Cours d'informatique théorique 2 de M. Arfi

FMdKdD
fmdkdd [à] free.fr

Université du Havre
Année 2010–2011

Table des matières

Chapitre 1

Préliminaires et compléments

Le mot anglo-saxon pour désigner un ordinateur est *computer*, qui signifie « calculer ». Il s'agit d'un mot plus révélateur que le mot français. Une définition de l'informatique : traitement automatique de l'information.

1.1 Historique

Rappel d'histoire (version courte) : fatigué de chasser les mammouths, l'Homme se met à réfléchir, et commence par compter. Chronologie :

- Le boulier chinois (IX^{e} siècle avant J.-C.) : appareil comprenant des tringles métalliques sur lesquelles sont enfilées des boules, et servant à compter.
- Les automates :
  - Les horloges : automates mécaniques permettant de mesurer le temps. Améliorés ensuite par l'adjonction d'un programme interne figé. Les Jaquemarts que l'on trouve encore dans certaines églises en sont un bon exemple.
  - La pascaline : machine arithmétique de Pascal (1642). Il s'agit d'une machine à calculer mécanique, gérant les retenues, capable de faire des additions et des soustractions, grâce à un système de roues dentées. La nouveauté réside dans la possibilité d'introduire les données dans la machine.
  - Métier à tisser de Jacquard (1801) : automatisation du fonctionnement des métiers à tisser. Le programme est, chose nouvelle, introduit à l'extérieur à l'aide de cartons perforés, réutilisables et modifiables.
- Le machine analytique de Babbage (1833) : il s'agit d'un projet de machine arithmétique comportant deux parties : le « moulin » où sont effectuées les opérations, et la « mémoire » qui contient les données à traiter et les résultats intermédiaires. L'introduction des opérations a réaliser est effectuée à l'aide de cartes perforées. Cette machine regroupe les deux propriétés des machines précédemment décrites. Faute de moyens financiers et technologiques, la machine reste inachevée.
- Les machines électromagnétiques :
  - Les machines électromécaniques.
  - Les machines électriques.
  - Les machines électroniques.

  Le résultat : l'utilisation de la représentation binaire pour les informations.

  - 1924 : naissance d'IBM (USA).
  - 1933 : naissance de BULL (France).

  Durant la seconde guerre mondiale, l'ENIAC réalise de calculs de balistique.
- Progrès théoriques : sur le plan théorique, les travaux d'Alan Turing sur la calculabilité constituent le progrès le plus remarquable de l'intervalle séparant les deux guerres. Il décrit une machine théorique capable de calculer toute fonction supposée intuitivement calculable. L'hypothèse d'alors, selon laquelle toute fonction intuitivement calculable l'est par une machine de Turing appropriée, est devenue de nos jours une définition de la calculabilité.
- Naissance de l'ordinateur : en 1946 est énoncé le principe d'une machine théorique, dont la paternité sera attribuée à Von Neumann, et qui est à la base de tous les ordinateurs modernes. Le modèle de Von Neumann, contrairement à celui de Turing, est opérationnel. La première machine commerciale, l'UNIVAC I, date de 1950. Il s'agit d'une machine à tubes à vide.
- Ordinateurs modernes : les ordinateurs modernes sont à base de transistors, qui apparaissent en 1960. Ils seront remplacés par des ordinateurs de troisième génération, à base de circuits intégrés, puis de quatrième génération, toujours à base de circuits intégrés, mais à très haut niveau d'intégration. La commercialisation du premier microprocesseur (Intel) a facilité une telle évolution. Ce sont en fait des unités centrales d'ordinateurs que les techniques de miniaturisation ont permis de condenser sur des pastilles de silicium de quelques millimètres carrés (voire beaucoup moins), et dont le prix s'élève à quelques dizaines d'euros.

1.2 Limites du calculable

On ne peut pas tout calculer en utilisant un ordinateur. On verra en fait qu'il y a bien plus de problèmes non calculables que de problèmes calculables.

À tout problème, on peut associer un langage sur un certain alphabet qui vise à coder le dit problème. Inversement, à tout langage, il est possible d'associer un problème, ne serait-ce que l'appartenance d'un mot quelconque au dit langage. Or, selon la thèse de Church-Turing, conjecture jamais contredite à ce jour, tout problème que l'on peut résoudre à l'aide d'une procédure effective, c'est à dire un algorithme, on peut le résoudre en utilisant une machine de Turing, et vice-versa. On verra pas la suite que le nombre de machines de Turing est une quantité dénombrable, alors que celui des problèmes (ou langages) n'est pas dénombrable. On en déduit qu'il existe dans la nature infiniment plus de problèmes que d'algorithmes pour les résoudre.

Exemple. Problème de l'arrêt d'un algorithme.

Soit P un prédicat sur un ensemble E, c'est à dire une propriété susceptible d'être vérifiée par les éléments de E. On peut considérer P comme une application de E dans $\mathrm{B} = \{\text{vrai}, \text{faux}\}$. On dit que P est décidable (calculable) s'il existe un algorithme permettant de savoir si un élément quelconque $x \in \mathrm{E}$ satisfait ou non P. En particulier, une partie $\mathrm{X} \in \mathscr{P}(\mathrm{E})$ est dite décidable si le prédicat « $x \in \mathrm{X}$ » défini sur E est calculable.

Montrer que le problème de l'arrêt d'un algorithme est indécidable.

Supposons par l'absurde ce problème décidable. Cela signifie qu'il existe un algorithme A_0, acceptant en donnée un algorithme A et un jeu de données D pour A,

ou plutôt le codage de ces deux derniers, et produisant la sortie « vrai » si (A, < D >) s'arrête, et faux sinon. (< D > représentant le codage de D).

Considérons dans l'ensemble de tous les algorithmes, la classe de ceux qui prennent en donnée un autre algorithme, ou plutôt le codage de ce dernier. Et dans cette classe, soit l'algorithme particulier A_1 suivant :

```
Entrée : un algorithme A de la classe précédente.

début
  tantque (A_0,<A,A>) faire
fin
```

Appliquons l'algorithme A_1 à lui-même. On obtient :

$$\begin{aligned}(A_1, < A_1 >) \text{ s'arrête} &\iff (A_0 < A_1, A_1 >) \text{ est faux} \\ &\iff (A_1, < A_1 >) \text{ ne s'arrête pas}\end{aligned}$$

Contradiction.

On va procéder dans les chapitres suivants à une étude de la calculabilité sous trois angles différents, mais tous équivalents :
- les fonctions calculables,
- les machines à registres,
- les machines de Turing.

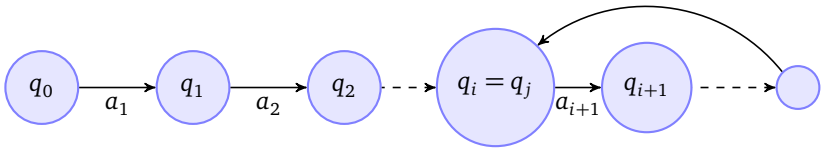
1.3 Grammaires formelles

Nous avons étudié l'an passé exclusivement les langages rationnels, sur un alphabet fini, c'est à dire ceux reconnus par un automate fini, que l'on peut même supposer déterministe complet. Mais il existe, sur un tel alphabet, infiniment plus de langages qui ne sont pas rationnels.

Exemple. Considérons, sur l'alphabet $\Sigma = \{a, b\}$, le langage $L = \{a^n b^n / n \geqslant 0\}$. Montrons que L n'est pas rationnel.

Supposons par l'absurde L rationnel. Il est alors reconnaissable, en vertu du théorème de Kleene. Il existe donc un automate fini, que l'on peut même prendre déterministe complet, $A = (Q, \Sigma, \delta, q_0, F)$ reconnaissant L. Posons $k = |Q|$ et notons $q.w$, pour simplifier l'écriture, l'unique état atteint dans l'automate par le mot w à partir de l'état q. Si $w = a_1 a_2 \dots a_n$, $a_i \in \Sigma$ pour $1 \leqslant i \leqslant n$, on note $(q_0, q_1, \dots, q_n)$ le chemin associé à w dans l'automate A en partant de l'état initial q_0. On a alors $q_i.a_{i+1} = q_{i+1}$, $\forall i \in \{0, \dots, n-1\}$.

Soit alors le mot $u = a^k b^k$. Il est clair que $u \in L$, c'est à dire que $q_0.u \in F$. Si $(q_0, \dots, q_k, \dots, q_{2k})$ désigne le chemin associé à u dans A à partir de q_0, $(q_0, \dots, q_k)$ sera celui associé à a^k à partir du même état. Puisque ce dernier chemin implique $k+1$ état, il existe nécessairement deux entiers i et j, $0 \leqslant i < j \leqslant k$ vérifiant $q_i = q_j$.



Considérons maintenant le mot $v = a^{k+j-i}b^k$. Il est évident que $v \notin L$, puisque $i \neq j$. Or, on obtient successivement :

$$\begin{aligned} q_0.v &= q_0.a^{k+j-i}b^k \\ &= q_0.a^{j-i+k}b^k \\ &= q_0.a^j a^{k-i}b^k \\ &= q_j.a^{k-i}b^k \\ &= q_i.a^{k-i}b^k \\ &= q_0.a^i a^{k-i}b^k \\ &= q_0.a^k b^k \\ &= q_0.u \in F \end{aligned}$$

On en déduit que $v \in L$, ce qui aboutit à une contradiction.

Définition 1.1. Une grammaire est un quadruplet $G = (V, A, P, S)$ où :
- V est un ensemble fini de symboles, appelés variables ou symboles non terminaux.
- A est un ensemble fini de lettres, appelées symboles terminaux, vérifiant $V \cap A = \emptyset$.
- $S \in V$ est le symbole initial, ou l'axiome.
- P est une partie finie du produit cartésien $\left[(V \cup A)^+ \setminus A^+\right] \times (V \cup A)^*$. Les éléments de P sont appelés productions de la grammaire. On note habituellement $\alpha \to_G \beta$, ou simplement $\alpha \to \beta$ pour signifier que $(\alpha, \beta) \in P$.

Exemple. Considérons la grammaire $G = (V, A, P, S)$ avec $V = \{S\}$, $A = \{a, b\}$ et les productions $S \to aSb$ et $S \to \varepsilon$, que l'on peut contracter en la seule écriture $S \to aSb|\varepsilon$.

Définitions 1.2. On dit qu'un mot $w' \in (V \cup A)^*$ dérive directement du mot w, et on note $w \Rightarrow_G w'$, s'il existe $u, v, x, y \in (V \cup A)^*$ tels que $w = xuy$, $w' = xvy$ et $u \to v$ est une production de G. Par exemple, en utilisant la grammaire précédente, on peut écrire

$$S \to aSb \Rightarrow a^2Sb^2 \Rightarrow a^3Sb^3 \Rightarrow a^3b^3$$

Un mot w' dérive du mot w, et on note $w \Rightarrow_G^* w'$, si $w = w'$ ou s'il existe une chaîne de mots $w_0, \ldots, w_n \in (V \cup A)^*$ tels que $w = w_0$, $w' = w_n$ et $w_i \Rightarrow_G w_{i+1}$, $\forall i \in \{0, \ldots, n-1\}$. La relation $\Rightarrow^*$ représente en fait sur les mots de $(V \cup A)^*$ la fermeture à la fois réflexive et transitive de la relation $\Rightarrow$, c'est à dire la plus petite relation réflexo-transitive (de pré-ordre) contenant $\Rightarrow$.

Définition 1.3. Soit $G = (V, A, P, S)$ une grammaire. On appelle langage engendré par G, et on note L(G), celui défini par :

$$L(G) = \{w \in A^* / S \Rightarrow_G^* w\}$$

On dira que deux grammaires G et G′ sont équivalentes lorsqu'elles engendrent le même langage.

Exemple. Le langage qui correspond à la grammaire G de l'exemple précédent est exactement :

$$\mathrm{L(G)} = \{a^n b^n / n \geqslant 0\}$$

Exemple. Considérons la grammaire $\mathrm{G}' = (\{\mathrm{E}\}, \mathrm{A}, \mathrm{P}, \mathrm{E})$ avec $\mathrm{A} = \{(,), +, \times, x_1, x_2, \ldots, x_n\}$ et les productions :

$$\mathrm{E} \to (\mathrm{E}) | \mathrm{E} + \mathrm{E} | \mathrm{E} \times \mathrm{E} | x_1 | \ldots | x_n$$

L(G′) représente l'ensemble des expressions arithmétiques bien parenthésées en les variables $x_1, \ldots, x_n$.

Exemple. Soit la grammaire $\mathrm{G}'' = (\{\mathrm{S}, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}\}, \{a\}, \mathrm{S}, \mathrm{P})$ ayant pour productions :

$$\begin{aligned}
\mathrm{S} &\to \mathrm{AC}a\mathrm{D} \\
\mathrm{C}a &\to a^2\mathrm{C} \\
\mathrm{CB} &\to \mathrm{DB} | \mathrm{E} \\
\mathrm{AD} &\to \mathrm{D}a \\
\mathrm{AD} &\to \mathrm{AC} \\
\mathrm{AE} &\to \mathrm{E}a \\
\mathrm{AE} &\to \varepsilon
\end{aligned}$$

On constate que $\mathrm{L(G}'') = \{a^{2^n} / n > 0\}$. Établissons un schéma de la preuve.

Notons provisoirement $\mathrm{L} = \{a^{2^n} / n > 0\}$. Pour montrer $\mathrm{L(G}'') \subset \mathrm{L}$, on peut démontrer par récurrence sur le nombre de pas d'une dérivation que si la production $\mathrm{CB} \to \mathrm{E}$ n'est jamais utilisée, on obtient l'une des possibilités suivantes :

1. S
2. $\mathrm{A}a^i\mathrm{C}a^j\mathrm{B}$, $i + 2j$ est une puissance de 2
3. $\mathrm{A}a^i\mathrm{D}a^j\mathrm{B}$, $i + j$ est une puissance de 2

Si l'on utilise la production $\mathrm{CB} \to \mathrm{E}$, on arrive à un résultat de la forme $\mathrm{A}a^i\mathrm{E}$, où i est une puissance de 2. La seule possibilité ensuite est d'appliquer i fois $a\mathrm{E} \to \mathrm{E}a$ et une fois $\mathrm{AE} \to \varepsilon$.

Pour montrer $\mathrm{L} \subset \mathrm{L(G}'')$, il est encore possible de montrer par récurrence que l'on obtient :

$$\begin{aligned}
\mathrm{S} &\Rightarrow^* \mathrm{A}a^{2^n}\mathrm{CB}, \quad \forall n > 0 \\
&\Rightarrow^* a^{2^n}
\end{aligned}$$

Tout mot de L est un résultat de la grammaire G″.

1.4 Hiérarchie de Chomsky

Une grammaire formelle sans aucune restriction sur les productions, autre que celle figurant dans la définition, est dite de type 0 (ou semi-Thue).

Une grammaire $G = (V, A, P, S)$ est dite sensible au contexte (*context sensitive*), ou de type 1, si toute production de P est de la forme :

$$u\alpha v \to uxv$$

avec $\alpha \in V$, $x \in (V \cup A)^+$, et $u, v \in (V \cup A)^*$.

On dit que G est libre du contexte (*context free*), ou de type 2, si toute production de P a pour forme :

$$\alpha \to x$$

avec $\alpha \in V$, et $x \in (V \cup A)^*$.

La grammaire G est dite linéaire droite, ou de type 3, si ses productions sont toutes de la forme :

$$\alpha \to x\beta, \quad \text{ou} \quad \alpha \to y$$

avec $\alpha, \beta \in V$ et $x, y \in A^*$.

On a aussi la notion, peu utilisée, de grammaire linéaire gauche.

Si l'on désigne par $\mathscr{L}_i$ la classe des langages, sur un alphabet fini A, engendrés par une grammaire de type i, on a :

$$\mathscr{L}_3 \subset \mathscr{L}_2 \subset \mathscr{L}_1 \subset \mathscr{L}_0$$

et toutes ces inclusions sont strictes.

Les langages de ces classes et les machines qui les reconnaissent :

- $\mathscr{L}_3$: langages rationnels, reconnus par les automates finis.
- $\mathscr{L}_2$: langages algébriques (non contextuels), reconnus par les automates à pile.
- $\mathscr{L}_1$: langages contextuels $\subsetneq$ langages récursifs, reconnus par les machines de Turing.
- $\mathscr{L}_0$: langages récursivement énumérables, reconnus également par les machines de Turing.

Définition 1.4. On dit qu'un ensemble E est dénombrable (*countable*) s'il existe une bijection de $\mathbb{N}$ sur E. Autrement dit, s'il existe une suite $(x_n)_{n\geqslant 0}$ d'éléments de E à la fois injective et surjective :

1. $\forall i, j \in \mathbb{N}, i \neq j \Rightarrow x_i \neq x_j$
2. On a $E = \{x_n / n \geqslant 0\}$

Exemple. L'application identique $\mathrm{Id}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}$ est clairement bijective. L'ensemble $\mathbb{N}$ est par conséquent dénombrable.

L'ensemble des langages récursivement énumérables, c'est à dire ceux reconnus par une machine de Turing ou engendrés par une grammaire, est dénombrable. Or, on a vu que $\mathscr{P}(\Sigma^*)$ n'est pas dénombrable, pour un alphabet Σ. Cela sous-entend qu'il existe encore une infinité de langages pour lesquels on ne dispose d'aucun moyen de reconnaissance (non récursivement énumérables).

On verra plus tard des exemples de tels langages et aussi de langages récursivement énumérables et récursifs. Pour ce qui est des langages récursifs dits « contextuels » (engendrés par une grammaire de type 1), le langage $L = \{a^{2^n} / n > 0\}$ en est un exemple. On peut démontrer en effet qu'il est possible de transformer la grammaire l'engendrant (donnée en cours) pour obtenir une grammaire sensible au contexte, telle que définie dans la hiérarchie de Chomsky.

Pour clore cette section, on va voir des exemples de langages rationnels (engendrés par une grammaire linéaire droite, de type 3).

Exemple. Soit L un langage fini sur un alphabet Σ. Donner une grammaire linéaire droite pour L, sachant que L est rationnel.

On a : $L = \{w_1, w_2, \ldots, w_n\}$. Une grammaire $G = (\{S\}, \Sigma, P, S)$ l'engendrant possède n productions : $S \to w_i$, $1 \leqslant i \leqslant n$. Il est possible de trouver plus simple, en exploitant les propriétés de L.

Exemple. Donner une grammaire engendrant $L = a^*ba^*$ sur $\Sigma = \{a, b\}$.

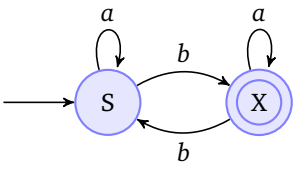
Une solution serait $G = (\{S, X\}, \Sigma, P, S)$ avec

$$P = \{S \to aS|bX, X \to aX|\varepsilon$$

Exemple. $L = \Sigma^*b$, $\Sigma = \{a, b\}$. Une solution simple : $G = (\{S\}, \Sigma, P, S)$ avec les productions

$$S \to aS|bS|b$$

Exemple. $L = \{w \in \Sigma^* / |w|_b \text{ impair }\}$, $\Sigma = \{a, b\}$. Si la grammaire linéaire droite n'est pas évidente à trouver, une grammaire peut toujours être tirée d'un automate reconnaissante L. Or, on sait que l'automate minimal de L est le suivant :



D'où la grammaire $G = (\{S, X\}, \Sigma, P, S)$ avec :

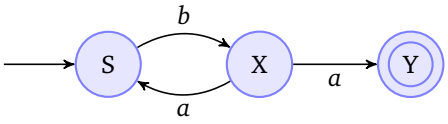
$$S \to aS|bX$$
$$X \to aX|bS|\varepsilon$$

Exemple. À contrario, il est possible d'exhiber un automate reconnaissant un langage rationnel L à partir d'une grammaire donnait L. Considérons pour exemple la grammaire suivante : $G = (\{S\}, \Sigma, P, S)$, avec $\Sigma = \{a, b\}$ et les productions $S \to baS|ba$. Donner un automate reconnaissant L(G).

Pour être exploitable dans ce sens, la grammaire G va devoir être transformée, quitte à lui rajouter des variables. On va considérer la grammaire équivalente $G' = (\{S, X\}, \Sigma, P', S)$ avec les éléments de P' suivants :

$$S \to bX$$
$$X \to aS|aY$$
$$Y \to \varepsilon$$

D'où l'automate :



On voit ici que $L(G') = (ba)^+$.

1.5 Langages algébriques et automates à pile

Rappelons que, par définition, un langage est algébrique s'il est engendré par une grammaire (de type 2) dont les productions sont toutes de la forme : $X \to \alpha$, avec $X \in V$ et $\alpha \in (V \cup A)^*$, V étant l'ensemble des variables de la grammaire et A celui des symboles terminaux. Toutefois, il est possible de normaliser ce genre de grammaire, et ce de deux manières différentes.

Forme normale de Chomsky (CNF). Tout langage algébrique propre (ne contenant pas ε) peut être engendré par une grammaire (de type 2) dont toutes les productions sont de la forme : $X \to YZ$ ou $X \to a$, avec $X, Y, Z \in V$ et $a \in A$ (alphabet terminal).

Forme normale de Greibach (GNF). Tout langage algébrique propre peut être engendré par une grammaire (de type 2) dont toutes les productions ont pour forme : $X \to a\alpha$, avec $X \in V$, $a \in A$ et αV^*.

Exemple. $L = \{a^n b^n / n > 0\}$, sur $A = \{a, b\}$. On sait que L peut être engendré par la grammaire $G = (\{S\}, A, P, S)$ avec pour seules productions : $S \to aSb|ab$.

Sa CNF s'obtient en posant $X = aS$, $Y = b$ et $Z = a$, d'où les productions :

$$\begin{aligned} S &\to XY|ZY \\ X &\to ZS \\ Y &\to b \\ Z &\to a \end{aligned}$$

Sa GNF, plus simple :

$$\begin{aligned} S &\to ASX|aX \\ X &\to b \end{aligned}$$

Exemple. $L = \{wx\tilde{w} / w \in A^*, x \in A\}$, sur $A = \{a, b\}$, où $\tilde{w}$ est l'image miroir de w. L est donc le langage des palindromes de longueur impaire sur A.

Une grammaire $G = (\{S\}, A, P, S)$ pour L aura les productions :

$$S \to a|b|aSa|bSb$$

Sa GNF aura pour productions :

$$\begin{aligned} S &\to a|b|aSX|bSY \\ X &\to a \\ Y &\to b \end{aligned}$$

Les productions de sa CNF :

$$\begin{aligned} S &\to a|b|XY|ZT \\ X &\to YS \\ Y &\to a \\ Z &\to TS \\ T &\to b \end{aligned}$$

Définition 1.5. Un automate à pile est un septuplet $\mathscr{A} = (Q, A, \Gamma, \delta, q_0, Z_0, F)$, où :
- Q est un ensemble fini d'états ;
- A est un alphabet, dit d'entrée ;
- Γ est l'alphabet de pile ;
- $q_0 \in Q$ est l'état initial ;
- $Z_0 \in \Gamma$ est le symbole initial de la pile ;
- $F \subset Q$ est l'ensemble des états finaux ;
- δ est une application de $Q \times (A \cup \{\varepsilon\}) \times \Gamma \to \mathscr{P}_f(Q \times \Gamma^*)$, où $\mathscr{P}_f$ est l'ensemble des parties finies.

On écrit $(q, aw, Z\alpha) \vdash (p, w, \beta\alpha)$ si $(p, \beta) \in \delta(q, a, Z)$, où $p, q \in Q$, $a \in A \cup \{\varepsilon\}$, $w \in A^*$, $\alpha, \beta \in \Gamma^*$, et $Z \in \Gamma$.

On note $\vdash^*$ une suite finie (éventuellement vide) d'applications de règles de la forme $\vdash$. La relation $\vdash^*$ représente la fermeture réflexo-transitive de la relation $\vdash$.

On dispose de deux modes d'acceptation différents, s'agissant d'un automate à pile.

Acceptation d'un langage par état final.

$$L(\mathscr{A}) = \{w \in A^* / \exists p \in F, \exists \gamma \in \Gamma^*, (q_0, w, Z_0) \vdash^* (p, \varepsilon, \gamma)\}$$

Acceptation d'un langage par pile vide.

$$N(\mathscr{A}) = \{w \in A^* / \exists p \in Q, (q_0, w, Z_0) \vdash^* (p, \varepsilon, \varepsilon)\}$$

On a le théorème suivant.

Théorème 1.1. *Soit* $L \subset A^*$ *un langage. Les assertions suivantes sont équivalentes :*

1. L *est algébrique.*
2. *Il existe un automate à pile* $\mathscr{A}$ *tel que* $L = N(\mathscr{A})$.
3. *Il existe un automate à pile* $\mathscr{A}$ *tel que* $L = L(\mathscr{A})$.

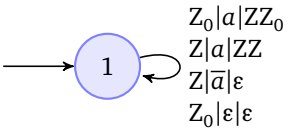
Définition 1.6. Un automate à pile $\mathscr{A}$ est dit déterministe si les deux conditions suivantes sont réunies :

1. $\forall q \in Q, \forall Z \in \Gamma, \delta(q, \varepsilon, Z) \neq \emptyset \Rightarrow \delta(q, a, Z) = \emptyset, \forall a \in A$. Autrement dit, une ε-transition n'est possible que si aucune autre transition pour une lettre de A n'est possible pour un même état et symbole de pile.
2. $\forall q \in Q, \forall Z \in \Gamma, \forall a \in A \cup \{\varepsilon\}, \left|\delta(q, a, Z)\right| \leq 1$. Autrement dit, pour un état, un symbole de pile et une lettre donnés, au plus une seule transition est possible.

Remarque. Contrairement aux automates des langages rationnels, il existe des langages algébriques qu'il est impossible de reconnaître en utilisant un automate à pile déterministe.

Deux familles, qui étaient confondues pour les langages rationnels, deviennent disjointes pour les langages algébriques.

Exemple. Donner un automate à pile acceptant le langage des « bons parenthèsages » sur l'alphabet $A = \{a, \overline{a}\}$.



Comme un automate classique, un automate à pile peut être représenté graphiquement. On a ici :

$$Q = \{1\}, \Gamma = \{Z_0, Z\}, q_0 = 1, F = \emptyset$$

et

$$\begin{aligned}\delta(1, \varepsilon, Z_0) &= (1, \varepsilon)\\ \delta(1, a, Z_0) &= (1, ZZ_0)\\ \delta(1, a, Z) &= (1, ZZ)\\ \delta(1, \overline{a}, Z) &= (1, \varepsilon)\end{aligned}$$

La suite d'applications de règles pour le mot $a\overline{a}a\overline{a}$ sera :

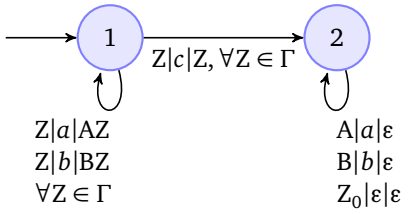
$$\begin{aligned}(1, a\overline{a}a\overline{a}, Z_0) &\vdash (1, \overline{a}a\overline{a}, ZZ_0)\\ &\vdash (1, a\overline{a}, Z_0)\\ &\vdash (1, \overline{a}, ZZ_0)\\ &\vdash (1, \varepsilon, Z_0)\\ &\vdash (1, \varepsilon, \varepsilon)\end{aligned}$$

Donc $(1, a\overline{a}a\overline{a}, Z_0) \vdash^* (1, \varepsilon, \varepsilon)$, c'est à dire que $a\overline{a}a\overline{a} \in N(\mathscr{A})$.
Par contre, $a\overline{a}\overline{a} \notin N(\mathscr{A})$; ce n'est pas un bon parenthèsage :

$$\begin{aligned}(1, a\overline{a}\overline{a}, Z_0) &\vdash (1, \overline{a}\overline{a}, ZZ_0)\\ &\vdash (1, \overline{a}, Z_0)\end{aligned}$$

Aucune transition n'est possible, et pourtant il reste encore une lettre, et la pile n'est pas vide. Donc le mot n'est pas accepté.

Exemple. Donner un automate à pile acceptant le langage $L = \{wc\tilde{w}/w \in \{a, b\}^*\}$ sur $A = \{a, b, c\}$.



Remarque. Dans ces deux exemples, on a utilisé uniquement le mode de reconnaissance par pile vide, qui paraît plus naturel dans le cadre des automates à pile. L'autre mode (par état final) est moins fréquemment employé.

Chapitre 2

Machines à registres

Nous allons détailler dans ce chapitre un premier volet relatif à la notion de calculabilité que constituent les machines à registres, également appelées machines RAM.

Il s'agit de machines abstraites qui ressemblent à des processeurs. Les seules données pouvent être traitées par ces machines sont les entiers naturels, que l'on peut supposer arbitrairement grands. Les entrées et les sorties s'effectuent à travers respectivement une bande d'entrée et une bande de sortie, munie chacune d'une tête de lecture (resp. d'écriture).

2.1 Description d'une machine à registres

Plus précisement, une machine à registres est formée :
- d'un programme consistant en une suite finie d'instructions, numérotées à partir de 0 ;
- d'une suite infinie de registres $(R_n)_{n\geqslant 0}$ numérotés également en partant de 0 ;
- d'un registre spécial appelé accumulateur, et noté A ;
- d'une bande d'entrée sur laquelle sont lues les données ;
- d'une bande de sortie rassemblant les résultats.

2.1.1 Registres

L'accumulateur A ainsi que les registres $(R_n)_{n\geqslant 0}$ servent à contenir les données manipulées par la machine tout au long du calcul. Ils sont tous initialisés à la valeur 0 au lancement du programme et peuvent stocker n'importe quelle valeur entière positive.

2.1.2 Instructions

Les instructions autorisées pour une machine RAM se divisent en quatre catégories :
- Manipulation de registres ;
- Opérations arithmétiques ;
- Ruptures de séquence ;

– Entrées / sorties.

Manipulation de registres

On dispose de deux instructions à cet effet : `load` et `store`.

L'instruction `load` permet de charger une valeur dans l'accumulateur à partir d'un registre. Elle admet trois déclinaisons :

1. `load #n`, où n est un entier naturel. Il s'agit d'un adressage absolu. La valeur de l'entier n est chargée dans l'accumulateur.
2. `load n`, où $n \in \mathbb{N}$. C'est l'adressage direct. Le contenu du registre R_n est mis dans l'accumulateur.
3. `load (n)`, où $n \in \mathbb{N}$. C'est l'adressage indirect. Le contenu du R_p, où p est la valeur de R_n, est placé dans l'accumulateur. Le registre R_n joue, dans cette variante, un rôle de pointeur.

Réciproquement, l'instruction `store` permet de stocker dans un registre une valeur issue de l'accumulateur. Elle possède deux variantes :

1. `store n`, où $n \in \mathbb{N}$. C'est un adressage direct. Le contenu de l'accumulateur est placé dans R_n.
2. `store (n)`, $n \in \mathbb{N}$. Il s'agit d'un adressage indirect. Le contenu de A est mis dans R_p, p étant la valeur de stockée dans R_n.

Opérations arithmétiques

Les machines RAM sont dotées de deux opérations arithmétiques uniquement : `incr` et `decr`. Ces deux instructions ne prennent pas de paramètres. L'opération `incr` augmente d'une unité le contenu de l'accumulateur, et l'opération `decr` réalise l'opération inverse lorsque le contenu de A n'est pas nul.

On va se restreindre à ces deux seules opérations. Il est néanmoins possible d'en ajouter d'autres, comme l'addition et la soustraction par exemple. Mais l'on démontre que cela ne change en rien la puissance de calcul de ces machines.

Ruptures de séquence

Il existe pour ces machines trois instructions de cette sorte : `stop`, `jump`, `jz` (« jump zéro »).

L'instruction `stop` ne prend pas de paramètres. Elle arrête le programme.

Les deux instructions `jump` et `jz` nécessitent comme paramètre un numéro d'instruction (existant). L'instruction `jump` est appelée saut inconditionnel : l'exécution se poursuit au numéro d'instruction mentionné. L'instruction `jz` est un saut conditionnel : l'exécution du programme saute à l'instruction mentionnée seulement si le contenu de A est nul, et se poursuit en séquence sinon.

Instructions d'entrées et de sorties

Elles sont au nombre de deux et sans paramètres : `read` et `write`.

L'instruction `read` place l'entier couvrant de la bande de lecture dans A et provoque le déplacement d'un cran de la tête de lecture (vers la droite).

L'instruction `write` provoque quant à elle, l'impression de la valeur de A sur la bande de sortie.

Remarque. On a fait mention au début de la ressemblance des machines RAM aux processeurs. Des différences importantes cependant méritent d'être relevées :

- Le nombre de registres d'une machine RAM est illimité, ce qui n'est pas le cas d'un processeur, qui dispose d'une mémoire limitée en pratique.
- Chaque registre d'une machine RAM peut contenir un entier positif de taille quelconque, alors qu'un microprocesseur est soumis à la manipulation de mots mémoire d'une capacité limitée.
- Le jeu d'instructions ouvertes à une machine à registres est bien plus réduit que celui attribué classiquement à un processeur.

Exemple. Le programme suivant utilise une indirection, c'est à dire un adressage inderect, pour remplir chaque registre R_n, $n > 0$, avec la valeur n. Bien sûr, il ne s'arrête jamais.

```
1  incr
2  store 0
3  store (0)
4  jump 0
```

Exemple. Écrire le programme d'une machine RAM, qui réalise la somme de deux entiers, en supposant que ces deux nombres sont placés sur la bande d'entrée.

```
1  read
2  store 0
3  read
4  store 1
5  load 0
6  jz 12
7  decr
8  store 0
9  load 1
10 incr
11 store 1
12 jump 4
13 load 1
14 write
```

On démontre que les machines à registres sont équivalentes, du point de vue puissance de calcul, aux machines de Turing, c'est à dire que toute fonction calculable au moyen d'une machine RAM l'est par une machine de Turing, et vice-versa.

Chapitre 3

Fonctions primitives récursives et fonctions récursives

On va aborder dans ce chapitre un second aspect de la notion de calculabilité, incarné par les fonctions calculables, dites aussi récursives.

3.1 Fonctions primitives récursives

On considère ici les fonctions de $\mathbb{N}^k$ dans $\mathbb{N}$, $k \geqslant 0$. On convient de l'existence de fonctions d'arité 0, que l'on peut assimiler à une constante. On note généralement $\overline{n}$ le k-uplet $(n_1, \ldots, n_k)$.

On définit d'abord ce qu'on appelle les fonctions primitives récursives de base :

1. La fonction d'arité 0 : $0() : \mathbb{N}^0 \to \mathbb{N}, \overline{n} \mapsto 0$.
2. Les projections $\pi_i^k : \mathbb{N}^k \to \mathbb{N}, \overline{n} \mapsto n_i$, $k > 0$.
3. La fonction successeur $\sigma : \mathbb{N} \to \mathbb{N}, n \mapsto n + 1$.

On définit d'autre part des schémas de construction d'autres fonctions à partir de ces fonctions de base.

Définition 3.1 (Schéma de composition)**.** Soient $g : \mathbb{N}^l \to \mathbb{N}$ une fonction à l arguments, et $h_1, \ldots, h_l$ l fonctions de $\mathbb{N}^k \to \mathbb{N}$, à k arguments. On considère la fonction $f = g(h_1, \ldots, h_l)$ définie par :

$$f(\overline{n}) = g(h_1(\overline{n}), \ldots, h_l(\overline{n}))$$

Il s'agit d'une fonction de $\mathbb{N}^k \to \mathbb{N}$, appelée composée de la fonction g et des fonctions h_i, $1 \leqslant i \leqslant l$.

Définition 3.2 (Schéma de récursion primitive)**.** Soient g une fonction à k arguments, et h une fonction à $k + 2$ arguments. On définit une fonction $f : \mathbb{N}^{k+1} \to \mathbb{N}$ par :

$$f(\overline{n}, 0) = g(\overline{n})$$
$$f(\overline{n}, m + 1) = h(\overline{n}, m, f(\overline{n}, m))$$

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

Définition 3.3. Les fonctions primitives récursives sont :
- les fonctions primitives récursives de base ;
- toutes les fonctions obtenues à partir des fonctions be base en appliquant une succession finie de schémas de composition et de récursion primitive.

Définition 3.4 (équivalente)**.** L'ensemble des fonctions primitives récursives est la plus petite famille des fonctions de $\mathbb{N}^k \to \mathbb{N}$, $k \geqslant 0$, contenant les fonctions de base et stable (fermée) pour les schémas de composition et de récursion primitive.

Exemple. La fonction prédécesseur est définie par :

$$pred(n) = \begin{cases} 0 & \text{si } n = 0 \\ n-1 & \text{sinon} \end{cases}$$

On peut écrire, en utilisant le schéma de récursion primitive :

$$\begin{cases} \text{pred}(0) = 0() \\ \text{pred}(m+1) = \pi_1^2(m, \text{pred}(m)) \end{cases}$$

La fonction pred est primitive récursive car elle se déduit des fonctions de base.

Exemple. Montrer que $\text{plus}(n_1, n_2) = n_1 + n_2$ est primitive récursive.

$$\begin{cases} \text{plus}(n_1, 0) = \pi_1^1(n_1) \\ \text{plus}(n_1, n_2 + 1) = h(n_1, n_2, \text{plus}(n_1, n_2)) \\ h = \sigma(\pi_3^3) \end{cases}$$

Exemple. Montrer que $\text{produit}(n_1, n_2) = n_1 n_2$ est primitive récursive.

$$\begin{cases} \text{produit}(n_1, 0) = 0_1(n_1) \\ \text{produit}(n_1, n_2 + 1) = h(n_1, n_2, \text{produit}(n_1, n_2)) \\ h = \text{plus}(\pi_3^3, \pi_1^3) \end{cases}$$

Exemple. Montrer que $j() :\to \mathbb{N}, \mapsto j$ est primitive récursive.

$$j() = \underbrace{\sigma(\sigma(\ldots\sigma}_{j \text{ fois}}(0())\ldots))$$

Exemple. Montrer que la fonction factorielle est primitive récursive.

$$\begin{cases} \text{fact}(0) = 1() \\ \text{fact}(n+1) = h(n, \text{fact}(n)) \\ h = \text{produit}(\sigma(\pi_1^2), \pi_2^2) \end{cases}$$

On verra dans la section suivante que les fonctions primitives récursives ne constituent en fait qu'une sous-classe propre (stricte) de toutes les fonctions calculables.

3.2 Fonctions récursives

Rappelons la thèse de Church-Turing, selon laquelle toute fonction calculable (par procédure effective, c'est à dire son algorithme) l'est par une machine de Turing et vice-versa.

Commençons par constater qu'il existe des fonctions pourtant calculables, qui ne sont pas des primitives récursives. Remarquons que l'ensemble des fonctions primitives récursives est dénombrable. Cet ensemble est infini. De plus, comme on le verra pour les machines de Turing, il est possible de coder injectivement toute fonction primitive récursive par un mot sur un alphabet fini fixé Σ (donc Σ^* est dénombrable).

Si l'on se restreint aux fonctions de $\mathbb{N} \to \mathbb{N}$ primitives récursives : elles constituent aussi un ensemble dénombrable (infini car il contient en particulier toutes les fonctions $j_1, j \geqslant 0$). Il existe une suite de fonctions deux à deux distinctes qui couvre l'ensemble des fonctions primitives récursives de $\mathbb{N}$ dans $\mathbb{N}$.

A	0	1	…	p	…
f_0	$f_0(0)$	$f_0(1)$	…	$f_0(p)$	…
f_1	$f_1(0)$	$f_1(1)$	…	$f_1(p)$	…
f_2	$f_2(0)$	$f_2(1)$	…	$f_2(p)$	…
⋮					

Considérons maintenant la fonction $f : \mathbb{N} \to \mathbb{N}$, donnée par : $f(n) = f_n(n)+1 = \mathrm{A}[n,n]+1$. La fonction f ainsi définie ne peut être primitive récursive, car elle ne correspond à aucune des fonctions figurant dans le tableau, censé contenir toutes les fonctions primitives récursives. Il suffit en effet d'écrire un algorithme (machine de Turing), pour calculer $f(n)$, énumérant toutes les fonctions f_i jusqu'au rang n, d'évaluer $f_n(n)$ et d'ajouter 1.

Un autre exemple explicite de fonction calculable non primitive récursive est celui de la fonction dite d'Ackermann. Il s'agit de la fonction $\mathrm{A} : \mathbb{N}^3 \to \mathbb{N}$ définie par :

$$\mathrm{A}(k,m,n) = \begin{cases} n+1 \text{ si } k = 0 \\ m \text{ si } k = 1 \text{ et } n = 0 \\ 0 \text{ si } k = 2 \text{ et } n = 0 \\ 1 \text{ si } k \neq 1,2 \text{ et } n = 0 \\ \mathrm{A}(k-1,m,\mathrm{A}(k,m,n-1)) \text{ sinon} \end{cases}$$

Proposition 3.1. *On a* $\mathrm{A}(1,m,n) = m+n$, $\mathrm{A}(2,m,n) = m \times n$, $\mathrm{A}(3,m,n) = m^n$.

Une version simplifiée de la fonction de $\mathbb{N}^2$ dans $\mathbb{N}$ peut être obtenue en prenant $m = 2$. Cependant, on prend souvent comme définition celle qui est légèrement différente, donnée par :

$$\mathrm{A}(0,n) = n+1$$
$$\mathrm{A}(k+1,0) = \mathrm{A}(k,1)$$
$$\mathrm{A}(k+1,n+1) = \mathrm{A}(k,\mathrm{A}(k+1,n))$$

Définition 3.5. On dit qu'une fonction $g : \mathbb{N} \to \mathbb{N}$ majore une fonction $f : \mathbb{N}^k \to \mathbb{N}$ si $f(\overline{n}) \leqslant g(\max(n1, \ldots, n_k))$.

On démontre le résultat suivant, qui exprime le fait que toute fonction primitive récursive est dominée par celle d'Ackermann.

Proposition 3.2. *Pour toute fonction primitive récursive $f : \mathbb{N}^k \to \mathbb{N}$, il existe un entier t tel que f soit majorée par la fonction $g(n) = \mathrm{A}(t, n)$.*

De cette proposition on peux immédiatement déduire le corollaire qui suit.

Corollaire 3.1. *La fonction d'Ackermann n'est pas primitive récursive. Cette fonction est cependant calculable.*

Pour caractériser les fonctions récursives, on va introduire un nouvel opérateur sur les prédicats, appelé minimisation non bornée.

Soit $p : \mathbb{N}^{k+1} \to \mathbb{N}$ un prédicat. On note :

$$f(\overline{n}) = \mu_i p(\overline{n}, i) = \begin{cases} \text{le plus petit } i \text{ vérifiant } p(\overline{n}, i) = 1 \\ \text{indéfini, sinon} \end{cases}$$

Exemple. Pour $p(\overline{n}, i) = 0$, $\forall \overline{n}, i$, $f(\overline{n})$ est une fonction de domaine vide.

Pour $p(\overline{n}, i) = 1$, $\forall i, \overline{n}$, $f(\overline{n}) = 0, \forall \overline{n}$.

Définition 3.6. On dit qu'un prédicat $p : \mathbb{N}^{k+1} \to \mathbb{N}$ est sûr (*safe*) si, $\forall \overline{n} \in \mathbb{N}^k, \exists i \geqslant 0$, $p(\overline{n}, i) = 1$.

Définition 3.7. L'ensemble des fonctions récursives (μ-récursives) est le plus petit ensemble de fonctions de $\mathbb{N}^k \to \mathbb{N}$, $k \geqslant 0$, contenant les fonctions de base $[\sigma, \pi, 0()]$ et fermé par composition, récursion primitive et minimisation non bornée des prédicats sûrs.

On démontre que toute fonction récursive ainsi définie est calculable par machine de Turing, et vice-versa (machine de Turing qui s'arrête toujours, quelque soit la donnée, car ici les fonctions sont supposées totales).

Cependant, il existe des fonctions calculables, qu'on ne peut partout définir. Par exemple, la fonction $\div : \mathbb{N} \times \mathbb{N}^3 \to \mathbb{N}$, $(m, n) \to m \div n$ est calculable mais n'a pas de valeur lorsque $n = 0$. Une telle fonction porte le nom de fonction partielle.

Pour intégrer les fonctions partielles calculables, on reformule la définition précédente comme suit :

Définition 3.8. L'ensemble des fonctions calculables (partielle) est le plus petit ensemble de fonctions de $\mathbb{N}^k \to \mathbb{N}$, $k \geqslant 0$, contenant les fonctions de base, et fermé par composition, récursion primitive et minimisation non bornée.

On démontre que cet ensemble coïncide exactement avec celui des fonctions calculables par machine de Turing, pouvant éventuellement ne pas s'arrêter pour certaines données.

Chapitre 4

Machines de Turing

Nous allons étudier en détail dans ce chapitre un ultime aspect de la calculabilité, revêtu par les machines de Turing.

Les machines de Turing représentent une abstraction des ordinateurs. Elles sont constituées d'une partie dite de contrôle, elle-même formée d'un nombre fini d'états possibles et de transitions régissant les calculs, et d'une bande infinie sur laquelle peuvent être lus ou écrits des symboles par l'intermédiaire d'une tête de lecture-écriture.

La partie de contrôle peut être comparée à un processeur. La bande simule la mémoire de l'ordinateur. Elle est théoriquement infinie, ce qui se justifie par le fait que la mémoire de l'ordinateur puisse être étendue à l'infini en y ajoutant des périphériques. La tête de lecture incarne le bus qui relie la mémoire au processeur. La différence importante entre un ordinateur et une machine de Turing réside dans le mode d'accès à la mémoire. Dans un ordinateur, ce dernier est aléatoire, alors qu'il est séquentiel s'agissant d'une machine de Turing.

4.1 Définitions

Plus formellement, une machine de Turing est un septuplet $\mathrm{M} = (\mathrm{Q}, \Sigma, \Gamma, \delta, q_0, \mathrm{F}, \#)$, dans lequel :

- Q est l'ensemble fini des états de contrôle de la machine.
- Σ est l'alphabet d'entrée. Il s'agit d'un ensemble fini de symboles ne contenant pas le symbole $\#$. Cet alphabet est utilisé pour écrire la donnée initiale sur la bande.
- Γ est l'alphabet de bande. C'est un ensemble fini, qui comprend tous les symboles pouvant figurer sur la bande, y compris $\#$ et ceux de Σ.
- δ est un ensemble fini de transitions de la forme (p, a, q, b, x), où p et q sont des états, a et b des symboles de Γ, et x un élément de $\{\mathrm{L}, \mathrm{R}\}$. On note $p, a \to q, b, x$ une telle transition.
- q_0 est l'état initial, l'état de départ de tout calcul.
- $\mathrm{F} \subseteq \mathrm{Q}$ est l'ensemble des états finaux, également dits états d'acceptation.
- $\#$ est le symbole « blanc », qui remplit en particulier au départ toutes les cellules de bande autres que celles contenant la donnée initiale, supposée écrite de façon connexe, tout à gauche de la bande.

Remarque. En lieu et place de L et R, on utilise parfois, dans certaines littérateures, les flèches $\leftarrow$ et $\rightarrow$.

Une machine de Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, F, \#)$ est dite déterministe si pour tout état p et toute lettre $a \in \Gamma$, il existe au plus dans δ une transition de la forme $p, a \rightarrow q, b, x$.

Souvent, l'ensemble δ des transitions fait place à une application, encore notée $\delta : Q \times \Gamma \rightarrow \mathscr{P}(Q \times \Gamma \times \{L, R\})$. Dans ce cas, l'application δ associe à chaque couple $(p, a) \in Q \times \Gamma$ l'ensemble des triplets (q, b, x) tels que $p, a \rightarrow q, b, x$ soit une transition de la machine. Lorsque cette dernière est déterministe, on voit bien que l'application δ devient en fait littéralement une fonction de $Q \times \Gamma \rightarrow \mathbb{Q} \times \Gamma \times \{L, R\}$.

On dispose de deux modes de représentation d'une machine de Turing : par table de transition ou par graphe d'états. Dans le premier mode, on utilise un tableau possédant une ligne pour chaque symbole de Γ, et une colonne pour chaque état. La case située à l'intersection de la ligne et de la colonne repérées respectivement par a et p contient tous les triplets (q, b, x) pour lesquels $p, a \rightarrow q, b, x$ est une transition de δ. Si la machine est déterministe, il est clair que chaque cellule va contenir au plus un tel triplet.

Exemple. Soit la machine $M = (\{0\}, \{a, b\}, \{a, b, \#\}, \delta, 0, \{0\}, \#)$. L'ensemble δ contient les transitions :

$$0, a \rightarrow 0, \#, R$$
$$0, b \rightarrow 0, \#, R$$
$$0, \# \rightarrow 0, \#, L$$

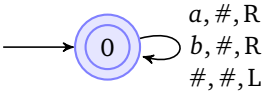
La représentation par table de M tient dans le tableau suivant :

	0
a	$0, \#, R$
b	$0, \#, R$
$\#$	$0, \#, L$

Constatons au passage que M est une machine déterministe.

Il est commode et plus fréquent encore de représenter une machine à l'aide d'un graphe orienté dont les sommets sont les états de Q. Ce graphe possède une arête joignant l'état p à q étiquetée a, b, x si $p, a \rightarrow q, b, x$ est une transition. L'état initial est indiqué par une flèche entrante, et les états finaux par une flèche sortante, parfois un double cercle.

Par exemple, s'agissant de la machine précédente, on obtient le graphe orienté valué :



4.2 Configurations, étapes et calculs

Le principe de fonctionnement d'une machine de Turing est le suivant. Une entrée, c'est à dire un mot fini sur l'alphabet Σ, se trouve au départ sur la bande,

tout à gauche et en un seul morceau. Le reste de la bande est rempli avec le symbole blanc. La tête de lecture est positionnée sur le premier symbole de la donnée et l'état de contrôle est l'état initial. La machine commence ensuite son calcul, étape par étape, jusqu'à l'obtention d'un état bloquant ou éventuellement acceptant.

Pour décrire précisément ce fonctionnement, on a besoin d'introduire la notion de configuration. Il s'agit de la situation de la machine à un instant donné du calcul. Une configuration comprend :

– l'état de contrôle courant, élément de Q,
– le contenu de la bande,
– la position de la tête de lecture.

Si la machine se trouve dans un état q, la configuration courante s'écrit uqv, où u est le contenu de la bande précédant strictement la tête de lecture et v celui situé à droite de cette tête. Le symbole juste sous la tête constitue donc le premier symbole de v. Les symboles blancs qui se trouvent dans la partie infinie à droite sont ignorés dans l'écriture d'une configuration.

Exemple.

La configuration courante de cette machine s'écrit :

$$ab\mathrm{A}aa\mathrm{BA}bqabbb\mathrm{BA}$$

Les symboles # figurant à l'infini à droite sont omis.

Au départ de tout calcul, on a la configuration dite initiale q_0w, w étant la donnée d'entrée : $w \in \Sigma^*$.

Un calcul d'une machine de Turing va se décomposer en étapes. Une étape consiste à passer d'une configuration à la suivante en appliquant une transition de δ. Elle effectue alors les actions suivantes :

– changer l'état de contrôle,
– remplacer le symbole sous la tête,
– déplacer la tête d'une position à gauche ou à droite.

On note $\mathrm{C} \Rightarrow \mathrm{C}'$ une étape de calcul passant de la configuration C à C′.

Remarque. Une transition de la forme $p, a \to q, b, \mathrm{L}$ ne peut être appliquée que lorsque la tête ne se trouve pas sur le premier symbole de la bande.

Les étapes possibles de calcul sont les suivantes, u et v étant des mots de Γ^* et a, b, c des lettres :

$$ucpav \Rightarrow uqcbv, \text{ si } p, a \to q, b, \mathrm{L}$$
$$upav \Rightarrow ubqv, \text{ si } p, a \to q, b, \mathrm{R}$$

Du fait de la présence implicite de symboles # en fin de bande, on en vient aussi à considérer les étapes :

$$ucp \Rightarrow uqcb, \text{ si } p, \# \to q, b, \mathrm{L}$$
$$up \Rightarrow ubq, \text{ si } p, \# \to q, b, \mathrm{R}$$

En définitive, un calcul d'une machine de Turing est une suite finie $\mathrm{C}_0, \ldots, \mathrm{C}_n$ de configurations telle que $\mathrm{C}_0 \Rightarrow \cdots \Rightarrow \mathrm{C}_n$, C_0 étant la configuration initiale. On

note en abrégé $C_0 \Rightarrow^* C_n$, ou encore $C_0 \Rightarrow_M^* C_n$ lorsque la machine impliquée dans le calcul doit être précisée. Il est à noter, par rapprochement avec les grammaires, que la relation $\Rightarrow^*$ représente en fait la fermeture (clôture) réflexive et transitive de la relation $\Rightarrow$.

Exemple. Considérons la machine $M = (\{0\}, \{a, b\}, \{a, b, \#\}, \delta, 0, \{0\}, \#)$ décrite dans l'avant-dernier exemple.

Supposons que l'on fournisse à M la donnée initiale $w = aba$. Sa configuration de départ est alors $0aba$. Recensons les configurations suivantes.

C_0	$0aba$
C_1	$\#0ba$
C_2	$\#\#0a$
C_3	$\#\#\#0$
C_4	$\#\#0$
C_5	$\#0$
C_6	0

On se trouve en C_6 dans une configuration bloquante ; on ne peut plus appliquer aucune transition (cf. la remarque précédente). On obtient alors le calcul suivant :

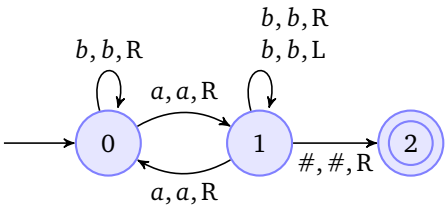
$$\begin{aligned} 0aba &\Rightarrow \#0ba \\ &\Rightarrow \#\#0a \\ &\Rightarrow \#\#\#0 \\ &\Rightarrow \#\#0 \\ &\Rightarrow \#0 \\ &\Rightarrow 0 \end{aligned}$$

Définition 4.1. Une configuration C est dite bloquante s'il n'existe pas de configuration C' telle que $C \Rightarrow C'$.

Une configuration uqv est dite acceptante si l'état q est acceptant, c'est à dire final.

Un calcul $C_0 \Rightarrow^* C_n$ est dit acceptant s'il atteint une configuration acceptante.

Exemple. Soit la machine $M = (\{0, 1, 2\}, \{a, b\}, \{a, b, \#\}, \delta, 0, \{2\}, \#)$ donnée par le graphe :



Le mot d'entrée $w = ab^2a$ conduit aux différents calculs suivants (M n'est pas déterministe) :

$$0abba \Rightarrow a1bba \Rightarrow ab1ba \Rightarrow abb1a \Rightarrow abba0$$

La dernière configuration est bloquante, et ce calcul n'est pas acceptant.

$$0abba \Rightarrow a1bba \Rightarrow 1abba \Rightarrow a0bba \Rightarrow ab0ba$$
$$\Rightarrow abb0a \Rightarrow abba1 \Rightarrow abba\#2$$

Configuration finale bloquante et calcul acceptant.

$$0abba \Rightarrow a1bba \Rightarrow ab1ba \Rightarrow a1bba \Rightarrow ab1ba$$
$$\Rightarrow \ldots$$

Un tel calcul ne se termine jamais et n'est pas acceptant.

En fin de compte, le mot ab^2a est accepté par la machine puisqu'au moins un des calculs le concernant est acceptant. Par contre, il est facile de constater que a^2 n'est pas un mot accepté, car le seul calcul possible l'impliquant est $0aa \Rightarrow a1a \Rightarrow aa0$, qui n'est pas acceptant.

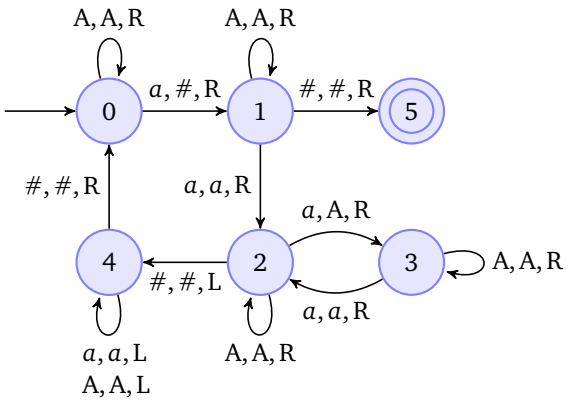
Définition 4.2. Soit $M = (Q, \Sigma, \Gamma, \delta, q_0, F, \#)$ une machine de Turing. On a les définitions suivantes :

- Un mot $w \in \Sigma^*$ est accepté par M si et seulement si $\exists u, v\Gamma^*, q \in F, q_0 w \Rightarrow^* uqv$.
- On appelle reconnu (ou accepté) par M, le langage des mots Σ^* acceptés par M :
$$L(M) = \left[w \in \Sigma^* / \exists u, v \in \Gamma^*, q \in F, q_0 \Rightarrow^* uqv\right]$$
- On dira que deux machines de Turing sont équivalentes si et seulement si elles reconnaissent le même langage.

Proposition 4.1. *Toute machine de Turing est équivalente à une machine de Turing qui est déterministe.*

Exemple. Machine de Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, F, \#)$ reconnaissant le langage $L = \{a^{2^n} / n \geqslant 0\}$ sur l'alphabet $\Sigma = \{a\}$:

- $Q = \{0, 1, 2, 3, 4, 5\}$
- $\Gamma = \{a, A, \#\}$
- $q_0 = 0$
- $F = \{5\}$



On dispose essentiellement de deux modes d'utilisation d'une machine de Turing, parfois d'une combinaison des deux : accepteur ou calculateur. Le mode accepteur vient d'être illustré. En mode calculateur (pur), on considère un ensemble d'états finaux vide. Pour un mot d'entrée, on effectue le (ou les) calcul jusqu'à atteindre éventuellement une configuration bloquante. Si la machine tourne indéfiniment, le résultat est indéfini. Par contre, s'il y a blocage, le résultat est le contenu de la bande à cet instant.

Pour effectuer des calculs sur des entiers naturels, le codage que l'on utilise généralement pour la donnée initiale est l'écriture bâtons : on code l'entier n par le mot 0^n, sur l'alphabet $\Sigma = \{0\}$. Lorsque la donnée appartient à $\mathbb{N}^k$, $k \geqslant 1$, on considère l'alphabet d'entrée $\Sigma = \{0, 1\}$ et on code le k-uplet $(n_1, \ldots, n_k)$ par le mot $0^{n_1}10^{n_2}1 \ldots 10^{n_k}$. Le symbole 1 sert ici de séparateur.

4.3 Codage d'une machine de Turing

On a parfois besoin, notamment dans certaines preuves, de faire travailler une machine de Turing sur une autre machine de Turing, qui lui est fournie en donnée. Il faut pouvoir représenter (coder) une machine de Turing par un mot sur un certain alphabet A. L'alphabet le plus souvent utilisé est $\{0, 1, (,), ,\}$ (le dernier symbole est la virgule).

L'idée maîtresse est de tout coder en binaire, en utilisant la virgule comme séparateur et les parenthèses comme délimiteurs de bloc. Chaque état de la machine et chaque symbole de son alphabet de bande se trouve ainsi codé en binaire (sur le sous-alphabet $\{0, 1\}$, dans l'ordre décroissant des puissances de 2).

Soit $\mathrm{M} = (\mathrm{Q}, \Sigma, \Gamma, \delta, q_0, \mathrm{F}, \#)$ une machine de Turing disposant de n états, numérotés par convention de 0 à $n-1$. Soit m le nombre de symboles de Γ, eux aussi numérotés de 0 à $m-1$. On suppose que le symbole blanc possède le numéro 0, et que les symboles de Σ ont les numéros 1 à k, avec $k = |\Sigma|$. Les écritures binaires des entiers n, m, et k seront respectivement notées $< n >$, $< m >$, et $< k >$. Pour un état q, le codage binaire du numéro de q sera noté $< q >$.

Le codage de la machine M commence par le symbole '(' et se termine par le symbole ')'. Entre ces deux symboles se distinguent six parties séparées par des virgules. Les trois premières parties sont exactement $< n >$, $< k >$ et $< m >$. La quatrième partie code les transitions de δ, la cinquième est $< q_0 >$, et la sixième code l'ensemble F des états finaux.

Le codage de δ commence par le symbole '(' et finit par le symbole ')'. Il est constitué de la suite des codages de toutes les transitions, séparés par des virgules. Chaque transition $p, a \to q, b, x$ sera codée par la suite de symboles $(< p >, < a >, < q >, < b >, < x >)$, où $< x >= 0$ si $x = \mathrm{L}$, et 1 sinon. Quant au codage de F, il est constitué des codages $< q >$ pour tout $q \in \mathrm{F}$, séparés par des virgules, et délimité par des parenthèses.

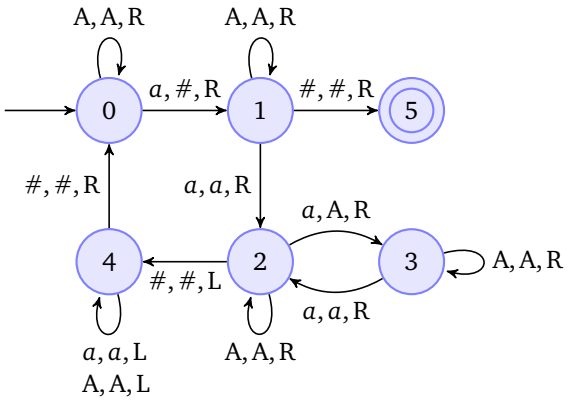
Le codage d'une machine M va donc revêtir la forme générale suivante :

$$\big(< n >, < k >, < m >, \big((< p_1 >, < a_1 >, < q_1 >, < b_1 >, < x_1 >), \ldots\big), < q_0 >, \big(< f_i >, \ldots\big)\big)$$

Avec ces conventions, toute machine de Turing possède un codage (et même plusieurs) sur l'alphabet A. Cependant, tout mot sur A^* ne représente pas nécessairement un codage de machine. En particulier, un tel codage doit respecter certaines règles syntaxiques, comme commencer par une parenthèse ouvrante et

s'achever par une parenthèse fermante. Il doit aussi vérifier des règles sémantiques. Par exemple, les numéros des états qui apparaissent dans le codage de δ doivent être inférieurs au nombre d'états de la machine ; de même pour les numéros des symboles figurant dans le même codage, qui doivent tous être inférieurs à m. Il n'est d'ailleurs pas très difficile de construire une machine de Turing qui prend en entrée un mot de A et teste s'il s'agit d'un codage valide ou non.

Exemple. Donner le codage de la machine de l'exemple précédent reconnaissant $L = \{a^{2^n} | n \geqslant 0\}$:



On a ici $n = 6$, $k = 1$ et $m = 3$. Posons, dans l'ordre, $\Gamma = \{\#, a, A\}$. Un codage de cette machine sera donc :

$$(110, 1, 11, ((0, 10, 0, 10, 1), (0, 1, 1, 0, 1), (1, 10, 1, 10, 1), (1, 0, 101, 0, 1),$$
$$(1, 1, 10, 1, 1), (10, 1, 11, 10, 1), (11, 10, 11, 10, 1),$$
$$(11, 1, 10, 1, 1), (10, 10, 10, 10, 1), (10, 0, 100, 0, 0),$$
$$(100, 1, 100, 1, 0), (100, 10, 100, 10, 0), (100, 0, 0, 0, 1)), 0, (101))$$

4.4 Variantes des machines de Turing

Il existe plusieurs variantes des machines de Turing, presqu'autant que de livres traitant du sujet. Parmi ces variantes, certaines sont mineures et d'autres sont plus classiques, telles que les machines à bande biinfinie, ou à plusieurs bandes. Cependant, toutes les variantes considérées par les différents auteurs sont équivalentes, en termes de puissance de calcul. Déjà en tant que variante du modèle standard, dans lequel la tête ne dispose que de deux déplacements possibles, il est possible de considérer une troisième alternative : demeurer dans la même position (symbole S, pour *stay*). Cette petite extension ne confère rien de spécial au modèle standard. Par contre, on va voir qu'elle simplifie sensiblement la description des machines à bandes multiples.

4.4.1 Machine à bande biinfinie

Sa définition reste identique à celle du modèle standard, excepté le fait que la bande est également considérée comme étant infinie à gauche. Cela signifie de

façon plus formelle que la bande constitue une suite de positions indexées par tous les entiers. Au début du calcul, la donnée initiale occupe les positions 0 à k, $k+1$ étant sa longueur ; tout le reste de la bande est alors rempli de symboles #. On suppose en outre que la tête pointe sur la position 0.

4.4.2 Machine à plusieurs bandes

Au lieu de disposer d'une seule bande infinie à droite, comme le requiert le modèle standard, ces machines peuvent utiliser plusieurs bandes indépendantes, chacune pourvue d'une tête de lecture propre. Plus formellement, une machine de Turing à k bandes ($k > 0$) est une septuplet $M = (Q, \Sigma, \Gamma, \delta, q_0, F, \#)$, avec en particulier $\Sigma = \Pi_{i=1}^{k} \Sigma_i$ et $\Gamma = \Pi_{i=1}^{k} \Gamma_i$, avec $\Sigma_i \subset \Gamma_i$, $\forall i$.

Les transitions de δ doivent revêtir la forme $p, a_1, \ldots, a_k \rightarrow q, b_1, \ldots, b_k, x_1, \ldots, x_k$, où $p, q \in Q$, $a_i, b_i \in \Gamma$, et $x_i \in \{L, R, S\}$, $\forall i$.

Remarque. Les machines à bandes multiples interviennent beaucoup dans les preuves de certains énoncés ; on en verra d'ailleurs un exemple d'utilisation.

4.5 Langages récursivement énumérables

On a déjà défini un langage récursivement énumérable comme étant un langage engendré par une grammaire (de type 0). Voici une définition équivalente :

Définition 4.3. Un langage est récursivement énumérable s'il est reconnu par une machine de Turing.

On est d'emblée sûr qu'il existe des langages (même une infinité) non récursivement énumérables. La raison en est que les machines de Turing sont dénombrables et que les langages ne le sont pas. On verra aussi plus loin un exemple d'un tel langage.

On appelle énumérateur une machine de Turing utilisée d'une manière particulière. Cette machine est lancée avec une entrée vide et seules ses sorties nous intéressent. On suppose que la tête ne revient jamais en arrière (pas d'effacement). Les symboles autorisées pour l'écriture sont ceux de Σ, ou #, ce dernier faisant figure de séparateur. La sortie se décompose sous la forme $w_0 \# \ldots \# w_p$, avec $w_i \in \Sigma^*$, $0 \leqslant i \leqslant p$. On dira alors que les mots $w_0, \ldots, w_p$ sont énumérés par la machine. Il faut cependant noter qu'ils ne sont pas tous nécessairement différents (un même mot peut très bien être obtenu plusieurs fois).

Définition 4.4. Un langage est dit énuméré par une machine de Turing (vue comme un énumérateur) s'il est égal à l'ensemble de mots énumérés sur la bande de sortie de cette machine.

Théorème 4.1. *Un langage est récursivement énumérable si et seulement s'il est énuméré par une machine de Turing (ce qui justifie la terminologie adoptée).*

Un autre résultat important concernant ces langages est précisé par l'énoncé suivant.

Proposition 4.2. *L'union et l'intersection de deux langages récursivement énumérables le sont également.*

Démonstration. Supposons deux langages L_1 et L_2 reconnus respectivement par des machines de Turing du modèle standard $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_{0_1}, F_1, \#)$ et $M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q_{0_2}, F_2, \#)$. On va construire deux machines de Turing à deux bande, $M_\cup$ et $M_\cap$, acceptant respectivement $L_1 \cup L_2$ et $L_1 \cap L_2$. L'idée de ces machines consiste à simuler simultanément les machines M_1 et M_2, en effectuant alternativement une transition dans chacune d'elles. La première bande simule M_1 et la seconde M_2. La machine $M_\cup$ possède au départ la même entrée sur ses deux bandes. Ses états de contrôle sont les triplets (q_1, q_2, n), avec $q_1 \in Q_1$, $q_2 \in Q_2$ et $n \in 1, 2$. L'entier n indique le numéro de la prochaine machine à simuler. On a donc $Q_\cup = Q_1 \times Q_2 \times \{1, 2\}$ et $\Gamma_\cup = \Gamma_1 \times \Gamma_2$. Ses transitions sont données par :

$$\begin{aligned}\delta = &\{(p_1, p_2, 1), (a_1, a_2) \to (q_1, p_2, 2), (b_1, a_2), (x, S) | p_1, a_1 \to q_1, b_1, x \in \delta_1\} \\ &\cup \{(p_1, p_2, 2), (a_1, a_2) \to (p_1, q_2, 1), (a_1, b_2), (S, x) | p_2, a_2 \to q_2, b_2, x \in \delta_2\}\end{aligned}$$

Lorsque la dernière composante de l'état de $M_\cup$ est égale à 1 elle effectue une transition qui modifie p_1 en q_1 et écrit sur la première bande. Lorsque cette composante vaut 2, on modifie p_2 en q_2 et on écrit sur la seconde bande. La machine accepte son entrée dès que l'une des deux machines M_1 ou M_2 accepte. On a alors $F_\cup = F_1 \times Q_2 \times \{1, 2\} \cup Q_1 \times F_2 \times \{1, 2\}$.

La machine $M_\cap$ ressemble beaucoup à $M_\cup$. La seule différence réside dans la détermination des états finaux. En effet, $M_\cap$ doit accepter lorsque les deux machines M_1 et M_2 acceptent. Si l'une des deux accepte à un moment donné, $M_\cap$ va simuler les transitions de l'autre machine jusqu'à son acceptation éventuelle. □

4.5.1 Exemple de langage non récursivement énumérable

On commence tout d'abord par numéroter tous les mots de l'alphabet d'entrée selon l'ordre hiérarchique (par longueur puis par ordre lexicographique dans chaque tranche de longueur). On fait de même avec toutes les machines de Turing, en utilisant le codage sur l'alphabet fixé A. On obtient alors deux suites infinies de mots $w_0, w_1, \ldots$ et de machines $M_0, M_1, \ldots$, Soit maintenant le langage $L = \{w_i | i \geqslant 0, M_i \text{ n'accepte pas } w_i\}$.

Proposition 4.3. *Le langage* L *n'est pas récursivement énumérable.*

Démonstration. Supposons par l'absurde que L est récursivement énumérable. Il est alors reconnu par une machine de Turing M_k de l'énumération précédente. Considérons alors le mot W_k, portant le même numéro, et distinguons les deux cas :

- w_k est accepté par M_k. Donc $w_k \notin L$. Cela contredit le fait que M_k reconnaît L.
- w_k n'est pas accepté par M_k. Donc $w_k \in L$. On aboutit à la même contradiction.

Conclusion : L ne saurait être récursivement énumérable. □

Remarque. Les langages récursivement énumérables sont également qualifiés de semi-décidables, tout comme les problèmes qu'ils peuvent décrire.

4.6 Langages récursifs

Définition 4.5. Un langage est dit récursif s'il est reconnu par une machine de Turing (que l'on peut même prendre déterministe) qui n'a pas de calcul infini, c'est-à-dire qui s'arrête toujours quelque soit la donnée.

Les deux mots « récursif » et « décidable » sont synonymes ; mais on parle plutôt de problème décidable et de langage récursif. La machine en question peut être supposée déterministe, car on a déjà vu qu'une machine de Turing est toujours équivalente à une machine déterministe. De plus, si la machine non déterministe s'arrête toujours, on peut trouver une machine déterministe vérifiant cette même propriété.

Proposition 4.4.
- *Tout langage récursif est récursivement énumérable (évident).*
- *L'union et l'intersection de deux langages récursifs sont des langages récursifs.*
- *Le complémentaire d'un langage récursif est récursif.*
- *Si un langage et son complémentaire sont tous deux récursivement énumérables, alors ils sont tous deux récursifs.*

Démonstration. S'agissant de l'union et de l'intersection, la démonstration est analogue à celle déjà effectuée pour les langages récursivement énumérables. Il suffit de remarquer que si les machines de départ M_1 et M_2 n'ont pas de calcul infini, alors $M_\cup$ et $M_\cap$ n'en ont pas non plus.

Pour établir la fermeture par complémentation de la famille des langages récursifs, considérons un langage L de cette famille. Par définition, L est accepté par une machine M sans calcul infini, que l'on peut considérer déterministe. Soit la machine M' obtenue en échangeant dans M les états finaux avec les autres états. Si $M = (Q, \Sigma, \Gamma, \delta, q_0, F, \#)$, on aura $M' = (Q, \Sigma, \Gamma, \delta, q_0, Q \setminus F, \#)$. Puisque M et M' sont toutes deux déterministes, il y a pour toute entrée w, un unique calcul partant de la configuration initiale $q_0 w$. Le mot w est alors accepté par M ou par M'. Par définition des états finaux de M, ce calcul est acceptant dans M' si et seulement s'il ne l'est pas dans M. On en conclut donc que M' reconnaît $L^c = \Sigma^* \setminus L$.

Reste à établir la dernière propriété de la liste. Supposons pour cela L et L^c récursivement énumérables, c'est-à-dire reconnus respectivement par les machines M et M'. Construisons une machine P sans calcul infini qui accepte L. On va considérer à cet effet la machine P qui, pour une entrée w, simule simultanément M et M' sur la même donnée w, comme on l'a déjà fait pour l'union et l'intersection. Dès que l'une des deux machines M ou M' accepte, P s'arrête. Si c'est M qui a accepté ($w \in L$), P accepte, sinon ($w \notin L$) P rejette. Il est clair que P reconnaît L. De plus, P n'a pas de calcul infini. Alors L est récursif. De même, on prouve de manière symétrique que L^c est récursif. □

Remarque. À cause de cette dernière propriété, les langages récursivement énumérables sont aussi appelés semi-récursifs.

4.6.1 Exemple de langage récursivement énumérable non récursif

Pour une machine de Turing M et un mot w, on note $< \mathrm{M}, w >$ le codage de couple (M, w). On considère alors le langage suivant :

$$\mathrm{L} = \{< \mathrm{M}, w > | \mathrm{M} \text{ accepte } w\}$$

Ce langage code le fameux problème appelé « problème de l'acceptation », c'est-à-dire celui de savoir si une machine M donnée accepte ou non un mot w donné.

Proposition 4.5. *Le langage* L *est récursivement énumérable, mais non récursif.*

Avant d'en effectuer la preuve, on peut remarquer que L^c n'est pas récursivement énumérable ; car s'il l'était, L serait récursif. Cela nous donne un second exemple de langage non récursivement énumérable.

Démonstration. Supposons par l'absurde L récursif. Cela signifie que L est reconnu par une machine de Turing A qui s'arrête toujours. Considérons alors la machine P qui prend en entrée le codage $< \mathrm{M} >$ d'une machine M quelconque :

Machine P avec en entrée $< \mathrm{M} >$
- Calculer si A accepte l'entrée $< \mathrm{M}, < \mathrm{M} >>$
- Accepter si A n'accepte pas.

Examinons ce qui se produit lorsque P agit sur $< \mathrm{P} >$. On obtient que P accepte $< \mathrm{P} >$ si et seulement si A n'accepte pas $< \mathrm{P}, < \mathrm{P} >>$, ce c'est-à-dire $< \mathrm{P}, < \mathrm{P} >> \in \mathrm{L} \iff < \mathrm{P}, < \mathrm{P} >> \notin \mathrm{L}$: contradiction. Donc L ne peut être récursif.

Reste à voir que L est néanmoins récursivement énumérable. Il faudrait pour cela pouvoir construire une machine qui accepte l'entrée $< \mathrm{M}, w >$ si M accepte w, c'est-à-dire pouvant simuler n'importe quelle machine qui lui est fournie en donnée (avec sa propre entrée). Une telle machine porte le nom de machine universelle. Cette construction n'est pas très compliquée. On utilise une machine tri-bandes. La première bande sert à stocker le codage de M (la donnée) ainsi que celui de la donnée initiale de M. La deuxième bande contient l'état courant de M. La dernière permet de stocker le contenu de la bande de M durant la simulation. Un symbole spécial y est employé pour marquer la position de la tête de lecture de M. □

Chapitre 5

Complexité temporelle

La complexité sert à mesurer la performance d'un algorithme, aussi bien en temps (complexité temporelle) qu'en espace mémoire requis pour le fonctionnement de l'algorithme (complexité spatiale). Pour aborder cette partie consacrée au premier type de complexité, nous aurons besoin de quelques notions préliminaires.

5.1 Notations de Landau

Soient f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$. On dit que f est en $O(g)$ ou $f = O(g)$ s'il existe $n_0 \in \mathbb{N}$ et $c > 0$ tels que $\forall n \geqslant n_0$, $f(n) \leqslant cg(n)$. On dit aussi que f est dominée asymptotiquement par g. On a par exemple $2n = O(n^2)$, mais aussi $2n = O(n)$. La relation O ainsi définie représente une relation de préordre (réflexive et transitive) sur l'ensemble des fonctions considéré.

Autre relation de préordre $\Omega : f = \Omega(g) \iff g = O(f)$. La fonction g est alors qualifiée de minorant asymptotique de f.

On dit que f est en $\Theta(g)$, ou $f = \Theta(g)$ si $f = O(g)$ et $g = O(f)$, c'est-à-dire s'il existe $c, d \in \mathbb{R}_+^*$ et $n_0 \in \mathbb{N}$ tels que $cg(n) \leqslant f(n) \leqslant dg(n)$, $\forall n \geqslant n_0$. Nous dirons alors que f et g ont même ordre de grandeur asymptotique.

Il est facile de s'apercevoir que la relation Θ définit une équivalence sur les fonctions en question. Elle est plus précise que la notation O. Par exemple, on a toujours $2n = \Theta(n)$, mais $2n \neq \Theta(n^2)$. Cependant, la notation O est souvent abusivement utilisée dans la littérature à la place de Θ.

Proposition 5.1. *Si* Λ *désigne l'un des trois opérateurs* O, Ω *ou* Θ *:*

$$\left.\begin{array}{l} f_1 = \Lambda(g_1) \\ f_2 = \Lambda(g_2) \end{array}\right] \Rightarrow f_1 + f_2 = \Lambda(g_1 + g_2), f_1 + f_2 = \Lambda(\sup(g_1, g_2)), f_1 f_2 = \Lambda(g_1 g_2)$$

$$f = \mathrm{O}(g) \Rightarrow \frac{1}{f} = \Omega\left(\frac{1}{g}\right)$$

$$f = \Omega(g) \Rightarrow \frac{1}{f} = \mathrm{O}\left(\frac{1}{g}\right)$$

$$f = \Theta(g) \Rightarrow \frac{1}{f} = \Theta\left(\frac{1}{g}\right)$$

$$\left.\begin{array}{l} f_1 = \mathrm{O}(g_1) \\ f_2 = \Omega(g_2) \end{array}\right] \Rightarrow \frac{f_1}{f_2} = \mathrm{O}\left(\frac{g_1}{g_2}\right)$$

$$\left.\begin{array}{l} f_1 = \Omega(g_1) \\ f_2 = \mathrm{O}(g_2) \end{array}\right] \Rightarrow \frac{f_1}{f_2} = \Omega\left(\frac{g_1}{g_2}\right)$$

$$\left.\begin{array}{l} f_1 = \Theta(g_1) \\ f_2 = \Theta(g_2) \end{array}\right] \Rightarrow \frac{f_1}{f_2} = \Theta\left(\frac{g_1}{g_2}\right)$$

La notion d'ordre de grandeur asymptotique a une grande importance pratique en matière de complexité. Supposons par exemple que l'on dispose pour résoudre un problème donné de sept algorithmes dont les complexités ont respectivement pour ordre de grandeur 1, $\log n$, n, $n\log n$, n^2, n^3 et 2^n. Supposons d'autre part l'ordinateur capable d'effectuer 10^6 op/s.

n/complexité	1	$\log n$	n	$n\log n$	n^2	n^3	2^n
10^2	$1\mu s$	$6,6\mu s$	$0,1\mu s$	$0,6\mu s$	$10\mu s$	$1s$	$4\times 10^{16}a$
10^3	”	$9,9$	1	$9,9$	$1s$	$16,6m$	∞
10^4	”	$13,3$	10	$0,1s$	$100s$	$11,5j$	∞
10^5	”	$16,6$	$0,1s$	$1,6$	$2,7h$	$31,7a$	∞
10^6	”	$19,9$	1	$19,9$	$11,5j$	$31,7\times 10^3 a$	∞

Avec pour convention, $\infty > 10^{100}a$.
Ce tableau illustre les faits suivants :

- Les algorithmes utilisables pour des données de grande taille sont ceux qui s'exécutent en temps :
  - constant
  - logarithmique
  - linéaire
  - $nlogn$
- Les algorithmes qui prennent un temps polynomial, c'est à dire en $\Theta(n^k)$ avec $k > 0$, ne sont vraiment utilisables que pour $k < 2$. Lorsque $2 \leqslant k \leqslant 3$, on ne peut traiter que des données de taille moyenne. Si k dépasse 3, on ne peut traiter que des données de petite taille.
- Les algorithmes en temps exponentiel sont à peu près inutilisables, saut pour des données de très petite taille.

5.2 Définitions

Soient M une machine de Turing et w un mot sur son alphabet d'entrée. On va supposer que M s'arrête toujours (sans calcul infini). La longueur d'un calcul de M sur w est le nombre d'étapes nécessaires pour atteindre une configuration bloquante. Le temps de calcul $c_{\mathrm{M}}(w)$ de la machine M sur w est la longueur du plus long calcul admettant w comme entrée (la machine M pouvant très bien ne pas être déterministe). On prend en compte le nombre d'étapes effectuées dans le pire des cas.

On définit alors la complexité temporelle t_{M} de M (en fonction de la taille de la donnée) par la formule :

$$t_{\mathrm{M}}(n) = \max_{|w|=n} c_{\mathrm{M}}(w)$$

Cela représente la longueur du plus long calcul impliquant un mot de longueur n.

Proposition 5.2 (accélération)**.** *Soit t une fonction de $\mathbb{N} \to \mathbb{R}_+$ telle que $n = \mathrm{O}(t(n))$ et soit* M *une machine de Turing en temps $t(n)$. Pour toute constante k, il existe une machine de Turing* M′ *équivalente à* M *et en temps $t(n)/k$.*

Cette proposition justifie le sens de l'utilisation de la notation O*.*

Proposition 5.3. *Soit $t : \mathbb{N} \to \mathbb{R}_+$ une fonction telle que $t(n) \geqslant n$, $\forall n \geqslant 0$. Une machine à plusieurs bandes qui fonctionne en temps $t(n)$ est équivalente à une machine à une seule bande en temps $t^2(n)$.*

Proposition 5.4. *Soit $t : \mathbb{N} \to \mathbb{R}_+$ une fonction vérifiant les mêmes hypothèses que la proposition précédente. Une machine non déterministe en temps $t(n)$ est équivalente à une machine déterministe en temps $\mathrm{O}(2^{t(n)})$.*

5.3 Classes de complexité

On va distinguer les machines déterministes des non déterministes. Soit $t : \mathbb{N} \to \mathbb{R}_+$ une fonction. Pour les machines déterministes, on note $\mathrm{TIME}(t(n))$ la classe des problèmes pouvant être résolus en temps $t(n)$. Pour les machines non déterministes, la classe est notée $\mathrm{NTIME}(t(n))$:

$$\mathrm{TIME}(t(n)) = \{\mathrm{L} | \mathrm{L} \text{ peut être décidé en temps } t(n) \text{ par une machine de Turing déterministe.}\}$$
$$\mathrm{NTIME}(t(n)) = \{\mathrm{L} | \mathrm{L} \text{ peut être décidé en temps } t(n) \text{ par une machine du Turing non déterministe.}\}$$

La classe importante est celle des problèmes qui peuvent être résolus en temps polynomial par une machine déterministe. Cette classe est notée P :

$$\mathrm{P} = \bigcup_{k \geqslant 0} \mathrm{TIME}(n^k)$$

De manière analogue, on définit la classe des problèmes pouvant être résolus par une machine de Turing (non déterministe) en temps polynomial :

$$\mathrm{NP} = \bigcup_{k \geqslant 0} \mathrm{NTIME}(n^k)$$

On a bien sûr l'inclusion triviale P $\subset$ NP. Le problème de savoir s'il y a égalité ou non est un problème très difficile, encore ouvert à ce jour.

On considère aussi la classe EXPTIME des problèmes qui peuvent être résolus en temps exponentiel par une machine déterministe :

$$\text{EXPTIME} = \bigcup_{k \geqslant 0} \text{TIME}(2^{n^k})$$

On a bien sûr les inclusions P $\subset$ NP $\subset$ EXPTIME.

5.4 Exemples de problèmes dans P et NP

5.4.1 Accessibilité dans un graphe

Le problème est de savoir si un graphe orienté G contient un chemin de s à t pour deux sommets s et t également donnés. Le langage le codant peut s'écrire :

$$\text{L} = \{< \text{G}, s, t > | \text{G contient un chemin de } s \text{ à } t.\}$$

Ce problème peut être résolu par une parcours en largeur de G qui s'effectue en temps linéaire. Le langage L peut donc être décidé en temps polynomial par une machine de Turing déterministe. On est donc en présence d'un problème de la classe P.

5.4.2 Clique d'un graphe

Une clique de taille k d'un graphe orienté $\text{G} = (\text{V}, \text{E})$ est un sous-ensemble $\{v_1, \ldots, v_k\}$ de sommets de G tel que toutes les arêtes (v_i, v_j), pour $1 \leqslant i \leqslant j \leqslant k$, sont présentes dans G. Le problème CLIQUE consiste à savoir si un graphe donné contient une clique de taille k donnée. Il est codé par le langage :

$$\text{L} = \{< \text{G}, k > | \text{G possède une clique de taille } k\}$$

On dispose d'un algorithme non déterministe décidant ce langage. Il commence par choisir de façon non déterministe k sommets. Il peut le faire en temps linéaire en la taille de G. Ensuite, l'algorithme vérifie, ce qui peut être fait en temps polynomial (de l'ordre de k^2 arêtes à tester), que toutes les arêtes (v_i, v_j) sont dans G, pour $1 \leqslant i \leqslant j \leqslant k$. Le problème CLIQUE est dans NP.

5.4.3 Satisfiabilité d'une formule logique

Il s'agit du problème noté SAT. Il dispose d'une variante 3-SAT. Une formule est formée à partir de variables booléennes, en utilisant les trois connecteurs $\neg$, $\vee$ et $\wedge$. Par exemple, la formule $\Phi = (x \vee \neg y \vee z) \wedge (\neg x \vee t)$ utilise les quatre variables x, y, z et t.

Un littéral est soit une variable, soit la négation d'une variable, comme x ou $\neg x$. $\neg x$ est aussi souvent noté $\overline{x}$. Une clause est la disjonction ($\vee$) d'un nombre fini de littéraux. Par exemple, $x \vee \neg y \vee z$. Une formule est dite en forme conjonctive si elle est la conjonction ($\wedge$) d'un nombre fini de clauses.

Une formule est dite satisfiable s'il existe une distribution de valeurs sur les variables donnant la valeur 1 (vrai) à la formule. Le problème SAT consiste à savoir

si une formule donnée est satisfiable. Le problème 3-SAT consiste à savoir si une formule donnée sous la forme conjonctive avec 3 littéraux est satisfiable. Il est clair que 3-SAT représente un cas particulier de SAT. Une instance de 3-SAT est une formule de la forme :

$$\Phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$$

Il est clair que SAT et 3-SAT sont tous deux NP. Le traitement est similaire au problème CLIQUE.

5.4.4 Classe NP et vérification en temps polynomial

Les deux problèmes énoncés ci-dessus (CLIQUE et SAT) admettent des algorithmes similaires, constitués en deux parties. Dans la première, on choisit un objet de façon non déterministe, et dans la seconde on vérifie si cet objet répond au problème. Dans les deux cas, la vérification a lieu en temps polynomial. Il s'agit là d'un principe général : tout problème de la classe NP est équivalent à un problème de vérification en temps polynomial.

On appelle vérificateur en temps polynomial pour un langage L une machine de Turing déterministe V avec des entrées de la forme $< w, c >$ et avec un temps de calcul polynomial telle que :

$$\mathrm{L} = \{w / \exists c, \mathrm{V} \text{ accepte } < w, c > \}$$

Proposition 5.5. *Un langage* L *est dans la classe NP si et seulement s'il existe un vérificateur en temps polynomial pour* L.

5.5 Réductions

On va commencer par voir l'utilité d'une réduction sur un exemple : le problème de l'arrêt.

Considérons le langage $\mathrm{L} = \{< \mathrm{M}, w > / \mathrm{M} \text{ s'arrête sur } w \}$, M étant une machine de Turing, et w une donnée de M. Ce langage code le problème dit de « l'arrêt ».

Proposition 5.6. *Le problème de l'arrêt n'est pas décidable (*L *est non récursif).*

Démonstration. Pour le montrer, on va utiliser le fait que le problème de l'acceptation n'est pas décidable. Par l'absurde, supposons L récursif, c'est-à-dire accepté par une machine H sans calcul infini. Considérons la machine A suivante :

Machine A avec en entrée $< \mathrm{M}, w >$
Si H n'accepte pas $< \mathrm{M}, w >$ rejeter.
Sinon, simuler la machine M sur w jusqu'à l'arrêt et
accepter si M accepte et rejeter si M rejette.

La machine A est sans calcul infini et accepte les entrées $< \mathrm{M}, w >$ telles que M accepte w. Elle reconnaît donc le langage codant le problème de l'acceptation, ce qui est contradictoire puisqu'une telle machine n'existe pas. Alors H n'existe pas, et le problème de l'arrêt est du coup indécidable. □

L'idée de cette preuve est de ramener le problème de l'acceptation à celui de l'arrêt. Ce principe, général, porte le nom de réduction, car il consiste à réduire la résolution d'un problème à celle d'un autre.

Définition 5.1. Soient A et B deux problèmes codés par des langages L_A et L_B sur des alphabets Σ_A et Σ_B. Une réduction de A à B est une fonction $f : \Sigma_A^* \to |Sigma_B^*$ calculable telle que $w \in L_A \iff f(w) \in L_B$.

Cela signifie alors que le problème A n'est pas plus compliqué que B, puisque résoudre B suffit à résoudre A. On note $A \leqslant B$ lorsqu'il existe une réduction de A à B.

On a les énoncés suivants :

Proposition 5.7. *Si* $A \leqslant B$ *est* B *est décidable, alors* A *est décidable.*

Corollaire 5.1. *Si* $A \leqslant B$ *et* A *est indécidable, alors* B *est indécidable.*

Définition 5.2. Lorsque la fonction f est calculable en temps polynomial, on dit que la réduction est polynomiale. On note plus précisément $A \leqslant_P B$.

Proposition 5.8. *Il existe une réduction polynomiale du problème 3-SAT au problème CLIQUE.*

5.6 NP-complétude

Intuitivement, un problème est NP-complet s'il figure parmi les problèmes les plus difficiles de la classe NP.

Définition 5.3. Un problème A est dit NP-complet s'il satisfait les deux conditions suivantes :

1. $A \in NP$,
2. $\forall B \in NP$, on a $B \leqslant_P A$.

Si seule la seconde condition est vérifiée, on dit que A est NP-difficile (*NP-hard*).

Remarque. Il suffit qu'un seul problème NP-complet soit dans P pour avoir l'égalité P = NP. Mais ce problème demeure ouvert.

Le théorème suivant, dû à Cook et Levin, établit l'existence d'au moins un problème NP-complet.

Théorème 5.1 (Cook-Levin)**.** *Les problèmes SAT et 3-SAT sont tous deux NP-complets.*

Démonstration. On a déjà vu que SAT et 3-SAT sont dans NP. Pour montrer qu'ils sont NP-complets, on va montrer que tout problème de NP se réduit à eux deux.

Dans une première étape, on réduit un problème quelconque de NP à SAT. SAT est alors NP-complet. Dans un second temps on réduit SAT à 3-SAT. En composant les deux réductions, on s'aperçoit que 3-SAT est également NP-complet. □

Autres problèmes NP-complets

Pour montrer qu'un problème A est NP-complet, il faut d'abord montrer qu'il est dans la classe NP. Ensuite, il suffit de montrer qu'un seul problème NP-complet B s'y réduit. En effet :

Proposition 5.9. *Si* B *est NP-complet et si* $B \leqslant_P A$*, alors* A *est NP-difficile.*

Démonstration. Puisque B est NP-complet, on a $C \leqslant_P B$, $\forall C \in NP$. Si on a aussi $B \leqslant_P A$, on obtient $C \leqslant_P A$, en combinant les deux réductions. □

Cette proposition fournit une technique permettant de savoir si un problème est NP-complet en utilisant un autre problème su pour être NP-complet.

1. On sait que $CLIQUE \in NP$. On a aussi vu que $3-SAT \leqslant_P CLIQUE$. Il découle de la proposition précédente que CLIQUE est NP-complet.
2. Le problème de la couverture de sommet (VERTEX-COVER) est également NP-complet. En effet, il est dans NP et 3-SAT s'y réduit polynomialement.
3. Le problème du chemin hamiltonien : un chemin qui passe une fois et une seule par chaque sommet du graphe. Le problème HAM-PATH consiste à savoir si un graphe G contient un chemin hamiltonien reliant un sommet s à t. G pouvant être orienté ou non. Ce problème est NP-complet. Il est d'abord dans NP et on démontre ensuite que 3-SAT s'y réduit polynomialement.
4. Le problème de la somme (SUBSET-SUM). Soient $s, x_1, \ldots, x_k$, $k+1$ entiers. Est-il possible d'extraire de $x_1, \ldots, x_k$ une sous-suite de somme s ? Ce problème est NP-complet. Il est clairement dans NP. Pour montrer ensuite que SUBSET-SUM est NP-difficile, on prouve que 3-SAT s'y réduit polynomialement.