Éléments d'Informatique Théorique

Support de cours

Olivier Biberstein José Rolim

Août 1997

Université de Genève

Table des matières

Chapitre 1

Notions de base

Puisque ce document s'intéresse particulièrement aux notions de mot et de langage qui seront constamment employées par la suite, ce premier chapitre présente brièvement les quelques définitions de base relatives à ces notions.

1.1 Notion d'alphabet, de mot et de langage

Définition 1.1.1 *Alphabet, mot*
Un *alphabet* Σ est un ensemble fini de symboles comme par exemple des lettres, des chiffres ou d'autres sigles. Un *mot* w défini sur un alphabet Σ est une suite, une chaîne ou une séquence finie de symboles appartenant à Σ. $\Diamond$

Définition 1.1.2 *Langage*
Un *langage* L défini sur un alphabet Σ est un ensemble de mots définis sur l'alphabet en question. Cet ensemble de mots peut être infini. $\Diamond$

Par exemple, $L = \{\mathsf{acbb}, \mathsf{accbb}, \mathsf{acccbb}, \ldots\}$ est un langage défini sur l'alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ qui est composé de tous les mots débutant par le symbole a suivi d'au moins un symbole c et se terminant par deux symboles bb.

1.1.1 Opérations sur les mots

L'opération de concaténation consiste à juxtaposer deux mots afin d'en obtenir un nouveau. C'est une opération associative mais non commutative.

Définition 1.1.3 *Concaténation*
On note l'opération de *concaténation* de deux mots $w_1 = s_1 s_2 \cdots s_m$ et $w_2 = t_1 t_2 \cdots t_n$ définis sur un alphabet Σ par $w_1 \cdot w_2 = s_1 \cdots s_m t_1 \cdots t_n$. On note aussi cette opération simplement $w_1 w_2$. $\Diamond$

Définition 1.1.4 *Longueur*
La *longueur* d'un mot w que l'on note généralement $|w|$ fait correspondre à chaque mot le nombre de symboles qui le composent. $\Diamond$

On définit un mot particulier appelé *mot vide* que l'on note ϵ. Ce mot n'est composé d'aucun caractère, sa longueur est donc nulle ($|\epsilon| = 0$). Ainsi, pour tout mot w appartenant à un langage L on a $w \cdot \epsilon = \epsilon \cdot w = w$.

Définition 1.1.5 *Miroir*
L'opération *miroir* associe à un mot $w = s_1 s_2 \ldots s_n$ le même mot renversé. On la note $w^R = s_n \ldots s_2 s_1$. ◊

Propriétés:

1. Si $w = w^R$ alors le mot w est appelé un *palindrome*, par exemple `radar` ou `esoperesteicietserepose`,
2. $(w^R)^R = w$,
3. $(w_1 w_2)^R = w_2^R w_1^R$.

Définition 1.1.6 *Puissance*
La *puissance* d'un mot w, notée w^n où $n \geq 0$, est définie par:

1. $w^0 = \epsilon$,
2. $w^{n+1} = w^n w$. ◊

Par exemple, le mot $w = \mathtt{abbc}$ élevé à la puissance 3 vaut $w^3 = \mathtt{abbcabbcabbc}$.

1.1.2 Opérations sur les langages

Définissons tout d'abord l'ensemble des mots définis sur un alphabet donné Σ et de longueur finie supérieure ou égale à 1. On note cet ensemble Σ^+. Par exemple, si $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ alors l'ensemble vaudra $\Sigma^+ = \{\mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \mathtt{aaa}, \mathtt{aab}, \mathtt{aba}, \mathtt{abb}, \mathtt{baa}, \mathtt{bab}, \ldots\}$. Cet ensemble est donc infini si et seulement si $\Sigma \neq \emptyset$. On défini également ce même ensemble muni du mot vide, on le note alors Σ^*.

Définition 1.1.7 Σ^+, Σ^*
Soit un alphabet Σ. L'ensemble Σ^+ des mots définis sur Σ de longueur finie supérieure ou égale à un est défini inductivement par

1. tout élément de Σ appartient à Σ^+,
2. si $a \in \Sigma$ et $w \in \Sigma^+$ alors $aw \in \Sigma^+$,
3. Σ^+ ne contient que des objets obtenus en appliquant un nombre fini de fois les règles 1 et 2.

De plus, $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$. ◊

1.1.2.1 Opérations ensemblistes

Comme nous l'avons vu les langages sont des ensembles de mots. Les opérations habituelles concernant les ensembles telles que l'union, l'intersection et la complémentation sont alors tout à fait applicables aux langages.

Définition 1.1.8 *Union, intersection*
L'*union* de deux langages L_1 et L_2 définis sur un alphabet Σ $(L_1 \cup L_2)$ et leur *intersection* $(L_1 \cap L_2)$ correspondent exactement à l'union et à l'intersection d'ensembles. ◊

Définition 1.1.9 *Complémentation*
Soit un langage L défini sur un alphabet Σ, la *complémentation* de L, notée $\overline{L}$, est définie par $\overline{L} = \Sigma^* \setminus L$. ◊

1.1.2.2 Opérations non ensemblistes

Définition 1.1.10 *Concaténation*
Soient L_1 et L_2 deux langages définis sur l'alphabet Σ, l'opération de *concaténation* est définie comme suit:

$$L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \text{ et } w_2 \in L_2\}$$

$\Diamond$

L'élément neutre de cette opération est donc l'ensemble $L_\epsilon = \{\epsilon\}$ $(L \cdot L_\epsilon = L_\epsilon \cdot L = L)$ et l'élément absorbant le langage $\varnothing$ $(L \cdot \varnothing = \varnothing \cdot L = \varnothing)$. Il faut noter que ces deux ensembles ne sont pas identiques $(L_\epsilon \neq \varnothing)$. Par exemple, soient $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ et les deux langages $L_1 = \{\mathtt{a}, \mathtt{b}\}$ et $L_2 = \{\mathtt{bac}, \mathtt{b}, \mathtt{a}\}$. La concaténation de L_1 et L_2 vaut $L_1 \cdot L_2 = \{\mathtt{abac}, \mathtt{ab}, \mathtt{aa}, \mathtt{bbac}, \mathtt{bb}, \mathtt{ba}\}$.

Définition 1.1.11 *Puissance*
La *puissance* d'un langage L, notée L^n où $n \geq 0$, est définie par:

1. $L^0 = L_\epsilon = \{\epsilon\}$,
2. $L^{n+1} = L^n \cdot L$. $\Diamond$

Définition 1.1.12 *Fermeture itérative*
La *fermeture itérative* ou *fermeture de Kleene* d'un langage L, notée L^*, est l'ensemble des mots résultant d'une concaténation d'un nombre fini de mots de L. Formellement on a:

$$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \ldots = \bigcup_{i \geq 0} L^i$$

Le mot vide appartient donc à L^* mais pas forcément à L^+ qui est définie comme suit:

$$L^+ = L^1 \cup L^2 \cup L^3 \ldots = \bigcup_{i \geq 1} L^i = L \cdot L^* = L^* \cdot L$$

$\Diamond$

1.1.3 Formalismes de spécification des langages

Pour spécifier un langage, c'est-à-dire décrire formellement le langage, différents formalismes sont à disposition. La première solution consiste à énumérer de manière exhaustive les composants du langage, mais il est clair que cette tâche est irréalisable lorsqu'il s'agit d'ensembles infinis. Pour décrire des langages infinis il faut alors utiliser des formalismes plus riches comme les *automates à états fini*, les *expressions régulières*, les *grammaires*, les *machines de Turing*, etc.

Chapitre 2

Langages réguliers

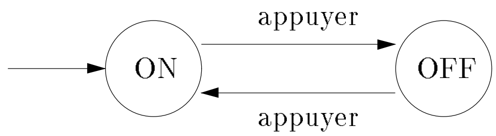
2.1 Automates à états finis

Un *automate à états fini* (souvent résumé par AF) est un modèle d'un système et de son évolution, c'est-à-dire une description formelle du système et de la manière dont il se comporte. Les automates sont des modèles très simples et ne sont évidemment pas en mesure de décrire tous les systèmes que l'on est amené à modéliser. Tout au long des chapitres suivants les modèles vont être enrichis afin de pouvoir décrire des systèmes de plus en plus sophistiqués.

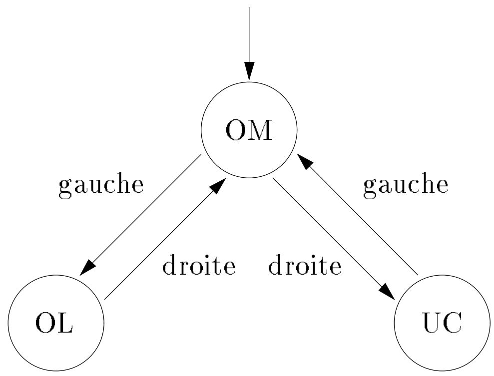
Un AF est composé d'un *ensemble fini d'états* (représentés graphiquement par des cercles), d'une *fonction de transition* décrivant l'action qui permet de passer d'un état à l'autre (ce sont les flèches) et d'un *état initial* (l'état que désigne la flèche isolée). Un AF est donc un graphe orienté où les noeuds correspondent aux états et les arcs aux flèches.

Exemples 2.1.1

1. Un bouton poussoir passant alternativement d'un état "ON" à un état "OFF" par l'action d'appuyer sur le bouton peut être représenté par:



2. Un sélecteur rotatif des canaux OL, OM et UC d'un poste de radio pouvant être tourné dans les deux sens, du canal OL à UC en passant par OM:



Définition 2.1.1 *Automate à états fini*
Un *automate à états fini* est un 4-tuple $\langle Q, \Sigma, \delta, q_0 \rangle$ où

- Q est l'ensemble (fini) des états de l'automate,
- Σ est l'ensemble fini des symboles,
- $\delta : Q \times \Sigma \to Q$ est la fonction de transition exprimant le passage entre deux états,
- $q_0 \in Q$ est l'état initial. $\Diamond$

2.1.1 Fonction δ^*

La fonction $\delta : Q \times \Sigma \to Q$ exprime le symbole (ou l'action) qui permet le passage d'un état à l'état suivant. Son extension est la fonction $\delta^* : Q \times \Sigma^* \to Q$ qui fournit l'état suivant pour une séquence de symboles. Rappelons que Σ^* correspond à l'ensemble de tous les mots que l'on peut former sur Σ.

Définition 2.1.2 *Fonction δ^**
Pour un état $q \in Q$,un mot $w \in \Sigma^*$ et un symbole $s \in \Sigma$, la fonction $\delta^* : Q \times \Sigma^* \to Q$ est définie par:

1. $\delta^*(q, \epsilon) = q$,
2. $\delta^*(q, ws) = \delta(\delta^*(q, w), s)$. $\Diamond$

Exemple 2.1.2 Soit l'automate à états fini correspondant au sélecteur de l'exemple 2.1.1 (2) dont la fonction de transition δ vaut:

$$\begin{aligned}
\delta(\mathrm{OM}, \mathrm{droite}) &= \mathrm{UC} \\
\delta(\mathrm{UC}, \mathrm{gauche}) &= \mathrm{OM} \\
\delta(\mathrm{OM}, \mathrm{gauche}) &= \mathrm{OL} \\
\delta(\mathrm{OL}, \mathrm{droite}) &= \mathrm{OM}
\end{aligned}$$

Les trois applications suivantes de la fonction δ^* sont dont bien correctes:

$$\begin{aligned}
\delta^*(\mathrm{OM}, \mathrm{gauche} \cdot \mathrm{droite}) &= \mathrm{OM} \\
\delta^*(\mathrm{OL}, \mathrm{droite} \cdot \mathrm{droite} \cdot \mathrm{gauche}) &= \mathrm{OM} \\
\delta^*(\mathrm{OM}, \mathrm{droite}) &= \mathrm{UC}
\end{aligned}$$

2.1.2 Accepteur de langages

Les automates finis peuvent être considérés comme des modèles de langages, mais aussi comme une machine réalisant un certain travail; c'est le cas des automates finis dits *accepteurs* de langages ou *reconnaisseurs* de langages. Ce type d'automates correspond à une machine "lisant" au fur et à mesure une suite de symboles formant un mot pour progresser dans l'ensemble des états. Ces automates sont munis d'un ensemble d'états appelés *états finaux*; un mot est *accepté* ou *reconnu* par un automate si ce dernier atteint un état final. L'ensemble des mots acceptés par une telle machine forme donc un langage qui possède la caractéristique d'être *régulier*.

Définition 2.1.3 *Accepteur de langage*
Un *accepteur de langage* est un AF muni d'un ensemble d'états finaux F, c'est donc un 5-tuple $\langle Q, \Sigma, \delta, q_0, F\rangle$ où

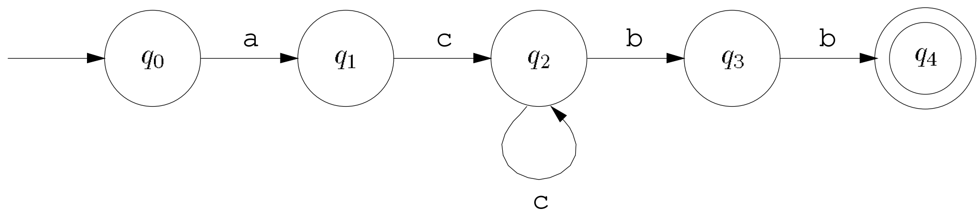
- Q est l'ensemble (fini) des états de l'automate,
- Σ est l'ensemble fini des symboles,
- $\delta : Q \times \Sigma \to Q$ est la fonction de transition,
- $q_0 \in Q$ est l'état initial,
- F est l'ensemble des états finaux avec $F \subseteq Q$. $\diamond$

Définition 2.1.4 *Acceptation*
Un mot $w \in \Sigma^*$ est *accepté* ou *reconnu* par un AF $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ si et seulement si $\delta^*(q_0, w) \in F$. Le *langage accepté* ou *reconnu* par un AF $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ est noté $L(M)$; c'est l'ensemble de tous les mots acceptés par M. $\diamond$

Définition 2.1.5 *Langage régulier*
On appelle langage *régulier* tout langage accepté par un automate à états fini accepteur. $\diamond$

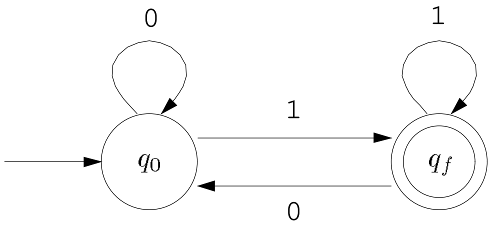
Exemples 2.1.3

1. Le langage L de l'exemple du § 1.1 est accepté par l'automate suivant:



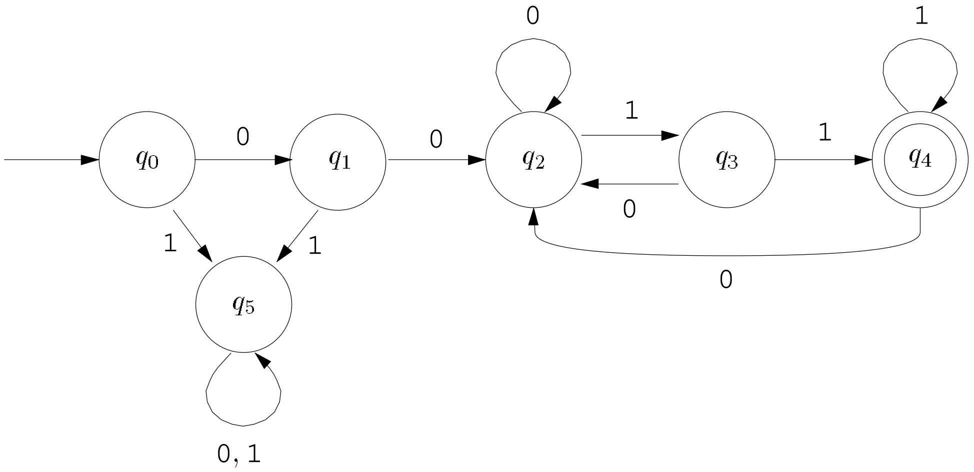
   Graphiquement on distingue les états finaux des états non finaux par le double cercle.

2. Soit l'automate accepteur A défini formellement par $A = \langle Q, \Sigma, \delta, q_0, F\rangle$ où $Q = \{q_0, q_f\}$, $\Sigma = \{0, 1\}$, $F = \{q_f\}$ et
$$\begin{aligned}\delta(q_0, 0) &= q_0\\ \delta(q_0, 1) &= q_f\\ \delta(q_f, 0) &= q_0\\ \delta(q_f, 1) &= q_f\end{aligned}$$
et graphiquement par



   En considérant que les mots formés sur l'alphabet $\Sigma = \{0, 1\}$ dénotent des nombres binaires, le langage accepté par A est $L(A) = \{x \mid x \in \{0, 1\}^* \text{ et } x \text{ est impair}\}$.

3. Soit l'accepteur B représenté graphiquement



le langage accepté par B est $L(B) = \{0^n x 1^m \mid n \geq 2, m \geq 2 \text{ et } x \in \{0,1\}^*\}$.

2.1.3 Configurations et dérivations

Nous avons défini ce que représentait un mot accepté en terme de fonction de transition (un mot w est accepté par un automate M si et seulement si $\delta^*(q_0, w) \in F$). Afin de préciser le “chemin” parcouru par un automate on introduit la notion de *configuration*. Une configuration d'un automate est un couple $(q, w) \in Q \times \Sigma^*$ où q représente l'état courant dans lequel se trouve l'automate et w est le mot en entrée qui reste à traiter. Une configuration représente donc l'état proprement dit d'un automate, à ne pas confondre avec l'état q ou les états de l'ensemble Q qui, quant à eux, représentent les états du système que décrit l'automate.

Au fur et à mesure de la lecture, l'automate “passe” d'une configuration à l'autre. Ce passage s'appelle aussi *dérivation* et représente une (ou plusieurs) étapes d'exécution de l'automate.

Définition 2.1.6 *Dérivation en une étape*
Pour un automate M, la *dérivation en une étape* de la configuration (q, w) à la configuration (q', w') est notée par $(q, w) \vdash_M (q', w')$ si $w = sw'$ et $q' = \delta(q, s)$. ◊

Définition 2.1.7 *Dérivation en plusieurs étapes*
Pour un automate M, la *dérivation en plusieurs étapes* de la configuration (q, w) à la configuration (q', w') est notée par $(q, w) \vdash_M^* (q', w')$ si $w = w'' \cdot w'$ et $q' = \delta^*(q, w'')$. ◊

Cette notation permet d'exprimer formellement l'exécution d'un automate sur un mot w à partir d'un état q :

$$(q, w) \vdash (q_1, w_1) \vdash (q_2, w_2) \vdash \cdots \vdash (q_n, w_n)$$

On dit aussi qu'à partir d'un état q et pour une entrée w, l'état q_n est *accessible* ou *atteignable*. Un mot w est accepté par un automate M si $(q_0, w) \vdash_M^* (q_n, \epsilon)$ avec $q_n \in F$. Le langage accepté par M peut être exprimé par

$$L(M) = \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (q, \epsilon) \text{ avec } q \in F\}.$$

2.1.4 Traducteur de langages

Un automate peut être considéré comme une machine acceptant un langage mais aussi comme une machine traduisant chaque symbole lu par un ou plusieurs autres symboles. Ce type d'automate délivre donc en sortie une information à partir d'une suite de symboles en entrée.

Définition 2.1.8 *Traducteur de langage*
Un *traducteur de langage* est un 6-tuple $\langle Q, \Sigma, \delta, q_0, O, \eta \rangle$ où

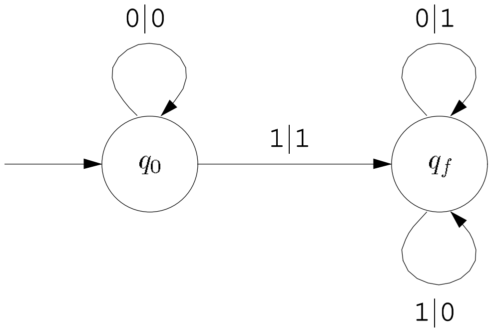
- Q est l'ensemble (fini) des états de l'automate,
- Σ est l'ensemble fini des symboles,
- $\delta : Q \times \Sigma \rightarrow Q$ est la fonction de transition,
- q_0 est l'état initial,
- O est l'ensemble des symboles de sortie,
- $\eta : Q \times \Sigma \rightarrow O^*$ est la fonction de sortie.

Pour un état $q \in Q$, un mot $w \in \Sigma^*$ et un symbole $s \in \Sigma$, l'extension $\eta^* : Q \times \Sigma^* \rightarrow O^*$ de la fonction η est définie par:

1. $\eta^*(q, \epsilon) = \epsilon$,
2. $\eta^*(q, ws) = \eta^*(q, w) \cdot \eta(\delta^*(q, w), s)$. $\diamondsuit$

Graphiquement on symbolise la traduction par $s|o$ où $s \in \Sigma$ et $o \in O^*$, s représente donc le symbole lu et o le mot de sortie.

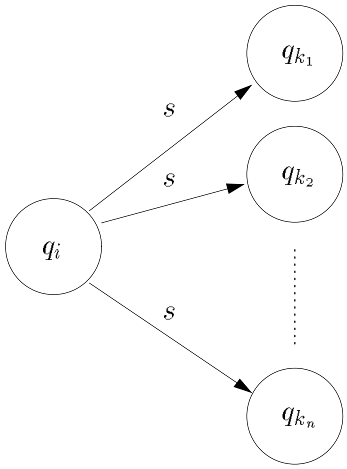
Exemple 2.1.4 Le traducteur suivant traduit un mot binaire en son complément à deux pour autant que la lecture commence par le bit de poids le moins significatif:



2.1.5 Automates à états finis non déterministes

Les automates définis précédemment possèdent une fonction de transition de $Q \times \Sigma$ vers Q, c'est-à-dire qu'à partir d'un état et d'un symbole, un unique état peut être "atteint". On appelle ce type d'automates des automates *déterministes* (AFD). Cependant, il existe des modèles d'automates dits automates à états finis *non déterministes* (AFN) qui possèdent la particularité suivante: à partir d'un état et pour un symbole donné, plusieurs états peuvent être atteints; le choix entre les différents "chemins" est équiprobable, ce qui signifie que la probabilité d'emprunter une voie plutôt qu'une autre est identique.

Graphiquement ce non déterminisme est représenté par plusieurs flèches partant d'un état pour un même symbole.



La fonction de transition d'un automate non déterministe ne possède, par conséquent, pas le même co-domaine que celle d'un automate déterministe. En effet, ce n'est plus un seul état qui peut être atteint mais un ensemble d'états. Le co-domaine de la fonction de transition d'un automate non déterministe sera donc l'ensemble des parties de l'ensemble des états Q. On note cet ensemble $\mathbf{P}(Q)$.

Définition 2.1.9 *Automate non déterministe*
Un *automate à états fini accepteur non déterministe* est un 5-tuple $\langle Q, \Sigma, \delta, q_0, F\rangle$ où

- Q est l'ensemble (fini) des états de l'automate,
- Σ est l'ensemble fini des symboles,
- $\delta : Q \times \Sigma \to \mathbf{P}(Q)$ est la fonction de transition,
- q_0 est l'état initial,
- F est l'ensemble des états finaux avec $F \subseteq Q$,

Pour un état $q \in Q$, un mot $w \in \Sigma^*$ et un symbole $s \in \Sigma$, l'extension $\delta^* : Q \times \Sigma^* \to \mathbf{P}(Q)$ de la fonction δ est définie par:

a) $\delta^*(q, \epsilon) = \{q\}$,

b) $\delta^*(q, ws) = \bigcup_{q' \in \delta^*(q,w)} \delta(q', s)$.

Il est aussi utile d'étendre le domaine de δ, δ^* à $\mathbf{P}(Q)$ comme ceci

$$\begin{aligned} \delta \quad &: \quad \mathbf{P}(Q) \times \Sigma \to \mathbf{P}(Q) \\ &\quad\;\; (p, s) \mapsto \cup_{p \in P} \delta(p, s) \end{aligned} \qquad\qquad \begin{aligned} \delta^* \quad &: \quad \mathbf{P}(Q) \times \Sigma^* \to \mathbf{P}(Q) \\ &\quad\;\; (p, w) \mapsto \cup_{p \in P} \delta^*(p, w) \end{aligned}$$

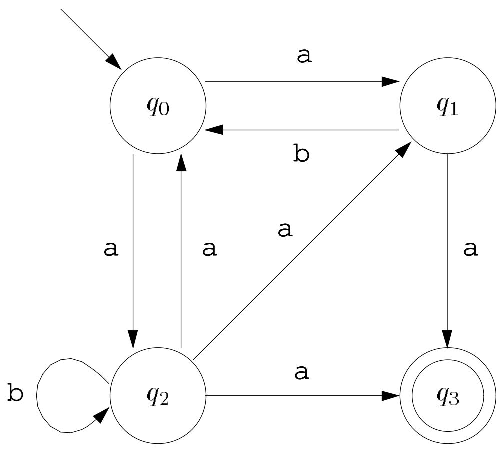
Un mot $w \in \Sigma^*$ est *accepté* si et seulement si $\delta^*(q_0, w) \cap F \neq \emptyset$. Le *langage accepté* par un automate non déterministe est l'ensemble des mots acceptés par l'automate. $\Diamond$

Remarques :

1. Un mot est accepté par un automate non déterministe s'il existe au moins un chemin allant de l'état initial à l'état final.

2. Les transitions d'un état à l'autre ont été définies par une fonction dont le co-domaine est l'ensemble des parties de Q pour exprimer tous les états "suivants". On peut aussi définir les transitions à l'aide d'une simple relation de transition $\Delta \subseteq (Q \times \Sigma \times Q)$.

Exemple 2.1.5 L'automate non déterministe qui suit accepte le mot `abaa` puisqu'il existe au moins un chemin de q_0 à q_3 (il en existe en réalité trois).



2.1.6 Équivalence entre AFD et AFN

Dans cette section on s'intéresse à déterminer si une notion d'équivalence existe entre les automates déterministes et les automates non déterministes. En premier lieu il faut définir cette notion d'équivalence.

Définition 2.1.10 *Équivalence d'automates*
Deux automates à états finis M_1 et M_2 sont *équivalents* si et seulement s'ils acceptent le même langage, i.e. $L(M_1) = L(M_2)$. $\Diamond$

Théorème 2.1.1 *Équivalence AFD-AFN*
A tout automate à états fini non déterministe correspond un automate à états fini déterministe équivalent et réciproquement.

Preuve
Montrer qu'à tout AF déterministe correspond un AF non déterministe est évident puisque les AF déterministes sont des cas particuliers des AF non déterministes.

La réciproque n'est pas aussi simple. Pour montrer qu'à tout AFN correspond un AFD, choisissons $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ un AFN et construisons un AFD $M' = \langle Q', \Sigma, \delta', q'_0, F'\rangle$ comme suit:

1. $Q' = \mathbf{P}(Q)$, un élément de Q' sera noté $[q_1, q_2, \ldots, q_i]$ où $q_1, \ldots, q_i$ appartiennent à Q. Cet élément dénote un unique état de la machine déterministe correspondant à un ensemble d'états de la machine non déterministe,

2. F' est l'ensemble des membres de Q' comportant au moins un état final de M,

3. $q'_0 = [q_0]$,
4. $\delta'([q_1, \ldots, q_i], s) = [p_1, \ldots, p_j] \iff \delta(\{q_1, \ldots, q_i\}, s) = \{p_1, \ldots, p_j\}$.

Il reste à montrer que tout mot w accepté par M l'est aussi par M', et que s'il n'est pas accepté par M alors il ne l'est pas non plus par M'. Ceci revient à montrer (par induction sur la longueur de w) que

$$\delta'^*(q'_0, w) = [q_1, \ldots, q_i] \iff \delta^*(q_0, w) = \{q_1, \ldots, q_i\}.$$

Cas de base: trivial pour $|w| = 0$ car $q'_0 = [q_0]$ et $w = \epsilon$.
Induction: supposons la formule ci-dessus vraie pour les mots de longueur $\leq m$. Soit $s \in \Sigma$ et ws une chaîne de longueur $m+1$. Par définition de δ'^*

$$\delta'^*(q'_0, ws) = \delta'(\delta'^*(q'_0, w), s).$$

Par hypothèse d'induction,

$$\delta'^*(q'_0, w) = [p_1, \ldots, p_j] \iff \delta^*(q_0, w) = \{p_1, \ldots, p_j\}.$$

Mais par définition de δ',

$$\delta'([p_1, \ldots, p_j], s) = [r_1, \ldots, r_k] \iff \delta(\{p_1, \ldots, p_j\}, s) = \{r_1, \ldots, r_k\}.$$

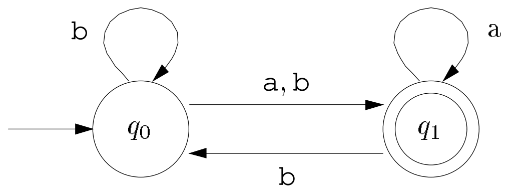
Par conséquent,

$$\delta'^*(q'_0, ws) = [r_1, \ldots, r_k] \iff \delta^*(q'_0, ws) = \{r_1, \ldots, r_k\},$$

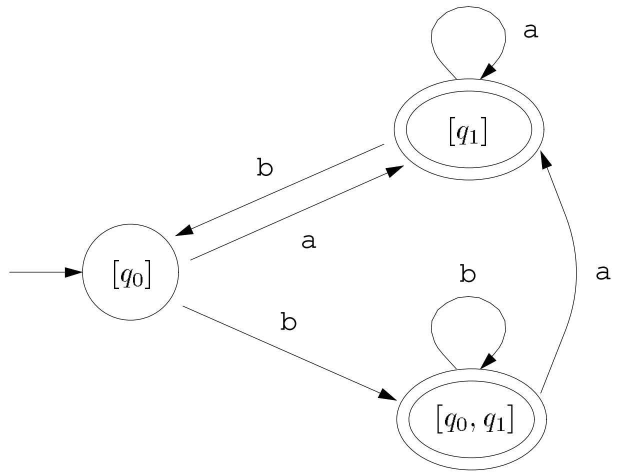
ce qui confirme notre hypothèse d'induction.

Finalement, il faut ajouter le fait que $\delta'^*(q'_0, w) \in F'$ uniquement lorsque $\delta^*(q_0, w)$ contient un état de Q qui appartient à F. Ainsi $L(M) = L(M')$. □

Exemple 2.1.6 Soit l'automate non déterministe qui suit:



L'élimination du non déterminisme consiste à construire un automate déterministe selon la méthode décrite dans la preuve du théorème 2.1.1 ce qui donne:



2.1.7 Automates avec transitions vides

Dans le modèle précédent (automates non déterministes) les transitions sont causées par un symbole appartenant à un alphabet. Nous allons étendre ce modèle en autorisant les transitions causées par le mot vide. On appelle ces transitions des transitions *spontanées* ou *vides*. Lorsque ce modèle étendu sera défini nous nous intéresserons, à nouveau, à la notion d'équivalence entre automates non déterministes avec "transitions vides" et automates non déterministes sans transition vide.

Définition 2.1.11 *Automate avec transitions vides*
Un automate à états fini avec *transitions vides* se définit de la même manière qu'un automate sans transition vide exception faite de son alphabet qui comporte le symbole vide ϵ. ◊

Définition 2.1.12 ϵ-Fermeture
Pour un état q appartenant à l'ensemble d'états Q d'une machine M, ϵ-Fermeture(q) est l'ensemble de tous les états de M atteignables depuis l'état q par une suite de transitions vides.

$$\epsilon\text{-Fermeture}(q) = \{p \in Q \mid (q,w) \vdash_M^* (p,w)\},$$

$$\epsilon\text{-Fermeture}(P) = \bigcup_{p \in P} \epsilon\text{-Fermeture}(p)$$

où P dénote un ensemble d'états. ◊

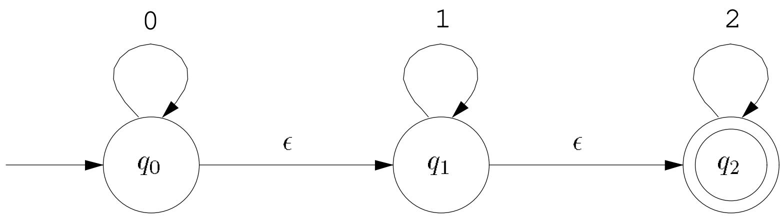
Définition 2.1.13 δ^*
Pour un AFN avec transitions vides on définit une nouvelle fonction δ^*:

1. $\delta^*(q,\epsilon) = \epsilon$-Fermeture$(q)$,
2. $\forall w \in \Sigma^*$ et $s \in \Sigma$, $\delta^*(q, ws) = \epsilon$-Fermeture$(P)$, où $P = \{p \mid$ pour un $r \in \delta^*(q,w), p \in \delta(r,s)\}$.
   On étend δ et δ^* à un ensemble d'états:
3. $\delta(R,s) = \cup_{q \in R}\delta(q,s)$,
4. $\delta^*(R,w) = \cup_{q \in R}\delta^*(q,w)$. ◊

2.1.7.1 Suppression des transitions vides

La méthode d'élimination des transitions vides d'un automate non déterministe consiste à calculer la ϵ-Fermeture pour tous les états de l'automate ce qui définit simplement la fonction δ^*. Une fois cette opération effectuée, il faut ajouter l'état initial à l'ensemble des états finaux dans le cas où le mot vide est accepté par l'automate avec transitions vides.

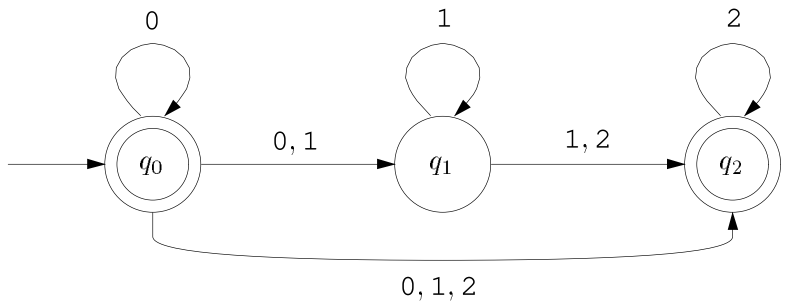
Exemple 2.1.7 Soit l'automate avec transitions vides:



Voici la nouvelle fonction de transition δ^* :

δ^*	0	1	2
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1	$\varnothing$	$\{q_1, q_2\}$	$\{q_2\}$
q_2	$\varnothing$	$\varnothing$	$\{q_2\}$

il faut ajouter q_0 à l'ensemble des états finaux, ce qui donne la représentation graphique du nouvel automate :



Théorème 2.1.2
Si un langage L est accepté par un AF non déterministe avec transitions vides, il l'est aussi par un AF non déterministe sans transitions vides.

Preuve
Prenons $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ un AFN avec transitions vides et construisons un AFN $M' = \langle Q, \Sigma, \delta', q_0, F' \rangle$ sans transitions vides, où

$$F' = \begin{cases} F \cup \{q_0\} \text{ si } \epsilon\text{-Fermeture}(q_0) \text{ contient un état de } F, \\ F \text{ sinon} \end{cases}$$

et $\delta'(q, s) = \delta^*(q, s)$ avec $q \in Q$, $s \in \Sigma$ et δ^* de la définition 2.1.13.

Il reste à montrer par induction sur $|x|$ avec $x \in \Sigma$ que $\delta'^*(q_0, x) = \delta^*(q_0, x)$.
<u>Cas de base</u>: $|x| = 1$, trivial par définition de δ'.
<u>Induction</u>: pour $|x| > 1$, on pose $x = ws$ avec $w \in \Sigma^*$ et $s \in \Sigma$, alors

$$\delta'^*(q_0, ws) = \delta'(\delta'^*(q_0, w), s).$$

Par hypothèse d'induction on a $\delta'^*(q_0, w) = \delta^*(q_0, w)$. En posant $P = \delta^*(q_0, w)$, l'égalité précédente devient

$$\delta'(P, s) = \delta^*(q_0, ws)$$

mais

$$\delta'(P, s) = \bigcup_{q \in P} \delta'(q, s) = \bigcup_{q \in P} \delta^*(q, s).$$

comme $P = \delta^*(q_0, w)$, on a

$$\bigcup_{q \in P} \delta^*(q, s) = \delta^*(q_0, ws)$$

et par conséquent

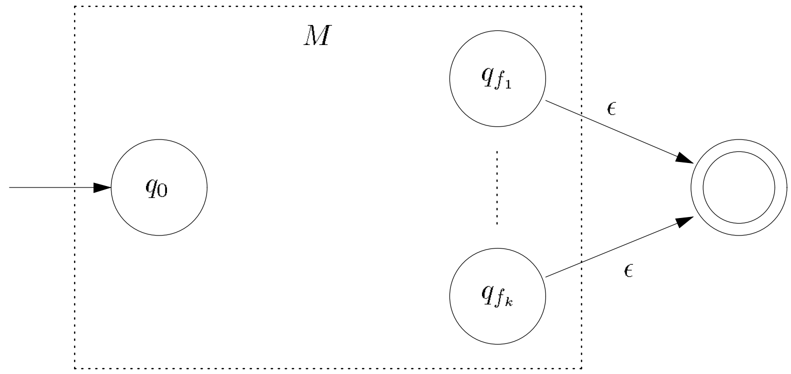
$$\delta'^*(q_0, ws) = \delta^*(q_0, ws).$$

Pour que la preuve soit complète il faut montrer que $\delta'^*(q_0, x)$ contient bien un état de F' si et seulement si $\delta^*(q_0, x)$ contient un état de F. (Laissé en exercice). $\square$

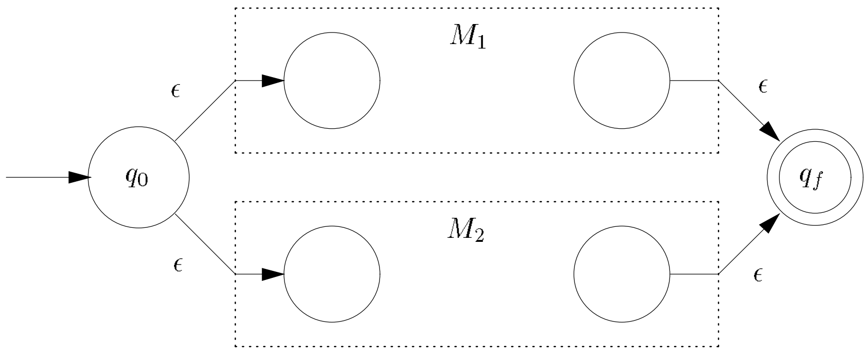
2.1.8 Propriétés des langages réguliers

La classe des langages réguliers, c'est-à-dire la classe des langages acceptés par des AF, possède diverses propriétés que nous allons exposer ci-dessous:

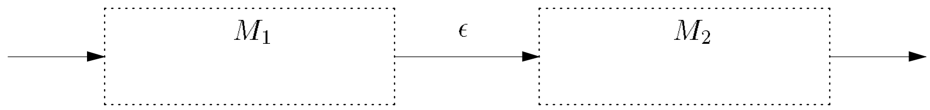
1. Soit M, l'AF possédant plusieurs états finaux, on peut construire un AF équivalent avec transitions vides ne possédant qu'un seul état final de la manière suivante :



2. l'union de deux langages réguliers est un langage régulier. Soit M_1 et M_2 deux AF, l'union des deux langages acceptés par les deux machines est un langage régulier puisque l'on peut construire un automate acceptant cette union comme suit :

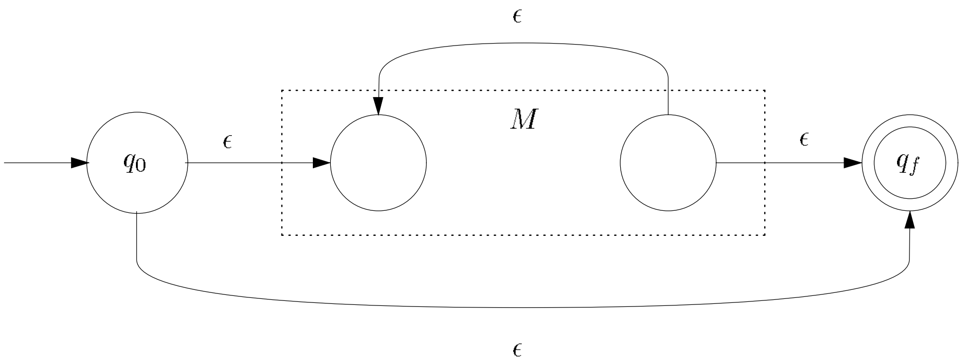


3. la concaténation de deux langages réguliers est un langage régulier car :



4. la complémentation d'un langage régulier L ($\overline{L} = \Sigma^* \setminus L$) est aussi un langage régulier. Pour construire un automate acceptant $\overline{L}$, il suffit d'intervertir le rôle des états finaux avec celui des états non finaux en faisant attention que la fonction de transition soit partout définie.

5. l'itération de deux langages réguliers est un langage régulier car :



2.2 Grammaires

Cette section introduit la notion de grammaire. Une grammaire est un formalisme permettant, comme les automates, de spécifier des langages. Quatre types importants de grammaire nous intéresseront. Ces types de grammaire correspondent chacun à un type de machine que nous avons ou allons étudier. Définissons tout d'abord ce qu'est une grammaire.

Intuitivement, une grammaire est un système de règles permettant de "réécrire" un terme en un autre terme. Les règles de la grammaire sont composées d'une partie gauche et d'une partie droite reliées par une flèche, où chacune est elle-même composée de symboles *terminaux* et de symboles *non terminaux*. Les symboles terminaux correspondent aux symboles qui forment les mots engendrés par la grammaire alors que les symboles non terminaux servent principalement à la "réécriture". Par terme on entend une séquence de ces deux types de symboles. La réécriture d'un terme en un autre terme consiste à appliquer une règle au terme original, c'est-à-dire à substituer la partie gauche de la règle qui apparaît dans le terme par la partie droite de la règle, et obtenir ainsi le nouveau terme. En commençant la réécriture à partir d'une règle de départ particulière appelée *axiome* et en appliquant successivement les règles, on finit (pas obligatoirement) par obtenir un terme pour lequel aucune règle ne s'applique. Un terme ainsi obtenu ne comportant aucun symbole non terminal est donc un mot défini sur l'alphabet des symboles terminaux. L'ensemble de ces mots forment alors un langage que l'on appelle *langage engendré* par la grammaire.

Définition 2.2.1 *Grammaire*
Une *grammaire* (définition générale) est un 4-tuple $G = \langle V_T, V_N, P, S\rangle$ où

- V_T est l'ensemble des symboles terminaux ou l'alphabet formant les mots du langage,
- V_N est l'alphabet des symboles non terminaux, avec $V_T \cap V_N = \varnothing$,
- P est l'ensemble des règles grammaticales, ou productions, de la forme $\alpha \to \beta$ où α et β sont des séquences de terminaux ou de non terminaux et où α contient au moins un non terminal,
- S est l'*axiome* ou le *symbole initial*, un non terminal particulier qui correspond à la racine de la grammaire. $\Diamond$

Définition 2.2.2 *Dérivation*
Soient une grammaire $G = \langle V_T, V_N, P, S\rangle$ et $A \to B$ une production de P. La séquence de terminaux et non terminaux $\alpha A\beta \in (V_T \cup V_N)^*$ peut être *dérivée* en une séquence $\alpha B\beta$ en *appliquant* la

production, c'est-à-dire en substituant A par B. On appelle *dérivation en une étape* le passage d'une séquence α à une séquence β par l'application d'une production de la grammaire G. On note cette dérivation $\alpha \Rightarrow_G \beta$. On appelle *dérivation en plusieurs étapes* une succession de dérivations et l'on note cette dérivation $\alpha \Rightarrow_G^* \beta$. $\Diamond$

Définition 2.2.3 *Langage engendré*
Le *langage engendré* par une grammaire $G = \langle V_T, V_N, P, S\rangle$, noté $L(G)$ est l'ensemble des mots formés uniquement de terminaux que l'on peut dériver de l'axiome S. $L(G) = \{w \mid S \Rightarrow_G^* w, w \in V_T^*\}$. $\Diamond$

Notation : Les symboles terminaux sont habituellement représentés par des lettres minuscules et les symboles non terminaux par des lettres majuscules.

Exemples 2.2.1

1. La grammaire $G = \langle\{\mathtt{a},\mathtt{b}\}, \{S, A\}, P, S\rangle$ où $P = \{S \to \mathtt{bb}A, A \to \mathtt{aa}A, A \to \mathtt{aa}\}$ engendre le langage $L(G) = \{\mathtt{bb}\} \cdot \{\mathtt{aa}\}^n$ où $n \geq 1$,

2. la grammaire $G = \langle\{\mathtt{0},\mathtt{1}\}, \{S\}, P, S\rangle$ où $P : \{S \to \mathtt{1}S, S \to \mathtt{0}S, S \to \mathtt{1}\}$ engendre le langage $L(G) = \{\mathtt{0},\mathtt{1}\}^* \cdot \{\mathtt{1}\}$,

3. la grammaire $G = \langle\{\mathtt{a},\mathtt{b}\}, \{S\}, P, S\rangle$ où $P = \{S \to \mathtt{a}S\mathtt{b}, S \to \epsilon\}$ engendre le langage $L(G) = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$,

4. le langage $L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 1\}$ est engendré par la grammaire

$$G = \langle\{\mathtt{a},\mathtt{b},\mathtt{c}\}, \{S, A, B, C, D\}, P, S\rangle$$

   où $P = \{S \to \mathtt{a}ACD, A \to \mathtt{a}AC, A \to \epsilon, B \to \mathtt{b}, CD \to BD\mathtt{c}, CB \to BC, D \to \epsilon\}$

2.2.1 Grammaires régulières

La définition ci-dessus des grammaires n'impose aucune restriction sur le profil des règles exception faite de la partie gauche de chaque règle qui doit comporter au moins un non terminal.

Définition 2.2.4 *Grammaire régulière*
On appelle *régulières à droite* les grammaires dont toutes les productions de la forme $\alpha \to \beta$ possèdent les restrictions suivantes :

1. $|\alpha| = 1$,

2. $\beta = aB$, $\beta = a$ ou $\beta = \epsilon$ avec $B \in V_N$ et $a \in V_T$.

Une grammaire est *régulière à gauche* si toutes les productions ont la forme :

1. $|\alpha| = 1$,

2. $\beta = Ba$, $\beta = a$ ou $\beta = \epsilon$ avec $B \in V_N$ et $a \in V_T$.

On appelle aussi simplement *grammaire régulière* toute grammaire régulière à droite ou à gauche. $\Diamond$

2.2.2 Grammaires régulières et automates à états finis

Théorème 2.2.1
Si L est accepté par un AF alors L est engendré par une grammaire régulière.

Preuve
Soit M l'AFD acceptant L avec $M = \langle Q, \Sigma, \delta, q_0, F\rangle$. Montrer que L est engendré par une grammaire régulière (à droite) G revient à construire cette grammaire puis à montrer que tout mot accepté par M est engendré par G et vice versa.

La grammaire G est construite comme suit: $G = \langle \Sigma, Q, P, q_0\rangle$, où P est l'ensemble des productions de la forme $p \to aq$ si $\delta(p, a) = q$ et $p \to \epsilon$ pour tout $p \in F$ avec $a \in \Sigma$ et $p, q \in Q$.

Montrons par induction sur la longueur des dérivations que $\delta^*(p, w) = q \iff p \Rightarrow_G^* wq$. Par définition de G, la propriété est vraie pour une longueur de 1. En supposant vraie $\delta^*(p, w) = q \iff p \Rightarrow_G^n wq$, on a que $\delta(\delta^*(p, w), a) = q' \iff p \Rightarrow_G^n wq \Rightarrow_G waq'$ et donc $\delta^*(p, wa) = q' \iff p \Rightarrow_G^{n+1} waq'$.

Finalement, tout mot w accepté par M est engendré par G. En effet, on a $\delta^*(q_0, w) = p \quad (p \in F) \implies q_0 \Rightarrow_G^* wp$; puisque p est final, il existe la production $p \to \epsilon$ et donc $q_0 \Rightarrow_G^* w$. De même, tout mot w engendré par G est accepté par M. En effet, il existe une production de la forme $p \to \epsilon$ qui permette d'écrire $q_0 \Rightarrow_G^* wp \Rightarrow_G w \iff \delta(\delta^*(q_0, w), \epsilon) = p$ avec $p \in F$. On a donc bien

$$\delta^*(q_0, w) = p \text{ avec } p \in F \iff q_0 \Rightarrow_G^* w$$

et l'on conclu que $L(M) = L(G)$. □

Théorème 2.2.2
Si le langage L est engendré par une grammaire régulière, alors L est accepté par un AF.

Preuve
Supposons que $L = L(G)$ pour une grammaire régulière à droite $G = \langle V_T, V_N, P, S\rangle$. On construit un AFN avec transitions vides, $M = \langle V_N \cup \{\mathit{final}\}, V_T, \delta, S, \{\mathit{final}\}\rangle$ où

1. $\delta(A, a) = B$ si $A \to aB$ appartient à P,
2. $\delta(A, a) = \mathit{final}$ si $A \to a$ appartient à P,
3. $\delta(A, \epsilon) = \mathit{final}$ si $A \to \epsilon$ appartient à P

pour $A, B \in V_N$ et $a \in V_T$.

Il reste à vérifier qu'un mot est engendré par G si et seulement s'il est accepté par M. Par induction sur la longueur des dérivations on montre facilement que $A \Rightarrow_G^* wB \iff \delta^*(A, w) = B$. Finalement on particularise cette propriété pour conclure que

$$S \Rightarrow_G^* w \iff \delta^*(S, w) = \mathit{final}$$

et donc que $L(G) = L(M)$. □

2.3 Expressions régulières

Outre la notation emsembliste classique, nous avons déjà vu deux formalismes permettant de spécifier des langages réguliers, il s'agit des automates à états finis et des grammaires régulières. Dans cette section nous allons aborder une troisième manière de spécifier un langage régulier: les *expressions régulières (ER)*. Nous allons tout d'abord définir ce qu'est une expression régulière puis énoncer deux théorèmes exprimant l'équivalence entre ce nouveau formalisme et les deux vus précédemment.

Définition 2.3.1 *Expression régulière*
Soit un alphabet Σ, une *expression régulière* définie sur Σ ainsi que les ensembles qu'elle dénote sont définis récursivement de la manière suivante :

1. $\varnothing$ est une expression régulière dénotant l'ensemble vide,
2. ϵ est une expression régulière dénotant l'ensemble $\{\epsilon\}$,
3. pour tout $a \in \Sigma$, a est une expression régulière dénotant l'ensemble $\{a\}$,
4. si r et s sont des expressions régulières dénotant respectivement les ensembles R et S, alors $(r+s)$, (rs) et (r^*) sont des expressions régulières dénotant respectivement les ensembles $R \cup S$, $R \cdot S$ et R^*,
5. rien d'autre n'est considéré comme expression régulière.

Le langage que dénote une expression régulière r est noté $L(r)$. $\Diamond$

Exemples 2.3.1

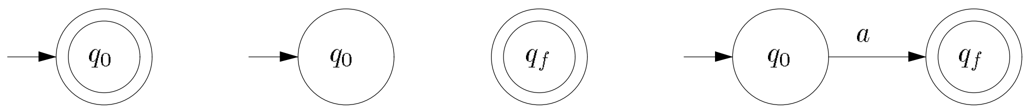
a) Soit l'alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ et l'expression régulière $r = (\mathtt{a}\,(\mathtt{a}+\mathtt{b})^*)$, $L(r) = \{\mathtt{a}\} \cdot \{\mathtt{a}, \mathtt{b}\}^*$.

b) L'expression régulière $r = (\mathtt{0}^*\mathtt{1})^*$ dénote l'ensemble $L(r) = \{\{\mathtt{0}\}^* \cdot \{\mathtt{1}\}\}^*$ ou bien $L(r) = \{x \mid x \in \{\mathtt{0}, \mathtt{1}\}^*$ et x représente un nombre impair$\}$.

Théorème 2.3.1
Soit r une expression régulière, il existe alors un automate A à états finis non déterministe avec des transitions vides tel que A accepte $L(r)$.

Preuve
Montrer qu'un automate à état finis A existe pour toute expression régulière r définie sur un alphabet Σ se fait par induction sur le nombre d'opérateurs i apparaissant dans l'expression régulière.

<u>Cas de base</u>: $i = 0$
Si r ne contient aucun opérateur alors r ne peut être que ϵ, $\varnothing$, ou un symbole $a \in \Sigma$; l'automate A acceptant r est respectivement:



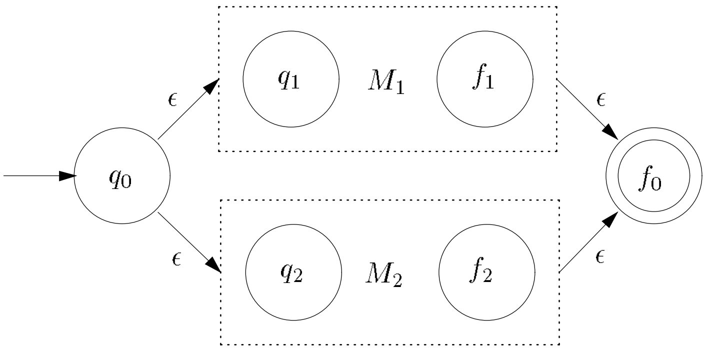
$r = \epsilon$ $r = \varnothing$ $r = a$

<u>Induction</u>: $i > 0$
Nous supposons que le théorème est vrai pour un nombre d'opérateurs inférieur à i. On distingue alors trois cas:

1. $r = r_1 + r_2$, où r_1 et r_2 possèdent un nombre d'opérateurs $< i$, à toutes deux correspond donc un automate à états finis non déterministe avec transitons vides, M_1 et M_2 respectivement.

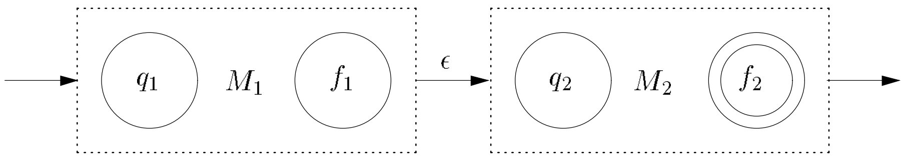
$M_1 = \langle Q_1, \Sigma_1, \delta_1, q_1, \{f_1\}\rangle$, $M_2 = \langle Q_2, \Sigma_2, \delta_2, q_2, \{f_2\}\rangle$ tel que $L(M_1) = L(r_1)$ et $L(M_2) = L(r_2)$. En utilisant les propriétés des AF nous pouvons construire l'automate A acceptant $L(r)$:
$A = \langle Q_1 \cup Q_2 \cup \{q_0, f_0\}, \Sigma_1 \cup \Sigma_2, \delta, q_0, \{f_0\}\rangle$ avec la fonction de transition définie comme ci-dessous:

$$\begin{aligned}
&\delta(q_0, \epsilon) = \{q_1, q_2\}\\
&\delta(q, a) = \delta_1(q, a) \text{ pour } q \in Q_1 \setminus \{f_1\} \text{ et } a \in \Sigma_1 \cup \{\epsilon\}\\
&\delta(q, a) = \delta_2(q, a) \text{ pour } q \in Q_2 \setminus \{f_2\} \text{ et } a \in \Sigma_2 \cup \{\epsilon\}\\
&\delta(f_1, \epsilon) = \delta_1(f_2, \epsilon) = \{f_0\}
\end{aligned}$$



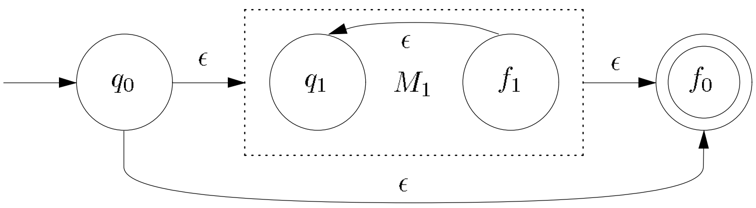
ainsi $L(A) = L(M_1) \cup L(M_2) = L(r)$.

2. $r = r_1 r_2$, en utilisant les mêmes hypothèses que dans le cas 1) nous pouvons construire l'automate A acceptant $L(r)$:
$A = \langle Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta, q_1, \{f_2\}\rangle$ où
$$\begin{aligned}
&\delta(q, a) = \delta_1(q, a) \text{ pour } q \in Q_1 \setminus \{f_1\} \text{ et } a \in \Sigma_1 \cup \{\epsilon\}\\
&\delta(q, a) = \delta_2(q, a) \text{ pour } q \in Q_2 \setminus \{f_2\} \text{ et } a \in \Sigma_2 \cup \{\epsilon\}\\
&\delta(f_1, \epsilon) = \{q_2\}
\end{aligned}$$



ainsi $L(A) = L(M_1) \cdot L(M_2) = L(r)$.

3. $r = {r_1}^*$, soit M_1 l'automate à états finis non déterministe avec transitions vides acceptant $L(r_1)$. La construction de l'automate A acceptant $L(r)$ se fait toujours selon les conditions énoncées dans le cas 1):
$A = \langle Q_1 \cup \{q_0, f_0\}, \Sigma_1, \delta, q_0, \{f_0\}\rangle$ où
$$\begin{aligned}
&\delta(q_0, \epsilon) = \delta(f_1, \epsilon) = \{q_1, f_0\}\\
&\delta(q, a) = \delta_1(q, a) \text{ pour } q \in Q_1 \setminus \{f_1\} \text{ et } a \in \Sigma_1 \cup \{\epsilon\}
\end{aligned}$$



ainsi $L(A) = L(M_1)^* = L(r)$. □

Théorème 2.3.2
Si le langage L est accepté par un AF déterministe, alors il existe une expression régulière r qui dénote ce langage L.

Preuve
Soit L un langage accepté par un AF déterministe $M = \langle\{q_1, \dots, q_n\}, \Sigma, \delta, q_1, F\rangle$, et soit R_{ij}^k l'ensemble de toutes les chaînes qui font passer l'automate de l'état q_i à l'état q_j sans passer par un état numéroté par une valeur plus grande que k, (i ou j peuvent être plus élevés que k), R_{ij}^k se définit récursivement comme suit:

$$R_{ij}^k = R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1} \cup R_{ij}^{k-1},$$

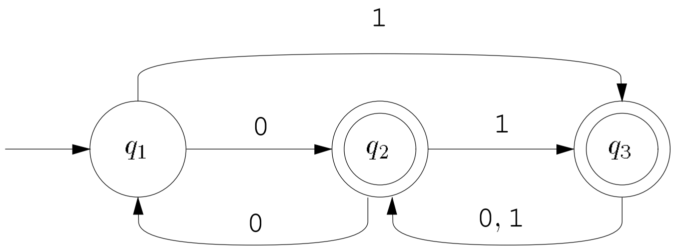
$$R_{ij}^0 = \begin{cases} \{a \mid \delta(q_i, a) = q_j\} & \text{si} \quad i \neq j, \\ \{a \mid \delta(q_i, a) = q_j\} \cup \{\epsilon\} & \text{si} \quad i = j. \end{cases}$$

A partir de ces données nous pouvons montrer par induction sur k qu'il existe une expression régulière r_{ij}^k qui dénote le langage R_{ij}^k:

<u>Cas de base</u>: (k=0), l'ensemble R_{ij}^0 est un ensemble fini de chaînes lesquelles sont soit ϵ soit un unique symbole. L'expression régulière dénotant cet ensemble s'écrit alors: $r_{ij}^0 = a_1 + a_2 + \cdots + a_p$ si $i \neq j$ ou $r_{ij}^0 = a_1 + a_2 + \cdots + a_p + \epsilon$ si $i = j$ avec $\{a_1, a_2, \dots a_p\}$ étant l'ensemble de tous les symboles a tel que $\delta(q_i, a) = q_j$ ou $\varnothing$ (ϵ si $i = j$) si aucun a existe.

<u>Induction</u>: par hypothèse d'induction, pour chaque l et m il existe une expression régulière r_{lm}^{k-1} tel que $L(r_{lm}^{k-1}) = R_{lm}^{k-1}$, ainsi $r_{ij}^k = r_{ik}^{k-1}(r_{kk}^{k-1})^* r_{kj}^{k-1} + r_{ij}^{k-1}$ est régulière. Nous remarquons alors que $L(M) = \cup_{q_j \in F} R_{1j}^n$ puisque R_{1j}^n dénote les séquences de tous les chemins allant de q_i à q_j et, par conséquent $L(M)$ est dénoté par l'expression régulière $r_{1j_1}^n + r_{1j_2}^n + \cdots + r_{1j_p}^n$, où $F = \{q_{j_1}, q_{j_2}, \dots, q_{j_p}\}$. □

Exemple 2.3.2 Soit l'automate $M = \langle\{q_1, q_2, q_3\}, \{0, 1\}, \delta, q_1, \{q_2, q_3\}\rangle$:



Le tableau suivant donne les valeurs de r_{ij}^k pour tout i,j et $k = 0, 1, 2$. Les valeurs du tableau sont calculées en utilisant la formule: $r_{ij}^k = r_{ik}^{k-1}(r_{kk}^{k-1})^* r_{kj}^{k-1} + r_{ij}^{k-1}$.

	$k=0$	$k=1$	$k=2$
${r_{11}}^k$	ϵ	ϵ	$(00)^*$
${r_{12}}^k$	0	0	$0(00)^*$
${r_{13}}^k$	1	1	0^*1
${r_{21}}^k$	0	0	$0(00)^*$
${r_{22}}^k$	ϵ	$\epsilon + 00$	$(00)^*$
${r_{23}}^k$	1	$1 + 01$	0^*1
${r_{31}}^k$	$\varnothing$	$\varnothing$	$(0+1)(00)^*0$
${r_{32}}^k$	$0+1$	$0+1$	$(0+1)(00)^*$
${r_{33}}^k$	ϵ	ϵ	$\epsilon + (0+1)0^*1$

Ainsi nous pouvons construire l'expression régulière dénotant $L(M)$: $r^3_{12} + r^3_{13}$

$$\begin{aligned} r^3_{12} &= r^2_{13}(r^2_{33})^* r^2_{32} + r^2_{12} \\ &= 0^*1(\epsilon + (0+1)0^*1)^*(0+1)(00)^* + 0(00)^* \\ &= 0^*1((0+1)0^*1)^*(0+1)(00)^* + 0(00)^* \end{aligned}$$

$$\begin{aligned} r^3_{13} &= r^2_{13}(r^2_{33})^* r^2_{33} + r^2_{13} \\ &= 0^*1(\epsilon + (0+1)0^*1)^*(\epsilon + (0+1)0^*1) + 0^*1 \\ &= 0^*1((0+1)0^*1)^* \end{aligned}$$

$$r^3_{12} + r^3_{13} = 0^*1((0+1)0^*1)^*(\epsilon + (0+1)(00)^*) + 0(00)^*.$$

2.4 Lemme de pompage

Le lemme qui suit, appelé *lemme de pompage*, affirme que lorsqu'un langage est régulier, il est possible de trouver, pour tout mot w du langage, au moins un découpage $w = xyz$ tel que si y est élevé à une puissance $i \geq 0$, alors xy^iz appartient aussi au langage.

Lemme 2.4.1 *Pompage*
Soit un langage régulier L infini. Il existe alors une constante n telle que pour tout mot w de L, avec $|w| \geq n$, nous pouvons écrire $w = xyz$ de façon à avoir $|xy| \leq n$, $|y| \geq 1$ et pour tout $i \geq 0$, xy^iz appartient à L.

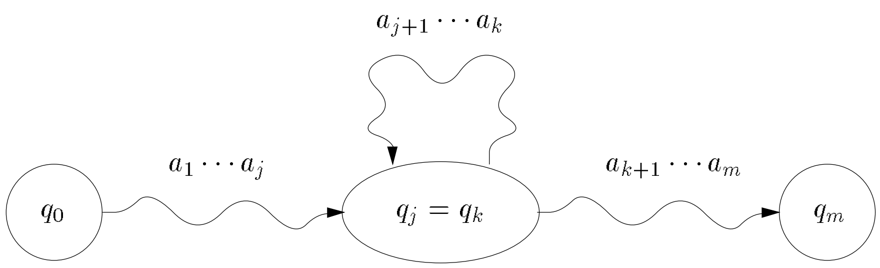
Preuve
Soit $M = \langle Q, \Sigma, \delta, s, F\rangle$ l'automate qui accepte L et qui possède n états distincts. Considérons le mot $w = a_1a_2 \cdots a_m$, $m \geq n$ appartenant à L. Alors la séquence de configurations suivante représente l'acceptation de w par M:

$$(q_0, a_1 \cdots a_m) \vdash_M (q_1, a_2 \cdots a_m) \vdash_M \cdots \vdash_M (q_{m-1}, a_m) \vdash_M (q_m, \epsilon)$$

avec $q_i \in Q$, $0 \leq i \leq m$, $q_0 = s$ et $q_m \in F$. Si l'on considère les n premières dérivations, il n'est pas possible que les $n+1$ états $q_0, q_1, \ldots, q_n$ soient distincts car M ne possède que n états. Ceci signifie qu'il existe deux entiers j et k, $(0 \leq j < k \leq n)$ tel que $q_j = q_k$. Ces valeurs déterminent la décomposition $w = xyz$ où

$$x = a_1 \cdots aj, \quad y = a_{j+1} \cdots a_k, \quad z = a_{k+1} \cdots a_m.$$

On observe que, par construction, $|y| \geq 1$ et $|xy| \leq n$. Nous pouvons schématiser ceci par:



Puisque $w = a_1a_2 \cdots a_m$ appartient à L alors $a_1a_2 \cdots a_ja_{k+1} \cdots a_m$ appartient également à L. En effet, $(q_0, xz) \vdash_M^* (q_j, z)$, $q_j = q_k$ et $(q_k, z) \vdash_M^* (q_m, \epsilon)$. De même, puisque $(q_j, y) \vdash_M^* (q_k, \epsilon)$ et $q_j = q_k$ on a pour tout $i \geq 0$ que xy^iz appartient à L. Nous avons donc prouvé que pour tout mot accepté par un automate à états fini nous pouvons itérer ou "pomper" une sous-chaîne de la chaîne initiale, acceptée, elle aussi, par cet automate. □

2.4.1 Application du lemme de pompage

Le lemme de pompage est employé pour prouver qu'un langage n'est pas régulier! On procède alors par contraposée, ce qui revient à trouver au moins un mot dont aucun découpage ne peut être "pompé", et ceci indépendamment de la constante n du lemme.

Exemples 2.4.1

1. Le langage dont les mots sont composés du même nombre de symboles $\mathtt{a}$ que de symboles $\mathtt{b}$ n'est pas régulier. En effet, le mot particulier $w = \mathtt{a}^n\mathtt{b}^n$ avec n la constante du lemme ($|w|$ est donc bien supérieure à n) ne peut en aucun cas être pompé. Les seuls découpages possibles respectant les contraintes du lemme sont $w = xyz$ où $xy = \mathtt{a}^k \quad (k \leq n)$ et $xy^iz \quad (\forall i \geq 0)$ n'appartient alors pas au langage car le nombre de $\mathtt{a}$ dépasse le nombre de $\mathtt{b}$ pour certains i.

2. Le langage $L = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ n'est pas régulier car si l'on considère le mot $w = \mathtt{a}^k\mathtt{b}^k \in L$ $(\forall k \geq 0)$ aucun découpage de la forme $w = xyz$ n'entraine $xy^iz \in L$ $(\forall i \geq 0)$. En effet, pour tous les découpages possibles le mot $w = xy^iz \notin L$ $(\forall i \geq 0)$:

   - $x = \mathtt{a}^p$, $y = \mathtt{a}^q$ et $z = \mathtt{a}^r\mathtt{b}^k$ $(\forall p, q, r$ avec $|y| > 0)$ on a $xy^iz = \mathtt{a}^p\mathtt{a}^{iq}\mathtt{a}^r\mathtt{b}^k = \mathtt{a}^{p+iq+r}\mathtt{b}^k \notin L$ $(\forall i \geq 0)$,
   - idem pour la symétrie,
   - $x = \mathtt{a}^{k-l}$, $y = \mathtt{a}^l\mathtt{b}^m$, $z = \mathtt{b}^{k-m}$ $(\forall l, m$ avec $|y| > 0)$ on a $xy^iz = \mathtt{a}^{k-l}(\mathtt{a}^l\mathtt{b}^m)^i\mathtt{b}^{k-m} \notin L$ $(\forall i \geq 0)$.

   On note que la preuve de l'exemple 1) aurait également pu être adoptée.

3. Le langage $L = \{\mathtt{a}^n \mid n$ est premier$\}$ n'est pas régulier. En effet, pour un $w = \mathtt{a}^{p+q+r} (\forall q > 0, p, r \geq 0) \in L$, le mot $\mathtt{a}^{p+iq+r} \notin L$ $(\forall i \geq 0)$ car si l'on prend $i = p + 2q + r + 2$ on obtient $p + iq + r = (q + 1)(p + 2q + r)$ qui est le produit de deux nombres supérieurs à 1 et donc évidemment pas premier.

2.5 Résumé

La classe des langages engendrés par les grammaires régulières est identique à celle des langages acceptés par les automates à états finis ainsi qu'à la classe des langages dénotés par les expressions régulières.

L'automate à états fini en tant qu'accepteur est particulièrement simple et pas suffisamment "puissante" dans le sens où certains langages ne sont acceptés par aucun automate à états fini (cf. lemme de pompage). Le chapitre qui suit ainsi que le suivant introduisent des modèles plus sophistiqués afin de répondre à ce manque de puissance.

Chapitre 3

Langages hors-contexte

3.1 Automate à pile déterministe

Le lemme de pompage vu au chapitre précédent montre clairement que certains langages ne peuvent être acceptés par des automates à états finis. En particulier, aucun AF ne permet d'accepter le langage $\{\mathtt{a}^n, \mathtt{b}^n \mid n \geq 1\}$. Pour qu'une machine puisse accepter un tel langage il faudrait que cette dernière soit à même de mémoriser le nombre de symboles a lu pour n'accepter que les mots comportant ce même nombre de symboles b. Les automates à états finis ne possédant pas cette fonctionnalité nous introduisons dans ce chapitre les *automates à pile déterministes (*APD*)*. Un automate à pile est un automate à états fini doté d'une pile jouant le rôle de mémoire. En effet, l'utilisation d'une pile permet de mémoriser les actions précédemment effectuées par l'automate. Les transitions d'un automate à pile s'effectuent donc en fonction du symbole lu et du symbole présent au sommet de la pile.

Exemple 3.1.1 L'automate à pile acceptant le langage $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 1\}$ se comportera comme suit:

1. tant qu'un symbole a peut être lu ce même symbole est placé en sommet de pile,
2. au premier symbole b rencontré un symbole a est dépilé,
3. tant qu'un symbole b peut être lu un symbole a est dépilé,
4. le processus se termine en présence d'une pile vide et lorsque tous les symboles d'entrée ont été consommés.

Définition 3.1.1 *Automate à pile déterministe*
Un *automate à pile déterministe* est un 6-tuple $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \triangleleft \rangle$ où:

- Q est l'ensemble (fini) des états de l'automate,
- Σ est l'alphabet d'entrée,
- Γ est l'alphabet auxiliaire (symboles de la pile) avec $\Gamma \cap \Sigma = \varnothing$,
- δ est la fonction de transition, $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$,
- $q_0 \in Q$ est l'état initial,
- $\triangleleft \in \Gamma$ est le symbole initial de la pile (pour indiquer que la pile est vide). $\Diamond$

3.1.1 Configurations et dérivations

Définition 3.1.2 *Configuration*
Une *configuration* d'un automate à pile est un triplet $c = \langle q, x, v\rangle$ où q représente l'état courant, $x \in \Sigma^*$ les symboles non encore consommés de l'entrée et $v \in \Gamma^*$ le contenu de la pile. $\Diamond$

Définition 3.1.3 *Dérivation*
Une *dérivation en une étape* d'une configuration à une autre pour un automate M est notée $\vdash_M$ et définie par $\langle q, ax, Z\beta\rangle \vdash_M \langle q', x, \alpha\beta\rangle$ avec $q, q' \in Q$, $a \in \Sigma \cup \{\epsilon\}$, $x \in \Sigma^*$, $Z \in \Gamma$ et $\alpha, \beta \in \Gamma^*$ si et seulement si $\delta(q, a, Z) = (q', \alpha)$.
Une *dérivation en plusieurs étapes* est notée $\vdash_M^*$. $\Diamond$

3.1.2 Accepteur de langage

L'automate à pile peut être vu comme un accepteur de langage auquel on ajoute un ensemble $F \subseteq Q$ d'états finaux, $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \lhd, F\rangle$ où F est l'ensemble des états finaux, $F \subseteq Q$,

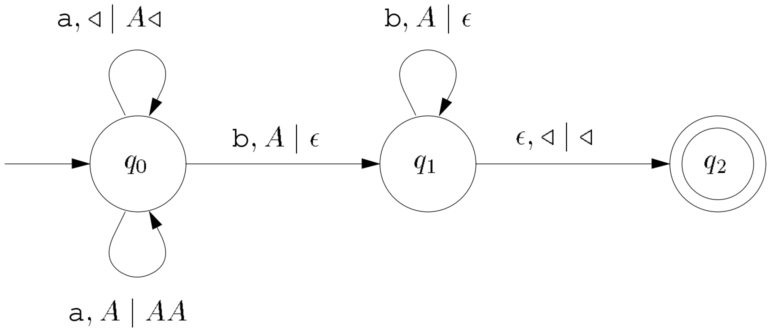
Définition 3.1.4 *Acceptation*
Un mot $x \in \Sigma^*$ est *accepté* par un APD M si et seulement si $\langle q_0, x, \lhd\rangle{\vdash_M}^*\langle q, e, v\rangle$ avec $q \in F$. De même, le *langage accepté* par M est défini par $L(M) = \{x \mid x \in \Sigma^* \text{ et } \langle q_0, x, \lhd\rangle{\vdash_M}^*\langle q, e, v\rangle \text{ et } q \in F\}$ $\Diamond$

Exemple 3.1.2 Construction d'un automate à pile M acceptant le langage $L = \{\mathtt{a}^n\mathtt{b}^n | n \geq 1\}$

$$M = \langle\{q_0, q_1, q_2\}, \{\mathtt{a}, \mathtt{b}\}, \{A, \lhd\}, q_0, \lhd, \{q_2\}\rangle$$

$$\begin{array}{rcl}
\delta(q_0, \mathtt{a}, \lhd) & = & (q_0, A\lhd) \\
\delta(q_0, \mathtt{a}, A) & = & (q_0, AA) \\
\delta(q_0, \mathtt{b}, A) & = & (q_1, \epsilon) \\
\delta(q_1, \mathtt{b}, A) & = & (q_1, \epsilon) \\
\delta(q_1, \epsilon, \lhd) & = & (q_2, \epsilon)
\end{array}$$

Graphiquement cet automate M peut être représenté comme ci-dessous avec la convention: $(a, Z \mid \alpha)$ signifiant: a est le caractère lu en entrée par l'automate, Z le symbole du sommet de la pile avant l'application de la fonction de transition et α est l'état de la pile après application de la fonction de transition.



Pour une chaîne en entrée $\mathtt{a}^i\mathtt{b}^i$ l'automate à pile effectue la séquence de mouvements suivante:

$$\langle q_0, \mathtt{a}^i\mathtt{b}^i, \lhd\rangle \vdash_M \langle q_0, \mathtt{a}^{i-1}\mathtt{b}^i, A\lhd\rangle \vdash_M \cdots \vdash_M \langle q_0, \mathtt{ab}^i, A^{i-1}\lhd\rangle \vdash_M \langle q_0, \mathtt{b}^i, A^i\lhd\rangle \vdash_M$$
$$\langle q_1, \mathtt{b}^{i-1}, A^{i-1}\lhd\rangle \vdash_M \cdots \vdash_M \langle q_1, \epsilon, \lhd\rangle \vdash_M \langle q_2, \epsilon, \lhd\rangle$$

Cet séquence indique que la chaîne $\mathtt{a}^i\mathtt{b}^i$ est acceptée par l'automate.

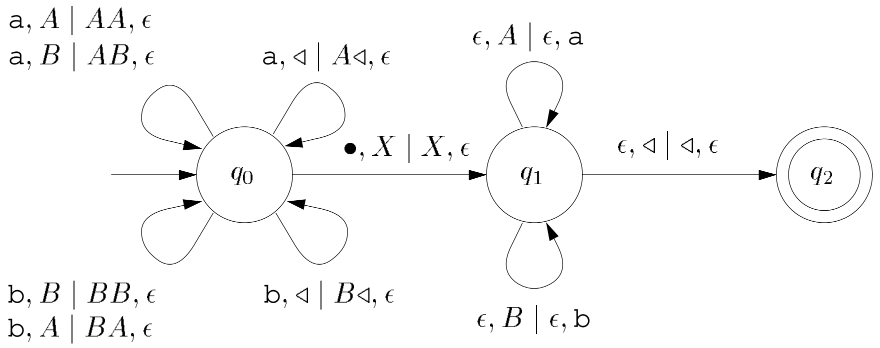
3.1.3 Traducteur de langage

Un *traducteur* basé sur un automate à pile n'est autre qu'un automate à pile accepteur muni d'un alphabet de sortie et d'une fonction exprimant la traduction effectuée pour chaque symbole d'entrée et chaque symbole en sommet de pile.

Définition 3.1.5 *Traducteur*
Un automate à pile *traducteur* T est défini par le 8-tuple $T = \langle Q, \Sigma, \Gamma, \delta, q_0, \triangleleft, O, h\rangle$ où O est l'alphabet de sortie et $h : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow O^*$ la fonction de traduction, les autres éléments restant identiques à ceux de la définition des automates à piles accepteurs. $\Diamond$

Définition 3.1.6 *Configuration, dérivation*
Une *configuration* d'un traducteur M est un quadruplet $c = \langle q, x, g, z\rangle$ où $q \in Q$, $x \in \Sigma^*$, $g \in \Gamma^*$ et $z \in O^*$. Une *dérivation* d'une configuration à une autre pour un automate traducteur T est notée $\vdash_T$ et définie par $\langle q, ax, Z\beta, z\rangle \vdash_T \langle q', x, \alpha\beta, zh(q, a, Z)\rangle$ avec $q, q' \in Q$, $a \in \Sigma \cup \{\epsilon\}$, $x \in \Sigma^*$, $Z \in \Gamma$, $\alpha, \beta \in \Gamma^*$ et $z \in O^*$ si et seulement si $\delta(q, a, Z) = (q', \alpha)$.
Une *dérivation en plusieurs étapes* est notée $\vdash_T^*$. $\Diamond$

Exemple 3.1.3 Le traducteur proposé consiste à traduire un mot en entrée en son mot miroir. Le mot à renverser appartient à $\{\mathtt{a,b}\}^*$ et le symbole $\bullet$ représente le dernier caractère du mot. Ainsi, pour un mot $w\bullet$ avec $w \in \{\mathtt{a,b}^*\}$, le mot w^r sera produit en sortie. La spécification de ce traducteur pourrait être $T = \langle\{q_0, q_1, q_2\}, \{\mathtt{a}, \mathtt{b}, \bullet\}, \{A, B, \triangleleft\}, \delta, q_0, \{q_2\}, \{\mathtt{a}, \mathtt{b}\}, h\rangle$



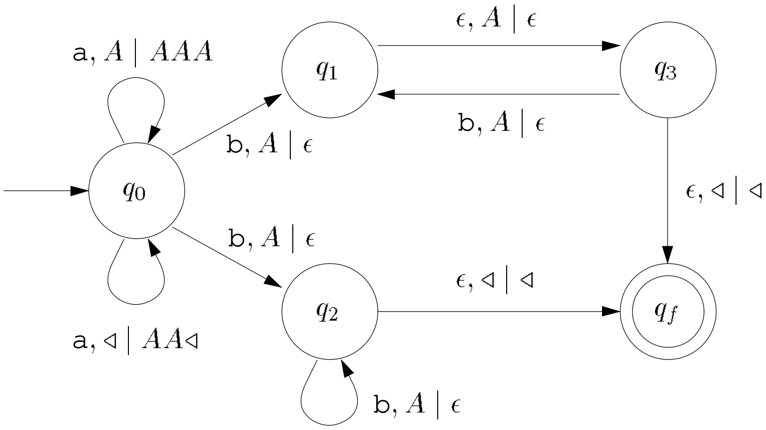
3.2 Automates à pile non déterministes

Certains langages ressemblant au langage $\{\mathtt{a}^n\mathtt{b}^n\}$ comme par exemple le langage $L = \{a^n b^m \mid m = n \text{ ou } m = 2n\}$ ne peuvent être acceptés par un automate muni d'une seule pile. Pour le langage L il faudrait disposer d'un modèle d'automate doté de deux piles, la première mémorisant le nombre n et la seconde le nombre $2n$. En revanche, un automate à deux piles n'est pas en mesure d'accepter le langage $\{\mathtt{a}^n\mathtt{b}^m \mid m = n \text{ ou } m = 2n \text{ ou } m = 3n\}$. Pour ce type de langage il faudrait trois piles. Afin de disposer d'un modèle plus général nous introduisons le non déterminisme dans les automates à pile.

Définition 3.2.1 *Automate à pile non déterministe*
Un automate à pile non déterministe est un 7-tuple $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \triangleleft, F\rangle$ avec $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow \mathbf{P}(Q \times \Gamma^*)$ où $\mathbf{P}(Q \times \Gamma^*)$ dénote l'ensemble des parties de $Q \times \Gamma^*$. $\Diamond$

La notion de dérivation est identique à celle des automates à pile déterministes tout comme la notion d'acceptation.

Exemple 3.2.1 Soit $L = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 1\} \cup \{\mathtt{a}^n\mathtt{b}^{2n} \mid n \geq 1\}$, un automate à pile non déterministe acceptant L pourrait être



Remarque : Notons que nous ne pouvons pas construire un automate à pile déterministe qui accepte le langage L de l'exemple et que, par conséquent, les automates à pile non déterministes acceptent une classe plus large de langages que les automates à pile déterministes. On dit aussi que les automates à pile non déterministes sont "plus puissants" que les automates à pile déterministes.

3.3 Grammaires hors-contexte

Définition 3.3.1 *Grammaire, langage hors-contexte*
Une *grammaire hors-contexte* $G = \langle V_T, V_N, P, S\rangle$ est une grammaire dont les productions ont la forme $\alpha \rightarrow \beta$ où

a) $|\alpha| = 1$ et $\alpha \in V_N$

b) β est une séquence de terminaux ou de non-terminaux.

Le langage généré par une grammaire hors-contexte G est appelé *langage hors-contexte*. On le note $L(G)$. $\Diamond$

Exemples 3.3.1

1. Soit le langage $L_1 = \{\mathtt{a}^n\mathtt{b}^n | n \geq 1\}$, la grammaire qui engendre L_1 peut être définie par $G_1 = \langle\{\mathtt{a}, \mathtt{b}\}, \{S\}, P_1, S\rangle$ où $P_1 = \{S \rightarrow \mathtt{a}S\mathtt{b}, S \rightarrow \mathtt{ab}\}$. On remarque que G_1 est bien hors-contexte et qu'aucune grammaire régulière ne génère L_1 puisque L_1 n'est pas régulier.

2. Soit $G_2 = \langle\{\mathtt{a}, \mathtt{b}\}, \{S, A, B\}, P_2, S\rangle$ avec $P_2 = \{S \rightarrow \mathtt{a}B, S \rightarrow \mathtt{b}A, A \rightarrow \mathtt{a}, A \rightarrow \mathtt{a}S, A \rightarrow \mathtt{b}AA, B \rightarrow \mathtt{b}, B \rightarrow \mathtt{b}S, B \rightarrow \mathtt{a}BB\}$. G_2 est hors-contexte et engendre un langage où tout mot possède le même nombre de symboles $\mathtt{a}$ que de symboles $\mathtt{b}$.

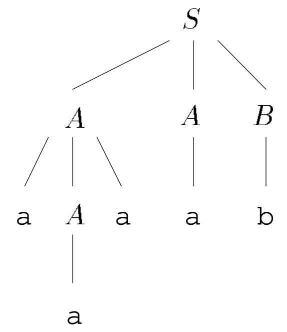
3. Soit G_3 une grammaire produite par l'ensemble de productions $\{S \rightarrow X\mathtt{aa}X, X \rightarrow \mathtt{a}X, X \rightarrow \mathtt{b}X, X \rightarrow \epsilon\}$. G_3 est donc bien hors-contexte et $L(G_3)$ correspond à $L((\mathtt{a}+\mathtt{b})^*\mathtt{aa}(\mathtt{a}+\mathtt{b})^*)$. $L(G_3)$ est donc un langage régulier puisqu'une expression régulière dénote ce langage. Il existe donc également une grammaire régulière générant $L(G_3)$.

3.3.1 Arbres d'analyse et ambiguïté

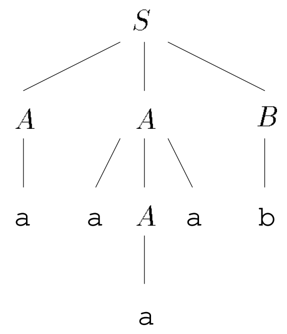
On remarque que les dérivations peuvent être vues comme des arbres: *arbres d'analyse*. Le fait de pouvoir déduire plusieurs arbres d'analyse différents pour la génération d'un même mot nous conduit à la notion de *grammaires ambiguës*.

Exemples 3.3.2

1. Soient les productions $S \to AAB$, $A \to \mathtt{a}A\mathtt{a}$, $A \to \mathtt{a}$, $B \to \mathtt{b}$ et les dérivations $S \Rightarrow AAB \Rightarrow \mathtt{a}A a AB \Rightarrow \mathtt{aaa}AB \Rightarrow \mathtt{aaaa}B \Rightarrow \mathtt{aaaab}$. Elles construisent l'arbre d'analyse



   Par contre, les dérivations $S \Rightarrow AAB \Rightarrow \mathtt{a}AB \Rightarrow \mathtt{aa}A\mathtt{a}B \Rightarrow \mathtt{aaaa}B \Rightarrow \mathtt{aaaab}$ donnent l'arbre d'analyse



   Ces deux ensembles de dérivations génèrent le même mot alors que deux arbres différents leur sont associés. On parle alors de grammaire ambiguë.

2. En revanche, on vérifie facilement que la grammaire donnée par les productions $S \to AB$, $A \to \mathtt{a}A$, $A \to \mathtt{a}$, $B \to \mathtt{b}A$, $B \to \mathtt{b}$ est non ambiguë. C'est-à-dire à tout mot engendré correspond une et une seule dérivation.

Définition 3.3.2 *Grammaire ambiguë et langage ambigu*
Une grammaire hors-contexte G est *ambiguë* s'il existe un mot $w \in L(G)$ tel que w possède deux arbres d'analyse différents. G est non ambiguë si tout $w \in L(G)$ possède un unique arbre d'analyse.

Un langage L est dit *ambigu* si toutes les grammaires qui engendrent L sont ambiguës. Un langage L est non ambigu si $L = L(G)$ pour une G non ambiguë. $\Diamond$

3.3.2 Formes normales

Il est parfois utile de représenter les grammaires sous une forme particulière. Il y a deux formes normales pour les grammaires hors-contexte que l'on va citer ici sans prouver leur équivalence. Nous commençons par la forme normale de Chomsky puis nous donnerons la forme normale de Greibach.

3.3.2.1 Forme normale de Chomsky

Théorème 3.3.1 *Forme normale Chomsky*
Tous les langages hors-contextes sans mot vide peuvent être engendrés par une grammaire dont les productions sont de la forme:

$$A \to BC \text{ ou } A \to a$$

où A, B, C sont des symboles non-terminaux et a est terminal,

Preuve
cf. références. □

Exemple 3.3.3 La grammaire

$$\langle\{S, A, B\}, \{\mathtt{a}, \mathtt{b}\}, P, S\rangle \text{ avec } P = \{S \to \mathtt{b}A \mid \mathtt{a}B, A \to \mathtt{b}AA \mid \mathtt{a}S \mid \mathtt{a}, B \to \mathtt{a}BB \mid \mathtt{b}S \mid \mathtt{b}\}$$

génère le même langage que celle en forme normale de Chomsky

$\langle\{S, A, B, C, D, E, F\}, \{\mathtt{a}, \mathtt{b}\}, P, S\rangle$ avec
$P = \{S \to DA \mid CB, A \to CS \mid DE \mid \mathtt{a}, B \to DS \mid CF \mid \mathtt{b}, E \to AA, F \to BB, C \to \mathtt{a}, D \to \mathtt{b}\}$.

3.3.2.2 Forme normale de Greibach

Théorème 3.3.2 *Forme normale de Greibach*
Tout langage hors-contexte sans mot vide peut être engendré par une grammaire dont les productions sont de la forme:

$$A \to a\alpha \text{ ou } A \to a$$

où A est un symbole non-terminal, α une séquence de non-terminaux et a un terminal.

Preuve
cf. références. □

3.4 Équivalence APN et grammaires hors contexte

Afin de simplifier la démonstration du théorème concernant l'équivalence entre les automates à pile non déterministes et les grammaires hors contexte, nous introduisons une définition de l'acceptation d'un automate à pile légèrement différente de la précédente. Cette fois, pour que l'automate accepte un mot non seulement l'automate doit se trouver dans un état final mais, de plus, la pile doit être vide. Les deux définitions sont tout à fait équivalentes, pour convertir un automate à pile acceptant sur état final en un automate acceptant sur pile vide, il suffit de modifier l'automate de telle façon qu'il puisse vider sa pile lorsqu'il est dans un état final. Inversement, un automate à pile acceptant sur pile vide peut être modifié de telle façon qu'il accepte sur état final en lui ajoutant un nouvel état final qui ne peut être atteint que si la pile est vide.

Définition 3.4.1 *Acceptation sur pile vide*
Un automate à pile M accepte un mot w sur *pile vide* s'il existe au moins une dérivation des configurations telle que

$$\langle q_0, x, \lhd\rangle \vdash_M^* \langle q, e, e\rangle \textit{ avec } q \in Q$$

◊

Théorème 3.4.1 *Équivalence entre APN et grammaires hors contexte*
Si un langage L est accepté par un automate à pile non déterministe M, alors L est un langage hors-contexte.

Preuve
Soit M un automate à pile vide défini par $\langle Q, \Sigma, \Gamma, \delta, q_0, \triangleleft, \emptyset \rangle$ et $G = \langle \Sigma, V, P, S \rangle$ une grammaire hors-contexte où V est l'ensemble des éléments de la forme $[q, A, p]$ avec $q, p \in Q$ et $A \in \Gamma$, S correspond à l'axiome de la grammaire G et P est l'ensemble des productions:

1. $S \to [q_0, \triangleleft, q]$ pour tout $q \in Q$,

2. $[q, A, q_{m+1}] \to a[q_1, B_1, q_2][q_2, B_2, q_3] \ldots [q_m, B_m, q_{m+1}]$, $\forall q, q_1, q_2, \ldots q_{m+1} \in Q$, $a \in \Sigma \cup \{\epsilon\}$ et $A, B_1, B_2, \ldots, B_m \in \Gamma$, tel que $\delta(q, a, A)$ contienne $(q_1, B_1 B_2 \ldots B_m)$. (Si $m = 0$, alors la production est $[q, A, q_1] \to a$).

Afin de mieux comprendre la preuve, on remarque que G a été défini de façon à simuler le comportement de A sur un mot w par une dérivation à gauche de ce mot. On montre par induction sur le nombre d'étapes dans une dérivation pour G ou le nombre de déplacements de l'automate à pile que

$$[q, x, A] \Rightarrow^* x \text{ si et seulement si } (q, x, A) \vdash^* (p, \epsilon, \epsilon)$$

On commence par montrer que si $(q, x, A) \vdash^i (p, \epsilon, \epsilon)$, alors $[q, A, p] \Rightarrow^* x$.

Si $i = 1$, alors $\delta(q, x, A)$ doit contenir (p, ϵ) et par conséquent $[q, A, p] \to x$ est une production de G.

Si $i \geq 1$, on pose $x = ay$, $(q, ay, A) \vdash (q_1, y, B_1, B_2, \ldots, B_n) \vdash^{i-1} (p, \epsilon, \epsilon)$ avec $y = y_1 y_2 \ldots y_n$, où chaque y_i correspond à une action de B_i sur la pile. Alors, par construction, il existe des états $q_2, q_3, \ldots, q_{n+1}$ où $q_{n+1} = p$ tels que $(q_j, y_j, B_j) \vdash^* (q_{j+1}, \epsilon, \epsilon)$ avec moins de i déplacements. Alors, l'hypothèse de l'induction s'applique, $[q_j, B_j, q_{j+1}] \Rightarrow^* y_j$ pour $1 \leq j \leq n$.

Alors, on a $(q, ay, A) \vdash (q_1, y, B_1, B_2, \ldots, B_n)$ comme premier déplacement. Donc, on a la production:

$$[q, A, p] \Rightarrow a[q_1, B_1, q_2][q_2, B_2, q_3] \ldots [q_n, B_n, q_{n+1}]$$

par construction de G. On obtient finalement $[q, A, p] \Rightarrow^* a y_1 y_2 \ldots y_n = x$.

On doit maintenant prouver l'inverse. On suppose que $[q, A, p] \Rightarrow^i x$ et on montre, par induction sur i, que $(q, x, A) \vdash^* (p, \epsilon, \epsilon)$.

Pour $i = 1$, $[q, A, p] \to x$ est une production de G et donc $\delta(q, x, A)$ doit contenir (p, ϵ).

On suppose que $[q, A, p] \Rightarrow a[q_1, B_1, q_2] \ldots [q_n, B_n, q_{n+1}] \Rightarrow^{i-1} x$ où $q_{n+1} = p$. On peut écrire $x = a x_1 \ldots x_n$, où $[q_j, B_j, q_{j+1}] \Rightarrow^* x_j$ pour $1 \leq j \leq n$ où chaque dérivation se fait en nombre d'étapes inférieur à i. Par l'hypothèse inductive, $(q_j, x_j, B_j) \vdash^* (q_{j+1}, \epsilon, \epsilon)$, pour $1 \leq j \leq n$. Si on insère au fond de la pile la séquence $B_{j+1} \ldots B_n$ on obtient $(q_j, x_j, B_j B_{j+1} \ldots B_n) \vdash^* (q_j{+}1, \epsilon, B_{j+1} \ldots B_n)$, et nous savons que $(q, x, A) \vdash (q_1, x_1 \ldots x_n, B_1 B_2 \ldots B_n)$. Alors pour $j = 1, \ldots, n$ $(q, x, A) \vdash^* (p, \epsilon, \epsilon)$.

Pour terminer on fait $q = q_0$, $A = \triangleleft$, pour obtenir:

$$[q_0, \triangleleft, p] \Rightarrow^* x \text{ si et seulement si } (q_0, x, \triangleleft) \vdash^* (p, \epsilon, \epsilon)$$

Alors, si on ajoute les règles (1) de la grammaire G on obtient:

$$S \Rightarrow^* x \text{ si et seulement si } (q_0, x, \triangleleft) \vdash^* (p, \epsilon, \epsilon)$$

c'est-à-dire $x \in L(G)$ si et seulement si $x \in L(A)$. $\square$

Théorème 3.4.2
Si L est un langage hors-contexte, alors il existe un automate à pile non déterministe M tel que $L = L(M)$.

Preuve
Soit L un langage hors-contexte sans le mot vide ϵ et soit $G = \langle T, V, P, S \rangle$ une grammaire en forme normale de Greibach engendrant L. On définit un automate à pile vide défini par $M = \langle \{a\}, T, V, \delta, q, S, \rangle$, où $\delta(q, a, A)$ contient (q, γ) s'il existe une production de la forme $A \rightarrow a\gamma \in P$.

L'automate à pile M simule chaque séquence obtenue par une dérivation à gauche de la grammaire. Alors, G est en forme normale de Greibach, c'est-à-dire que toutes les productions ont la forme $x\alpha$ où $x \in T$, et α est une séquence de non-terminaux; donc M va lire a avec α dans la pile. Formellement, on doit prouver que:

$$S \Rightarrow^* x\alpha \text{ par une dérivation à gauche si et seulement si } (q, x, S) \vdash^* (q, \epsilon, \alpha).$$

Supposons que $(q, x, S) \vdash^i (q, \epsilon, \alpha)$ et montrons par induction sur i que $S \Rightarrow^* x\alpha$.

Pour $i = 0$, il suffit de faire $x = \epsilon$ et $\alpha = S$.

Pour $i \geq 1$, posons $x = ya$ et considérons l'expression suivante:

$$(q, ya, S) \vdash^{i-1} (q, a, \beta) \vdash (q, \epsilon, \alpha)$$

Donc, $(q, y, S) \vdash (q, \epsilon, \alpha)$ car a n'a pas d'effet sur les $i - 1$ premiers mouvements de M. Alors, $S \Rightarrow^* y\beta$ par hypothèse d'induction. Le mouvement $(q, a, \beta) \vdash (q, \epsilon, \alpha)$ implique que $\beta = A\gamma$ pour $A \in V$, $A \rightarrow a\eta \in P$ et $\alpha = \eta\gamma$. Ainsi $S \Rightarrow^* y\beta \Rightarrow ya\eta\gamma = x\alpha$; ce qui prouve la première partie de notre expression.

Pour compléter, supposons que $S \Rightarrow^i x\alpha$ obtenue par une dérivation à gauche. Montrons par induction sur le nombre de dérivations i que $(q, x, S) \vdash^* (q, \epsilon, \alpha)$.

Le cas de base pour $i = 0$ est trivial. Pour $i \geq 1$, supposons vraies les implications suivantes

$$S \Rightarrow^{i-1} yA\gamma \Rightarrow ya\eta\gamma,$$

où $x = ya$ et $\alpha = \eta\gamma$. Par hypothèse $(q, y, S) \vdash^* (q, \epsilon, A\gamma)$ et par conséquent $(q, ya, S) \vdash^* (q, a, A\gamma)$. Si $A \rightarrow a\eta \in P$, alors on a $(q, \eta) \in \delta(q, a, A)$. Donc, $(q, x, S) \vdash^* (q, a, A\gamma) \vdash (q, \epsilon, \alpha)$. Ce qui complète la preuve inductive.

Pour conclure, si l'on fait $\alpha = \epsilon$, on obtient $S \Rightarrow^* x$ si et seulement si $(q, x, S) \vdash^* (q, \epsilon, \epsilon)$. Ainsi x appartient à $L(G)$ si et seulement si x appartient à $L(M)$. □

3.4.1 Lemme de pompage

Comme pour les langages réguliers, il existe un résultat nous permettant de déterminer si un langage n'est pas hors-contexte. On énonce ce résultat sans le démontrer.

Lemme 3.4.1
Si L est un langage hors-contexte alors il existe une constante n telle que pour tout mot w appartenant à L, avec $|w| \geq n$, on puisse découper le mot $w = uvxyz$ de manière à ce que

1. $|vy| \geq 1$,
2. $|vxy| \leq n$ et
3. $\forall i \geq 0$, $uv^ixy^iz \in L$.

Preuve
cf. références. □

Exemple 3.4.1 Soit le langage $L = \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i \mid i \geq 1\}$. Montrons que ce langage ne peut être hors-contexte au moyen du lemme ci-dessus. Par l'absurde, supposons L hors-contexte et prenons le mot $w = \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$. Nous allons montrer qu'aucun découpage du mot w ne respecte les 3 points du lemme et que, par conséquent le langage n'est pas hors-contexte.

Découpons $w = uvxyz$, comme $|vxy| \leq n$, il n'est pas possible pour v et y d'avoir des a et des c. Si v et y n'ont que des a, alors uxz ne peut pas avoir n symboles a, car $|vy| \geq 1$. Mais uxz doit appartenir à L par le lemme de pompage ($uxz = uv^0xy^0z$), ce qui n'est pas le cas.

On peut analyser de la même manière les autres découpages: lorsqu'il n'y a que des symboles b dans v et y ou que des c ou que des a et des b ou encore que des b et c. Dans chaque situation, on obtient une contradiction. Ainsi L ne peut pas être hors-contexte.

Nous avons vu que les langages hors-contexte sont acceptés par des automates à pile non déterministes ou générés par des grammaires hors-contextes. Il reste donc à montrer qu'il existe des langages hors-contextes qui ne peuvent être acceptés par des automates à pile déterministes.

Théorème 3.4.3
Il existe un langage L hors-contexte, tel que L n'est accepté par aucun automate à pile déterministe.

Preuve
Considérons par exemple:

$$L = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i \neq j \text{ ou } j \neq k\}$$

Il est aisé de trouver un automate à pile non déterministe acceptant L. L est donc hors-contexte. Supposons maintenant que L soit accepté par un automate à pile déterministe et montrons une contradiction. Considérons le langage complémentaire de L, $\bar{L} = \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i \mid i \geq 1\}$; $\bar{L}$ n'est pas hors-contexte comme nous venons de le prouver à l'aide du lemme de pompage. Mais si L était accepté par un automate à pile déterministe on pourrait inverser les états finaux et non-finaux pour obtenir un automate à pile déterministe qui accepte $\bar{L}$. Mais $\bar{L}$ n'est pas hors-contexte, d'où contradiction. □

Chapitre 4

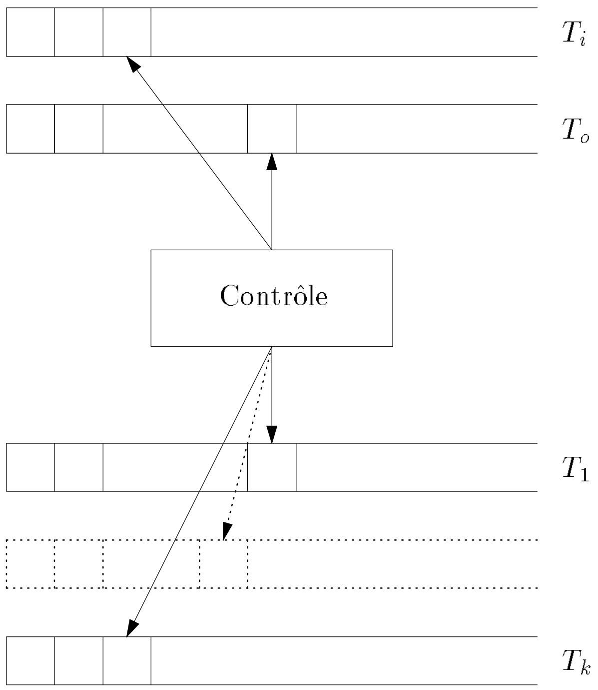
Langages récursivement énumérables

4.1 Machine de Turing déterministe

Introduite la première fois par Alan Turing en 1936, la machine qui porte son nom, machine de Turing (MT), est un modèle mathématique très simplifié d'un calculateur mais suffisamment puissant pour accomplir tout ce qu'un calculateur est en mesure de réaliser. Il existe plusieurs variantes de ce modèle toutes équivalentes en puissance. Informellement, une machine de Turing est constituée:

1. d'une bande d'entrée T_i et de sa tête de lecture,
2. d'une bande de sortie T_o et de sa tête d'écriture,
3. de k bandes de travail et de leurs tête de lecture/écriture et
4. d'un bloc de contrôle, partie correspondant à l'automate.

On peut schématiser une telle machine comme ceci:



Une bande est une sorte de tampon avec un nombre infini de cases, chaque case pouvant contenir au plus un symbole. Les bandes $T_1, \ldots, T_k$ sont accessibles en lecture et en écriture avec la possibilité d'agir simultanément sur plusieurs bandes, et cela de manière complètement indépendante. Par contre, la bande d'entrée T_i est accessible uniquement en lecture alors que la bande de sortie T_o l'est uniquement en écriture. Initialement la bande T_i contient le mot en entrée et toutes les autres cases contiennent un symbole particulier appelé blanc et noté $\Box$.

Un *pas* ou *déplacement* dans une machine de Turing est constitué:

1. du passage d'un état à un autre état en fonction des symboles se trouvant en face des têtes des bandes d'entrée et de travail (lecture).
2. de la modification du symbole se trouvant en face des têtes des bandes de sortie et de travail (écriture).
3. d'un déplacement des têtes des bandes soit vers la gauche (L) soit vers la droite (R) soit stationnaire (S).

On représente graphiquement un tel pas de la manière suivante:

$$q_1 \xrightarrow{\quad i, r_1, \ldots, r_k \mid r'_1, \ldots, r'_k, o, \langle M_0, \ldots, M_{k+1} \rangle \quad} q_2$$

Cette représentation signifie que lorsque la machine se trouve dans l'état q_1 et passe à l'état q_2, si le symbole i se trouve en face de la tête de la bande d'entrée T_i et les symboles $r_1, \ldots, r_k$ sont en face des bandes de travail $T_1, \ldots, T_k$. Alors, la machine remplace les symboles des bandes de travail $r_1, \ldots, r_k$ par $r'_1, \ldots, r'_k$ et écrit le symbole o sur la bande de sortie T_o avant de déplacer les $k+2$ têtes suivant le tuple $\langle M_0, \cdots, M_{k+1} \rangle$. Chaque M_i peut avoir la valeur L, R, ou S. M_0 représente le mouvement de T_i et M_{k+1} celui de T_o qui doit être différent de L.

Exemple 4.1.1

1. Reprenons le langage $L_1 = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 1\}$; la reconnaissance de L_1 par une machine de Turing à une bande de travail se fait suivant les étapes:

   - écrire tous les symboles $\mathtt{a}$ (déplacement vers la droite) sur la bande de travail jusqu'à ce qu'on rencontre le premier symbole $\mathtt{b}$.
   - consommer un symbole $\mathtt{b}$ pour chaque symbole $\mathtt{a}$ présent sur la bande de travail (déplacement vers la gauche).
   - si tous les symboles $\mathtt{b}$ ont été consommés et qu'aucun symbole $\mathtt{a}$ ne peut être lu sur la bande de travail, la machine s'arrête alors dans un état final et accepte, sinon la machine n'accepte pas la chaîne en entrée.

2. Soit $L_2 = \{\mathtt{a}^n\mathtt{b}^{2n} | n \geq 1\}$, la machine de Turing pour ce langage fonctionne selon le même principe que celui de l'exemple précédent. Cette fois-ci deux symboles $\mathtt{b}$ sont consommés pour chaque symbole $\mathtt{a}$.

3. La machine de Turing acceptant $L = L_1 \cup L_2$ utilise deux bandes de travail et fonctionne de la manière suivante:

   (a) la machine écrit pour chaque $\mathtt{a}$ en entrée, un $\mathtt{a}$ sur la première bande de travail et deux $\mathtt{a}$ sur la deuxième bande jusqu'au premier $\mathtt{b}$ rencontré,

(b) change le sens du déplacement vers la gauche et consomme un `b` pour chaque `a` sur les deux bandes,

(c) s'il n'y a plus de `a` sur la première bande et plus de `b` en entrée alors M s'arrête dans un état final,

(d) sinon consomme les `b` restants avec les `a` de la deuxième bande,

(e) s'il n'y a plus de `a` sur la deuxième bande et plus de `b` en entrée alors M s'arrête dans un état final,

(f) dans tous les autres cas M n'accepte pas la chaîne en entrée.

4.1.1 Définitions formelles

Définition 4.1.1 *Machine de Turing déterministe*
Une *machine de Turing déterministe à $k \geq 0$ bandes de travail* est définie par le 8-tuple $M = \langle Q, \Sigma, \Gamma, O, \delta, q_0, \Box, F\rangle$ où

- Q est l'ensemble (fini) des états,
- Σ est l'alphabet d'entrée,
- Γ est l'alphabet auxiliaire,
- O est l'alphabet de sortie,
- $F \subseteq Q$ est l'ensemble des états finaux,
- q_0 est l'état initial,
- $\Box$ est le symbole initial blanc,
- δ est la fonction de transition, $\delta : Q \times \Sigma \times \Gamma^k \to Q \times \Gamma^k \times O \times \{R, S\} \times \{R, L, S\}^k \times \{R, S\}$, R,L et S représentent le sens du déplacement des têtes R: vers la droite, L: vers la gauche, S: stationnaire. $\Diamond$

Définition 4.1.2 *Configuration*
Une *configuration* d'une machine de Turing est représentée par:

$$c = \langle q, x \uparrow iy, \alpha_1 \uparrow A_1\beta_1, \cdots, \alpha_k \uparrow A_k\beta_k, u \uparrow o\rangle$$

avec $q \in Q$, $A_j \in \Gamma$, $\alpha_j, \beta_j \in \Gamma^*$, $i \in \Sigma$, $x, y \in \Sigma^*$ et $o, u \in O^*$.
Une configuration initiale est de la forme $c_0 = \langle q_0, \uparrow ix, \uparrow \Box, \cdots, \uparrow \Box, \uparrow \Box\rangle$ $\Diamond$

Définition 4.1.3 *relation de transition ou dérivation*
La *relation de transition* ou *dérivation* d'une configuration $c = \langle q, x \uparrow iy, \alpha_1 \uparrow a_1\beta_1, \cdots, \alpha_k \uparrow a_k\beta_k, u \uparrow ov\rangle$ vers une configuration $c' = \langle q', x' \uparrow i'y', \alpha'_1 \uparrow a'_1\beta'_1, \cdots, \alpha'_k \uparrow a'_k\beta'_k, u' \uparrow o'v'\rangle$ notée $c \vdash_M c'$ est appelée *mouvement* ou *pas* de M et est définie comme suit:
$c \vdash_M c'$ si et seulement si $\delta(q, i, a_1, \cdots, a_k) = \langle q', c_1 \cdots c_k, N_i, N_1, \cdots, N_k, N_o\rangle$ où les c_i appartiennent à l'alphabet des bandes de travail et les N_i représentent $\{R, L, S\}$ avec les conditions:

1. $x' = x, i' = i, y' = y, N_i = S$ ou bien
2. $x' = xi, y = i'y', N_i = R$

avec $x, x', y, y' \in \Sigma^*$ et $i, i' \in \Sigma$ et pour chaque r $(1 \leq r \leq k)$

1. $\alpha'_r = \alpha_r, \beta'_r = \beta_r, N_r = S$ ou bien
2. $\alpha'_r = \alpha_r a_r, \beta_r = a'_r \beta'_r, N_r = R$ ou bien
3. $\alpha_r = \alpha'_r a'_r, \beta'_r = a_r \beta_r$

avec $\alpha_r, alpha'_r, \beta_r, \beta'_r \in \Gamma^*$ et $a_r, a'_r \in \Gamma$, de plus

1. $u' = u, o' = o, v' = v, N_o = S$ ou bien
2. $u' = uo, v = o'v', N_o = R$

avec $u, u', v, v' \in O^*$ et $o, o' \in O$. $\Diamond$

Définition 4.1.4 *Mot et langage accepté*
Un mot $x \in \Sigma^*$ est accepté par une machine M si et seulement si

$$c_0 = \langle q_0, x, \uparrow\Box, \cdots, \uparrow\Box\rangle \vdash_M^* \langle q, x'\uparrow iy, \alpha_1\uparrow A_1\beta_1, \cdots, \alpha_k\uparrow A_k\beta_k\rangle \text{ avec } q \in F.$$

Un langage L est accepté par une machine M si et seulement si

$$L = \{x \mid x \in \Sigma^* \text{ et } x \text{ est accepté par } M\}$$

. Le langage accepté par une MT M est noté $L(M)$. $\Diamond$

Définition 4.1.5 *Langage récursivement énumérable*
Un langage accepté par une machine de Turing est appelé un langage *récursivement énumérable*. $\Diamond$

Exemple 4.1.2 Soit le langage $L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 1\}$ et soit

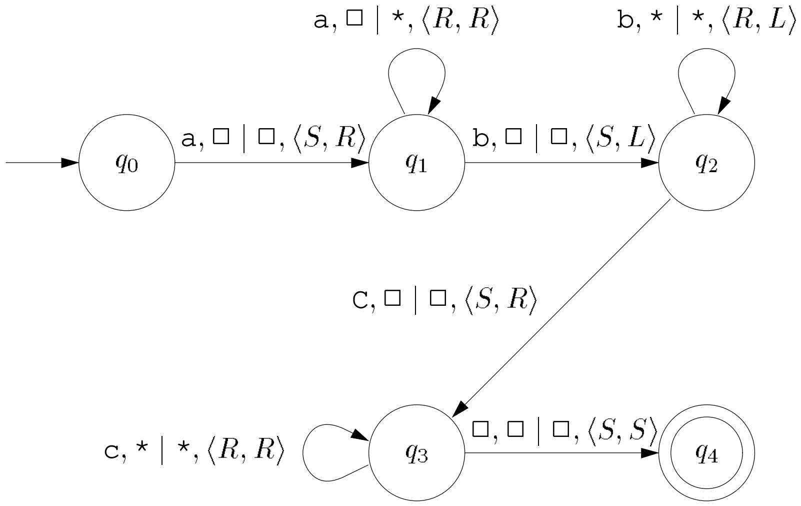
$$M = \langle\{q_0, q_1, q_2, q_3, q_4\}, \{\mathtt{a,b,c}\}, \{*, \Box\}, \emptyset, \delta, q_0, \Box, \{q_4\}\rangle$$

une machine de Turing avec une bande de travail qui accepte L.

Le principe de fonctionnement de cette machine est décrit selon les étapes suivantes:

1. Lors du premier a on inscrit un blanc sur la bande de travail qui servira de marque de début de bande par la suite, (représenté par le passage de q_0 à q_1 dans le diagramme qui suit),
2. Tant qu'il y a des a en entrée on écrit le symbole * sur la bande de travail, et cela jusqu'au premier b rencontré (représenté par l'état q_1),
3. tant qu'il y a des b en entrée on lit un symbole * sur la bande en avançant à gauche et cela jusqu'au premier c rencontré (représenté par l'état q_2),
4. tant qu'il y a des c en entrée on lit un symbole * sur la bande en avançant à droite et cela jusqu'au premier blanc suivant le mot d'entrée (représenté par l'état q_3),
5. si le nombre de a est égal au nombre de b et de c alors la machine s'arrête dans un état final (représenté par l'état q_4).

Ce fonctionnement correspond à une fonction de transition représentée graphiquement par le diagramme qui suit. La machine n'employant pas la bande de sortie le symbole correspondant à cette bande à été omis.

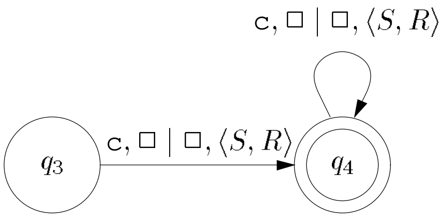


et formellement la fonction de transition est:

$$
\begin{array}{rcl}
\delta(q_0, \mathtt{a}, \Box) & = & (q_1, \Box, S, R) \\
\delta(q_1, \mathtt{a}, \Box) & = & (q_1, \star, R, R) \\
\delta(q_1, \mathtt{b}, \Box) & = & (q_2, \Box, S, L) \\
\delta(q_2, \mathtt{b}, \star) & = & (q_2, \star, R, L) \\
\delta(q_2, \mathtt{c}, \Box) & = & (q_3, \Box, S, R) \\
\delta(q_3, \mathtt{c}, \star) & = & (q_3, \star, R, R) \\
\delta(q_3, \Box, \Box) & = & (q_4, \Box, R, R)
\end{array}
$$

Remarque : Comme dans le cas des automates à états finis, pour que la fonction de transition soit totale il est nécessaire d'ajouter un état supplémentaire, appelé aussi état "puits". Si ce dernier est déclaré état final et que l'on adopte la convention suivante: si la machine s'arrête dans l'état "puits" le mot en entrée n'appartient pas au langage accepté, on a alors défini la notion d'état acceptant et d'état rejetant.

L'ajout de cet état puits ne signifie pas que la machine termine son exécution dans tous les cas. En effet, si l'on ajoute ce qui suit ainsi qu'un état rejetant à notre exemple précédent on obtient une nouvelle machine M' acceptant bien le même langage que la machine M $(L(M) = L(M'))$. Cependant elle n'est pas en mesure de décider dans tous les cas si une entrée n'appartient pas au langage accepté puisque M' peut ne pas s'arrêter.



4.1.2 Machine de Turing comme évaluateur de fonctions

L'évaluation des fonctions entières à résultat entier, peut être accomplie par une machine de Turing. L'approche traditionnelle consiste à représenter les entiers en base unaire, c'est-à-dire que l'entier $i \geq 0$ est représenté par la chaîne $\mathtt{0}^i$. Si une fonction possède k arguments $i_1, i_2, \cdots, i_k$, alors ces entiers sont représentés par la chaîne $\mathtt{0}^{i_1}\mathtt{1}\mathtt{0}^{i_2}\mathtt{1}\cdots\mathtt{1}\mathtt{0}^{i_k}$. Lorsque la machine s'arrête avec un contenu de bande de sortie égal à $\mathtt{0}^m$ on dit alors que $f(i_1, i_2, \cdots, i_k) = m$, où f est une fonction à k arguments.

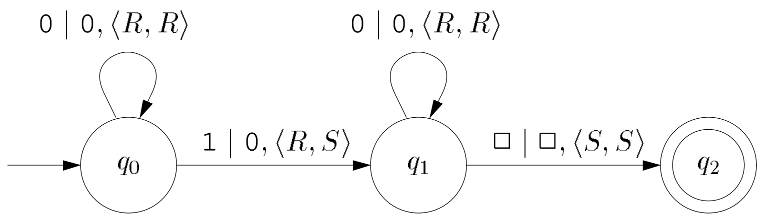
Exemples 4.1.3

1. Pour l'addition de deux nombres entiers m, n aucune bande de travail n'est nécessaire. En effet, pour une entrée de la forme $\mathtt{0}^m\mathtt{1}\mathtt{0}^n$ il suffit de copier les m et les n $\mathtt{0}$ de la bande d'entrée sur la bande de sortie.

   La machine $M = \langle\{q_0, q_1, q_3\}, \{\mathtt{0}, \mathtt{1}\,\Box\}, \{\mathtt{0}, \mathtt{1}\}, \delta, q_0, \Box, \{q_2\}\rangle$ avec la fonction δ suivante effectue ce calcul:

$$\begin{array}{rcl} \delta(q_0, \mathtt{0}) & = & (q_0, \mathtt{0}, R, R) \\ \delta(q_0, \mathtt{1}) & = & (q_1, \mathtt{0}, R, S) \\ \delta(q_1, \mathtt{0}) & = & (q_1, \mathtt{0}, R, R) \\ \delta(q_1, \Box) & = & (q_2, \Box, S, S) \end{array}$$

   Graphiquement cela se représente par:



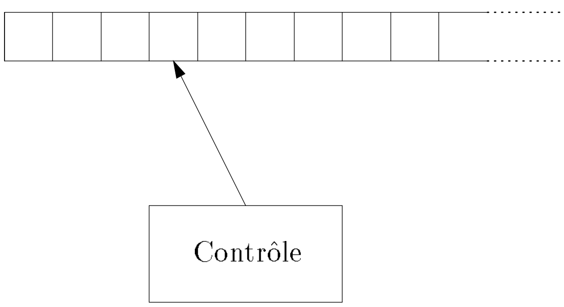
2. La multiplication de deux nombres entiers m, n peut s'effectuer par additions successives, une bande de travail simplifie alors les choses. Le fonctionnement d'une machine réalisant cette opération revient à copier m $\mathtt{0}$ sur la bande de travail, puis pour chaque $\mathtt{0}$ de n la machine copie m $\mathtt{0}$ sur la bande de sortie.

4.2 Modification de la machine de Turing

Comme nous l'avons dit, il existe plusieurs variantes de la machine de Turing décrite précédemment. Dans cette section nous allons exposer quelques unes de ces variantes et montrer que tous ces modèles sont équivalents en "puissance", c'est-à-dire que toutes ces variantes sont en mesure de réaliser exactement les mêmes travaux.

4.2.1 Machine de Turing à une seule bande

Ce type de machine de Turing ne possède pas de bande de travail ni de bande de sortie. La machine ne dispose que d'une seule bande accessible en lecture/écriture jouant le rôle de bande d'entrée et de bande de sortie. La tête peut se déplacer dans les deux sens mais ne peut rester immobile. A l'initialisation la bande contient le mot à traiter et le symbole blanc $\Box$ pour toutes les autres cases. Le dessin suivant schématise une telle machine:



Définition 4.2.1 *Machine de Turing à une seule bande*
Une *machine de Turing à une seule bande* M est un 7-tuple $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \Box, F\rangle$ où

- Q est l'ensemble (fini) des états,
- Γ, l'ensemble fini des symboles de la bande,
- Σ est l'alphabet d'entrée, un sous-ensemble de Γ n'incluant pas $\Box$,
- δ est la fonction de transition, $\delta : Q \times \Sigma \to Q \times \Gamma \times \{L, R\}$ avec L pour un déplacement vers la gauche et R vers la droite,
- q_0 est l'état initial,
- $\Box$ est le symbole initial blanc,
- $F \subseteq Q$ est l'ensemble des états finaux. $\Diamond$

Exemple 4.2.1

1. Une fois de plus, étudions le langage $L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 1\}$. L est accepté, entre autres, par la machine de Turing à une seule bande se comportant de la manière suivante:

   - pour chaque symbole `a` la machine substitue le `a` par le symbole `A`, recherche le premier `b` et le change en `X`, puis ramène la tête sur le symbole `a` le plus à gauche,
   - si le nombre de `a` est différent du nombre de `b` il doit alors persister soit des symboles `a` ou `b`, dans ce cas la machine rejette le mot,
   - dans le cas contraire il suffit de réappliquer le même traitement pour les symboles `X` et `c` en changeant les `X` en `B` et les `c` en `C`,
   - la machine accepte donc le mot en entrée si et seulement si il ne persiste plus de symboles `a`, `b`, `c` ou `X`.

   Par exemple, le mot `aaabbbccc` sera transformé en `AAABBBCCC` et sera donc accepté. Par contre le mot `aaabbbbccc` sera transformé en `AAAXXXbccc` et sera rejeté tout comme `aabbcccc` puisque ce dernier deviendra `AABBCCcc`.

2. Cet exemple utilise une machine de Turing à une seule bande afin de réaliser l'opération de soustraction propre entre deux nombres entiers $n, m \geq 0$.

Soit la fonction de *soustraction propre* définie par:

$$m \ominus n = \begin{cases} m - n & \text{pour } m \geq n \\ 0 & \text{pour } m < n \end{cases}$$

La machine de Turing $M = \langle \{q_0, q_1, \cdots, q_6\}, \{\texttt{0}, \texttt{1}, \Box\}, \{\texttt{0}, \texttt{1}\}, \delta, q_0, \Box, \{q_6\} \rangle$ calcule la soustraction propre d'une entrée de la forme $\texttt{0}^m\texttt{1}\texttt{0}^n$. Il restera donc sur la bande $m \ominus n$ symboles 0. Le comportement de la machine M est le suivant:

(a) M remplace le premier 0 par $\Box$,

(b) recherche (vers la droite) un 1 suivi d'un 0 et remplace ce dernier par un 1,

(c) recherche (vers la gauche) le premier $\Box$ et retourne en 1),

(d) M s'arrête lorsque:

   i. si en recherchant à droite un 0 M rencontre un $\Box$. Les n 0 de n ont été changés en 1 et $n+1$ 0 de m ont été changés en $\Box$. M remplace alors les 1 par $\Box$ et le $n+1$-ème $\Box$ par 0. Il reste donc sur la bande $m \ominus n$ 0.

   ii. En retournant en 1), M ne trouve plus de 0 à remplacer par $\Box$, car les premiers m 0 ont tous été changés. On se trouve alors dans le cas où $n \geq m$, et le résultat est nul. M remplace alors tous les 0 et 1 restant par des $\Box$.

Voici la fonction de transition de cette machine:

(a) $\delta(q_0, \texttt{0}) = (q_1, \Box, R)$
   Début de la boucle. M remplace le premier 0 par $\Box$,

(b) $\delta(q_1, \texttt{0}) = (q_1, \texttt{0}, R)$
   $\delta(q_1, \texttt{1}) = (q_2, \texttt{1}, R)$
   M cherche (vers la droite) le premier 1,

(c) $\delta(q_2, \texttt{1}) = (q_2, \texttt{1}, R)$
   $\delta(q_2, \texttt{0}) = (q_3, \texttt{1}, L)$
   M cherche (vers la droite) le premier 0 après un 1, puis le change en 1,

(d) $\delta(q_3, \texttt{0}) = (q_3, \texttt{0}, L)$
   $\delta(q_3, \texttt{1}) = (q_3, \texttt{1}, L)$
   $\delta(q_3, \Box) = (q_0, \Box, R)$
   M se déplace (vers la gauche) jusqu'au premier $\Box$ et retourne à 1).

(e) $\delta(q_2, \Box) = (q_4, \Box, L)$
   $\delta(q_4, \texttt{1}) = (q_4, \Box, L)$
   $\delta(q_4, \texttt{0}) = (q_4, \texttt{0}, L)$
   $\delta(q_4, \Box) = (q_6, \texttt{0}, R)$
   Description du cas 5) a),

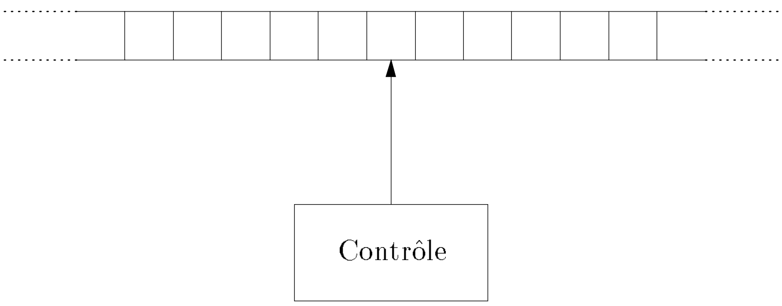
(f) $\delta(q_0, \texttt{1}) = (q_5, \Box, R)$ $\quad\delta(q_5, \texttt{0}) = (q_5, \Box, R)$
   $\delta(q_5, \texttt{1}) = (q_5, \Box, R)$ $\quad\delta(q_5, \Box) = (q_6, \Box, R)$
   Description du cas 5) b).

Théorème 4.2.1
Un langage L est accepté par une machine de Turing à k bandes de travail si et seulement si L est accepté par une machine à une seule bande.

Preuve
Exercice. $\Box$

4.2.2 Machine de Turing à une seule bande infinie des deux côtés

Cette variante de la machine à une seule bande a la particularité de posséder une bande infinie aussi bien à droite qu'à gauche. La définition d'une telle machine reste identique à celle de la machine à une seule bande (c.f. déf. 4.2.1) mais sa tête de lecture est libre de se déplacer dans les deux sens quelque soit son emplacement. La figure qui suit illustre une telle machine:



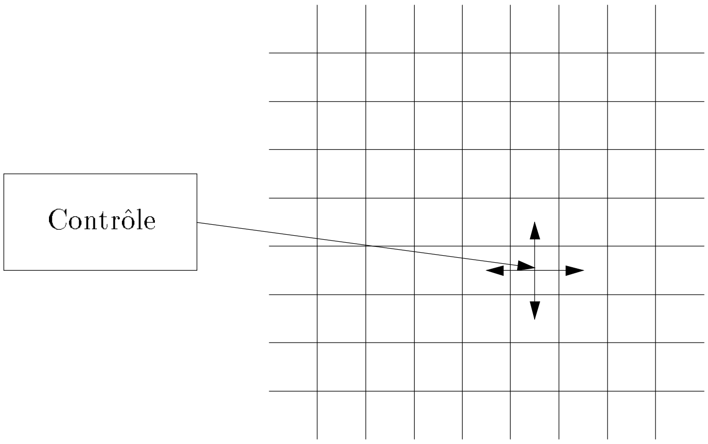
Théorème 4.2.2
Un langage L est accepté par une machine de Turing à une bande infinie des deux côtés si et seulement si L est accepté par une machine à une seule bande.

Preuve
Exercice. □

4.2.3 Machine de Turing multidimensionnelle

La machine de Turing multidimensionnelle est une machine travaillant sur un tableau de dimension k (constante), infini pour les $2k$ directions. La machine possède une tête de lecture/écriture pouvant se déplacer selon les $2k$ directions.

La figure suivante illustre la machine de Turing multidimensionnelle à deux dimensions:



Théorème 4.2.3
Si le langage L est accepté par une machine de Turing à deux dimensions M_2 alors L est accepté par une machine à une dimension M_1.

Preuve
On va définir une machine M_1 à une dimension avec deux bandes qui accepte le même langage que

M_2. Soit la machine de Turing à deux dimensions M_2, dont la bande est représentée par le tableau suivant:

$\Box$	$\Box$	$\Box$	a_1	$\Box$	$\Box$	$\Box$
$\Box$	$\Box$	a_2	a_3	a_4	a_5	$\Box$
a_6	a_7	a_8	a_9	$\Box$	a_{10}	$\Box$
$\Box$	a_{11}	a_{12}	a_{13}	$\Box$	a_{14}	a_{15}
$\Box$	$\Box$	a_{16}	a_{17}	$\Box$	$\Box$	$\Box$

Soit M_1 la machine de Turing à une dimension, dont l'une des bandes représente la bande de M_2 et a l'aspect suivant:

$$**\Box\Box\Box a_1\Box\Box\Box*\Box\Box a_2a_3a_4a_5\Box*a_6a_7a_8a_9\Box a_{10}\Box*\Box a_{11}a_{12}a_{13}\Box a_{14}a_{15}*\Box\Box a_{16}a_{17}\Box\Box\Box**$$

Si M_2 effectue un déplacement horizontal, alors M_1 se déplace d'une case vers la gauche ou vers la droite. La situation se complique lorsque M_2 effectue un déplacement vertical. Alors M_1 utilise la seconde bande comme compteur de cases. Si le mouvement se fait vers le bas alors M_1 déplace sa tête vers la droite et se repère par rapport au symbole $*$ en utilisant le compteur pour connaître la distance entre sa tête et le symbole $*$. Si ce mouvement s'effectue vers le haut, alors M_1 déplace sa tête vers la gauche et se positionne à la case correspondant à celle pointée par M_2 sur la bande à deux dimensions en utilisant le compteur.

Remarquons que la preuve est basée sur le fait d'avoir toujours un rectangle limité par des symboles $*$. Au début, on a l'entrée en une ligne horizontale sur M_2 et on peut la copier sur la première bande de M_1 en la délimitant par des $*$. Il reste à considérer la situation, où M_2 sort du rectangle formé. Si la sortie est verticale, M_1 doit additionner un nouveau bloc de symboles B vers la gauche ou vers la droite, en utilisant le compteur pour savoir combien il en faut exactement. Si le mouvement est horizontal, alors M_1 doit additionner un nouveau $\Box$ à chaque bloc de symboles.

Alors, M_1 simule M_2 acceptant un mot w si et seulement si M_2 le fait aussi. $\Box$

4.3 Machine de Turing non déterministe

La machine de Turing non déterministe est une machine qui, pour un état et un symbole donné, possède un nombre fini de choix pour le prochain mouvement. Chaque choix consiste en un nouvel état, un symbole de bande à imprimer, et un changement de direction. La machine de Turing non déterministe accepte son entrée, s'il existe au moins un chemin qui mène à un état acceptant parmi tous les chemins possibles.

On montre que cette nouvelle sorte de machine est équivalente aux machines déterministes. Cela signifie que le modèle non déterministe n'est pas plus "puissant" que le modèle déterministe.

Définition 4.3.1 *Machine de Turing non déterministe*
Une *machine de Turing non déterministe* est définie par le 9-tuple $M = \langle Q, \Sigma, \Gamma, O, \delta, q_0, \Box, F\rangle$ où seule la fonction de transition

$$\delta : Q \times \Sigma \times \Gamma^k \to \mathbf{P}(Q \times \Gamma^k \times \{R, L, S\}^{k+1})$$

se différencie de la définition des MT déterministes (c.f. déf. 4.1.1). $\Diamond$

Exemple 4.3.1

1. Soit le langage défini par l'ensemble: $L = \{x\texttt{abaab}y \mid x, y \in \{\texttt{a}, \texttt{b}\}^*\}$

   Une machine non déterministe pour L doit choisir au "hasard" où commencer à vérifier si la chaîne `abaab` est présente dans l'entrée ou pas.

2. Soit le langage $L = \{0^n \mid n \text{ est un nombre composé}\}$. Une machine non déterministe qui accepte L doit, d'une manière non déterministe choisir deux nombres p et q et comparer si $0^n = 0^{pq}$ ou non. S'il existe au moins deux des ces nombres, alors 0^n est accepté.

Théorème 4.3.1 *Équivalence entre MT dét. et non dét.*
Si M est une machine de Turing non déterministe acceptant un langage L alors il existe une machine de Turing déterministe M' acceptant également L.

Preuve
La preuve de ce théorème se base sur la construction d'une machine déterministe M' à 3 bandes de travail. Celle-ci simule toutes les dérivations possibles de la machine non déterministe à une seule bande M sur un mot w et accepte w s'il existe au moins une dérivation de M sur w qui mène à un état final de M. Pour réaliser ceci une solution serait de simuler toutes les dérivations une par une. Malheureusement cette solution possède un inconvénient. Si par malchance la simulation du mot w arrive à une dérivation infinie, la simulation ne pourra se terminer alors que w est peut-être accepté par une autre séquence finie de dérivations. Pour éviter ce problème M' va simuler toutes les dérivations possibles dans un ordre particulier. La machine M' simulera tous les chemins de longueur 1 puis 2 puis 3 etc. On peut illustrer ceci par un parcours en largeur de l'arbre des dérivations plutôt qu'en profondeur.

Pour tout couple (état, symbole) de M, il existe un nombre fini de choix pour le prochain état, soit r ce nombre maximal. Ces choix successifs peuvent être représentés par une séquence de chiffres compris chacun entre 1 et r. Ainsi, une séquence de longueur 4 représente un chemin de longueur 4 où chaque chiffre symbolise un choix pour un couple donné.

La machine M' possède 3 bandes. La première pour contenir l'entrée qui n'est jamais modifiée. Dans la seconde M' génère les séquences de 1 à r de manière systématique. L'ordre de simulation doit être capable d'éviter les boucles infinies. Ainsi, les séquences de longueur inférieure doivent apparaître en premier et les séquences de même longueur doivent suivre l'ordre numérique.

La machine M' considère les séquences de nombres les unes après les autres. Pour chaque séquence générée, M' copie l'entrée w dans la troisième bande, puis simule M selon les chiffres composant la séquence. Si M entre dans un état acceptant, alors M' accepte. En revanche, M' n'accepte pas w dans le cas où elle ne trouve pas de séquences menant à un état acceptant. Ainsi, M et M' acceptent le même langage. $\square$

4.4 Grammaires sans restriction

Comme pour les automates à états finis et les automates à pile il existe un équivalent en terme de grammaire. Pour les langages récursivement énumérables on parle de grammaire sans restriction.

Définition 4.4.1 *Grammaire sans restriction*
Une *grammaire sans restriction* est une $G = \langle V_T, V_N, P, S\rangle$ où les productions on la forme $\alpha \to \beta$ avec $\alpha \neq \epsilon$ et α, β sont des séquences de terminaux ou non terminaux. $\Diamond$

Exemple 4.4.1 Soit le langage $L = \{a^i \mid i \text{ est une puissance de } 2 \ \geq 0\}$. Une grammaire sans restriction qui génère ce langage est $G = \langle\{\mathsf{a}\}, \{S, A, X\}, P, S\rangle$ avec $P = \{S \to X\mathsf{a}A, S \to \mathsf{a}, A \to AA, \mathsf{a}A \to A\mathsf{aa}, XA \to X, X \to \epsilon\}$. Cette grammaire génère tout d'abord des mots de la forme $X\mathsf{a}A^n$ puis par la règle $\mathsf{a}A \to A\mathsf{aa}$ produit finalement des mots de la forme $XA^n\mathsf{a}^{2^n}$. Pour terminer, les deux règles encore non utilisées permettent d'éliminer les non-terminaux A et X, ce dernier servant uniquement de délimiteur. Remarquons que la dernière règle peut être appliquée à tout moment et que les mots ainsi obtenus contiennent des non-terminaux alors qu'aucune règle ne peut être appliquée. En fait, ces mots n'appartiennent par définition pas au langage $L(G)$.

4.4.1 Équivalence grammaire sans restriction machine de Turing

Théorème 4.4.1
Si un langage L est généré par une grammaire sans restriction alors L est accepté par une machine de Turing.

Preuve
Le principe consiste à essayer toutes les dérivations possibles de la grammaire jusqu'à ce qu'on arrive au mot w écrit sur la bande d'entrée. On utilise la notion de non-déterminisme afin d'essayer toutes les combinaisons des règles.

Il s'agit de construire une machine de Turing M non déterministe à deux bandes qui reconnaisse le langage L généré par la grammaire sans restriction $G = \langle V_T, V_N, P, S\rangle$. La première bande contiendra l'entrée w et la seconde contiendra α: la partie gauche d'une règle de production de G. M initialise α à S. Les étapes qu'effectuera M sont:

1. d'une manière non déterministe M choisit une position i comprise entre 1 et la longueur de α et déplace sa tête en position i,
2. d'une manière également non déterministe, M choisit une production $\beta \to \gamma$ de G,
3. Si la position i pointe sur β, alors β est remplacé par γ,
4. M compare le contenu de la deuxième bande avec celui de la première, si les deux contenus sont identiques M accepte sinon retourne à l'étape 1.

La machine M accepte donc un mot w si et seulement s'il existe une séquence de dérivations $S \Rightarrow^* w$. Ainsi $L(M) = L(G)$. □

Théorème 4.4.2
Si L est accepté par une machine de Turing alors il existe une grammaire G sans restriction telle que $L = L(G)$.

Preuve
Le principe de la démonstration consiste à construire une grammaire dont les mots générés correspondent aux configurations possibles de la machine de Turing. Une fois qu'on génère une configuration qui accepte le mot d'entrée, la grammaire doit accepter ce mot. Alors, pour pouvoir le générer il faut le mémoriser. Au début la grammaire doit faire deux copies du même mot, simuler la machine sur une copie, et si la machine accepte ce mot, la grammaire doit arriver à la fin avec la deuxième copie du mot.

Soit une machine de Turing $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \Box, F\rangle$ déterministe qui accepte L. Il s'agit de construire une grammaire G sans restriction qui génère d'une "manière non déterministe" deux copies d'une chaîne de Σ^* et simule l'action de M sur une de ces copies.

Soit $G = \langle \Sigma, V_N, P, A_1\rangle$, où $V_N = ((S \cup \{\epsilon\}) \times \Gamma) \cup \{A_1, A_2, A_3\}$, les règles de production de P sont:

1. $A_1 \to q_0 A_2$,
2. $A_2 \to [a, a]A_2$ pour tout a dans Σ,

3. $A_2 \to A_3$,
4. $A_3 \to [e, B]A_3$,
5. $A_3 \to e$,
6. $q[a, X] \to [a, Y]p$ pour chaque $a \in \Sigma \cup \{e\}$, $\forall q \in Q$, et $\forall X, Y \in \Gamma$, tel que $\delta(q, X) = (p, Y, R)$,
7. $[b, Z]q[a, X] \to p[b, Z][a, Y]$, $\forall X, Y, Z \in \Gamma$, $\forall a, b \in \Sigma \cup \{e\}$, et $\forall q \in Q$, tel que $\delta(q, X) = (p, Y, L)$,
8. $[a, X]q \to qaq$, $q[a, X] \to qaq$, et $q \to e$ $\forall a \in \Sigma \cup \{e\}$, $\forall X \in \Gamma$, $\forall q \in F$.

En utilisant les règles 1) et 2) nous obtenons: $A_1 \Rightarrow^* q_0[a_1, a_1][a_2, a_2] \cdots [a_n, a_n]A_2$. Si M accepte une chaîne $a_1a_2 \cdots a_n$ alors il existe un m tel que M utilise au plus m cases de son entrée, en appliquant 3) et 4) nous obtenons $A_1 \Rightarrow * \, q_0[a_1, a_1][a_2, a_2] \cdots [a_n, a_n][\epsilon, \Box]^m$. À ce stade on ne peut appliquer que 6) et 7).

On montre par induction sur le nombre de mouvements que si

$$q_0a_1a_2 \cdots a_n \vdash^* X_1 \cdots X_{r-1}qX_r \cdots X_s$$

alors

$$q_0[a_1, X_1][a_2, X_2] \cdots [a_{r-1}, X_{r-1}]q[a_r, X_r] \cdots [a_{n+m}, X_{n+m}]$$

où les $a_i \in \Sigma$, et $a_{n+1} = a_{n+2} = \cdots = a_{n+m} = e$, $X_i \in \Gamma$, et $X_{s+1} = X_{s+2} = \cdots = X_{n+m} = \Box$.

Supposons l'hypothèse d'induction vraie à l'ordre $k - 1$ mouvement,

$$q_0a_1a_2 \cdots a_n \vdash^{k-1} X_1 \cdots X_{r-1}qX_r \cdots X_s \vdash Y_1 \cdots Y_{t-1}pY_t \cdots Y_u.$$

par hypothèse d'induction,

$$q_0[a_1, a_1][a_2, a_2] \cdots [a_n, a_n][\epsilon, \Box]^m \Rightarrow * \, [a_1, Y_1][a_2, Y_2] \cdots [a_{t-1}, Y_{t-1}]p[a_t, Y_t] \cdots [a_{n+m}, Y_{n+m}]$$

où $Y_i = B$ pour $i > u$.

En utilisant la règle 8), si p appartient à F alors on obtient

$$[a_1, Y_1][a_2, Y_2] \cdots [a_{t-1}, Y_{t-1}]p[a_t, Y_t] \cdots [a_{n+m}, Y_{n+m}] \Rightarrow^* a_1a_2 \cdots a_n$$

.

Ainsi si $w \in L(M)$, alors $A_1 \Rightarrow^* w \in L(G)$. □

4.5 Langage sensible au contexte

Définition 4.5.1 *Langage sensible au contexte*
Un *langage sensible au contexte* est généré par une grammaire qui possède des productions de la forme $\alpha \to \beta$ où la longueur de β est au moins égale à celle de α ($|\alpha| \leq |\beta|$). ◊

Exemple 4.5.1 Le langage $L = \{\mathsf{a}^n\mathsf{b}^n\mathsf{c}^n \mid n \geq 1\}$ est engendré par la grammaire

$$G = \langle \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}, \{S, A, B, C, D\}, P, S \rangle$$

où $P = \{S \to \mathsf{a}ACD, A \to \mathsf{a}AC, A \to \epsilon, B \to \mathsf{b}, CD \to BD\mathsf{c}, CB \to BC, D \to \epsilon\}$. Cette grammaire n'est pas sensible au contexte car les deux productions $A \to \epsilon$ et $D \to \epsilon$ ne correspondent pas à la définition ($|\epsilon| = 0$). En revanche, L est donc bien sensible au contexte car il existe bel et bien une grammaire sensible au contexte qui engendre ce dernier. Pour obtenir cette grammaire il suffit d'éliminer les deux productions en question sans oublier de répercuter sur les autres productions cette élimination.

4.5.1 Automate borné linéairement

Un *automate borné linéairement* (ABL) est une machine de Turing non déterministe pour laquelle il existe deux constantes k_1 et k_2, telles que lors de toute exécution sur un mot d'entrée w, la machine n'utilise jamais plus de $k_1|w| + k_2$ cases des bandes de travail.

Théorème 4.5.1 *Équivalence ABL et grammaires sensibles au contexte*
Si L est un langage sensible au contexte, alors il existe un ABL M tel que $L = L(M)$.

Preuve
La preuve est essentiellement identique à celle du théorème 4.4.1. La différence se situe dans la phase 4). Dans le cas présent, la nature particulière des règles de production permet de rejeter l'entrée w si l'a longueur de la deuxième bande excède la longueur de w. Ainsi, la machine M accepte le mot w si et seulement si une séquence de dérivations $S \Rightarrow^* w$ existe, mais il ne peut y avoir une séquence de dérivations $S \Rightarrow^* \alpha \Rightarrow^* w$ avec $|\alpha| > |w|$. La machine M utilise donc au maximum un nombre de cases égal à $|w|$ ($k_1 = 1$ et $k_2 = 0$). Ainsi M est un ABL et $L(M) = L(G) = L$. □

Théorème 4.5.2
Si $L = L(M)$, M étant un ABL alors $L \setminus \{\epsilon\}$ est un langage sensible au contexte.

Preuve
Difficile et fastidieuse, cf. références. □

4.6 Résumé

Avec ce chapitre nous avons terminé notre étude des langages formels et nous sommes prêts à étudier la notion de calculabilité et la notion de complexité des problèmes. Il reste à remarquer que les quatre classes des langages étudiés forment ce que l'on appelle la *hiérarchie de Chomsky*. Chaque classe de langage possède son propre modèle opérationnel (automate) et sa propre caractérisation grammaticale. Ces quatre classes sont:

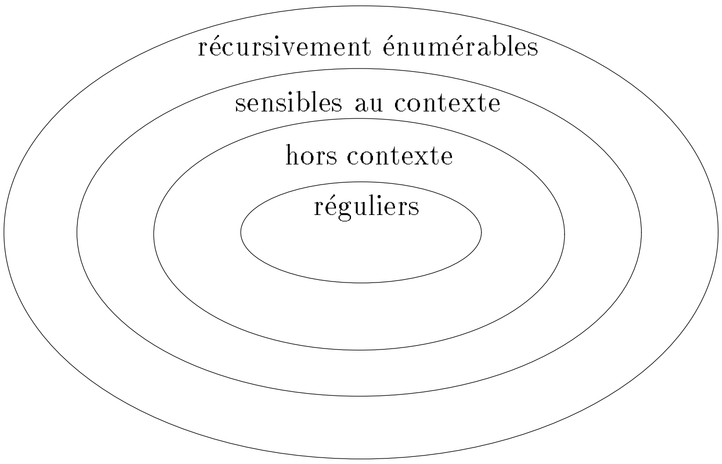
3- les langages réguliers,

2- les langages hors-contexte,

1- les langages sensibles au contexte,

0- les langages récursivement énumérables.

On peut résumer l'inclusion stricte de ces classes par la figure suivante:



Chapitre 5

Calculabilité

Parmi les automates ou les systèmes de réécriture que nous avons étudiés jusqu'à présent, la machine de Turing et les grammaires sans restriction sont les plus puissants dans le sens où ils peuvent traiter une classe plus large de problèmes que leur prédécesseurs. Nous avons vu au moyen des machines de Turing que les notions de problème et de langage sont intimement liées. Une question fondamentale reste posée: existe-t-il des problèmes qu'une machine de Turing ne puisse résoudre? Dans ce chapitre nous allons continuer notre classification des problèmes pour nous apercevoir que de tels problèmes existent et en déduire qu'aucune machine existante ne peut les résoudre.

5.1 Quelques définitions

Un langage est *décidé* par une machine de Turing s'il existe une machine de Turing qui accepte le langage et qui s'arrête toujours. En d'autres termes, une telle machine acceptera ou rejettera un mot donné dans un temps fini et ceci quelque soit les mots que l'on lui soumettra.

Un langage est *accepté* par une machine de Turing s'il existe une machine de Turing qui accepte dans un temps fini tous les mots appartenant au langage. Une telle machine ne rejettera pas forcément un mot n'appartenant pas au langage et peut ne jamais atteindre un état final dans ce type de situations.

On dit qu'un langage est *récursivement énumérable* si ce langage est accepté par une machine de Turing. Le langage est dit *récursif* s'il est décidé par une machine de Turing.

Un *algorithme* est constitué d'un ensemble fini d'instructions décrivant le déroulement des opérations à réaliser pour résoudre un certain problème. Pour un algorithme, ce déroulement des opérations doit toujours se terminer dans un temps fini.

Un problème pour lequel il existe une machine de Turing s'arrêtant toujours est dit *décidable* autrement il est dit *indécidable*.

5.2 Hypothèse de Church

Les machines de Turing présentent toutes les caractéristiques essentielles des ordinateurs réels. On peut donc considérer la machine de Turing comme une formalisation précise de la notion intuitive de calculabilité. Ce principe est connu sous le nom d'hypothèse de Church ou thèse de Church-Turing. Cette hypothèse n'est pas un théorème car elle ne découle pas d'un raisonnement mathématique. L'hypothèse de Church fait simplement une association entre la notion informelle de calculabilité et la notion formelle d'une machine de Turing. C'est une hypothèse comme l'hypothèse d'Einstein à propos de la vélocité de la lumière. Il se peut que, dans le futur, une nouvelle notion de calculabilité remplace la machine de Turing comme la mécanique newtonienne a été supplantée par la mécanique d'Einstein. Jusqu'à présent, rien ne nous a permis de mettre en doute la thèse de l'hypothèse de Church et tout laisse à croire qu'une telle affirmation est raisonnable.

En termes de langages on peut exprimer l'hypothèse de Church comme:

Un langage est calculable si et seulement si ce langage est récursif.

5.3 Autres modèles

Les logiciens ont présenté différents modèles de calculabilité tels que les fonctions récursives partielles et totales, qui correspondent respectivement aux langages récursivement énumérables et récursifs. Le λ-Calcul, les post-systèmes, les fonctions récursives générales, les machines RAM, etc. Tous ces modèles sont équivalents aux machines de Turing et l'hypothèse de Church reste valable.

Par exemple, on dit qu'une fonction est une *fonction récursive partielle* si cette fonction peut être évaluée par une machine de Turing. En plus, une fonction est *récursive totale* si elle est récursive partielle et définie pour *toutes les entrées possibles*. Les fonctions récursives partielles sont équivalentes aux langages *récursivement énumérables*, parce qu'elles sont évaluées par des machines qui peuvent ne pas s'arrêter pour toutes les entrées. Les fonctions récursives totales correspondent aux langages *récursifs* parce qu'elles sont évaluées par des machines qui s'arrêtent toujours. On pourrait transformer l'hypothèse de Church pour les fonctions en disant:

Une fonction est calculable si et seulement si cette fonction est récursive totale.

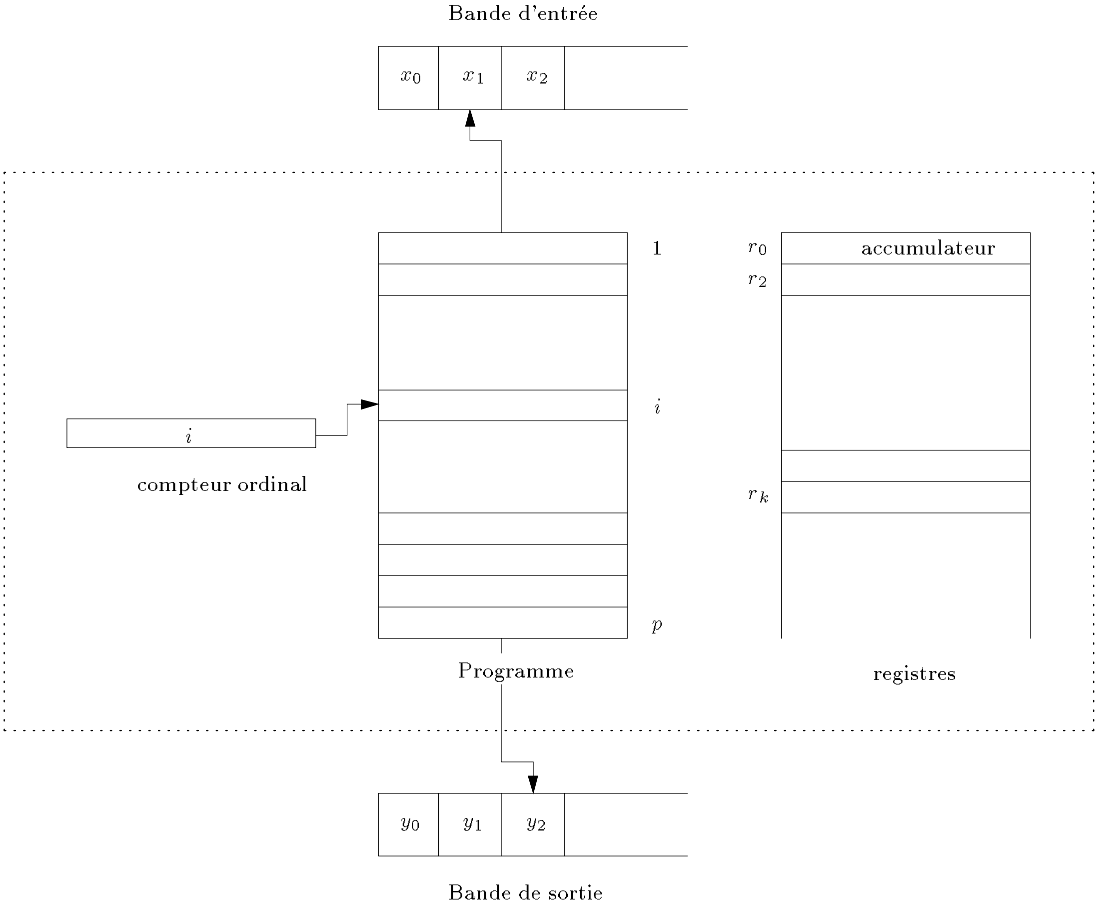
5.4 Machine RAM

Tout comme la machine de Turing, la machine RAM (Random Access Machine) est un modèle simple de calcul relativement proche d'une unité centrale d'un ordinateur actuel. En effet, une machine RAM est constituée de:

1. une bande d'entrée et une bande de sortie dont les cases sont des entiers arbitrairement grands,
2. un nombre de mémoires ou *registres* infini pouvant stocker chacun un nombre entier arbitrairement grand, les registres sont numérotés $r_0, r_1, \ldots, r_k, \ldots$, le registre r_0 joue le rôle particulier d'*accumulateur*,
3. une zone finie inaltérable de mémoires numérotées contenant les instructions composant le programme,
4. un registre *compteur ordinal* indiquant le numéro (ou l'*adresse*) de la mémoire programme courante.

Le jeu d'instructions de la machine vaut:

Incrémenter	*op*	Décrémenter	*op*
Ajouter	*op*	Soustraire	*op*
Multiplier	*op*	Diviser	*op*
Charger	*op*	Ranger	*op*
Lire	*op*	Écrire	*op*
Saut	*adr*	Saut_si_positif	*adr*
Saut_si_zéro	*adr*	Arrêt	



Les instructions sont composées de deux parties: la partie *code* et la partie *opérande*. Pour les instructions de rupture de séquence, l'*adresse* correspond au numéro de la prochaine instruction à exécuter. Autrement, les opérandes sont, soit un entier indiquant le numéro du registre concerné, soit un entier précédé de la lettre 'A' qui signifie absolu, soit d'un entier précédé de la lettre 'I' pour indirection. Pour une instruction d'entrée, l'opérande est transféré dans la case de la tête de lecture ou d'écriture en déplaçant d'une position vers la droite la tête en question.

La sémantique ou la signification de chaque instruction est donnée par le tableau suivant:

Incrémenter	n	Acc $\leftarrow [n] + 1$
Décrémenter	n	Acc $\leftarrow [n] - 1$
Ajouter	n	Acc $\leftarrow$ [Acc] $+ [n]$
Soustraire	n	Acc $\leftarrow$ [Acc] $- [n]$
Multiplier	n	Acc $\leftarrow$ [Acc] $* [n]$
Diviser	n	Acc $\leftarrow$ [Acc]$/[n]$
Charger	n	Acc $\leftarrow [n]$
Charger A	n	Acc $\leftarrow$ la valeur de n
Charger I	n	Acc $\leftarrow [[n]]$
Ranger	n	$n \leftarrow$ [Acc]
Ranger I	n	$[n] \leftarrow$ [Acc]
Lire	n	$[n] \leftarrow$ entier sous la tête et déplacement 1 à droite
Écrire	n	écrire $[n]$ sous la tête et déplacement 1 à droite
Saut	a	Co $\leftarrow a$

Saut_si_positif	a	Co $\leftarrow a$ si [Acc] ≥ 0
		Co $\leftarrow$ Co + 1 sinon
Saut_si_zéro	a	Co $\leftarrow a$ si [Acc] $= 0$
		Co $\leftarrow$ Co + 1 sinon
Arrêt		Arrêt de l'exécution

Les notations $[n]$ et $[[n]]$ représentent respectivement le contenu du registre r_n et le contenu du registre ayant pour numéro l'entier contenu dans le registre r_n. Acc représente le numéro 0 (accumulateur) et Co représente le compteur ordinal.

Il est aisé de voir qu'un programme RAM définit une *fonction récursive partielle*. De plus, si le programme s'arrête pour toutes les entrées on a affaire à une *fonction récursive totale*. On peut aussi voir dans un programme RAM un accepteur de langages. Le mot w écrit sur la bande de lecture est accepté par la RAM si le programme de la RAM lit w totalement, écrit un 1 sur la bande de sortie et s'arrête. Le *langage* accepté par le programme est l'ensemble de tous les mots acceptés par ce programme. L'hypothèse de Church est encore valable pour la RAM et pour la notion d'algorithme associé au concept de RAM; car un langage est *récursivement énumérable* si et seulement s'il est accepté par une RAM et un langage est *récursif* si et seulement s'il est accepté par une RAM qui s'arrête toujours. Pour démontrer l'équivalence entre les langages acceptés par une RAM et ceux acceptés par les machines de Turing, on montre comment simuler une RAM par une machine de Turing et vice-versa.

Théorème 5.4.1
Tout langage accepté par une machine de Turing est accepté par une RAM.

Preuve
exercice. □

Théorème 5.4.2
Tout langage accepté par une RAM est accepté par une machine de Turing.

Preuve
exercice. □

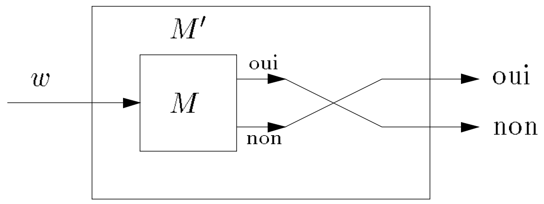
5.5 Propriétés

Quelques propriétés importantes des langages récursifs et des langages récursivement énumérables sont énoncées et démontrées dans les théorèmes qui suivent. La plupart des démonstrations sont accompagnées d'illustrations pour lesquelles les convention suivantes ont été adoptées: les rectangles correspondent à des machines de Turing, les flèches représentent les flots de données ou de contrôle alors que les mots "oui", "non" et "début" représentent respectivement l'acceptation, le rejet et le déclenchement de l'exécution d'une MT.

Théorème 5.5.1
Le complément d'un langage récursif est récursif.

Preuve
Soient L un langage récursif et M une machine de Turing qui s'arrête pour toutes ses entrées et qui décide L. On construit M' telle que, si M s'arrête en acceptant une entrée w, alors M' s'arrête sans l'accepter; inversement, si M s'arrête sans accepter w, M' s'arrête en l'acceptant. Ainsi la machine

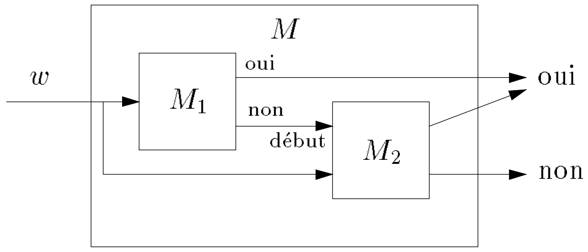
construite M' accepte tous les mots de $\overline{L}$ et fournit une réponse négative dans tous les autres cas; $\overline{L}$ est donc bien récursif. Cette construction peut être facilement représentée graphiquement comme suit:



□

Théorème 5.5.2
L'union de deux langages récursifs est un langage récursif.

Preuve
Soient L_1 et L_2 deux langages récursifs décidés respectivement par M_1 et M_2. Soit M la machine qui simule tout d'abord M_1, si M_1 accepte alors M accepte sinon M déclenche la simulation de M_2 et accepte si et seulement si M_2 accepte; nous pouvons dire alors que M décide $L_1 \cup L_2$.



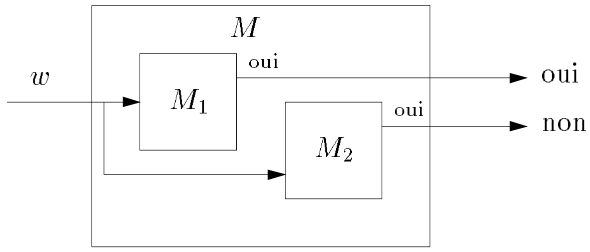
□

Théorème 5.5.3
L'union de deux langages récursivement énumérables est un langage récursivement énumérable.

Preuve
exercice. □

Théorème 5.5.4
Si L et $\overline{L}$ sont récursivement énumérables alors L est récursif (tout comme $\overline{L}$).

Preuve
Soient M_1 et M_2 deux machines acceptant respectivement L et $\overline{L}$. Il s'agit de construire M qui simule simultanément M_1 et M_2 tel que M accepte w si M_1 accepte w et ne l'accepte pas si M_2 l'accepte. Ainsi M répondra toujours par oui ou par non. C'est donc un algorithme qui décide L, L est alors récursif.



□

Des théorèmes 5.5.1 et 5.5.4 on obtient les conséquences suivantes: soient L et $\overline{L}$ deux langages complémentaires, on a alors soit

1. L et $\overline{L}$ sont récursifs,
2. ni L ni $\overline{L}$ ne sont récursivement énumérables,
3. si l'un des langages L ou $\overline{L}$ est récursivement énumérable mais pas récursif, alors son complément n'est pas récursivement énumérable.

Conclusion: Pour prouver qu'un langage L n'est pas récursif (décidable) il suffit de prouver que $\overline{L}$ n'est pas récursivement énumérable, c'est-à-dire qu'aucune machine de Turing n'existe pour accepter le langage. Cette remarque va être employée pour caractériser notre premier langage non récursivement énumérable.

5.6 Codage des machines de Turing

Avant de construire notre premier langage non récursivement énumérable, nous allons montrer que l'on peut attribuer un code à chaque machine de Turing. Pour simplifier considérons une machine de Turing M acceptant des mot $w \in \{0,1\}$ et dont l'alphabet de la bande d'entrée est augmenté du symbole $\Box$, M est donc caractérisée par:

$$M = \langle Q, \{0,1\}, \{0,1,\Box\}, \delta, q_1, \Box, \{q_2\}\rangle$$

La fonction de transition de cette machine est de la forme:

$$\delta(q_i, X_j) = (q_k, X_l, D_m) \quad \text{avec } X_j, X_l \in \{0,1,\Box\}, D_m \in \{R, L\},$$

chacune de ces transitions sont encodées comme suit:

$$0^i 1 0^j 1 0^k 1 0^l 1 0^m.$$

Finalement la machine M codée en binaire aura la forme:

$$111 \text{ code}_1 \ 11 \text{ code}_2 \ 11 \ \cdots \ 11 \text{ code}_r \ 111.$$

Exemple 5.6.1 Soit $M = \{\{q_1, q_2, q_3\}, \{0,1\}, \{0,1,\Box\}, \delta, q_1, \Box, \{q_2\}\}$ et

$$\begin{array}{rcl} \delta(q_1, 1) & = & (q_3, 0, R), \\ \delta(q_3, 0) & = & (q_1, 1, R), \\ \delta(q_3, 1) & = & (q_2, 0, R), \\ \delta(q_3, \Box) & = & (q_3, 1, L). \end{array}$$

En codant q_1 par 0, q_2 par 00, q_3 par 000, 0 par 0, 1 par 00, $\Box$ par 000, L par 0 et R par 00 le code de M vaut:

$$1110100100010100110001010100100110001001001010011000100010001001 0111$$

Remarques :

1. En définitive, quelque soit l'alphabet de la machine de Turing il est possible de lui attribuer un code binaire et nous ne perdons rien en généralité en ne considérant que des machines de Turing travaillant sur un alphabet binaire. Le codage choisi n'est évidemment pas le seul possible.

2. A toute machine de Turing correspond donc un code mais tout code binaire ne représente pas une MT. En revanche, à tout code représentant une machine de Turing correspond une unique machine.

3. L'énumération des mots binaires par ordre croissant de leur représentation numérique est appelée énumération *canonique* alors que l'énumération des machines selon le même critère est appelée énumération de *Gödel*.

4. L'ensemble des machines de Turing est un ensemble dénombrable.

5.7 Langages diagonalisables

Dans cette section nous allons construire un langage L_d dit langage diagonal et montrer que ce dernier n'est pas récursivement énumérable.

Soit une table infinie à deux indices. Verticalement sont énumérés par ordre canonique les mots et horizontalement sont énumérées les machines de Turing. L'indice i pour les lignes représente donc le code binaire du mot $w_i \in \{0,1\}^*$ et l'indice j pour les colonnes représente le code binaire de la machine M_j. La valeur de la table d'indices (i,j) vaut 0 si $w_i \notin L(M_j)$ et 1 si $w_i \in L(M_j)$.

En prenant les valeurs de la diagonale nous construisons le langage L_d qui satisfait la condition:

$$L_d = \{w_i \mid \text{valeur de la table d'indice } (i,i) = 0\},$$

c'est-à-dire l'ensemble des mots w_i qui ne sont pas acceptés par la machine M_i. Cette condition implique qu'aucune machine de Turing n'accepte le langage L_d comme l'exprime le théorème qui suit.

Théorème 5.7.1
Le langage L_d n'est pas récursivement énumérable.

Preuve
Raisonnons par l'absurde et supposons qu'une machine M_j accepte le langage L_d, $L_d = L(M_j)$. Dans ce cas, pour un mot $w_j \in L_d$ on a, par définition de L_d, que l'entrée $(j,j) = 0$ or, par définition de la table, ceci implique que $w_j \notin L(M_j)$ ce qui contredit $L_d = L(M_j)$. Inversement, si $w_j \notin L_d$ l'entrée $(j,j) = 1$ mais ceci implique que $w_j \in L(M_j)$ ce qui contredit de nouveau $L_d = L(M_j)$. Par conséquent aucune machine de Turing accepte L_d et donc L_d n'est pas récursivement énumérable.
□

Corollaire 5.7.1
Le langage $\overline{L_d}$ n'est pas récursif.

Preuve
$\overline{L_d}$ n'est pas récursif sinon par le théorème 5.5.1 L_d le serait aussi ce qui n'est pas le cas. □

Théorème 5.7.2
Le langage $\overline{L_d}$ est récursivement énumérable.

Preuve
exercice □

5.8 Langage universel et machine universelle

Définition 5.8.1
On note par $\langle M, w\rangle$ le code du mot w juxtaposé au code de la machine de Turing M. Ce code $\langle M, w\rangle$ peut alors très bien correspondre à l'entrée d'une autre machine de Turing. $\Diamond$

Définition 5.8.2 *Langage universel*
On appelle *langage universel* le langage $L_u = \{\langle M, w\rangle \mid M \text{ accepte } w\}$. Le langage L_u correspond à l'ensemble de tous les couples (mots) $\langle M, w\rangle$ tels que la machine M accepte le mot w. $\Diamond$

Le langage universel correspond au problème: "Est-ce qu'une machine de Turing M accepte un mot w". Les deux théorèmes qui suivent démontrent que le langage L_u est récursivement énumérable mais pas récursif. Le problème correspondant à L_u est donc indécidable, c'est-à-dire qu'aucune machine de Turing ayant comme entrée le code d'une machine M suivit du code d'un mot w ($\langle M, w\rangle$) ne peut déterminer si oui ou non la machine M accepte le mot w.

Théorème 5.8.1
Le langage universel L_u est récursivement énumérable.

Preuve
Construisons une machine M_u acceptant le langage L_u. L'entrée de cette machine est un mot de la forme $\langle M, w\rangle$ et son rôle consiste à simuler la machine M sur le mot w. Les mouvements de M sont décrits par la chaîne codée, précédemment définie par:

$$111 \text{ code}_1 \ 11 \text{ code}_2 \ 11 \ \cdots \ 11 \text{ code}_r \ 111.$$

M_u doit analyser le mot w en utilisant les codes qui représentent la fonction de transition de M.

La première bande de M_u est la bande d'entrée, sa tête est utilisée pour localiser les mouvements de M lorsque l'entrée est $\langle M, w\rangle$. La seconde bande de M_u simule la bande de M. L'alphabet de M est $\{0, 1, B\}$. La troisième bande de M_u contient l'état de M, q_i représenté par 0^i.

Le fonctionnement de M_u est décrit par les étapes suivantes:

1. Vérifier que la chaîne en entrée est codée correctement et que si $0^i10^j10^k10^l10^m$ est un code, alors $1 \leq j \leq 3, 1 \leq l \leq 3,$ et $1 \leq m \leq 2$.

2. Initialiser la deuxième bande avec le mot w, initialiser la troisième bande avec le symbole 0. Le symbole 0 représente l'état q_1. Les trois têtes des bandes sont positionnées sur les symboles les plus à gauche.

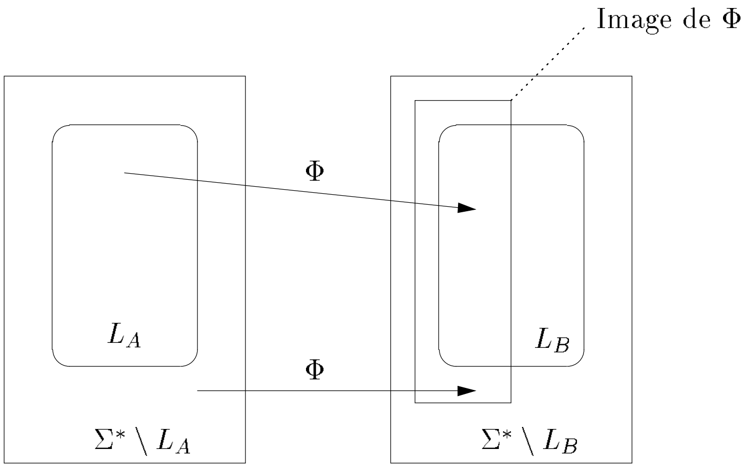
3. Si la troisième bande contient le code de l'état acceptant alors M_u accepte et s'arrête.

4. Soit X_j le symbole courant analysé par la deuxième bande, et soit 0^i le contenu courant de la troisième bande. M_u va chercher dans la première bande une sous-chaîne qui commence par 110^i10^j1 entre les deux premiers 111. Si cette chaîne n'est pas trouvée, alors elle s'arrête en rejetant. Sinon, cette sous-chaîne est suivie de $10^k10^l10^m$, 0^k est alors écrit sur la troisième bande, X_l est écrit sur la case analysée par la deuxième tête et est déplacée dans la direction de D_m.

En analysant le comportement de M_u, on remarque qu'il est similaire à celui de M.En effet M_u accepte $\langle M, w\rangle$ si et seulement si M accepte w; si M boucle indéfiniment sur w, alors M_u bouclera indéfiniment sur $\langle M, w\rangle$, et si M s'arrête en rejetant w alors M_u s'arrête en rejetant $\langle M, w\rangle$. On appelle M_u machine universelle. $\square$

5.8.1 Réduction d'un problème à un autre

Pour un alphabet Σ donné, un problème A se *réduit* à un problème B s'il existe une machine de Turing s'arrêtant toujours et réalisant la fonction Φ qui met en correspondance tous les éléments du langage L_A avec des éléments de L_B et tous les éléments de $\Sigma^* \setminus L_A$ avec des éléments de $\Sigma^* \setminus L_B$. On note cette réduction $A \leq_M B$ qui est aussi appelée réduction "many-one".



En d'autres termes, $A \leq_M B$ si il existe une fonction récursive totale Φ telle que $w \in L_A \iff \Phi(w) \in L_B$.

Proposition 5.8.1
En supposant que le problème A se réduit au problème B, si B est décidable alors A est décidable.

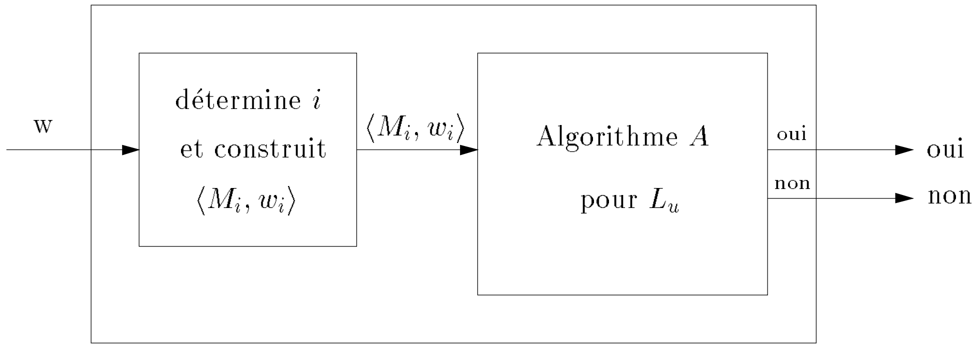
Preuve
Soient M la machine de Turing qui décide L_B et M' la machine qui effectue la réduction (Φ) du problème A au problème B. On voit facilement que la cascade M puis M' est une machine qui s'arrête toujours et qui décide L_A. □

En raisonnant par l'absurde ce principe de réduction va nous permettre, à partir d'un langage indécidable connu, de prouver qu'un autre problème l'est également.

Théorème 5.8.2
Le langage L_u n'est pas récursif.

Preuve
Supposons qu'il existe un algorithme A décidant L_u, en réduisant le langage $\overline{L_d}$, que l'on sait non récursif par le théorème 5.5.1, à L_u nous allons en déduire une contradiction ce qui nous permettra de conclure que L_u n'est pas décidable.

La fonction Φ de notre principe de réduction reviendra à déterminer à quel indice i correspond un mot $w = w_i$ par énumération canonique des mots. Puis, par énumération de Gödel, on détermine la machine M_i. Ces deux opérations se terminent bien évidemment dans tous les cas. On dispose donc d'un algorithme pour $\overline{L_d}$ ce qui est en contradiction avec le théorème précédent. On en déduit alors que L_u n'est pas décidable ou récursif.



Algorithme pour $\overline{L_d}$

□

5.9 D'autres exemples de problèmes indécidables

5.9.1 Halting problem

Déterminer si une machine de Turing M s'arrête sur une entrée w est un problème indécidable. Ce problème est connu sous le nom du "Halting problem"

Théorème 5.9.1 *Halting problem*
Le langage $L_h = \{\langle M, w \rangle \mid M \text{ s'arrête sur } w\}$ n'est pas récursif.

Preuve
Exercice. □

5.9.2 Le langage accepté par une MT est-il non vide?

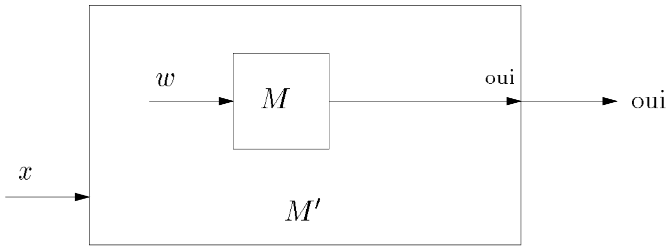
Le problème de déterminer si le langage accepté par une machine M n'est pas vide ($L(M) \neq \emptyset$) correspond au langage $L_{ne} = \{\langle M \rangle \mid L(M) \neq \emptyset\}$.

Théorème 5.9.2
Le langage $L_{ne} = \{\langle M \rangle \mid L(M) \neq \emptyset\}$ est récursivement énumérable mais pas récursif.

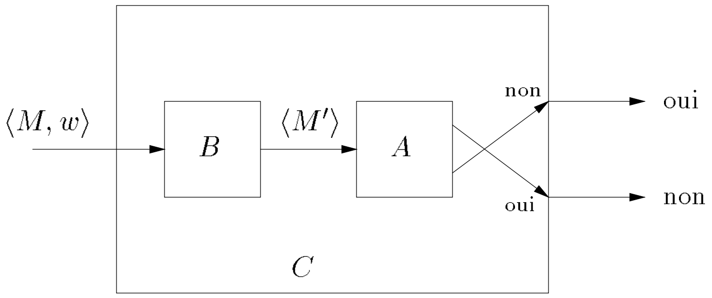
Preuve
Pour montrer que L_{ne} est récursivement énumérable, il faut construire une machine de Turing M_{ne} qui reconnaisse les codes des machines qui acceptent un ensemble non vide de mots.

Pour un code $\langle M_i \rangle$ en entrée, M_{ne} choisit, de manière non déterministe, une chaîne x acceptée par M_i et vérifie que x est acceptée par M_i en simulant M_i avec l'entrée x. Si M_i accepte alors M_{ne} accepte $\langle M_i \rangle$.

Montrons maintenant que le complément de L_{ne}, noté $L_e = \{\langle M \rangle \mid L(M) = \emptyset\}$, n'est pas récursif. En supposant que L_e est récursif et en effectuant une réduction à partir de L_u il va apparaître une contradiction. Soit A l'algorithme hypothétique décidant L_e. Soit B un algorithme qui, étant donné une entrée $\langle M, w \rangle$, construit une nouvelle MT M'. Cette machine produite par l'algorithme B se comportera comme suit: M' fonctionne selon le schéma suivant: M' accepte $\varnothing$ (n'accepte aucun mot) si M n'accepte pas w et accepte $x \in \Sigma^*$ si M accepte w.



Une fois ceci établi, on construit un algorithme C décidant le langage L_u.



D'après le théorème précédent nous savons que cet algorithme C n'existe pas, par conséquent l'algorithme hypothétique A n'existe pas et donc L_e n'est pas récursif.

D'après les propriétés des langages et leur complément énoncés précédemment, L_{ne} est récursivement énumérable mais pas récursif. $\square$

5.9.3 Le langage accepté par une MT est-il récursif?

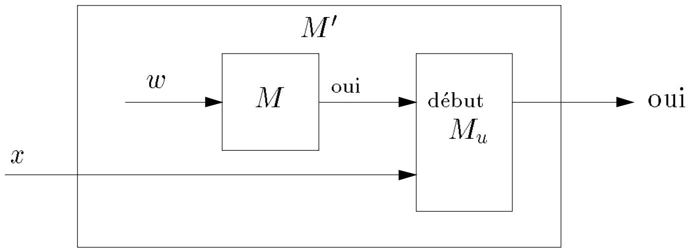
Tous les langages non récursivement énumérables vu jusqu'à présent leur langage complément était récursivement énumérable (pas récursif évidemment). Les deux langages ci-dessous ont la particularité d'être complémentaires et tous deux non récursivement énumérables.

$$L_r = \{\langle M\rangle \mid L(M) \text{ est récursif}\} \qquad L_{nr} = \{\langle M\rangle \mid L(M) \text{ est non récursif}\}$$

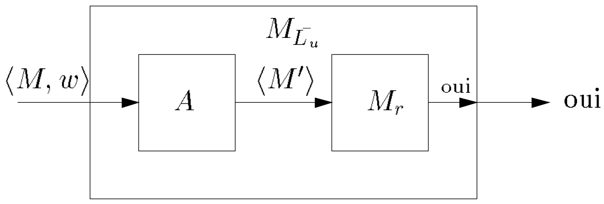
Afin de montrer que L_r n'est pas récursivement énumérable nous allons supposer qu'il l'est et effectuer une réduction du langage $\bar{L_u}$ à L_r. Construisons une MT acceptant $\bar{L_u}$ tout en sachant que cette dernière n'existe pas car L_u n'est pas récursif. Soit M_r une MT acceptant L_r, nous pouvons construire un algorithme A qui, pour une entrée de la forme $\langle M, w\rangle$, produit une machine M' telle que

$$L(M') = \begin{cases} \varnothing & \text{si } M \text{ n'accepte pas } w \\ L_u & \text{si } M \text{ accepte } w \end{cases}$$

Il faut noter que puisque L_u n'est pas récursif la machine M' accepte un langage récursif, le langage vide, si et seulement si M n'accepte pas w.



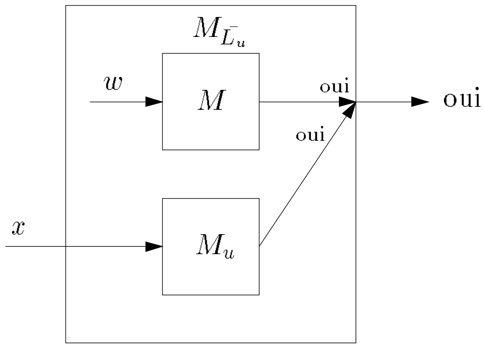
La machine ci-dessous correspond à la machine hypothétique pour $\bar{L}_u$



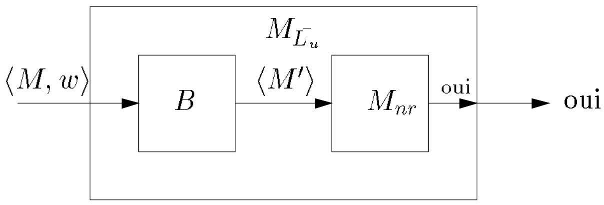
Pour une entrée $\langle M, w\rangle$ l'algorithme A produit la machine M'. M_r est employée afin de déterminer si l'ensemble accepté par M' est récursif. La machine $M_{\bar{L}_u}$ accepte son entrée si et seulement si $L(M')$ est récursif. Mais $L(M')$ est récursif ssi $L(M') = \emptyset$ ce qui signifie que M n'accepte pas w. Ainsi, notre machine hypothétique pour $\bar{L}_u$ accepte $\langle M, w\rangle$ ssi $\langle M, w\rangle$ appartient à $\bar{L}_u$, d'où contradiction.

Les choses sont similaires pour le langage L_{nr}. On suppose qu'il existe une machine de Turing M_{nr} qui accepte L_{nr}, et un algorithme B qui, pour une entrée $\langle M, w\rangle$, produit une machine M' avec les caractéristiques suivantes:

$$L(M') = \begin{cases} \Sigma^* & \text{si } M \text{ accepte } w \\ L_u & \text{si } M \text{ n'accepte pas } w \end{cases}$$

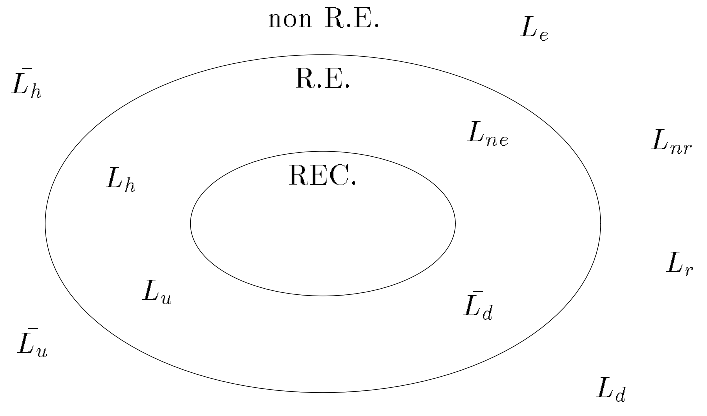


Notons que M' accepte un langage récursif si et seulement si M accepte w. A partir de l'algorithme B on construit une machine de Turing $M'_{\bar{L}_u}$ pour le langage $\bar{L}_u$.



En tenant le même raisonnement que précédemment on vérifie que la machine $M'_{\bar{L_u}}$ accepte $\langle M, w\rangle$ ssi $\langle M, w\rangle \in \bar{L_u}$ et l'on conclut une contradiction. Puisqu'aucune machine n'accepte $\bar{L_u}$ alors L_r et L_{nr} ne sont pas récursivement énumérables.

Voici un résumé de l'appartenance de ces langages un peu particuliers. La classe des langages récursifs est abrégée par "REC.", celle des langages récusivement énumérables par "R.E.", tandis que les langages qui ne sont pas récursivement récusivement énumérables sont classés dans "non R.E.".



5.10 Théorèmes de Rice

5.10.1 Théorème de Rice pour les langages récursifs

A l'aide de la technique de réduction nous avons montré que les problèmes "le langage accepté par une machine est-il vide?" et "le langage accepté par une machine est-il récursif?" sont indécidables. Dans ces deux cas, la propriété qu'un langage accepté par une machine soit vide ou récursif débouche sur un problème indécidable. Le théorème qui va suivre généralise ce résultat pour une propriété quelconque.

Définition 5.10.1 *Propriété de langages récursivement énumérables*
On appelle *propriété de langages récursivement énumérables* un ensemble P de langages r.e. Un langage L possède une propriété P si L est un élément de P. On dit que P est une propriété *triviale* si P est vide ou si P correspond à l'ensemble de tous les langages r.e. On note par L_P le langage $\{\langle M\rangle \mid L(M) \in P\}$. ◊

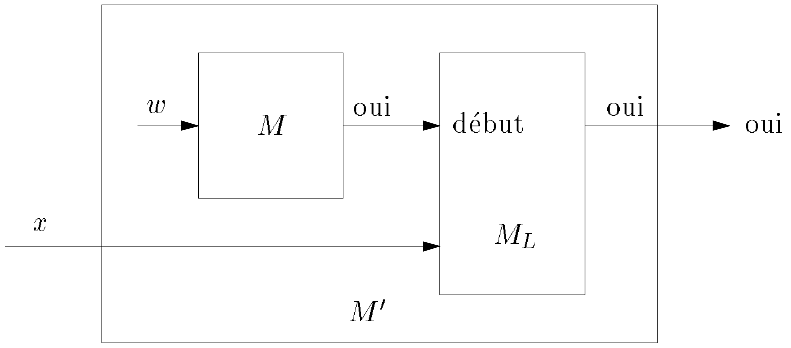
Théorème 5.10.1
Toute propriété P non triviale de langages récursivement énumérables est indécidable ($L_P = \{\langle M\rangle \mid L(M) \in P\}$ n'est pas récursif).

Preuve
Sans perte de généralité nous pouvons supposer que le langage vide n'appartient pas à P (si c'est le cas on peut très bien considérer $\overline{P}$ et tenir le même raisonnement grâce au théorème 5.5.1). Puisque P est non triviale il existe alors un langage L avec la propriété P.

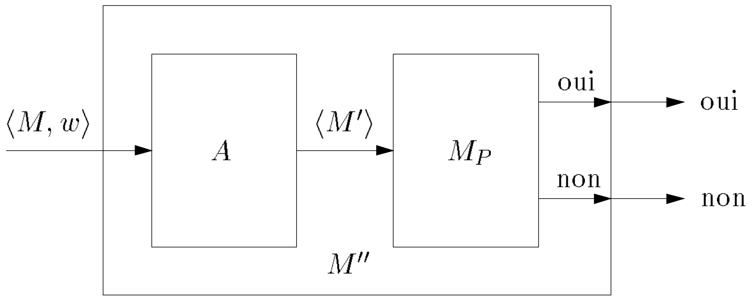
Soit M_L une machine de Turing acceptant L. Supposons que la propriété P soit décidable, il existe alors un algorithme M_P décidant L_P. Construisons un algorithme pour le langage L_u au moyen de M_L et M_P.

Tout d'abord on construit un algorithme A qui possède l'entrée $\langle M, w\rangle$ et délivre en sortie $\langle M'\rangle$ de façon à ce que $L(M')$ appartienne à P si et seulement si M accepte w. Autrement formulé, $L(M')$ appartient à P si et seulement si $\langle M, w\rangle \in L_u$.

Le comportement de M' est le suivant: dans un premier temps M' ignore son entrée et simule M sur w. Si M n'accepte pas w, alors M' n'accepte pas x. Si M accepte w, alors M' simule M_L sur x, et accepte x si et seulement si M accepte w. Précisons que dans le cas où M n'accepte pas w le langage accepté par M' correspond au langage vide et ce langage n'appartient pas à P par notre hypothèse de départ.



En utilisant notre algorithme A et l'algorithme hypothétique M_P nous pouvons construire un algorithme M'' qui décide L_u. Pour une entrée $\langle M, w\rangle$ l'algorithme A produit la machine M' qui correspond à l'entrée de M_P. Puisque $L(M')$ appartient à P si et seulement si $\langle M, w\rangle \in L_u$, M'' décide bien L_u.



On en déduit donc une contradiction et l'on conclut que le langage L_P n'est pas récursif, ce qui revient à dire que la propriété P est indécidable. □

Corollaire 5.10.1
Parmi les propriétés des langages récursivement énumérables qui sont indécidables citons :

1. $L = \varnothing$,
2. L est fini,
3. L est régulier,
4. L est hors contexte.

5.10.2 Théorème de Rice pour les langages récursivement énumérables

Le théorème précédent affirme que pour toute propriété P le langage $L_P = \{\langle M\rangle \mid L(M) \in P\}$ n'est pas récursif. Les conditions sur P pour que L_P soit récursivement énumérable sont nettement plus complexes.

Théorème 5.10.2
L_P est récursivement énumérable si et seulement si P satisfait les trois conditions:

1. si L appartient à P et $L \subseteq L'$ pour tout L' récursivement énumérable alors L' appartient à P,
2. si L est un langage infini appartenant à P alors il existe un langage $L' \subset L$ tel que $L' \in P$,
3. l'ensemble des langages finis appartenant à P est dénombrable.

Preuve
Démonstration fastidieuse et compliquée (c.f. références). □

Corollaire 5.10.2
Les propriétés suivantes ne sont pas récursivement énumérables :

1. $L = \varnothing$,
2. $L = \Sigma^*$,
3. L est récursif,
4. L n'est pas récursif,
5. L est composé d'un seul mot,
6. L est un ensemble régulier,
7. $L \setminus L_u \neq \varnothing$.

Pour les cas a), c), d), e) et f) la condition 1) du théorème n'est pas satisfaite. Pour b) la condition 2) n'est pas satisfaite et pour g) c'est la troisième condition qui ne l'est pas.

Corollaire 5.10.3
Les propriétés suivantes de langages récursivement énumérables sont récursivement énumérables :

1. $L \neq \varnothing$,
2. L est composé d'au moins 10 éléments,
3. w est dans L, w fixé,
4. $L \cap L_u \neq \varnothing$.

5.11 Problème de correspondance de Post

Le problème de correspondance de Post (PCP) est un problème indécidable important dans la mesure où il permet, toujours au moyen de réductions, de montrer l'indécidabilité de nombreux autres problèmes.

Une instance du PCP est constituée de deux listes de chaînes formées sur un alphabet Σ:

$$A = w_1, \ldots, w_k \qquad B = x_1, \ldots, x_k$$

Cette instance possède une solution s'il existe une séquence d'entiers $i_1, i_2, \ldots, i_m$ avec $m \geq 1$ tel que

$$w_{i_1} w_{i_2} \cdots w_{i_m} = x_{i_1} x_{i_2} \cdots x_{i_m}.$$

La séquence $i_1, \ldots, i_m$ est une solution de cette instance du PCP.

Exemple 5.11.1

1. Soient $\Sigma = \{0, 1\}$ et les listes $A = 1, 10111, 10$ et $B = 111, 10, 0$.

   Avec $m = 4$, la séquence d'entiers $i_1 = 2, i_2 = 1, i_3 = 1, i_4 = 3$ est une solution de cette instance du PCP puisque $w_2 w_1 w_1 w_3 = x_2 x_1 x_1 x_3 = 101111110$.

2. Soit $\Sigma = \{0, 1\}$, $A = 10, 011, 101$ et $B = 101, 11, 011$.

   Pour cette instance du PCP on remarque qu'aucune séquence d'entiers n'est solution du problème.

5.11.1 Décidabilité du problème de correspondance de Post

Théorème 5.11.1
Le problème de correspondance de Post est indécidable.

Preuve
Démonstration fastidieuse et compliquée (c.f. réferences). □

Chapitre 6

Complexité

Dans les chapitres précédents nous avons tout d'abord classifié les langages selon la "puissance" de la machine nécessaire pour les reconnaître. Dans un deuxième temps, nous nous sommes intéressés à caractériser les langages/problèmes par rapport à la notion de calculabilité. Ce troisième volet se préoccupe toujours de classifier les problèmes mais principalement selon le "temps" et l'"espace" nécessaire à une machine de Turing pour accomplir son travail. On appelle ce type de classification: *complexité de calcul*. Ces notions de temps et d'espace sont fondamentales mais bien d'autres critères de classification peuvent être envisagés.

6.1 Notions de base

6.1.1 Complexité en temps et en espace

Définition 6.1.1 *Machine de Turing off-line*
On appelle machine de Turing *off-line* une machine constituée d'une bande d'entrée accessible en lecture uniquement et de k bandes (finies d'un côté, infinies de l'autre) de travail accessibles en lecture/écriture. Le mot d'entrée est encadré par deux symboles particuliers ($\triangleright$ et $\triangleleft$ par exemple) et la tête de la bande d'entrée n'est pas autorisée à se déplacer au-delà de ces deux symboles. Il y a clairement équivalence entre les machines off-line et les autres définitions des machines de Turing. $\diamond$

Définition 6.1.2 *Complexité en espace*
On dit qu'une machine de Turing off-line déterministe M possède une *complexité en espace* $S(n)$ si et seulement si pour chaque mot d'entrée de longueur n, M visite au plus $S(n)$ cases sur une de ses bandes de travail. La complexité en espace n'est donc pas la totalité des cases des bandes de travail visitées mais le maximum de ces bandes (parmi toutes les bandes de travail visitées). On parle aussi de machine bornée en espace par $S(n)$. $\diamond$

Définition 6.1.3 *Complexité en temps*
Soit M une machine de Turing déterministe munie de k bandes infinies des deux côtés accessibles en lecture/écriture et dont une de ces bandes contient l'entrée. La complexité en temps d'une telle machine est définie par le nombre de mouvements avant que la machine atteigne un état final. Si pour un mot de longueur n la machine M réalise au plus $T(n)$ mouvements, on dit que M possède une complexité en temps $T(n)$ ou qu'elle est bornée en temps par $T(n)$. $\diamond$

Le fait de considérer une machine de Turing off-line comme modèle pour la complexité en espace nous permet de considérer une complexité inférieure à une progression linéaire ($S(n) \leq n$) puisque la bande d'entrée n'est pas prise en compte. Dans le cas où $S(n) \geq n$ on peut alors très bien considérer une machine composée d'une seule bande accessible en lecture/écriture sans changer la classe des langages acceptés en $S(n)$. Par contre, pour la complexité en temps il n'est pas possible d'adopter comme modèle une machine à une seule bande au lieu d'une machine multibandes sans modifier la complexité.

Exemple 6.1.1 Soit le langage $L = \{w\mathtt{c}w^r \mid w \in \{\mathtt{0}, \mathtt{1}\}^*\}$, il existe une machine M_1 à deux bandes tel que w est copié dans la première bande et w^r dans la deuxième bande. Nous remarquons que M_1 doit faire au maximum $n+1$ mouvements pour une entrée de longueur n, M_1 est donc de complexité en temps $T(n) = n+1$.

Pour la complexité en espace, il est préférable d'utiliser une machine M_2 pour avoir une complexité en espace de l'ordre de $\log_2(n)$. M_2 est constituée de deux bandes de travail utilisées comme compteurs binaires. M_2 vérifie d'abord qu'il n'existe qu'un seul symbole c dans la chaîne d'entrée, et que le nombre de symboles à gauche et à droite de c est le même, ensuite vérifie chaque symbole gauche avec le symbole droit lui correspondant et accepte si les symboles comparés sont les mêmes.

Définition 6.1.4 *Complexité en temps et non-déterminisme*
La complexité en temps d'une machine de Turing non déterministe correspond au nombre de mouvements maximum parmi les diverses possibilités dues au non-déterminisme. $\Diamond$

Définition 6.1.5 *Complexité en espace et non-déterminisme*
La complexité en espace d'une machine de Turing non déterministe correspond à l'espace nécessaire maximum parmi les diverses possibilités dues au non déterminisme. $\Diamond$

Remarque : On peut facilement admettre qu'une machine de Turing utilise au moins 1 case pour n'importe quelle entrées et effectue au moins $n+1$ mouvements pour effectuer la lecture de son entrée. Afin de définir totalement les complexités en espace et en temps, on considère implicitement $\max(1, \lceil S(n) \rceil)$ pour $S(n)$ et $max(n+1, \lceil T(n) \rceil)$ pour $T(n)$.

6.1.2 Classes de complexité

Pour une complexité $f(n)$ d'un certain type (espace, temps, ...) les langages pouvant être acceptés avec cette complexité forment une classe dite *classe de complexité* (en espace, en temps, ...).

Définition 6.1.6 *Classes de complexité*
La classe de complexité DSPACE($S(n)$) correspond à l'ensemble des langages acceptés par une machine de Turing déterministe ayant une complexité en espace $S(n)$.

La classe de complexité NSPACE($S(n)$) correspond à l'ensemble des langages acceptés par une machine de Turing non déterministe ayant une complexité en espace $S(n)$.

La classe de complexité DTIME($T(n)$) correspond à l'ensemble des langages acceptés par une machine de Turing déterministe ayant une complexité en temps $T(n)$.

La classe de complexité NTIME($T(n)$) correspond à l'ensemble des langages acceptés par une machine de Turing non déterministe ayant une complexité en temps $T(n)$. $\Diamond$

On a clairement les relations d'inclusion suivantes entre classes déterministes et non déterministes:

$$\begin{aligned} \mathrm{DSPACE}(f(n)) &\subseteq \mathrm{NSPACE}(f(n)) \\ \mathrm{DTIME}(f(n)) &\subseteq \mathrm{NTIME}(f(n)) \end{aligned}$$

Bien d'autres classes de complexité seront abordées par la suite, nous pouvons mentionner, parmi celles jouant un rôle très important, la classe $\mathcal{P}$ qui représente la classe des langages acceptés par une machine déterministe bornée en temps par un polynôme et la classe $\mathcal{NP}$ qui décrit la classe des langages acceptés par une machine non déterministe bornée en temps par un polynôme.

Exemple 6.1.2 Le langage $L = \{w\mathtt{c}w^r \mid w \in \{\mathtt{0}, \mathtt{1}\}^*\}$ appartient aux classes de complexité DTIME$(n+1)$, DSPACE$(\log_2 n)$, et NTIME$(n+1)$, mais aussi à NTIME(n^2) puisque DTIME$(n+1) \subseteq$ DTIME$(n^2) \subseteq$ NTIME(n^2).

6.2 Propriétés

6.2.1 Compression de bandes

Le nombre de symboles de l'alphabet d'une machine de Turing pouvant être arbitrairement grand, l'espace nécessaire pour reconnaître un langage peut être réduit par un facteur constant en construisant une machine équivalente employant un alphabet plus riche.

Théorème 6.2.1 *Compression de bandes*
Un langage L appartenant à la classe de complexité DSPACE$(S(n))$ appartient également à DSPACE$(cS(n))$ avec $0 < c < 1$.

Preuve
Soit la machine de Turing M_1 de complexité en espace $S(n)$ acceptant le langage L. Il s'agit de construire une machine M_2 capable de simuler M_1 et qui appartienne à $cS(n)$ avec $0 < c < 1$. On construit M_2 telle que chaque case de M_2 contienne un symbole représentant r cases adjacentes de M_1. La valeur de r est déterminée par la relation $rc \geq 2$. De cette façon M_2 peut simuler M_1 en n'utilisant que $\lceil S(n)/r \rceil \leq cS(n)$ cases car $\lceil S(n)/r \rceil = \lceil cS(n)/(rc) \rceil \leq \lceil cS(n)/2 \rceil \leq cS(n)$. □

Corollaire 6.2.1
Un langage L appartenant à la classe de complexité NSPACE$(S(n))$ appartient également à NSPACE$(cS(n))$ avec $0 < c < 1$.

Preuve
Similaire à la preuve du théorème. □

6.2.2 Réduction du nombre de bandes (espace)

Théorème 6.2.2
Un langage L accepté par une machine déterministe à k bandes de travail de complexité $S(n)$ est également accepté par une machine déterministe à une seule bande de travail de même complexité $S(n)$.

Preuve
Soit M_1 une machine à k bandes de travail et L le langage accepté par M_1. On construit une machine à une seule bande de travail M_2 en regroupant, dans l'unique bande, toutes les bandes de M_1 au moyen d'un alphabet plus riche. La figure qui suit illustre cette construction, chaque bande

de M_1 sera représentée par deux pistes, la première piste contient les données de la bande de M_1 et la seconde un témoin pour marquer la position courante de la tête de M_1.

bande 1	A_1	A_2	A_3		A_n
tête 1			X		
bande 2	B_1	B_2			B_n
tête 2	X				

Un symbole de M_2 représentera donc les pistes de données et les pistes contenant les marqueurs. A chaque pas de simulation, la machine M_2 doit lire sa bande afin de visiter toutes les pistes de données en regard des marqueurs puis mettre à jour toutes ces pistes dans un second parcours de sa bande. Clairement la complexité en espace de M_2 est bien la même que celle de M_1. $\square$

Corollaire 6.2.2
Un langage L accepté par une machine non déterministe à k bandes de travail de complexité $S(n)$ est également accepté par une machine non déterministe à une seule bande de travail de même complexité $S(n)$.

Preuve
Similaire à la preuve du théorème. $\square$

6.2.3 Accélération linéaire

Comme pour la complexité en espace il est possible de réduire d'un facteur constant la complexité en temps. Cependant la lecture de la bande d'entrée qui représente $n+1$ mouvements est à prendre en compte et cette réduction peut avoir lieu dès que le rapport $T(n)/n$ tend vers l'infini lorsque n tend vers l'infini.

Notation :

1. $\sup\limits_{n\to\infty} f(n)$ dénote la limite, lorsque n tend vers l'infini, du plus petit majorant de $\{f(n+k) \mid k \geq 0\}$.
2. $\inf\limits_{n\to\infty} f(n)$ dénote la limite, lorsque n tend vers l'infini, du plus grand minorant de $\{f(n+k) \mid k \geq 0\}$.

Exemple 6.2.1

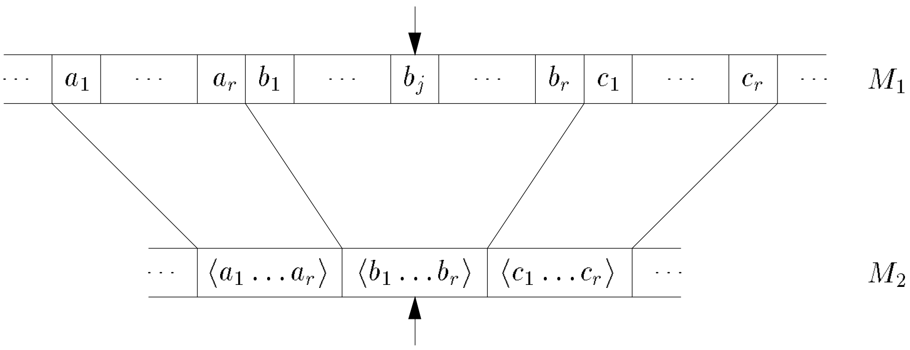
1. Soit la fonction
$$f(n) = \begin{cases} 1/n & \text{si } n \text{ est pair} \\ n & \text{sinon} \end{cases}$$
$\sup\limits_{n\to\infty} f(n) = \infty$ et $\inf\limits_{n\to\infty} f(n) = 0$.
2. Soit la fonction
$$f(n) = \frac{n}{n+1}$$
$\sup\limits_{n\to\infty} f(n) = 1$ et $\inf\limits_{n\to\infty} f(n) = 1$.

Théorème 6.2.3 *Accélération linéaire*
Un langage L accepté par une machine de Turing M à k bandes de complexité en temps $T(n)$ telle que $\inf_{n\to\infty} T(n)/n = \infty$ est également accepté par une machine à $k+1$ bandes de complexité en temps $cT(n)$ avec $k > 1, c > 0$.

Preuve
L'idée de la preuve est semblable à celle du théorème 6.2.1, il s'agit de construire une nouvelle machine qui regroupe r symboles adjacents en un. Cependant certaines précautions supplémentaires au cas de la compression en espace sont nécessaires.

Soit M_1 la machine à k bandes de complexité en temps $T(n)$ acceptant le langage L. On construit une machine M_2 à $k+1$ bandes simulant M_1. Dans un premier temps la machine M_2 copie sa bande d'entrée sur la bande de travail $k+1$ en regroupant r cases adjacentes en une au moyen d'un alphabet plus riche, puis se repositionne en première position sur cette bande. Cette phase nécessite donc $n + \lceil n/r \rceil$ mouvements où n correspond à la longueur de l'entrée. Lorsque cette première phase est accomplie M_2 simule la machine M_1 en utilisant cette bande comme entrée.

La simulation de r pas de M_1 consistera pour M_2 à visiter (pour toutes les bandes) à partir de la position courante la position à gauche, la position courante, la position à droite pour finalement se repositionner en position courante. La figure qui suit illustre la correspondance entre une bande de M_1 et une bande de M_2. La notation $\langle a_1 \dots a_r \rangle$ représente le symbole de l'alphabet enrichi de la machine M_2 correspondant aux symboles $a_1, \dots, a_r$ de M_1. En supposant que la tête de M_2 soit en regard de la flèche, M_2 visitera la position $\langle a_1 \dots a_r \rangle$, $\langle b_1 \dots b_r \rangle$, $\langle c_1 \dots c_r \rangle$ puis se repositionnera en position $\langle b_1 \dots b_r \rangle$.



A ce moment M_2 "connaît" $3r$ cases de M_1, $a_1, \dots, a_r, b_1, \dots, b_r, c_1, \dots, c_r$. Puisque ce contenu consiste en $|\Sigma|^{3r}$ valeurs différentes, le contrôle de M_2 peut représenter ces valeurs au moyen des états. Lorsque la machine M_1 opère sur ces $3r$ cases elle peut:

1. s'arrêter en acceptant ou en refusant,
2. ne jamais déborder de cette zone pour aucune des bandes,
3. effectuer au moins $r+1$ mouvements sur une certaine bande et déborder de la zone.

Le cas 1 et 2 peuvent se décider sans déplacement de tête, uniquement au moyen du contrôle de la machine. Dans le cas 3, M_2 simule r mouvements de M_1 en effectuant au plus deux mouvements. On observe que si la tête de M_1 se trouve en une position b_j, r mouvements vers la gauche ne permettent pas de quitter la zone $a_1, \dots, a_r$ ni de visiter à la fois la zone $a_1, \dots, a_r$ et $c_1, \dots, c_r$. Ces r mouvements de M_1 peuvent donc être effectués en au plus 2 mouvements de M_2.

Ainsi, r mouvements de M_1 sont simulés en au plus 6 mouvements par M_2, quatre pour la visite des deux voisins et deux pour la simulation des r mouvements de M_1. Ceci implique que si la machine M_1 effectue $T(n)$ mouvements pour reconnaître un mot de longueur n, la machine M_2 simulera M_1 en un nombre de mouvements inférieur ou égal à

$$n + \left\lceil \frac{n}{r} \right\rceil + 6 \left\lceil \frac{T(n)}{r} \right\rceil \leq 2n + \frac{n}{r} + 6\frac{T(n)}{r}$$

en sachant que $\lceil x \rceil < x + 1, \forall x \geq 0$ et $n + 7 < 2n, \forall n > 7$. Rappelons que si $\inf_{n\to\infty} T(n)/n = \infty$ alors pour tout $\delta > 0$ il existe une constante n_δ telle que, quelque soit $n \geq n_\delta$, on a $T(n)/n \geq \delta$. Ainsi, pour tout $n \geq n_\delta$

$$2n + \frac{n}{r} + 6\frac{T(n)}{r} \leq T(n) \left(\frac{2}{\delta} + \frac{1}{r\delta} + \frac{6}{r} \right)$$

Clairement, pour tout $c > 0$ donné il est possible de trouver un r et un δ tel que

$$\frac{2}{\delta} + \frac{1}{r\delta} + \frac{6}{r} \leq c.$$

Il reste a considérer l'ensemble des mots de longueur inférieure à $\max(7, n_\delta)$. Ceci ne pose pas un véritable problème car l'ensemble étant fini il est possible de le reconnaître par un automate fini et évidemment par une machine de Turing de complexité $n_\delta + 1$. Rappelons que par $cT(n)$ on entend implicitement $\max(n + 1, cT(n))$. □

Corollaire 6.2.3
Si $\inf_{n\to\infty} T(n)/n = \infty$ et $c > 0$ on a alors

$$\text{DTIME}(T(n)) = \text{DTIME}(cT(n)) \quad \text{et} \quad \text{NTIME}(T(n)) = \text{NTIME}(cT(n))$$

6.2.4 Réduction du nombre de bandes (temps)

Théorème 6.2.4
Un langage accepté par une machine déterministe (respectivement non déterministe) à k bandes bornée en temps par $T(n)$ est également accepté par une machine monobande déterministe (respectivement non déterministe) bornée en temps par $T^2(n)$.

Preuve
La machine M_2 du théorème 6.2.2 possède une complexité en temps $2kT^2(n)$ qui peut être réduite par le théorème 6.2.3 à $T^2(n)$. □

Théorème 6.2.5
Un langage accepté par une machine déterministe (respectivement non déterministe) à k bandes de complexité en temps $T(n)$ est également accepté par une machine déterministe (respectivement non déterministe) à deux bandes de complexité $T(n) \log T(n)$.

Preuve
Preuve fastidieuse et compliquée (c.f. références) □

6.3 Hiérarchie de langages

Dans cette section nous allons répondre à deux questions légitimes concernant les classes de complexité. La première de ces questions, à laquelle il sera répondu par la négation, est la suivante : "Existe-t-il une complexité en temps $T(n)$ telle que tout problème puisse être résolu en $T(n)$?". La seconde peut se formuler ainsi: "Si deux complexités T_1 et T_2 telles que pour tout n $T_1(n) < T_2(n)$, est-ce que DTIME($T_1(n)$) $\subset$ DTIME($T_2(n)$) ?". Les deux questions ont été formulées par rapport à la classe DTIME qu'en est-il pour les classes DSPACE, NTIME et NSPACE? Le théorème 6.3.1 répond à la première question quelque soit le type de classe considéré, en revanche la deuxième question doit être traitée de manière différente pour les complexités en temps et en espace. Le théorème 6.3.2 donne une réponse à la deuxième question pour une complexité en espace alors que théorème 6.3.3 apporte une solution pour une complexité en temps.

Théorème 6.3.1
Pour toute fonction récursive totale donnée $f(n)$ il existe un langage récursif L tel que L n'appartient pas à DTIME($f(n)$).

Preuve
Rappelons que précédemment nous avons montré que toute machine de Turing travaillant sur un alphabet $\{\mathtt{0}, \mathtt{1}, \square\}$ peut être codée en une chaîne binaire et que l'on dispose d'un algorithme pour l'énumération de ces codes. Il est assez aisé de voir que ces deux résultats peuvent être étendus aux machines de Turing possédant un alphabet quelconque.

Définissons le langage suivant

$$L = \{w_i \mid M_i \text{ n'accepte pas } w_i \text{ en moins de } f(|w_i|) \text{ mouvements}\}$$

et montrons qu'il est, d'une part, récursif et, d'autre part, qu'il n'appartient pas à DTIME($f(n)$).

Premièrement, L est récursif. Pour déterminer si un mot w_i appartient ou pas à L, on recherche à quel M_i il correspond en énumérant les codes de machine, puis on simule w_i sur M_i. Si la simulation excède $f(|w_i|)$ mouvements alors $w \in L$ et $w \notin L$ dans le cas contraire. Cette simulation se termine dans tous les cas puisque $f(n)$ est récursive totale.

Deuxièmement, L n'appartient pas à DTIME($f(n)$). En effet par l'absurde, supposons que $L \in$ DTIME($f(n)$) ce qui revient à dire qu'une machine, disons M_i, accepte L avec M_i bornée en temps par $f(n)$. Si cette machine M_i accepte w_i en moins de $f(|w_i|)$ mouvements on a $w_i \in L$ or, par définition de L, si M_i accepte w_i en moins de $f(|w_i|)$ mouvements on a $w_i \notin L$ d'où contradiction. De la même manière, si cette machine M_i n'accepte pas w_i en moins de $f(|w_i|)$ mouvements on a $w_i \notin L$ or, par définition de L, si M_i n'accepte pas w_i en moins de $f(|w_i|)$ mouvements on a $w_i \in L$ d'où contradiction. $\square$

Le théorème 6.3.1 nous indique que la complexité déterministe en temps engendre une hiérarchie de classes de complexité infinie. En effet, considérons la complexité $T(n)$, le théorème 6.3.1 dit qu'il existe un langage récursif L n'appartenant pas à DTIME($T(n)$), en revanche il est accepté par une machine de complexité disons $T_L(n)$. Prenons $T'(n) = \max\{T(n), T_L(n)\}$, on a clairement DTIME($T(n)$) $\subset$ DTIME($T'(n)$) puisque le langage L appartient à DTIME($T'(n)$) mais pas à DTIME($T(n)$).

Le théorème 6.3.1 s'applique également si l'on remplace DTIME par DSPACE, NTIME ou NSPACE.

6.3.1 Hiérarchie en espace

Définition 6.3.1 *Fonction constructible en espace*
Une fonction $S(n)$ est dite *constructible en espace* s'il existe une machine de Turing M de complexité en espace $S(n)$ telle que pour chaque n il y a au moins une entrée de longueur n pour laquelle M utilise exactement $S(n)$ cases. Si toutes les entrées de longueur n de M utilisent exactement $S(n)$ cases alors $S(n)$ est dite *pleinement constructible en espace.* $\Diamond$

L'ensemble des fonctions constructibles en espace contient, entre autres, les fonctions bien connues $\log n$, n^k, 2^n, $n!$, etc. Si $S_1(n)$ et $S_2(n)$ sont constructibles en espace alors $S_1(n)S_2(n)$, $2^{S_1(n)}$ et $S_1(n)^{S_2(n)}$ sont des fonctions constructibles en espace. Cet ensemble est donc particulièrement riche.

Lemme 6.3.1
Si un langage L est accepté par une machine de Turing M bornée en espace par $S(n) \geq \log_2 n$ alors L est accepté par une machine bornée en espace par $S(n)$ et qui s'arrête sur toutes ses entrées.

Preuve
Rappelons que la configuration d'une machine de Turing est composée d'un état, de la position de ses têtes et du contenu de toutes ses bandes. Une séquence de configurations représente donc l'évolution d'une machine pour une certaine entrée. Une configuration apparaissant plusieurs fois dans la séquence représente donc obligatoirement un comportement ne se terminant pas.

Soit une machine off-line M bornée en espace par $S(n)$ avec s états et un alphabet de bandes composé de t symboles. Si M accepte un mot elle le fait obligatoirement en au plus $(n+2)sS(n)t^{S(n)}$ mouvements. Ce produit représente le nombre maximum de configurations possibles. Pour toute évolution ne se terminant pas le nombre de mouvements dépassera cette valeur. L'expression $(n+2)$ correspond à la longueur de l'entrée avec les deux symboles qui l'encadrent et $t^{S(n)}$ représente les différents contenus des bandes. Par le théorème 6.2.2 on peut considérer une machine à une seule bande de travail.

Ainsi, en ajoutant une bande servant de compteur du nombre de mouvements effectués (mis à jour par M elle-même), M peut alors décider d'arrêter sa propre exécution si ce compteur dépasse

$$(4st)^{S(n)} \geq (n+2)sS(n)t^{S(n)} \quad \text{si } S(n) \geq \log_2 n.$$

M effectue l'incrémentation de son compteur en base $4st$ ce qui permet d'arrêter l'exécution si la taille du compteur excède $S(n)$ cases ($\log_{4st} 4st^{S(n)} = S(n)$). Pour ne pas avoir à calculer la taille maximum du compteur ($S(n)$ en réalité), la machine M initialise le compteur à une longueur $\log_2 n$ et chaque fois qu'une case de la bande de travail de plus est nécessaire la taille du compteur est augmentée de un. Dès que le compteur déborde de sa taille M s'arrête en rejetant. La machine M accepte donc L et s'arrête sur toutes ses entrées tout en restant bornée en espace par $S(n)$. $\Box$

Théorème 6.3.2
Si une fonction $S_2(n)$ est pleinement constructible en espace avec

$$\inf_{n\to\infty} \frac{S_1(n)}{S_2(n)} = 0,$$

$S_1(n)$ et $S_2(n) \geq \log_2 n$, alors il existe un langage L appartenant à DSPACE($S_2(n)$) et non à DSPACE($S_1(n)$).

Preuve
Considérons l'énumération binaire des machines de Turing off-line possédant $\{0, 1\}$ comme alphabet d'entrée et une bande de travail. Pour l'occasion nous autorisons un nombre de $\{1\}$ quelconque précédant le code, dans ce cas il y a alors une infinité de codes pour la même machine.

On construit une machine M qui utilise $S_2(n)$ cases et qui ne correspond pas, sur au moins une entrée à une machine de Turing de complexité $S_1(n)$. La machine travaillera sur des mots représentant le code d'une machine suivit de l'entrée de la machine codée.

Tout d'abord, sur une entrée w ($|w| = n$), la machine M commence par marquer $S_2(n)$ cases de sa bande de travail. M pourra ainsi elle-même interrompre son fonctionnement au cas où elle déborderait de ces $S_2(n)$ cases et rejeter w. Ceci nous garantit une complexité en espace $S_2(n)$. Comme la fonction S_2 est pleinement constructible, cette initialisation peut se faire en simulant une machine de Turing utilisant exactement $S_2(n)$ cases pour chaque entrée de longueur n.

Puis, M commence une simulation sur une entrée w d'une machine M_w (la machine suivie de son entrée x encodée par une chaîne binaire w). Si M_w est de complexité $S_1(n)$ et possède t symboles de bande, cette simulation occupera $\lceil \log_2 t \rceil S_1(n)$. Le fonctionnement de M consiste à accepter w si elle peut effectuer sa simulation en moins de $S_2(n)$ cases et que M_w rejette son entrée. Puisque M est bornée en espace par $S_2(n)$ alors $L(M) \in \mathrm{DSPACE}(S_2(n))$ mais $L(M) \notin \mathrm{DSPACE}(S_1(n))$. Voyons que $L(M)$ n'appartient pas à $\mathrm{DSPACE}(S_1(n))$ en supposant le contraire. Supposons qu'une machine M' de complexité $S_1(n)$ avec t symboles de bande accepte $L(M)$, le lemme 6.3.1 nous permet d'affirmer que M' s'arrête sur toutes ses entrées. Puisque M' apparaît un nombre de fois infini dans l'énumération des codes de machine et

$$\inf_{n\to\infty} \frac{S_1(n)}{S_2(n)} = 0,$$

il existe alors un mot w suffisamment long ($|w| = n$) tel que

$$\lceil \log_2 t \rceil S_1(n) < S_2(n).$$

Prenons comme M_w cette machine M'. Sur une entrée w, M possède suffisamment d'espace pour simuler M_w et donc accepte w si M_w rejette. $L(M)$ ne peut donc pas être identique à $L(M_w)$ ce qui est une contradiction. On en déduit que $L(m) \notin \mathrm{DSPACE}(S_1(n))$. □

6.3.2 Hiérarchie en temps

Définition 6.3.2 *Fonction constructible en temps*
Une fonction $T(n)$ est dite *constructible en temps* s'il existe une machine de Turing multibandes M de complexité en temps $T(n)$ telle que pour chaque n il y a au moins une entrée de longueur n pour laquelle M réalise exactement $T(n)$ mouvements. Si pour toutes les entrées de longueur n, M réalise exactement $T(n)$ mouvements alors $T(n)$ est dite *pleinement constructible en temps.* ◊

Théorème 6.3.3
Si T_2 est une fonction pleinement constructible en temps avec

$$\inf_{n\to\infty} \frac{T_1(n) \log T_1(n)}{T_2(n)} = 0,$$

alors il existe un langage L appartenant à $\mathrm{DTIME}(T_2(n))$ et non à $\mathrm{DTIME}(T_1(n))$.

Preuve
Preuve similaire à la preuve du théorème 6.3.2. □

Exemple 6.3.1 Soient les fonctions $T_1(n) = 2n$ et $T_2(n) = n^2 2^n$. Le théorème 6.3.3 nous indique que $\text{DTIME}(2^n) \neq \text{DTIME}(n^2 2^n)$ mais puisque $T_1(n) \leq T_2(n)$ pour tout n on en conclut que $\text{DTIME}(2^n) \subset \text{DTIME}(n^2 2^n)$

6.4 Rapports entre classes de complexité

Théorème 6.4.1

1. Si $L \in \text{DTIME}(f(n))$ alors $L \in \text{DSPACE}(f(n))$,
2. Si $L \in \text{DSPACE}(f(n))$ et $f(n) \geq \log_2 n$ alors il existe une constante c, dépendante de L, telle que $L \in \text{DTIME}(c^{f(n)})$,
3. Si $L \in \text{NTIME}(f(n))$ alors il existe une constante c, dépendante de L, telle que $L \in \text{DTIME}(c^{f(n)})$.

Preuve

1. Si une machine M ne réalise pas plus de $f(n)$ mouvements, elle ne peut analyser plus de $f(n)+1$ cases d'une bande. En groupant deux symboles par case (en modifiant M), le nombre de cases utilisées sera de $\lceil (f(n)+1)/2 \rceil$ qui est inférieur ou égal à $f(n)$.
2. Si une machine M_1 bornée en espace par $f(n)$ et qui possède s états et t symboles de bande alors son nombre total de configurations différentes pour une entrée de longueur n est inférieur ou égal à $(n+2)sf(n)t^{f(n)}$. Puisque $f(n) \geq \log_2 n$, il existe une constante c, telle que pour tout $n \geq 1$ on a $c^{f(n)} \geq (n+2)sf(n)t^{f(n)}$. On construit à partir de M_1 une machine multibandes M_2 qui utilise une bande comme compteur de mouvements et deux autres bandes pour simuler M_1. M_2 comptabilise durant la simulation le nombre de mouvements de M_1, si M_1 n'a pas accepté avant que le compteur atteigne $c^{f(n)}$ alors M_2 s'arrête sans accepter. La machine M_2 est donc bornée en temps par $c^{f(n)}$ et ainsi $L \in \text{DTIME}(c^{f(n)})$.
3. Considérons la machine non déterministe M_1 bornée en temps par $f(n)$ possédant k bandes. Le nombre total de configurations différentes pour une entrée de longueur n est inférieur ou égal à $(f(n)+1)^k st^{kf(n)}$. Pour $d = s(t+1)^{3k}$ on a
$$d^{f(n)} \geq (f(n)+1)^k st^{kf(n)} \quad \text{pour tout } n \geq 1.$$
Une machine multibandes déterministe peut déterminer si M_1 accepte une entrée w de longueur n en construisant la liste de toutes les configurations accessibles depuis la configuration initiale. Ceci peut se faire en un temps inférieur au carré de la longueur de la liste. Puisque la liste des configurations accessibles n'est pas plus longue que $d^{f(n)}$ fois la longueur d'une configuration qui peut être codée par $1 + k(f(n)+1)$ symboles, le temps est alors borné par $c^{f(n)}$ pour une constante c.

□

Théorème 6.4.2 *Savitch*
Si $L \in \text{NSPACE}(S(n))$ alors $L \in \text{DSPACE}(S^2(n))$ pour autant que $S(n)$ soit pleinement constructible en espace et que $S(n) \geq \log_2 n$.

Preuve
c.f. références. □

Chapitre 7

$\mathcal{NP}$-complétude

Dans le chapitre précédent nous avons introduit la notion de classe de complexité en temps et en espace (déterministe ou non) pour les problèmes décidables et exprimé certaines relations entre ces différentes classes. Dans ce chapitre nous allons nous intéresser principalement à la complexité en temps des problèmes et essayer, une fois de plus, de classifier ces derniers par rapport à cette complexité.

Parmi les complexités en temps nous ferons la distinction entre les complexités dites *polynomiales* et celles qui ne le sont pas. Nous considérerons les problèmes possédant un algorithme borné en temps par un polynôme comme étant des problèmes ayant une solution "efficace". Bien que le degré du polynôme puisse être élevé, et donc correspondre à une solution relativement peu efficace, on considère tout de même ce genre de solution comme telle.

Bien d'autres problèmes ne possèdent pas de solutions en temps polynomial mais, par exemple, en temps exponentiel; naturellement ces problèmes sont considérés comme beaucoup plus "difficiles". Toute une série de problèmes très répandus dans la pratique possèdent des solutions exponentielles sans que l'on ait encore réussi à démontrer qu'il n'existe pas de solutions polynomiales à ces problèmes. En revanche, plusieurs de ces problèmes sont solubles par des machines de Turing non déterministes bornées en temps par un polynôme. Cette catégorie de problèmes joue un rôle prépondérant et l'on montrera que si un algorithme polynomial était découvert pour un des problèmes les plus "difficiles" de cette catégorie alors tous les problèmes de la catégorie posséderaient une solution polynomiale, ce qui semble très peu probable. La classe des problèmes résolus par une machine de Turing déterministe bornée en temps par un polynôme est notée $\mathcal{P}$ et celle correspondant à des machines non déterministes bornées également par un polynôme est notée $\mathcal{NP}$. La question "la classe $\mathcal{P}$ est-elle identique à la classe $\mathcal{NP}$ ou non" reste, à l'heure qu'il est, une des questions ouvertes les plus importantes du domaine de la complexité.

7.1 Classes polynomiales

La classe *déterministe polynomiale* regroupe tous les problèmes solubles par une machine de Turing déterministe bornée en temps par un polynôme. Cette classe contient, par exemple, les problèmes de recherche dichotomique, de tri, de multiplication matricielle etc.

Définition 7.1.1
La classe déterministe polynomiale, notée $\mathcal{P}$, est définie par:

$$\mathcal{P} = \bigcup_{i \in \mathbf{N}} \mathrm{DTIME}(n^i) = \{L \mid L(M), M \text{ déterministe}, T_M(n) \leq P(n) \text{ où } P(n) \text{ est un polynôme}\}$$

$\Diamond$

La classe *non déterministe polynomiale* $\mathcal{NP}$ regroupe les problèmes solubles par une machine de Turing non déterministe bornée en temps par un polynôme. Les problèmes appartenant à cette classe mais non à $\mathcal{P}$ ont une grande importance au niveau pratique. Parmi ces problèmes on trouve par exemple le problème du circuit d'Hamilton (HC) qui peut s'énoncer ainsi: "Un graphe donné possède-t-il un chemin qui visite chaque sommet une et une seule fois ?". Ce problème possède un algorithme déterministe exponentiel évident consistant à énumérer tous les chemins possibles tout en vérifiant si figure une solution parmi cette énumération. Par contre, il existe également un algorithme non déterministe borné par un polynôme qui consiste à choisir de manière non déterministe un chemin et de vérifier si ce chemin est une solution au problème. Nous reviendrons sur ce problème d'Hamilton ultérieurement tout comme sur bien d'autres problèmes appartenant à cette fameuse classe $\mathcal{NP}$.

Définition 7.1.2
La classe non déterministe polynomiale, notée $\mathcal{NP}$, est définie par:

$$\begin{aligned}\mathcal{NP} &= \bigcup_{i\in\mathbf{N}} \mathrm{NTIME}(n^i) \\ &= \{L \mid L(M), M \text{ non déterministe}, T_M(n) \leq P(n) \text{ où } P(n) \text{ est un polynôme}\}\end{aligned} \qquad \Diamond$$

On définit les deux classes de complexité relatives à l'espace de manière identique:

Définition 7.1.3

$$\mathrm{PSPACE} = \bigcup_{i\in\mathbf{N}} \mathrm{DSPACE}(n^i) \quad \text{et} \quad \mathrm{NSPACE} = \bigcup_{i\in\mathbf{N}} \mathrm{NSPACE}(n^i) \qquad \Diamond$$

Il y a plusieurs raisons pour considérer plus en détail les classes $\mathcal{P}$ et $\mathcal{NP}$. Une de ces raisons est le fait que si un langage n'appartient pas à $\mathcal{P}$ alors, d'un point de vue pratique, ce langage ne peut raisonnablement pas être calculé par un ordinateur à l'exception d'un nombre très petit d'entrées. Il est donc important de savoir quels langages font partie de la classe $\mathcal{P}$.

Bien que l'expression *calculable en temps* polynomial puisse changer de degré selon le modèle formel considéré, l'appartenance à $\mathcal{P}$ est invariable pour la majorité des modèles connus d'ordinateurs. On a cité plusieurs modèles, machine de Turing: on-line, off-line, avec une seule bande, avec plusieurs bandes, multidimensionnelle, machine RAM, etc; la version déterministe de tous ces modèles accepte en temps polynomial précisément la même classe de complexité $\mathcal{P}$ et la version non déterministe la classe $\mathcal{NP}$. Dans ces conditions l'hypothèse de Church peut être reformulée ainsi:

Hypothèse de Cook
Si un langage appartient à $\mathcal{P}$ ou $\mathcal{NP}$, il sera alors accepté en temps polynomial par les modèles "décents" respectivement déterministes ou non déterministes.

Par "décent" on entend les modèles comme les machines de Turing et les RAM. Le manque d'une définition précise de cet adjectif fait que l'hypothèse de Cook, comme celle de Church, est bien une hypothèse et non un théorème.

Il existe une très large classe de problèmes pour lesquels il a été facilement prouvé leur appartenance à la classe $\mathcal{NP}$ alors que personne n'a encore pu déterminer leur appartenance ou leur non-appartenance à $\mathcal{P}$, voire même quelle est la perte minimale en efficacité pour simuler de manière déterministe la version non déterministe. Plusieurs problèmes combinatoires, en théorie des graphes, recherche opérationnelle, systèmes opérationnels, etc, appartiennent à $\mathcal{NP}$ sans que l'on ait encore trouvé une version déterministe efficace.

La question $\mathcal{P} = \mathcal{NP}$? est question ouverte et considérée par plusieurs chercheurs comme la question ouverte la plus importante de notre époque. Formulée différemment elle correspond à "tous les problèmes résolus de manière non déterministe en temps polynomial peuvent-il être résolus de manière déterministe en temps polynomial?"

Une façon d'aborder les problèmes qui appartiennent à $\mathcal{NP}$ et qui, peut-être, ne peuvent pas appartenir à $\mathcal{P}$ est de considérer les réductions polynomiales ainsi que les problèmes plus difficiles dans $\mathcal{NP}$, les problèmes $\mathcal{NP}$-complets.

7.2 Réductions polynomiales

Les réductions polynomiales sont des cas particuliers des réductions introduites lors de notre étude concernant la calculabilité. Ces réductions vont nous permettre de tirer certaines conclusions quant à la structure de la classe $\mathcal{NP}$.

Définition 7.2.1
Soient deux langages $L_1 \in \Sigma_1^*$ et $L_2 \in \Sigma_2^*$. Une *réduction polynomiale* de L_1 vers L_2, notée $L_1 \propto L_2$, est une fonction $\Phi : \Sigma_1^* \rightarrow \Sigma_2^*$ réalisée par une machine de Turing bornée en temps par un polynôme telle que $\Phi(x) \in L_2$ si et seulement si $x \in L_1$. On dit aussi que L_1 est *polynomialement réductible* à L_2. $\Diamond$

Lemme 7.2.1
Soient deux langages L_1 et L_2 avec L_1 polynomialement réductible à L_2:

1. si L_2 appartient à $\mathcal{P}$ alors L_1 appartient à $\mathcal{P}$,
2. si L_2 appartient à $\mathcal{NP}$ alors L_1 appartient à $\mathcal{NP}$.

Preuve

1. L_2 appartenant à $\mathcal{P}$ il existe donc un algorithme borné en temps par un polynôme $p(n)$ décidant ce langage. Puisque $L_1 \propto L_2$ un polynôme $q(n)$ borne la fonction de réduction et l'application de l'algorithme décidant L_2 à la réduction produit donc un algorithme décidant L_1. Cet algorithme est bien borné en temps polynomial car pour un mot $w \in L_1$ de longueur n la réduction est bornée en temps par $q(n)$ et produira un mot de longueur inférieure à $p(n)$ (Le résultat est de longueur inférieure au temps de calcul car l'écriture de chaque symbole nécessite au moins une étape de calcul). Le temps total pour décider si un mot w de longueur n appartient à L_1 ou non est inférieur à $q(n) + p(q(n))$ qui correspond bien à un polynôme. L_1 appartient bien à $\mathcal{P}$.
2. similaire au point 1). $\square$

Notons que si l'on trouve un langage $L \in \mathcal{P}$ pour lequel tous les langages de $\mathcal{NP}$ lui sont polynomialement réductibles, on trouverait automatiquement des algorithmes polynomiaux déterministes pour tous les langages appartenant à $\mathcal{NP}$, en conséquence de la première partie du lemme précédent. Alors, on aurait prouvé que $\mathcal{P} = \mathcal{NP}$ et ce langage L doit être au moins aussi difficile que tous les autres langages appartenant à $\mathcal{NP}$. Nous investiguerons l'existence des tels langages ultérieurement.

7.3 La notation “ordre de”

Avant de poursuivre, il est utile d'introduire un peu plus formellement la notation “ordre de”.

Définition 7.3.1
Dire qu'une fonction $g(n)$ est $O(f(n))$ (notation $g(n) \sim O(f(n))$) s'il existe deux constantes c et n_0 telles que: $g(n) \leq cf(n) \quad \forall n \geq n_0$ $\Diamond$

Si on prends $g(n)$ comme mesure de complexité en temps (espace) alors la notation O correspond au théorème de l'accélération linéaire (compression d'alphabet) qui fait disparaître la constante c et par conséquent n_0.

Cette définition correspond à la notion de borne supérieure; la fonction $g(n)$ majore $f(n)$. D'une manière similaire on peut définir borne inférieure:

Définition 7.3.2
Une fonction $g(n)$ est $\Omega(f(n))$ (notation $g(n) \sim \Omega(f(n))$) s'il existe deux constantes c et n_o telles que: $g(n) \geq cf(n) \quad \forall n \geq n_0$ $\Diamond$

Une fonction peut aussi avoir une borne optimale:

Définition 7.3.3
Une fonction $g(n)$ est $\Theta(f(n))$ (notation $g(n) \sim \Theta(f(n))$) si et seulement si $g(n) \sim O(f(n))$ et $g(n) \sim \Omega(f(n))$ $\Diamond$

Exemples 7.3.1

1. $7n^2 - 99999n + 777777777 \sim O(n^2)$
2. $n^6 \sim O(2^n)$
3. $n^2 \sim O(n^3)$
4. $n^2 \sim O(n^4)$
5. $n^2 \sim \Omega(n)$
6. $n^2 \sim \Omega(\log n)$
7. $k \sim O(1)$
8. $5n^2 + 78n + 999 \sim \Theta(n^2)$

7.4 $\mathcal{NP}$-complétude

Définition 7.4.1 *Langage $\mathcal{NP}$-hard*
On dit que le langage L est $\mathcal{NP}$-hard si pour tout L' appartenant à $\mathcal{NP}$, L' est polynomialement réductible à L ($L' \propto L$). $\Diamond$

Remarquez que l'on ne demande pas l'appartenance à $\mathcal{NP}$ d'un problème $\mathcal{NP}$-hard. En conséquence, un problème $\mathcal{NP}$-hard peut être plus difficile que la classe de complexité $\mathcal{NP}$.

Théorème 7.4.1
Si un langage L_1 est $\mathcal{NP}$-hard et polynomialement réductible à un langage L_2, alors L_2 est aussi $\mathcal{NP}$-hard.

Preuve
Soit L un langage quelconque dans $\mathcal{NP}$. Par hypothèse L_1 est $\mathcal{NP}$-hard, donc $L \propto L_1$. Aussi par hypothèse L_1 est polynomialement réductible à L_2, alors $L_1 \propto L_2$. On a $L \propto L_1 \propto L_2$, donc $L \propto L_2$. Alors, L_2 est $\mathcal{NP}$-hard. □

Si l'on exige qu'un langage $\mathcal{NP}$-hard appartienne à $\mathcal{NP}$, on obtient le concept de $\mathcal{NP}$-complétude.

Définition 7.4.2 *$\mathcal{NP}$-complétude*
On dit que le langage L est $\mathcal{NP}$-complet si

1. L appartient à $\mathcal{NP}$
2. L est $\mathcal{NP}$-hard. ◊

La principale caractéristique d'un problème $\mathcal{NP}$-complet est que si l'on arrive à trouver un algorithme déterministe polynomial pour le résoudre, alors $\mathcal{P} = \mathcal{NP}$.

Théorème 7.4.2
Si L est $\mathcal{NP}$-complet alors $\mathcal{P} = \mathcal{NP}$ si et seulement si L appartient à $\mathcal{P}$.

Preuve
Dans un sens, si L est $\mathcal{NP}$-complet appartient à $\mathcal{NP}$ par définition de la $\mathcal{NP}$-complétude et donc appartient à $\mathcal{P}$ par l'hypothèse de $\mathcal{P} = \mathcal{NP}$. Pour prouver la réciproque: supposez que L $\mathcal{NP}$-complet appartient à $\mathcal{P}$. Alors, L est aussi $\mathcal{NP}$-hard par définition de la $\mathcal{NP}$-complétude. Ainsi tous les langages $L \in \mathcal{NP}$ lui sont polynomialement réductibles et comme L doit appartenir à $\mathcal{P}$ par le lemme 7.2.1 (point 1) on a bien $\mathcal{P} = \mathcal{NP}$. □

7.5 Premier problème $\mathcal{NP}$-complet

Pour pouvoir utiliser la théorie de la $\mathcal{NP}$-complétude, on doit trouver un premier problème $\mathcal{NP}$-complet pour employer nos réductions polynomiales afin de déterminer d'autres problèmes $\mathcal{NP}$-complets. Ce premier problème est la satisfaisabilité pour le calcul des expressions booléennes.

Définition 7.5.1
On dit qu'une expression booléenne F est *satisfaisable* s'il existe au moins une combinaison de valeurs de ses variables (0 ou 1) qui la rende vraie. ◊

Exemples 7.5.1

1. La formule $x \vee y \wedge z$ est satisfaisable,
2. La formule $x \wedge y \wedge \bar{x}$ n'est pas satisfaisable.

Le problème de la satisfaisabilité est la détermination si une expression booléenne donnée est satisfaisable ou non. Il peut être représenté par le langage L_{SAT}:

$$L_{\mathrm{SAT}} = \{F \mid F \text{ une expression booléenne satisfaisable}\}$$

Théorème 7.5.1 *Cook*
Le langage L_{SAT} est $\mathcal{NP}$-complet.

Preuve
Cette preuve se déroule en deux parties, la première se charge de déterminer un algorithme non déterministe polynomial qui montre que L_{SAT} appartient à $\mathcal{NP}$. La deuxième partie est nettement plus ardue car il s'agit de montrer que tout problème de $\mathcal{NP}$ se réduit polynomialement à L_{SAT} c'est-à-dire que ce dernier est $\mathcal{NP}$-hard. De ces deux points on en déduit la $\mathcal{NP}$-complétude de L_{SAT}.

Le langage L_{SAT} appartient à la classe $\mathcal{NP}$ car l'algorithme non déterministe suivant, dont l'entrée est la formule F, est clairement linéaire par rapport à la longueur de son entrée:

1. vérifier que F est bien une expression booléenne,
2. pour chaque variable x de F, fixer de manière non déterministe x à 0 ou 1,
3. calculer F et accepter si le résultat vaut 1 et rejeter sinon.

Montrer que L_{SAT} appartient à $\mathcal{NP}$-hard est plus difficile. Pour ceci, on doit montrer que *tout* langage L dans $\mathcal{NP}$ est polynomialement réductible à L_{SAT}. On doit transformer tout langage L en une formule F telle que si un mot w appartient à L, alors F est satisfaisable. Notons que F doit dépendre aussi du mot w.

Soit un L quelconque qui appartienne à $\mathcal{NP}$. Alors, il y existe une machine de Turing non déterministe qui accepte L en temps polynomial $p(n)$. Soit w un mot accepté par M. On peut décrire les pas de M sur w selon la formule F composée des prédicats configuration, prochain initiale et finale.

$$\text{accepté}(w) = \wedge_{i=1}^{p(n)} \text{configuration}(\alpha_i) \wedge_{i=1}^{p(n)-1} \text{prochain}(\alpha_i, \alpha_{i+1}) \ \wedge\ \text{initiale}(\alpha_1, w)\ \wedge\ \text{finale}(\alpha_{p(n)})$$

Cette formule simule un chemin de calcul de M; elle signifie que w est accepté si à partir de la configuration initiale α_1 pour w, chaque configuration α_{i+1} est la configuration logique suivante à partir de α_i selon la fonction de transition de M et que la configuration finale $\alpha_{p(n)}$ est acceptante. Notons que cette formule décrit un seul chemin de M; mais M est non déterministe ce qui veut dire qu'il peut y avoir plusieurs chemins calcul pour M. On règle ceci en disant que s'il existe une combinaison de 0 et 1 qui rendre F vraie alors il existe un chemin acceptant de calcul de M qui accepte w et vice versa.

Il reste à prouver qu'une telle formule peut être formulée en temps polynomial, et que donc la réduction est bien polynomiale. Pour le montrer on va détailler chaque prédicat de F:

1. configuration(α) doit être vrai si et seulement si α reçoit une chaîne de 0 et de 1 qui encode une configuration de M sur w.

   Alors configuration(α) est composé d'une sous-formule qui encode les symboles de la bande de M, d'une sous-formule qui encode la position de la tête de la bande, et d'une sous formule qui encode l'état interne de M. La longueur de toutes ces formules est polynomiale, puisque M est bornée par $p(n)$. Finalement, un nombre polynomial de disjonctions peut tester si l'état et la tête de M apparaît au moins une fois et un nombre polynomial de conjonctions peuvent tester que la tête apparaît une seule fois. La formule finale configuration(α) est une conjonctions de toutes ces sous-formules et dont la longueur est toujours polynomiale.

2. prochain(α, β) doit être vrai si et seulement si α, β reçoivent une chaîne de 0 et de 1 qui encode des configurations de M de longueur $p(n)$ et que M peut aller de α à β en une étape.

   Alors, cette formule commence avec la conjonction configuration$(\alpha \ \wedge$ configuration(β). Une troisième formule est ajoutée pour tester si le changement des symboles entre α et β suivent la relation de transition de M. Une quatrième formule est nécessaire pour s'assurer que les autres symboles ne changent pas entre α et β. De nouveau la longueur de la formule prochain(α, β) est bornée polynomialement, puisque les sous-formules configuration le sont, par l'argument précédent et les autres deux sous-formules représentent les contenus de bande qui doivent être bornés par $p(n)$ puisque M est bornée par $p(n)$ par hypothèse.

3. initiale(α, w) doit être vraie si et seulement si α reçoit une chaîne de 0 et de 1 qui encode la configuration initiale de M sur w.

   Cette formule est composée d'une formule configuration(α) et d'une autre formule qui teste si la tête se trouve au début de la bande, si l'état interne correspond à l'état initial de M et que le contenu de la bande d'entrée est w suivi par des blancs. Toutes ces sous-formules sont bornées polynomialement.

4. finale(α) doit être vraie si et seulement si α reçoit une chaîne de 0 et de 1 qui encode une configuration finale acceptante de M.

   Cette formule est composée de $config(\alpha)$ et d'une sous-formule qui teste si l'état de α est un des états finaux de M; ce qui peut être fait avec une borne polynomiale.

Alors, la formule F qui est écrite comme accepté(w) peut être écrite en temps polynomial, puisque chaque partie est de longueur polynomiale. De plus, accepté(w) est satisfaisable si et seulement s'il existe une séquence de 0 et de 1 pour les variables α_i telle que la machine M commence avec une configuration initiale, suit les configurations prochaines et arrive à une configuration finale. Donc, $w \in L$ si et seulement si $accepte(w) \in L_{\text{SAT}}$. Alors, la fonction qui à chaque w assigne la formule $accepte(w)$ est une réduction polynomiale de L à L_{SAT}. Donc, L_{SAT} est $\mathcal{NP}$-hard. Finalement, on a que:

- L_{SAT} appartient à $\mathcal{NP}$ et
- L_{SAT} est $\mathcal{NP}$-hard.

et donc, L_{SAT} est $\mathcal{NP}$-complet. □

7.6 Quelques exemples de réduction

Une fois que l'on connaît un problème $\mathcal{NP}$-complet, disons L, on peut déduire, au moyen de la réduction polynomiale, qu'un autre problème, disons L', est également $\mathcal{NP}$-complet en prouvant les deux points suivants:

1. L' appartient à $\mathcal{NP}$,
2. $L \propto L'$ et donc L' est $\mathcal{NP}$-hard.

Cette technique de réduction polynomiale est fréquemment employée pour prouver qu'un nouveau problème est $\mathcal{NP}$-complet. Dans cette section nous allons explorer quelques problèmes classiques et montrer au moyen de cette technique qu'ils sont $\mathcal{NP}$-complets.

7.6.1 Problèmes de satisfaisabilité

Commençons par le problème de la satisfaisabilité des expressions booléennes sous forme normale conjonctive (CNF), c'est-à-dire, F est une formule de la forme:

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

avec chaque C_i composé uniquement de disjonctions de variables. On appelle ces C_i des clauses. En termes de langage ce problème correspond au langage

$$L_{\mathrm{SAT-CNF}} = \{F \mid F \text{ est une expression CNF satisfaisable}\}$$

Théorème 7.6.1
Le langage $L_{\text{SAT-CNF}}$ est $\mathcal{NP}$-complet.

Preuve
On doit prouver deux choses:

1. $L_{\text{SAT-CNF}} \in \mathcal{NP}$
2. $L_{\text{SAT}} \propto L_{\text{SAT-CNF}}$

Le même algorithme que celui pour L_{SAT} fonctionne puisqu'il s'agit d'un cas particulier de L_{SAT}.

Une formule booléenne est facilement transformable en une formule booléenne sous forme normale conjonctive en temps polynomial et la longueur de la formule résultante a un rapport polynomial avec la longueur de la formule originale.

Soit F la formule booléenne initiale, on construit une formule F' sous forme normale conjonctive telle que F' soit satisfaisable si et seulement si F est satisfaisable. On sait qu'il existe une fonction polynomiale g tel que pour tout F, $g(F) = F'$ est une formule booléenne sous forme normale conjonctive, et F est dans L_{SAT} si et seulement si $F' = g(F)$ est dans $L_{\text{SAT-CNF}}$. $L_{\mathrm{SAT-CNF}}$ est alors $\mathcal{NP}$-hard.

Puisque $L_{\text{SAT-CNF}}$ appartient à $\mathcal{NP}$ et que $L_{\text{SAT}} \propto L_{\text{SAT-CNF}}$ on en conclut que $L_{\text{SAT-CNF}}$ est $\mathcal{NP}$-complet. □

Une autre variante de ce genre de problèmes est la satisfaisabilité des formules k-CNF. On dit qu'une expression booléenne est k-CNF si cette expression est la conjonction de la disjonction d'au plus k variables.

Exemple 7.6.1 La formule $(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\bar{x}_1 \vee x_2)$ est 2-CNF.

Le problème k-SAT correspond à la détermination de la satisfaisabilité d'une expression booléenne sous forme k-CNF. Pour $k \leq 2$, on peut trouver des algorithmes déterministes polynomiaux pour tester la satisfaisabilité, mais à partir de 3-SAT les choses se compliquent.

Théorème 7.6.2
3-SAT est un problème $\mathcal{NP}$-complet.

Preuve
Considérez le langage;

$$L_{\text{3-SAT}} = \{F \mid F \text{ une expression 3-CNF satisfaisable}\}$$

qui formalise le problème 3-SAT. Pour montrer que 3-SAT est $\mathcal{NP}$-complet, on doit prouver deux choses:

1. $L_{\text{3-SAT}} \in \mathcal{NP}$ et
2. $L_{\text{3-SAT}}$ est $\mathcal{NP}$-hard.

Le langage $L_{\text{3-SAT}} \subseteq L_{\text{SAT}}$ et donc le même algorithme que celui pour L_{SAT} fonctionne pour $L_{\text{3-SAT}}$. $L_{\text{3-SAT}}$ appartient bien à $\mathcal{NP}$.

On montre par réduction polynomiale de $L_{\text{SAT-CNF}}$ à $L_{\text{3-SAT}}$ que 3-SAT est $\mathcal{NP}$-hard. Soit F une formule en forme normale conjonctive. Soit $(x_1 \vee x_2 \vee \ldots \vee x_k)$ une clause quelconque de F. On remplace cette clause par:

$$(x_1 \vee x_2 \vee y_1) \wedge (x_3 \vee \bar{y}_1 \vee y_2) \wedge (x_4 \vee \bar{y}_2 \vee y_3) \wedge \ldots \wedge (x_{k-2} \vee \bar{y}_{k-4} \vee y_{k-3}) \wedge (x_{k-1} \vee x_k \vee \bar{y}_{k-3})$$

où les y_i sont de nouvelles variables. Cette nouvelle clause est satisfaisable si et seulement si la clause originale est satisfaisable. En effet, si la clause originale est vraie alors il y a au moins un $x_i = 1$. On assigne alors à chaque $y_k \leq i-2$ la valeur 1 et à chaque $y_k \geq i-1$ la valeur 0 tout en maintenant les x_i. Il est aisé de voir que la nouvelle clause est également vraie.

Réciproquement, pour que la nouvelle clause soit être rendue vraie il faut avoir au moins un x_i égal à 1, sinon tous les y_k doivent être 1, ce qui rend $x_{k-1} \vee x_k \vee \bar{y}_{k-3}$ faux. Alors, on doit avoir au moins une variable $x_1 = 1$ qui rend la clause originale vraie.

Cette construction peut remplacer chaque disjonction de la formule originale par un ensemble de disjonctions 3-CNF en temps polynomial. Alors, $L_{\text{SAT-CNF}} \propto L_{\text{3-SAT}}$ et donc $L_{\text{3-SAT}}$ est $\mathcal{NP}$-hard.

Finalement, $L_{\text{3-SAT}}$ est $\mathcal{NP}$-complet. □

7.6.2 Problème de la Clique

Toutes les réductions ne sont aussi faciles que les précédentes. Certaines réductions nécessitent imagination et habileté. Par exemple, la réduction qui suit est intéressante parce qu'elle réalise la réduction d'une formule booléenne à un problème de graphe. Il existe une grande variété de problèmes de graphe qui sont $\mathcal{NP}$-complets. Ici, on considère le problème de la *clique*. Une *clique* est un sous-graphe complet d'un graphe. Le problème est de déterminer si un graphe donné contient une clique de taille donnée.

Théorème 7.6.3
Étant donné un graphe non orienté, $G = (V, E)$ et un entier k, existe-t-il $V' \subseteq V$, $|V'| \geq k$ et pour tout v et v' de V' on a $(v, v') \in E$? Ce problème est $\mathcal{NP}$-complet.

Preuve
Comme toujours il faut prouver que:

1. Le problème de la clique appartient à $\mathcal{NP}$.

   Soit l'algorithme non déterministe suivant:

   (a) entrée $G = (V, E)$ et k,

   (b) choisir d'une manière non déterministe $V' \subseteq V$,

   (c) tester si $|V'| \geq k$,

   (d) tester si tous les sommets de V' sont connectés,

(e) accepter si c'est le cas sinon rejeter.

Cet algorithme est au pire des cas, de complexité $O(n^2)$, où n est le nombre de sommets de G. Le problème de la clique appartient donc clairement à $\mathcal{NP}$.

2. Le problème de la clique est $\mathcal{NP}$-hard.

Effectuons la réduction suivante: $L_{\text{SAT-CNF}} \propto L_{\text{Clique}}$.

Une formule CNF consiste en un ensemble de littéraux $\{x_1, x_2 \ldots x_r\}$ où chaque x_i peut être une variable ou son complément ainsi qu'un ensemble de clauses $\{C_1, \ldots, C_s\}$. La fonction de réduction d'une formule CNF à *Clique* revient à construire un graphe $G = (V, E)$ qui possède une clique d'au moins s sommets si et seulement si la formule est satisfaisable:

$$\begin{aligned} V &= \{[x_i, C_j] \mid x_i \text{ est dans } C_j\} \\ E &= \{([x_i, C_j], [x_m, C_l]) \mid j \neq l \text{ et } \bar{x}_i \neq x_m\} \\ k &= s \end{aligned}$$

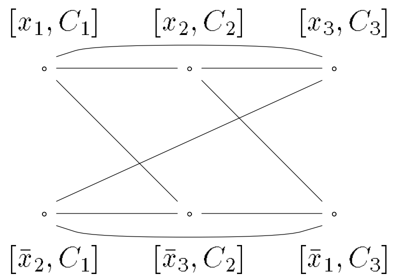
La fonction de réduction peut clairement se faire en temps polynomial. Reste à montrer qu'une telle réduction est une véritable réduction. Si la formule CNF est satisfaisable alors chaque clause C_n possède au moins un littéral x_{j_n} qui rende C_n vraie. Donc les sommets:

$$[x_{j_1}, C_1], [x_{j_2}, C_2], \ldots, [x_{j_k}, C_k]$$

forment une clique de G, car chacun de ces sommets sont voisins par définition de la fonction de réduction. Réciproquement, supposons qu'il existe une clique dans G. Alors, aucun des littéraux peut être la négation d'un autre, par construction de l'ensemble E. Donc, si l'on rend chacun de ces littéraux vrai, chacune des clauses est vraie et la formule originale aussi.

Conclusion, le problème de la clique appartient à $\mathcal{NP}$, il est $\mathcal{NP}$-hard et donc $\mathcal{NP}$-complet. □

Exemple 7.6.2 Soit la formule satisfaisable $\underbrace{(x_1 \vee \bar{x}_2)}_{C_1} \wedge \underbrace{(x_2 \vee \bar{x}_3)}_{C_2} \wedge \underbrace{(x_3 \vee \bar{x}_1)}_{C_3}$. Le graphe construit par la réduction possède bien une clique de 3 sommets:



7.6.3 Couverture de sommets (vertex cover VC)

Soit $G = (V, E)$ un graphe non orienté, V l'ensemble de ses sommets et E l'ensemble de ses arêtes. On dit qu'un sous-ensemble $S \subseteq V$ est une couverture de sommets du graphe G, si pour toute arête $(v, v') \in E$ on a $v \in S$ ou $v' \in S$. Le problème de la couverture de sommets consiste en: Étant donné un graphe G et un entier $k \leq |V|$, G possède-t-il une couverture inférieure ou égale à k sommets?

Théorème 7.6.4
Le problème de la couverture de sommets est $\mathcal{NP}$-complet.

Preuve
Pour prouver que ce problème appartient à $\mathcal{NP}$, il suffit qu'une machine de Turing non déterministe choisisse un ensemble de sommets et vérifie s'ils sont une couverture de taille inférieure à k.

Pour montrer que le problème est $\mathcal{NP}$-hard, on réalise une réduction polynomiale du problème de la clique à VC (Vertex Cover).

Soient $G = (V, E)$ et k les entrées pour le problème de la clique. La réduction consiste à construire le graphe complément de G, noté $\overline{G}$ et de définir l'entier $\overline{k} = |V| - k$:

$$\overline{G} = \{V, \overline{E}\} \text{ et } \overline{E} = \{(i, j) \mid i, j \in V \text{ et } (i, j) \notin E\}.$$

Il reste à montrer que cette transformation est correcte. Si les sommets $v_1, \dots v_k$ déterminent une clique pour G, alors comme il n'y a pas d'arête dans $\overline{G}$ qui relie ces v_i, toute arête dans $\overline{G}$ doit avoir un élément de $V \setminus \{v_1, \dots v_k\}$. Donc, cet ensemble est une couverture pour $\overline{G}$ avec $\overline{k}$ sommets.

Réciproquement, supposons que $U = \{u_1, \dots u_{\overline{k}}\}$ est une couverture des sommets pour $\overline{G}$. Alors, toute arête de $\overline{G}$ possède un élément de U et donc deux sommets de $V \setminus U$ ne peuvent pas être liés par une arête de $\overline{G}$. Mais, tous les deux sommets sont liés par une arête de G ou par une arête de son complément, et donc tous les deux sommets de $V \setminus U$ sont liés par une arête de G. Alors, $V \setminus U$ est une clique pour G avec k sommets.

Cette transformation de G à $\overline{G}$ peut être clairement réalisée en temps polynomial, notre réduction de la clique au VC est donc bien une réduction polynomiale. Le problème de la couverture de sommets est $\mathcal{NP}$-hard et donc $\mathcal{NP}$-complet. □

7.6.4 Problème de l'ensemble indépendant

Le problème de l'ensemble indépendant consiste à savoir, étant donné un graphe non orienté $G = (V, E)$ et un entier positif k, si G contient un ensemble $V' \subseteq V$, $|V'| \geq k$ tel qu'aucune paire de sommets de V' ne corresponde à une arête de E. Cet ensemble s'appelle l'ensemble indépendant.

Théorème 7.6.5
Le problème de l'ensemble indépendant est $\mathcal{NP}$-complet.

Preuve
Pour prouver l'appartenance à $\mathcal{NP}$, considérez la machine de Turing qui choisisse d'une manière non-déterministe un sous ensemble de V et teste si le nombre d'éléments de ce sous ensemble est plus petit ou égal à k et teste si aucune paire de sommets ne soient liés par une arête. Cette machine de Turing est bornée par $O(n^2)$, où $|V| = n$, et alors le problème est $\mathcal{NP}$.

Pour prouver que ce problème est $\mathcal{NP}$-hard, on fait la réduction du problème de la clique à l'ensemble indépendant. La fonction de réduction est exactement la même qu'entre les problèmes de la clique et de la couverture de sommets, à savoir: on construit le complément du graphe donné comme entrée pour savoir s'il y a une clique. C'est facile à voir qu'une clique en G se convertit en un ensemble indépendant en $\overline{G}$ et vice-versa. Évidemment le complément de G peut être trouvé en temps polynomial. Le problème de l'ensemble indépendant est alors $\mathcal{NP}$-hard, étant donné que le problème de la clique l'est aussi. Le problème de l'ensemble indépendant est $\mathcal{NP}$-complet. □

7.6.5 Problème du chemin de Hamilton pour graphe orienté

Étant donnés un graphe orienté $G = (V, E)$ et deux sommets particuliers $s, t \in V$, ce problème consiste à déterminer s'il existe un parcours du graphe qui commence en s et termine en t contenant chaque sommet une et une seule fois. Ce problème est très important, car il est très utilisé en pratique. Malheureusement il est $\mathcal{NP}$-complet. La preuve de complétude est assez technique et compliquée. On donne une réduction à partir du problème de la couverture de sommets pour le lecteur intéressé.

Théorème 7.6.6
Le problème du chemin de Hamilton pour un graphe orienté est $\mathcal{NP}$-complet.

Preuve
Pour montrer que le problème appartient à $\mathcal{NP}$, il suffit de former une liste quelconque d'arcs et de vérifier si ces arcs forment un chemin commençant par s, terminant par t et comprenant tous les sommets. Cet algorithme est clairement polynomial.

Pour montrer la $\mathcal{NP}$-complétude, on réduit le problème de la couverture de sommets au problème du chemin de Hamilton. Soient $G = (V, E)$ et k, les entrées pour la couverture de sommets. Pour chaque sommet v, soit $(v,1),(v,2)\dots(v,d(v))$ les arêtes incidentes sur v, où $d(v)$ est le degré de v.

Le graphe orienté $G' = (V', E')$ de la réduction, entrée pour le problème du chemin de Hamilton, est défini comme suit:

$$V' = \{a_0, a_1, \dots, a_k\} \cup \bigcup_{v\in V} \{(v,1,i),(v,2,i) | 1 \leq i \leq d(v)\}$$

Notons que les a_i sont des nouveaux symboles et que pour chaque $v \in G$ il y a $2d(v)$ sommets dans G'. Les arêtes sont définies comme:

$$\begin{gathered} E' = \{(a_i, (v,1,1)) \mid 0 \leq i < k \text{ et } v \in V\} \cup \\ \{((v,2,d(v)), a_i) \mid 0 < i \leq k \text{ et } v \in V\} \cup \\ \{((u,1,i),(v,1,j)),((u,2,i),(v,2,j)) \mid (u,i) = (v,j) \text{ dans } G\} \cup \\ \{((v,1,i),(v,2,i)) \mid v \in V \text{ et } 1 \leq i \leq d(v)\} \cup \\ \{((v,2,i),(v,1,i+1)) \mid v \in V \text{ et} 1 \leq i < d(v)\} \end{gathered}$$

Les deux dernières parties de la définition de E' décrivent un chemin direct de $(v,1,1)$ à $(v,2,d(v))$ qui passe par tous les autres $2d(v) - 2$ sommets de G' associés à v; on appelle ce chemin la trace de v. Pour toutes les arêtes (u,v) de G il y a une connexion entre les traces de u et de v comme spécifié dans la troisième partie de la définition de E'. Pour $0 \leq i < k$ et $v \in V$, a_i est connecté à $(v,1,1)$ par la première partie de la définition de E'. Par la deuxième partie de la définition de E' si $0 < i \leq k$ *et* $v \in V$, les $(v,2,d(v))$ sont connectées à a_i. Alors, les a_i sont utilisés comme des passages entre les traces.

Soit $s = a_0$ et $t = a_k$; ce qui complète la définition de la fonction de réduction. On doit montrer que G a une couverture de sommets de longueur k si et seulement s'il y a un chemin de Hamilton de a_0 à a_k dans G'.

Supposons que $C = \{v_1, v_2 \dots v_k\}$ soit une couverture de sommets pour G. On peut construire un chemin de Hamilton dans G' comme suit: on commence avec une arête de a_0 à $(v_1,1,1)$, on utilise maintenant le trace de v_1 pour aller à $(v_1,2,d(v_1))$, après de a_1 à $(v_2,1,1))$, alors on utilise la trace de v_2 pour aller à $v_2,2,d(v_2))$, etc. Finalement, de a_k à $(v_k,2,d(v_k))$. Encore, pour toutes les arêtes (u,v) de G, pour lesquelles un sommet u appartient à C mais l'autre v n'appartient pas à C, les sommets $(v,1,j)$ et $(v,2,j)$ pour $(v,j) \in V$ sont inclus dans le chemin en faisant un détour de trace u; pour compléter ces cas soit (u,i) une arête, alors au lieu d'aller de $(u,1,i)$ à $(u,2,i)$ directement, on fait le détour $(u,1,i) \to (v,1,j) \to (v,2,j) \to (u,2,i)$. Puis que C est une couverture de sommets, on peut inclure tous les sommets dans les traces qui ne sont pas utilisées; ce qui donne le chemin de Hamilton.

Réciproquement, si P est un chemin de Hamilton de a_0 à a_k dans G', on peut construire une couverture de sommets S dans G avec k sommets. Le sommet $v \in S$ si et seulement si la trace de v est utilisée dans P. Le nombre de sommets de S est exactement k. Considérez un sommet (u, v) de G. Si les traces de v et de u sont utilisées dans P, alors l'arête (u, v) est couverte deux fois. Sinon, la seule manière d'avoir $(u, 1, i), (u, 2, i), (v, 1, j)$ et $(v, 2, j)$ dans P est d'utiliser soit la trace de u soit la trace de v. Alors, l'arête (u, v) est couverte soit par u soit par v. Donc, toutes les arêtes sont couvertes et S est une couverture de sommets.

On vient de prouver la réduction polynomiale du problème de la couverture des sommets au problème du chemin de Hamilton. Ce qui complète la preuve de que le problème du chemin de Hamilton est $\mathcal{NP}$-complet. □

7.6.6 Chemin de Hamilton pour un graphe non orienté

L'énoncé est identique au problème précédent à la différence que le graphe en question est non orienté.

Théorème 7.6.7
Le problème du chemin de Hamilton pour un graphe non orienté est $\mathcal{NP}$-complet.

Preuve
L'appartenance à $\mathcal{NP}$ est triviale à démontrer et ressemble beaucoup à celle du théorème précédent. Pour prouver l'appartenance à $\mathcal{NP}$-hard, on fait une réduction polynomiale du problème du chemin de Hamilton pour graphes orientés aux graphes non orientés. Soit $G = (V, E)$, s et t les entrées pour le cas des graphes orientés. La fonction de réduction donnera le graphe non orienté suivant:

$$\begin{aligned} V' &= \{(v,0),(v,1),(v,2) \mid v \in V\} \\ E' &= \{(v,0) \leftrightarrow (v,1), (v,1) \leftrightarrow (v,2) \mid v \in V\} \cup \\ &\quad\ \{(u,2) \leftrightarrow (v,0) \mid u \to v \in E\} \end{aligned}$$

Les deux points terminaux seront $(s, 0)$ et $(t, 2)$.

Il est aisé de remarquer que s'il y a un chemin de Hamilton P de s à t dans G, alors il y a un chemin de Hamilton P' dans G' commençant par $(s, 0)$ et terminant par $(t, 2)$. P' est composé de toutes les arêtes dans la première partie de la définition de E', et si $u \to v \in P$, alors $(u, 2) \leftrightarrow (v, 0)$ est inclue dans P'.

Supposons maintenant que P' est une solution pour G'. Alors, toutes les arêtes de la première partie de la définition de E' doivent apparaître dans P', sinon il n'y a pas manière de passer par $(v, 1)$. Alors, on peut examiner P' de $(s, 0)$ à $(t, 2)$; il doit commencer avec $(s, 0) \leftrightarrow (s, 1) \leftrightarrow (s, 2)$ et terminer avec $(t, 0) \leftrightarrow (t, 1) \leftrightarrow (t, 2)$. A chaque fois que $(u, 2) \leftrightarrow (v, 0)$ est utilisé dans P', on utilise (u, v) dans P. Le chemin résultant passe par s, termine par t et passe par tous les sommets de G une seule fois.

Donc, la réduction est correcte et le problème est $\mathcal{NP}$-complet. □

7.6.7 L'association d'éléments tridimensionnels (3DM)

Un problème important comme problème de base pour prouver la $\mathcal{NP}$-complétude de plusieurs autres problèmes est le 3DM. Soit l'ensemble $M \subseteq X \times Y \times Z$, où X, Y, Z sont des ensembles disjoints et ayant le même nombre d'éléments q. Le problème 3DM (*3-dimensional matching*) consiste à faire une association entre les éléments des 3 sous-ensembles en formant un sous-ensemble $M' \subseteq M$ tel que $|M'| = q$ et qu'aucun élément de M' ne coïncide avec un autre du même ensemble sur une coordonnée. Le problème consiste à déterminer s'il existe un sous-ensemble M' avec toutes ces conditions. Remarquez que ce problème est la généralisation du problème du mariage classique; donnés n hommes célibataires et n femmes célibataires et une liste de gens qui voudraient se marier, on doit sortir n mariages de telle manière que la polygamie soit évitée et que tous les célibataires trouvent un conjoint. Le 3DM correspond à un mariage entre 3 types différents de célibataires. Remarquons que le problème du mariage classique peut être résolu en temps polynomial mais le 3DM est $\mathcal{NP}$-complet et sa preuve est assez compliquée.

Théorème 7.6.8
Le problème 3DM est $\mathcal{NP}$-complet.

Preuve
Il est évident de voir que le problème 3DM est dans $\mathcal{NP}$. En effet, il suffit d'appliquer un algorithme non déterministe pour former les triplets à partir de M, et de vérifier en un temps polynomial que les triplets ainsi formés ne coïncident sur aucune coordonnée.

La deuxième étape consiste à transformer le problème 3-SAT au problème 3DM. Soit F une formule sous forme normale conjonctive avec l'ensemble de clauses $C = \{C_1, C_2, \ldots, C_m\}$ et $x_1, x_2 \ldots c_n$ les variables dans la formule.

L'idée de la preuve est d'avoir un ensemble de triplets AC pour représenter la consistance entre les variables; c'est-à-dire pour garantir qu'une variable dans C donne toujours la même valeur et son complément la valeur complémentaire. On doit avoir un autre ensemble de triplets SC pour représenter la satisfaisabilité; c'est-à-dire pour garantir que toutes les clauses seront satisfaisables. Finalement, il y aura un autre ensemble des triplets, GC, pour assurer de trouver un matching si toutes les conditions précédentes sont remplies.

On construit les ensembles disjoints X, Y et Z avec la condition $|X| = |Y| = |Z|$ de la manière suivante:

$$X = \{x_{ij}, \bar{x}_{ij} \mid 1 \leq i \leq n, 1 \leq j \leq m\}$$

Alors, pour chaque variable x_i et chaque clause C_j il y a un élément x_{ij} pour représenter la présence possible de x_i dans C_j; de même pour $\bar{x}_i$ dans C_j.

$$Y = A_1 \cup S \cup G \text{ et } Z = A_2 \cup S \cup G$$

ou A_i, S et G sont disjoints pair à pair. Ils sont définis comme:

$$\begin{aligned} A_1 &= \{a_{ij} \mid 1 \leq i \leq n, 1 \leq j \leq m\} \\ A_2 &= \{b_{ij} \mid 1 \leq i \leq n, 1 \leq j \leq m\} \\ S &= \{s_j \mid 1 \leq j \leq m\} \\ G &= \{g_k \mid 1 \leq k \leq m(n-1)\} \end{aligned}$$

Maintenant, on fait $M = AC \cup SC \cup GC$, avec AC, SC et GC disjoints pair à pair et définis comme:

$$AC = \bigcup_{i=1}^{n} AC_i$$

où,

$$AC_i = \{(x_{ij}, a_{ij}, b_{ij}) \mid 1 \leq j \leq m\} \cup \{(\bar{x}_{i,j}, a_{i,j+1}, b_{ij}) \mid 1 \leq j \leq m-1\} \cup \\ \{(\bar{x}_{i,j}, a_{ij}, b_{im})\}$$

Remarquez que chaque a_{ij} et b_{ij} participent dans seulement deux triplets, alors un matching possible M' doit avoir soit tous les triplets de la forme (x_{ij}, a_{ij}, b_{ij}) ou de l'autre forme, mais il n'y aura pas de mélange possible. Ça représente la consistance, à savoir: soit x_i est vrai soit il est faux dans toutes les clauses.

$$SC = \bigcup_{j=1}^{m} SC_j$$

où

$$SC_j = \{(x_{ij}, s_j, s_j) \mid x_i \in C_j\} \cup \{(\bar{x}_{ij}, s_j, s_j) \mid \bar{x}_i \in C_j\}$$

Puisque l'on utilise 3-SAT, chaque SC_j contient trois triplets. Pour couvrir s_j, comme une composante aux deuxième et troisième dimensions, un et seulement un des triplets peut appartenir au matching M'. Ça ne peut arriver que si x_{ij} ($\bar{x}_{ij}$) n'a pas été utilisé encore dans $M \cap AC$, quand x_i est rendu vrai (faux).

Jusqu'à maintenant $M' \cap (AC \cup SC)$ utilise $mn + m$ des $2mn$ éléments de W. Il faut pour compléter la réduction couvrir le restant $m(n-1)$ éléments, alors:

$$GC = \{(x_{ij}, g_k, g_k), (\bar{x}_{ij}, g_k, g_k) \mid 1 \leq i \leq n, 1 \leq j \leq n, 1 \leq k \leq m(n-1)\}$$

Donc, on a un matching M' si et seulement si la formule F est satisfaisable. Et la réduction est réalisée en temps polynomial. Alors ce problème et $\mathcal{NP}$-hard, ce qui complète la preuve. $\square$

7.7 D'autres problèmes $\mathcal{NP}$-complets

On énonce ici quelques problèmes $\mathcal{NP}$-complets sans prouver leur réduction. Il y a de très nombreux problèmes qui sont connus comme $\mathcal{NP}$-complets, on trouvera les preuves dans la littérature.

Couverture exacte par trois ensembles (Exact Cover by 3 Sets – X3C)

Étant donné un ensemble S avec $3q$ éléments et une collection C de sous-ensembles de S avec 3 éléments, on demande si C a une couverture exacte de S, c'est-à-dire une sous-collection $C' \subseteq C$ telle que chaque élément de S apparaisse exactement dans un composant de C'.

Remarque : Réduction assez simple à partir de 3DM.

Problème du circuit de Hamilton

Soit G un graphe donné, il s'agit de trouver un chemin qui visite tous les sommets du graphe exactement une fois et revienne au point de départ.

Remarque : Réduction assez simple du problème du chemin de Hamilton.

Problème de la partition

Etant donné l'ensemble A et la fonction à résultat entier $s(a) > 0, a \in A$, le problème de la partition consiste à trouver un sous-ensemble $A' \subseteq A$ dont les éléments vérifient l'égalité suivante:

$$\sum_{a \in A \setminus A'} s(a) = \sum_{a \in A'} s(a)$$

Remarque : Réduction à partir de 3DM.

Problème du KNAPSACK

Soient un ensemble fini U, la fonction "taille" ("size") $s(u) > 0$ et la fonction "valeur" $v(u) > 0$, pour tout $u \in U$, il existe des valeurs seuils, $B > 0$ pour la taille limite, et $K > 0$ pour la valeur limite. Il s'agit alors de trouver un sous-ensemble $U' \subseteq U$ tel que:

$$\sum_{u \in U'} s(u) \leq B \; et \; \sum_{u \in U'} v(u) \geq K$$

Remarque : Pour la preuve, il suffit de faire la réduction de la partition au Knapsack, avec $s(u) = v(u)$ et $B = K = \frac{1}{2}\sum_{u \in U} s(u)$.

Arbre de recouvrement borné

Soit un graphe $G = (V, E)$ et un entier positif $k \leq |V| - 1$. Il s'agit de trouver un arbre de recouvrement pour G tel que pour tout sommet $v \in V$, $d(v) \leq k$ où $d(v)$ est le degré de v et un sous ensemble $E' \subseteq E$ tel que $|E'| = |V| - 1$. Le graphe $G' = (V, E')$ est connecté et aucun sommet $v \in V$ n'est connecté à plus de k arêtes de E'. Remarque : La réduction du problème de l'arbre de recouvrement se fait avec le problème du circuit de Hamilton.

Sous-graphe isomorphique

Soient deux graphes, $G_1 = (V_1, E_1)$ et $G_2 = (V_2, E_2)$. Il s'agit de trouver un sous-graphe de G_1 isomorphe à G_2 dont le sous-ensemble des sommets et le sous-ensemble des arêtes $E \subseteq E_1$, et $|V| = |V_2|$, $|E| = |E_2|$, et il existe une fonction $f : V_2 \to V$ satisfaisant $(u, v) \in E_2$ si et seulement si $(f(u), f(v)) \in E$.

Remarque : La réduction de ce problème se fait avec le problème de la clique.

Problème du voyageur de commerce

Soit un ensemble fini $C = \{c_1, c_2, \ldots, c_n\}$ de villes. Pour toute ville $c_i, c_j \in C$, $d(c_i, c_j) > 0$, où $d(c_i, c_j)$ est la distance entre les villes c_i, c_j. On se donne une borne $B > 0$. Il s'agit de trouver un circuit qui visite toutes les villes une seule fois et revienne au point de départ avec la distance totale du parcours plus petite ou égale à B.

Remarque : Réduction triviale à partir du problème du circuit de Hamilton, en définissant $d(c_i, c_j) = 1$ et B comme le nombre des sommets du graphe.

7.8 La classe co-$\mathcal{NP}$

Considérez le problème $\overline{\text{SAT}}$ comme le complément de SAT, c'est-à-dire le problème de déterminer si une formule booléenne n'est pas satisfaisable. Soit M une machine non déterministe qui résolve SAT en temps polynomial et soit $\overline{M}$ la machine que l'on obtient en interchangeant les états finaux et non finaux de M. Il est facile de voir que le langage accepté par $\overline{M}$ n'est pas $\overline{\text{SAT}}$ mais le langage composé de formules qui ont au moins une interprétation qui ne les satisfait pas. Tous les essais pour obtenir une machine non déterministe qui résolve $\overline{\text{SAT}}$ en temps polynomial ont échoué jusqu'à présent. Plus formellement, on définit la classe complémentaire de $\mathcal{NP}$ comme suit.

Définition 7.8.1
La classe co-$\mathcal{NP}$ est l'ensemble des compléments des langages appartenant à la classe $\mathcal{NP}$; c'est-à-

dire:

$$\text{co-}\mathcal{NP} = \{L \mid \overline{L} \in \mathcal{NP}\}$$

$\diamondsuit$

Remarquons que le problème $\overline{\text{SAT}}$ est co-$\mathcal{NP}$-complet. Jusqu'à maintenant personne n'a encore trouvé si $\mathcal{NP}$ = co-$\mathcal{NP}$ ou pas. Notons aussi qu'il existe une forte connexion entre les problèmes $\mathcal{NP}$-complets et la conjecture $\mathcal{NP} \neq$ co-$\mathcal{NP}$.

Théorème 7.8.1
Soit L $\mathcal{NP}$-complet, si $\overline{L} \in \mathcal{NP}$, alors co-$\mathcal{NP} = \mathcal{NP}$.

Preuve
Soit $L_1 \in \mathcal{NP}$. Alors $L_1 \propto L$, car L est $\mathcal{NP}$-complet par hypothèse et $L \propto \overline{L}$ trivialement, donc $L_1 \propto \overline{L}$ Comme $\overline{L} \in$ co-$\mathcal{NP}$, car $L \in \mathcal{NP}$ par hypothèse, alors $L_1 \in$ co-$\mathcal{NP}$. Réciproquement, supposons $L_1 \in$ co-$\mathcal{NP}$. Alors, $L_1 \propto \overline{L}$, car L est $\mathcal{NP}$-complet et donc $\overline{L}$ est co-$\mathcal{NP}$-complet, en plus $\overline{L} \propto L$ trivialement. Donc, $L_1 \propto L$ et $L \in \mathcal{NP}$, alors $L_1 \in \mathcal{NP}$. $\square$

Une application intéressante de ce résultat est quand un langage L et $\overline{L}$ sont dans $\mathcal{NP}$. A moins que $\mathcal{NP}$ = co-$\mathcal{NP}$, L ne peut pas être $\mathcal{NP}$-complet. Un exemple d'un tel langage est le suivant:

$$L_{\text{Premier}} = \{w \mid w \text{ est premier}\}$$

En réalité aucun langage a été trouvé satisfaisant les conditions du théorème; alors la conjecture $\mathcal{NP} \neq$ co-$\mathcal{NP}$ a été proposée, mais cette conjecture est encore plus forte que la conjecture $\mathcal{P} \neq \mathcal{NP}$, puisque:

Théorème 7.8.2
Si co-$\mathcal{NP} \neq \mathcal{NP}$, alors $\mathcal{P} \neq \mathcal{NP}$.

Preuve
Par contradiction on suppose que $\mathcal{P} = \mathcal{NP}$, ce qui implique co-$\mathcal{P}$ = co-$\mathcal{NP}$. Mais $\mathcal{P}$ = co-$\mathcal{P}$, donc $\mathcal{NP}$ = co-$\mathcal{NP}$, ce qui va à l'encontre de l'hypothèse co-$\mathcal{NP} \neq \mathcal{NP}$. $\square$

7.9 Complétude pour autres classes

La notion de complétude n'est pas exclusive à la classe $\mathcal{NP}$; en réalité elle est valable pour toutes les classes de complexité. Cette notion a été exhaustivement étudiée pour $\mathcal{NP}$, mais il y a d'autres exemples intéressantes au-dessous et au-dessus de $\mathcal{NP}$, comme pour les classes $\mathcal{P}$ et PSPACE.

7.9.1 Problèmes $\mathcal{P}$-complets

On peut définir un problème $\mathcal{P}$-complet par un langage L tel que:

1. $L \in \mathcal{P}$
2. $\forall L' \in \mathcal{P} : L' \propto L$

Il y a plusieurs exemples de problèmes $\mathcal{P}$-complets; un exemple relativement naturel est le 2-SAT, la satisfaisabilité des formules booléennes sous forme normale conjonctive, ou chaque disjonction est composée par au plus 2 variables. En plus d'être complets pour $\mathcal{P}$, ces problèmes semblent ne pas avoir d'algorithmes parallèles efficaces pour les résoudre. En réalité, un algorithme séquentiel pour un problème $\mathcal{P}$-complet semble être plus efficace qu'une solution parallèle.

7.9.2 Problèmes PSPACE-complets

On peut définir un problème PSPACE-complet par un langage L tel que:

1. $L \in$ PSPACE
2. $\forall L' \in$ PSPACE : $L' \propto L$

Un exemple de problème complet pour PSPACE est celui de la satisfaisabilité pour les formules booléennes quantifiées (QBF). Une formule booléenne quantifiée est une formule construite à partir de variables, d'opérateurs *et*, *ou*, *complément* , des parenthèses, et des quantificateurs $\forall$ et $\exists$. Le problème QBF consiste à déterminer si une formule QBF peut avoir la valeur vraie.

Exemple 7.9.1 Soit la formule booléenne quantifiée suivante n'est pas satisfaisable :

$$\forall x(\forall x(\exists y(x \vee y)) \wedge \bar{x})$$

Notons que le problème de la satisfaisabilité pour une formule QBF est assez compliqué. Malheureusement, la classe PSPACE n'est pas la dernière en matière de degré de difficulté. Il y a d'autres classes comme E, EXP, NE, NEXP en relation avec la complexité en temps qui sont au-dessus de PSPACE; il y a la hiérarchie polynomiale qui tombe exactement sur PSPACE. Si l'on change les modèles de calculabilité pour des machines de Turing probabilistes on définit des classes de complexité utiles qui sont entre $\mathcal{P}, \mathcal{NP}$ et PSPACE. On peut aussi définir les systèmes interactifs de preuve comme la classe de complexité IP; qui a été prouvé récemment correspondre exactement à la classe PSPACE.