

Langages formels, Calculabilité et Complexité

Olivier Carton

Formation prédoctorale
École Normale Supérieure

Version du 12 avril 2007

Rédaction

Une toute première version de ce support de cours a été rédigée pendant l'année 2004/2005 par les élèves Gaëtan Bisson, François Garillot, Thierry Martinez et Sam Zoghaib. Ensuite, il a été complètement repris et très largement étoffé par mes soins. Même si la version actuelle est relativement éloignée de la première version, elle lui doit l'impulsion initiale sans laquelle elle n'existerait peut-être pas. Certains passages ont aussi bénéficié de quelques travaux de rédaction des élèves. Malgré tous ces efforts, ce support de cours contient encore de trop nombreuses erreurs et omissions. Une certaine indulgence est demandée au lecteur. Les corrections et suggestions sont bien sûr les bienvenues.

Objectifs

Ce support de cours reflète les deux objectifs de ce cours. Le premier est d'acquérir les principales notions élémentaires en langages formels, calculabilité et complexité. Le second est de ne pas rester uniquement au niveau des définitions et trivialités et de montrer quelques jolis résultats de ces différents domaines. Ce choix fait que des résultats de difficultés très différentes se côtoient. Le premier chapitre sur les langages rationnels contient par exemple le lemme de l'étoile mais aussi la caractérisation de Schützenberger des langages sans étoile.

Plan

La structure de ce document reprend la division du cours en deux grandes parties : les langages formels d'une part, calculabilité et complexité d'autre part. Chacune de ces deux parties est scindée en deux chapitres. Les deux premiers chapitres sont consacrés aux langages rationnels et aux langages algébriques et les deux suivants à la calculabilité et à la complexité.

Téléchargement

Ce support de cours est disponible à l'URL
<http://www.liafa.jussieu.fr/~carton/Enseignement/Complexite/ENS/Support/>.

Table des matières

I	Langages formels	5
1	Langages rationnels	7
1.1	Premières définitions	7
1.2	Opérations rationnelles	8
1.3	Combinatoire des mots	9
1.3.1	Périodicités	9
1.3.2	Mots infinis	13
1.3.3	Motifs inévitables	14
1.4	Un peu d'ordre	18
1.4.1	Quasi-ordres sur les mots	20
1.4.2	Ordres sur les mots	22
1.4.3	Quasi-ordres sur les arbres	22
1.5	Langages rationnels	25
1.5.1	Expressions rationnelles	25
1.5.2	Automates	26
1.6	Automates déterministes	31
1.7	Automate minimal	34
1.7.1	Quotients	34
1.7.2	Congruence de Nerode	36
1.7.3	Calcul de l'automate minimal	37
1.8	Propriétés de clôture	40
1.8.1	Opérations booléennes	40
1.8.2	Morphisme et morphisme inverse	40
1.8.3	Mélange	40
1.9	Lemme de l'étoile et ses variantes	41
1.10	Hauteur d'étoile	44
1.11	Reconnaissance par morphisme	45
1.12	Langages sans étoile	52
1.13	Compléments	57
1.13.1	Conjecture de Černý	57
1.13.2	Rationnels d'un monoïde quelconque	58
2	Langages algébriques	61
2.1	Grammaires	61
2.1.1	Définitions et exemples	61
2.1.2	Grammaires réduites	64
2.1.3	Grammaires propres	65
2.1.4	Forme normale quadratique	66

2.2	Systèmes d'équations	67
2.2.1	Substitutions	67
2.2.2	Système d'équations associé à une grammaire	68
2.2.3	Existence d'une solution pour $\mathcal{S}(G)$	68
2.2.4	Unicité des solutions propres	69
2.2.5	Théorème de Parikh	70
2.2.6	Systèmes d'équations en commutatifs	71
2.2.7	Solutions rationnelles des systèmes commutatifs	71
2.3	Arbres de dérivation	74
2.3.1	Ambiguïté	74
2.3.2	Lemme d'itération	75
2.3.3	Applications du lemme d'itération	76
2.3.4	Ambiguïté inhérente	78
2.4	Propriétés de clôture	78
2.4.1	Opérations rationnelles	78
2.4.2	Substitution algébrique	79
2.4.3	Morphisme alphabétique inverse	79
2.4.4	Intersection avec un rationnel	80
2.4.5	Théorème de Chomsky-Schützenberger	81
2.5	Forme normale de Greibach	82
2.6	Automates à pile	85
2.6.1	Définitions et exemple	85
2.6.2	Différents modes d'acceptation	86
2.6.3	Équivalence avec les grammaires	87
2.6.4	Automates à pile déterministes	88
2.7	Compléments	89
2.7.1	Réécriture	90
2.7.2	Contenus de pile	92
2.7.3	Groupe libre	93

II Calculabilité et complexité 95

3	Calculabilité 97
3.1	Introduction 97
3.1.1	Notion de problème 97
3.1.2	Notion de codage 98
3.1.3	Machines de Turing 98
3.1.4	Normalisation 103
3.1.5	Variantes 105
3.2	Langages récursivement énumérables 110
3.3	Langages décidables 112
3.4	Problème de correspondance de Post 116
3.4.1	Présentation 116
3.4.2	Indécidabilité 117
3.4.3	Application aux grammaires 119
3.5	Théorème de récursion 121
3.6	Machines linéairement bornées 123
3.6.1	Définition 123
3.6.2	Grammaires contextuelles 123

3.6.3	Décidabilité	125
3.6.4	Complémentation	126
3.7	Décidabilité de théories logiques	126
3.7.1	Modèles et formules logiques	126
3.7.2	Arithmétique de Presburger	127
3.7.3	Addition et multiplication	129
3.7.4	Logique monadique du second ordre	129
3.8	Fonctions récursives	130
3.8.1	Fonctions primitives récursives	130
3.8.2	Fonctions récursives	134
3.8.3	Équivalence avec les machines de Turing	134
3.8.4	Thèse de Church	135
3.9	Compléments	135
3.9.1	Écritures des entiers dans une base	135
3.9.2	Machines de Turing sans écriture sur l'entrée	138
4	Complexité	141
4.1	Introduction	141
4.1.1	Objectifs	141
4.1.2	Définition des complexités	141
4.1.3	Graphes des configurations	142
4.2	Complexité en temps	142
4.2.1	Théorème d'accélération	142
4.2.2	Changements de modèles	143
4.2.3	Classes de complexité en temps	144
4.2.4	NP-complétude	146
4.2.5	NP-complétude de SAT et 3SAT	147
4.2.6	Exemples de problèmes NP-complets	151
4.3	Complexité en espace	158
4.3.1	Changement de modèle	158
4.3.2	Classes de complexité en espace	159
4.3.3	Complexités en temps et en espace	159
4.3.4	Exemples de problèmes dans PSPACE	160
4.3.5	PSPACE-complétude	161
4.3.6	Espace logarithmique	162
4.4	Théorèmes de hiérarchie	163
4.5	Machines alternantes	164
4.5.1	Définitions	164
4.5.2	Algorithme alternant pour QSAT	164
4.5.3	Automates alternants	165
4.5.4	Classes de complexité	165
4.6	Compléments	167
4.6.1	Machines à une bande en temps $o(n \log n)$	167
	Bibliographie	174
	Index	175

Première partie

Langages formels

Chapitre 1

Langages rationnels

Les langages rationnels sont le premier niveau de la hiérarchie de Chomsky. Cette famille est constituée des langages acceptés par les automates finis qui sont le modèle le plus simple de machines. Cette famille de langages jouit de propriétés remarquables. Elle est en particulier closes par de très nombreuses opérations.

La théorie des automates est une théorie relativement riche et certains de ses résultats sont de véritables perles. Elle entretient aussi des liens avec beaucoup d'autres domaines comme la dynamique symbolique, la combinatoire, l'algèbre, la topologie, la théorie des jeux, l'arithmétique, la logique et l'algorithmique. Certains de ces liens avec l'algèbre et la logique sont abordés dans ce support de cours (cf. sections 1.11 et 3.7.2 par exemple).

Les automates se sont aussi imposés comme un outil incontournable d'un point de vue pratique. Tout éditeur de texte un peu évolué comprend une fonction de recherche à partir d'une expression rationnelle. Cette recherche commence toujours par la construction d'un automate équivalent à l'expression. Les automates sont aussi utilisés dans le domaine de la vérification. Ils s'agit dans ce domaine de vérifier qu'un objet tel un composant logiciel ou un protocole satisfait bien une spécification.

Ce chapitre introduit les notions de mots et de langages. Après quelques éléments de combinatoire des mots, il développe les langages rationnels et les automates finis qui les acceptent. Une dernière partie est consacrée à la reconnaissance par morphismes des langages rationnels et à la caractérisation des langages sans étoile. Pour l'essentiel du chapitre, une bonne référence est [Per90].

1.1 Premières définitions

On commence par quelques définitions élémentaires. La terminologie est empruntée à la linguistique et provient du fait qu'un des objectifs initiaux de ce domaine était la modélisation des langues naturelles.

- Définition 1.1.**
- Un *alphabet* est un ensemble fini (souvent noté A ou Σ) dont les éléments sont appelés *lettres* ou *symboles*.
 - Un *mot* w sur l'alphabet A est une suite finie $w_1w_2 \cdots w_n$ de lettres de A . L'entier n est appelé la *longueur* de w qui est notée $|w|$.
 - Le *mot vide* noté ε ou parfois 1 correspond à l'unique mot de longueur 0 .

- Un *langage* sur l’alphabet A est un ensemble de mots sur A .
- L’ensemble de tous les mots sur l’alphabet A est noté A^* et il est appelé le *monoïde libre* sur A .

La *concaténation* des deux mots $u = u_1 \cdots u_m$ et $v = v_1 \cdots v_n$ est le mot noté $u \cdot v$ ou uv et égal à $u_1 \cdots u_m v_1 \cdots v_n$ obtenu par simple juxtaposition. C’est une opération associative dont le mot vide ε est l’élément neutre. C’est pour cette raison que le mot vide est parfois noté 1.

Exemple 1.2. Si $u = abaa$ et $v = bab$, on a $uv = abaabab$ et $vu = bababaa$. La concaténation n’est pas commutative.

Comme la concaténation est une loi de composition interne associative avec un élément neutre, elle munit l’ensemble A^* de tous les mots d’une structure de monoïde. Pour cette raison, la concaténation est aussi appelée *produit*. Ce monoïde est dit libre parce que les seules équations qu’il satisfait sont celles qui sont conséquences de la définition de monoïde. Ce fait est énoncé de manière rigoureuse à la proposition 1.106 (p. 46). La notion de *structure libre* peut être définie de manière très générale dans le cadre de l’algèbre universelle [Alm94].

Un mot u est un *préfixe* (resp. *suffixe*) d’un mot w s’il existe un mot v tel que $w = uv$ (resp. $w = vu$). Un mot u est un *facteur* d’un mot w s’il existe deux mots v et v' tels que $w = vuv'$.

Exercice 1.3. Soit $w = w_1 \cdots w_n$ un mot de longueur n . Par un léger abus, on utilise la notation w_i même si l’entier i n’est pas dans l’intervalle $[1; n]$. On note ainsi la lettre $w_{i'}$ où i' est l’unique entier tel que $1 \leq i' \leq n$ et $i' \equiv i \pmod n$. On appelle *occurrence circulaire* d’un mot $u = u_1 \cdots u_p$ un entier k dans l’intervalle $[1; n]$ tel que $w_{k+i-1} = u_i$ pour chaque $1 \leq i \leq p$.

Montrer que pour tout alphabet A et tout entier k , il existe un mot w_k tel que tous les mots de longueur k sur A ont exactement une occurrence circulaire dans w_k . Un tel mot est appelé un mot de *de Bruijn* d’ordre k . Les mots 1100 et 11101000 sont des mots de de Bruijn d’ordre 2 et 3 sur l’alphabet $\{0, 1\}$.

On pourra considérer l’automate dont l’ensemble des états est A^{k-1} et dont l’ensemble des transitions est $\{au \xrightarrow{a} ub \mid a, b \in A \text{ et } u \in A^{k-2}\}$. Montrer qu’il y a bijection entre les chemins eulériens de cet automate et les mots de de Bruijn d’ordre k .

Soit A et B deux alphabets. Un morphisme de monoïdes μ de A^* dans B^* est une application de A^* dans B^* qui est compatible avec la structure de monoïde. Ceci signifie que l’image du mot vide est le mot vide ($\mu(\varepsilon) = \varepsilon$) et que $\mu(uv) = \mu(u)\mu(v)$ pour tout mot u et v de A^* . L’image d’un mot $w = w_1 \cdots w_n$ est donc égal à $\mu(w) = \mu(w_1) \cdots \mu(w_n)$. Le morphisme est complètement déterminé par les images des lettres. Un morphisme μ est dit *effaçant* s’il existe une lettre a tel que $\mu(a) = \varepsilon$.

Exemple 1.4. Soit A l’alphabet $\{a, b\}$ et soit $\mu : A^* \rightarrow A^*$ le morphisme de monoïde déterminé par $\mu(a) = ab$ et $\mu(b) = a$. Ce morphisme est non-effaçant. L’image du mot aba par μ est le mot $abaab = ab \cdot a \cdot ab$.

1.2 Opérations rationnelles

On définit maintenant les opérations rationnelles qui permettent de construire de nouveaux langages.

Définition 1.5 (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 1.6. Soient les deux langages $L = \{u \in A^* \mid |u| \text{ pair}\}$ et $L' = \{u \in A^* \mid |u| \text{ impair}\}$. On a alors les égalités suivantes.

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

Définition 1.7 (Étoile). Soit $L \subseteq A^*$, on définit :

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

Cette définition justifie donc *a posteriori* la notation A^* puisque l'ensemble de tous les mots est bien l'étoile de l'alphabet. Il faut remarquer qu'en général L^i est différent de $\{u^i \mid u \in L \text{ et } i \geq 0\}$.

Un *sous-monoïde* de A^* est un langage de A^* qui contient le mot vide et qui est clos pour la concaténation. On vérifie que pour tout langage L , le langage L^* est le plus petit sous-monoïde de A^* qui contient L . Inversement tout sous-monoïde K de A^* est de la forme L^* puisque $K = K^*$.

Exemple 1.8. – Si $L = \{a, ba\}$, alors L^* est constitué de tous les mots dans lesquels chaque b est suivi d'un a .

- Si $L = \{a^n b \mid n \geq 0\}$, alors L^* est constitué du mot vide ε et de tous les mots qui se terminent par un b .
- Si $L = \emptyset$, alors L^* est le langage $\{\varepsilon\}$ uniquement constitué du mot vide.

Pour un langage L , on note L^+ le langage $LL^* = L^*L = \bigcup_{i \geq 1} L^i$. Si L contient le mot vide, on a $L^+ = L^*$ et sinon on a $L^+ = L^* \setminus \{\varepsilon\}$. Le langage A^+ est par exemple l'ensemble des mots non vides.

1.3 Combinatoire des mots

La combinatoire des mots est l'étude des propriétés structurelles des mots. Dans cette partie, on s'intéresse d'abord à des questions de périodicité puis à des questions de motifs inévitables.

1.3.1 Périodicités

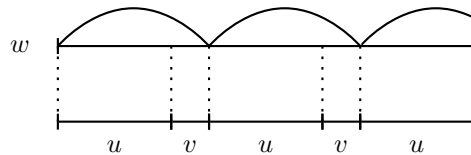


FIG. 1.1 – Période d'un mot w

Soit $w = w_1 \cdots w_n$ un mot sur un alphabet A . Une *période* de w est un entier p entre 1 et $|w|$ tel que $w_i = w_{i+p}$ pour tout $1 \leq i \leq |w| - p$. Par

convention, la longueur de w est une période de w . On note $P(w)$ l'ensemble des périodes de w . Il faut remarquer qu'une période ne divise pas nécessairement pas la longueur du mot. Si p est une période de w , le mot w se factorise $w = (uv)^k u$ où l'entier k vérifie $k \geq 1$, l'entier p est égal à $|uv|$ et le mot v est non vide (cf. figure 1.1).

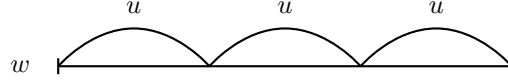


FIG. 1.2 – Mot non primitif $w = u^3$

Un mot w est *primitif* s'il ne s'écrit pas $w = u^k$ avec $k \geq 2$. Autrement dit, un mot est primitif si sa longueur est l'unique période qui divise sa longueur (cf. figure 1.2). Tout mot w s'écrit de manière unique $w = u^k$ où l'entier k vérifie $k \geq 1$ et le mot u est primitif. L'entier k est égal à 1 quand w est primitif.

Exemple 1.9. Le mot $w = 010101$ a $P(w) = \{2, 4, 6\}$ pour ensemble de périodes. Il n'est pas primitif car il s'écrit $w = (01)^3$. Le mot $w' = 01001010010$ a $P(w') = \{5, 8, 10, 11\}$ pour ensemble de périodes. Il est primitif car aucune de ses périodes autre que sa longueur ne divise sa longueur.

Le plus grand commun diviseur de deux entiers p et q est noté $p \wedge q$. Le théorème de Fine et Wilf est le résultat de base de la combinatoire des mots. Les preuves de beaucoup d'autres résultats font appel à ce théorème.

Théorème 1.10 (Fine-Wilf 1965). *Si p et q sont deux périodes d'un mot w de longueur supérieure ou égale à $p + q - p \wedge q$, alors $p \wedge q$ est aussi une période de w .*

Avant de donner la preuve du théorème, on montre que la borne du théorème est optimale. Soit $(f_n)_{n \geq 0}$ la suite de mots définie par récurrence par $f_0 = 1$, $f_1 = 0$ et $f_{n+2} = f_{n+1}f_n$. Les premiers mots de cette suite sont $f_2 = 01$, $f_3 = 010$, $f_4 = 01001$ et $f_5 = 01001010$. Les mots de cette suite sont appelés *mots de Fibonacci*. En effet, la longueur de chaque mot f_n est le nombre de Fibonacci F_n où la suite $(F_n)_{n \geq 0}$ est définie par $F_0 = F_1 = 1$ et la relation de récurrence $F_{n+2} = F_{n+1} + F_n$ pour $n \geq 0$.

Les mots de Fibonacci possèdent beaucoup de propriétés remarquables. Ils constituent en particulier des cas extrémaux pour le théorème de Fine et Wilf. Soit, pour $n \geq 2$, le mot g_n obtenu en supprimant de f_n ses deux dernières lettres qui sont 01 si n est pair et 10 sinon. Les premiers mots de cette suite sont donc $g_2 = \varepsilon$, $g_3 = 0$, $g_4 = 010$ et $g_5 = 010010$. On va montrer que le mot g_n admet F_{n-1} et F_{n-2} pour périodes mais pas $F_{n-1} \wedge F_{n-2} = 1$ alors que sa longueur $|g_n|$ est égale à $F_n - 2 = F_{n-1} + F_{n-2} - (F_{n-1} \wedge F_{n-2}) - 1$. On vérifie facilement que deux nombres des Fibonacci consécutifs F_n et F_{n+1} sont premiers entre eux : $F_n \wedge F_{n+1} = 1$.

On montre par récurrence que $g_{n+2} = f_{n+1}g_n = f_n g_{n+1}$ pour tout $n \geq 2$. Les premières valeurs $g_2 = \varepsilon$, $g_3 = 0$ et $g_4 = 010$ permettent de vérifier ces égalités pour $n = 2$. La formule $f_{n+2} = f_{n+1}f_n$ et la définition de g_n donnent immédiatement l'égalité $g_{n+2} = f_{n+1}g_n$. On a ensuite $g_{n+2} = f_n f_{n-1} g_n = f_n g_{n+1}$ par hypothèse de récurrence. Les relations $g_{n+2} = f_n f_{n-1} g_n = f_n g_{n+1}$ montrent que g_{n+2} admet $F_n = |f_n|$ et $F_{n+1} = |f_n f_{n-1}|$ pour périodes. Par

contre, g_n n'admet pas $F_n \wedge F_{n+2} = 1$ pour période si $n \geq 4$ car il commence par le préfixe 01.

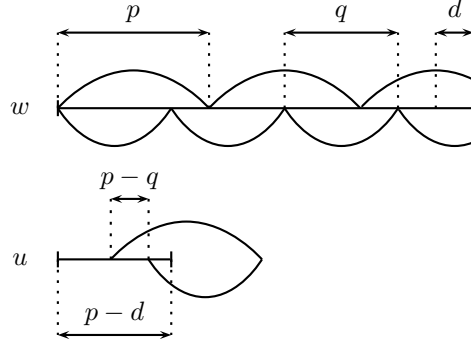


FIG. 1.3 – Preuve du théorème de Fine et Wilf

Une preuve un peu différente est donnée en [Lot83, p. 9].

Preuve. Soit $d = p \wedge q$ le plus grand commun diviseur de p et q . On fixe d et on prouve le résultat par récurrence sur la valeur de $p+q$. Si $p+q \leq d$, c'est-à-dire $p = d$ et $q = 0$ ou $p = 0$ et $q = d$, le résultat est trivial et on suppose qu'il est vrai pour les entiers inférieurs à $p+q$. Soit w un mot tel que $|w| \geq p+q-d$ et ayant p et q comme périodes. Par symétrie, on peut supposer que $p \geq q$. Le mot w est alors factorisé $w = uv$ où u est de longueur $p-d$ (cf. figure 1.3). Pour tout $1 \leq i \leq q-d$, on a $u_i = w_i = w_{i+p} = w_{i+p-q} = u_{i+p-q}$ et u a donc $p-q$ comme période. Comme u a aussi q comme période, l'hypothèse de récurrence implique que u a $d = (p-q) \wedge q$ comme période. Comme $|u| \geq q$, le préfixe de w de longueur q a aussi d comme période. Comme w a q comme période et que d divise q , alors w a aussi d comme période. $\square$

On va utiliser le théorème de Fine et Wilf pour prouver le théorème de Guibas et Odlyzko qui est un résultat assez surprenant. Il montre en effet que tout mot a les mêmes périodes qu'un mot sur un alphabet binaire. Ce théorème est contraire à l'intuition que plus de lettres dans l'alphabet donne plus de liberté pour choisir les périodes. Il est la première étape dans la caractérisation des ensemble d'entiers qui peuvent apparaître comme ensemble de période d'un mot.

Théorème 1.11 (Guibas-Odlyzko 1981). *Pour tout mot w , il existe un mot w' sur un alphabet binaire tel que $P(w) = P(w')$.*

Puisque l'ensemble des périodes d'un mot comprend sa longueur, le mot w' fourni par le théorème précédent est toujours de même longueur que le mot w .

La preuve du théorème est décomposée en plusieurs lemmes. Le premier est un corollaire immédiat du théorème de Fine et Wilf. Il donne une première idée de la structure de l'ensemble des périodes d'un mot. Il dit essentiellement que seules les grandes périodes sont irrégulières puisque les autres sont les multiples de la plus petite période. La plus petite période d'un mot w est toujours définie puisque l'ensemble $P(w)$ des périodes contient au moins la longueur de w .

Lemme 1.12. *Soit w un mot et p sa plus petite période. Si q est une période de w telle que $q \leq |w| - p$, alors q est un multiple de p .*

Preuve. Puisque $p + q \leq |w|$, l'entier $p \wedge q$ est aussi une période de w par le théorème de Fine et Wilf (théorème 1.10). Par définition de p , on a $d = p$ et p divise q . $\square$

Le second lemme permet de prolonger n'importe quel mot en un mot primitif. Sa preuve utilise encore le théorème de Fine et Wilf.

Lemme 1.13. *Pour tout mot w sur $\{0, 1\}$, $w0$ ou $w1$ est primitif.*

Preuve. Si $w = \varepsilon$, alors $w0$ et $w1$ sont primitifs. Supposons par l'absurde que w est non vide et que les mots $w0$ et $w1$ s'écrivent $w0 = u^k$ et $w1 = v^l$ pour des entiers $k, l \geq 2$ et des mots primitifs u et v . Les longueurs $|u|$ et $|v|$ sont des périodes de w et comme $|w| = k|u| - 1 = l|v| - 1 \geq 2 \max(|u|, |v|) - 1 \geq |u| + |v| - 1$, l'entier $d = |u| \wedge |v|$ est aussi une période de w par le théorème de Fine et Wilf (théorème 1.10). Comme d divise $|u|$ et $|v|$ et que les mots u et v sont primitifs, on a $|u| = d = |v|$. Comme u et v sont deux préfixes de w , on doit avoir $u = v$ et ceci contredit le fait que les mots u et v se terminent respectivement avec les lettres 0 et 1. $\square$

On a vu que si p est une période du mot w , celui-ci se factorise $w = (uv)^k u$ où $|uv| = p$ et v est non vide. On applique ce résultat à la plus petite période du mot w . Dans les trois lemmes qui suivent, le mot w est factorisé de la manière suivante.

$$w = (uv)^k u \quad \text{où } p = |uv| = \min P(w), v \neq \varepsilon \text{ et } k \geq 1. \quad (1.1)$$

La preuve du théorème de Guibas et Odlyzko se fait par récurrence sur la longueur du mot w . Elle utilise la factorisation ci-dessus pour se ramener au mot uv qui est plus petit sauf si $|w|$ est l'unique période de w (ce dernier cas est trivial). Les deux premiers lemmes traitent le cas $k \geq 2$ alors que le dernier traite le cas $k = 1$.

Lemme 1.14. *Soit w décomposé comme en (1.1) avec $k \geq 2$ et soit q tel que $|w| - p < q < |w|$. Soit $q = (k - 1)p + r$ où $|u| < r < |u| + p$. Alors q est une période de w si et seulement si r est une période de uvu .*

Preuve. Pour tout $0 < i < |w| - q = p + |u| - r$, on a $w_i = (uvu)_i$ et $w_{i+q} = (uvu)_{i+r}$. On a donc $w_i = w_{i+q}$ si et seulement si $(uvu)_i = (uvu)_{i+r}$ ce qui signifie exactement que $q \in P(w)$ si et seulement si $r \in P(uvu)$. $\square$

Lemme 1.15. *Soit w décomposé comme en (1.1) avec $k \geq 1$. Si $|uv| = |u'v'|$ et $P(u'v'u') = P(uvu)$, alors $P(w) = P(w')$ où $w' = (u'v')^k u'$.*

Preuve. Le cas $k = 1$ est trivial et on suppose dorénavant que $k \geq 2$. L'égalité $P(u'v'u') = P(uvu)$ implique $|uvu| = |u'v'u'|$ et donc $|w| = |w'|$. On remarque ensuite que $p = |uv|$ est une période de w' . On fixe un entier $1 \leq q \leq |w|$ et on montre que $q \in P(w)$ si et seulement si $q \in P(w')$. On considère deux cas suivant que q vérifie $q \leq |w| - p$ ou $q > |w| - p$.

On suppose d'abord que $q \leq |w| - p$. Si $q \in P(w')$, on a $|w'| = |w| \geq p + q$. D'après le théorème de Fine et Wilf (théorème 1.10), l'entier $d = p \wedge q$ est aussi une période de w' . Comme $d \leq p = |u'v'|$, d est aussi une période de $u'v'u'$ et donc de uvu . Comme d divise p , d est finalement une période de w . La minimalité

de p implique $d = p$. Comme p divise q , q est une période de w . Si à l'inverse $q \in P(w)$, d'après le lemme 1.12, p divise q et q est une période de w' .

On suppose maintenant que $|w| - p < q < |w|$ et on pose $r = q - (k - 1)p$ qui vérifie $|u| < r < p + |u|$. D'après le lemme précédent, $q \in P(w)$ si et seulement si $r \in P(uvu) = P(u'v'u')$ qui est, encore d'après le lemme précédent, équivalent à $q \in P(w')$. $\square$

Lemme 1.16. *Soit $w = uvu$ comme en (1.1) avec $k = 1$. Soit $u' \in \{0, 1\}^*$ tel que $P(u') = P(u)$. Si $b \in \{0, 1\}$ est tel que $u'1^{|v|-1}b$ soit primitif alors on a $P(w) = P(w')$ où $w' = u'1^{|v|-1}bu'$.*

Preuve. Soit q une période de w qui vérifie donc $q \geq p = |u'1^{|v|-1}b|$. Elle est donc aussi période de w' puisque $P(u) = P(u')$. Ceci montre l'inclusion $P(w) \subseteq P(w')$. Supposons par l'absurde que cette inclusion soit stricte et soit q le plus petit entier de $P(w') \setminus P(w)$. Comme u' commence par un 0, q vérifie $q < |u'|$ ou $p - 1 \leq q < |w|$.

Si q vérifie $q < |u'|$, il est, par définition, la plus petite période de w' . Le lemme 1.12 implique que q divise p et que le mot $u'1^{|v|-1}b$ n'est pas primitif, ce qui est une contradiction.

Si q est égal à $p - 1$, on a alors $b = 0$ et l'égalité $u'1 = 0u'$ est à nouveau impossible. On a donc $p < q < |w|$ et on pose $r = q - p$. L'entier r est une période de u' et donc de u . Ceci implique que q est une période de w qui est en contradiction avec la définition de q . $\square$

On est maintenant en mesure de prouver le théorème de Guibas et Odlyzko en combinant les différents lemmes.

Preuve du théorème 1.11. On raisonne par récurrence sur la longueur de w . Le résultat est trivial si $|w| \leq 2$. On suppose le résultat vrai pour tout mot de longueur inférieure à n et soit w un mot de longueur n qui s'écrit comme en 1.1.

Si $k \geq 2$, on a $|uvu| < n$ et par hypothèse de récurrence, il existe u' et v' tels que $|uv| = |u'v'|$ et $P(uvu) = P(u'v'u')$. Le lemme 1.15 permet alors de conclure.

Si $k = 1$, on a $|u| < n$ puisque v n'est pas le mot vide. Par hypothèse de récurrence, il existe u' tel que $P(u) = P(u')$. Si $u = \varepsilon$, on a alors $P(w) = \{|w|\}$ et $w' = 01^{|w|-1}$ vérifie $P(w) = P(w')$. Sinon, on peut supposer par symétrie que u' commence par 0. D'après le lemme 1.13, il existe $b \in \{0, 1\}$ tel que $u'1^{|v|-1}b$ est primitif. Le lemme 1.16 permet alors de conclure. $\square$

La preuve du théorème est effective. Il est même possible d'en déduire un algorithme qui calcule en temps linéaire en la longueur de w le mot w' tel que $P(w) = P(w')$.

Exercice 1.17. Soient six mots x, y, z, x', y' et z' tels que $xyz \neq x'y'z'$. Montrer qu'il existe un entier k_0 tel que pour tout entier $k \geq k_0$, on a $xy^kz \neq x'y'^kz'$.

1.3.2 Mots infinis

On introduit ici les mots infinis parce qu'ils sont très utiles pour l'étude des motifs inévitables.

Un *mot infini* x sur un alphabet A est une suite infinie $x = x_0x_1x_2 \dots$ de lettres de A . L'ensemble des mots infinis sur A est noté A^ω . Un mot $x =$

$x_0x_1x_2\cdots$ est *périodique* s'il existe un entier p appelé *période* tel que $x_n = x_{n+p}$ pour tout $n \geq 0$. Il est *ultimement périodique* s'il existe deux entiers l et p tels que $x_n = x_{n+p}$ pour tout $n \geq l$. Pour un mot fini u de longueur p , le mot infini de période p et dont les p premières lettres coïncident avec celles de u est noté u^ω . Le lemme 3.75 (p. 137) fournit une caractérisation en terme de facteurs des mots ultimement périodiques.

On décrit maintenant une façon très classique de construire des mots infinis. Un morphisme μ de A^* dans B^* est naturellement étendu aux mots infinis en posant $\mu(x) = \mu(x_0)\mu(x_1)\mu(x_2)\cdots$ pour tout mot infini $x = x_0x_1x_2\cdots$ sur A . Quelques précautions sont nécessaires si le morphisme μ est effaçant car $\mu(x)$ n'est pas nécessairement infini.

Soit μ un morphisme de A^* dans A^* tel que $\mu(a)$ commence par la lettre a , c'est-à-dire $\mu(a) = au$ pour $u \in A^*$. On définit par récurrence la suite de mots finis $(w_n)_{n \geq 0}$ par $w_0 = a$ et $w_{n+1} = \mu(w_n)$. On montre par récurrence que chaque mot w_n est préfixe du mot w_{n+1} . Si de plus $\lim_{n \rightarrow \infty} |w_n| = +\infty$, la suite de mots w_n converge vers un mot infini noté $\mu^\omega(a)$. Ce mot est appelé *point fixe* du morphisme μ car il vérifie $\mu(\mu^\omega(a)) = \mu^\omega(a)$.

Exemple 1.18. Soit A l'alphabet $\{0, 1, 2\}$ et $\psi : A^* \rightarrow A^*$ le morphisme défini de la manière suivante.

$$\psi : \begin{cases} 0 \mapsto 01 \\ 1 \mapsto 221 \\ 2 \mapsto 2 \end{cases}$$

On vérifie sans peine que le point fixe $\psi^\omega(0)$ est le mot $a_0a_1a_2\cdots$ où $a_0 = 0$, $a_i = 1$ si i est un carré non nul et $a_i = 2$ sinon.

L'exemple précédent montre que le mot caractéristique des carrés (qui a 1 aux positions qui sont des carrés et 0 ailleurs) est un mot de la forme $\mu(\psi^\omega(0))$ pour des morphismes μ et ψ . Cette propriété reste encore vraie pour l'ensemble $\{P(n) \mid n \geq 0\}$ des valeurs prises par un polynôme P tel que $P(n)$ est entier pour tout $n \geq 0$.

Exemple 1.19. Soit le morphisme μ défini de la manière suivante.

$$\mu : \begin{cases} 0 \mapsto 01 \\ 1 \mapsto 0 \end{cases}$$

La suite de mots $f_n = \mu^n(0)$ vérifie la relation de récurrence $f_{n+2} = f_{n+1}f_n$ et les valeurs initiales sont $f_0 = 0$ et $f_1 = 01$. Il s'agit donc de la suite des mots de Fibonacci déjà rencontrée au théorème de Fine et Wilf (théorème 1.10). Par extension, le mot $f = \mu^\omega(0)$ est encore appelé *mot de Fibonacci*. Le début de ce mot est le suivant.

$$f = 010010100100101001010010010100100101001001010010100101001010 \cdots$$

1.3.3 Motifs inévitables

Le but de cette partie est d'étudier les mots qui évitent, c'est-à-dire ne contiennent pas comme facteurs certains mots. On suppose fixé un ensemble F de mots et on s'intéresse aux mots de $A^* \setminus A^*FA^*$. Plus particulièrement, la question principale est de déterminer si cet ensemble est fini ou non. Le lemme suivant reformule cette question en terme de mots infinis.

Lemme 1.20. *Soit F un sous-ensemble de mots sur un alphabet fini A . L'ensemble $A^* \setminus A^*FA^*$ est infini si et seulement si il existe un mot infini n'ayant aucun facteur dans F .*

Preuve. Il est clair que si le mot infini x n'a aucun facteur dans F , alors chacun de ses préfixes a la même propriété et l'ensemble en question est infini. La réciproque est une conséquence directe du lemme de König (lemme 3.18 p. 109). $\square$

Un *carré* est un mot de la forme uu pour un mot u . L'ensemble F est ici l'ensemble des carrés non vides. Un mot est dit *sans carré* s'il ne contient pas de facteur, autre que le mot vide, qui ne soit un carré. Nous allons voir que sur un alphabet à trois lettres, il existe des mots sans carré arbitrairement longs.

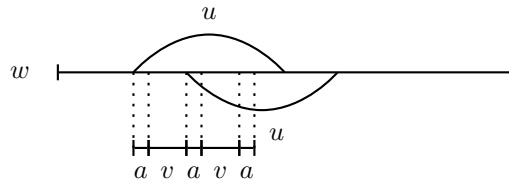


FIG. 1.4 – Chevauchement de deux occurrences d'un facteur u

Sur l'alphabet à deux lettres $\{0, 1\}$, les seuls mots sans carré sont les mots 010 et 101 et leurs facteurs. Pour un alphabet à trois lettres, on commence par étudier les chevauchements. Un *chevauchement* est un mot de la forme $uvuvu$ où u est un mot non vide et v un mot quelconque. Un mot est dit *sans chevauchement* s'il ne contient pas de facteur qui ne soit un chevauchement. La terminologie est justifiée par la remarque suivante. Un mot w contient un chevauchement si et seulement si un de ses facteurs a deux occurrences qui se chevauchent (cf. figure 1.4). Si w a un chevauchement $uvuvu$, le mot uvu a deux occurrences qui se chevauchent dans w . Réciproquement, soit w un mot ayant deux occurrences d'un facteur u qui se chevauchent. Soit a la première lettre de u et soit v le reste de u jusqu'au début de l'occurrence suivante (cf. figure 1.4). Le mot $avava$ est un chevauchement de w . Ceci montre en particulier qu'un mot ayant un chevauchement a toujours un chevauchement de la forme $avava$ où a est une lettre.

On commence par montrer qu'il existe des mots arbitrairement longs sans chevauchement sur l'alphabet $A = \{0, 1\}$. Soit le morphisme de Thue-Morse τ de A^* dans A^* défini de la manière suivante.

$$\tau : \begin{cases} 0 \mapsto 01 \\ 1 \mapsto 10 \end{cases}$$

Le mot infini $\tau^\omega(0)$ est appelé *mot de Thue-Morse*. On prouve sans difficulté que la n -ième lettre du mot de Thue-Morse est 0 si le nombre de 1 dans l'écriture binaire de n est pair et 1 sinon. Le début du mot de Thue-Morse est le suivant.

01101001100101101001011001101001...

Proposition 1.21. *Le mot de Thue-Morse est sans chevauchement.*

Soit X l'ensemble $(01 + 10)^*$. Par définition même du morphisme τ , tout mot $\tau^n(0)$ appartient à X pour $n \geq 1$. On commence par établir une propriété élémentaire de l'ensemble X .

Lemme 1.22. *Si x appartient à X , alors $0x0$ et $1x1$ n'appartiennent pas à X .*

Preuve. On raisonne par récurrence sur la longueur de x . Si x est le mot vide, ni 00 ni 11 n'appartient à X . Soit $x = x_1 \cdots x_n$ un mot de X de longueur $n \geq 1$. Si $0x0$ appartient à X , on a $x_1 = x_n = 1$ et le mot $x_2 \cdots x_{n-1}$ appartient à X . Ceci aboutit à une contradiction. Le raisonnement pour $1x1$ est similaire. $\square$

Preuve de la proposition. Il suffit de montrer que les mots $\tau^n(0)$ sont sans chevauchement. On raisonne par récurrence sur n . Pour $n = 1$, le mot $\tau(0) = 01$ ne contient pas de chevauchement. Supposons par l'absurde que le mot $w = \tau^{n+1}(0)$ contienne un chevauchement. Comme cela a été vu, on peut supposer que ce chevauchement est de la forme $avava$ où $a \in \{0, 1\}$ et $v \in \{0, 1\}^*$. Le mot w se factorise donc $w = xavavay$. On peut supposer que x est de longueur paire. L'autre cas s'obtient en passant au mot miroir puisque x et y ont des longueurs de parités différentes. Si v est de longueur paire, la seconde occurrence de v commence à une position paire. On en déduit que v et ava sont des mots de X^* et ceci est une contradiction d'après le lemme précédent. Si v est de longueur impaire, il se factorise $v = \bar{a}v'$. Comme le premier et le dernier a de $avava$ sont à des positions de même parité, la première lettre de y est aussi $\bar{a}$ et y se factorise $y = \bar{a}y'$. Le mot w se factorise finalement $w = xa\bar{a}v'a\bar{a}v'a\bar{a}y'$. Comme x est de longueur paire, le mot $a\bar{a}v'a\bar{a}v'a\bar{a}$ est l'image d'un facteur $auaua$ de $\tau^n(0)$. Ceci est en contradiction avec l'hypothèse de récurrence. $\square$

On considère le morphisme σ de $\{0, 1, 2\}$ dans $\{0, 1\}$ défini de la manière suivante.

$$\sigma : \begin{cases} 0 \mapsto 0 \\ 1 \mapsto 01 \\ 2 \mapsto 011 \end{cases}$$

Chaque mot $\tau^n(0)$ ne contient pas de facteur 111 et commence par 0 . Il appartient donc à l'ensemble $(0+01+011)^*$. On en déduit que pour tout entier n , il existe un mot w_n sur l'alphabet $\{0, 1, 2\}$ tel que $\sigma(w_n) = \tau^n(0)$. Le mot w_n est en outre unique. Chaque mot w_n est préfixe du mot w_{n+1} . La suite $(w_n)_{n \geq 0}$ converge vers un mot infini que l'on note $\sigma^{-1}(\tau^\omega(0))$. Le début de ce mot est le suivant.

21020121012021020120210121020120...

Théorème 1.23 (Thue 1906). *Le mot $\sigma^{-1}(\tau^\omega(0))$ est sans carré.*

Preuve. Il suffit de prouver que chacun des mots w_n est sans carré. Supposons par l'absurde que w_n contienne un carré uu . On peut supposer que uu a une occurrence qui n'est pas à la fin de w_n . Sinon, on remplace w_n par w_{n+1} . Comme uua est un facteur de w_n , $\sigma(u)\sigma(u)\sigma(a)$ est un facteur de $\tau^n(0)$. Les mots $\sigma(u)$ et $\sigma(a)$ commencent par 0 . Le mot $\tau^n(0)$ a alors un chevauchement $avava$ contrairement au résultat de la proposition précédente. $\square$

Le mot infini $\sigma^{-1}(\tau^\omega(0))$ peut être directement obtenu en itérant un morphisme. Soit le morphisme μ défini de la manière suivante.

$$\mu : \begin{cases} 0 \mapsto 1 \\ 1 \mapsto 20 \\ 2 \mapsto 210 \end{cases}$$

La suite de mots $(\mu^n(2))_{n \geq 0}$ converge vers le mot $\sigma^{-1}(\tau^\omega(0))$. En effet, on vérifie l'égalité $\tau \circ \sigma = \sigma \circ \mu$. En utilisant en outre l'égalité $\sigma(2)0 = \tau^2(0)$, on montre par récurrence que $\sigma(\mu^n(2))\tau^n(0) = \tau^{n+2}(0)$.

Soit le morphisme ν défini de la manière suivante.

$$\nu : \begin{cases} 0 \mapsto 010 \\ 1 \mapsto 011 \end{cases}$$

Soit $\nu^\omega(0)$ le point fixe de ce morphisme. Ce mot infini est sans cube. En observant les façons dont les mots de longueur 2 se découpent sur $\{010, 011\}$, on prouve facilement que si $\nu^\omega(0)$ contenait un cube, il contiendrait un cube de longueur strictement plus petite.

Les carrés et les chevauchements sont des cas particuliers des répétitions qui sont des puissances non entières de mots. Soit u un mot de longueur p et soit r un nombre rationnel de la forme $r = q/p$. On note u^r , l'unique mot fini de longueur q qui est préfixe du mot u^ω . Ainsi un carré est mot de la forme u^r avec $r = 2$ alors qu'un chevauchement est un mot de la forme u^r avec $r > 2$.

Plus généralement, un *motif* est un mot p sur un alphabet X de variables. Un mot sur un alphabet A *contient* le motif p , s'il possède un facteur de la forme $\mu(p)$ pour un morphisme $\mu : X^* \rightarrow A^*$ non effaçant. Dans le cas contraire, on dit qu'il *évite* le motif p . Les carrés correspondent au motif xx et les chevauchements $uvuv$ correspondent au motif $xyxy$ si v est non vide et au motif xxx si v est vide. Un motif p est dit *k-inévitable* si l'ensemble des mots sur un alphabet à k lettres qui évitent p est fini. Il est dit *inévitable* s'il est inévitable pour tout entier k . La référence sur les motifs inévitables est [Cas02]. Le lemme suivant permet de construire des motifs inévitables.

Lemme 1.24. *Si p est un motif inévitable et si la lettre x n'apparaît pas dans p , alors le motif pxp est encore inévitable.*

Preuve. Soit A un alphabet. L'ensemble des mots sur A qui évitent p est fini. Il existe un entier l tel que tout mot de longueur l contient le motif p . Soit $n = |A|^l$ le nombre de mots sur A de longueur l . Un mot w de longueur $nl + l + n$ se factorise $w = u_0 a_1 u_1 a_2 \cdots a_n u_n$ où $a_1, \dots, a_n$ sont des lettres et $u_0, \dots, u_n$ sont des mots de longueur l . Par définition de n , deux mots u_i et u_j pour $i < j$ sont égaux. Le mot w se factorise donc $v_0 u_i v_1 u_i v_2$. Comme u_i est de longueur l , il se factorise $u_i = v_3 \mu(p) v_4$ où μ est une morphisme de $(X \setminus \{x\})^*$ dans A^* . En prolongeant μ à X^* par $\mu(x) = v_4 v_1 v_3$, le mot w contient le facteur $\mu(pxp)$. $\square$

Soit X l'alphabet infini $\{x_0, x_1, x_2, \dots\}$. On définit la suite $(z_n)_{n \geq 0}$ des mots de Zimin par $z_0 = \varepsilon$ et $z_{n+1} = z_n x_n z_n$ pour tout $n \geq 0$. Grâce au lemme précédent, chacun des mots z_n est un motif inévitable. La suite des mots de Zimin converge vers un mot infini z . Pour tout $n \geq 0$, la n -ième lettre de z est x_k si 2^k est la plus grande puissance de 2 qui divise n . Le début du mot z est le suivant.

$$z = x_0 x_1 x_0 x_2 x_0 x_1 x_0 x_3 x_0 x_1 x_0 x_2 x_0 x_1 x_0 x_4 x_0 x_1 x_0 x_2 x_0 x_1 x_0 x_3 \cdots$$

Il existe un algorithme dû à Zimin qui calcule si un motif donné est inévitable ou non. Cet algorithme est détaillé en [Cas02]. Par contre, aucun algorithme n'est connu pour déterminer si un mot est k -inévitale pour un entier k donné.

1.4 Un peu d'ordre

Dans cette partie, on s'intéresse aux bons quasi-ordres et aux théorèmes de Higman et de Kruskal. Ces ordres sont souvent utilisés pour prouver la terminaison de systèmes de réécriture ou de procédures. On rappelle aussi quelques ordres classiques sur l'ensemble des mots.

Une relation binaire $\preceq$ sur un ensemble E est un *quasi-ordre* (on dit aussi *préordre*) si elle est réflexive et transitive. Contrairement à un ordre, un quasi-ordre n'est pas nécessairement antisymétrique. On peut très bien avoir $x \preceq y$ et $y \preceq x$ pour deux éléments distincts x et y . On écrit $y \succ x$ pour $x \preceq y$. On écrit $x \prec y$ si $x \preceq y$ et $y \not\preceq x$. À un quasi-ordre $\preceq$ sur E est associée une relation d'équivalence $\approx$ définie par $x \approx y$ si $x \preceq y$ et $y \preceq x$. Le quasi-ordre $\preceq$ induit alors un ordre sur l'ensemble quotient $E/\approx$. Deux éléments x et y sont dits *incomparables* (pour $\preceq$) si $x \not\preceq y$ et $y \not\preceq x$.

Exemple 1.25. Soit $E = \mathbb{N}$ l'ensemble des entiers naturels. La relation de divisibilité est un ordre sur $\mathbb{N}$. Soit la relation $\preceq$ définie sur $\mathbb{N}$ par $m \preceq n$ s'il existe un entier k tel que m divise n^k , c'est-à-dire si tout diviseur premier de m est aussi diviseur premier de n . C'est un quasi-ordre qui n'est pas un ordre. On a en effet $6 \preceq 12$ et $12 \preceq 6$ car 6 et 12 ont les mêmes diviseurs premiers.

Une relation d'équivalence $\sim$ sur un ensemble E est un quasi-ordre sur E dont la relation d'équivalence associée $\approx$ est justement la relation $\sim$.

Soit $\preceq$ un quasi-ordre sur E . Une *chaîne décroissante* est une suite $(x_i)_{i \geq 0}$ finie ou infinie telle que $x_i \succ x_{i+1}$ pour tout $i \geq 0$. Un quasi-ordre est dit *bien fondé* s'il ne possède pas de chaîne infinie décroissante. Autrement dit, le quasi-ordre $\preceq$ est bien fondé si la relation $\succ$ induite sur $E/\approx$ est noethérienne (cf. section 2.7.1). Une *antichaîne* est un ensemble d'éléments incomparables deux à deux. Un *idéal* (d'ordre) est un ensemble I tel que si x appartient à I et $x \preceq y$, alors y appartient aussi à I . Une *base* d'un idéal I est un sous-ensemble B de I tel que $I = \{x \mid \exists b \in B \ b \preceq x\}$. On dit alors que l'idéal I est *engendré* par la base B . Le théorème suivant donne plusieurs propriétés équivalentes qui définissent la notion fondamentale de bon quasi-ordre.

Théorème 1.26 (Higman). *Les propriétés suivantes sont équivalentes pour un quasi-ordre sur E .*

- i) *tout idéal possède une base finie,*
- ii) *toute chaîne croissante $I_0 \subseteq I_1 \subseteq I_2 \subseteq \dots$ d'idéaux est constante à partir d'un certain rang,*
- iii) *toute suite infinie d'éléments de E contient une sous-suite infinie croissante,*
- iv) *toute suite infinie d'éléments de E contient une sous-suite croissante de longueur 2,*
- v) *toute chaîne décroissante est finie et toute antichaîne est finie,*
- vi) *tout quasi-ordre $\preceq'$ qui prolonge $\preceq$ est bien fondé.*

Un quasi-ordre qui satisfait l'une de ces propriétés est appelé bon quasi-ordre.

Il y a une différence de terminologie entre le français et l'anglais. En français, un ordre est implicitement total et le qualificatif *partiel* est ajouté s'il ne l'est pas. En anglais, un ordre est supposé partiel et on utilise l'adjectif *total* ou *linear* pour préciser qu'il est total. En anglais, un bon quasi-ordre est appelé *well quasi-order* souvent abrégé en *wqo*.

On a vu qu'une relation d'équivalence est un quasi-ordre. Elle est un bon quasi-ordre si et seulement si elle a un nombre fini de classes. En effet, deux éléments non équivalents sont incomparables. Pour qu'il n'existe pas d'antichaine infinie, le nombre de classes doit être fini. La réciproque est évidente.

Exemple 1.27. L'ordre naturel des entiers naturels est un bon quasi-ordre sur $\mathbb{N}$. L'ordre naturel des entiers relatifs n'est pas bien fondé sur $\mathbb{Z}$ et il n'est donc pas un bon quasi-ordre. La divisibilité est un ordre bien fondé sur les entiers mais elle n'est pas un bon quasi-ordre car les nombres premiers forment une antichaine infinie.

Preuve. On commence par montrer l'équivalence entre i) et ii). Soit $I_0 \subseteq I_1 \subseteq I_2 \subseteq \dots$ une chaîne croissante d'idéaux. L'ensemble $I = \bigcup_{n \geq 0} I_n$ est encore un idéal. Si le quasi-ordre a la propriété de base finie, cet idéal est engendré par une base finie B . Il existe alors un entier k tel que $B \subseteq I_k$. On a alors $I = I_k$ et $I_{k+n} = I_k$ pour tout $n \geq 0$. Réciproquement soit I un idéal. Si I ne possède pas de base finie, on construit par récurrence une suite $(x_n)_{n \geq 0}$ d'éléments de I de la manière suivante. L'élément x_0 est choisi de façon arbitraire dans I . Les éléments $x_0, \dots, x_k$ étant déjà choisis, l'élément x_{k+1} est choisi en dehors de l'idéal engendré par $B_k = \{x_0, \dots, x_k\}$. Soit I_k l'idéal engendré par B_k . Par construction, la chaîne d'idéaux $I_0 \subsetneq I_1 \subsetneq I_2 \subsetneq \dots$ est strictement croissante.

On montre maintenant l'équivalence entre iii), iv) et v). Il est clair que iii) implique iv) et que iv) implique à nouveau v). Soit $(x_n)_{n \geq 0}$ une suite d'éléments. On extrait de cette suite une sous-suite infinie croissante $(x_{i_n})_{n \geq 0}$ de la façon suivante. On appelle élément *minimal* d'un sous-ensemble X , un élément x de X tel que pour tout x' de X , $x' \preceq x$ implique $x \preceq x'$ et donc $x \approx x'$. Soit $X = \{x_n \mid n \geq 0\}$ l'ensemble des valeurs de la suite. Puisque il n'y a pas de chaîne infinie décroissante, il existe au moins un élément minimal dans X . Pour deux éléments minimaux x et x' , soit ils sont incomparables soit ils vérifient $x \approx x'$. Puisqu'il n'y a pas d'antichaine infinie, il existe un nombre fini de classes de $\approx$ formées d'éléments minimaux. Soient $z_1, \dots, z_k$ des représentants de ces classes. Pour tout élément x_n de la suite, il existe au moins un représentant z_j tel que $x_n \succeq z_j$. Il existe donc au moins un représentant z_{j_0} tel qu'il existe une infinité d'éléments x_n vérifiant $x_n \succeq z_{j_0}$. On pose alors $x_{i_0} = z_{j_0}$ et on recommence avec la sous-suite des éléments x_n tels que $n > i_0$ et $x_n \succeq x_{i_0}$.

On montre ensuite l'équivalence entre i)-ii) et iii)-iv)-v). Soit $(x_n)_{n \geq 0}$ une suite d'éléments. Soit I_k l'idéal engendré par la base $B_k = \{x_0, \dots, x_k\}$. Puisque la chaîne d'idéaux $I_0 \subseteq I_1 \subseteq I_2 \subseteq \dots$ est constante à partir d'un certain rang, il existe deux indices $k < l$ tel que $x_k \preceq x_l$, ce qui montre iv). Réciproquement, soit $I_0 \subseteq I_1 \subseteq I_2 \subseteq \dots$ une chaîne d'idéaux. Si cette chaîne n'est pas constante à partir d'un certain rang, on peut supposer, quitte à supprimer quelques idéaux de la chaîne, que chaque inclusion $I_n \subsetneq I_{n+1}$ est stricte. Soit x_n un élément de $I_{n+1} \setminus I_n$. La suite $(x_n)_{n \geq 0}$ ne contient aucune sous-suite croissante de longueur 2. Ceci contredit l'hypothèse que la chaîne d'idéaux n'est pas constante à partir d'un certain rang.

On montre finalement l'équivalence entre vi) et les autres propriétés. Soit $\preceq'$ un quasi-ordre qui prolonge $\preceq$. Soit $(x_n)_{n \geq 0}$ une suite d'éléments. D'après iv), il existe deux indices $k < l$ tel que $x_k \preceq x_l$ et donc $x_k \preceq' x_l$. Ceci montre que le quasi-ordre $\preceq'$ n'a pas de chaîne infinie décroissante. Réciproquement, le quasi-ordre $\preceq$ est son propre prolongement. Ceci montre qu'il est bien fondé. Il reste à montrer qu'il n'a pas d'antichaîne infinie. On montre que si A est une antichaîne de $\preceq$ et si $\preceq_0$ est quasi-ordre quelconque sur A , il est possible de prolonger $\preceq$ en un quasi-ordre $\preceq'$ tel que la restriction de $\preceq'$ à A est $\preceq_0$. Soit $\preceq'$ la clôture transitive de la relation $\preceq \cup \preceq_0$. C'est bien sûr un quasi-ordre. Il reste à prouver que pour tous éléments a et b de A , $a \preceq' b$ implique $a \preceq_0 b$ puisque la réciproque est évidente. Si $a \preceq' b$, il existe une suite finie $x_0, \dots, x_n$ telle que $x_0 = a$, $x_n = b$ et $x_i \preceq x_{i+1}$ ou $x_i \preceq_0 x_{i+1}$ pour tout $0 \leq i \leq n-1$. Si $n = 1$, le résultat est acquis puisque a et b sont incomparables pour $\preceq$. On suppose maintenant que $n \geq 2$. En utilisant la transitivité de $\preceq_0$, on peut supposer que les éléments $x_1, \dots, x_{n-1}$ n'appartiennent pas à A . On a alors $x_i \preceq x_{i+1}$ pour tout $0 \leq i \leq n-1$ et donc $a \preceq b$ contrairement à l'hypothèse que a et b sont incomparables pour $\preceq$. Ceci montre que $\preceq'$ coïncide avec $\preceq_0$ sur A . Si le quasi-ordre $\preceq$ a une antichaîne infinie A , on peut le prolonger en utilisant un quasi-ordre $\preceq_0$ sur A qui contient une chaîne infinie décroissante. Le quasi-ordre obtenu contient encore une chaîne infinie décroissante et n'est pas bien fondé. $\square$

Soient $\preceq_1$ et $\preceq_2$ deux quasi-ordres sur des ensembles E_1 et E_2 . Le produit cartésien $E_1 \times E_2$ est naturellement muni d'un quasi-ordre $\preceq$ défini par $(x_1, x_2) \preceq (y_1, y_2)$ si $x_1 \preceq_1 y_1$ et $x_2 \preceq_2 y_2$. Le résultat suivant est parfois attribué à Nash-Williams.

Lemme 1.28. *Si $\preceq_1$ et $\preceq_2$ sont des bons quasi-ordres sur E_1 et E_2 , alors $\preceq$ est un bon quasi-ordre sur $E_1 \times E_2$.*

La preuve découle directement de la caractérisation iii) du théorème précédent. L'ordre naturel sur les entiers induit un ordre naturel sur les k -uplets d'entiers défini par $(m_1, \dots, m_k) \leq (n_1, \dots, n_k)$ si $m_i \leq n_i$ pour tout $1 \leq i \leq k$. Le lemme précédent a pour conséquence le résultat suivant appelé lemme de Dickson.

Lemme 1.29 (Dickson 1913). *Tout sous-ensemble de $\mathbb{N}^k$ a un nombre fini d'éléments minimaux.*

1.4.1 Quasi-ordres sur les mots

Soit $\preceq$ un quasi-ordre sur un ensemble A . On définit un quasi-ordre $\preceq^*$ sur l'ensemble A^* des mots finis sur A de la façon suivante. Deux mots $w = a_1 \dots a_m$ et $w' = a'_1 \dots a'_n$ vérifient $w \preceq^* w'$ s'il existe une suite croissante $1 \leq j_1 < j_2 < \dots < j_m \leq n$ d'indices tels que $a_i \preceq a'_{j_i}$ pour tout $1 \leq i \leq m$. Pour une relation R , on note parfois R^* la clôture réflexive et transitive de la relation R (cf. section 2.7.1). Ici le quasi-ordre $\preceq^*$ n'est pas du tout la clôture réflexive et transitive de $\preceq$ (qui est justement égale à $\preceq$). L'exposant $*$ rappelle seulement que $\preceq^*$ est l'extension à A^* du quasi-ordre $\preceq$. On vérifie facilement que $\preceq^*$ est le plus petit (au sens de l'inclusion) quasi-ordre sur A^* tel que :

1. $uv \preceq^* uav$ pour tous mots $u, v \in A^*$ et toute lettre $a \in A$,

2. $uav \preceq^* ubv$ si $a \preceq b$ pour tous mots $u, v \in A^*$ et toutes lettres $a, b \in A$.

Soit $w = w_1 \cdots w_n$ un mot de longueur n . Un *sous-mot* de w est mot u tel qu'il existe une sous-suite croissante $1 \leq i_1 < \cdots < i_k \leq n$ d'indices vérifiant $u = w_{i_1} \cdots w_{i_k}$. Autrement dit, le mot u est obtenu en supprimant certaines lettres du mot w . La relation *être sous-mot* est bien sûr un ordre partiel sur l'ensemble des mots. La notion de sous-mot est à distinguer de celle de facteur où les lettres choisies sont consécutives.

Exemple 1.30. Si le quasi-ordre $\preceq$ sur A est l'égalité, le quasi-ordre $\preceq^*$ est l'ordre des sous-mots.

Théorème 1.31 (Higman 1952). *Si $\preceq$ est un bon quasi-ordre sur A , alors $\preceq^*$ est un bon quasi-ordre sur A^* .*

Preuve. On dit qu'une suite $(u_n)_{n \geq 0}$ est *mauvaise* s'il n'existe pas deux indices $k < l$ tels que $u_k \preceq^* u_l$ et on utilise la caractérisation iv) du théorème 1.26. On raisonne par l'absurde et on suppose qu'il existe au moins une mauvaise suite. On construit alors par récurrence une mauvaise suite minimale $(u_n)_{n \geq 0}$. Soit u_0 un mot le plus court possible tel que u_0 soit le premier élément d'une mauvaise suite. Les mots $u_0, \dots, u_n$ étant choisis, u_{n+1} est un mot le plus court tel que $u_0, \dots, u_{n+1}$ sont les premiers éléments d'une mauvaise suite. La suite ainsi construite est mauvaise. Puisque le mot vide vérifie $\varepsilon \preceq^* w$ pour tout mot w , aucun des mots u_n n'est vide. Chaque mot u_n s'écrit donc $u_n = a_n v_n$ où a_n est une lettre et v_n un mot. Comme $\preceq$ est un bon quasi-ordre, il existe une suite extraite $(a_{i_n})_{n \geq 0}$ qui est croissante. On considère alors la suite suivante

$$u_0, u_1, u_2, \dots, u_{i_0-2}, u_{i_0-1}, v_{i_0}, v_{i_1}, v_{i_2}, \dots$$

obtenue en prenant les i_0 premiers éléments $u_0, u_1, \dots, u_{i_0-1}$ de la suite $(u_n)_{n \geq 0}$ puis tous les éléments de la suite $(v_{i_n})_{n \geq 0}$. Cette nouvelle suite est encore mauvaise. On ne peut pas avoir $k < l < i_0$ et $u_k \preceq^* u_l$, car $(u_n)_{n \geq 0}$ est mauvaise. La relation $u_k \preceq^* v_{i_l}$ implique la relation $u_k \preceq^* u_{i_l}$ et la relation $v_{i_k} \preceq^* v_{i_l}$ implique $u_{i_k} \preceq^* u_{i_l}$ car $a_{i_k} \preceq a_{i_l}$. On obtient alors une contradiction car v_{i_0} est plus court que u_{i_0} . $\square$

Exemple 1.32. D'après le théorème de Higman, l'ordre des sous-mots sur un alphabet fini est un bon quasi-ordre. En effet, l'égalité sur A est bien sûr un bon quasi-ordre sur A .

Le quasi-ordre $\preceq^*$ défini sur les suites finies peut être étendu en un quasi-ordre $\preceq^\omega$ sur A^ω en posant $(a_n)_{n \geq 0} \preceq^\omega (a'_n)_{n \geq 0}$ s'il existe une suite strictement croissante d'indices $(i_n)_{n \geq 0}$ telle que $a_n \preceq a'_{i_n}$ pour tout $n \geq 0$. L'exemple suivant montre que le théorème de Higman ne se généralise pas à $\preceq^\omega$.

Exemple 1.33. Soit $A = \mathbb{N} \times \mathbb{N}$ l'ensemble des paires d'entiers et soit l'ordre $\preceq$ défini sur A par $(m, n) \preceq (m', n')$ si $m = m'$ et $n \leq n'$ ou $m < m'$ et $n < m'$. On vérifie que c'est un bon ordre partiel. Pour chaque entier i , soit $x_i \in A^\omega$ donné par

$$x_i = (i, 0), (i, 1), (i, 2), (i, 3), \dots$$

La suite $(x_i)_{i \geq 0}$ ne contient pas de sous-suite croissante de longueur 2 pour $\preceq^\omega$.

Pour obtenir une généralisation du théorème de Higman aux suites infinies, des meilleurs quasi-ordres (better quasi-order en anglais souvent abrégé en *bqo*) ont été introduits par Nash-Williams. Pour une définition des meilleurs quasi-ordre et quelques résultats, on pourra consulter [Alm94, p. 33].

Exemple 1.34. L'ordre des sous-mots est défini sur A^ω par $(a_n)_{n \geq 0} \preceq^\omega (a'_n)_{n \geq 0}$ s'il existe une suite strictement croissante d'indices $(i_n)_{n \geq 0}$ telle que $a_n = a'_{i_n}$ pour tout $n \geq 0$. Si l'alphabet A est fini, c'est un bon quasi-ordre sur A^ω . Ceci provient du fait que l'égalité est un meilleur quasi-ordre sur A .

1.4.2 Ordres sur les mots

L'ordre des sous-mots n'est pas l'ordre le plus naturel sur les mots. Le plus familier est sans aucun doute l'*ordre lexicographique* utilisé dans le dictionnaire. On suppose que A est muni d'un ordre noté $<$. Deux mots w et w' vérifient $w <_{\text{lex}} w'$ soit si w est préfixe de w' soit s'il existe deux lettres a et b vérifiant $a < b$ et trois mots u, v et v' tels que $w = uav$ et $w' = ubv'$. Le mot u est en fait le plus long préfixe commun à w et w' qui est noté $w \wedge w'$. Dans le cas où w est préfixe de w' , on a $w \wedge w' = w$.

On pourra remarquer que la fonction d définie par $d(w, w') = |w| + |w'| - 2|w \wedge w'|$ est une distance sur l'ensemble des mots.

L'ordre lexicographique est total mais il n'est pas bien fondé dès que l'alphabet a au moins deux lettres. Sur l'alphabet $A = \{0, 1\}$, la suite de mots $w_n = 0^n 1$ est infinie et décroissante. Pour cette raison, on considère aussi l'*ordre hiérarchique* où les mots sont d'abord classés par longueur puis par ordre lexicographique. Deux mots w et w' vérifient $w <_{\text{hié}} w'$ si $|w| < |w'|$ ou $|w| = |w'|$ et $w <_{\text{lex}} w'$. Cet ordre a l'avantage d'être total et bien fondé.

1.4.3 Quasi-ordres sur les arbres

Le théorème de Higman admet une généralisation aux arbres finis due à Kruskal. La construction de bons quasi-ordres sur les arbres est essentielle pour la terminaison de systèmes de réécriture de termes, souvent utilisés pour donner une sémantique aux langages de programmation fonctionnels.

On commence par rappeler la notion de domaine d'arbre. On note $\mathbb{N}^*$ l'ensemble des suites finies d'entiers. Un *domaine d'arbre* est sous-ensemble D de $\mathbb{N}^*$ qui vérifie les deux conditions suivantes.

1. D est clos par préfixe, c'est-à-dire si uv appartient à D , alors u appartient aussi à D pour toutes suites $u, v \in \mathbb{N}^*$.
2. D est clos par valeurs inférieures, c'est-à-dire si ui appartient à D , alors uj appartient aussi à D pour tout $u \in \mathbb{N}^*$ et tous entiers $0 \leq j \leq i$.

Soit A un alphabet non nécessairement fini. On appelle *arbre sur A* une fonction $t : D \rightarrow A$ où D est un domaine d'arbre. Un élément du domaine D de l'arbre est appelé un *nœud* de l'arbre. L'arbre est dit *vide* (resp. *fini*) si son domaine D est vide (resp. fini). On appelle *taille* de t et on note $|t|$ le cardinal du domaine de t . Le domaine d'un arbre t est noté $\text{dom}(t)$. Un arbre est représenté comme un graphe ayant les nœuds pour sommets et les paires de nœuds de la forme (u, ui) pour arêtes.

Un mot fini $w = a_0 \cdots a_n$ s'identifie avec un arbre t de domaine $D = \{\varepsilon, 0, 0^2, \dots, 0^n\} = (\varepsilon + 0)^n$ et défini par $t(0^k) = a_k$ pour tout $0 \leq k \leq n$.

Exemple 1.35. Soit le domaine d'arbre D égal à l'ensemble

$$D = \{\varepsilon, 0, 1, 00, 10, 11, 12, 000, 001, 110, 111, 1110\}$$

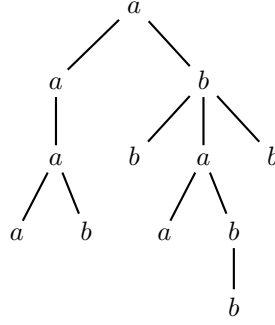


FIG. 1.5 – L'arbre de l'exemple 1.35

et soit l'arbre t défini sur ce domaine par $t(w) = a$ si w a un nombre pair de 1 et $t(w) = b$ sinon. Cet arbre t est représenté à la figure 1.5.

Soit t un arbre de domaine D et soit w un mot sur $\mathbb{N}$. L'arbre $w^{-1}t$ est l'arbre de domaine $w^{-1}D$ défini par $(w^{-1}t)(u) = t(wu)$ pour tout $u \in w^{-1}D$. Cet arbre est appelé le *sous-arbre de t enraciné en w* . Si w n'appartient pas au domaine de t , l'arbre $w^{-1}D$ est vide.

Soit $\preceq$ un quasi-ordre sur un ensemble A . On définit un quasi-ordre $\preceq^\Delta$ sur l'ensemble de tous les arbres finis sur A de la façon suivante. Deux arbres t et t' de domaines respectifs D et D' vérifient $t \preceq^\Delta t'$ s'il existe une fonction injective f de D dans D' telle que pour tous w et w' dans D , on ait les propriétés suivantes.

1. $t(w) \preceq t'(f(w))$,
2. $f(w \wedge w') = f(w) \wedge f(w')$ et
3. Si $w <_{\text{lex}} w'$ alors $f(w) <_{\text{lex}} f(w')$

où $w \wedge w'$ dénote le plus long préfixe commun de w et w' .

La fonction f généralise la suite croissante d'indices $j_1, \dots, j_m$ utilisée dans le cas des mots pour définir l'ordre $\preceq^*$. La première condition vérifiée par f est l'analogue de la condition $a_i \preceq a'_{j_i}$ pour les mots. Les deux autres conditions assurent que f est bien un plongement de l'arbre t dans l'arbre t' . La seconde condition exprime que la relation de parenté est préservée et la dernière que l'ordre des fils est également respecté.

Ce quasi-ordre peut être aussi défini de manière récursive sur les arbres.

Théorème 1.36 (Kruskal 1960). *Si $\preceq$ est un bon quasi-ordre sur A , alors $\preceq^\Delta$ est un bon quasi-ordre sur l'ensemble des arbres finis sur A .*

Dans les arbres que nous avons définis, les fils de chaque nœud sont ordonnés et ils forment une suite. Il est possible de définir des arbres où l'ordre des fils est sans importance. Le théorème de Kruskal reste vrai pour cette variante des arbres.

Preuve. La preuve reprend les grandes lignes de la preuve du théorème de Higman. On dit qu'une suite $(t_n)_{n \geq 0}$ est *mauvaise* s'il n'existe pas deux indices $k < l$ tels que $t_k \preceq^\Delta t_l$ et on utilise la caractérisation iv) du théorème 1.26. On raisonne par l'absurde et on suppose qu'il existe au moins une mauvaise suite.

On construit alors par récurrence une mauvaise suite minimale $(t_n)_{n \geq 0}$. Soit t_0 un arbre de taille minimale tel que t_0 soit le premier élément d'une mauvaise suite. Les arbres $t_0, \dots, t_n$ étant choisis, t_{n+1} est un arbre de taille minimale tel que $t_0, \dots, t_{n+1}$ sont les premiers éléments d'une mauvaise suite. La suite ainsi construite est mauvaise.

Il existe un nombre fini d'indices n tels que $|t_n| \leq 2$. Sinon il existe une sous-suite infinie d'arbres de taille 1 et on peut trouver deux indices $k < l$ tels que $t_k(\varepsilon) \preceq t_l(\varepsilon)$ qui est équivalent à $t_k \preceq^\Delta t_l$ si t_k et t_l sont de taille 1. Quitte à supprimer un nombre fini d'éléments de la suite, on peut supposer que $|t_n| \geq 2$ pour tout $n \geq 0$.

Puisque $\preceq$ est un bon quasi-ordre sur A , il existe une suite extraite $(t_{i_n})_{n \geq 0}$ telle que $t_{i_0}(\varepsilon) \preceq t_{i_1}(\varepsilon) \preceq t_{i_2}(\varepsilon) \preceq \dots$. Quitte à remplacer la suite $(t_n)_{n \geq 0}$ par la suite extraite $(t_{i_n})_{n \geq 0}$, on peut supposer que la suite $(t_n)_{n \geq 0}$ vérifie $t_0(\varepsilon) \preceq t_1(\varepsilon) \preceq t_2(\varepsilon) \preceq \dots$.

Soit $\mathcal{D}$ l'ensemble des arbres qui apparaissent juste sous la racine d'un des arbres t_n . Plus formellement, on définit l'ensemble $\mathcal{D}$ par

$$\mathcal{D} = \{i^{-1}t_n \mid i \in \mathbb{N} \text{ et } i \in \text{dom}(t_n)\}.$$

On prétend que $\preceq^\Delta$ est un bon quasi-ordre sur $\mathcal{D}$. Sinon, il existe une mauvaise suite $(r_n)_{n \geq 0}$ d'arbres de $\mathcal{D}$. On peut extraire de cette suite une nouvelle suite $(r'_n)_{n \geq 0}$ telle que, pour tous $k < l$, les arbres r'_k et r'_l sont des sous-arbres de t_m et t_n avec $m < n$. On peut donc supposer que la suite $(r_n)_{n \geq 0}$ satisfait cette propriété. Soit alors n_0 le plus petit indice tel que r_0 est un sous-arbre de t_{n_0} . On considère la suite

$$t_0, t_1, \dots, t_{n_0-1}, r_0, r_1, r_2, \dots$$

et on constate que cette suite est encore mauvaise et qu'elle contredit la minimalité de la suite $(t_n)_{n \geq 0}$ construite. Cette contradiction prouve que $\preceq^\Delta$ est bien un bon quasi-ordre sur $\mathcal{D}$. Par le théorème de Higman, le quasi-ordre $\preceq^{\Delta*}$ est un bon quasi-ordre sur $\mathcal{D}^*$.

Pour chaque arbre t_n , on note k_n l'entier $\max\{i \mid i \in \text{dom}(t_n)\}$. Les sous-arbres sous la racine de t_n sont donc les arbres $0^{-1}t_n, 1^{-1}t_n, \dots, k_n^{-1}t_n$. Puisque $\preceq^{\Delta*}$ est un bon quasi-ordre sur $\mathcal{D}^*$, il existe deux indices $m < n$ tels que

$$(0^{-1}t_m, 1^{-1}t_m, \dots, k_m^{-1}t_m) \preceq^{\Delta*} (0^{-1}t_n, 1^{-1}t_n, \dots, k_n^{-1}t_n)$$

Puisqu'on a en outre $t_m(\varepsilon) \preceq t_n(\varepsilon)$, on a finalement $t_m \preceq^\Delta t_n$ qui prouve que $\preceq^\Delta$ est un bon quasi-ordre sur l'ensemble de tous les arbres finis. $\square$

Les théorèmes de Higman et de Kruskal peuvent encore être généralisés aux graphes. Le théorème de Robertson et Seymour établit que l'ordre par mineur des graphes est un bon quasi-ordre. On rappelle qu'un *mineur* d'un graphe G est un graphe obtenu par contraction d'arêtes d'un sous-graphe de G . Un *sous-graphe* est le graphe induit par un sous-ensemble de sommets. La contraction d'une arête consiste à identifier les deux sommets qu'elle relie. Beaucoup de classes de graphes comme celle des graphes planaires ou celle des graphes triangulés sont caractérisées par des mineurs interdits. Elles sont égales à l'ensemble des graphes qui ne contiennent pas certains graphes fixés comme mineur. La preuve du résultat de Robertson et Seymour est excessivement difficile et s'étend sur plusieurs centaines de pages.

L'exercice suivant introduit une extension du théorème de Ramsey (théorème 1.98) aux arbres.

Exercice 1.37. Pour $n \geq 0$, on note B_n l'ensemble $\{0, 1\}^{\leq n}$ des mots de longueur au plus n sur l'alphabet $\{0, 1\}$. Un arbre t est dit *monochrome* si pour tous $w, w' \in \text{dom}(t)$, on a $t(w) = t(w')$. Montrer que pour tout arbre t de domaine B_{2n} sur l'alphabet $A = \{a, b\}$, il existe un arbre t' monochrome de domaine B_n tel que $t' \preceq^\Delta t$ où l'ordre $\preceq$ sur A est l'égalité.

1.5 Langages rationnels

La classe des langages rationnels est le premier niveau de la hiérarchie de Chomsky. Ces langages sont très simples mais ils possèdent de très nombreuses propriétés remarquables. Ils peuvent être introduits par des définitions de natures très différentes. Ceci est une preuve de leur rôle central en théorie des langages formels.

1.5.1 Expressions rationnelles

On commence par la définition de ces langages qui utilise les opérations rationnelles et justifie la terminologie.

Définition 1.38 (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 $\{a\} \in \mathcal{R}$ pour toute lettre a ;
- $\mathcal{R}$ est close pour les opérations rationnelles (l'union, le produit et l'étoile).

Exemple 1.39. Quelques exemples de langages rationnels.

- Le langage $\{\varepsilon\}$ est rationnel car il s'écrit $\emptyset^*$.
- Le langage A est rationnel puisqu'il s'écrit $A = \bigcup_{a \in A} \{a\}$.
- Le langage L des mots de longueur paire est rationnel puisqu'il s'écrit $L = (AA)^* = (A^2)^*$.
- Le langage L' des mots de longueur impaire est rationnel puisqu'il s'écrit $L' = AL$.
- Le langage des mots qui contiennent un facteur aba est rationnel puisqu'il s'écrit A^*abaA^* .

La notion d'*expression* n'est pas formellement définie. Elle doit être entendue dans le sens usuel en mathématiques comme c'est le cas des expressions arithmétiques. Les opérateurs arithmétiques sont remplacés dans notre cadre par les opérateurs rationnels et les parenthèses sont encore utilisées pour lever les ambiguïtés.

Définition 1.40 (Expressions rationnelles). La classe $\mathcal{E}$ des *expressions rationnelles* est la plus petite famille d'expressions telles que :

- $\emptyset \in \mathcal{E}$, et $a \in \mathcal{E}$ pour toute lettre a ;
- pour toutes expressions E et E' de $\mathcal{E}$, les expressions $E + E'$, $E \cdot E'$ et E^* sont encore dans $\mathcal{E}$.

Notons que $+$, $\cdot$ et $*$ sont vus ici des symboles inertes et non des opérations.

Notation 1.41. Pour alléger les notations, Le point dénotant le produit et les accolades autour des singletons sont omis dans l'écriture des expressions rationnelles. Ainsi l'expression $(\{a\} + \{b\} \cdot \{a\})^*$ est écrite $(a + ba)^*$. De plus, Si $A = \{a_1, \dots, a_n\}$, on utilise A comme une abréviation pour $a_1 + \dots + a_n$.

Exemple 1.42. Quelques exemples d'expressions rationnelles.

A^*	tous les mots
aA^*	mots commençant par a
A^*a	mots finissant par a
$(b + ab)^*(a + \varepsilon)$	mots n'ayant pas deux a consécutifs
$a^* + b^*$	mots n'ayant que des a ou que des b
$(aa + b)^*$	mots avec des blocs de a de longueur paire
$(ab^*a + b)^*$	mots ayant un nombre pair de a

1.5.2 Automates

Une autre définition des langages rationnels peut être donnée en utilisant les automates. Il s'agit d'un type de machines très simples qui sont des cas particuliers des machines de Turing.

Définition 1.43 (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 1.44. Une transition (p, a, q) est notée $p \xrightarrow{a} q$.

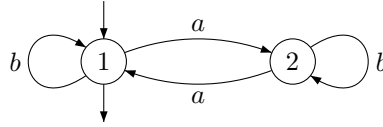


FIG. 1.6 – Un automate avec quatre transitions

Exemple 1.45. Soit $\mathcal{A} = (\{1, 2\}, \{a, b\}, \{(1, b, 1), (1, a, 2), (2, b, 2), (2, a, 1)\}, \{1\}, \{1\})$ un automate représenté à la figure 1.6. Cet automate a les deux états 1 et 2. L'état 1 est à la fois initial (marqué d'une petite flèche entrante) et final (marqué d'une petite flèche sortante). Il possède quatre transitions représentées comme des arêtes d'un graphe.

Définition 1.46 (Chemin). Un *chemin* dans un automate (Q, A, E, I, F) est une suite finie de transitions consécutives

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$$

noté aussi de manière concise $q_0 \xrightarrow{w} q_n$. L'état q_0 est l'*état de départ* et q_n est l'*état d'arrivée* du chemin. Le mot $a_1 \dots a_n$ est l'*étiquette* du chemin.

Exemple 1.47. 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. Son étiquette est le mot *abbab*.

Définition 1.48 (Acceptation). Un chemin est *acceptant* ou *réussi* lorsque l'état de départ est initial et l'état d'arrivée est final. Un mot est *accepté* par l'automate $\mathcal{A}$ s'il est l'étiquette d'un chemin acceptant de $\mathcal{A}$. Le langage des mots acceptés par l'automate $\mathcal{A}$ est noté $L(\mathcal{A})$.

Exemple 1.49. L'ensemble des mots acceptés par l'automate de la figure 1.6 est le langage $(ab^*a + b)^*$ des mots ayant un nombre pair d'occurrences de a .

Le théorème de Kleene établit l'équivalence entre les expressions rationnelles et les automates finis dans le sens où ces deux notions définissent les mêmes langages. Ce résultat est remarquable car il relie deux notions de natures différentes, combinatoire pour les expressions rationnelles et opérationnelle pour les automates. Le théorème de Kleene admet de nombreuses extensions à des structures telles les arbres et les mots infinis ou même transfinis.

Théorème 1.50 (Kleene 1956). *Un langage L est rationnel si et seulement s'il existe un automate (fini) $\mathcal{A}$ tel que $L = L(\mathcal{A})$.*

La preuve du théorème utilise les notions d'automate émondé et normalisé, que nous définissons successivement.

Définition 1.51 (Automate émondé). Un automate est *émondé* si par tout état passe au moins un chemin acceptant.

Il est clair que les états par lesquels ne passe aucun chemin acceptant peuvent être supprimés sans changer l'ensemble des mots acceptés par l'automate. Un état q apparaît sur un chemin acceptant s'il est accessible d'un état initial et si un état final est accessible à partir de q (on dit que q est co-accessible). L'ensemble des états accessibles et co-accessibles peut être calculé par deux parcours en largeur de l'automate considéré comme un graphe. On suppose souvent dans la suite que les automates sont émondés.

Définition 1.52 (Automate normalisé). Un automate est *normalisé* s'il possède un unique état initial qui est l'état d'arrivée d'aucune transition et un unique état final qui est l'état de départ d'aucune transition.

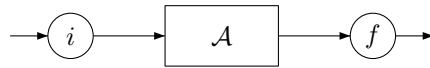


FIG. 1.7 – Schéma d'un automate normalisé

Si l'état initial et l'état final d'un automate normalisé coïncident, aucune transition n'est adjacente à cet état et l'automate accepte uniquement le mot vide. Sinon, l'automate n'accepte pas le mot vide. La proposition suivante n'est pas nécessaire à la preuve du théorème mais la construction utilisée dans la preuve est intéressante en soi.

Proposition 1.53 (Normalisation). *Pour tout automate $\mathcal{A}$, il existe un automate normalisé $\mathcal{A}'$ tel que $L(\mathcal{A}') = L(\mathcal{A}) \setminus \{\varepsilon\}$.*

Preuve. Soit $\mathcal{A} = (Q, A, E, I, F)$ un automate. Soient i et f deux nouveaux états n'appartenant pas à Q . L'automate $\mathcal{A}'$ est égal à $(Q \cup \{i, f\}, A, E', \{i\}, \{f\})$ où

l'ensemble E' des transitions est donné par

$$\begin{aligned} E' = E \cup \{ & i \xrightarrow{a} q \mid \exists p \in I \ p \xrightarrow{a} q \in E \} \\ & \cup \{ p \xrightarrow{a} f \mid \exists q \in F \ p \xrightarrow{a} q \in E \} \\ & \cup \{ i \xrightarrow{a} f \mid \exists p \in I \ \exists q \in F \ p \xrightarrow{a} q \in E \}. \end{aligned}$$

C'est pure routine de vérifier que $L(\mathcal{A}') = L(\mathcal{A}) \setminus \{\varepsilon\}$. $\square$

Preuve. Pour un langage L , on note respectivement $\varepsilon(L)$ et $\alpha(L)$ les langages $L \cap \{\varepsilon\}$ et $L \setminus \{\varepsilon\}$. Les formules suivantes montrent qu'on peut calculer récursivement $\alpha(L)$ et $\varepsilon(L)$ à partir d'une expression rationnelle de L .

$$\begin{aligned} \varepsilon(L + L') &= \varepsilon(L) + \varepsilon(L') & \alpha(L + L') &= \alpha(L) + \alpha(L') \\ \varepsilon(LL') &= \varepsilon(L)\varepsilon(L') & \alpha(LL') &= \alpha(L)\alpha(L') + \varepsilon(L)\alpha(L') + \alpha(L)\varepsilon(L') \\ \varepsilon(L^*) &= \varepsilon & \alpha(L^*) &= \alpha(L)^+ \end{aligned}$$

On montre par induction sur (la longueur de) l'expression rationnelle que si L est rationnel alors $\alpha(L)$ est accepté par un automate normalisé. Pour obtenir un automate acceptant L , il suffit d'ajouter éventuellement un nouvel état à la fois initial et final pour accepter le mot vide.

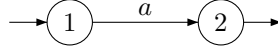


FIG. 1.8 – Automate normalisé acceptant $L = a$

On considère d'abord les cas de base. Le langage $L = a$ est accepté par l'automate de la figure 1.8

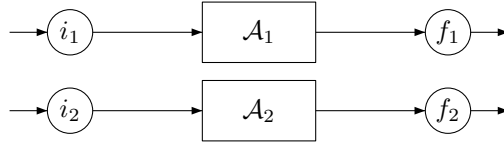


FIG. 1.9 – Automates normalisés $\mathcal{A}_1$ et $\mathcal{A}_2$

Soient L_1 et L_2 deux langages rationnels et soient $\mathcal{A}_1$ et $\mathcal{A}_2$ deux automates normalisés acceptant respectivement les langages $\alpha(L_1)$ et $\alpha(L_2)$. Ces deux automates sont schématisés à la figure 1.9

Si $L = L_1 + L_2$, on utilise la formule $\alpha(L) = \alpha(L_1) + \alpha(L_2)$. Un automate acceptant $\alpha(L)$ est construit en faisant l'union disjointe des deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$. Cet automate est ensuite normalisé en fusionnant i_1 avec i_2 et f_1 avec f_2 (cf. figure 1.10).

Si $L = L_1 L_2$, on utilise la formule $\alpha(L) = \alpha(L_1)\alpha(L_2) + \varepsilon(L_1)\alpha(L_2) + \alpha(L_1)\varepsilon(L_2)$. D'après le cas précédent, on sait construire un automate normalisé pour l'union. Il suffit donc de savoir construire un automate pour $\alpha(L_1)\alpha(L_2)$. Un automate normalisé acceptant $\alpha(L_1)\alpha(L_2)$ est construit en faisant l'union

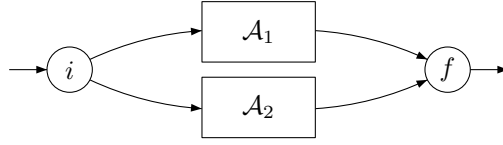


FIG. 1.10 – Automate pour l'union



FIG. 1.11 – Automate pour le produit

disjointe des deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$ puis en fusionnant f_1 avec i_2 (cf. figure 1.11).

Si $L = L_1^*$, on utilise la formule $\alpha(L) = (\alpha(L_1))^+$. Un automate normalisé acceptant $(\alpha(L_1))^+$ est construit en dupliquant chacun des états i_1 et f_1 en des états i'_1 et f'_1 puis en fusionnant i'_1 avec f'_1 (cf. figure 1.12). Cet automate peut aussi être obtenu en fusionnant d'abord i_1 avec f_1 puis en normalisant ensuite l'automate grâce à la proposition précédente.

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

Il existe de nombreuses autres méthodes pour convertir une expression rationnelle en automate. Une variante de la méthode présentée ci-dessus consiste à utiliser des automates *semi-normalisés* où seules les conditions sur l'état initial sont satisfaites. Les constructions pour l'automate du produit et de l'étoile sont un peu plus délicates. Par contre, ces automates peuvent accepter le mot vide et les constructions limitent le nombre d'états engendrés. Cette méthode est pratique pour des constructions à la main. La méthode de Thompson utilise des automates avec ε -transitions qu'il faut ensuite supprimer. D'autres méthodes sont basées sur les quotients à gauche calculés directement sur l'expression. On peut obtenir un automate déterministe par la méthode Brzozowski ou un automate non déterministe par celle d'Antimirov (cf. [Sak03]).

Algorithme (McNaughton-Yamada).

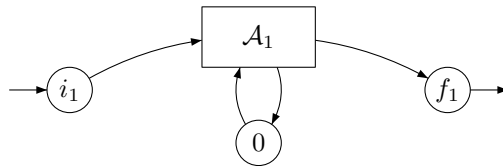


FIG. 1.12 – Automate pour l'étoile

On suppose $Q = \{1, \dots, n\}$. Pour $0 \leq k \leq n$, et deux états $q, q' \in Q$, on définit l'ensemble $L_{q,q'}^k$ de mots qui étiquettent un chemin de q à q' ne passant que par des états intermédiaires dans l'ensemble $\{1, \dots, k\}$. Plus formellement, l'ensemble $L_{q,q'}^k$ est défini par

$$L_{q,q'}^k = \{a_1 \dots a_n \mid n \geq 0 \text{ et } q \xrightarrow{a_1} p_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q' \text{ avec } \forall i, p_i \in \{1, 2, \dots, k\}\}.$$

Ainsi, on a : $L(\mathcal{A}) = \bigcup_{i \in I, f \in F} L_{i,f}^n$. Les formules suivantes permettent de calculer les langages $L_{q,q'}^k$ par récurrence sur k .

$$\begin{aligned} L_{q,q'}^0 &= \{a \mid q \xrightarrow{a} q' \in E\} \cup \{\varepsilon \mid \text{si } q = q'\} \\ L_{q,q'}^{k+1} &= L_{q,q'}^k + L_{q,k+1}^k (L_{k+1,k+1}^k)^* L_{k+1,q'}^k. \end{aligned}$$

Comme les formules ci-dessus ne font intervenir que les opérations rationnelles, on montre facilement par récurrence sur k que tous les langages $L_{q,q'}^k$ et donc aussi $L(\mathcal{A})$ sont rationnels.

Algorithme (Méthode par élimination).

Cet algorithme manipule des automates dont les transitions sont étiquetées par des expressions rationnelles. La première étape consiste à normaliser l'automate en utilisant la proposition 1.53. La seconde étape remplace pour chaque paire d'états (p, q) , toutes les transitions $p \xrightarrow{a} q$ par une seule transition étiquetée par l'expression rationnelle $\sum_{(p,a,q) \in E} a$. À chaque étape suivante, un des états autre que l'état initial et l'état final est supprimé. L'algorithme s'arrête lorsqu'il ne reste plus que l'état initial et l'état final ainsi qu'une seule transition entre les deux. Le résultat est alors l'expression rationnelle qui étiquette cette transition.

À chaque suppression d'un état s on ajoute, pour chaque paire (p, q) d'états restants, à l'expression portée par la transition de p à q l'expression xy^*z où x , y et z sont les expressions rationnelles portées respectivement par les transitions de p à s , de s à s et de s à q .

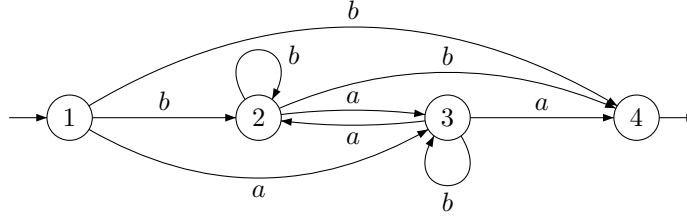


FIG. 1.13 – Méthode par élimination : normalisation

Exemple 1.54. Si on applique la méthode par élimination à l'automate de la figure 1.6, on obtient successivement les automates des figures 1.13, 1.14 et 1.15.

Lemme 1.55 (Lemme d'Arden). *Soient K et L deux langages et soit l'équation $X = KX + L$ en le langage X .*

- si $\varepsilon \notin K$, l'unique solution de l'équation est $X = K^*L$.
- si $\varepsilon \in K$, les solutions sont de la forme $X = K^*(L + Y)$ où $Y \subseteq A^*$.

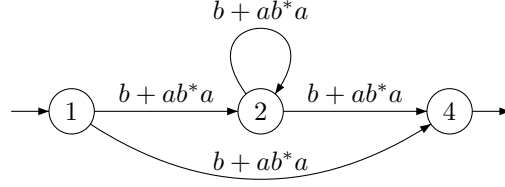


FIG. 1.14 – Méthode par élimination : suppression de 3

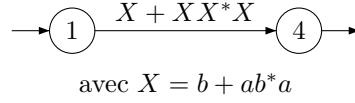


FIG. 1.15 – Méthode par élimination : suppression de 2

Preuve. Soit X une solution de l'équation $X = KX + L$. Si $Y \subset X$, on montre par récurrence sur n que $K^n Y \subset X$ et donc $K^* Y \subset X$. En considérant $Y = L$, on constate que $K^* L \subset X$. Réciproquement, l'égalité $K = KK^* + \varepsilon$ implique que $K^* L$ est une solution de l'équation. Ceci démontre que $K^* L$ est la plus petite (pour l'inclusion) solution de l'équation. Ce résultat est en fait un cas particulier de la proposition 2.20 concernant les solutions d'un système polynomial.

- Si $\varepsilon \notin K$, il reste à montrer que $K^* L$ est en fait l'unique solution de l'équation. Montrons par l'absurde que $X = K^* L$ en supposant que $X \setminus K^* L$ est non vide.
Soit w un mot de $X \setminus K^* L$ de longueur minimale. Comme $w \notin L$, ce mot s'écrit $w = kx$ avec $k \in K$ et $x \in X$. Comme $k \neq \varepsilon$, le mot x est de longueur strictement inférieure à w . La minimalité de $|w|$ implique que x appartient à $K^* L$. Ceci aboutit à la contradiction que $w \in K^* L$.
- Si $\varepsilon \in K$ la solution X s'écrit $X = K^* L + Y$. Comme $Y \subset X$, on a aussi $K^* Y \subset X$ et donc l'égalité $X = K^* (L + Y)$. Comme $KK^* = K^*$, tout langage de la forme $K^* (L + Y)$ est solution de l'équation.

□

Algorithme (Méthode de Gauss).

Pour chaque $q \in Q$, soit X_q l'ensemble des mots qui étiquettent 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 :

$$X_p = \begin{cases} \sum_{(p,a,q) \in E} aX_q + \varepsilon & \text{si } p \text{ est final} \\ \sum_{(p,a,p) \in E} aX_q & \text{sinon} \end{cases} \quad \text{pour } p \in Q.$$

Exemple 1.56. Pour l'automate de la figure 1.6, on obtient le système

$$\begin{aligned} X_1 &= bX_1 + aX_2 + \varepsilon \\ X_2 &= aX_1 + bX_2 \end{aligned}$$

En utilisant le lemme d'Arden sur la seconde équation, on obtient $X_2 = b^* a X_1$. En substituant dans la première équation, on obtient $X_1 = (ab^* a + b) X_1 + \varepsilon$ qui donne finalement $L(\mathcal{A}) = X_1 = (ab^* a + b)^*$ grâce au lemme d'Arden.

1.6 Automates déterministes

La notion de machine ou de modèle de calcul déterministe est fondamentale et elle apparaît tout au long de ce cours. De manière intuitive, une machine est déterministe si pour chaque entrée, un seul calcul est possible. Pour certains modèles comme les automates ou les machines de Turing, toute machine est équivalente à une machine déterministe. Par contre, le passage à une machine déterministe a un prix qui est le nombre d'états pour les automates (cf. exemples 1.62 et 1.63) ou le temps de calcul pour les machines de Turing. Pour d'autres modèles comme les automates à pile, les machines déterministes sont moins puissantes.

Une propriété essentielle des automates finis est que tout automate est équivalent à un automate déterministe qui accepte le même langage. Cette propriété a un intérêt aussi bien théorique que pratique. D'un point de vue théorique, elle permet de montrer la clôture par complémentation des langages rationnels. D'un point de vue pratique, les automates déterministes sont plus faciles à implémenter. La commande UNIX **grep** utilise par exemple un automate déterministe pour rechercher une occurrence d'une expression rationnelle dans un texte. En fait, la commande **grep** calcule d'abord un automate non déterministe puis le détermine de manière paresseuse. Elle ne calcule que les états vraiment parcourus par la lecture du texte dans l'automate.

De manière intuitive, un automate est déterministe si une seule transition est possible à chaque instant.

Définition 1.57 (Automate déterministe). Un automate $\mathcal{A} = (Q, A, E, I, F)$ est *déterministe* si :

- il a un unique état initial : $|I| = 1$;
- si (p, a, q) et $(p, a, q') \in E$ alors $q = q'$.

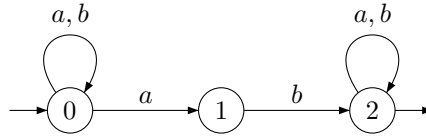


FIG. 1.16 – Automate non déterministe

Exemple 1.58. L'automate de la figure 1.6 (p. 26) est déterministe. Par contre, l'automate de la figure 1.16 n'est pas déterministe. Il accepte le langage A^*abA^* des mots qui contiennent au moins un facteur ab .

Proposition 1.59. *Tout automate est équivalent à (accepte le même langage que) un automate déterministe.*

Preuve. Soit $\mathcal{A} = (Q, A, E, I, F)$ un automate donné. On construit un automate déterministe équivalent $\hat{\mathcal{A}}$ dont les états sont les parties de Q .

$$\hat{\mathcal{A}} = (\mathfrak{P}(Q), A, \hat{E}, \{I\}, \{P \subseteq Q \mid P \cap F \neq \emptyset\})$$

avec $\hat{E} = \{P \xrightarrow{a} P' \mid P' = \{q \mid \exists p \in P \ p \xrightarrow{a} q\}\}.$

Ainsi, $\hat{\mathcal{A}}$ est déterministe et accepte le même langage que $\mathcal{A}$ comme le montre le lemme suivant. $\square$

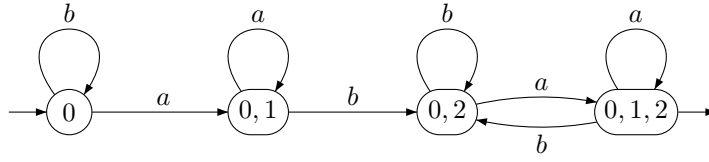


FIG. 1.17 – Déterminisation de l'automate de la figure 1.16

Exemple 1.60. En appliquant la construction utilisée dans la preuve de la proposition 1.59 à l'automate de la figure 1.16, on obtient l'automate de la figure 1.17

Le fait que l'automate $\hat{\mathcal{A}}$ soit équivalent à $\mathcal{A}$ découle directement du lemme suivant.

Lemme 1.61. *Pour tout $w \in A^*$, il existe un chemin de I à P dans $\hat{\mathcal{A}}$ étiqueté par w si et seulement si $P = \{q \mid \exists i \in I \ i \xrightarrow{w} q \text{ dans } \mathcal{A}\}$*

La preuve du lemme se fait par récurrence sur la longueur de w .

Un des problèmes de la construction par sous-ensembles utilisée dans la preuve de la proposition 1.59 est l'explosion du nombre d'états. Si l'automate $\mathcal{A}$ a n états, l'automate déterministe $\hat{\mathcal{A}}$ équivalent peut avoir jusqu'à 2^n états comme le montrent les exemples suivants.

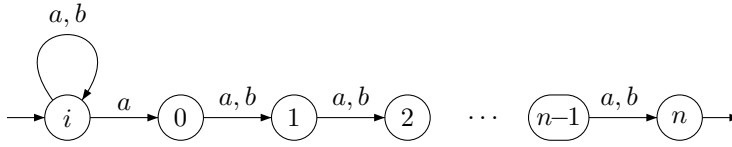


FIG. 1.18 – Automate non déterministe pour A^*aA^n

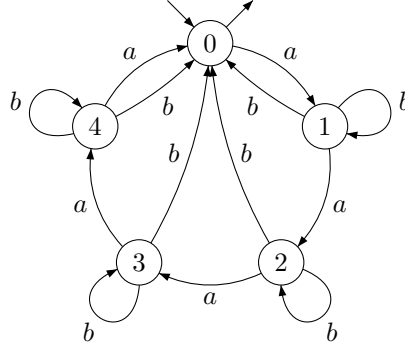
Exemple 1.62. Soient $A = \{a, b\}$ et n un entier. Le langage A^*aA^n est accepté par un automate non déterministe ayant $n + 2$ états. Par contre tout automate déterministe acceptant ce langage a au moins 2^n états.

Exemple 1.63. Soient $A = \{a, b\}$, n un entier et $Q_n = \{0, \dots, n-1\}$. On considère l'automate $\mathcal{A}_n = (Q_n, A, E_n, \{0\}, \{0\})$ où l'ensemble des transitions est donné par

$$E_n = \{i \xrightarrow{a} i+1 \mid 0 \leq i \leq n-2\} \cup \{n-1 \xrightarrow{a} 0\} \\ \cup \{i \xrightarrow{b} i \mid 1 \leq i \leq n-1\} \cup \{i \xrightarrow{b} 0 \mid 1 \leq i \leq n-1\}.$$

Tout automate déterministe et complet qui est équivalent à $\mathcal{A}_n$ possède au moins 2^n états.

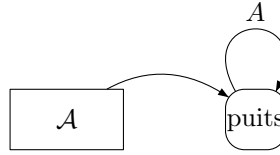
Définition 1.64 (Automate complet). Un automate est *complet* si pour tout paire (p, a) de $Q \times A$, il existe un état q tel que $p \xrightarrow{a} q$ soit une transition.

FIG. 1.19 – Automate $\mathcal{A}_5$ de l'exemple 1.63

Notation 1.65. Dans un automate déterministe complet $\mathcal{A} = (Q, A, E, I, F)$, on notera pour tout état $q \in Q$ et toute lettre $a \in A$, l'état $q \cdot a$ l'unique état p tel que $q \xrightarrow{a} p$ soit une transition de $\mathcal{A}$.

Proposition 1.66. *Tout automate (déterministe) est équivalent à un automate (déterministe) complet.*

Preuve. On introduit un nouvel état $p \notin Q$ appelé *puits* et on pose $Q' = Q \cup \{p\}$. On note $R = \{(q, a) \mid q \in Q' \text{ et } \forall q' \ q \xrightarrow{a} q' \notin E\}$ l'ensemble des transitions manquantes. L'automate complété est l'automate $\mathcal{A}'$ défini par $\mathcal{A}' = (Q', A, E', I, F)$, avec $E' = E \cup \{q \xrightarrow{a} p \mid (q, a) \in R\}$. Notons que si $\mathcal{A}$ est déterministe, l'automate complété $\mathcal{A}'$ l'est également. $\square$

FIG. 1.20 – Complétion par ajout d'un état *puits*

Corollaire 1.67 (Clôture par complémentation). *Si $L \subseteq A^*$ est rationnel, alors $A^* \setminus L$ est rationnel.*

Preuve. Soit $\mathcal{A} = (Q, A, E, I, F)$ un automate déterministe et complet acceptant le langage L . L'automate $\mathcal{A}' = (Q, A, E, I, Q \setminus F)$ accepte le langage $L(\mathcal{A}') = A^* \setminus L$. $\square$

1.7 Automate minimal

Dans cette partie, on montre que tout langage rationnel est accepté par un automate minimal qui est le quotient de tout automate déterministe acceptant ce langage. On étudie ensuite quelques algorithmes permettant de calculer cet automate minimal. Une très bonne référence pour cette partie est [BBC93].

1.7.1 Quotients

Les quotients à gauche constituent un outil indispensable à l'étude des automates déterministes acceptant un langage L puisque le nombre de quotients donne un minorant du nombre d'états. Ils fournissent également une caractérisation très utile des langages rationnels (Proposition 1.72). Celle-ci est, par exemple, utilisée de manière cruciale dans la preuve du théorème 1.97 (p. 43).

Définition 1.68 (Quotients à gauche). Soit $L \subseteq A^*$ un langage. Le *quotient à gauche* de L par un mot $u \in A^*$ est le langage $u^{-1}L = \{v \mid uv \in L\}$. Le *quotient à gauche* de L par un langage $K \subseteq A^*$ est $K^{-1}L = \bigcup_{k \in K} k^{-1}L$.

De manière symétrique, on peut aussi définir les *quotients à droite* d'un langage. L'intérêt des quotients à gauche réside dans leurs liens avec les automates déterministes.

Exemple 1.69. Soit $L = (ab^*a + b)^*$ le langage des mots ayant un nombre pair d'occurrences de a . On a les égalités $a^{-1}L = b^*aL$ et $b^{-1}L = L$ où b^*aL est le langage des mots ayant un nombre impair d'occurrences de a .

Les formules suivantes permettent de calculer explicitement un quotient $a^{-1}L$ puis un quotient $w^{-1}L$ en procédant par récurrence sur la longueur de w . On parvient ainsi à calculer tous les quotients à gauche d'un langage.

Proposition 1.70. Pour toute lettre a , tous mots u, v et w et tous langages K et L , on a les relations suivantes qui permettent de calculer les quotients à gauche.

- $w^{-1}(K + L) = w^{-1}K + w^{-1}L$
- $a^{-1}(KL) = (a^{-1}K)L + \varepsilon(K)a^{-1}L$
- $w^{-1}(KL) = (w^{-1}K)L + \sum_{uv=w} \varepsilon(u^{-1}K)v^{-1}L$ en notant $\varepsilon(L) = L \cap \varepsilon$.
- $a^{-1}(L^*) = (a^{-1}L)L^*$
- $w^{-1}L^* = \sum_{uv=w} \varepsilon(u^{-1}L^*)(v^{-1}L)L^*$
- $(uv)^{-1}L = v^{-1}(u^{-1}L)$.

Le lemme suivant établit un lien fondamental entre les quotients à gauche d'un langage et les automates déterministes acceptant ce même langage.

Lemme 1.71. Soit $\mathcal{A} = (Q, A, E, \{i\}, F)$ un automate déterministe acceptant un langage L . Pour tout chemin $i \xrightarrow{u} q$, on a $u^{-1}L = \{v \mid q \xrightarrow{v} f \text{ avec } f \in F\}$.

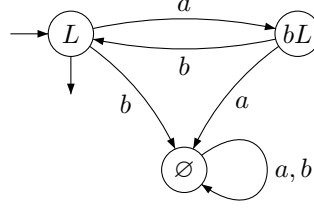
La preuve du lemme est triviale. Le lemme a pour corollaire immédiat que le nombre de quotients à gauche d'un langage est inférieur au nombre d'états de tout automate déterministe et complet acceptant ce langage. En particulier, le nombre de quotients à gauche d'un langage rationnel est fini. Cette propriété est en fait caractéristique des langages rationnels.

Proposition 1.72. Un langage L est rationnel si et seulement si il a un nombre fini de quotients à gauche.

Pour la preuve de la proposition, on introduit la définition suivante.

Définition 1.73. Soit L un langage rationnel. L'*automate minimal* de L est automate $\mathcal{A}_L = (Q, A, E, I, F)$ où

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

FIG. 1.21 – Automate minimal de $L = (ab)^*$

Exemple 1.74. Les différents quotients à gauche du langage $L = (ab)^*$ sont les langages $a^{-1}L = bL$, $b^{-1}L = \emptyset$, $a^{-1}(bL) = \emptyset$ et $b^{-1}(bL) = L$. En appliquant la construction donnée dans la preuve de la proposition précédente, on obtient l'automate minimal de L de la figure 1.21.

Le nombre d'états de l'automate minimal est bien sûr minimal puisqu'il est égal au nombre de quotients à gauche. La proposition découle alors directement du lemme suivant.

Lemme 1.75. *L'automate minimal $\mathcal{A}_L$ accepte le langage L .*

Preuve. On montre facilement par récurrence sur la longueur que pour tout mot u , on a un chemin $L \xrightarrow{u} u^{-1}L$ dans l'automate $\mathcal{A}_L$. La définition des états finaux donne immédiatement le résultat. $\square$

1.7.2 Congruence de Nerode

On montre dans cette partie que l'automate minimal a en outre la propriété d'être le quotient de tout autre automate déterministe acceptant le même langage. Il peut toujours être obtenu en fusionnant des états.

Pour toute paire $(p, a) \in Q \times A$ d'un automate déterministe, on note s'il existe $q \cdot a$ l'unique état q tel que $p \xrightarrow{a} q$ soit une transition. Si de plus l'automate est complet, $q \cdot a$ est toujours défini. Cette notation peut être étendue à tous les mots en définissant par récurrence $p \cdot (ua) = (p \cdot u) \cdot a$. L'état $q \cdot w$ est alors l'unique état q tel que $p \xrightarrow{w} q$ soit un chemin.

Définition 1.76 (Congruence). Soit $\mathcal{A} = (Q, A, E, \{i\}, F)$ un automate déterministe et complet. Une *congruence* sur $\mathcal{A}$ est une relation d'équivalence $\sim$ sur Q qui vérifie pour tous $q, q' \in Q$ et $a \in A$:

$$\begin{aligned} q \sim q' &\implies (q \in F \iff q' \in F), \\ q \sim q' &\implies q \cdot a \sim q' \cdot a. \end{aligned}$$

La première propriété signifie que chaque classe d'une congruence ne contient que des états finaux ou que des états qui ne sont pas finaux. La seconde propriété est qu'une congruence est compatible avec les transitions de l'automate. Rien dans cette définition ne concerne l'état initial.

Si $\sim$ est une congruence sur un automate $\mathcal{A} = (Q, A, E, \{i\}, F)$, l'automate quotient $\mathcal{A}/\sim$ est l'automate $(Q/\sim, A, E', \{[i]\}, \{[f] \mid f \in F\})$ où l'ensemble E' des transitions est donné par

$$E' = \{[q] \xrightarrow{a} [q \cdot a] \mid q \in Q \text{ et } a \in A\}.$$

Cet automate est encore déterministe car la classe $[q \cdot a]$ ne dépend que de la classe de q . Le lemme suivant montre qu'un automate quotient accepte le même langage.

Lemme 1.77. *Soit $\sim$ une congruence sur un automate $\mathcal{A}$, alors l'automate quotient $\mathcal{A}/\sim$ accepte le langage $L(\mathcal{A})$.*

Preuve. Soit un chemin

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$$

d'étiquette $w = a_1 \dots a_n$ dans l'automate $\mathcal{A}$. On en déduit que

$$[q_0] \xrightarrow{a_1} [q_1] \xrightarrow{a_2} \dots \xrightarrow{a_n} [q_n]$$

est un chemin étiqueté par w dans $\mathcal{A}/\sim$. Si de plus q_0 est initial et q_n est final dans $\mathcal{A}$, alors $[q_0]$ est initial et $[q_n]$ est final dans $\mathcal{A}/\sim$. Ceci prouve l'inclusion $L(\mathcal{A}) \subseteq L(\mathcal{A}/\sim)$.

Soit maintenant un chemin

$$[q_0] \xrightarrow{a_1} [q_1] \xrightarrow{a_2} \dots \xrightarrow{a_n} [q_n]$$

d'étiquette $w = a_1 \dots a_n$ dans l'automate $\mathcal{A}/\sim$. On montre par récurrence sur n que pour tout état q'_0 de la classe $[q_0]$, il existe un chemin

$$q'_0 \xrightarrow{a_1} q'_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q'_n$$

d'étiquette w tel que $q'_i \sim q_i$ pour tout $0 \leq i \leq n$. Si le chemin dans $\mathcal{A}/\sim$ est acceptant, on peut choisir $q'_0 = i$ et l'état q'_n est alors final. Ceci montre l'inclusion $L(\mathcal{A}/\sim) \subseteq L(\mathcal{A})$. $\square$

Définition 1.78 (Congruence de Nerode). Soit $\mathcal{A} = (Q, A, E, \{i\}, F)$ un automate déterministe et complet. La congruence de Nerode est définie pour tous états q et q' par

$$q \sim q' \stackrel{\text{def}}{\iff} \forall w \in A^* \quad (q \cdot w \in F \iff q' \cdot w \in F).$$

La proposition suivante donne comment obtenir l'automate minimal à partir de n'importe quel automate déterministe.

Proposition 1.79. *Soit $\mathcal{A}$ un automate déterministe et complet acceptant un langage L . L'automate minimal $\mathcal{A}_L$ est égal à $\mathcal{A}/\sim$ où $\sim$ est la congruence de Nerode de $\mathcal{A}$.*

Preuve. Pour un état q , on note L_q le langage $\{w \mid q \cdot w \in F\}$. Deux états q et q' vérifient $q \sim q'$ si et seulement si $L_q = L_{q'}$. D'après le lemme 1.71, chaque langage L_q est de la forme $u^{-1}L$ pour un mot u qui étiquette un chemin de l'état initial à q . On peut donc identifier les classes de la congruence de Nerode avec les quotients à gauche de L . $\square$

1.7.3 Calcul de l'automate minimal

D'après la proposition précédente, calculer l'automate minimal revient à calculer la congruence de Nerode d'un automate déterministe. Ce calcul peut être fait de manière très efficace. Une première méthode naïve donne un temps de calcul en $O(n^2)$ où n est le nombre d'états de l'automate. Une seconde méthode basée sur le principe *diviser pour régner* permet d'obtenir un temps de calcul en $O(n \log n)$. Le résultat de cette méthode est étonnant car le principe *diviser pour régner* n'est pas évident à appliquer à la minimisation d'automates.

Méthode itérative

Soit $\mathcal{A} = (Q, A, E, \{i\}, F)$ un automate déterministe. On définit par récurrence une suite $(\sim_i)_{i \geq 0}$ de relations d'équivalence sur Q par

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

Proposition 1.80. *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 Nerode.*

Preuve. Les deux premières affirmations sont évidentes et on conclut par récurrence. Si $q \not\sim_k q'$ alors (comme $\sim_k = \sim_{k+1}$), $q \cdot a \not\sim_k q' \cdot a$, 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 Nerode.

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

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

Ce qui implique que $q \sim_N q'$. □

Algorithme de Hopcroft

L'algorithme de Hopcroft calcule la congruence de Nerode d'un automate en temps $O(n \log n)$. Il s'agit d'un algorithme particulièrement astucieux. Dans cette partie, on identifie une relation d'équivalence sur un ensemble Q avec la partition de Q en classes qu'elle induit.

Définition 1.81 (Stabilité et coupure). Soit $\mathcal{A} = (Q, A, E, I, F)$ un automate complet déterministe et émondé, et soient $a \in A$ et $B, C \subseteq Q$. La partie B est *stable* pour (C, a) si $B \cdot a \subseteq C$ ou si $B \cdot a \cap C = \emptyset$ avec $B \cdot a = \{q \cdot a \mid q \in B\}$. Sinon, la paire (C, a) *coupe* B en les deux parties B_1 et B_2 définies par

$$B_1 = \{q \in B \mid q \cdot a \in C\} \quad \text{et} \quad B_2 = \{q \in B \mid q \cdot a \notin C\}$$

La notion de stabilité qui vient d'être introduite permet de reformuler le fait d'être une congruence pour une partition. Une partition $\{P_1, \dots, P_k\}$ de Q compatible avec F est une congruence si et seulement si pour toutes part P_i est stable pour toute paire (P_j, a) où P_j est une part et a une lettre.

La preuve de la proposition suivante est immédiate. Par contre, le résultat de celle-ci est le principe même sur lequel est basé l'algorithme de Hopcroft.

Proposition 1.82. *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 $|Q|$.

Validité

À la terminaison, la partition calculée correspond à la congruence de Nerode. On dit qu'une partition $\mathcal{P}$ est *stable* pour (C, a) si $\forall P \in \mathcal{P}$, P est stable pour (C, a) .

On a l'invariant suivant :

Proposition 1.83. *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 Nerode, cela se fait par récurrence.

Complexité

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

□

Double déterminisation

Les deux méthodes précédentes s'appliquent lorsque l'automate de départ est déterministe. Si ce n'est pas le cas, il peut toujours être d'abord déterminisé grâce à la proposition 1.59 puis ensuite minimisé. On va voir une alternative qui consiste à appliquer deux déterminisations.

Pour un automate $\mathcal{A} = (Q, A, E, I, T)$, on note $d(\mathcal{A})$ la partie accessible de l'automate $\hat{\mathcal{A}}$ construit dans la preuve de la proposition 1.59. On note aussi $t(\mathcal{A})$ l'automate (Q, A, E^t, F, I) où l'ensemble E^t des transitions est donné par

$$E^t = \{p \xrightarrow{a} q \mid q \xrightarrow{a} p \in E\}.$$

De manière intuitive, l'automate $t(\mathcal{A})$ est obtenu en *renversant* les transitions et en échangeant états initiaux et états finaux. Le résultat de la proposition suivante est surprenant car il contredit la première intuition que la déterminisation fait augmenter le nombre d'états.

Proposition 1.84. *Soit $\mathcal{A}$ acceptant un langage L . L'automate $d(t(d(t(\mathcal{A}))))$ est l'automate minimal $\mathcal{A}_L$ de L .*

Preuve. Soit $\mathcal{A}'$ l'automate $d(t(d(t(\mathcal{A}))))$. Par construction, cet automate est déterministe. On vérifie en outre qu'il accepte le langage L . Il reste donc à vérifier que cet automate est minimal.

Soient P et P' deux états distincts de $\mathcal{A}'$. Par définition, P et P' sont deux ensembles états de l'automate $d(t(\mathcal{A}))$. Soit R un état de $d(t(\mathcal{A}))$ tel que $R \in P$ et $R \notin P'$. Par définition de $d(t(\mathcal{A}))$, R est un ensemble d'états de $\mathcal{A}$. Grâce au lemme 1.61, il existe un mot w tel que $R = \{q \mid q \xrightarrow{w} f \text{ avec } f \in F\}$. On en déduit que dans $\mathcal{A}'$, il existe un chemin de P à un état final étiqueté par w alors que ce n'est pas le cas pour P' . Ceci montre que P et P' ne sont pas équivalents pour la congruence de Nerode. $\square$

1.8 Propriétés de clôture

La classe des langages rationnels est très robuste. Elle est close pour d'innombrables opérations. Ceci est une propriété importante de ces langages et justifie l'intérêt qui leur est porté. Cette partie donne les principales propriétés de clôture mais la liste est loin d'être exhaustive.

1.8.1 Opérations booléennes

Il a déjà été vu que le complémentaire d'un langage rationnel est encore rationnel (cf. corollaire 1.67). Comme la classe des langages rationnels est par définition close par union, elle est aussi close par intersection.

1.8.2 Morphisme et morphisme inverse

La classe des langages rationnels est close par morphisme et par morphisme inverse.

Proposition 1.85. *Soit μ un morphisme de A^* dans B^* . Si $L \subseteq A^*$ est rationnel, alors $\mu(L)$ est rationnel. Si $K \subseteq B^*$ est rationnel, alors $\mu^{-1}(K)$ est aussi rationnel.*

Preuve. Il est possible d'obtenir une expression rationnelle pour $\mu(L)$ à partir d'une expression rationnelle pour L en remplaçant chaque occurrence d'une lettre a par $\mu(a)$.

Soit $\mathcal{A} = (Q, A, E, I, F)$ un automate acceptant le langage K . On construit un automate $\mathcal{A}'$ pour $\mu^{-1}(K)$ ayant même ensemble d'états. Il y a une transition $p \xrightarrow{a} q$ dans $\mathcal{A}'$ s'il y a un chemin $p \xrightarrow{\mu(a)} q$ dans $\mathcal{A}$. En gardant les mêmes états initiaux et finaux, l'automate $\mathcal{A}$ accepte le langage $\mu^{-1}(K)$. On peut remarquer que si $\mathcal{A}$ est déterministe alors $\mathcal{A}'$ est aussi déterministe. $\square$

1.8.3 Mélange

Soient K et L deux langages sur un alphabet A . Le *mélange* $K \text{ III } L$ des langages K et L est le langage sur A^* défini par la formule suivante.

$$K \text{ III } L = \{u_0 v_0 u_1 \cdots u_n v_n \mid u_i, v_i \in A^*, u_0 \cdots u_n \in K \text{ et } v_0 \cdots v_n \in L\}.$$

On peut remarquer que u est un sous-mot de w si et seulement si le mot w appartient au langage $u \text{ III } A^*$. La classe des langages rationnels est close par mélange.

Proposition 1.86. *Si les langages L et L' sont rationnels, alors le langage $L \text{ III } L'$ est encore rationnel.*

Preuve. Soient $\mathcal{A} = (Q, A, E, I, F)$ et $\mathcal{A}' = (Q', A, E', I', F')$ deux automates acceptant respectivement les langages L et L' . On construit l'automate dont l'ensemble d'états est $Q \times Q'$, les ensembles d'états initiaux et finaux sont $I \times I'$ et $F \times F'$ et dont l'ensemble des transitions est

$$\{(p, p') \xrightarrow{a} (q, p') \mid p \xrightarrow{a} q \in E\} \cup \{(p, p') \xrightarrow{a} (p, q') \mid p' \xrightarrow{a} q' \in E'\}.$$

On vérifie sans difficulté que cet automate accepte $L \text{ III } L'$. $\square$

Le théorème de Higman (théorème 1.31) permet de montrer qu'un langage de la forme $L \text{ III } A^*$ est toujours rationnel même si L n'est pas rationnel.

Exercice 1.87. 1. Soient les langages L_n définis par récurrence par $L_0 = \varepsilon$ et $L_{n+1} = L_n \text{ III } (ab)^*$. Montrer que L_n est l'ensemble des mots w tels que $|w|_a = |w|_b$ et $|u|_b \leq |u|_a \leq |u|_b + n$ pour tout préfixe u de w .

2. Soit une suite croissante $n_1, \dots, n_k$ d'entiers telle que $n_1 = 1$ et pour tout $1 \leq i < n$, on a $n_i \leq n_{i+1} \leq n_0 + \cdots + n_i + 1$. Montrer que le langage $(a^{n_1} b^{n_1})^* \text{ III } (a^{n_2} b^{n_2})^* \text{ III } \cdots \text{ III } (a^{n_k} b^{n_k})^*$ est encore égal à l'ensemble des mots w tels que $|w|_a = |w|_b$ et $|u|_b \leq |u|_a \leq |u|_b + n_1 + \cdots + n_k$ pour tout préfixe u de w . On peut remarquer qu'une suite $n_1, \dots, n_k$ d'entiers vérifie les conditions ci-dessus si et seulement si pour tout entier s tel que $1 \leq s \leq n_1 + \cdots + n_k$, il existe une suite d'indices $i_1 < \cdots < i_r$ telle que $s = n_{i_1} + \cdots + n_{i_r}$.

Exercice 1.88. Soit L un langage. Pour tout entier n tel que L contienne au moins un mot de longueur n , w_n est défini comme le mot le plus petit pour l'ordre lexicographique de $L \cap A^n$. Montrer que si L est rationnel, le langage $\{w_n \mid L \cap A^n \neq \emptyset\}$ est encore rationnel.

Exercice 1.89. Soit $s = (s_n)_{n \geq 0}$ une suite strictement croissante d'entiers. Pour un mot $w = w_0 \cdots w_n$, on note $w[s]$ le mot $w_{s_0} w_{s_1} \cdots w_{s_k}$ où k est l'unique entier tel que $s_k \leq n < s_{k+1}$. Pour un langage $L \subseteq A^*$, on note $L[s]$, le langage $\{w[s] \mid w \in L\}$. Montrer pour chacune des suites suivantes que si L est rationnel, alors $L[s]$ est encore rationnel.

$$s_n = n!, \quad s_n = 2^n, \quad s_n = n^2.$$

1.9 Lemme de l'étoile et ses variantes

L'intérêt principal de ce lemme, également appelé *lemme d'itération* ou *lemme de pompage* est de donner une condition nécessaire pour qu'un langage soit rationnel. Cette condition est souvent utilisée pour montrer qu'un langage donné n'est pas rationnel. 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, ou de sa version plus évoluée, le théorème de Ramsey.

Proposition 1.90 (Lemme de l'étoile V1). *Pour tout langage rationnel L , il existe un entier n tel que pour tout mot f ,*

$$\left. \begin{array}{l} |f| \geq n \\ f \in L \end{array} \right\} \implies \left\{ \begin{array}{l} \exists u, v, w \in A^* \quad v \neq \varepsilon \\ f = uvw \text{ et } uv^*w \subseteq L \end{array} \right.$$

Preuve. Soit n le nombre d'états d'un automate acceptant le langage L , par exemple l'automate minimal de L . Si $|f| \geq n$, un chemin acceptant pour f doit repasser au moins deux fois par un même état q . Si u, v et w sont les étiquettes d'un chemin de l'état initial à q , du circuit de q à q et du chemin de q à un état final, on obtient une factorisation convenable de f . $\square$

Exemple 1.91. Grâce à ce lemme, on peut montrer que $L = \{a^n b^n \mid n \geq 0\}$ n'est pas rationnel.

Par contre, $L \cup A^* b a A^*$ vérifie la condition de rationalité de ce lemme sans pour autant être rationnel. Ceci montre que cette condition n'est pas suffisante pour garantir la rationalité.

Les trois propositions suivantes raffinent successivement l'énoncé de la précédente mais les preuves demeurent essentiellement identiques.

Proposition 1.92 (Lemme de l'étoile V2). *Pour tout langage rationnel L , il existe un entier n tel que pour tout mot f ,*

$$\left. \begin{array}{l} |f| \geq n \\ f \in L \end{array} \right\} \implies \left\{ \begin{array}{l} \exists u, v, w \in A^* \quad v \neq \varepsilon \text{ et } |uw| \leq n \\ f = uvw \text{ et } uv^*w \subseteq L \end{array} \right.$$

Proposition 1.93 (Lemme de l'étoile V3). *Pour tout langage rationnel L , il existe un entier n tel que pour tout mot f ,*

$$\left. \begin{array}{l} f = uvw \text{ avec } |v| \geq n \\ f \in L \end{array} \right\} \implies \left\{ \begin{array}{l} \exists v_1, v_2, v_3 \quad v_2 \neq \varepsilon \text{ et } |v_1 v_3| \leq n \\ v = v_1 v_2 v_3 \text{ et } uv_1 v_2^* v_3 w \subseteq L \end{array} \right.$$

Proposition 1.94 (Lemme de l'étoile V4). *Pour tout langage rationnel L , il existe un entier n tel que pour tout mot f ,*

$$\left. \begin{array}{l} f = uv_1 \cdots v_n w \\ f \in L \text{ et } v_i \in A^+ \end{array} \right\} \implies \left\{ \begin{array}{l} \exists i, j \ 0 \leq i < j \leq n \\ uv_1 \cdots v_i (v_{i+1} \cdots v_j)^* v_{j+1} \cdots v_n w \subseteq L \end{array} \right.$$

Malheureusement, cette condition n'est toujours pas suffisante, comme le montre l'exemple suivant.

Exemple 1.95. Soient les alphabets $A = \{a, b\}$ et $B = \{a, b, \#\}$. Soit le langage L sur B défini par $K' + A^* K A^* \# A^* + A^* \# A^* K A^*$ où les langages K et K' sont définis par $K = \{uu \mid u \in A^*\}$ et $K' = \{u\#v \mid u, v \in A^* \text{ et } u \neq v\}$. On vérifie que le langage L vérifie les conclusions du lemme précédent bien qu'il ne soit pas rationnel.

Pour présenter la version la plus aboutie du lemme de l'étoile, nous avons besoin de quelques définitions. On pourra consulter [Sak03, p. 127] pour cette version forte du lemme.

Définition 1.96 (Propriétés σ_k et σ'_k). Un langage L vérifie la propriété σ_k (resp. σ'_k) pour $k \geq 0$ si pour toute factorisation $f = uv_1 v_2 \cdots v_k w$ où tous les mots v_i sont non vides, il existe deux indices $1 \leq i < j \leq k$ tels que

$$\begin{aligned} \forall n \geq 0 \quad f \in L &\iff uv_1 \cdots v_i (v_{i+1} \cdots v_j)^n v_{j+1} \cdots v_k w \in L && \text{pour } \sigma_k \\ f \in L &\iff uv_1 \cdots v_i v_{j+1} \cdots v_k w \in L && \text{pour } \sigma'_k \end{aligned}$$

Le théorème suivant établit que les propriétés σ_k et σ'_k qui généralisent les lemmes de l'étoile sont caractéristiques des langages rationnels.

Théorème 1.97 (Ehrenfeucht, Parikh et Rozenberg 1981). *Pour tout langage L , les propositions suivantes sont équivalentes.*

1. L est rationnel.
2. Il existe $k \geq 0$ tel que L vérifie σ_k .
3. Il existe $k \geq 0$ tel que L vérifie σ'_k .

La preuve du théorème s'appuie sur le théorème suivant. Si E est un ensemble, on notera $\mathfrak{P}_k(E)$ l'ensemble des parties à k éléments de E . On admettra le théorème de Ramsey dont la démonstration peut être consultée en [vLW92].

Théorème 1.98 (Ramsey 1929). *Pour tout triplet d'entiers naturels (k, m, r) , il existe un entier $N(k, m, r)$ tel que pour tout :*

- ensemble E tel que $|E| \geq N(k, m, r)$,
- ensemble C tel que $|C| = m$,
- fonction $f : \mathfrak{P}_k(E) \rightarrow C$,

il existe $F \subseteq E$ tel que :

- $|F| \geq r$;
- $|f(\mathfrak{P}_k(F))| \leq 1$.

Preuve du lemme de l'étoile fort.

Trivialités

Il est tout d'abord évident que (1.) implique (2.) et que (2.) implique (3.). Il reste donc à montrer que (3.) implique (1.).

Démarche

On montre dans un premier temps que si L satisfait σ'_k , alors $v^{-1}L$ satisfait σ'_k pour tout mot v . 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 1.72 (p. 35) permet de conclure qu'il est rationnel.

Premier temps

Soit L un langage qui vérifie σ'_k pour un entier k fixé et soit un mot $v \in A^*$. On montre que le langage $v^{-1}L$ vérifie aussi σ'_k . Soit alors f , un mot qui s'écrit $f = uv_1v_2 \cdots v_kw$ où les mots v_i ne sont pas vides pour $1 \leq i \leq k$. Les mots u et w sont quelconques.

Ainsi, $f \in v^{-1}L \iff vf \in L \iff vuv_1 \cdots v_iv_{j+1} \cdots v_kw \in L$ pour deux indices $1 \leq i < j \leq k$ obtenus en appliquant la propriété σ'_k au mot $f' = vf$ pour le langage L . Il est alors immédiat que $v^{-1}L$ satisfait σ'_k .

Deuxième temps

En appliquant le théorème de Ramsey pour $k = m = 2$ et $r = k + 1$, on obtient un entier $N = N(2, 2, k + 1)$.

Soient L et K deux langages vérifiant σ'_k pour un entier k fixé. On montre que si K et L coïncident sur les mots de longueur au plus N , c'est-à-dire $L \cap (\varepsilon + A)^N = K \cap (\varepsilon + A)^N$, alors K et L sont égaux. Ceci montre que le nombre de langages vérifiant σ'_k est fini puisque $(\varepsilon + A)^N$ est fini.

On suppose que $L \cap (\varepsilon + A)^N = K \cap (\varepsilon + A)^N$ et on montre par récurrence sur $|f|$ que $f \in K \iff f \in L$ pour tout mot f . Cette propriété est bien sûr immédiate lorsque $|f| \leq N$.

Si $|f| > N$, le mot f s'écrit $f = a_0 \cdots a_N w$ où $a_0, \dots, a_N$ sont des lettres et w est un mot.

Posons maintenant :

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

Si l'on pose $E = \{1, \dots, N\}$, $C = \{0, 1\}$ 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 $F = \{i_0, \dots, i_k\}$ 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 $xv_1v_2 \cdots v_k y$ 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 $f \in K \iff a_0 \cdots a_{i_\alpha} a_{i_\beta+1} \cdots a_N w \in K$.

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

□

1.10 Hauteur d'étoile

On étudie ici une classification des langages rationnels basée sur le nombre d'étoiles imbriquées des expressions rationnelles. La raison pour laquelle on s'intéresse à cette opération est qu'elle est celle qui donne toute leur puissance aux expressions rationnelles. Les seuls langages à pouvoir être décrits sans étoile sont les langages finis.

La *hauteur d'étoile* $h(e)$ d'une expression rationnelle e est définie par récurrence sur la structure de l'expression par les formules suivantes.

$$\begin{aligned} h(a) &= h(\varepsilon) = 0 \text{ pour toute lettre } a \in A \\ h(e + f) &= \max(h(e), h(f)) \\ h(e f) &= \max(h(e), h(f)) \\ h(e^*) &= 1 + h(e) \end{aligned}$$

Exemple 1.99. Les expressions $ab+ba$, $(ab+a)^*$ et $(ab^*a+b)^*$ sont respectivement de hauteur d'étoile 0, 1 et 2.

Il faut remarquer qu'un même langage peut être décrit par plusieurs expressions rationnelles ayant des hauteurs d'étoile différentes. Par exemple, les deux expressions $(a + b)^*$ et $(a^*b)^*a^*$ sont de hauteurs d'étoile 1 et 2 mais décrivent le même langage de tous les mots sur l'alphabet $\{a, b\}$. La *hauteur d'étoile* d'un langage L est définie comme la hauteur minimale d'une expression rationnelle décrivant L . Les langages de hauteur d'étoile 0 sont les langages finis.

Exemple 1.100. Le langage $(a + b)^*$ est de hauteur d'étoile 1 puisqu'il est décrit par une expression rationnelle de hauteur 1 et qu'il ne peut être décrit par une expression de hauteur 0 parce qu'il est infini.

Un premier résultat sur la hauteur d'étoile est que pour tout entier n , il existe un langage rationnel de hauteur d'étoile n . Il existe un algorithme qui calcule la hauteur d'étoile d'un langage rationnel donné mais celui-ci est très complexe et dépasse le cadre de cet ouvrage.

Comme la classe des langages rationnels est close pour toutes les opérations booléennes, il est possible de considérer des expressions utilisant non seulement les opérations rationnelles mais aussi toutes les opérations booléennes. On peut d'ailleurs se limiter à la complémentation puisque l'union et la complémentation permettent d'exprimer les autres opérations booléennes. La hauteur d'étoile peut être étendue à ces expressions généralisées en posant

$$\begin{aligned} h(\emptyset) &= 0, \\ h(e^c) &= h(e). \end{aligned}$$

La *hauteur d'étoile généralisée* d'un langage L est la hauteur d'étoile minimale d'une expression utilisant les opérations rationnelles et booléennes et décrivant L .

Exemple 1.101. Le langage $(a + b)^* = \emptyset^c$ est de hauteur d'étoile généralisée 0.

Les langages de hauteur d'étoile généralisée 0 sont appelés langages sans étoile. Ils sont étudiés en détail et caractérisés de manière algébrique à la section 1.12 (p. 52). C'est encore un problème ouvert de savoir s'il existe des langages de hauteur d'étoile généralisée strictement plus grande que 1.

1.11 Reconnaissance par morphisme

Les langages rationnels peuvent être décrits par des expressions rationnelles ou par des automates. On donne maintenant une troisième façon plus algébrique de définir ces langages.

Définition 1.102 (Monoïde). On appelle *monoïde* tout ensemble muni d'une loi de composition interne associative qui possède un élément neutre noté 1_M .

Exemple 1.103. Quelques exemples classiques de monoïdes :

- A^* muni de la concaténation (pour A , alphabet quelconque),
- l'ensemble $\mathfrak{P}(A^*)$ des parties de A^* muni du produit des langages,
- tout groupe G muni de sa loi naturelle,
- l'ensemble des matrices $n \times n$ sur un anneau et plus généralement sur un semi-anneau (pas d'inverse pour l'addition),
- $M = \{1, \alpha, \beta\}$ où 1 est l'élément neutre et où la loi est définie pour les autres éléments par $\alpha\beta = \alpha^2 = \alpha$ et $\beta^2 = \beta\alpha = \beta$ (loi $xy = x$),
- $M = \{1, \alpha, \beta\}$ où 1 est l'élément neutre et où la loi est définie pour les autres éléments par $\alpha^2 = \beta\alpha = \alpha$ et $\alpha\beta = \beta^2 = \beta$ (loi $xy = y$),
- $M = \{1\} \cup I \times J$ pour deux ensembles I et J où la loi est $(i, j)(i', j') = (i, j')$.

Définition 1.104 (Morphisme de monoïdes). Soit M et M' deux monoïdes. Un *morphisme* de M dans M' est une application $\mu : M \rightarrow M'$ qui vérifie :

- $\mu(1_M) = 1_{M'}$,
- $\mu(xy) = \mu(x)\mu(y)$, pour tous éléments x et y de M .

Exemple 1.105. L'ensemble $\mathbb{N}$ muni de l'addition est un monoïde. L'application $w \mapsto |w|$ est un morphisme de A^* dans $\mathbb{N}$.

La proposition suivante justifie le fait que le monoïde A^* soit appelé monoïde libre. Cette propriété caractérise le monoïde libre engendré par A .

Proposition 1.106. *Toute fonction $\mu : A \rightarrow M$ de A dans un monoïde M se prolonge de façon unique en un morphisme de monoïde de A^* dans M .*

Preuve. Si le morphisme $\hat{\mu}$ prolonge μ , on a pour tout mot $w = a_1 \cdots a_n$ de A^* ,

$$\hat{\mu}(w) = \hat{\mu}(a_1) \cdots \hat{\mu}(a_n) = \mu(a_1) \cdots \mu(a_n).$$

Ceci définit $\hat{\mu}$ de manière unique. Inversement, il est facile de vérifier que la formule ci-dessus définit bien un morphisme. $\square$

Définition 1.107 (Reconnaissance par monoïde). Un langage $L \subseteq A^*$ est *reconnu* par un morphisme $\mu : A^* \rightarrow M$ si et seulement si il existe une partie P de M telle que $L = \mu^{-1}(P)$. Par extension, un monoïde M *reconnaît* le langage L s'il existe un morphisme $\mu : A^* \rightarrow M$ qui reconnaît L .

Quand un morphisme de monoïdes reconnaît un langage L , on peut toujours prendre la partie P égale à $P = \mu(L)$. Autrement dit, un morphisme $\mu : A^* \rightarrow M$ reconnaît L si et seulement si $L = \mu^{-1}(\mu(L))$. Si le morphisme μ n'est pas surjectif, on peut remplacer le monoïde M par le sous-monoïde $M' = \mu(A^*)$.

Exemple 1.108. On considère le monoïde M égal au groupe $\mathbb{Z}/2\mathbb{Z}$ à deux éléments.

$$\begin{aligned} - \mu : A^* &\rightarrow \mathbb{Z}/2\mathbb{Z} && (\text{dans ce cas : } \mu^{-1}(0) = (A^2)^*) \\ &w \mapsto |w| \bmod 2 \\ - \mu : A^* &\rightarrow \mathbb{Z}/2\mathbb{Z} && (\text{dans ce cas : } \mu^{-1}(0) = (ab^*a + b)^*) \\ &w \mapsto |w|_a \bmod 2 \end{aligned}$$

Exemple 1.109. On considère $A = \{a, b\}$ et le monoïde $M = \{1, \alpha, \beta\}$ avec $\alpha^2 = \beta\alpha = \alpha$ et $\alpha\beta = \beta^2 = \beta$ (cf. exemple 1.103). Le morphisme $\mu : A^* \rightarrow M$ défini par :

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

Proposition 1.110 (Rationalité par morphisme). *Un langage $L \subseteq A^*$ est rationnel si et seulement s'il est reconnu par un monoïde fini.*

Preuve. Soit un langage L et un morphisme $\mu : A^* \rightarrow M$ tel que $L = \mu^{-1}(P)$ pour une partie P de M . On construit l'automate $\mathcal{A} = (Q, A, E, I, F)$ où $Q = M$, $I = \{1\}$, $F = \mu(L)$ et $E = \{(m, a, m\mu(a)) \mid m \in M, a \in A\}$. Cet automate reconnaît L , ce qui prouve un sens de l'équivalence. $\square$

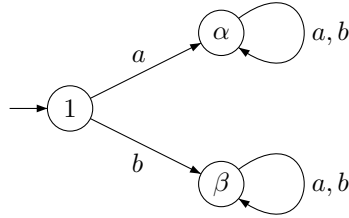


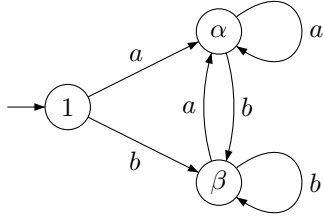
FIG. 1.22 – Automate du monoïde $M = \{1, \alpha, \beta\}$ avec la loi $xy = x$

Exemple 1.111. Si on applique la construction précédente au monoïde $M = \{1, \alpha, \beta\}$ avec la loi $xy = x$ (cf. exemple 1.103), on obtient l'automate de la figure 1.22.

Exemple 1.112. Si on applique la construction précédente au monoïde $M = \{1, \alpha, \beta\}$ avec la loi $xy = y$ (cf. exemple 1.103), on obtient l'automate de la figure 1.23.

Pour montrer que tout langage rationnel est reconnu par un monoïde fini, on introduit quelques définitions. Soient r et r' deux relations binaires sur un même ensemble E . On définit la composition de ces deux relations, notée rr' par la formule suivante.

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

FIG. 1.23 – Automate du monoïde $M = \{1, \alpha, \beta\}$ avec la loi $xy = y$

On vérifie sans peine que la loi de composition ainsi définie est associative et que l'identité est un élément neutre. On note $\mathcal{R}_E$ le *monoïde des relations binaires* sur l'ensemble E muni de cette loi de composition interne.

Définition 1.113 (Monoïde des transitions). Soit un automate $\mathcal{A}$ égal à (Q, A, E, I, F) . L'application $\mu : A^* \rightarrow \mathcal{R}_Q$ définie par :

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

est un morphisme de A^* dans $\mathcal{R}_Q$. Son image $\mu(A^*)$ dans $\mathcal{R}_Q$ est appelée *monoïde des transitions* de l'automate $\mathcal{A}$.

Avec ce morphisme, il suffit de prendre :

$$P = \{r \mid r \cap (I \times F) \neq \emptyset\}$$

pour avoir le résultat recherché.

Définition 1.114 (Sous-monoïde). M' est appelé *sous-monoïde* de M lorsque $M' \subseteq M$ et $M' \xrightarrow{id} M$ est un morphisme.

Exemple 1.115. Quelques exemples de sous-monoïdes.

- Les sous-monoïdes de A^* sont les langages de la forme L^* pour $L \subset A^*$.
- L'ensemble des parties rationnelles de A^* est un sous-monoïde de $\mathfrak{P}(A^*)$.

Exercice 1.116. On dit qu'un langage est *préfixe* si $L \cap LA^+ = \emptyset$, c'est-à-dire si un préfixe stricte d'un mot de L n'est jamais un mot de L . Montrer que l'ensemble des langages préfixes est un sous-monoïde de $\mathfrak{P}(A^*)$ et que dans ce monoïde, l'égalité

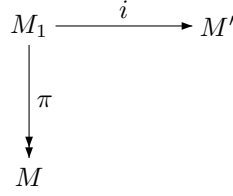
$$L_1 \cdots L_m = K_1 \cdots K_n$$

n'est vérifiée que si $m = n$ et $L_i = K_i$ pour tout $1 \leq i \leq m$. Un tel monoïde est dit *libre* car il est isomorphe à un monoïde de la forme B^* .

Définition 1.117 (Monoïde quotient). On dit que M est *quotient* de M' s'il existe un morphisme surjectif $\mu : M' \twoheadrightarrow M$.

Définition 1.118 (Monoïde diviseur). On dit que M *divise* M' et on note $M \triangleleft M'$ si M est un quotient d'un sous-monoïde M_1 de M' (cf. figure 1.24).

la notion de *diviseur* pour des objets algébriques est l'équivalent pour les graphes de la notion de *mineur*.

FIG. 1.24 – Relation de division $M \triangleleft M'$

Proposition 1.119 (Ordre de la division). *La relation de division est une relation d'ordre sur l'ensemble des monoïdes finis.*

On perd l'anti-symétrie lorsque les monoïdes sont infinis comme le montre l'exemple suivant.

Exemple 1.120. 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 définis par :

$$\begin{aligned}
\mu : M &\rightarrow M' & : w &\mapsto w \\
\mu : M' &\rightarrow M & : \begin{cases} a \mapsto aa \\ b \mapsto bb \\ c \mapsto ab \\ d \mapsto ba \end{cases}
\end{aligned}$$

Preuve. La réflexivité est évidente et l'anti-symétrie se vérifie facilement par équipotence.

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

$$\begin{array}{ccccc}
M_2 & \xrightarrow{id} & M'_1 & \xrightarrow{id} & M'' \\
\downarrow \pi' & & \downarrow \pi' & & \\
M_1 & \xrightarrow{id} & M' & & \\
\downarrow \pi & & & & \\
M & & & &
\end{array}$$

On vérifie alors que le monoïde $M_2 = \pi'^{-1}(M_1)$ est un sous-monoïde de M'_1 et donc de M'' et que $\pi(\pi'(M_2)) = \pi(M_1) = M$. $\square$

Définition 1.121 (Congruence). Une relation d'équivalence $\sim$ définie sur un monoïde M est appelée *congruence* (de monoïde) si pour tous x, x', y et y' dans M , on a :

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

Pour qu'une relation d'équivalence $\sim$ soit une congruence, il suffit que pour tous y et y' dans M , l'implication $y \sim y' \implies \forall x, z \, xyz \sim xy'z$ soit satisfaite.

Proposition 1.122. *Si $\sim$ est une congruence sur M , le quotient $M/\sim$ est un monoïde pour la loi définie par $[x][y] = [xy]$ où $[x]$ désigne la classe de x dans le quotient.*

Définition 1.123 (Congruence syntaxique). Pour tout langage $L \subseteq A^*$, on appelle *contexte* de $y \in A^*$ l'ensemble $C(y) = \{(x, z) \in (A^*)^2 : xyz \in L\}$. La relation $\sim_L$ définie par :

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

est appelée *congruence syntaxique* de L .

La proposition suivante justifie la terminologie.

Proposition 1.124 (Monoïde syntaxique). *Pour tout langage L , la congruence syntaxique $\sim_L$ est effectivement une congruence et le monoïde quotient $A^*/\sim_L$ est appelé monoïde syntaxique de L .*

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

la proposition suivante montre que le monoïde syntaxique d'un langage est en quelque sorte l'équivalent algébrique de l'automate minimal. Il est le plus petit monoïde reconnaissant le langage. Beaucoup de propriétés d'un langage peuvent être déduites de son monoïde syntaxique.

Proposition 1.125. *Un monoïde M reconnaît un langage $L \subseteq A^*$ si et seulement si le monoïde syntaxique $M(L)$ divise M .*

On commence par prouver un premier lemme qui montre que la relation de division est compatible avec la reconnaissabilité.

Lemme 1.126. *Pour tout langage $L \subseteq A^*$ et tout monoïde M , les propositions suivantes sont vérifiées.*

1. *Si M reconnaît L si M est sous-monoïde de M' , alors M' reconnaît L .*
2. *Si M reconnaît L si M est quotient de M' , alors M' reconnaît L .*
3. *Si M reconnaît L si M divise M' , alors M' reconnaît L .*

Preuve. La proposition 1. découle du fait qu'un morphisme de A^* dans M peut être vu comme un morphisme de A^* dans M' . La proposition 3. est conséquence de 1. et de 2.

Pour la proposition 2, soit μ un morphisme de A^* dans M et π un morphisme surjectif de M' dans M . Pour toute lettre a de A , on pose $\mu'(a) = m_a$ où m_a est un élément quelconque de M' vérifiant $\pi(m_a) = \mu(a)$. D'après la proposition 1.106, μ' se prolonge en un morphisme de A^* dans M' qui vérifie $\pi(\mu'(w)) = \mu(w)$. Il est alors facile de vérifier que la partie $P' = \pi'^{-1}(\mu(L))$ de M' vérifie $\mu'^{-1}(P') = L$ et que μ' reconnaît L . $\square$

Ce second lemme établit qu'un langage est bien reconnu par son monoïde syntaxique.

Lemme 1.127. *Pour tout langage L , le monoïde syntaxique $M(L)$ reconnaît L .*

Preuve. Il faut remarquer qu'un mot w appartient à L si et seulement si son contexte $C(w)$ contient la paire $(\varepsilon, \varepsilon)$. Il est alors clair que si une classe de la congruence syntaxique $\sim_L$ contient un mot de L , elle est alors entièrement contenu dans L .

L'application canonique de A^* dans $M(L) = A^*/\sim$ qui associe à tout mot w sa classe $[w]$ est un morphisme. D'après la remarque précédente, ce morphisme reconnaît L puisque $L = \bigcup_{w \in L} [w]$. $\square$

On procède maintenant à la preuve de la proposition.

Preuve. En utilisant les lemmes 1.127 et 1.126, on obtient immédiatement que, si $M(L)$ divise M , alors M reconnaît aussi L .

Réciproquement, si M reconnaît L , il existe un morphisme $\mu : A^* \rightarrow M$ tel que $L = \mu^{-1}(P)$ pour une partie $P \subseteq M$. On note π la morphisme canonique de A^* dans $M(L)$. Le monoïde $M' = \mu(A^*)$ est un sous-monoïde de M . On va montrer que $M(L)$ est un quotient de M' ce qui prouvera que $M(L)$ divise M .

On montre que pour tous mots w et w' , si $\mu(w) = \mu(w')$, alors on a aussi $\pi(w) = \pi(w')$. Une paire (u, v) appartient au contexte $C(w)$ si uwv appartient à L , c'est-à-dire si $\mu(u)\mu(w)\mu(v)$ appartient à P . Ceci montre que les contextes $C(w)$ et $C(w')$ sont égaux et que $\pi(w) = \pi(w')$.

Pour tout élément m de M' , on peut définir $\pi'(m) = \pi(w)$ où w est un mot tel que $\mu(w) = m$ puisque $\pi(w)$ est indépendant du choix de w . On vérifie alors sans difficulté que π' ainsi défini est un morphisme surjectif de M' dans $M(L)$.

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

$\square$

Pour le calcul du monoïde syntaxique, on utilise la proposition suivante.

Proposition 1.128. *Le monoïde syntaxique $M(L)$ d'un langage rationnel L est égal au monoïde des transitions de l'automate minimal de L .*

Preuve. D'après la proposition 1.110, le monoïde des transitions de l'automate minimal de L reconnaît L . Il est donc divisible par le monoïde syntaxique $M(L)$ d'après la proposition 1.125. Soient deux mots w et w' ayant des images différentes dans le monoïde des transitions de l'automate minimal. Puisque cet automate est déterministe, il existe un état p tel que $p \cdot w \neq p \cdot w'$. Soient q et q' les états $p \cdot w$ et $p \cdot w'$. Puisque cet automate est minimal, il existe un mot v tel que $q \cdot v$ est final et $q' \cdot v$ n'est pas final (ou l'inverse). Il existe aussi un mot u tel que $i \cdot u = p$. On en déduit que la paire (u, v) appartient à $C(w)$ mais pas à $C(w')$ et que w et w' ont des images différentes dans le monoïde syntaxique de L . $\square$

Exemple 1.129. Soit le langage $L = (ab)^*$ dont l'automate minimal est représenté à la figure 1.21. On construit son monoïde syntaxique en déterminant toutes les relations de transitions de l'automate. Comme l'automate est déterministe, ces relations sont des fonctions. On obtient la table ci-dessous.

Le monoïde associé comporte 6 éléments $\{1, 0, \alpha, \beta, \alpha\beta, \beta\alpha\}$ qui vérifie les relations suivantes. Ceci constitue en fait une présentation par générateurs et relations de ce monoïde.

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

	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 – Fonctions de transitions de l'automate minimal de $(ab)^*$

La reconnaissance par monoïdes des langages rationnels permet de résoudre rapidement certaines questions. Pour illustrer le propos, nous allons montrer la proposition suivante dont la preuve est grandement facilitée par les monoïdes. Le plus intéressant n'est pas le résultat en soi mais la technique de preuve. Le résultat est en fait un cas particulier d'une construction beaucoup plus générale.

Proposition 1.130. *Soit σ une substitution de A^* dans B^* et soit $K \subset B^*$ un langage rationnel. Les deux langages $\{w \mid \sigma(w) \cap K \neq \emptyset\}$ et $\{w \mid \sigma(w) \subseteq K\}$ sont rationnels.*

Pour un monoïde M , l'ensemble $\mathfrak{P}(M)$ des parties de M est naturellement muni d'une structure de monoïde par le produit $P \cdot Q = \{pq \mid p \in P \text{ et } q \in Q\}$ pour deux parties $P, Q \subseteq M$.

Preuve. On va montrer que si K est reconnu par le monoïde M , alors les deux langages $L = \{w \mid \sigma(w) \cap K \neq \emptyset\}$ et $L' = \{w \mid \sigma(w) \subseteq K\}$ sont reconnus par le monoïde $\mathfrak{P}(M)$. Soit $\mu : B^* \rightarrow M$ un morphisme de B^* dans un monoïde fini M tel que $K = \mu^{-1}(P)$ pour une partie $P \subseteq M$. On définit le morphisme $\hat{\mu} : A^* \rightarrow \mathfrak{P}(M)$ par $\hat{\mu}(a) = \mu(\sigma(a))$. On vérifie que $\hat{\mu}(w) = \mu(\sigma(w))$ pour tout mot $w \in A^*$. Ceci implique que $L = \hat{\mu}^{-1}(Q)$ et $L' = \hat{\mu}^{-1}(Q')$ où Q et Q' sont respectivement les ensembles $\{T \mid T \cap P \neq \emptyset\}$ et $\{T \mid T \subseteq P\}$ des parties de M qui intersectent P et qui sont incluses dans P . $\square$

1.12 Langages sans étoile

Le but de cette partie est d'illustrer l'utilisation du monoïde syntaxique et de montrer la puissance de cet outil. On introduit la classe des langages appelés *sans étoile* et on prouve le théorème de Schützenberger qui donne une élégante caractérisation de ces langages en terme de groupes contenus dans leurs monoïdes syntaxiques.

Définition 1.131 (Langage sans étoile). La famille des *langages sans étoile* est la plus petite famille $\mathcal{E}$ telle que :

- $\emptyset \in \mathcal{E}$ et $\{a\} \in \mathcal{E}$ pour toute lettre a ;
- $\mathcal{E}$ est close par union, complémentation et produit. Ceci signifie que si les langages K et L appartiennent à $\mathcal{E}$, alors les langages $K + L$, $A^* \setminus L$ et KL appartiennent aussi à $\mathcal{E}$.

Les langages sans étoile sont aussi les langages de hauteur d'étoile généralisée 0 (cf. section 1.10 p. 44).

Exemple 1.132. Quelques exemples et contre-exemples de langages sans étoile.

- $A^* = A^* \setminus \emptyset$ et est donc un langage sans étoile.
- $A^*aA^*bA^*$ est sans-étoile.
- $\{\varepsilon\} = A^* \setminus \bigcup_{a \in A} A^*aA^*$ est 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.

Le fait que le langage $(aa)^*$ ne soit effectivement pas sans étoile n'est pas évident *a priori*. Il est facile de montrer qu'un langage est sans étoile en donnant explicitement une expression sans étoile pour ce langage. Par contre, il est plus ardu de prouver qu'un langage n'est pas sans étoile. La suite de cette partie est consacrée à une caractérisation algébrique des langages sans étoile qui permet facilement de prouver qu'un langage donné n'est pas sans étoile.

Définition 1.133 (Groupe contenu dans un monoïde). On dit qu'un groupe G est *contenu* dans un monoïde M si :

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

Un monoïde M est *apériodique* s'il ne contient que des groupes triviaux, c'est-à-dire réduits à un seul élément.

Lorsqu'un groupe G est contenu dans un monoïde M , l'élément neutre de G ne coïncide par nécessairement avec l'élément neutre de M ($1_G \neq 1_M$).

Les résultats suivants montrent que les monoïdes apériodiques sont à l'opposé des groupes. Soit x un élément d'un monoïde M . On considère l'ensemble $\{x^k \mid k \geq 0\}$ des puissances de x qui est en fait le sous-monoïde engendré par x . Si M est fini, il existe deux entiers $l < k$ tels que $x^k = x^l$. Si k est en outre choisi le plus petit possible, les éléments $1, x, x^2, \dots, x^{k-1}$ sont distincts et on a le diagramme suivant, communément appelé *poêle à frire*.

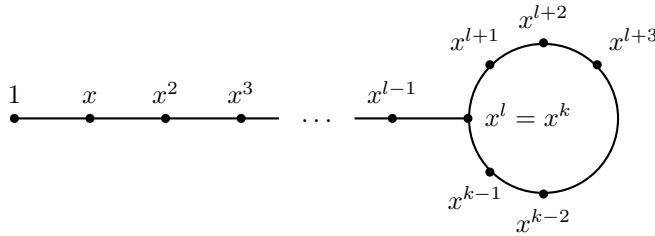


FIG. 1.25 – Diagramme de la poêle à frire

Si k est le plus petit entier pour lequel il existe l tel que $x^k = x^l$, les entiers k et l sont uniques. On note alors $l_x = l$ et $p_x = k - l$. Le lemme suivant caractérise les groupes et les monoïdes apériodiques parmi les monoïdes finis.

Lemme 1.134. *Pour un monoïde fini M , les trois propriétés suivantes sont équivalentes.*

1. M est un groupe.
2. Pour tout $x \in M$, on a $l_x = 0$ et donc $x^{p_x} = 1$.

3. Il existe un entier ω tel que $x^\omega = 1$ pour tout $x \in M$.

Pour un monoïde fini M , les trois propriétés suivantes sont équivalentes.

1. M est apériodique.
2. Pour tout $x \in M$, on a $p_x = 1$ et donc $x^{l_x+1} = x^{l_x}$.
3. Il existe un entier ω tel que $x^{\omega+1} = x^\omega$ pour tout $x \in M$.

L'énoncé de ce lemme se traduit en termes imagés de la manière suivante. Dans un groupe, la poêle à frire est donc *sans manche* alors qu'au contraire, elle est *réduite au manche* dans un monoïde apériodique.

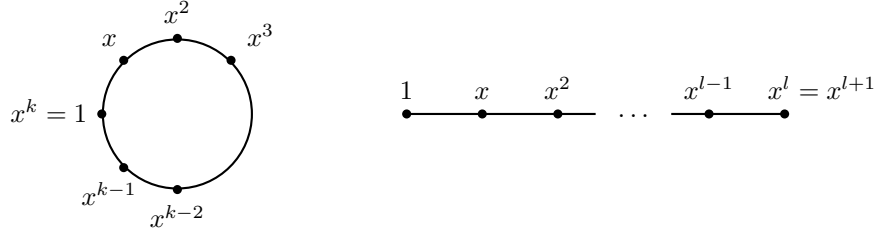


FIG. 1.26 – Cas des groupes et des monoïdes apériodiques

Preuve. Le cas des groupes n'est pas utilisé dans la suite. La preuve est laissée à titre d'exercice.

Pour le cas des monoïdes apériodiques, la preuve de $(1 \Rightarrow 2)$ découle de la remarque suivante. L'ensemble $\{x^{l_x}, x^{l_x+1}, \dots, x^{l_x+p_x-1}\}$ est un monoïde isomorphe à $\mathbb{Z}/p_x\mathbb{Z}$. Par contre, il n'est pas un sous-monoïde de M si $l_x > 0$. Son élément neutre est x^m où m est l'unique entier entre l_x et $l_x + p_x - 1$ tel que $m \equiv 0 \pmod{p_x}$. Si M est apériodique, on a donc nécessairement $p_x = 1$.

La preuve de $(2 \Rightarrow 3)$ est obtenue en posant $\omega = \max\{l_x \mid x \in M\}$.

$(1 \Rightarrow 3)$ Si G est un groupe contenu dans M , on a l'égalité $g^\omega = g^{\omega+1}$. Cette égalité implique immédiatement que g est l'identité de G et que le groupe G est trivial. $\square$

Exemple 1.135. Le monoïde syntaxique du langage $(aa)^+$ est constitué des trois éléments $\{1, \alpha, \alpha^2\}$ avec la relation $\alpha^3 = \alpha$. Le sous-ensemble $\{\alpha, \alpha^2\}$ est isomorphe au groupe $\mathbb{Z}/2\mathbb{Z}$.

Le théorème suivant caractérise en terme de monoïde syntaxique les langages sans étoile. Il permet en particulier de décider effectivement si un langage donné est sans étoile et de calculer explicitement une expression sans étoile si c'est possible.

Théorème 1.136 (Schützenberger 1965). *Pour un langage L , les conditions suivantes sont équivalentes.*

1. Le langage L est sans étoile.
2. Le monoïde syntaxique $M(L)$ de L est apériodique.
3. Le langage L est reconnu par un monoïde apériodique.

Le monoïde syntaxique du langage $(aa)^*$ est le groupe $\mathbb{Z}/2\mathbb{Z}$ qui n'est bien sûr pas apériodique. Le théorème précédent montre donc que ce langage n'est pas sans étoile.

On peut remarquer qu'il suffit qu'un langage soit reconnu par un monoïde apériodique pour que son monoïde syntaxique soit apériodique. On vérifie sans difficulté qu'un monoïde qui divise un monoïde apériodique est encore apériodique. La preuve du théorème se décompose en les sept lemmes techniques suivants.

Le lemme suivant traduit directement en terme de langages le résultat du lemme 1.134.

Lemme 1.137. *Le monoïde syntaxique d'un langage rationnel L est apériodique si et seulement si il existe un entier ω tel que*

$$\forall x, y, z \in A^* \quad xy^\omega z \in L \iff xy^{\omega+1} z \in L.$$

Pour un langage tel que $M(L)$ apériodique, on appelle *indice* et on note $i(L)$ le plus petit entier ω tel que $\forall x, y, z \in A^* \quad xy^\omega z \in L \iff xy^{\omega+1} z \in L$.

Lemme 1.138. *Pour tous langages $K, L \subseteq A^*$ rationnels, on a*

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

Le lemme précédent permet de montrer par récurrence sur (la taille de) l'expression rationnelle sans étoile que le monoïde syntaxique d'un langage sans étoile est apériodique. La réciproque est plus technique.

Dans un monoïde apériodique, l'élément neutre est le seul élément inversible puisque l'ensemble des éléments inversibles est un groupe. Le lemme suivant raffine cette remarque.

Lemme 1.139 (Simplification). *Soient p, m et q trois éléments d'un monoïde apériodique fini. L'égalité $m = pmq$ implique les deux égalités $m = pm = mq$.*

Une conséquence directe du lemme est que l'élément neutre est le seul élément inversible à droite (ou à gauche). Si on a $mp = 1_M$, alors on a aussi $mpm = m$ et $mp = m$ d'après le lemme.

Preuve. D'après le lemme 1.134, il existe un entier ω tel que $s^\omega = s^{\omega+1}$ pour tout élément s du monoïde. On écrit alors les égalités

$$m = pmq = p^\omega mq^\omega = p^\omega mq^{\omega+1} = mq$$

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

Le lemme précédent permet de montrer le lemme suivant.

Lemme 1.140. *Pour tout élément m d'un monoïde apériodique fini M , on a l'égalité*

$$\{m\} = (mM \cap Mm) \setminus J \quad \text{où} \quad J = \{x \in M \mid m \notin MxM\}.$$

Preuve. Tout d'abord m appartient à $mM \cap Mm$ mais il n'appartient pas à J puisque $m \in MmM$. On en déduit l'inclusion $\{m\} \subseteq (mM \cap Mm) \setminus J$ qui est vraie même si M n'est pas apériodique.

On montre maintenant l'inclusion inverse. Soit maintenant m' un élément de $(mM \cap Mm) \setminus J$. Puisque $m' \in mM \cap Mm$, il existe deux éléments p et q de M tels que $m' = mp = qm$. Puisque m' n'appartient pas à J , il existe aussi deux éléments s et r de M tels que $m = sm'r$. En combinant $m' = mp$ et $m = sm'r$, on obtient $m = smpr$ qui donne $m = mpr$ par le lemme de simplification et donc $m = m'r$. En combinant cette dernière égalité avec $m' = qm$, on obtient $m' = qm'r$ et donc $m' = m'r$ par le lemme de simplification. $\square$

Lemme 1.141. *Soit $\mu : A^* \rightarrow M$ un morphisme dans un monoïde apériodique fini M et soit $m \neq 1_M$ un élément de M différent de l'élément neutre. On a alors l'égalité*

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

où les trois ensembles $U, V, W \subset A^*$ de mots sont donnés par

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

et

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

Preuve. Puisque $m \neq 1_M$, les deux ensembles mM et Mm ne contiennent pas l'élément neutre 1_M d'après le lemme de simplification.

D'après le lemme précédent, un mot w vérifie $\mu(w) = m$ si et seulement si $\mu(w) \in mM \cap Mm$ et $m \in M\mu(w)M$.

On commence par montrer que $\mu(w) \in mM$ si et seulement si w se factorise uav tel que $n = \mu(u)$ vérifie $n \notin mM$ et $n\mu(a)M = mM$. L'appartenance $\mu(w) \in Mm$ se traite de manière symétrique. Soit u le plus long préfixe de w tel que $\mu(u) \notin mM$. Ce mot u existe puisque le mot vide est un préfixe de w et $\mu(\varepsilon) = 1_M$ n'appartient pas à mM . Ce mot u est par définition différent de w . Le mot w se factorise $w = uav$ tel que $n = \mu(u)$ vérifie $n \notin mM$ et $n\mu(a)M = mM$.

Soit w un mot tel que $m \notin M\mu(w)M$. On montre que w se factorise soit $w = uav$ où $m \notin M\mu(a)M$ soit $w = uavbt$ tel que $n = \mu(v)$ vérifie $m \in M\mu(a)nM \cap Mn\mu(b)M$ et $m \notin M\mu(a)n\mu(b)M$. Soit v' un plus petit facteur de w tel que $m \notin M\mu(v')M$. Un tel mot existe car $m \notin M\mu(w)M$. Si v' est réduit à une lettre a , on a $w = uav$ où $m \notin M\mu(a)M$. Si v' est de longueur au moins 2, v' s'écrit $v' = avb$ avec les propriétés requises. $\square$

Pour tout m , on pose $r(m) = |MmM|$. La preuve que $\mu^{-1}(m)$ est un langage sans étoile est faite par récurrence sur $|M| - r(m)$:

- si $r(m) = |M|$, alors $m = 1_M$ et on a

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

- sinon, on écrit $\mu^{-1}(m) = (UA^* \cap A^*V) \setminus A^*WA^*$ grâce au lemme précédent. On applique alors l'hypothèse de récurrence aux langages U , V et W .

Pour que cette preuve fonctionne, il reste à prouver les résultats suivants qui assure que les valeurs de n utilisées dans les définitions de U , V et W du lemme précédent vérifient bien $r(n) > r(m)$ et que l'hypothèse de récurrence peut être appliquée.

Lemme 1.142. *Pour tous éléments m et n d'un monoïde apériodique fini M , on a l'implication*

$$\left. \begin{array}{l} m \in nM \\ n \notin mM \end{array} \right\} \implies r(n) > r(m).$$

Preuve. De la relation $m \in nM$, on tire l'inclusion $MmM \subset MnM$. Pour montrer que cette inclusion est stricte, on montre que $n \notin MmM$. Puisque $m \in nM$, il existe un élément p tel que $np = m$. Si on suppose par l'absurde que $n \in MmM$, il existe deux éléments s et r tels que $n = smr$. En combinant ces deux égalités, on déduit que $n = snpr$ et donc $n = npr$ par le lemme de simplification. Ceci conduit à $n = mr$ qui contredit $n \notin mM$. Ceci prouve l'inégalité $r(n) > r(m)$. $\square$

Lemme 1.143. *Pour tous éléments m , n , α et β d'un monoïde apériodique fini M , on a l'implication*

$$\left. \begin{array}{l} m \in M\alpha nM \cap Mn\beta M \\ m \notin M\alpha n\beta M \end{array} \right\} \implies r(n) > r(m).$$

Preuve. De la relation $m \in M\alpha nM$, on tire l'inclusion $MmM \subset MnM$. Pour montrer que cette inclusion est stricte, on montre que $n \notin MmM$. Puisque $m \in M\alpha nM \cap Mn\beta M$, il existe quatre éléments p , q , p' et q' tels que $m = p\alpha nq = p'n\beta q'$. Si on suppose par l'absurde que $n \in MmM$, il existe deux éléments s et r tels que $n = smr$. En combinant cette dernière égalité avec $m = p'n\beta q'$, on obtient $n = sp'n\beta q'r$ et donc $n = n\beta q'r$ par le lemme de simplification. En substituant dans $m = p\alpha nq$, on obtient $m = p\alpha n\beta q'r$ qui contredit $m \notin M\alpha n\beta M$. Ceci prouve l'inégalité $r(n) > r(m)$. $\square$

On pourra consulter [Pin84] pour un exposé plus complet.

Exemple 1.144. 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 $U = a, V = a, W = aa + bb$).

Exercice 1.145. Donner une expression sans étoile pour le langage $(ab + ba)^*$.

1.13 Compléments

1.13.1 Conjecture de Černý

On s'intéresse ici à un problème ouvert connu sous le nom de conjecture de Černý. Pour un état p d'un automate déterministe $\mathcal{A}$ et un mot w , on note $p \cdot w$ l'unique état q , s'il existe, tel qu'il y ait un chemin dans $\mathcal{A}$ de p à q étiqueté par w . Un automate $\mathcal{A}$ est dit *synchronisant* s'il existe un mot w tel que $p \cdot w$ est égal à un état constant q pour tout état p de $\mathcal{A}$. La lecture du mot w ramène

l'automate dans l'état q quelque soit l'état de départ. Le mot w est alors dit *synchronisant* pour $\mathcal{A}$.

Une question naturelle est de savoir la longueur minimale d'un mot synchronisant d'un automate à n états. La conjecture de Černý énonce qu'un tel mot est de longueur au plus $(n-1)^2$. Cette borne ne peut être améliorée. Pour tout entier n , il existe un automate synchronisant $\mathcal{C}_n$ dont le mot synchronisant le plus court est de longueur $(n-1)^2$. La meilleure borne connue pour l'instant est $(n-1)^3/6$ même si la conjecture a déjà été prouvée pour des familles particulières d'automates.

L'automate $\mathcal{C}_n$ est défini de la manière suivante. Son ensemble d'états est $Q_n = \{0, \dots, n-1\}$. La lettre a laisse invariant tous les états sauf l'état 0 qui est envoyé sur 1. La lettre b effectue une permutation circulaire des états.

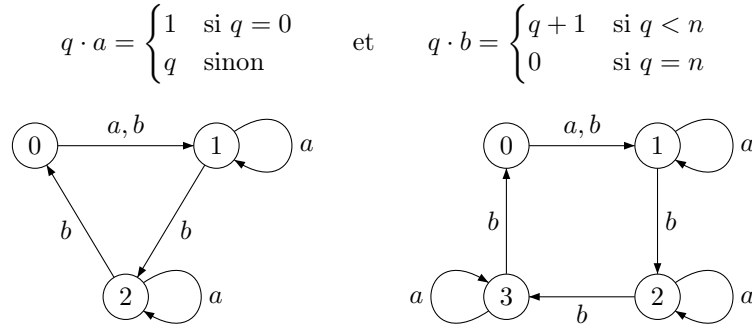


FIG. 1.27 – Les automates $\mathcal{C}_3$ et $\mathcal{C}_4$

Exemple 1.146. Les automates $\mathcal{C}_3$ et $\mathcal{C}_4$ de la figure 1.27 sont synchronisants. Ils sont effectivement synchronisés par les mots ab^2a et ab^3ab^3a de longueur 4 et 9. Il est un plus délicat de prouver que ce sont les mots synchronisants les plus courts de $\mathcal{C}_3$ et $\mathcal{C}_4$. On montre de même que le mot synchronisant le plus court de $\mathcal{C}_n$ est le mot $(ab^{n-1})^{n-2}a$ de longueur $(n-1)^2$.

1.13.2 Rationnels d'un monoïde quelconque

Jusqu'à maintenant, on a uniquement étudié les parties des monoïdes libres. La définition d'une partie rationnelle s'étend à n'importe quel monoïde. Il est intéressant de considérer d'autres monoïdes comme les monoïdes $A^* \times B^*$ ou plus généralement $A_1^* \times \dots \times A_k^*$, les monoïdes commutatifs libres ou le groupe libre.

Les opérations rationnelles s'étendent de manière évidente aux parties d'un monoïde quelconque. Pour des parties K et L d'un monoïde M , le produit KL et l'étoile K^* sont définies de la manière suivante.

$$KL = \{kl \mid k \in K \text{ et } l \in L\} \quad \text{et} \quad K^* = \{k_1 \dots k_n \mid k_1, \dots, k_n \in K\}.$$

L'étoile K^* est en fait le plus petit sous-monoïde de M contenant K et il est appelée le sous-monoïde *engendré* par K . La notation comporte une certaine ambiguïté puisque K^* peut noter le monoïde libre sur l'alphabet K où le sous-monoïde engendré par K dans M . Cette confusion est sans conséquence dans la mesure où on identifie très souvent une suite finie $k_1, \dots, k_n$ avec son produit $k_1 \dots k_n$ dans M . L'application qui envoie toute suite $k_1, \dots, k_n$ sur $k_1 \dots k_n$ est d'ailleurs un morphisme du monoïde libre K^* dans le monoïde M .

Définition 1.147. Soit M un monoïde. La famille notée $\text{Rat } M$ des *parties rationnelles* est la plus petite famille de parties de M telle que

- $\emptyset \in \text{Rat } M$ et $\{a\} \in \text{Rat } M$ pour toute lettre a ;
- $\text{Rat } M$ est close pour les opérations rationnelles (l'union, le produit et l'étoile).

Un automate sur un monoïde M est un automate dont les étiquettes des transitions sont des éléments de M . Plus formellement, un automate $\mathcal{A}$ sur M est un quintuplet (Q, M, E, I, F) où Q est un ensemble fini d'états et E est un sous-ensemble de $Q \times M \times Q$. L'*étiquette* d'un chemin

$$q_0 \xrightarrow{m_1} q_1 \xrightarrow{m_2} \cdots \xrightarrow{m_n} q_n$$

est le produit $m_1 \cdots m_n$ dans M des étiquettes des transitions qui constituent le chemin. Le *comportement* d'un automate est l'ensemble des étiquettes des chemins acceptants (c'est-à-dire commençant dans un état initial et se terminant dans un état final). Le théorème de Kleene s'étend facilement à un monoïde quelconque. La preuve faite dans le cas d'un monoïde libre reste valide.

Théorème 1.148. Une partie d'un monoïde M est rationnelle si et seulement si elle est le comportement d'un automate sur M .

La construction du groupe libre ainsi que ses parties rationnelles sont décrites à la section 2.7.3 (p. 93).

Pour illustrer ces notions, on va donner quelques exemples dans le cas d'un monoïde $A^* \times B^*$ pour deux alphabets A et B . Une partie rationnelle de $A^* \times B^*$ est aussi appelée une *relation rationnelle* ou encore une *transduction rationnelle*. Un élément de $A^* \times B^*$ est une paire (u, v) de mots sur A et B qui est souvent notée $u|v$ en particulier pour les étiquettes des automates.

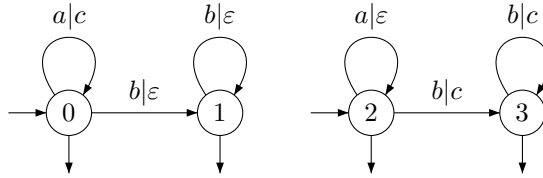


FIG. 1.28 – Un transducteur pour $(a, c)^*(b, \varepsilon)^* + (a, \varepsilon)^*(b, c)^*$

Exemple 1.149. La partie rationnelle $(a, c)^*(b, \varepsilon)^* + (a, \varepsilon)^*(b, c)^*$ du monoïde $\{a, b\}^* \times \{c\}^*$ est le comportement du transducteur de la figure 1.28.

Exemple 1.150. L'automate de l'addition donné à la figure 3.10 (p. 128) est en fait un transducteur dont le comportement est la partie du monoïde $\{0, 1\}^* \times \{0, 1\}^* \times \{0, 1\}^*$ constituée des triplets (u, v, w) tels que la valeur binaire de w est égale à la somme des valeurs binaires de u et v .

Une partie K d'un monoïde M est dite *reconnaissable* s'il existe un morphisme μ de M dans un monoïde fini N et une partie P de N tels que $K = \mu^{-1}(P)$. L'ensemble des parties reconnaissables de M est notée $\text{Rec } M$. Il découle directement de la définition que la famille des parties reconnaissables est close pour les opérations booléennes (union, intersection et complémentation). Un monoïde est dit *finiment engendré* s'il est égal au sous-monoïde engendré par une de ses parties finies.

Proposition 1.151. *Si M est finiment engendré, alors $\text{Rec } M \subseteq \text{Rat } M$.*

Les deux familles $\text{Rec } M$ et $\text{Rat } M$ coïncident lorsque M est un monoïde libre mais l'inclusion de la proposition précédente peut être stricte si M n'est pas libre. Si M est un monoïde de la forme $A^* \times B^*$, les parties reconnaissables sont les unions finies d'ensembles de la forme $K \times L$ où K et L sont respectivement des parties rationnelles de A^* et B^* . La diagonale $\Delta = \{(w, w) \mid w \in A^*\} = \{(a, a) \mid a \in A\}^*$ est une partie rationnelle de $A^* \times A^*$ qui n'est pas reconnaissable.

Preuve. Soit A un ensemble fini qui engendre M et soit μ un morphisme de M dans un monoïde fini N tel que $K = \mu^{-1}(P)$ pour une partie $P \subseteq N$. On construit l'automate $\mathcal{A}$ sur M égal à $(N, A, E, \{1_N\}, P)$ dont l'ensemble des transitions est $E = \{(n, a, n\mu(a)) \mid a \in A \text{ et } n \in N\}$. Il est facile de vérifier que pour tout m de M , il existe un chemin d'étiquette m de 1_N à $\mu(m)$. $\square$

Proposition 1.152. *Soit M un monoïde. Si K est une partie reconnaissable et L une partie rationnelle de M , alors $K \cap L$ est une partie rationnelle.*

Preuve. Soit μ un morphisme de M dans un monoïde fini N tel que $K = \mu^{-1}(P)$ pour une partie $P \subseteq N$. Soit $\mathcal{A} = (Q, M, E, I, F)$ un automate sur M dont le comportement est L . On construit un automate $\mathcal{A}'$ dont le comportement est $K \cap L$. L'ensemble des états est $Q \times N$, les états initiaux sont ceux de $I \times \{1_N\}$ et les états finaux sont ceux de $F \times P$. Les transitions sont les triplets de la forme $((p, n), m, (q, n\mu(m)))$ où (p, m, q) est une transition de $\mathcal{A}$. $\square$

Chapitre 2

Langages algébriques

Les langages algébriques, aussi appelés langages *hors contexte* (*context-free* en anglais) constituent le second niveau de la hiérarchie de Chomsky. Il furent initialement introduits pour modéliser les langues naturelles. Cette approche a été relativement abandonnée même si des grammaires locales sont encore utilisées pour décrire des constructions grammaticales. Par contre, les grammaires se sont révélées particulièrement adaptées pour décrire la syntaxe des langages de programmation. D'un point de vue plus pratique, des outils tels **yacc** ou **bison** permettent de construire automatiquement un analyseur syntaxique à partir d'une grammaire qui satisfait quelques hypothèses supplémentaires.

Une première partie de ce chapitre traite des grammaires qui permettent d'engendrer ces langages algébriques. Une seconde partie est consacrée aux automates à pile qui acceptent les langages algébriques. Ces automates étendent les automates finis en utilisant une mémoire externe sous la forme d'une pile. Les liens avec les systèmes d'équations sont également abordés et utilisés pour prouver le théorème de Parikh. Pour tout ce chapitre, on se reportera à [Aut87] ou à [ABB97].

2.1 Grammaires

2.1.1 Définitions et exemples

Une grammaire est en quelque sorte un système de réécriture qui engendre un ensemble de mots à partir d'un axiome. Afin que les réécritures terminent, l'alphabet est partagé en lettres terminales et en lettres non terminales appelées variables. Seules les variables peuvent être réécrites.

Définition 2.1 (Grammaire). Une *grammaire* G est un triplet (A, V, P) où A et V sont des alphabets finis et disjoints et où P est une partie finie de $V \times (A \cup V)^*$. Les symboles de A sont appelés *terminaux* et ceux de V sont appelés *non terminaux* ou *variables*. Les éléments de P sont appelés *règles*. Chaque règle est une paire (X, u) où X est une variable et u est un mot sur l'alphabet $A + V$.

On note $X \rightarrow u$ lorsque $(X, u) \in P$ est une règle de la grammaire. Par extension, si $(X, u_1), \dots, (X, u_n)$ sont des règles, on écrit $X \rightarrow u_1 + \dots + u_n$.

Les éléments X et u d'une règle $X \rightarrow u$ sont respectivement appelés *membre gauche* et *membre droit* de la règle.

Exemple 2.2. Soit la grammaire $G = (A, V, P)$ définie par

$$A = \{a, b\}, \quad V = \{S\}, \quad P = \{S \rightarrow aSb + \varepsilon\}$$

On verra que le langage engendré par cette grammaire est $\{a^n b^n \mid n \geq 0\}$.

Lorsqu'une grammaire est donnée de manière explicite, on se contente souvent de donner les règles. Les variables de la grammaire sont alors implicitement les symboles qui apparaissent comme membre gauche des règles et toutes les autres lettres sont implicitement des lettres terminales. Pour la grammaire donnée ci-dessus, on aurait pu simplement écrire $P = \{S \rightarrow aSb + \varepsilon\}$.

Définition 2.3 (Dérivation). Soit $G = (A, V, P)$ une grammaire et soient u et v deux mots sur $A + V$. On dit que u se *dérive* en (ou *produit*) v et on note $u \rightarrow v$ lorsque il existe $\alpha, \beta \in (A + V)^*$ et $X \in V$ tels que :

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

On note $u \xrightarrow{k} v$ s'il existe des mots $u_1, u_2, \dots, u_{k-1}$ tels que $u \rightarrow u_1 \rightarrow \dots \rightarrow u_{k-1} \rightarrow v$. Plus généralement, on notera $u \xrightarrow{*} v$ s'il existe un entier $k \geq 0$ tel que $u \xrightarrow{k} v$. La relation $\xrightarrow{*}$ est donc la clôture réflexive et transitive de la relation $\rightarrow$.

Les langages algébriques sont aussi appelés langages hors contexte parce que lors d'une dérivation, le remplacement de la variable X par w dans le mot $u = \alpha X \beta$ est indépendant des contextes α et β . Il existe aussi des grammaires contextuelles (cf. section 3.6.2) dont les membres gauches de règle ne sont plus simplement des variables mais des mots qui permettent de spécifier le contexte de remplacement d'une variable.

Définition 2.4 (Langage engendré). Soit $G = (A, V, P)$ une grammaire et soit u un mot sur $(A + V)^*$. On note respectivement $\widehat{L}_G(u)$ et $L_G(u)$ les langages $\{v \in (A + V)^* \mid u \xrightarrow{*} v\}$ et $\widehat{L}_G(u) \cap A^*$.

Dans une grammaire G , une variable S_0 appelée *axiome* est parfois distinguée. Le langage engendré par la grammaire est alors le langage $L_G(S_0)$.

Définition 2.5 (Langage algébrique). Un langage est dit *algébrique* s'il peut être engendré par une grammaire, c'est-à-dire s'il est égal $L_G(S)$ pour une variable S d'une grammaire G .

Grâce au lemme fondamental 2.8 et à la clôture des langages algébriques par produit (cf. proposition 2.46), on constate que $L_G(u)$ est encore algébrique pour tout mot $u \in (A + V)^*$. La notation L_G peut être étendue aux langages en posant $L_G(K) = \bigcup_{u \in K} L_G(u)$ pour $K \subset (A + V)^*$. La clôture des langages algébriques par substitution algébrique (cf. proposition 2.48) montre que $L_G(K)$ reste algébrique quand K est algébrique.

Exemple 2.6. Quelques exemples de grammaires et de langages algébriques.

1. $G : S \rightarrow aS + b \quad L_G(S) = a^*b$
2. $G : S \rightarrow aSa + bSb + a + b + \varepsilon \quad L_G(S) = \{w \mid w \text{ est un palindrome}\}$

3. Grammaire

$$\begin{cases} A = \{a, b, c\} \\ S \rightarrow AS + \varepsilon \\ A \rightarrow AB + a \\ B \rightarrow BC + b \\ C \rightarrow CA + c \end{cases}$$

4. Langage de Dyck sur n paires de parenthèses.

$$\begin{cases} A_n = \{a_1, \dots, a_n, \bar{a}_1, \dots, \bar{a}_n\} \\ S \rightarrow ST + \varepsilon \\ T \rightarrow a_1 S \bar{a}_1 + \dots + a_n S \bar{a}_n \end{cases}$$

Les langages $D_n = L_G(T)$ et $D_n^* = L_G(S)$ sont respectivement appelés *langage de Dyck primitif* et *langage de Dyck*. On vérifie que D_n^* est l'ensemble des mots qui se réduisent au mot vide en appliquant les réductions de la forme $a_i \bar{a}_i \rightarrow \varepsilon$.

5. Langage de Luckasiewicz

$$S \rightarrow aSS + \bar{a}$$

Le langage $\mathbb{L} = L_G(S)$ est l'ensemble des mots w tels que $|w|_{\bar{a}} = |w|_a + 1$ et $|u|_{\bar{a}} \leq |u|_a$ pour tout préfixe (propre) u de w . On a l'égalité $\mathbb{L} = D_1^* \bar{a}$.

6. Langage de Goldstine

$$L = \{a^{n_0} b a^{n_1} b \dots a^{n_k} b \mid k \geq 0 \text{ et } \exists j \geq 0, n_j \neq j\}$$

Le langage L est égal à $L_G(S)$ où la grammaire G est constituée des règles suivantes.

$$\begin{aligned} T_0 &\rightarrow aT_0 + \varepsilon \\ T_1 &\rightarrow T_0 b \\ T_2 &\rightarrow T_1 T_2 + \varepsilon \\ T_3 &\rightarrow T_2 T_3 a + T_1 T_2 + aT_0 \\ S &\rightarrow T b T_2 \end{aligned}$$

On vérifie facilement que $L_G(T_0) = a^*$, $L_G(T_1) = a^* b$, $L_G(T_2) = (a^* b)^*$, $L_G(T_3) = \{a^{n_0} b a^{n_1} b \dots a^{n_k} b \mid k \geq 0 \text{ et } n_k \neq k\}$ et $L_G(S) = L$.

Exemple 2.7. Quelques exemples et contre-exemples laissés en exercice.

- $\{w \# w' \mid w \neq w'\}$ n'est pas algébrique.
- $\{w \# w' \mid w = w'\}$ n'est pas algébrique.
- $\{w \# w' \mid w \neq w' \text{ et } |w| = |w'|\}$ n'est pas algébrique.
- $\{w w' \mid w \neq w' \text{ et } |w| = |w'|\}$ est algébrique.

Lemme 2.8 (Fondamental). *Soit $G = (A, V, P)$ une grammaire et soit u et v deux mots sur $(A + V)^*$. On suppose que u se factorise $u = u_1 u_2$. Il existe une dérivation $u \xrightarrow{k} v$ de longueur k si et seulement si v se factorise $v = v_1 v_2$ et s'il existe deux dérivations $u_1 \xrightarrow{k_1} v_1$ et $u_2 \xrightarrow{k_2} v_2$ où $k = k_1 + k_2$.*

Une manière un peu moins précise d'énoncer le lemme fondamental est d'écrire que pour tous mots u et v sur $(A + V)^*$, on a l'égalité

$$\widehat{L}_G(uv) = \widehat{L}_G(u) \widehat{L}_G(v).$$

Preuve. Il est aisé de combiner les dérivations $u_1 \xrightarrow{k_1} v_1$ et $u_2 \xrightarrow{k_2} v_2$ pour obtenir une dérivation $u_1 u_2 \xrightarrow{k} v_1 v_2$ de longueur $k_1 + k_2$. La réciproque se montre par induction sur k . $\square$

2.1.2 Grammaires réduites

La notion de grammaire réduite est l'analogue pour les grammaires de la notion d'automate émondé. De manière intuitive, une grammaire est réduite si elle ne contient pas de variable trivialement inutile.

Définition 2.9 (Grammaire réduite). Une grammaire $G = (A, V, P)$ est dite *réduite* pour $S_0 \in V$ si

1. pour tout $S \in V$, $L_G(S) \neq \emptyset$,
2. pour tout $S \in V$, il existe $u, v \in (A + V)^*$ tels que $S_0 \xrightarrow{*} uSv$.

La première condition signifie que toute variable peut produire un mot et la seconde condition signifie que toute variable peut être atteinte à partir de la variable S_0 .

Proposition 2.10 (Réduction). *Pour toute grammaire $G = (A, V, P)$ et toute variable $S_0 \in V$, il existe une grammaire G' réduite pour T telle que $L_G(S_0) = L_{G'}(T)$.*

Preuve. Pour réduire, on applique successivement les deux étapes suivantes. Il est à noter que l'ordre des étapes est important.

La première étape consiste à supprimer des variables S telles que $L_G(S) = \emptyset$. On définit par récurrence une suite $(U_n)_{n \geq 0}$ d'ensembles de symboles de $A + V$ par

$$\begin{aligned} U_0 &= A, \\ U_{n+1} &= U_n \cup \{S \in V \mid S \rightarrow w \text{ et } w \in U_n^*\} \quad \text{pour } n \geq 0. \end{aligned}$$

Puisque l'ensemble $A + V$ est fini et que la suite $(U_n)_{n \geq 0}$ est croissante, celle-ci est constante à partir d'un certain rang. Soit U l'ensemble $\bigcup_{n \geq 0} U_n$. On prouve par récurrence sur la longueur d'une plus petite dérivation que $U \cap V$ est l'ensemble $\{S \mid L_G(S) \neq \emptyset\}$. On supprime ainsi les variables qui ne sont pas dans $U \cap V$.

La seconde étape consiste à supprimer les variables inaccessibles à partir de l'axiome S_0 . On définit par récurrence une suite $(W_n)_{n \geq 0}$ d'ensembles de variables par

$$\begin{aligned} W_0 &= \{S_0\}, \\ W_{n+1} &= W_n \cup \{S \in V \mid \exists S' \in W_n, S' \rightarrow uSv \text{ avec } u, v \in (A + V)^*\}. \end{aligned}$$

Puisque V est fini et que la suite $(W_n)_{n \geq 0}$ est croissante, celle-ci est constante à partir d'un certain rang. On montre que l'ensemble $W = \bigcup_{n \geq 0} W_n$ est égal à l'ensemble $\{S \mid S_0 \xrightarrow{*} uSv \text{ avec } u, v \in A^*\}$ en utilisant le fait que $L_G(S)$ est non vide pour toute variable S . Ainsi, W ne garde que les variables accessibles depuis S_0 . $\square$

2.1.3 Grammaires propres

Définition 2.11 (Grammaire propre). Une grammaire $G = (A, V, P)$ est *propre* si elle ne contient aucune règle de la forme $S \rightarrow \varepsilon$ où de la forme $S \rightarrow S'$ pour $S, S' \in V$.

La proposition suivante établit que toute grammaire peut être rendue propre en perdant éventuellement le mot vide.

Proposition 2.12. *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 $L_{G'}(S') = L_G(S) \setminus \{\varepsilon\}$.*

La preuve de la proposition donne une construction effective d'une grammaire propre. Elle utilise la notion de substitution qui est maintenant définie.

Définition 2.13 (Substitution). Soient A et B deux alphabets. Une *substitution* de A^* dans B^* est un morphisme $\sigma : A^* \rightarrow \mathfrak{P}(B^*)$ où $\mathfrak{P}(B^*)$ est muni du produit des langages.

Preuve. En utilisant la proposition précédente, on peut supposer que la grammaire G est réduite. La construction se décompose en les deux étapes suivantes. L'ordre des étapes a de nouveau son importance.

La première étape consiste à supprimer les règles de la forme $S \rightarrow \varepsilon$. On commence par calculer l'ensemble $U = \{S \mid S \xrightarrow{*} \varepsilon\}$ des variables qui produisent le mot vide. On définit par récurrence la suite $(U_n)_{n \geq 0}$ d'ensembles de variables par

$$\begin{aligned} U_0 &= \{S \mid S \rightarrow \varepsilon\}, \\ U_{n+1} &= U_n \cup \{S \mid S \rightarrow w \text{ et } w \in U_n^*\}. \end{aligned}$$

La suite $(U_n)_{n \geq 0}$ est constante à partir d'un certain rang. On montre par récurrence que $U = \bigcup_{n \geq 0} U_n$ est l'ensemble $\{S \mid S \xrightarrow{*} \varepsilon\}$ des variables qui produisent le mot vide. On introduit la substitution σ de $(A + V)^*$ dans $(A + V)^*$ par

$$\begin{aligned} \sigma(a) &= a \quad \text{pour } a \in A \\ \sigma(S) &= \begin{cases} S + \varepsilon & \text{si } S \xrightarrow{*} \varepsilon \\ S & \text{sinon} \end{cases} \quad \text{pour } S \in V. \end{aligned}$$

La première étape consiste alors à procéder aux deux transformations suivantes sur la grammaire G . L'idée intuitive est de remplacer par le mot vide dans les membres droits de règles les variables qui produisent le mot vide.

1. Supprimer les règles $S \rightarrow \varepsilon$.
2. Ajouter toutes les règles $S \rightarrow u$ où $S \rightarrow w$ est une règle et $u \in \sigma(w)$.

On montre par récurrence sur la longueur des dérivations que dans la nouvelle grammaire, chaque variable engendre les mêmes mots au mot vide près.

La seconde étape consiste à supprimer les règles de la forme $S \rightarrow S'$ où S et S' sont deux variables. Puisqu'aucune variable ne produit le mot vide, on a une dérivation $S \xrightarrow{*} S'$ si et seulement si il existe des variables $S_1, \dots, S_n$ telles que $S_1 = S$, $S_n = S'$ et $S_i \rightarrow S_{i+1}$ pour tout $1 \leq i \leq n-1$. La relation $\xrightarrow{*}$ restreinte aux variables est donc la clôture transitive et réflexive de la relation $\rightarrow$ restreinte aux variables.

La seconde étape consiste alors à procéder aux deux transformations suivantes sur la grammaire G . L'idée intuitive est de remplacer par une seule dérivation une suite de dérivations $S_i \rightarrow S_{i+1}$ suivie d'une dérivation $S_n \rightarrow w$ où $w \notin V$.

1. Supprimer les règles $S \rightarrow S'$.
2. Ajouter les règles $S \rightarrow w$ où $S \xrightarrow{*} S'$, et $S' \rightarrow w$ est une règle avec $w \notin V$.

On vérifie sans difficulté que la grammaire obtenue est propre et qu'elle engendre le même langage.

Un autre méthode consiste à d'abord quotienter l'ensemble V par la relation d'équivalence $\equiv$ définie par

$$S \equiv S' \stackrel{\text{def}}{\iff} (S \xrightarrow{*} S' \text{ et } S' \xrightarrow{*} S).$$

La grammaire obtenue engendre le même langage car $S \equiv S'$ implique que $L_G(S) = L_G(S')$. La relation $\xrightarrow{*}$ devient alors un ordre sur le quotient. On supprime alors les règles $S \rightarrow S'$ où S est maximal pour cet ordre en ajoutant les règles $S' \rightarrow w$ pour chaque règle $S \rightarrow w$. Le processus est itéré avec les nouvelles variables devenues maximales pour l'ordre $\xrightarrow{*}$. $\square$

Exemple 2.14. La grammaire $\{S \rightarrow aS\bar{a}S + \varepsilon\}$ est transformée en la grammaire $\{S \rightarrow aS\bar{a}S + aS\bar{a} + a\bar{a}S + a\bar{a}\}$ par la première étape de la preuve.

2.1.4 Forme normale quadratique

Définition 2.15 (Forme normale quadratique). Une grammaire est en *forme normale quadratique* si toutes ses règles sont d'une des formes suivantes :

- $S \rightarrow S_1S_2$ où $S_1, S_2 \in V$
- $S \rightarrow a$ où $a \in A$

Il faut remarquer qu'une grammaire en forme quadratique est nécessairement propre. En particulier, elle ne peut pas engendrer le mot vide.

Proposition 2.16 (Forme normale quadratique). *Pour toute grammaire $G = (A, V, P)$ et tout $S \in V$, il existe une grammaire en forme normale quadratique $G' = (A, V', P')$ et $S' \in V'$ telle que $L_{G'}(S') = L_G(S) \setminus \{\varepsilon\}$.*

Preuve. Grâce à la proposition 2.12, on peut supposer que la grammaire G est propre. Ensuite, la construction se décompose à nouveau en deux étapes.

la première étape consiste à se ramener à une grammaire où tous les membres droits de règles sont soit une lettre terminale soit un mot formé uniquement de variables, c'est-à-dire d'une des formes suivantes.

- $S \rightarrow a$ où $a \in A$
- $S \rightarrow S_1S_2 \cdots S_n$ où $S_1, \dots, S_n \in V$

Grâce à la proposition précédente, on peut supposer que G est grammaire propre. Soit $V' = \{V_a \mid a \in A\}$ un ensemble de nouvelles variables en bijection avec A . Soit G' la grammaire $(A, V \cup V', P')$ où P' est l'ensemble $\{V_a \rightarrow a \mid a \in A\} \cup \{S \rightarrow \sigma(w) \mid S \rightarrow w \in P\}$ et où σ est la substitution définie par $\sigma(S) = S$ pour $S \in V$ et $\sigma(a) = V_a$ pour $a \in A$.

Dans la seconde étape, les règles $S \rightarrow S_1 \cdots S_n$ avec $n > 2$ sont supprimées et remplacées par d'autres nouvelles règles. Dans ce but, on introduit les nouvelles

variables $S'_1, \dots, S'_{n-1}$ et on remplace $S \rightarrow S_1 \cdots S_n$ par les n règles suivantes.

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

□

2.2 Systèmes d'équations

On va voir que les langages algébriques sont les solutions des systèmes polynomiaux, ce qui justifie la terminologie. On utilisera cette approche pour montrer le théorème de Parikh qui établit que tout langage algébrique est commutativement équivalent à un langage rationnel.

2.2.1 Substitutions

À une grammaire, on peut associer un système d'équations en langages. Soit par exemple la grammaire $G = (A, V, P)$ avec $A = \{a, b\}$, $V = \{X_1, X_2\}$ et P donné par les règles suivantes :

$$\begin{cases} X_1 \longrightarrow X_1 X_2 + \varepsilon \\ X_2 \longrightarrow a X_2 + b. \end{cases}$$

On associe à cette grammaire le système

$$\begin{cases} L_1 = L_1 L_2 + \varepsilon \\ L_2 = a L_2 + b \end{cases}$$

en les langages L_1 et L_2 sur A .

On introduit quelques notations pratiques qui aident à formaliser ceci. Soit $G = (A, V, P)$ une grammaire dont l'ensemble des variables est $V = \{X_1, \dots, X_n\}$ et soit $L = (L_1, \dots, L_n)$ un n -uplet de langages sur A . On définit successivement des opérations de substitution par

$$\begin{aligned} \varepsilon(L) &= \{\varepsilon\} \\ a(L) &= \{a\} \text{ pour } a \in A \\ X_i(L) &= L_i \\ xy(L) &= x(L)y(L) \text{ pour } x, y \in (A + V)^* \\ K(L) &= \bigcup_{w \in K} w(L) \text{ pour } K \subset (A + V)^* \end{aligned}$$

On notera la cohérence dans le cas particulier $L = (X_1, \dots, X_n)$ avec la notation $\alpha(X_1, \dots, X_n)$ marquant une simple dépendance en $X_1, \dots, X_n$.

Le point important est de voir qu'on l'on procède bel est bien à une substitution des X_i par les L_i correspondants.

2.2.2 Système d'équations associé à une grammaire

Définition 2.17. Soit $G = (A, V, P)$ une grammaire où $V = \{X_1, \dots, X_n\}$. Le système associé à G est formé par les n équations

$$L_i = \sum_{X_i \longrightarrow w} w(L) \quad \text{pour } 1 \leq i \leq n$$

en les variables $L_1, \dots, L_n$.

Notation 2.18. On notera également

$$L_G := (L_G(X_1), \dots, L_G(X_n)).$$

La proposition suivante est essentiellement une reformulation du lemme fondamental 2.8 (p. 63).

Proposition 2.19. Pour tout mot $w \in (A + V)^*$, on a

$$L_G(w) = w(L_G).$$

Preuve. Le résultat est déjà vrai sur les symboles de base ε , a et X_i :

$$\begin{aligned} L_G(\varepsilon) &= \varepsilon = \varepsilon(L_G) \\ L_G(a) &= a = a(L_G) \\ L_G(X_i) &= X_i(L_G). \end{aligned}$$

On écrit $w = w_1 \dots w_n$ où $w_1, \dots, w_n$ sont les lettres de w et on applique les règles de substitution données ci-dessus.

$$\begin{aligned} L_G(w) &= L_G(w_1 \dots w_n) \\ &= L_G(w_1) \dots L_G(w_n) \\ &= w_1(L_G) \dots w_n(L_G) \\ &= w_1 \dots w_n(L_G) \\ &= w(L_G). \end{aligned}$$

□

Voyons à présent le lien entre solutions du système associé et langages engendrés par les X_i .

2.2.3 Existence d'une solution pour $\mathcal{S}(G)$

Proposition 2.20 (Minimalité des langages engendrés). Soit $G = (A, V, P)$ une grammaire avec $V = \{X_1, \dots, X_n\}$. Alors L_G est la solution minimale (pour l'inclusion composante par composante) du système $\mathcal{S}(G)$ associé à G .

Lemme 2.21. Soit $G = (A, V, P)$ une grammaire et soit L une solution du système $\mathcal{S}(G)$. Si les deux mots w et w' sur $(A + V)^*$ vérifient $w \xrightarrow{*} w'$, alors $w(L) \supset w'(L)$.

Preuve. Il suffit de montrer le résultat pour une seule dérivation. Le cas général se prouve par récurrence sur la longueur de la dérivation. Supposons que $w = uX_iv \rightarrow uzv = w'$ où $X_i \rightarrow z$ est une règle de la grammaire. Puisque L est solution de $\mathcal{S}(G)$, on a l'égalité $L_i = \sum_{X_i \rightarrow \gamma} \gamma(L)$ et donc l'inclusion $L_i \supset z(L)$. Il en découle que $w(L) \supset w'(L)$. $\square$

On peut maintenant procéder à la preuve de la proposition.

Preuve. – L_G est bien une solution de $\mathcal{S}(G)$:

$$\begin{aligned} L_G(X_i) &= \sum_{X_i \rightarrow \alpha} L_G(\alpha) \\ &= \sum_{X_i \rightarrow \alpha} \alpha(L_G) \end{aligned}$$

d'après la proposition 2.19.

- Par ailleurs, soit $L = (L_1, \dots, L_n)$ une solution de $\mathcal{S}(G)$. Pour tout mot $w \in L_G(X_i)$, on a par définition une dérivation $X_i \xrightarrow{*} w$ et par le lemme précédent l'inclusion $w(L) \subset X_i(L)$. Comme $w \in A^*$, on $w(L) = w$ et donc $w \in L_i$. Ceci prouve l'inclusion $L_G(X_i) \subset L_i$ pour tout $1 \leq i \leq n$. $\square$

La proposition précédente justifie a posteriori la terminologie des langages algébriques. Il sont les solutions minimales des systèmes d'équations polynomiales.

Bien que la proposition nous donne toujours l'existence d'une solution de $\mathcal{S}(G)$, l'unicité est généralement fautive. Considérons par exemple la grammaire $X \rightarrow XX$ dont le système associé est l'équation $L = LL$. Une solution est $L_G(X) = \emptyset$, mais tous les langages de la forme L^* pour $L \subset A^*$ sont également solutions.

La proposition suivante donne des conditions pour avoir l'unicité.

2.2.4 Unicité des solutions propres

Définition 2.22 (Solution propre). Une solution L de $\mathcal{S}(G)$ est dite *propre* si tous les L_i sont propres, c'est-à-dire ne contiennent pas le mot vide ε .

Proposition 2.23 (Unicité des solutions propres). *Soit G une grammaire propre. Alors L_G est l'unique solution propre de $\mathcal{S}(G)$.*

Preuve. Puisque G est propre, L_G est propre, et la proposition précédente nous dit que L_G est une solution de $\mathcal{S}(G)$. Ainsi L_G est une solution propre.

Pour un entier l , on introduit la relation $\overset{l}{\sim}$ qui capture le fait que deux langages coïncident pour les mots de longueur inférieure à l . Pour deux langages K et K' et un entier l , on note $K \overset{l}{\sim} K'$ si l'égalité

$$\{w \in K \mid |w| \leq l\} = \{w \in K' \mid |w| \leq l\},$$

est vérifiée. Cette notation est étendue aux n -uplets composante par composante. Soient $L = (L_1, \dots, L_n)$ et $L' = (L'_1, \dots, L'_n)$ deux solutions propres de G . On va montrer par récurrence sur l que $L \overset{l}{\sim} L'$ pour tout entier $l \geq 0$ ce qui prouve que $L = L'$.

Pour $1 \leq i \leq n$, on introduit le langage fini $S_i = \{w \mid X_i \rightarrow w\}$ de sorte que le système $\mathcal{S}(G)$ s'écrit maintenant

$$L_i = S_i(L) \quad \text{pour } 1 \leq i \leq n.$$

Comme L et L' sont propres, aucune de leurs composantes ne contient le mot vide et on a $L \stackrel{0}{\sim} L'$.

Supposons $L \stackrel{l}{\sim} L'$ pour $l \geq 0$ et montrons $L \stackrel{l+1}{\sim} L'$, c'est-à-dire

$$L_i \stackrel{l+1}{\sim} L'_i \quad \text{pour } 1 \leq i \leq n$$

ou encore

$$S_i(L) \stackrel{l+1}{\sim} S_i(L') \quad \text{pour } 1 \leq i \leq n.$$

Soit donc $1 \leq i \leq n$ fixé, et soit $w \in S_i$ (c'est-à-dire un mot tel que $X_i \rightarrow w$), que l'on écrit $w = w_1 \dots w_p$ où chaque w_j est soit une lettre de A , soit une variable $X_k \in V$. L'entier p est non nul car la grammaire G est propre.

Soit ensuite $u \in w(L)$ de longueur inférieure à $l+1$. Il se décompose $u = u_1 \dots u_p$ où $u_j = w_j$ si w_j appartient à A et $u_j \in L_k$ si $w_j = X_k$. Comme la grammaire est propre chaque mot u_j est non vide. Il s'ensuit que chaque mot u_j vérifie $|u_j| \leq l$. Sinon on aurait $p = 1$ et la grammaire G contiendrait une règle $X_i \rightarrow X_k$. Ceci est exclu car la grammaire G est propre.

Par hypothèse de récurrence, on en déduit que $u_j \in L'_k$ si $w_j = X_k$ et que u appartient aussi à $w(L')$. On a montré que $L_i \subset L'_i$ pour tout $1 \leq i \leq n$. Les inclusions inverses sont aussi vérifiées par symétrie. Ceci montre que $L \stackrel{l+1}{\sim} L'$ et termine la preuve que $L = L'$. $\square$

2.2.5 Théorème de Parikh

On montre ici que tout langage algébrique est commutativement équivalent à un langage rationnel.

Définition 2.24. Deux mots w et w' sont dits *anagrammes* s'il existe n lettres $a_1, \dots, a_n$ et une permutation σ de $\{1, \dots, n\}$ telle que $w = a_1 \dots a_n$ et $w' = a_{\sigma(1)} \dots a_{\sigma(n)}$. L'ensemble des anagrammes d'un mot w est noté $\bar{w}$.

Pour un langage L , on appelle *image commutative* et on note $\bar{L}$ l'ensemble $\{\bar{w} \mid w \in L\}$ des classes des mots de L . Deux langages L et M sont dits *commutativement équivalents* si $\bar{L} = \bar{M}$. Pour tous langages L et M , les deux égalités suivantes sont bien sûr vérifiées.

$$\overline{L+M} = \bar{L} + \bar{M} \quad \text{et} \quad \overline{LM} = \bar{M}\bar{L}.$$

Nous sommes maintenant en mesure d'énoncer le théorème de Parikh.

Théorème 2.25 (Parikh). *Tout langage algébrique L est commutativement équivalent à un langage rationnel R ($\bar{L} = \bar{R}$).*

Les grandes étapes de la preuve sont les suivantes. On associe à une grammaire un système d'équations en commutatif et on montre que les langages engendrés par une grammaire propre sont l'unique solution du système. On montre de manière indépendante que tout système en commutatif admet aussi une solution rationnelle. Une autre preuve plus élémentaire se trouve en [Koz97, p. 201].

L'exemple suivante illustre l'énoncé du théorème.

Exemple 2.26. Au langage algébrique $L = \{a^n b^n \mid n \geq 0\}$, on peut par exemple associer le langage rationnel $R = (ab)^*$ tel que $\overline{L} = \overline{R}$.

2.2.6 Systèmes d'équations en commutatifs

Nous montrons ici que des langages algébriques engendrés par une grammaire propre sont la solution unique du système commutatif associé à la grammaire.

Soit $G = (A, V, P)$ une grammaire dont l'ensemble des variables est $V = \{X_1, \dots, X_n\}$ et soit $L = (L_1, \dots, L_n)$ un n -uplet de langages sur A . Soient u et v deux mots sur l'alphabet $A + V$. Si $\overline{u} = \overline{v}$ alors $u(L) = v(L)$. Cette propriété s'étend aisément aux langages. Si les deux langages J et K sur $A + V$ vérifient $\overline{J} = \overline{K}$ alors l'égalité $\overline{J(L)} = \overline{K(L)}$ est vérifiée.

Soit $G = (A, V, P)$ une grammaire dont on suppose par commodité que l'ensemble des variables est $\{X_1, \dots, X_n\}$. On associe à cette grammaire un système d'équations $\overline{\mathcal{S}(G)}$ obtenu en considérant le système $\mathcal{S}(G)$ pour les langages de la forme $\overline{K}$. Une solution de $\overline{\mathcal{S}(G)}$ est donc un n -uplet $L = (L_1, \dots, L_n)$ de langages tel que

$$\overline{L_i} = \sum_{X_i \longrightarrow w} \overline{w(L)} \quad \text{pour } 1 \leq i \leq n.$$

Il s'agit d'établir la proposition suivante, ou plutôt de la rétablir dans le cas commutatif.

Proposition 2.27 (Unicité des solutions propres en commutatif). *Soit G une grammaire propre. Alors $\overline{L_G}$ est l'unique solution propre de $\overline{\mathcal{S}(G)}$.*

La proposition précédente contient un léger abus de langage. Conformément à la définition, c'est L_G et non pas $\overline{L_G}$ qui est la solution de $\overline{\mathcal{S}(G)}$. On a employé $\overline{L_G}$ pour insister sur le fait que l'unicité doit être entendue à image commutative près. Ceci signifie que si L est une solution de $\overline{\mathcal{S}(G)}$, alors $\overline{L_G} = \overline{L}$.

Preuve. La preuve de la proposition 2.23 peut être reprise *mutatis mutandis*. En effet l'argument essentiel de cette preuve est argument de longueur qui reste inchangé dans le cas commutatif. $\square$

2.2.7 Solutions rationnelles des systèmes commutatifs

Nous montrons ici que tout système d'équations en commutatif associé à une grammaire admet une solution rationnelle. Pour cela, on considère des grammaires généralisées où l'ensemble des productions de chaque variable n'est plus un ensemble fini mais un ensemble rationnel de mots.

Définition 2.28. Une *grammaire étendue* G est un triplet (A, V, P) où A et V sont des alphabets finis et disjoints et où P est une partie de $V \times (A \cup V)^*$ telle que pour tout $X \in V$, l'ensemble $\{w \mid (X, w) \in P\}$ est rationnel.

Les notions de dérivation et de mot engendré se généralisent aux grammaires étendues. Il est facile de voir que l'ensemble des mots engendrés par une grammaire étendue est algébrique. Il suffit pour cela de rajouter des variables (autant que d'états) qui simulent chacun des automates acceptant les langages rationnels $\{w \mid (X, w) \in P\}$.

Théorème 2.29 (Solutions rationnelles). *Soit $G = (A, V, P)$ une grammaire étendue où $V = \{X_1, \dots, X_n\}$. Il existe des langages rationnels $R_1, \dots, R_n$ sur A tels que $(\overline{R_1}, \dots, \overline{R_n})$ soit solution de $\overline{\mathcal{S}(G)}$.*

On commence par un lemme.

Lemme 2.30. *Soit K un langage sur $A + \{X\}$ et soit L et M deux langages sur A . Alors l'égalité $\overline{K(L^*M)L^*} = \overline{K(M)L^*}$ est vérifiée.*

Preuve. Soit u un mot de K . En regroupant les occurrences de la variable X à la fin, on obtient $\overline{u} = \overline{wX^n}$ où w est un mot sur A et où n est le nombre d'occurrences de X dans u . On vérifie alors facilement que l'égalité $\overline{u(L^*M)L^*} = \overline{u(M)L^*}$. On obtient le résultat du lemme en sommant les égalités précédentes sur tous les éléments de K . $\square$

On procède maintenant à la preuve du théorème.

Preuve. La preuve du théorème se fait par récurrence sur le nombre n de variables de la grammaire étendue G .

Commençons par une grammaire $G = (A, \{X\}, P)$ ayant seule variable X . Les règles de G sont notées $S \rightarrow S(X)$ où $S(X)$ est une expression rationnelle sur l'alphabet $(A + \{X\})^*$. Une solution du système $\overline{\mathcal{S}(G)}$ est un langage L tel que $\overline{L} = \overline{\mathcal{S}(L)}$.

On isole dans $S(X)$ les mots ayant au moins une occurrence de X et les mots ayant aucune occurrence de X . On pose donc $P(X) = S(X) \cap (A + \{X\})^*X(A + \{X\})^*$ et $T = S(X) \cap A^*$. Les deux langages $P(X)$ (on note $P(X)$ plutôt que P pour insister sur le fait que les mots de P contiennent des occurrences de la variable X) et T sont deux langages rationnels puisque la classe des langages rationnels est close par intersection.

Comme tous les mots de $P(X)$ contiennent au moins une occurrence de X , on peut écrire modulo la commutativité

$$\overline{P(X)} = \overline{Q(X)X}$$

où $Q(X)$ peut être choisi rationnel. Il suffit de prendre pour $Q(X)$ l'ensemble des mots de $P(X)$ où la première occurrence de X a été supprimée.

Le système $\overline{\mathcal{S}(G)}$ s'écrit alors

$$\overline{L} = \overline{Q(L)L + T}.$$

On va vérifier que le langage rationnel $R = Q(T)^*T$ est effectivement une solution du système $\overline{\mathcal{S}(G)}$.

$$\begin{aligned} \overline{Q(R)R + T} &= \overline{Q(Q(T)^*T)Q(T)^*T + T} \\ &= \overline{Q(T)Q(T)^*T + T} && \text{par le lemme précédent} \\ &= \overline{Q(T)^*T} && Q(T)^* = Q(T)Q(T)^* + \varepsilon \\ &= \overline{R}. \end{aligned}$$

Ceci termine le cas où la grammaire a une seule variable.

Soit n un entier positif. On considère maintenant une grammaire étendue G ayant $n + 1$ variables $X_0, X_1, \dots, X_n$. Les règles de G s'écrivent

$$X_i \longrightarrow S_i(X_0, \dots, X_n) \quad \text{pour } 0 \leq i \leq n$$

où chaque S_i est un langage rationnel sur $A + X$ avec $X = \{X_0, \dots, X_n\}$. Par commodité, on note X' l'ensemble X privé de la variable X_0 , c'est-à-dire l'ensemble $\{X_1, \dots, X_n\}$.

On introduit la grammaire G_0 obtenue en considérant les variables $X_1, \dots, X_n$ comme des lettres terminales et en ne prenant que les règles associées à X_0 dans G . Plus formellement, la grammaire G_0 est donnée par

$$G_0 = (A + X', \{X_0\}, \{X_0 \rightarrow S_0(X_0, X')\}).$$

Comme la grammaire G_0 a une seule variable, on utilise le cas précédent pour obtenir une solution $R_0(X')$ du système $\overline{\mathcal{S}(G_0)}$ qui vérifie donc l'égalité suivante.

$$\overline{R_0(X')} = \overline{S_0(R_0(X'), X')} \quad (2.1)$$

On introduit la grammaire étendue G' en remplaçant, dans les autres règles, X_0 par la solution $R_0(X')$ de $\overline{\mathcal{S}(G_0)}$. Plus formellement, la grammaire G' est donnée par

$$G' = (A, X', \{X_i \rightarrow S_i(R_0(X'), X') \mid 1 \leq i \leq n\}).$$

Il faut remarquer que la grammaire G' est étendue même si la grammaire G ne l'est pas. En effet, la variable X_0 est substituée par un langage rationnel. Ceci explique la nécessité de considérer des grammaires étendues pour la preuve du théorème.

Puisque G' a n variables, l'hypothèse de récurrence peut être appliquée. Il existe n langages rationnels $R_1, \dots, R_n$ sur A qui forment une solution $R' = (R_1, \dots, R_n)$ du système $\overline{\mathcal{S}(G')}$. Ils vérifient donc les égalités

$$\overline{R_i} = \overline{S_i(R_0(R'), R')} \quad \text{pour } 1 \leq i \leq n.$$

Il reste à vérifier que le $n + 1$ -uplet $R = (R_0(R'), R_1, \dots, R_n)$ constitue effectivement une solution du système $\overline{\mathcal{S}(G)}$ de la grammaire initiale G . En substituant X' par R' dans l'équation (2.1), on obtient une équation qui complète les n équations ci-dessus pour montrer que le $n + 1$ -uplet R est bien une solution rationnelle de $\overline{\mathcal{S}(G)}$. Ceci termine la preuve du théorème. $\square$

Preuve du théorème de Parikh

Soit L un langage algébrique. Il existe une grammaire $G = (A, V, P)$ où $X = \{X_1, \dots, X_n\}$ telle que $L = L_G(X_1)$. Si L est propre, on peut supposer G propre. D'après la proposition 2.27, $\overline{L}_G$ est l'unique solution propre du système $\overline{\mathcal{S}(G)}$. D'après le théorème 2.29, ce même système admet une solution rationnelle $R = (R_1, \dots, R_n)$. En combinant les deux résultats, on conclut que $\overline{L} = \overline{R_1}$.

Si L n'est pas propre, on applique le résultat précédent au langage $L' = L \setminus \{\varepsilon\}$ qui est propre et aussi algébrique. On obtient un langage rationnel R' tel que $\overline{L'} = \overline{R'}$. Le langage $R = R' + \varepsilon$ vérifie alors l'égalité $\overline{L} = \overline{R}$ souhaitée.

2.3 Arbres de dérivation

Dans une dérivation, les règles qui s'appliquent à des variables différentes peuvent être appliquées de manière indépendante et donc dans un ordre quelconque. Ceci conduit à avoir plusieurs dérivation différentes même lorsqu'il y a essentiellement qu'une seule façon d'obtenir un mot. Les arbres de dérivation permettent de capturer cette notion d'ambiguïté des grammaires. Ils permettent aussi de formaliser un lemme d'itération.

Définition 2.31 (Arbre de dérivation). Soit $G = (A, V, P)$ une grammaire. Un *arbre de dérivation* est un arbre fini étiqueté par $A \cup V \cup \{\varepsilon\}$ vérifiant la propriété suivante. Si S est l'étiquette d'un nœud interne et si $a_1, \dots, a_n$ sont les étiquettes de ses fils alors $S \rightarrow a_1 \cdots a_n$ est une règle de G . La *frontière* d'un arbre de dérivation est le mot obtenu par concaténation des étiquettes des feuilles de gauche à droite. Si la frontière contient au moins une variable, l'arbre est un *arbre de dérivation partielle*.

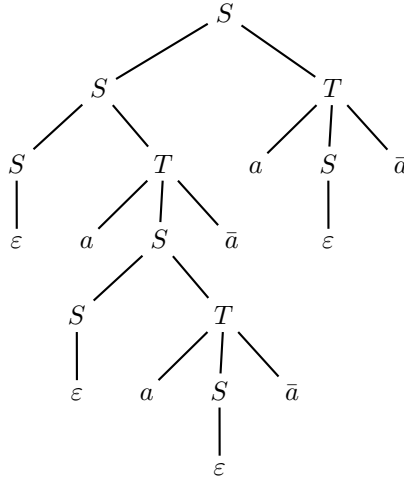


FIG. 2.1 – Arbre de dérivation

Exemple 2.32. Soit la grammaire définie par les règles $\{S \rightarrow ST + \varepsilon, T \rightarrow aSa\}$. La figure 2.1 exhibe un arbre de dérivation pour cette grammaire. La frontière de cet arbre est le mot $aa\bar{a}a\bar{a}a$.

Proposition 2.33. *Le langage $L_G(S)$ (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 2.34 (Dérivation gauche). On appelle *dérivation gauche* une dérivation où la variable remplacée est systématiquement la plus à gauche de l'image de la règle.

2.3.1 Ambiguïté

La notion d'ambiguïté est l'équivalent pour les grammaires du non déterminisme des automates. De manière intuitive, une grammaire est non-ambiguë,

si elle engendre chaque mot d'au plus une façon. Contrairement au cas des automates, tout langage algébrique n'est pas engendré par une grammaire non-ambiguë.

Définition 2.35 (Grammaire ambiguë). Une grammaire G est dite *ambiguë* s'il existe un mot ayant au moins deux arbres de dérivation distincts étiquetés à la racine par le même S .

Définition 2.36 (Langage ambigu). Un langage est dit *non ambigu* s'il existe au moins une grammaire non ambiguë qui l'engendre. Sinon, il est dit *inhéremment ambigu*.

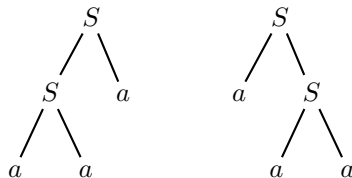


FIG. 2.2 – Deux arbres de dérivation pour aaa

Exemple 2.37. La grammaire $S \rightarrow SS + a$ est ambiguë car le mot aaa a deux arbres de dérivations (cf. figure 2.2). Par contre, le langage engendré $L_G(S) = a^+$ est non ambigu car il est également engendré par la grammaire $S \rightarrow aS + a$ qui est non ambiguë.

Les propositions 2.44 et 2.45 établissent qu'il existe des langages algébriques inhéremment ambigus.

2.3.2 Lemme d'itération

Le but de cette partie est d'établir un lemme d'itération pour les langages algébriques. On commence par un lemme purement combinatoire sur les arbres.

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 2.38. 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.

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 $r - 1$ nœuds particuliers. $\square$

Le lemme suivant est l'analogue du lemme de l'étoile pour les langages algébriques. Pour cette raison, il est aussi appelé *lemme d'itération*. Il permet de montrer que certains langages ne sont pas algébriques ou que certains langages algébriques sont inhéremment ambigus.

Lemme 2.39 (d'Ogden). *Pour toute grammaire $G = (A, V, P)$ et toute variable $S \in V$, 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 $\alpha, u, \beta, v, \gamma \in (A + V)^*$, avec :*

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

Les conditions du lemme montrent que les tous les mots de la forme $\alpha u^n \beta v^n \gamma$ pour $n \geq 0$ appartiennent au langage $\widehat{L}_G(S)$. Une telle paire (u, v) est appelée une *paire itérante*.

Preuve. Soit m la longueur maximale des membres droits des règles. Il 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 la figure 2.3. $\square$

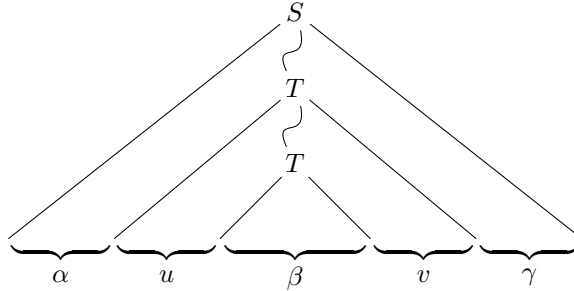


FIG. 2.3 – Découpage d'un mot par le lemme d'Ogden

Dans le cas où toutes les lettres de f sont marquées, on obtient le corollaire suivant.

Corollaire 2.40 (Théorème de Bar-Hillel, Perles, Shamir). *Pour tout langage algébrique L , il existe $N \geq 0$ tel que pour tout mot $f \in L$, si $|f| \geq N$ alors on peut trouver une factorisation $f = \alpha u \beta v \gamma$ tel que $|uv| > 0$, $|u \beta v| < N$ et $\alpha u^n \beta v^n \gamma \in L$ pour tout $n \geq 0$.*

2.3.3 Applications du lemme d'itération

Dans cette partie, on applique le lemme d'itération pour montrer que certains langages classiques ne sont pas algébriques et que certains langages algébriques sont inhéremment ambigus.

Proposition 2.41. *Le langage $L = \{a^n b^n c^n \mid n \geq 0\}$ n'est pas algébrique.*

Preuve. Le corollaire précédent suffit pour prouver que le langage L n'est pas algébrique. Soit N l'entier fourni par le corollaire. On considère le mot $f_1 = a^N b^N c^N \in L$ qui se factorise $f_1 = \alpha u \beta v \gamma$ tel que $f_n = \alpha u^n \beta v^n \gamma \in L$ pour tout $n \geq 0$. Chacun des mots u et v ne contient qu'une seule des lettres a , b ou c . Sinon le mot f_2 n'appartient pas à $a^* b^* c^* \supset L$. Il en découle que f_2 ne peut pas appartenir à L car les nombres d'occurrences de a , b et c ne sont plus égaux. $\square$

La proposition précédente a pour corollaire immédiat.

Corollaire 2.42. *La classe des langages algébriques n'est close ni par intersection ni par complémentation.*

Preuve. Le langage $\{a^n b^n c^n \mid n \geq 0\}$ est l'intersection des deux langages algébriques $\{a^m b^m c^n \mid m, n \geq 0\}$ et $\{a^m b^n c^n \mid m, n \geq 0\}$. $\square$

Proposition 2.43. *Le langage $L = \{a^m b^n c^m d^n \mid n, m \geq 0\}$ n'est pas algébrique.*

Preuve. Supposons qu'il existe une grammaire G telle que $L = L_G(S)$. Soit k l'entier fourni par le lemme d'itération. On applique le résultat au mot $f = a^k b^k c^k d^k$ où les k lettres distinguées sont les lettres b . D'après le lemme d'itération, il existe des dérivations

$$S \xrightarrow{*} a^k b^{k_1} T d^{k_2} \quad T \xrightarrow{*} b^i T d^i \quad T \xrightarrow{*} b^{k'_1} c^k d^{k'_2}$$

avec les relations $k = k_1 + i + k'_1 = k_2 + i + k'_2$. On applique à nouveau le lemme d'itération au mot $a^k b^{k_1} T d^{k_2}$ où les k lettres distinguées sont les lettres a . On obtient immédiatement une contradiction car la paire itérante obtenue contient des lettres a mais aucune lettre d . $\square$

Proposition 2.44. *Le langage $L = \{a^m b^m c^n \mid m, n \geq 0\} \cup \{a^m b^n c^n \mid m, n \geq 0\}$ est inhéremment ambigu.*

Preuve. Chacun des langages $\{a^m b^m c^n \mid m, n \geq 0\}$ et $\{a^m b^n c^n \mid m, n \geq 0\}$ est bien sûr algébrique et le langage L est donc aussi algébrique. Soit $G = (A, V, P)$ une grammaire telle que $L = L_G(S_0)$. On montre que la grammaire G est ambiguë en montrant qu'un mot de la forme $a^n b^n c^n$ a au moins deux arbres de dérivation.

Soit k l'entier fourni par le lemme d'itération. On applique le résultat au mot $f_1 = a^k b^k c^{k+k!}$ où les k lettres distinguées sont les b . D'après le lemme d'itération, il existe des dérivations

$$S_0 \xrightarrow{*} \alpha S \gamma \xrightarrow{*} \alpha u S v \gamma \xrightarrow{*} \alpha u \beta v \gamma = f_1$$

Les conditions impliquent que $u = a^i$ et $v = b^i$ pour un entier $0 \leq i \leq k$. En itérant $k!/i$ fois la dérivation $S \xrightarrow{*} u S v$ on obtient un arbre de dérivation pour le mot $f = a^{k+k!} b^{k+k!} c^{k+k!}$. Cet arbre contient un sous-arbre dont la frontière ne contient que des lettres a et b dont au moins $k! - k$ lettres b .

En appliquant le même procédé au mot $f_2 = a^{k+k!} b^k c^k$, on obtient un autre arbre de dérivation pour le même mot f . Cet arbre contient un sous-arbre dont la frontière ne contient que des lettres b et c dont au moins $k! - k$ lettres b . Cet arbre est donc différent du premier arbre trouvé. $\square$

2.3.4 Ambiguïté inhérente

Il est souvent difficile de montrer qu'un langage algébrique donné est inhéremment ambigu. Lorsque le lemme d'itération n'est pas adapté, une autre méthode consiste à utiliser la fonction génératrice du langage L définie par

$$f_L(z) = \sum_{n \geq 0} a_n z^n \quad \text{où} \quad a_n = |L \cap A^n|.$$

Si le langage est non ambigu, la fonction $f_L(z)$ est algébrique car elle est solution du système commutatif associé à une grammaire non ambiguë qui engendre L . En utilisant cette méthode, on peut par exemple montrer le théorème suivant (cf. exemple 2.6 p. 62 pour la définition du langage de Goldstine).

Proposition 2.45 (Flageolet). *Le langage de Goldstine est inhéremment ambigu.*

Un mot n'est pas dans le langage de Goldstine soit parce qu'il se termine par la lettre a soit parce qu'il est de la forme $baba^2ba^3b \cdots a^n b$ pour $n \geq 0$. Ceci montre que la fonction génératrice du langage de Goldstine est égale à $(1-z)/(1-2z) - g(z)$ où la fonction $g(z)$ est égale à

$$g(z) = z + z^3 + z^6 + \cdots = \sum_{n \geq 1} z^{n(n+1)/2}.$$

Une fonction de la forme $\sum_{n \geq 0} z^{c_n}$ où la suite $(c_n)_{n \geq 0}$ vérifie $\sup(c_{n+1} - c_n) = \infty$ converge sur le bord de son disque de convergence. La fonction g admet donc une infinité de singularités et ne peut être algébrique comme f_L .

2.4 Propriétés de clôture

On montre dans cette partie quelques opérations de clôture classiques pour les langages algébriques. Le théorème de Chomsky-Schützenberger qui établit que les langages de Dyck sont les langages algébriques génériques est également prouvé.

2.4.1 Opérations rationnelles

On commence par les opérations rationnelles qui montrent que les langages algébriques généralisent les langages rationnels.

Proposition 2.46 (Opérations rationnelles). *Les langages algébriques sont clos par union, concaténation et étoile.*

Preuve. Soient $G = (A, V, P)$ et $G' = (A, V', P')$ deux grammaires telles que $L = L_G(S)$ et $L' = L_{G'}(S')$. On suppose sans perte de généralité que $V \cap V' = \emptyset$.

- Le langage $L + L'$ est égal à $L_{G_1}(S_0)$ où la grammaire G_1 est égale à $(A, \{S_0\} \cup V \cup V', \{S_0 \rightarrow S + S'\} \cup P \cup P')$
- Le langage LL' est égal à $L_{G_2}(S_0)$ où la grammaire G_2 est égale à $(A, \{S_0\} \cup V \cup V', \{S_0 \rightarrow SS'\} \cup P \cup P')$
- Le langage L^* est égal à $L_{G_3}(S_0)$ où la grammaire G_3 est égale à $(A, \{S_0\} \cup V, \{S_0 \rightarrow SS_0 + \varepsilon\} \cup P)$

□

Le corollaire suivant découle directement de la proposition précédente.

Corollaire 2.47. *Les langages rationnels sont aussi algébriques.*

Pour obtenir directement le corollaire, on peut également construire une grammaire qui engendre le langage $L(\mathcal{A})$ des mots acceptés par un automate normalisé $\mathcal{A} = (Q, A, E, \{i\}, \{f\})$. La grammaire $G = (A, Q, P)$ où l'ensemble des règles est $P = \{p \rightarrow aq \mid p \xrightarrow{a} q \in E\} \cup \{f \rightarrow \varepsilon\}$ vérifie que $L_G(i) = L(\mathcal{A})$. Réciproquement, une grammaire dont toutes les règles sont de la forme $S \rightarrow aT$ pour $a \in A$ et $S, T \in V$ engendre un langage rationnel.

2.4.2 Substitution algébrique

Proposition 2.48 (Substitution algébrique). *Si $L \subset A^*$ est un langage algébrique et σ une substitution algébrique de A^* dans B^* , alors le langage $\sigma(L) = \{\sigma(w) \mid w \in L\}$ est aussi algébrique.*

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

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

2.4.3 Morphisme alphabétique inverse

Définition 2.49 (Morphisme alphabétique). Un morphisme $\sigma : A^* \rightarrow B^*$ est dit *alphabétique* (resp. *strictement alphabétique*) si pour tout $a \in A$, $|\sigma(a)| \leq 1$ (resp. $|\sigma(a)| = 1$).

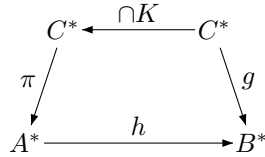


FIG. 2.4 – Factorisation d'un morphisme

Lemme 2.50 (Factorisation d'un morphisme). *Pour tout morphisme $h : A^* \rightarrow B^*$, il existe deux morphismes alphabétiques $g : C^* \rightarrow B^*$ et $\pi : C^* \rightarrow A^*$ et un langage rationnel $K \subseteq C^*$ tel que $h^{-1}(w) = \pi(g^{-1}(w) \cap K)$ pour tout mot $w \in B^*$.*

Preuve. On définit l'alphabet C et les deux parties C_0 et C_1 de C par

$$C = \{(a, i) \mid a \in A \text{ et } 0 \leq i \leq |h(a)|\}$$

$$C_0 = \{(a, 0) \mid a \in A\} \quad C_1 = \{(a, |h(a)|) \mid a \in A\}$$

puis les langages rationnels locaux W et K de C^* par

$$\begin{aligned} W &= C^2 \setminus (\{(a, i)(a, i+1) \mid a \in A \text{ et } 0 \leq i < |h(a)|\} \cup C_1 C_0) \\ K &= (C_0 C^* \cap C^* C_1) \setminus C^* W C^* \end{aligned}$$

Les mots de K sont les mots de la forme $u_1 \cdots u_n$ où chaque mot u_i pour $1 \leq i \leq n$ est un mot de la forme $(a_i, 0)(a_i, 1) \cdots (a_i, |h(a_i)|)$ pour une lettre a_i de A .

On définit finalement les deux morphismes $g : C^* \rightarrow B^*$ et $\pi : C^* \rightarrow A^*$ par

$$g(a, i) = \begin{cases} \varepsilon & \text{si } i = 0 \\ h(a)[i] & \text{si } i \geq 1 \end{cases} \quad \text{et} \quad \pi(a, i) = \begin{cases} a & \text{si } i = 0 \\ \varepsilon & \text{si } i \geq 1 \end{cases}$$

où $h(a)[i]$ est une notation pour la i -ième lettre du mot $h(a)$. $\square$

Proposition 2.51 (Morphisme inverse). *Si $h : A^* \rightarrow B^*$ est un morphisme et $L \subseteq B^*$ est un langage algébrique, alors $h^{-1}(L)$ est aussi algébrique.*

Preuve. Montrons d'abord la propriété lorsque h est alphabétique. Soit $G = (B, V, P)$ une grammaire et S_0 une variable telle que $L = L_G(S_0)$.

Puisque le morphisme h est alphabétique, l'alphabet A se partitionne en les deux parties A_0 et A_1 définies par $A_0 = \{a \in A \mid h(a) = \varepsilon\}$ et $A_1 = \{a \in A \mid h(a) \in B\}$. Le morphisme h est étendu à $(A + V)^*$ en posant $h(S) = S$ pour toute variable $S \in V$.

On définit la grammaire $G' = (A, V, P_0 \cup P_1)$ où les ensembles de règles P_0 et P_1 sont respectivement donnés par

$$\begin{aligned} P_0 &= \{T \rightarrow \sum_{a \in A_0} aT + \varepsilon\}, \\ P_1 &= \{S \rightarrow Tu_1T \cdots Tu_nT \mid u_i \in (V \cup A_1)^*, S \rightarrow h(u_1 \cdots u_n) \in P\}. \end{aligned}$$

On vérifie que la grammaire G' engendre le langage $h^{-1}(L)$. Ensuite, pour étendre ce résultat à un morphisme h quelconque, il suffit d'appliquer le lemme précédent. L'intersection avec le rationnel K découle de la propriété suivante. $\square$

2.4.4 Intersection avec un rationnel

Proposition 2.52 (Intersection avec un rationnel). *Si L est un langage algébrique et K un langage rationnel, alors $K \cap L$ est algébrique.*

L'intersection de deux langages algébriques n'est pas algébrique en général comme cela a été vu au corollaire 2.42 (p. 77).

Preuve. Soit $G = (A, V, P)$ une grammaire telle que $L = L_G(T)$. Sans perte de généralité, on peut supposer que G est en forme normale quadratique. Cette condition n'est pas indispensable à la preuve mais simplifie les notations.

Méthode des automates Soit $\mathcal{A} = (Q, A, E, \{i\}, \{f\})$ un automate normalisé acceptant la langage K . Le langage $K \cap L$ est engendré par la grammaire

$G' = (A, V', P')$ définie de la façon suivante.

$$\begin{aligned} V' &= \{S_{pq} \mid S \in V \text{ et } p, q \in Q\} \\ P' &= \{S_{pq} \rightarrow a \mid S \rightarrow a \in P \text{ et } p \xrightarrow{a} q \in E\} \\ &\cup \{S_{pq} \rightarrow R_{pr}T_{rq} \mid r \in Q \text{ et } S \rightarrow RT \in P\}. \end{aligned}$$

Le langage $K \cap L$ algébrique car il est égal à $L_{G'}(T_{if})$.

Méthode des monoïdes Soit $\mu : A^* \rightarrow M$ un morphisme de A^* dans un monoïde M fini tel que $K = \mu^{-1}(H)$ où H est une partie de M . Le langage $K \cap L$ est engendré par la grammaire $G' = (A, V', P')$ définie de la façon suivante.

$$\begin{aligned} V' &= \{S_m \mid S \in V \text{ et } m \in M\} \\ P' &= \{S_m \rightarrow a \mid S \rightarrow a \in P \text{ et } m = \mu(a)\} \\ &\cup \{S_m \rightarrow R_{m_1}T_{m_2} \mid S \rightarrow RT \in P \text{ et } m = m_1m_2\}. \end{aligned}$$

Le langage $K \cap L$ est algébrique car il est égal à l'union $\bigcup_{m \in H} L_{G'}(T_m)$.

□

2.4.5 Théorème de Chomsky-Schützenberger

Le théorème suivant établit que les langages de Dyck (cf. exemple 2.6 p. 62 pour la définition) sont des langages algébriques *génériques*. Ils sont générateurs du cône des langages algébriques. On peut consulter [Aut87] pour une preuve plus détaillée de ce théorème.

Théorème 2.53 (Chomsky-Schützenberger). *Un langage L est algébrique si et seulement si $L = \varphi(D_n^* \cap K)$ pour un entier n , un langage rationnel K et un certain morphisme φ alphabétique.*

Preuve. La condition est suffisante grâce aux propriétés de clôture des langages algébriques.

Réciproquement, soit $G = (A, V, P)$ une grammaire telle que $L = L_G(S_0)$ pour $S_0 \in V$. D'après la proposition 2.16, on peut supposer que G en forme normale quadratique. Les règles de G sont donc de la forme $S \rightarrow S_1S_2$ pour $S_1, S_2 \in V$ ou de la forme $S \rightarrow a$ pour $a \in A$. Pour chaque règle r du premier type, on introduit six nouveaux symboles $a_r, b_r, c_r, \bar{a}_r, \bar{b}_r$ et $\bar{c}_r$. Pour chaque règle r du second type, on introduit deux nouveaux symboles a_r et $\bar{a}_r$. Soit A' l'ensemble de toutes les nouvelles lettres introduites :

$$A' = \{a_r, b_r, c_r, \bar{a}_r, \bar{b}_r, \bar{c}_r \mid r = S \rightarrow S_1S_2\} \cup \{a_r, \bar{a}_r \mid r = S \rightarrow a\}.$$

Soit D_n le langage de Dyck sur l'alphabet de parenthèses A' . L'entier n est donc égal à trois fois le nombre de règles de la forme $S \rightarrow S_1S_2$ plus le nombre de règles de la forme $S \rightarrow a$. On définit alors la grammaire $G' = (A', V, P')$ où les règles de G' sont en correspondance avec les règles de G . Pour chaque règle r de G , il existe une règle r' de G' définie de la manière suivante.

- Si r est la règle $S \rightarrow S_1S_2$, alors r' est la règle $S \rightarrow a_rb_rS_1\bar{b}_rc_rS_2\bar{c}_r\bar{a}_r$.
- Si r est la règle $S \rightarrow a$, alors r' est la règle $S \rightarrow a_r\bar{a}_r$.

On définit finalement le morphisme $\varphi : A'^* \rightarrow A^*$ de la façon suivante. Si r est une règle de la forme $S \rightarrow S_1 S_2$, on pose φ par $\varphi(a_r) = \varphi(b_r) = \varphi(c_r) = \varphi(\bar{a}_r) = \varphi(\bar{b}_r) = \varphi(\bar{c}_r) = \varepsilon$. Si r est une règle de la forme $S \rightarrow a$, on pose $\varphi(a_r) = a$ et $\varphi(\bar{a}_r) = \varepsilon$.

Les définitions des règles de G' et du morphisme φ impliquent immédiatement que $L = L_G(S_0) = \varphi(L_{G'}(S_0))$. L'inclusion $L_{G'}(S_0) \subset D_n^*$ découle directement de la forme particulière des règles de G' . Il reste maintenant à définir un langage rationnel K tel que $L_{G'}(S_0) = D_n^* \cap K$.

On va définir un langage local K qui exprime les contraintes entre les paires de lettres consécutives de A' dans $L_{G'}(S_0)$. Soit A'_0 l'ensemble des lettres a_r où r est une règle ayant S_0 pour membre gauche. Soit K le langage local $K = A_0 A'^* \setminus A'^* W A'^*$ où l'ensemble W est défini par

$$\begin{aligned} A'^2 \setminus W = & \{a_r b_r, \bar{b}_r c_r, \bar{c}_r \bar{a}_r \mid r = S \rightarrow S_1 S_2\} \\ & \cup \{a_r \bar{a}_r \mid r = S \rightarrow a\} \\ & \cup \{b_r a_t, \bar{a}_t \bar{b}_r \mid r = S \rightarrow S_1 S_2 \text{ et } t = S_1 \rightarrow \dots\} \\ & \cup \{c_r a_t, \bar{a}_t \bar{c}_r \mid r = S \rightarrow S_1 S_2 \text{ et } t = S_2 \rightarrow \dots\}. \end{aligned}$$

L'inclusion $L_{G'}(S_0) \subset D_n^* \cap K$ découle directement du choix du langage K . L'inclusion inverse se montre par récurrence sur la longueur du mot. $\square$

Pour un entier n , on note A_n l'alphabet $\{a_1, \dots, a_n, \bar{a}_1, \dots, \bar{a}_n\}$ ayant n paires de parenthèses.

Lemme 2.54. *Pour tout entier n , il existe un morphisme $\psi : A_n^* \rightarrow A_2^*$ tel que $D_n^* = \psi^{-1}(D_2^*)$.*

Preuve. Il est facile de vérifier que le morphisme ψ défini par $\psi(a_k) = a_1 a_2^k a_1$ et $\psi(\bar{a}_k) = \bar{a}_1 \bar{a}_2^k \bar{a}_1$ pour $1 \leq k \leq n$ convient parfaitement. $\square$

$$\begin{array}{ccc} & \xleftarrow{\cap K} & D_n^* \\ \varphi \swarrow & & \searrow \psi \\ L & & D_2^* \end{array}$$

FIG. 2.5 – Théorème de Chomsky-Schützenberger

D'après le lemme précédent, le théorème de Chomsky-Schützenberger établit que tout langage algébrique L s'écrit $L = \varphi(\psi^{-1}(D_2^*) \cap K)$ pour des morphismes φ et ψ et pour un langage rationnel K (cf. figure 2.5). Une transformation de la forme $X \mapsto \varphi(\psi^{-1}(X) \cap K)$ s'appelle une *transduction rationnelle*. Elle peut être réalisée par un automate ayant deux bandes, une pour l'entrée et une pour la sortie.

2.5 Forme normale de Greibach

La forme normale de Greibach est une forme particulière de grammaire. Toute grammaire est équivalente à une grammaire en forme normale de Greibach mais la preuve n'est pas triviale, contrairement à la forme normale quadratique.

Cette forme normale permet en particulier de montrer que les ε -transitions ne sont pas nécessaires dans les automates à pile.

Définition 2.55 (Forme normale de Greibach). Une grammaire $G = (A, V, P)$ est en *forme normale de Greibach* si chacune de ses règles est de la forme $S \rightarrow w$ où w appartient à AV^* . Si de plus chaque w appartient à $A + AV + AV^2$, la grammaire est en *forme normale de Greibach quadratique*.

Proposition 2.56 (Forme normale de Greibach). *Toute grammaire propre est équivalente à une grammaire en forme normale de Greibach quadratique.*

La preuve initiale de Greibach donnait seulement une grammaire en forme normale non quadratique. Nous allons d'abord une première preuve plus élémentaire qui donne une grammaire en forme normale de Greibach. Nous donnons ensuite la preuve due à Rosenkrantz qui est plus concise et donne en plus une grammaire en forme normale quadratique. Le théorème peut encore est raffiné en introduisant des formes normales bilatères où les membres droits des règles appartiennent à $A + AA + AVA + AVVA$. La preuve de ce résultat dû à Hotz est cependant plus difficile.

Preuve élémentaire. Il suffit de trouver une grammaire avec des règles de la forme $S \rightarrow w$ où $w \in A(A + V)^*$. On se ramène alors à une grammaire en forme normale de Greibach en introduisant une nouvelle variable T_a avec la règle $T \rightarrow a$ pour chaque lettre terminale a et remplaçant chaque occurrence de a par T_a .

Soit $G = (A, V, P)$ une grammaire où on suppose que $V = \{X_1, \dots, X_n\}$. On pose $G_0 = G$ et on définit par récurrence une suite $G_0, \dots, G_n$ de grammaires telles que dans chaque grammaire G_i les variables $X_1, \dots, X_i$ n'apparaissent pas en tête des membres droits de règles. On suppose avoir construit la grammaire G_{i-1} et on construit la grammaire G_i par les deux étapes suivantes.

1. On commence par supprimer X_i en tête des productions de X_i . Soient

$$X_i \rightarrow X_i u_1 + \dots + X_i u_k + w_1 + \dots + w_p$$

les règles de G_{i-1} ayant X_i comme membre gauche où les mots $w_1, \dots, w_p$ ne commencent pas par la variable X_i . On introduit une nouvelle variable X'_i et on remplace les règles précédentes par les nouvelles règles

$$\begin{aligned} X_i &\rightarrow w_1 X'_i + \dots + w_p X'_i + w_1 + \dots + w_p \\ X'_i &\rightarrow u_1 X'_i + \dots + u_k X'_i + u_1 + \dots + u_k. \end{aligned}$$

2. La transformation précédente a pu faire apparaître des variables X_j (pour $1 \leq j \leq i$) en tête des mots u_l . En outre, la variable X_i peut être la première lettre de membres droits d'autres règles. Toutes ces occurrences des variables X_j (pour $1 \leq j \leq i$) en tête de membres droits sont supprimées en remplaçant chaque règle $S \rightarrow X_j v$ par toutes les règles $S \rightarrow uv$ où u parcourt les productions de la variable X_j .

□

Exemple 2.57. En appliquant l'algorithme de la preuve précédente à la grammaire $G = G_0$, on obtient successivement les grammaires G_1 , G_2 et G_3 .

$$\begin{array}{ll}
 G_0 \left\{ \begin{array}{l} A \rightarrow AB + a \\ B \rightarrow BC + b \\ C \rightarrow CA + c \end{array} \right. & G_1 \left\{ \begin{array}{l} A \rightarrow aA' + a \\ A' \rightarrow BA' + B \\ B \rightarrow BC + b \\ C \rightarrow CA + c \end{array} \right. \\
 G_2 \left\{ \begin{array}{l} A \rightarrow aA' + a \\ A' \rightarrow bB'A' + bA' + bB' + b \\ B \rightarrow bB' + b \\ B' \rightarrow CB' + bB' \\ C \rightarrow CA + c \end{array} \right. & G_3 \left\{ \begin{array}{l} A \rightarrow aA' + a \\ A' \rightarrow bB'A' + bA' + bB' + b \\ B \rightarrow bB' + b \\ B' \rightarrow cC'B' + cB' + cC' + c \\ C \rightarrow cC' + c \\ C' \rightarrow aA'C' + aC' + aA' + a \end{array} \right.
 \end{array}$$

Preuve de Rosenkrantz. Soit $G = (A, V, P)$ une grammaire où l'ensemble des variables est $V = \{X_1, \dots, X_n\}$. Sans perte de généralité, on peut supposer que G est forme normale quadratique. Cette hypothèse est uniquement nécessaire pour que la grammaire en forme normale de Greibach qui est obtenue soit aussi quadratique. Pour $1 \leq i \leq n$, on définit le langage P_i par $P_i = \{w \mid X_i \rightarrow w\}$. Les règles de G s'écrivent alors $X_i \rightarrow P_i$ pour $1 \leq i \leq n$. On introduit le vecteur $\vec{X} = (X_1, \dots, X_n)$ et le vecteur $\vec{P} = (P_1, \dots, P_n)$ de sorte que les règles s'écrivent

$$\vec{X} \rightarrow \vec{P}.$$

On introduit la matrice $R = (R_{ij})$ de dimensions $n \times n$ où chaque entrée R_{ij} est définie par $R_{ij} = X_i^{-1}P_j$. On définit aussi le vecteur $\vec{S} = (S_1, \dots, S_n)$ où $S_i = P_i \cap AV^*$. Les règles de la grammaire s'écrivent alors sous forme matricielle de la manière suivante

$$\vec{X} \rightarrow \vec{X}R + \vec{S}.$$

Par analogie avec le système $X = XR + S$ dont la solution est SR^* , on introduit la matrice $Y = (Y_{ij})$ où chaque entrée Y_{ij} est une nouvelle variable. On introduit aussi la grammaire G' dont les règles sont données par

$$\begin{array}{l}
 X \rightarrow SY \\
 Y \rightarrow RY + I
 \end{array}$$

où I est la matrice ayant ε sur la diagonale et $\emptyset$ partout ailleurs. La grammaire G' est équivalente à la grammaire G . Comme G est en forme normale quadratique, chaque entrée R_{ij} est une somme de variables de V . En utilisant $X \rightarrow SY$, on définit finalement la matrice $R' = (R'_{ij})$ où chaque entrée R'_{ij} est donnée par la formule

$$R'_{ij} = \sum_{X_k \in R_{ij}} (SY)_k.$$

La grammaire G'' dont les règles sont données par

$$\begin{array}{l}
 X \rightarrow SY \\
 Y \rightarrow R'Y + I
 \end{array}$$

est encore équivalente à la grammaire G . On supprime les ε -règles de G'' pour obtenir une grammaire en forme normale de Greibach quadratique équivalente à G . $\square$

2.6 Automates à pile

Les automates à pile sont une extension des automates finis. Outre un contrôle par un nombre fini d'états, ils possèdent une mémoire auxiliaire. Celle-ci est organisée sous la forme d'une pile contenant des symboles. Il est seulement possible d'empiler ou de dépiler des symboles. Seul le symbole en sommet de la pile est accessible. Celui-ci participe au choix de la transition effectuée. Ces automates acceptent les langages algébriques.

2.6.1 Définitions et exemple

Comme un automate fini, un automate à pile a un ensemble fini Q d'états. Il y a un alphabet A sur lequel sont écrits les mots d'entrée ainsi qu'un alphabet de pile qui contient tous les symboles qui peuvent être mis dans la pile. La transition effectuée par l'automate dépend de l'état de contrôle, de la lettre lue dans le mot d'entrée et du symbole au sommet de la pile. Chaque transition remplace le symbole au sommet de la pile par un mot éventuellement vide sur l'alphabet de pile. Ce dernier cas correspond à un dépilement. Effectuer une transition fait passer à la lettre suivante du mot d'entrée qui est donc lu séquentiellement de gauche à droite. On introduit aussi des ε -transitions qui ne lisent aucune lettre du mot d'entrée. Ces transitions ne sont pas absolument nécessaires dans le cas des automates non déterministes. Il est possible de les supprimer même si ce n'est pas immédiat. Par contre, ces ε -transitions sont indispensables pour les automates déterministes.

Définition 2.58 (Automate à pile). Un *automate à pile* est constitué d'un alphabet d'entrée A , d'un alphabet de pile Z dont un symbole initial $z_0 \in Z$, d'un ensemble fini d'états Q dont un état initial q_0 et de transitions de la forme $q, y, z \rightarrow q', h$ avec $q, q' \in Q$, $y \in A \cup \{\varepsilon\}$, $z \in Z$ et $h \in Z^*$.

L'état initial q_0 est l'état de contrôle dans lequel se trouve l'automate à pile au début d'un calcul. Tout au long d'un calcul, la pile de l'automate n'est jamais vide. Le calcul s'arrête dès que la pile devient vide. Le symbole de pile initial z_0 est le symbole qui est mis dans pile avant de commencer tout calcul.

Une configuration d'un automate est un état instantané de l'automate qui comprend l'état de contrôle, le contenu de la pile ainsi que le reste du mot à lire. Le contenu de la pile est vu comme un mot sur l'alphabet de pile. On appelle *configuration*, un triplet (q, f, h) de $Q \times A^* \times Z^*$. La notion de calcul pour un automate à pile est l'équivalent d'un chemin pour un automate fini. Il est constitué d'une suite d'étapes de calcul qui sont des paires de configurations consécutives.

Définition 2.59 (Calcul d'un automate à pile). Une *étape de calcul* est une paire (C, C') de configurations notée $C \rightsquigarrow C'$ telles que $C = (p, yf, zw)$ et $C' = (q, f, hw)$ pour une transition $p, y, z \rightarrow q, h$ de l'automate.

Un *calcul* de l'automate est une suite d'étapes de calcul. Pour un mot f , la configuration initiale est (q_0, f, z_0) .

Dans la définition précédente, le contenu de la pile est écrit de haut en bas. Le sommet de pile se situe donc à gauche du mot de pile. Certains ouvrages utilisent la convention inverse où le sommet de pile se situe à droite. Aucune des conventions n'est parfaite et n'évite de passer au mot miroir dans certaines

preuves. La convention adoptée dans ce cours simplifie la preuve d'équivalence entre grammaires et automates à pile (cf. théorème 2.63). Par contre, la preuve en complément que l'ensemble des contenus forme un langage rationnel (cf. théorème 2.75) nécessite un passage au mot miroir.

2.6.2 Différents modes d'acceptation

Il reste à définir les configurations finales qui déterminent les calculs acceptants et par conséquent les mots acceptés par l'automate. Il existe plusieurs modes d'acceptation pour définir ces configurations finales. Dans tous les cas, les configurations finales sont de la forme (q, ε, w) et imposent que l'automate a lu l'intégralité du mot d'entrée pour accepter. Les principaux modes utilisés sont les suivants. Ils sont tous équivalents pour les automates non déterministes. Pour les automates déterministes, le mode d'acceptation par pile vide est plus faible car le langage accepté est nécessairement un code préfixe.

pile vide : les configurations finales sont toutes les configurations de la forme $(q, \varepsilon, \varepsilon)$ où la pile est vide.

état final : les configurations finales sont toutes les configurations de la forme (q, ε, w) où l'état q appartient à un sous-ensemble F d'états distingués de Q et où le contenu de la pile est quelconque.

sommet de pile : les configurations finales sont les configurations (q, ε, zw) où le sommet de pile z appartient à un sous-ensemble Z_0 de symboles de pile distingués de Z .

combinaison : toute combinaison des trois premiers.

Un mot f est *accepté* par un automate à pile s'il existe un calcul de la configuration initiale (q_0, f, z_0) à une configuration finale où l'automate a entièrement lu le mot f .

Exemple 2.60. Soit l'automate à pile défini sur les alphabets d'entrée et de pile $A = \{a, b\}$ et $Z = \{z\}$ avec les états $Q = \{q_0, q_1\}$ et comportant les trois transitions suivantes :

$$q_0, a, z \rightarrow q_0, zzz \quad q_0, \varepsilon, z \rightarrow q_1, \varepsilon \quad q_1, b, z \rightarrow q_1, \varepsilon$$

Un calcul valide pour cet automate est par exemple

$$q_0, abb, z \rightsquigarrow q_0, bb, zzz \rightsquigarrow q_1, bb, zz \rightsquigarrow q_1, b, z \rightsquigarrow q_1, \varepsilon, \varepsilon.$$

Si on choisit un arrêt par pile vide, cet automate accepte le langage $L_1 = \{a^n b^{2n} \mid n \geq 0\}$. Si on choisit l'arrêt par état final avec l'ensemble $F = \{q_1\}$, il accepte le langage $L_2 = \{a^n b^p \mid 0 \leq p \leq 2n\}$.

Exemple 2.61. Soit l'automate à pile défini sur les alphabets d'entrée et de pile $A = \{a, b\}$ et $Z = \{A, B\}$ avec $z_0 = A$ comme symbole de pile initial. Il possède les états $Q = \{q_0, q_1, q_2\}$ et comporte les transitions suivantes :

$$\begin{array}{lll} q_0, a, A \rightarrow q_1, A & & q_2, a, A \rightarrow q_2, \varepsilon \\ q_0, b, A \rightarrow q_1, B & & q_2, b, B \rightarrow q_2, \varepsilon \\ q_1, a, A \rightarrow q_1, AA & q_1, b, A \rightarrow q_2, BA & q_1, a, A \rightarrow q_2, A \\ q_1, a, B \rightarrow q_1, AB & q_1, b, B \rightarrow q_2, BB & q_1, a, B \rightarrow q_2, B \\ q_1, b, A \rightarrow q_1, BA & q_1, a, A \rightarrow q_2, AA & q_1, b, A \rightarrow q_2, A \\ q_1, b, B \rightarrow q_1, BB & q_1, a, B \rightarrow q_2, AB & q_1, b, B \rightarrow q_2, B \end{array}$$

Cet automate à pile accepte l'ensemble des palindromes (cf. exemple 2.6 p. 62).

Proposition 2.62 (Équivalence des modes d'acceptation). *Les différents modes d'acceptation sont équivalents dans la mesure où ils permettent tous d'accepter exactement les mêmes langages (les langages algébriques).*

Un des problèmes des automates à pile est que le calcul se bloque dès que la pile devient vide. Cette difficulté peut être contournée en utilisant des automates à fond de pile testable qui peuvent tester si le sommet de pile est le dernier symbole dans la pile. Un automate est dit à *fond de pile testable* si son alphabet de pile peut être partitionné $Z = Z_0 \uplus Z_1$ telle que le contenu de la pile de toute configuration accessible commence par un symbole de Z_0 c'est-à-dire est dans $\varepsilon + Z_1^* Z_0$. Lorsque l'automate voit un sommet de pile de Z_0 il sait que celui-ci est le dernier symbole de la pile.

Il est possible de transformer un automate à pile quelconque en un automate à fond de pile testable. Pour cela, il faut doubler la taille de l'alphabet de pile en introduisant une copie $\bar{z}$ de chaque lettre z de Z . On pose alors $Z_0 = \{\bar{z} \mid z \in Z\}$ et $Z_1 = Z$. Chaque transition $p, y, z \rightarrow q, h$ donne alors les deux transitions $p, y, z \rightarrow q, h$ et $p, y, \bar{z} \rightarrow q, z_1 \cdots z_{n-1} \bar{z}_n$ si $h = z_1 \cdots z_n$.

Preuve. Voir Tds. □

2.6.3 Équivalence avec les grammaires

On montre dans cette partie que les automates à pile sont équivalents aux grammaires dans le sens où les langages acceptés par les automates à pile sont exactement les langages engendrés par les grammaires.

Théorème 2.63 (Équivalence grammaires/automates à pile). *Un langage $L \subset A^*$ est algébrique si et seulement si il existe un automate à pile qui accepte L .*

Preuve. On commence par montrer que pour toute grammaire G , il existe un automate à pile qui accepte les mots engendrés par G . Soit $G = (A, V, P)$ une grammaire telle que $L = L_G(S_0)$. On suppose que toute règle de G est soit de la forme $S \rightarrow w$ avec $w \in V^*$, soit de la forme $S \rightarrow a$ avec $a \in A$. Il suffit pour cela d'introduire une nouvelle variable V_a pour chaque lettre a de A . Cette transformation est identique à la première étape de la mise en forme normale quadratique (cf. la preuve de la proposition 2.16).

On construit un automate à pile sur l'alphabet d'entrée A . Son alphabet de pile est l'ensemble V des variables de G . Il a un unique état q_0 . Il possède une transition pour chaque règle de G . Son ensemble de transitions est donné par

$$\{q_0, a, S \rightarrow q_0, \varepsilon \mid S \rightarrow a \in P\} \cup \{q_0, \varepsilon, S \rightarrow q_0, h \mid S \rightarrow h \in P\}.$$

Pour les règles du premier ensemble, une lettre de l'entrée est lue et un symbole de pile est dépilé. Pour les règles du second ensemble, aucune lettre de l'entrée n'est lue et un ou plusieurs symboles de piles sont empilés. Il est facile de vérifier que cet automate simule les dérivations droites de la grammaire.

Lorsque la grammaire est en forme normale de Greibach, l'automate équivalent peut être construit sans utiliser d' ε -transition. On suppose que toute règle de S est de la forme $S \rightarrow aw$ où $a \in A$ et $w \in V^*$. L'ensemble des transitions de l'automate est alors donné par

$$\{q_0, a, S \rightarrow q_0, w \mid S \rightarrow aw \in P\}.$$

Pour la réciproque, on utilise la méthode des triplets de Ginsburg. Soit un langage L accepté par un automate à pile $\mathcal{A}$ acceptant par pile vide. Pour simplifier les notations, on suppose que chaque règle de l'automate empile au plus deux symboles. Si cette hypothèse n'est pas vérifiée, il suffit d'ajouter quelques états et quelques transitions pour la satisfaire. On note q_0 et z_0 l'état initial et le symbole de pile initial de $\mathcal{A}$.

Soient q et q' deux états et z un symbole de pile de l'automate. On note $L_{q,q',z}$ l'ensemble des mots f tels qu'il existe un calcul de la configuration (q, f, z) à la configuration $(q, \varepsilon, \varepsilon)$. On a alors les égalités suivantes.

$$\begin{aligned} L &= \bigcup_{q' \in Q} L_{q_0, q', z_0} \\ L_{q, q', z} &= \{y \mid q, y, z \rightarrow q', \varepsilon\} \\ &\cup \bigcup_{q, y, z \rightarrow q_1, z_1} y L_{q_1, q', z_1} \\ &\cup \bigcup_{q, y, z \rightarrow q_1, z_1 z_2} y L_{q_1, q_2, z_1} L_{q_2, q', z_2} \end{aligned}$$

Ces égalités se traduisent directement en une grammaire ayant pour variables les triplets de la forme (q, q', z) . Il est à remarquer que si l'automate ne possède pas d' ε -transition, alors la grammaire obtenue est en forme normale de Greibach. $\square$

Il faut remarquer que la traduction d'une grammaire en un automate donne un automate ayant un seul état. Ceci montre que tout automate à pile est en fait équivalent à un automate à pile ayant un seul état. Si de plus la grammaire est en forme normale de Greibach, l'automate n'a pas d' ε -transition. Ceci montre que tout automate à pile est équivalent à un automate à pile sans ε -transition.

Exemple 2.64. La construction d'un automate à partir de la grammaire $\{S \rightarrow aSa + bSb + a + b + \varepsilon\}$ pour engendrer les palindromes donne l'automate suivant. Cet automate a comme alphabet de pile $Z = \{S, A, B\}$ avec $z_0 = S$ comme symbole de pile initial. Cet automate a un seul état q et ses transitions sont les suivantes.

$$\begin{array}{lll} q, \varepsilon, S \rightarrow q, ASA & q, \varepsilon, S \rightarrow q, A & q, \varepsilon, S \rightarrow q, \varepsilon \\ q, \varepsilon, S \rightarrow q, BSB & q, \varepsilon, S \rightarrow q, B & q, a, A \rightarrow q, \varepsilon \\ & & q, b, B \rightarrow q, \varepsilon \end{array}$$

Cet automate est à comparer avec l'automate de l'exemple 2.61.

Exemple 2.65. La méthode des triplets appliquée à l'automate de l'exemple 2.60, donne la grammaire $\{R \rightarrow aRRR, S \rightarrow aRRS + aRST + aSTT + \varepsilon, T \rightarrow b\}$ où les variables R , S , et T correspondent aux triplets (q_0, q_0, z) , (q_0, q_1, z) et (q_1, q_1, z) . Après réduction et substitution de T par b , on obtient la grammaire $\{S \rightarrow aSbb + \varepsilon\}$ qui engendre bien le langage $L_1 = \{a^n b^{2n} \mid n \geq 0\}$.

2.6.4 Automates à pile déterministes

La définition d'un automate à pile déterministe est technique même si l'idée intuitive est relativement simple. L'intuition est qu'à chaque étape de calcul, il

n'y a qu'une seule transition possible. Ceci signifie deux choses. D'une part, si une ε -transition est possible, aucune autre transition est possible. D'autre part, pour chaque lettre de l'alphabet d'entrée, une seule transition est possible.

Définition 2.66. Un automate à pile (Q, A, Z, E, q_0, z_0) est *déterministe* si pour toute paire (p, z) de $Q \times Z$,

- soit il existe une unique transition de la forme $p, \varepsilon, z \rightarrow q, h$ et il n'existe aucune transition de la forme $p, a, z \rightarrow q, h$ pour $a \in A$,
- soit il n'existe pas de transition de la forme $p, \varepsilon, z \rightarrow q, h$ et pour chaque lettre $a \in A$, il existe au plus une transition de la forme $p, a, z \rightarrow q, h$.

Exemple 2.67. L'automate de l'exemple 2.60 n'est pas déterministe. Pour la paire (q_0, z) , cet automate possède une ε -transition $q_0, \varepsilon, z \rightarrow q_1, \varepsilon$ et une transition $q_0, a, z \rightarrow q_0, zzz$. Le langage $L_1 = \{a^n b^{2^n} \mid n \geq 0\}$ est accepté par pile vide par l'automate à pile déterministe ayant les états $Q = \{q_0, q_1, q_2\}$ et comportant les quatre transitions suivantes :

$$q_0, a, z \rightarrow q_1, zz \quad q_1, a, z \rightarrow q_1, zzz \quad q_1, b, z \rightarrow q_2, \varepsilon \quad q_2, b, z \rightarrow q_2, \varepsilon.$$

Dans le cas des automates à pile déterministes, les différents modes d'acceptation ne sont plus équivalents. L'acceptation par pile vide permet seulement d'accepter des langages préfixes, c'est-à-dire des langages tels que deux mots du langage ne sont jamais préfixe l'un de l'autre.

Un langage algébrique est dit *déterministe* s'il est accepté par un automate à pile déterministe. L'intérêt de cette classe de langages est sa clôture par complémentation qui montre par ailleurs que cette classe est une sous-classe stricte des langages algébriques (cf. corollaire 2.42 p. 77).

Proposition 2.68. *Le complémentaire d'un langage algébrique déterministe est un langage algébrique déterministe.*

Une autre propriété importante des langages déterministe est la suivante.

Proposition 2.69. *Tout langage algébrique déterministe est non ambigu.*

La réciproque de la proposition précédente est fausse comme le montre l'exemple suivant. Cet exemple montre également que la classe des langages déterministes n'est pas fermée pour l'union.

Exemple 2.70. Le langage $L = \{a^n b^n \mid n \geq 0\} \cup \{a^n b^{2^n} \mid n \geq 0\}$ est non ambigu. Par contre, il n'est pas déterministe. Même si l'idée intuitive de ce fait est relativement claire, la preuve en est particulièrement technique.

2.7 Compléments

Cette partie est consacrée à quelques compléments sur le groupe libre et les contenus de pile des automates. Dans les deux cas, l'approche est basée sur des réécritures. On commence donc par rappeler quelques résultats très classiques de réécriture.

2.7.1 Réécriture

On rappelle dans cette partie quelques résultats sur la terminaison et la confluence des relations qui seront utilisés par la suite. Pour une relation binaire notée $\rightarrow$, on note $\xrightarrow{*}$ la clôture réflexive et transitive de la relation $\rightarrow$. On dit qu'un élément x se réduit en y si $x \xrightarrow{*} y$.

Un élément x est dit *irréductible* ou *en forme normale* s'il n'existe pas d'élément y tel que $x \rightarrow y$. Une relation est *noethérienne* s'il n'existe pas de suite infinie $(x_n)_{n \geq 0}$ telle que $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \dots$. L'intérêt des relations noethériennes est que tout élément se réduit en un élément en forme normale.

Proposition 2.71. *Si la relation $\rightarrow$ est noethérienne, tout élément se réduit en un élément en forme normale.*

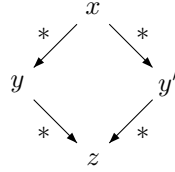


FIG. 2.6 – Propriété de confluence

Une relation $\rightarrow$ est *confluente* si pour tous éléments x, y et y' tels que $x \xrightarrow{*} y$ et $x \xrightarrow{*} y'$, il existe un élément z tel que $y \xrightarrow{*} z$ et $y' \xrightarrow{*} z$. Cette propriété est représentée par la figure 2.6. Elle aussi appelée propriété diamant. La confluence est parfois appelée propriété de Church-Rosser par référence à la confluence de la β -réduction du λ -calcul qui a été prouvée par Church et Rosser. L'intérêt principal de la confluence est qu'elle garantit l'unicité de la forme normale.

Proposition 2.72. *Si la relation $\rightarrow$ est confluente, tout élément se réduit en au plus un élément en forme normale.*

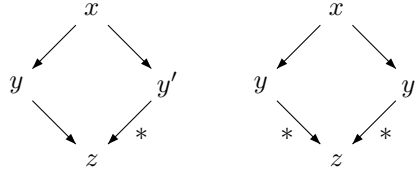


FIG. 2.7 – Confluences forte et locale

La confluence est souvent difficile à vérifier. Pour contourner cette difficulté, on introduit deux propriétés plus faciles à vérifier. Une relation $\rightarrow$ est *fortement confluente* si pour tous éléments x, y et y' tels que $x \rightarrow y$ et $x \rightarrow y'$, il existe un élément z tel que $y = z$ ou $y \rightarrow z$ et $y' \xrightarrow{*} z$. Une relation $\rightarrow$ est *localement confluente* si pour tous éléments x, y et y' tels que $x \rightarrow y$ et $x \rightarrow y'$, il existe un élément z tel que $y \xrightarrow{*} z$ et $y' \xrightarrow{*} z$. Les propriétés de confluence forte et de confluence locale sont représentées à la figure 2.7. La terminologie est cohérente car la confluence forte implique bien la confluence.

Proposition 2.73. *Toute relation fortement confluente est confluente.*

Preuve. Soient trois éléments x, y et y' tels que $x \xrightarrow{*} y$ et $x \xrightarrow{*} y'$. Par définition, il existe deux suites $y_0, \dots, y_m$ et $y'_0, \dots, y'_n$ d'éléments telles que $y_0 = y'_0 = x$, $y_m = y$, $y'_n = y'$, $y_i \rightarrow y_{i+1}$ pour $0 \leq i \leq m-1$ et $y'_i \rightarrow y'_{i+1}$ pour $0 \leq i \leq n-1$. On montre l'existence de z tel que $y \xrightarrow{*} z$ et $y' \xrightarrow{*} z$ par récurrence sur n . Si $n = 0$, le résultat est trivial puisque $y' = x$ et que $z = y$ convient. On suppose maintenant que $n \geq 1$. On définit par récurrence une suite d'éléments $z_0, \dots, z_m$ de la manière suivante. On pose $z_0 = y'_1$. On suppose avoir défini z_i tel que $y_i \rightarrow z_i$. On applique alors la confluence locale pour trouver z_{i+1} tel que $y_{i+1} \rightarrow z_{i+1}$ et $z_i \xrightarrow{*} z_{i+1}$. On peut alors appliquer l'hypothèse de récurrence à y'_1 puisque y'_1 se réduit en $n-1$ étapes à y' et en certain nombre d'étapes à z_m . On trouve alors z tel que $z_m \xrightarrow{*} z$ et $y' \xrightarrow{*} z$ et donc tel que $y \xrightarrow{*} z$ et $y' \xrightarrow{*} z$. $\square$

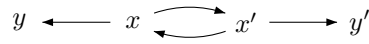


FIG. 2.8 – Relation localement confluente mais non confluente

La confluence locale n'implique pas la confluence dans le cas général. La relation représentée à la figure 2.8 est localement confluente mais elle n'est pas confluente. Par contre, elle n'est pas noethérienne puisque qu'on a la suite de réductions $x \rightarrow x' \rightarrow x \rightarrow \dots$. Le résultat suivant est souvent appelé lemme de Newman.

Proposition 2.74 (Newman). *Toute relation noethérienne et localement confluente est confluente.*

La confluence d'une relation de réécriture est souvent une façon d'obtenir une forme normale unique. Dans ce cas, la réécriture doit aussi être noethérienne pour assurer l'existence de la forme normale. L'hypothèse du lemme précédent est alors naturelle et peu contraignante.

Preuve. Soit une relation $\rightarrow$ noethérienne et localement confluente. Puisque $\rightarrow$ est noethérienne, tout élément se réduit à au moins un élément en forme normale. On va montrer que tout élément se réduit en fait à un seul élément en forme normale. Cela suffit pour prouver la confluence de la relation $\rightarrow$. Si les trois éléments x, y , et y' vérifient $x \xrightarrow{*} y$ et $x \xrightarrow{*} y'$, il existe des éléments z et z' en forme normale tels que $y \xrightarrow{*} z$ et $y' \xrightarrow{*} z'$. On déduit des relations $x \xrightarrow{*} z$ et $z \xrightarrow{*} z'$ que $z = z'$.

Supposons qu'il existe deux éléments en forme normale z et z' tels que $x \xrightarrow{*} z$ et $x \xrightarrow{*} z'$. On construit par récurrence une suite d'éléments $(x_n)_{n \geq 0}$ tels que $x_0 \rightarrow x_1 \rightarrow \dots$. On pose $x_0 = x$ et on suppose avoir défini x_i tel qu'il existe deux éléments z_i et z'_i en forme normale tels que $x_i \xrightarrow{*} z_i$ et $x_i \xrightarrow{*} z'_i$. Il existe alors des éléments y et y' tels que $x_i \rightarrow y \xrightarrow{*} z_i$ et $x_i \rightarrow y' \xrightarrow{*} z'_i$. Si $y = y'$, on choisit $x_{i+1} = y$. Sinon, on applique la confluence locale pour trouver z en forme normale tel que $y \xrightarrow{*} z$ et $y' \xrightarrow{*} z$. Si $z \neq z_i$, on choisit $x_{i+1} = y$, $z_{i+1} = z'_{i+1}$ et $z'_{i+1} = z$. Sinon on choisit $x_{i+1} = y'$, $z_{i+1} = z$ et $z'_{i+1} = z'_i$. $\square$

2.7.2 Contenus de pile

Nous allons utiliser les résultats sur les relations confluentes pour montrer que les contenus de pile des configurations accessibles d'un automate à pile forment un langage rationnel. Soit un automate à pile (cf. définition 2.58) d'alphabet d'entrée A , d'alphabet de pile Z , d'état initial q_0 et de symbole de pile initial z_0 . On définit le langage H de la manière suivante.

$$H = \{h \in Z^* \mid \exists f \in A^* \exists q \in Q \ (q_0, f, z_0) \xrightarrow{*} (q, \varepsilon, h)\}$$

Théorème 2.75. *Pour tout automate à pile, le langage H est rationnel.*

Monoïde polycyclique

Soit A_n l'alphabet $\{a_1, \dots, a_n, \bar{a}_1, \dots, \bar{a}_n\}$. On définit la relation $\rightarrow$ sur A_n^* de la manière suivante. Deux mots w et w' vérifient $w \rightarrow w'$ s'il existe deux mots u et v sur A_n et un indice i tel que $w = ua_i\bar{a}_i v$ et $w' = uv$. La relation $\rightarrow$ est noethérienne puisque $w \rightarrow w'$ implique $|w'| = |w| - 2$. On vérifie aussi facilement qu'elle est fortement confluyente et donc confluyente d'après la proposition 2.73. On note $\rho(w)$ l'unique mot irréductible tel que $w \xrightarrow{*} \rho(w)$. On définit alors la relation d'équivalence $\sim$ sur A_n^* par $w \sim w'$ si $\rho(w) = \rho(w')$. Cette relation est en fait une congruence sur A_n^* . Le *monoïde polycyclique* engendré par A_n est alors le monoïde quotient $A_n^*/\sim$.

Le lemme suivant établit un lien entre le monoïde polycyclique et le langage de Dyck (cf. exemple 2.6 p. 62).

Lemme 2.76. *Le langage $\{w \mid \rho(w) = \varepsilon\}$ est égal au langage de Dyck D_n^* sur n paires de parenthèses.*

Preuve. On montre par récurrence sur la longueur de la dérivation que tout mot de langage se réduit au mot vide. Inversement, on montre par récurrence sur la longueur de la réduction que tout mot qui se réduit au mot vide est dans le langage de Dyck. $\square$

On définit la substitution σ de A_n^* dans A_n^* par $\sigma(a) = D_n^* a D_n^*$ pour toute lettre a de A_n . Le lemme suivant établit la propriété clé de cette substitution.

Lemme 2.77. *Pour tout mot $w \in A_n^*$, on a $\sigma(w) = \{w' \mid w' \xrightarrow{*} w\}$.*

Preuve. Pour tout mot $w = w_1 \cdots w_k$, on a $\sigma(w) = D_n^* w_1 D_n^* w_2 \cdots D_n^* w_k D_n^*$. Il est alors évident d'après le lemme précédent que tout mot w' de $\sigma(w)$ vérifie $w' \xrightarrow{*} w$. Inversement, on prouve par récurrence sur la longueur de la réduction de w' à w que tout mot qui vérifie $w' \xrightarrow{*} w$ est dans $\sigma(w)$. $\square$

Corollaire 2.78. *Pour tout langage rationnel K de A_n^* , le langage $\rho(K) = \{\rho(w) \mid w \in K\}$ est rationnel.*

Preuve. Le langage $L = \{w' \mid \exists w \in K \ w \xrightarrow{*} w'\}$ est égal à $\sigma^{-1}(K)$ est il est donc rationnel d'après la proposition 1.130. Le langage $\rho(K)$ est égal à $L \setminus \bigcup_{i=1}^n A_n^* a_i \bar{a}_i A_n^*$ est il est donc aussi rationnel. $\square$

Nous sommes maintenant en mesure d'établir le théorème 2.75.

Preuve du théorème 2.75. Pour un mot $w = w_1 \cdots w_n$, on note $\tilde{w}$ le mot miroir $w_n \cdots w_1$ obtenu en renversant le mot w . On montre que le langage miroir $\tilde{H} = \{\tilde{h} \mid h \in H\}$ est rationnel, ce qui implique immédiatement que H est aussi rationnel.

Soit $\mathcal{A}$ un automate à pile. Quitte à renommer les symboles de pile, on suppose que l'alphabet de pile de $\mathcal{A}$ est $\{a_1, \dots, a_n\}$. Soit E l'ensemble des transitions de $\mathcal{A}$. On définit un morphisme μ de E^* dans A_n^* en posant $\mu(\tau) = \tilde{z}\tilde{h}$ pour toute transition τ égale à $q, y, z \rightarrow q', h$. Deux transitions τ et τ' de $\mathcal{A}$ sont dites consécutives si l'état d'arrivée de τ est l'état d'arrivée de τ' . Il faut bien noter que deux transitions consécutives ne peuvent pas nécessairement être enchaînées dans un calcul. En effet, le dernier symbole de pile empilé par τ n'est pas le même que le symbole de pile dépilé par τ' . Le fait qu'elles puissent être enchaînées dépend même du contenu de pile dans le cas où le mot empilé par τ est vide. On note K l'ensemble des suites consécutives de transitions de $\mathcal{A}$. Cet ensemble est bien sûr rationnel. On a alors l'égalité

$$\tilde{H} = \rho(z_0\mu(K)) \cap \{a_1, \dots, a_n\}^*$$

qui montre, grâce au corollaire précédent, que $\tilde{H}$ est rationnel. $\square$

2.7.3 Groupe libre

Soit A_n l'alphabet $\{a_1, \dots, a_n, \bar{a}_1, \dots, \bar{a}_n\}$. La fonction $a \mapsto \bar{a}$ est étendue à tout A_n en posant $\bar{\bar{a}}_i = a_i$ pour tout $1 \leq i \leq n$. Elle devient alors une involution sur A_n .

On définit la relation $\rightarrow$ sur A_n^* de la manière suivante. Deux mots w et w' vérifient $w \rightarrow w'$ s'il existe deux mots u et v sur A_n et une lettre a de A_n tel que $w = ua\bar{a}v$ et $w' = uv$. La relation $\rightarrow$ est noethérienne puisque $w \rightarrow w'$ implique $|w'| = |w| - 2$. On vérifie aussi facilement qu'elle est fortement confluente et donc confluente d'après la proposition 2.73. L'ensemble des mots irréductibles est $I = A_n^* \setminus A_n^*(\sum_{a \in A_n} a\bar{a})A_n^*$. On note $\rho(w)$ l'unique mot irréductible tel que $w \xrightarrow{*} \rho(w)$. On définit alors la relation d'équivalence $\sim$ sur A_n^* par $w \sim w'$ si $\rho(w) = \rho(w')$. Cette relation est en fait une congruence sur A_n^* . Le *groupe libre* engendré par A_n , noté $F(A_n)$, est alors le monoïde quotient $A_n^*/\sim$. Comme chaque classe d'équivalence de $\sim$ contient un seul mot irréductible, on peut identifier les éléments du $F(A_n)$ avec l'ensemble I des mots irréductibles sur A_n . Le produit dans $F(A_n)$ est alors défini par $u, v \mapsto \rho(uv)$ pour deux mots irréductibles u et v . Pour un élément x de $F(A_n)$, c'est-à-dire une classe de $\sim$, on note $\iota(x)$ l'unique mot de x qui est irréductible. L'application ι est l'*injection canonique* de $F(A_n)$ dans A_n^* . Pour un ensemble $X \subseteq F(A_n)$, on note également $\iota(X)$ l'ensemble $\{\iota(x) \mid x \in X\}$.

La terminologie est justifiée par la propriété suivante. Pour tout mot $w = w_1 \cdots w_k$, le mot w^{-1} égal à $\bar{w}_k \cdots \bar{w}_1$ est un inverse de w dans $A_n^*/\sim$. On vérifie en effet que $\rho(ww^{-1}) = \rho(w^{-1}w) = \varepsilon$. Le monoïde $A_n^*/\sim$ est en fait un groupe.

Le corollaire 2.78 reste vrai dans le cas de la réduction qui vient d'être introduite pour définir le groupe libre. Il en découle le résultat classique suivant.

Théorème 2.79 (Benois 1969). *Une partie $X \subseteq F(A_n)$ est une partie rationnelle de $F(A_n)$ si et seulement si $\iota(X)$ est une partie rationnelle de A_n^* .*

Le corollaire suivant découle immédiatement du théorème de Benoit.

Corollaire 2.80. *La famille des parties rationnelles de $F(A_n)$ est close pour les opérations booléennes (union, intersection et complémentation).*

Les parties reconnaissables du groupe libre sont les unions finies de classes d'un sous-groupe d'indice fini. En effet, si μ est un morphisme de monoïde de $A_n^*/\sim$ dans un monoïde fini M , alors M est nécessairement un groupe et μ est en fait un morphisme de groupe.

Sous-groupes rationnels

Lorsque le monoïde sous-jacent est un groupe G , les sous-groupes rationnels de G jouent un rôle important car ils ont la caractérisation suivante.

Proposition 2.81. *Un sous-groupe H d'un groupe G est une partie rationnelle de G si et seulement si il est finiment engendré.*

Preuve. Si le sous-groupe H est engendré par l'ensemble fini $\{g_1, \dots, g_k\}$, alors il est égal à l'étoile K^* de l'ensemble $K = \{g_1, \dots, g_k, g_1^{-1}, \dots, g_k^{-1}\}$ et il est donc rationnel.

Réciproquement, soit $\mathcal{A} = (Q, G, E, I, F)$ un automate qui accepte le sous-groupe H . Un chemin dans $\mathcal{A}$ est dit *simple* s'il ne repasse pas deux fois par le même état hormis peut-être pour l'état de départ et l'état d'arrivée qui peuvent coïncider. Un chemin est dit *presque simple* s'il se décompose $p \xrightarrow{u} q \xrightarrow{v} q \xrightarrow{w} r$ où les deux chemins $p \xrightarrow{u} q \xrightarrow{w} r$ et $q \xrightarrow{v} q$ sont simples. L'ensemble des calculs presque simples est fini puisqu'un tel chemin est de longueur au plus $2|Q|$. On note K l'ensemble fini des étiquettes des chemins réussis presque simples. On montre alors que H est engendré par K . Il est clair que $K \subseteq H$. Supposons par l'absurde que K n'engendre pas H et soit un chemin réussi c le plus court possible tel que son étiquette $h \in H$ n'est pas dans le sous-groupe engendré par K . Puisque le chemin c n'est pas simple (car sinon $h \in K$), il se décompose $p \xrightarrow{u} q \xrightarrow{v} q \xrightarrow{w} r$ où le calcul $q \xrightarrow{v} q \xrightarrow{w} r$ est presque simple. Il existe alors un calcul $i \xrightarrow{u'} q$ tel que le calcul $i \xrightarrow{u'} q \xrightarrow{v} q \xrightarrow{w} r$ soit réussi et presque simple. On a alors les égalités $h = uvw = gw(u'w)^{-1}u'vw$ qui montrent que h appartient au sous-groupe engendré par K . $\square$

Dans le cas du groupe libre, la proposition précédente et le corollaire du théorème de Benoit permettent de retrouver un résultat très classique.

Théorème 2.82 (Howson). *L'intersection de deux sous-groupes finiment engendrés d'un groupe libre $F(A_n)$ est encore un sous-groupe finiment engendré.*

Deuxième partie

Calculabilité et complexité

Chapitre 3

Calculabilité

L'objectif de ce chapitre est de définir la notion de calculable et de montrer que les réponses à certaines questions ne sont pas calculables. Cette notion va être introduite par l'intermédiaire des machines de Turing qui fournissent un modèle de calcul abstrait pas trop éloigné du fonctionnement d'un ordinateur. Il existe d'autres modèles de calcul comme les fonctions récursives ou le λ -calcul. Pour une grande partie de ce chapitre, une bonne référence est [Sip97].

3.1 Introduction

On commence par définir la notion de problème puis de codage qui permet de passer d'un problème à un langage. On introduit ensuite les machines de Turing qui nous serviront de modèle de calcul.

3.1.1 Notion de problème

On commence par définir un problème qui fixe un ensemble de questions appelées instances du problème ainsi qu'un sous ensemble de questions ayant une réponse positive. On s'intéresse à des problèmes de décisions pour lesquels les réponses aux questions sont *oui* ou *non*.

Définition 3.1 (Problème de décision). Un *problème de décision* est la donnée d'un ensemble E d'instances et d'un sous-ensemble $P \subseteq E$ des instances positives pour lesquelles la réponse est *oui*.

Exemple 3.2. On peut considérer les problèmes suivants :

- Nombres premiers : l'ensemble E des instances est l'ensemble $\mathbb{N}$ des entiers naturels et $P = \{n \mid n \text{ premier}\}$.
- 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 : $E = \{G \mid G \text{ graphe fini}\}$, $P = \{G \mid G \text{ est connexe}\}$.
- Grammaires ambiguës : $E = \{G \mid G \text{ grammaire}\}$, $P = \{G \mid G \text{ ambiguë}\}$ ou encore $P' = \{G \mid L_G(S) \text{ ambiguë}\}$.

3.1.2 Notion de codage

Définition 3.3 (Langage d'un problème). Pour associer un langage à un problème, on utilise un *codage* qui est une fonction *naturelle* de E dans Σ^* . Le codage de $x \in E$ est noté $\langle x \rangle$. Le *langage* associé à un problème P est l'ensemble $L_P = \{\langle x \rangle \mid x \in P\}$ des codages des instances positives.

Exemple 3.4. Quelques exemples de codages.

Nombres premiers. On peut prendre pour $\langle n \rangle$ l'écriture en base 10 ou en base 2 de n ou même l'écriture en base 1 où le codage de n est le mot 1^n sur l'alphabet $\Sigma = \{1\}$.

Grammaires. On peut adopter le codage suivant sur l'alphabet Σ formés des chiffres 0 et 1, des parenthèses (et) et de la virgule ','. Soit $G = (A, V, P)$ une grammaire. On associe à chaque symbole de $A + V$ un numéro entre 0 et $|A| + |V|$. Le codage $\langle a \rangle$ d'un symbole a est l'écriture binaire du numéro de a . Le codage $\langle w \rangle$ d'un mot $w = a_1 \cdots a_n$ est $(\langle a_1 \rangle, \dots, \langle a_n \rangle)$. Chaque règle $S \rightarrow w$ de P est codée par $(\langle S \rangle, \langle w \rangle)$. L'ensemble $P = \{r_1, \dots, r_m\}$ est codé par $(\langle r_1 \rangle, \dots, \langle r_m \rangle)$ et la grammaire est finalement codée par $\langle G \rangle = (\langle |A| \rangle, \langle |V| \rangle, \langle P \rangle)$.

3.1.3 Machines de Turing

Introduction

Une *machine de Turing* se compose d'une partie de contrôle et d'une bande infinie sur laquelle se trouvent écrits des symboles. La partie de contrôle est constituée d'un nombre fini d'états possibles et de transitions qui régissent les calculs de la machine. Les symboles de la bande sont lus et écrits par l'intermédiaire d'une *tête de lecture*.

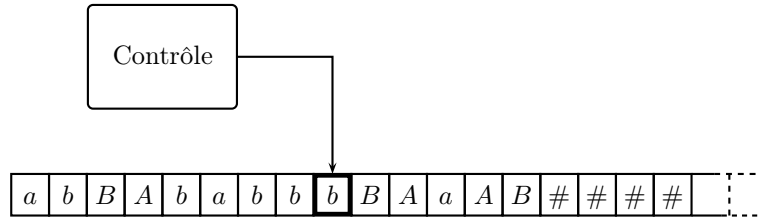


FIG. 3.1 – Machine de Turing

Les machines de Turing sont une abstraction des ordinateurs. La partie de contrôle représente le microprocesseur. Un élément essentiel est que le nombre d'états est *fini*. Ceci prend en compte que les microprocesseurs possèdent un nombre déterminé de registres d'une taille fixe et que le nombre de configurations possibles est fini. La bande représente la mémoire de l'ordinateur. Ceci comprend la mémoire centrale ainsi que les mémoires externes telles les disques durs. La tête de lecture représente la bus qui relie le microprocesseur à la mémoire. Contrairement à un ordinateur, la mémoire d'une machine de Turing est infinie. Ceci prend en compte qu'on peut ajouter des disques durs à un ordinateur de façon (presque) illimitée. Une autre différence entre une machine de Turing et

un ordinateur est que l'ordinateur peut accéder à la mémoire de manière directe (appelée aussi aléatoire) alors que la tête de lecture de la machine de Turing se déplace que d'une position à chaque opération.

Le contrôle de la machine est constitué d'un nombre fini d'états $q_0, \dots, q_n$. À chaque instant, la machine se trouve dans un de ces états. Au départ, la machine se trouve dans l'état q_0 qu'on appelle *état initial*. Le calcul d'une machine de Turing est formé d'une suite d'*étapes de calcul* qui sont effectuées par la machine. Chaque étape consiste à changer l'état de contrôle, écrire un symbole sous la tête de lecture et déplacer la tête de lecture. Les étapes de calcul possibles sont décrites par les transitions de la machine. Les transitions constituent en quelque sorte le programme de la machine.

Définition

Nous donnons maintenant la définition précise d'une machine de Turing. De manière formelle,

Définition 3.5. Une machine de Turing M est un septuplet $(Q, \Sigma, \Gamma, E, q_0, F, \#)$ où

- $Q = \{q_0, \dots, q_n\}$ est un ensemble fini d'*états de contrôle*.
- Σ est l'*alphabet d'entrée*. C'est un ensemble *fini* de symboles qui ne contient pas le symbole blanc $\#$. Cet alphabet est utilisé pour écrire la donnée initiale sur la bande.
- Γ est l'*alphabet de bande*. C'est un ensemble *fini* qui comprend tous les symboles qui peuvent être écrits sur la bande. Ceci inclut bien sûr l'alphabet d'entrée Σ et le symbole blanc $\#$.
- E est un ensemble fini de *transitions* de la forme (p, a, q, b, x) où p et q sont des états, a et b sont des symboles de bande et x est un élément de $\{L, R\}$. Une transition (p, a, q, b, x) est aussi notée $p, a \rightarrow q, b, x$.
- $q_0 \in Q$ est l'*état initial* dans lequel se trouve machine au début d'un calcul.
- F est l'ensemble des *états finaux* appelés aussi *états d'acceptation*.
- $\#$ est le *symbole blanc* qui, au départ, remplit toutes les positions de la bande autres que celles contenant la donnée initiale.

Une machine de Turing est *déterministe* si pour tout état p et tout symbole a , il existe au plus une transition de la forme $p, a \rightarrow q, b, x$. Lorsque la machine est déterministe, l'ensemble E de transitions est aussi appelé *fonction de transition* et est noté δ . La fonction δ associe à chaque paire (p, a) l'unique triplet (q, b, x) , s'il existe, tel que $p, a \rightarrow q, b, x$ soit une transition. Lorsque la machine n'est pas déterministe, l'ensemble E peut encore être vu comme une fonction δ . Dans ce cas, la fonction δ associe à chaque paire (p, a) l'ensemble des triplets (q, b, x) tels que $p, a \rightarrow q, b, x$ soit une transition.

À la place de L et de R , on utilise parfois les flèches $\leftarrow$ et $\rightarrow$ pour désigner les déplacements de la tête de lecture. C'est en particulier le cas sur certaines figures.

Exemple 3.6. La machine représentée à la figure 3.2 accepte les mots sur l'alphabet $\Sigma = \{a\}$ dont la longueur est une puissance de 2.

Exemple 3.7. La machine représentée à la figure 3.3 accepte l'ensemble $\{w \mid |w|_a = |w|_b\}$ des mots sur l'alphabet $\Sigma = \{a, b\}$ ayant le même nombre d'occurrences de a et de b .

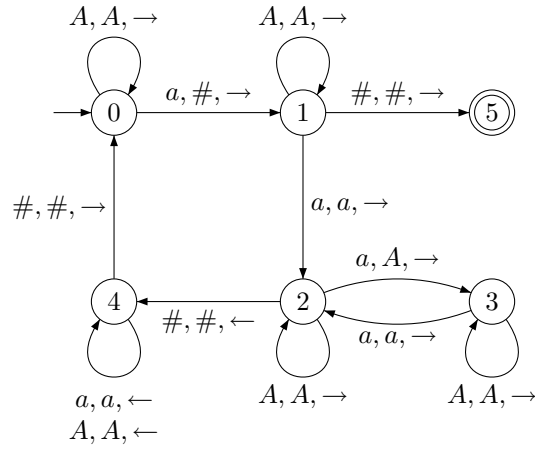


FIG. 3.2 – Un exemple de machine de Turing

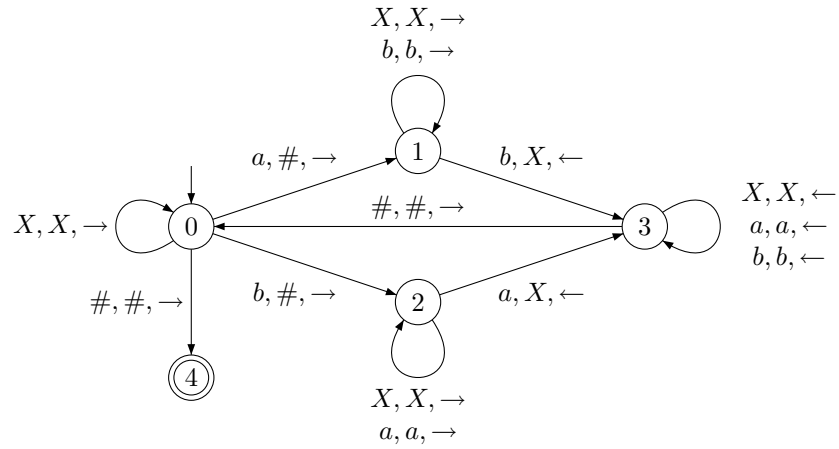


FIG. 3.3 – Un autre exemple de machine de Turing

Configurations et calculs

Le principe global de fonctionnement d'une machine de Turing est le suivant. Une entrée, c'est-à-dire un mot fini sur l'alphabet d'entrée, est écrit sur la bande la machine. Le reste de la bande est rempli avec le symbole blanc. La tête de lecture est placée sur le premier symbole du mot d'entrée. Ensuite, la machine commence son calcul jusqu'à arriver éventuellement dans un état où elle accepte l'entrée.

Pour décrire le fonctionnement précis d'une machine de Turing, il est nécessaire d'introduire la notion de configuration. Une *configuration* d'une machine de Turing est l'état *global* de la machine à un instant donné. Elle comprend

1. l'état de contrôle qui est un élément de Q ,
2. le contenu de la bande,
3. la position de la tête de lecture sur la bande.

Si la machine se trouve dans un état q , la configuration est écrite uqv où u est le contenu de la bande (strictement) à gauche de la tête de lecture et v est le contenu de la tête à droite de la tête de lecture (cf. la figure ci-dessous). Le symbole sous la tête de lecture est donc le premier symbole de v . Quand on décrit une configuration de la machine, les symboles blancs qui se trouvent dans la partie infinie à droite de la bande sont omis. Ceci permet de décrire une configuration de façon finie. Si par exemple, la machine de la figure 3.4 se trouve dans l'état q , sa configuration est $abBAbabqbBAaAB$ parce que le contenu de la bande avant la tête de lecture est $abBAbab$ et que le contenu après la tête de lecture est $bBAaAB$ suivi de symboles blancs. Tous les symboles $\#$ à droite sont implicites et ne sont pas écrits.

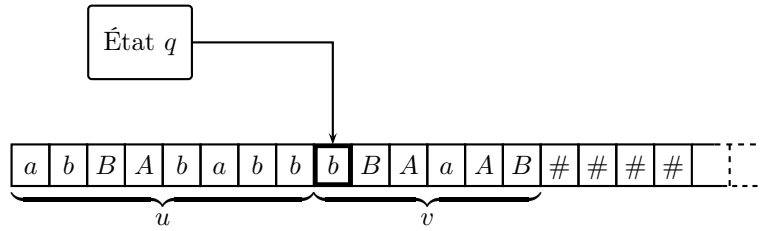


FIG. 3.4 – Configuration $abBAbabqbBAaAB$

Au départ, la bande contient la donnée initiale et la tête de lecture se trouve sur la première position de la bande. La *configuration initiale* s'écrit donc q_0w où q_0 est l'état initial et w est la donnée initiale.

Un calcul d'une machine de Turing se décompose en étapes. Une étape d'un calcul consiste à passer d'une configuration à une autre configuration en appliquant une des transitions de l'ensemble E . Une étape de calcul comprend les trois actions suivantes :

1. changer l'état de contrôle,
2. écrire un symbole à la place du symbole sous la tête de lecture,
3. déplacer la tête de lecture d'une position vers la gauche ou la droite.

Une transition de la forme $p, a \rightarrow q, b, x$ peut uniquement être appliquée si la machine se trouve dans l'état p et si le symbole sous la tête de lecture

est a . Après application de cette transition, la machine se trouve dans l'état q , le symbole a est remplacé sur la bande par b et la tête de lecture se déplace d'une position vers la gauche si $x = L$ (Left) ou d'une position vers la droite si $x = R$ (Right).

Définition 3.8 (Étape de calcul). Une *étape de calcul* est une paire de configuration (C, C') notée $C \rightsquigarrow C'$ telle que :

- soit $C = ucpav$ et $C' = uqcbv$ et $p, a \rightarrow q, b, L$ est une transition
- soit $C = upav$ et $C' = ubqv$ et $p, a \rightarrow q, b, R$ est une transition.

La présence de symboles blancs est implicite à la fin de d'une configuration. Dans la définition précédente, on s'autorise à ajouter un symbole blanc à la place de v si celui-ci est vide afin de pouvoir effectuer une transition de la forme $p, \# \rightarrow q, b, x$.

Il faut noter qu'une transition de la forme $p, a \rightarrow q, b, L$ (avec déplacement de la tête de lecture vers la gauche) peut uniquement être appliquée si la tête de lecture ne se trouve pas sur le premier symbole de la bande.

Définition 3.9 (Calcul). Un *calcul* est une suite de configurations successives $c_0 \rightsquigarrow c_1 \rightsquigarrow \dots \rightsquigarrow c_k$. Un calcul est dit *acceptant* si la configuration c_0 est *initiale*, c'est-à-dire $c_0 = q_0w$ pour $w \in \Sigma^*$ et si la configuration c_k est *finale*, c'est-à-dire $c_k = uqv$ avec $q \in F$.

Utilisation

Il y a deux modes d'utilisation des machines de Turing. Le premier mode est d'utiliser une machine comme *accepteur* et le second est d'utiliser une machine comme un *calculateur*.

Dans le mode *accepteur*, on fournit un mot d'entrée à la machine et celle-ci répond par *oui* ou par *non*. Quand la machine répond *oui*, on dit que la machine *accepte* le mot. La machine définit alors l'ensemble des mots qui sont acceptés. Par convention, on dit que la machine accepte un mot w s'il existe *au moins* un calcul acceptant avec w comme entrée, c'est-à-dire qui commence avec la configuration q_0w . L'élément important de cette définition est qu'un seul calcul acceptant suffit pour que la machine accepte même si d'autres calculs bloquent ou n'atteignent jamais une configuration acceptante.

Dans le mode *calculateur*, on fournit un mot d'entrée à la machine et celle-ci retourne un ou plusieurs mots de sortie. Quand la machine ne donne toujours qu'un seul mot de sortie, elle calcule une fonction qui associe le mot de sortie au mot d'entrée. Les mots de sortie sont par convention les contenus de la bande des dernières configurations des calculs acceptants. On met donc le mot d'entrée sur la bande, la machine effectue un calcul jusqu'à ce qu'elle atteigne un état final et le contenu de la bande constitue alors un des mots de sortie. Comme il peut y avoir plusieurs calculs acceptants, il peut y avoir plusieurs mots de sortie. Dans le cas d'une machine déterministe, il y a au plus un seul calcul acceptant et donc au plus un seul mot de sortie.

Définition 3.10 (Langage accepté). On dit que w , un mot de Σ^* est *accepté* par M s'il existe un calcul acceptant de configuration initiale q_0w . L'ensemble des mots acceptés par M est noté $L(M)$.

3.1.4 Normalisation

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_- .

Plus formellement, on a la proposition suivante.

Proposition 3.11 (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_- .

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.

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 la machine soit dans état p et que le symbole de bande sous la tête soit le symbole a . Si la machine n'a pas de transition de la forme $p, a \rightarrow q, b, x$ avec $x \in \{L, R\}$, 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 la machine soit dans état p , que la tête de lecture soit sur la première position de la bande et que le symbole de bande sous cette tête soit 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.

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 ces 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 $p, a \rightarrow q, b, x$ avec $x \in \{L, R\}$. 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 $\bar{a}$. L'alphabet de bande M' est donc $\Gamma' = \Gamma \cup \{\bar{a} \mid a \in \Gamma\}$. 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 $\bar{a}$ 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 a par un caractère b et un caractère $\bar{a}$ par un caractère $\bar{b}$. Ainsi, la première position de la bande contient toujours un symbole de la forme $\bar{a}$ et toutes les autres positions contiennent des symboles de la forme a .

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 mêmes 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 : $\Gamma' = \Gamma \cup \{\bar{a} \mid a \in \Gamma\}$
- 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 a de $\Sigma \cup \{\#\}$ de l'entrée par le caractère $\bar{a}$ correspondant.

$$E_0 = \{q'_0, a \rightarrow q'_1, \bar{a}, R \mid a \in \Gamma\}.$$

$$p = q'_1$$

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.

$$E_1 = \{q'_1, a \rightarrow q_0, a, L \mid a \in \Gamma\}.$$

$$p = q_+ \text{ ou } p = q_-$$

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

$$p \in Q$$

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 écrivant à la position -1 un nouveau symbole $\$$ n'appartenant pas à Γ . Comme aucune transition ne permet de lire ce symbole, la machine se bloque dès qu'elle repasse par cette position de la bande.

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. La case de numéro $-i$ se retrouve en dessous la case de numéro i . Une case fictive remplie d'un nouveau symbole $\$$ est mise sous la case de numéro 0 (cf. figure ci-dessous). Ceci permet en outre à la machine de reconnaître la première position de la bande.

0	1	2	3	4	5	6	7	
A	b	a	B	#	B	#	#	...
\$	#	b	#	B	#	#	#	...
	-1	-2	-3	-4	-5	-6	-7	

Le nouvel alphabet de bande est $\Gamma \times (\Gamma \cup \{\$\})$ puisqu'on considère à chaque position i de la nouvelle bande le couple des caractères aux positions $(i, -i)$ dans l'ancienne bande. De plus la case de numéro 0 contient un symbole de $\Gamma \times \{\$\}$.

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 première opération de la machine consiste à mettre le symbole $\$$ dans la partie inférieure de la première case de la bande en remplaçant le symbole $(a, \#)$ par $(a, \$)$.

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. $\square$

Machines à plusieurs bandes

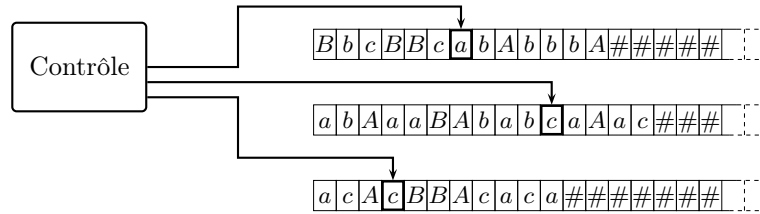


FIG. 3.6 – Machine de Turing à trois bandes

Définition 3.14 (Machine à plusieurs bandes). Une *machine à k bandes* est une machine disposant de k bandes, chacune lue par une tête de lecture indépendante (cf. figure 3.6). Une transition est alors un élément de l'ensemble $Q \times \Gamma^k \times Q \times \Gamma^k \times \{L, R, S\}^k$, où S (pour *Stay*) permet éventuellement de laisser immobile certaines têtes de lecture.

Lorsqu'une machine à une seule bande, il n'est pas vraiment nécessaire d'avoir la possibilité de laisser la tête de lecture à sa position. En effet, le symbole lu à la transition suivante est celui qui vient d'être écrit. Par contre, lorsque la machine a plusieurs mode, il est plus commode de pouvoir laisser une tête en place tout en déplaçant les autres.

Proposition 3.15 (Équivalence). *Touche machine de Turing M à plusieurs bandes est équivalente à une machine de Turing M' à une seule bande qui accepte les mêmes entrées. De plus, si M est déterministe (resp. s'arrête sur chaque entrée), M' est aussi déterministe (resp. s'arrête aussi sur chaque entrée).*

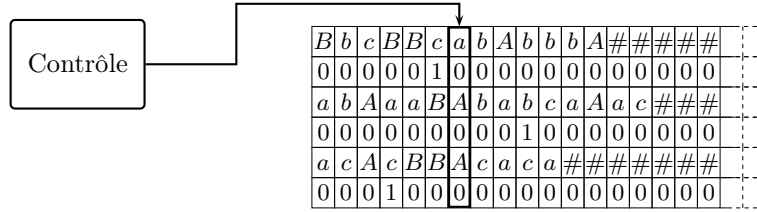


FIG. 3.7 – Conversion de trois bandes en une seule

Preuve. Soit M une machine de Turing à k bandes pour un entier k . On construit une machine de Turing M' à une seule bande qui accepte les mêmes entrées.

Une première solution est de construire une machine M' qui stocke les contenus des k bandes sur une seule bande en les séparant par un nouveau symbole $\$$. Outre le contenu des bandes, M' doit aussi garder trace des positions des k têtes de lecture. Elle possède donc un symbole de bande $\downarrow$ qui est placé juste avant les symboles qui étaient sous les têtes de lecture de M . La simulation d'une seule transition de M se fait en parcourant toute la bande pour noter dans l'état de contrôle les symboles sous les k têtes de lectures. En fin de bande, la machine M' choisit la transition de M à effectuer en fonction de l'état et des k symboles. La machine effectue un nouveau parcours de la bande pour changer les caractères sous les k têtes de lecture puis déplacer les k têtes de lecture.

Une seconde solution consiste à *coller* les k bandes pour en faire une seule sur l'alphabet Γ^k . Comme il faut aussi garder trace des positions des k têtes de lecture, on agrandit encore l'alphabet pour en avoir une seule bande sur l'alphabet $(\Gamma \times \{0, 1\})^k$ (cf figure 3.7). De nouveau, la simulation d'une seule transition de M se fait par deux parcours successifs de toute la bande de M . $\square$

Déterminisme

Définition 3.16 (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 $(q, b, x) \in Q \times \Gamma \times \{L, R\}$ tel que $p, a \rightarrow q, b, x \in E$.

La définition ci-dessus a été donnée pour une machine à une seule bande. Elle s'étend facilement au cas des machines à plusieurs bandes. Les machines des figures 3.2 et 3.3 sont déterministes.

Proposition 3.17. *Toute machine de Turing M est équivalente à une machine M' déterministe. Si de plus M est sans calcul infini, la machine M' est aussi sans calcul infini.*

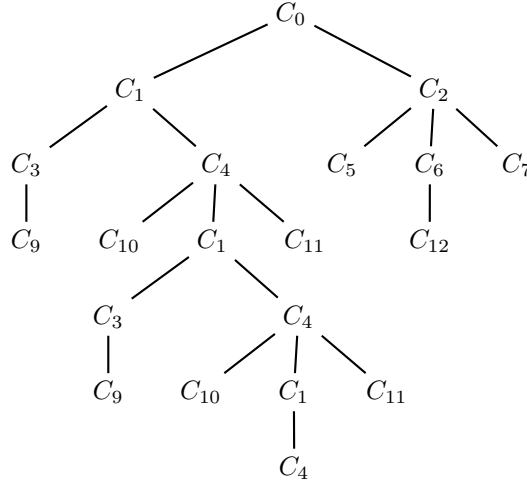


FIG. 3.8 – Un arbre de calcul

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 rencontre un état acceptant au cours de cette exploration, elle accepte l'entrée. Sinon, la simulation de D ne se termine pas.

Soit $w \in \Sigma^*$ une entrée de la machine N . On considère l'ensemble des calculs de N sur w comme un arbre. Les nœuds sont les calculs finis et valides de N . La racine de l'arbre est le calcul vide consistant en la configuration initiale $C_0 = q_0w$. Les fils d'un calcul γ sont tous les calculs obtenus en prolongeant γ par une étape de calcul. Si γ est le calcul $C_0 \rightsquigarrow \dots \rightsquigarrow C_n$, ses fils sont alors tous les calculs de la forme $C_0 \rightsquigarrow \dots \rightsquigarrow C_n \rightsquigarrow C_{n+1}$. Le nœud γ a donc autant de fils qu'il existe de configurations C telles que $C_n \rightsquigarrow C$. Sur la figure 3.8, chaque nœud γ est étiqueté par sa dernière configuration C_n . Le calcul peut être reconstitué en suivant la suite des configurations le long de la branche. Dans cette représentation, une configuration peut apparaître plusieurs fois puisque plusieurs calculs peuvent conduire à une même configuration. Pour une configurations C donnée, le nombre de configurations C' telles que $C \rightsquigarrow C'$ est borné par le nombre de transitions de la machine. L'arbre des calculs est donc de degré borné. L'idée générale est de faire parcourir cet arbre par la machine D . Comme certaines branches peuvent être infinies même si la machine N accepte w , il est impossible d'effectuer un parcours en profondeur. La machine D pourrait descendre indéfiniment le long d'une même branche infinie et manquer une configuration acceptante sur une autre branche. La machine D va donc effectuer un parcours en largeur de l'arbre des calculs.

Soit un mot $x = \tau_1 \cdots \tau_n$ sur l'alphabet E des transitions de N . Le mot x détermine l'unique calcul obtenu en effectuant à partir de la configuration q_0w les transitions $\tau_1, \dots, \tau_n$. Bien sûr, tout mot x ne correspond pas nécessairement à un calcul valide et à un nœud de l'arbre. La machine D va parcourir les nœuds de l'arbre dans l'ordre des mots correspondants où les mots sont d'abord ordonnés par longueur puis par ordre lexicographique pour les mots d'une même longueur. Pour chaque mot sur E , la machine D vérifie en simulant N si ce mot correspond à un calcul valide. Si la configuration atteinte est acceptante la machine D accepte. Sinon, elle passe au mot suivant et continue.

Dans l'implémentation, les transitions de N sont numérotées par les entiers de 0 à $k-1$ où k est le nombre de ces transitions. On identifie alors un mot sur l'alphabet $\{0, \dots, k-1\}$ avec un mot sur E .

Mise en application. La machine déterministe D est construite en utilisant trois bandes. D'après la proposition 3.17, cette machine est à nouveau équivalente à une machine déterministe avec une seule bande. La machine D utilise ses trois bandes de la manière suivante. La première bande contient toujours l'entrée w de la machine et n'est jamais modifiée. La seconde bande contient une copie de la bande de N au cours de son calcul le long d'une branche. La troisième bande contient un mot sur l'alphabet $\{0, \dots, k-1\}$ qui détermine éventuellement un calcul de N .

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 le long du calcul déterminé par la troisième bande. Pour chaque étape de la simulation de N , le symbole de la troisième bande de D donne la transition de N qui doit être appliquée. Si cette transition ne peut pas être appliquée ou s'il ne reste plus de symboles sur la troisième bande, la simulation de ce calcul est abandonnée et la machine D passe à l'étape 4. Si une configuration acceptante de N est rencontrée, la machine D s'arrête et accepte l'entrée w .
4. Remplacer la chaîne de la troisième bande avec la chaîne qui suit dans l'ordre hiérarchique. La machine D retourne à l'étape 2 pour simuler ce nouveau calcul.

Dans le cas où la machine M est sans calcul infini, l'arbre des calculs est fini d'après le lemme de König (cf. lemme 3.18) et donc de hauteur finie. Ceci signifie qu'il existe une borne sur la longueur des calculs partant de la configuration q_0w . Pour que la machine D soit sans calcul infini si N est sans calcul infini, la machine D procède de la manière suivante. Pour chaque longueur l de mots sur la troisième bande, la machine D vérifie si elle a trouvé au moins un calcul de longueur l . Si elle n'en a trouvé aucun, elle s'arrête en rejetant l'entrée w . Ceci garantit que la machine D s'arrête sur toute entrée. $\square$

On prouve ici le lemme de König qui a été utilisé dans la preuve de la proposition précédente. Ce lemme est en fait un argument de compacité sous une forme un peu déguisée.

Lemme 3.18 (König 1936). *Tout arbre infini dont tout nœud est de degré fini contient une branche infinie.*

Preuve. On construit par récurrence une suite $(x_n)_{n \geq 0}$ de nœuds qui forment une branche infinie de l'arbre. On prend pour x_0 la racine de l'arbre. On suppose avoir choisi un nœud x_n tel que le sous-arbre enraciné en x_n est infini. Puisque x_n a un nombre fini de fils, au moins un des sous-arbres enracinés en l'un de ses fils est infini. On choisit alors comme nœud x_{n+1} un tel fils de x_n . $\square$

3.2 Langages récursivement énumérables

Définition 3.19 (Langages récursivement énumérables). Un langage $L \subseteq \Sigma^*$ est *récursivement énumérable* s'il existe une machine de Turing M telle que $L = L(M)$. Par extension, un problème P est *récursivement énumérable* si L_P est récursivement énumérable.

La terminologie est justifiée par la proposition suivante (proposition 3.21). Les langages récursivement énumérables sont aussi appelés *semi-récurifs* ou *semi-décidables*. Le terme *récurif* provient des fonctions récursives (cf. section 3.8). Le terme *semi* est justifié par la proposition 3.26.

Définition 3.20 (Énumérateur). Un énumérateur est une machine de Turing déterministe qui écrit sur sa première bande des mots de Σ^* séparés par $\# \notin \Sigma$. La tête de lecture de cette première bande ne se déplace jamais vers la gauche.

Par définition, l'ensemble des mots énumérés est l'ensemble des mots sur Σ qui apparaissent au moins une fois sur la première bande de l'énumérateur. Il n'y a aucune contrainte sur la façon dont les mots sont énumérés. Ils peuvent apparaître dans n'importe quel ordre et chaque mot peut apparaître une ou plusieurs fois. La proposition suivante justifie la terminologie *récursivement énumérable* employée pour un langage accepté par une machine de Turing.

Proposition 3.21. *Un langage $L \subseteq \Sigma^*$ est récursivement énumérable si et seulement si L est l'ensemble des mots énumérés par un énumérateur.*

Preuve. On va montrer qu'à partir d'une machine de Turing qui accepte L , on peut construire un énumérateur qui énumère exactement les mots de L et qu'inversement on peut construire une machine de Turing qui accepte L à partir d'un énumérateur.

On suppose d'abord disposer d'une machine M qui accepte L . Grâce à la proposition 3.17, on peut supposer que M est déterministe. L'énumérateur ne peut pas utiliser directement la machine M sur chaque mot car celle-ci peut avoir des calculs infinis. L'idée consiste à simuler la machine sur un nombre maximal de transitions puis d'augmenter progressivement ce nombre maximal de transitions. On obtient alors l'algorithme ci-dessous.

Algorithme (Machine de Turing $\rightarrow$ Énumérateur).

```

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

```


end for

On suppose maintenant disposer d'un énumérateur pour L . La machine de Turing simule l'énumérateur et compare successivement chaque mot imprimé par celui-ci avec son entrée. Elle accepte en cas d'égalité. Si le mot n'est jamais énuméré, le calcul de la machine se poursuit indéfiniment sans que la machine n'accepte. On obtient alors l'algorithme ci-dessous.

Algorithme (Énumérateur $\rightarrow$ 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

```

□

Un argument de cardinalité montre qu'il existe des langages non récursivement énumérables. Pour tout alphabet Σ , l'ensemble de tous les langages sur Σ n'est pas dénombrable. Au contraire, l'ensemble des machines de Turing et donc des langages récursivement énumérables est dénombrable. Cet argument est semblable à celui qui montre qu'il existe des nombres transcendants parce que l'ensemble des nombres algébriques est dénombrable. L'exemple suivant exhibe un langage qui n'est pas récursivement énumérable. La construction de ce langage est basée sur un argument diagonal très classique.

Exemple 3.22 (Langage diagonal). Soit un alphabet Σ fixé égal par exemple à $\{a, b\}$. Puisque l'ensemble des machines de Turing est dénombrable, on indexe par les entiers les machines de Turing d'alphabet d'entrée Σ . On obtient une suite $M_0, M_1, M_2, \dots$ de toutes les machines de Turing sur Σ . On indexe aussi les mots de Σ^* pour obtenir une suite $w_0, w_1, w_2, \dots$ de tous les mots sur Σ . On définit alors le langage L par

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

On montre par l'absurde que L n'est pas récursivement énumérable. On suppose qu'il existe une Machine de Turing M telle que $L = L(M)$. On note k l'indice de cette machine, c'est-à-dire l'entier k tel que $M = M_k$. Si $w_k \in L(M_k)$, alors $w_k \notin L(M)$ et si $w_k \notin L(M_k)$, alors $w_k \in L(M)$, ce qui conduit dans les deux cas à une contradiction.

La proposition suivante établit que la classe des langages récursivement énumérables est close pour l'union et l'intersection.

Proposition 3.23. *Si les langages L et L' sont récursivement énumérables, alors les deux langages $L \cup L'$ et $L \cap L'$ sont aussi récursivement énumérables.*

Preuve. Soient L et L' deux langages acceptés par des machines M et M' qu'on peut supposer à une seule bande grâce à la proposition 3.15. On construit des machines $M_\cup$ et $M_\cap$ qui acceptent respectivement les langages $L \cup L'$ et $L \cap L'$.

On construit une machine $M_{\cup}$ avec deux bandes. La machine commence dans un premier temps par recopier son entrée de la première bande sur la seconde bande. Dans un second temps, la machine alterne, transition par transition la simulation de M sur la première bande et la simulation de M' sur la seconde bande. La machine $M_{\cup}$ accepte dès qu'une des deux machines M et M' accepte.

Pour la machine $M_{\cap}$, on peut procéder de la même façon en alternant les simulations des deux machines M et M' et en acceptant lorsque les deux ont accepté. Une autre solution est de simuler d'abord M sur l'entrée après avoir sauvegardé cette entrée sur une autre bande, puis de simuler M' lorsque M a accepté. Si l'une des deux machines ne s'arrête pas, le calcul de $M_{\cap}$ est également infini. Si les deux machines acceptent, la machine $M_{\cap}$ accepte. $\square$

3.3 Langages décidables

Les problèmes et les langages décidables sont ceux pour lesquels il existe un algorithme qui donne la réponse à chaque instance. Par algorithme, on entend une méthode qui donne la réponse correcte en un temps fini.

Définition 3.24 (Langage décidable). Un langage $L \subseteq \Sigma^*$ est *décidable* s'il existe une machine de Turing M sans calcul infini (*i.e.* la machine M s'arrête sur toute entrée) telle que $L = L(M)$. On dit que la machine M *décide* le langage L .

Les langages décidables sont parfois appelés *rékursifs*. Par définition même, les langages décidables sont récursivement énumérables. On verra à la proposition 3.28 qu'il existe des langages récursivement énumérables qui ne sont pas décidables. On commence par quelques propriétés de clôture des langages décidables.

Proposition 3.25 (Opérations booléennes). *Si les langages $L \subset \Sigma^*$ et L' sont décidables, alors les langages $L \cup L'$, $L \cap L'$ et $\Sigma^* \setminus L$ sont aussi décidables.*

Preuve. Soient M et M' deux machines de Turing sans calcul infini qui acceptent les langages L et L' . Les machines $M_{\cup}$ et $M_{\cap}$ construites dans la preuve de la proposition 3.23 sont sans calcul infini et elles acceptent les langages $L \cup L'$ et $L \cap L'$.

D'après les propositions 3.17 et 3.11, on peut supposer que la machine M est déterministe et normalisée. En échangeant les rôles des états q_+ et q_- , on obtient une machine qui accepte le complémentaire $\Sigma^* \setminus L$ de L . $\square$

La proposition suivante justifie la terminologie *semi-décidables* ou *semi-rékursifs* qui est parfois employée pour les langages récursivement énumérables.

Proposition 3.26 (Complémentation et décidabilité). *Soit $L \subset \Sigma^*$ un langage. Si L et son complémentaire $\Sigma^* \setminus L$ sont récursivement énumérables, alors L et $\Sigma^* \setminus L$ sont décidables.*

Preuve. On suppose que les machines M et M' acceptent les langages L et $\Sigma^* \setminus L$. Par contre, ces deux machines peuvent avoir des calculs infinis. On construit une machine M_1 semblable à la machine $M_{\cup}$ utilisée dans la preuve de la proposition 3.23 mais avec une condition d'arrêt différente. La machine M_1 simule en parallèle les deux machines M et M' et s'arrête dès qu'une des deux

machines M ou M' accepte. Si la machine M accepte, la machine M_1 accepte et si la machine M' accepte, la machine M_1 rejette. Comme les machines M et M' acceptent les langages L et $\Sigma^* \setminus L$, une des deux machines doit accepter pour toute entrée w . Ceci assure que la machine M_1 s'arrête sur toute entrée. $\square$

Pour une machine M et un mot w , on note respectivement $\langle M \rangle$, $\langle w \rangle$ et $\langle M, w \rangle$ les codages de M , w et de la paire (M, w) . Le codage de (M, w) peut être par exemple $\langle M \rangle \$ \langle w \rangle$ où le symbole $\$$ n'apparaît ni dans le codage de M ni dans le codage de w . On définit maintenant le *langage d'acceptation* noté L_{acc} par

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

Proposition 3.27 (Machine universelle). *Le langage L_{acc} est récursivement énumérable. Une machine de Turing M_U telle que $L(M_U) = L_{acc}$ est appelée machine universelle.*

Une machine universelle est en quelque sorte une machine capable de simuler n'importe qu'elle autre machine qui lui est passée en paramètre. Les interpréteurs de langages comme la machine virtuelle de Java (*Java Virtual Machine*) sont des machines universelles.

Preuve. On construit une machine universelle M_U avec deux bandes. La première bande reçoit l'entrée qui est le codage $\langle M, w \rangle$ d'une machine de Turing M et d'une entrée w . La machine M_U va simuler le calcul de M sur l'entrée w . La seconde bande de M_U est utilisée pour contenir la configuration de la machine M au cours du calcul. Cette bande est initialisée avec la configuration $q_0 w$ obtenue en recopiant l'état initial de M et la donnée w . Ensuite, pour chaque étape de calcul, la machine M_U recherche dans $\langle M \rangle$ une transition applicable sur la configuration puis l'applique pour obtenir la configuration suivante. $\square$

Proposition 3.28. *Le langage L_{acc} n'est pas décidable.*

D'après la proposition 3.26, si un langage L et son complémentaire sont récursivement énumérables, alors L est décidable. La proposition précédente a immédiatement le corollaire suivant.

Corollaire 3.29. *Le complémentaire de L_{acc} n'est pas récursivement énumérable.*

La preuve de la proposition précédente se fait par un argument diagonal même si celui-ci n'est pas complètement explicite. La machine Q est en effet lancée sur son propre codage pour aboutir à une contradiction.

Preuve. On montre le résultat par un raisonnement par l'absurde. On suppose que le langage L_{acc} est décidable. Soit A une machine de Turing qui décide le langage L_{acc} . Ceci signifie que $L(A) = L_{acc}$ et que la machine A s'arrête sur toute entrée. On construit alors la machine Q de la manière suivante.

```

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

```

On montre maintenant que l'on a une contradiction en lançant la machine Q sur l'entrée $\langle Q \rangle$.

- Si Q accepte $\langle Q \rangle$, alors A accepte l'entrée $\langle Q, \langle Q \rangle \rangle$ par définition de A . Mais si A accepte $\langle Q, \langle Q \rangle \rangle$, Q rejette $\langle Q \rangle$ par définition de Q .
- Si Q rejette $\langle Q \rangle$, alors A rejette l'entrée $\langle Q, \langle Q \rangle \rangle$ par définition de A . Mais si A rejette $\langle Q, \langle Q \rangle \rangle$, Q accepte $\langle Q \rangle$ par définition de Q .

Dans les deux cas, on a abouti à une contradiction. Ceci montre que le langage L_{acc} n'est pas décidable $\square$

Il n'est pas toujours aussi facile d'établir directement qu'un langage ou un problème est indécidable. On utilise alors une réduction pour se ramener à un autre problème indécidable. On commence par la définition d'une fonction calculable.

Définition 3.30. Une fonction $f : \Sigma^* \rightarrow \Gamma^*$ est *calculable* s'il existe une machine de Turing qui pour toute entrée w s'arrête avec $f(w)$ sur la bande.

Comme toute machine de Turing est équivalente à une machine de Turing déterministe, toute fonction calculable l'est aussi par une machine de Turing déterministe. La classe des fonctions calculables est close par de nombreuses opérations. En particulier, la composée de deux fonctions calculables est encore une fonction calculable.

Définition 3.31 (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 $f : \Sigma_A^* \rightarrow \Sigma_B^*$ calculable telle que $w \in L_A \iff f(w) \in L_B$. On note $A \leq_m B$ lorsque A se réduit à B .

La notation $A \leq_m B$ suggère que le problème A est moins compliqué que le problème B . L'indice m provient de l'expression anglaise *many to one* qui s'oppose à *one to one* et qui indique que la fonction f n'est pas nécessairement injective. Plusieurs instances de A peuvent être envoyées par la réduction sur la même instance de B . Puisque la composée de deux fonctions calculables est calculable, la relation $\leq_m$ est bien sûr transitive.

Proposition 3.32 (Réduction). Si $A \leq_m B$ et si 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 $w \in L_A \iff w' \in L_B$.

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 . $\square$

Par contraposition de la proposition précédente, on obtient le corollaire suivant qui s'utilise très souvent pour montrer qu'un problème B est indécidable.

Corollaire 3.33 (Réduction). Si $A \leq_m B$ et si A est indécidable, alors B est indécidable.

Afin d'illustrer le corollaire précédent on va prouver la proposition suivante qui donne deux autres problèmes indécidables.

Proposition 3.34. *Les deux langages $L_\emptyset = \{\langle M \rangle \mid L(M) \neq \emptyset\}$ et $L_\neq = \{\langle M_1, M_2 \rangle \mid L(M_1) \neq L(M_2)\}$ sont indécidables.*

Preuve. Pour utiliser le corollaire précédent et montrer que le langage $L_\emptyset$ est indécidable, on construit une réduction de L_{acc} à $L_\emptyset$. Pour toute paire (M, w) on considère la machine M_w définie de la manière suivante.

```

input  $u$ 
if  $u = w$  then
    simuler  $M$  sur  $w$ 
else
    rejeter
end if

```

La fonction f qui associe $\langle M_w \rangle$ à $\langle M, w \rangle$ est bien sûr calculable par une machine de Turing déterministe. De plus on a aussi l'équivalence $w \in L(M) \iff L(M') \neq \emptyset$ car le mot w est le seul mot qui puisse être accepté par M_w . La fonction f ainsi définie est donc une réduction de L_{acc} à $L_\emptyset$. Le corollaire précédent assure que le langage $L_\emptyset$ est indécidable.

Pour utiliser à nouveau le corollaire précédent et montrer que le langage $L_\neq$ est indécidable, on construit une réduction de $L_\emptyset$ à $L_\neq$. On considère une machine $M_\emptyset$ qui accepte le langage vide. Il suffit de prendre pour $M_\emptyset$ une machine ayant aucun état final. La fonction g qui associe $\langle M, M_\emptyset \rangle$ à $\langle M \rangle$ est bien sûr calculable par une machine de Turing déterministe. De plus on a l'équivalence $L(M) \neq \emptyset \iff L(M) \neq L(M_\emptyset)$. La fonction g ainsi définie est donc une réduction de $L_\emptyset$ à $L_\neq$. Le corollaire précédent assure que le langage $L_\neq$ est indécidable. $\square$

Les deux résultats de la proposition précédente sont en fait deux cas particuliers d'un théorème plus général. On dit qu'une propriété P sur les langages est *non triviale* s'il existe au moins un langage récursivement énumérable L qui vérifie P et au moins un langage récursivement énumérable L' qui ne vérifie pas P .

Théorème 3.35 (Rice). *Pour toute propriété non triviale P sur les langages, le problème de savoir si le langage $L(M)$ d'une machine de Turing M vérifie P est indécidable.*

Si une propriété P est triviale, tous les langages récursivement énumérables vérifient ou ne vérifient pas la propriété P . Du coup, il est facile de faire une machine de Turing qui accepte les codages $\langle M \rangle$ des machines M telles que $L(M)$ vérifie P .

Preuve. Soit le langage $L_P = \{\langle M \rangle \mid L(M) \text{ vérifie } P\}$ des codages des machines de Turing dont le langage vérifie la propriété P . Pour montrer que ce langage est indécidable, trouve une réduction du langage L_{acc} au langage L_P et on conclut encore grâce au corollaire précédent.

Quitte à remplacer la propriété P par sa négation, on peut toujours supposer que le langage vide ne vérifie pas la propriété P . Puisque P n'est pas triviale, il existe une machine de Turing M_0 telle $L(M_0)$ vérifie P .

Pour toute paire (M, w) on considère la machine M_w définie de la manière suivante.

```

input  $u$ 
if  $M$  accepte  $w$  then
  simuler  $M_0$  sur  $u$  et retourner le résultat
else
  rejeter
end if

```

Dans la définition précédente, si la machine M ne termine pas sur l'entrée w la machine M_w ne termine pas non plus. Si M accepte w le langage de la machine M_w est égal au langage $L(M_0)$. Si au contraire la machine M rejette w , la machine M_w rejette tous les mots u et son langage est vide. On en déduit l'équivalence $w \in L(M) \iff L(M_w) \in P$. La fonction f qui associe $\langle M_w \rangle$ à $\langle M, w \rangle$ est bien sûr calculable par une machine de Turing déterministe. Elle est donc une réduction de L_{acc} à L_P . Le corollaire précédent permet de conclure que L_P n'est pas décidable. $\square$

3.4 Problème de correspondance de Post

3.4.1 Présentation

Définition 3.36 (PCP). Le *problème de correspondance de Post* est le suivant.

- Une *instance* est la donnée d'un entier m et de deux suites $u_1, \dots, u_m$ et $v_1, \dots, v_m$ de mots sur un alphabet Σ .
- Une *solution* est une suite d'indices $i_1, i_2, \dots, i_n$ de $\{1, \dots, m\}$ telle que :

$$u_{i_1} u_{i_2} \cdots u_{i_n} = v_{i_1} v_{i_2} \cdots v_{i_n}.$$

- Les instances positives sont celles ayant au moins une solution.

Le problème de correspondance de Post peut être visualisé à l'aide de dominos sur lesquels sont inscrits deux mots. Pour chaque paire de mots (u_i, v_i) , on construit un domino sur lequel les mots u_i et v_i sont écrits l'un au dessus de l'autre. On obtient alors les m dominos suivants.

$$\begin{array}{|c|} \hline u_1 \\ \hline v_1 \\ \hline \end{array}, \begin{array}{|c|} \hline u_2 \\ \hline v_2 \\ \hline \end{array}, \dots, \begin{array}{|c|} \hline u_n \\ \hline v_n \\ \hline \end{array}$$

La question est alors de savoir s'il existe une façon de disposer côte-à-côte des dominos pris dans la liste de telle sorte qu'on lise le même mot en haut et en bas. Comme certains indices de $i_1, \dots, i_n$ peuvent être égaux, il est possible de réutiliser plusieurs fois le même domino de la liste.

Le problème PCP peut aussi être formulé en termes de morphismes. Soit A l'alphabet $\{a_1, \dots, a_m\}$ et soient μ et ν les deux morphismes de A^* dans Σ^* définis respectivement par $\mu(a_i) = u_i$ et $\nu(a_i) = v_i$. L'instance de PCP donnée par les deux suites $u_1, \dots, u_m$ et $v_1, \dots, v_m$ de mots a une solution si et seulement si il existe un mot $w \in A^*$ tel que $\mu(w) = \nu(w)$.

Exemple 3.37. Soit l'instance du problème de Post donnée par $m = 4$ et les mots $u_1 = ab$, $u_2 = c$, $u_3 = ab$, $u_4 = abc$, $v_1 = a$, $v_2 = bcab$, $v_3 = b$ et $v_4 = bc$. Cette instance du problème de Post admet la solution $n = 4$, $i_1 = 1$, $i_2 = 2$,

$i_2 = 1$ et $i_4 = 4$. En effet, on a l'égalité $u_1 u_2 u_1 u_4 = v_1 v_2 v_1 v_4 = abcababc$ qui se visualise sous forme de dominos par

$$\begin{array}{|c|} \hline ab \\ \hline a \\ \hline \end{array} \begin{array}{|c|} \hline c \\ \hline bcab \\ \hline \end{array} \begin{array}{|c|} \hline ab \\ \hline a \\ \hline \end{array} \begin{array}{|c|} \hline abc \\ \hline bc \\ \hline \end{array} = \begin{array}{|c|} \hline abcababc \\ \hline abcababc \\ \hline \end{array}.$$

Exemple 3.38. Soit l'instance du problème de Post donnée par $m = 4$ et les mots $u_1 = ab$, $u_2 = a$, $u_3 = bc$, $u_4 = c$, $v_1 = bc$, $v_2 = ab$, $v_3 = ca$ et $v_4 = a$. Un argument de longueur permet de conclure que cette instance n'a pas de solution. On a en effet $|u_i| \leq |v_i|$ pour tout $1 \leq i \leq m$. Dans ce cas, l'instance a une solution si et seulement si $u_i = v_i$ pour au moins un entier $1 \leq i \leq m$.

Afin de montrer que PCP est indécidable, on introduit une variante très proche de PCP. Elle diffère seulement dans le fait que le premier domino de la solution est imposé. Son intérêt principal est de faciliter la preuve de l'indécidabilité de PCP.

Définition 3.39 (PCPM). Le *problème de correspondance de Post modifié* est le suivant.

- Une *instance* est la donnée d'un entier m et de deux suites $u_1, \dots, u_m$ et $v_1, \dots, v_m$ de mots sur un alphabet Σ .
- Une *solution* est une suite d'indices $i_1, i_2, \dots, i_n$ de $\{1, \dots, m\}$ telle que :

$$i_1 = 1 \quad \text{et} \quad u_1 u_{i_2} \cdots u_{i_n} = v_1 v_{i_2} \cdots v_{i_n}.$$

- Les instances positives sont celles ayant au moins une solution.

3.4.2 Indécidabilité

On va montrer que les deux problèmes PCP et PCPM sont indécidables. On commence par montrer que les deux problèmes sont équivalents.

Lemme 3.40 (Équivalence). *PCP est décidable si et seulement si PCPM est décidable.*

Preuve. On montre d'abord que si PCPM est décidable, alors PCP est aussi décidable. Soit une instance de PCP formée par les deux suites de mots $u_1, \dots, u_m$ et $v_1, \dots, v_m$. Une solution $i_1, \dots, i_n$ de cette instance de PCP est aussi une solution de l'instance de PCPM donnée par les suites $u_{i_1}, u_1, \dots, u_m$ et $v_{i_1}, v_1, \dots, v_m$ où les mots u_{i_1} et v_{i_1} ont été placés en tête. L'instance de PCP a une solution si et seulement si une des m instances de PCPM obtenues en mettant en tête l'un des dominos a une solution. Ceci montre que si PCPM est décidable alors PCP est décidable.

Il faut remarquer qu'on a ramené une instance de PCP à m instances de PCPM. Ce n'est pas une réduction $\leq_m$ telle que cela a été défini (cf. définition 3.36). Il s'agit en fait d'une réduction de Turing qui est notée $\leq_T$.

On montre maintenant que si PCP est décidable, alors PCPM est aussi décidable. On effectue une véritable réduction qui transforme une instance de PCPM en une instance de PCP équivalente.

Soit une instance de PCPM donnée par un entier m et deux suites $u_1, \dots, u_m$ et $v_1, \dots, v_m$ de mots sur un alphabet Σ . On introduit un nouveau symbole $\$$ n'appartenant pas à Σ et on pose $\Sigma' = \Sigma \cup \{\$\}$. On définit les deux morphismes

p et s de Σ^* dans Σ'^* par $p(a) = \$a$ et $s(a) = a\$$ pour toute lettre a de Σ . On vérifie que pour tout mot w de Σ^* , on a l'égalité $p(w)\$ = \$s(w)$.

On définit alors l'instance de PCP donnée par l'entier $m' = 2m + 1$ et les deux suites de mots $u'_1, \dots, u'_{2m+1}$ et $v'_1, \dots, v'_{2m+1}$ définis sur l'alphabet Σ' de la manière suivante.

$$u'_k = \begin{cases} p(u_k) & \text{si } 1 \leq k \leq m \\ p(u_{k-m})\$ & \text{si } m < k \leq 2m \\ p(u_1) & \text{si } k = 2m + 1 \end{cases} \quad \text{et} \quad v'_k = \begin{cases} s(v_k) & \text{si } 1 \leq k \leq m \\ s(v_{k-m}) & \text{si } m < k \leq 2m \\ \$s(v_1) & \text{si } k = 2m + 1 \end{cases}$$

Soit $1, i_2, \dots, i_n$ une solution de l'instance de PCPM, c'est-à-dire une suite telle l'égalité $u_1 u_{i_2} \dots u_{i_n} = v_1 v_{i_2} \dots v_{i_n}$ soit vérifiée. En utilisant la relation $p(w)\$ = \$s(w)$, on obtient l'égalité $p(u_1)p(u_{i_2}) \dots p(u_{i_n})\$ = \$s(v_1)s(v_{i_2}) \dots s(v_{i_n})$ qui se traduit par l'égalité $u'_{2m+1} u'_{i_2} \dots u'_{i_{n-1}} u'_{i_n+m} = v'_{2m+1} v'_{i_2} \dots v'_{i_{n-1}} v'_{i_n+m}$. On constate que la suite $2m+1, i_2, \dots, i_{n-1}, i_n + m$ est une solution de l'instance de PCP.

On vérifie que les seules solutions minimales de cette instance de PCP sont les suites d'indices $i_1, \dots, i_n$ vérifiant les conditions suivantes :

- $i_1 = 2n + 1$,
- $1 \leq i_k \leq m$ pour tout $2 \leq k \leq n - 1$,
- $m + 1 \leq i_n \leq 2m$.

et alors, par construction, $1, i_2, \dots, i_{n-1}, i_n - m$ est à chaque fois une solution de l'instance de PCPM. Les solutions non minimales sont obtenues en prenant des concaténations de solutions minimales. On retrouve ces concaténations par les occurrences du mot $\$ \$$ dans le mot $u'_{i_1} \dots u'_{i_n} = v'_{i_1} \dots v'_{i_n}$. Ceci montre que l'instance de départ de PCPM a une solution si et seulement si l'instance de PCP associée a une solution. La fonction qui calcule l'instance de PCP associée à l'instance de PCPM donnée est calculable par machine de Turing déterministe. On en déduit qu'on a trouvé une réduction de PCPM à PCP. $\square$

Théorème 3.41 (Post 1946). *Les deux problèmes PCP et PCPM sont indécidables.*

Dans le résultat précédent, la taille de l'alphabet sur lequel sont écrits les mots u_i et v_i n'est pas fixée. En fait, le problème PCP reste indécidable même si la taille de cet alphabet est fixée à 2. Par contre PCP est décidable sur un alphabet unaire.

Preuve. Comme PCP et PCPM sont équivalents, il suffit de montrer que PCPM est indécidable. On va exhiber une réduction du langage L_{acc} à PCPM. Soit une instance de L_{acc} formée par une machine de Turing M et une entrée w . D'après la proposition 3.11, on peut supposer que la machine M est normalisée. On peut en outre supposer que si la machine accepte, elle termine avec une bande remplie de symboles blancs $\#$. Pour cela, on ajoute quelques transitions qui se chargent de remplir la bande avec $\#$ une fois que la machine a atteint l'état d'acceptation q_+ . Ceci signifie qu'on peut supposer que la dernière configuration d'un calcul acceptant est la configuration q_+ .

Le mot w est accepté par la machine M s'il existe un calcul $q_0 w = C_0 \rightsquigarrow C_1 \rightsquigarrow \dots \rightsquigarrow C_l = q_+$ de la configuration initiale $q_0 w$ à la configuration acceptante q_+ . On construit une instance de PCPM telle que ses solutions minimales produisent le mot $C_0 \$ C_1 \$ \dots \$ C_l$.

Cette instance est formée par toutes les paires (u_i, v_i) de mots suivantes :

- la première paire $u_1 = \$q_0w\$$ et $v_1 = \$$ pour initier le calcul,
- une paire $u_a = a$ et $v_a = a$ pour tout a de Γ pour recopier cette lettre,
- une paire $u = \$$ et $v = \$$ pour passer à la configuration suivante,
- une paire $u = \$$ et $v = \#\$$ pour passer à la configuration suivante en ajoutant un $\#$ implicite à la fin de la configuration,
- une paire $u_\tau = pa$ et $v_\tau = bq$ pour toute transition $\tau = p, a \rightarrow q, b, R$ de M ,
- une paire $u_\tau = cpa$ et $v_\tau = qcb$ pour toute transition $\tau = p, a \rightarrow q, b, L$ de M et toute lettre $c \in \Gamma$,
- une paire $u = aq_+$ et $v = q_+$ et une autre paire $u = q_+a$ et $v = q_+$ pour toute lettre $a \in \Gamma$ afin de réduire la configuration une fois l'état q_+ atteint,
- une paire $u = q_+\$$ et $v = \varepsilon$ pour terminer.

Sous forme de dominos, on a les paires

$$\begin{array}{ccccccc}
 \boxed{\begin{array}{c} \$ \\ \$q_0w\$ \end{array}} & , & \boxed{\begin{array}{c} a \\ a \end{array}} & , & \boxed{\begin{array}{c} b \\ b \end{array}} & \cdots & , & \boxed{\begin{array}{c} \# \\ \# \end{array}} & , & \boxed{\begin{array}{c} \$ \\ \$ \end{array}} & , & \boxed{\begin{array}{c} \$ \\ \#\$ \end{array}} \\
 \boxed{\begin{array}{c} pa \\ bq \end{array}} & , \cdots , & \boxed{\begin{array}{c} cpa \\ qcb \end{array}} & , \cdots , & \boxed{\begin{array}{c} aq_+ \\ q_+ \end{array}} & , & \boxed{\begin{array}{c} q_+a \\ q_+ \end{array}} & , \cdots , & \boxed{\begin{array}{c} q_+\$ \end{array}} .
 \end{array}$$

On vérifie (laborieusement) que toute solution de cette instance de PCPM provient nécessairement d'un calcul acceptant le mot w . $\square$

3.4.3 Application aux grammaires

On utilise maintenant l'indécidabilité de PCP pour montrer l'indécidabilité de questions très naturelles sur les grammaires. Le résultat est à chaque fois obtenu par une réduction de PCP au problème considéré.

Soit une instance de PCP donnée par deux suites $u_1, \dots, u_m$ et $v_1, \dots, v_m$ de mots sur un alphabet Σ . On introduit un alphabet $A = \{a_1, \dots, a_m\}$ formé de m nouvelles lettres n'appartenant pas à Σ . On définit ensuite les deux langages L_u et L'_u sur l'alphabet $A + \Sigma$ par

$$\begin{aligned}
 L_u &= \{a_{i_1}a_{i_2} \cdots a_{i_n}u_{i_n}u_{i_{n-1}} \cdots u_{i_1} \mid n \geq 0 \text{ et } 1 \leq i_k \leq m\} \\
 L'_u &= \{a_{i_1}a_{i_2} \cdots a_{i_n}w \mid n \geq 0, w \in \Sigma^* \text{ et } w \neq u_{i_n} \cdots u_{i_1}\}.
 \end{aligned}$$

On alors le lemme.

Lemme 3.42. *Les deux langages L_u et L'_u sont algébriques.*

Preuve. Le langage L_u est facilement engendré par la grammaire

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

Pour le langage L'_u , on considère trois cas suivant que w est un suffixe propre de $u_{i_n} \cdots u_{i_1}$, que w est de longueur supérieure à $u_{i_n} \cdots u_{i_1}$ ou que w et $u_{i_n} \cdots u_{i_1}$ diffèrent par une lettre. Le langage L'_u est égal à $L_G(S)$ où la grammaire G

possède les règles suivantes.

$$\begin{aligned}
S &\rightarrow \sum_{i=1}^n a_i S u_i + \sum_{\substack{1 \leq i \leq m \\ |u|=|u_i| \\ u \neq u_i}} a_i R u + \sum_{\substack{1 \leq i \leq m \\ |u| < |u_i|}} a_i T u + \sum_{\substack{1 \leq i \leq m \\ b \in \Sigma}} a_i V b u_i \\
R &\rightarrow \sum_{i=1}^n a_i R + \sum_{b \in \Sigma} R b + \varepsilon \\
T &\rightarrow \sum_{i=1}^n a_i T + \varepsilon \\
V &\rightarrow \sum_{b \in \Sigma} V b + \varepsilon
\end{aligned}$$

□

Lemme 3.43. On a les égalités $(A + \Sigma)^* \setminus L_u = L'_u \cup ((A + \Sigma)^* \setminus A^* \Sigma^*)$

Preuve. Il suffit de remarquer que $L_u \subseteq A^* \Sigma^*$ et $A^* \Sigma^* \setminus L_u = L'_u$. □

Proposition 3.44. 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 grammaire G , est-ce que $L_G(S) = A^*$?
4. pour une grammaire G , est-ce que G est ambiguë ?

Preuve. 1 Le problème de correspondance de Post a une solution si et seulement si l'intersection $L_u \cap L_v$ est non vide. Les grammaires engendrant les langages L_u et L_v peuvent être calculées par une machine de Turing déterministe à partir de l'instance de PCP. On a donc construit une réduction de PCP au problème considéré.

3 L'intersection $L_u \cap L_v$ est vide si et seulement si l'union $(A + \Sigma)^* \setminus L_u \cup (A + \Sigma)^* \setminus L_v$ est égale à $(A + \Sigma)^*$. On a donc une réduction du problème 1 au problème 3. Ceci montre que le problème 3 est aussi indécidable.

2 Comme le problème 3 se réduit au problème 2, ce dernier problème est également indécidable.

4 Le langage $(L_u + L_v) \setminus \{\varepsilon\}$ est égal à $L_G(S)$ où la grammaire G est donnée par les règles suivantes.

$$\begin{aligned}
S &\rightarrow S_1 + S_2 \\
S_1 &\rightarrow \sum_{i=1}^m a_i S_1 u_i + \sum_{i=1}^m a_i u_i \\
S_2 &\rightarrow \sum_{i=1}^m a_i S_2 v_i + \sum_{i=1}^m a_i v_i.
\end{aligned}$$

Cette grammaire est ambiguë si et seulement si l'intersection $L_u \cap L_v$ est non vide. On a donc une réduction du problème 1 au problème 4 qui montre que le problème 4 est encore indécidable.

□

3.5 Théorème de récursion

Le théorème de récursion est un résultat central de la calculabilité. Il intervient dès qu'un modèle de calcul est équivalent aux machines de Turing. Ce théorème est souvent vu comme un paradoxe car il permet de construire des machines qui sont capables de se reproduire. Il est alors un ingrédient essentiel à la fabrication de virus informatiques. Il a aussi des implications en logique mathématique. La proposition suivante est un cas particulier du théorème de récursion. Ce premier résultat plus simple permet de comprendre les mécanismes mis en œuvre.

Proposition 3.45 (Quines). *Il existe des machines de Turing qui écrivent leur propre codage. De telles machines sont appelées Quines.*

La proposition a pour conséquence que dans tout langage de programmation équivalent aux machines de Turing, on peut écrire un programme qui affiche son propre code.

Preuve. On considère des machines de Turing qui calculent des fonctions et qui s'arrêtent sur toute entrée. On suppose que ces machines sont normalisées. Elles possèdent un unique état final f où se termine tout calcul. Pour simplifier la composition, on suppose aussi que les machines s'arrêtent avec la tête de lecture sur la première position de la bande. Pour deux machines M et M' , on note MM' la machine obtenue en composant de manière séquentielle les machines M et M' . Pour une entrée w , la machine MM' donne le résultat du calcul de M' qui a pris pour entrée le résultat du calcul de M sur w .

On commence par construire les machines suivantes.

1. Étant donnés les codages $\langle M \rangle$ et $\langle M' \rangle$ de deux machines de Turing, la machine C calcule le codage $\langle MM' \rangle$ de la machine MM' .
2. Étant donné un mot w , la machine P_w calcule le mot w .
3. Pour une entrée w , la machine R_0 affiche le codage $\langle P_w \rangle$ de la machine qui affiche w .
4. Pour une entrée w de la forme $w = \langle M \rangle$, la machine R calcule le codage $\langle P_w M \rangle$ de la composition de P_w et de M .

On considère alors la machine $Q = P_{\langle R \rangle} R$ qui est la composition séquentielle des machines $P_{\langle R \rangle}$ et R . Après le fonctionnement de la machine $P_{\langle R \rangle}$, la bande contient $\langle R \rangle$. Après le fonctionnement de la machine R , la bande contient $\langle P_{\langle R \rangle} R \rangle$ qui est bien le codage de la machine Q . Ceci est schématisé de la façon suivante.

$$\varepsilon \xrightarrow{P_{\langle R \rangle}} \langle R \rangle \xrightarrow{R} \langle P_{\langle R \rangle} R \rangle$$

□

On illustre la preuve de la proposition en montrant comment celle-ci permet de donner un programme en langage C qui affiche son propre code. La même méthode pourrait être appliquée dans n'importe quel autre langage de programmation.

Le premier problème à résoudre est un problème de notation. Lorsqu'un caractère `'''` ou `'\'` est inclus dans une chaîne il doit être précédé d'un caractère d'échappement `'\'`. Pour une chaîne w sur l'alphabet ASCII, on note

$\hat{w}$ la chaîne obtenue en insérant un caractère `'\'` avant chaque caractère `'\"'` ou `'\'`. Si w est par exemple la chaîne `char *\"'a'\"`, alors la chaîne $\hat{w}$ est `char *\"\\'a'\\\\\"`.

Le second problème est qu'un programme C ne peut pas relire ce qu'il a écrit. La machine P_w est donc transformée en une fonction qui retourne une chaîne de caractères. Pour la chaîne w égale à `char *\"'a'\"`, la fonction P_w s'écrit de la manière suivante en langage C.

```
char *P(){ return "char *\"\\'a'\\\\\"";}
```

La machine R est transformée en une fonction qui affiche le code de P_w . Elle s'écrit de la manière suivante en langage C.

```
void R(char* w) {
    printf("char *P(){ return \";
    while(*w != '\0') {
        if (*w == '\\\' || *w == '\\\'')
            printf("\\");
        printf("%c", *w++);
    }
    printf("\";}");
}
```

Le code de la machine Q est finalement le suivant.

```
void char* P(){ return "...";}
void main(){
    char* w = P();
    printf("char *P(){ return \";
    while(*w != '\0') {
        if (*w == '\\\' || *w == '\\\'')
            printf("\\");
        printf("%c", *w++);
    }
    printf("\";}");
    printf("%s", w);
}
```

où les `...` sont remplacés par $\hat{w}$ où w est le code `void main(){ ... }` de la fonction `main`.

Il est bien sûr possible de faire plus concis comme l'exemple suivant qui contourne le problème du caractère d'échappement en utilisant le code ASCII de la quote `'\"'` qui est 34 en décimal.

```
main(){char *s="main(){char *s=%c%s%c;printf(s,34,s,34);}";
        printf(s,34,s,34);}
```

Le théorème suivant étend la proposition précédente dans la mesure où une machine peut combiner son propre codage avec d'autres données.

Théorème 3.46 (Théorème de récursion). *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. On commence par construire la machine R' , qui pour une entrée de la forme $u\$ \langle M \rangle$ calcule $u\$ \langle P_{\langle M \rangle} M \rangle$. Soit T une machine de Turing qui calcule la fonction t . On considère la machine $M = P_{\langle R'T \rangle} R'T$ qui est la composition séquentielle des machines $P_{\langle R'T \rangle}$, R' et T . On a alors le schéma de fonctionnement suivant.

$$w\$ \xrightarrow{P_{\langle R'T \rangle}} w\$ \langle R'T \rangle \xrightarrow{R'} w\$ \langle P_{\langle R'T \rangle} R'T \rangle \xrightarrow{T} t(w, \langle M \rangle)$$

□

Théorème 3.47 (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) :

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

□

3.6 Machines linéairement bornées

Les machines linéairement bornées sont des machines qui utilisent uniquement la portion de la bande où est écrite l'entrée. Ces machines jouent un rôle important pour plusieurs raisons. D'une part, elles caractérisent les langages engendrés par les grammaires contextuelles qui généralisent les grammaires pour les langages algébriques (cf. définition 2.1 p. 61). D'autre part, elles se situent à la frontière de l'indécidabilité puisque le problème de l'acceptation est décidable alors que le problème du vide est indécidable.

3.6.1 Définition

Une machine de Turing M est dite *linéairement bornée* si elle n'écrit pas en dehors de l'espace utilisé par la donnée.

Comme on peut toujours supposer qu'une machine de Turing n'écrit pas le symbole blanc $\#$ (en le dupliquant éventuellement en un autre symbole $\#'$), on peut aussi dire qu'une machine linéairement bornée est une machine telle que si $p, \# \rightarrow q, b, x$ est une transition alors, $b = \#$ et $x = L$. Ceci empêche la machine de modifier les symboles blancs présents sur la bande.

L'espace utilisé par une machine linéairement bornée peut être augmenté en ajoutant de nouveaux symboles à l'alphabet de bande mais l'espace ainsi disponible reste proportionnel à la taille de l'entrée. Ceci justifie la terminologie.

3.6.2 Grammaires contextuelles

L'objectif de cette partie est de définir les grammaires contextuelles et de montrer qu'elles ont même pouvoir d'expression que les machines linéairement bornées. Ceci signifie qu'un langage est accepté par une machine linéairement bornée si et seulement si il est engendré par une grammaire contextuelle.

Définition 3.48. Une *grammaire contextuelle* est un quadruplet (A, V, P, S) où A est un alphabet fini, V un ensemble fini de variables, $S \in V$ est l'axiome et P est un ensemble de règles de la forme suivante.

- soit la règle $S \rightarrow 1$ qui permet d'engendrer le mot vide à partir de l'axiome,
- soit une règle $uTv \rightarrow uvv$ où $T \in V$ est une variable, $u, v \in (A + V)^*$ sont des mots formés de lettres et de variables et $w \in (A + V \setminus \{S\})^+$ est un mot non vide formé de lettres et de variables autres que l'axiome.

L'élément important de cette définition est que seul l'axiome peut engendrer le mot vide. Hormis la règle $S \rightarrow 1$, le membre droit d'une règle est toujours de longueur supérieure ou égale au membre gauche.

Définition 3.49. Une grammaire contextuelle (A, V, P) est *croissante* si chaque règle $u \rightarrow v$ vérifie $|u| \leq |v|$.

La proposition suivante montre que grammaires contextuelles et grammaires croissantes sont équivalentes pour les langages ne contenant pas le mot vide.

Proposition 3.50. *Un langage $L \subset A^+$ est engendré par une grammaire contextuelle si et seulement si il est engendré par une grammaire contextuelle croissante.*

Une grammaire contextuelle qui ne contient pas la règle $S \rightarrow 1$ est une grammaire croissante. Si L est engendré par une grammaire contextuelle, celle-ci ne contient pas la règle $S \rightarrow 1$.

La réciproque est un peu plus délicate. Elle consiste à transformer une grammaire croissante en une grammaire contextuelle équivalente. Chaque règle de la grammaire croissante est remplacée par plusieurs règles dans la grammaire croissante construite. Quitte à introduire une nouvelle variable pour chaque lettre de A , on peut supposer que les règles de la grammaire croissante sont des deux formes suivantes.

- soit de la forme $S \rightarrow a$ pour une variable $S \in V$ et une lettre $a \in A$,
- soit de la forme $S_{i_1} \cdots S_{i_k} \rightarrow S_{j_1} \cdots S_{j_m}$ avec $m \geq n \geq 1$.

Les règles de la forme $S \rightarrow a$ sont laissées telles quelles. Chaque règle $S_{i_1} \cdots S_{i_k} \rightarrow S_{j_1} \cdots S_{j_m}$ est remplacée par les règles suivantes.

- $S_{i_1} \cdots S_{i_k} \rightarrow Z_1 S_{i_2} \cdots S_{i_k} \rightarrow Z_1 Z_2 S_{i_3} \cdots S_{i_k} \rightarrow Z_1 Z_2 Z_3 S_{i_4} \cdots S_{i_k} \rightarrow \cdots \rightarrow Z_1 \cdots Z_{k-1} S_{i_k} \rightarrow Z_1 \cdots Z_{k-1} S_{j_k} \cdots S_{j_m}$
- $Z_1 \cdots Z_{k-1} S_{j_k} \cdots S_{j_m} \rightarrow Z_1 \cdots Z_{k-2} S_{j_{k-1}} \cdots S_{j_m} \rightarrow Z_1 \cdots Z_{k-3} S_{j_{k-2}} \cdots S_{j_m} \rightarrow \cdots \rightarrow Z_1 S_{j_2} \cdots S_{j_m} \rightarrow S_{j_1} \cdots S_{j_m}$

où $Z_1, Z_2, \dots, Z_{k-1}$ sont $k-1$ nouvelles variables propres à la règle transformée. Il est facile de vérifier que les nouvelles règles simulent l'ancienne règle.

Les grammaires croissantes permettent de démontrer le résultat suivant.

Théorème 3.51. *Un langage est engendré par une grammaire contextuelle si et seulement s'il est accepté par une machine linéairement bornée.*

Étant donné une grammaire quelconque, on peut toujours faire une machine de Turing non déterministe qui devine la dérivation à l'envers. La machine accepte lorsqu'elle trouve l'axiome. Si la grammaire est contextuelle, l'espace mémoire utilisé diminue au cours du calcul. On obtient donc une machine linéairement bornée.

Réciproquement soit une machine linéairement bornée $M = (Q, \Sigma, \Gamma, E, q_0, F, \#)$. Sans perte de généralité, on peut supposer que la machine M a un seul état final q_f , c'est-à-dire que $F = \{q_f\}$. On construit alors la grammaire contextuelle (A, V, P, S) suivante. L'alphabet A est l'alphabet d'entrée Σ de M . L'ensemble V de variables est $\{S\} \cup \Gamma \times \Sigma \cup Q \times \Gamma \times \Sigma$ qui contient l'axiome S , les paires (a, b) formées d'une lettre $a \in \Gamma$ de l'alphabet de bande et d'une lettre $b \in \Sigma$ de l'alphabet d'entrée ainsi que les triplets (q, a, b) formés d'un état $q \in Q$, d'une lettre $a \in \Gamma$ de l'alphabet de bande et d'une lettre $b \in \Sigma$ de l'alphabet d'entrée. L'ensemble P contient les règles des trois types suivants.

- Règles d'initialisation
 - une règle $S \rightarrow S(a, a)$ pour chaque $a \in A$,
 - une règle $S \rightarrow (q_0, a, a)$ pour chaque $a \in A$,
- Règles de simulation
 - une règle $(q, a, c)(d, e) \rightarrow (b, c)(p, d, e)$ pour chaque transition $q, a \rightarrow p, b, R$ de M et chaque $c, e \in \Sigma$ et $d \in \Gamma$,
 - une règle $(d, e)(q, a, c) \rightarrow (p, d, e)(b, c)$ pour chaque transition $q, a \rightarrow p, b, L$ de M et chaque $c, e \in \Sigma$ et $d \in \Gamma$,
- Règles de terminaison
 - une règle $(q_f, b, a)(d, e) \rightarrow a(q_f, d, e)$ pour chaque $a, e \in \Sigma$ et $b, d \in \Gamma$,
 - une règle $(q_f, b, a) \rightarrow a$ pour chaque $a \in \Sigma$ et $b \in \Gamma$,

3.6.3 Décidabilité

Les machines linéairement bornées se situent à la frontière entre la décidabilité et l'indécidabilité. Certains problèmes comme celui de l'acceptation deviennent décidables pour les machines linéairement bornées alors que d'autres comme celui du vide restent indécidables.

Théorème 3.52. *Il peut être décidé si une machine linéairement bornée accepte un mot w .*

Pour une entrée de longueur n , le nombre de configurations possibles de la machine est $n|Q||\Gamma|^n$ où $|Q|$ est le nombre d'états et $|\Gamma|$ le nombre de symboles de bande. On peut construire effectivement le graphe des calculs de la machine sur l'entrée w et décider s'il y a, dans ce graphe, un chemin de la configuration initiale q_0w à une configuration acceptante.

Théorème 3.53. *Il est indécidable de savoir si une machine M linéairement bornée n'accepte aucun mot, c'est-à-dire $L(M) = \emptyset$.*

Preuve. Soit M une machine de Turing quelconque et soit w un mot d'entrée de M . Soit $\$$ un nouveau symbole qui n'appartient pas à $Q \cup \Gamma$. On définit le langage X_w de la manière suivante.

$$X_w = \{C_0\$C_1\$ \cdots \$C_k \mid q_0w = C_0 \rightsquigarrow \cdots \rightsquigarrow C_k \text{ est un calcul acceptant } w\}.$$

Le langage X_w est accepté par une machine linéairement bornée M' qui peut être effectivement construite à partir de la machine M et de w . La machine M' doit vérifier que $C_0 = q_0w$, que $C_i \rightsquigarrow C_{i+1}$ pour $0 \leq i < k$ et que C_k est acceptante. On a alors les équivalences

$$w \in L(M) \iff X_w \neq \emptyset \iff L(M') \neq \emptyset$$

qui montrent que le problème de l'acceptation pour les machines de Turing se réduit au problème du vide pour les machines linéairement bornées. Le problème du vide pour les machines linéairement bornées est donc indécidable. $\square$

3.6.4 Complémentation

Nous montrons dans cette partie que la famille des langages acceptés par des machines linéairement bornées est fermée par complémentation.

Théorème 3.54 (Immerman). *Pour toute machine M linéairement bornée et non-déterministe, il existe une machine M' linéairement bornée et non déterministe qui accepte le complémentaire du langage accepté par M .*

Soit L un langage accepté par une machine linéairement bornée M . Pour une entrée $w \in \Sigma^*$ de M , on note $N(w)$ le nombre de configurations de M qui peuvent être atteintes par M à partir de l'entrée w (c'est-à-dire de la configuration initiale q_0w). On a l'inégalité $N(w) \leq K^{|w|}$ pour une constante K bien choisie. Le nombre $N(w)$ peut donc être stocké sur une bande de longueur $|w|$.

On construit une machine M' linéairement bornée et non déterministe qui accepte le complémentaire Σ^*L du langage L . Pour une entrée w , le calcul de M' comprend les deux étapes principales suivantes.

1. calcul de $N(w)$
2. déterminer si $w \in L$ en utilisant $N(w)$

On commence par décrire l'étape 2 puis on décrit l'étape 1.

Étape 2

Pour l'entrée w , on note $C(w)$, l'ensemble des configurations contenues dans l'espace $|w|$.

Étape 1

On note $N_i(w)$ le nombre de configurations qui peuvent être atteinte en au plus i étapes de calcul de M . On a $N_0(w) = 1$ et $N(w) = N_i(w)$ où i est le plus petit entier tel que $N_i(w) = N_{i+1}(w)$. Pour calculer $N(w)$, il suffit donc de savoir calculer $N_{i+1}(w)$ à partir de $N_i(w)$. La procédure suivante calcule $N_{i+1}(w)$ à partir de $N_i(w)$.

3.7 Décidabilité de théories logiques

3.7.1 Modèles et formules logiques

Une logique est d'abord déterminée par l'ensemble sous-jacent appelé *modèle* qui contient les éléments sur lequel on travaille. Elle est ensuite fixée par les opérateurs logiques utilisés dans les formules ainsi que les formules atomiques autorisées sur les éléments. La manière dont sont utilisés les quantificateurs est aussi importante. Dans certains cas, on s'autorise uniquement à quantifier sur les éléments du modèle. On parle alors de logique du *premier ordre*. Dans d'autres cas, les quantificateurs peuvent porter sur des ensembles ou des relations. On parle alors de logique du *second ordre*. La logique du second ordre est qualifiée

de *monadique* lorsque les quantifications portent seulement sur les ensembles mais pas sur les relations ou les fonctions.

Dans toute cette partie, le modèle est toujours l'ensemble $\mathbb{N}$ des entiers. Les opérations atomiques sur les entiers que l'on considère sont l'addition, la multiplication et l'ordre.

- modèle : $\mathbb{N}$.
- connecteurs booléens : $\wedge, \vee, \neg$
- quantificateur : $\forall, \exists$
- opérations : $+, \times, <$

On va s'intéresser essentiellement à la logique du premier ordre des entiers avec l'addition seulement ou l'addition et la multiplication ainsi qu'à la logique du second ordre des entiers avec l'ordre.

On rappelle auparavant quelques définitions élémentaires de logique.

Définition 3.55 (Formules closes). Une *formule close* est une formule sans variable libre, c'est-à-dire une formule où toute variable est sous la portée d'un quantificateur.

Définition 3.56. On dit qu'une *théorie logique* est *décidable* s'il est décidable de savoir si une formule close est vraie.

Définition 3.57 (Forme prénexe). La formule φ est en *forme prénexe* si elle a la forme $\varphi = Q_1x_1Q_2x_2\ldots Q_nx_n\psi$ où $Q_i = \forall$ ou $Q_i = \exists$ pour $1 \leq i \leq n$ et où la formule ψ est sans quantificateur.

3.7.2 Arithmétique de Presburger

L'arithmétique de Presburger est la théorie du premier ordre des entiers munis de l'addition mais pas de la multiplication.

Théorème 3.58 (Presburger 29). *La théorie au premier ordre des entiers munis de l'addition est décidable.*

Preuve. On montre le résultat en utilisant des automates. On montre en effet que pour toute formule φ de la logique de Presburger, l'ensemble des n -uplets qui satisfont la formule φ est un langage rationnel. La preuve se fait par récurrence sur le nombre de quantificateurs de φ .

Soit φ une formule close qu'on suppose sous forme prénexe. Elle s'écrit donc $\varphi = Q_1x_1Q_2x_2\ldots Q_nx_n\psi$. Pour tout entier $1 \leq k \leq n$, on définit φ_k par $\varphi_k = Q_{k+1}x_{k+1}\ldots Q_nx_n\psi$ avec les conventions $\varphi_0 = \varphi$ et $\varphi_n = \psi$. Pour $0 \leq k \leq n$, la formule φ_k a k variables libres et elle s'écrit donc $\varphi_k(x_1, x_2, \ldots, x_k)$.

On montre par récurrence sur $n - k$ que l'ensemble des k -uplets qui satisfont φ_k est un ensemble rationnel. On définit d'abord un codage qui permet d'écrire les k -uplets d'entiers. Chaque entier est écrit en binaire sur l'alphabet $\{0, 1\}$. Un k -uplet d'entiers est écrit sur l'alphabet $\Sigma_k = \{0, 1\}^k$ en ajoutant éventuellement des zéros en tête des écritures pour les rendre de la même longueur. On définit l'ensemble X_k par

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

On construit par récurrence sur $n - k$, un automate $\mathcal{A}_k$ qui accepte les écritures sur Σ_k des éléments de X_k . Ainsi, l'automate $\mathcal{A}_0$ (défini sur un alphabet unaire) accepte au moins un mot si et seulement si la formule φ est vraie.

On commence par la construction de l'automate $\mathcal{A}_n$ qui accepte les n -uplets qui satisfont la formule ψ . Celle-ci se décompose comme une combinaison booléenne de formules atomiques de la forme $x_i = x_j$ ou $x_i + x_j = x_k$. Comme la classe des langages rationnels est close pour toutes les opérations booléennes, il suffit de construire un automate pour chacune des formules atomiques.

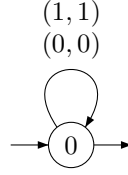


FIG. 3.9 – Automate de égalité

L'automate pour l'égalité s'obtient très facilement (cf.figure 3.9). Pour l'ad-

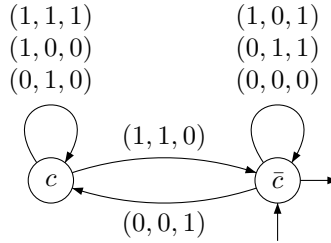


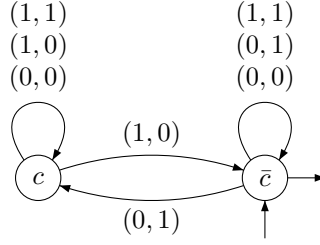
FIG. 3.10 – Automate de l'addition

dition, on construit un automate codéterministe en faisant comme s'il lisait de la droite vers la gauche. Sur la figure 3.10, l'état c est l'état avec retenue et l'état $\bar{c}$ est l'état sans retenue.

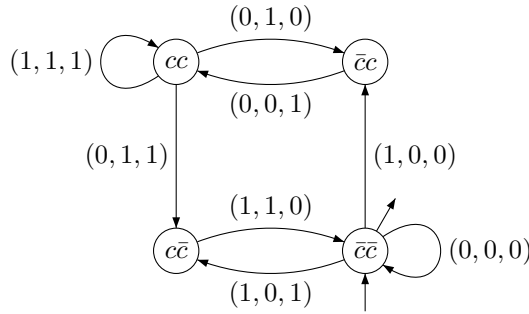
Pour le passage de $\mathcal{A}_{k+1}$ à $\mathcal{A}_k$, on écrit $\varphi_k = Q_{k+1}x_{k+1}\varphi_{k+1}$ et on considère deux cas suivant que Q_{k+1} est le quantificateur $\forall$ ou le quantificateur $\exists$. Dans le premier cas, on se ramène au second cas en écrivant que $\varphi_k = \neg\exists x_{k+1}\neg\varphi_{k+1}$ et en utilisant la clôture par complémentation des langages rationnels. Il ne reste plus qu'à traiter le cas où Q_{k+1} est le quantificateur existentiel.

L'automate $\mathcal{A}_k$ est obtenu à partir de l'automate $\mathcal{A}_{k+1}$ en oubliant la dernière composante de l'alphabet, c'est-à-dire en projetant l'alphabet Σ^{k+1} sur l'alphabet Σ^k par le morphisme π_k défini par $\pi_k(b_1, \dots, b_k, b_{k+1}) = (b_1, \dots, b_k)$. L'automate $\mathcal{A}_k$ devine en quelque sorte l'entier x_{k+1} . L'automate $\mathcal{A}_k$ a même ensemble d'états que l'automate $\mathcal{A}_{k+1}$. Les transitions de $\mathcal{A}_k$ sont toutes les transitions de la forme $p \xrightarrow{\pi_k(x)} q$ où $p \xrightarrow{x} q$ est une transition de $\mathcal{A}_{k+1}$. Les états finaux de $\mathcal{A}_k$ sont ceux de $\mathcal{A}_{k+1}$. Les états initiaux de $\mathcal{A}_k$ sont les états qui peuvent être atteints en partant d'un état initial de $\mathcal{A}_{k+1}$ en lisant des lettres égales à la lettre $(0, \dots, 0)$ de Σ^k . Ce changement permet de prendre en compte les entiers x_{k+1} ayant une écriture binaire plus longue que celles des entiers $x_1, \dots, x_k$. □

La construction de l'automate $\mathcal{A}_k$ à partir de l'automate $\mathcal{A}_{k+1}$ est illustrée par l'exemple suivant.

FIG. 3.11 – Automate de la relation $x \leq y$

Exemple 3.59. La relation $x \leq y$ est équivalente à la formule $\exists z \ x + z = y$. On obtient donc un automate pour la formule $x \leq y$ en partant de l'automate de la figure 3.10 et en projetant l'alphabet Σ^3 sur Σ^2 par $\pi(b_1, b_2, b_3) = (b_1, b_3)$ (cf. figure 3.11).

FIG. 3.12 – Automate de la relation $x = y + z \wedge z = y + y$

Exemple 3.60. L'équation $x \equiv 0 \pmod 3$ est équivalente à la formule $\exists y, z \ x = y + z \wedge z = y + y$. En combinant les automates des relations $x = y + z$ et $z = y + y$, on obtient l'automate de la figure 3.12. En projetant l'alphabet sur sa première composante, on obtient un automate non déterministe qui accepte les écritures binaires des multiples de 3. En déterminisant cet automate, on trouve l'automate de la figure 3.13.

3.7.3 Addition et multiplication

Théorème 3.61 (Tarski). *La théorie au premier ordre des entiers munis de l'addition et de la multiplication 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. $\square$

3.7.4 Logique monadique du second ordre

Pour les liens entre les automates et la logique, [Str94] et [Lib04] sont des très bonnes références.

Théorème 3.62 (Büchi 60). *La logique monadique du second ordre de $\langle \mathbb{N}, < \rangle$ est décidable.*

Théorème 3.63 (McNaughton-Papert). *Un langage est définissable en logique du premier ordre si et seulement si il est sans étoile.*

3.8 Fonctions récursives

Les fonctions récursives constituent une autre façon de définir la notion de *calculabilité*. Elles sont définies à partir d'un nombre restreint de fonctions élémentaires en utilisant la composition, un schéma de récurrence et un schéma de minimisation.

Les fonctions récursives sont habituellement définies comme des fonctions qui manipulent des entiers mais il est également possible de considérer des fonctions de mots comme en [Aut92, p. 68]. Chaque fonction récursive est une fonction d'un sous-ensemble de $\mathbb{N}^k$ dans $\mathbb{N}^r$ pour des entiers k et r . On commence par définir la sous-famille des fonctions primitives récursives qui sont des fonctions totales.

3.8.1 Fonctions primitives récursives

Notations

Comme les fonctions récursives manipulent des k -uplets d'entiers, on adopte quelques conventions pour simplifier les notations. Un k -uplet d'entiers est noté $(n_1, \dots, n_k)$. Si m et n sont les deux k et l -uplets $(m_1, \dots, m_k)$ et $(n_1, \dots, n_l)$, le $k+l$ -uplet $(m_1, \dots, m_k, n_1, \dots, n_l)$ obtenu par concaténation est simplement noté m, n ou encore (m, n) . En particulier, si f est une fonction dont le domaine est inclus dans $\mathbb{N}^{k+l}$, on écrit $f(m, n)$ pour $f(m_1, \dots, m_k, n_1, \dots, n_l)$. Cette notation est étendue aux suites de k -uplets. Soit p un entier et soient p entiers $k_1, \dots, k_p$. Soit k l'entier égal à la somme $k_1 + \dots + k_p$. Si chaque n_i est un k_i -uplet $(n_{i,1}, \dots, n_{i,k_i})$, on note $n_1, \dots, n_p$ ou $(n_1, \dots, n_p)$ le k -uplet $(n_{1,1}, \dots, n_{p,k_p})$ obtenu en concaténant les k_i -uplets. Si f est une fonction dont le domaine est inclus dans $\mathbb{N}^k$, on écrit $f(n_1, \dots, n_p)$ pour $f(n_{1,1}, \dots, n_{p,k_p})$.

Définitions

On introduit maintenant un certain nombre de fonctions primitives récursives de base.

- On appelle *identité* et on note id la fonction qui associe à tout k -uplet $(n_1, \dots, n_p)$ le k -uplet $(n_1, \dots, n_p)$.
- Pour tous entiers $k \geq 0$ et $i \geq 0$, on appelle *fonction constante* la fonction qui à tout k -uplet associe l'entier fixe i . Cette fonction est directement notée par l'entier i .
- Pour des entiers k et i tels que $0 \leq i \leq k$, la i -ème *projection* $p_{k,i}$ est définie par $p_{k,i}(n_1, \dots, n_k) = n_i$. Lorsqu'il n'y a pas d'ambiguïté sur la valeur de k , cette fonction est simplement notée p_i .
- Pour tout entiers $r \geq 0$, la *fonction de duplication* d_r est définie par $d_r(n) = (n, \dots, n)$ où l'entier n est répété r fois.
- La *fonction successeur* s est définie par $s(n) = n + 1$.

On définit maintenant les schémas de composition et de récurrence.

Soit p un entier et soient $p + 2$ entiers $k, r, k_1, \dots, k_p$. Soient p fonctions $f_1, \dots, f_p$ où chaque fonction f_i pour $1 \leq i \leq p$ est une fonction de $\mathbb{N}^{k_i}$ dans $\mathbb{N}^{k_i}$ et soit g une fonction de $\mathbb{N}^{k_1 + \dots + k_p}$ dans $\mathbb{N}^r$, alors la *composée* $g(f_1, \dots, f_p)$ est la fonction de $\mathbb{N}^k$ dans $\mathbb{N}^r$ qui à chaque k -uplet $n = (n_1, \dots, n_k)$ associe le r -uplet $g(f_1(n), \dots, f_p(n))$.

Soit k et r deux entiers. Soit f une fonction de $\mathbb{N}^k$ dans $\mathbb{N}^r$ et soit g une fonction de $\mathbb{N}^{k+r+1}$ dans $\mathbb{N}^r$. La fonction $h = \text{Rec}(f, g)$ est la fonction de $\mathbb{N}^{k+1}$ dans $\mathbb{N}^r$ définie récursivement de la manière suivante pour tout entier n et pour tout k -uplet $m = (m_1, \dots, m_k)$.

$$\begin{aligned} h(0, m) &= f(m), \\ h(n + 1, m) &= g(n, h(n, m), m) \end{aligned}$$

La famille des *fonctions primitives récursives* est la plus petite famille de fonctions qui vérifie les conditions suivantes.

1. Les fonctions constantes, les projections, les fonctions de duplication et la fonction successeur sont primitives récursives.
2. La famille des fonctions primitives récursives est close pour les schémas de composition et de récurrence.

Exemples

On donne ici les exemples classiques de fonctions primitives récursives. Ces fonctions seront utilisées dans la preuve de l'équivalence avec les machines de Turing.

Somme, prédécesseur et différence. La fonction *sum* définie par $\text{sum}(n, m) = n + m$ est primitive récursive. Elle peut en effet être définie de la manière suivante.

$$\begin{aligned} \text{sum}(0, m) &= m \\ \text{sum}(n + 1, m) &= s(\text{sum}(n, m)) \end{aligned}$$

La fonction somme est donc égale à $\text{Rec}(p_1, s(p_2))$.

La fonction *prédécesseur* p définie par $p(n) = \max(0, n - 1)$ est primitive récursive. Elle peut en effet être définie de la manière suivante.

$$\begin{aligned} p(0) &= 0 \\ p(n + 1) &= n \end{aligned}$$

La fonction prédécesseur a pu être définie car le schéma de récurrence permet à la fonction g qui calcule $h(n + 1, m)$ d'utiliser la valeur de n . Sinon la fonction prédécesseur est difficile à définir. Une autre possibilité est de la mettre dans les fonctions de base à côté de la fonction successeur.

La fonction *sub* définie par $\text{sub}(n, m) = n - m$ si $n \geq m$ et 0 sinon est aussi primitive récursive. Elle peut en effet être définie de la manière suivante.

$$\begin{aligned} \text{sub}(n, 0) &= n \\ \text{sub}(n, m + 1) &= p(\text{sub}(n, m)) \end{aligned}$$

Comme la récurrence porte sur le second argument, la fonction *sub* est égale à $\text{sub}'(p_2, p_1)$ où la fonction sub' est égale à $\text{Rec}(p_1, p(p_2))$.

Produit. La fonction *prod* définie par $prod(n, m) = nm$ est primitive réursive. Elle peut en effet être définie de la manière suivante.

$$\begin{aligned} prod(0, m) &= 0 \\ prod(n + 1, m) &= sum(prod(n, m), m) \end{aligned}$$

La fonction somme est donc égale à $Rec(0, sum(p_2, p_3))$.

Égalité. La fonction *eq0* définie par $eq0(m) = 1$ si $m = 0$ et $eq0(m) = 0$ sinon est primitive réursive. Elle est égale à $Rec(1, 0)$. La fonction *eq* définie par $eq(m, n) = 1$ si $m = n$ et $eq(m, n) = 0$ sinon est primitive réursive. Elle est égale $eq0(sum(sub(p_1, p_2), sub(p_2, p_1)))$.

Division et reste. Les fonctions *div* et *mod* où $div(n, m)$ et $mod(n, m)$ sont respectivement le quotient et le reste de la division entière de n par m sont primitives réursives. La fonction *div* peut être définie de la manière suivante.

$$\begin{aligned} div(0, m) &= 0, \\ div(n + 1, m) &= div(n, m) + eq(m(1 + div(n, m)), n + 1) \end{aligned}$$

La fonction *mod* peut être alors définie par $mod(n, m) = n - m \cdot div(n, m)$.

Puissance, racine et logarithme. La fonction *pow* où $pow(n, m) = m^n$ est primitive réursive. Elle peut être définie de la manière suivante.

$$\begin{aligned} pow(0, m) &= 1, \\ pow(n + 1, m) &= prod(pow(n, m), m) \end{aligned}$$

La fonction *root* où $root(n, m)$ est la racine m -ième de n , c'est-à-dire le plus grand entier k tel que $k^m \leq n$ est primitive réursive. Elle peut être définie de la manière suivante.

$$\begin{aligned} root(0, m) &= 0, \\ root(n + 1, m) &= root(n, m) + eq(m, pow(1 + root(n, m), m), n + 1) \end{aligned}$$

La fonction *log* où $log(n, m)$ est le logarithme en base m de n , c'est-à-dire le plus grand entier k tel que $m^k \leq n$ est primitive réursive. Elle peut être définie de la manière suivante.

$$\begin{aligned} log(0, m) &= 0, \\ log(n + 1, m) &= log(n, m) + eq(pow(1 + log(n, m), m), n + 1) \end{aligned}$$

Nombres premiers. On commence par définir la fonction *ndiv*(n) qui donne le nombre de diviseurs de l'entier n . Cette fonction est définie par $ndiv(n) = pndiv(n, n)$ où la fonction *pndiv*(p, n) qui donne le nombre de diviseurs de n inférieurs à p est définie de la manière suivante.

$$\begin{aligned} pndiv(0, n) &= 0, \\ pndiv(p + 1, n) &= pndiv(p, n) + eq0(mod(n, p + 1)). \end{aligned}$$

La fonction premier est alors définie par $premier(n) = eq(ndiv(n), 2)$.

Valuation. Montrer que la fonction *val* où $val(n, m)$ est le plus grand entier k tel que m^k divise n est primitive récursive.

Fonction d'Ackermann

On définit la fonction d'Ackermann A de $\mathbb{N}^3$ dans $\mathbb{N}$ par

$$\begin{aligned} A(k, m, n) &= n + 1 && \text{si } k = 0, \\ A(k, m, n) &= m && \text{si } k = 1 \text{ et } n = 0, \\ A(k, m, n) &= 0 && \text{si } k = 2 \text{ et } n = 0, \\ A(k, m, n) &= 1 && \text{si } k \neq 1, 2 \text{ et } n = 0, \\ A(k, m, n) &= A(k - 1, m, A(k, m, n - 1)) && \text{sinon.} \end{aligned}$$

La proposition suivante donne des formules explicites de $A(k, m, n)$ pour les premières valeurs de k .

Proposition 3.64. *Pour tous m et n , on a les égalités*

$$\begin{aligned} A(1, m, n) &= m + n, \\ A(2, m, n) &= mn = \underbrace{m + \cdots + m}_n, \\ A(3, m, n) &= m^n = \underbrace{m \times \cdots \times m}_n, \\ A(4, m, n) &= m^{\underbrace{m^{\cdots m}}_n}. \end{aligned}$$

Une version simplifiée de la fonction d'Ackermann peut être obtenue en prenant $m = 2$ dans la définition ci-dessus. On obtient alors une fonction de $\mathbb{N}^2$ dans $\mathbb{N}$. On prend aussi souvent comme définition la définition suivante qui est légèrement différente.

$$\begin{aligned} A(0, n) &= n + 1, \\ A(k + 1, 0) &= A(k, 1), \\ A(k + 1, n + 1) &= A(k, A(k + 1, n)) \quad \text{pour } n \geq 0 \end{aligned}$$

On dit qu'une fonction g de $\mathbb{N}$ dans $\mathbb{N}$ *major*e une fonction f de $\mathbb{N}^k$ dans $\mathbb{N}$ si $f(n_1, \dots, n_k) \leq g(\max(n_1, \dots, n_k))$ pour tout k -uplet $(n_1, \dots, n_k)$ d'entiers.

Proposition 3.65. *Pour toute fonction primitive récursive f de $\mathbb{N}^k$ dans $\mathbb{N}$, il existe un entier k tel que f est majorée par la fonction $g(n) = A(k, n)$.*

Corollaire 3.66. *La fonction d'Ackermann n'est pas primitive récursive.*

Le corollaire précédent donne explicitement une fonction qui n'est pas primitive récursive. Un argument diagonal fournit aussi une fonction qui n'est pas primitive récursive. Soit $(f_n)_{n \geq 0}$ une énumération de toutes les fonctions primitives récursives de $\mathbb{N}$ dans $\mathbb{N}$. La fonction g définie par $g(n) = f_n(n) + 1$ n'est pas primitive récursive.

3.8.2 Fonctions récursives

Pour définir les fonctions récursives, on introduit un schéma de minimisation. Soit k et r deux entiers et soit f une fonction de $\mathbb{N}^{k+1}$ dans $\mathbb{N}^r$ éventuellement partielle. On définit alors la fonction $g = \text{Min}(f)$ de $\mathbb{N}^k$ dans $\mathbb{N}$ de la manière suivante. Pour m dans $\mathbb{N}^k$, $g(m)$ est le plus petit entier n , s'il existe, tel que $f(n, m) = d_r(0)$ (on rappelle que $d_r(0)$ est $(0, \dots, 0)$ où 0 est répété r fois). Si un tel entier n n'existe pas, $g(m)$ n'est pas défini. Même si la fonction f est totale, la fonction $g = \text{Min}(f)$ peut être partielle.

La famille des *fonctions récursives* est la plus petite famille de fonctions qui vérifie les conditions suivantes.

1. Les fonctions primitives récursives sont récursives.
2. La famille des fonctions récursives est close pour les schémas de composition, de récurrence et de minimisation.

L'argument diagonal permet encore d'exhiber une fonction qui n'est récursive. Soit $(f_n)_{n \geq 0}$ une énumération de toutes les fonctions récursives de $\mathbb{N}$ dans $\mathbb{N}$. La fonction g définie par $g(n) = f_n(n) + 1$ si $f_n(n)$ est défini et $g(n) = 0$ sinon n'est pas récursive.

3.8.3 Équivalence avec les machines de Turing

Nous allons maintenant montrer le théorème suivant qui établit que les notions de calculabilité définies par les fonctions récursives et les machines de Turing sont identiques.

Théorème 3.67. *Une fonction de $\mathbb{N}^k$ dans $\mathbb{N}^r$ est récursive si et seulement si elle est calculable par une machine de Turing.*

Pour les machines de Turing, il faut préciser comment sont codés les entiers. Ils peuvent être codés en binaire, en base 10 ou en base 1. Le choix du codage est en réalité sans importance puisqu'il est possible de passer d'un quelconque de ces codages à n'importe quel autre avec une machine de Turing.

Il est facile de se convaincre qu'une fonction récursive est calculable par une machine de Turing. Les fonctions de base comme les fonctions constantes, les projections, les fonctions de duplication sont bien sûr calculables par machine de Turing. Ensuite avec des machines calculant des fonctions $f, g, f_1, \dots, f_p$, il n'est pas difficile de construire des machines de Turing pour les fonctions $g(f_1, \dots, f_p)$, $\text{Rec}(f, g)$ et $\text{Min}(f)$.

Nous allons maintenant montrer qu'une fonction calculable par une machine de Turing est récursive. Soit M une machine de Turing qu'on suppose déterministe et avec un seul état d'acceptation q_+ . On suppose que la machine ne se bloque jamais. Soit n le nombre d'états de M et soit m le cardinal de son alphabet de bande. On suppose que les états de M sont les entiers $\{0, 1, \dots, n-1\}$ et que les symboles de bande sont les entiers $\{0, 1, \dots, m-1\}$, le symbole blanc $\#$ étant l'entier 0. Pour tout mot $w = a_0 \cdots a_k$ sur l'alphabet de bande, on associe les deux entiers $\alpha(w) = \sum_{i=0}^k a_i m^{k-i}$ et $\beta(w) = \sum_{i=0}^k a_i m^i$ obtenus en lisant w comme un nombre écrit en base m (avec les unités respectivement à droite et à gauche pour α et β). Les fonctions α et β permettent de considérer les mots de la bande comme des entiers.

Soit w une entrée de M et soit $C_0 \rightsquigarrow \dots \rightsquigarrow C_n$ le calcul de M sur w . On suppose que chaque configuration C_k est égale à $u_k q_k v_k$. On va montrer que la fonction c définie par

$$c(\beta(w), k) = (\alpha(u_k), q_k, \beta(v_k))$$

est primitive récursive. À partir des transitions de M , on peut définir une fonction t telle que

$$t(q, a) = (p, b, x) \quad \text{si } q, a \rightarrow p, b, x \text{ est une transition de } M.$$

On complète t sur $\mathbb{N}^2$ en posant $t(i, j) = (0, 0, 0)$ si $i \geq n$ ou $j \geq m$. La fonction t est bien sûr primitive récursive. On note t_1 , t_2 et t_3 les fonctions $p_1(t)$, $p_2(t)$ et $p_3(t)$ qui donnent respectivement le nouvel état, la symbole à écrire et le déplacement de la tête de lecture. On a alors la définition récursive de c .

- $c(\beta(w), 0) = (0, q_0, \beta(w))$,
- Soit $a_k = \text{mod}(\beta(v_k), m)$. Si $t_3(q_k, a_k) = R$,

$$\begin{aligned} \alpha(u_{k+1}) &= m\alpha(u_k) + t_2(q_k, a_k), \\ q_{k+1} &= t_1(q_k, a_k), \\ \beta(v_{k+1}) &= \text{div}(\beta(v_k), m), \end{aligned}$$

- Soit $a_k = \text{mod}(\beta(v_k), m)$. Si $t_3(q_k, a_k) = L$,

$$\begin{aligned} \alpha(u_{k+1}) &= \text{div}(\alpha(u_k), m), \\ q_{k+1} &= t_1(q_k, a_k), \\ \beta(u_{k+1}) &= m(\beta(v_k) - a_k + t_2(q_k, a_k)) + \text{mod}(u_k, m) \end{aligned}$$

3.8.4 Thèse de Church

Après avoir introduit les machines de Turing, on a montré que les différentes variantes de ces machines conduisaient toutes à la même notion de calculable. De même, on a établi précédemment que les fonctions récursives sont équivalentes aux machines de Turing. Une fonction est récursive si et seulement si elle est calculable par une machine de Turing. Beaucoup d'autres modèles de calcul comme le λ -calcul ou les machines RAM ont été introduits. Chacun de ces modèles s'est avéré équivalent (ou plus faible) aux machines de Turing. Ceci a conduit Church à postuler que tous les modèles de calcul définissent la même notion de calculable. Ce postulat ne peut bien sûr pas être démontré mais il est universellement admis.

3.9 Compléments

3.9.1 Écritures des entiers dans une base

Pour la preuve du théorème de Presburger (théorème 3.58), on a utilisé des automates qui lisent des entiers en base 2. Il a été construit à l'exemple 3.60 un automate acceptant les multiples de 3 en base 2. De manière plus générale, pour des entiers k , q et r tels que $k \geq 2$ et $q > r \geq 0$, il est possible de construire un automate qui accepte les écritures en base k des entiers congrus à r modulo q .

Soit l'ensemble $Q = \{0, \dots, q-1\}$ et l'alphabet $B = \{0, \dots, k-1\}$. Soit l'automate déterministe $\mathcal{A} = (Q, B, E, \{0\}, \{r\})$ où l'ensemble E des transitions est donné par

$$E = \{p \xrightarrow{d} p' \mid kp + d \equiv p' \pmod{q}\}.$$

On vérifie par récurrence sur l qu'il y a un chemin étiqueté par $d_l \dots d_0$ de l'état p à l'état p' si l'entier $n = \sum_{i=0}^l k^i d_i$ vérifie $k^{l+1}p + n \equiv p' \pmod{q}$. En appliquant ce résultat à $p = 0$ et $p' = r$, on trouve que l'automate accepte bien les écritures $d_l \dots d_0$ des entiers n congrus à r modulo q .

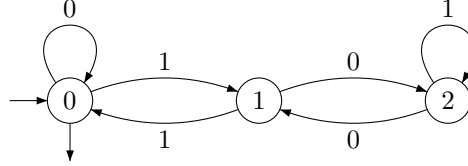


FIG. 3.13 – Automate des multiples de 3 en base 2

Exemple 3.68. Pour $k = 2$, $q = 3$ et $r = 0$, on obtient l'automate de la figure 3.13 qui accepte les écritures binaires des multiples de 3.

Définition 3.69. Soit X un ensemble d'entiers et soit $k \geq 1$ un entier. L'ensemble X est dit *k-rationnel* si l'ensemble des écritures des éléments de X en base k est rationnel.

Le cas $k = 1$ nécessite quelques commentaires. L'ensemble $\mathbb{N}$ s'identifie au monoïde libre 0^* sur un alphabet unaire. En effet l'application de $\mathbb{N}$ dans 0^* qui envoie tout entier n sur le mot 0^n est un isomorphisme de monoïde. Un ensemble d'entiers est donc 1-rationnel si et seulement si il correspond à un ensemble rationnel de mots sur 0^* . Pour cette raison, on parle simplement d'ensemble rationnel d'entiers. Ces ensembles ont une structure très simple que nous allons décrire.

Une *progression arithmétique* est un ensemble de la forme $\{kp+l \mid 0 \leq k \leq q\}$ pour des entiers l et p fixés et pour q égal soit à ∞ soit à un entier fixé. Un ensemble X d'entiers est *ultimement périodique* s'il existe deux entiers $l \geq 0$ et $p \geq 1$ tels que pour tout entier n supérieur à l , n appartient à X si et seulement si $n+p$ appartient à X . Ces définitions permettent d'énoncer la proposition suivante qui décrit complètement les ensembles rationnels d'entiers.

Proposition 3.70. *Pour un ensemble X d'entiers, les conditions suivantes sont équivalentes.*

1. X est rationnel.
2. X est ultimement périodique.
3. X est une union finie de progressions arithmétiques.

Preuve. L'implication $(1 \Rightarrow 2)$ se déduit directement de la forme des automates déterministes sur un alphabet unaire. Pour l'implication $(2 \Rightarrow 3)$, il est facile de décomposer un ensemble ultimement périodique en progressions arithmétiques. Pour l'implication $(3 \Rightarrow 1)$, une progression arithmétique est bien sûr un ensemble rationnel et le résultat découle de la clôture par union des ensembles rationnels. $\square$

Nous allons maintenant étudier les liens entre les notions de k -rationalité pour différentes valeurs de k . La construction d'un automate acceptant les écritures en base k des entiers congrus à r modulo q implique immédiatement la proposition suivante.

Proposition 3.71. *Tout ensemble rationnel d'entiers est k -rationnel pour tout entier k .*

Proposition 3.72. *Pour tout entiers $k \geq 1$ et $n \geq 1$, un ensemble X d'entiers est k -rationnel si et seulement si il est k^n -rationnel.*

Preuve. Les chiffres utilisés en base k^n sont les entiers de 0 à $k^n - 1$ qui s'écrivent avec n chiffres en base k . Cette remarque permet de passer d'un automate pour X en base k à un automate en base k^n . $\square$

La réciproque de la proposition est fausse. L'ensemble $\{2^n \mid n \geq 0\}$ des puissances de 2 n'est pas ultimement périodique. Par contre, l'ensemble de ses écritures en base 2 est le langage rationnel 10^* . Il découle immédiatement de la proposition que si $k^m = l^n$ pour des entiers $m, n \geq 1$, alors X est k -rationnel si et seulement si X est l -rationnel. Cette remarque motive la définition suivante.

Définition 3.73. Deux entiers k et l sont *multiplicativement indépendants* si $k^m \neq l^n$ pour tous les entiers $m, n \geq 1$.

Les entiers k et l sont multiplicativement indépendants si et seulement si le nombre réel $\log k / \log l$ est irrationnel. On a en effet $\log k / \log l = n/m$ dès que $k^m = l^n$. Si k et l sont multiplicativement indépendants, les nombres de la forme $p \log k - q \log l$ sont denses dans $\mathbb{R}$ et les nombres de la forme k^m / l^n sont denses dans $\mathbb{R}^+$.

Théorème 3.74 (Cobham 1969). *Tout ensemble d'entiers qui est k et l -rationnel pour deux entiers k et l multiplicativement indépendants est rationnel.*

La preuve de ce théorème est (très) difficile.

Un ingrédient essentiel de la preuve est le résultat suivant. Il faut remarquer qu'un ensemble X d'entiers est ultimement périodique si et seulement si son *mot caractéristique* x défini par $x_i = 1$ si $i \in X$ et $x_i = 0$ sinon est ultimement périodique. Un facteur w d'un mot infini x est *récurrent* s'il a une infinité d'occurrences dans x . Le lemme suivant donne une caractérisation des mots ultimement périodiques.

Lemme 3.75. *Un mot x est ultimement périodique si et seulement si il existe un entier n tel que x ait au plus n facteurs récurrents de longueur n .*

Preuve. Soit $r(n)$ le nombre de facteurs récurrents de x de longueur n . Puisque tout facteur récurrent est le préfixe d'un facteur récurrent plus long, on a l'inégalité $r(n) \leq r(n+1)$ pour tout entier n . Soit n_0 le plus petit entier tel que $r(n_0) = n_0$. Par définition de n_0 , on a $r(n_0 - 1) = n_0$. Ceci montre que tout facteur récurrent de longueur $n_0 - 1$ a un unique prolongement en un facteur récurrent de longueur n_0 . Ceci implique immédiatement que x est ultimement périodique. $\square$

Pour un mot infini x , la *fonction de complexité* (en facteurs) de x est la fonction $p_x(n)$ qui donne pour tout entier n le nombre de facteurs de x de longueur n . Le lemme précédent montre que tout mot x tel que $p_x(n) \leq n$ pour au moins un entier n est ultimement périodique. Les mots infinis x tels que $p_x(n) = n + 1$ pour tout entier n sont dits *sturmiens*. Ce sont des mots non ultimement périodiques de complexité minimale. Le mot de Fibonacci déjà considéré à l'exemple 1.19 (p. 14) est sturmien.

La première étape de la preuve du théorème de Cobham est de montrer que l'ensemble X est à lacune bornée. Un ensemble X d'entiers est à *lacune bornée* s'il existe un entier d tel que tout intervalle de longueur d contient au moins un élément de X .

Un ensemble L de mots sur un alphabet A est dit *dense à droite* si pour tout mot u sur A , il existe un mot v tel que $uv \in L$. Ceci signifie que le quotient $u^{-1}L$ est non vide pour tout mot u .

Lemme 3.76. *Soit X un ensemble d'entier k -rationnel et soit L l'ensemble de ses écritures en base k . L'ensemble X est à lacune bornée si et seulement si L est dense à droite.*

3.9.2 Machines de Turing sans écriture sur l'entrée

On s'intéresse dans cette partie aux machines qui n'écrivent pas sur leur entrée. Si la machine a plusieurs bandes, elle peut recopier l'entrée de la première bande sur une autre bande en parcourant l'entrée avec une tête de lecture et en écrivant en même temps sur une autre bande. Si la machine ne possède qu'une seule bande, l'idée intuitive est que la machine ne peut pas copier cette entrée sans la modifier. Pour effectuer la copie, l'unique tête de lecture doit faire des aller-retours entre l'entrée et le reste de la bande. La machine doit d'une façon ou d'une autre marquer là où elle en est dans la copie.

Théorème 3.77. *Soit M une machine de Turing à une seule bande qui n'écrit jamais sur son entrée, alors le langage $L(M)$ des mots acceptés est rationnel.*

Le théorème précédent énonce que pour une telle machine de Turing, il existe un automate qui accepte les mêmes entrées. Par contre, le passage de la machine de Turing à l'automate n'est pas calculable.

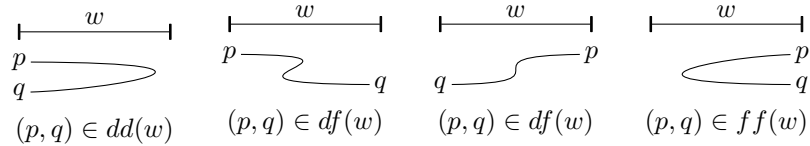


FIG. 3.14 – Définitions de dd , df , fd et ff

Preuve. Pour montrer que le langage $L(M)$ est rationnel, on montre qu'il est saturé par une congruence d'indice fini (avec un nombre fini de classes). On introduit quatre fonctions dd , df , fd et $ff : \Sigma^* \rightarrow \mathfrak{P}(Q \times Q)$ qui associent à chaque mot w de Σ^* des ensembles de paires d'états. L'ensemble $dd(w)$ est défini comme l'ensemble des paires (p, q) d'états de M telle qu'il existe un calcul de M de l'état p avec la tête sur la première lettre de w à l'état q avec la tête

après la dernière lettre de w qui ne lit que des lettres de w (cf. figure 3.14). Les trois fonctions df , fd et ff sont définies de manière similaire en considérant les chemins qui restent aussi sur w mais qui partent ou arrivent au début ou à la fin de w (cf. figure 3.14).

On définit maintenant la relation $\sim$ sur Σ^* par

$$w \sim w' \stackrel{\text{def}}{\iff} \begin{cases} dd(w) = dd(w') \\ df(w) = df(w') \\ fd(w) = fd(w') \\ ff(w) = ff(w') \end{cases}$$

On montre successivement les quatre propriétés suivantes de la relation $\sim$.

- La relation $\sim$ est une relation d'équivalence.
- Le nombre de ses classes est majoré par $2^{4|Q|^2}$.
- La relation $\sim$ est une congruence de monoïde sur Σ^* .
- La relation $\sim$ sature le langage $L(M)$. Ceci signifie que la relation $w \sim w'$ implique l'équivalence $w \in L(M) \iff w' \in L(M)$.

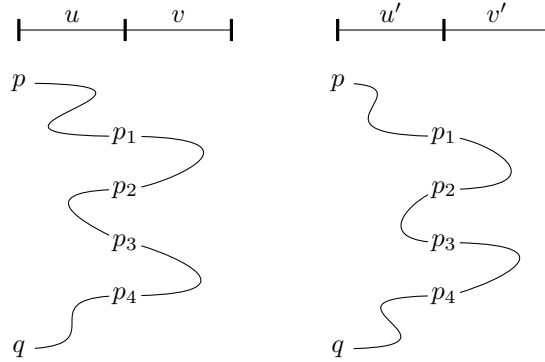


FIG. 3.15 – Égalité $dd(uv) = dd(u'v')$

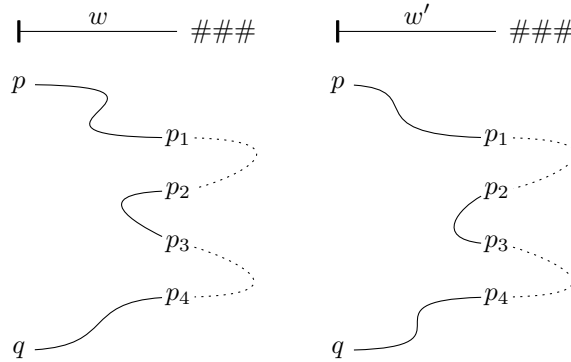


FIG. 3.16 – La relation $\sim$ sature $L(M)$

Par définition, la relation est relation d'équivalence. Puisque chacune des quatre fonctions dd , df , fd et ff peut prendre au plus $2^{|Q|^2}$ valeurs différentes, le nombre de classes de $\sim$ est majoré par $2^{4|Q|^2}$. On prouve maintenant que

$\sim$ est une congruence de monoïde (cf. définition 1.121 p. 49). Soient u, u', v et v' des mots sur Σ tels que $u \sim u'$ et $v \sim v'$. On montre d'abord l'égalité $dd(uv) = dd(u'v')$. Les trois autres égalités pour df , fd et ff se montrent de manière identique. Un chemin à l'intérieur de uv se décompose en une suite de chemins à l'intérieur de u ou v (cf. figure 3.15). Comme $u \sim u'$ et $v \sim v'$ les chemins à l'intérieur de u et v peuvent être remplacés par des chemins à l'intérieur de u' et v' . On montre maintenant que la relation $\sim$ sature $L(M)$. Soient w et w' deux mots sur Σ tels que $w \sim w'$. Un calcul sur l'entrée w se décompose en une suite de calculs à l'intérieur de w et hors de w (cf. figure 3.16). Comme $w \sim w'$, les calculs à l'intérieur de w peuvent être remplacés par des calculs à l'intérieur de w' pour obtenir un calcul sur l'entrée w' . On a finalement montré que le langage $L(M)$ est reconnu par le monoïde fini $\Sigma^*/\sim$. $\square$

Le théorème précédent montre que si l'on introduit des automates finis qui peuvent avancer ou reculer d'une position la tête de lecture à chaque transition, ces automates acceptent encore des langages rationnels. De tels automates sont appelés *automates boustrophédons* (*2-way automata* en anglais).

Chapitre 4

Complexité

4.1 Introduction

4.1.1 Objectifs

Lorsqu'il existe des algorithmes pour résoudre un problème, ceux-ci peuvent se révéler inutilisables dans la pratique parce qu'ils requièrent soit trop de temps soit trop de mémoire même pour des petits exemples. L'objectif de cette partie est d'identifier et de caractériser des problèmes pour lesquels il y a peu d'espoir de trouver des algorithmes efficaces. Il s'agit en quelque sorte d'une approche complémentaire de l'algorithmique dont l'objectif est de développer des algorithmes performants. L'idée est de fournir des arguments pour se convaincre que c'est probablement impossible pour certains problèmes. Un objectif est aussi de classifier les problèmes selon leur complexité en temps ou en espace mémoire mais avec des échelles beaucoup plus grossières qu'en algorithmique. Il est d'ailleurs question de distinguer les complexités polynomiale et exponentielle que les complexités linéaire et $n \log n$.

Dans cette partie, on ne considère naturellement que des problèmes décidables. On suppose implicitement que toutes les machines de Turing considérées s'arrêtent sur toutes leurs entrées.

4.1.2 Définition des complexités

Définition 4.1 (Complexité). Soit une machine de Turing M (*a priori* non déterministe) et soit $\gamma = q_0 w \rightsquigarrow C_1 \cdots \rightsquigarrow C_m$ un calcul de M d'entrée w .

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

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

$$t_M(w) = \max_{\gamma} \text{temps}(\gamma) \quad \text{et} \quad s_M(w) = \max_{\gamma} \text{espace}(\gamma)$$

On peut alors définir les complexités en temps t_M et en espace s_M de la machine M par :

$$t_M(n) = \max_{|w|=n} t_M(w) \quad \text{et} \quad s_M(n) = \max_{|w|=n} s_M(w)$$

C'est bien sûr plus le comportement asymptotique des fonctions t_M et s_M quand n devient grand qui est intéressant plus que des valeurs particulières de ces deux fonctions.

Comme à chaque transition de la machine, la tête de lecture se déplace d'au plus une position, la taille de la configuration augmente d'au plus une unité.

Lemme 4.2. *Si $n \leq t_M(n)$ alors $s_M(n) \leq t_M(n)$.*

Dans le modèle utilisé, la première configuration a une taille égale à celle de l'entrée. L'espace utilisé est donc au moins linéaire. Afin d'étudier des classes d'algorithmes utilisant peu d'espace mémoire, il faut introduire des machines de Turing spécifiques. Celles-ci ont deux bandes dédiées l'une à l'entrée et l'autre à la sortie. L'espace mémoire utilisé par la machine ne prend plus en compte ces deux bandes.

4.1.3 Graphes des configurations

Beaucoup de problèmes sur les machines de Turing se ramènent à des problèmes sur les graphes en considérant le graphe des configurations. Du coup, les algorithmes sur les graphes jouent un rôle essentiel. Le théorème de Savitch (théorème 4.22) peut par exemple être vu comme un résultat sur l'accessibilité dans les graphes en espace $\log^2 n$.

Le *graphe des configurations* d'une machine de Turing M est le graphe dont l'ensemble des sommets est l'ensemble de toutes les configurations de M et dont les arêtes sont les paires (C, C') de configurations telles que $C \rightsquigarrow C'$. Un chemin dans ce graphe est donc un calcul de la machine M .

L'acceptation d'une entrée par la machine se ramène à un problème d'accessibilité dans le graphe des configurations. En effet, l'entrée w est acceptée par M si et seulement si il y a un chemin de la configuration initiale q_0w à une configuration acceptante.

Le graphe des configurations est infini. Par contre, on s'intéresse souvent à une partie finie de ce graphe. Si la machine M s'arrête sur toutes les entrées w , l'ensemble des configurations accessibles à partir d'une configuration initiale est fini. La recherche d'une configuration acceptante peut alors se faire par un parcours du sous-graphe des configurations accessibles à partir de w . Si on a une borne sur le temps ou l'espace utilisé par la machine sur une entrée de taille n , on a alors une borne sur la taille des configurations qui détermine un sous-graphe fini du graphe des configurations.

La détermination d'une machine de Turing (cf. proposition 3.17) se fait par un parcours en largeur de l'arbre des calculs. Cet arbre est en fait un dépliage du graphe des configurations. La machine déterministe équivalente à une machine M effectue en fait un parcours en largeur du graphe des configurations de M .

4.2 Complexité en temps

4.2.1 Théorème d'accélération

Le théorème suivant énonce qu'on peut toujours accélérer une machine d'un facteur constant à condition que le temps de calcul est au moins proportionnel

à la taille de l'entrée. Il existe des machines qui fonctionnent en temps sous-linéaire. Le langage $a\Sigma^*$ peut par exemple être décidé en temps constant puisque la machine doit juste examiner la première lettre de l'entrée. Il est cependant vrai que dans la majorité des problèmes intéressants, la machine doit au moins lire entièrement l'entrée.

Théorème 4.3 (d'accélération). *Soit $k \geq 0$ un entier et soit M une machine de Turing. Si $n = O(t_M(n))$, alors il existe une machine de Turing M' , équivalente à M telle que $t_{M'}(n) \leq t_M(n)/k$*

Preuve.

$$\Sigma \quad \begin{array}{|c|c|c|c|c|c|c|c|} \hline a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & & \cdots \\ \hline \end{array}$$

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

$$\Sigma' \quad \begin{array}{|c|c|c|c|c|c|} \hline a_0, a_1 & a_2, a_3 & a_4, a_5 & & & \cdots \\ \hline \end{array}$$

□

Ainsi, lorsqu'on étudie la complexité d'un problème, les constantes multiplicatives ne sont pas significatives. Les complexités des machines de Turing et des problèmes sont exprimées sous la forme $O(f)$.

4.2.2 Changements de modèles

On s'intéresse maintenant à l'indidence d'un changement de modèle 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 -ième étape est de longueur $s_M(n) \leq t_M(n)$.

Au final, on a donc :

$$t_{M'}(n) = O(t_M^2(n))$$

– **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. L'entier k est le cardinal maximal des ensembles $\delta(p, a) = \{(q, b, x) \mid p, a \rightarrow q, b, x \in E\}$ pour tous $p \in Q$ et $a \in \Gamma$. On en tire, si $t(n) \geq n$:

$$t_{M'}(n) = O(t_M(n)k^{t_M(n)}) = 2^{O(t_M(n))}$$

Il faut noter le changement radical de complexité induit par cette transformation.

4.2.3 Classes de complexité en temps

Définition 4.4. Pour une fonction $f : \mathbb{N} \rightarrow \mathbb{R}^+$, on définit les classes

- $\text{TIME}(f(n))$ est l'ensemble des problèmes décidés par une machine de Turing *déterministe* (à plusieurs bandes) en temps $O(f(n))$.
- $\text{NTIME}(f(n))$ est l'ensemble des problèmes décidés par une machine de Turing *non déterministe* (à plusieurs bandes) en temps $O(f(n))$.

On définit alors les classes de complexité suivantes.

$$\begin{aligned} P &= \bigcup_{k \geq 0} \text{TIME}(n^k) & NP &= \bigcup_{k \geq 0} \text{NTIME}(n^k) \\ \text{EXPTIME} &= \bigcup_{k \geq 0} \text{TIME}(2^{n^k}) & \text{NEXPTIME} &= \bigcup_{k \geq 0} \text{NTIME}(2^{n^k}) \end{aligned}$$

Proposition 4.5. On a les inclusions triviales $P \subseteq NP \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME}$.

Exemple 4.6. Quelques exemples de problèmes de la classe P .

1. Problème d'accessibilité

Une instance est un triplet (G, s, t) où $G = (V, E)$ est un graphe orienté et s et t deux sommets de G . L'instance est positive s'il existe un chemin de s à t dans G . Un tel chemin peut être trouvé par un parcours en largeur du graphe, et donc en temps linéaire en la taille du graphe (nombre d'arêtes). C'est donc un problème polynomial. Ce problème est en fait dans la classe NL des problèmes résolubles en espace logarithmique par une machine non déterministe. Il est même complet pour cette classe (cf. [Sip97, p. 237])

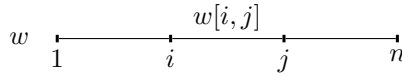


FIG. 4.1 – facteur $w[i, j]$ de w

2. Langages algébriques

On présente ici l'algorithme de Cocke-Kasami-Younger pour décider en temps cubique si un mot est engendré par une grammaire. On suppose fixé un langage algébrique L donné par une grammaire G qu'on peut supposer en forme normale quadratique grâce à la proposition 2.16. Une instance est un mot w et cette instance est positive si $w \in L$. Ce problème peut être résolu en temps cubique par un algorithme de programmation dynamique. Si w est le mot $a_1 \cdots a_n$, on note $w[i, j]$ le facteur $a_i \cdots a_j$ de w (cf. figure 4.1). Pour tous les entiers $1 \leq i \leq j \leq n$, on calcule l'ensemble $E_{i,j}$ des variables S telles que $w[i, j] \in L_G(S)$. Les ensembles $E_{i,j}$ sont calculés par récurrence sur la différence $j - i$. Si $i = j$, le facteur $w[i, i] = a_i$ appartient à $L_G(S)$ si et seulement si G contient une règle $S \rightarrow a_i$. Les ensembles $E_{i,i}$ sont donc facilement calculés. Pour $j - i \geq 1$, le facteur $w[i, j]$ appartient à $L_G(S)$ si et seulement si il existe une règle $S \rightarrow S_1 S_2$ de G et un entier $i \leq k < j$ tels que $w[i, k] \in L_G(S_1)$ et $w[k + 1, j] \in L_G(S_2)$. L'ensemble $E_{i,j}$ peut donc être calculé à partir des ensembles $E_{i,k}$ et $E_{k+1,j}$. Comme le temps pour calculer chaque $E_{i,j}$ est proportionnel à la différence $j - i$, le temps global de l'algorithme est cubique. Il faut remarquer que cette analyse ne prend pas en compte la taille

de la grammaire car celle-ci est supposée fixe. Cette taille est intégrée à la constante.

Exemple 4.7. Quelques exemples classiques de problèmes de la classe NP.

1. Chemin hamiltonien (HAM-PATH)

Un *chemin hamiltonien* dans un graphe G est un chemin qui passe une fois et une seule par chaque sommet de G (à ne pas confondre avec un *chemin eulérien* qui passe une fois et une seule par chaque arête du graphe). 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. Une instance est donc un triplet (G, s, t) et elle est positive s'il existe un chemin hamiltonien de s à t . 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 (SAT)

On considère des formules du calcul propositionnel, c'est-à-dire construites à partir de variables booléennes avec les opérateurs logiques $\neg$, $\wedge$ et $\vee$. Une telle formule est *satisfiable* s'il est possible d'affecter une valeur vrai (notée 1) ou faux (notée 0) à chacune des variables de telle façon que la formule ait la valeur vrai. Un exemple d'une instance de SAT est la formule suivante.

$$\varphi = ((\neg x_1 \vee x_4) \wedge (x_2 \vee (\neg x_1 \wedge \neg x_5))) \vee (x_1 \wedge (x_2 \vee \neg x_3))$$

La classe P est la classe de problèmes pour lesquels il est possible de calculer une solution en temps polynomial. On va voir que la classe NP est la classe des problèmes pour lesquels il est possible de vérifier une solution en temps polynomial.

Définition 4.8 (Vérificateur). Un *vérificateur* en temps polynomial pour un langage L est une machine déterministe M qui accepte des entrées de la forme $\langle w, c \rangle$ en temps polynomial en $|w|$ telle que $L = \{w \mid \exists c \langle w, c \rangle \in L(M)\}$.

L'élément c est en quelque sorte la *preuve* que w appartient bien à L .

Il est important que la machine M soit en temps polynomial en la taille de w et non en la taille de la paire $\langle w, c \rangle$. Cette définition impose que c peut être choisi de taille polynomiale en w . En effet, la machine M n'utilise qu'une partie de c de taille polynomiale en w . La partie non utilisée peut être supprimée.

Proposition 4.9 (Équivalence vérificateur et NP). *Un langage L est dans la classe NP si et seulement s'il existe un vérificateur polynomial pour L .*

Preuve. On montre que L est accepté par une machine (non déterministe) en temps polynomial si et seulement si il existe un vérificateur en temps polynomial pour L .

- $\Rightarrow$: Soit M une machine acceptant L en temps polynomial. Le vérificateur V prend en entrée c , une suite de transitions (de taille polynomiale) faites par M en calculant sur l'entrée w . Le vérificateur V simule alors M sur w pour vérifier que $w \in L(M)$. Ceci se fait en suivant c et donc en temps polynomial. Dans ce cas, la preuve que w appartient à L est le calcul acceptant de M .

- $\Leftarrow$: Soit V un vérificateur en temps polynomial pour L . La machine M choisit c de façon non déterministe et *simule* le calcul de V sur $\langle w, c \rangle$. Puisque V calcule en temps polynomial, c est de longueur polynomiale et le calcul de M est aussi polynomial.

□

Exemple 4.10. On revient sur les deux problèmes donnés à l'exemple 4.7

- Étant donné un graphe G , deux sommets s et t de G et un chemin γ dans G , il est facile de vérifier en temps linéaire que le chemin γ est hamiltonien.
- Étant donné une formule φ du calcul propositionnel et une affectation des variables, il est facile de calculer en temps linéaire la valeur de la formule induite par l'affectation et de vérifier que la formule est satisfiable.

4.2.4 NP-complétude

Réduction polynomiale

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

Définition 4.11 (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 $f : \Sigma_A^* \rightarrow \Sigma_B^*$ calculable en temps polynomial par une machine de Turing déterministe telle que :

$$w \in L_A \iff f(w) \in L_B.$$

L'existence d'une réduction polynomiale de A à B se note $A \leq_P B$.

L'idée intuitive est qu'un problème A se réduit à un problème B si le problème B est *plus compliqué* que le problème A puisqu'on peut ramener la résolution de A à la résolution de B . Il est facile de vérifier que la relation $\leq_P$ est transitive et réflexive (c'est un pré-ordre).

La proposition suivante découle directement des définitions. Il suffit de composer la machine de Turing qui calcule la réduction avec la machine de Turing qui décide B en temps polynomial pour obtenir une machine qui décide A en temps polynomial.

Proposition 4.12. Si $A \leq_P B$ et $B \in P$, alors $A \in P$.

Complétude

Définition 4.13 (NP-difficile). Un problème A est dit NP-*difficile* si tout problème B de NP se réduit à A , i.e. $B \leq_P A$.

Définition 4.14 (NP-complet). Un problème A est NP-*complet* s'il est dans la classe NP et s'il est NP-difficile.

L'idée intuitive est que les problèmes NP-complets sont les plus difficiles des problèmes NP puisque tous les problèmes de NP se ramènent à ces problèmes. Pour montrer qu'un problème est NP-difficile et en particulier NP-complet, on utilise généralement la proposition suivante qui découle directement des définitions.

Proposition 4.15. Si A est NP-difficile et si $A \leq_P B$, alors B est NP-difficile.

Pour utiliser la proposition précédente afin de montrer qu'un problème est NP-complet, il faut déjà disposer d'un problème NP-complet. Le théorème de Cook et Levin en fournit deux.

4.2.5 NP-complétude de SAT et 3SAT

La notion de NP-complétude est pertinente parce qu'il existe des problèmes NP-complets. C'est un résultat étonnant car on aurait pu imaginer une hiérarchie infinie de problèmes dans la classe NP. Les premiers problèmes NP-complets à avoir été trouvés sont les problèmes SAT et 3SAT. Le problème 3SAT est souvent très utile pour prouver que d'autres problèmes sont NP-complets car il se prête bien à des réductions.

Le problème SAT est le problème de la satisfiabilité d'une formule du calcul propositionnel (cf. exemples 4.7 et 4.10). Le problème 3SAT est le problème de la satisfiabilité d'une formule du calcul propositionnel où cette formule est en forme normale conjonctive avec au plus trois littéraux par clause. Une instance de 3SAT est donc une conjonction $c_1 \wedge \dots \wedge c_k$ de k clauses où chaque clause c_i est une disjonction $l_{i,1} \vee l_{i,2} \vee l_{i,3}$ de trois littéraux qui sont chacun soit une variable x_k soit la négation $\neg x_k$ d'une variable. La formule φ ci-dessous est un exemple d'une telle formule sur les variables x , y et z .

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

Le théorème suivant montre que la forme de la formule ne change rien à la complexité de la satisfiabilité puisque les deux problèmes sont NP-complets.

Théorème 4.16 (Cook & Levin). *Les deux problèmes SAT et 3SAT sont NP-complets.*

La preuve du théorème se fait par deux réductions successives. Une première réduction polynomiale de n'importe quel problème de NP au problème SAT est donnée. C'est le cœur même de la preuve. Ensuite une seconde réduction polynomiale de SAT à 3SAT est explicitée.

NP-Complétude de SAT

On commence par donner une réduction de n'importe quel problème de NP à SAT. Soit A un problème NP qui est décidé par une machine M machine non déterministe en temps polynomial. Pour toute entrée w de M , on montre qu'il existe une formule φ_w de taille polynomiale en $|w|$ telle que φ_w est satisfiable si et seulement si w est acceptée par M .

On note $n = |w|$ la taille de l'entrée w . Sans perte de généralité, on peut supposer que M décide A en temps n^k pour un entier k . Quitte à modifier légèrement la machine M , on suppose que tout calcul acceptant sur w est de longueur exactement n^k .

Comme la machine 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.

Ces configurations sont écrites les unes sous les autres pour obtenir le tableau de symboles éléments de l'alphabet $A = \Gamma \cup Q$ comme ci-dessous. 4.1.

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 .

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	...	#
$C_3 =$	...	...	...	...	...	#
$\vdots$						$\vdots$
$C_{n^k} =$	...	...	...	...	...	...

TAB. 4.1 – Tableau formé par les configurations

Pour chaque paire (i, j) telle que $0 \leq i, j \leq n^k$ et pour chaque symbole a de A , soit $x_{i,j,a}$ une variable qui code le fait que la case (i, j) contienne ou non le symbole a . Le nombre de ces variables est $|A|n^{2k+2}$ qui est polynomial en n . La formule φ_w est écrite en utilisant les variables $x_{i,j,a}$.

La formule φ_w se décompose en une conjonction $\varphi_0 \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ où chacune des quatre formules φ_0 , φ_1 , φ_2 et φ_3 code un des aspects de l'existence d'un chemin acceptant. Soit φ_0 la formule codant le fait que chaque cellule contient un unique symbole de A , φ_1 celle codant que la première ligne du tableau est bien q_0w , φ_2 celle codant le fait que chaque ligne est obtenue en appliquant une transition de M et φ_3 celle codant que le calcul est bien acceptant. La formule φ_w est alors égale à la conjonction $\varphi_0 \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3$. Il reste à expliciter les quatre formules φ_0 , φ_1 , φ_2 et φ_3 .

La formule φ_0 est la conjonction d'une formule pour chaque paire (i, j) . Cette dernière garantit qu'au moins une des variables $x_{i,j,a}$ a la valeur 1 pour un symbole a de A mais que deux variables $x_{i,j,a}$ et $x_{i,j,b}$ pour $a \neq b$ ne peuvent avoir la valeur 1 simultanément. La formule φ_0 s'écrit de la façon suivante

$$\varphi_0 = \bigwedge_{0 \leq i, j \leq n^k} \left[\left(\bigvee_{a \in A} x_{i,j,a} \right) \wedge \left(\bigwedge_{\substack{a, a' \in A \\ a \neq a'}} (\neg x_{i,j,a} \vee \neg x_{i,j,a'}) \right) \right]$$

La formule φ_1 s'écrit directement de la façon suivante.

$$\varphi_1 = x_{0,0,q_0} \wedge x_{0,1,w_1} \wedge \cdots \wedge x_{0,n,w_n} \wedge x_{0,n+1,\#} \wedge \cdots \wedge x_{0,n^k,\#}.$$

La formule φ_3 assure qu'au moins une des cases de la dernière ligne du tableau contient un état final. Elle s'écrit simplement de la façon suivante.

$$\varphi_3 = \bigvee_{q \in F} \left(\bigvee_{0 \leq j \leq n^k} x_{n^k,j,q} \right)$$

La formule φ_2 est un peu plus délicate à écrire. Il faut remarquer que le contenu d'une case (i, j) dépend uniquement des contenus des trois cases au-dessus $(i-1, j-1)$, $(i-1, j)$ et $(i-1, j+1)$ et de la transition qui a été effectuée par la machine pour passer de la configuration C_{i-1} à la configuration C_i . En fait, il est seulement nécessaire de connaître la transition effectuée lorsqu'une des trois cases au-dessus contient l'état de la configuration C_{i-1} . Si les trois

symboles contenus dans les trois cases au-dessus sont des symboles de bande, alors le symbole de la case (i, j) est identique au symbole de la case $(i-1, j)$. Ces remarques impliquent qu'il est possible de vérifier que la ligne i est bien obtenue à partir de la ligne $i-1$ uniquement en regardant les contenus des fenêtres de taille 2×3 . La machine M étant fixée, on considère tous les contenus possibles des fenêtres de taille 2×3 . Le nombre de ces contenus possibles ne dépend que de l'alphabet et des transitions de la machine. C'est donc un nombre fixe qui ne dépend pas de n . Le fait que six cases du tableau correspondent s'écrit comme une conjonction de six variables $x_{i,j,a}$. Le fait que toutes les parties de six cases du tableau correspondent à un des contenus possibles des fenêtre s'exprime par la conjonction pour $0 \leq i, j \leq n^k$ d'une disjonction sur les différents contenus. Ceci est une formule de taille polynomiale en n .

Réduction de SAT à 3SAT

On montre maintenant que le problème SAT se réduit polynomialement au problème 3SAT. Pour toute formule φ , il existe une formule φ' en forme normale conjonctive avec trois littéraux par clause et calculable en temps polynomial telle que φ est satisfiable si et seulement si φ' est satisfiable. Le calcul de φ' se fait en trois étapes. La première étape consiste à ramener toutes les négations devant les variables pour obtenir une formule sans négation autre que dans les littéraux. La seconde étape consiste à obtenir une formule en forme normale conjonctive mais avec des clauses ayant des nombres quelconques de littéraux. La troisième et dernière étape produit finalement la formule φ' en forme normale conjonctive avec trois littéraux par clause. Les première et troisième étapes sont faciles. Par contre la deuxième étape recèle une difficulté.

Première étape. La première étape fait descendre les négations à l'intérieur de la formule en utilisant les lois de De Morgan : $\neg(A \vee B) \equiv \neg A \wedge \neg B$ et $\neg(A \wedge B) \equiv \neg A \vee \neg B$. Cette descente des négations s'arrête lorsque toutes celles-ci se trouvent devant des variables. La formule obtenue est équivalente à la formule de départ. Cette transformation des formules se fait en temps linéaire si les formules sont représentées sous formes d'arbres et donc en temps polynomial sur une machine de Turing.

Deuxième étape. La formule de départ n'utilise plus de négation hormis dans les littéraux. La formule est donc uniquement constituée des opérateurs $\wedge$ et $\vee$. Une première idée naïve serait d'utiliser la distributivité de chacun de ces opérateurs par rapport à l'autre pour obtenir une formule équivalente en forme normale conjonctive. Cette approche ne marche pas car la formule obtenue peut avoir une taille exponentielle en la formule de départ. La formule φ ci-dessous est un exemple pathologique.

$$\varphi = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n)$$

Si la formule φ est développée, on obtient la conjonction de tous les monômes de la forme $z_1 \vee z_2 \vee \cdots \vee z_n$ où chaque z_i est soit x_i soit y_i . Comme il y a 2^n monômes de cette forme, la formule développée a une taille de l'ordre $n2^n$ alors que la taille de φ est de l'ordre de n .

L'idée pour éviter ce problème est de calculer une formule φ' qui n'est pas nécessairement équivalente à la formule de départ. Par contre la propriété d'être satisfiable reste préservée par la formule φ' . L'idée intuitive est que les affectations qui rendent vraie la formule φ se prolongent en plusieurs affectations qui rendent vraie la formule φ' .

Le calcul de la formule se fait par induction sur la forme de la formule φ .

- Si φ est de la forme $\varphi = E \wedge F$, on calcule récursivement des formules E' et F' en forme normale conjonctive pour E et F et on prend comme formule $\varphi' = E' \wedge F'$.
- Si φ est de la forme $\varphi = E \vee F$, on calcule récursivement des formules E' et F' en forme normale conjonctive. Supposons que ces deux formules sont respectivement donnée par $E' = E_1 \wedge \dots \wedge E_k$ et $F' = F_1 \wedge \dots \wedge F_m$ où les formules E_i et F_i sont des disjonctions de littéraux. On prend alors la formule φ' donnée par

$$\varphi' = (y \vee E_1) \wedge \dots \wedge (y \vee E_k) \wedge (\neg y \vee F_1) \wedge \dots \wedge (\neg y \vee F_m)$$

où y est une nouvelle variable qui est introduite.

Dans les deux cas, la formule φ' est satisfiable si et seulement si la formule φ est satisfiable.

On vérifie par récurrence sur la profondeur de la formule que le nombre de clauses de φ' est au plus proportionnel à la taille de φ . On montre également par récurrence que la taille de φ' est au plus quadratique en la taille de φ et que φ' se calcule effectivement en temps quadratique.

Troisième étape. Cette dernière étape consiste à remplacer les clauses n'ayant pas trois littéraux par une ou plusieurs clauses ayant exactement trois littéraux.

Si une clause a seulement un ou deux littéraux, on peut facilement la transformer en une clause à trois littéraux en répétant plusieurs fois le même littéral. Ainsi la clause $x_1 \vee x_2$ devient la clause $x_1 \vee x_2 \vee x_2$ qui est bien sûr équivalente.

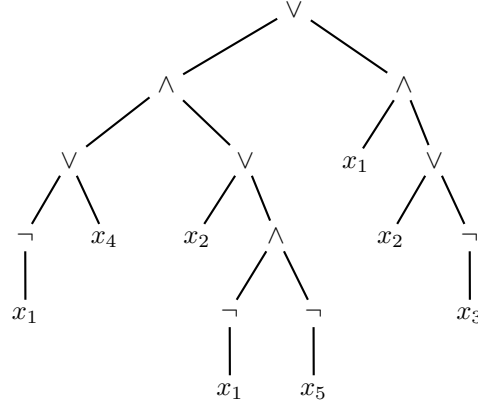
Si au contraire une clause a plus de trois littéraux, on effectue la transformation suivante. Supposons que la clause soit égale à la disjonction $l_1 \vee \dots \vee l_k$ des k littéraux $l_1, \dots, l_k$. On la remplace par la conjonction suivante de $k - 2$ clauses à trois littéraux.

$$(l_1 \vee l_2 \vee y_1) \wedge (\neg y_1 \vee l_3 \vee y_2) \wedge (\neg y_2 \vee l_4 \vee y_3) \wedge \dots \wedge (\neg y_{k-4} \vee l_{k-2} \vee y_{k-3}) \wedge (\neg y_{k-3} \vee l_{k-1} \vee l_k)$$

où les variables $y_1, \dots, y_{k-3}$ sont nouvellement introduites. On vérifie que toute affectation qui rend vraie la clause $l_1 \vee \dots \vee l_k$ se prolonge en une affectation aux variables $y_1, \dots, y_{k-3}$ qui rend vraie la conjonction. Réciproquement, toute affectation qui rend vraie la conjonction rend vraie la clause initiale. Cette dernière transformation se fait en temps linéaire en la taille de la formule et donc en temps polynomial sur une machine de Turing.

Il existe une autre façon de construire une formule φ' en forme normale conjonctive à partir d'une formule φ quelconque. On considère l'*arbre syntaxique* de la formule. Les nœuds internes de cet arbre sont étiquetés par les opérateurs $\neg$, $\wedge$ et $\vee$ et les feuilles par les variables. L'arbre syntaxique de la formule

$$\varphi = ((\neg x_1 \vee x_4) \wedge (x_2 \vee (\neg x_1 \wedge \neg x_5))) \vee (x_1 \wedge (x_2 \vee \neg x_3))$$

FIG. 4.2 – Arbre syntaxique de φ

est représenté à la figure 4.2.

On introduit une nouvelle variable pour chaque nœud interne de l'arbre. Suivant qu'un nœud interne est étiqueté par l'opérateur $\neg$, $\wedge$ ou $\vee$, on lui associe l'égalité $x_i = \neg x_j$, $x_i = x_j \wedge x_k$ ou $x_i = x_j \vee x_k$ où x_i est la variable correspondant à ce nœud et x_j et x_k les variables correspondants à ses fils. On considère la formule φ' qui est la conjonction des égalités associées à tous les nœuds internes et de la clause x_r où x_r est la variable introduite pour la racine.

Ensuite, on remplace chaque égalité par une conjonction de clauses. On a en effet les équivalences suivantes

$$\begin{aligned} x = \neg y &\equiv (x \vee y) \wedge (\neg x \vee \neg y) \\ x = y \wedge z &\equiv (x \vee \neg y \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg x \vee z) \\ x = y \vee z &\equiv (\neg x \vee y \vee z) \wedge (x \vee \neg y) \wedge (x \vee \neg z) \end{aligned}$$

qui permettent de remplacer chacune des égalités de φ' par une conjonction de deux ou trois clauses. Comme la taille de l'arbre est proportionnel à la taille de φ , la taille de φ' est également proportionnel à celle de φ .

4.2.6 Exemples de problèmes NP-complets

Dans cette partie, on donne quelques exemples classiques de problèmes qui sont NP-complets. On commence par trois problèmes sur les graphes, le chemin hamiltonien, la clique et la couverture de sommets et on termine par le problème de la somme d'entiers. Dans chacun des cas, la NP-complétude est prouvée en donnant une réduction polynomiale de 3SAT au problème considéré.

Chemin hamiltonien

On rappelle qu'un chemin hamiltonien d'un graphe G 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.

Proposition 4.17. *Le problème du chemin hamiltonien est NP-complet.*

Preuve. Cette réduction est écrite pour le problème du chemin hamiltonien sur un graphe orienté. On trouvera la réduction de HAM-PATH sur les graphes orientés à HAM-PATH sur les graphes non orientés en [HU79, p. 336]. $\square$

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

Pour montrer que le problème HAM-PATH est NP-difficile, on va réduire polynomialement le problème 3SAT à HAM-PATH. À chaque instance de 3SAT, on associe une instance de HAM-PATH qui a une solution si et seulement si l'instance de 3SAT en a une. Soit φ une instance de 3SAT, 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 appelée *gadget*. Pour chaque variable, ce graphe possède $2k + 2$ sommets et il est identique à celui représenté sur la figure 4.3.

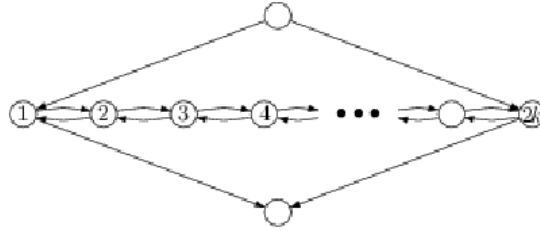
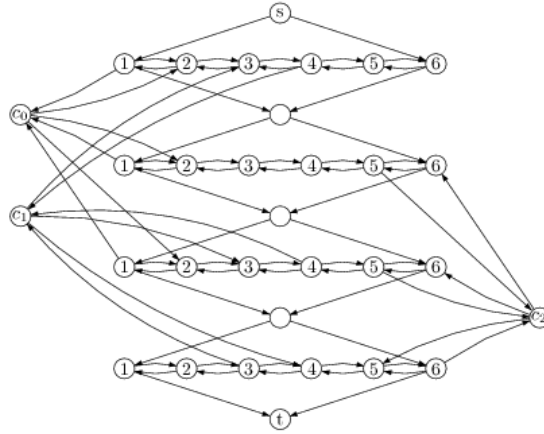


FIG. 4.3 – *gadget* associé à une variable

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 (cf. figure 4.4). 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 t est le dernier sommet du gadget de la dernière variable. On vérifie qu'il y a un chemin hamiltonien dans le graphe construit si et seulement si la formule φ est satisfiable.

La construction est illustrée sur la formule $\varphi = (x_0 \vee x_1 \vee x_2) \wedge (\neg x_0 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$. Les entiers k et m sont égaux à 3 et à 4 et on obtient le graphe représenté à la figure 4.4.

FIG. 4.4 – graphe associé à la formule φ

Clique

On commence par la définition d'une clique dans un graphe. Une clique de taille k est un sous-graphe complet de taille k .

Définition 4.18 (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 arête.

Proposition 4.19. *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 $v_1, \dots, v_k$ de G . Ce choix se fait en temps linéaire en la taille du graphe. Ensuite, l'algorithme vérifie que 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. $\square$

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

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

On introduit alors le graphe non orienté G dont l'ensemble des sommets est l'ensemble $V = \{l_1, \dots, l_{3k}\}$ 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 $\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$, on obtient le graphe représenté à la figure 4.5.

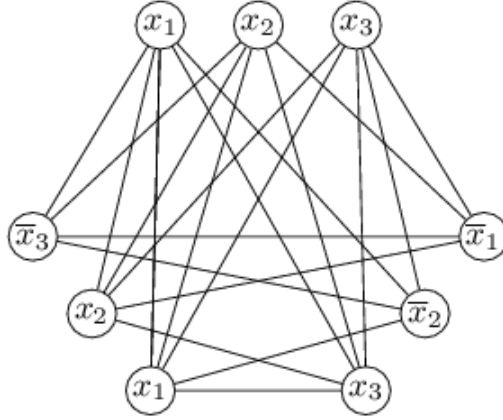


FIG. 4.5 – graphe associé à la formule φ

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 .

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 .

Proposition 4.20. *Le problème de savoir si un graphe a une couverture de sommets de taille k 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 $u_1, \dots, u_k$ puis vérifie que chaque arête du graphe est bien adjacente à un de ces sommets.

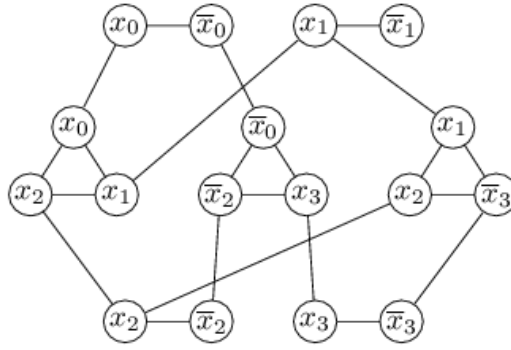
Pour montrer que le problème VERTEX-COVER est NP-difficile, on va réduire polynomialement le problème 3SAT à VERTEX-COVER. À chaque instance de 3SAT, on associe une instance de VERTEX-COVER qui a une solution si et seulement si l'instance de 3SAT en a une. Soit φ une instance de 3SAT, c'est-à-dire une 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 $\varphi = (x_0 \vee x_1 \vee x_2) \wedge (\neg x_0 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$. Les entiers k et m sont égaux à 3 et 4 et on obtient le graphe représenté à la figure 4.6.

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

FIG. 4.6 – graphe associé à la formule φ

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 . $\square$

Somme d'entiers

Le problème de la somme d'entiers est le suivant. Soient donnés une suite finie d'entiers $x_1, \dots, x_k$ et un entier s . Le problème est de savoir s'il est possible d'extraire une sous-suite de la suite donnée de manière à obtenir une suite dont la somme est égale à s . Plus formellement, il s'agit de trouver suite croissante d'indices $1 \leq i_1 < i_2 < \dots < i_n \leq k$ telle que $x_{i_1} + \dots + x_{i_n} = s$.

Proposition 4.21. *Le problème de la somme d'entiers 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 $i_1, i_2, \dots, i_n$ puis vérifie que la somme $x_{i_1} + \dots + x_{i_n}$ a pour valeur s .

Pour montrer que le problème SUBSET-SUM est NP-difficile, on va réduire polynomialement le problème 3SAT à SUBSET-SUM. À chaque instance de 3SAT, on associe une instance de SUBSET-SUM qui a une solution si et seulement si l'instance de 3SAT en a une. Soit φ une instance de 3SAT, 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 $c_0, \dots, c_{k-1}$ les k clauses de φ et soient $x_0, \dots, x_{m-1}$ les m variables de φ . Pour une variable x_i , 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 x_i 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 p_i et n_i définis de la façon suivante.

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

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

Les entiers p_i et n_i s'écrivent en base 10 avec $m + k$ chiffres égaux à 0 ou à 1. 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. 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.

À 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 à 1. Le chiffre à la position j est un 1 et tous les autres sont des 0.

On définit finalement le nombre s par la formule suivante.

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

L'entier s s'écrit en base 10 avec des chiffres 1 et 3. Son écriture en base 10 a la forme $1 \dots 13 \dots 3$ où le premier bloc comporte m chiffres 1 et le second bloc comporte k chiffres 3.

Nous allons illustrer cette construction sur la formule

$$\varphi = (x_0 \vee x_1 \vee x_2) \wedge (\neg x_0 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3).$$

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 4.2.

	x_3	x_2	x_1	x_0	c_2	c_1	c_0	Valeur
$p_0 =$	0	0	0	1	0	0	1	1001
$n_0 =$	0	0	0	1	0	1	0	1010
$p_1 =$	0	0	1	0	1	0	1	10101
$n_1 =$	0	0	1	0	0	0	0	10000
$p_2 =$	0	1	0	0	1	0	1	100101
$n_2 =$	0	1	0	0	0	1	0	100010
$p_3 =$	1	0	0	0	0	1	0	1000010
$n_3 =$	1	0	0	0	1	0	0	1000100
$q_0, q'_0 =$	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. 4.2 – 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 .

Le nombre d'entiers utilisés est égal à deux fois la somme du nombre de variables et du nombre de clauses. Il est donc proportionnel à la taille de la formule. La taille de ces entiers, c'est-à-dire le nombre de chiffres de leur écriture en base 10 est également proportionnel à la taille de la formule. La taille de l'instance de SUBSET-SUM est donc quadratique en la taille de la formule et peut être calculé en un temps de même ordre. La réduction de 3SAT à SUBSET-SUM est donc bien polynomiale. $\square$

4.3 Complexité en espace

On rappelle que la complexité en espace d'un calcul est le nombre de cellules de la bande qui sont utilisées, c'est-à-dire la taille maximale d'une configuration apparaissant dans le calcul. La fonction de complexité en espace d'une machine M est la fonction s_M qui donne pour chaque entier n la complexité en espace maximale d'un calcul sur une entrée de taille n (cf. définition 4.1 p. 141).

4.3.1 Changement de modèle

Les changements de modèles en nombre et forme de bandes sont sans grand changement sur la complexité en espace.

Théorème 4.22 (Savitch). *Soit f une fonction de $\mathbb{N}$ dans $\mathbb{R}^+$ telle que $f(n) \geq n$ pour tout n assez grand. Une machine non déterministe qui fonctionne en espace $f(n)$ est équivalente à une machine de Turing déterministe en espace $O(f^2(n))$.*

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  $t \leq 1$  then
    return  $C = C'$ 
else
    for all  $C''$  de taille  $\leq s(|w|)$  do

```



```

    if (ACCESS( $C, C'', \lceil t/2 \rceil$ )  $\wedge$  ACCESS( $C'', C', \lfloor t/2 \rfloor$ )) 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 $C_0 = q_0w$ 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 $2^{O(s(n))}$ (cf. proposition 4.24). Si w est accepté, il y a un calcul de C_0 à C_f en $t = 2^{Ks(n)}$.

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

Reste à évaluer la complexité en espace de ACCESS :

- taille de la pile des appels récursifs : $Ks(n)$
- espace d'un appel : C, C', C'' (configurations, espace $\leq s(n)$) t (en binaire $\leq Ks(n)$)

Au final, la machine utilise un espace $K^2s^2(n)$. $\square$

4.3.2 Classes de complexité en espace

Définition 4.23. Pour une fonction $f : \mathbb{N} \rightarrow \mathbb{R}^+$, on définit les classes

- $\text{SPACE}(f(n))$ est l'ensemble des problèmes décidés par une machine de Turing *déterministe* (à plusieurs bandes) en espace $O(f(n))$.
- $\text{NSPACE}(f(n))$ est l'ensemble des problèmes décidés par une machine de Turing *non déterministe* (à plusieurs bandes) en espace $O(f(n))$.

Le théorème de Savitch (théorème 4.22) garantit qu'il n'est pas nécessaire de distinguer les machines déterministes et non déterministes pour l'espace polynomial ou exponentiel. On définit les classes suivantes.

$$\begin{aligned} \text{PSPACE} &= \bigcup_{k \geq 0} \text{SPACE}(n^k) = \text{NPSpace} = \bigcup_{k \geq 0} \text{NSpace}(n^k) \\ \text{EXPSPACE} &= \bigcup_{k \geq 0} \text{SPACE}(2^{n^k}) \end{aligned}$$

4.3.3 Complexités en temps et en espace

On rappelle que si machine fonctionne en temps $t(n)$, elle fonctionne en espace au plus $t(n)$.

Proposition 4.24. *On a les inclusions suivantes*

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME} \subseteq \text{EXPSPACE}.$$

Le théorème de hiérarchie (corollaire 4.34 p. 163) permet de montrer que les inclusions $\text{P} \subsetneq \text{EXPTIME}$ et $\text{PSPACE} \subsetneq \text{EXPSPACE}$ sont strictes. Pour les inclusions intermédiaires, rien n'est connu même si la thèse généralement admise est qu'elles sont toutes strictes.

Preuve. Les inclusions $P \subseteq PSPACE$, $NP \subseteq NPSpace$ et $EXPTIME \subseteq EXPSPACE$ découlent directement du lemme 4.2.

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 $|Q|s(n)k^{s(n)} = 2^{O(s(n))}$. Ceci montre qu'une machine qui fonctionne en espace $s(n)$ fonctionne en temps $2^{O(s(n))}$. On a donc l'inclusion $PSPACE \subseteq EXPTIME$. $\square$

4.3.4 Exemples de problèmes dans PSPACE

Exemple 4.25. Quelques exemples de problèmes dans la classe 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 tous les mots sur A sont acceptés par $\mathcal{A}$, c'est-à-dire $L(\mathcal{A}) = A^*$.

(a) Si l'automate $\mathcal{A}$ est déterministe, il est très facile de savoir s'il accepte tous les mots sur A . Sans perte de généralité, on peut supposer que $\mathcal{A}$ est émondé. Dans ce cas, l'égalité $L(\mathcal{A}) = A^*$ est vraie si et seulement si $\mathcal{A}$ est complet et si tous ses états sont finaux.

(b) Le problème est intéressant si $\mathcal{A}$ n'est pas 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 de longueur inférieure à 2^n qui n'est pas accepté par $\mathcal{A}$. On utilise cette remarque pour construire un algorithme non déterministe en espace polynomial pour décider si $L(\mathcal{A}) \neq A^*$. Grâce au théorème de Savitch (théorème 4.22), cet algorithme peut être transformé en un algorithme déterministe.

L'algorithme procède la façon suivante. De manière non déterministe, l'algorithme devine un mot w de longueur inférieure à 2^n et simule le déterminisé de $\mathcal{A}$ sur ce mot pour montrer qu'il n'est pas accepté. Pour que l'espace reste polynomial, le mot w n'est pas mémorisé. Seule sa longueur est mémorisée.

- 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.
- L'algorithme s'arrête après 2^n étapes et rejette.
- L'algorithme s'arrête lorsqu'il obtient un groupe d'états comprenant un état non final et accepte.

L'algorithme utilise un espace linéaire pour stocker le nombre d'étapes effectuées et l'ensemble des états accessibles. Le principe de l'algorithme consiste à calculer de manière incrémentale l'automate déterminisé, au lieu de le calculer entièrement avant de l'analyser.

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 $\wedge, \vee, \neg$. La formule suivante est un exemple d'une telle formule.

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

Le problème est de savoir si une formule quantifiée close (sans variable libre) est vraie.

Proposition 4.26. QSAT est dans PSPACE.

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

Algorithme (QSAT(Φ) | Φ est close).

```

if  $\Phi$  n'a plus de quantificateurs then
  eval( $\Phi$ )
end if
if  $\Phi = \exists x \psi$  then
  QSAT( $\psi[x \leftarrow 0]$ )  $\vee$  QSAT( $\psi[x \leftarrow 1]$ )
end if
if  $\Phi = \forall x \psi$  then
  QSAT( $\psi[x \leftarrow 0]$ )  $\wedge$  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. $\square$

4.3.5 PSPACE-complétude

La définition suivante à l'analogue de la NP-complétude pour la classe PSPACE.

Définition 4.27. Un problème A est dit PSPACE-complet si :

1. A est PSPACE,
2. Tout problème B de PSPACE se réduit polynomialement *en temps* à A , (c'est-à-dire $B \leq_P A$).

Il est essentielle que la notion de PSPACE-complétude soit définie avec des réductions polynomiales en temps. En effet, comme le théorème de Savitch (théorème 4.22) permet de rendre déterministes les machines de Turing en espace polynomial, tout problème de la classe PSPACE se réduit en espace polynomial au problème trivial.

De manière plus générale, la complétude relative à une classe est toujours définie en utilisant des réductions qui sont plus faibles que les machines de la classe. Le théorème suivant montre qu'il existe des problèmes PSPACE-complets.

Théorème 4.28. Le problème 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 $O(n^k)$.

Soient C et C' deux configurations traduites en variables en nombre $O(n^k)$. On construit une formule $\Phi_{t,C,C'}$ qui est vraie si et seulement si il existe un calcul de longueur au plus t de C à C' . La formule $\Phi_{t,C,C'}$ est construite par récurrence

sur la valeur de t . Dans les définitions ci-dessous, on s'autorise à quantifier sur les configurations. Il faut voir ces quantifications comme des abréviations des quantifications sur les variables qui codent les configurations.

1. Pour $t = 1$, $\Phi_{1,C,C'}$ est la formule $(C = C') \vee (C \rightsquigarrow C')$.
2. Pour $t > 1$, $\Phi_{t,C,C'}$ est la formule

$$\begin{aligned} \exists C'' \quad \forall D, D' \\ [(D = C \wedge D' = C'') \vee (D = C'' \wedge D' = C')] \implies \Phi_{\lfloor t/2 \rfloor, D, D'}. \end{aligned}$$

La définition de $\Phi_{t,C,C'}$ pour $t > 1$ suscite plusieurs remarques. L'idée générale est de décomposer un calcul de longueur t en deux calculs de longueur $\lfloor t/2 \rfloor$ comme dans la preuve du théorème de Savitch (théorème 4.22). Cette définition donne une formule équivalente à la formule $\exists C'' \quad \Phi_{\lfloor t/2 \rfloor, C, C''} \wedge \Phi_{\lfloor t/2 \rfloor, C'', C'}$. Par contre, cette dernière définition ne conduirait pas une formule de taille polynomiale pour $t = 2^{K n^k}$ puisque la taille serait proportionnelle à t .

Cette définition est inexacte lorsque t est impaire mais ceci est sans conséquence car les valeurs de t utilisées sont des puissances de 2. Lorsque $t = 2t' + 1$, le calcul de longueur t devrait être décomposé en une étape de calcul suivie de deux calculs de longueur t' . L'étape de calcul est exprimée en utilisant la formule $\Phi_{1,C,C'}$. $\square$

Pour montrer qu'un problème est PSPACE-complet, on procède par réduction comme pour la NP-complétude en utilisant le lemme suivant.

Lemme 4.29. *Si $B \in \text{PSPACE}$, $A \leq_P B$ et A est PSPACE-complet, alors B est aussi PSPACE-complet.*

Le lemme découle directement des définitions et de la composition des réductions en temps polynomial.

4.3.6 Espace logarithmique

On s'intéresse à des problèmes qui peuvent être résolus avec un espace logarithmique en la taille de l'entrée. Cette question est uniquement pertinente si l'espace occupé par l'entrée n'est pas pris en compte dans l'espace utilisé par la machine. On introduit pour cela une variante des machines de Turing. Dans cette partie, on suppose que chaque machine à deux bandes spéciales appelée *bande d'entrée* et *bande de sortie*. La machine n'écrit jamais sur la bande d'entrée. Plus formellement, chaque transition écrit sur cette bande le caractère qui vient d'y être lu. Sur la bande de sortie, la tête de lecture ne se déplace que vers la droite. Cette dernière contrainte signifie que la machine ne peut jamais effacer ce qu'elle y a écrit. Cette bande se comporte un peu comme une imprimante. En contrepartie de ces deux contraintes, l'espace utilisé ne prend pas en compte ces deux bandes particulières. La taille d'une configuration est le nombre de cellules utilisées sur les autres bandes. En utilisant cette convention sur les machines de Turing, on peut définir les deux classes suivantes.

$$L = \text{SPACE}(\log n) \quad \text{et} \quad NL = \text{NSPACE}(\log n).$$

La base du logarithme est sans importance puisque toutes les complexités sont définies à un facteur multiplicatif près. Ces deux classes de complexités jouent

un rôle important d'une part parce que des problèmes intéressants s'y trouvent et d'autre part parce qu'elles fournissent des sous-classes de P. On commence par voir que l'accessibilité dans un graphe orienté est dans la classe NL.

Exemple 4.30. Soit $G = (V, E)$ un graphe orienté et soit s et t deux sommets de G . Pour trouver de façon non déterministe un chemin de s à t , il suffit de passer d'un sommet à un autre du chemin. Il est seulement nécessaire de mémoriser simultanément deux sommets du graphes. Ceci peut être fait en mémorisant les positions (comme des pointeurs) des deux sommets dans l'entrée. Ceci utilise un espace de l'ordre de $\log n$.

Exemple 4.31. Le problème 2SAT de la satisfiabilité d'une formule en forme normale conjonctive avec au plus deux littéraux par clause est dans la classe co-NL. Soit φ une instance de 2SAT. On introduit le graphe orienté G dont les sommets sont les variables x_i et leur négations $\neg x_i$. Chaque clause $l_i \vee l_j$ de φ induit les deux arêtes $\neg l_i \rightarrow l_j$ et $\neg l_j \rightarrow l_i$. On montre que φ n'est pas satisfiable si et seulement si il existe une variable x telle qu'il y ait dans G un chemin de x à $\neg x$ et de $\neg x$ à x .

4.4 Théorèmes de hiérarchie

On montre que si le temps ou l'espace alloué est augmenté, la classe des problèmes qui peuvent être résolus croît strictement. Il est cependant nécessaire de supposer que la ressource supplémentaire soit réellement utilisable. Ceci fait l'objet de la définition suivante.

Définition 4.32. Une fonction $f : \mathbb{N} \rightarrow \mathbb{N}$ est dite *constructible* en temps (resp. en espace) s'il existe une machine de Turing qui calcule pour une entrée 1^n le mot $1^{f(n)}$ en temps $O(f(n))$ (resp. en espace $O(f(n))$).

Il est facile de vérifier que la classe des fonctions constructibles en temps ou en espace contient toutes les fonctions usuelles $n \mapsto \lfloor \log n \rfloor$, $n \mapsto \lfloor \sqrt{n} \rfloor$, etc. et qu'elle est close pour la somme, le produit, l'exponentiation, la composition et toutes les opérations raisonnables.

Théorème 4.33 (de hiérarchie). *Soient $f : \mathbb{N} \rightarrow \mathbb{N}$ et $g : \mathbb{N} \rightarrow \mathbb{N}$ deux fonctions telles que $g = o(f)$. Si f est constructible en temps (resp. en espace), l'inclusion $\text{TIME}(g(n)) \subsetneq \text{TIME}(f(n))$ (resp. $\text{SPACE}(g(n)) \subsetneq \text{SPACE}(f(n))$) est stricte.*

La preuve que chacune des inclusions est stricte est basée sur un argument diagonal. Il est important de noter que ces inclusions ne concernent que les classes de complexités des machines déterministes.

Preuve. La preuve est pratiquement identique pour les deux inclusions. On construit une machine de Turing déterministe qui fonctionne en temps (ou en espace) $O(f(n))$ et qui accepte un langage qui ne peut pas être accepté par une machine déterministe en temps (ou en espace) $O(g(n))$.

On suppose fixé un codage des machines de Turing sur un alphabet Σ . On ajoute un nouveau symbole $\$$ et on construit une machine M_0 d'alphabet d'entrée $\Sigma \cup \{\$\}$. La machine M_0 accepte des entrées de la forme $\langle M \rangle \k où M est une machine de Turing sur l'alphabet $\Sigma \cup \{\$\}$ et k est un entier. Si une entrée w n'est pas de cette forme, elle est rejetée par M_0 . Si l'entrée w est de la

forme $\langle M \rangle \k , M_0 simule la machine M sur l'entrée w en ne dépassant pas le temps (ou l'espace) $f(|w|)$. Pour ce faire, M_0 a au préalable calculé $1^{f(|w|)}$ sur une bande. Si la simulation de la machine M dépasse le temps (ou l'espace) $f(|w|)$, l'entrée w est rejetée par M_0 . Si la simulation de M sur w aboutit, M_0 accepte si et seulement si M rejette. $\square$

Corollaire 4.34. *Les inclusions $P \subsetneq \text{EXPTIME}$ et $\text{PSPACE} \subsetneq \text{EXPSPACE}$ sont strictes.*

Preuve. Le théorème précédent montre que l'inclusion $\text{TIME}(2^n) \subsetneq \text{TIME}(2^{n^2})$ est stricte. Comme on a bien sûr l'inclusion $P \subset \text{TIME}(2^n)$, on a le résultat pour la complexité en temps. Le résultat pour la complexité en espace s'obtient de la même façon. $\square$

4.5 Machines alternantes

4.5.1 Définitions

Définition 4.35 (Machine alternante). Une *machine alternante* est une machine de Turing où l'ensemble Q des états est partitionné $Q = Q_\vee \uplus Q_\wedge$. Les états de $Q_\vee$ et $Q_\wedge$ sont appelés états *existentiels* et *universels*. Par extension, une configuration $C = uqv$ est dite existentielle (resp. universelle) si q est existentiel (resp. universel).

Définition 4.36 (Calcul). Un *calcul* d'une machine alternante est un arbre fini, étiqueté par des configurations et vérifiant les deux conditions suivantes.

1. Si un nœud x de l'arbre est étiqueté par une configuration existentielle C , alors x a un seul fils étiqueté par une configuration C' tel que $C \rightsquigarrow C'$.
2. Si un nœud x de l'arbre est étiqueté par une configuration universelle C , alors x a un fils étiqueté C' pour chaque configuration C' telle que $C \rightsquigarrow C'$.

Définition 4.37 (Acceptation). Une branche est *acceptante* si un état final apparaît sur celle-ci. Un calcul est *acceptant* si

- l'étiquette de la racine est la configuration initiale q_0w ,
- toutes les branches sont acceptantes.

4.5.2 Algorithme alternant pour QSAT

Algorithme ($\text{QSAT_ALT}(\Phi)$).

```

if  $\Phi$  sans  $Q$  then
    eval( $\Phi$ )
end if
if  $\Phi = \exists \psi \Phi$  then
     $\text{QSAT\_ALT}(\Phi[\psi \leftarrow 0]) \vee \text{QSAT\_ALT}(\Phi[\psi \leftarrow 1])$ 
end if
if  $\Phi = \forall \psi \Phi$  then
     $\text{QSAT\_ALT}(\Phi[\psi \leftarrow 0]) \wedge \text{QSAT\_ALT}(\Phi[\psi \leftarrow 1])$ 
end if
```

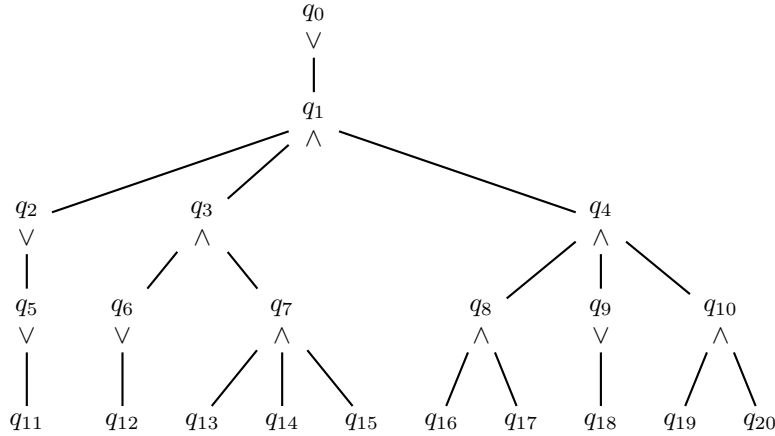


FIG. 4.7 – Un calcul dans un automate alternant

4.5.3 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 figure 4.7).

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.

4.5.4 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.

Définition 4.38. Pour une fonction $f : \mathbb{N} \rightarrow \mathbb{R}^+$, on définit les classes

- $\text{ATIME}(f(n))$ est l'ensemble des problèmes décidés par une machine de Turing alternante en temps $O(f(n))$.
- $\text{ASPACE}(f(n))$ est l'ensemble des problèmes décidés par une machine de Turing alternante en espace $O(f(n))$.

On définit alors les classes de complexité suivantes.

$$\text{AP} = \bigcup_{k \geq 0} \text{ATIME}(n^k)$$

$$\text{AL} = \text{ASPACE}(\log n) \qquad \text{APSPACE} = \bigcup_{k \geq 0} \text{ASPACE}(n^k)$$

Le théorème suivant relie les classes de complexité pour les machines alternantes aux classes pour les machines non déterministes.

Théorème 4.39. *Si la fonction $t : \mathbb{N} \rightarrow \mathbb{R}^+$ vérifie $t(n) \geq n$ pour n assez grand, on a*

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

$$\text{AP} = \text{PSPACE}.$$

Si la fonction $t : \mathbb{N} \rightarrow \mathbb{R}^+$ vérifie $t(n) \geq \log n$ pour n assez grand, on a

$$\text{ASPACE}(t(n)) = \text{TIME}(2^{O(t(n))})$$

$$\text{AL} = \text{P} \quad \text{et} \quad \text{APSPACE} = \text{EXPTIME}$$

Preuve. On commence par montrer l'inclusion $\text{ATIME}(t(n)) \subseteq \text{SPACE}(t(n))$. Soit A une machine alternante en espace $t(n)$ et soit w une entrée de taille n . La machine déterministe D effectue un parcours en profondeur du graphe des configuration (cf. p. 142) pour déterminer si la configuration initiale peut donner un calcul acceptant. Au cours du calcul, la machine D ne mémorise pas la configuration courante. Elle se contente de mettre dans la pile les transitions effectuées. Comme le nombre de transitions effectuées est borné par $t(n)$, l'espace de D est de taille $O(t(n))$.

On montre ensuite l'inclusion $\text{SPACE}(t(n)) \subseteq \text{ATIME}(t^2(n))$. Soit M une machine en espace $t(n)$ et soit w une entrée de taille n . La machine alternante A simule l'algorithme donné pour la preuve du théorème de Savitch (théorème 4.22). Cet algorithme détermine s'il existe un calcul de longueur k d'une configuration C à une configuration C' en trouvant une configuration C'' telle qu'il existe un calcul de longueur $\lceil k/2 \rceil$ de C à C'' et un calcul de longueur $\lfloor k/2 \rfloor$ de C'' à C' . La machine A utilise des configurations existentielles pour déterminer la configuration intermédiaire C'' et elle utilise une configuration universelle pour vérifier que d'une part, il existe un chemin de C à C'' et que d'autre part il existe un chemin de C'' à C' . La machine A fonctionne en temps $O(t^2(n))$.

On montre maintenant l'inclusion $\text{ASPACE}(t(n)) \subseteq \text{TIME}(2^{O(t(n))})$. Soit A une machine alternante en espace $t(n)$ et soit w une entrée de taille n . La machine déterministe D construit le graphe des configurations (cf. p. 142) de la machine A qui sont accessibles à partir de la configuration initiale q_0w . Ce graphe est de taille $2^{O(t(n))}$ et il peut être construit en temps $2^{O(t(n))}$. Ensuite la machine D marque toutes les configurations qui sont le début d'un calcul acceptant de A . Elle commence par marquer toutes les configurations acceptantes. Une configuration existentielle C est marquée dès qu'une configuration C' telle $C \rightsquigarrow C'$ est marquée. Une configuration universelle C est marquée dès que toutes les configurations C' telle $C \rightsquigarrow C'$ sont marquées. Ce processus de marquage se poursuit jusqu'à ce que plus aucune configuration ne puisse être marquée. L'entrée w est acceptée si la configuration initiale q_0w est marquée. Ce marquage peut être mené de façon déterministe en temps $2^{O(t(n))}$.

On montre finalement la dernière inclusion $\text{TIME}(2^{O(t(n))}) \subseteq \text{ASPACE}(t(n))$. C'est la construction la plus astucieuse. Soit M une machine déterministe en temps $2^{O(t(n))}$ et soit w une entrée de taille n . La machine alternante A vérifie que le tableau introduit dans la preuve du théorème de Cook et Levin (théorème 4.16) peut être rempli. Comme la machine M est déterministe, ce tableau peut être rempli de manière unique. La machine A utilise des configurations

universelles pour vérifier que chaque case de la dernière ligne peut être remplie. Le contenu de la case est choisi par une configuration existentielle. Ensuite, la machine utilise à nouveau des configurations universelles pour vérifier que le contenu de chaque case est cohérent avec les contenus des cases au dessus qui sont à leur tour choisies. La machine ne stocke pas le tableau. Elle utilise uniquement un nombre fini de pointeurs sur des cases du tableau. Elle utilise donc un espace $O(t(n))$. $\square$

4.6 Compléments

4.6.1 Machines à une bande en temps $o(n \log n)$

On montre dans cette partie que toute machine de Turing à une seule bande qui fonctionne en temps $o(n \log n)$ accepte un langage rationnel. L'ingrédient essentiel de la preuve est l'utilisation des suites de franchissements (*cutting sequences* en anglais) qui constituent un outil très classique. L'intérêt du théorème suivant est autant dans le résultat que dans la technique de preuve.

Théorème 4.40 (Hartmanis 1968). *Une machine de Turing à une seule bande qui fonctionne en temps $o(n \log n)$ accepte nécessairement un langage rationnel.*

Pour montrer que ce résultat est optimal, on commence par donner un exemple de machine de Turing à une seule bande qui fonctionne en temps $O(n \log n)$ et qui accepte un langage non rationnel.

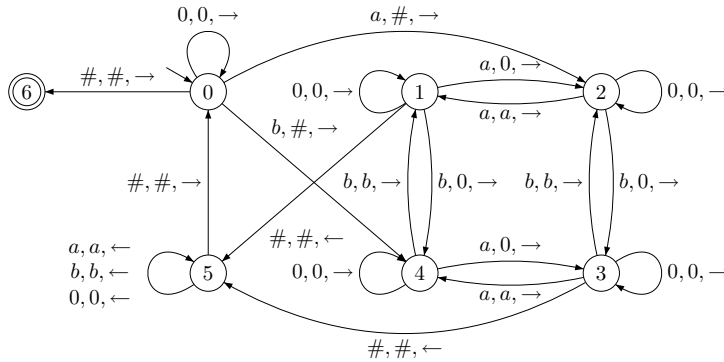
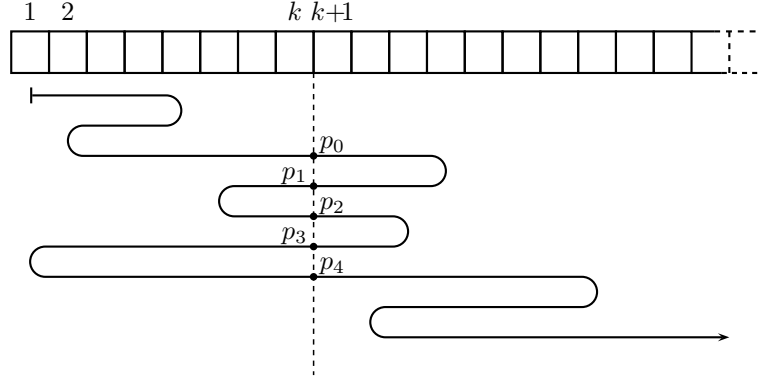


FIG. 4.8 – Machine acceptant $\{w \in (a + b)^* \mid |w|_a = |w|_b\}$

Exemple 4.41. La machine représentée à la figure 4.8 accepte l'ensemble $\{w \mid |w|_a = |w|_b\}$ des mots sur l'alphabet $\Sigma = \{a, b\}$ ayant le même nombre d'occurrences de a et de b . Le temps de calcul de cette machine pour une entrée de longueur n est de l'ordre $n \log n$. Comme ce langage n'est pas rationnel, le théorème précédent implique que cette machine est optimale.

Cette machine fonctionne de la manière suivante. Elle vérifie que le nombre d'occurrences de a est égal au nombre d'occurrences de b en remplaçant progressivement les a et les b par des 0. Lors d'un premier passage sur l'entrée, la machine remplace par 0 un a sur 2 et un b sur 2, tout en vérifiant que la parité

du nombre de a est égale à la parité du nombre de b . Ceci est réalisé par les états de 1 à 4. L'état 5 ramène la tête de lecture au début de la bande. Lors d'un second passage, la machine ignore les 0 et remplace par 0 un a sur 2 qui reste et un b sur 2 qui reste, tout en vérifiant encore que la parité du nombre de a est égale à la parité du nombre de b . La machine poursuit ainsi ses passages qui divisent par 2 les nombres de a et de b restant. La machine accepte quand il ne reste plus aucun a ou b sur la bande. Le nombre de passages effectués est proportionnel à $\log_2 n$ et la machine fonctionne bien en temps $O(n \log n)$. Si $|w|_a \neq |w|_b$, la machine échoue au k -ième passage où k est le plus petit entier tel que $|w|_a \not\equiv |w|_b \pmod{2^k}$.

FIG. 4.9 – Franchissements en k

Soit M une machine de Turing et soit $k \geq 1$ un entier. On dit qu'une étape de calcul $C \rightsquigarrow C'$ de M *franchit* la frontière entre les cases k et $k+1$ si $C = upav$, $C' = ubqv$ et $|u| = k$ en appliquant une transition $p, a \rightarrow q, b, R$ ou si $C = ucav$, $C' = uc'bv$ et $|u| = k$ en appliquant une transition $p, a \rightarrow q, b, L$. Soit $\gamma = C_0 \rightsquigarrow \dots \rightsquigarrow C_n$ un calcul de M où chaque configuration C_i est égale à $u_i q_i v_i$. Soit $i_0 < \dots < i_m$ la suite des indices où les étapes $C_i \rightsquigarrow C_{i+1}$ franchissent la frontière k . On appelle *suite de franchissements* de γ en k , la suite d'états $p_0, \dots, p_m$ où $p_j = q_{i_j+1}$. C'est l'état de la machine juste après le franchissement qui est pris en considération (cf. figure 4.9).

On commence par démontrer quelques propriétés des suites de franchissements puis on montre un résultat intermédiaire où on suppose que la machine fonctionne en temps linéaire. Dans la suite, on appelle *contenu de bande*, un mot infini $x = a_0 a_1 a_2 \dots$ constitué de symboles blancs $\#$ à partir d'un certain rang. On dit que γ est un calcul sur un contenu de bande x si x est le contenu de la bande au début de γ . On appelle le *type* d'un calcul le fait qu'il soit acceptant ou non. Pour une machine M , on note respectivement $t_M(n)$ et $r_M(n)$ la longueur maximale d'un calcul et la longueur maximale d'une suite de franchissements d'un calcul sur une entrée de longueur n . On note $|Q|$ le nombre d'états de M .

Le lemme suivant permet de combiner deux calculs où apparaît la même suite de franchissements.

Lemme 4.42. *Soient $x = uy$ et $x' = u'y'$ deux contenus de bande. Soient γ et γ' deux calculs sur x et x' tels que la suite de franchissements de γ en $|u|$ est identique à la suite de franchissements de γ' en $|u'|$. Alors, il existe un calcul γ'' sur uy' qui est identique à γ sur u et à γ' sur y' .*

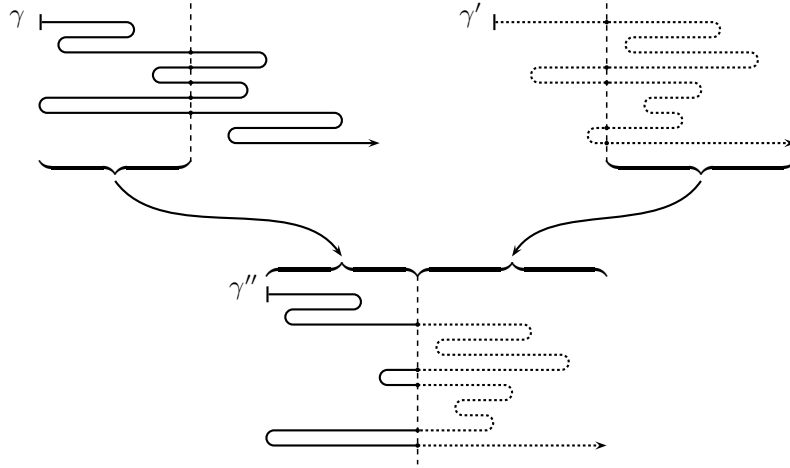


FIG. 4.10 – Jonction de calculs

Si la suite de franchissements commune à γ et γ' en $|u|$ et en $|u'|$ est de longueur paire, le chemin γ'' se termine comme γ . Si au contraire, cette suite est de longueur impaire, le chemin γ'' se termine comme γ' . L'idée de la preuve est illustrée à la figure 4.10.

Preuve. On construit un calcul γ'' sur uy' en combinant les deux calculs γ et γ' . Soit $p_0, \dots, p_n$ la suite de franchissements de γ en $|u|$ et de γ' en $|u'|$. Le calcul γ'' est obtenu en raccordant les calculs γ et γ' au niveau des franchissements en $p_0, \dots, p_n$. Le début de γ'' est le début de γ jusqu'au premier franchissement en p_0 . Ensuite le calcul γ'' se poursuit comme le calcul γ' après le premier franchissement et ceci jusqu'au deuxième franchissement en p_1 . Après le deuxième franchissement, il reprend comme γ jusqu'au troisième franchissement en p_2 . L'alternance se poursuit ainsi jusqu'au dernier franchissement. Le calcul γ'' se termine comme le calcul γ si n impair et comme γ' si n est pair. $\square$

Corollaire 4.43. *Soit γ un calcul sur une entrée $x = uv$ tel que les suites de franchissements en $|u|$ et $|uv|$ sont identiques. Il existe alors des calculs sur uy et $uvvy$ de même type que γ .*

Preuve. Il suffit d'appliquer le lemme précédent au deux factorisations $x = u \cdot vy = uv \cdot y$ de l'entrée x . $\square$

Proposition 4.44. *Une machine de Turing M à une seule bande telle que $r_M(n) \leq K$ pour tout $n \geq 0$ accepte nécessairement un langage rationnel.*

On rappelle qu'une relation d'équivalence $\sim$ sur un ensemble E sature un sous-ensemble F de E si toute classe d'équivalence de $\sim$ est soit incluse dans F soit disjointe de F . Dans ce cas, F est égal à l'union des classes qui l'intersectent.

On dit qu'une relation d'équivalence $\sim$ sur A^* est dite *régulière à droite* si $w \sim w'$ implique $wa \sim w'a$ pour tous mots w et w' et toute lettre $a \in \Sigma$. Une telle relation est en quelque sorte une demi congruence puisqu'une congruence est une relation d'équivalence qui est simultanément régulière à droite et à gauche. Un langage L est rationnel s'il est saturé par une relation d'équivalence d'indice

fini qui est régulière à droite. D'une part, la congruence syntaxique est bien sûr régulière à droite. D'autre part, il est très facile de construire un automate déterministe à partir d'une relation d'équivalence régulière à droite. Il suffit de prendre pour états les classes de cette relation d'équivalence et les transitions de la forme $[w] \xrightarrow{a} [wa]$ où $[w]$ dénote la classe de w .

Preuve. On commence par remplacer la machine M par une machine M' qui accepte uniquement en rencontrant un symbole $\#$. On suppose la machine M normalisée avec un état acceptant q_+ . On ajoute une transition $q_+, a \rightarrow q_+, a, R$ pour chaque symbole $a \in \Gamma \setminus \{\#\}$ et une transition $q_+, \# \rightarrow q_{++}, \#$ où q_{++} est un nouvel état qui devient l'état final. La nouvelle machine M' vérifie $r_{M'}(n) \leq K + 1$.

Pour un mot d'entrée w , on note F_w l'ensemble des suites de franchissements en $|w|$ des calculs sur une entrée de la forme wv pour un mot v . On définit la relation $\sim$ sur Σ^* par $w \sim w'$ si et seulement si $F_w = F_{w'}$. Comme les suites de franchissements de M sont de longueur bornée par K , la relation $\sim$ est d'indice fini.

Il reste à montrer que la relation $\sim$ est régulière à droite et qu'elle sature le langage L accepté par M . Soient w et w' deux mots tels que $w \sim w'$ et soit a une lettre de Σ . Soit $p_0, \dots, p_n$ une suite de franchissements de F_{wa} . Il existe un mot v tel que $p_0, \dots, p_n$ est la suite de franchissements en $|wa|$ d'un calcul γ sur une entrée wav . Puisque $F_w = F_{w'}$, il existe un calcul γ' sur une entrée $w'v$ ayant même suite de franchissements en $|w'|$ que γ en $|w|$. En appliquant le lemme 4.42, on trouve un chemin γ'' sur l'entrée $w'' = w'av$ ayant même suite de franchissements en $|w'|$ que γ en $|w|$ et qui se comporte comme γ sur la partie av de l'entrée. Ceci montre que $p_0, \dots, p_n$ est aussi la suite de franchissements de γ'' en $w'a$ et appartient donc à $F_{w'a}$. Par symétrie, on a $F_{wa} = F_{w'a}$ et la relation $\sim$ est donc régulière à droite.

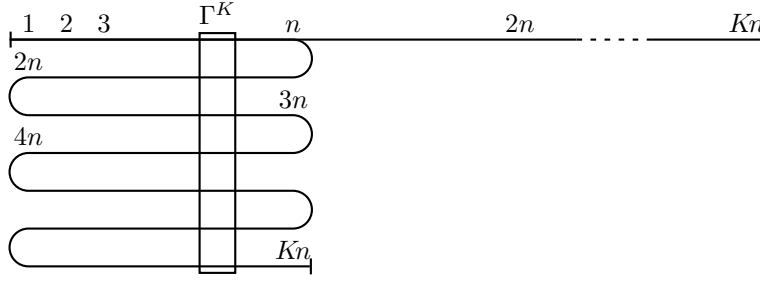
Si w est mot d'entrée accepté par la machine M , il existe un calcul acceptant γ sur w qui se termine sur un symbole $\#$. Si w' vérifie $w' \sim w$, il existe un calcul γ' sur une entrée $w'v$ ayant même suite de franchissements en $|w'|$ que γ en $|w|$. En appliquant à nouveau le lemme 4.42, il existe un calcul γ'' sur w' ayant même suite de franchissements en $|w'|$. Ce calcul peut être choisi acceptant et w' est aussi accepté par M . $\square$

Lemme 4.45. *Soit une machine M telle que $t_M(n) \leq Kn$ pour une constante K . Il existe une machine M' équivalente à M telle que $t_{M'}(n) \leq 2Kn$ et qui n'utilise que la partie de la bande où est écrite l'entrée.*

Preuve. Comme la machine M vérifie $t_M(n) \leq Kn$, elle utilise au plus Kn cellules de bande. L'idée générale de la construction est de replier la bande sur ses n premières cellules. Ceci revient à travailler sur le nouvel alphabet Γ^K . La nouvelle machine M' effectue une transition de plus lorsqu'elle passe sur les positions où est repliée la bande. Comme il y a au plus une telle transition supplémentaire pour chaque transition de M , on a la majoration $t_{M'} \leq 2Kn$. $\square$

La proposition suivante due à Hennie est une première étape dans la preuve du théorème.

Proposition 4.46 (Hennie 1965). *Une machine de Turing M à une seule bande telle que $t_M(n) \leq Kn$ accepte nécessairement un langage rationnel.*

FIG. 4.11 – Repliement des Kn cellules sur les n premières cellules

La preuve de la proposition est basée sur le lemme suivant.

Lemme 4.47. *Soit M une machine de Turing à une seule bande telle que $t_M(n) \leq Kn$ et qui n'utilise que la partie de la bande où est écrite l'entrée. Si un calcul a une suite de franchissements de longueur supérieure à $2K|Q|^K + K$, alors il existe au moins trois suites de franchissements identiques de longueur strictement inférieure à K .*

Preuve. Soit γ un calcul sur une entrée de longueur n ayant au moins une suite de franchissements de longueur supérieure à $2K|Q|^K + K$. Pour $i \geq 1$, la longueur de la suite de franchissements de γ en i est notée l_i . Puisque la machine est supposée n'utiliser que la partie de la bande où est écrite l'entrée, la valeur de l_i est égale à 0 pour tout $i > n$. La longueur de γ est égale à la somme $\sum_{i=1}^n l_i$. Puisque la machine vérifie $t_M(n) \leq Kn$, on a l'inégalité $\sum_{i=1}^n l_i \leq Kn$. Soit L le nombre de suites de franchissement de longueur strictement inférieure à K . Parmi les $n - L$ suites de franchissements de longueur au moins égale à K , au moins une est de longueur supérieure à $2K|Q|^K + K$. On a alors les inégalités

$$2K|Q|^K + K + (n - L - 1)K \leq \sum_{i=1}^n l_i \leq Kn.$$

En simplifiant, on obtient la minoration $L \geq 2|Q|^K$. Le nombre de suites différentes de longueur strictement inférieure à K est $1 + |Q| + \dots + |Q|^{K-1}$ qui est majoré par $|Q|^K - 1$. Comme $L \geq 2|Q|^K$, il existe au moins trois suites égales de longueur strictement inférieure à K . $\square$

On est maintenant en mesure d'établir la proposition 4.46.

Preuve de la proposition 4.46. Grâce au lemme 4.45, il peut être supposé que la machine M utilise uniquement la portion de la bande où est écrite l'entrée. On montre que $r_M(n)$ est borné par $2K|Q|^K + K$. Supposons par l'absurde qu'il existe un calcul sur une entrée w qui a une suite de franchissements de longueur supérieure à $2K|Q|^K + K$. Le chemin est en outre choisi de telle sorte que w soit de longueur minimale. D'après le lemme précédent, il existe trois suites de franchissements identiques de longueur inférieure à K . Deux au moins parmi ces trois suites se situent du même côté de la suite de longueur supérieure à $2K|Q|^K + K$. Grâce au corollaire 4.43, il existe un autre calcul sur une entrée strictement plus courte que w ayant encore une suite de franchissements de

longueur supérieure à $2K|Q|^K + K$. Ceci contredit le choix de w et montre que $r_M(n)$ est effectivement borné par $2K|Q|^K + K$. Grâce à la proposition 4.44, le langage accepté par la machine M est rationnel. $\square$

On termine par la preuve du théorème 4.40 qui utilise des arguments similaires à la preuve de la proposition 4.46 mais de manière plus fine.

Preuve du théorème 4.40. Soit M une machine de Turing à une seule bande telle que $\lim_{n \rightarrow \infty} t_M(n)/\log n = 0$. Si $(r_M(n))_{n \geq 0}$ est bornée, le résultat est acquis grâce à la proposition 4.44. On suppose donc maintenant que $(r_M(n))_{n \geq 0}$ n'est pas bornée. On définit par récurrence une suite d'entiers $(n_i)_{i \geq 0}$ de la façon suivante. On pose $n_0 = 0$ et n_i étant défini, on choisit pour n_{i+1} le plus petit entier tel que $r_M(n_{i+1}) > r_M(n_i)$. Par construction, la suite $(n_i)_{i \geq 0}$ vérifie $r_M(n_0) < r_M(n_1) < r_M(n_2) < \dots$ et elle est strictement croissante. Pour chaque $i \geq 0$, soit γ_i un calcul sur une entrée w_i tel que w_i est de longueur n_i et γ_i a une suite de franchissements de longueur $r_M(n_i)$.

Chaque calcul γ_i n'a pas trois suites de franchissements identiques parmi les n_i premières suites. Sinon, deux de ces suites de franchissements se situent du même côté de la suite de longueur $r_M(n_i)$. Grâce au corollaire 4.43, il existe un autre calcul sur une entrée strictement plus courte que w_i ayant encore une suite de franchissements de longueur $r_M(n_i)$. Ceci contredit le choix de n_i . Le nombre de suites de franchissements distinctes de longueur inférieure ou égale à $r_M(n_i)$ est $1 + |Q| + \dots + |Q|^{r_M(n_i)}$. Ce nombre est majoré par $|Q|^{r_M(n_i)+1} - 1$. Comme chaque suite de franchissements apparaît au plus deux fois parmi les n_i premières, on obtient l'inégalité $2|Q|^{r_M(n_i)+1} \geq n_i$. En prenant le logarithme, on a $r_M(n_i) \geq \log_{|Q|} n_i - \log_{|Q|} 2 - 1 \geq \log_{|Q|} n_i - 2$. Le temps minimal d'un calcul γ_i survient lorsque toutes les suites de franchissements de longueur j pour $0 \leq j \leq r$ apparaissent deux fois, l'entier r étant le plus petit possible pour que ces suites soient distinctes. On obtient donc l'inégalité

$$t_M(n_i) \geq 2 \sum_{j=1}^r j|Q|^j \geq 2r|Q|^r \quad \text{où} \quad r = \lfloor \log_{|Q|} n_i \rfloor - 3.$$

En utilisant finalement l'inégalité $r \geq \log_{|Q|} n_i - 3$, on arrive à l'inégalité $t_M(n_i) \geq 2(\log_{|Q|} n_i - 3)n_i|Q|^{-3}$ qui implique que $t_M(n_i)/n_i \log n_i$ ne tend pas vers 0 contrairement à l'hypothèse de départ. Ceci montre que la $r_M(n)$ est en fait borné et que le langage accepté par M est rationnel. $\square$

Bibliographie

- [ABB97] J.-M. Autebert, J. Berstel, and L. Boasson. Context-free languages and pushdown automata. In *Handbook of formal languages*, volume 1, pages 111–174. Springer, 1997.
- [Alm94] J. Almeida. *Finite Semigroups and Universal Algebra*. World Scientific, 1994.
- [Aut87] J.-M. Autebert. *Langages algébriques*. Masson, 1987.
- [Aut92] J.-M. Autebert. *Calculabilité et Décidabilité*. Masson, 1992.
- [BB90] J. Bestel and L. Boasson. Context-free languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 2, pages 59–102. Elsevier, 1990.
- [BBC93] G. Beauquier, J. Berstel, and P. Chrétienne. *Éléments d’algorithmique*. Masson, 1993.
- [Cas02] J. Cassaigne. *Algebraic Combinatorics on Words*, chapter Unavoidable patterns. Cambridge University Press, 2002.
- [HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
- [Koz97] D. C. Kozen. *Automata and Computability*. Springer, 1997.
- [Lib04] L. Libkin. *Elements of Finite Model theory*. Springer, 2004.
- [Lot83] M. Lothaire. *Combinatorics on Words*, volume 17 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, Reading, MA, 1983.
- [Pap95] C. Papadimitriou. *Computational complexity*. Addison-Wesley, 1995.
- [Per90] D. Perrin. Finite automata. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 1, pages 1–57. Elsevier, 1990.
- [Pin84] J.-É. Pin. *Variétés de Langages Formels*. Masson, 1984.
- [Sak03] J. Sakarovitch. *Éléments de théorie des automates*. Vuibert, 2003.
- [Sip97] M. Sipser. *Introduction to the Theory of Computation*. PWS publishing Company, 1997.
- [Ste90] J. Stern. *Fondements mathématiques de l’informatique*. McGraw-Hill, 1990.
- [Str94] H. Straubing. *Finite Automata, Formal Logic and Circuits Complexity*. Progress in theoretical computer science. Birkhäuser, 1994.

- [vLW92] J. H. van Lint and R. M. Wilson. *A Course in Combinatorics*. Cambridge University Press, 1992.

La bibliographie a été délibérément limitée aux références essentielles. Elle propose uniquement des livres et des chapitres de livres destinés à des non spécialistes. Les articles originaux ont été écartés au profit de documents plus pédagogiques. Par contre, beaucoup des ouvrages cités contiennent une bibliographie plus complète sur chacun des sujets traités.

Certains livres couvrent l'ensemble des quatre chapitres du cours. Dans cette catégorie, la référence la plus classique est [HU79]. D'autres livres comme [Sip97] ou [Koz97] proposent un traitement un peu plus moderne.

Un certain nombre de chapitres de livres de référence sont cités. Ces chapitres présentent l'intérêt de couvrir un sujet tout en étant relativement concis. Les chapitres [Per90] et [BB90] traitent ainsi des langages rationnels et des langages algébriques. Le chapitre [Cas02] est consacré aux motifs inévitables.

Des livres plus spécialisés offrent un approfondissement de certains sujets. Le livre [Sak03] traite de manière très complète les langages rationnels. Le livre [Pap95] donne un survol très complet de la théorie de la complexité.

D'autres livres couvrent des sujets à peine effleurés dans ce cours. Le livre [BBC93] couvre l'algorithmique. Un de ses chapitres traite la minimisation d'automates et l'algorithme d'Hopcroft. Le livre [vLW92] contient une preuve du théorème de Ramsey. Les livres [Pin84] et [Alm94] développent la théorie des semigroupes. Le livre [Str94] couvre très bien les liens entre automates, semigroupes et logique. Le livre [Lib04] traite de la théorie des modèles et en particulier des liens entre les automates et la logique.

Index

- algorithme
 - de Cocke-Kasami-Younger, 144
 - de Hopcroft, 38
 - de McNaughton-Yamada, 29
 - par élimination, 30
- arbre, 22
 - de dérivation, 74
 - des calculs, 108, 142
 - domaine d', 22
 - sous-arbre, 22
 - syntaxique, 150
- argument
 - diagonal, 111, 113, 133, 134, 163
- automate, 26
 - émondé, 27
 - à pile, 85
 - à pile déterministe, 88
 - alternant, 165
 - boustrophédon, 140
 - complet, 33
 - déterministe, 31
 - minimal, 34
 - normalisé, 27
 - quotient, 36
- chemin
 - eulérien, 8
 - hamiltonien, 145, 146, 151
- classe
 - AL, 165, 166
 - APSPACE, 165
 - AP, 165, 166
 - ASPACE, 165, 166
 - ATIME, 165
 - L, 162
 - P, 144, 159, 166
 - EXPSPACE, 159
 - EXPTIME, 144, 159, 166
 - NEXPTIME, 144, 159
 - NL, 144, 162
 - NP, 144, 159
 - PSPACE, 159, 166
 - de complexité, 144, 159, 165
- clique, 153
- codage, 98
- congruence, 169
 - d'automate, 36
 - de monoïde, 49, 138
 - de Nerode, 36
 - syntaxique, 49
- couverture de sommets, 155
- entiers
 - mult. indépendants, 137
- énumérateur, 110
- facteur, 8
 - récurrent, 137
- fonction
 - calculable, 114
 - d'Ackermann, 133
 - primitive récursive, 130
 - récursive, 130
- franchissement, 167
- grammaire, 61, 119, 144
 - ambiguë, 74
 - contextuelle, 124
 - contextuelle croissante, 124
 - en forme de Greibach, 82
 - en forme quadratique, 66
 - propre, 65
 - réduite, 64
- graphe
 - des configurations, 142
- hauteur
 - d'étoile, 44
 - d'étoile généralisée, 45
- idéal
 - d'ordre, 18

- langage
 - algébrique, 62, 144
 - d'un problème, 98
 - décidable, 112
 - de Dyck, 63, 81, 92
 - de Goldstine, 63, 78
 - récursivement énumérable, 110
 - rationnel, 25, 70, 92, 127, 137, 138, 167
 - sans étoile, 45, 52
- lemme
 - d'Arden, 30
 - d'Ogden, 76
 - de Dickson, 20
 - de König, 109
 - de l'étoile, 41
 - de Newman, 91
- mélange
 - de langages, 40
- machine de Turing, 99
 - à plusieurs bandes, 106
 - alternante, 164
 - déterministe, 107
 - linéairement bornée, 123
 - normalisée, 103
- mineur
 - de graphe, 24
- monoïde, 45
 - apériodique, 52
 - des transitions, 47
 - finiment engendré, 59
 - libre, 8, 46
 - polycyclique, 92
 - quotient, 48
 - syntactique, 49, 51
- morphisme, 40, 59, 116
 - alphabétique, 79
 - de monoïdes, 8, 14, 46
 - effaçant, 8
- mot
 - de de Bruijn, 8
 - de Fibonacci, 10, 14, 138
 - de Thue-Morse, 15
 - de Zimin, 17
 - infini, 13
 - infini périodique, 13
 - primitif, 10
 - sans chevauchement, 15
 - sans carré, 15
 - sturmien, 138
 - ultimement périodique, 137
 - ultimement périodique, 14
- occurrence
 - circulaire, 8
- ordre
 - des sous-mots, 21
 - hiérarchique, 22
 - lexicographique, 22, 41
- période
 - d'un mot, 9
- parcours
 - en largeur, 108, 144
- partie
 - reconnaissable, 59
- préfixe, 8
- préordre, voir quasi-ordre
- problème
 - NP-complet, 146, 151
 - NP-difficile, 146
 - PSPACE-complet, 161
 - d'accessibilité, 144
 - de correspondance de Post, 116
 - de décision, 97
 - de la somme, 156
- programmation
 - dynamique, 144
- quasi-ordre, 18
 - bien fondé, 18
 - bon, 18
- Quines, 121
- réduction
 - algorithmique, 114
 - polynomiale, 146–156
- relation
 - confluente, 90
 - fortement confluente, 90
 - localement confluente, 90
 - noethérienne, 90
 - rationnelle, 58
- relation d'équivalence
 - régulière à droite, 169
- SAT, 145–147
- 3SAT, 147, 151
- satisfiabilité, voir SAT et 3SAT
- sous-graphe, 24

- sous-monoïde, 9
- sous-mot, 20, 40
- substitution, 65
 - algébrique, 79
 - inverse, 51
- suffixe, 8
- théorème
 - d’Immerman, 126
 - de Büchi, 130
 - de Benois, 93
 - de Chomsky-Schützenberger, 81
 - de Cobham, 137
 - de Cook-Levin, 147
 - de Fine et Wilf, 10
 - de Guibas et Odlyzko, 11
 - de hiérarchie, 163
 - de Higman, 21
 - de Howson, 94
 - de Kleene, 26
 - de Kruskal, 23
 - de Parikh, 70
 - de Post, 118
 - de Presburger, 127
 - de récursion, 122
 - de Ramsey, 43
 - de Rice, 115
 - de Savitch, 158
 - de Schützenberger, 54
- thèse de Church, 135
- transduction
 - rationnelle, 58
- vérificateur, 145