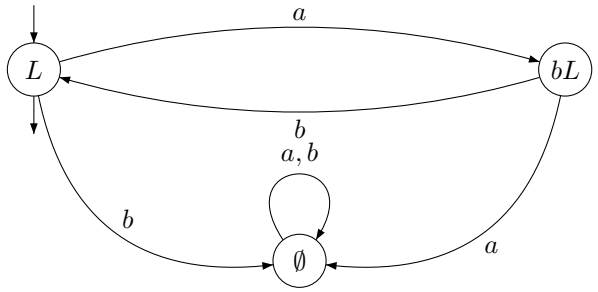


FIG. 1.15 – Exemple 1.5 : Automate minimal obtenu par l'énumération des quotients du langage

1.6 Morphismes d'automates

Définition 1.6.1 (Morphisme).

Soit $\mathcal{A} = (Q, A, E, I = \{i\}, F)$ et $\mathcal{A}' = (Q', A, E', I' = \{i'\}, F')$, deux automates déterministes. Une application $\mu : Q \to Q'$ est un *morphisme d'automates* si :

- $\mu(i) = i'$
- $\mu(q) \in F' \Leftrightarrow q \in F$
- $(q, a, q') \in E \Rightarrow (\mu(q), a, \mu(q')) \in E'$.

Proposition 1.6.1.

Soit $\mathcal{A}$ et $\mathcal{A}'$ deux automates déterministes complets et μ un morphisme de $\mathcal{A}$ vers $\mathcal{A}'$. Alors $L(\mathcal{A}) = L(\mathcal{A}')$.

Preuve.

Si $u = a_1 a_2 \cdots a_n \in L(\mathcal{A})$, on a :

$$i \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \to \cdots \xrightarrow{a_n} q_n \in F$$

et ainsi

$$i' = \mu(i) \xrightarrow{a_1} \mu(q_1) \xrightarrow{a_2} \mu(q_2) \to \cdots \xrightarrow{a_n} \mu(q_n) \in F'$$

Si $u = a_1 a_2 \cdots a_n \notin L(\mathcal{A})$, on a :

$$i \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \to \cdots \xrightarrow{a_n} q_n \notin F$$

et ainsi

$$i' = \mu(i) \xrightarrow{a_1} \mu(q_1) \xrightarrow{a_2} \mu(q_2) \to \cdots \xrightarrow{a_n} \mu(q_n) \notin F'$$

□

Définition 1.6.2 (Surjectivité).
Un morphisme $\mu : Q \to Q'$ est dit *surjectif* si $\mu(Q) = Q'$

Définition 1.6.3 (Quotient).
Un automate $\mathcal{A}'$ est dit *quotient* de l'automate $\mathcal{A}$ s'il existe un morphisme surjectif de $\mathcal{A}$ sur $\mathcal{A}'$.

Proposition 1.6.2.
L'automate $\mathcal{A}_L$ (défini dans la preuve de la proposition 1.5.2, 17) est un quotient de tout automate déterministe, complet et émondé (dont tous les états sont accessibles) qui accepte L.

Preuve.
Soit $\mathcal{A} = (Q, A, E, I, F)$ déterministe, complet et émondé.
Pour $q \in Q$, on définit :

$$\begin{aligned} L_q &= \{v \mid \text{il existe un chemin de } q \text{ à un état final étiqueté par } v\} \\ &= L(Q, A, E, \{q\}, F) \end{aligned}$$

Ainsi, s'il existe un chemin de i à q étiqueté par u alors $L_q = u^{-1}L$. Définissons μ par $\mu : q \in Q \mapsto L_q$ (et donc $\mu(i) = L$).

Alors, $q \in F \Leftrightarrow \varepsilon \in L_q \Leftrightarrow \mu(q)$ final dans $\mathcal{A}_L$. De plus, lorsque $i \overset{u}{\rightsquigarrow} q \xrightarrow{a} q'$ dans $\mathcal{A}$, on a :

$$\begin{aligned} L_q &= u^{-1}L \\ \text{donc } L_{q'} &= (ua)^{-1}L = a^{-1}(u^{-1}L) \end{aligned} \tag{1.1}$$

Ainsi, (L_q, a, L'_q) est une transition de $\mathcal{A}_L$, ce qui montre que μ est un morphisme d'automate (la surjectivité est évidente). □

Définition 1.6.4 (Congruence).
On appelle *congruence* sur $\mathcal{A}$ (automate déterministe complet) toute une relation d'équivalence $\sim$ sur Q qui vérifie (pour tout q, q' et a) :
– $q \sim q' \Rightarrow (q \in F \Leftrightarrow q' \in F)$
– $q \sim q' \Leftrightarrow q \cdot a \sim q' \cdot a$

Proposition 1.6.3.
Si μ est un morphisme de $\mathcal{A}$ sur $\mathcal{A}'$ et si $\mathcal{A}'$ est muni d'une congruence $\sim'$, alors, la relation $\sim$ définie par : $q \sim q' \Leftrightarrow \mu(q) \sim' \mu(q')$ est une congruence.

Proposition 1.6.4.
Si $\sim$ est une congruence sur $\mathcal{A}$, posons :

$$\begin{aligned} \mathcal{A}' &= \big(Q/\sim, A, E, I = \{[i]\}, F = \{[q] \mid q \in F\}\big) \\ \textit{avec } E &= \{([q], a, [qa]) \mid q \in Q\} \end{aligned} \tag{1.2}$$

Alors, $\mu : Q \to Q/\sim$ est un morphisme.

([q] dénote bien entendu la classe de l'état q pour la relation d'équivalence $\sim$)

Proposition 1.6.5.

La relation $\sim_N$ définie par : $q \sim_N q' \Leftrightarrow L_q = L_{q'}$ est une congruence appelée congruence de Nérode.

1.6.1 Calcul de la congruence de Nérode

Méthode itérative

Définissons :

$$\begin{aligned} q \sim_0 q' &\Leftrightarrow (q \in F \Leftrightarrow q' \in F) \\ q \sim_{i+1} q' &\Leftrightarrow q \sim_i q' \text{ et } \forall a \in A, q \cdot a \sim_i q' \cdot a \end{aligned} \tag{1.3}$$

Proposition 1.6.6.

Il existe $k \leq |Q|$ tel que $\sim_k = \sim_{k+1}$. De plus, $\sim_k = \sim_{k+n}$ pour tout $n \geq 0$ et $\sim_k$ est la congruence de Nérode.

Preuve.

Les deux premières affirmations sont évidentes et on conclut par récurrence. Si $q \not\sim_k q'$ alors (comme $\sim_k = \sim_{k+1}$), $qa \not\sim_k q'a$, et donc $L_q \neq L_{q'}$ d'où $q \not\sim_N q'$. Cela montre que $\sim_k$ est moins fine que la congruence de Nérode.

Mais, si $q \sim_k q'$, alors pour tout $u = a_1 \cdots a_n$ on a :

$$q \xrightarrow{a_1} \cdots \xrightarrow{a_n} q_n \in F \Leftrightarrow q' \xrightarrow{a_1} \cdots \xrightarrow{a_n} q'_n \in F$$

Ce qui implique que $q \sim_N q'$. □

Algorithme de Hopcroft

Définition 1.6.5 (Stabilité et coupure).

Soit $\mathcal{A} = (Q, A, E, I, F)$ un automate complet déterministe émondé, $a \in A$ et $B, C \subseteq Q$.

On dit que B est *stable* pour (C, a) si l'une des deux conditions suivantes est remplie :

- $B \cdot a \subseteq C$
- $B \cdot a \cap C = \emptyset$

où $B \cdot a = \{q \cdot a \mid q \in B\}$.

Sinon, on dit que (C, a) *coupe* B et on pose :

$$\begin{aligned} B_1 &= \{q \in B \mid q \cdot a \in C\} \\ B_2 &= \{q \in B \mid q \cdot a \notin C\} \end{aligned} \tag{1.4}$$

Remarque.

Ainsi, une partition $\{P_1, \ldots, P_p\}$ de Q compatible avec F est une congruence si et seulement si $\forall P_i, P_j, a \in A$, P_i est stable pour (P_j, a).

Proposition 1.6.7.

Soit $B \subseteq Q$, $C = C_1 \uplus C_2$ ($\uplus$ représente l'union disjointe), alors :

– *Si B est stable pour (C_1, a) et (C_2, a) alors B est stable pour (C, a).*
– *Si B est stable pour (C_1, a) et (C, a) alors B est stable pour (C_2, a).*

Algorithme.

$\mathcal{P} := (F, Q \setminus F)$
if $|F| \leq |Q \setminus F|$ **then**
  $\mathcal{S} := \{(F, a) \mid a \in A\}$
else
  $\mathcal{S} := \{(Q \setminus F, a) : a \in A\}$
end if
while $\mathcal{S} \neq \emptyset$ **do**
  $(C, a) :=$ un élément de $\mathcal{S}$
  $\mathcal{S} = \mathcal{S} \setminus \{(C, a)\}$
  if (C, a) coupe B en B_1, B_2 **then**
    remplacer B par B_1, B_2 dans $\mathcal{P}$
    for all $b \in A$ **do**
      (boucle en complexité proportionnelle à $|C||a|$)
      if $(B, b) \in \mathcal{S}$ **then**
        remplacer (B, b) par (B_1, b) et (B_2, b)
      else if $|B_1| \leq |B_2|$ **then**
        ajouter (B_1, b) à $\mathcal{S}$
      else
        ajouter (B_2, b) à $\mathcal{S}$
      end if
    end for
  end if
end while

Voir [BBC93]

Preuve.

Terminaison

Le nombre de coupures est borné par $|Q|$.

À la terminaison, la partition calculée correspond à la congruence de Nérode.

Validité

Notation.

$\mathcal{P}$ est stable pour (C, a) si $\forall P \in \mathcal{P}$, P est stable pour (C, a).

On a l'invariant suivant :

Proposition 1.6.8.

Pour tout $a \in A$ et $P \in \mathcal{P}$, P est combinaison booléenne de :
– *parties C telles que $\mathcal{P}$ est stable pour (C, a) ;*
– *parties C telles que $(C, a) \in \mathcal{S}$.*

On obtient donc bien une congruence. Il reste à montrer qu'il s'agit de la congruence de Nérode, cela se fait par récurrence.

Complexité

La complexité totale est en $\Theta\left(|A||Q|\log|Q|\right)$.

□

1.7 Lemmes de l'étoile

On les appelle également lemmes d'itération, ou de pompage. On consultera pour ce critère de rationalité [Sak03], p. 77.

Il existe une multitude de variantes pour ce lemme ; nous en présentons les principales. Leurs démonstrations sont toutes basées sur un usage plus ou moins fin du principe des tiroirs (les tiroirs étant les états d'un automate reconnaissant le langage), ou de sa version plus évoluée : le théorème de Ramsey.

Proposition 1.7.1 (Lemme de l'étoile faible 1).
Si L est rationnel, il existe un entier n tel que :

$$\begin{cases} |f| \geq n \\ f \in L \end{cases} \Rightarrow \begin{cases} \exists\,(u,v,w) \in A^*, v \neq \varepsilon \\ f = uvw \text{ et } uv^*w \subseteq L \end{cases}$$

Exemple.
Grâce à ce lemme, on peut montrer que $L = \{a^n b^n \mid n \geq 0\}$ n'est pas rationnel.

Par contre, $L \cup (A^*baA^*)$ vérifie la condition de rationnalité de ce lemme sans pour autant être rationnel – ce qui montre que cette condition n'est que suffisante.

Proposition 1.7.2 (Lemme de l'étoile faible 2).
Si L est rationnel, il existe un entier n tel que :

$$\left.\begin{array}{r} |f| \geq n \\ f \in L \end{array}\right\} \Rightarrow \begin{cases} \exists\,(u,v,w) \in A^*, v \neq \varepsilon \text{ et } |uv| \leq n \\ f = uvw \text{ et } uv^*w \subseteq L \end{cases}$$

Proposition 1.7.3 (Lemme de l'étoile faible 3).
Si L est rationnel, il existe un entier n tel que :

$$\left.\begin{array}{r} f = xyz \text{ avec } |y| \geq n \\ f \in L \end{array}\right\} \Rightarrow \begin{cases} \exists u,v,w \mid v \neq \varepsilon \\ y = uvw \text{ et } xuv^*wz \subseteq L \end{cases}$$

Proposition 1.7.4 (Lemme de l'étoile faible 4).
Si L est rationnel, il existe un entier n tel que :

$$\left.\begin{array}{r} f = xu_1 \cdots u_n z \\ f \in L \text{ et } (\forall i) u_i \in A^+ \end{array}\right\} \Rightarrow \begin{cases} \exists\,(i,j)\,, 0 \leq i < j \leq n \\ xu_1 \cdots u_i (u_{i+1} \cdots u_j)^* u_{j+1} \cdots u_n z \subseteq L \end{cases}$$

Malheureusement, cette condition n'est toujours pas nécessaire, comme le montre le langage $L = \{a^{i_1} b a^{i_2} b \cdots a^{i_n} b \mid \exists j, i_j \neq j\}$

Pour présenter la version la plus aboutie du lemme de l'étoile, nous avons besoin de quelques définitions.

Définition 1.7.1 (Propriétés σ_k et σ'_k).
L vérifie la *propriété σ_k (resp. σ'_k)* pour $k \geq 0$ si :

$$\forall i, u_i \in A^+,\ \forall f = xu_1u_2\cdots u_kz,\ \exists\,(i,j)\,, 0 \leq i < j \leq k \text{ tels que :}$$

$$f \in L \Leftrightarrow \forall n \geq 0,\ xu_1\cdots u_i(u_{i+1}\cdots u_j)^n u_{j+1}\cdots u_kz \in L$$

$$(\text{resp. } f \in L \Leftrightarrow xu_1\cdots u_iu_{j+1}\cdots u_kz \in L)$$

Proposition 1.7.5 (Lemme de l'étoile fort).
Soit L, un langage. Il y a équivalence entre :

1. *L est rationnel.*
2. *Il existe $k \in \mathbb{N}$ tel que L vérifie σ_k.*
3. *Il existe $k \in \mathbb{N}$ tel que L vérifie σ'_k.*

Notation.
Si E est un ensemble, on notera $\mathfrak{P}_k(E)$ l'ensemble des parties à k éléments de E.

On admettra le théorème suivant :

Théorème 1.7.1 (Ramsey).
Pour tout triplet d'entiers naturels (k, m, r), il existe un entier $N(k, m, r)$ tel que pour tout :
- *E, ensemble tel que $|E| \geq N$;*
- *C, ensemble tel que $|C| = m$;*
- *f, une fonction : $\mathfrak{P}_k(E) \to C$.*

il existe $F \subseteq E$ tel que :
- $|F| \geq r$;
- $|f(\mathfrak{P}_k(F))| \leq 1$.

Pour la preuve du théorème de Ramsey, on consultera [vLW92].

Preuve du lemme de l'étoile fort.

Trivialités

Tout d'abord, il est évident que (1.) $\Rightarrow$ (2.) et que (2.) $\Rightarrow$ (3.). Il reste donc à montrer que (3.) $\Rightarrow$ (1.).

Démarche

On montre dans un premier temps que si L satisfait σ'_k, alors $u^{-1}L$ satisfait σ'_k pour tout mot u.

Dans un deuxième temps, on montre que le nombre de langages vérifiant σ'_k est fini.

Finalement, avec ces deux résultats, on obtient qu'un langage vérifiant σ'_k a un nombre fini de quotients à gauche. La proposition 1.5.2 montre alors qu'il est rationnel.

Premier temps

Soit un mot u et un entier k, fixés. Supposons que L vérifie σ'_k.

Soit alors f, un mot qui s'écrit : $xu_1u_2\cdots u_kz$ avec $\forall i, u_i \neq \varepsilon$.

Ainsi, $f \in u^{-1}L \Leftrightarrow uf \in L \Leftrightarrow uxu_1\cdots u_i\cdots u_{j+1}\cdots u_kz \in L$.

Il est alors immédiat que $u^{-1}L$ satisfait σ'_k.

Deuxième temps

Appliquons le théorème de Ramsey pour $k = m = 2$ et $r = k + 1$, on a donc un entier $N = N(2, 2, k + 1)$.

Montrons maintenant que pour deux langages L et K vérifiant σ'_k, si $L \cap (\varepsilon + A)^N = K \cap (\varepsilon + A)^N$ alors $K = L$. Et cela permettra de conclure puisque $(\varepsilon + A)^N$ est fini.

On montre par récurrence sur $|f|$ que $f \in K \Leftrightarrow f \in L$:

Déjà, cette propriété est immédiate lorsque $|f| \leq N$.

Sinon, $|f| \geq N + 1$, et on écrit $f = a_0 \cdots a_N w$ où $\forall i, a_i \in A$ et $w \in A^*$.

Posons maintenant :

$$X = \{(i, j) \mid a_0 \cdots a_i a_{j+1} \cdots a_N w \in L\}$$

Si l'on pose $E = \{1, \ldots, N\}$, $C = \{0, 1\}$ et $f(i, j) = \begin{cases} 0 & \text{si } (i, j) \in X \\ 1 & \text{sinon} \end{cases}$ le théorème de Ramsey (pour $(k = 2, m = 2, r = k + 1)$) nous donne un ensemble à $k + 1$ éléments $F = \{i_0, \ldots, i_k\}$ tel que l'on est dans l'un des deux cas :

- Pour tout couple d'entiers (m, n) vérifiant $0 \leq m < n \leq k$, on a $(i_m, i_n) \in X$ (soit $a_0 \cdots a_{i_m} a_{i_n+1} \cdots a_N w \in L$).
- Pour tout couple d'entiers (m, n) vérifiant $0 \leq m < n \leq k$, on a $(i_m, i_n) \notin X$ (soit $a_0 \cdots a_{i_m} a_{i_n+1} \cdots a_N w \notin L$).

On peut alors écrire f sous la forme $x v_1 v_2 \cdots v_k y$ avec :

- $x = a_0 a_1 \cdots a_{i_0-1}$;
- $v_j = a_{i_{j-1}} a_{i_{j-1}+1} \cdots a_{i_j-1}$ pour $1 \leq j \leq k$;
- $y = a_{i_k+1} \cdots a_N$.

On utilise alors la propriété σ'_k de K avec la factorisation de f ci-dessus. On obtient donc deux entiers α et β tels que $f \in K \Leftrightarrow a_0 \cdots a_{i_\alpha} a_{i_\beta+1} \cdots a_N w \in K$.

Or, par récurrence, $a_0 \cdots a_{i_\alpha} a_{i_\beta+1} \cdots a_N w \in K \Leftrightarrow a_0 \cdots a_{i_\alpha} a_{i_\beta+1} \cdots a_N w \in L$. Comme on avait $f \in L \Leftrightarrow a_0 \cdots a_{i_\alpha} a_{i_\beta+1} \cdots a_N w \in L$, on a bien l'équivalence recherchée.

Voir [Sak03]. □

1.8 Reconnaissance par morphisme

Définition 1.8.1 (Monoïde).

On appele *monoïde* tout ensemble muni d'une loi de composition interne associative qui possède un élément neutre.

Exemple.

- A^*, muni de la concaténation (pour A, alphabet quelconque)
- G groupe, muni de sa loi naturelle (par exemple : $\mathbb{Z}/2\mathbb{Z}$)
- $M = \{1, \alpha, \beta\}$, où 1 est l'élément neutre et où la loi sur les autres éléments est définie ainsi :

- $\alpha\beta = \alpha^2 = \alpha$
- $\beta^2 = \beta\alpha = \beta$

Définition 1.8.2 (Morphisme de monoïdes).
Soit M et M' deux monoïdes. On appelle *morphisme de M dans M'* toute application $\mu : M \to M'$ qui vérifie :
- $\mu(1_M) = 1_{M'}$
- $\mu(xy) = \mu(x)\mu(y)$, pour tout couple de mots (x, y).

Définition 1.8.3 (Reconnaissance par morphismes).
On dit qu'un langage, $L \subseteq A^*$, est *reconnu* par le morphisme $\mu : A^* \to M$ si et seulement si il existe une partie P de M telle que $L = \mu^{-1}(P)$ (on a donc $P = \mu(L)$ — P est unique si et seulement si le morphisme est surjectif[1]).

Exemple (pour $M = \mathbb{Z}/2\mathbb{Z}$).
- $\mu : \begin{array}{ccc} A^* & \to & M \\ w & \mapsto & |w| \bmod 2 \end{array}$ (dans ce cas : $\mu^{-1}(0) = (A^2)^*$)
- $\mu : \begin{array}{ccc} A^* & \to & M \\ w & \mapsto & |w|_a \bmod 2 \end{array}$ (dans ce cas : $\mu^{-1}(0) = (ab^*a + b)^*$)

Exemple (pour $M = \{1, \alpha, \beta\}$ et $A = \{a, b\}$).
$\mu : A^* \to M$ défini par :

$$\mu(w) = \begin{cases} 1 & \text{si } w = \varepsilon \\ \alpha & \text{si } w \in aA^* \\ \beta & \text{si } w \in bA^* \end{cases}$$

Définition 1.8.4 (Reconnaissance par monoïde).
On dit qu'un monoïde M *reconnaît* le langage L s'il existe un morphisme $\mu : A^* \to M$ qui reconnaît L.

Proposition 1.8.1 (Rationnalité par morphisme).
Soit $L \subseteq A^*$. L est rationnel si et seulement s'il existe M un monoïde fini et un morphisme $\mu : A^* \to M$, qui reconnaît L.

Preuve.
Si L est reconnu par $\mu : A^* \to M$ (M pouvant plus généralement être un monoïde quelconque), on construit l'automate $\mathcal{A} = (Q, A, E, I, F)$ définit par :
- $Q = M$
- $I = \{1\}$
- $F = \mu(L)$
- $E = \{(m, a, m\mu(a)) \mid m \in M, a \in A\}$

Cet automate reconnaît L, ce qui prouve un sens de l'équivalence.

Exemple.
Pour $M = \{1, \alpha, \beta\}$ et la loi déjà rencontrée en 1.8 : Voir FIG. 1.16, page 30.

Pour le sens direct, on utilise les définitions suivantes :

Définition 1.8.5 (Monoïde des relations binaires).
Soient r et r' deux relations binaires sur un même ensemble E. On définit la composition de ces deux relations, notée rr' par :

$$rr' = \{(x, z) \mid \exists y (x, y) \in r et (y, z) \in r'\}$$

[1] On peut toujours supposer que μ est surjectif en le co-restreignant à son image.

On notera $\mathcal{R}_E$ le *monoïde des relations binaires* sur l'ensemble E (muni de la loi décrite ci-dessus).

Définition 1.8.6 (Morphisme des transitions).
Étant donné un automate $\mathcal{A} = (Q, A, E, I, F)$, le morphisme $\mu : A^* \to \mathcal{R}_Q$ définit par :

$$\mu(w) = r_w = \{(q, q') \mid q \xrightarrow{w} q' \text{ dans } \mathcal{A}\}$$

est appelé *morphisme des transitions* de $\mathcal{A}$ ($\mu(A^*)$ est appelé monoïde des transitions).

Avec ce morphisme, il suffit de prendre :

$$P = \{r \mid \exists i \in I, \exists f \in F, (i, f) \in r\}$$

pour avoir le résultat recherché. □

Définition 1.8.7 (Sous-monoïde).
M' est appelé *sous-monoïde* de M lorsque $M' \subseteq M$ et $M' \overset{id}{\hookrightarrow} M$ est un morphisme.

Définition 1.8.8 (Monoïde quotient).
On dit que M' est *quotient* de M s'il existe un morphisme surjectif $\mu : M \mapsto M'$.

Définition 1.8.9 (Monoïde diviseur).
On dit que M' *divise* M si M' est quotient d'un sous-monoïde M_1 de M.

Notation (M' divise M).
On note cela : $M' \lhd M$

Voir le schéma de ces relations FIG. 1.17, page 30.

Proposition 1.8.2 (Ordre de la division).
La relation divise est une relation d'ordre sur l'ensemble des monoïdes finis.

Remarque.
On perd l'antisymétrie lorsque les monoïdes sont infinis.

Exemple.
On peut illustrer ce fait avec $M = \{a, b\}^*$ et $M' = \{a, b, c, d\}^*$ (remarquez que $M \subsetneq M'$). Et en considérant les morphismes définit par :

$$\mu : M \to M' \quad : \quad w \mapsto w$$

$$\mu : M' \to M \quad : \quad \begin{cases} a \mapsto aa \\ b \mapsto bb \\ c \mapsto ab \\ d \mapsto ba \end{cases}$$

Preuve.
La réflexivité est évidente et l'antisymétrie se vérifie facilement par équipotence.

Pour la transitivité, supposons que $M \lhd M' \lhd M''$, alors :

$$\begin{array}{ccccc} M_2 & \overset{id}{\hookrightarrow} & M_1' & \overset{id}{\hookrightarrow} & M'' \\ \downarrow \pi & & \downarrow \pi' & & \\ M_1 & \overset{id}{\hookrightarrow} & M' & & \\ \downarrow \pi & & & & \\ M & & & & \end{array}$$

□

Définition 1.8.10 (Congruence).
Une relation d'équivalence $\sim$ définie sur M est appelée *congruence* si :

$$\left.\begin{array}{l} x \sim x' \\ y \sim y' \end{array}\right\} \Rightarrow xy \sim x'y'$$

Proposition 1.8.3.
$(M/\sim)$ est un monoïde pour la loi définie par $[x][y] = [xy]$ (où $[x]$ désigne la classe de x dans l'ensemble quotient).

Remarque.
Il suffit de montrer que $y \sim y' \Rightarrow \forall x, z\ xyz \sim xy'z$.

Définition 1.8.11 (Contexte).
Pour tout langage $L \subseteq A^*$, on appelle *contexte* de $y \in A^*$ l'ensemble $C(y) = \{(x, z) \in (A^*)^2 : xyz \in L\}$.

Proposition 1.8.4.
La relation $\sim_L$ définie ci-dessous est une congruence :

$$y \sim_L y' \Leftrightarrow C(y) = C(y') \Leftrightarrow \forall x, z \in A^*\ (xyz \in L \Leftrightarrow xy'z \in L)$$

Preuve.
Si $y \sim_L y'$, on a $\forall x, z \in A^*$, $C(xyz) = C(xy'z)$ et donc $xyz \sim_L xy'z$. □

Définition 1.8.12 (Congruence syntaxique de L).
On appelle *congruence syntaxique* la congruence $\sim_L$.

Définition 1.8.13 (Monoïde syntaxique).
Le *monoïde syntaxique* est : $M(L) = A^*/\sim_L$

Proposition 1.8.5.

1. $\left.\begin{array}{l} M \text{ reconnaît } L \\ M \overset{id}{\hookrightarrow} M' \end{array}\right\} \Rightarrow M' \text{ reconnaît } L$
2. $\left.\begin{array}{l} M \text{ reconnaît } L \\ M \text{ quotient de } M' \end{array}\right\} \Rightarrow M' \text{ reconnaît } L$
3. $\left.\begin{array}{l} M \text{ reconnaît } L \\ M \lhd M' \end{array}\right\} \Rightarrow M' \text{ reconnaît } L$

Preuve.

1. et 3. sont évidents.

2. : pour $a \in A$, définissons $\mu'(a) = m \in M'$ pour m vérifiant $\pi(m) = \mu(a)$.

Ensuite, on définit μ sur tout mot $\omega = a_1 \cdots a_n$ par $\mu(\omega) = \mu(a_1) \dots \mu(a_n)$.

Alors, par définition $\pi(\mu'(\omega)) = \mu(\omega)$.

Et en posant $P' = \pi^{-1}(P)$, on a bien $\mu'^{-1}(P') = \mu^{-1}(P)$. □

Proposition 1.8.6.

$M(L)$ reconnaît L.

Preuve.

En effet, $\pi : \begin{array}{ccc} A^* & \to & M(L) \\ \omega & \mapsto & \overline{\omega} \end{array}$ est un morphisme et $L = \pi^{-1}(P)$ où $P = \{\overline{\omega} \mid \omega \in L\}$. □

Proposition 1.8.7.

Soit $L \in A^*$ et M un monoïde. Alors,

$$M \text{ reconnaît } L \Leftrightarrow M(L) \lhd M$$

Preuve.

En utilisant les propositions 1.8.5 et 1.8.6, on obtient immédiatement que, si $M(L) \lhd M$, alors M reconnaît L.

Réciproquement, si M reconnaît L, soit un morphisme $\mu : A^* \to M$ tel que $L = \mu^{-1}(P)$ pour un certain $P \subseteq M$; $M' = \mu(A^*)$ est un sous-monoïde de M.

$$\begin{array}{lll} A^* & \xrightarrow{\mu} & M' \\ \downarrow \pi & \swarrow \pi' & \\ M(L) & & \end{array}$$

On peut poser $\pi'(\mu(\omega)) = \pi(\omega)$ car si $\mu(\omega) = \mu(\omega')$ alors $\mu(x)\mu(\omega)\mu(y) \in P \Leftrightarrow \mu(x)\mu(\omega')\mu(y) \in P$ donc $x\omega y \in L \Leftrightarrow x\omega' y \in L$. □

Exercice.

Montrer que $M(L)$ est le monoïde de transition de l'automate minimal de L.

Exemple.

Soit $L = (ab)^*$. L'automate minimal est dessiné à la figure 1.15.

On a alors la table de lecture 1.1 (page 30).

Le monoïde associé comporte 6 éléments $\{1, 0, \alpha, \beta, \alpha\beta, \beta\alpha\}$ et a les propriétés suivantes :

- $\alpha\beta\alpha = \alpha$
- $\beta\alpha\beta = \beta$
- $\alpha^2 = \beta^2 = 0$

	0	1	2
$1 = \varepsilon$	0	1	2
$aba = a$	1	2	2
$bab = b$	2	0	2
$0 = bb = aa$	2	2	2
ab	0	2	2
ba	2	1	2
bb	2	2	2
aba	1	2	2
bab	2	0	2

TAB. 1.1 – Table de construction du morphisme associé à FIG. 1.15, page 19

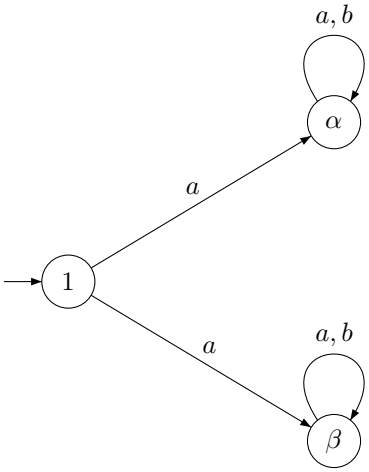


FIG. 1.16 – Exemple d'automate construit à l'aide d'un morphisme de monoïdes reconnaissant un langage rationnel

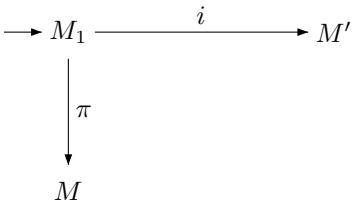


FIG. 1.17 – Diagramme des relations *sous-monoïde*, *quotient* impliquées par la relation *divise*.

1.9 Langages sans étoile

Définition 1.9.1 (Langage sans étoile).
La famille des *langages sans étoile* est la plus petite famille $\mathcal{E}$ telle que :

- $\emptyset \in \mathcal{E}$ et $\forall a \in A, \{a\} \in \mathcal{E}$;
- $\mathcal{E}$ est stable par union ($(K, L) \in \mathcal{E}^2 \Rightarrow K + L$), concaténation ($(K, L) \in \mathcal{E}^2 \rightarrow KL$) et complémentation ($K \in A^* \setminus K$).
  Ainsi, si K et L appartiennent à $\mathcal{E}$, alors $K + L$ et KL appartiennent à L

Exemple.
On a :
- $A^* = A \backslash \emptyset$ et est donc un langage sans étoile.
- $(ab)^* \in \mathcal{E}$ car il s'écrit aussi $(ab)^* = (aA^* \cap A^*b) \setminus A^*(aa + bb)A^*$.
- $(aa)^*$ n'est pas sans étoile.

Définition 1.9.2 (Groupe contenu dans un monoïde).
On dit qu'un groupe G est *contenu* dans un monoïde M si :

1. $G \subseteq M$
2. $\forall (g, g') \in G^2, g._G g' = g._M g'$

Attention, à priori $1_G \neq 1_M$. On dit que M est apériodique s'il ne contient pas de groupe non trivial.

Théorème 1.9.1 (Schützenberger).
L est sans étoile si et seulement si $M(L)$ est apériodique.

Preuve.
Le théorème se prouve aisément une fois démontrés les six lemmes suivants :

Lemme 1.9.2.
Pour tout monoïde fini M,

$$M \text{ est apériodique} \Longleftrightarrow \forall\ m \in M,\ \exists\ \omega \in \mathbb{N}^*, m^\omega = m^{\omega+1}$$

Preuve.
$\Longrightarrow$: Montrons cela par contraposition.
Supposons que dans un monoïde, M, il existe m tel que : $\forall \omega \geq 0, m^\omega \neq m^{\omega+1}$. Alors, M étant fini, $\exists k, l \in \mathbb{N}^*, m^{k+l} = m^k$. Soit $p = \min\{l \in \mathbb{N}^* \mid m^{k+l} = m^k$, comme $p \geq 2$, on peut définir $G = \{m^k, m^{k+1}, \ldots, m^{k+p-1}\}$; G est un groupe isomorphe à $\mathbb{Z}\ /\ p\mathbb{Z}$, non trivial.
$\Longleftarrow$: Si $\forall m \in M, \exists \omega, m^\omega = m^{\omega+1}$, alors pour tout groupe G contenu dans M on a $\forall g \in G, g^{\omega'} = g^{\omega'+1}$ (avec $\omega' = ppcm_{g \in G}\omega(g)$) ce de quoi on déduit que G est un groupe trivial. □

Lemme 1.9.3.
Pour $L \subseteq A^*$ rationnel :

$$M(L) \text{ apériodique } \Longleftrightarrow (\exists w \geq 1 \mid \forall (x, y, z), xy^w z \in L \Leftrightarrow xy^{w+1}z \in L)$$

Notation.
Pour un langage tel que $M(L)$ apériodique, on note $i(L)$ le plus petit entier tel que $\forall (x, y, z), xy^{i(L)}z \in L \Leftrightarrow xy^{i(L)+1}z \in L$.

Lemme 1.9.4.
Pour $K, L \subseteq A^*$ rationnel :

$$\begin{array}{rcl} i(\{a\}) & = & 1 \\ i(K+L) & \leq & \max(i(K), i(L)) \\ i(KL) & \leq & i(K)+i(L)+1 \\ i(A^* \setminus K) & = & i(K) \end{array}$$

On en déduit qui si L est sans étoile alors $M(L)$ est apériodique.

Lemme 1.9.5 (Effacement).
Soit M apériodique et $p, m, q \in M$.

$$m = pmq \Rightarrow m = pm = mq$$

Preuve.
On a :

$$m = pmq = p^w mq^w = p^w mq^{w+1} = mq$$

et on procède pareillement pour $m = pm$. □

Lemme 1.9.6.
Pour M, un monoïde apériodique :

$$\{m\} = (mM \cap Mm) \setminus J$$

où $J = \{x \in M \mid m \notin MxM\}$.

Preuve.
Tout d'abord, $m \in mM \cap Mm$ et de plus $m \in MmM$ donc $m \notin J$. D'où $\{m\} \subseteq (mM \cap Mm) \setminus J$.

Soit maintenant $m' \in (mM \cap Mm) \setminus J$,
On a $m' \in mM \cap Mm$ donc $m' = mp = qm$.
D'autre part, $m' \notin J$ alors $m \in Mm'M$ donc $m = um'v$.
$\begin{cases} m' = qm \\ m = um'v \end{cases}$ donc $m = uqmv$. Et par effacement, $m = uqm = mv$.
$\begin{cases} m = uqm \\ m' = qm \end{cases}$ donc $m = um'$.
$\begin{cases} m' = mp \\ m = um' \end{cases}$ donc $m' = um'p$. Et par effacement, $m' = um'$.
Donc $m = m'$ d'où $(mM \cap Mm) \setminus J \subseteq \{m\}$. □

Lemme 1.9.7.
Soit M, un monoïde apériodique, $\mu : A^* \to M$ un morphisme, et $m \neq 1$. Alors :

$$\mu^{-1}(m) = (UA^* \cap A^*V) \setminus (A^*WA^*)$$

où

$$U = \bigcup_{\substack{n\mu(a)M = mM \\ n \notin mM}} \mu^{-1}(n)a$$

$$V = \bigcup_{\substack{Mm=M\mu(a)n \\ n\notin Mm}} a\mu^{-1}(n)$$

et

$$W = \{a \mid m \notin M\mu(a)M\} \cup \left(\bigcup_{\substack{m\notin M\mu(a)\cap\mu(b)M \\ m\in(M\mu(a)nM\cap Mn\mu(b)M)}} a\mu^{-1}(n)b \right)$$

□

Preuve $\mu^{-1}(m)$ sans étoile.

Posons $r(m) = |MmM|$. Faisons la démonstration par récurrence sur $|M| - r(m)$:

– si $r(m) = |M|$, alors $m = 1$ et :

$$\mu^{-1}(1) = \{a \mid \mu(a) = 1\}^* = A^* \setminus A^*\{a \mid \mu(a) \neq 1\}A^*$$

– *sinon*, $\mu^{-1}(m) = (UA^* \cap A * V) \setminus A^*WA^*$

□

Lemme 1.9.8.

$$\left.\begin{array}{l} n\mu(a)M = mM \\ n \notin mM \end{array}\right\} \Rightarrow r(n) > r(m)$$

Voir [Pin84] pour un exposé complet.

Exemple.

Prenons $L = (ab)^*$ et considérons $M = \{1, \alpha, \beta, \alpha\beta, \beta\alpha, 0\}$. On a $M\alpha M = \{0, \alpha\beta, \beta\alpha, \beta, \alpha\}$, ainsi :

– $\mu^{-1}(1) = \varepsilon$

– $\mu^{-1}(a) = (ab^*)a = (aA^* \cap A^*a) \setminus A^*(aa + bb)A^*$
(en prenant $U = a, V = a, W = aa + bb$).

Chapitre 2

Langages algébriques

Les langages algébriques sont parfois appelés *langages hors contexte*, ou encore *context-free* dans la littérature anglaise. Pour tout ce chapitre, on se reportera à [Aut87].

2.1 Grammaires algébriques

2.1.1 Définitions et exemples

Définition 2.1.1 (Grammaire).
Une *grammaire* G est un triplet (A, V, P) où A, V et P sont des ensembles finis vérifiant :

$$P \subseteq V \times (A \cup V)^*$$

A est l'alphabet terminal, V est l'alphabet non terminal (constitué des variables), et P est l'ensemble de règles.

Notation.
On note $v \to u$ lorsque $(v, u) \in P$. Par extension, on note $v \to u_1 + u_2 + \cdots + u_n$ lorsque $\forall i, (v, u_i) \in P$.

Exemple.
On définit une grammaire par :

$$A = \{a, b\}, V = \{S\}, P : \text{´ } S \to aSb + \varepsilon \text{ ˇ}$$

On verra que, pour cette grammaire, $L_G(S) = \{a^n b^n \mid n \in \mathbb{N}\}$.

Définition 2.1.2 (Dérivation).
On dit que u se *dérive* [1] en v (pour $(u, v) \in ((A + V)^*)^2$), et on note $u \to v$ lorsque il existe $(\alpha, \beta) \in ((A + V)^*)^2, X \in V$ tels que :

$$u = \alpha X \beta, v = \alpha w \beta \text{ et } (X \to w) \in P$$

Notation (Dérivation itérée).
On note $u \to^k v$ s'il existe $u_1, u_2, \cdots, u_{k-1}$ tels que $u \to u_1 \to u_2 \to \cdots \to u_{k-1} \to v$.

Plus généralement, on notera $u \to^* v$ s'il existe un k tel que $u \to^k v$.

[1] On parle aussi de *production*.

Proposition 2.1.1 (Clôture de la dérivation).
$\to^*$ est la clôture réflexive et transitive de $\to$.

Définition 2.1.3 (Langage engendré par une grammaire).
Soit $G = (A, V, P)$, une grammaire, et $S \in V$. On définit :

$$L_G(S) = \{w \in A^* \mid S \to^* w\}$$

Notation.
Par extension, on note $\widehat{L_G(S)} = \{w \in (V \cup A)^* \mid S \to^* w\}$.

Définition 2.1.4 (Langage algébrique).
Un langage est dit *algébrique* s'il peut être engendré par une grammaire (*ie.* il s'écrit $L_G(S)$ pour un certain couple (G, S)).

Exemple.
- $G : S \to aS + b \qquad L_G(S) = a^*b$
- $G : S \to aSa + bSb + a + b + \varepsilon \qquad L_G = \{palindromes\}$

- *Langage de Dyck*

$$\begin{cases} A = \{a,\ \bar{a},\ b,\ \bar{b}\} \\ S \to ST + \varepsilon \\ T \to aS\bar{a} + bS\bar{b} \end{cases}$$

  $D_2 = L_G(T)$ langage de Dyck primitif
  $D_2^* = L_G(S)$ langage de Dyck

- *Langage de Luckasuwicz*

$$S \to aSS + b$$

  $L_G(S) = \{w \mid |w|_b = |w|_a + 1$ et pour tout préfixe $u,\ |u|_b \leq |u|_a\}$
  Ce langage est noté Ł. Intuitivement, les mots de ce langage peuvent représenter :
  - un jeu de pile ou face où, sauf égalité, a mène la partie tout le temps (plus de victoires que b), sauf au dernier coup où b remporte.
  - une succession de `push` et de `pop`, en partant d'une pile vide, en commençant par un `push`, et en stoppant au dernier `pop` car effectué sur pile vide.

- *Langage de Goldstine*

$$L = \{a^{n_1}ba^{n_2}b\cdots a^{n_k}b \mid k > 0 \text{ et } \exists\, j \in \mathbb{N}, n_j \neq j\}$$

  $L = L_G(S)$ avec G pouvant être définie comme suit :

$$\begin{cases} T_{(A^*)} & \to & aT_{(A^*)} + bT_{(A^*)} + \varepsilon \\ T_{(a^*b)} & \to & aT_{(a^*b)} + b \\ T & \to & T_{(a^*b)}Ta + \varepsilon + aa \\ S & \to & TbT_{(A^*)}b \end{cases}$$

En effet, nous avons clairement $L_G(S) = L_- \cup L_+$ où :

$$\begin{aligned} L_- &= \{(a^*b)^n a^n b A^* b\} \\ L_+ &= \{(a^*b)^n a^{n+2} b A^* b\} \end{aligned}$$

et alors :

$$\begin{aligned} u = a^{n_1} b a^{n_2} b \cdots a^{n_k} b \in L &\Leftrightarrow \exists j \mid n_j < j \text{ ou } \exists j \mid n_j > j \\ &\Leftrightarrow u \in L_- \text{ ou } u \in L_+ \end{aligned}$$

Quelques exemples et contre-exemples laissés en exercice :
- $\{wAw' \mid w \neq w'\}$ n'est pas algébrique ;
- $\{wAw' \mid w = w'\}$ n'est pas algébrique ;
- $\{wAw' \mid w \neq w' \text{ et } |w| = |w'|\}$ n'est pas algébrique ;
- $\{ww' \mid w \neq w' \text{ et } |w| = |w'|\}$ est algébrique.

2.1.2 Grammaires réduites

Définition 2.1.5 (Grammaire réduite).
G est dite *réduite* pour $S_0 \in V$ si :

$$\begin{cases} \forall\ S \in V, L_G(S) \neq \emptyset \\ \forall\ S \in V, \exists \alpha, \beta \in (A+V)^*, S_0 \rightarrow^* \alpha S \beta \end{cases}$$

Théorème 2.1.1 (Réduction).
Pour toute grammaire $G = (A, V, P)$ et tout $S_0 \in V$ fixé [2], il existe une grammaire G' réduite pour T telle que $L_G(S_0) = L_{G'}(T)$

Preuve.
1re étape[3] : suppression des variables S telles que $L_G(S) = \emptyset$.
Posons $U_0 = A$, $\forall i \in \mathbb{N}, U_{i+1} = U_i \cup \{S \in V \mid S \rightarrow w \text{ et } w \in U_i^*\}$.
On a $U_i \subseteq A \cup V$ fini et $(U_i)_{i\in\mathbb{N}}$ est croissante donc cette suite est constante à partir d'un certain rang.
Soit $U = \bigcup_{i\geq 0} U_i$. On prouve par récurrence que $U \cap V = \{S \mid L_G(S) \neq \emptyset\}$.
On supprime ainsi les variables qui ne sont pas dans $U \cap V$.
2e étape : Suppression des variables inaccessibles à partir de l'axiome
On défini récursivement la suite d'ensembles de variables (W_i) :
- $W_0 = \{S_0\}$, $\forall\ i \in \mathbb{N}$
- $W_{i+1} = W_i \cup \{S \in V \mid \exists\ S' \in W_i, S' \rightarrow u_1 S u_2 \text{ avec } u_1, u_2 \in (A+V)^*\}$

On a $W_i \subseteq V$ fini et $(W_i)_{i\in\mathbb{N}}$ est croissante donc cette suite est constante à partir d'un certain rang.
Soit $W = \bigcup_{i\geq 0} W_i$. On a $W = \{S \mid S_0 \rightarrow^* uSv \text{ pour un certain couple } (u, v)\}$.
Ainsi, W ne garde que les variables accessibles depuis S_0. □

Lemme 2.1.2 (Fondamental).

Si $u_1 u_2 \rightarrow^k v$ alors v s'écrit $v_1 v_2$ et vérifie $\begin{cases} u_1 \rightarrow^{k_1} v_1 \\ u_2 \rightarrow^{k_2} v_2 \end{cases}$ *et $k = k_1 + k_2$.*

[2] S_0 est souvent appelé l'*axiome*.
[3] L'ordre des étapes est important.

2.1.3 Grammaires propres

Définition 2.1.6 (Grammaire propre).
Une grammaire $G = (A, V, P)$ est dite *propre* si P ne contient pas de règle d'une des deux formes suivantes :
- $S \to \varepsilon$
- $S \to S'$.

Définition 2.1.7 (Substitution).
Étant donné un alphabet A, on dit qu'une application $\sigma : A^* \to \mathfrak{P}(A^*)$ est une *substitution* si pour tout $w = w_1 w_2 \dots w_n \in A^*$, on a $\sigma(w) = \sigma(w_1)\sigma(w_2)\dots\sigma(w_n)$.

Proposition 2.1.2.
Pour toute grammaire $G = (A, V, P)$ et tout $S \in V$, il existe une grammaire propre $G' = (A, V', P')$ et $S' \in V'$ telle que $L_{G'}(S') = L_G(S) \setminus \{\varepsilon\}$.

Preuve.
Construisons $\{S \mid S \to^* \varepsilon\}$.
Posons $U_0 = \{S \mid S \to \varepsilon\}, U_{i+1} = U_i \cup \{S \mid S \to w \text{ et } w \in U_i^*\}$.
Ainsi, $(U_i)_{i\in\mathbb{N}}$ est constante à partir d'un certain rang ; et, par récurrence, on peut montrer :

$$u = \bigcup_{i\geq 0} U_i = \{S \mid S \to^* \varepsilon\}$$

Introduisons la substitution σ définie par :
- $\sigma(a) = a$ pour $a \in A$
- $\sigma(S) = S + \varepsilon$ si $S \to^* \epsilon$
- $\sigma(S) = S$ sinon

Étape 1 :
- on supprime les règles $S \to \varepsilon$.
- on ajoute les règles $S \to u$ lorsque $S \to w$ est une règle et $u \in \sigma(w)$.

Étape 2 :
On définit une relation d'équivalence par :

$$S \equiv S' \Leftrightarrow (S \to^* S' \text{ et } S' \to^* S)$$

On a $S \equiv S' \Rightarrow L_G(S) = L_G(S')$.
On passe au quotient $V/\equiv$: la relation $\to^*$ est alors un ordre (on vient d'ajouter l'antisymétrie). On retire alors les règles de la forme $S \to S$.
Si S est maximal, alors il n'y a pas de règle $S \to S'$, et on remplace chaque règle $S' \to S$, où S est maximal, par les règles $S' \to w$, où $S \to w$.
On supprime alors les éléments maximaux de la relation d'ordre et on recommence. □

2.1.4 Système d'équations associé à une grammaire

Remarque (Pourquoi dit-on algébrique ?).
À une grammaire G, on peut associer un système d'équations. Par exemple, pour $S \to ST + \varepsilon$, $T \to aT + b$, on a le système en langages suivant, pour $S, T \subseteq A^*$:

$$\begin{cases} S = ST + \varepsilon \\ T = aT + b \end{cases}$$

Proposition 2.1.3 (Minimalité des langages associés).
Soit $G = (A, V, P)$ une grammaire avec $V = \{X_1, X_2, \ldots, X_n\}$.
Alors $L_G(X_1), L_G(X_2), \ldots, L_G(X_n)$ est la solution minimale (au sens de l'inclusion) du système associé à G.

Remarque.
Cette solution n'est pas unique ! Par exemple, pour la grammaire $X \to XX$, on a $L_G(X) = \emptyset$. Pourtant, $X = XX$ admet de nombreuses solutions, par exemple : $A^*, (A^2)^*$.

Définition 2.1.8 (Solution propre).
Une solution est dite *propre* si aucun langage ne contient le mot vide.

Proposition 2.1.4 (Unicité des langages associés à une grammaire propre).
Soit G une grammaire propre : $G = (A, V, P)$ et $V = \{X_1, X_2, \ldots, X_n\}$.
Alors $L_G(X_1), L_G(X_2), \ldots, L_G(X_n)$ est l'unique solution propre du système.

Preuve.
Soient $L_1, L_2, \ldots, L_n$ et $L'_1, L'_2, \ldots, L'_n$ deux solutions propres du système. On montre par récurrence sur k que $L_i \cap (A + \varepsilon)^k = L'_i \cap (A + \varepsilon)^k$. □

Définition 2.1.9 (Anagrammes).
On dit que w et w' sont *commutativement équivalents* ou encore que w est un *anagramme* de w' lorsque $\forall a \in A, |w|_a = |w'|_a$.

Notation.
On note $w \equiv w'$ si w et w' sont commutativement équivalents et $\overline{w}$ la classe de w ; $\overline{L} = \{\overline{w} \mid w \in L\}$, pour $L \subseteq A^*$.

Théorème 2.1.3 (Parikh).
Pour tout langage algébrique L, il existe un langage rationnel L' tel que $\overline{L} = \overline{L'}$.

Exemple.
À $L = \{a^n b^n \mid n \geq 0\}$ on peut par exemple associer $L' = (ab)^*$.

Preuve.
Les propositions précédentes restent vraies lorsque l'on suppose la concaténation commutative.

Commençons par une grammaire à une variable : $X \to P(X)$. Cette grammaire est, à commutation près, solution du système $X = R(X)X + S$ (où $R(X)$ dénote une expression rationelle en X) avec $S \in A^*$. Posons $G = R(S)$. Alors, G^*S est aussi solution du système d'après le lemme d'Arden (voir le lemme 1.3.2, 14).

Pour un système à plusieurs variables, on résoud pareillement chaque équation en fixant les autres variables comme des lettres. □

2.1.5 Arbres de dérivation

Définition 2.1.10 (Arbre de dérivation).
Soit $G = (A, V, P)$ une grammaire. Un *arbre de dérivation* est un arbre fini étiqueté par $A \cup V \cup \{\varepsilon\}$ tel que :

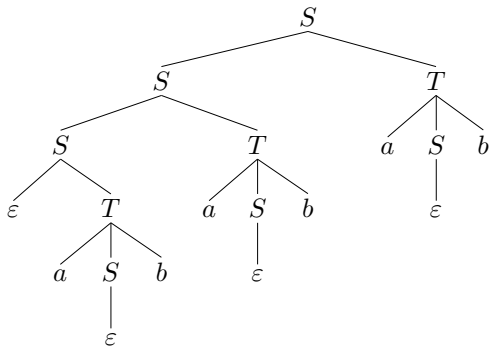


FIG. 2.1 – Un arbre de dérivation

- les feuilles sont étiquetées par un élément de $A \cup \{\varepsilon\}$.
- si un nœud interne est étiqueté par S et ses fils par $w_1, w_2, \ldots, w_n$, alors $S \to w_1 w_2 \ldots w_n$ est une règle.

Définition 2.1.11 (Arbre de dérivation partielle).
Un arbre de dérivation partielle admet également des feuilles étiquetées par une variable (les feuilles sont donc étiquetées par un élément de $A \cup V \cup \{\varepsilon\}$).

Exemple.
Soit la grammaire :
- $S \to ST + \varepsilon$
- $T \to aSb$

La FIG. 2.1, page 40 exhibe un arbre de dérivation.

Définition 2.1.12 (Frontière).
La *frontière* d'un arbre de dérivation est le mot obtenu par concaténation des feuilles (de gauche à droite).

Proposition 2.1.5.
*$L_G(S)$ (*resp. *$\widehat{L_G}(S)$) est l'ensemble des mots $w \in A^*$ (*resp. *$(A \cup V)^*$) tels qu'il existe un arbre de dérivation (*resp. *partielle) ayant S à la racine et dont la frontière est w.*

Définition 2.1.13 (Dérivation gauche).
On appelle *dérivation gauche* une dérivation où la variable remplacée est systématiquement la plus à gauche de l'image de la règle.

2.1.6 Ambiguïté

Définition 2.1.14 (Grammaire ambiguë).
Une grammaire G est dite *ambiguë* s'il existe un mot ayant au moins deux arbres de dérivation distincts étiqueté à la racine par le même S.

Définition 2.1.15 (Langage non ambigu).
Un langage est dit *non ambigu* s'il existe au moins une grammaire non ambiguë qui l'engendre.

Exemple.

$S \to SS + a$ est une grammaire ambiguë, mais le langage engendré (a^*) est non ambigu car il est également engendré par $S \to aS + a$.

On verra qu'il existe des langages inhéremment ambigus, c'est à dire qui ne peuvent pas être engendrés par une grammaire non ambiguë.

2.1.7 Forme normale quadratique

Définition 2.1.16 (Forme normale quadratique).

Une grammaire $G = (A, V, P)$ est dite en *forme normale quadratique* si toutes les règles de G sont d'une des formes suivantes :

- $S \to S_1 S_2$, $(S_1, S_2) \in V^2$
- $S \to a$, $a \in A$

Proposition 2.1.6 (Mise en forme normale quadratique).

Toute grammaire est équivalente à une grammaire en forme normale quadratique.

Preuve.

1re étape : on se ramène à une grammaire où les règles sont d'une des formes suivantes :

- $S \to S_1 S_2 \dots S_n$, $(S_1, S_2, S_n) \in V^n$
- $S \to a$, $a \in A$

On peut supposer que G est grammaire propre. Soit V' en bijection avec A ($V' = \{V_a \mid a \in A\}$). Soit $G' = (A, V \cup V', P')$ où $P' = \{V_a \to a \mid a \in A\} \cup \{S \to \sigma(w) \mid S \to w \in P\}$, où σ est la substitution définie par $\sigma(S) = S$ pour $S \in V$, et $\sigma(a) = V_a$ pour $a \in A$.

2e étape : on supprime les règles $S \to S_1 S_2 \dots S_n$ avec $n > 2$. Pour cela, on introduit $S'_1, S'_2, \dots, S'_{n-1}$ et on remplace $S \to S_1 S_2 \dots S_n$ par

$$\begin{cases} S \to S_1 S'_2 \\ S'_i \to S_i S'_{i+1} \text{ pour } 2 \leq i < n-1 \\ S'_{n-1} \to S_{n-1} S_n \end{cases}$$

. □

2.2 Lemme d'itération

Soit un arbre où certaines feuilles sont *distinguées*. On dit que :

- un nœud est *distingué* lorsque le sous-arbre dont il est racine contient des feuilles distinguées.
- un nœud est *particulier* lorsqu'il a au moins deux fils distingués.

Lemme 2.2.1.

Soit t un arbre de degré m avec k feuilles distinguées. Si chaque branche contient au plus r nœuds particuliers, alors $k \leq m^r$.

Preuve.

Montrons cela par récurrence sur r :

Si $r = 1$, alors on prend le nœud particulier de hauteur minimale (il existe), c'est le seul.

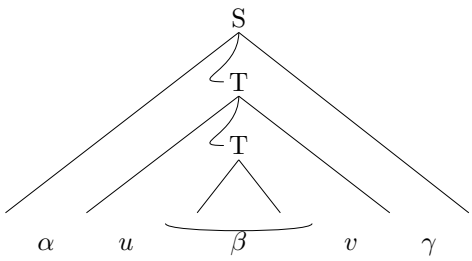


FIG. 2.2 – Découpage d'un mot par Ogden

Si $r \geq 1$, alors on prend les nœuds particuliers les plus bas possibles et on applique le lemme pour $r = 1$ dans leurs sous-arbres. Puis on les remplace par des feuilles et cela donne un nouvel arbre qui possède au plus $r - 1$ noeuds particuliers. □

Lemme 2.2.2 (d'Ogden [4]).

Soit $G = (A, V, P)$ une grammaire et $S \in V$. Alors, il existe un entier K tel que tout mot $f \in \widehat{L_G}(S)$ ayant au moins K lettres distinguées se factorise en $f = \alpha u \beta v \gamma$, où $\alpha, u, \beta, v, \gamma \in (\Sigma \cup V)^*$, avec :

1. *$S \to^* \alpha T \gamma$, $T \to^* uTv + \beta$.*
2. *soit α, u, β, soit β, v, γ contiennent des lettres distinguées.*
3. *$u\beta v$ contient moins de K lettres distinguées.*

Preuve.

Soit m la longueur maximal des membres droits des règles ; m majore le degré de l'arbre.

On choisit $r = 2|V| + 2$ et on pose $K = m^r + 1$. Si l'arbre de dérivation a K feuilles distinguées, alors, par contraposition du lemme précédent, on a au moins r nœuds particuliers sur une branche.

Il y en a de deux types : des nœuds particuliers droits, et des nœuds particuliers gauches.

On a au moins $|V|+1$ nœuds particuliers de même type sur une branche. On suppose qu'ils sont de types gauches. Soit à considérer uniquement les $|V| + 1$ derniers ; deux d'entre eux sont étiquetés par la même variable non-terminale T.

Le mot est alors découpé comme le suggère l'illustration FIG. 2.2, page 42. □

Corollaire 2.2.1 (Théorème de Bar-Hillel, Perles, Shamir).

Soit L un langage algébrique, il existe $N \geq 0$ tel que pour tout $\omega \in L$, si $|\omega| \geq N$ alors on peut trouver une factorisation $\omega = \alpha u \beta v \gamma \in L$ tel que $|uv| > 0$, $|u\beta v| < N$ et $\omega = \alpha u^n \beta v^n \gamma \in L$ pour tout $n \geq 0$.

[4] Ce lemme est également appelé *lemme d'itération*.

2.3 Propriétés de clôture des langages algébriques

2.3.1 Opérations rationnelles

Proposition 2.3.1 (Opérations rationnelles).
Les langages algébriques sont clos par union, concaténation et étoile.

Preuve.
Soient $G = (A, V, P)$, $L = L_G(S)$ et $G' = (A, V', P')$, $L = L'_G(S')$, avec $V \cap V' = \emptyset$.
- *Union* : $G'' = (A, \{S_0\} \cup V \cup V', \{S_0 \to S + S'\} \cup P \cup P')$
- *Concaténation* : $G'' = (A, \{S_0\} \cup V \cup V', \{S_0 \to SS'\} \cup P \cup P')$
- *Étoile* : $G'' = (A, \{S_0\} \cup V, \{S_0 \to SS_0 + \varepsilon\} \cup P)$

□

Corollaire 2.3.1.
Les langages rationnels sont aussi algébriques.

2.3.2 Substitution algébrique

Proposition 2.3.2 (Clôture par substitution algébrique).
Si L algébrique et σ substitution algébrique, alors $\sigma(L) = \{\sigma(w) \mid w \in L\}$ est aussi algébrique.

Preuve.
Soient $G = (A, V, P)$ une grammaire pour $L = L_G(S)$ et $\sigma : A \to \mathfrak{P}(B^*)$, une substitution algébrique telle que pour tout $a \in A$, $G_a = (B, V_a, P_a)$ est une grammaire pour $\sigma(a) = L_{G_a}(S_a)$.

On considère alors $G' = (B, V \cup (\bigcup_a V_a), P' \cup (\bigcup_a P_a))$ où $P' = \{S \to \rho(w) \mid S \to w \in P\}$ avec ρ la substitution telle que $\rho(S) = S$ pour tout $S \in V$, et $\rho(a) = S_a$ pour tout $a \in A$. Et cette grammaire engendre $\sigma(L)$. □

2.3.3 Morphisme alphabétique inverse

Définition 2.3.1 (Morphisme alphabétique).
Un morphisme $\sigma : A^* \to B^*$ est dit *alphabétique* si pour tout $a \in A$, $|\sigma(a)| \leq 1$.

Lemme 2.3.1 (Factorisation d'un morphisme).
Pour tout morphisme $h : A^* \to B^*$, il existe :
- *deux morphismes alphabétiques $g : C^* \to B^*$ et $\pi : C^* \to A^*$,*
- *un langage rationnel $K \subseteq C^*$*

tels que pour tout $w \in B^*$, $h^{-1}(w) = \pi(g^{-1}(w) \cap K)$.

Preuve.
On pose :
- $C = \{a_i \mid a \in A \text{ et } 0 \leq i \leq |h(a)|\}$
- $C_0 = \{a_0 \mid a \in A\}$
- $C_1 = \{a_k \mid a \in A \text{ et } k = |h(a)|\}$

et
- $W = C^2 \setminus (\{a_i a_{i+1} \mid 0 \leq i < |h(a)|\} \cup C_0 \cup C_1)$
- $K = (C_0 C^* \cap C^* C_1) \setminus C^* W C^*$

On définit alors $g : \begin{cases} a_0 \mapsto \epsilon \\ a_i \mapsto \text{la } i^{\text{e}} \text{ lettre de } h(a) \text{ pour } i \geq 1 \end{cases}$

et $\pi : \begin{cases} a_0 \mapsto a \\ a_i \mapsto \epsilon \text{ pour } i \geq 1 \end{cases}$. □

Proposition 2.3.3 (Morphisme inverse).
Si $h : A^* \to B^*$ morphisme et $L \subseteq B^*$ algébrique, alors $h^{-1}(L)$ est algébrique.

Preuve.
Montrons d'abord la propriété lorsque h est alphabétique.
Soit $G = (B, V, P)$ une grammaire telle que $L = L_G(S_0)$.
On pose $A_0 = \{a \in A \mid h(a) = \varepsilon\}$ et $A_1 = \{a \in A \mid h(a) \in B\}$.
Pour $S \in V$, on pose $h(S) = S$.
On définit la grammaire $G' = (A, V, P_0 \cup P_1)$ avec :
– $P_0 = \{T \to \sum_{a \in A_0} aT + \varepsilon\}$
– $P_1 = \{S \to Tu_1Tu_2 \dots Tu_nT \mid u_i \in (V \cup A_i)^*, S \to h(u_1u_2 \dots u_n) \in P\}$
Ensuite, pour étendre ce résultat à h morphisme quelconque, il suffit d'appliquer le lemme précédent. □

2.3.4 Intersection avec un rationnel

Proposition 2.3.4 (Intersection avec un rationnel).
Si L est algébrique et K est rationnel, alors $K \cap L$ est algébrique.

Remarque.
L'intersection de deux algébriques n'est pas algébrique en général.
Par exemple, soient $L_1 = \{a^nb^nc^* \mid n \geq 0\}$ et $L_2 = \{a^*b^nc^n \mid n \geq 0\}$, on a vu que L_1, L_2 étaient algébriques mais que $L_1 \cap L_2 = \{a^nb^nc^n \mid n \geq 0\}$ ne l'était pas.

Preuve.
Soit $G = (A, V, P)$ une grammaire telle que $L = L_G(S_0)$ en forme normale quadratique.

Méthode des automates Soit $\mathcal{A} = (Q, A, E, \{i\}, \{f\})$, un automate en forme normale acceptant K.
Le langage $K \cap L$ est engendré par la grammaire $G' = (A, V', P')$ avec :
– $V' = \{S_{p,q} \mid S \in V, (p, q) \in Q^2\}$
– $P' = \{S_{p,q} \to a \mid S \to a \in P \text{ et } p \xrightarrow{a} q \in E\} \cup \{S_{p,q} \to R_{p,r}T_{r,q} \mid S \to RT \in P\}$

Méthode des monoïdes Soit $\mu : A^* \to M$, un morphisme avec M fini reconnaissant K : $K = \mu^{-1}(P)$.
Le langage $K \cap L$ est engendré par la grammaire $G' = (A, V', P')$ avec :
– $V' = \{S_m \mid m \in M\}$
– $P' = \{S_m \to a \mid S \to a \in P \text{ et } m = \mu(a)\} \cup \{S_m \to R_{m_1}T_{m_2} \mid S \to RT \in P \text{ et } m = m_1m_2\}$

□

2.3.5 Théorème de Chomsky-Schützenberger

On consultera [Per90] pour une preuve élémentaire de ce théorème.

Définition 2.3.2 (Langage de Dyck).
Pour tout $n \in \mathbb{N}^*$, soit un alphabet $A_n = \{a_1, a_2, \ldots, a_n\}$. Le *langage de Dyck* D_n^* est défini par la grammaire $G = (A_n \cup \overline{A_n}, \{S, T\}, P)$, où $P = \{S \to ST, T \to \sum_{a \in A_n} aS\overline{a}\}$.

Théorème 2.3.2 (Chomsky - Schützenberger).
Un langage L est algébrique si et seulement si $L = \varphi(D_n^* \cap K)$ pour un langage rationnel K et un certain morphisme φ alphabétique.

Preuve.
La condition est suffisante grâce aux propriétés de clôture.

Réciproquement, soit G une grammaire en forme normale quadratique. On définit $A = \{a_r, b_r, c_r, \overline{a_r}, \overline{b_r}, \overline{c_r} \mid r = S \to S_1S_2\} \cup \{d_r, \overline{d_r} \mid r = S \to a\}$.

On définit la grammaire G' en remplaçant les règles :
- $S \to S_1S_2$ par $S \to a_r b_r S_1 \overline{b_r} c_r S_2 \overline{c_r a_r}$
- $S \to a$ par $S \to d_r \overline{d_r}$

et on définit φ par $\varphi(a_r) = \varphi(b_r) = \varphi(c_r) = \varphi(\overline{a_r}) = \varphi(\overline{b_r}) = \varphi(\overline{c_r}) = \varphi(\overline{d_r}) = \varepsilon$ et $\varphi(d_r) = a$. Alors $L = L_G(S) = \varphi(L_{G'}(S))$. Or $L_{G'}(S) = D_n^* \cap K$ où K décrit les contraintes (a_r suivit d'un b_r, $\overline{b_r}$ suivit d'un c_r, $\overline{c_r}$ suivit d'un $\overline{a_r}$ et d_r suivit d'un $\overline{d_r}$). □

2.4 Automates à pile

2.4.1 Définitions et exemple

Définition 2.4.1 (Automate à pile).
Un *automate à pile* est défini par :
- un alphabet d'entrée, A.
- un alphabet de pile, Z, parmi lesquels un symbole de pile initial z_0.
- des états, en nombre fini, Q, parmi lesquels un état initial q_0.
- des transitions de type :

$$q, y, z \to q', h$$

avec $q, q' \in Q$, $y \in A \cup \{\varepsilon\}$, $z \in Z$, $h \in Z^*$.

Son comportement est proche de celui d'un automate classique : il peut de plus interagir avec sa pile.

Définition 2.4.2 (Configuration).
On appele configuration, un triplet :

$$(q, f, h) \in Q \times A^* \times Z^*$$

correspondant à un "état" de l'automate à pile (c'est tout ce que l'automate a en mémoire à un moment donné).

Remarque.
C'est une pile LIFO.

Définition 2.4.3 (Calcul sur un automate à pile).

Une *étape de calcul* est un couple (C, C') d'éléments de $Q \times A^* \times Z^*$ tels que :

$$C = (p, yf, zw) \text{ et } C' = (q, f, hw)$$

où p, y, z, q, h constituent une transition $p, y, z \to q, h$. On la note $C \to C'$.

On appelle *calcul* (sur le mot m) une suite d'étapes de calcul partant de la configuration initiale (q_0, m, z_0).

2.4.2 Différents modes d'acceptation

Définition 2.4.4 (Configurations finales sur un automate à pile).

On peut choisir de considérer différentes conventions d'acceptation ; cela revient à *choisir les configurations finales*. Voici les cas usuels :

- pile vide : (q, m, ε)
- état final : (q, m, z) pour tout $q \in F \subseteq Q$
- une union des deux précédents

Exemple.

Soient l'automate à pile définit sur $A = \{a, b\}$, $Z = \{z\}$ et $Q = q_0, q_1, q_2$ et comportant les transitions suivantes :

$$\begin{aligned} q_0, b, z &\to q_2, z \\ q_0, a, z &\to q_1, zz \\ q_1, a, z &\to q_1, zz \\ q_1, b, z &\to q_2, \varepsilon \\ q_2, b, z &\to q_2, \varepsilon \end{aligned}$$

Si on choisit un arrêt par pile vide, il reconnaît le langage

$$L_1 = \{a^n b^p \mid 1 \leq n \leq p\}$$

Si on choisit l'arrêt par état final $F = q_2$, il reconnaît

$$L_2 = \{a^n b^p \mid n \geq 1,\ 1 \leq p\}$$

Proposition 2.4.1 (Équivalence des différents modes d'acceptation).

*Les différents modes d'acceptation (*i.e. *les différentes conventions de* configurations finales*) sont équivalents dans la mesure où aucune ne permet de reconnaître plus de langages que les autres.*

Preuve.

Voir TD5, exercice 3 (4 novembre).

On introduit l'équivalence par rapport à un automate à *fond de pile testable*, ie. pour lesquels il existe une partition de l'alphabet de pile $Z = Z_1 \cup Z_2$ telle que la pile de toute configuration accessible est dans $\{\{\varepsilon\} \cup Z_2^* Z_1\}$. □

2.4.3 Équivalence avec les grammaires

Définition 2.4.5 (Forme normale de Greibach).
Une grammaire $G = (A, V, P)$ est *sous forme normale de Greibach* si chacune de ses règles est de la forme

$$S \to w, w \in AV^*$$

Proposition 2.4.2 (Mise en forme normale de Greibach).
Toute grammaire est équivalente à une grammaire en forme normale de Greibach.

Théorème 2.4.1 (Equivalence langages algébriques/automates à pile).

$L \subseteq A^*$ est algébrique si et seulement si il existe un automate à pile qui reconnaît L.

Preuve.
Soit G une grammaire telle que $L = L_G(S_0)$.
En la supposant sous forme normale de Greibach, ses règles sont de la forme :

$$S \to aw, \text{ où } w \in V^*$$

Ce qui permet de construire un automate à pile reconnaissant L avec des transitions du type :

$$q_0, a, S \to q_0, w$$

Réciproquement, on peut remonter récursivement le caractère algébrique du langage reconnu par un automate à pile : en partant d'un état final et en montrant que le préfixe à gauche par les transitions de l'automate conserve le caractère algébrique. □

Deuxième partie

Calculabilité et complexité

Chapitre 3

Calculabilité et machines de Turing

3.1 Introduction

3.1.1 Notion de problème

Pour tout le début de ce chapitre, on se réfèrera à [Sip97].

Définition 3.1.1 (Problème de décision).
Un *problème de décision* est l'énoncé d'une question à laquelle on peut répondre par oui ou par non. Pour chaque problème, on fixe un ensemble E des *instances* et un sous-ensemble $P \subseteq E$ des instances pour lesquelles la réponse est « oui ».

Exemple.
On peut considérer les problèmes suivants :
- Nombres premiers : $E = \mathbb{N}$, $P = \{n \mid n \text{ premier}\}$.
- Automates (acceptance) : $E = \{(\mathcal{A}, w) \mid \mathcal{A} \text{ automate et} w \text{ mot}\}$, $P = \{(\mathcal{A}, w) \mid \mathcal{A} \text{ accepte} w\}$.
- Graphes connexes : $E = \{G \mid G \text{ graphe fini}\}$, $P = \{G \mid G \text{ fortement connexe}\}$.
- Grammaires ambiguës : $E = \{G \mid G \text{ grammaire}\}$, $P = \{G \mid G \text{ ambiguë}\}$ ou encore $P' = \{G \mid L_G(S) \text{ ambigu}\}$.

3.1.2 Notion de codage

Définition 3.1.2 (Codage associé à un problème).
Pour associer un langage à un problème, on utilise un *codage* qui est une fonction de E dans Σ^* (fonction « naturelle »). Le codage de $x \in E$ est noté $\langle x \rangle$.

Notation (Langage associé à un problème).
On définit $L_P = \{\langle x \rangle \mid x \in P\}$.

Exemple (Exemples de codage).

Nombres premiers On peut prendre pour $\langle n \rangle$ l'écriture en base 10 ou en base 2 de n ; ou encore en « base 1 » (écriture bâton) ($\Sigma = \{1\}$). Remarquons l'importance du caractère « naturel » du codage : si on prend

comme codage la factorisation de l'entier en produit de nombres premiers, caractériser P devient trivial.

Grammaires $G = (A, V, P)$, $\$ \notin A$; un codage convenable peut être : $\langle G \rangle = (\langle |A| \rangle \$ \langle |V| \rangle \$ \langle r_1 \rangle \$ \cdots \$ \langle r_n \rangle)$, où la règle $r_i = S \to w$, $w = a_1 a_2 \cdots a_n$, est codée par $\langle r_i \rangle = \langle S \rangle £ \langle n \rangle £ \langle a_1 \rangle £ \langle a_2 \rangle £ \cdots £ \langle a_n \rangle$.

3.1.3 Machines de Turing

Définition 3.1.3 (Machine de Turing).

Une *machine de Turing* est un 7-uplet $(Q, \Sigma, \Gamma, E, q_0, F, \#)$, où :

Q	états	q_0	état initial
Σ	alphabet d'entrée	F	états finaux
Γ	alphabet de bande	$\#$	symbole blanc
E	transitions		

avec $\Sigma \subseteq \Gamma$, $\# \in \Gamma \setminus \Sigma$, $F \subseteq Q$, $q_0 \in Q$ et $E \subseteq Q \times \Gamma \times Q \times \Gamma \times \{\leftarrow, \rightarrow\}$.

On représente la bande sur laquelle la machine de Turing travaille par :

a	z	#	b	c	z	b	#	x	u	r	#	#	#	#	#	#	#	⋯

La bande est infinie à droite et ne contient que des # à partir d'un certain rang. Un unique contrôle (la tête de lecture de la machine de Turing) est situé sur un caractère $a \in \Sigma$, distingué, de la bande ; elle est dans un état $q \in Q$. On repère le mot $u \in \Sigma^*$ situé strictement à gauche du caractère a ainsi distingué, et le mot $v \in \Sigma^*$ situé strictement à droite.

Une configuration est donc caractérisée par la donnée du découpage uav de la bande, ainsi que par l'état q : une configuration est usuellement notée $uqav$.

Les transitions sont notées : $q, a \to q', b, x$ où $x \in \{\leftarrow, \rightarrow\}$ (déplacement de la tête de lecture à gauche ou à droite).

Définition 3.1.4 (Calcul).

Un *calcul* est une suite de configurations successives $c_0 \rightsquigarrow c_1 \rightsquigarrow \cdots \rightsquigarrow c_k$, où chaque *étape* $c_i \rightsquigarrow c_{i+1}$ est associée à une transition $q, a \to q', b, x$ de la façon suivante :

- si $x = \rightarrow$ alors l'étape est : $uqav \rightsquigarrow ubq'v$.
- si $x = \leftarrow$ alors l'étape est : $udqav \rightsquigarrow uq'dbv$ ($d \in \Sigma$ est le caractère situé immédiatement à gauche du contrôle)[1].

Définition 3.1.5 (Calcul acceptant).

Un calcul $c_0 \rightsquigarrow c_1 \rightsquigarrow \cdots \rightsquigarrow c_k$ est dit *acceptant* lorsque :

- c_0 est *initiale*, c'est-à-dire $c_0 = q_0 w$ avec $w \in \Sigma^*$;
- c_k est finale, c'est-à-dire $c_k = uqv$ avec $q \in F$.

Définition 3.1.6 (Langage accepté).

On dit que w, un mot de Σ^* est *accepté* par M s'il existe un calcul acceptant de configuration initiale $q_0 w$.

[1] On remarquera que cette transition est uniquement possible si la tête de lecture n'est pas sur le premier symbole de la bande

On en déduit le *langage accepté* par M :

$$L(M) = \{w \mid w \text{ accepté par } M\}$$

.

Proposition 3.1.1 (Normalisation).
Pour toute machine de Turing M, il existe une machine de Turing M' telle que :

1. *$L(M) = L(M')$ et M se bloque si et seulement si M' se bloque ;*
2. *M' a deux états q_+ et q_- qui vérifient :*
   - *$F' = \{q_+\}$ (états finaux de M') ;*
   - *M' s'arrête en q_+ et en q_- ;*
   - *M' ne s'arrête qu'en q_+ et en q_-.*

On consultera, pour le principe de normalisation, l'appendice A, page 85.

3.1.4 Variantes

Bandes bi-infinies

Définition 3.1.7 (Machine à bande bi-infinie).
Une *machine à bande bi-infinie* est formellement identique à une machine de Turing, mais la bande est indicée par $\mathbb{Z}$.

	-3	-2	-1	0	1	2	3							
···	#	#	#	#	#	b	#	x	u	r	a	v	d	#	#	#	#	···

Proposition 3.1.2 (Équivalence).
Il y a équivalence entre les machines à bande bi-infinie et les machines de Turing, c'est à dire que pour toute machine de Turing à bande bi-infinie, il existe une machine de Turing à bande unique qui reconnaît le même langage.

Preuve.
Pour simuler les calculs d'une machine de Turing sur une machine à bande bi-infinie, il suffit de marquer le début de bande en fixant un caractère bloquant (par exemple $\& \notin \Gamma$) en position -1.

Pour simuler les calculs d'une machine à bande bi-infinie sur une machine de Turing, l'idée est de replier la bande en 0.

r	a	v	d	#	#	#	#	···
\$	u	x	#	b	#	#	#	···

Le nouvel alphabet de bande est $\Gamma \times (\Gamma \cup \{\$\})$: on considère à chaque position i dans la nouvelle bande le couple des caractères aux positions $(i, -i)$ dans l'ancienne bande, en plaçant sous le caractère 0, le symbole \$, supposé ne pas appartenir à l'alphabet de bande initiale.

On écrit l'entrée sur la partie supérieure de la bande : l'alphabet d'entrée devient donc $\Sigma \times \{\#, \$\}$, le nouveau caractère blanc $(\#, \#)$ et il est donc toujours exclu de l'alphabet d'entrée.

La machine doit mémoriser si elle lit la partie supérieure ou la partie inférieure de la bande, les états sont donc des paires $Q \times \{U, D\}$, avec U pour *Up* et D pour *Down*.

On transcrit alors les transitions de la machine d'origine, en prenant garde que la nouvelle machine se déplace désormais en sens contraire lorsqu'elle lit la partie inférieure de la bande. □

Machines à plusieurs bandes

Définition 3.1.8 (Machine à plusieurs bandes).
Une *machine à k bandes* est une machine disposant de k têtes de lecture (ou de contrôle) indépendantes, une sur chaque bande.

Une transition est alors un élément de l'ensemble $E \subseteq Q \times \Gamma^k \times Q \times \Gamma^k \times \{\overleftarrow{L}, \overrightarrow{R}, S\}^k$, où S (*Stay*) permet éventuellement de laisser immobile certaines têtes de lecture.

Proposition 3.1.3 (Équivalence).
Il y a équivalence entre les machines à plusieurs bandes et les machines de Turing.

Preuve.
Montrons comment convertir une machine M à k bandes en une machine de Turing mono-bande S équivalente.

S simule l'effet des k bandes en stockant leur information sur une unique bande, et utilise le symbole % pour délimiter le contenu des différentes bandes. Outre le contenu des bandes, S doit aussi garder la trace de la position des têtes. Elle possède donc un symbole de bande qui consiste à ajouter un point au-dessus des positions des têtes «virtuelles» de M. Les symboles additionnés d'un point ont ainsi été rajoutés à l'alphabet de bande. La figure 3.1 page 54 montre comment simuler une machine de Turing à trois bandes à l'aide d'une machine de Turing mono-bande.

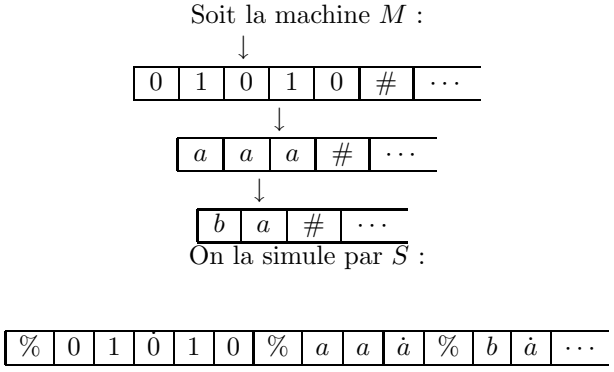
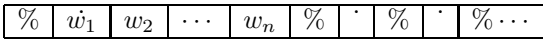


FIG. 3.1 – Simulation d'une machine à trois bandes.

Lorsque S prend en entrée un mot $w = w_1 \cdots w_n$:

1. S commence par organiser sa bande selon l'organisation décrite pour simuler les k bandes de M. Sa bande contient alors :



2. Pour simuler un mouvement donné, S se déplace sur la bande à partir du premier %, qui marque l'origine, jusqu'à au $(k+1)$-ème %, qui marque l'extrémité droite, pour déterminer quels symboles sont sous les têtes virtuelles. Puis, S fait un second passage pour mettre à jour les bandes en fonction de ce que la fonction de transition de M indique.
3. Si à n'importe quel moment S déplace l'une des têtes virtuelles sur un %, alors M a déplacé la tête correspondante sur une portion vierge et jamais lue de sa bande. S marque alors un symbole d'espacement sur cette case, et décale le contenu de toute la bande après cette case d'un rang sur la droite, puis continue la simulation.

□

Déterminisme

Définition 3.1.9 (Machine de Turing déterministe).
Une machine M est *déterministe* si pour chaque paire $(p, a) \in Q \times \Gamma$, il existe au plus un triplet $(q, b, x) \in Q \times \Gamma \times \{\leftarrow, \rightarrow\}$ tel que $p, a \rightarrow q, b, x \in E$.

Proposition 3.1.4.
Toute machine de Turing est équivalente à une machine déterministe.

Preuve.
Idée de la preuve :

On montre qu'il est possible de simuler toute machine non-déterministe N avec une machine déterministe D. L'idée de la simulation est de faire essayer à D toutes les branches possibles du calcul de N. Si D se trouve dans un état acceptant au cours de cette exploration, D accepte, et sinon, la simulation de D ne termine pas.

Le calcul de N sur une entrée w est vue comme un arbre : chaque branche de l'arbre représente une des possibilités d'exécution non-déterministe, chaque noeud de l'arbre est une configuration de N, et la racine de l'arbre est la configuration de départ. La machine D parcourt cet arbre pour trouver une configuration acceptante. Il est crucial cependant d'effectuer cette recherche avec soin pour que D visite bien tout l'arbre. Une première idée serait d'utiliser un algorithme DFS, mais D pourrait descendre indéfiniment le long d'une même branche infinie et manquer une configuration acceptante sur une branche voisine. On utilise donc un algorithme BFS.
Mise en application :

La machine déterministe D simulant N a trois bandes. D'après la proposition précédente, cet arrangement est équivalent à une machine à simple bande. La amchine D utilise ses trois bandes de manière spécifique : la première bande contient toujours l'entrée de la machine et n'est jamais modifiée, la seconde bande contient une copie de la bande de N selon une branche de son calcul, et la troisième bande enregistre la position de D sur l'arbre des calculs de N. La figure 3.2, page 56 montre l'organisation des bandes de D.

Considérons tout d'abord la représentation des données sur la troisème bande : chaque noeud de l'arbre peut avoir au plus b fils, ou b est le plus grand nombre de choix possible rencontré dans la fonction de transition de N. A chaque noeud de l'arbre, on associe alors une addresse qui est une chaîne prise dans l'alphabet $\Sigma_b = \{1, 2, \ldots, b\}$. On assigne ainsi l'addresse 231 au noeud atteint par le calcul

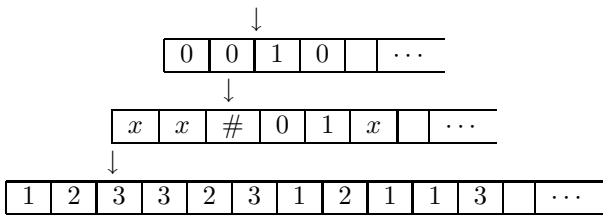


FIG. 3.2 – Une machine déterministe D simulant N

qui part de la racine, puis choisit son second fils, puis le troisième fils de celui-ci, puis le premier fils de ce dernier. Chaque symbole de la chaîne nous montre ainsi les choix à faire lorsqu'on simule une étape du calcul de N. Remarquons qu'un symbole peut parfois ne correspondre à aucun choix si un noeud donné a trop peu de fils. La troisième bande de D contient ainsi une chaîne sur l'alphabet $\sum_b$: elle représente à chaque case le chemin de calcul de la racine au noeud étiqueté par cette case (sauf si l'addresse est invalide). On peut alors décrire le fonctionnement de D :

1. Initialement, la première bande contient l'entrée w, et les bandes 2 et 3 sont vides.
2. Copier la première bande sur la seconde.
3. Utiliser la seconde bande pour simuler N avec l'entrée w sur une branche de son calcul. Avant chaque étape de N consulter le symbole suivant sur la troisième bande, pour déterminer quel choix faire parmi ceux autorisés par la fonction de transisition de N. S'il ne reste plus de symboles sur la troisième bande, ou que ce choix est invalide, on abandonne cette branche en allant à l'étape 4. De même, on abandonne si on rencontre une configuration bloquante. Si une configuration acceptante est rencontreée, on *accepte* l'entrée w.
4. Remplacer la chaîne de la troisième bande avec la chaîne qui suit dans l'ordre lexicographique (BFS). Simuler cette branche de N, en retournant à l'étape 2.

□

3.2 Langages/problèmes récursivement énumérable

Définition 3.2.1 (Langages *r.e.*).
Le langage $L \subseteq A^*$ est dit *récursivement énumérable* (*r.e.*) s'il existe une machine de Turing M telle que $L = L(M)$. Par extension, un problème P est dit *r.e.* si L_P est *r.e.*.

Définition 3.2.2 (Énumérateur).
Une machine M (éventuellement à plusieurs bandes) est un énumérateur si elle écrit sur sa (première) bande des mots sur Σ^* séparés par $\# \notin \Sigma$.

Proposition 3.2.1.
L est r.e. *si et seulement si L est l'ensemble des mots énumérés par un énumérateur.*

Preuve.

La thèse de Church affirme que tout ce qui est calculable (donc, en particulier les algorithmes) l'est par machine de Turing. Parfois, on utilisera implicitement ce fait, et on écrira donc des programmes au lieu de décrire une machine de Turing.

Algorithme (passer d'une machine de Turing à un énumérateur).

for all $k \geq 0$ **do**
  for all $w \mid |w| \leq k$ **do**
    if M accepte w en au plus k étapes **then**
      imprimer w
    end if
  end for
end for

Algorithme (passer d'un énumérateur à une machine de Turing).

loop
  repeat
    Exécuter l'énumérateur
  until l'énumérateur imprime #
  if w est identique au mot imprimé **then**
    accepter
  end if
end loop

□

Exemple (Langage diagonal).

Pour un alphabet $\Sigma = \{a, b\}$ fixé, indexons par $\mathbb{N}$ l'ensemble des machines de Turing sur Σ : $M_0, M_1, \ldots$ (cela est possible puisqu'elles sont en nombre dénombrable) ; et indiçons aussi les mots de Σ^* : $w_1, w_2, \ldots$.

On pose alors $L = \{w_i \mid w_i \notin L(M_i)\}$.

Montrons que L n'est pas *r.e.* : s'il existe une Machine de Turing M telle que $L = L(M)$, notons i l'indice de cette machine ($M = M_i$). Si $w_i \in L(M_i)$, alors $w_i \notin L(M)$ et si $w_i \notin L(M_i)$, alors $w_i \in L(M)$, ce qui induit une contradiction.

Il existe donc des langages non reconnaissables par une machine de Turing.

Proposition 3.2.2 (Clôture par union et par intersection).

Si L et L' sont r.e.*, alors $L \cup L'$ et $L \cap L'$ sont* r.e..

Preuve.

L et L' sont chacun reconnus par une machine à une bande, respectivement M et M'.

$M_\cup$ (qui reconnait $L \cup L'$) peut se construire avec deux bandes : on commence par copier l'entrée de la première bande sur la deuxième, puis on alterne la simulation de M sur la première bande et celle de M' sur la deuxième.

$M_\cap$ (qui reconnait $L \cap L'$) est plus simple, car si une des deux machines M ou M' boucle, on peut se permettre de boucler aussi. Il suffit donc de simuler M, et si l'entrée est acceptée, alors on simule M'. □

3.3 Langage décidable

Les langages décidables sont parfois appelés *langages récursifs*, et les langages récusivement énumérables *langages semi-récursifs*.

Définition 3.3.1 (Langage décidable).
On dit que $L \subseteq \Sigma^*$ est *décidable* lorsqu'il existe M, une machine de Turing sans calcul infini (*i.e.* la machine M s'arrête sur toute entrée) telle que $L = L(M)$.

Proposition 3.3.1 (Sous-classe des langages *r.e.*).
Si L est décidable, alors L est r.e..

Preuve.
Si L est décidable, alors, par définition, il existe une machine M telle que $L(M) = L$. □

Proposition 3.3.2 (Clôture par union et par intersection).
Si L et L' sont décidables, alors $L \cup L'$ et $L \cap L'$ sont décidables.

Preuve.
Les machines construites pour la clôture des machines *r.e.* sont sans calcul infini lorsque les machines M et M' sont elles-mêmes sans calul infini. □

Lemme 3.3.1 (Kônig).
Si M s'arrête toujours, alors pour tout mot $\omega \in \Sigma^*$, les chemins d'évaluation sont bornés.

Proposition 3.3.3 (Clôture par complémentation).
Si $L \subseteq \Sigma^*$ est décidable, alors $\Sigma^* \setminus L$ est décidable.

Preuve.
Soit M, une machine de Turing déterministe, normalisée (qui accepte en q_+ et rejette en q_-) et sans calcul infini reconnaissant L.
Si l'on note $M = (Q, \Sigma, \Gamma, E, q_0, \{q_+\}, \#)$,
la machine $M' = (Q, \Sigma, \Gamma, q_0, \{q_-\}, \#)$, sans calcul infini, reconnait $\Sigma^* \setminus L$. □

Proposition 3.3.4 (Complémentation et décidabilité).
Soit $L \subseteq \Sigma^*$, si L et $\Sigma^* \setminus L$ sont r.e.*, alors L et $\Sigma^* \setminus L$ sont décidables.*

Preuve.
Il suffit de construire $M_\cup$ pour M et M', des machines de Turing qui reconnaissent respectivement L et $\Sigma^* \setminus L$; cette machine s'arrête pour tout mot w (puisqu'il est soit dans L, soit dans $\Sigma^* \setminus L$).
Il ne reste plus qu'à rejeter les mots de $\Sigma^* \setminus L$ pour reconnaître L (ou *vice-versa*). □

Notation (Codage d'une machine de Turing).
Si M machine, on note $\langle M \rangle$ le codage de M.

Notation (Codage d'un mot).
Si w mot, on note $\langle w \rangle$ le codage de w.

Notation (Codage d'une requête).

Si M machine et w mot, on note $\langle M, w\rangle$ le codage du couple (M, w) (par exemple, $< M > \$ < w >$, où \$ est un caractère qui n'apparaît ni dans les codages des machines de Turing ni dans les codages de mots).

Définition 3.3.2 (Langage d'acceptation).

$$L_{acceptation} = \{\langle M, w\rangle \mid w \in L(M)\}$$

Proposition 3.3.5 (Machine universelle).

Il existe une machine M_U telle que $L(M_U) = L_{acceptation}$. Une telle machine est dite universelle.

Cette machine peut être construite sur trois bandes : la première bande prend l'entrée $\langle M, w\rangle$. Les données w sont recopiées sur la bande de simulation, disons la troisième bande. M_U simule le comportement de M sur w. La machine M n'est pas forcément déterminisite : durant la simulation, on maintient l'état courant sur la seconde bande.

Proposition 3.3.6 (Indécidabilité du langage d'acceptation).

$L_{acceptation}$ *est* r.e. *mais pas décidable.*

Preuve.

D'après la définition précédente, $L_{acceptation} = L(M_U)$. $L_{acceptation}$ est donc *r.e.*.

Supposons que $L_{acceptation}$ est décidable : il existe une machine A qui « décide » $L_{acceptation}$ (c'est-à-dire qui accepte $L_{acceptation}$ et qui s'arrête toujours).

Définissons la machine Q par :

entrée $\langle M\rangle$
if A accepte $\langle M, \langle M\rangle\rangle$ **then**
  rejeter
else
  accepter
end if

Lançons alors Q sur l'entrée $\langle Q\rangle$.

- Si Q accepte $\langle Q\rangle$, c'est que A refuse $\langle Q, \langle Q\rangle\rangle$. Or A refuse $\langle Q, \langle Q\rangle\rangle$ signifie précisément que Q refuse $\langle Q\rangle$, par définition de A.
- Si Q n'accepte pas $\langle Q\rangle$, c'est que A accepte $\langle Q, \langle Q\rangle\rangle$. Or A accepte $\langle Q, \langle Q\rangle\rangle$ signifie précisément que Q accepte $\langle Q\rangle$.

Dans les deux cas, on a une contradiction ; ce qui montre que $L_{acceptation}$ n'est pas décidable □

Définition 3.3.3 (Réduction).

Soient A et B deux problèmes, d'alphabets respectifs Σ_A et Σ_B et de langages respectifs L_A et L_B. Une réduction de A à B est une fonction $f : \Sigma_A^* \to \Sigma_B^*$ calculable par une machine de Turing telle que $w \in L_A \Leftrightarrow f(w) \in L_B$.

Notation (Réduction).

On notera $A \leq_m B$ lorsque A se réduit à B.

Proposition 3.3.7 (Décidabilité par réduction).

Si $A \leq_m B$ et B est décidable, alors A est décidable.

Preuve.

$A \leq_m B$ donc il existe une machine M qui s'arrête toujours et qui, pour toute entrée $w \in \Sigma_A^*$, calcule $w' \in \Sigma_B^*$, tel que $w \in L_A \Leftrightarrow w' \in L_B$.

B est décidable donc il existe une machine M' qui s'arrête toujours telle que $L(M') = L_B$.

On définit une machine M'' qui prend en entrée un mot $w \in \Sigma_A^*$, qui exécute M sur cet entrée, puis qui exécute M' sur la sortie de M. M'' accepte si M' accepte et rejette si M' rejette.

Alors M'' s'arrête toujours et par construction, $M(L(M'')) = L(M') = L_B$ donc $L(M'') = L_A$ par définition de M. □

Corollaire 3.3.1 (Indécidabilité par réduction).

Par contraposition, si $A \leq_m B$ et A est indécidable, alors B est indécidable.

Exemple.

Soient les langages :

- $L_\emptyset = \{\langle M \rangle \mid L(M) \neq \emptyset\}$
- $L_\neq = \{\langle M_1, M_2 \rangle \mid L(M_1) \neq L(M_2)\}$

$L_\emptyset$ et $L_\neq$ sont indécidables.

- Pour $L_\emptyset$, considérons la machine A de $L_{acceptant}$, prenant en entrée un codage $\langle M, w \rangle$, et la machine M' définie par :

  entrée u
  if $u = w$ **then**
  if A accepte $< M, u >$ **then**
  accepter
  else
  rejeter
  end if
  else
  rejeter
  end if

  Soit f la fonction $\langle M, w \rangle \mapsto \langle M' \rangle$.
  On a $w \in L(M) \Leftrightarrow w \in L(M') \Leftrightarrow L(M') \neq \emptyset$ (car M' n'accepte que w).
  f est donc une réduction, soit $L_{acceptation} \leq_m L_\emptyset$, ce qui conclut.
- Pour $L_\neq$, considérons la machine de $L_\emptyset$, prenant en entrée un codage $\langle M \rangle$, et la machine M' définie par :

  entrée u
  rejeter

  Soit f la fonction $\langle M \rangle \mapsto \langle M, M' \rangle$. On a $L(M) \neq \emptyset \Leftrightarrow L(M) \neq L(M')$, donc f est une réduction de $L_\emptyset$ à $L_\neq$.

3.4 Problème de correspondance de Post (PCP)

3.4.1 Présentation et indécidabilité

Définition 3.4.1 (Problème de correspondance de Post).

- Instance : un entier, $m > 0$, et $\boxed{\begin{matrix} u_1 \\ v_1 \end{matrix}}, \boxed{\begin{matrix} u_2 \\ v_2 \end{matrix}}, \ldots, \boxed{\begin{matrix} u_m \\ v_m \end{matrix}}$ une suite de couples de mots : $(u_i, v_i) \in (\Sigma^*)^2$. On peut voir chaque couple comme un domino sur lequel deux mots sont écrit, l'un en dessous de l'autre.
- Solution : un entier, $n > 0$, et une suite d'indices, $i_1, i_2, \ldots, i_n \in \{1, \ldots, m\}$, telle que :
$$u_{i_1} u_{i_2} \ldots u_{i_n} = v_{i_1} v_{i_2} \ldots v_{i_n}$$
- Problème : Y a-t-il au moins une solution ? C'est à dire, y-a-t-il une disposition des dominos côte-à-côte telle qu'on lise le même mot sur la partie supérieure et inférieure des dominos ?

Définition 3.4.2 (Problème de correspondance de Post modifié (PCPM)).

- Solution : un entier, $n > 0$, et une suite d'indices, $i_1, i_2, \ldots, i_n \in \{1, \ldots, m\}$, telle que
  1. $u_{i_1} u_{i_2} \ldots u_{i_n} = v_{i_1} v_{i_2} \ldots v_{i_n}$;
  2. $i_1 = 1$

Lemme 3.4.1 (Équivalence des deux problèmes).

$$\text{PCP } \textit{indécidable} \Leftrightarrow \text{PCPM } \textit{indécidable}$$

Preuve.

1. PCPM décidable $\Rightarrow$ PCP décidable
   On teste PCPM pour les m instances où on fait passer chaque domino en tête.
   Remarque.
   Vu qu'on a réduit une instance de PCP à plusieurs instances de PCPM, il ne s'agit pas d'une réduction $\leq_m$, mais d'une réduction de Turing, $\leq_T$.
2. PCP décidable $\Rightarrow$ PCPM décidable
   Soit une instance de PCPM : $m > 0, \boxed{\begin{matrix} u_1 \\ v_1 \end{matrix}}, \boxed{\begin{matrix} u_2 \\ v_2 \end{matrix}}, \ldots, \boxed{\begin{matrix} u_m \\ v_m \end{matrix}}$.
   En notant Σ l'alphabet utilisé par PCPM, on se place dans l'alphabet $\Sigma \cup \{\$\}$, avec $\$ \notin \Sigma$.
   On définit deux morphismes de Σ^* dans $(\Sigma \cup \{\$\})^*$ par :
   - $p(a_1 a_2 \ldots a_k) = \$a_1 \$a_2 \ldots \$a_k$;
   - $s(a_1 a_2 \ldots a_k) = a_1 \$a_2 \ldots \$a_k \$$.

   On observe que $p(w)\$ = \$s(w)$.
   On définit alors l'instance de PCP suivante sur l'alphabet $\Sigma \cup \{\$\}$:
$$2m+1, \boxed{\begin{matrix} u'_1 \\ v'_1 \end{matrix}}, \boxed{\begin{matrix} u'_2 \\ v'_2 \end{matrix}}, \ldots, \boxed{\begin{matrix} u'_{2m+1} \\ v'_{2m+1} \end{matrix}}$$
   où on a posé :
   - pour $1 \leq k \leq m$, $u'_k = p(u_k)$ et $v'_k = s(v_k)$;
   - pour $1 \leq k \leq m$, $u'_{m+k} = p(u_k)\$$ et $v'_{m+k} = s(v_k)$;
   - $u'_{2m+1} = p(u_1)$ et $v'_{2m+1} = \$s(v_1)$.

Les seules solutions de l'instance de PCP ont la forme $i_1, i_2, \ldots, i_n$, où :
- $i_1 = 2n + 1$;
- $m + 1 \leq i_n \leq 2m$;
- $1 \leq i_l \leq m$ pour $2 \leq l \leq n - 1$.

et alors, par construction, $1, i_2, \ldots, i_{n-1}, (i_n - m)$ est à chaque fois une solution de l'instance de PCPM.

Remarque.

Il s'agit d'une réduction $\leq_m$.

Remarque.

On doit éventuellement couper au premier « \$\$ ».

□

Théorème 3.4.2 (Indécidabilité).
PCP *et* PCPM *sont indécidables.*

Preuve.

Montrons que $L_{acceptant}$ se réduit à PCPM. Si $w \in L(M)$, on a les configurations successives $C_0 = q_0 w \Rightarrow C_1 \rightsquigarrow \cdots \rightsquigarrow C_l$, avec acceptation en C_l[2].

On va construire une instance PCPM qui « emboîte » le mot « $C_0\$C_1\$\ldots\$C_l$ ».

Soit l'instance dans laquelle les dominos sont :
- domino initial : $\boxed{\begin{matrix}\$ \\ \$q_0 w\$\end{matrix}}$
- pour tout $a \in \Gamma \cup \{\$\}$, $\boxed{\begin{matrix}a \\ a\end{matrix}}$.
- pour tout $p, a \to q, b, R \in E$, $\boxed{\begin{matrix}pa \\ bq\end{matrix}}$.
- pour tout $p, a \to q, b, L \in E$, $\boxed{\begin{matrix}cpa \\ qcb\end{matrix}}$.
- deux dominos $\boxed{\begin{matrix}\$ \\ \#\$\end{matrix}}$ et $\boxed{\begin{matrix}\$ \\ \$\end{matrix}}$.

Ramenons-nous à $F = \{q_+\}$ unique état final, $\boxed{\begin{matrix}aq_+ \\ q_+\end{matrix}}$ et $\boxed{\begin{matrix}q_+a \\ q_+\end{matrix}}$. □

3.4.2 Application aux grammaires

Définition 3.4.3 (Grammaires déduites d'un PCP).
Pour toute instance $\boxed{\begin{matrix}u_1 \\ v_1\end{matrix}}, \boxed{\begin{matrix}u_2 \\ v_2\end{matrix}}, \ldots, \boxed{\begin{matrix}u_m \\ v_m\end{matrix}}$ sur A^*, on introduit les m nouvelles lettres : $X = \{a_1, a_2, \ldots, a_m\}$.

On pose alors :

$$L_U = \{a_{i_1} a_{i_2} \ldots a_{i_n} u_{i_n} u_{i_{n-1}} \ldots u_{i_1} \mid 1 \leq i_k \leq m, n \geq 0\}$$

$$L'_U = \{a_{i_1} a_{i_2} \ldots a_{i_n} w \mid w \in A^*, w \neq u_{i_n} u_{i_{n-1}} \ldots u_{i_1}\}$$

Lemme 3.4.3 (Grammaires algébriques).
L_U *et* L'_U *sont algébriques.*

[2]On suppose ainsi que la machine efface la bande avant de s'arrêter

Preuve.
On obtient L_U à partir de la règle :

$$S \to \sum_{i=1}^{n} a_i S u_i + \varepsilon$$

On obtient L'_U à partir des règles (avec S pour axiome) :

- $S \to \sum_{i=1}^{n} a_i S u_i + T$
- $T \to \sum_{\substack{1\leq i\leq m \\ |u|=|u_i| \\ u\neq u_i}} a_i R u + \sum_{\substack{1\leq i\leq m \\ |u|<|u_i|}} a_i V u$
- $R \to \sum_{i=1}^{n} a_i R + \sum_{b\in A} R b + \varepsilon$
- $V \to \sum_{i=1}^{n} a_i V + \varepsilon$

□

Lemme 3.4.4.
On a $(A+X)^* \setminus L_U = L'_U \cup ((A+X)^* \setminus X^*A^*)$

Preuve.
Il suffit de remarquer que $L_U \subseteq X^*A^*$ et $X^*A^* \setminus L_U = L'_U$. □

Proposition 3.4.1.
Les problèmes suivants sont indécidables :

1. *pour deux grammaires G et G', est-ce que $L_G(S) \cap L_{G'}(S') = \emptyset$?*
2. *pour deux grammaires G et G', est-ce que $L_G(S) = L_{G'}(S')$?*
3. *pour une grammaires G, est-ce que $L_G(S) = A^*$?*
4. *pour une grammaire G, est-ce que G est ambiguë ?*

Preuve.
Pour 1., le PCP a une solution si et seulement si $L_U \cap L_V \neq \emptyset$.
Pour 3., $L_U \cap L_V = \emptyset \Leftrightarrow L_U^c \cup L_V^c = A^*$. Et ainsi, on réduit 1. à 3..
Pour 4., lorsqu'on a $S \to S_1 + S_2$,
On obtient L_U par $S_1 \to \sum_{i=1}^{m} a_i S_1 u_i + \sum_{i=1}^{m} a_i u_i$.
On obtient L_V par $S_1 \to \sum_{i=1}^{m} a_i S_1 v_i + \sum_{i=1}^{m} a_i v_i$. □

3.5 Quines, théorème de récursion et point fixe

Théorème 3.5.1 (Théorème de récursion).
Il existe des machines de Turing qui écrivent leur propre codage.

Preuve.
On introduit les machines suivantes :

1. étant donné w, un mot, la machine A_w affiche w

2. pour une entrée w, la machine B' affiche $\langle A_w \rangle$.
   Il faut, bien entendu, contourner les problèmes de notations — ie. remplacement de " par \", \ par \\ — qu'on rencontre dans les langages de programmation usuels.
3. pour une entrée w, la machine B affiche $\langle A_w \rangle.w$

On considère la machine $\boxed{A_{\langle B \rangle} \mid B}$. Autrement dit, on lance $A_{\langle B \rangle}$, puis B avec pour entrée la sortie produite par $A_{\langle B \rangle}$.

Mécanisme : $\overset{A_{\langle B \rangle}}{\rightarrow} \langle B \rangle \overset{B}{\rightarrow} \langle A_{\langle B \rangle} \rangle \langle B \rangle$ □

Définition 3.5.1 (Quines).
On appelle *quines* de telles machines.

Proposition 3.5.1.
Soit $t : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ une fonction calculable par machine de Turing. Il existe une machine de Turing M qui calcule la fonction $m : w \mapsto t(w, \langle M \rangle)$.

Preuve.
Soit T, une machine qui calcule t. Considérons la machine $\boxed{A_{\langle BT \rangle} \mid B \mid T}$
$\overset{A_{\langle BT \rangle}}{\rightarrow} \langle BT \rangle \overset{B}{\rightarrow} \langle A_{\langle BT \rangle} \rangle \langle BT \rangle$
Il ne reste plus qu'à insérer w en tête de la sortie ainsi obtenue, et de passer ceci en entrée de T. □

Théorème 3.5.2 (Point fixe).
Soit t une fonction calculable par machine de Turing qui à $\langle M \rangle$ associe une machine de Turing $M' = t(\langle M \rangle)$. Alors, il existe une machine M telle que M et $t(M)$ sont équivalentes.

Preuve.
On considère la machine M suivante (qui est correctement définie d'après la proposition précédente) :

entrée w
$x := \langle M \rangle$
$y := t(x)$
simuler y sur l'entrée w

□

3.6 Décidabilité de théorie logique

3.6.1 Modèles logiques sur $\mathbb{N}$

- Modèle : $\mathbb{N}$.
- Opérateurs booléens : $\vee$, $\wedge$, $\neg$
- Quantificateur : $\forall$, $\exists$
- Opérations : $+$, $\times$

Définition 3.6.1 (Formules closes).
Une *formule close* est une formule f où toute variable dépend d'un quantificateur.

Définition 3.6.2.
On dit qu'une *théorie logique* est *décidable* s'il est décidable de savoir si une formule close est vraie.

Intéressons-nous aux deux théories : $\mathsf{Th}\langle\mathbb{N},+\rangle$ et $\mathsf{Th}\langle\mathbb{N},+,\times\rangle$.

Définition 3.6.3 (Forme prénexe).
La formule φ est en *forme prénexe* lorsque :

$$\varphi = Q_1 x_1 Q_2 x_2 \dots Q_n x_n \psi$$

où, pour tout i, $Q_i = \forall$ ou $Q_i = \exists$ et ψ ne contient plus de quantificateur.

Théorème 3.6.1 (Pressburger).
La théorie $\mathsf{Th}\langle\mathbb{N},+\rangle$ est décidable.

Preuve.
On prend une formule close φ, qu'on suppose sous forme prénexe : $\varphi = Q_1 x_1, Q_2 x_2, \dots, Q_n x_n \psi$.
On pose $\varphi_k = Q_{k+1} x_{k+1}, \dots, Q_n x_n \psi$ avec $\varphi_0 = \varphi$ et $\varphi_n = \psi$.
Alors, $\varphi_k(x_1, x_2, \dots, x_k)$ a k variables libres.
On se place sur l'alphabet $\Sigma_k = \{0,1\}^k$ et on pose

$$X_k = \{(n_1, \dots, n_k) \mid \varphi_k(n_1, n_2, \dots, n_k) \text{ est vraie}\}$$

On construit par récurrence décroissante sur $k \leq n$, un automate $\mathcal{A}_k$ qui « accepte » X_k.
Ainsi, l'automate $\mathcal{A}_0$ (définit sur l'alphabet vide) accepte ε si et seulement si φ est vraie.
Construction de l'automate $\mathcal{A}_n$ (qui correspond à la formule ψ) :
On peut se ramener au cas où ψ est une combinaison booléenne de

$$\begin{cases} x_i = x_j \\ x_i + x_j = x_k \end{cases}$$

.

Automate de l'addition

On construit un automate « codéterministe » en faisant comme s'il lisait de la droite vers la gauche (C est l'état avec retenue, NC est l'état sans retenue). Voir Fig. 3.3, page 66.

Passage de $\mathcal{A}_{k+1}$ à $\mathcal{A}_k$

$\varphi_k = Q_{k+1} x_{k+1}, \varphi_{k+1}$

1. si $Q_{k+1} = \exists$: chaque transition de l'automate A_k provient d'une transition de l'automate A_{k+1}, cette dernière dépendant en plus du bit du $k+1$ nombre : on considère donc les deux alternatives possibles pour ce bit. Si le $(k+1)^{\text{e}}$ nombre est plus long que tous les autres, il suffit de le tronquer en oubliant les lectures initiales, Fig. 3.4, page 66.
2. si $Q_{k+1} = \forall$: $\varphi_k = \forall x_{k+1}, \varphi_{k+1} = \neg\exists x_{k+1}, \neg\varphi_{k+1}$

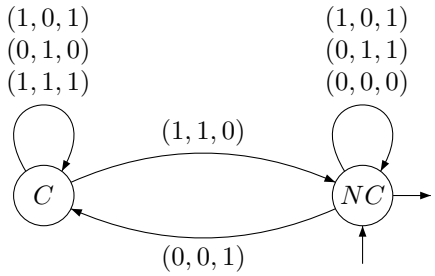


FIG. 3.3 – Automate de l'addition

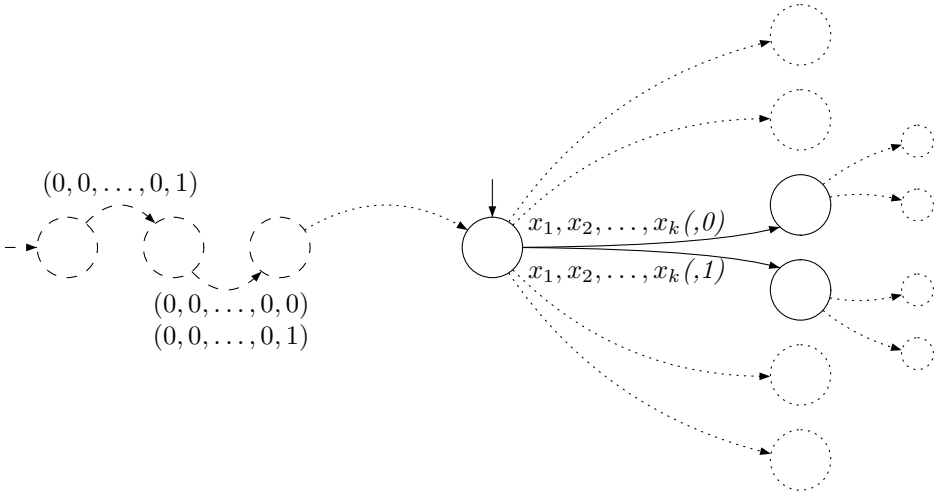


FIG. 3.4 – Passage de A_{k+1} à A_k pour $Q_{k+1} = \exists$

□

Théorème 3.6.2 (Gödel).
La théorie $\mathsf{Th}\langle \mathbb{N}, +, \times \rangle$ *est indécidable.*

Preuve.
On réduit le problème de l'acceptation à ce problème : $M, w \rightsquigarrow \exists x, \varphi_{M,w}$, où x code le calcul et il existe si et seulement si w est acceptant. Voir preuve dans le cours de logique, chapitre 8. □

3.6.2 Critères de divisibilité

Plaçons nous en base b, soit avec l'alphabet $\{0, 1, \ldots, b-1\}$. Considérons le problème suivant : comment, dans cette base, déterminer si un langage est rationnel ?

On peut par exemple construire l'automate discutant le reste de la division euclidienne par 3 en base 2 (voir FIG. 3.5, page 67). Plus généralement, pour discuter le reste de la division euclidienne par q en base r, on construit l'automate à q états représentant les restes euclidiens $0, 1, \ldots, q-1$, et $r \xrightarrow{d} r'$ est une

transition de l'automate si et seulement si $br + d = qk + r'$ (voir FIG. 3.6, page 67).

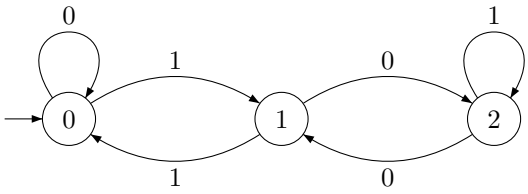


FIG. 3.5 – Automate de la division euclidienne par 3 en base 2

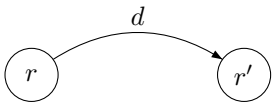


FIG. 3.6 – Division euclidienne par q en base b : $br + d = qk + r'$

Définition 3.6.4 (Sous-ensemble de $\mathbb{N}$ ultimement périodique).

$E \subseteq \mathbb{N}$ est *ultimement périodique* lorsque $\exists N, \exists p \mid \forall n \geq N, n \in E \Leftrightarrow n + p \in E$

.

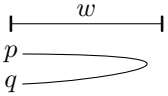
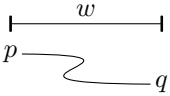
Définition 3.6.5 (Entiers multiplicativement indépendants).
b et b' sont *multiplicativement indépendants* lorsque $\forall n, m > 0, b^n \neq b'^m$.

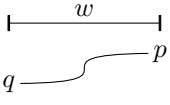
Théorème 3.6.3 (Cobhem).
Soient b et b' deux entiers multiplicativement indépendants, si $E \subseteq \mathbb{N}$ est rationnel en bases b et b', alors E est ultimement périodique.

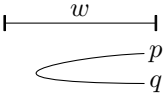
3.6.3 Machine de Turing avec une entrée en lecture seule

Théorème 3.6.4.
Soit M une machine de Turing qui n'écrit jamais sur son entrée, alors $L(M)$ est rationnel.

Preuve.
Pour $w \in \Sigma^*$, soient les ensembles de couples d'états suivants (entre lesquels il existe un chemin de la forme indiquée) :

$dd(w) = \{(p,q) \mid$  $\}$, $df(w) = \{(p,q) \mid$  $\}$ $fd(w) =$



$\}$, $ff(w) = \{(p,q) \mid$  $\}$

On définit : $w \sim w' \Leftrightarrow \begin{cases} dd(w) = dd(w') \\ df(w) = df(w') \\ fd(w) = fd(w') \\ ff(w) = ff(w') \end{cases}$

Alors :

– $\sim$ est une relation d'équivalence.

– le nombre de ses classes est majoré par $2^{4 \cdot |Q|^2}$

– $\sim$ est une congruence (voir FIG. 3.7, page 69)

– $w \sim w' \Rightarrow (w \in L(M) \Leftrightarrow w' \in L(M))$ (voir FIG. 3.8, page 69)

donc L est reconnu par $A^*/\sim$. Toutefois, notez bien que le passage de la machine à l'automate n'est pas décidable. □

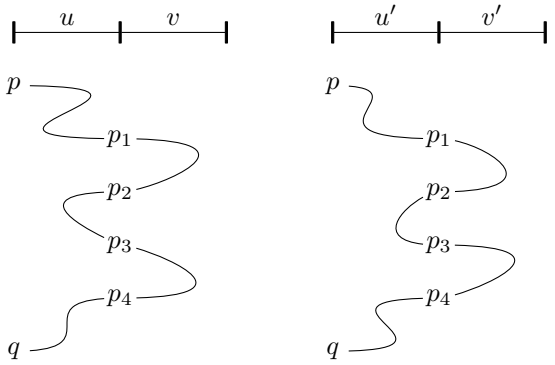


FIG. 3.7 – $\sim$ est une congruence

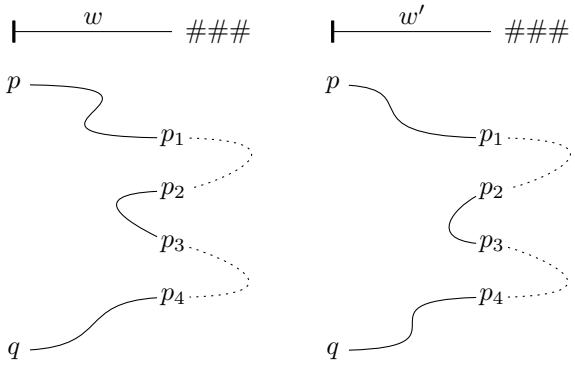


FIG. 3.8 – $w \sim w' \Rightarrow (w \in L(M) \Leftrightarrow w' \in L(M))$

Chapitre 4

Complexité en temps et en espace

4.1 Introduction

4.1.1 Objectifs

Le problème de la complexité se pose en temps comme en espace.

Il s'agit de déterminer s'il est possible de trouver un algorithme efficace pour résoudre un problème (par symétrie, de l'algorithmique qui cherche à déterminer le-dit algorithme). On tentera par ailleurs de classifier les problèmes selon leur complexité, par des méthodes beaucoup plus larges que celles de l'algorithmique.

On ne considérera naturellement ici que des problèmes décidables (*i.e.* décidés par des machines de Turing qui s'arrêtent toujours).

4.1.2 Représentation de la complexité

Définition 4.1.1 (Complexité).

Soit une machine de Turing M (*a priori* non déterministe).

Soit un calcul $C_0 = q_0 w \rightsquigarrow C_1 \cdots \rightsquigarrow C_m$ de M d'entrée w :

– le *temps* de ce calcul est m.

– l'*espace* de ce calcul est $\max_{0 \leq i \leq m} |C_i|$.

On définit alors la complexité en temps (resp. espace) pour un mot w comme la plus grande complexité en temps (resp. espace) des calculs de w :

$$\begin{aligned} t_M(w) &= \max_{\text{calculs}} \text{temps(calcul)} \\ s_M(w) &= \max_{\text{calculs}} \text{espace(calcul)} \end{aligned}$$

On peut alors définir la complexité de la machine M par :

$$\begin{aligned} t_M(n) &= \max_{|w|=n} t_M(w) \\ s_M(n) &= \max_{|w|=n} s_M(w) \end{aligned}$$

Proposition 4.1.1.

Si $t_M(n) \geq n$ *alors* $s_M(n) \leq t_M(n)$.

4.2 Complexité en temps

4.2.1 Théorème d'accélération

Théorème 4.2.1 (d'accélération).
Soit $k \in \mathbb{N}$ et M, une machine de Turing. Si $n = \mathbf{O}(t_M(n))$, alors il existe une machine de Turing M', équivalente à M telle que $t_{M'}(n) \leq (1/k)t_M(n)$

Preuve.

Σ : | a_0 | a_1 | | | | | | | $\cdots$ |

On opère la transformation $\Sigma' = \Sigma \times \Sigma$ (ou, plus généralement, $\Sigma' = \Sigma^k$) :

Σ' : | (a_0, a_1) | | | | | | | | $\cdots$ |

□

Ainsi, lorsqu'on étudie la complexité d'un problème, les constantes multiplicatives ne sont pas significatives : on préfère donc exprimer une complexité sous la forme $\mathbf{O}(f)$.

4.2.2 Changement de modèle

Quelle est l'influence du modèle de machine sur la complexité ?

- **machine à bande bi-infinie** :
  La transformation faite, la complexité de la simulation reste la même.
- **machine à plusieurs bandes** :
  Dans l'algorithme de transformation, on fait des allers-retours pour reconstituer le k-uplet représentant l'état de la machine à k bandes. Au pire, le parcours fait pour simuler la i^{e} étape est de longueur $s_M(n) \leq t_M(n)$. Au final, on a donc :
  $$t_{M'}(n) = \mathbf{O}(t_M^2(n))$$
- **machines non déterministes** :
  Soit M une machine non déterministe, et M' une machine déterministe qui la simule en essayant successivement tous ses calculs à l'aide d'un arbre. Dans l'arbre des calculs, on ne se préoccupe pas du problème de l'arrêt, par hypothèse résolu. On peut donc se contenter de parcourir l'arbre des calculs en profondeur d'abord.
  Le nombre de calculs possibles pour M', pour une entrée de taille n est borné par $k^{t_M(n)}$, où k est le nombre maximal de transitions qui peuvent être effectuées à partir d'une configuration quelconque : k est le cardinal maximal des $\delta(p, a) = \{(q, b, x) \mid p, a \to q, b, x \in E\}$ pour tous p et a. On en tire, si $t(n) \geq n$:
  $$t_{M'}(n) = \mathbf{O}(t_M(n)k^{t_M(n)}) = 2^{\mathbf{O}(t_M(n))}$$
  Remarquons le changement radical de complexité induit par cette transformation.

4.2.3 Classes de complexité en temps

Définition 4.2.1 (Classe).
Soit f une fonction $\mathbb{N} \to \mathbb{R}_+$. Par définition :
- $\mathsf{TIME}(f(n))$ est l'ensemble des problèmes décidés par une machine de Turing déterministe en temps $\mathbf{O}(f(n))$.
- $\mathsf{NTIME}(f(n))$ est l'ensemble des problèmes décidés par une machine de Turing non déterministe en temps $\mathbf{O}(f(n))$

On peut alors définir :

$$\mathsf{P} = \bigcup_{k \geq 0} \mathsf{TIME}(n^k)$$

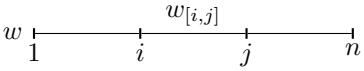
$$\mathsf{NP} = \bigcup_{k \geq 0} \mathsf{NTIME}(n^k)$$

$$\mathsf{EXPTIME} = \bigcup_{k \geq 0} \mathsf{TIME}(2^{n^k})$$

Proposition 4.2.1.
On a les inclusions triviales : $\boxed{\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXPTIME}}$

Exemple (Problèmes dans P*).*

1. Problème d'accessibilité : *un sommet t d'un graphe G donné est-il accessible à partir du sommet s ?*
   La résolution se fait en parcourant le graphe en largeur ; cela se fait en temps linéaire. Voir [Sip97], p. 237.[1]
2. Problèmes décrits par des langages algébriques :
   étant donné L, un langage algébrique, est-ce que $L \in \mathsf{P}$? :



$$G = (A, V, P)$$

$$S \in V, w \in L_G(S)$$

$$\begin{array}{rcl} w[i,j] & \in & L_G(S) \\ S & \to & S_1 S_2 \\ w[i,k] \in L_G(S_1) & et & w[k+1,j] \in L_G(S_2) \end{array}$$

Exemple (Problèmes dans NP*).*

[1] On remarquera que ce même problème est dans NL (c'est à dire $\mathsf{NSPACE}(log(n))$, défini un peu plus loin) et même NL-complet, lorsque le graphe est orienté. Lorsqu'il s'agit d'un graphe non orienté, le problème d'accessibilité est de classe L (*i.e.* $\mathsf{SPACE}(log(n))$). on consultera pour ces résultats [Sip97], p. 295.

1. Chemin hamiltonien (HAM-PATH) ($\neq$ chemin eulérien[2]) :
   Un chemin hamiltonien dans un graphe G orienté est un chemin qui passe une fois et une seule par chaque sommet de G. Le problème du chemin hamiltonien est de savoir si un graphe G donné contient un chemin hamiltonien de s à t pour deux sommets s et t également donnés. Ce problème peut être posé pour un graphe orienté ou pour un graphe non orienté mais ces deux problèmes se ramènent aisément de l'un à l'autre.
2. Satisfiabilité d'une formule – du calcul propositionnel : Une formule est dite satisfiable s'il est possible d'affecter une valeur vrai (noté 1) ou faux (noté 0) à chacune des variables de telle façon que la formule ait la valeur vrai.

Définition 4.2.2 (Vérificateur).
Une machine M déterministe est un vérificateur en temps polynomial pour L si M décide sur des entrées de la forme $\langle w, c\rangle$ en temps polynomial en $|w|$ et est telle que :

$$L = \{w \mid \exists c, \langle w, c\rangle \in L(M)\}$$

Proposition 4.2.2 (Equivalence vérificateur et NP).
$L \in \mathsf{NP}$ *si et seulement s'il existe un vérificateur polynomial pour* L.

Preuve.
Soit une machine de Turing non déterministe, M', qui reconnait L en temps polynomial ; on établit une association [3]

$$V \text{ (vérificateur)} \leftrightarrow M' \text{ (machine non déterministe)}$$

– $\Rightarrow$: V prend en entrée c, une suite de transitions (de taille polynomiale) faites par M' en calculant w. V *simule* alors M sur w pour vérifier $w \in L(M')$. Ceci se fait en suivant c, donc en temps polynomial.
– $\Leftarrow$: M' *choisit* c de façon non déterministe et *simule* le calcul de V sur $\langle w, c\rangle$. V est en temps polynomial, donc c polynomial et le calcul de M' est polynomial.

□

4.2.4 Réduction polynomiale

Premières définitions

Il s'agit d'introduire un ordre de complexité parmi les problèmes NP.

Définition 4.2.3 (Réduction polynomiale).
Soient A et B, des problèmes codés respectivement par L_A et L_B sur les alphabets Σ_A et Σ_B. Une réduction polynomiale de A à B est une fonction $f : \Sigma_A^* \to \Sigma_B^*$ calculable en temps polynomial par une machine de Turing déterministe telle que :

$$w \in L_A \Leftrightarrow f(w) \in L_B$$

Notation.
L'existence d'une réduction polynomiale se note :

$$A \leq_P B$$

[2] Circuit qui passe une fois exactement par chaque *arête*.
[3] On utilise pour cela l'axiome du choix dénombrable.

Proposition 4.2.3.
Si $A \leq_P B$ et $B \in P$, alors $A \in P$.

NP-complétude

Définition 4.2.4 (NP-difficile).
Un problème, A, est dit NP-difficile si $\forall B \in \mathsf{NP},\ B \leq_P A$

Définition 4.2.5 (NP-complet).
Un problème, A, est dit NP-complet si

1. $A \in \mathsf{NP}$;
2. A est NP-difficile.

NP-complétude de SAT et 3-SAT

NP-Complétude de SAT

Théorème 4.2.2 (Cook & Levin).
Les problèmes SAT *et 3-*SAT *sont* NP*-complets.*

Où SAT est le problème de la satisfiabilité d'une formule, 3-SAT celui de la satisfiabilité d'une formule en forme normale conjonctive où chaque clause est la conjonction de trois littéraux.

Preuve de Cook & Levin.
Preuve que SAT est NP-complet : Soit $A \in \mathsf{NP}$, M machine non déterministe en temps n^k qui décide A. On veut montrer que, pour toute entrée w, Φ_w est satisfiable si et seulement si w est acceptée par M.

On note $n = |w|$ la taille de l'entrée w. M fonctionne en temps polynomial donc $\exists k$ tel que tout calcul sur w soit de longueur $\leq n^k$. Quitte à modifier légèrement M, on suppose que tout calcul acceptant sur w est de longueur exactement n^k.

Comme M fonctionne en temps n^k, elle utilise au plus n^k cellules. Les configurations sont de longueur au plus n^k, qu'on supposera égale à n^k quitte à modifier l'écriture des configurations.

Ecrivons ces configurations les unes sous les autres pour obtenir le tableau de symboles éléments de l'alphabet $A = \Gamma \cup Q$, à la figure 4.1.

Conf.	0	1	2	3	...	n^k
$C_0 =$	q_0	w_1	w_2	w_3	...	#
$C_1 =$	w'_1	q_1	w_2	w_3	...	#
$C_2 =$	w'_1	w'_2	q_2	w_3	...	#
$C_3 =$	...	...	...	...	...	#
...	...	...	...	...	...	...
C_{n^k}	...	...	...	...	...	...

TAB. 4.1 – Tableau formé par les configurations

On cherche à écrire Φ_w qui code l'existence d'un tel tableau formé par les configurations successives d'un calcul acceptant sur w.

Pour chaque $(i,j) \mid 0 \leq i,j \leq n^k$, et pour chaque $z \in \Gamma \cup Q$, soit $x_{(i,j,z)}$ qui code le fait que la case (i,j) contienne ou non le symbole z [4].

Notons Φ_{cell} la formule codant le fait que chaque cellule contient un unique symbole de A, Φ_{start} celle codant que la première ligne du tableau est bien $q_0 w$, Φ_{move} codant le fait que chaque ligne est obtenue en appliquant une transition de M, Φ_{accept} codant que le calcul est bien acceptant.

La formule φ_{move} dit que chaque C_i suit C_{i-1} par la transition apparaissant dans le symbole à une case (i,j) donnée. Or, le contenu d'une case (i,j) dépend uniquement de la case au-dessus $(i-1,j)$ et des deux cases qui y sont adjacentes, par fonctionnement de la machine.

On peut donc définir un prédicat $f(W,X,Y,Z)$ qui est vrai si et seulement si le symbole Z peut apparaître en position i,j d'une configuration donnée, sachant que les symboles W, X, Y sont les symboles aux positions $(i-1,j-1)$, $(i-1,j)$ et $(i-1,j+1)$ [5].

$$
\begin{aligned}
\Phi_w &= \Phi_{\text{cell}} \wedge \Phi_{\text{start}} \wedge \Phi_{\text{accept}} \wedge \Phi_{\text{move}} \\
\Phi_{\text{cell}} &= \bigwedge_{0 \leq i,j \leq n^k} \left[\left(\bigvee_{z \in \Gamma \cup Q} x_{i,j,z} \right) \wedge \left(\bigwedge_{\substack{z,z' \in \Gamma \cup Q \\ z \neq z'}} (\neg x_{i,j,z} \vee \neg x_{i,j,z'}) \right) \right] \\
\Phi_{start} &= x_{0,0,q_0} \wedge x_{0,1,w_1} \wedge x_{0,2,w_2} \ldots \wedge x_{0,n,w_n} \wedge x_{0,n+1,\#} \ldots \wedge x_{0,n^k,\#} \\
\Phi_{accept} &= \bigvee_{q \in F} \left(\bigvee_{0 \leq j \leq n^k} x_{n^k,j,q} \right) \\
\Phi_{\text{move}} &= \bigwedge_{0 \leq i,j \leq n^k} \left(\bigvee_{\substack{W,X,Y,Z \\ \mid f(W,X,Y,Z)}} \left(x_{i-1,j-1,W} \wedge x_{i-1,j,Y} \wedge x_{i-1,j+1,X} \wedge x_{i,j,Z} \right) \right)
\end{aligned}
$$

Ces formules étant de taille polynomiale en n, leur conjonction l'est aussi. □

Réduction de SAT à 3-SAT

On montre que SAT se réduit polynômialement à 3-SAT, ce qui suffit à montrer que SAT est NP-complet. Soit une formule Φ quelconque, trouvons Φ' à trois littéraux par clause, de même satisfiabilité. Les étapes du calcul de Φ' sont :

1. Descente des négations sur les variables, en utilisant les lois de De Morgan. Cette descente produit en temps linéaire pour un programme (donc polynomial pour une machine de Turing) une formule où toutes les négations sont devant des variables.
2. $\Phi \rightarrow \Phi\,'$ en forme normale conjonctive[6] : par induction sur Φ
   - si $\Phi = \Phi_1 \wedge \Phi_2$ alors $\Phi' = \Phi'_1 \wedge \Phi'_2$
   - si $\Phi = \Phi_1 \vee \Phi_2$ par induction on calcule :
     $\Phi_1 \rightarrow \Phi'_1 = c_1 \wedge c_2 \wedge \ldots \wedge c_k$

[4] Le nombre de ces variables est $|A|n^{2k}$, polynomial en n.

[5] Si $j = 1$, $W = \#$ et si $j = n^k$, $Y = \#$.

[6] Ceci ne peut se résoudre par distributivité, car la taille de la formule finale pourrait ainsi croître exponentiellement.

$\Phi_2 \rightarrow \Phi'_2 = c'_1 \wedge c'_2 \wedge \ldots \wedge c'_{k'}$
Comme on ne peut pas les réunir par une disjonction, on introduit une nouvelle variable y :
$\Phi' = (y \vee c_1) \wedge \ldots \wedge (y \vee c_k) \wedge (\neg y \vee c'_1) \wedge \ldots \wedge (\neg y \vee c'_{k'})$
Φ' n'est pas logiquement équivalente à Φ mais a même *satisfiabilité*. De plus elle est de taille au plus quadratique en $|\Phi|$, et se calcule en temps quadratique.

3. Puis on met sous forme 3-SAT en remplaçant les clauses n'ayant pas trois littéraux par une ou plusieurs clauses ayant exactement trois littéraux. Si une clause a moins de trois littéraux, on répète la disjonction sur un littéral de la clause pour en ajouter. Dans le cas d'une clause à plus de trois littéraux $(l_1 \vee \ldots \vee l_k)$:
On introduit les variables $y_1 \ldots y_{k-3}$ et on remplace par :
$(l_1 \vee l_2 \vee y_1) \wedge (\neg y_1 \vee l_3 \vee y_2) \wedge \ldots \wedge (\neg y_{k-3} \vee l_{k-1} \vee l_k)$
Toute affectation qui rend vraie la clause initiale se prolonge en une affectation sur $y_1 \ldots y_{k-3}$ qui permet de maintenir l'équivlaence de satisfiabilité. Cette dernière transformation se fait en temps linéaire en la taille de la formule, donc polynomial sur une machine de Turing

3-SAT est donc NP-complet.

Autres problèmes NP-complets

Proposition 4.2.4.
Le problème du chemin hamiltonien est NP*-complet.*

Preuve.
On trouvera la réduction de 3-SAT à HAM-PATH, [7] en annexe B, page 89. Cette réduction est écrite pour le problème du chemin hamiltonien sur un graphe orienté. On trouvera la réduction de HAM-PATH sur graphe orienté à HAM-PATH sur graphe non orienté dans [HU79], page 336. □

Définition 4.2.6 (Clique).
Soit G un graphe. Une clique de taille k de G est un ensemble de k sommets parmi lesquels deux sommets distincts sont toujours reliés par une arrête.

Proposition 4.2.5.
Le problème de savoir si un graphe a une clique de taille k est NP*-complet.*

Preuve de CLIQUE$\in$ NP*.*
Un algorithme non déterministe pour décider ce langage L peut fonctionner de la manière suivante. L'algorithme commence par choisir de façon non déterministe k sommets $v_1, \ldots, v_k$ de G. Ce choix se fait en temps linéaire en la taille du graphe. Ensuite, l'algorithme vérifie que les toutes les arêtes (v_i, v_j) pour tout $1 \leq i < j \leq k$ sont présentes dans G : il accepte si c'est le cas. Cette vérification peut se faire en temps polynomial puisqu'il y a au plus k^2 arêtes à tester. Cet algorithme décide si le graphe G possède une clique de taille k. En effet, il y a un calcul de l'algorithme pour chaque choix possible de k sommets. Un de ces calculs est acceptant si G contient une clique. Le problème de la clique appartient donc à la classe NP. □

[7] `http://www.liafa.jussieu.fr/~carton/Enseignement/Complexite/MMFAI/Cours/MasterInfo/time.html`

Preuve de la NP*-complétude de* CLIQUE*.*

On trouvera la preuve de la réduction de 3-SAT à CLIQUE, en annexe C, page 91. □

On trouvera la présentation d'autres algorithmes NP-complets ainsi que la réduction de 3-SAT à ceux-ci dans les annexes.

4.3 Complexité en espace

Soit une machine de Turing M et $s_M(w)$ taille maximale d'une configuration d'un calcul sur l'entrée w. On définit similairement à la complexité en temps, et comme précédemment, $s_M(n) = \max_{|w|=n} s_M(w)$.

4.3.1 Changement de modèle

Les changements de configurations en nombre et forme de bandes sont sans grands changements sur la complexité en espace.

Théorème 4.3.1 (Savitch).
Soit s une fonction de $\mathbb{N}$ dans $\mathbb{R}^+$ telle que $s(n) \geq n$. [8] *Une machine* non déterministe *qui fonctionne en espace $s(n)$ est équivalente à une machine de Turing* déterministe *en espace $\mathbf{O}(s^2(n))$.*

Preuve.

Soit M une machine de Turing en espace $s(n)$. On simule M par une machine déterministe m' décrite par l'algorithme ACCESS ci-dessous : ACCESS(C, C', t) retourne vrai si la machine M peut passer de la configuration C à la configuration C' en au plus t étapes de calcul.

Algorithme (ACCESS(C, C', t)**).**

```
if t ≤ 1 then
  return C==C'
else
  for all C'' de taille ≤ s(|w|) do
    if (ACCESS(C, C'', ⌈t/2⌉) ∧ ACCESS(C'', C', ⌊t/2⌋)) then
      return true
    end if
  end for
  return false
end if
```

L'objectif de l'algorithme global est de déterminer si $w \in L(M)$ en supposant $F = \{q_f\}$. Reste à trouver, en notant $C_0 = q_0 w$ et $C_f = q_f$, si on peut passer de C_0 à C_f : quel t faut-il choisir ?

Si M fonctionne en espace $s(n)$, alors M fonctionne en temps $2^{\mathbf{O}(s(n))}$, voir page 79.

Si w est accepté, il y a un calcul de C_0 à C_f en $t = 2^{Ks(n)}$.

[8] A priori, on pourrait penser que cette configuration est toujours vérifiée, dans la mesure où l'entrée de taille n est toujours dans la configuration. On verra dans les extensions qu'il est en fait possible de définir une machine de Turing de complexité sous-linéaire en espace avec des configurations un peu différentes.

$$w \in L(M) \Leftrightarrow \textsc{Access}(C_0, C_f, 2^{Ks(n)})$$

Reste à évaluer la complexité en espace de Access :
- taille de la pile des appels récurisfs : $Ks(n)$
- espace d'un appel : C,C',C" (configurations, espace $\leq s(n)$) t (en binaire $\leq Ks(n)$)

Au final, la machine utilise en espace $K^2 s^2(n)$. □

4.3.2 Classes de complexité en espace

Notation.
$\mathsf{SPACE}(f(n))$ l'ensemble des problèmes décidables en espace $f(n)$ par une machine déterministe.
$\mathsf{NSPACE}(f(n))$ l'ensemble des problèmes décidables en espace $f(n)$ par une machine non déterministe.
Le théorème de Savitch garantit qu'il n'est pas utile de distinguer les machines déterministes et non déterministes, d'où l'égalité :

$$\mathsf{PSPACE} = \bigcup_{k\geq 0} \mathsf{SPACE}(n^k) = \mathsf{NPSPACE} = \bigcup_{k\geq 0} \mathsf{NSPACE}(n^k)$$

4.3.3 Relations entre les complexités en temps et en espace

Lemme 4.3.2.
Si M fonctionne en temps $t(n)$ alors M fonctionne en espace au plus $t(n)$

Proposition 4.3.1.

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME}$$

Preuve.
$\mathsf{P} \subseteq \mathsf{PSPACE}$ et $\mathsf{NP} \subseteq \mathsf{NPSPACE}$ prouvés par le lemme.

Si une machine de Turing s'arrête toujours, un calcul de cette machine ne peut pas repasser deux fois par la même configuration. En effet, le nombre de configurations d'une telle machine est $|Q|s(n)k^{s(n)} = 2^{\mathbf{O}(s(n))}$. Ceci montre qu'une machine qui fonctionne en espace $s(n)$ fonctionne en temps $2^{\mathbf{O}(s(n))}$. On a donc l'inclusion $\mathsf{PSPACE} \subseteq \mathsf{EXPTIME}$. [9] □

4.3.4 Exemples de problèmes dans PSPACE

Exemple (Problèmes dans PSPACE*).*

1. *Universalité* d'un automate fini :
   Soit $\mathcal{A}$ un automate sur l'alphabet A. Le problème de l'universalité est de savoir si :
   $$L(\mathcal{A}) = A^*$$

[9] On sait que $\mathsf{NP} \subset \mathsf{EXPTIME}$, les inégalités strictes entre les autres ensembles restant à ce jour indéterminées.

(a) $\mathcal{A}$ déterministe et émondé
Dans ce cas, $L(\mathcal{A}) = A^*$ si et seulement si $\mathcal{A}$ complet et tous les états sont finaux.

(b) $\mathcal{A}$ non déterministe :
L'automate $\mathcal{A}$ est équivalent à un automate déterministe ayant au plus 2^n états. Si $L(\mathcal{A}) \neq A^*$, il existe un mot w, tel que $|w| \leq 2^n$ qui n'est pas accepté par $\mathcal{A}$.
Pour que l'automate accepte tous les mots, il faut que tous leurs états d'arrivée soient finaux. Pour étudier ceci, on décompose ces calculs en transitions sur les lettres, arrivant de manière non déterministe sur des groupes d'états.
- L'algorithme simule le calcul de $\mathcal{A}$ en marquant les états accessibles en lisant le mot w. À chaque étape de calcul, l'algorithme choisit une lettre a de A et calcule les états accessibles en effectuant des transitions.
- on s'arrête après 2^n étapes.
- on s'arrête lorsqu'on arrive sur un groupe d'état comprenant un état non final.

L'algorithme utilise un espace linéaire pour stocker le nombre d'étapes effectuées.[10]

2. *Formules booléennes quantifiées* (QSAT) On considère les formules booléennes avec des quantificateurs existentiels et universels portant sur exactement toutes leurs variables. Les seuls opérateurs autorisés sont $\wedge, \vee, \neg$. Par exemple :

$$\forall x \exists y \forall z (1 \wedge y) \vee (y \wedge \neg z) \vee (y \wedge \neg 1)$$

Une formule quantifiée close[11] est-elle vraie ?

Proposition 4.3.2.
QSAT *est dans* PSPACE.

Preuve.
On résout QSAT à l'aide de l'algorithme récursif suivant :

Algorithme ($\mathsf{QSAT}(\Phi) \mid \Phi$ *est close).*
if Φ n'a plus de quantificateurs **then**
eval(Φ)
end if
if $\Phi = \exists \psi$ **then**
$\mathsf{QSAT}(\psi[x \leftarrow 0]) \vee \mathsf{QSAT}(\psi[x \leftarrow 1])$
end if
if $\Phi = \forall x \psi$ **then**
$\mathsf{QSAT}(\psi[x \leftarrow 0]) \wedge \mathsf{QSAT}(\psi[x \leftarrow 1])$
end if

L'algorithme correctement programmé utilise une seule copie de Φ, et une pile de taille proportionnelle au nombre de variables. Il fonctionne donc en espace linéaire. □

[10] On remarquera que cela revient à calculer de manière incrémentale l'automate déterminisé – au lieu de le calculer entièrement avant de l'analyser.
[11] Sans variable libre

4.3.5 PSPACE-complétude

Définition 4.3.1 (PSPACE-complet).
Un problème A est dit PSPACE-complet si :

1. A est PSPACE
2. Tout problème B de PSPACEse réduit polynomialement **en temps** à A, i.e.
$$B \leq_P A$$

Théorème 4.3.3 (PSPACE-complétude de QSAT).
QSAT *est* PSPACE*-complet.*

Preuve.
Soit M machine en espace polynomial. On veut ramener l'acceptation de w par M au calcul de la valuation d'une formule quantifiée.

On suppose donc qu'un problème B est décidé par une machine de Turing M en espace polynomial qu'on peut supposer être n^k pour un entier k fixé. Soit w une entrée de M de taille n. L'existence d'un calcul acceptant w est codé de la manière suivante :

Chaque configuration C de la machine M est codée par des variables qui décrivent chacun des symboles de la configuration. Puisque chaque configuration est de taille n^k, le nombre de variables nécessaires est au plus $\mathbf{O}(n^k)$.

Soit C une configuration, traduite en variable au nombre de $\mathbf{O}(n^k)$. $\Phi_{t,(C,C')}$ est vraie si et seulement si $C \Rightarrow^{\leq t} C'$

1. $t \leq 1 : \Phi_{t,(C,C')} = (C = C' \vee C \Rightarrow C')$
2. $t \geq 1 : \Phi_{t,(C,C')} = \big(\exists\ C''$
$\forall\ (D, D')\,[(D = C \wedge D' = C'') \vee (D = C'' \wedge D' = C')] \Rightarrow \Phi_{\lfloor t/2 \rfloor,(D,D')}\big)$

Dans le cas $t \geq 1$, définir $\Phi_{t,(C,C')}$ par une disjonction de même que pour la fonction Access de la preuve de Cook & Levin ne fonctionne pas, car un calcul de M est de longueur allant jusqu'à 2^{Kn^k} pour une constante K fixée. La formule $\Phi_{t,(C,C')}$ «à la diviser pour régner» pour $t = 2^{Kn^k}$ n'est pas de taille polynomiale. □

4.4 Machine alternante

4.4.1 Définitions

Définition 4.4.1 (Machine alternante).
Une machine M alternante est une machine de Turing où l'ensemble Q des états est partitionné en :
– $Q_\vee$ ensemble des états existentiels
– $Q_\wedge$ ensemble des états universels
Par extension, on dira q'une configuration $C = uqv$ est existentielle (resp. universelle) si q est existentiel (resp. universel).

Définition 4.4.2 (Calcul sur une machine alternante).
Un calcul est un arbre étiqueté par des configurations, vérifiant :

- si un noeud x de l'arbre est étiqueté par une configuration existentielle C alors x a un seul fils étiqueté par C' tel que $C \Rightarrow C'$
- si un noeud x de l'arbre est étiqueté par une configuration universelle C alors x a un fils étiqueté C' pour chaque C' telle que $C \Rightarrow C'$

Définition 4.4.3 (Calcul acceptant).
Une branche est acceptante si un état final apparaît sur la branche.
Un calcul est acceptant si
- l'étiquette de la racine est la configuration initiale $q_0 w$.
- toutes les branches sont acceptantes

4.4.2 Algorithme alternant pour QSAT

Algorithme (QSAT_ALTERNANT(Φ)**).**
if Φ sans Q **then**
eval(Φ)
end if
if $\Phi = \exists\psi\Phi$ **then**
QSAT_ALTERNANT($\Phi[\psi \leftarrow 0]$) $\vee$ QSAT_ALTERNANT($\Phi[\psi \leftarrow 1]$)
end if
if $\Phi = \forall\psi\Phi$ **then**
QSAT_ALTERNANT($\Phi[\psi \leftarrow 0]$) $\wedge$ QSAT_ALTERNANT($\Phi[\psi \leftarrow 1]$)
end if

4.4.3 Automates alternants

Un automate alternant est un automate où chaque transition mène d'un état à un ensemble d'états et est étiquetée par une lettre de l'alphabet et par $\vee$ ou $\wedge$. Un calcul dans un automate alternant est un *arbre* dont les nœuds sont les états traversés au cours de la lecture du mot (voir FIG. 4.1, page 83).

Pour chaque nœud interne :
- si la transition empruntée à l'issue de l'état est étiquetée par $\vee$, le nœud a exactement un fils, correspondant à l'un des états pointés par la transition.
- si elle est étiquetée par $\wedge$, le nœud a exactement autant de fils que la transition pointe d'états, et chacun de ces fils correspond naturellement à chacun des états pointés.

Tous les chemins dans l'arbre ont par conséquent la longueur du mot lu : le calcul est acceptant si toutes les feuilles correspondent à des états finaux.

4.4.4 Classes de complexité

Le temps d'un calcul d'une machine de Turing alternante est par définition la profondeur de l'arbre. L'espace du calcul est la taille maximale d'une configuration apparaissant dans le calcul.

Notation.
pour une fonction t de $\mathbb{N}$ dans $\mathbb{R}^+$, on définit :

- ATIME($t(n)$) classe des problèmes décidés par des machines de Turing alternantes en temps $t(n)$

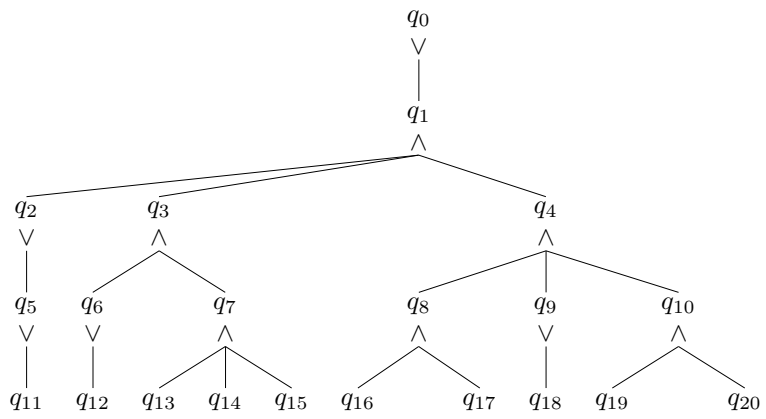


FIG. 4.1 – Un calcul dans un automate alternant

- $\mathsf{AP}(t(n))$classe des problèmes décidés par des machines de Turing alternantes en temps polynomial en $t(n)$
- $\mathsf{ASPACE}(t(n))$classe des problèmes décidés par des machines de Turing alternantes en espace $t(n)$
- $\mathsf{APSPACE}(t(n))$ classe des problèmes décidés par des machines de Turing alternantes en espace polynomial en $t(n)$

Théorème 4.4.1.
Supposant $t(n) \geq n$ *:*

$$\mathsf{ATIME}(t(n)) \subseteq \mathsf{SPACE}(t(n)) \subseteq \mathsf{ATIME}(t^2(n))$$

$$\mathsf{AP} = \mathsf{PSPACE}$$

Supposant $t(n) \geq \log(n)$ *:*

$$\mathsf{ASPACE}(t(n)) \subseteq \mathsf{TIME}(2^{\mathbf{O}(t(n))})$$

$$\mathsf{ASPACE} = \mathsf{EXPTIME}$$

Preuve.

Machine qui vérifie, dans le tableau de la preuve de Cook & Levin, pour chaque paire de lignes et pour chaque paire de cases, qu'une transition a bien été effectuée. Elle ne construit pas le tableau entièrement, mais seulement aux endroits indexés par ses pointeurs, qui sont de taille $t(n)$. On se reportera pour la preuve complète à [Sip97]. □

Annexe A

Normalisation de machines de Turing

A.1 Résultat

Le but de cette partie est de montrer qu'on peut toujours supposer qu'une machine de Turing possède deux états spéciaux q_+ et q_- tels que :

- q_+ est final ($q_+ \in F$) et q_- n'est pas final ($q_- \notin F$),
- la machine se bloque dès qu'elle est dans un des états q_+ ou q_-,
- la machine se bloque uniquement dans un des états q_+ ou q_-.

Pour montrer ce résultat, nous allons montrer qu'à partir d'une machine M, on peut toujours construire une autre machine de Turing M' qui accepte les mêmes entrées que M et qui vérifie les propriétés ci-dessus. Nous allons montrer en outre que si la machine M est déterministe alors la machine M' est aussi déterministe.

A.2 Analyse

Une machine de Turing peut se bloquer dans un état q pour deux raisons.

1. La première raison est qu'aucune transition n'est possible. Supposons que machine est dans état p et que le symbole de bande sous la tête est le symbole a. Si la machine n'a pas de transition de la forme $p, a \rightarrow q, b, x \mid x \in \{L, R\}$, alors elle reste bloquée en p.
2. La seconde raison est que certaines transitions sont possibles mais que ces transitions conduisent à un déplacement de la tête de lecture vers la gauche alors que cette tête est justement sur la première position de la bande. Supposons que machine est dans état p, que la tête de lecture est sur la première position de la bande et que le symbole de bande sous cette tête est le symbole a. Si la machine a des transitions de la forme $p, a \rightarrow q, b, L$ mais pas de transitions de la forme $p, a \rightarrow q, b, R$, alors elle reste bloquée en p. En effet, la définition des machines de Turing stipule qu'une machine n'a pas le droit d'effectuer une transition qui déplace la tête de lecture vers la gauche lorsque cette tête de lecture est justement sur la première position de la bande.

A.3 Idées générales de la construction

L'idée générale de la construction de M' est d'ajouter deux nouveaux états q_+ et q_- à M et d'ajouter les transitions pour que la machine M' passe dans un de ce deux états quand la machine M se bloque dans un état p. Le premier type de blocage est facile à détecter dans la machine M. Il suffit de trouver toutes les paires (p, a) telle qu'il n'existe pas de transition $p, a \to q, b, L \mid x \in \{L, R\}$. Le second type de blocage est plus délicat à détecter puisque la machine M' doit savoir quand sa tête de lecture se trouve sur la première position de la bande. Pour contourner cette difficulté, on ajoute de nouveaux symboles de bandes à la machine M'. Pour chaque symbole de bande a de M, la machine M' contient le symbole a et un autre symbole correspondant $\underline{a}$. L'alphabet de bande M' est donc $\Gamma' = \Gamma \cup \{\underline{a} \mid a \in \Gamma\}$. Ces nouveaux symboles seront uniquement utilisés dans la première position de la bande et serviront à la machine pour détecter cette position. L'alphabet d'entrée de M' est identique à celui de M. La première opération de la machine M' est de remplacer le premier symbole a de l'entrée par le symbole $\underline{a}$ correspondant pour marquer la première position de la bande. À cette fin, on ajoute deux nouveaux états q'_0 et q'_1 et quelques transitions. Ensuite toutes les autres transitions de M' remplacent un caractère non souligné a par un caractère non souligné b et un caractère souligné $\underline{a}$ par un caractère souligné $\underline{b}$. Ainsi, la première position de la bande contient toujours un symbole souligné et toutes toutes les autres positions contiennent des symboles non soulignés.

A.4 La construction proprement dite

Soit $M = (Q, \Sigma, \Gamma, E, q_0, F, \#)$ une machine de Turing. Nous décrivons la machine de M' qui accepte les même entrées mais qui se bloque uniquement dans un des deux états q_+ ou q_-. Par rapport à la machine M, la machine M' possède quatre nouveaux états qui sont les états q'_0, q'_1, q_+ et q_-.

- États : $Q' = Q \cup \{q'_0, q'_1, q_+, q_-\}$
- État initial : q'_0
- États finaux : $F' = \{q_+\}$
- Alphabet de bande : $\Gamma' = \Gamma \cup \{\underline{a} \mid a \in \Gamma\}$
- Alphabet d'entrée : $\Sigma' = \Sigma$
- Caractère blanc : $\#$

Il reste à décrire l'ensemble E' des transitions de M'. Cet ensemble est décomposé en $E' = E_0 \cup E_1 \cup E_2 \cup E_3 \cup E_4$ suivant l'état p dans lequel se trouve la machine. Les ensembles E_0, E_1, E_2, E_3 et E_4 sont définis ci-dessous.

$p = q'_0$

La première transition est chargée de remplacer le premier caractère de l'entrée par le caractère souligné correspondant.

$$E_0 = \{q'_0, a \to q'_1, \underline{a}, R \mid a \in \Gamma\}$$

.

$p = q'_1$

La seconde transition est chargée de remettre la tête de lecture sur la première position de la bande et de passer dans l'ancien état final q_0 pour commencer le calcul.

$$E_1 = \{q'1, a \to q_0, a, L \mid a \in \Gamma\}$$

$p = q_+ \vee p = q_-$

Pour que la machine M' se bloque en q_+ et en q_- elle ne possède aucune transition de la forme $q_+, a \to q, b, x$ et aucune transition de la forme $q_-, a \to q, b, x$ pour $x \in \{L, R\}$.

$p \in Q$

On commence par ajouter à M' toutes les transitions de M pour les lettres non soulignées et les lettres soulignées. On pose :

$$E_2 = \{p, a \to q, b, x \mid p, a \to q, b, x \in E\}$$
$$\cup\{p, \underline{a} \to q, \underline{b}, R \mid p, a \to q, b, R \in E\}$$

Ensuite, on ajoute des transitions spécifiques pour éviter que M' ne se bloque dans des états autres que q_+ et q_-. On pose U l'ensemble des paires (p, a) telles que M ne possède pas de transitions $p, a \to q, b, x \mid x \in \{L, R\}$ et V l'ensemble des paires (p, a) telles que M ne possède pas de transitions $p, a \to q, b, R$. On définit alors les deux ensembles E_3 et E_4.

$$E_3 = \{p, a \to q_+a, R \mid \text{ si } (p, a) \in U \text{ et } p \in F\}$$
$$\cup\{p, a \to q_-a, R \mid \text{ si } (p, a) \in U \text{ et } p \notin F\}$$

$$E_4 = \{p, \underline{a} \to q_+\underline{a}, R \mid \text{ si } (p, a) \in V \text{ et } p \in F\}$$
$$\cup\{p, \underline{a} \to q_-\underline{a}, R \mid \text{ si } (p, a) \in V \text{ et } p \notin F\}$$

.

Il est facile de vérifier que si la machine M est déterministe, alors la machine M' est également déterministe.

Annexe B

Réduction de 3-SAT à HAM-PATH

Le problème HAM-PATH est dans NP. Un algorithme pour résoudre ce problème commence par choisir de façon nondéterministe une suite $u_1, \dots u_n$ de sommets puis vérifie ensuite qu'il s'agit d'un chemin hamiltonien de s à t.

Pour montrer que le problème HAM-PATH est NP-difficile, on va réduire polynomialement le problème 3-SAT à HAM-PATH. À chaque instance de 3-SAT, on associe une instance de HAM-PATH qui a une solution si et seulement si l'instance de 3-SAT en a une. Soit Φ une instance de 3-SAT, c'est à dire une formule en forme conjonctive telle que chaque clause de Φ contienne trois littéraux. On note k le nombre de clauses de Φ et m le nombre de variables apparaissant dans Φ. Si une variable apparaît positivement et négativement dans la même clause, cette clause est toujours satisfaite. On suppose dans la suite que chaque variable apparaît au plus une fois dans chaque clause.

À cette formule Φ, on associe un graphe orienté ayant $2km+2m+k$ sommets. À chaque clause est associée un seul sommet. À chaque variable est associée une partie du graphe appellée *gadget*. Pour chaque variable, ce graphe possède $2k+2$ sommets et est identique à celui représenté sur la B.1.

th=8cm]gadgham-gadget

FIG. B.1 – *gadget* associé à une variable

Le graphe global est obtenu en mettant bout à bout les gadgets pour obtenir une sorte de chapelet et en ajoutant les sommets des clauses. Le gadget d'une variable est relié au sommet de chaque clause où elle apparaît. Si la variable apparaît positivement dans la $(j-i)^{\text{e}}$clause, il y a une arête du sommet $2j-1$ du gadget vers le sommet de la clause et une arête du sommet de la clause vers le sommet $2j$ du gadget. Si la variable apparaît négativement dans la $(j-i)^{\text{e}}$clause, il y a une arête du sommet $2j$ du gadget vers le sommet de la clause et une arête du sommet de la clause vers le sommet $2j-1$ du gadget.

Le sommet s est le premier sommet du gadget de la première variable et le sommet s est le dernier sommet du gadget de la dernière variable. On vérifie qu'il y a un chemin hamiltonien dans la graphe construit si et seulement si la formule Φ est satisfiable.

La construction est illustrée sur la formule $\Phi = (x_0 \vee x_1 \vee x_2) \wedge (\neg x_0 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$. Les entiers k et m sont égaux à 3 et 4 et on obtient le graphe représenté à la figure B.2.

th=8cm]graph3sat2ham

FIG. B.2 – graphe associé à la formule Φ

Annexe C

Réduction de 3-SAT à CLIQUE

Soit Φ une instance de 3-SAT, c'est-à-dire une formule en forme conjonctive telle que chaque clause de Φ contienne trois littéraux.

$$\Phi = (l_1 \vee l_2 \vee l_3) \wedge \ldots (l_{3k-2} \vee l_{3k-1} \vee l_{3k})$$

On introduit alors le graphe non orienté G dont l'ensemble des sommets est l'ensemble $V = \{l_1, \ldots, l_3k\}$ de tous les littéraux de Φ. Deux sommets de G sont reliés par une arête s'ils n'appartiennent pas à la même clause et s'ils ne sont pas contradictoires. Par non contradictoire, on entend que l'un n'est pas égal à la négation de l'autre. L'ensemble E des arêtes est donc défini de la manière suivante.

$$E = \{(l_i, l_j) \mid \lfloor (i-1)/3 \rfloor \neq \lfloor (j-1)/3 \rfloor \wedge l_i \neq l_j\}$$

En effet, le numéro de la clause d'un littéral l_i est égal à $\lfloor (i-1)/3 \rfloor$ si les clauses sont numérotées à partir de 0.

Pour la formule $\Phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$, on obtient le graphe représenté à la figure C.1.

th=8cm]graph23sat2clique

Fig. C.1 – graphe associé à la formule Φ

Nous allons voir que la formule Φ est satisfiable si et seulement si le graphe G contient une clique de taille k. On remarque que deux littéraux d'une même clause ne sont jamais reliés par une arête. Une clique peut donc contenir au plus un littéral par clause et elle est de taille au plus k.

Supposons d'abord que la formule Φ est satisfiable. Il existe donc une affectation des variables telle que Φ vaille 1. Ceci signifie qu'au moins un littéral par clause vaut la valeur 1. Choisissons un tel littéral dans chacune des clauses pour former un ensemble de k littéraux. Comme tous ces littéraux valent 1, deux d'entre eux ne peuvent pas être contradictoires et ils sont donc reliés par des arêtes. C'est donc une clique de taille k dans G.

Supposons maintenant que G contienne une clique de taille k. Comme les littéraux d'une même clause ne sont pas reliés, cette clique contient un littéral exactement dans chaque clause. Montrons alors qu'il existe une affectation qui rend tous ces littéraux égaux à 1. Chaque littéral de cette clique est égal à x_i ou à $\neg x_i$. Pour que ce littéral vaille 1, on impose la valeur 1 ou 0 à la variable correspondante x_i. Comme tous les littéraux de la clique sont reliés par une arête, ils ne sont pas contradictoires deux à deux. Ceci signifie que deux littéraux quelconques de la clique concernent deux variables distinctes x_i et x_j avec $i \neq j$ ou alors ils concernent la même variable x_i mais ils imposent la même valeur à la variable x_i. On obtient alors une affectation cohérente des variables apparaissant dans la clique. En affectant n'importe quelle valeur à chacune des autres variables, on obtient une affectation qui donne la valeur 1 à la formule Φ.

Annexe D

Couverture de sommets

Une arête (u, v) d'un graphe est dite *adjacente* à un sommet s si s est égal à u ou à v. Une couverture de taille k d'un graphe $G = (V, E)$ est un sous-ensemble C de k sommets tel que toute arête de G est adjacente à au moins un des sommets de C.

Problème (VERTEX-COVER).
Un graphe G donné contient une couverture d'une taille k également donnée ?

Proposition.
Le problème VERTEX-COVER *est* NP*-complet.*

Preuve.

Le problème VERTEX-COVER est dans NP. Un algorithme pour résoudre ce problème commence par choisir de façon non déterministe les k sommets $u_1, \ldots, u_k$ puis vérifie que chaque arête du graphe est bien adjacente à un de ces sommets.

Pour montrer que le problème VERTEX-COVER est NP-difficile, on va réduire polynomialement le problème 3-SATà VERTEX-COVER. À chaque instance de 3-SAT, on associe une instance de VERTEX-COVER qui a une solution si et seulement si l'instance de 3-SAT en a une. Soit Φ une instance de 3-SAT, c'est-à-dire une formule en forme conjonctive telle que chaque clause de Φ contienne trois littéraux. On note k le nombre de clauses de Φ et m le nombre de variables apparaissant dans Φ.

À cette formule Φ, on associe un graphe non orienté ayant $3k + 2m$ sommets. Chaque sommet du graphe est en outre étiqueté par un littéral. À chaque variable x_i correspondent deux sommets étiquetés par les littéraux x_i et $\neg x_i$. Ces deux sommets sont reliés par une arête. Cette partie du graphe est appelée le gadget de la variable x_i. À chaque clause correspondent trois sommets, un étiqueté par chaque littéral de la clause. Ces trois sommets sont reliés entre eux par trois arêtes. On ajoute en outre une arête entre chacun des trois sommets d'une clause et le sommet de la variable qui est étiqueté par le même littéral. Cette partie du graphe est appelée le gadget de la clause.

La construction est illustrée sur la formule $\Phi = (x_0 \vee x_1 \vee x_2) \wedge (\neg x_0 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$. Les entiers k et m sont égaux à 3 et 4 et on obtient le graphe représenté à la figure D.1.

th=8cm]graph33sat2cover

FIG. D.1 – graphe associé à la formule Φ

Nous allons voir que la formule Φ est satisfiable si et seulement si le graphe G contient une couverture de taille $2k + m$. Pour chaque variable x_i, il faut qu'un des deux sommets associés soit dans la couverture pour couvrir l'arête entre ces deux sommets. De même pour chaque clause, il faut que deux des trois sommets associés soient dans la couverture pour couvrir les trois arêtes entre ces sommets. Ceci montre qu'une couverture du graphe doit contenir au moins $2k + m$ sommets.

Supposons d'abord que la formule Φ est satisfiable. Il existe donc une affectation des variables telle que Φ vaille 1. Ceci signifie qu'au moins un littéral par clause vaut la valeur 1. Pour chaque variable x_i, on met dans la couverture le sommet x_i ou le sommet $\neg x_i$ suivant que x_i vaille 1 ou 0 dans l'affectation. Pour chaque clause, on met dans la couverture deux sommets du gadget correspondant en prenant au moins les littéraux qui ont la valeur 0 et d'autres pour compléter. Ces choix construisent une couverture. Toutes les arêtes à l'intérieur des gadgets sont couvertes. Chaque arête entre les gadgets des variables et des clauses, relie une variable au littéral correspondant. Si la variable vaut 1, le sommet dans le gadget de la variable a été choisi et si la variable vaut 0,le sommet dans le gadget de la clause a été choisi. Dans les deux cas, l'arête est couverte.

Supposons maintenant que G possède une couverture de taille $2k + m$. Il est clair que cette couverture a exactement un sommet dans chaque gadget associé à une variable et deux sommets dans chaque gadget associé à une clause. Il est facile de vérifier que le choix des sommets dans les gadgets des variables définit une affectation qui donne la valeur 1 à la formule Φ. □

Annexe E

Problème de la somme

Problème (SUBSET-SUM).

Une suite d'entiers $x_1, \ldots, x_k$ ainsi qu'un entier s sont donnés.

Est-il possible d'extraire une sous-suite de la suite donnée de manière à obtenir une suite dont la somme est égale à s ?

La solution attendue est donc une suite croissante d'entiers $1 \leq i_1 < i_2 < \ldots < i_n \leq k$ telle que :

$$x_{i_1} + x_{i_2} + \ldots + x_{i_n} = s$$

Proposition.

Le problème SUBSET-SUM *est* NP*-complet.*

Preuve.

Le problème SUBSET-SUM est dans NP. Un algorithme pour résoudre ce problème commence par choisir de façon non déterministe les indices $i_1, i_2, \ldots, i_n$ puis vérifie que la somme $x_{i_1} + \ldots + x_{i_n}$ a pour valeur s.

Pour montrer que le problème SUBSET-SUM est NP-difficile, on va réduire polynomialement le problème 3-SAT à SUBSET-SUM. À chaque instance de 3-SAT, on associe une instance de SUBSET-SUM qui a une solution si et seulement si l'instance de 3-SAT en a une. Soit Φ une instance de 3-SAT, c'est-à-dire une formule en forme conjonctive telle que chaque clause de Φ contienne trois littéraux. On note k le nombre de clauses de Φ et m le nombre de variables apparaissant dans Φ. Soient $c_0, \ldots, c_{k-1}$ les k clauses de Φ et soient $x_0, \ldots, x_{m-1}$ les m variables de Φ. Pour une variable x_i de variable on note $p(i)$ l'ensemble des numéros des clauses où x_i apparaît positivement et $n(i)$ l'ensemble des numéros des clauses où xi apparaît négativement.

À cette formule Φ, on associe un ensemble de $2(m + k)$ entiers qui vont s'écrire avec $m + k$ chiffres en base 10. À chaque variable x_i correspond deux entiers n_i et p_i définis de la façon suivante.

$$n_i = 10^{k+i} + \sum_{j \in n(i)} 10^j$$

$$pi = 10^{k+i} + \sum_{j \in p(i)} 10^j$$

Les entiers n_i et p_i s'écrivent en base 10, avec $m+k$ chiffres égaux à 0 ou 1. Pour n_i, le chiffre à la position $k+i$ et les chiffres aux positions de $n(i)$ sont des 1 et tous les autres chiffres sont des 0. Pour p_i, le chiffre à la position $k+i$ et les chiffres aux positions de $p(i)$ sont des 1 et tous les autres chiffres sont des 0.

A chaque clause c_j, on associe deux entiers q_j et q'_j qui sont tous les deux égaux à 10^j. Les entiers q_j et $q'j$ s'écrivent en base 10, avec k chiffres égaux à 0 ou 1. Le chiffre à la position j est un 1 et tous les autres sont des 0.

On définit le nombre s par la formule suivante.

$$s = \sum_{O \leq i < m} (10^{k+i} + 3) \sum_{0 \leq j < k} (10^j)$$

L'entier s s'écrit en base 10 avec des chiffres 1 et 3. Son écriture en base 10 a la forme $1 \ldots 13 \ldots 3$ où le premier bloc comporte m chiffres 1 et le second bloc comporte k chiffres 3.

Nous allons illustrer cette construction sur la formule

$$\Phi = (x_0 \vee x_1 \vee x_2) \wedge (\neg x_0 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$$

. Les entiers k et m sont égaux à 3 et 4. Les entiers $n_0, p_0, n_1, p_1, n_2, p_2, n_3, p_4, q_0, q_1, q_2$ et s sont donnés dans le tableau E.1.

n	x_3	x_2	x_1	x_0	c_2	c_1	c_0	Valeur
$n_0 =$	0	0	0	1	0	0	1	1001
$p_0 =$	0	0	0	1	0	1	0	1010
$n_1 =$	0	0	1	0	1	0	1	10101
$p_1 =$	0	0	1	0	0	0	0	10000
$n_2 =$	0	1	0	0	1	0	1	100101
$p_2 =$	0	1	1	0	0	1	0	110010
$n_3 =$	1	0	0	0	0	1	0	100010
$p_3 =$	1	0	0	0	0	0	1	1000001
$q_0, q'_0 =$	0	0	0	0	0	0	1	1
$q_1, q'_1 =$	0	0	0	0	0	1	0	10
$q_2, q'_2 =$	0	0	0	0	1	0	0	100
$s =$	1	1	1	1	3	3	3	1111333

TAB. E.1 – Les entiers associés à la formule Φ

La preuve que l'instance du problème SUBSET-SUM a une solution si et seulement si la formule Φ est satisfiable découle des remarques suivantes. La première remarque est que pour chaque colonne, il y a au plus cinq entiers qui ont un chiffre 1 dans cette colonne. Ceci signifie que quelque soit le choix des entiers, leur somme se fait sans retenue. La somme est donc calculée colonne par colonne.

Comme le chiffre de s est 1 dans chaque colonne associée à la variable x_i, il est nécessaire d'utiliser exactement un des deux entiers n_i et p_i. Ce-sont en effet les seuls qui ont un chiffre non nul dans cette colonne et il n'est pas possible de prendre les deux. Le fait de prendre de choisir n_i ou p_i correspond à affecter la valeur 0 ou 1 à la variable x_i. Chaque entier n_i ou p_i ajoute 1 dans chaque colonne associée à une clause égale à 1 pour le choix de la valeur de la variable x_i. Comme le chiffre de s est 3 dans chaque colonne associée à clause c_j, il

faut exactement trois entiers qui apportent une contribution dans cette colonne. Deux contributions peuvent être apportées par q_j et q'_j mais une contribution au moins doit être apportée par un entier n_i et p_i. Ceci garantit que toutes les causes sont vraies. Si plusieurs variables rendent vraie la même clause, on adapte la somme dans cette colonne en retirant un ou deux des entiers q_j et q'_j. □

Bibliographie

[ABB97] Jean-Michel Autebert, Jean Berstel, and Luc Boasson. Context-free languages and pushdown automata. In *Handbook of formal languages*, volume 1, pages 111–174. Springer, 1997.

[Aut87] Jean-Michel Autebert. *Langages algébriques*. Masson, 1987.

[BBC93] Gérard Beauquier, Jean Berstel, and Philippe Chrétienne. *Éléments d'algorithmique*. Masson, 1993.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Per90] Dominique Perrin. Finite automata. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 1, pages 1–57. Elsevier, 1990.

[Pin84] Jean-Éric Pin. *Variétés de Langages Formels*. Masson, 1984.

[Sak03] Jacques Sakarovitch. *Éléments de théorie des automates*. Vuibert, 2003.

[Sip97] M. Sipser. *Introduction to the Theory of Computation*. PWS publishing Company, 1997.

[vLW92] J. H. van Lint and R. M. Wilson. *A Course in Combinatorics*. Cambridge University Press, 1992.