Cours INF561
de l'Ecole Polytechnique

Algorithmes et Complexité

Olivier Bournez

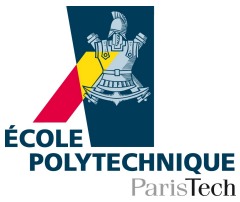


Table des matières

Chapitre 1

Introduction

Objectifs

Ce cours s'intéresse aux algorithmes et à leur efficacité.

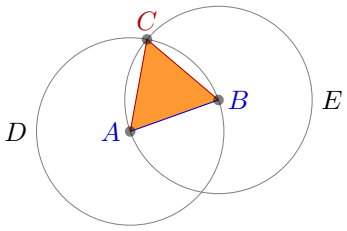
Autrement dit, l'objectif de ce cours est de répondre à la question suivante : quelles sont les limites des algorithmes, et des systèmes de calcul ?

Algorithme ?

Le mot algorithme vient du nom du mathématicien Al-Khawarizmi (latinisé au Moyen Âge en Algoritmi), qui au IXe siècle écrivit plusieurs ouvrages sur la résolution des équations.

On peut définir informellement un algorithme comme une liste de tâches bien définies pour résoudre un problème.

L'exemple suivant est inspiré du manuel du paquetage *TikZ-PGF* version 2.0, lui-même inspiré des *Éléments* d'*Euclide*.



Algorithme:
Pour construire un triangle équilatéral ayant pour coté AB: tracer le cercle de centre A de rayon AB; tracer le cercle de centre B de rayon AB. Nommer C l'une des intersection de ces deux cercles. Le triangle ABC est la solution recherchée.

Nous tenterons de donner une définition formelle de ce que l'on appelle un algorithme.

Nous nous intéresserons ensuite à différents problèmes, à distinguer les problèmes qui admettent une solution par algorithme, puis à discuter les ressources (temps, mémoire, etc) nécessaires pour la résolution d'un problème.

Contenu du document

Plus précisément, dans le chapitre 2, nous introduisons un certain nombre de concepts de base de l'informatique, et de rappels nécessaires à la suite du document.

Dans le chapitre 3, nous présentons quelques algorithmes classiques. Cela nous permet de discuter des questions liées à d'une part ce qu'est un algorithme, et sur quel type de structures ils fonctionnent. D'autre part, cela nous permet de discuter des questions liées à la terminaison des algorithmes, et à leur complexité en temps. Nous en profiterons finalement pour formaliser et clarifier un peu la façon dont nous écrivons les algorithmes.

Dans le chapitre 4, nous définissons formellement la notion d'algorithme en utilisant l'approche axiomatique de [Gurevich, 2000].

Dans le chapitre 5, nous établissons l'équivalence entre cette notion d'algorithme et d'autres façons de les définire en utilisant les modèles classiques de la calculabilité et complexité.

Dans le chapitre 6, nous introduisons la théorie de la calculabilité et ses principaux résultats. La théorie de la calculabilité permet de discuter de l'existence, ou de la non-existence d'algorithmes pour résoudre un problème donné.

Dans le chapitre 7, nous introduisons la notion de circuit booléen, et établissons quelques relations reliant leur taille à leur profondeur.

Dans le chapitre 8, nous introduisons les classes de complexité en temps, déterministes et non-déterministes, leurs relations avec les circuits, la notion de problème complet, et la question $P = NP$.

Dans le chapitre 9, nous introduisons les classes de complexité en espace, déterministes et non-déterministes, leurs relations avec les classes de complexité en temps, et l'existence de problèmes complets pour les classes de complexité en espace usuelles.

Dans le chapitre 11, nous introduisons les classes de complexité parallèles. Nous introduisons le modèle de calcul parallèle PRAM. Nous présentons ensuite les modèles parallèles basés sur les circuits, et relions ces modèles parallèles, et aussi ces modèles aux modèles séquentiels.

Dans le chapitre 11, nous présentons quelques algorithmes probabilistes, en cherchant à présenter quelques techniques probabilistes utilisées en algorithmique.

Dans le chapitre 12 nous présentons les classes probabilistes en cherchant à les positionner par rapport aux autres classes de complexité.

Les chapitres 13 et 14 présentent brièvement la hiérarchie polynomiale et les systèmes de preuve interactifs.

Chapitre 2

Préliminaires

Dans ce chapitre préliminaire, nous introduisons quelques notions standards : la notion de fonction booléenne, de graphe, d'arbre, de mot, de langage, de codage, pour en arriver à la notion de structure et de modèle en logique du premier ordre.

2.1 Algèbre

On notera $\mathbb{N}$ l'ensemble des entiers naturels, et $\mathbb{Z}$ l'ensemble des entiers relatifs. $\mathbb{Z}$ est un anneau.

Lorsque p et q sont deux entiers, on notera $p \ mod \ q$ le reste de la division euclidienne de p par q, et $p \ div \ q$ le quotient : autrement dit, $p = q * (p \ div \ q) + (p \ mod \ q)$. On rappelle que la relation de congruence modulo p est une relation d'équivalence : deux entiers n et n' sont dans la même classe si $n \ mod \ p = n' \ mod \ p$. On notera, comme il en est l'habitude, $\mathbb{Z}_p$ pour l'anneau quotient obtenu en quotientant $\mathbb{Z}$ par cette relation d'équivalence.

$\mathbb{Z}_p$ est un corps lorsque p est un entier naturel premier. En particulier, pour $p = 2$, $\mathbb{Z}_2$ est un (l'unique) corps à deux éléments, obtenu en raisonnant modulo 2 dans $\mathbb{Z}$. Rappelons que dans $\mathbb{Z}_2$, tous les calculs se font modulo 2 : c'est-à-dire que l'on a $0 + 0 = 0$, $0 + 1 = 1 + 0 = 1$, $1 + 1 = 0$, $1.1 = 1$, $1.0 = 0.1 = 0.0 = 0$.

On notera $\mathbb{R}$ l'ensemble des réels, et $\mathbb{C}$ l'ensemble des complexes. $\mathbb{R}$ et $\mathbb{C}$ sont des corps. On notera $\mathbb{R}^{\geq 0}$ l'ensemble des réels positifs ou nuls. D'autre part, lorsque $\mathbb{K}$ est un corps, on notera $\mathbb{K}[X]$ pour l'anneau des polynômes à coefficients sur $\mathbb{K}$.

2.2 Calculs booléens

Une *fonction booléenne* n-aire est une application de $\{0,1\}^n$ dans $\{0,1\}$.

Par commodité, on peut voir $\{0,1\}$ comme $\mathbb{Z}_2$. Cela nous permettra d'écrire certaines fonctions booléennes très facilement comme des fonctions de ${\mathbb{Z}_2}^n \rightarrow \mathbb{Z}_2$, et aussi comme des polynômes sur $\mathbb{Z}_2$.

	¬
0	**1**
1	**0**

		∧
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

		∨
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

FIG. 2.1 – Table de vérités de ¬, ∧ et ∨.

Par exemple :
- Il n'y a que quatre fonctions booléennes unaires (1-aire) : l'identité, et la négation ¬, qui à $x \in \mathbb{Z}_2$, associe $1 + x$, et les fonctions constantes 0 et 1 qui à $x \in \mathbb{Z}_2$ associent 0 et 1 respectivement.
- Il y a 16 fonctions binaires (2-aire) : parmi ces fonctions, il y a l'identité, la conjonction ∧, qui à $x, y \in \mathbb{Z}_2$ associe $\wedge(x, y) = xy$, la disjonction ∨ qui à $x, y \in \mathbb{Z}_2$ associe $\vee(x, y) = x + y + xy$, l'implication ⇒ qui à $x, y \in \mathbb{Z}_2$ associe $\Rightarrow (x, y) = y \vee (\neg x)$, c'est-à-dire $xy + x + 1$, et l'équivalence qui à $x, y \in \mathbb{Z}_2$ associe $\wedge(\Rightarrow (x, y), \Rightarrow (y, x))$, c'est-à-dire $1 + x + y$.

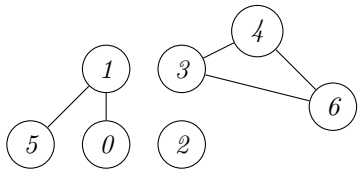
Bien entendu, ces fonctions portent ces noms, car si l'on identifie $vrai$ avec 1 et $faux$ avec 0, elles correspondent à la fonction logique naturelle associée : par exemple, $\wedge(x, y)$ vaut $vrai$ si et seulement si x vaut $vrai$ et si y vaut $vrai$. Le symbole ∧ correspond donc à la conjonction logique : voir la figure 2.1

2.3 Graphes

Un graphe $G = (V, E)$ est donné par un ensemble V, dont les éléments sont appelés *sommets*, ou nœud, et d'une partie de $E \subset V \times V$, dont les éléments sont appelés des *arcs*. Un chemin de s à t est une suite $(s = s_0, \ldots, s_n = t)$ de nœuds tels que, pour $1 \leq i \leq n$, (s_{i-1}, s_i) soit un arc. Un *circuit* est un chemin de longueur non nulle dont l'origine coïncide avec l'extrémité.

Si les arcs ne sont pas orientés, c'est-à-dire, si l'on considère que (u, v) et (v, u) sont équivalents, on dit que le graphe G est *non-orienté*, et les éléments de E sont appelés des *arêtes*. Lorsque il y a un arête entre u et v, c'est-à-dire lorsque $(u, v) \in E$, on dit que u et v sont voisins.

Exemple 2.1 *Le graphe (non-orienté) $G = (V, E)$ avec*
- $V = \{0, 1, \ldots, 6\}$
- $E = \{(0, 1), (3, 4), (5, 1), (6, 3), (6, 4)\}$.



Un graphe est dit *connexe* si deux quelconques de ses nœuds sont reliés par un chemin.

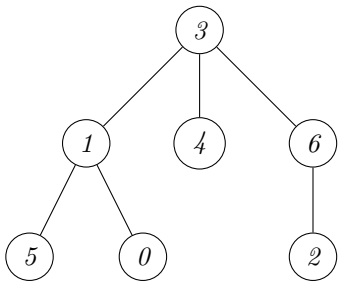
Exemple 2.2 *Le graphe de l'exemple 2.1 n'est pas connexe.*

2.4 Arbres

Nous aurons besoin de la notion d'arbre (étiqueté). Les arbres peuvent être présentés de plusieurs façons.

Une façon est de passer par la notion de graphe. Un *arbre* est un graphe non-orienté connexe et sans circuit. Une feuille est un nœud de l'arbre qui ne possède qu'un seul voisin. Un noeud qui n'est pas une feuille est dit *un noeud interne*.

Exemple 2.3 (Un arbre) *Représentation graphique d'un arbre, dont les feuilles sont les nœuds* $5, 0, 4$ *et* 2.



En fait, presque tous nos arbres seront étiquetés : Soit A un ensemble dont les éléments sont appelés des *étiquettes*. Un *arbre étiqueté par* A est un arbre $G = (V, E)$ et une fonction qui associe à chaque sommet de V un élément de A.

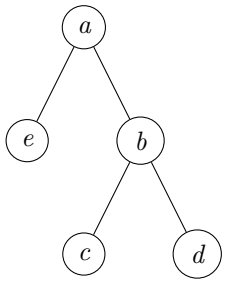
Une autre façon de définir un arbre est de passer par une définition inductive : par exemple, un arbre étiqueté par A peut se définir de la façon suivante : un arbre étiqueté par A, et soit un élément de A (une feuille), ou un élément de A (une racine r) et une liste d'arbres (ses fils $f_1, \cdots, f_k$). S'il on préfère, les arbres étiquetés correspondent au plus petit ensemble qui contienne A, et tel que si $f_1, \cdots, f_k$ sont des arbres, alors le couple constitué d'un élément r de A, et de la liste $f_1, \cdots, f_k$ est un arbre.

Cette dualité entre ces deux façons de définir les arbres, soit inductivement, comme le plus petit ensemble qui contienne quelque chose, et qui soit clos par une opération construisant un nouvel arbre à partir d'arbres, soit par les graphes, se retrouvera à de multiples reprises dans ce document.

Par ailleurs, à plusieurs reprises dans ce document, nous aurons besoin de décrire des arbres. Formellement, dans le cas général, on notera $a(f_1, \cdots, f_k)$ l'arbre de racine a ayant pour fils les arbres $f_1, \cdots, f_k$ respectivement.

Exemple 2.4 *$a(e, b(c, d))$ désigne l'arbre dont la racine, étiquetée par a possède le fils gauche étiqueté par e et un fils droit étiqueté par b, ayant deux fils étiquetés par c et d.*

Graphiquement, cet arbre peut se représenté comme :



Exemple 2.5 *Par exemple, si A est l'ensemble d'étiquette $A = \{\vee, \wedge\}$, l'arbre $\vee(a, \wedge(b, a))$ désigne l'arbre dont la racine, étiquetée par $\vee$ possède le fils gauche a et un fils droit étiqueté par $\wedge$, ayant pour fils b et a.*

2.5 Alphabets, Mots, Langages

Nous rappelons maintenant quelques définitions élémentaires sur les mots et les langages.

La terminologie, empruntée à la linguistique, rappelle que les premiers travaux sont issus de la modélisation de la langue naturelle.

On fixe un ensemble Σ que l'on appelle alphabet. Contrairement à de nombreux ouvrages, nous ne supposerons pas Σ nécessairement fini.

Les éléments de Σ sont appelés des *lettres* ou des symboles. Un mot w sur l'alphabet Σ est une suite finie $w_1 w_2 \cdots w_n$ de lettres de Σ. L'entier n est appelé la *longueur* du mot w. Il sera noté $|w|$.

Le mot vide ϵ est le seul mot de longueur 0. Un *langage sur* Σ est un ensemble de mots sur Σ. L'ensemble de tous les mots sur l'alphabet Σ est noté Σ^*.

On définit une opération de concaténation sur les mots : la concaténation du mot $u = u_1 u_2 \cdots u_n$ et du mot $v = v_1 v_2 \cdots v_m$ est le mot noté $u.v$ défini par $u_1 u_2 \cdots u_n v_1 v_2 \cdots v_m$, c'est-à-dire le mot dont les lettres sont obtenues en juxtaposant les lettres de v à la fin de celles de u. L'opération de concaténation notée . est associative, mais non-commutative. Le mot vide est un élément neutre à droite et à gauche de cette opération. On appelle aussi Σ^* le *monoïde* libre sur l'alphabet Σ.

On note aussi uv pour la concaténation $u.v$. En fait, tout mot $w_1 w_2 \cdots w_n$ peut se voir comme $w_1.w_2 \cdots .w_n$, où w_i représente le mot de longueur 1 réduit à la lettre w_i. Cette confusion entre les lettres et les mots de longueur 1 est souvent très pratique.

Exemple 2.6 *Par exemple, si Σ est l'ensemble $\Sigma = \{a, b\}$, aaab est le mot de longueur 4 dont les trois premières lettres sont a, et la dernière est b.*

Un mot u est un *préfixe* d'un mot w, s'il existe un mot z tel que $w = u.z$. Un mot u est un *suffixe* d'un mot w s'il existe un mot z tel que $w = z.u$.

Étant donnés deux langages L et L' sur un même alphabet Σ, leur union, notée $L \cup L'$ est définie par $L \cup L' = \{w | w \in L \text{ ou } w \in L'\}$. Leur intersection, notée $L \cap L'$, est définie par $L \cap L' = \{w | w \in L \text{ et } w \in L'\}$. Leur concaténation, notée $L.L'$ est définie par $L.L' = \{w | w = w_1.w_2, w_1 \in L, w_2 \in L'\}$.

Le complément d'un langage L sur l'alphabet Σ est le langage L^c défini par $L^c = \{w | w \notin L\}$.

2.6 Changement d'alphabet

Il est souvent utile de pouvoir réécrire un mot sur un alphabet en un mot sur un autre alphabet : on procède par réécriture lettre par lettre.

Exemple 2.7 *Par exemple, si Σ est l'alphabet $\Sigma = \{a, b, c\}$, et $\Gamma = \{0, 1\}$, on peut coder les mots de Σ^* sur Γ^* par la fonction h telle que $h(a) = 01$, $h(b) = 10$, $h(c) = 11$. Le mot abab se code alors par $h(abab) = 01100110$, c'est-à-dire par le mot obtenu en codant lettre par lettre.*

Très formellement, étant donnés deux alphabets Σ et Γ, un *homomorphisme* est une application de Σ^* dans Γ^* telle que

- $h(\epsilon) = \epsilon$
- $h(u.v) = h(u).h(v)$ pour tout mots u et v.

Tout homomorphisme est parfaitement déterminé par son image sur les lettres de Σ, et peut s'étendre aux mots de Σ^* par

$$h(w_1 w_2 \cdots w_n) = h(w_1).h(w_2).\cdots.h(w_n)$$

pour tout mot $w = w_1 w_2 \cdots w_n$.

2.7 Codage, Décodage

Nous aurons très souvent besoin de coder des mots, des entiers, des listes, des arbres, des graphes ...par des mots. Il nous faut pour cela parler de codage.

Formellement, un *codage* d'un ensemble E sur un alphabet Σ est une fonction injective de E vers Σ^* : on notera systématiquement $< x >$ pour le codage de $x \in E$.

En fait, dans beaucoup de contextes, on souhaite implicitement des codages *"naturels"* : ici, "naturel" fait référence à une notion intuitive, difficile à définir. Cependant, l'idée est que l'on souhaite qu'un codage soit facile à calculer (encoder), et facile à décoder.

Si l'on veut être plus précis, on peut par exemple fixer les conventions suivantes : on suppose que l'on travaille sur des alphabets qui contiennent au moins deux symboles, l'un noté **0**, et l'autre **1**.

Exemple 2.8 (Codage des entiers) *On peut coder les entiers*

- *en base* 2 *:* $< x >$ *serait l'écriture en binaire de* x *sur l'alphabet* $\Sigma = \{\mathbf{0}, \mathbf{1}\}$. *En particulier,* $< 12 >= \mathbf{1100}$.
- *en base* 10 *:* $< x >$ *serait l'écriture usuelle de* n *sur l'alphabet* $\Sigma = \{\mathbf{0}, \mathbf{1}, \cdots, 9\}$. *En particulier,* $< 12 >= \mathbf{12}$.
- *en unaire :* $< x >$ *serait le mot* $\mathbf{1}^x$, *c'est-à-dire le mot de longueur* x *sur l'alphabet* $\Sigma = \{\mathbf{1}\}$. *En particulier,* $< 12 >= \mathbf{111111111111}$.

(Sauf mention du contraire) on fixe que tout entier est codé en binaire, donc comme un mot de $\{\mathbf{0}, \mathbf{1}\}^*$.

On a souvent besoin de coder des paires (ordonnées). On suppose construite une fonction $< ., . >$ qui à deux mots $u, v \in \Sigma^*$ sur l'alphabet Σ associe un mot $< u, v >$ qui code la paire u, v.

Exemple 2.9 *Par exemple, prenons la convention suivante. On choisit un symbole* $\#$ *qui n'est pas dans l'alphabet* Σ. *On fixe un homomorphisme* h *de* $\Sigma \cup \{\#\}$ *sur* Σ, *et on définit* $< u, v >$ *comme* $h(u\#v)$ *pour tout* u, v. *Par exemple, l'homorphisme* h *peut être celui qui réécrit* $h(x)$ *en* $\mathbf{1}x$ *pour* $x \neq \#$, *et* $h(\#) = \mathbf{00}$.

Ce n'est qu'une façon de faire, d'autres sont possibles. Il nous importe dans ce qui suit simplement que l'on puisse facilement retrouver u et v à partir du mot $< u, v >$.

Une fois que l'on sait encoder une paire, on peut encoder des n-uplets : on définit pour les mots u, v, w, $< u, v, w >$ par $< u, v, w >=<< u, v >, w >$, puis pour les mots u, v, w, x, $< u, v, w, x >$ par $< u, v, w, x >=<<< u, v >, w >, x >$, ...etc. On code ainsi toutes les suites de longueur fixée, ou si l'on préfère toutes les listes de longueur fixée.

Les suites ou les listes de longueur variables peuvent aussi se coder de cette façon, mais il est souvent utile de connaître la longueur de la suite. Aussi, il est plus pertinent de fixer le codage suivant :

$$< x_1, x_2, \cdots, x_n >=<<<< x_1, x_2 > \cdots, x_n >, n >$$

Encore une fois, il nous importe simplement dans ce qui suit simplement que l'on puisse facilement retrouver chacun des x_i à partir du codage de la suite.

Toutes ces notions se composent naturellement.

Exemple 2.10 *Par exemple, pour coder une paire d'entiers* u, v, *on utilisera le mot* $<< u >, < v >, >$, *c'est-à-dire le codage de la paire dont le premier élément est le codage de* u *(donc son écriture en binaire) et le deuxième élément le codage de* v *(donc son écriture en binaire).*

Exemple 2.11 *Par exemple, puisque nous avons vu qu'un arbre peut se définir inductivement comme une racine et une liste d'arbres (de fils de la racine), on peut coder l'arbre* $a(e, b(c, d))$ *par* $<< a >, << b >, << c >, < d >>>>$.

Les graphes peuvent se coder soit par leur matrice d'adjacence, ou par des listes d'adjacence. Nous laissons au lecteur le soin de fixer les détails.

2.8 Logique Propositionnelle

Nous allons maintenant introduire la logique propositionnelle.

La logique propositionnelle permet de parler des fonctions booléennes. La logique du premier ordre permet de parler de formules mathématiques générales, avec par exemple des quantifications.

Pour définir formellement la logique, il faut distinguer la syntaxe (le vocabulaire avec lequel on écrit) de la sémantique (le sens que l'on donne à cette syntaxe).

2.8.1 Syntaxe de la logique propositionnelle

Formellement, on considère un ensemble fini P non-vide d'éléments, que l'on appelle des *variables propositionnelles*.

Une *formule propositionnelle* (sur P) est un arbre, dont les feuilles sont étiquetées par les variables propositionnelles de P, dont les nœuds internes ont soit un ou deux fils. Chaque nœuds ayant

- un fils est étiqueté par $\neg$,
- deux fils est étiqueté par $\vee$, $\wedge$, $\Rightarrow$, ou $\Leftrightarrow$.

Si l'on préfère[1], l'ensemble des formules propositionnelles sur P est le plus petit sous-ensemble qui contient P, tel que si F est une formule propositionnelle, alors $\neg F$ aussi, et tel que si F et G sont des formules propositionnelles, alors $F \vee G$, $F \wedge G$, $F \Rightarrow G$, et $F \Leftrightarrow G$ aussi.

Exemple 2.12 *Par exemple, l'arbre $\vee(a, \wedge(b, a))$ est une formule propositionnelle. On note aussi cet arbre / cette formule $a \vee (b \wedge a)$.*

2.8.2 Sémantique de la logique propositionnelle

Une formule propositionnelle ayant n feuilles distinctes décrit une fonction booléenne n-aire comme on s'y attend.

C'est-à-dire formellement : une *valuation* σ est une fonction qui associe à chaque élément de P un élément de $\mathbb{Z}_2 = \{0, 1\}$, c'est-à-dire une fonction qui décrit la valeur des variables (propositionnelles).

Pour connaître la valeur (l'interprétation) d'une formule sur P pour la valuation σ, on associe à (évalue) chaque feuille étiquetée par la variable propositionnelle p par $\sigma(p)$ et on propage les valeurs de bas en haut : en chaque nœud interne étiqueté par $g \in \{\neg\}$, (respectivement $g \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$, on évalue ce nœud par $g(f_1)$ (resp. $g(f_1, f_2)$), où f_1 est l'évaluation de son fils (resp. f_1 et f_2 sont les évaluations de son fils droit et de son fils gauche. La valeur de la fonction se lit alors comme l'évaluation de la racine à l'issu de ce processus.

Exemple 2.13 *Par exemple, pour la valuation $\sigma(a) = 0$, $\sigma(b) = 1$, la formule $a \vee (b \wedge a)$ s'évalue en $\vee(\sigma(a)), \wedge(\sigma(b), \sigma(a)) = \vee(0, \wedge(1, 0)) = \vee(0, 0) = 0$.*

[1] Il ne s'agit que d'une instance de notre remarque sur les différentes façon de définir un arbre

On peut assez facilement se convaincre du résultat suivant

Proposition 2.1 *Toute* fonction booléenne *n-aire f, donc de ${\mathbb{Z}_2}^n \to \mathbb{Z}_2$, peut s'écrire comme un polynôme à n-variables à coefficients dans $\mathbb{Z}_2$.*

En fait, si $x_1, \cdots, x_n$ sont ses variables, la fonction f s'écrit

$$f(x_1, \cdots, x_n) = \sum_{x_1, x_2, \cdots, x_n \in \{0,1\}} f(x_1, x_2, \cdots, x_n) \overline{x_1}.\overline{x_2} \cdots \overline{x_n},$$

où $\overline{x_i}$ est x_i quand x_i vaut 1, et $1 + x_i$ lorsque x_i vaut 0.

Le produit $. : x, y \mapsto x.y$ dans $\mathbb{Z}_2$ correspondant à la fonction $\wedge(x, y)$, $x \mapsto 1 + x$ à la fonction $\neg(x)$, et la somme $+ : x, y \mapsto x + y$ correspondant à la fonction $\neg(\wedge(\neg(x), \neg(y)))$, en développant le polynôme f dans le corps $\mathbb{Z}_2$ et en effectuant ces substitutions de sommes et de produits par ces expressions, on obtient :

Corollaire 2.1 *Toute* fonction booléenne *n-aire correspond à une formule booléenne : pour toute fonction booléenne n-aire, on peut construire une formule booléenne qui la décrit.*

Il y a au moins une, et en fait toujours une infinité de formules booléennes qui correspond à la fonction : par exemple, $\neg a$ peut aussi s'écrire $(\neg a) \vee (a \wedge \neg a)$, ou comme $(\neg a) \vee (a \wedge \neg a) \vee (a \wedge \neg a)$, etc...

Exemple 2.14 *Considérons la fonction booléenne* sélecteur *qui nous sera utile par la suite, définie par $S(1, y, z) = y$, et $S(0, y, z) = z$. Cette fonction $S(x, y, z)$ correspond au polynôme $x.y + (1 - x).z$, qui est décrit par la formule booléenne $\vee(\wedge(\neg(x), y), \wedge(x, z))$.*

La logique propositionnelle est donc complète pour les fonctions booléennes : elle permet d'exprimer toutes les fonctions booléennes.

2.9 Structures du premier ordre

Elle reste cependant très limitée.

Nous allons faire un pas vers un formalisme plus général, en autorisant la notion de symbole de fonction et de symbole de relation. Nous ne présenterons pas en fait toute la logique du premier ordre mais seulement la notion de structure.

2.9.1 Signature

Une *signature* $\Sigma = (\mathcal{F}, \mathcal{R})$ *(d'un langage du premier ordre)* est la donnée de deux ensembles de symboles : les éléments du premier ensemble $\mathcal{F}$ sont appelés les *symboles de fonctions* ; les éléments du second ensemble $\mathcal{R}$ sont appelés les *symboles de prédicats (aussi appelés symboles de relations).* A chaque symbole de fonction et de prédicat est associé une *arité*, c'est-à-dire un entier (qui se veut un nombre d'arguments). Une fonction d'arité 0 (sans argument) s'appelle *une constante.*

Exemple 2.15 *Par exemple,* $\Sigma = (\{\mathbf{0}, \mathbf{1}, +, -\}, \{=\})$ *correspond à la signature qui possède* 4 *symboles de fonctions (*$\mathbf{0}$ *et* $\mathbf{1}$ *d'arité* 0*,* $+$ *d'arité* 2*, et* $-$ *d'arité* 2*) et un symbole de relation* $=$ *d'arité* 2*.*

Pour éviter nombre de problèmes techniques, on va faire les hypothèses suivantes sur les signatures que l'on considère dans ce document.

1. On ne considère que des signatures qui ont au moins deux symboles de constantes $\mathbf{0}$ et $\mathbf{1}$, et un symbole de relation $=$ d'arité 2.

2.9.2 Terme clos

Un *terme (clos)* est un arbre, dont les feuilles sont étiquetées par les symboles de constantes, dont les nœuds internes ayant k fils sont étiquetés par un symbole de fonction ou de relation d'arité k.

Exemple 2.16 *Par exemple,* $+(\mathbf{1}, -(\mathbf{0}, \mathbf{1}))$ *est un terme clos sur la signature précédente. On convient de noter aussi cet arbre* $\mathbf{1} + (\mathbf{0} - \mathbf{1})$*.*

Exemple 2.17 *Par exemple,* $=(\mathbf{1}, \mathbf{0})$ *est un terme clos sur la signature précédente. On convient de noter aussi cet arbre* $\mathbf{1} = \mathbf{1}$*.*

2.9.3 Sémantique

Classiquement, on définit la sémantique de la façon suivante :

Une *réalisation* d'une signature $\Sigma = (\mathcal{F}, \mathcal{R})$ (encore appelé *modèle* de Σ, ou encore *structure (du premier ordre) sur* Σ) est la donnée d'un ensemble non vide M, appelé *ensemble de base*, et pour chaque symbole de fonction $f \in \mathcal{F}$ d'arité n d'une fonction $[\![f]\!] : M^n \to M$, appelée *interprétation de* f, et pour chaque symbole de prédicat $r \in \mathcal{R}$ d'arité n d'un sous-ensemble $[\![r]\!]$ de M^n, appelé *interprétation de* r.

Exemple 2.18 *Par exemple, une réalisation de la signature* $\Sigma = (\{\mathbf{0}, \mathbf{1}, +, -\}, \{=\})$ *correspond à l'ensemble de base* $\mathbb{N}$ *des entiers naturels, où* $\mathbf{0}$ *est interprété par l'entier* 0*,* $\mathbf{1}$ *par* 1*,* $+$ *par l'addition,* $-$ *par la soustraction, et* $=$ *par l'égalité sur les entiers.*

Exemple 2.19 *Une autre une réalisation de cette signature correspond à l'ensemble de base* $\mathbb{R}$ *des réels, où* $\mathbf{0}$ *est interprété par le réel* 0*,* $\mathbf{1}$ *est interprété par le réel* 1*,* $+$ *par l'addition,* $-$ *la soustraction, et* $=$ *par l'égalité sur les réels.*

Dans le contexte de calculabilité et de complexité qui nous intéresse dans ce document, il est en fait plus simple de considérer qu'en fait on a que des fonctions. Nos structures contiennent au moins deux éléments notés $\mathbf{0}$ et $\mathbf{1}$. On convient que $\mathbf{1}$ s'interprète par *vrai* et $\mathbf{0}$ par *faux*. A un symbole de prédicat $r \in \mathcal{R}$ d'arité n, plutôt que d'associer un sous-ensemble de M^n, on associe une fonction $[\![r]\!]$ (la fonction caractéristique de cet ensemble) qui vaut soit l'interprétation de $\mathbf{0}$, notée $vrai$, soit l'interprétation de $\mathbf{1}$, notée $faux$.

Le sous-ensemble se relie à cette fonction par le fait que cette dernière est son ensemble caractéristique : $[\![r]\!]$ vaut $vrai$ en un n-uplet si et seulement si ce n-uplet est dans le sous-ensemble $[\![r]\!]$, et $faux$ sinon. C'est la convention que l'on fixe pour la suite.

On fixe aussi les conventions suivantes que l'on peut qualifier de "bon sens".

1. Dans les structures considérées, l'interprétation de l'égalité est l'égalité : c'est-à-dire $[\![=]\!]$ est la fonction à deux arguments qui vaut $vrai$ ssi ses deux arguments sont les mêmes.
2. Dans le même esprit, on utilisera les symboles de fonctions et de prédicats usuels avec leurs interprétations usuelles : $+$, s'interprète comme l'addition (c'est-à-dire formellement $[\![+]\!]$ est l'addition), $-$ comme la soustraction, $=$ pour l'égalité, etc.
3. On notera 1 ou $vrai$, pour l'interprétation de $\mathbf{1}$ (c'est-à-dire pour $[\![1]\!]$) et 0 ou $faux$, pour celle de $\mathbf{0}$. On suppose que ces interprétations $vrai$ et $faux$ ne correspondent pas à la même valeur.

Étant donnés un terme t l'interprétation du terme t, notée $[\![t]\!]$ est un élément de M qui se définit inductivement comme on s'y attend : si t est un symbole de constante c alors $[\![t]\!]$ est $[\![c]\!]$; sinon t est de la forme $f(t_1, \cdots, t_k)$, pour un symbole de fonction ou de relation f, et alors $[\![f]\!]$ est $[\![f]\!]([\![t_1]\!], \cdots, [\![t_k]\!])$.

Exemple 2.20 *Par exemple, sur les structures des deux exemples précédents terme* $(\mathbf{1}+\mathbf{1})+\mathbf{1}$*, c'est-à-dire l'arbre* $+(+(\mathbf{1},\mathbf{1}),\mathbf{1})$*, s'interprète par l'entier ou le réel* 3*.*

Exemple 2.21 *Par exemple, sur les structures des deux exemples précédents, le terme* $\mathbf{1}=\mathbf{0}$*, c'est-à-dire l'arbre* $=(\mathbf{1},\mathbf{0})$*, s'interprète par* $faux$*, c'est-à-dire* 0*.*

Nous noterons souvent une structure par $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ cette notation désigne la structure de signature $(\{f_1, \cdots, f_u\}, \{r_1, \cdots, r_v\})$ sur l'ensemble de base M, où chaque fonctions f_i et chaque relation r_i possède son interprétation "usuelle" sur M.

Exemple 2.22 *Par exemple, sur la structure* $\mathfrak{M} = (\mathbb{N}, \mathbf{0}, \mathbf{1}, +, =)$*, le terme* $(\mathbf{1}+\mathbf{1})+\mathbf{1}$ *s'interprète par l'entier* 3*.*

2.9.4 Combinaisons booléennes

Un *terme booléen* est soit le terme $\mathbf{1}$, soit le terme $\mathbf{0}$, soit un terme dont la racine est étiqueté par un symbole de relation. Un terme booléen s'interprète donc soit par $vrai$ (noté aussi 1) soit par $faux$ (noté 0).

On va fixer en outre la convention suivante :

1. Les structures considérées possèdent au moins un terme $S(x, y, z)$ dont l'interprétation $[\![S]\!]$ vérifie $[\![S]\!](1, y, z) = y$ et $[\![S]\!](0, y, z) = z$.

Puisque $\neg x$ correspond alors à $[\![S]\!](x, 1, 0)$, $\vee(x, y)$ alors à $[\![S]\!](x, y, 1)$ et $\wedge(x, y)$ alors à $[\![S]\!](x, 0, y)$, cela permet d'affirmer que dans la structure toute combinaison booléenne de termes booléens s'exprime par un terme.

2.9.5 Isomorphisme de structures

Comme souvent, on raisonne souvent à isomorphisme près.

Formellement : soient X et Y deux réalisations d'une même signature Σ. Un *isomorphisme* est une fonction bijective i de l'ensemble de base de X vers l'ensemble de base de Y telle que $[\![f]\!](i(x_1), \cdots, i(x_k)) = i(x_0)$ dans Y à chaque fois que $[\![f]\!](x_1, \cdots, x_k) = x_0$ dans X, pour chaque symbole de fonction ou de prédicat f. Ici k est l'arité de f.

2.10 Notes bibliographiques

Ce chapitre ne contient que des choses très standards en mathématique, ou en logique mathématique. Pour les arbres et les graphes, on ne peut que conseiller l'excellent cours INF421 de l'école polytechnique. Pour la partie logique, le texte de ce chapitre est en fait très fortement inspiré de leurs présentations dans [Cori and Lascar, 1993], [Dowek, 2008], [Gurevich, 2000]ou encore dans [Poizat, 1995]

Chapitre 3

Quelques algorithmes séquentiels

En guise d'apéritif, nous présentons quelques algorithmes intéressants, et certaines de leurs propriétés. Cela nous permet de discuter des questions liées à d'une part ce qu'est un algorithme, et sur quel type de structures ils fonctionnent. D'autre part, cela nous permet de discuter des questions liées à la terminaison des algorithmes, et à leur complexité en temps.

Nous en profiterons finalement pour formaliser et clarifier un peu la façon dont nous écrivons les algorithmes.

3.1 Quelques exemples

3.1.1 L'algorithme d'Euclide

L'algorithme d'Euclide permet de calculer le plus grand commun diviseur, le *pgcd*, de deux entiers a et b.

Il se base sur le principe que le *pgcd* de a et de b est aussi le *pgcd* de b et de $a \ mod \ b$. Puisque $a \ mod \ b$ est toujours plus petit que b, la suite des restes décroît, et finit par atteindre 0. Dans ce cas, le dernier reste est le *pgcd*.

Autrement dit, l'algorithme s'écrit :

```
read < a, b >
repeat
  if b = 0 then out ←a
  else par
          a ← b
          b ← a mod b
       endpar
until out!=undef
write out
```

Ici, on note **par** $\cdots$ **endpar** l'exécution en parallèle, c'est-à-dire simultanément : on exécute les deux instructions a $\leftarrow$ b et b $\leftarrow$ a mod b en parallèle, ce qui évite d'utiliser une variable intermédiaire pour effectuer l'opération. On suppose (ici et dans toute la suite), que **out** est une variable particulière, qui vaut initialement la valeur **undef** (qui signifie "non défini").

Cet algorithme fonctionne si a et b sont des entiers, mais en fait aussi sur n'importe quel anneau euclidien : par exemple, si a et b sont des polynômes de $\mathbb{K}[X]$, où $\mathbb{K}$ est un corps.

Formellement, un anneau euclidien A est un anneau commutatif intègre muni d'une division euclidienne : pour tout $a, b \in A$, il existe $q, r \in A$ tel que $a = bq+r$, et il existe d'autre part une fonction ν de $A - \{0\} \to \mathbb{N}$ telle que soit $r = 0$ ou $\nu(r) < \nu(b)$.

Pour les entiers relatifs, ν peut être pris comme la valeur absolue. Pour $\mathbb{K}[X]$, on peut prendre ν comme la fonction qui associe à un polynôme son degré. Les anneaux euclidiens incluent aussi par exemple les entiers de Gauss, c'est-à-dire $\mathbb{Z}[i]$ avec $\nu(a + ib) = a^2 + b^2$, avec une fonction ν adéquate.

L'existence de la fonction ν garantit la terminaison de l'algorithme, puisqu'elle décroît à chaque étape.

3.1.2 L'algorithme de Syracuse

L'algorithme précédent termine en raison de l'existence d'une fonction ν que l'on sait prouver décroissante.

L'algorithme suivant est plus curieux, car il termine, sans que l'on ne sache vraiment pourquoi.

```
read n
repeat
  if n = 1 then out←1
  else if n mod 2=0 then n←n div 2
  else n← 3∗n+1
until out!=undef
write out
```

En fait, à ce jour, on a trouvé aucun entier n pour lequel il ne termine pas, mais on ne connaît pas non plus aucune preuve du fait que l'algorithme termine pour tout n.

Bien entendu, s'il termine, c'est avec la réponse 1. Mais rien ne permet simplement d'affirmer qu'il doit nécessairement terminer.

C'est le problème de la suite de Syracuse, discuté pour la première fois par Collatz en 1928, qui a mené à de nombreux travaux de recherche dans les années 1960.

3.1.3 L'ensemble de Mandelbrot

Considérons l'algorithme suivant qui fonctionne sur le corps des complexes : c et z sont des complexes.

```
read c
z ← 0
repeat
  if |z|>2 then out ← true
  else     z ← z^2 + c
until out!=undef
write out
```

L'algorithme termine pour certains c, et ne termine pas pour d'autre. L'ensemble des c pour lequel il termine correspond à *l'ensemble de Mandelbrot*.

Il s'agit d'un ensemble fractal que l'on peut sait prouver compact, connexe, d'intérieur non vide. Il est conjecturé qu'il est localement connexe.

Cet algorithme montre que l'ensemble de Mandelbrot est *récursivement énumérable* sur les complexes : on sait construire un algorithme qui termine pour c dans l'ensemble, mais qui ne termine pas pour c dans son complémentaire.

En fait, on sait prouver par des arguments de topologie qu'il est indécidable, c'est-à-dire :

Théorème 3.1 *Il n'existe aucun algorithme sur les complexes (avec la signature, c'est-à-dire les opérations de l'algorithme précédent) qui termine toujours, et qui répond vrai pour c dans l'ensemble de Mandelbrot, et faux pour c dans son complémentaire.*

C'est équivalent au résultat suivant :

Théorème 3.2 *Le complémentaire de l'ensemble de Mandelbrot n'est pas récursivement énumérable sur les complexes.*

3.1.4 Algorithme de Bissection

Considérons une fonction $f : [0, 1] \to \mathbb{R}$ telle que $f(0) < 0$ et $f(1) > 0$. Si la fonction est continue, le théorème des valeurs intermédiaires permet d'affirmer qu'il existe une valeur x^* telle que $f(x^*) = 0$.

Comment peut-on approcher cette valeur ? L'algorithme suivant en retourne une valeur approchée par bissection à la précision e près.

```
read e
par
 a ←0
 b ←1
endpar
repeat
  let m = (a+b)/2 in
  if |f(m)| < e then out ← m
  else if f(m)>0 then b ← m
  else a ← m
```

```
until out!=undef
write out
```

Ici **let** var ← expr instruction consiste à remplacer syntaxiquement dans instruction chaque occurrence de var par expr.

3.1.5 Le problème du sac à dos réel

On se donne des poids d'objets en kg, et l'on veut savoir si l'on peut remplir exactement un sac qui supporte un total de 1 kg.

Formellement, on se donne des réels $x_1, x_2, \cdots, x_n \in \mathbb{R}$ et on souhaite décider s'il existe un sous ensemble $S \subset \{1, 2, \cdots, n\}$ tel que $\sum_{i \in S} x_i = 1$

```
read < n, x >
j ← 1
repeat
   [sous-routine d'énumération: avec l'entrée (n, j),
   produire b^j la jème suite de {0,1}^n ]
   b ← b^j
   if Σ_{i∈b} x_i = 1 then out ←vrai
   else if j = 2^n then out ←faux
   else j ← j +1
until out!=undef
write out
```

Ici la notation [$\cdots$ description $\cdots$] désigne une sous-procédure : du code laissé au lecteur qui a pour effet d'effectuer l'opération indiquée par description.

Cet algorithme nécessite dans le pire cas $\mathcal{O}(2^n)$ opérations, et utilise des opérations qui consistent en des additions, et des tests d'égalité : il s'agit d'un algorithme sur la structure $(\mathbb{R}, \mathbf{0}, \mathbf{1}, +, -, =)$.

En fait, on peut prouver que sur cette structure, on ne peut pas faire mieux :

Théorème 3.3 *Il n'existe pas d'algorithme sur la structure* $(\mathbb{R}, \mathbf{0}, \mathbf{1}, +, -, =)$ *qui résout le problème en temps polynomial.*

3.1.6 Équations polynomiales à une inconnue

Si l'on se donne un polynôme P de degré 2 de $\mathbb{R}[X]$, il est de la forme $P(X) = aX^2 + bX + c$. On enseigne au lycée, comment déterminer si un tel polynôme possède ou non une racine réelle, et même à calculer explicitement les solutions dans ce cas, en utilisant des formules qui font intervenir des radicaux.

Cela se généralise très mal à des degrés supérieurs, si l'on veut explicitement les solutions : il est connu (voir un cours sur la théorie de Galois) que les solutions ne sont pas représentables par radicaux dans le cas général pour des polynômes de degré 5.

Mais si seulement l'existence de racines réelles nous intéresse, la situation est plus simple. L'algorithme suivant fonctionne pour les polynômes de degré 2.

```
read < a, b, c >
repeat
  if b^2 − 4 ∗ a ∗ c > 0 then out ←vrai
  else out ← faux
until out!=undef
write out
```

On peut en construire un qui prend un polynôme de degré arbitraire et qui détermine s'il possède une racine réelle ou non :

Théorème 3.4 *Il existe un algorithme qui détermine si un polynôme P de $\mathbb{R}[X]$ de degré arbitraire possède ou non une racine réelle.*

Passons maintenant à plusieurs inconnues. Commençons par le cas simple : des équations linéaires.

3.1.7 Équations linéaires à n-inconnues : l'élimination de Gauss

L'algorithme de Gauss est un grand classique pour résoudre les systèmes d'équations linéaires.

On va se limiter à l'existence de solutions : on se donne un tableau $A[i,j]$ et un tableau $B[i]$ de rationnels, avec $1 \leq i \leq n$, $1 \leq j \leq m$, et l'on veut savoir si le système

$$\begin{cases} A[1,1]x_1 + \cdots A[1,m]x_m &= B[1] \\ \cdots \\ A[n,1]x_1 + \cdots A[n,m]x_m &= B[n] \end{cases}$$

admet une solution rationnelle.

L'algorithme de Gauss permet de répondre à la question :

```
read < A, B >
i ← 1
j ← 1
repeat
  while (i ≤ m and j ≤ n) do
    // Trouver le pivot dans la colonne j,
    //          en partant de la ligne i:
     maxi ← i
     for k ← i+1 to m do
       if abs(A[k,j]) > abs(A[maxi,j]) then maxi ← k
     endfor
     if A[maxi,j] ≠ 0 then
       [ échanger ligne i et maxi de A et B]
       // A[i,j] contient l'ancienne valeur de A[maxi,j].
       [ diviser chaque coeff. de la ligne i de A et B
         par A[i,j] ]
```

```
        // A[i,j] a maintenant la valeur 1.
        for u ← i+1 to m do
          [ soustraire A[u,j]* ligne i de la ligne u de A
            et B]
          // A[u,j] est maintenant 0
        endfor
      i ← i + 1
    endif
    else out ← faux
    j ← j + 1
endwhile
out ← vrai
until out!=undef
write out
```

Cet algorithme fonctionne sur les rationnels, sur les réels, les complexes, et en fait sur tout corps $\mathbb{K}$.

L'implémentation plus haut fonctionne en temps $\mathcal{O}(n^3)$. On sait faire mieux : Strassen a proposé un algorithme en temps de l'ordre de $O(n^{2.81})$. Le meilleur algorithme connu fonctionne en temps $\mathcal{O}(n^{2.376})$.

Lorsque l'on travaille sur un corps fini, comme $\mathbb{Z}_2$, il y a un autre algorithme, que l'on pourrait qualifier de brutal : essayer toutes les valeurs possibles pour les variables $x_1, \cdots, x_m$, (il y en a 2^n), et regarder si cela correspond à une solution. Cet algorithme de complexité $\mathcal{O}(2^n)$ est nettement moins efficace (en temps) que celui écrit plus haut.

Sur un corps qui n'est pas fini (c'est-à-dire qui n'a pas un nombre fini d'éléments), cet algorithme brutal ne fonctionne pas, car on ne peut pas énumérer toutes les valeurs possibles.

3.1.8 Équations polynomiales à n-inconnues

Si l'on se donne un système d'équations polynomiales cette fois, la situation devient plus délicate.

On se donne un ensemble de n-polynômes $P_1, \cdots, P_n$ de $\mathbb{K}[X_1, \cdots, X_m]$, et l'on veut savoir si le système $P_1, \cdots, P_n$ possède une solution, c'est-à-dire si $P_1, \cdots, P_n$ possèdent une racine commune dans $\mathbb{K}^m$.

Sur les réels, la situation est la suivante.

Théorème 3.5 *Il existe un algorithme exponentiel (en* $\max(n, m)$*) pour résoudre ce problème sur le corps* $\mathbb{R}$ *des réels. Aucun algorithme polynomial n'est connu.*

En fait, ce problème est $\mathrm{NP}_{\mathbb{R}}$ complet pour la structure $(\mathbb{R}, \mathbf{0}, \mathbf{1}, +, -, *, /, =, <)$. En d'autres termes, produire un algorithme polynomial (en $\max n, m$) reviendrait à démontrer que $P_{\mathbb{R}} = \mathrm{NP}_{\mathbb{R}}$, c'est-à-dire à montrer que $P = NP$ sur cette structure. Cette question est à priori distincte de la question $P = NP$ classique discutée dans la plupart des ouvrages et des cours.

Sur le corps fini $\mathbb{Z}_2$, la situation est la suivante.

Théorème 3.6 *Il existe un algorithme exponentiel pour résoudre ce problème sur le corps $\mathbb{Z}_2$. Il en existe un algorithme polynomial si et seulement si* P = NP.

En fait, l'algorithme exponentiel consiste en l'algorithme brutal déjà évoqué pour les systèmes linéaires : on essaye pour tous les m-uplets de valeurs.

Le deuxième résultat vient du fait que le problème sur $\mathbb{Z}_2$ est NP-complet (au sens classique usuel).

3.2 Une discussion de cosmétique

3.2.1 Notre écriture des algorithmes

Les algorithmes précédents sont tous des algorithmes qui prennent en entrée (le codage de) certaines entrées et qui produisent en sortie un résultat, codé dans une variable particulière que nous notons **out**. Celle-ci est initialisée à la valeur **undef**, qui signifie "non défini". Dès que la variable **out** est affectée, l'algorithme s'arrête.

Autrement dit, nos algorithmes sont de la forme :

```
read <donnée>
<initialisations>
repeat
  <instructions>
until out!=undef
write out
```

Puisque dans ce document, nous considérons des algorithmes qui ne sont pas réactifs (qui n'interagissent pas avec leur environnement si ce n'est lors de la toute première instruction **read**, et lors de leur affichage final du résultat), cela n'est pas restrictif.

Dans cette écriture, plus précisément :

< initialisations > code en fait l'état initial des variables.

Dans <instructions> nous avons déjà utilisé la construction **let** var $\leftarrow$ expr **in** instruction qui est du pur *sucre syntaxique* : c'est simplement un moyen commode de noter le résultat du remplacement dans instruction de chaque occurrence de var par expr.

Nous avons déjà aussi discuté de la notation [$\cdots$ <description> $\cdots$] qui consiste à laisser au lecteur le soin de remplir les détails, de façon à produire un code qui réalise ce qui est décrit.

3.2.2 États élémentaires

En fait, si l'on veut parler de nombre d'étapes, c'est-à-dire de temps d'exécution, il faut préciser un peu ce que l'on appelle *étape*.

Clairement, on s'attend à ce que le code

```
for k ← 1 to m do
  <instruction>
endfor
```

ne corresponde pas à une étape, mais plutôt à m étapes (en fait $\mathcal{O}(m)$ étapes serait aussi acceptable), alors travaillons un tout petit peu.

Instruction par

On a déjà utilisé, et on utilisera la syntaxe **par** <instruction1> $\cdots$ <instructionk> **endpar** pour indiquer que les k instructions sont effectuées en parallèle.

L'intérêt pratique de cette opération est de pouvoir faire par exemple des échanges de variables sans passer par des variables intermédiaires :

Par exemple, échanger a et b correspond à

```
par
  a ← b
  b ← a
endpar
```

On notera aussi $\{< instruction1 >< instruction2 >\}$ pour
par <instruction1> <instruction2> **endpar**

Séquence

En fait, on utilise un principe implicite dans nos programmes, et dans de nombreux langages de programmation : le fait que deux instructions qui se suivent doivent s'effectuer l'une après l'autre.

Autrement dit, le fait que lorsque l'on écrit

```
<instruction>
<instruction'>
```

on effectue <instruction> puis <instruction'>.

A un moment donné, l'algorithme peut effectuer l'une des instructions, mais pas les deux : ce qui est implicite est que dans une machine séquentielle, est qu'il y a quelque part une variable, que l'on notera *ctl_state* qui code quelle est l'instruction courante.

Pour cela on va supposer qu'il y a toujours une variable *ctl_state*. On suppose par ailleurs que l'on numérote les instructions de façon unique : autrement dit, un programme devient une suite $\pi_1, \pi_2, \cdots, \pi_k$ d'instructions. π_i est l'instruction numéro i. On indiquera parfois explicitement les indices à gauche comme dans :

```
1 <instruction>
2 <instruction'>
3 ...
```

En fait, le code complet correspondant au code précédent est :

```
1 if ctl_state = 1 then {<instruction> ctl_state ←2}
2 else if ctl_state = 2 then {<instruction '> ctl_state ←3}
3 ...
```

En d'autres termes, chaque ligne <instruction> de numéro i est en fait FSM(i,<instruction>,i+1) où FSM(i,<instruction>,j) désigne l'instruction **if** *ctl_state*=i **then** {<instruction> *ctl_state* ← j}.

Nous utiliserons la construction **seq** ...**endseq** dans la suite, quand nous voulons insister sur le fait qu'il existe implicitement une variable *ctl_state* qui est mise à jour en chaque instruction en *ctl* ← *ctl_state* + 1, sauf pour les instructions qui la modifient explicitement. Nous noterons ainsi

```
seq
 <instruction>
 <instruction '>
endseq
```

pour le code précédent.

Boucle for

En fait, fort des remarques précédentes, les boucles **for** peuvent être éliminées dans chacun de nos codes.

Par exemple, on peut voir

```
1 for k ← 1 to m do
2   <instruction>
3   ...
4 endfor
5 ...
```

comme un synonyme de

```
if ctl_state = 1 then {k ←1 ctl_state ←2}
else if ctl_state = 2 and (k ≤m) then
   {<instruction> ctl_state ←3}
...
else if ctl_state = 2 and (k>m) then ctl_state ←5
else if ctl_state =4 then {k ←k+1 ctl_state ←2}
...
```

Boucle while

De même

```
1 while <condition>
2   <instruction>
```

```
3   ...
4 endwhile
5 ...
```

est un synonyme de

```
1 if ctl_state = 1 and not <condition> {ctl_state ←5}
2 else if ctl_state = 1 then
3    {<instruction>  ctl_state ←2}
4 ...
```

Résultat final

Rappelons que les instructions sont englobées dans une boucle de type

```
repeat
  <instructions>
until out!=undef
```

Si l'on fait cette manipulation systématiquement sur le code <instructions>, on élimine toutes les instructions **while** et **for** dans nos codes, et aussi toutes les séquences d'instructions, remplacées par des suites de **if then else** et de **par endpar** imbriquées.

D'autre part, le nombre d'itérations de la la boucle englobante

```
repeat
  <instructions>
until out!=undef
```

correspondra bien alors au nombre d'étapes de <instructions>, c'est-à-dire au temps de l'algorithme, dans le sens où l'on l'entend intuitivement (à une constante multiplicative près), pour le programme obtenu.

Remarque 3.1 *On voit que chacun des algorithmes précédents peut en fait s'écrire uniquement avec des* ***if then else***, ***par endpar*** *et des affectations . Nous verrons dans le chapitre suivant, que c'est le cas de tout algorithme. Nous verrons en outre, que tout algorithme peut s'écrire sous une forme normalisée.*

3.3 Notes bibliographiques

Les exemples de ce chapitre sont inspirés de ceux de [Gurevich, 2000] (pour Euclide), de [Wolper, 2001] (pour Syracuse), de [Blum et al., 1998] et leur reformulation dans [Arora and Barak, 2009] (pour Mandelbrot et le problème du sac à dos), d'un exposé de Gurevitch (pour Bissection), de [Poizat, 1995] (pour les équations polynomiales et linéaires), ainsi que de *Wikipedia* (en particulier pour le code de l'élimination de Gauss).

La section relative à la réécriture des algorithmes est un peu inspirée de [Börger and Stärk, 2003], qui utilise un langage graphique, peut être plus clair que nos discussions.

Chapitre 4

Qu'est-ce qu'un algorithme séquentiel ?

4.1 Introduction

Nous allons tenter dans ce chapitre de définir formellement la notion d'algorithme. Nous allons pour cela utiliser les idées de [Gurevich, 2000].

Nous nous restreignons dans ce chapitre aux algorithmes séquentiels (c'est-à-dire sur une seule machine et un seul processeur), non réactifs et déterministes (c'est-à-dire sans interaction avec l'environnement au cours du fonctionnement).

Nous allons énoncer un certain nombre de postulats, que doivent vérifier les algorithmes.

Exemple 4.1 (Algorithme d'Euclide) *L'algorithme d'Euclide, qui nous servira d'illustration tout au long de ce chapitre, correspond à l'algorithme suivant :*

```
read < a, b >
repeat
  if b = 0 then out ← a
  else par
          a ← b
          b ← a mod b
       endpar
until out!=undef
write out
```

où **par** $\cdots$ **endpar** *désigne la mise en parallèle.*

4.2 Premier postulat

Le premier postulat affirme qu'un algorithme correspond à une suite d'états, et à des transitions entre ces états : l'idée est que les états correspondent à des descriptions instantanées de l'algorithme.

Postulat 1 (Temps séquentiel) *A tout algorithme (non-déterministe séquentiel) A est associé un* système de transitions*, c'est-à-dire,*

- *un ensemble $S(A)$, dont les éléments sont appelés des* états *(parfois* configurations*),*
- *un sous-ensemble $I(A) \subset S(A)$ dont les éléments sont dits* initiaux*,*
- *et une application $\tau_A : S(A) \to S(A)$ que l'on appelle* évolution en un pas *de A.*

Une *exécution* du système de transition est une suite $X_0, X_1, \cdots, X_n, \cdots$ où X_0 est un état initial ($X_0 \in I(A)$), chaque X_i est un état ($X_i \in S(A)$), et $X_{i+1} = \tau_A(X_i)$ pour tout i. Lorsque X est une configuration, on appelle $\tau_A(X)$ sa *configuration successeur.*

Exemple 4.2 *Dans l'algorithme d'Euclide, à un instant fixé, a, b, et out possèdent une certaine valeur. Le triplet de ces valeurs peut être vu comme un état. Maintenant, les états initiaux, sont ceux au début de l'exécution. L'application τ_A est celle qui décrit l'évolution de l'algorithme : elle associe à l'état à un certain temps à l'état au temps d'après.*

4.3 Second postulat

Il nous faut maintenant dire que les états correspondent à quelque chose que l'on sait décrire en langage mathématique : le bon formalisme pour cela est la notion de structure du premier ordre (voir le chapitre 2)

Postulat 2 (Etats abstraits)

- *Les états de A correspondent à des structures du premier ordre.*
- *Les états sont tous sur la même signature.*
- *La fonction d'évolution en un pas τ_A ne modifie pas l'ensemble de base d'aucun état.*
- *$S(A)$ et $I(A)$ sont clos par isomorphisme. De plus, tout isomorphisme de X vers Y est un isomorphisme de $\tau_A(A)$ vers $\tau_A(Y)$.*

Autrement dit, le premier point dit que les états se décrivent par des structures du premier ordre. La pratique des mathématiques depuis plusieurs siècles montre qu'une situation peut se décrire fidèlement avec le langage des mathématiques, c'est-à-dire avec la logique mathématique. L'expérience montre que la logique du premier ordre (la seule introduite dans ce document) est suffisante pour décrire les algorithmes.

En pratique, on décrit un algorithme en utilisant une signature finie fixée. Le deuxième point permet alors d'exprimer que l'on travaille sur une signature fixée, qui ne varie pas tout au long de l'exécution de l'algorithme.

Entre deux états, la seule chose qui change est l'interprétation de certaines fonctions ou relations, mais pas l'ensemble de base. C'est le sens du troisième point.

Exemple 4.3 *Dans notre algorithme d'Euclide, les états correspondent à des structures du premier ordre sur la signature*

$$\Sigma = (\{a, b, out, mod\}, \{=\})$$

constituée de trois symboles de constantes a, b et out et du symbole de fonction mod. Par définition, une structure sur cette signature correspond au choix d'un ensemble de base (par exemple les entiers), d'un choix pour l'interprétation de la fonction mod (l'opération reste de la division euclidienne sur les entiers), et d'une valeur dans cet ensemble de base pour chacun de ces symboles, c'est-à-dire d'un choix de la valeur de a, b, out, i.e. d'un triplet. Les transitions entre états ne modifient que l'interprétation des valeurs a, b, et out.

Le dernier point, qui peut paraître plus mystérieux, dit en fait simplement que l'on raisonne toujours à isomorphisme près : en fait, c'est naturellement le cas, donc dans un premier temps, on peut oublier cette assertion. Si l'on veut mieux comprendre, plus concrètement, on exprime en fait que l'on travaille à un niveau d'abstraction fixé, complètement décrit par l'algorithme : un algorithme n'est pas autorisé à dépendre de représentations internes de l'algorithme qui ne sont pas décrites par l'algorithme. En langage logique, les détails en lesquels l'algorithme dépend, doivent être exprimés par le *type d'isomorphisme* de l'état.

En pratique, on peut distinguer les symboles de fonctions ou de relations *statiques* de ceux *dynamiques* : un symbole de fonction ou de relation est *dynamique* si son interprétation peut varier d'un état à son successeur. Il est statique dans le cas contraire.

Exemple 4.4 *Dans notre algorithme d'Euclide, les symboles de constantes a, b et out sont dynamiques : la valeur de a, b et de out peut changer d'une étape à la suivante. Le symbole de fonction mod est statique : l'interprétation de la fonction mod reste fixée d'une étape à l'étape suivante.*

4.4 Structures versus mémoire

En fait, en abstrayant autour de l'exemple de l'algorithme d'Euclide, il est très pratique de voir une structure comme une certaine mémoire : si $a \in \Sigma$ est un symbole de constante (c'est-à-dire un symbole de fonction de la signature Σ d'arité 0), on peut voir a comme le nom d'un variable : sa sémantique $[\![a]\!]$ correspond à sa valeur.

Plus généralement : si $f \in \Sigma$ est un symbole de fonction d'arité k, son interprétation $[\![f]\!]$ est une fonction de M^k dans M : on peut voir cette fonction

comme un tableau de dimension k. Aussi, étant donné un k-uplet $\overline{m}$ d'éléments de M, une paire $(f, \overline{m})$ sera appelée *emplacement*. On appelle *contenu* la valeur de $[\![f]\!](\overline{m})$, que l'on notera parfois $[\![(f, \overline{m})]\!]$: on peut voir $(f, \overline{m})$ comme désignant l'élément d'indice $\overline{m}$ du tableau f : son contenu $[\![(f, \overline{m})]\!]$ correspond à sa sémantique $[\![f]\!](\overline{m})$.

4.5 Mises à jour

Dans cet esprit, nous avons aussi besoin d'une notation pour les mises à jour : si $(f, \overline{m})$ est un emplacement, et si b est un élément de M, alors on notera le triplet $(f, \overline{m}, b)$ aussi par $(f, \overline{m}) \leftarrow b$, et on appelle un tel triplet une *mise à jour*. Une mise à jour est dite *triviale* si $[\![f]\!](\overline{m}) = b$. Exécuter une mise à jour $(f, \overline{m}) \leftarrow b$ consiste à changer $[\![f]\!](\overline{m})$ en b.

Deux mises à jour sont contradictoires si elles font référence au même emplacement, mais elles sont distinctes.

Exemple 4.5 *$(f, \overline{m}) \leftarrow \mathbf{1}$ et $(f, \overline{m}) \leftarrow \mathbf{0}$ sont deux mises à jour contradictoires.*

Un ensemble de mises à jour est consistant s'il ne contient pas de mises à jour contradictoires. Pour exécuter un ensemble de mise à jour consistant, on exécute simultanément toutes les mises à jour de l'ensemble. Le résultat de l'exécution de l'ensemble de mises à jour Δ sur la structure X sera noté $X + \Delta$.

Passer d'un état à un autre correspond à faire des mises à jour :

Lemme 4.1 *Si X et Y sont deux structures de même vocabulaire et de même ensemble de base, il existe un unique ensemble de mises à jour non triviales consistantes tel que $Y = X + \Delta$.*

Démonstration: Δ est l'ensemble des mises à jour $(f, \overline{m}) \leftarrow b$, pour tous les f et $\overline{m}$ tels que $[\![f]\!](\overline{m})$ diffèrent dans X et dans Y, avec b choisi comme la valeur de $[\![f]\!](\overline{m})$ dans Y. Notons que l'ensemble Δ obtenu est bien consistant. □

Cet ensemble Δ sera noté $Y - X$. En particulier, lorsque X est un état, on notera

$$\Delta(A, X) = \tau_A(X) - X,$$

c'est-à-dire l'ensemble des mises à jour qui correspondent à passer de l'état X à son successeur.

En fait, cet ensemble est bien défini à isomorphisme près. C'est ce que dit le résultat technique suivant, que l'on peut ignorer dans un premier temps.

Lemme 4.2 *Supposons que i soit un isomorphisme de l'état X de A sur un état Y de A. Étendons i de la façon naturelle pour que son domaine contienne aussi les uplets d'éléments et donc les mises à jour. Alors $\Delta(A, Y) = i(\Delta(A, X))$.*

4.6 Troisième postulat

Jusqu'à maintenant, nous n'avons fait aucune hypothèse sur les transitions entre états. La clé de ce que l'on appelle un algorithme est dans le postulat qui suit : soit T un ensemble de termes sur la signature des états. On dira que deux états (donc deux structures) X et Y coïncident sur T si $[\![t]\!]$ possède la même valeur dans la structure X et dans la structure Y pour tout $t \in T$.

Postulat 3 (Exploration bornée) *Il existe un ensemble fini de termes T sur le vocabulaire de l'algorithme A, tel que $\Delta(A, X) = \Delta(A, Y)$ à chaque fois que les états X et Y de A coïncident sur T.*

Intuitivement, l'algorithme A n'explore qu'un nombre fini de l'état courant pour déterminer le prochain état. T est appelé un *témoin d'exploration bornée.*

Exemple 4.6 *Dans notre algorithme d'Euclide, on peut prendre $T = \{b = \mathbf{0}, a, b, \mathbf{0}, \mathbf{1}, a \bmod b, \textbf{\textit{out}}\}$.*

4.7 Définition formelle d'un d'algorithme

Nous avons tous les ingrédients :

Définition 4.1 *Un algorithme déterministe séquentiel est un objet A qui satisfait les trois postulats.*

Exemple 4.7 *L'algorithme d'Euclide est un algorithme (déterministe séquentiel).*

4.8 Une forme normale pour les algorithmes

En fait, ces discussions permettent de prouver que tout algorithme peut s'écrire sous une certaine forme, que nous appelerons *forme normale.*

4.8.1 Termes critiques

Soit T un témoin d'exploration bornée. Sans perte de généralité, on peut supposer que

1. T est *clos par sous-terme* : si $t_1 \in T$, t_2 est un sous-terme (sous-arbre) de t_1, alors $t_2 \in T$.
2. T contient les termes $\mathbf{0}$ et $\mathbf{1}$.

Un terme de T est dit *un terme critique.* Pour tout état X, les valeurs des éléments des termes critiques de X seront appelées les *éléments critiques* de X.

Une conséquence du postulat 3 est que les modifications sur les états se décrivent par les termes critiques. Les valeurs des emplacements à modifier et leurs nouvelles valeurs correspondent nécessairement à des valeurs critiques :

Lemme 4.3 (Les mises à jour impliquent des éléments critiques) *Soit A un algorithme qui vérifie les trois postulats. Si X est un état de A, et si*

$$(f, (a_1, \cdots, a_m)) \leftarrow a_0$$

est une mise à jour de $\Delta(A, X)$, alors tous les éléments $a_1, \cdots, a_m$ et a_0 sont des éléments critiques de X.

Démonstration: Par contradiction, supposons que l'un des a_i ne soit pas critique. Soit Y une structure isomorphe à l'état X qui s'obtient en remplaçant a_i par un élément frais (c'est-à-dire qui n'était pas déjà dans l'ensemble de base de X) b. Par le postulat 2, Y est un état. On vérifie facilement que $[\![t]\!]$ coïncide sur X et sur Y pour tout terme critique t. Par le choix de T, $\Delta(A, Y)$ doit être égal à $\Delta(A, X)$, et donc il doit contenir la mise à jour $(f, (a_1, \cdots, a_m)) \leftarrow a_0$. Mais a_i n'est pas dans l'ensemble de base de Y. Par le troisième point du postulat 2, a_i n'est pas dans l'ensemble de base de $\tau_A(Y)$ non-plus. Et donc il ne peut pas être dans $\Delta(A, Y) = \tau_A(Y) - Y$. Ce qui mène à une contradiction. □

4.8.2 Affectation

Nous introduisons maintenant un langage de programmation minimal :

Définition 4.2 (Affectation) *Soit Σ une signature. Une* règle de mise à jour *(ou encore une* règle d'affectation*) de vocabulaire Σ est de la forme*

$f(t_1, \cdots, t_k) \leftarrow t_0$

où f est un symbole k-aire de la signature Σ, et $t_0, \cdots, t_k$ sont des termes sur la signature Σ.

Pour exécuter une telle règle R dans l'état X, on calcule la valeur des $[\![t_i]\!]$ et on exécute la mise à jour $(f, ([\![t_1]\!], \cdots, [\![t_k]\!])) \leftarrow [\![t_0]\!]$. Notons cette mise à jour $\Delta(R, X)$.

Le lemme 4.3 permet d'affirmer que pour tout algorithme A, et pour tout état X, chaque mise à jour de $\Delta(A, X)$ s'écrit comme $\Delta(R, X)$ pour une telle règle R : en outre, chacun des $t_0, t_1, \cdots, t_k$ peut être choisi parmi les termes critiques.

4.8.3 Mise en parallèle

$\Delta(A, X)$ est constitué d'un ensemble de règles de mise à jour. Nous avons besoin d'une règle qui permet d'effectuer en parallèle plusieurs règles.

Définition 4.3 (Mise en parallèle) *Soit Σ une signature. Si k est un entier, et $R_1, \cdots, R_k$ sont des règles de vocabulaire Σ, alors*

par
  R_1
  R_2

$\cdots$
R_k
endpar

est une règle R de vocabulaire Σ.

Pour exécuter une telle règle R, on exécute simultanément les règles $R_1, \cdots, R_k$. La règle correspond aux mises à jour $\Delta(R, X) = \Delta(R_1, X) \cup \cdots \cup \Delta(R_k, X)$.

Lemme 4.4 *Pour tout algorithme A sur la signature Σ, et pour tout état X de A, on peut construire une règle R^X de vocabulaire Σ qui corresponde à $\Delta(A, X)$, c'est-à-dire telle que $\Delta(R^X, X) = \Delta(A, X)$. En outre cette règle R^X n'implique que des termes critiques.*

Démonstration: Cette règle correspond à la mise en parallèle des mises à jours de $\Delta(A, X)$. T étant fini, cet ensemble de mise à jour est fini. □

Lemme 4.5 *Soient X et Y deux états qui coïncident sur l'ensemble T des termes critiques. Alors $\Delta(A, Y)$ correspond aussi à la règle R^X, c'est-à-dire $\Delta(R^X, Y) = \Delta(A, Y)$.*

Démonstration: Cela découle du fait que $\Delta(R^X, Y) = \Delta(R^X, X) = \Delta(A, X) = \Delta(A, Y)$. La première égalité découle du fait que R^X n'implique que des termes critiques, et que ceux ci ont même valeur en X et Y. La deuxième égalité vient de la définition de R^X. La troisième du postulat 3. □

4.8.4 Construction "si . . .alors"

Définition 4.4 (Si alors) *Soit Σ une signature. Si ϕ est un terme booléen sur la signature Σ, et si R_1 est une règle de vocabulaire Σ, alors*

if ϕ ***then*** R_1

est une règle de vocabulaire Σ.

Pour exécuter une telle règle, on exécute la règle R_1 si l'interprétation de ϕ dans la structure est *vrai*, et on ne fait rien sinon.

4.8.5 Forme normale

En chaque état X, l'égalité entre éléments critiques induit une relation d'équivalence : deux termes critiques t_1 et t_2 sont équivalents, noté $E_X(t_1, t_2)$ si et seulement si $[\![t_1]\!] = [\![t_2]\!]$ dans X. On dit que deux états X et Y sont *T-similaires* si $E_X = E_Y$.

En fait, on peut prouver le résultat suivant, qui dit que seule la classe de T-similarité compte :

Lemme 4.6 $\Delta(R^X, Y) = \Delta(A, Y)$ *pour tout état Y T-similaire à X.*

Démonstration: Supposons que X et Y soient des états, et que $\Delta(R^X, Z) = \Delta(A, Z)$ pour un état Z isomorphe à Y. Alors $\Delta(R^X, Y) = \Delta(A, Y)$.

En effet, soit i un isomorphisme de Y sur un Z approprié. i peut s'étendre naturellement aux uplets, et emplacements et mises à jour. On vérifie facilement que $i(\Delta(R^X, Y)) = \Delta(R^X, Z)$. Par le choix de Z, $\Delta(R^X, Z) = \Delta(A, Z)$. Par le lemme 4.2, $\Delta(A, Z) = i(\Delta(A, Y))$. Alors $i(\Delta(R^X, Y)) = i(\Delta(A, Y))$. Il suffit alors d'appliquer l'inverse de i à chacun des cotés de l'égalité.

Il suffit donc de construire un Z isomorphe à Y avec $\Delta(R^X, Z) = \Delta(A, Z)$.

Si Y est disjoint de X, considérons Z comme la structure isomorphe à Y obtenue en remplaçant $[\![t]\!]$ dans Y par $[\![t]\!]$ dans X pour tout terme critique t. Par le postulat 2, Z est un état. Puisque X et Z coïncident sur T, on a $\Delta(R^X, Z) = \Delta(A, Z)$.

Dans le cas général, remplaçons chaque élément de Y qui est aussi dans X par un élément frais. Cela donne une structure Z isomorphe à Y mais différente de X. Par le postulat 2, Z est un état. Puisque Z est isomorphe à Y, il est T-similaire à Y, et donc T-similaire à X. En raison du cas précédent, $\Delta(R^X, Z) = \Delta(A, Z)$. □

Pour tout état X, il existe un terme [1] ϕ^X qui vaut *vrai* dans une structure Y si et seulement si Y est T-similaire à X : ce terme indique que la relation d'égalité entre les termes critiques définit exactement la relation d'équivalence E_X.

Exemple 4.8 *Sur l'algorithme d'Euclide, les classes d'équivalence distinguent les cas* $b = \mathbf{0}$ *vrai de* $b = \mathbf{0}$ *faux (et en fait beaucoup plus de choses, puisqu'elles distinguent aussi l'égalité/inégalité entre tous les termes critiques : par exemple, ils distinguent aussi le cas* $out = \mathbf{0}$*, de sa négation.).*

En pratique, il est utile uniquement de considérer les termes de T qui sont dynamiques, puisque les termes qui impliquent des symboles statiques ne changent pas d'état à son successeur. Cela permet de réduire le nombre de classes d'équivalence pertinentes.

Puisque T est fini, il y a un nombre fini de classes d'équivalences. Il y a donc un nombre fini d'états $X_1, X_2, \cdots, X_m$ tel que tout état est T-similaire à l'un des états X_i.

Rappelons que nos algorithmes sont de la forme :

```
read <donnée>
<initialisations>
repeat
  <instructions>
until out!=undef
write out
```

On obtient le résultat suivant.

Théorème 4.1 (Forme normale d'un algorithme) *Soit* Σ *une signature. Tout algorithme séquentiel déterministe A sur la signature* Σ *s'écrit avec <instructions> réduit à une règle de vocabulaire* Σ *de la forme*

```
par
    if φ^X1 then R^X1
```

[1] On utilise ici le fait qu'une combinaison booléenne de termes booléens correspond à un terme : voir les conventions fixées dans le chapitre 2, et la discussion de la section 2.

if ϕ^{X_2} ***then*** R^{X_2}
$\cdots$
if ϕ^{X_m} ***then*** R^{X_m}
endpar

chaque règle R^{X_i} *étant de la forme* ***par*** $<aff_1> \cdots <aff_{i_k}>$ *end* ***par****, où chaque instruction* $<aff_j>$ *est une instruction de mise à jour (affectation).*

Autrement dit, tout algorithme peut toujours s'écrire avec des mises à jours, des mises en parallèle, et des si alors.

Exemple 4.9 *L'algorithme d'Euclide peut aussi s'écrire*

read $< a, b >$
repeat
par
if b=**0** ***then*** $d \leftarrow a$
if $b \neq$**0** ***then***
par
$a \leftarrow b$
$b \leftarrow a \ mod \ b$
endpar
endpar
until out!=undef
write out

Remarque 4.1 *Le programme de cet exemple n'est pas exactement la forme normale du théorème précédent pour l'algorithme d'Euclide : la mise sous forme normale obtenue par le théorème précédent comporte en fait beaucoup plus de cas, car on distingue l'égalité/inégalité entre tous les termes critiques (par exemple distingue* $out = \mathbf{0}$ *de* $d \neq \mathbf{0}$*, etc ...). Il s'agit ici d'une forme "réduite" aux tests pertinents.*

4.9 Notion d'algorithme sur une structure $\mathfrak{M}$

4.9.1 Terminologie

Dans ce document, nous utiliserons la convention suivante :

Définition 4.5 *Nous appelerons* algorithme sur

$$\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$$

un algorithme sur la signature de $\mathfrak{M}$ *ou sur une signature étendue*

$$(M, f_1, \cdots, f_u, f'_1, \cdots, f'_\ell, r_1, \cdots, r_v),$$

où $f'_1, \cdots, f'_\ell$ sont des symboles de fonctions dynamiques, qui travaille[2] *sur le domaine M.*

En particulier, lorsque l'on parlera d'algorithme sur une structure

$$\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v),$$

on ne mentionnera en général dans l'écriture de $\mathfrak{M}$ que les symboles de fonctions et de relations statiques, étant donné qu'implicitement on s'autorise des symboles de fonctions dynamiques.

Rappelons qu'un symbole de fonction est dynamique si son interprétation peut varier d'un état à son sucesseur, par opposition à statique.

Exemple 4.10 *L'algorithme d'Euclide sur les réels sera appelé un algorithme sur la structure $(\mathbb{R}, mod)$, car il est en fait sur la signature $\Sigma = (\{a, b, out, mod\}, \{=\})$.*

4.9.2 Conventions

Pour éviter plusieurs problèmes techniques, on se limitera aux algorithmes qui respectent les conventions suivantes :

- La signature possède un symbole de constante **undef** dont l'intepétation sera notée (par abus de langage) **undef**. On dira qu'un emplacement $(f, \overline{m})$ est *utile*, si f est un symbole de fonction dynamique, et son contenu $[\![(f, \overline{m})]\!]$ n'est pas **undef**.
- L'état initial d'un algorithme respecte la propriété suivante : il possède qu'un nombre fini d'emplacements utiles.

Un état sera dit *terminal* s'il correspond à un état d'arrêt de l'algorithme, c'est-à-dire, un état tel que l'interprétation de **out** n'est pas **undef** pour les algorithmes considérés ici. On dira qu'un algorithme *termine* sur un certain état initial, ou sur une entrée, si l'exécution correspondante mène à (contient) un état terminal.

4.10 Notes bibliographiques

Le contenu de ce chapitre est basé sur l'axiomatisation et les résultats de [Gurevich, 2000]. Nous avons aussi parfois utilisé leur présentation dans le livre [Börger and Stärk, 2003].

[2] Formellement, l'ensemble de base de chacune des états est M.

Chapitre 5

Quelques modèles séquentiels, et leur équivalence

Nous avons maintenant défini ce qu'était un algorithme. La notion d'algorithme au sens précédent est de très haut niveau par rapport à la notion basée sur les machines de Turing, ou sur d'autres langages bas niveau, comme celle qui est utilisée dans la plupart des ouvrages de calculabilité ou de complexité.

Nous ne pouvons éviter de montrer que notre notion d'algorithme correspond à la notion usuelle. L'objet de ce chapitre est de définir quelques modèles et de montrer qu'ils sont équivalents à notre notion d'algorithme.

5.1 Machines de Turing

Revenons sur le modèle classique de la machine de Turing.

5.1.1 Description

Une machine de Turing (déterministe) à k rubans est composée des éléments suivants :

1. Une mémoire infinie sous forme de k rubans. Chaque ruban est divisé en cases. Chaque case peut contenir un élément d'un ensemble M (qui se veut un alphabet). On suppose dans cette section que l'alphabet M est un ensemble fini.
2. *k têtes de lecture* : chaque tête se déplace sur l'un des k rubans.
3. Un programme donné par une *fonction de transition* qui pour chaque état de la machine, précise selon le symbole sous l'une des têtes de lecture,
   (a) l'état suivant ;

(b) le nouvel élément de M à écrire à la place de l'élément de M sous la tête de lecture d'un des rubans ;

(c) un sens de déplacement pour la tête de lecture de ce dernier ruban.

L'exécution d'une machine de Turing peut alors se décrire comme suit : initialement, l'entrée se trouve sur le début du premier ruban. Les cases des rubans qui ne correspondent pas à l'entrée contiennent toutes l'élément **B** (symbole de blanc), qui est un élément particulier de M. A chaque étape de l'exécution, la machine, selon son état, lit le symbole se trouvant sous l'une des têtes de lecture, et selon ce symbole :

- remplace le symbole sous une des têtes de lecture par celui précisé par sa fonction transition ;
- déplace (ou non) cette tête de lecture d'une case vers la droite ou vers la gauche suivant le sens précisé par la fonction de transition ;
- change d'état vers un nouvel état.

5.1.2 Formalisation

La notion de machine de Turing se formalise de la façon suivante :

Définition 5.1 (Machine de Turing) Une machine de Turing à k-rubans *est un 8-uplet*

$$M = (Q, \Sigma, \Gamma, \mathbf{B}, \delta, q_0, q_a, q_r)$$

où

1. Q *est l'ensemble fini des états.*
2. Σ *est un alphabet* fini.
3. Γ *est l'alphabet de travail* fini *:* $\Sigma \subset \Gamma$.
4. $\mathbf{B} \in \Gamma$ *est le caractère blanc.*
5. $q_0 \in Q$ *est l'état initial.*
6. $q_a \in Q$ *est l'état d'acceptation.*
7. $q_r \in Q$ *est l'état de rejet (ou d'arrêt).*
8. δ *est la fonction de transition :* δ *est constitué d'une fonction* δ_1 *de* Q *dans* $\{1, 2, \cdots, k\}$*, et d'une fonction* δ_2 *de* $Q \times \Gamma$ *dans* $Q \times \{1, 2, \cdots, k\} \times \Gamma \times \{\leftarrow, |, \rightarrow\}$.

Le langage accepté par une machine de Turing se définit à l'aide des notions de *configurations* et de dérivations entre configurations.

Formellement : *Une configuration* est donnée par la description des rubans, par les positions des têtes de lecture/écriture, et par l'état interne : une configuration peut se noter $C = (q, u_1 \# v_1, \ldots, u_k \# v_k)$, avec $u_1, \ldots, u_n, v_1, \ldots, v_n \in (\Gamma - \{\mathbf{B}\})^*$, $q \in Q$: u_i et v_i désignent le contenu respectivement à gauche et à droite de la tête de lecture du ruban i, la tête de lecture du ruban i étant sur la première lettre de v_i. Pour simplifier la présentation, on fixe la convention que les mots v_i sont écrits de gauche à droite (la lettre numéro $i+1$ de v_i est à

droite de la lettre numéro i) alors que les u_i sont écrits de droite à gauche (la lettre numéro $i+1$ de u_i est à gauche de la lettre numéro i, la première lettre de u_i étant à gauche de la tête de lecture numéro i).

Une telle configuration est dite *acceptante* si $q = q_a$, *rejetante* si $q = q_r$.

Pour $w \in \Sigma^*$, la configuration initiale correspondante à w est la configuration $C[w] = (q_0, \#w, \#, \ldots, \#)$ (u_1 et les u_i et v_i pour $i \neq 2$ correspondent au mot vide).

On note : $C \vdash C'$ si la configuration C' est le successeur direct de la configuration C par le programme (donné par δ) de la machine de Turing. Formellement, si $C = (q, u_1\#v_1, \ldots, u_k\#v_k)$, et si $a_1, \cdots, a_k$ désignent les premières lettres[1] de $v_1, \cdots, v_k$, et si

$$\delta_1(q) = \ell$$

$$\delta_2(q, a_\ell) = (q', \ell', a', m')$$

alors $C \vdash C'$ si

- $C' = (q', u'_1\#v'_1, \cdots, u'_k\#v'_k)$, et
- pour $i \neq \ell'$, $u'_i = u_i$, $v'_i = v_i$
- si $i = \ell'$,
  - si $m' = |$, $u'_i = u_i$, et v'_i est obtenu en remplaçant la première lettre a_i de v_i par a'.
  - si $m' = \leftarrow$, $v'_i = a'v_i$, et u'_i est obtenu en supprimant la première lettre de u_i.
  - si $m' = \rightarrow$, $u'_i = a'u_i$, et v'_i est obtenu en supprimant la première lettre a_i de v_i.

On appelle *calcul de M sur un mot $w \in \Sigma^*$*, une suite de configurations $(C_i)_{i \in N}$ telle que $C_0 = C[w]$ et pour tout i, $C_i \vdash C_{i+1}$.

Le mot w est dit *accepté* si le calcul sur ce mot est tel qu'il existe un entier t avec C_t acceptante. On dit dans ce cas que w est accepté *en temps t*. Le mot w est dit *refusé* si le calcul sur ce mot est tel qu'il existe un entier t avec C_t rejetante. On dit dans ce cas que w est refusé *en temps t*.

Un langage $L \subset \Sigma^*$ est *dit accepté par M* si pour tout $w \in \Sigma^*$, $w \in L$ si et seulement si w est accepté.

Un langage $L \subset \Sigma^*$ est *dit décidé par M* si pour tout $w \in \Sigma^*$, $w \in L$ si et seulement si w est accepté, et $w \notin L$ si et seulement si w est rejeté.

5.1.3 Une machine de Turing est un algorithme

En fait, cela peut aussi se décrire à un plus haut niveau naturellement comme un algorithme.

Proposition 5.1 *Une machine de Turing à k rubans correspond à un programme du type*

```
ctl_state ←q0
repeat
```

[1] Avec la convention que la première lettre du mot vide est le blanc **B**.

```
  par
 <instructions>
  if ctl_state = q_a then out ←vrai
  if ctl_state = q_r then out ←faux
 endpar
until out!=undef
write out
```

où <instructions> est une suite finie d'instructions du type

```
if ctl_state=q and tape_ℓ(head_ℓ) = a then
par
  tape_ℓ'(head_ℓ') ←a'
  head_ℓ' ←head_ℓ' + m
  ctl_state ←q'
endpar
```

où $q \neq q_a$, $q \neq q_r$, $head_1, \cdots, head_k$ sont des symboles de constantes à valeurs entières, qui codent la position de chacune des têtes de lecture, $tape_1, \cdots, tape_k$ sont des symboles de fonctions d'arité 1, qui codent le contenu de chacun des rubans. Chaque règle de ce type code le fait que $\delta_1(q) = \ell$, $\delta_2(q, a) = (q', \ell', a', m')$, avec m qui vaut respectivement $-1, 0$ ou 1 lorsque m' vaut $\leftarrow, |, \rightarrow$ respectivement.

5.2 Machines de Turing sur une structure $\mathfrak{M}$

Autrement dit, une machine qui travaille sur l'alphabet fini

$$\Sigma \subset \Gamma = \{\gamma_1, \gamma_2, \cdots, \gamma_m\},$$

ne sait écrire que des lettres γ_i, et tester si le symbole sous l'une des têtes de lecture est égal à l'une des lettres γ_i (et déplacer ses têtes de lecture).

Autrement dit, les seules opérations en écriture autorisées sont les constantes $\gamma_1, \gamma_2, \cdots, \gamma_m$, et les seuls tests autorisés correspondent à l'égalité à l'un des symboles γ_i : on peut voir cela comme une machine de Turing sur la signature

$$\mathfrak{M} = (\Gamma, \{\gamma_1, \gamma_2, \cdots, \gamma_m\}, \{\gamma_1?, \gamma_2?, \cdots, \gamma_m?, =\}),$$

où γ_i désigne le symbole de constante qui s'interprète par la lettre γ_i, et $\gamma_i?$ désigne le symbole de prédicat d'arité 1 qui est vrai si et seulement si son argument est la lettre γ_i.

En effet, rien ne nous interdit de considérer des machines de Turing (à k rubans) qui travaillent sur une structure arbitraire plus générale. Une machine qui travaille sur la structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$, est autorisée à calculer les fonctions f_i et à tester les relations r_j de la structure. Les cases des rubans contiennent cette fois des éléments de l'ensemble de base M, qui n'est pas nécessairement fini.

5.2.1 Description

La description de ce que l'on appelle une machine de Turing ne change pas, si ce n'est que les rubans contiennent des éléments de M, l'ensemble de base de la structure, possiblement infini, et que la machine est autorisée à effectuer les tests $r_1, \cdots, r_v$ qui correspondent aux relations de la structure, et à effectuer les opérations $f_1, \cdots, f_u$ qui correspondent aux fonctions de la structure.

En d'autres termes, à coup de copier, coller :

Une machine de Turing (déterministe) à k rubans sur la structure

$$\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$$

est composée des éléments suivants.

1. Une mémoire infinie sous forme de k rubans. Chaque ruban est divisé en cases. Chaque case peut contenir un élément de M.
2. *k têtes de lecture* : chaque tête se déplace sur l'un des k rubans.
3. Une *fonction de transition* qui pour chaque état de la machine, et selon le résultat éventuel d'un test de relation de la structure sur l'un des rubans, précise
   (a) l'état suivant ;
   (b) le nouvel élément de M à écrire à la place de l'élément de M sous la tête de lecture d'un des rubans, obtenu comme le résultat de l'application d'une fonction de la structure ;
   (c) un sens de déplacement pour la tête de lecture de ce dernier ruban.

On fixe la convention que l'on lit les arguments d'une relation ou d'une fonction de la structure sur les premières cases à droite de la tête de lecture : si f_i (respectivement r_j) est une fonction (resp. relation) d'arité k, ses arguments sont lus sur les k premiers cases à droite de la tête de lecture.

On fixe aussi les conventions suivantes, de façon à éviter de nombreux problèmes :

1. Les structures considérées contiennent au moins la fonction *id* identité d'arité 1, et la relation *vrai* d'arité 1 qui est toujours vraie : cela permet dans la définition qui suit d'autoriser les machines à recopier le contenu d'une case sur un ruban sur un autre ruban.
2. Les structures considérées contiennent au moins la fonction $\mathbf{0}$ (respectivement : $\mathbf{1}$, $\mathbf{B}$) d'arité 1 qui teste si son argument est $\mathbf{0}$ (resp. $\mathbf{1}$, $\mathbf{B}$) : cela permet de tester si un élément est blanc, par exemple.

Puisque les structures contiennent toujours ces symboles de fonctions et de relations, nous éviterons de les noter explicitement, mais elles sont présentes dans les structures considérées.

5.2.2 Formalisation

Définition 5.2 (Machine de Turing) ***Une machine de Turing à k-rubans*** *sur une structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ est un 8-uplet*

$$(Q, \Sigma, \Gamma, \mathbf{B}, \delta, q_0, q_a, q_r)$$

où

1. *Q est l'ensemble fini des états*
2. *Σ est l'alphabet, et correspond à un sous-ensemble de l'ensemble de base de la structure : $\Sigma \subset M$.*
3. *Γ est l'alphabet de travail, et correspond à un sous-ensemble de l'ensemble de base de la structure : $\Sigma \subset \Gamma \subset M$.*
4. *$\mathbf{B} \in \Gamma$ est le caractère blanc.*
5. *$q_0 \in Q$ est l'état initial*
6. *$q_a \in Q$ est l'état d'acceptation*
7. *$q_r \in Q$ est l'état de rejet (ou arrêt).*
8. *δ est la fonction de transition : δ est constitué d'une fonction δ_1 de Q dans $\{1, 2, \cdots, k\} \times \{1, 2, \cdots, v\}$, et d'une fonction δ_2 de $Q \times \{vrai, faux\}$ dans $Q \times \{1, 2, \cdots, k\} \times \{1, 2, \cdots, u\} \times \{\leftarrow, |, \rightarrow\}$.*

Le langage accepté par une machine de Turing se définit toujours à l'aide des notions de *configurations* et de dérivations entre configurations.

On utilise toujours la même convention pour représenter les configurations, et on laisse inchangée la notion de configuration acceptante ou rejetante. Comme auparavant, les mots u_i et v_i sont des mots sur l'alphabet M, mais cette fois cet alphabet est possiblement infini.

On note : $C \vdash C'$ si la configuration C' est le successeur direct de la configuration C par le programme (donné par δ) de la machine de Turing. Formellement, si $C = (q, u_1 \# v_1, \ldots, u_k \# v_k)$, et si

$$\delta_1(q) = (\ell, j)$$

et l'on pose r comme $vrai$ (respectivement $faux$) ssi $r_j(a_{\ell,1}, a_{\ell,2}, \cdots, a_{\ell,k})$ est vrai (resp. faux), où $a_{\ell,1}, a_{\ell,2}, \cdots, a_{\ell,k}$ désignent les k premières lettres du mot v_ℓ, et k est l'arité du symbole de prédicat r_j, et si

$$\delta_2(q, r) = (q', \ell', i, m'),$$

et si l'on pose a' comme l'interprétation de $f_i(a_{\ell,1}, a_{\ell,2}, \cdots, a_{\ell,k'})$ où les symboles $a_{\ell,1}, a_{\ell,2}, \cdots, a_{\ell,k'}$ désignent les k' premières lettres du mot v_ℓ, et k' l'arité de f_i, alors $C \vdash C'$ si

- $C' = (q', u'_1 \# v'_1, \cdots, u'_k \# v'_k)$, et
- pour $i \neq \ell'$, $u'_i = u_i$, $v'_i = v_i$
- si $i = \ell'$,

- si $m = |$, $u'_i = u_i$, et v'_i est obtenu en remplaçant la première lettre a_i de v_i par a'.
- si $m = \leftarrow$, $v'_i = a'v_i$, et u'_i est obtenu en supprimant la première lettre de u_i.
- si $m = \rightarrow$, $u'_i = u_i a'$, et v'_i est obtenu en supprimant la première lettre a_i de v_i.

On laisse inchangée la notion de calcul, de mot accepté, ou refusé.

5.2.3 Une machine de Turing sur une structure $\mathfrak{M}$ est un algorithme sur $\mathfrak{M}$

Proposition 5.2 *Une machine de Turing à k rubans sur la structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ correspond à un programme du type*

```
ctl_state ←q0
repeat
   par
 <instructions>
  endpar
    if ctl_state = qa then out ←vrai
    if ctl_state = qr then out ←faux
  endpar
until out!=undef
write out
```

où <instructions> est une suite finie d'instructions du type

```
if ctl_state=q and
    rj(tapeℓ(headℓ), tapeℓ(headℓ + 1), ··· , tapeℓ(headℓ + k)) then
par
   tapeℓ′(headℓ′) ←fi(tapeℓ(headℓ), tapeℓ(headℓ + 1), ··· , tapeℓ(headℓ + k′))
   headℓ′ ←headℓ′ + m
   ctl_state ←q′
endpar
```

ou alors

```
if ctl_state=q and
   not rj(tapeℓ(headℓ), tapeℓ(headℓ + 1), ··· , tapeℓ(headℓ + k)) then
par
   tapeℓ′(headℓ′) ←fi(tapeℓ(headℓ), tapeℓ(headℓ + 1), ··· , tapeℓ(headℓ + k′))
   headℓ′ ←headℓ′ + m
   ctl_state ←q′
endpar
```

où $q \neq q_a$, $q \neq q_r$, $head_1, \cdots, head_k$ sont des symboles de constantes à valeur entières qui codent la position de chacune des têtes de lecture, $tape_1, \cdots, tape_k$ sont des symboles de fonctions d'arité 1, qui codent le contenu de chacun des

rubans. Chaque règle de ce type code le fait que $\delta_1(q) = (\ell, j)$, $\delta_2(q, r) = (q', \ell', i, m')$, *avec* m *qui vaut respectivement* $-1, 0$ *ou* 1 *lorsque* m' *vaut* $\leftarrow, |, \rightarrow$ *respectivement.* k *et* k' *désignent les arités de* r_j *et* f_i.

5.3 Robustesse du modèle

5.3.1 Structures finies comme structures arbitraires

Les définitions précédentes sont conçues précisément pour que le résultat suivant soit vrai :

Proposition 5.3 *Une machine de Turing au sens de la section 5.1, avec alphabet fini* $\Sigma \subset \Gamma = \{\gamma_1, \gamma_2, \cdots, \gamma_m\}$ *correspond à (se simule par) une machine de Turing sur la structure*[2]

$$\mathfrak{M} = (\Gamma, \{\gamma_1, \gamma_2, \cdots, \gamma_m\}, \{\gamma_1?, \gamma_2?, \cdots, \gamma_m?, =\}),$$

où γ_i *désigne le symbole de constante qui s'interprète par la lettre* γ_i, *et* $\gamma_i?$ *désigne le symbole de prédicat d'arité* 1 *qui est vrai si et seulement son argument est la lettre* γ_i.

Démonstration: On remplace chaque instruction de la machine de Turing **if** *ctl_state* $= q$ **and** $tape_\ell(head_\ell) = a$ **then par** $tape_{\ell'}(head_{\ell'}) \leftarrow a' \ldots$ par l'instruction **if** *ctl_state*=q **and** $a?(tape_\ell(head_\ell))$ **then par** $tape_{\ell'}(head_{\ell'}) \leftarrow a' \ldots$ □

Puisque tester si le symbole sous l'une des têtes de lecture γ_i peut aussi se faire en recopiant le symbole sous la tête de lecture sur un autre ruban, en écrivant à sa droite le symbole γ_i et en testant l'égalité, quitte à augmenter le nombre de rubans de 1, on peut aussi affirmer :

Proposition 5.4 *Une machine de Turing au sens de la section 5.1, avec alphabet fini* $\Sigma \subset \Gamma = \{\gamma_1, \gamma_2, \cdots, \gamma_m\}$ *correspond à (se simule par) une machine de Turing sur la structure*

$$\mathfrak{M} = (\Gamma, \{\gamma_1, \gamma_2, \cdots, \gamma_m\}, \{=\}).$$

5.3.2 Des structures finies vers les booléens

On appelle *booléens* la structure

$$\mathfrak{B} = (\{0, 1\}, \mathbf{0}, \mathbf{1}, =).$$

[2]Rappelons nos conventions sur les structures considérée : $\mathfrak{M}$ contient aussi d'autres symboles de fonctions et de relations (comme l'identité, la fonction $vrai$, ...) que nous ne précisons pas pour ne pas alourdir les notations.

Théorème 5.1 *Une machine de Turing sur une structure*

$$\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$$

avec M fini (et donc une machine de Turing au sens de la section 5.1) se simule par une machine de Turing sur la structure

$$\mathfrak{B} = (\{0,1\}, \mathbf{0}, \mathbf{1}, =).$$

Principe: Puisque M est fini, on peut coder chaque élément de M par une suite de 0 et de 1 (son écriture en binaire). Il s'agit alors simplement de construire une machine de Turing qui travaille sur ces écritures en binaire. Puisque M est fini, chaque fonction ou relation ne prend qu'un nombre fini de valeurs : la table de chacune des fonctions et relations peut être stockée dans l'ensemble des états finis Q de la machine de Turing, ou si l'on préfère par une suite finie de **if then else** dans l'algorithme. □

Autrement dit, la calculabilité (les fonctions qui sont calculables) sont les même sur toute structure finie.

Si la structure n'est pas finie (c'est-à-dire si son ensemble de base M n'est pas fini), il n'y aucune raison que la notion de fonction calculable sur la structure et sur les booléens coïncide.

5.3.3 Programmer avec des machines de Turing

La programmation avec des machines de Turing relève de programmation extrêmement bas niveau.

Lemme 5.1 *Fixons deux entiers i et j distincts. On peut écrire un sous-programme d'une machine de Turing à $k \geq \max(i,j)$ rubans qui recopie le contenu à droite de la tête de lecture numéro i sur le ruban numéro j.*

Principe: Le programme consiste à déplacer la tête de lecture numéro j à droite jusqu'à rencontrer un blanc. Puis à déplacer à gauche cette tête de lecture en écrivant à chaque étape sur le ruban numéro j le symbole blanc, de façon à complètement effacer ce ruban, et ramener sa tête de lecture tout à gauche. On déplace alors la tête de lecture numéro i et j case par case vers la droite, en copiant le symbole sous la tête numéro i sur le ruban j, jusqu'à rencontrer un blanc sous la tête numéro i. □

Nous laissons en exercice les problèmes suivants :

Exercice 5.1 *Construire une machine de Turing qui ajoute* 1 *au nombre écrit en binaire (donc avec des* **0** *et* **1***) sur son ruban numéro i.*

Exercice 5.2 *Construire une machine de Turing qui soustrait* 1 *au nombre écrit en binaire (donc avec des* **0** *et* **1***) sur son ruban numéro i.*

Exercice 5.3 *Construire une machine de Turing qui accepte les chaînes de caractère avec le même nombre de* **0** *et de* **1**.

5.4 Machines à $k \geq 2$ piles

5.4.1 Sur une structure finie

Une *machine à k piles*, possède un nombre fini k de piles $r_1, r_2, \cdots, r_k$, qui correspondent à des piles d'éléments de M. Les instructions d'une machine à piles permettent seulement d'empiler un symbole sur l'une des piles, tester la valeur du sommet d'une pile, ou dépiler le symbole au sommet d'une pile.

Si l'on préfère, on peut voir une pile d'éléments de M comme un mot w sur l'alphabet M. Empiler le symbole $a \in M$ correspond à remplacer w par aw. Tester la valeur du sommet d'une pile correspond à tester la première lettre du mot w. Dépiler le symbole au sommet de la pile correspond à supprimer la première lettre de w.

Définition 5.3 *Une machine à k-piles correspond à un programme du type*

```
ctl_state ←0
repeat
   seq
 <instructions>
   if ctl_state = q_a then out ←vrai
   if ctl_state = q_r then out ←faux
   endseq
until out!=undef
write out
```

où <instructions> est une suite finie d'instructions de l'un des types suivants

1. $push_i(a)$
2. pop_i
3. **if** $top_i = a$ **then** $ctl\_state := q$ **else** $ctl\_state := q'$

où $i \in \{1, 2, \cdots, k\}$, a est un symbole d'un alphabet fini Γ, q et q' sont des entiers, et $push_i(a)$, pop_i, top_i désignent respectivement empiler le symbole a sur la pile i, dépiler le sommet de la pile i, et le symbole au sommet de la pile i.

Théorème 5.2 *Toute machine de Turing à k rubans au sens de la section 5.1 peut être simulée par une machine à $2k$ piles.*

Démonstration: Selon la formalisation page 48, une configuration d'une machine de Turing correspond à $C = (q, u_1\#v_1, \ldots, u_k\#v_k)$, où u_i et v_i désignent le contenu respectivement à gauche et à droite de la tête de lecture du ruban i. On peut voir u_i et v_i comme des piles. Si l'on relit attentivement la formalisation page 48, on observe que les opérations effectuées par le programme de la machine de Turing pour passer de la configuration C a sa configuration successeur C' correspondent à des opérations qui se codent trivialement par des *push*, *pop*, et *top* : on construit donc une machine à $2k$ piles, chaque pile codant l'un des u_i ou v_i (le contenu à droite et à gauche de chacune des têtes de lecture), et qui simule étape par étape la machine de Turing. □

5.4.2 Sur une structure arbitraire

On peut généraliser les machines à k piles sur une structure arbitraire $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$. Une *machine à k piles*, possède un nombre fini k de piles $r_1, r_2, \cdots, r_k$, qui correspondent à des piles d'éléments de l'ensemble de base M. Les instructions d'une machine à piles permettent seulement d'empiler le résultat sur l'une des piles de l'application d'une fonction f_i de la structure à l'une des piles (c'est-à-dire que si f_i est d'arité k, et si $a_1, \cdots, a_k$ désignent les premières lettres d'une des piles, on empile sur l'une des piles $f_i(a_1, \cdots, a_k)$) , tester la valeur d'une relation r_j d'une pile (c'est-à-dire que si r_j est d'arité k, et si $a_1, \cdots, a_k$ désignent les premières lettres d'une des piles, on teste la valeur de $r_j(a_1, \cdots, a_k)$) ou dépiler le symbole au sommet d'une pile.

Selon le même principe que précédemment :

Théorème 5.3 *Toute machine de Turing à k rubans sur une structure arbitraire $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ peut être simulée par une machines à $2k$ piles sur $\mathfrak{M}$.*

5.5 Cas des structures finies : Machines à compteurs

Nous introduisons maintenant un modèle extrêmement rudimentaire : une *machine à compteurs* possède un nombre fini k de compteurs $r_1, r_2, \cdots, r_k$, qui contiennent des entiers naturels. Les instructions d'une machine à compteur permettent seulement de tester l'égalité d'un des compteurs à 0, d'incrémenter un compteur ou de décrémenter un compteur. Tous les compteurs sont initialement nuls, sauf celui codant l'entrée.

Autrement dit :

Définition 5.4 *Une machine à k-compteurs correspond à un programme du type*

```
ctl_state ←0
repeat
   seq
 <instructions>
   if ctl_state = q_a then out ←vrai
   if ctl_state = q_r then out ←faux
   endseq
until out!=undef
write out
```

où <instructions> est une suite finie d'instructions de l'un des types suivants

1. $x_i \leftarrow x_i + 1$
2. $x_i \leftarrow x_i \ominus 1$
3. **if** $x_i = 0$ **then** $ctl\_state := q$ **else** $ctl\_state := q'$

où $i \in \{1, 2, \cdots, k\}$, q *et* q' *sont des entiers, et* $x \ominus 1$ *vaut* $x - 1$ *si* $x \neq 0$, *et* 0 *pour* $x = 0$. *Chacun des registres* x_i *contient un entier naturel.*

Théorème 5.4 *Toute machine à k-piles sur une structure finie (c'est-à-dire sur un alphabet fini) peut être simulée par une machines à* $k + 1$ *compteurs.*

Démonstration: L'idée est de voir une pile $w = a_1 a_2 \cdots a_n$ sur l'alphabet Σ de cardinalité $r - 1$ comme un entier i en base r : sans perte de généralité, on peut voir Σ comme $\Sigma = \{0, 1, \cdots, r-1\}$. Une pile (un mot) w correspond à l'entier $i = a_n r^{n-1} + a_{n-1} r^{n-2} + \cdots a_2 r + a_1$.

On utilise ainsi un compteur i pour chaque pile. Un $k+1$ème compteur (que l'on appellera *compteur supplémentaire*) est utilisé pour ajuster les compteurs et simuler chaque opération (empilement, dépilement, lecture du sommet) sur l'une des piles.

Dépiler correspond à remplacer i par i *div* r. En partant avec le compteur supplémentaire à 0, on décrémente le compteur i de r (en r étapes) et on incrémente le compteur supplémentaire de 1. On répète cette opération jusqu'à ce que le compteur i atteigne 0. On décrémente alors le compteur supplémentaire de 1 en incrémentant le compteur i de 1 jusqu'à ce que le premier soit 0. A ce moment, on lit bien le résultat correct dans le compteur i.

Empiler le symbole a correspond à remplacer i par $i * r + a$. On multiplie d'abord par r : en partant avec le compteur supplémentaire à 0, on décrémente le compteur i de 1 et on incrémente le compteur supplémentaire de r (en r étapes) jusqu'à ce que le compteur i soit à 0. On décrémente alors le compteur supplémentaire de 1 en incrémentant le compteur i de 1 jusqu'à ce que le premier soit 0. A ce moment, on lit $i * r$ dans le compteur i. On incrémente alors le compteur i de a (en a incrémentations).

Lire le sommet d'une pile i correspond à calculer i *mod* r. En partant avec le compteur supplémentaire à 0, on décrémente le compteur i de 1 et on incrémente le compteur supplémentaire de 1. Lorsque le compteur i atteint 0 on s'arrête. On décrémente alors le compteur supplémentaire de 1 en incrémentant le compteur i de 1 jusqu'à ce que le premier soit 0. On fait chacune de ces opérations en comptant en parallèle modulo r dans l'état interne de la machine (ou si l'on préfère en le codant dans l'instruction courante *ctl_state*). □

Observons que nous nous sommes limités ici aux structures finies, car sur une structure infinie, on ne peut pas coder aussi facilement une pile par un entier.

Théorème 5.5 *Toute machine à* $k \geq 3$ *compteurs se simule par une machine à 2 compteurs.*

Démonstration: Supposons d'abord $k = 3$. L'idée est coder trois compteurs i, j et k par l'entier $m = 2^i 3^j 5^k$. L'un des compteurs stocke cet entier. L'autre compteur est utilisé pour faire des multiplications, divisions, calculs modulo m, pour m valant 2, 3, ou 5.

Pour incrémenter i, j ou k de 1, il suffit de multiplier m par 2, 3 ou 5 en utilisant le principe de la preuve précédente.

Pour tester si i, j ou $k = 0$, il suffit de savoir si m est divisible par 2, 3 ou 5, en utilisant le principe de la preuve précédente.

Pour décrémenter i, j ou k de 1, il suffit de diviser m par 2, 3 ou 5 en utilisant le principe de la preuve précédente.

Pour $k > 3$, on utilise le même principe, mais avec les k premiers nombres premiers au lieu de simplement 2, 3, et 5. □

En combinant les résultats précédents, on obtient :

Corollaire 5.1 *Toute machine de Turing (sur un alphabet fini) se simule par une machine à 2 compteurs.*

Observons que la simulation est particulièrement inefficace : la simulation d'un temps t de la machine de Turing nécessite un temps exponentiel pour la machine à 2 compteurs.

5.6 Machines RAM

5.6.1 Introduction

Les *machines RAM (Random Access Machine)* sont des modèles de calcul qui étendent le modèle rudimentaire des machines à compteur, et qui ressemblent plus aux langages machines actuels. Une machine RAM possède des registres qui contiennent des entiers naturels (nuls si pas encore initialisés). Les instructions autorisées dépendent des applications et des ouvrages que l'on consulte, mais elles incluent en général la possibilité de

1. copier le contenu d'un registre dans un autre
2. l'adressage indirect : récupérer/écrire le contenu d'un registre dont le numéro est donné par la valeur d'un autre registre
3. effectuer des opérations élémentaires sur un ou des registres, par exemple additionner 1, soustraire 1 ou tester l'égalité à 0.
4. effectuer d'autres opérations sur un ou des registres, par exemple l'addition, la soustraction, la multiplication, division, les décalages binaires, les opérations binaires bit à bit.

Dans ce qui suit, nous réduirons la discussion aux *SRAM (Successor Random Access Machine)* qui ne possèdent que des instructions des types 1., 2. et 3.

Clairement, tout ce qui peut être fait par une machine à compteurs peut être fait par un SRAM.

Clairement, tout ce qui peut être fait par un SRAM est faisable avec une RAM. La réciproque sera prouvée ultérieurement, puisque nous prouverons que tout ce qui peut être fait par algorithme (et donc par RAM) est faisable par machine de Turing, et donc par une machine à compteur.

5.6.2 Structures finies

Définition 5.5 *Une machine SRAM (Successor Random Access Machine) correspond à un programme du type*

```
ctl_state ←0
repeat
   seq
 <instructions>
   if ctl_state = q_a then out ←vrai
   if ctl_state = q_r then out ←faux
  endseq
until out!=undef
write out
```

où <instructions> est une suite finie d'instructions de l'un des types suivants

1. $x_i \leftarrow 0$
2. $x_i \leftarrow x_i + 1$
3. $x_i \leftarrow x_i \ominus 1$
4. $x_i \leftarrow x_j$
5. $x_i \leftarrow x_{x_j}$
6. $x_{x_i} \leftarrow x_j$
7. ***if*** $x_i = 0$ ***then*** *ctl_state* $\leftarrow j$ ***else*** *ctl_state* $\leftarrow j'$

Chacun des i désigne un entier naturel. Chaque x_i désigne un registre qui contient un entier naturel. x_{x_i} désigne le registre dont le numéro est x_i.

Remarque 5.1 *Si l'on souhaite être puriste, observons qu'il s'agit bien d'un algorithme dans le sens du chapitre 4 : on peut considérer x comme un symbole de fonction d'arité 1. x_i désigne en fait $x(i)$, où i désigne un symbole de constante, dont l'interprétation est fixée à l'entier i. x_{x_i} désigne $x(x(i))$. L'ensemble de base est constitué des entiers naturels.*

Puisqu'une machine à compteurs est une machine SRAM particulière, on obtient :

Corollaire 5.2 *Toute machine de Turing à k rubans se simule par une machine SRAM.*

Par la discussion plus haut, en fait seulement 2 registres suffisent sur les structures finies, si l'on ignore l'efficacité.

5.6.3 Sur une structure arbitraire

On peut généraliser la notion de machine SRAM (ou RAM) aux structures arbitraires

$$\mathfrak{M} = (S, f_1, \cdots, f_u, r_1, \cdots, r_v),$$

et aussi obtenir des simulations plus efficaces même dans le cas des structures finies.

Définition 5.6 *Une machine SRAM (Successor Random Access Machine) sur une structure* $\mathfrak{M} = (S, f_1, \cdots, f_u, r_1, \cdots, r_v)$ *possède deux types de registres : des registres* entiers $x_1, x_2, \cdots, x_n, \cdots$ *et des registres* arbitraires $x'_1, x'_2, \cdots, x'_n, \cdots$. *Les premiers contiennent des entiers naturels. Les autres des valeurs du domaine de base* M *de la structure.*

Le programme de la machine correspond à un programme du type

```
ctl_state ←0
repeat
   seq
 <instructions>
   if ctl_state = q_a then out ←vrai
   if ctl_state = q_r then out ←faux
  endseq
until out!=undef
write out
```

où <instructions> est une suite finie d'instructions de l'un des types suivants

1. $x_i \leftarrow 0$
2. $x_i \leftarrow x_i + 1$
3. $x_i \leftarrow x_i \ominus 1$
4. $x_i \leftarrow x_j$
5. $x_i \leftarrow x_{x_j}$
6. $x_{x_i} \leftarrow x_j$
7. **if** $x_i = 0$ **then** $ctl\_state \leftarrow j$ **else** $ctl\_state := j'$
8. $x'_i \leftarrow x'_j$
9. $x'_i \leftarrow x'_{x_j}$
10. $x'_{x_i} \leftarrow x'_j$
11. **if** $r_j(x'_{\ell_1}, x'_{\ell_2}, \cdots, x'_{\ell_k})$ **then** $ctl\_state \leftarrow j$ **else** $ctl\_state := j'$
12. $x'_i \leftarrow f_i(x'_{\ell_1}, x'_{\ell_2}, \cdots, x'_{\ell_k})$
13. $x'_{x_i} \leftarrow f_i(x'_{\ell_1}, x'_{\ell_2}, \cdots, x'_{\ell_k})$

Chacun des ℓ_i *désigne un entier.* r_j *désigne une des relations de la structure,* f_i *une des fonctions de la structure, et* k *son arité.*

Remarque 5.2 *Comme auparavant, si l'on souhaite être puriste, on peut bien voir cela comme un algorithme dans le sens du chapitre 4, en écrivant* $x(i)$ *pour* x_i, *ou* $x'(i)$ *pour* x'_i, *...etc.*

Théorème 5.6 *Toute machine de Turing sur la structure*

$$\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$$

se simule par une machine SRAM sur $\mathfrak{M}$.

Principe:

Nous traitons le cas où la machine de Turing possède un unique ruban ($k = 1$).

La machine RAM commence par copier l'entrée vers les registres x'_1 à x'_n. Les registres x_1 et x_2 seront utilisés pour la simulation. Le registre x_3 contiendra la longueur du mot sur le ruban de la machine de Turing. On met ensuite la valeur **1** dans le registre x_1 qui contient la position de la tête de lecture. Le registre x_2 code l'état interne de la machine de Turing. A partir de ce moment, la machine RAM simule les étapes de la machine de Turing étape par étape.

Pour simuler l'instruction $\delta_1(q) = (1, j)$, $\delta_2(q, r) = (q', 1, i, m')$, où le symbole de relation r_j est d'arité k, on teste si x_2 vaut q par une suite d'instructions du type $x_3 \leftarrow x_2$ suivi d'une suite de q décrémentations de x_3 en testant s'il est nul à l'issu de cette décrémentation, et bien non nul à chaque étape auparavant.

Si c'est le cas, on charge les symboles à droite de la tête de lecture dans les registres $x'_1, x'_2, \cdots, x'_k$ par des instructions du type $x_2 \leftarrow x_1 \quad x'_1 \leftarrow x'_{x_2} \quad x_2 \leftarrow x_2 + 1$ $x'_2 \leftarrow x'_{x_2} \quad x_2 \leftarrow \quad x_2 + 1 \quad \ldots \; x'_k \leftarrow x'_{x_2}$.

On teste alors la relation r_j par une instruction du type **if** $r_j(x'_1, x'_2, \cdots, x'_k)$ **then** *ctl_state* $\leftarrow q$, où l'instruction q correspond au code à effectuer : c'est-à-dire à écrire dans un premier temps $f_i(x'_1, x'_2, \cdots, x'_k)$ sous la tête de lecture. Pour cela on commence par charger les symboles à droite de la tête de lecture si l'arité de f_i est supérieure à k. On calcule $f_i(x'_1, x'_2, \cdots, x'_k)$ par $x'_1 := f_i(x'_1, x'_2, \cdots, x'_k)$. On met le résultat en place par $x'_{x_1} := x'_1$. On déplace ensuite la tête de lecture en incrémentant ou décrémentant (ou préservant) x_1 selon la valeur de m'. On change l'état interne par une instruction du type $x_2 := 0$ suivi de q incrémentations de x_2.

Si la machine de Turing possède k rubans, on généralise la construction en stockant la case numéro i du ruban ℓ dans le registre x'_i avec $i = k * (q - 1) + \ell$.

Nous laissons à notre lecteur le soin de compléter précisément tous les détails.
□

5.6.4 Équivalence avec les machines de Turing

Nous allons montrer que toute machine RAM peut être simulée par une machine de Turing.

Nous allons en fait prouver dans la section suivante que tout algorithme, et donc toute machine RAM sur une structure arbitraire peut être simulée par machine de Turing.

Pour aider à comprendre la construction très générale de la section qui suit, nous allons montrer dans un premier temps que les machines RAM sur les structures finies, c'est-à-dire au sens de la définition 5.5, peuvent être simulées par les machines de Turing.

Pour cela, nous allons réduire le nombre d'instructions des machines RAM à un ensemble réduit d'instructions (*RISC reduced instruction set* en anglais) en utilisant un unique registre x_0 comme accumulateur.

Définition 5.7 *Une machine RISC (sur une structure finie) est une machine SRAM dont les instructions sont uniquement de la forme*

1. $x_0 \leftarrow 0$
2. $x_0 \leftarrow x_0 + 1$
3. $x_0 \leftarrow x_0 \ominus 1$
4. ***if*** $x_0 = 0$ ***then*** $ctl\_state \leftarrow j$
5. $x_0 \leftarrow x_i$
6. $x_i \leftarrow x_0$
7. $x_0 \leftarrow x_{x_i}$
8. $x_{x_0} \leftarrow x_i$

Clairement, tout programme RAM (au sens de la définition 5.5, c'est-à-dire sur une structure finie) peut être converti en un programme RISC équivalent, en passant systématiquement par l'accumulateur x_0.

Théorème 5.7 *Toute machine RISC peut être simulée par une machine de Turing.*

Démonstration: La machine de Turing qui simule la machine RISC possède 4 rubans. Les deux premiers rubans codent les couples (i, x_i) pour x_i non nul. Le troisième ruban code l'accumulateur x_0 et le quatrième est un ruban de travail.

Plus concrètement, le premier ruban code un mot de la forme

$$\ldots \mathbf{BB} < i_o > \mathbf{B} < i_1 > \cdots \mathbf{B} \cdots < i_k > \mathbf{BB} \cdots .$$

Le second ruban code un mot de la forme

$$\ldots \mathbf{BB} < x_{i_o} > \mathbf{B} < x_{i_1} > \cdots \mathbf{B} \cdots < x_{i_k} > \mathbf{BB} \cdots$$

Les têtes de lecture des deux premiers rubans sont sur le deuxième **B**. Le troisième ruban code $< x_0 >$, la tête de lecture étant tout à gauche. On appelle *position standard* une telle position des têtes de lecture.

Au départ, la donnée du programme RAM est copié sur le second ruban, et **0** est placé sur le ruban 1 signifiant que x_0 contient la donnée du programme.

La simulation est décrite pour trois exemples. Notre lecteur pourra compléter le reste.

1. $x_0 \leftarrow x_0 + 1$: on déplace la tête de lecture du ruban 3 tout à droite jusqu'à atteindre un symbole **B**. On se déplace alors d'une case vers la gauche, et on remplace les **1** par des **0**, en se déplaçant vers la gauche tant que possible. Lorsqu'un **0** ou un **B** est trouvé, on le change en **1** et on se déplace à gauche pour revenir en position standard.

2. $x_{23} \leftarrow x_0$: on parcourt les rubans 1 et 2 vers la droite, bloc délimité par **B** par bloc, jusqu'à atteindre la fin du ruban 1, ou ce que l'on lise un bloc **B10111B** (**10111** correspond à 23 en binaire).

   Si la fin du ruban 1 a été atteinte, alors l'emplacement 23 n'a jamais été vu auparavant. On l'ajoute en écrivant **10111** à la fin du ruban 1, et on recopie le ruban 3 (la valeur de x_0) sur le ruban 2. On retourne alors en position standard.

   Sinon, c'est que l'on a trouvé **B10111B** sur le ruban 1. On lit alors $< x_{23} >$ sur le ruban 2. Dans ce cas, il doit être modifié. On fait cela de la façon suivante :

   (a) On copie le contenu à droite de la tête de lecture numéro 1 sur le ruban 4.

   (b) On copie le contenu du ruban 3 (la valeur de x_0) à la place de x_{23} sur le ruban 2.

   (c) On écrit **B**, et on recopie le contenu du ruban 4 à droite de la tête de lecture du ruban 2, de façon à restaurer le reste du ruban 2.

   (d) On retourne en position standard.

3. $x_0 \leftarrow x_{x_{23}}$: En partant de la gauche des rubans 1 et 2, on parcourt les rubans 1 et 2 vers la droite, bloc délimité par **B** par bloc, jusqu'à atteindre la fin du ruban 1, ou ce que l'on lise un bloc **B10111B** (**10111** correspond à 23 en binaire).

   Si la fin du ruban 1 a été atteinte, on ne fait rien, puisque x_{23} vaut 0 et le ruban 3 contient déjà $< x_0 >$.

   Sinon, c'est que l'on a trouvé **B10111B** sur le ruban 1. On lit alors $< x_{23} >$ sur le ruban 2, que l'on recopie sur le ruban 4. Comme ci-dessus, on parcours les rubans 1 et 2 en parallèle jusqu'à trouver $\mathbf{B} < x_{23} > \mathbf{B}$ où atteindre la fin du ruban 1. Si la fin du ruban 1 est atteinte, alors on écrit **0** sur le ruban 3, puisque $x_{x_{23}} = x_0$. Sinon, on copie le bloc correspondant sur le ruban 1 sur le ruban 3, puisque le bloc sur le ruban 2 contient $x_{x_{23}}$, et on retourne en position standard.

□

En fait, on observera que toute opération $x_0 \leftarrow x_0$ ''opération'' x_i peut être simulée ainsi, dès que ''opération'' correspond à une opération calculable.

5.7 Équivalence entre algorithmes et machines de Turing

Nous sommes prêts à montrer l'équivalence entre notre notion d'algorithme et la notion de machines de Turing sur une structure arbitraire.

Nous avons déjà vu qu'un programme de machine de Turing sur une structure arbitraire $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ correspond à un algorithme sur $\mathfrak{M}$ (étendue par des symboles de fonctions dynamiques).

Il nous reste la partie délicate : montrer la réciproque.

Théorème 5.8 *Tout algorithme sur une structure* $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ *peut se simuler par machine de Turing sur la structure* $\mathfrak{M}$.

Démonstration: En suivant la discussion du chapitre 4, nous savons qu'à un algorithme est associé un ensemble fini T de termes critiques, tel que l'algorithme puisse se mettre sous la forme normale indiquée par le théorème 4.1.

Pour simuler un tel algorithme, il suffit d'être capable d'évaluer chacun des emplacements $(f, \overline{m})$, pour f un symbole de fonction de la structure. On peut se restreindre aux emplacements utiles, au sens de la définition du chapitre 4 : c'est-à-dire aux symboles f dynamiques, puisque les autres ont une valeur fixée, et avec $[\![(f, \overline{m})]\!]$ qui ne vaut pas **undef**.

Il y a un nombre fini de tels symboles f. On utilise le principe de la preuve du théorème 5.7, en utilisant 2 rubans par tel symbole f d'arité ≥ 1 (comme pour le symbole x dans la preuve de ce théorème), un ruban par symbole d'arité 0 (comme pour le symbole x_0), en plus de rubans de travail en nombre fini.

Pour un symbole f d'arité $r \geq 1$, les deux rubans codent des mots de la forme

$$\ldots \mathbf{BB}\overline{a}_0\mathbf{B}\overline{a}_1\mathbf{B}\cdots\mathbf{B}\ldots\overline{a}_k\mathbf{BB}\cdots .$$

Le second ruban code un mot de la forme

$$\ldots \mathbf{BB}[\![(f, \overline{a}_0)]\!]\mathbf{B}[\![(f, \overline{a}_1)]\!]\mathbf{B}\cdots[\![(f, \overline{a}_k)]\!]\mathbf{BB}\cdots$$

où $\overline{a}_i$ désigne un r-uplet d'éléments de M, et $[\![(f, \overline{a}_i)]\!]$ le contenu de l'emplacement $(f, \overline{a}_i)$ (et donc désigne un élément de M).

Comme dans la preuve du théorème 5.7, tout emplacement qui n'est pas dans la liste codée par ces deux rubans correspond à une valeur indéfinie, car jamais encore accédée.

Comme dans la preuve du théorème 5.7, on simule instruction par instruction de l'algorithme, en utilisant éventuellement les rubans de travail : les mises à jour sont calculées en utilisant les anciennes valeurs dans les listes, avant d'écraser par les nouvelles valeurs en masse pour simuler les instructions **par endpar**.

Le calcul des relations et des fonctions de la structure de l'algorithme est réalisé en utilisant les relations et les fonctions de la structure par la machine de Turing. □

Remarque 5.3 *La preuve du théorème 5.7 travaille sur les codages binaires des entiers et des emplacements. Dans le théorème 5.8, on travaille directement sur des éléments de* M.

5.8 Synthèse du chapitre

En résumé, nous venons d'obtenir[3].

[3]Bien entendu, il faut comprendre derrière cette terminologie impropre que chacun de ces modèles correspond à un algorithme, et que tout algorithme peut être simulé par chacun de ces modèles.

Corollaire 5.3 *Sur une structure arbitraire* $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$, *les modèles suivants se simulent deux à deux*

1. *Les machines de Turing*
2. *Les machines à* $k \geq 2$ *piles*
3. *Les machines RAM*
4. *Les algorithmes au sens du chapitre 4*

Corollaire 5.4 *Sur une structure finie, ou sur un alphabet fini, les modèles suivants se simulent deux à deux*

1. *Les machines de Turing*
2. *Les machines à* $k \geq 2$ *piles*
3. *Les machines à compteurs*
4. *Les machines à* 2 *compteurs*
5. *Les machines RAM*
6. *Les algorithmes au sens du chapitre 4*

5.9 Notes bibliographiques

Le modèle de machine de Turing est dû à [Turing, 1936]. L'idée de machines de Turing travaillant sur une structure du premier ordre arbitraire est due à [Goode, 1994], et se trouve développée (moins formellement mais peut être aussi clairement) dans [Poizat, 1995]. Dans le reste du chapitre, nous avons utilisé ici diverses sources dont essentiellement [Jones, 1997] et [Hopcroft et al., 2001]. La preuve de l'équivalence entre algorithmes et machines de Turing est due et inspirée de [Dershowitz and Gurevich, 2008]. La notion d'algorithme utilisée ici est, comme nous l'avons dit dans le chapitre 4 due à [Gurevich, 2000].

Chapitre 6

Calculabilité

6.1 Introduction

Dans ce chapitre, nous présentons les principaux résultats de la théorie de la calculabilité. La théorie de la calculabilité permet de discuter de l'existence, ou de la non-existence d'algorithmes pour résoudre un problème donné.

Nous discutons la calculabilité en toute généralité, c'est-à-dire sur une structure quelconque, pas nécessairement fini. Cependant, tous les résultats ne sont pas vrais dans toute structure. En fait, on doit parfois se limiter aux structures suffisamment expressives, ou finis, c'est-à-dire aux structures qui permettent de coder/décoder des listes d'éléments de la structure, ou à des structures dont l'ensemble de base est fini.

Formellement :

Définition 6.1 (Structure finie) *Une structure* $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ *est dite* finie *si l'ensemble de base* M *est fini.*

Définition 6.2 (Algorithme classique) *Un algorithme sur une structure finie sera dit* classique.

Lorsque nous ne le précisons pas, un algorithme est un algorithme classique dans ce document. Nous rappelons que la notion d'algorithme classique est la même sur toute structure finie, selon la discussion du chapitre 5.

Lorsqu'une structure n'est pas finie, nous aurons besoin de distinguer parfois celles en lesquels il est possible de coder les listes :

Définition 6.3 (Structure suffisemment expressive) *Une structure* $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ *est dite* suffisamment expressive *s'il existe un symbole* f_i *de fonction (pour coder les paires) et deux symboles* $f_{i'}$ *et* $f_{i''}$ *de fonctions de la structure pour les décoder : pour tous termes* $t, t' \in M$, $[\![f_{i'}(f_i(t, t'))]\!] = [\![t]\!]$, $[\![f_{i''}(f_i(t, t'))]\!] = [\![t']\!]$.

Selon la discussion du chapitre 2, savoir coder les paires permet de coder les triplets, les quadruplets, ..., et plus généralement les listes. Rappelons que nos structures contiennent toujours au moins deux symboles **0** et **1**, et donc que l'on peut coder un nombre entier en binaire (par une liste de **0** et de **1**). Comme dans le chapitre 2, on notera alors $< ., . >$, $< ., ., . >$, ...etc pour le codage d'une paire, d'un triplet, ...etc.

6.2 Existence d'une machine universelle

6.2.1 Algorithmes et Arbres

Suivant la discussion du chapitre 2, un terme sur une signature Σ correspond à un arbre étiqueté par les symboles de fonctions ou de relations (incluant les constantes) de Σ.

En suivant la discussion du chapitre 4, un algorithme sur la signature Σ peut s'écrire à partir d'affectations, de mises en parallèles, et de tests : on peut donc le voir comme un arbre étiqueté, dont chaque nœud est :

- soit étiqueté par le symbole $\leftarrow$: le nœud possède alors deux fils, l'un correspondant à un terme t (et donc à un arbre) et l'autre à un terme t' (et donc à un arbre) : un tel nœud code l'affectation t$\leftarrow$ t' ;
- soit étiqueté par le symbole **par** : le nœud possède alors 2 fils, l'un correspondant à une instruction R_1 (donc un arbre), l'autre à une autre instruction (donc à un arbre) R_2 : un tel nœud code l'instruction **par** R_1 R_2 **endpar**. Une instruction **par** R_1 R_2 $\cdots$ R_n **endpar** pour $n \geq 3$ pouvant aussi s'écrire **par** R_1 **par** R_2 $\cdots$ R_n **endpar endpar**, on peut se limiter au cas $n = 2$;
- soit étiqueté par le symbole **if then** : le nœud possède alors deux fils, l'un correspondant à un terme t (donc à un arbre), l'autre à une instruction (donc à un arbre) R : un tel nœud code l'instruction **if** t **then** R.

En résumé, un algorithme sur la signature Σ peut se voir comme un arbre étiqueté par les symboles de fonctions et de relations de Σ, union $\{\leftarrow$,**if then**, **par**$\}$.

En suivant la discussion du chapitre 2, un arbre étiqueté comme celui là peut se voir comme un mot fini sur l'alphabet M, où M est l'ensemble de base de la structure.

Un algorithme classique correspond à un mot sur un alphabet fini[1]. Un algorithme sur une structure non finie correspond à un mot sur un alphabet non fini.

6.2.2 Machine universelle

L'étape suivante est de se convaincre alors du résultat suivant.

[1]Celui de la structure. Par les discussions du chapitre précédent, on peut se ramener à l'alphabet $\{\mathbf{0}, \mathbf{1}\}$.

Théorème 6.1 (Existence d'une algorithme/d'une machine universelle) *Il existe un algorithme (classique) qui prend en entrée $< A, d >$, où*

1. *A est un algorithme*[2]
2. *d est une donnée*[3]

qui simule l'algorithme (classique) A sur la donnée (l'entrée) d.

Cela fonctionne en fait aussi sur toute structure suffisamment expressive.

Théorème 6.2 (Existence d'une algorithme/d'une machine universelle) *Fixons une structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ suffisamment expressive.*
Il existe un algorithme sur $\mathfrak{M}$ qui prend en entrée $< A, d >$, où

1. *A est algorithme*[4] *sur $\mathfrak{M}$*
2. *d est une donnée*[5]

qui simule l'algorithme A sur la donnée (l'entrée) d.

Démonstration: Suite à la discussion du chapitre 4, et en utilisant le principe de la preuve dans le chapitre 5 de la simulation d'un algorithme A (respectivement sur une structure $\mathfrak{M}$) par une machine de Turing (respectivement sur une structure $\mathfrak{M}$), il suffit d'être capable de maintenir la liste de chacun des emplacements $(f, \overline{m})$, pour f un symbole de fonction de la structure, de l'exécution de A.

On peut se restreindre aux emplacements utiles, au sens de la définition du chapitre 4 : c'est-à-dire aux symboles f dynamiques, puisque les autres ont une valeur fixée, et avec $[\![(f, \overline{m})]\!]$ qui ne vaut pas **undef**.

Simuler l'algorithme A sur la donnée d correspond alors à faire des manipulations de listes analogues à celles dans la preuve de la simulation d'un algorithme A (respectivement sur une structure $\mathfrak{M}$) par une machine de Turing. □

6.3 Langages et problèmes (semi-)décidables

6.3.1 Problèmes de décision

En calculabilité et complexité, on s'intéresse essentiellement aux problèmes de décision, c'est-à-dire aux problèmes dont la réponse est soit positive, soit négative.

Définition 6.4 *Un* problème de décision *P est la donnée d'un ensemble E d'instances, et d'un sous-ensemble E^+ d'instances dites* positives *pour lequel la réponse est* oui.

Exemple 6.1 *On peut considérer les problèmes suivants :*

[2] C'est-à-dire le codage par un mot d'un arbre codant un algorithme.
[3] C'est-à-dire un mot sur l'alphabet M.
[4] C'est-à-dire le codage par un mot d'un arbre codant un algorithme.
[5] C'est-à-dire un mot sur l'alphabet M.

1. Nombre premiers : *l'ensemble E est l'ensemble des entiers naturels, et E^+ est le sous-ensemble des entiers premiers.*
2. Algorithme : *l'ensemble E est l'ensemble des mots sur l'alphabet M, et E^+ est le sous-ensemble de ceux qui codent un algorithme.*
3. Chemin : *l'ensemble E est l'ensemble des triplets constitués d'un graphe, d'un sommet u et d'un sommet v de ce graphe, et E^+ est le sous-ensemble de ces instances pour lesquelles il existe un chemin entre u et v dans le graphe.*

En fait, à un problème est associé implicitement une fonction de codage, qui permet de coder les instances, c'est-à-dire les éléments de E, par un mot sur un certain alphabet M (fini ou non-fini). On voit donc E comme un sous-ensemble de M^*, où M est un certain alphabet. On peut alors voir un problème comme un langage : à un problème P, on associe le langage $L(P)$ correspondant à l'ensemble des mots codant un instance de E, qui appartient à E^+ :

$$L(P) = \{w | w \in E^+\}.$$

Réciproquement, on peut voir tout langage L comme un problème de décision : prendre E comme l'ensemble des mots sur l'alphabet du langage, et prendre $E^+ = L$.

Par conséquent, dans la suite, nous utiliserons indifféremment la terminologie langage ou problème.

6.3.2 Langages semi-décidables

Soit A un algorithme. À l'algorithme A, on peut associer le langage $L(A)$ des entrées sur lequel A termine. Un langage $L \subset M^*$ est dit *semi-décidable* s'il correspond à un $L(A)$ pour un certain algorithme A.

On peut généraliser le concept aux algorithmes sur une structure $\mathfrak{M}$: soit A un algorithme sur une structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$. À l'algorithme A, on peut associer le langage $L(A)$ des entrées sur lequel A termine. Un langage $L \subset M^*$ est dit *semi-décidable sur la structure* $\mathfrak{M}$ s'il correspond à un $L(A)$ pour un certain algorithme A sur la structure $\mathfrak{M}$.

Si l'on préfère, cela peut se reformuler en :

Définition 6.5 (Langages semi-décidables) – *Un langage $L \subset M^*$ est dit* semi-décidable *s'il existe un algorithme (classique) qui sur toute entrée $w \in M^*$ termine si et seulement si $w \in L$.*

– *Un langage $L \subset M^*$ est dit* semi-décidable sur la structure

$$\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$$

s'il existe un algorithme sur la structure $\mathfrak{M}$ qui sur toute entrée $w \in M^*$ termine si et seulement si $w \in L$.

Un langage semi-décidable est aussi dit *récursivement énumérable*.

On note RE la classe des langages et des problèmes semi-décidables.

6.3.3 Langages décidables

Définition 6.6 (Langage décidable) – *Un langage $L \subset M^*$ est dit* décidable *s'il est semi-décidable et son complémentaire aussi.*
– *Un langage $L \subset M^*$ est dit* décidable sur la structure

$$\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$$

s'il est semi-décidable sur cette structure et son complémentaire aussi.

Un langage ou un problème décidable est aussi dit *récursif*. Un langage qui n'est pas décidable, est dit *indécidable*.

On note R pour la classe des langages et des problèmes *décidables*.

Corollaire 6.1 $R \subset RE$

Cela se généralise bien entendu à toute structure.

La terminologie de *décidable* vient du résultat suivant :

Théorème 6.3 *Un problème ou un langage L sur un alphabet fini M est décidable si et seulement s'il existe un algorithme (classique) qui*
– *termine sur toute entrée w ;*
– *termine avec la réponse 1 (vrai) lorsque $w \in L$;*
– *termine avec la réponse 0 (faux) lorsque $w \notin L$.*

Démonstration: sens $\Rightarrow$. Supposons que L soit récursif. Il existe un algorithme M_1 qui termine sur L, et un algorithme M_2 qui termine sur son complémentaire. On construit un algorithme qui, sur une entrée w, simule en parallèle M_1 et M_2, jusqu'à ce que l'un des deux termine. Si M_1 termine, il répond 1. Si c'est M_2 il répond 0. L'algorithme produit vérifie les bonnes propriétés.

sens $\Leftarrow$. Réciproquement, supposons qu'il existe un tel algorithme A. On construit un algorithme B qui termine sur L en construisant un algorithme qui simule A. Si A répond 1, l'algorithme B s'arrête alors, et si A répond 0, l'algorithme B rentre dans une boucle qui ne termine jamais. On construit un algorithme B' qui termine sur le complémentaire de L en inversant le rôle de 1 et 0 plus haut. □

La preuve montre clairement que cela se généralise aux structures quelconques.

Théorème 6.4 *Un problème ou un langage L sur une structure $\mathfrak{M}$ est décidable si et seulement s'il existe un algorithme sur $\mathfrak{M}$ qui*
– *termine sur toute entrée w*
– *termine avec la réponse 1 lorsque $w \in L$*
– *termine avec la réponse 0 lorsque $w \notin L$*

Ces résultats expliquent la terminologie de *décidable* par rapport à la définition que nous en avions posé à partir de la notion de *semi-décidable*.

Observons que le principe de la preuve précédente montre aussi que pour un ensemble semi-décidable, on a la propriété suivante :

Théorème 6.5 *Un problème ou un langage L (respectivement sur une structure $\mathfrak{M}$) est semi-décidable si et seulement s'il existe un algorithme sur $\mathfrak{M}$ qui*
- *termine avec la réponse* 1 *lorsque* $w \in L$
- *soit termine avec la réponse* 0*, soit ne termine pas, lorsque* $w \notin L$*.*

La terminologie de *récursif* fait référence à la notion de fonction récursive : voir le cours [Dowek, 2008] qui présente par exemple la calculabilité par l'intermédiaire des fonctions récursives.

6.3.4 Semi-décision et énumération

La notion de *énumérable* dans *récursivement énumérable* s'explique par le résultat suivant. Notez que l'on se limite ici aux structures finies, car l'on a besoin dans la preuve que l'ensemble des mots soit dénombrable.

Théorème 6.6 *Un langage $L \subset M^*$ est récursivement énumérable si et seulement si l'on peut produire un algorithme classique qui affiche un à un (énumère) tous les mots du langage L.*

Démonstration: sens $\Rightarrow$. Supposons que L soit récursivement énumérable. Il existe un algorithme A qui termine sur les mots de L.

L'ensemble de couples (t, w), où t est un entier, w est un mot est dénombrable. On peut se convaincre assez facilement qu'il est en fait effectivement dénombrable : on peut construire un algorithme qui produit le codage $< t, w >$ de tous les couples (t, w). Par exemple, on considère une boucle qui pour $t = 1, 2, \cdots$ jusqu'à l'infini, considère tous les mots w de longueur inférieure ou égale à t, et produit pour chacun le couple $< t, w >$.

Considérerons un algorithme B qui en plus à chaque couple produit (t, w), simule en outre t étapes de la machine A. Si la machine A termine en exactement t étapes, B affiche alors le mot w. Sinon B n'affiche rien pour ce couple.

Un mot du langage L, est accepté par A en un certain temps t. Il sera alors écrit par B lorsque celui-ci considérera le couple (t, w). Clairement, tout mot w affiché par B est accepté par A, et donc est un mot de L.

sens $\Leftarrow$. Réciproquement, si l'on a un algorithme classique B qui énumère tous les mots du langage L, on peut construire un algorithme A, qui étant donné un mot w, simule B, et à chaque fois que B produit un mot compare ce mot au mot w. S'ils sont égaux, A s'arrête. Sinon, A continue à jamais. Si éventuellement il s'avère en simulant B que B termine (ce qui signifierait que L soit fini) alors A rentre dans une boucle sans fin.

Par construction, sur une entrée w, A termine si w se trouve parmi les mots énumérés par B, c'est-à-dire si $w \in L$. Si w ne se trouve pas parmi ces mots, par hypothèse, $w \notin L$, et donc par construction, A ne terminera jamais. □

6.3.5 Propriétés de clôture

Théorème 6.7 *L'ensemble des langages semi-décidables (respectivement sur une structure $\mathfrak{M}$) est clos par union et par intersection : autrement dit, si L_1 et L_2 sont semi-décidables, alors $L_1 \cup L_2$ et $L_1 \cap L_2$ le sont.*

Démonstration: Supposons que L_1 corresponde à $L(A_1)$ pour un algorithme A_1 et L_2 à $L(A_2)$ pour un algorithme A_2. Alors $L_1 \cup L_2$ correspond à $L(A)$ pour l'algorithme A qui simule en parallèle A_1 et A_2 et qui termine dès que le premier des deux algorithmes A_1 et A_2 termine.

$L_1 \cap L_2$ correspond à $L(A)$ pour l'algorithme A qui simule en parallèle A_1 et A_2 et qui termine dès que les deux algorithmes A_1 et A_2 terminent.

□

Théorème 6.8 *L'ensemble des langages décidables (respectivement sur une structure $\mathfrak{M}$) est clos par union, intersection, et complément : autrement dit, si L_1 et L_2 sont décidables, alors $L_1 \cup L_2$, $L_1 \cap L_2$, et L_1^c le sont.*

Démonstration: Par la définition 6.6, la clôture par complémentaire est triviale.

Il reste à montrer qu'avec les hypothèses, chaque langage $L_1 \cup L_2$ et $L_1 \cap L_2$ et son complémentaire est semi-décidable. Mais cela est clair par le théorème précédent, et par les lois de Morgan (le complémentaire d'une union est l'intersection des complémentaires, et inversement). □

6.4 Indécidabilité

6.4.1 Un premier problème indécidable

En fait, l'ensemble des langages décidables est strictement inclus dans l'ensemble des langages semi-décidables. Nous allons le prouver en utilisant un argument de *diagonalisation*, c'est-à-dire un procédé analogue à la diagonalisation de Cantor qui permet de montrer que l'ensemble des parties de $\mathbb{N}$ n'est pas dénombrable.

Définition 6.7 (Langage L_{univ}) *On appelle langage universel (respectivement langage universel sur une structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$), noté L_{univ}, le langage*

$$L = \{< A, w > | A \text{ termine sur l'entrée } w\}$$

constitué des paires $< A, w >$ où A code un algorithme (respectivement sur une structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$), w code une entrée de cet algorithme, et A termine sur l'entrée w.

Tout d'abord ce langage est bien semi-décidable : en effet, le théorème 6.1 permet d'affirmer.

Corollaire 6.2 *Le langage universel L_{univ} (respectivement sur une structure $\mathfrak{M}$) est semi-décidable (resp. sur la structure $\mathfrak{M}$).*

Démonstration: Pour décider si on doit accepter un mot w', il suffit de vérifier que ce mot est de la forme $w' =< A, w >$, puis de simuler l'algorithme A sur l'entrée w. On s'arrête si l'on détecte que l'algorithme A s'arrête sur l'entrée w. □

Il s'avère qu'il n'est pas décidable.

Théorème 6.9 *Le langage universel L_{univ} (respectivement sur une structure $\mathfrak{M}$) n'est pas décidable (resp. sur la structure $\mathfrak{M}$).*

Démonstration: On prouve le résultat par un raisonnement par l'absurde. Supposons que L_{univ} soit décidé par un algorithme A, qui s'arrête sur toute entrée (qui existe par le théorème 6.3).

On construit alors un algorithme B de la façon suivante :
- B prend en entrée un mot $< C >$ codant un algorithme C
- B appelle l'algorithme A sur le mot $<< C >, < C >>$
- Si l'algorithme A
  - accepte ce mot, B refuse.
  - refuse ce mot, B accepte.

On montre qu'il y a une contradiction, en appliquant l'algorithme B sur le mot $< B >$:
- Si B accepte $< B >$, cela signifie, par définition de L_{univ} et de A, que A accepte $<< B >, < B >>$. Mais si A accepte ce mot, B est construit pour refuser son entrée $< B >$. Contradiction.
- Si B refuse $< B >$, cela signifie, par définition de L_{univ} et de A, que A refuse $<< B >, < B >>$. Mais si A refuse ce mot, B est construit pour accepter son entrée $< B >$. Contradiction.

□

Autrement dit,

Corollaire 6.3 $\mathrm{R} \subset \mathrm{RE}$*, et l'inclusion est stricte.*

Une fois que nous avons obtenu un premier problème indécidable, nous allons en obtenir d'autres.

Tout d'abord, en vertu du théorème 6.3, on peut en obtenir un très rapidement.

Corollaire 6.4 *Le complémentaire du langage L_{univ} (respectivement sur une structure $\mathfrak{M}$) n'est pas décidable (resp. sur la structure $\mathfrak{M}$).*

6.4.2 Notion de réduction

Nous connaissons deux langages indécidables, L_{univ} et son complémentaire. Notre but est maintenant d'en obtenir d'autres, ou de savoir comparer les problèmes. Nous introduisons pour cela la notion de réduction entre problèmes ou entre langages.

Tout d'abord, nous pouvons généraliser la notion de *calculable* aux fonctions et pas seulement aux langages et problèmes de décision.

Définition 6.8 (Fonction calculable) *Soient M et N deux alphabets. Une fonction $f : M^* \to N^*$ est* calculable *s'il existe un algorithme A, tel que pour tout mot w, A avec l'entrée w termine, et écrit en sortie $f(w)$.*

On peut se convaincre facilement que la composée de deux fonctions calculable est calculable.

Cela nous permet d'introduire une notion de réduction entre problèmes : l'idée est que si A se réduit à B, alors le problème A est plus facile que le problème B, ou si l'on préfère, le problème B est plus difficile que le problème A.

Définition 6.9 (Réduction) *Soient A et B deux problèmes d'alphabet respectifs M_A et M_B, et de langages respectifs M_A et M_B. Une* réduction de A vers B *est une fonction $f : M_A^* \to M_B^*$ calculable telle que $w \in L_A$ ssi $f(w) \in L_B$. On note $A \leq_m B$ lorsque A se réduit à B.*

Intuitivement, si un problème est plus facile qu'un problème décidable, alors il est décidable. Formellement.

Proposition 6.1 (Réduction) *Si $A \leq_m B$, et si B est décidable (respectivement : sur une structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$) alors A est décidable (resp. sur $\mathfrak{M}$).*

Démonstration: A est décidé par l'algorithme qui, sur une entrée w, calcule $f(w)$, puis simule l'algorithme qui décide B sur l'entrée $f(w)$. Puisqu'on a $w \in A$ si et seulement si $f(w) \in B$, l'algorithme est correct. □

Proposition 6.2 (Réduction) *Si $A \leq_m B$, et si A est indécidable, alors B est indécidable.*

Démonstration: Il s'agit de la contraposée de la proposition précédente. □

6.4.3 Quelques autres problèmes indécidables

Cette idée va nous permettre d'obtenir immédiatement la preuve de l'indécidabilité de plein d'autres problèmes.

Il n'est pas possible de déterminer si un algorithme s'arrête sur au moins une entrée :

Proposition 6.3 *Le langage $L_\emptyset$ constitué des mots w qui correspondent au codage d'un algorithme A tel que $L(A) \neq \emptyset$, c'est-à-dire*

$$L_\emptyset = \{< A > | L(A) \neq \emptyset\}$$

est indécidable.

Démonstration: On construit une réduction de L_{univ} vers $L_\emptyset$: pour toute paire $<< A >, w >$, on considère l'algorithme A_w défini de la manière suivante :

- A_w prend en entrée un mot u.
- Si $u = w$, A_w simule A sur w,
- sinon A rejette (rentre dans une boucle sans fin).

La fonction f qui à $<< A >, w >$ associe $< A_w >$ est bien calculable. De plus on a $<< A >, w >\in L_{univ}$ si et seulement si $L(A_w) \neq \emptyset$, c'est-à-dire $< A_w >\in L_\emptyset$: en effet, le mot w est le seul qui puisse être accepté par A_w, et il l'est si et seulement si A accepte w. □

Proposition 6.4 *L'ensemble $L_{\neq}$ constitué des mots $< A, A' >$ qui correspondent au codage d'un algorithme A et d'un algorithme A' tels que $L(A) \neq L(A')$, c'est-à-dire*

$$L_{\neq} = \{< A, A' > | L(A) \neq L(A')\}$$

est indécidable.

Démonstration: On construit une réduction de $L_\emptyset$ vers $L_{\neq}$. On considère un algorithme fixe B qui accepte le langage vide : prendre par exemple un algorithme B qui rentre immédiatement dans une boucle sans fin. La fonction f qui à $< A >$ associe $< A, B >$ est bien calculable. De plus on a $< A >\in L_\emptyset$ si et seulement si $L(A) \neq \emptyset$ si et seulement si $< A, B >\in L_{\neq}$. □

6.4.4 Théorème de Rice

En fait, les deux résultats précédents peuvent être vus comme les conséquences d'un résultat très général qui affirme que toute propriété non-triviale des algorithmes est indécidable.

Théorème 6.10 (Théorème de Rice) *Toute propriété non-triviale des langages semi-décidables est indécidable.*

Autrement dit, soit une propriété P non-triviale, c'est-à-dire qui n'est pas vraie pour tout langage semi-décidable, et qui n'est pas non plus fausse non plus pour tout langage semi-décidable.

Alors le langage L_P constitué des mots w qui codent un algorithme A tel que $L(A)$ vérifie la propriété P est indécidable.

Remarquons que si une propriété P est triviale, au sens plus haut, il est facile de décider L_P : construire un algorithme qui ne lit même pas son entrée et qui accepte (respectivement : refuse).

Démonstration:

Il nous faut démontrer que L_P est indécidable.

Quitte à remplacer P par sa négation, on peut supposer que le langage vide ne vérifie pas la propriété P (prouver l'indécidabilité de L_p est par la définition 6.6 équivalent à prouver l'indécidabilité de son complémentaire). Puisque P est non triviale, il existe un moins un algorithme B avec $L(B)$ qui vérifie P.

On construit une réduction de L_{univ} vers le langage L_P. Étant donnée une paire $<< A >, w >$, on considère l'algorithme A_w défini de la façon suivante :

- A_w prend en entrée un mot u.
- Sur le mot u, A_w simule A sur le mot w.

– Si A accepte w, alors A_w simule B sur le mot u : A_w accepte si seulement si B accepte u.

Autrement dit, A_w accepte, si et seulement si A accepte w et si B accepte u. Si w est accepté par A, alors $L(A_w)$ vaut $L(B)$, et donc vérifie la propriété P. Si w n'est pas accepté par A, alors $L(A_w) = \emptyset$, et donc ne vérifie pas la propriété P.

La fonction f qui à $<< A >, w >$ associe $< A_w >$ est bien calculable. □

6.4.5 Notion de complétude

Notons que l'on peut aussi introduire une notion de complétude.

Définition 6.10 (RE-complétude) *Un problème A est dit* RE*-complet, si*

1. *il est récursivement énumérable*
2. *tout autre problème récursivement énumérable B est tel que $B \leq_m A$.*

Autrement dit, un problème RE-complet est maximal pour $\leq_m$ parmi les problèmes de la classe RE.

Théorème 6.11 *Le problème L_{univ} est* RE*-complet.*

Démonstration: L_{univ} est semi-décidable. Maintenant, soit L un langage semi-décidable. Par définition, il existe un algorithme A qui termine sur L. Considérons la fonction f qui à w associe le mot $<< A >, w >$. On a $w \in L$ si et seulement si $f(w) \in L_{univ}$, et donc $L \leq_m L_{univ}$. □

6.5 Problèmes indécidables naturels

On peut objecter que les problèmes précédents, relatifs aux algorithmes sont "artificiels", dans le sens où ils parlent de propriétés d'algorithmes, les algorithmes ayant eux même été définis par la théorie de la calculabilité.

Il est difficile de définir formellement ce qu'est un problème *naturel*, mais on peut au moins dire qu'un problème qui a été discuté avant l'invention de la théorie de la calculabilité est naturel.

6.5.1 Le dixième problème de Hilbert

C'est clairement le cas du dixième problème de Hilbert, identifié par Hilbert parmi les problèmes intéressants pour le 20ième siècle en 1900 : peut-on déterminer si une équation polynomiale à coefficients entiers possède une solution entière.

Théorème 6.12 *Le dixième problème de Hilbert (étant donné un polynôme $P \in \mathbb{N}[X_1, \cdots, X_n]$ à coefficients entiers, déterminer s'il possède une solution entière) est indécidable.*

La preuve de ce résultat, due à Matiyasevich [Matiyasevich, 1970], qui s'obtient toujours à partir d'une réduction à partir d'un des problèmes précédents, est hors de l'ambition de ce document.

6.5.2 Le problème de la correspondance de Post

La preuve de l'indécidabilité du problème de la correspondance de Post est plus facile, même si nous ne la donnerons pas ici. On peut considérer ce problème comme "naturel", dans le sens où il ne fait pas référence directement à la notion d'algorithme, ou aux machines de Turing.

Définition 6.11 (Problème de la correspondance de Post) *Le* problème de la correspondance de Post *est le suivant :*

- *Une instance est une suite* $(u_1, v_1), \cdots, (u_n, v_n)$ *de paires de mots sur l'alphabet* Σ.
- *Une solution est une suite d'indices* $i_1, i_2, \cdots, i_m$ *de* $\{1, 2, \cdots, n\}$ *telle que*

$$u_{i_1} u_{i_2} \cdots u_{i_m} = v_{i_1} v_{i_2} \cdots v_{i_m}$$

- *Une instance est positive si et seulement si elle possède au moins une solution.*

Théorème 6.13 *Le problème de la correspondance de Post (c'est-à-dire étant donnée une instance, déterminer si elle est positive) est indécidable.*

6.5.3 Décidabilité/Indécidabilité de théories logiques

Dans cette sous-section, on considère des logiques du premier ordre où le modèle est l'ensemble $\mathbb{N}$ des entiers naturels. Les opérations autorisées sur les entiers sont l'égalité, l'addition et la multiplication.

Définition 6.12 *Une théorie logique est dite* décidable *si le problème de savoir si une formule close est vraie est décidable.*

Théorème 6.14 (Presburger) *La théorie du premier ordre des entiers muni de l'addition (mais pas de la multiplication) est décidable.*

Nous ne donnerons pas la preuve de ce résultat, dont l'idée est de passer par la théorie des automates. On montre en effet que pour toute formule, l'ensemble des entiers qui la satisfont est un langage rationnel (reconnu par automate). La preuve se fait par induction sur les formules. La décidabilité découle alors du fait que l'on peut décider si un automate accepte tous les mots.

Théorème 6.15 *La théorie du premier ordre des entiers munis de l'addition et de la multiplication est indécidable.*

La preuve de ce résultat est beaucoup plus technique : elle consiste à réduire le problème L_{univ} à ce problème : à toute instance $< A, w >$ on associe une formule $\exists x\ \phi_{A,w}$ qui est vrai si et seulement si le mot w est accepté par l'algorithme A. L'idée étant de coder un chemin acceptant de l'algorithme par une formule.

6.6 Théorème du point fixe

Les résultats de cette section sont très subtiles, et extrêmement puissants.

Commençons par une version simple, qui nous aidera à comprendre les preuves.

Proposition 6.5 *Il existe un algorithme A^* qui écrit son propre algorithme : il produit en sortie $< A^* >$.*

En d'autres termes, dans tout langage de programmation qui est équivalent aux machines de Turing, il est possible d'écrire un programme qui affiche son propre code.

En shell *UNIX* par exemple, le programme suivant

```
x='y=‘echo . | tr . "\47" ‘; echo "x=$y$x$y;$x"'
y=‘echo . | tr . "\47"‘; echo "x=$y$x$y;$x"
```

produit

```
x='y=‘echo . | tr . "\47" ‘; echo "x=$y$x$y;$x"';y=‘echo .
| tr . "\47" ‘; echo "x=$y$x$y;$x"
```

qui est une commande, qui exécutée, affiche son propre code.

On appelle parfois de tels programme des *quines*, en l'honneur du philosophe Willard van Orman Quine, qui a discuté l'existence de programmes autoreproducteurs.

Démonstration: On considère des algorithmes qui terminent sur tout entrée. Pour deux tels algorithmes A et A', on note AA' l'algorithme qui est obtenu en composant de façon séquentielle A et A'. Formellement, AA' est l'algorithme qui effectue d'abord A, et puis lorsque A est terminé avec la sortie w, il effectue A' sur l'entrée w.

On construit les algorithmes suivants :

1. Étant donné un mot w, l'algorithme $Print_w$ termine avec le résultat w.
2. Pour une entrée w de la forme $w =< X >$, où X est un algorithme, l'algorithme B produit en sortie le codage de l'algorithme $Print_w X$, c'est-à-dire le codage de l'algorithme obtenu en composant $Print_w$ et X.

On considère alors l'algorithme A^* donné par $Print_{<B>}B$, c'est-à-dire la composition séquentielle des machines $Print_{<B>}$ et B.

Déroulons le résultat de cet algorithme : l'algorithme $Print_{<B>}$ produit en sortie $< B >$. La composition par B produit alors le codage de $Print_{<B>} < B >$, qui est bien le codage de l'algorithme A^*. □

Le théorème de récursion permet des auto-références dans un langage de programmation. Sa démonstration consiste à étendre l'idée derrière la preuve du résultat précédent.

Théorème 6.16 (Théorème de récursion) *Soit $t : M^* \times M^* \to M^*$ une fonction calculable. Alors il existe un algorithme R qui calcule une fonction $r : M^* \to M^*$ telle que pour tout mot w*

$$r(w) = t(< R >, w).$$

Son énoncé peut paraître technique, mais son utilisation reste assez simple. Pour obtenir un algorithme qui obtient sa propre description et qui l'utilise pour calculer, on a besoin simplement d'un algorithme T qui calcule une fonction t comme dans l'énoncé, qui prend une entrée supplémentaire qui contient la description de l'algorithme. Alors le théorème de récursion produit un nouvel algorithme R qui opère comme T mais avec la description de $< R >$ gravée dans son code.

Démonstration: Il s'agit de la même idée que précédemment. Soit T un algorithme calculant la fonction t : T prend en entrée une paire $< u, w >$ et produit en sortie un mot $t(u, w)$.

On considère les algorithmes suivants :

1. Étant donné un mot w, l'algorithme $Print_w$ prend en entrée un mot u et termine avec le résultat $< w, u >$.
2. Pour une entrée w' de la forme $<< X >, w >$, l'algorithme B
   (a) calcule $<< Print_{<X>}X >, w >$, où $Print_{<X>}X$ désigne l'algorithme qui compose $Print_{<X>}$ avec X,
   (b) puis passe le contrôle à l'algorithme T.

On considère alors l'algorithme R donné par $Print_{<B>}B$, c'est-à-dire l'algorithme obtenu en composant $Print_{<B>}$ avec B.

Déroulons le résultat $r(w)$ de cet algorithme R sur une entrée w : sur une entrée w, l'algorithme $Print_{<B>}$ produit en sortie $<< B >, w >$. La composition par B produit alors le codage de $<< Print_{<B>}B >, w >$, et passe le contrôle à T. Ce dernier produit alors $t(< Print_{<B>}B >, w) = t(< R >, w) = r(w)$. □

On obtient alors.

Théorème 6.17 (Théorème du point fixe de Kleene) *Soit une fonction calculable qui à chaque mot $< A >$ codant un algorithme associe un mot $< A' >$ codant un algorithme. Notons $A' = f(A)$.*

Alors il existe un algorithme A^* tel que $L(A^*) = L(f(A^*))$.

Démonstration: Considérons une fonction $t : M^* \times M^* \to M^*$ telle que $t(< A >, x)$ soit le résultat de la simulation de l'algorithme $f(A)$ sur l'entrée x. Par le théorème précédent, il existe un algorithme R qui calcule une fonction r telle que $r(w) = t(< R >, w)$. Par construction $A^* = R$ et $f(A^*) = f(R)$ ont donc même valeur sur w pour tout w. □

On peut interpréter les résultat précédents en lien avec les virus informatiques. En effet, un virus est un programme qui vise à se diffuser, c'est-à-dire à s'autoreproduire, sans être détecté. Le principe de la preuve du théorème de récursion est un moyen de s'autoreproduire, en dupliquant son code.

6.7 Notes bibliographiques

Ce chapitre contient des résultats standards en calculabilité. Nous nous sommes inspirés ici essentiellement de leur présentation dans [Wolper, 2001],

[Carton, 2008], [Jones, 1997], [Kozen, 1997], [Hopcroft et al., 2001], ainsi que dans [Sipser, 1997].

Chapitre 7

La notion de circuit

7.1 La notion de circuit

Puisque les ordinateurs modernes sont constitués de circuits électroniques digitaux, il est assez naturel que la notion de *circuit*, et en particulier de *circuit booléen* joue un rôle fondamental lorsque l'on parle d'algorithmes et de complexité.

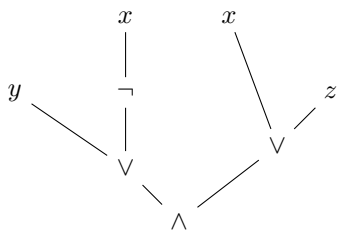
7.1.1 Circuits booléens

On peut voir un *circuit booléen* comme un moyen de décrire comment on produit une sortie booléenne en appliquant une suite d'opération *OU* ($\vee$), *AND* ($\wedge$) et NOT ($\neg$) sur des bits en entrée.

Définition 7.1 (Circuit booléen) *Soit n un entier. Un* circuit booléen a n entrées et une sortie *est un graphe orienté sans cycle (*DAG, directly oriented graph*) avec*

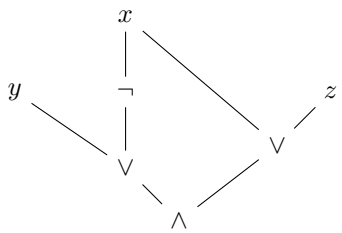
- n entrées, *c'est-à-dire n sommets sans arc entrant.*
- *exactement une* sortie, *c'est-à-dire un sommet sans arc sortant (un puit au sens de la théorie des graphes).*
- *chaque entrée est étiquetée soit par la constante* **0**, *soit par la constante* **1**, *soit par un symbole de variable $x_1, x_2, \cdots, x_n$.*
- *tout autre sommet est appelé une* porte *et est étiqueté soit par $\vee$, soit par $\wedge$, soit par $\neg$. On appelle* fanin *le degré entrant d'une porte.*
- *les portes étiquetées par $\vee$ ou $\wedge$ sont de fanin* 2.
- *les portes étiquetées par $\neg$ sont de fanin* 1.

Exemple 7.1 *Voici un exemple de circuit, correspondant à la fonction $S(x, y, z) = \wedge(\vee(\neg(x), y), \vee(x, z))$.*

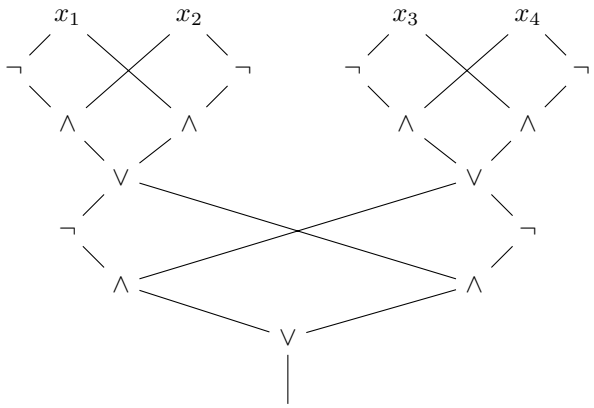


Notons que plusieurs entrées peuvent être étiquetées par un même symbole de constante ou de variable. On s'autorise graphiquement à partager les entrées.

Exemple 7.2 *Le circuit de l'exemple 7.1 peut aussi se représenter comme*



Exemple 7.3 *Voici un exemple de circuit moins trivial.*



Étant donné un circuit booléen C à n entrées, et $\overline{x} \in \{0,1\}^n$, la *sortie de* C sur $\overline{x} = (x_1, \cdots, x_n)$, notée $C(\overline{x})$ se définit comme on s'y attend.

Définition 7.2 (Fonction booléenne associée à un circuit) *Plus formellement, pour chaque sommet* ν *de* C*, on définit sa valeur* $val(\nu) \in \{0,1\}$ *sur* $\overline{x} = (x_1, \cdots, x_n)$ *comme suit :*

– *si* ν *est une entrée étiquetée par* 0*, alors* $val(\nu) = 0$ *;*

- *si ν est une entrée étiquetée par* 1, *alors* $val(\nu) = 1$;
- *si ν est une entrée étiquetée par une variable* x_i, *alors* $val(\nu) = x_i$;
- *si ν est une porte* $\wedge$, *alors* $val(\nu)$ *est la conjonction logique des valeurs* $val(e_1)$ *et* $val(e_2)$ *de ses entrées* e_1 *et* e_2 : $val(\nu) = \wedge(val(e_1), val(e_2))$;
- *si ν est une porte* $\vee$, *alors* $val(\nu)$ *est la disjonction logique des valeurs* $val(e_1)$ *et* $val(e_2)$ *de ses entrées* e_1 *et* e_2 : $val(\nu) = \vee(val(e_1), val(e_2))$;
- *si ν est une porte* $\neg$, *alors* $val(\nu)$ *est la négation de son entrée* e : $val(\nu) = \neg(val(e))$.

La valeur $C(\overline{x})$ *du circuit est alors donnée par la valeur* $val(\nu)$ *pour son unique sortie.*

Exemple 7.4 *Le circuit de l'exemple 7.1 calcule la fonction de* $\{0,1\}^3$ *dans* $\{0,1\}$ *définie par* $S(x,y,z) = \wedge(\vee(\neg(x), y), \vee(x,z))$, *c'est-à-dire la fonction* sélecteur.

Exemple 7.5 *Le circuit de l'exemple 7.3 calcule la fonction* $PARITY$ *de* $\{0,1\}^4$ *dans* $\{0,1\}$ *qui vaut* 1 *si et seulement si un nombre impair de ses arguments valent* 1.

La *taille* d'un circuit C, noté $taille(C)$ est le nombre de sommets du circuit. La *profondeur d'un circuit* C, notée $profondeur(C)$ est la longueur du plus long chemin entre une entrée et la sortie.

Exemple 7.6 *Le circuit de l'exemple 7.3 est de taille* 19 *et de profondeur* 6.

On appelle *sous-circuit* d'un circuit C ce que l'on a envie d'entendre par sous-circuit : à savoir, une partie C' du circuit qui a la propriété que si une porte p en fait partie, alors toute porte qui émet un arc vers cette porte aussi. En particulier, le *sous-circuit principal* associé une porte p est formé de l'ensemble des portes de C qui sont sur un chemin qui arrive à p. Les *sous-circuits immédiats* d'un circuit C sont le ou les sous-circuits principaux correspondant à la ou aux deux portes portes émettant un arc vers la sortie.

On peut relier la notion de terme à la notion de circuit booléen.

Définition 7.3 (Terme booléen) *Un* terme booléen *est un circuit à une sortie dans lequel toute porte émet un arc et un seul.*

Autrement dit, le graphe a la forme d'un arbre.

Exemple 7.7 *Le circuit de l'exemple 7.3 n'est pas un terme. Le circuit de l'exemple 7.1 est un terme.*

Remarque 7.1 *Attention, graphiquement, il se représente aussi comme dans l'exemple 7.2, mais il s'agit bien d'un terme.*

7.1.2 Relations entre taille et profondeur

On note $e(C)$ pour le nombre d'entrées du circuit C.

Proposition 7.1 *Si aucune entrée n'est une sortie, alors* $taille(C) \leq 3(taille(C) - e(C))$.

Démonstration: Toute entrée possède au moins un arc sortant, qui va vers une porte p, qui n'est pas une entrée. Comme p reçoit au plus deux arcs, elle ne sert que pour deux entrées au plus, soit $e(C) \leq 2(taille(C) - e(C))$, soit $3e(C) \leq 2taille(C)$, ce qui revient à l'inégalité. □

Pour les termes, il y a forte corrélation entre taille et nombre d'entrées.

Proposition 7.2 *Si C est un terme T, alors* $2e(T) - 1 \leq taille(T)$ *avec égalité si toutes les portes qui ne sont pas des entrées reçoivent deux arcs entrants.*

Démonstration: Par récurrence sur la profondeur de T. C'est clair si T est de taille 1. Si T se décompose en deux termes immédiats T_1 et T_2, alors $e(T) = e(T_1) + e(T_2)$, puisque les sous-termes n'ont pas d'entrées communes, et $taille(T) = taille(T_1) + taille(T_2)$ puisqu'ils sont disjoints. La récurrence fonctionne bien dans ce cas. Si T ne possède qu'un sous-terme immédiat T_1 alors $e(T) = e(T_1)$, $taille(T) = taille(T_1)$ et la récurrence est encore valide. □

Proposition 7.3 *Pour tout circuit C,*

$$profondeur(C) \leq taille(C) - e(C),$$

$$taille(C) \leq (2^{profondeur(C)+1} - 1).$$

Pour tout terme T, $\log(e(T)) \leq profondeur(T)$.

Démonstration: La première inégalité vient du fait qu'un chemin maximal de longueur $profondeur(C)$ passe par $profondeur(C + 1)$ portes, et que la première est une entrée.

La deuxième inégalité se prouve par récurrence sur $profondeur(C)$. C'est clair si $profondeur(C) = 0$, car la taille vaut alors 1. Sinon, la sortie possède un ou deux sous-circuits immédiats.

S'il y en a qu'un C_1, alors $taille(C) = taille(C_1) - 1$, $profondeur(C_1) \leq profondeur(C_1) - 1$, et donc $taille(C) \leq 2^{profondeur(C)} - 1 + 1 \leq 2^{profondeur(C)+1} - 1$.

S'il y en a deux, $taille(C) \leq taille(C_1) + taille(C_2) + 1$, et C_1 et C_2 sont de profondeur au plus $profondeur(C) - 1$ si bien que $taille(C) \leq 2(2^{profondeur(C)} - 1) + 1 \leq 2^{profondeur(C)+1} - 1$.

Si C est un terme T, alors $2e(T) - 1 \leq taille(T)$ par la proposition 7.2, et donc $e(T) \leq 2^{profondeur(P)}$ par ce qui précède.

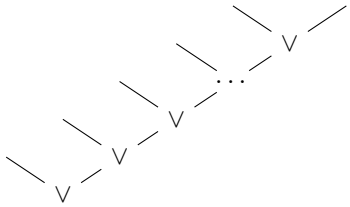
□

7.1.3 Effet Shannon

Commençons par une remarque simple :

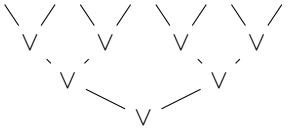
Lemme 7.1 *On peut construire un circuit de profondeur* $\log(n)+1$ *qui calcule la fonction disjonction à n variables : c'est-à-dire la fonction* $x_1 \vee x_2 \cdots \vee x_i \vee \cdots \vee x_n$*, notée* $\bigvee_{i=1}^{n} x_i$*, de* $\{0,1\}^n$ *dans* $\{0,1\}$*, qui vaut* **1** *si et seulement au moins l'un de ses arguments vaut* 1*.*

Démonstration: Une solution simple consisterait à construire un circuit avec uniquement des portes $\vee$ de la forme d'un peigne, c'est-à-dire de la forme



Cependant, un tel circuit n'est pas de profondeur logarithmique.

Il est plus astucieux de construire un arbre complet, ou presque complet.



Formellement, cela se prouve par récurrence sur n. Si $n = 2$, il suffit de considérer un circuit réduit à une porte $\vee$. Si $n > 2$, par hypothèse de récurrence, il existe un circuit de profondeur $\log(n)$ qui calcule la disjonction $\bigvee_{i=1}^{i=\lfloor n/2 \rfloor} x_i$, et un circuit de profondeur $\log(n)$ qui calcule la disjonction $\bigvee_{i=\lfloor n/2 \rfloor+1}^{n} x_i$. En ajoutant une porte $\vee$ qui prend en entrée ces deux circuits, on obtient un circuit de profondeur $\log(n)+1$. □

En remplaçant $\vee$ par $\wedge$ dans la preuve précédente, on a clairement aussi.

Lemme 7.2 *On peut construire un circuit de profondeur* $\log(n)+1$ *qui calcule la fonction conjonction à n variables : c'est-à-dire la fonction* $x_1 \wedge x_2 \cdots \wedge x_i \wedge \cdots \wedge x_n$*, notée* $\bigwedge_{i=1}^{n} x_i$*, de* $\{0,1\}^n$ *dans* $\{0,1\}$*, qui vaut* **1** *si et seulement chacun de ses arguments vaut* 1*.*

En fait, toute fonction peut se représenter, si l'on s'autorise une profondeur de l'ordre du nombre d'entrées, et de taille exponentielle.

Proposition 7.4 *Toute fonction booléenne en n variables peut se représenter par un terme booléen de profondeur au plus* $n+\log n+2$*, et de taille exponentielle en n.*

Démonstration: Si la fonction f est la constante 0, elle se représente par un circuit réduit à un sommet. Sinon, on considère chaque valeur $\overline{e} = (e_1, \cdots, e_n)$ telle que $f(\overline{e}) = 1$: à une telle valeur, on associe l'expression $\epsilon_1 x_1 \wedge \cdots \epsilon_i x_i \cdots \wedge \epsilon_n x_n$ où l'on convient que $\epsilon_i x_i$ vaut x_i si $e_i = 1$, et $\neg x_i$ si $e_i = 0$. Pour mettre cette expression sous la forme d'un circuit $C_{\overline{e}}$, il faut une profondeur 1 pour nier certaines variables, puis un arbre binaire de profondeur au plus $\log(n) + 1$ qui prend les conjonctions deux par deux comme dans les lemmes précédents.

$C_{\overline{e}}$ donne la valeur 1 au uplet $\overline{e}$ et la valeur 0 à tous les autres. f s'obtient comme la disjonction de tous les $C_{\overline{e}}$. Comme il y en a pas plus que 2^n, cela se fait en profondeur n, par les lemmes précédents. □

Exemple 7.8 *Pour la fonction* $=(x, y)$ *de* $\{0,1\}^2$ *dans* $\{0,1\}$ *qui vaut* 1 *si* $x = y$ *: on a* $\overline{e}_1 = (1,1)$*, qui correspond à l'expression* $x \wedge y$ *et* $\overline{e}_2 = (0,0)$ *qui correspond à l'expression* $\neg x \wedge \neg y$*. L'expression* $(x \wedge y) \vee (\neg x \wedge \neg y)$ *correspond donc à la fonction* $=$*.*

En fait, on ne peut pas espérer faire mieux. Il y a 2^{2^n} fonctions de $\{0,1\}^n$ dans $\{0,1\}$.

Combien il y a t'il de circuits de taille $p(n)$ avec seulement n entrées ? Il faut pour chaque paire de portes p et q décider s'il y a 0, 1 ou deux arcs de p vers q, ce qui ne laisse pas plus de $2^{p(n)^2 \log 3}$ choix pour placer ces arcs. Il faut ensuite étiqueter les portes par $\neg$, $\vee$, $\wedge$, 0, 1 ou $x_1, x_2, \cdots, x_n$, ce qui donne pas plus de $(n+6)^{p(n)}$ choix. Le nombre de ces circuits est donc majoré par $2^{p(n)^2 \log(3) + p(n) \log(n+6)}$. En conséquence si $p(n)$ est un polynôme en n, ou même une fonction négligeable devant 2^n, on est certain que pour n grand il y aura des fonctions n-aires non exprimables par un terme de taille $p(n)$, et même qu'elles constituent la majorité.

7.1.4 Bornes inférieures

On peut parfois obtenir des bornes inférieures.

Proposition 7.5 *L'expression de la fonction* $\sup(x_1, \cdots, x_n)$ *nécessite un circuit de profondeur au moins* $\log n$*.*

Démonstration: C'est la conséquence du résultat suivant : tout circuit acceptant tous les $00\cdots010\cdots00$ et refusant $00\cdots00$ possède une profondeur supérieure ou égale à $\log(n)$.

Par récurrence sur n. Cela est clair pour $n = 1$, car toute profondeur vaut au moins $0 = \log(1)$. Pour le cas général, on considère un tel circuit de profondeur minimal. Sa sortie est étiquetée par la négation, la porte précédant la sortie ne peut être étiquette par la négation car deux négations se neutralisent. L'antécédent est donc étiqueté par une conjonction ou une disjonction. En utilisant les lois de Morgan, on peut donc faire remonter les négations.

On peut donc supposer sans perte de généralité que l'étiquette de sortie est soit $\vee$ soit $\wedge$. Soient A et B ses circuits immédiats. Si l'étiquette est $\wedge$, l'un de ces deux sous-circuits au moins refuse $00\cdots00$, tandis que tous les deux doivent

accepter $00\cdots010\cdots00$. Autrement dit, l'un des deux possède la propriété ce qui contredit la minimalité de la profondeur.

L'étiquette de sortie est donc $\vee$. Soit X l'ensemble des variables x_i telles que A donne la valeur 1 sur l'entrée $00\cdots010\cdots00$, le 1 étant à la $i$ème place, et soit Y l'ensemble de variables ayant cette propriété pour B. On doit avoir $X \cup Y$ égal à toutes les entrées. Soit A' le circuit obtenu à partir de A en remplaçant par 0 toute variable hors de X, et B' celui obtenu à partir de B en remplaçant par 0 toute variable hors de Y. Comme A et B donnent 0 à la suite nulle, et que A' et B' ont strictement moins de variables (par minimalité de C), la récurrence s'applique et $profondeur(A') \geq \log(card(X)), profondeur(B') \geq \log(card(Y))$. Donc $profondeur(C) \geq \log(\max(card(X), card(Y)) + 1 \geq \log(n/2) + \log 2 = \log n$. □

7.1.5 Circuits à m sorties

On peut bien entendu considérer des circuits à plusieurs sorties : un *circuit booléen a n entrées et m-sorties* se définit exactement comme dans la définition 7.1, si ce n'est que l'on remplace la phrase "exactement une *sortie*", par "exactement m sorties".

Un circuit à n entrées et m sorties calcule une fonction $C(x)$ de $\{0,1\}^n$ dans $\{0,1\}^m$ comme on s'y attend : formellement, si l'on appelle C_i le sous-circuit correspondant à la sortie $1 \leq i \leq m$, le circuit C calcule la fonction qui à $\overline{x} = (x_1, \cdots, x_n)$ associe $C(\overline{x}) = (C_1(\overline{x}), C_2(\overline{x}), \cdots, C_m(\overline{x}))$.

7.1.6 Problème du temps parallèle

Théorème 7.1 (Spira) *Tout terme booléen de taille t est équivalent à un circuit (ou à un terme) de profondeur inférieure à $4\log(t)$.*

Le *problème du temps parallèle* est (le problème ouvert) de savoir si un calcul s'effectuant en temps séquentiel polynomial peut toujours s'effectuer, par une méthode éventuellement plus astucieuse, en temps parallèle logarithmique. En terme de circuits, cela se traduit ainsi : existe-t'il une constante A et une transformation qui à tout circuit C associe un circuit équivalent C^* telle que $profondeur(C^*) \leq A\log(taille(C))$?

Par le théorème de Spira, cela équivaut à la question suivante : existe-t'il une constante B et une transformation qui à tout circuit C associe un terme équivalent T telle que $taille(T) \leq taille(C)^B$?

7.1.7 Circuits sur une structure $\mathfrak{M}$

La notion de circuit se généralise à la notion de circuit sur une structure $\mathfrak{M}$ arbitraire.

Définition 7.4 (Circuit sur une structure) *Soit n un entier. Un* circuit booléen a n entrées et m sorties sur la structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ *est un graphe orienté sans cycle (*DAG, directly oriented graph*) avec*

- n entrées, *c'est-à-dire n sommets sans arc entrant.*
- *exactement* m sorties, *c'est-à-dire m sommets sans arc sortant.*
- *chaque entrée est étiquetée soit par une constante de la structure* $\mathfrak{M}$*, par un élément de* M *ou par un symbole de variable* $x_1, x_2, \cdots, x_n$*.*
- *tout autre sommet est appelé une* porte *et est étiqueté par une fonction ou une relation de la structure. Le fanin*[1] *de chaque sommet correspond à l'arité du symbole de fonction ou de relation avec lequel il est étiqueté.*

Une constante étiquetant une entrée du circuit qui ne correspond pas à une fonction 0-aire de la structure est qualifiée de *paramètre*.

Nous laissons au lecteur le soin de définir l'évaluation d'un circuit sur un élément de M^n.

7.2 Circuits et algorithmes

Nous allons maintenant utiliser la notion de circuit comme un modèle de calcul et comparer ce modèle aux algorithmes.

7.2.1 Principe fondamental

En fait, la remarque fondamentale est la suivante.

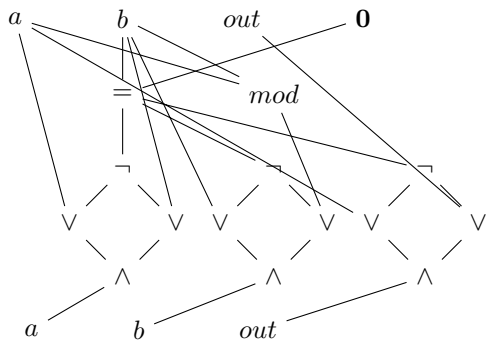
Proposition 7.6 *Soit A un algorithme (respectivement : sur la structure* $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$*). Soit n un entier. Soit t un entier.*

Il existe un circuit $C_{n,t}$ *(resp. sur la structure* $\mathfrak{M}$*) qui accepte exactement les mots de longueur n acceptés par l'algorithme A en (respectivement : moins de) t étapes.*

Commençons par un cas simple, l'algorithme d'Euclide.

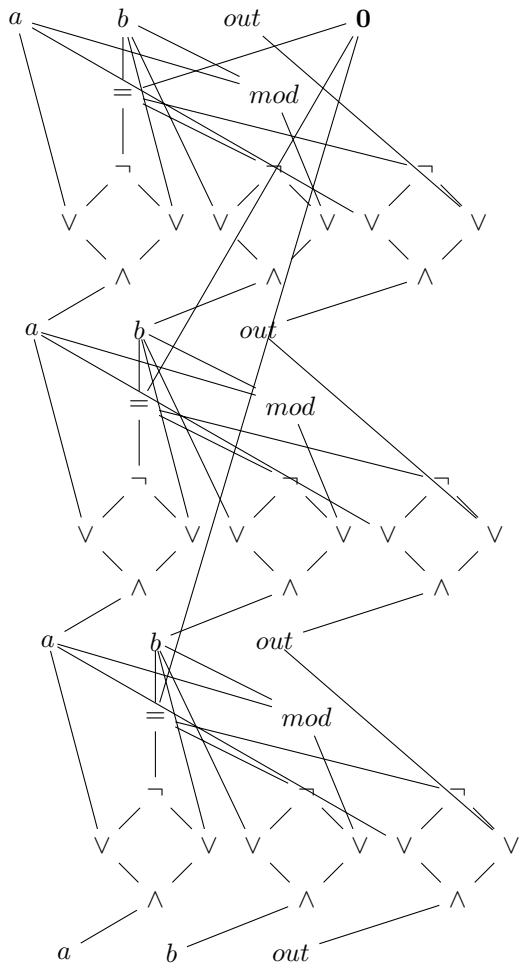
Exemple 7.9 *Chaque itération de l'algorithme d'Euclide du chapitre 4 remplace* (a, b, out) *par* $(S(b = 0, a, b), S(b = 0, b, a \bmod b), S(b = 0, a, out))$.

Une itération de l'algorithme correspond donc au circuit suivant :



[1] On ordonne aussi les arcs entrants de façon à savoir quel est le premier, le second, ...etc argument de la fonction, puisque les fonctions ne sont plus nécessairement symétriques.

On obtient le résultat de t itérations en répétant ce circuit : par exemple, pour $t = 3$.



Pour prouver ce résultat dans le cas général, commençons par quelques observations.

Selon la discussion du chapitre 4, à tout algorithme correspond un système de transitions, dont les états, aussi appelés configurations, correspondent à des structures.

Rappelons qu'un emplacement $(f, \overline{m})$ d'une structure est dit *utile*, si f est un symbole de fonction dynamique, et son contenu $[\![(f, \overline{m})]\!]$ n'est pas **undef**.

Lemme 7.3 (Codage des emplacements utiles d'une configuration) *On peut coder les emplacements utiles d'une structure C par un mot $[C]$ sur l'alphabet M de longueur de l'ordre du nombre d'emplacements utiles de cette structure.*

Démonstration: En effet, la signature Σ de l'algorithme possède un nombre fini de symboles de fonctions et de relations : on peut donc coder chaque symbole f de fonction ou de relation par un mot en binaire de $\{\mathbf{0},\mathbf{1}\}^*$. On notera $< f >$ le codage du symbole f. Quitte à ajouter des $\mathbf{0}$ à gauche, on peut supposer que la longueur de ce codage ne dépend pas du symbole f. Pour chaque emplacement utile $(f,\overline{m})$, on peut coder l'emplacement et son contenu $[\![(f,\overline{m})]\!]$ par le mot $w_{f,\overline{m}} =< f > \overline{m}1\cdots 1[\![(f,\overline{m})]\!]$, où les 1 sont utilisés pour rendre la longueur ℓ de ce mot indépendante du symbole f. Les emplacements utiles d'une structure C peuvent se coder alors par le mot $[C] = w_{f_1,\overline{m}_1} w_{f_2,\overline{m}_2}\cdots w_{f_k,\overline{m}_k}$. □

Lemme 7.4 (Passage d'une configuration à sa successeur) *On peut construire un circuit qui envoie le codage $[C]$ d'une configuration sur le codage $[C']$ de sa configuration successeur par le système de transition associé à l'algorithme.*

Ce circuit possède une taille de l'ordre du nombre d'emplacements utiles dans la configuration C.

Supposons ce lemme prouvé. La proposition 7.6 en découle. En effet :

Démonstration: Si l'on note C_t la configuration de l'algorithme au temps t, il suffit de construire par le lemme précédent pour chaque $t' = 0, t' = 1, \cdots, t' = t-1$ un circuit qui calcule la fonction qui envoie $C_{t'}$ sur $C_{t'+1}$. En partant de la configuration initiale C_0, en recopiant ce circuit, comme dans l'exemple d'Euclide, pour $t' = 0, 1, \cdots, t-1$, on obtiendra un circuit qui calcule la configuration au temps t, et donc le résultat. □

Il reste à prouver le lemme 7.4

Démonstration: Pour se convaincre que l'on peut calculer $[C']$ en fonction de $[C]$ par circuit, voici quelques fonctions, réalisables par circuits, qui nous seront utiles : (nous confondrons ici mots de longueur k et k-uplets, par abus de notations).

- EQUAL$(\overline{x},\overline{y})$, où $\overline{x} = (x_1,\cdots,x_n)$ et $\overline{y} = (y_1,\cdots,y_n)$ sont des vecteurs de même longueur, qui vaut 1 si et seulement si $\overline{x} = \overline{y}$: EQUAL est donnée par le terme $\bigwedge_{i=1}^{n} = (x_i, y_i)$.
- REPLACE$(s,\overline{x},\overline{y})$, où $\overline{x} = (x_1,\cdots,x_n)$ et $\overline{y} = (y_1,\cdots,y_n)$ sont des vecteurs de même longueur, qui vaut $\overline{y}$ si $s = 1$ et $\overline{x}$ sinon : s'obtient avec un circuit dont la sortie i vaut $S(s, y_i, x_i)$.
- EVALUE$(< f >, \overline{m}, C_t)$, où $(f,\overline{m})$ est un emplacement, qui retourne son contenu dans la configuration C_t : EVALUE se définit récursivement en fonction de la longueur de C_t. Pour $|C_t| < l$, EVALUE est réduit au circuit réduit au sommet $undef$. Pour $|C_t| \geq l$, notons $C_t = w_{\ell-1} v C'_t$, où $w_{\ell-1}$ est le préfixe de longueur $\ell - 1$, et v un élément de l'alphabet M. Dans ce cas, EVALUE$(< f >, \overline{m}, C_t)$ est donnée par le circuit

$$S(\text{EQUAL}(< f >, \overline{m}, 1, \cdots, 1, w_{\ell-1}), v,$$

$$S(\text{EQUAL}(w_\ell, 1, \cdots, 1), undef,$$

$$\text{EVALUE}(< f >, \overline{m}, C'_t)).$$

où les $1\cdots 1$ sont en nombre suffisant pour que ces circuits aient un sens.

– Soit $t = f(t_1, \cdots, t_k)$ un terme. $\mathrm{TVALUE}_t(C_t)$ retourne la valeur du terme t dans la configuration C_t : TVALUE_t est donnée par le circuit

$$\mathrm{EVALUE}(< f >, \mathrm{TVALUE}_{t_1}(C_t), \cdots, \mathrm{TVALUE}_{t_k}(C_t), C_t),$$

si f est un symbole dynamique, et par

$$f(\mathrm{TVALUE}_{t_1}(C_t), \cdots, \mathrm{TVALUE}_{t_k}(C_t))$$

sinon.

– Soit f un symbole. $\mathrm{UPDATE}_f(\overline{m}, b, C_t)$ retourne la configuration obtenue en effectuant la mise à jour $(f, \overline{m}) \leftarrow b$ sur C_t : se définit récursivement en fonction de la longueur de C_t. Pour $|C_t| < l$, UPDATE_f est réduit au circuit qui produit $< f > \overline{m}1 \cdots 1b$. Pour $|C_t| \geq l$, notons $C_t = w_{\ell-1} v C'_t$, où $w_{\ell-1}$ est le préfixe de longueur $\ell - 1$, et v un élément de l'alphabet M. Dans ce cas, $\mathrm{UPDATE}_f(\overline{m}, b, C_t)$ est donnée par le circuit

$$REPLACE(\mathrm{EQUAL}(< f >, \overline{m}, 1, \cdots, 1, w_\ell),$$

$$REPLACE(\mathrm{EQUAL}(w_\ell, 1, \cdots, 1), < f > w_{\ell-1} v, < f > \overline{m}1 \cdots 1b),$$

$$< f > \overline{m}1 \cdots 1b)$$

où les $1 \cdots 1$ sont en nombre suffisant pour que ces circuits aient un sens.

– Soit t← t' une instruction d'affectation. $\mathrm{ASSIGN}_{t:=t'}(C_t)$ retourne la configuration obtenue en effectuant cette instruction : si t s'écrit $t = f(t_1, \cdots, t_k)$, s'obtient par le circuit

$$\mathrm{UPDATE}_f(\mathrm{TVALUE}_{t_1}(C_t), \cdots, \mathrm{TVALUE}_{t_k}(C_t), \mathrm{TVALUE}_{t'}(C_t), C_t).$$

– Soit **par** R **endpar** la mise en parallèle d'instructions d'affectation consistantes. $\mathrm{PAR}_R(C_t)$ retourne la configuration obtenue en effectuant cette instruction : s'obtient en mettant en parallèle les circuits ASSIGN correspondants.

– Soit finalement une instruction **if** ϕ **then** R, où R est de la forme précédente : $\mathrm{DO}(C_t)$ retourne la configuration obtenue en effectuant cette instruction : s'obtient comme $\mathrm{REPLACE}(\mathrm{TVALUE}_\phi(C_t), C_t, \mathrm{PAR}_R(C_t))$

– Le passage de C_t à C_{t+1} s'obtient en mettant en parallèle chacun de ces circuits.

□

En fait, on peut même dire plus : rappelons qu'un circuit correspond à un graphe étiqueté. Il se code donc par un mot, en utilisant les principes du chapitre 2.

Proposition 7.7 *La taille du circuit $C_{n,t}$ est polynomiale en n et en t .*

On peut construire la suite $C_{n,t}$ effectivement : on peut construire un algorithme qui prend en entrée les entiers n et t et qui produit en sortie le codage $< C_{n,t} >$ de $C_{n,t}$.

Démonstration: La première observation résulte du fait que puisque chaque itération de l'algorithme (supposé en forme normale) ne modifie qu'un nombre fini d'emplacements, le nombre d'emplacements utile dans la configuration C_t au temps t reste affine en t, majoré par $k.t + n$, où k est une constante.

La seconde consiste simplement à observer que la construction de la preuve plus haut peut se traduire comme la description d'un algorithme pour construire $C_{n,t}$ en fonction de n et de t. □

7.2.2 Premières applications : indécidabilité

Théorème 7.2 *Tout ensemble semi-décidable sur la structure*

$$(\mathbb{R}, +, -, *, /, =, <)$$

possède un nombre dénombrable de composantes connexes.

Démonstration: En vertu des propositions précédentes, un ensemble semi-décidable doit s'écrire comme une union dénombrable de langages correspondant à l'union sur t et sur n des mots acceptés par les circuits $C_{n,t}$.

Un circuit sur la structure $(\mathbb{R}, +, -, *, /, =, <)$ reconnaît un ensemble *semi-algébrique* : c'est-à-dire un ensemble qui se définit comme une combinaison booléenne d'inégalités et d'égalités polynomiales.

On admettra que tout ensemble semi-algébrique possède un nombre fini de composantes connexes. □

Corollaire 7.1 *L'ensemble de Mandelbrot n'est pas décidable sur structure*

$$(\mathbb{R}, +, -, *, /, =, <)$$

Démonstration: Cela découle du fait (admis) que le complémentaire de l'ensemble de Mandelbrot ne possède pas un nombre dénombrable de composantes connexes. □

7.2.3 Premières applications : bornes inférieures

Théorème 7.3 *Le problème du sac à dos réel (on se donne des réels $x_1, x_2, \cdots, x_n \in \mathbb{R}$ et on souhaite décider s'il existe un sous ensemble $S \subset \{1, 2, \cdots, n\}$ tel que $\sum_{i \in S} x_i = 1$) ne peut pas se résoudre en temps polynomial en n (c'est-à-dire avec un nombre d'étapes polynomial en n) sur la structure $(\mathbb{R}, \mathbf{0}, \mathbf{1}, +, -, =)$.*

Démonstration: Observons qu'un circuit sur la signature $(\mathbb{R}, \mathbf{0}, \mathbf{1}, +, -, =)$ accepte un ensemble qui s'écrit comme une combinaison booléenne de relations affines (de *tests*) de la forme $\lambda_1 x_1 + \cdots + \lambda_n x_n = a$.

Supposons que le problème soit résolu par un algorithme en temps polynomial en n. Sur chaque instance x, le circuit correspondant effectue un certain nombre m de tests, avec m qui reste polynomial en n.

Parce que l'on peut effectuer qu'un nombre fini de tests en temps fini, il doit exister une instance négative x du problème du sac à dos réel tel que le résultat

des m relations affines $l_1(x), l_2(x), \cdots, l_m(x)$ du circuit sur cette instance soient tous négatifs.

Puisque m est polynomial en n, et que $\{0,1\}^n$ est de taille 2^n, il doit exister $\epsilon \in \{0,1\}^n$ tel que $\epsilon_1 x_1 + \cdots + \epsilon_n x_n = 1$ ne soit pas l'un des m tests. L'algorithme se trompe sur un $y \in \mathbb{R}^n$ tel que $\epsilon_1 y_1 + \cdots + \epsilon_n y_n = 1$ et tel que $l_1(y) \neq 0, \ldots, l_m(y) \neq 0$: un tel y existe car les m dernières contraintes enlèvent un espace vectoriel de dimension $n-2$ à l'hyperplan défini par la première contrainte. En effet, le résultat des tests sur x et sur y de chacun des tests est le même et pourtant x doit être refusé, alors que y doit être accepté. □

7.3 Notes bibliographiques

Les résultats de ce chapitre sur les circuits sont très largement inspirés de [Poizat, 1995]. L'application à l'indécidabilité de l'ensemble de Mandelbrot est inspirée de [Blum et al., 1989] et de [Blum et al., 1998]. Le lien entre les circuits et les algorithmes est un résultat classique développé dans la plupart des ouvrages de complexité. L'application au problème du sac à dos réel est repris de [Koiran, 1994].

Chapitre 8

Complexité en temps

8.1 Temps déterministe

8.1.1 Mesure du temps de calcul

Selon la discussion du chapitre 4, à un algorithme A correspond un système de transitions, c'est-à-dire un ensemble d'états $S(A)$ et une fonction d'évolution $\tau_A : S(A) \to S(A)$ qui décrit les évolutions entre ces états. Soit w une entrée d'un algorithme A, sur lequel A termine : le *temps de calcul* de l'algorithme A sur l'entrée w est le nombre d'étapes du système de transitions associé.

Plus formellement,

Définition 8.1 (Temps de calcul) *Soit A un algorithme. A chaque entrée w est associée une exécution $X_0, X_1, \cdots, X_t$ du système de transitions associé à A où X_0 est un état initial qui code w, et chaque X_i est un état, et $X_{i+1} = \tau_A(X_i)$ pour tout i. Supposons que w soit acceptée, c'est-à-dire qu'il existe un entier t avec X_t terminal. Le* temps de calcul sur l'entrée w, *noté* $\mathrm{TIME(A, w)}$, *est défini comme cet entier t.*

Si l'on observe qu'un algorithme mis sous la forme normale du chapitre 4 n'effectue qu'un nombre fini d'opérations par exécution d'une itération de la boucle englobante **repeat until**, on peut se convaincre que cela correspond à la notion intuitive de temps de calcul pour un algorithme, éventuellement à une constante multiplicative près. On raisonnera dans la suite de toute façon toujours à une constante multiplicative près.

Comme il en est l'habitude en informatique, on mesure en fait les complexités en fonction de la taille des entrées. Rappelons que si w est un mot sur un alphabet M, $|w|$ désigne sa longueur, c'est-à-dire son nombre de lettres (symboles).

Définition 8.2 (Temps de calcul) *Soit A un algorithme qui termine sur toute entrée. Le* temps de calcul *de l'algorithme A est la fonction $f : \mathbb{N} \to \mathbb{N}$ telle*

que pour tout entier n, $f(n)$ est le maximum de $\mathrm{TIME}(\mathrm{A}, \mathrm{w})$ *pour les entrées (mots) w de longueur n. Autrement dit,*

$$f(n) = \max_{|w|=n} \mathrm{TIME}(\mathrm{A}, \mathrm{w}).$$

8.1.2 Notation $\mathrm{TIME}(\mathrm{t}(\mathrm{n}))$

Puisque calculer précisément la fonction f est souvent problématique, comme il en est l'habitude en informatique, on travaille souvent à un ordre de grandeur près, via les notations $\mathcal{O}$. Rappelons les définitions suivantes :

Définition 8.3 (Notation $\mathcal{O}$) *Soient f et g deux fonctions $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$. On note $f(n) = \mathcal{O}(g(n))$ lorsqu'il existe des entiers c et n_0 tels que pour tout $n \geq n_0$,*

$$f(n) \leq cg(n).$$

Intuitivement, cela signifie que f est inférieur à g à une constante multiplicative près, pour les instances (données) de tailles suffisamment grandes.

De même on définit :

Définition 8.4 (Notations o, Ω, Θ) *Soient f et g deux fonctions $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$.*

- *On note $f(n) = o(g(n))$ lorsque pour tout réel c, il existe un entier n_0 tels que pour tout $n \geq n_0$,*
$$f(n) < cg(n).$$
- *On note $f(n) = \Omega(g(n))$ lorsqu'il existe des entiers c et n_0 tels que pour tout $n \geq n_0$,*
$$f(n) \geq cg(n).$$
- *On note $f(n) = \Theta(g(n))$ lorsque $f(n) = \mathcal{O}(g(n))$ et $f(n) = \Omega(g(n))$.*

On cherche alors à déterminer les problèmes ou les langages qui admettent une solution en un certain temps. On utilisera pour cela la notation suivante :

Définition 8.5 *Soit $t : \mathbb{N} \to \mathbb{N}$ une fonction. On définit la classe* $\mathrm{TIME}(\mathrm{t}(\mathrm{n}))$ *comme la classe des problèmes ou des langages qui sont solvables par un algorithme qui fonctionne en temps $\mathcal{O}(t(n))$. Si l'on préfère,*

$$\mathrm{TIME}(\mathrm{t}(\mathrm{n})) = \{L | L \text{ est un langage décidé par}$$

$$\text{un algorithme en temps } \mathcal{O}(t(n))\}.$$

Bien entendu, on peut aussi considérer les algorithmes sur une structure donnée. Il y a cependant une petite subtilité : pour certaines raisons que l'on comprendra qu'ultérieurement, on autorise par défaut les algorithmes à utiliser des paramètres, c'est-à-dire des éléments de la structure.

Définition 8.6 *Soit $t : \mathbb{N} \to \mathbb{N}$ une fonction. Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure.*

On définit la classe $\mathrm{TIME}^0_{\mathfrak{M}}(\mathrm{t(n)})$ *comme la classe des problèmes ou des langages qui sont solvables par un algorithme sur la structure $\mathfrak{M}$ qui fonctionne en temps $\mathcal{O}(t(n))$. Si l'on préfère,*

$$\mathrm{TIME}^0_{\mathfrak{M}}(\mathrm{t(n)}) = \{L | L \text{ est un langage décidé par}$$

$$\text{un algorithme sur } \mathfrak{M} \text{ en temps } \mathcal{O}(t(n))\}.$$

Définition 8.7 *Soit $t : \mathbb{N} \to \mathbb{N}$ une fonction. Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure.*

Un algorithme à paramètres sur la structure $\mathfrak{M}$ est un algorithme sur une structure $\mathfrak{M}' = (M, f_1, \cdots, f_u, c_1, \cdots, c_k, r_1, \cdots, r_v)$, où $c_1, \cdots, c_k$ sont des symboles de constantes. Autrement dit, un algorithme à paramètres sur la structure $\mathfrak{M}$ est un algorithme sur une structure obtenue en étendant $\mathfrak{M}$ par un nombre fini de constantes.

On définit la classe $\mathrm{TIME}_{\mathfrak{M}}(\mathrm{t(n)})$ *comme la classe des problèmes ou des langages qui sont solvables par un algorithme avec paramètres sur la structure $\mathfrak{M}$ qui fonctionne en temps $\mathcal{O}(t(n))$.*

$$\mathrm{TIME}_{\mathfrak{M}}(\mathrm{t(n)}) = \{L | L \text{ est un langage décidé par}$$

$$\text{un algorithme à paramètres sur } \mathfrak{M} \text{ en temps } \mathcal{O}(t(n))\}.$$

8.1.3 Version effective de la thèse de Church

Dans le chapitre 5, nous avons introduit différents modèles de calculs, et nous avons montré qu'ils se simulaient l'un et l'autre. Il est en fait possible d'en dire plus et de montrer que, si l'on met de côté les machines à compteurs, et les machines RAM telles que nous les avons définies, les simulations préservent le temps polynomial :

Définition 8.8 *On dira qu'une classe de modèles $\mathcal{M}_1$ simule une classe de modèles $\mathcal{M}_2$* en temps polynomial*, si chaque élément m_2 de $\mathcal{M}_2$ est simulé par un élément m_1 de $\mathcal{M}_1$, de telle sorte qu'il existe un polynôme p (qui peut dépendre de m_1 et m_2) tel qu'un temps de calcul $t(n)$ de m_2 soit simulé en un temps $p(t(n))$ par la machine m_1 correspondante.*

Théorème 8.1 *Sur une structure arbitraire $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$, les classes de modèles suivants se simulent deux à deux en temps polynomial :*

1. *Les machines de Turing*
2. *Les machines à $k \geq 2$ piles*
3. *Les algorithmes au sens du chapitre 4*

Démonstration: Nous avons montré dans le chapitre 5 qu'une machine de Turing se simulait par une machine à $k \geq 2$ piles : un examen de la preuve montre que la simulation se fait en un temps linéaire.

Nous avons montré d'autre part qu'une machine à $k \geq 2$ piles correspondait à un algorithme : la simulation se fait là aussi en temps linéaire.

Nous avons montré qu'un algorithme pouvait être simulé par une machine de Turing : la simulation de chacune des étapes de l'algorithme nécessite un nombre fini et constant de parcours du ruban de la machine de la simulation. Si l'on appelle T le temps de l'algorithme simulé, au temps T le ruban reste de taille $\mathcal{O}(T)$, et donc la simulation du temps T se fait en temps $\mathcal{O}(T^2)$. Un algorithme en temps $t(n)$ est donc simulé en temps $t(n)^2$ par la machine de Turing.

Le résultat suit par transitivité de la notion de simulation, et de la stabilité des polynômes par composition. □

Remarque 8.1 *Notons qu'il est possible de définir une variante du modèle de machine RAM du chapitre 5 qui serait aussi équivalente à chacun des modèles du théorème précédent à un temps polynomial près.*

8.1.4 Classe P

Les résultats précédents invitent à considérer la classe suivante.

Définition 8.9 P *est la classe des langages et des problèmes décidés par un algorithme en temps polynomial. En d'autres termes,*

$$\mathrm{P} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME}(\mathrm{n}^{\mathrm{k}}).$$

On dira aussi qu'un problème ou un langage de P est *calculable en temps polynomial*. En théorie de la complexité, on considère souvent que la notion de temps raisonnable correspond au temps polynomial.

Exemple 8.1 (Problème REACH**)** *Étant donné un graphe orienté $G = (V, E)$, et deux sommets $u, v \in V$, on peut déterminer en temps polynomial s'il existe un chemin entre u et v.*

Par exemple, en effectuant un parcours en profondeur à partir de u. S'il on atteint v dans ce parcours en profondeur, on sait que v est atteignable. Sinon, v n'est pas atteignable. Cela fonctionne en un temps de l'ordre du maximum du nombre d'arêtes du graphe, et du nombre de sommets du graphe.

Avec nos notations, pour le langage

$$\mathrm{REACH} = \{< G, u, v > | G \text{ est un graphe orienté avec un chemin entre les sommets } u \text{ et } v\}$$

on peut écrire

$$\mathrm{REACH} \in \mathrm{P}.$$

On peut généraliser cette notion aux structures arbitraires :

Définition 8.10 *Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure.*

$\mathrm{P}^0_{\mathfrak{M}}$ *est la classe des langages et des problèmes décidés par un algorithme sur la structure $\mathfrak{M}$ en temps polynomial. En d'autres termes,*

$$\mathrm{P}^0_{\mathfrak{M}} = \bigcup_{k\in\mathbb{N}} \mathrm{TIME}^0_{\mathfrak{M}}(\mathrm{n}^{\mathrm{k}}).$$

$\mathrm{P}_{\mathfrak{M}}$ *est la classe des langages et des problèmes décidés par un algorithme à paramètres sur la structure $\mathfrak{M}$ en temps polynomial. En d'autres termes,*

$$\mathrm{P}_{\mathfrak{M}} = \bigcup_{k\in\mathbb{N}} \mathrm{TIME}_{\mathfrak{M}}(\mathrm{n}^{\mathrm{k}}).$$

Exemple 8.2
- *Le problème* CIRCUITVALUE $= \{< C, \overline{x} > | C$ *est un circuit booléen et* $C(\overline{x}) = 1\}$ *est dans* P.
- *Le problème* CIRCUITVALUE $= \{< C, \overline{x} > | C$ *est un circuit sur* $\mathfrak{M}$ *et* $C(\overline{x}) = 1\}$ *est dans* $\mathrm{P}^0_{\mathfrak{M}}$ *et dans* $\mathrm{P}_{\mathfrak{M}}$.

En effet, pour déterminer si une entrée doit être acceptée, il suffit de simuler le calcul du circuit sur l'entrée $\overline{x}$, ce qui se fait en temps polynomial.

8.2 Liens avec les circuits

8.2.1 Reconnaissance par une famille de circuits

Un circuit booléen C à n entrées reconnaît un sous-ensemble de $\{0,1\}^n$: il reconnaît les mots $\overline{x} \in \{0,1\}^n$ tels que $C(\overline{x}) = 1$.

Avec un circuit, on peut donc parler uniquement de mots d'une longueur fixée.

Pour pouvoir parler de mots de longueurs arbitraires, on utilise des familles de circuits indexées par les entiers. Rappelons que si w est un mot, $|w|$ désigne sa longueur.

Définition 8.11 (Langage accepté par une famille de circuits) *Soit $(C_n)_{n\in\mathbb{N}}$ une famille de circuits booléens (respectivement : sur une structure $\mathfrak{M}$), où le circuit C_n possède exactement n entrées et 1 sortie. Le* langage L reconnu par cette famille *est défini par la propriété suivante : $w \in L$ si et seulement si le mot w est accepté par le circuit $C_{|w|}$.*

Autrement dit, on considère que l'on a une famille de circuits, un par taille d'entrée, et que chaque circuit C_n donne la réponse sur les entrées de taille n.

Il est alors naturel de s'intéresser aux familles de circuits de taille polynomiale.

Définition 8.12 (Classe $\mathbb{P}$) *On dira qu'un langage est dans la classe $\mathbb{P}$ s'il est reconnu par une famille de circuits booléens de taille polynomiale : L est reconnu par une famille de circuits $(C_n)_{n\in\mathbb{N}}$ et il existe un entier k tel que $taille(C_n) = \mathcal{O}(n^k)$.*

Définition 8.13 (Classe $\mathbb{P}^0_{\mathfrak{M}}$) *Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure.*

On dira qu'un langage est dans la classe $\mathbb{P}^0_{\mathfrak{M}}$ s'il est reconnu par une famille de circuits sur $\mathfrak{M}$ de taille polynomiale : L est reconnu par une famille de circuits $(C_n)_{n\in\mathbb{N}}$ et il existe un entier k tel que taille$(C_n) = \mathcal{O}(n^k)$.

Définition 8.14 (Classe $\mathbb{P}_{\mathfrak{M}}$) *Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure.*

On dira qu'un langage est dans la classe $\mathbb{P}_{\mathfrak{M}}$ s'il est reconnu par une famille de circuits sur $\mathfrak{M}$ à paramètres de taille polynomiale : il existe un ensemble fini d'éléments $c_1, c_2, \cdots, c_k \in M$, tel que L est reconnu par une famille de circuits $(C_n)_{n\in\mathbb{N}}$ avec les paramètres $c_1, c_2, \cdots, c_k$ (c'est-à-dire que $w \in L$ si et seulement $C_{|w|+k}(c_1, c_2, \cdots, c_k, w) = 1$) et il existe un entier k tel que taille$(C_n) = \mathcal{O}(n^k)$.

8.2.2 Temps polynomial et circuits de taille polynomiale

Théorème 8.2 *Soit A un algorithme (respectivement : sur la structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$), qui termine sur toute entrée en temps $t(n)$.*

Alors :

1. *Il existe alors une famille $(C_n)_{n\in\mathbb{N}}$ de circuits booléens (resp : de circuits sur la structure $\mathfrak{M}$), de taille $\mathcal{O}(t(n)^k)$ pour un certain entier k, qui reconnaît le langage décidé par A.*
2. *La fonction qui à l'entier n associe la description $< C_n >$ du circuit C_n est calculable en un temps polynomial en n.*

Le point 2. est ce que l'on appelle une condition *d'uniformité* : la suite des circuits est calculable, même calculable en temps polynomial en n.

Remarque 8.2 *Dans la deuxième condition, on parle bien d'un temps polynomial en l'entier n, et pas en son codage. Si l'on veut être formel, la condition 2. devrait être : la fonction qui à $\mathbf{1}^n$ (le mot constitué de n fois la lettre 1) associe la description $< C_n >$ du circuit C_n est calculable en un temps polynomial.*

Remarque 8.3 *Avec cette formulation, il n'y a pas ambiguïté, car le codage en binaire de $\mathbf{1}^n$ est de longueur n.*

Remarque 8.4 *Remarquons que le codage d'un circuit (sans paramètre) même sur une structure arbitraire, peut se voir comme un mot sur $\{\mathbf{0}, \mathbf{1}\}$: on peut considérer qu'il s'agit en fait d'un algorithme classique, dans la condition 2.*

Démonstration: Il s'agit d'une application directe de la proposition 7.6 du chapitre 7. En effet, cette dernière proposition stipule l'existence d'une famille $(C_{n,t})_{n,t}$, où $C_{n,t}$ accepte les mots de longueur n acceptés en moins de t étapes. Considérer la famille $(C_n)_{n\in\mathbb{N}}$ donnée par $C_n = C_{n,t(n)}$ pour tout n. □

En fait, le théorème 8.2 admet une réciproque qui fournit une caractérisation du temps polynomial.

Théorème 8.3 *Un problème est dans* P *si et seulement si*

1. *il est dans* $\mathbb{P}$
2. *la famille de circuits booléens correspondante est calculable en temps polynomial : on peut produire* $< C_n >$ *en un temps polynomial en* n.

Plus généralement :

Théorème 8.4 *Soit* $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ *une structure.*
Un problème est dans $\mathrm{P}^0_{\mathfrak{M}}$ *si et seulement si*

1. *il est dans* $\mathbb{P}^0_{\mathfrak{M}}$
2. *la famille de circuits booléens correspondante est calculable en temps polynomial : on peut produire* $< C_n >$ *en un temps polynomial en* n *par un algorithme classique.*

Théorème 8.5 *Soit* $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ *une structure.*
Un problème est dans $\mathrm{P}_{\mathfrak{M}}$ *si et seulement si*

1. *il est dans* $\mathbb{P}_{\mathfrak{M}}$
2. *la famille de circuits booléens correspondante est calculable en temps polynomial avec paramètres : on peut produire* $< C_n >$ *en un temps polynomial en* n *par un algorithme à paramètres sur la structure* $\mathfrak{M}$.

Démonstration: Le sens direct de chacun des théorèmes est donné par le théorème 8.2, en prenant $t(n)$ polynôme en n. Réciproquement, soit L un langage dans $\mathbb{P}$ (respectivement : $\mathbb{P}^0_{\mathfrak{M}}$, $\mathbb{P}_{\mathfrak{M}}$). On peut déterminer si $w \in L$ par l'algorithme qui consiste à produire le circuit $C_{|w|}$ (resp. $C_{|w|+k}$) puis à simuler ce circuit sur l'entrée w (resp. avec ses paramètres $c_1, \cdots, c_k$). Par hypothèse, tout cela se fait en temps polynomial, car CIRCUITVALUE est polynomial. □

8.3 Temps non-déterministe

8.3.1 Classe NP

On formalise maintenant la notion de *solution qui soit vérifiable efficacement* en introduisant la classe NP.

Définition 8.15 (Classe NP**)** *Soit* M *un alphabet fini. Un problème* $L \subset M^*$ *est dans* NP *s'il existe un polynôme* $p : \mathbb{N} \to \mathbb{N}$ *et un problème* A *(appelé* vérificateur pour L*) tel que pour tout mot* w*,*

$$w \in L \text{ si et seulement si } \exists u \in M^* \text{ tel que } < w, u > \in A,$$

et tel que déterminer si $< w, u > \in A$ *admette un algorithme en temps* $p(|w|)$.

Pour $w \in L$, et un $u \in M^*$, avec $< x, u > \in L$, on dit que u est *un certificat* pour l'entrée w (vis à vis du problème L et de A).

Cela peut aussi s'écrire :

Définition 8.16 (Classe NP**)** *Soit M un alphabet fini. Un problème $L \subset M^*$ est dans* NP *s'il existe un polynôme $p : \mathbb{N} \to \mathbb{N}$ et un problème A dans* P *(appelé* vérificateur pour L*) tel que pour tout mot w,*

$$w \in L \text{ si et seulement si } \exists u \in M^*, \text{ tel que } |u| \leq p(|w|) \text{ et } < w, u > \in A.$$

Les deux définitions sont équivalentes : en effet, être polynomial en $| < w, u > |$ implique être polynomial en $|w|$ d'une part, et d'autre part, en temps polynomial en $|w|$ on ne peut lire qu'un nombre polynomial de lettres de u, et donc on peut supposer le certificat réduit aux lettres que l'on lit, c'est-à-dire en un certificat de longueur polynomiale.

Par construction, on a :

Proposition 8.1 $\mathrm{P} \subset \mathrm{NP}$

La question de savoir si l'inclusion est stricte, est une des questions fondamentales ouvertes les plus célèbres de l'informatique théorique.

8.3.2 Classes $\mathrm{NP}_{\mathfrak{M}}$ et $\mathrm{NDP}_{\mathfrak{M}}$

Cela se généralise aux structures arbitraires, mais il y a une subtilité, selon que l'on quantifie sur les mots sur l'alphabet M comme ci-dessus, ou sur les mots constitués de **0** et de **1**.

Définition 8.17 (Classe $\mathrm{NDP}_{\mathfrak{M}}$**)** *Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure. Un problème $L \subset M^*$ est dans $\mathrm{NP}_{\mathfrak{M}}$ s'il existe un polynôme $p : \mathbb{N} \to \mathbb{N}$ et un problème A dans dans $\mathrm{P}_{\mathfrak{M}}$ (appelé* vérificateur pour L*) tel que pour tout mot w,*

$$w \in L \text{ si et seulement si } \exists u \in \{\mathbf{0}, \mathbf{1}\}^*, \text{ tel que } |u| \leq p(|w|) \text{ et } < w, u > \in A.$$

Définition 8.18 (Classe $\mathrm{NP}_{\mathfrak{M}}$**)** *Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure. Un problème $L \subset M^*$ est dans $\mathrm{NP}_{\mathfrak{M}}$ s'il existe un polynôme $p : \mathbb{N} \to \mathbb{N}$ et un problème A dans $\mathrm{P}_{\mathfrak{M}}$ (appelé* vérificateur pour L*) tel que pour tout mot w,*

$$w \in L \text{ si et seulement si } \exists u \in M^*, \text{ tel que } |u| \leq p(|w|) \text{ et } < w, u > \in A.$$

Pour les structures finies, il n'y a pas besoin de distinguer ces deux classes, car on peut toujours coder un élément par son écriture en binaire, et donc les deux classes coïncident.

Par définition, on a

Proposition 8.2 *Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure. On a*

$$\mathrm{P}_{\mathfrak{M}} \subset \mathrm{NDP}_{\mathfrak{M}} \subset \mathrm{NP}_{\mathfrak{M}}.$$

8.3.3 Temps non-déterministe polynomial et équivalence

Selon la discussion du chapitre 4 , un algorithme déterministe peut toujours s'écrire à l'aide d'instructions d'affectations, de **if then else**, et d'instructions **par endpar**.

On peut introduire la notion d'algorithme non-déterministe, en ajoutant une instruction **guess** x : c'est-à-dire qu'aux constructions du chapitre 4 , construisant des règles de mise à jour, des règles de mise en parallèle **par endpar**, et de test **if then else**, on ajoute la règle suivante :

Définition 8.19 (Instruction non-déterministe) *Soit Σ une signature, et t un symbole de constante sur la signature Σ.*

guess t

est une règle R de vocabulaire Σ.

Pour exécuter une règle R non-déterministe dans l'état X, il se produit une interaction avec l'environnement qui renvoie un élément $m \in M$ de l'ensemble de base de X. On exécute alors la mise à jour $(t, \emptyset) \leftarrow m$, c'est-à-dire que l'on affecte au contenu de l'emplacement t la valeur de m. Le résultat de l'exécution de cette règle en un état X peut donc mener à plusieurs nouveaux états X, selon la valeur choisie par l'environnement pour m.

Autrement dit, à un algorithme n'est plus associé un système de transition comme dans le postulat 1 mais un système de transition non-déterministe. Formellement :

Définition 8.20 (Système de transitions non-déterministe) *Un* système de transitions non-déterministe *est donné par*

- *un ensemble $S(A)$, dont les éléments sont appelés des* états *(parfois* configurations*),*
- *un sous-ensemble $I(A) \subset S(A)$ dont les éléments sont dits* initiaux*,*
- *et une relation $\tau_A \subset S(A) \times S(A)$ que l'on appelle* évolution en un pas *de A.*

Une *exécution* du système de transition non-déterministe est une suite X_0, X_1, $\cdots$, $X_n, \cdots$ où X_0 est un état initial ($X_0 \in I(A)$), chaque X_i est un état ($X_i \in S(A)$), et $(X_i, X_{i+1}) \in \tau_A(X_i)$ pour tout i.

On dit alors qu'un problème L est *décidé en temps non-déterministe $t(n)$*, s'il existe un algorithme dans ce sens étendu, tel que pour tout mot w, si l'on note $n = |w|$, il existe une exécution du système de transition non-déterministe associé à l'algorithme entre l'état initial X_0 correspondant à w qui mène à un de ses état terminal accepteur en moins de $t(n)$ étapes si et seulement si $w \in L$.

On peut aussi introduire la notion *d'algorithme non-déterministe digital* : on remplace la définition 8.19 par :

Définition 8.21 *Soit Σ une signature, et t un symbole de constante sur la signature Σ.*

guess $t \in \{\mathbf{0}, \mathbf{1}\}$

est une règle R de vocabulaire Σ.

Pour exécuter une telle règle R dans l'état X, il se produit une interaction avec l'environnement qui renvoie un élément $m \in \{\mathbf{0}, \mathbf{1}\}$. On exécute alors la mise à jour $(t, \emptyset) \leftarrow m$, c'est-à-dire que l'on affecte au contenu de l'emplacement t la valeur de m, qui cette fois ne peut être que 0 ou 1.

On dit alors qu'un problème L est *décidé en temps non-déterministe digital* $t(n)$, s'il existe un algorithme dans ce nouveau sens, tel que pour tout mot w, si l'on note $n = |w|$, il existe une exécution du système de transition non-déterministe associé à l'algorithme entre l'état initial X_0 correspondant à w qui mène à un de ses état terminal accepteur en moins de $t(n)$ étapes si et seulement si $w \in L$.

Définition 8.22 *Soit $t : \mathbb{N} \to \mathbb{N}$ une fonction. On définit la classe* NTIME(t(n)) *comme la classe des problèmes ou des langages qui sont solvables par un algorithme qui fonctionne en temps non-déterministe $\mathcal{O}(t(n))$. Si l'on préfère,*

$$\text{NTIME(t(n))} = \{L | L \textit{ est un langage décidé par}$$

$$\textit{un algorithme non-déterministe en temps } \mathcal{O}(t(n))\}.$$

Ces modèles de calculs correspondent aux classes précédentes.
En effet, on a :

Théorème 8.6 *Soit $t : \mathbb{N} \to \mathbb{N}$ une fonction. Un problème $L \subset M^*$ est dans* NTIME(t(n)) *si et seulement s'il existe un problème A (appelé* vérificateur pour *L) tel que pour tout mot w,*

$$w \in L \textit{ si et seulement si } \exists u \in M^*, \quad < w, u > \in A,$$

et tel que vérifier si $< w, u > \in A$ soit solvable par un algorithme en temps $t(|w|)$.

Démonstration: Clairement si un problème L admet un problème A comme plus haut, on peut simuler l'algorithme pour A, sur $< w, u >$ en remplaçant la lecture de chaque lettre de u par une instruction **guess** t qui retourne la valeur t de cette lettre. On reconnaitra bien le problème L en temps non-déterministe $t(|w|)$.

Réciproquement, si un problème L est reconnu en temps non-déterministe $t(|w|)$ par un algorithme A_L, considérer le problème $A = \{< w, u > | w$ est accepté par A_L avec la suite des résultats de chacun de ses choix non-déterministes donnés par les lettres de u dans leur ordre$\}$. A est résoluble par un algorithme en temps $t(|w|)$ qui simule A_L en remplaçant chaque instruction **guess** par la lecture de la lettre de u correspondante. □

Nous laissons au lecteur le soin de définir $\text{NTIME}_{\mathfrak{M}}(\text{t(n)})$ et $\text{NTIME}^0_{\mathfrak{M}}(\text{t(n)})$, et de généraliser ce théorème au cas d'une structure arbitraire.

On en déduit :

Théorème 8.7 *Soit M un alphabet fini. Un problème $L \subset M^*$ est dans* NP *si et seulement s'il est décidé en temps non-déterministe polynomial.*

Théorème 8.8 *Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure.*
- *Un problème $L \subset M^*$ est dans $\mathrm{NP}_{\mathfrak{M}}$ si et seulement s'il est décidé en temps non-déterministe polynomial à paramètres sur $\mathfrak{M}$.*
- *Un problème $L \subset M^*$ est dans $\mathrm{NDP}_{\mathfrak{M}}$ si et seulement s'il est décidé en temps non-déterministe digital polynomial à paramètres sur $\mathfrak{M}$.*

Remarquons que par définition :

Corollaire 8.1 *Soit $t : \mathbb{N} \to \mathbb{N}$ une fonction. On a* TIME(t(n)) ⊂ NTIME(t(n)).

8.4 Problèmes complets

8.4.1 Notion de réduction

Comme en calculabilité, on peut introduire une notion de réduction. La première étape est de parler de fonctions calculables en temps polynomial.

Définition 8.23 (Fonction calculable en temps polynomial) *Soient M et N deux alphabets. Une fonction $f : M^* \to N^*$ est* calculable en temps polynomial *s'il existe un algorithme A, et un entier k, tel que pour tout mot w, A avec l'entrée w termine en un temps $\mathcal{O}(n^k)$, où $n = |w|$, avec le résultat $f(w)$.*

On peut se convaincre facilement du résultat suivant :

Proposition 8.3 (Stabilité par composition) *La composée de deux fonctions calculables en temps polynomial est calculable en temps polynomial.*

Cela nous permet d'introduire une notion de réduction entre problèmes : l'idée est que si A se réduit à B, alors le problème A est plus facile que le problème B, ou si l'on préfère, le problème B est plus difficile que le problème A.

Définition 8.24 (Réduction) *Soient A et B deux problèmes d'alphabet respectifs M_A et M_B, et de langages respectifs M_A et M_B. Une* réduction de A vers B *est une fonction $f : M_A^* \to M_B^*$ calculable en temps polynomial telle que $w \in L_A$ ssi $f(w) \in L_B$. On note $A \leq B$ lorsque A se réduit à B.*

Cela se comporte comme on s'y attend : un problème est aussi facile (et difficile) que lui-même, et la relation "être plus difficile que" est transitive.

Théorème 8.9 *$\leq$ est un préordre :*

1. *$L \leq L$*
2. *$L_1 \leq L_2$, $L_2 \leq L_3$ implique $L_1 \leq L_3$.*

Démonstration: Considérer la fonction identité comme fonction f pour le premier point.

Pour le second point, supposons $L_1 \leq L_2$ via la fonction f, et $L_2 \leq L_3$ via la fonction g. On a $x \in L_1$ ssi $g(f(x)) \in L_2$. Il suffit alors de voir que $g \circ f$, en temps que composée de deux fonctions calculables en temps polynomial, est calculable en temps polynomial. □

Il ne s'agit pas d'un ordre, puisque $L_1 \leq L_2$, $L_2 \leq L_1$ n'implique pas $L_1 = L_2$.

En fait, il est naturel d'introduire :

Définition 8.25 *Deux problèmes L_1 et L_2 sont équivalents, noté $L_1 \equiv L_2$, si $L_1 \leq L_2$ et si $L_2 \leq L_1$.*

On a alors $L_1 \leq L_2$, $L_2 \leq L_1$ implique $L_1 \equiv L_2$.

Intuitivement, si un problème est plus facile qu'un problème polynomial, alors il est polynomial. Formellement.

Proposition 8.4 (Réduction) *Si $A \leq B$, et si $B \in \mathrm{P}$ (respectivement $B \in \mathrm{P}_{\mathfrak{M}}$) alors $A \in \mathrm{P}$ (resp. $A \in \mathrm{P}_{\mathfrak{M}}$).*

Démonstration: A est décidé par l'algorithme qui, sur une entrée w, calcule $f(w)$, puis simule l'algorithme qui décide B sur l'entrée $f(w)$. Puisqu'on a $w \in A$ si et seulement si $f(w) \in B$, l'algorithme est correct, et fonctionne bien en temps polynomial. □

Proposition 8.5 (Réduction) *Si $A \leq B$, et si $A \notin \mathrm{P}$ (respectivement $A \notin \mathrm{P}_{\mathfrak{M}}$)) alors $B \notin \mathrm{P}$ (resp. $B \notin \mathrm{P}_{\mathfrak{M}}$)).*

Démonstration: Il s'agit de la contraposée de la proposition précédente. □

8.4.2 Notion de complétude

On peut aussi introduire une notion de complétude : un problème NP-complet est un problème maximal pour $\leq$ parmi les problèmes de la classe NP. Si l'on préfère :

Définition 8.26 (NP-complétude) *Un problème A est dit* NP*-complet, si*

1. *il est dans* NP
2. *tout autre problème B de NP est tel que $B \leq A$.*

Corollaire 8.2 *Tous les problèmes* NP*-complets sont équivalents.*

Démonstration: Soient A et B deux problèmes NP-complets. Appliquer la propriété 2. en A relativement à $B \in \mathrm{NP}$, et en B relativement à $A \in \mathrm{NP}$. □

Plus généralement : un problème $\mathcal{C}$-complet est un problème maximal pour $\leq$ parmi les problèmes de la classe $\mathcal{C}$:

Définition 8.27 ($\mathcal{C}$-complétude) *Soit $\mathcal{C}$ une classe de langages. Un problème A est dit $\mathcal{C}$-complet, si*

1. *il est dans la classe $\mathcal{C}$*
2. *tout autre problème B de la classe $\mathcal{C}$ est tel que $B \leq A$.*

Corollaire 8.3 *Soit $\mathcal{C}$ une classe de langages. Tous les problèmes $\mathcal{C}$-complets sont équivalents.*

8.4.3 Existence de problème NP-complet

En fait, il n'est pas difficile de se convaincre de l'existence de problèmes complets : il suffit de jouer avec les définitions (rappelons que $\mathbf{1}^t$ désigne le mot de longueur t avec t fois la lettre $\mathbf{1}$) :

Théorème 8.10 *Le problème*

$$K = \{< A, w, \mathbf{1}^n, \mathbf{1}^t > | \exists u \in M^*, |u| = n \text{ tel que l'algorithme } A \text{ accepte l'entrée } < w, u > \text{ en temps } t\}$$

est NP*-complet.*

Démonstration: Par définition, il s'agit d'un problème de NP.

Maintenant soit L un problème de NP. L est décidé en temps polynomial $p(n)$ non-déterministe par un algorithme A'. Considérer l'algorithme A qui sur une entrée $< w, u >$ simule A' en lisant dans les lettres de u la suite des résultats des interactions avec l'environnement : la $i$ème lettre de u correspond au résultat de la $i$ème interaction. Soit $p'(n)$ un majorant du temps de simulation de l'algorithme A' sur entrée $< w, u >$, avec $|w| = n$.

La fonction qui à w, de longueur n, associe $< A, w, \mathbf{1}^{p(n)}, \mathbf{1}^{p'(n)} >$ réalise une réduction de L vers K. □

La même preuve, montre les résultats suivants :

Théorème 8.11 *Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure suffisamment expressive.*

Le problème

$$K_{\mathfrak{M}} = \{< A, w, \mathbf{1}^n, \mathbf{1}^t > | \exists u \in M^*, |u| = n \text{ tel que l'algorithme } A \text{ sur } \mathfrak{M} \text{ accepte l'entrée } < w, u > \text{ en temps } t\}$$

est $\mathrm{NP}_{\mathfrak{M}}$*-complet.*

et

Théorème 8.12 *Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure suffisamment expressive.*

Le problème

$$K_{\mathfrak{M}} = \{< A, w, \mathbf{1}^n, \mathbf{1}^t > | \exists u \in \{\mathbf{0}, \mathbf{1}\}^*, |u| = n \text{ tel que l'algorithme } A \text{ sur } \mathfrak{M} \text{ accepte l'entrée } < w, u > \text{ en temps } t\}$$

est $\mathrm{NDP}_{\mathfrak{M}}$*-complet.*

8.4.4 Premier problème NP-complet naturel

Les problèmes précédents sont cependant très artificiels, car faisant référence à la notion d'algorithme, et donc quelque part directement à la définition.

Il s'avère que l'on peut construire des problèmes NP-complets naturels assez facilement, si l'on a compris qu'algorithmes et circuits sont équivalents au niveau de la complexité.

On dit qu'un circuit booléen C est *satisfiable*, s'il est possible d'affecter ses variables $\overline{x} = (x_1, \cdots, x_n)$ par des éléments de $\{0, 1\}$, tel que $C(\overline{x}) = 1$.

Théorème 8.13 *Étant donné un circuit booléen, le problème de savoir s'il est satisfiable est* NP*-complet.*

Démonstration: Le problème est dans NP, car un circuit C est satisfiable si et seulement s'il existe un mot $\overline{x}$ de longueur n, où n est le nombre d'entrées du circuit, tel que $< C, \overline{x} > \in$ CIRCUITVALUE.

Maintenant soit L un langage de NP. Par définition, il existe un problème A dans P, et un polynôme p, tel que pour tout mot w, $w \in L$ si et seulement si il existe $u \in M^*$, $|u| \leq p(|w|)$ avec $< w, u > \in A$. Selon le théorème 8.2, déterminer si $< w, u > \in A$ correspond à un circuit C de taille polynomiale en $| < w, u > |$, et donc en $|w|$. Considérons le circuit C' obtenu en fixant les entrées correspondant au mot w aux symboles de w.

La fonction qui au mot w associe C' réalise une réduction de L vers le problème de satisfiabilité des circuits. En effet, $w \in L$ si et seulement si C' est satisfiable par un certain u. □

Autrement dit, le problème de la satisfiabilité des circuits est NP-complet.

Cela se généralise aux structures arbitraires de la façon suivante.

Le problème de *la satisfiabilité des circuits à paramètres* est le suivant : on se donne un circuit $C(\overline{x}, \overline{y})$ sur une structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$, et un mot $\overline{c} = (c_1, \ldots, c_k) \in M^*$ de même longueur que $\overline{x}$, et l'on veut savoir s'il est possible d'affecter les variables $\overline{y} = (y_1, \cdots, y_n)$ par des éléments de M, tels que $C(\overline{c}, \overline{y}) = 1$.

Le problème de *la satisfiabilité des circuits à paramètres par des entrées booléennes* est le suivant : on se donne un circuit $C(\overline{x}, \overline{y})$ sur une structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$, et un mot $\overline{c} = (c_1, \ldots, c_k) \in M^*$ de même longueur que $\overline{x}$, et l'on veut savoir s'il est possible d'affecter les variables $\overline{y} = (y_1, \cdots, y_n)$ par des éléments de $\{\mathbf{0}, \mathbf{1}\}$, tels que $C(\overline{c}, \overline{y}) = 1$.

Théorème 8.14 *Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure. Le problème de la satisfiabilité des circuits à paramètres sur $\mathfrak{M}$ est* $\mathrm{NP}_{\mathfrak{M}}$*-complet.*

Théorème 8.15 *Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure. Le problème de la satisfiabilité des circuits à paramètres sur $\mathfrak{M}$ par des entrées booléennes est* $\mathrm{NDP}_{\mathfrak{M}}$*-complet.*

Démonstration: Le problème est dans NP, car un circuit C avec la suite de paramètres $\overline{c}$ est satisfiable si et seulement s'il existe un mot $\overline{x}$ de longueur n, où n est le nombre d'entrées du circuit, tel que $< C, \overline{c}\ \overline{x} > \in$ CIRCUITVALUE.

Maintenant soit L un langage de NP. Par définition, il existe un problème A dans $\mathrm{P}_{\mathfrak{M}}$, et un polynôme p, tel que pour tout mot w, $w \in L$ si et seulement si il existe $u \in M^*$, $|u| \leq p(|w|)$ avec $< w, u > \in A$. Selon le théorème 8.2, déterminer si $< w, u > \in A$ correspond à un circuit C (éventuellement à paramètres $\overline{c}'$) de taille polynomiale en $| < w, u > |$, et donc en $|w|$.

La fonction qui au mot w associe $C(w, \overline{c}', \overline{x})$, où $\overline{x}$ désigne les entrées, et $w, \overline{c}'$ les paramètres, réalise une réduction de L vers le problème de satisfiabilité des circuits. En effet, $w \in L$ ssi $C(w, \overline{c}', \overline{x})$ est satisfiable par un certain u. □

8.4.5 Problèmes plus basiques NP-complets

On peut réécrire ce résultat de la façon suivante.

Définition 8.28 (Formules rudimentaires) *Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure. On appelle* formule rudimentaire sur $\mathfrak{M}$ *une conjonction finie de formules de l'un des types suivants*

1. $x = a$, *où a est un élément de M.*
2. $y = f_i(\overline{x})$, *où f_i est l'une des fonctions de la structure.*
3. $r(\overline{x}) \vee y = \mathbf{0}$, *où r_j est l'une des relations de la structure.*
4. $\neg r(\overline{x}) \vee y = \mathbf{1}$, *où r_j est l'une des relations de la structure.*
5. $x = \mathbf{0} \vee x = \mathbf{1}$.
6. $x = \epsilon \vee y = \epsilon'$, *où* $\epsilon, \epsilon' \in \{\mathbf{0}, \mathbf{1}\}$.
7. $x = \epsilon \vee y = \epsilon' \vee z = \epsilon''$, *où* $\epsilon, \epsilon', \epsilon'' \in \{\mathbf{0}, \mathbf{1}\}$.

Une formule rudimentaire est dite *satisfiable* s'il est possible d'affecter ses variables de telle sorte que toutes ses formules soient satisfaites.

Théorème 8.16 *Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure. Le problème de savoir si une formule rudimentaire est satisfiable est* $\mathrm{NP}_{\mathfrak{M}}$*-complet.*

Démonstration: C'est clairement un problème de $\mathrm{NP}_{\mathfrak{M}}$.

Réciproquement, on montre qu'un circuit est satisfiable si et seulement si une certaine formule rudimentaire de taille polynomiale est satisfaite : l'astuce est d'introduire une variable par porte du circuit pour représenter la valeur qui en sort (qui peut être un booléen résultat d'un test), et d'écrire que cette variable s'exprime en fonction des entrées de la porte, à l'aide de l'opérateur réalisé par la porte, et de réaliser que cela peut s'écrire à chaque fois comme une conjonction de formules des types plus haut. On écrit alors la conjonction de toutes les contraintes, et de la contrainte que la sortie doit être **1**. La formule rudimentaire obtenue est de taille polynomiale, et est satisfiable si et seulement si le circuit l'est. □

Remarque 8.5 *Il n'y a aucune raison que la satisfaction des formules rudimentaires par des entrées booléennes soit* $\mathrm{NDP}_{\mathfrak{M}}$*-complet puisqu'on introduit des variables comme $y = f_i(\overline{x})$ qui ne sont pas des booléens, mais des éléments de M l'ensemble de base de la structure.*

Le problème $\leq k$-SAT est le problème de la satisfiabilité d'une formule du calcul propositionnel en forme normale conjonctive avec au plus k-littéraux par clause : une instance de $\leq k$-SAT est donc une conjonction $c_1 \wedge \cdots c_n$ de n-clauses, chaque clause c_i étant une disjonction $l_{i,1} \vee l_{i,2} \cdots \vee l_{i,k_i}$, $k_i \leq k$, où chacun des $l_{i,j}$ est soit une variable x, soit la négation $\neg x$ d'une variable. La formule ϕ suivante est une instance de $\leq$ 3-SAT sur les variables x, y, z :

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

Plus généralement, le problème SAT est le problème de la satisfiabilité d'une formule du calcul propositionnel en forme normale conjonctive (sans contrainte sur le nombre de clauses).

Théorème 8.17 *Le problème de la satisfiabilité d'une formule de* $\leq$ 3-SAT *est* NP*-complet.*

Démonstration: Puisque $\leq$ 3-SAT est dans NP, il suffit de montrer qu'une formule rudimentaire peut toujours s'écrire comme une formule $\leq$ 3-SAT de taille polynomiale. Un algorithme classique correspond à la structure $\mathfrak{B} = (\{0,1\}, \mathbf{0}, \mathbf{1}, =)$. Un composant de formule rudimentaire de la forme $x = y \vee u = \mathbf{0}$ peut être remplacé par $(x = \mathbf{0} \vee y = \mathbf{1} \vee u = \mathbf{0}) \wedge (x = \mathbf{1} \vee y = \mathbf{0} \vee u = \mathbf{0})$. Un composant de formule rudimentaire de la forme $\neg x = y \vee u = \mathbf{1}$ peut être remplacé par $(x = \mathbf{0} \vee y = \mathbf{0} \vee u = \mathbf{1}) \wedge (x = \mathbf{1} \vee y = \mathbf{1} \vee u = \mathbf{1})$. □

Le problème k-SAT est le problème de la satisfiabilité d'une formule du calcul propositionnel en forme normale conjonctive avec exactement k-littéraux par clause.

Théorème 8.18 *Le problème de la satisfiabilité d'une formule de* 3-SAT *est* NP*-complet.*

Démonstration: Chaque clause de moins de 3 littéraux peut se remplacer par une conjonction de clauses à 3-littéraux. Par exemple : $\alpha \vee \beta$ peut se remplacer par $(\alpha \vee \beta \vee \gamma) \wedge (\alpha \vee \beta \vee \neg\gamma)$. La première est satisfiable si et seulement si la seconde l'est. □

Corollaire 8.4 *Le problème de la satisfiabilité d'une formule de* SAT *est* NP*-complet.*

Démonstration: Clairement 3-SAT $\leq$ SAT via la fonction identité. Il suffit alors de remarquer que SAT $\in$ NP. □

8.4.6 P = NP et élimination des quanteurs

On peut facilement déduire de la discussion précédente :

Le problème de la satisfiabilité des formules (libres) à paramètres consiste étant donnée une formule libre $f(\overline{x}, \overline{y})$ sans paramètres, et un uple $\overline{a}$ d'éléments de M, déterminer s'il existe un uple $\overline{b}$ tel que $f(\overline{a}, \overline{b})$ soit satisfait.

Théorème 8.19 *Dans toute structure $\mathfrak{M}$, le problème de la satisfiabilité des formules (libres) à paramètres est* $\mathrm{NP}_{\mathfrak{M}}$*-complet.*

Démonstration: Les formules rudimentaires sont des formules (libres) particulières, et donc le problème de la satisfiabilité des formules rudimentaires se réduit à ce problème. Maintenant, le problème est clairement dans $\mathrm{NP}_{\mathfrak{M}}$. □

On obtient le résultat suivant qui relie formules et circuits, via la question de l'élimination effective et efficace des quanteurs.

Théorème 8.20 *Une structure $\mathfrak{M}$ vérifie $\mathrm{P}_{\mathfrak{M}} = \mathrm{NP}_{\mathfrak{M}}$ si et seulement s'il existe un algorithme dans $\mathrm{P}_{\mathfrak{M}}$ avec paramètres $\overline{c}$ qui transforme toute formule existentielle (sans paramètres) $\exists \overline{y}\ f(\overline{x}, \overline{y})$ en un circuit équivalent $C(\overline{c}, \overline{x})$.*

Démonstration: Si cette dernière propriété est vérifiée, pour tester si $f(\overline{a}, \overline{y})$ est satisfiable, on commence par transformer la formule en son circuit équivalent $C(\overline{c}, \overline{a})$, et on calcule si $C(\overline{c}, \overline{a})$ vaut 1 ou 0.

Réciproquement, soit A un algorithme polynomial, utilisant les paramètres $\overline{c}$ qui quand on lui donne la formule $f(\overline{x}, \overline{y})$ détermine si $f(\overline{a}, \overline{y})$ est satisfiable. Lorsque la longueur de la formule et du uplet $\overline{a}$ sont fixés, cela se traduit par un circuit $C(\overline{c}, f, \overline{x})$ tel que $C(\overline{c}, f, \overline{a})$ vaut 1 si $f(\overline{a}, \overline{y})$ est satisfiable, 0 sinon. Le circuit $C(\overline{c}, f, \overline{x})$ est bien équivalent à la formule $\exists \overline{y}\ f(\overline{x}, \overline{y})$. □

8.4.7 Autres problèmes NP-complets

Une fois établi la NP-complétude de 3-SAT, nous renvoyons au polycopié du cours *INF550* pour la preuve des résultats suivants, où cette première était admise.

Un *stable d'un graphe* $G = (V, E)$ est un sous-ensemble $V' \subset V$ ne contenant aucune paire de sommets voisins dans G.

Théorème 8.21 *Étant donné un graphe G, et un entier k, le problème de savoir s'il admet un stable de taille $\geq k$ est* NP*-complet.*

Une *clique d'un graphe* $G = (V, E)$ est un sous-ensemble $V' \subset V$ tel que toute paire de sommets de V' soit reliée par une arête de G.

Théorème 8.22 *Étant donné un graphe G, et un entier k, le problème de savoir s'il admet une clique de taille $\geq k$ est* NP*-complet.*

Un *sous-ensemble couvrant* d'un graphe $G = (V, E)$ est un sous-ensemble $V' \subset V$ tel que toute arête de E ait au moins une de ses extrémités dans V'.

Théorème 8.23 *Étant donné un graphe G, et un entier k, le problème de savoir s'il admet une sous-ensemble couvrant de taille $\leq k$ est* NP*-complet.*

Un *chemin hamiltonien d'un graphe* G est un chemin qui passe une fois et une seule par chaque sommet de G. Le problème du chemin hamiltonien est de savoir si un graphe possède un chemin hamiltonien.

Théorème 8.24 *Le problème du chemin hamiltonien est* NP*-complet.*

Le problème de la somme partielle consiste, étant donné un ensemble fini d'entiers S et un entier s, à déterminer s'il existe un sous-ensemble $X \subset S$ tel que $\sum_{i \in X} i = s$.

Théorème 8.25 *Le problème de la somme partielle est* NP*-complet.*

8.4.8 Sur les réels

Théorème 8.26 *Dans* $(\mathbb{R}, +, -, *, =, <)$*, le problème de l'existence d'une racine pour un polynôme en* n *variables à coefficients réels et de degré total* 4 *est* $\mathrm{NP}_{(\mathbb{R},+,-,*,=,<)}$*-complet.*

8.5 Quelques résultats

8.5.1 Décision vs Construction

Nous avons défini la notion de temps non-déterministe polynomial en utilisant des problèmes de *décision*, c'est-à-dire dont la réponse est soit positive soit négative (par exemple : "la formule est elle satisfiable" ?) en opposition aux problèmes de *recherche* (par exemple : produire une affectation des variables qui la satisfait).

Clairement, produire une solution est plus difficile de savoir s'il en existe une, et donc si P $\neq$ NP, aucun de ces deux problèmes n'admet une solution pour tout problème NP-complet.

Cependant, si P = NP, il s'avère que si l'on sait résoudre le problème de décision, alors on sait aussi produire une solution :

Théorème 8.27 *Supposons que* P = NP*. Alors pour tout langage* $L \in$ NP*, et pour tout vérificateur* A *pour* L *(au sens de la définition 8.16), il existe un algorithme* B *qui sur toute entrée* $w \in L$ *produit en temps polynomial un certificat pour* w *(vis à vis du langage* L *et de* A*).*

Démonstration: Commençons par le prouver pour L correspondant au problème de la satisfaction de circuits. Supposons P = NP : on peut donc tester si un circuit C à n-entrées est satisfiable ou non. S'il est satisfiable, on peut fixer sa première entrée à $\mathbf{0}$, et tester si le circuit obtenu $C_{\mathbf{0}}$ est satisfiable. Si l'est, on écrit $\mathbf{0}$ et on recommence récursivement avec ce circuit $C_{\mathbf{0}}$ à $n-1$-entrées. Sinon, nécessairement tout certificat doit avoir sa première variable à $\mathbf{1}$, on écrit $\mathbf{1}$, et on recommence récursivement avec le circuit $C_{\mathbf{1}}$ dont la première entrée est fixée à $\mathbf{1}$, qui possède $n-1$-entrées. Puisqu'il est facile de vérifier si un circuit sans entrée est satisfiable, par cette méthode, on aura écrit un certificat.

Maintenant si L est un langage quelconque de NP, on peut utiliser le fait que la réduction produite par le théorème 8.13 est en fait une réduction de *Levin* : non seulement on a $w \in L$ si et seulement $f(w)$ est un circuit satisfiable,

mais on peut aussi retrouver un certificat pour w à partir d'un certificat de la satisfiabilité du circuit $f(w)$. On peut donc utiliser l'algorithme précédent pour retrouver un certificat pour L. □

En fait, on vient d'utiliser le fait que le problème de la satisfiabilité d'un circuit est *auto-réductible* à des instances de tailles inférieures.

8.5.2 EXPTIME **and** NEXPTIME

Considérons

$$\text{EXPTIME} = \bigcup_{c \geq 1} \text{TIME}(2^{n^c})$$

et

$$\text{NEXPTIME} = \bigcup_{c \geq 1} \text{NTIME}(2^{n^c})$$

Théorème 8.28 *Si* EXPTIME $\neq$ NEXPTIME *alors* P $\neq$ NP.

8.5.3 Théorèmes de hiérarchie

On dit qu'une fonction $f(n) \geq n\log(n)$ est *constructible en temps*, si la fonction qui envoie $\mathbf{1}^n$ sur la représentation binaire de $\mathbf{1}^{f(n)}$ est calculable en temps $\mathcal{O}(f(n))$.

La plupart des fonctions usuelles sont constructibles en temps : par exemple, $n\sqrt{n}$ est constructible en temps : sur l'entrée $\mathbf{1}^n$, on commence par compter le nombre de $\mathbf{1}$ en binaire : on peut utiliser pour cela un compteur, qui reste de taille $\log(n)$, que l'on incrémente : cela se fait donc en temps $\mathcal{O}(n\log(n))$ puisqu'on utilise au plus $\mathcal{O}(\log(n))$ étapes pour chaque lettre du mot en entrée. On peut alors calculer $\lfloor n\sqrt{n} \rfloor$ en binaire à partir de la représentation de n. N'importe quelle méthode pour faire cela fonctionne en temps $\mathcal{O}(n\log(n))$, puisque la taille des nombres impliqués est $\mathcal{O}(\log(n))$.

Théorème 8.29 (Théorème de hiérarchie) *Pour toute fonction $f : \mathbb{N} \to \mathbb{N}$ constructible en temps, il existe un langage L qui est décidable en temps $\mathcal{O}(f(n))$ mais pas en temps $o(f(n))$.*

Démonstration: Il s'agit d'une généralisation de l'idée de la preuve du théorème 9.13 du chapitre suivant : nous invitons notre lecteur à commencer par cette dernière preuve.

Nous prouverons une version plus faible que l'énoncé plus haut. Soit $f : \mathbb{N} \to \mathbb{N}$ une fonction constructible en temps.

On considère le langage (très artificiel) L qui est décidé par l'algorithme B suivant :

- sur une entrée w de taille n, B calcule $f(n)$ et mémorise $< f(n) >$ le codage en binaire de $f(n)$) dans un compteur binaire c.
- Si w n'est pas de la forme $< A > \mathbf{10}^*$, pour un certain algorithme A, alors l'algorithme B rejette.

- Sinon, B simule A sur le mot w pendant $f(n)$ étapes pour déterminer si A accepte en un temps inférieur à $f(n)$:
  - si A accepte en ce temps, alors B rejette ;
  - sinon B accepte :

Autrement dit B simule étape par étape A sur w, et décrémente le compteur c à chaque étape. Si ce compteur c atteint 0 ou A refuse, alors B accepte. Sinon, B rejette.

Par l'existence d'un algorithme universel, il existe des entiers k et d tels que L soit décidé en temps $df(n)^k$.

Supposons que L soit décidé par un algorithme A en temps $g(n)$ avec $g(n)^k = o(f(n))$. Il doit exister un entier n_0 tel que pour $n \geq n_0$, on ait $dg(n)^k < f(n)$.

Par conséquent, la simulation par B de A sera bien complète sur une entrée de longueur n_0 ou plus.

Considérons ce qui se passe lorsque B est lancé sur l'entrée $< A > \mathbf{10}^{n_0}$. Puisque cette entrée est de taille plus grande que n_0, B répond l'inverse de l'algorithme A sur la même entrée. Donc B et A ne décident pas le même langage, et donc l'algorithme A ne décide pas L, ce qui mène à une contradiction.

Par conséquent L n'est pas décidable en temps $g(n)$ pour toute fonction $g(n)$ avec $g(n)^k = o(f(n))$.

Le théorème est une généralisation de cette idée. Le facteur log vient de la construction d'un algorithme universel nettement plus efficace que ceux considérés dans ce document, introduisant seulement un ralentissement logarithmique en temps. □

Autrement dit :

Théorème 8.30 (Théorème de hiérarchie) *Soient $f, f' : \mathbb{N} \to \mathbb{N}$ des fonctions constructibles en temps telles que $f(n)\log(f(n)) = o(f'(n))$. Alors l'inclusion* $\mathrm{TIME(f)} \subset \mathrm{TIME(f')}$ *est stricte.*

On obtient par exemple :

Corollaire 8.5 $\mathrm{TIME}(\mathrm{n}^2) \subsetneq \mathrm{TIME}(\mathrm{n}^{\log \mathrm{n}}) \subsetneq \mathrm{TIME}(2^{\mathrm{n}})$.

On définit :

Définition 8.29 *Soit*

$$\mathrm{EXPTIME} = \bigcup_{c \in \mathbb{N}} \mathrm{TIME}(2^{\mathrm{n}^c})$$

On obtient :

Corollaire 8.6 $\mathrm{P} \subsetneq \mathrm{EXPTIME}$.

Démonstration: Tout polynome devient ultimement négligeable devant 2^n, et donc P est un sous-ensemble de $\mathrm{TIME}(2^{\mathrm{n}})$. Maintenant $\mathrm{TIME}(2^{\mathrm{n}})$, qui contient tout P est un sous-ensemble strict de, par exemple, $\mathrm{TIME}(2^{\mathrm{n}^3})$, qui est inclus dans EXPTIME. □

8.5.4 Problèmes booléens sur $\mathbb{R}$ ou $\mathbb{C}$

Lorsqu'on considère une structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ arbitraire, on considère des langages sur l'alphabet M. Si $L \subset M^*$ est un langage, on notera BOOLEAN(L) pour $L \cap \{\mathbf{0}, \mathbf{1}\}^*$. Si $\mathcal{C}$ est une classe de langage, on notera BOOLEAN($\mathcal{C}$) pour la classe des langages BOOLEAN(L) avec $L \in \mathcal{C}$.

Théorème 8.31

$$\text{BOOLEAN}(\text{P}_{(\mathbb{R},+,-,=)}) = \text{P}.$$

$$\text{BOOLEAN}(\mathbb{P}_{(\mathbb{R},+,-,=)}) = \mathbb{P}.$$

Théorème 8.32 $\text{BOOLEAN}(\text{P}_{(\mathbb{R},+,-,=,<)}) = \mathbb{P}$.

Théorème 8.33

$$\text{BOOLEAN}(\text{P}_{(\mathbb{C},+,-,*,=)}) = \text{BOOLEAN}(\text{P}_{(\mathbb{R},+,-,*,=)}) = \mathbb{P}$$

Théorème 8.34 *Tout problème $L \subset \{\mathbf{0}, \mathbf{1}\}^*$ est solvable par un algorithme en temps exponentiel sur la structure $(\mathbb{R}, +, -, *, =, <)$. Il existe des problèmes $L \subset \{\mathbf{0}, \mathbf{1}\}^*$ qui ne sont pas dans $\mathbb{P}_{(\mathbb{R},+,-,*,=,<)}$.*

8.6 Que signifie la question $P = NP$?

On peut voir NP comme classe des langages tel qu'en tester l'appartenance revient à déterminer s'il existe un certificat court (polynomial). On peut relier cela à l'existence d'une preuve en mathématiques. En effet, dans son principe même, la déduction mathématique consiste à produire des théorèmes à partir d'axiomes. On s'attend à ce que la validité d'une preuve soit facile à vérifier : il suffit de vérifier que chaque ligne de la preuve soit bien la conséquence des lignes précédentes, dans le système de preuve. En fait, dans la plupart des systèmes de preuves axiomatiques (par exemple en arithmétique de Peano, en théorie des ensembles de *Zermelo-Fraenkel*), cette vérification se fait en un temps qui est polynomial en la longueur de la preuve.

Autrement dit, le problème de décision suivant est NP pour chacun des systèmes de preuve axiomatiques usuels $\mathcal{A}$.

$$\text{THEOREMS} = \{< \phi, \mathbf{1}^n > | \phi \text{ possède une preuve de longueur } \leq n$$

$$\text{dans le système } \mathcal{A}\}.$$

Nous laisserons à notre lecteur l'exercice suivant :

Exercice 8.1 *La théorie des ensembles de* Zermelo-Fraenkel *est un des systèmes axiomatiques permettant d'axiomatiser les mathématiques avec une description finie. (Même sans connaître tous les détails de la théorie des ensembles de* Zermelo-Fraenkel*) argumenter à un haut niveau que le problème* THEOREMS *est* NP*-complet pour la théorie des ensembles de* Zermelo-Fraenkel*.*

Indice : la satisfiabilité d'un circuit booléen est un énoncé.

Autrement dit, en vertu du théorème 8.27, la question P = NP est celle (posée par Kurt Gödel) de savoir s'il existe un algorithme qui soit capable de produire la preuve mathématique de tout énoncé ϕ en un temps polynomial en la longueur de la preuve.

Cela semble t'il raisonnable ?

Rappelons que coNP est la classe des langages dont le complémentaire est dans NP. La question NP = coNP, est reliée à l'existence de preuve courte (de certicats) pour des énoncés qui ne semblent pas en avoir : par exemple, il est facile de prouver qu'un circuit est satisfiable (on produit une valuation de ses entrées, que l'on peut coder dans une preuve qui dit qu'en propageant les entrées vers les sorties, le circuit répond 1). Par contre, dans le cas général, il n'est pas facile d'écrire une preuve courte qu'un circuit donné est non satisfiable. Si NP = coNP, il doit toujours en exister une : la question est donc reliée à l'existence d'un autre moyen de prouver la non-satisfiabilité d'un circuit, que les méthodes usuelles.

On peut reformuler l'équivalent pour chacun des problèmes NP-complets évoqués.

8.7 Notes bibliographiques

Ce chapitre contient des résultats standards en complexité. Nous nous sommes essentiellement inspirés de leur présentation dans les ouvrages [Poizat, 1995], [Papadimitriou, 1994], [Arora and Barak, 2009]. Les considérations sur les structures arbitraires et sur les réels sont reprises de [Poizat, 1995], et de [Koiran, 1994]. La dernière partie discussion est standard, mais reprise ici essentiellement de sa formulation dans [Arora and Barak, 2009].

Chapitre 9

Complexité en espace

9.1 Classes de complexité en espace

9.1.1 Mesure de l'espace d'un calcul

Selon la discussion du chapitre 4, à un algorithme A correspond un système de transitions, c'est-à-dire un ensemble d'états $S(A)$ et une fonction d'évolution $\tau_A : S(A) \to S(A)$ qui décrit les évolutions entre ces états. Soit w une entrée d'un algorithme A, sur lequel A termine : dans le chapitre précédent, nous avons dit que le *temps de calcul* de l'algorithme A sur l'entrée w correspondait au nombre d'étapes du système de transitions associé.

Nous voulons maintenant parler de la mémoire utilisée. En fait, en théorie de la complexité, on parle plutôt *d'espace*, ou d'espace mémoire, que de mémoire. Pour cela, on va préciser comment on mesure l'espace d'une structure : rappelons que les états d'un algorithme correspondent à des structures sur une même signature.

Définition 9.1 *Considérons une structure du premier ordre sur la signature $(f_1, \cdots, f_u, r_1, \cdots, r_v)$. On rappelle que l'on dit qu'un emplacement $(f, \overline{m})$ est* utile*, si f_i est un symbole de fonction dynamique, et son contenu $[\![(f, \overline{m})]\!]$ n'est pas* ***undef****. La* taille (mémoire) *de la structure $\mathfrak{M}$, notée* size(X) *est le nombre d'emplacements utiles de X.*

On appelle *espace (mémoire) de calcul sur l'entrée w* le maximum de la taille (mémoire) de chacun des états de l'exécution de l'algorithme associée à l'entrée w :

Définition 9.2 *Soit A un algorithme. A chaque entrée w est associée une exécution $X_0, X_1, \cdots, X_t$ du système de transitions associé à A où $X_0 = X[w]$ est un état initial qui code w, et chaque X_i est un état, et $X_{i+1} = \tau_A(X_i)$ pour tout i. Supposons que w soit acceptée, c'est-à-dire qu'il existe un entier t avec X_t terminal. L'*espace (mémoire) de calcul sur l'entrée w*, noté* SPACE(A, w)*, est*

défini comme

$$\mathrm{SPACE(A, w)} = \max_{0 \leq i \leq t} \mathrm{size(X_i)}$$

Comme pour le temps, on raisonne à taille de donnée fixée.

Définition 9.3 (Espace de calcul) *Soit A un algorithme qui termine sur toute entrée. L'* espace (mémoire) *de l'algorithme A est la fonction $f : \mathbb{N} \to \mathbb{N}$ telle que pour tout entier n, $f(n)$ est le maximum de* SPACE(A, w) *pour les entrées (mots) w de longueur n. Autrement dit,*

$$f(n) = \max_{|w|=n} \mathrm{SPACE(A, w)}.$$

9.1.2 Notation SPACE(f(n))

Comme pour le temps, on introduit la notation suivante.

Définition 9.4 *Soit $t : \mathbb{N} \to \mathbb{N}$ une fonction. On définit la classe* SPACE(t(n)) *comme la classe des problèmes ou des langages qui sont solvables par un algorithme qui fonctionne en espace $\mathcal{O}(t(n))$. Si l'on préfère,*

$$\mathrm{SPACE(t(n))} = \{L | L \text{ est un langage décidé par un algorithme en espace } \mathcal{O}(t(n))\}.$$

Définition 9.5 *Soit $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ une structure. On définit la classe* $\mathrm{SPACE}_{\mathfrak{M}}$(f(n)) *comme la classe des problèmes ou des langages qui sont solvables par un algorithme qui fonctionne en espace $\mathcal{O}(f(n))$ sur $\mathfrak{M}$. Si l'on préfère,*

$$\mathrm{SPACE}_{\mathfrak{M}}\mathrm{(f(n))} = \{L | L \text{ est un langage décidé par un algorithme sur } \mathfrak{M} \text{ en espace } \mathcal{O}(f(n))\}.$$

En fait, cette façon de mesurer l'espace (mémoire) est problématique si l'on souhaite parler de fonctions $f(n)$ qui croient moins vite que n, comme $\log n$, car on compte (possiblement) dans la taille l'entrée et la sortie avec les définitions précédentes, et donc au moins n pour l'entrée. Pour éviter ce problème, on convient que l'entrée est accessible en *lecture uniquement*, et la sortie en *écriture uniquement* : l'entrée est codée par des symboles de fonctions statiques ; la sortie est codée par des symboles de fonctions dynamiques particuliers (comme par exemple **out**), et on suppose qu'un emplacement correspondant à ces symboles une fois écrit n'est jamais réécrit. On convient en outre dans la définition size(.) plus haut de ne pas compter les emplacements dynamiques qui correspondent à ces symboles de sortie (ceux d'entrées étant statiques, ils ne sont pas comptés de toute façon avec les conventions plus haut). C'est la convention qui sera utilisée dans la suite. D'autre part, on évitera d'utiliser des fonctions qui croient moins vite que $\log n$, pour éviter certains soucis[1]. Pour éviter des notations trop compliquées, nous laissons au lecteur le soin de corriger les définitions plus haut pour que ces conventions soient respectées.

[1] Comme par exemple que l'espace ne soit pas suffisant pour parcourir l'entrée et mesurer sa longueur en binaire.

9.1.3 Classe PSPACE

Comme dans le chapitre 8, on pourrait observer que les machines de Turing, les automates à $k \geq 2$-piles, où les algorithmes se simulent deux à deux en espace polynomial : la simulation de l'un par l'autre nécessite un espace qui reste polynomial en l'espace utilisé par le premier.

Cela invite aussi à considérer

Définition 9.6 PSPACE *est la classe des langages et des problèmes décidés par un algorithme en espace polynomial. En d'autres termes,*

$$\mathrm{PSPACE} = \bigcup_{k \in \mathbb{N}} \mathrm{SPACE}(\mathrm{n}^{\mathrm{k}}).$$

Plus généralement, on introduira

Définition 9.7 LOGSPACE *est la classe des langages et des problèmes décidés par un algorithme en espace logarithmique. En d'autres termes,*

$$\mathrm{LOGSPACE} = \mathrm{SPACE}(\log(\mathrm{n})).$$

9.1.4 Espace et structures non-finies

En fait, mesurer l'espace sur une structure dont l'ensemble de base est $\mathbb{N}$ ou $\mathbb{R}$ est souvent une mauvaise idée.

Sur $\mathbb{N}$, le problème est que tout algorithme peut se coder par une machine à 2 compteurs (voir le chapitre 5), et donc en espace $\mathcal{O}(1)$ (au prix d'une perte de temps qui peut être exponentielle).

Sur $\mathbb{R}$, on a le même type de phénomène :

Théorème 9.1 *Sur la structure* $(\mathbb{R}, 0, 1, +, -, *, =, <)$ *tout algorithme peut être réalisé par un algorithme qui fonctionne en espace constant (au prix d'une perte de temps qui peut être exponentielle).*

Nous n'en donnerons pas la preuve : on remarquera qu'il faut pour le prouver être capable d'effectuer toutes les opérations sur les réels en entrées en espace constant, ce qui nécessite plus que simplement dire qu'on utilise des machines à deux compteurs.

Pour ces raisons, on se limitera dans ce document aux structures finies dans tout ce qui concerne les discussions relatives à l'espace.

9.1.5 Espace non-déterministe

On peut aussi introduire une notion d'espace non-déterministe.

Définition 9.8 (Classe NPSPACE**)** *Soit M un alphabet fini. Un problème $L \subset M^*$ est dans* NPSPACE *s'il existe un polynôme $p : \mathbb{N} \to \mathbb{N}$ et un problème A (appelé* vérificateur pour L*) tels que pour tout mot w,*

$$w \in L \text{ si et seulement si } \exists u \in M^* \text{ tel que } < w, u > \in A,$$

et tels que déterminer si $< w, u > \in A$ admette un algorithme en espace $p(|w|)$.

Remarque 9.1 *C'est une définition sur le principe de la définition 8.15 pour le temps non-déterministe. Contrairement à ce qui se passe pour le temps non-déterministe, où l'on pouvait reformuler la définition 8.15 en la définition 8.16 définissant* NP *à partir de* P*, ici on ne peut pas reformuler cette définition facilement en définissant* NPSPACE *à partir de* PSPACE *: le problème est qu'un temps polynomial ne se relie pas directement à la longueur du certificat.*

On peut aussi se convaincre, en utilisant des arguments similaires au chapitre précédent, que cela correspond effectivement à l'espace non-déterministe polynomial pour les machines non-déterministes, en utilisant la notion de machine non-déterministe (avec des instructions **guess**) comme dans le chapitre précédent.

Théorème 9.2 *Soit M un alphabet fini. Un problème $L \subset M^*$ est dans* NPSPACE *si et seulement s'il est décidé en espace non-déterministe polynomial.*

9.2 Relations entre espace et temps

Pour éviter de compliquer inutilement certaines preuves, nous nous limiterons dans cette section à des fonctions $f(n)$ *de complexité propre* : on suppose que la fonction $f(n)$ est non-décroissante, c'est-à-dire que $f(n+1) \geq f(n)$, et qu'il existe un algorithme qui prend en entrée w et qui produit en sortie $\mathbf{1}^{f(n)}$ en temps $\mathcal{O}(n + f(n))$ et en espace $\mathcal{O}(f(n))$, où $n = |w|$.

Remarque 9.2 *Cela n'est pas vraiment restrictif, car toutes les fonctions usuelles non-décroissantes, comme* $\log(n)$*,* n*,* n^2*,* $\cdots$*,* $n \log n$*,* $n!$ *vérifient ces propriétés. En outre, ces fonctions sont stables par somme, produit, et exponentielle.*

Remarque 9.3 *En fait, nous avons besoin de cette hypothèse, car la fonction $f(n)$ pourrait ne pas être calculable, et donc il pourrait par exemple ne pas être possible d'écrire un mot de longueur $f(n)$ dans un des algorithmes décrits.*

Remarque 9.4 *Dans la plupart des assertions qui suivent, on peut se passer de cette hypothèse, au prix de quelques complications dans les preuves.*

9.2.1 Relations triviales

Un problème déterministe étant un problème non-déterministe particulier, on a :

Théorème 9.3 SPACE(f(n)) $\subset$ NSPACE(f(n)).

D'autre part :

Théorème 9.4 TIME(f(n)) $\subset$ SPACE(f(n)).

Démonstration: Un algorithme écrit au plus un nombre fini d'emplacements (mémoire) à chaque étape. L'espace mémoire utilisé reste donc linéaire en le temps utilisé. Rappelons que l'on ne compte pas l'entrée dans l'espace mémoire. □

9.2.2 Temps non-déterministe vs déterministe

De façon plus intéressante :

Théorème 9.5 *Pour tout langage de* NTIME(f(n))*, il existe un entier c tel que ce langage soit dans* TIME($c^{f(n)}$)*. Si l'on préfère :*

$$\text{NTIME(f(n))} \subset \bigcup_{c \in \mathbb{N}} \text{TIME}(c^{f(n)})$$

Démonstration: Soit L un problème de NTIME(f(n)). En utilisant le théorème 8.6, on sait qu'il existe un problème A tel que pour déterminer si un mot w de longueur n est dans L, il suffit de savoir s'il existe $u \in M^*$ avec $< w, u > \in A$, ce dernier test pouvant se faire en temps $f(n)$, où $n = |w|$. Puisqu'en temps $f(n)$ on ne peut pas lire plus que $f(n)$ lettres de u, on peut se limiter aux mots u de longueur $f(n)$. Tester si $< w, u > \in A$ pour tous les mots $u \in M^*$ de longueur $f(n)$ se fait facilement en temps $\mathcal{O}(c^{f(n)}) + \mathcal{O}(f(n)) = \mathcal{O}(c^{f(n)})$, où $c > 1$ est le cardinal de M : tester tous les mots u peut par exemple se faire en comptant en base c. □

Remarque 9.5 *Pour écrire le premier u à tester de longueur $f(n)$, nous utilisons le fait que cela doit être possible : c'est le cas, si l'on suppose $f(n)$ de complexité propre. On voit donc l'intérêt ici de cette hypothèse implicite. Nous éviterons de discuter ce type de problèmes dans la suite, qui ne se posent pas de toute façon pour les fonctions $f(n)$ usuelles.*

9.2.3 Temps non-déterministe vs espace

Théorème 9.6 NTIME(f(n)) ⊂ SPACE(f(n)).

Démonstration: On utilise exactement le même principe que dans la preuve précédente, si ce n'est que l'on parle d'espace. Soit L un problème de NTIME(f(n)). En utilisant le théorème 8.6, on sait qu'il existe un problème A tel que pour déterminer si un mot w de longueur n est dans L, il suffit de savoir s'il existe $u \in M^*$ de longueur $f(n)$ avec $< w, u > \in A$: on utilise un espace $\mathcal{O}(f(n))$ pour générer un à un les mots $u \in M^*$ de longueur $f(n)$ (par exemple en comptant en base c) puis on teste pour chacun si $< w, u > \in A$, ce dernier test se faisant en temps $f(n)$, donc espace $f(n)$. Le même espace pouvant être utilisé pour chacun des mots u, au total cela se fait au total un espace $\mathcal{O}(f(n))$ pour générer les u + $\mathcal{O}(f(n))$ pour les tests, soit $\mathcal{O}(f(n))$. □

9.2.4 Espace non-déterministe vs temps

Le problème de décision REACH, déjà mentionné dans le chapitre précédent, jouera un rôle important : on se donne un graphe orienté $G = (V, E)$, deux sommets u et v, et on cherche à décider dans ce problème s'il existe un chemin entre u et v. Nous avons vu dans le chapitre précédent que REACH est dans P.

Selon la discussion du chapitre 4 (le postulat 1) à tout algorithme A sur la structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ est associé un système de transition, dont les sommets sont appelés des états, ou encore des configurations, et les arcs correspondent à la fonction d'évolution en un pas de A. Dans le cas d'un algorithme non-déterministe, la fonction d'évolution n'est plus une fonction, mais une relation qui précise quelles sont les évolutions possibles en un pas. Ce système de transition, dans les deux cas peut être vu comme un graphe, dont les sommets sont les configurations, et les arcs correspondent à la fonction d'évolution en un pas.

Selon le lemme 7.3, chaque configuration X peut se décrire par un mot $[X]$ sur l'alphabet M longueur $\mathcal{O}$(size(X)) qui en décrit les emplacements utiles : si on fixe l'entrée w de longueur n, pour un calcul en espace $f(n)$, il y a donc moins de $\mathcal{O}(c^{f(n)})$) sommets dans ce graphe G_w, où $c > 1$ est le cardinal de l'alphabet M.

D'autre part, une très légère modification du lemme 7.4 montre qu'on peut construire un circuit C_w de taille $\mathcal{O}(f(n))$ tel que $C_w([X], [X']) = 1$ si et seulement si X' est une configuration successeur de la configuration X : les arcs de ce graphe sont donc donnés par un circuit C_w que l'on peut facilement déterminer.

Un mot w est accepté par l'algorithme A si et seulement s'il y a un chemin dans ce graphe G_w entre l'état initial $X[w]$ codant l'entrée w, et un état acceptant. On peut supposer sans perte de généralité qu'il y a un unique état acceptant X^*. Décider l'appartenance d'un mot w au langage reconnu par A est donc résoudre le problème REACH sur $< G_w, X[w], X^* >$.

On va alors traduire sous différentes formes tout ce que l'on sait sur le problème REACH. Tout d'abord, il est clair que le problème REACH se résout par exemple en temps et espace $\mathcal{O}(n^2)$, où n est le nombre de sommets, par un parcours en profondeur.

On en déduit :

Théorème 9.7 *Si $f(n) \geq \log n$, alors*

$$\text{NSPACE(f(n))} \subset \bigcup_{c \in \mathbb{N}} \text{TIME}(\text{c}^{\text{f(n)}}).$$

Démonstration: Soit L un problème de NSPACE(f(n)) reconnu par l'algorithme non-deterministe A. Par la discussion plus haut, on peut déterminer si $w \in L$ en résolvant le problème REACH sur $< G_w, X[w], X^* >$: on a dit que cela pouvait se fait en temps quadratique en le nombre de sommets, soit en temps $\mathcal{O}(c^{2\mathcal{O}(f(n))})$, où $c > 1$ est le cardinal de l'alphabet M.

□

9.2.5 Espace non-déterministe vs espace déterministe

On va maintenant affirmer que REACH se résout en espace $\log^2(n)$.

Proposition 9.1 REACH $\in$ SPACE($\log^2$n).

Démonstration: Soit $G = (V, E)$ le graphe orienté en entrée. Étant donnés deux sommets x et y de ce graphe, et i un entier, on note $CHEMIN(x, y, i)$ si et seulement s'il y a un chemin de longueur inférieure à 2^i entre x et y. On a $< G, u, v > \in$ REACH si et seulement si $CHEMIN(u, v, \log(n))$, où n est le nombre de sommets. Il suffit donc de savoir décider la relation $CHEMIN$ pour décider REACH.

L'astuce est de calculer $CHEMIN(x, y, i)$ récursivement en observant que l'on a $CHEMIN(x, y, i)$ si et seulement s'il existe un sommet intermédiaire z tel que $CHEMIN(x, z, i-1)$ et $CHEMIN(z, y, i-1)$. On teste alors à chaque niveau de la récursion chaque sommet possible z.

Pour représenter chaque sommet, il faut $\mathcal{O}(\log(n))$ bits. Pour représenter x, y, i, il faut donc $\mathcal{O}(\log(n))$ bits. Cela donne une récurrence de profondeur $\log(n)$, chaque étape de la récurrence nécessitant uniquement de stocker un triplet x, y, i et de tester chaque z de longueur $\mathcal{O}(\log(n))$. Au total, on utilise donc un espace $\mathcal{O}(\log(n)) * \mathcal{O}(\log(n)) = \mathcal{O}(\log^2(n))$. □

Théorème 9.8 (Savitch) *Si $f(n) \geq \log(n)$, alors*

$$\mathrm{NSPACE(f(n))} \subset \mathrm{SPACE(f(n)^2)}.$$

Démonstration: On utilise cette fois l'algorithme précédent pour déterminer s'il y a un chemin dans le graphe G_w entre $X[w]$ et X^*.

On remarquera que l'on a pas besoin de construire explicitement le graphe G_w mais que l'on peut utiliser l'algorithme précédent à la volée : plutôt que d'écrire complètement le graphe G_w, et ensuite de travailler en lisant dans cette écriture du graphe à chaque fois s'il y a un arc entre un sommet X et un sommet X', on peut de façon paresseuse, déterminer à chaque fois que l'on fait un test si $C_w(X, X') = 1$. □

Corollaire 9.1 PSPACE = NPSPACE

Principe: On a $\bigcup_{c\in\mathbb{N}} \mathrm{SPACE(n^c)} \subset \bigcup_{c\in\mathbb{N}} \mathrm{NSPACE(n^c)}$, par le théorème 9.3, et $\bigcup_{c\in\mathbb{N}} \mathrm{NSPACE(n^c)} \subset \bigcup_{c\in\mathbb{N}} \mathrm{SPACE(n^{2c})} \subset \bigcup_{c\in\mathbb{N}} \mathrm{SPACE(n^c)}$ par le théorème précédent. □

9.2.6 Espace logarithmique non-déterministe

En fait, on peut encore dire plus sur le problème REACH.

Définition 9.9 *On note*

$$\mathrm{NLOGSPACE} = \mathrm{NSPACE(log(n))}.$$

Théorème 9.9 REACH $\in$ NLOGSPACE.

Démonstration: Pour déterminer s'il y a un chemin entre u et v dans un graphe G, on devine le chemin arc par arc. Cela nécessite uniquement de garder le sommet que l'on est en train de visiter en plus de u et de v. Chaque sommet se codant par $\mathcal{O}(\log(n))$ bits, l'algorithme est en espace $\mathcal{O}(\log(n))$. □

Définition 9.10 *Soit $\mathcal{C}$ une classe de complexité. On note* co-$\mathcal{C}$ *pour la classe des langages dont le complémentaire est dans la classe $\mathcal{C}$.*

On parle ainsi de problème coNP, coNLOGSPACE, etc...
En fait, on peut montrer :

Théorème 9.10 *Le problème* REACH *est aussi dans* coNLOGSPACE.

Démonstration:
Supposons que l'on veuille savoir si t est accessible à partir d'un sommet s dans un graphe G à n sommets.

L'idée est d'arriver à calculer le nombre de sommets accessibles à partir d'un sommet s dans un graphe G. En effet, supposons que l'on connaisse ce nombre c : on peut alors prendre sommet par sommet, et deviner s'il est accessible ou non. Lorsqu'un sommet u est deviné comme accessible, l'algorithme vérifie ce fait en devinant un chemin de longueur au plus $n-1$ entre s et ce sommet u. Si l'algorithme échoue dans cette étape de deviner un chemin, l'algorithme termine en rejetant. L'algorithme compte alors le nombre d de sommets qui ont été vérifiés comme atteignables. Si d n'est pas c, l'algorithme refuse.

En d'autres termes, si l'algorithme devine correctement c sommets u comme atteignables à partir de s différent de t et arrive à se convaincre que chacun d'entre eux est atteignable en devinant un chemin entre s et chaque sommet u, l'algorithme peut être sûr que tout autre sommet n'est pas atteignable, et donc il peut accepter.

Cela donne un programme du type :

```
d ←0
for u sommet de G do
      test ←guess x ∈ {0,1}
       if test then
          [ suivre de façon non déterministe un chemin
          de longueur n−1 à partir de s et si aucun
          sommet rencontré sur ce chemin est u rejeter]
          if u = t then rejeter
          d ←d + 1
       endif
endfor
if d ≠ c then rejeter
else accepter
```

Il reste à se convaincre que l'on peut calculer c : l'idée est de calculer par récurrence le nombre de sommets du graphe c_i qui sont atteignables à partir de s par un chemin de longueur inférieure ou égale à i. Bien entendu, $c = c_{n-1}$. Le calcul de c_{i+1} se fait à partir de c_i selon le même principe que celui évoqué plus haut.

Concrètement, cela donne au final un algorithme comme celui là :

```
c_0 ←1
for i ← 0 to n − 2
    c_{i+1} ←0
    d ←0
    for v sommet de G do
      for u sommet de G do
        test ←guess x ∈ {0,1}
        if test then
            [ suivre de façon non déterministe un chemin
            de longueur i à partir de s et si aucun
            sommet rencontré est u rejeter]
            d ←d + 1
            if (u,v) est un arc de G then
                c_{i+1} ←c_{i+1} + 1
                [sortir de la boucle for u sommet de G
                pour passer au prochain v]
            endif
         endif
     endfor
     if d ≠ c_i then rejeter
endfor
d ←0
for u sommet de G do
       test ←guess x ∈ {0,1}
        if test then
            [ suivre de façon non déterministe un chemin
            de longueur n − 1 à partir de s et si aucun
            sommet rencontré est u refuser]
            if u = t then rejeter
            d ←d + 1
        endif
endfor
if d ≠ c_{n−1} then rejeter
else accepter
```

L'algorithme obtenu utilise bien un espace $\mathcal{O}(\log(n))$.

□

On en déduit :

Théorème 9.11 NLOGSPACE = coNLOGSPACE.

Démonstration: Il suffit de montrer que coNLOGSPACE ⊂ NLOGSPACE. L'inclusion inverse en découle car un langage L de NLOGSPACE aura son complémentaire dans coNLOGSPACE et donc aussi dans NLOGSPACE, d'où l'on déduit que L sera dans coNLOGSPACE.

Maintenant, pour décider si un mot w doit être accepté par un langage de coNLOGSPACE, on peut utiliser l'algorithme non-déterministe précédent qui utilise un espace logarithmique pour déterminer s'il existe un chemin entre $X[w]$ et X^* dans le graphe G_w. □

En fait, selon le même principe on peut montrer plus généralement.

Théorème 9.12 *Soit $f(n)$ une fonction telle que $f(n) \geq \log(n)$. Alors*

$$\text{NSPACE(f(n))} = \text{coNSPACE(f(n))}.$$

9.3 Quelques résultats & résumé

9.3.1 Théorèmes de hiérarchie

On dit qu'une fonction $f(n) \geq \log(n)$ est *constructible en espace*, si la fonction qui envoie $\mathbf{1}^n$ sur la représentation binaire de $\mathbf{1}^{f(n)}$ est calculable en espace $\mathcal{O}(f(n))$.

La plupart des fonctions usuelles sont constructibles en espace. Par exemple, n^2 est constructible en espace puisqu'un algorithme peut obtenir n en binaire en comptant le nombre de $\mathbf{1}$, et écrire n^2 en binaire par n'importe quelle méthode pour multiplier n par lui-même. L'espace utilisé est certainement en $\mathcal{O}(n^2)$.

Théorème 9.13 (Théorème de hiérarchie) *Pour toute fonction $f : \mathbb{N} \to \mathbb{N}$ constructible en espace, il existe un langage L qui est décidable en espace $\mathcal{O}(f(n))$ mais pas en espace $o(f(n))$.*

Démonstration: On considère le langage (très artificiel) L qui est décidé par l'algorithme B suivant :

- sur une entrée w de taille n, B calcule $f(n)$ en utilisant la constructibilité en espace de f, et réserve un espace $f(n)$ pour la simulation qui va venir.
- Si w n'est pas de la forme $< A > \mathbf{10}^*$, pour un certain algorithme A, alors l'algorithme B rejette.
- Sinon, B simule l'algorithme A sur le mot w pendant au plus $2^{f(n)}$ étapes pour déterminer si A accepte en ce temps avec un espace inférieur à $f(n)$.
  - si A accepte en ce temps et cet espace, alors B rejette ;
  - sinon B accepte.

Par construction, L est dans SPACE(f(n)), car la simulation n'introduit qu'un facteur constant dans l'espace nécessaire : plus concrètement, si A utilise un espace $g(n) \leq f(n)$, alors B utilise un espace au plus $dg(n)$ pour une constante d.

Supposons que L soit décidé par un algorithme A en espace $g(n)$ avec $g(n) = o(f(n))$. Il doit exister un entier n_0 tel que pour $n \geq n_0$, on ait $dg(n) < f(n)$. Par conséquent, la simulation par B de A sera bien complète sur une entrée de longueur n_0 ou plus.

Considérons ce qui se passe lorsque B est lancé sur l'entrée $< A > \mathbf{10}^{n_0}$. Puisque cette entrée est de taille plus grande que n_0, B répond l'inverse de

l'algorithme A sur la même entrée. Donc B et A ne décident pas le même langage, et donc l'algorithme A ne décide pas L, ce qui mène à une contradiction.

Par conséquent L n'est pas décidable en espace $o(f(n))$. □

Autrement dit :

Théorème 9.14 (Théorème de hiérarchie) *Soient $f, f' : \mathbb{N} \to \mathbb{N}$ des fonctions constructibles en espace telles que $f(n) = o(f'(n))$. Alors l'inclusion* $\mathrm{SPACE(f)} \subset \mathrm{SPACE(f')}$ *est stricte.*

Sur le même principe, on peut prouver :

Théorème 9.15 (Théorème de hiérarchie) *Soient $f, f' : \mathbb{N} \to \mathbb{N}$ des fonctions constructibles en espace telles que $f(n) = o(f'(n))$. Alors l'inclusion* $\mathrm{NSPACE(f)} \subset \mathrm{NSPACE(f')}$ *est stricte.*

On en déduit :

Théorème 9.16 $\mathrm{NLOGSPACE} \subsetneq \mathrm{PSPACE}$.

Démonstration: La classe NLOGSPACE est complètement incluse dans $\mathrm{SPACE}(\log^2 \mathrm{n})$ par le théorème de Savitch. Hors cette dernière est un sous-ensemble strict de SPACE(n), qui est inclus dans PSPACE. □

Sur le même principe, on obtient :

Définition 9.11 *Soit*

$$\mathrm{EXPSPACE} = \bigcup_{c \in \mathbb{N}} \mathrm{SPACE}(2^{\mathrm{n}^c}).$$

Théorème 9.17 $\mathrm{PSPACE} \subsetneq \mathrm{EXPSPACE}$.

Démonstration: La classe PSPACE est complètement incluse dans, par exemple, $\mathrm{SPACE}(\mathrm{n}^{\log(\mathrm{n})})$. Hors cette dernière est un sous-ensemble strict de $\mathrm{SPACE}(2^{\mathrm{n}})$, qui est inclus dans EXPSPACE. □

9.3.2 Classes de complexité usuelles

Les classes de complexité suivantes sont les classes les plus usuelles :

Définition 9.12

$$\mathrm{LOGSPACE} = \mathrm{SPACE}(\log \mathrm{n})$$

$$\mathrm{NLOGSPACE} = \mathrm{NSPACE}(\log \mathrm{n})$$

$$\mathrm{P} = \bigcup_{c \in \mathbb{N}} \mathrm{TIME}(\mathrm{n}^c)$$

$$\mathrm{NP} = \bigcup_{c \in \mathbb{N}} \mathrm{NTIME}(\mathrm{n}^c)$$

$$\mathrm{PSPACE} = \bigcup_{c \in \mathbb{N}} \mathrm{SPACE}(\mathrm{n}^c)$$

9.3.3 Relations entre classes

On a :

Théorème 9.18 LOGSPACE $\subset$ NLOGSPACE $\subset$ P $\subset$ NP $\subset$ PSPACE.

Démonstration: Chaque inclusion est une instance des résultats précédents. □

On a vu que NLOGSPACE $\subsetneq$ PSPACE mais on ne sait pas lesquelles des des inclusions strictes intermédiaires sont vraies.

Théorème 9.19 *Les classes déterministes sont closes par complément.*

Démonstration: Inverser l'état d'acceptation et de rejet dans un algorithme qui décide le langage. □

9.4 Problèmes complets

Nous allons maintenant montrer que les classes précédentes admettent des problèmes complets.

La notion de réduction, basée sur le temps polynomial, ne s'avère pas pertinente pour parler de problèmes complets pour des classes incluses dans P. Aussi, nous allons introduire une notion de réduction basée sur l'espace logarithmique.

9.4.1 Réduction en espace logarithmique

Nous commençons par introduire la notion de fonction calculable en espace logarithmique :

Définition 9.13 (Fonction calculable en espace logarithmique) *Soient M et N deux alphabets. Une fonction $f : M^* \to N^*$ est* calculable en espace logarithmique *s'il existe un algorithme A, tel que pour tout mot w, A avec l'entrée w termine en utilisant un espace $\mathcal{O}(\log(n))$, où $n = |w|$, avec le résultat $f(w)$. On utilise toujours la convention que l'on ne compte pas l'entrée, w qui est en lecture seulement, et la sortie $f(w)$ qui est en écriture seulement dans l'espace utilisé.*

En fait, il s'agit d'une contrainte plus forte que d'être calculable en temps polynomial.

Proposition 9.2 *Une fonction calculable en espace logarithmique est calculable en temps polynomial.*

Démonstration: Utiliser le principe de la preuve du théorème 9.7. □

Le résultat suivant est vrai (mais la preuve n'est pas directe contrairement au résultat similaire pour les fonctions calculables en temps polynomial. La difficulté est que le résultat intermédiaire de la première fonction ne peut pas être stocké en mémoire car sa taille n'est pas nécessairement logarithmique).

Proposition 9.3 (Stabilité par composition) *La composée de deux fonctions calculables en espace logarithmique est calculable en espace logarithmique.*

Démonstration: Pour calculer la composition de f et de g calculables en espace logarithmique, on ne peut pas, sur une entrée w, calculer $w' = f(w)$, puis $g(w')$ car $w' = f(w)$ peut être de taille trop importante. Cependant, chaque pas de calcul de $g(w')$ nécessite au plus de lire une lettre de w', disons la lettre numéro i de w'.

Or, étant donné le numéro i de cette lettre en binaire, on peut déterminer la valeur de cette lettre en espace logarithmique : simuler le calcul de f tout en maintenant un compteur qui compte le nombre de symboles j écrits jusque là, jusqu'à ce que $j = i$.

En maintenant la valeur de i, on simule le calcul de g sur $w' = f(w)$ sans jamais écrire complètement w' : à chaque fois que l'on a besoin de lire une nouvelle lettre de w', on utilise la procédure ci-dessus pour déterminer la valeur de cette lettre. □

On peut alors introduire :

Définition 9.14 (Réduction) *Soient A et B deux problèmes d'alphabet respectifs M_A et M_B, et de langages respectifs M_A et M_B. Une* réduction logspace de A vers B, *ou encore* réduction en espace logarithmique, *est une fonction $f : M_A^* \to M_B^*$ calculable en espace logarithmique telle que $w \in L_A$ ssi $f(w) \in L_B$. On note $A \leq_L B$ lorsque A se réduit à B de cette façon.*

Il découle de la proposition 9.2 que cette notion de réduction est plus contraignante que la précédente.

Corollaire 9.2 *Soient A et B deux problèmes. Supposons $A \leq_L B$. Alors $A \leq B$.*

Exactement pour les mêmes raisons qu'auparavant, on a :

Théorème 9.20 *$\leq_L$ est un préordre :*

1. *$L \leq_L L$*
2. *$L_1 \leq_L L_2$, $L_2 \leq_L L_3$ implique $L_1 \leq_L L_3$.*

Définition 9.15 *Deux problèmes L_1 et L_2 sont équivalents pour $\leq_L$, noté $L_1 \equiv_L L_2$, si $L_1 \leq_L L_2$ et si $L_2 \leq_L L_1$.*

et

Proposition 9.4 (Réduction) *Si $A \leq_L B$, et si $B \in$ P alors $A \in$ P.*

Proposition 9.5 (Réduction) *Si $A \leq_L B$, et si $A \notin$ P alors $B \notin$ P.*

Théorème 9.21 *Les mêmes résultats sont vrais si on remplace* P *par les classes de complexité* NLOGSPACE, NP, PSPACE *dans les propositions précédentes.*

Définition 9.16 ($\mathcal{C}$-complétude) *Soit $\mathcal{C}$ une classe de langages. Un problème A est dit $\mathcal{C}$-complet pour $\leq_L$ si*

1. *il est dans la classe $\mathcal{C}$*
2. *tout autre problème B de la classe $\mathcal{C}$ est tel que $B \leq_L A$.*

Corollaire 9.3 *Soit $\mathcal{C}$ une classe de langages. Tous les problèmes $\mathcal{C}$-complets sont équivalents au sens de $\equiv_L$.*

9.4.2 Retour sur la NP-complétude

Dans le chapitre précédent nous avons montré la NP-complétude d'un certain nombre de problèmes, pour la relation $\leq$ basée sur le temps polynomial. On observera que dans chacune des preuves, la fonction de réduction utilisée est en fait calculable non seulement en temps polynomial mais en espace logarithmique.

Autrement dit :

Théorème 9.22 *Chacun des problèmes* NP*-complets évoqués dans le chapitre précédent sont en fait* NP*-complets pour $\leq_L$ et pas seulement pour $\leq$.*

Lorsqu'on parlera de complétude dans la suite, on parlera toujours de complétude pour $\leq_L$.

9.4.3 Un problème PSPACE-complet

Comme pour la NP-complétude, en jouant avec les définitions, on peut prouver assez facilement l'existence d'un problème PSPACE-complet.

Théorème 9.23 *Le problème*

$$K = \{< A, w, \mathbf{1}^n > \mid \textit{l'algorithme}$$

$$A \textit{ accepte l'entrée } w \textit{ en espace } n\}$$

est PSPACE*-complet.*

Démonstration: Le problème est dans PSPACE car sur une entrée $< A, w, \mathbf{1}^n >$ il suffit de simuler l'algorithme A et de vérifier qu'il accepte w en espace n, ce qui se fait en espace linéaire.

Soit L un problème de PSPACE. L est accepté par un algorithme en espace $p(n)$ pour un certain polynôme p. L se réduit à K par la fonction qui à un mot w associe le triplet $< A, w, \mathbf{1}^{p(|w|)} >$. Cette fonction est bien calculable en espace logarithmique en la longueur de w. □

Cependant, le problème précédent n'est pas très naturel. Comme dans le chapitre précédent, nous allons parler de problèmes plus naturels en utilisant l'équivalence entre algorithmes et circuits.

Peut-être que le problème PSPACE-complet le plus fondamental est celui de la satisfiabilité quantifiée d'un circuit booléen, noté QCIRCUITSAT : étant

donné un circuit booléen $C(x_1, \cdots, x_n)$ à n variables, peut-on déterminer s'il existe une valeur pour x_1 telle que pour toute valeur de x_2, il existe une valeur pour la variable x_3 telle que pour toute valeur de x_4 il existe une valeur pour la variable x_5 telle que toute valeur de x_6 ...etc, telles que $C(x_1, x_2, \cdots, x_n) = 1$?

En d'autres termes, $\exists x_1 \forall x_2 \exists x_3 \cdots\ C(x_1, \cdots, x_n) = 1$?

Les quantificateurs de rang impairs sont existentiels, et ceux de rang pair sont universels.

Remarque 9.6 *En fait, pour s'assurer d'une alternance stricte de quantificateurs de ce type, devant une suite de quantificateurs qui ne serait pas exactement de ce type, on peut toujours insérer des variables "stupides" qui n'interviennent pas dans le circuit C pour s'y ramener. Par exemple $\forall x_1 \forall x_2 \exists x_3\ C(x_1, x_2, x_3) = 1$ peut se réécrire $\exists x \forall x_1 \exists y \forall x_2 \exists x_3\ C(x_1, x_2, x_3) = 1$.*

Théorème 9.24 *Le problème* QCIRCUITSAT *est* PSPACE*-complet.*

Démonstration: Le problème QCIRCUITSAT est dans PSPACE car il est facile de construire un algorithme récursif qui le résout : pour une formule de la forme $\exists x_1 \phi$ (respectivement : $\forall x_1 \phi$) on fixe la valeur de x_1 à 0, on propage cette valeur pour x_1 dans ϕ pour obtenir $\phi_{x_1=0}$, et on s'appelle récursivement pour savoir si la formule $\phi_{x_1=0}$ est satisfiable. On fixe alors la valeur de x_1 à 1, on propage cette valeur pour x_1 dans ϕ pour obtenir $\phi_{x_1=1}$, et on s'appelle récursivement pour savoir si la formule $\phi_{x_1=1}$ est satisfiable. On accepte si et seulement si $\phi_{x_1=0}$ ou (resp. et) $\phi_{x_1=1}$ sont satisfiables. L'algorithme fonctionne avec $\mathcal{O}(n)$ appels récursifs, chacun utilisant un espace constant, et donc en espace total $\mathcal{O}(n)$, où n est la taille de la formule.

Pour montrer qu'il est complet, on va utiliser le principe de la preuve du théorème de Savitch. Supposons que le problème L soit accepté en espace polynomial. On va écrire par un circuit quantifié $C_i([X], [X'])$ le fait qu'il existe un chemin de longueur inférieure à 2^i dans le graphe G_w entre les configurations X et X'. Le graphe G_w est de taille $2^{p(n)}$ pour un certain polynôme p, où $n = |w|$.

Ce circuit s'obtient par récurrence sur i. Pour $i = 0$, c'est le circuit qui donne les arcs du graphe G_w. Pour $i > 0$, on a envie d'écrire $C_i([X], [X'])$ comme

$$\exists [X'']\ C_{i-1}([X], [X'']) \wedge C_{i-1}([X''], [X']).$$

Cependant si l'on fait ainsi, le circuit obtenu pour C_i est au moins de taille double de celui de C_{i-1}, et donc C_i sera de taille exponentielle en i.

L'idée est de quantifiée en réutilisant l'espace comme dans la preuve du théorème de Savitch : on écrit $C_i([X], [X'])$ comme

$$\exists [X''] \forall [D_1] \forall [D_2]\ C'_i([X], [X'], [D_1], [D_2], [X''])$$

où $C'_i([X], [X'], [D_1], [D_2], [X''])$ teste la relation

$$(([D_1] = [X] \wedge [D_2] = [X'']) \vee (([D_1] = [X''] \wedge [D_2] = [X']) \Rightarrow C_{i-1}([D_1], [D_2]).$$

Cette fois la taille de C_i sera de l'ordre de celle de C_{i-1} plus $\mathcal{O}(p(n))$. Un mot w est dans le langage L si et seulement si $C_{p(n)}([X[w]], [X^*])$.

La fonction qui à w associe le circuit quantifié $C_{p(n)}([X[w]], [X^*])$ est bien calculable en espace logarithmique. □

En utilisant le fait que la satisfiabilité d'un circuit peut se ramener à celui d'une formule du calcul propositionnel en forme normale conjonctive (c'est-à-dire en introduisant une variable booléenne par porte du circuit comme dans la preuve du théorème 8.17) on obtient :

Le problème QSAT (aussi appelé QBF) consiste étant donnée une formule du calcul propositionnel en forme normale conjonctive ϕ avec les variables $x_1, x_2, \cdots, x_n$ (c'est-à-dire un instance de SAT) à déterminer si $\exists x_1 \forall x_2 \exists x_3 \cdots \ \phi(x_1, \cdots, x_n)$?

Théorème 9.25 *Le problème* QSAT *est* PSPACE*-complet.*

Les jeux stratégiques sur les graphes donnent naturellement naissance à des problèmes PSPACE-complet. Par exemple.

Le jeu GEOGRAPHY consiste à se donner un graphe orienté $G = (V, E)$. Le joueur 1 choisit un sommet u_1 du graphe. Le joueur 2 doit alors choisir un sommet v_1 tel qu'il y ait un arc de u_1 vers v_1. C'est alors au joueur 1 de choisir un autre sommet u_2 tel qu'il y ait un arc de v_1 vers u_2, et ainsi de suite. On a pas le droit de repasser deux fois par le même sommet. Le premier joueur qui ne peut pas continuer le chemin $u_1 v_1 u_2 v_2 \cdots$ perd. Le problème GEOGRAPHY consiste à déterminer étant donné un graphe G et un sommet de départ pour le joueur 1, s'il existe une stratégie gagnante pour le joueur 1.

Théorème 9.26 *Le problème* GEOGRAPHY *est* PSPACE*-complet.*

9.4.4 Un problème P complet

Rappelons que le problème CIRCUITVALUE consiste, étant donnés un circuit $C(x_1, \cdots, x_n)$, et un ensemble de valeurs booléennes pour ses variables $\overline{x} \in \{0,1\}^n$, à déterminer si $C(\overline{x}) = 1$.

Théorème 9.27 *Le problème* CIRCUITVALUE *est* P*-complet.*

Démonstration: On a déjà vu dans le chapitre précédent que le problème CIRCUITVALUE était dans P.

Soit L un langage de P. On sait qu'il existe une famille de circuits de taille polynomiale $(C_n)_{n\in\mathbb{N}}$ qui reconnaît L (Théorème 8.2). L se réduit au problème CIRCUITVALUE par la fonction qui à un mot w associe $<< C_{|w|} >, w >$, en observant que la preuve du théorème 8.2 ne montre pas seulement que la fonction qui à l'entier n associe la description $< C_n >$ du circuit C_n est calculable en temps polynomial, mais aussi que cette fonction est calculable en espace logarithmique. □

On peut en déduire que d'autres problèmes sont P-complets. Tout d'abord, comme d'habitude on peut exprimer l'équivalence entre circuits et formules du calcul propositionnel en forme normale conjonctive.

Le problème SATVALUE consiste, étant donnés une formule du calcul propositionnel en forme normale conjonctive ϕ avec les variables $x_1, x_2, \cdots, x_n$ (c'est-à-dire un instance de SAT), et un vecteur de valeurs $\overline{x} \in \{0,1\}^n$, à déterminer la valeur de $\phi(\overline{x})$.

Théorème 9.28 *Le problème* SATVALUE *est* P*-complet.*

Démonstration: Le problème est clairement dans P. D'autre part, on peut transformer tout circuit en une formule équivalente du calcul propositionnel en forme normale conjonctive en introduisant une variable par porte du circuit comme dans la preuve du théorème 8.17, le tout en espace logarithmique. □

En fait, on peut aussi prouver.

Le problème MONOTONECIRCUITVALUE consiste, étant donné un circuit booléen $C(x_1, \cdots, x_n)$ sans porte $\neg$ et un ensemble de valeurs booléennes pour ses variables $\overline{x} \in \{0,1\}^n$, à déterminer si $C(\overline{x}) = 1$.

Théorème 9.29 *Le problème* MONOTONECIRCUITVALUE *est* P*-complet.*

9.4.5 Un problème NLOGSPACE-complet

Théorème 9.30 *Le problème* REACH *est* NLOGSPACE*-complet.*

Démonstration: Nous avons déjà vu que le problème REACH était dans NLOGSPACE.

Montrons qu'il est complet. Soit L un langage de NLOGSPACE. L se réduit au problème REACH par la fonction qui à un mot w associe l'instance $< G_w, X[w], X^* >$: cette fonction est bien calculable en espace logarithmique. □

Rappelons que le problème 3-SAT est NP-complet. Le problème 2-SAT, c'est-à-dire celui de la satisfiabilité d'une formule du calcul propositionnel en forme normale conjonctive avec 2-littéraux par clause est lui solvable en temps polynomial.

En fait, il est NLOGSPACE-complet :

Théorème 9.31 *Le problème* 2-SAT *est* NLOGSPACE*-complet.*

9.4.6 Des problèmes EXPTIME et NEXPTIME-complets

Soit C un circuit tel que chaque porte soit le prédécesseur d'au plus deux portes. Chaque sommet du graphe associé à donc au plus 4-voisins que l'on peut numéroté de 0 à 3. Sa *représentation succincte* consiste à se donner un autre circuit avec plusieurs sorties : sur l'entrée $<< i >, < k >>$, où $< i >$ et $< k >$ sont les codages en binaire du numéro d'un sommet du graphe du circuit C, et de $k \in \{0,1,2,3\}$, cet autre circuit produit $<< j >, s >$ où s est l'étiquette de la porte, c'est-à-dire soit $\neg$, $\vee$, $\wedge$, **0** ou **1** où le numéro d'une variable en binaire, et $< j >$ est le codage en binaire du voisin numéro k.

Nous laissons en exercice les résultats suivants.

Le problème SUCCINTSAT consiste étant donné la représentation succincte d'un circuit à déterminer si le circuit correspondant est satisfiable.

Théorème 9.32 *Le problème* SUCCINTSAT *est* NEXPTIME*-complet.*

La représentation succincte d'une valuation de variables booléennes consiste à se donner un circuit qui prend en entrée le codage en binaire d'un entier représentant le numéro d'une variable et qui répond sa valeur.

Le problème SUCCINTSATVALUE consiste étant donné la représentation succincte d'un circuit, et la représentation succincte d'une valuation de ses variables, à déterminer si le circuit correspondant vaut 1 sur cette entrée.

Théorème 9.33 *Le problème* SUCCINTSATVALUE *est* EXPTIME*-complet.*

9.5 Notes bibliographiques

Ce chapitre contient des résultats classiques en complexité. Nous avons utilisé ici principalement leur présentation dans les ouvrages [Arora and Barak, 2009], [Papadimitriou, 1994], [Sipser, 1997] et [Kozen, 2006]. Les rares considérations sur les structures arbitraires sont tirées de [Poizat, 1995]. En particulier, le théorème 9.1 est dû à [Michaux, 1989].

Chapitre 10

Complexité parallèle

Une machine parallèle est une machine capable d'effectuer plusieurs opérations simultanément. Les machines parallèles peuvent résoudre certains problèmes plus rapidement que les machines séquentielles, qui elles ne peuvent effectuer qu'un nombre borné d'opérations à la fois.

Nous allons introduire le modèle de calcul parallèle PRAM, qui est basé sur le modèle RAM. Nous verrons ensuite les modèles parallèles basés sur les circuits. Nous présenterons alors des résultats reliant ces modèles parallèles, et aussi ces modèles aux modèles séquentiels.

10.1 Retour sur les machines RAM

10.1.1 Calculabilité et machines RAM

Avant de pouvoir parler de PRAM, nous devons revenir un peu sur le modèle des machines RAM, et sur le pourquoi nous ne les avions pas mises dans l'énoncé du théorème 8.1 du chapitre 8 affirmant l'équivalence de la plupart des modèles à un temps polynomial près.

Nous avons introduit le modèle des machines RAM dans la définition 5.5 du chapitre 5. Une machine RAM travaille sur les entiers, possède des registres $x_1, x_2, \cdots$, en nombre non-borné, et est autorisée, selon la définition 5.5 à effectuer des opérations du type :

Définition 10.1 (Opérations RAM)

1. $x_i \leftarrow 0$
2. $x_i \leftarrow x_i + 1$
3. $x_i \leftarrow x_i \ominus 1$
4. $x_i \leftarrow x_j$
5. $x_i \leftarrow x_{x_j}$
6. $x_{x_i} \leftarrow x_j$
7. ***if*** $x_i = 0$ ***then*** $ctl\_state \leftarrow j$ ***else*** $ctl\_state \leftarrow j'$

Nous avons montré que l'on pouvait simuler les machines RAM par une machine de Turing (théorème 5.7) travaillant sur un alphabet fini.

En fait, si l'on voit cette définition comme un algorithme sur une structure dont l'ensemble de base est les entiers, le théorème 5.8 montre qu'on peut aussi simuler une machine RAM par une machine de Turing qui travaille sur les entiers (c'est-à-dire dont les cases du ruban contiennent des entiers).

Du point de vue de la calculabilité, cela ne fait pas vraiment de différence, puisque l'on peut simuler des calculs sur les entiers par des calculs sur leur codage en binaire.

Dans le même esprit, du point de vue de la calculabilité, on pourrait aussi autoriser des types d'opérations plus générales dans le modèle comme par exemple ajouter aux opérations de la définition précédente les opérations du type $x_i \leftarrow \max(x_i, x_j)$, $x_i \leftarrow x_i + x_j$, $x_i \leftarrow x_i * x_j$, etc. En fait plus généralement, toute opération du type $x_i \leftarrow f(x_{i_1}, x_{i_2}, \cdots, x_{i_k})$ où f est une fonction calculable sur les entiers, ou dont la représentation binaire de la valeur est calculable à partir de la représentation binaire des arguments, si l'on préfère, calculable au sens des structures finies.

10.1.2 Complexité et machines RAM

S'il on veut parler de complexité, il est vrai qu'un calcul en temps t d'une machine de Turing se simule en un temps polynomial en t par une machine RAM au sens de la définition 5.5 : cela découle du théorème 5.6, en voyant une machine de Turing comme une machine de Turing sur la structure $\mathfrak{B} = (\{0,1\}, \mathbf{0}, \mathbf{1}, =)$.

Cependant la réciproque n'est pas nécessairement vraie, car une machine RAM comme définie dans la définition 5.5 peut à priori manipuler des entiers très grands en temps unitaire, et donc un temps t est pas simulable facilement en un temps polynomial en t, si la taille des entiers impliqués ne reste pas polynomiale en la taille des entrées et en t.

Aussi, si l'on souhaite parler de complexité sur une machine RAM, on mesure généralement la taille des entrées de la façon suivante : la taille d'un entier n est $|n| = \lceil \log(n) \rceil$ (le nombre de bits nécessaire pour l'écrire en binaire). Une entrée d'une machine RAM est un mot $w = w_1 w_2 \cdots w_n$ sur $\mathbb{N}^*$: on mesure sa taille $|w|$ comme $|w| = \sum_{i=1}^{n} |w_i|$.

Remarque 10.1 *Observons que cette subtilité est parfaitement inutile, si l'on ne parle que de problèmes et d'algorithmes impliquant des entiers de taille polynomiale en la taille des entrées. Mais à priori, dans le modèle RAM, on peut avoir des entiers de taille arbitraire.*

Si l'on utilise cette convention, un temps polynomial t en la (cette) taille des entrées d'une machine RAM se simule bien en temps polynomial en t sur une machine de Turing ou sur toute structure finie et réciproquement : il suffit d'observer les preuves.

Autrement dit, avec cette convention, le temps polynomial est le même au sens de tous les modèles considérés dans le chapitre 5 (mis à part les machines à k-compteurs).

On va fixer dorénavant cette convention, qui nous permettra de parler de machines RAM (et de PRAM) avec une notion de complexité qui est bien la complexité usuelle en algorithmique.

10.1.3 Étendre les opérations admissibles

Cette convention étant fixée, on peut aussi alors autoriser des types d'opérations plus générales dans le modèle comme par exemple $x_i \leftarrow \max(x_i, x_j)$, $x_i \leftarrow x_i + x_j$.

Mais pas des opérations comme $x_i \leftarrow x_i * x_j$: le problème est que cela permet d'écrire $x \leftarrow x^2$ et qu'en partant avec $x = 2$, en effectuant k opérations $x \leftarrow x^2$ consécutives on construit l'entier $x = 2^{2^k}$, qui est de taille 2^k : on peut donc créer des entiers énormes trop vite, et quitter le temps polynomial au sens des structures finies.

Définition 10.2 (Opération admissible) *En fait, on dira qu'une opération du type* $x_i \leftarrow f(x_{i_1}, x_{i_2}, \cdots, x_{i_k})$ *est* admissible *si*

1. *il est possible de calculer la représentation binaire de* $f(x_{i_1}, x_{i_2}, \cdots, x_{i_k})$ *en temps polynomial en la taille des représentations binaires de ses arguments.*
2. *la taille de* $f(x_{i_1}, x_{i_2}, \cdots, x_{i_k})$ *est au plus une constante plus le maximum de la taille de ses arguments.*

Remarque 10.2 *C'est la deuxième condition qui est problématique pour le produit, car* $\log(x_i x_j) = \log(x_i) + \log(x_j)$.

Remarque 10.3 *Observons que cette subtilité sur la deuxième condition est parfaitement inutile, si l'on ne considère des algorithmes dont les entiers manipulés restent de taille polynomiale en la taille des entrées.*

Avec des opérations admissibles, la taille des entiers au temps t reste polynomial en t et en la taille des entrées, et donc avec une notion de complexité qui est bien la complexité usuelle en algorithmique.

10.2 Le modèle PRAM

Ceci étant dit, revenons à la complexité parallèle.

10.2.1 Définitions

Plusieurs modèles ont été développés pour étudier les algorithmes parallèles. Peut-être que le plus prédominant en algorithmique parallèle est le modèle des machines PRAM (*Parallel Random Access Machine*), construit à partir du modèle RAM.

Dans le modèle PRAM différentes machines, ou *processeurs*, interagissent de façon synchrone par l'intermédiaire d'une mémoire partagée.

Chaque processeur correspond à une machine RAM, qui possède comme toute machine RAM, une mémoire, c'est-à-dire des registres, en nombre non-borné. Chaque processeur peut effectuer toutes les opérations usuelles des machines RAM sur sa propre mémoire, et ce, comme dans le modèle RAM en temps unité. Chaque processeur n'a pas accès aux registres de la mémoire locale d'un autre processeur.

Par contre, il y a aussi une mémoire globale, partagée : tous les processeurs ont accès en lecture et en écriture à cette mémoire partagée. L'accès par un processeur à un emplacement mémoire partagé se fait là aussi en temps unité.

Si l'on préfère, une machine PRAM peut se définir de la façon suivante :

Définition 10.3 *Une machine* PRAM (Parallel Random Access Machine) *correspond à un suite* $\Pi_1, \Pi_2, \cdots, \Pi_q$ *de programmes, chaque programme* Π_k *étant du type*

```
ctl_state ←0
repeat
   seq
 <instructions>
  endseq
until ctl_state = q_a
```

où <instructions> est une suite finie d'instructions de l'un des types suivants (c'est-à-dire une instruction d'une machine RAM au sens de la définition 10.1)

1. $x_{k,i} \leftarrow 0$
2. $x_{k,i} \leftarrow x_{k,i} + 1$
3. $x_{k,i} \leftarrow x_{k,i} \ominus 1$
4. $x_{k,i} \leftarrow x_{k,j}$
5. $x_{k,i} \leftarrow x_{k,x_{k,j}}$
6. $x_{k,x_{k,i}} \leftarrow x_{k,j}$
7. ***if*** $x_{k,i} = 0$ ***then*** $ctl\_state \leftarrow j$ ***else*** $ctl\_state \leftarrow j'$

ou de l'un des types suivants (c'est-à-dire une instruction d'accès en lecture ou écriture à la mémoire partagée)

1. $x_{k,i} \leftarrow x_{0,j}$
2. $x_{k,i} \leftarrow x_{0,x_{k,j}}$
3. $x_{0,i} \leftarrow x_{k,j}$
4. $x_{0,x_{k,i}} \leftarrow x_{k,j}$

Ici chaque variable est doublement indexée : $x_{k,i}$ *désigne la variable locale numéro* i *du programme* k. $x_{0,i}$ *désigne la variable globale numéro* i. *Bien entendu chaque variable ctl_state est locale.*

Le nombre q *de RAM dans le programme* PRAM *est supposé ne pas être une constante, mais une fonction* $q(n)$ *de la taille* n *de l'entrée. Pour éviter les*

cas pathologiques[1]*, on se restreint au cas* L-uniforme*, c'est-à-dire au cas où la fonction qui à* $\mathbf{1}^n$ *associe* $q(n)$ *et les programmes* $\Pi_1, \Pi_2, \cdots, \Pi_{q(n)}$ *est calculable en espace logarithmique.*

Une configuration $X = (X_1, \cdots, X_q)$ *d'un programme* PRAM *est la donnée des états de chacun des programmes (au sens du postulat* 1*, c'est-à-dire une structure qui code l'état des registres de chacun des programmes). On passe d'une configuration* X *à sa configuration successeur* $X' = (X'_1, \cdots, X'_q)$ *si chacun des* X'_k *est le successeur immédiat de* X_k *dans le programme* k *: autrement dit, en effectuant simultanément une étape de chaque programme.*

Un programme PRAM *termine si chacun des programmes* Π_k *termine, c'est-à-dire a sa variable* $ctl\_state = q_a$*. On lit le résultat éventuel d'un calcul dans la mémoire globale.*

Remarque 10.4 *Une machine* PRAM *avec un seul processeur (*$q = 1$*) correspond à une machine RAM.*

Remarque 10.5 *On mesure la taille des entrées comme dans le modèle RAM pour que le temps polynomial corresponde bien au temps polynomial dans le cas où il y n'y a qu'un nombre borné de processeurs.*

Cependant, il faut préciser comment sont gérés les éventuels conflits en lecture ou écriture dans la mémoire partagée.

Cela donne naissance à plusieurs variantes du modèle :

Définition 10.4

1. *Le modèle* CREW *(*Concurrent Read Exclusive Write*) est le modèle le plus usuel : le nombre de lectures simultanées d'un même emplacement mémoire partagé n'est pas borné. Par contre, l'accès en écriture d'un emplacement mémoire partagé n'est possible que par un processeur à la fois.*
2. *Le modèle* CRCW *(*Concurrent Read Concurrent Write*) est le modèle le plus puissant : le nombre d'accès en lecture et en écriture d'un même emplacement mémoire n'est pas borné. Il y a alors plusieurs variantes pour déterminer le résultat d'une écriture simultanée :*
   - *le mode* consistant *: on impose que tous les processeurs qui écrivent simultanément doivent écrire la même valeur.*
   - *le mode* arbitraire *: on prend la valeur d'un processeur qui écrit,*
   - *le mode* prioritaire *: on prend la valeur du processeur d'indice le plus petit.*
   - *le mode* fusion *: on effectue au vol le calcul d'une fonction commutative et associative des différentes valeurs écrites, comme un* ET *logique, ou un* OU *logique, ou un* max*, etc.*
3. *Le modèle* EREW (Exclusive Read Exclusive Write*) est le modèle le plus restrictif, mais peut-être aussi le plus proche des machines réelles : les accès simultanées en lecture et écriture ne sont pas autorisés.*

[1]Par exemple, où l'on utiliserait un nombre de processeurs non-calculable, et où l'on utiliserait cette information.

Clairement ce qui peut se programmer avec un EREW peut se programmer avec un CREW, et ce qui peut se programmer avec un CREW peut se programmer avec un CRCW en le même temps, et avec le même nombre de processeurs.

Remarque 10.6 *Exactement pour la même raison que pour les RAM, on peut plus généralement autoriser des instructions du type* $x_i \leftarrow \max(x_i, x_j)$, $x_i \leftarrow x_i + x_j$, *ou plus généralement des opérations admissibles* $x_i \leftarrow f(x_{i_1}, x_{i_2}, \cdots, x_{i_k})$ *dans les opérations de base autorisées par le modèle.*

Dans la suite, nous décrirons les algorithmes autant que possible à un haut niveau (celui des cours d'algorithmique parallèle dont nous nous sommes inspirés), en évitant de trop alourdir les notations : en particulier, les variables utilisées seront (sauf mention explicites) toutes des variables globales.

Exemple 10.1 *Par exemple, on s'autorisera à comparer des variables, même si de façon puriste pour comparer* x_i *à* x_j *il faudrait plutôt soustraire l'un à l'autre et regarder si le résultat strictement positif ou non. De même, on utilisera des constructions du type* ***for*** $i \leftarrow 1$ ***to*** n ***in parallel*** <*inst*> *qui signifie qu'on effectue sur chaque processeur* $1 \leq i \leq n$ *en parallèle l'instruction* <*inst*>.

10.2.2 Séparation des modèles

Commençons par introduire quelques algorithmes, de façon à avoir un peu d'intuition de ce qui se passe en programmation parallèle. Cela nous permettra d'autre part de séparer les modèles.

Voici un algorithme pour le calcul en temps constant du maximum d'un tableau A de n éléments sur une PRAM CRCW avec $\mathcal{O}(n^2)$ processeurs.

```
for i←1 to n in parallel
    m[i]←vrai
for i,j←1 to n in parallel
    if A[i] < A[j] then m[i]←faux
for i←1 to n in parallel
    if m[i] = vrai then out←A[i]
```

Ce programme PRAM opère en mode CRCW consistant, car tous les processeurs écrivent la même valeur, à savoir $faux$.

Le calcul du maximum ne pouvant se faire en séquentiel qu'en temps au moins n, on voit qu'une machine parallèle est capable de le réaliser en temps constant.

En fait, selon un principe similaire, on peut faire un *ou* logique ou un *et* logique en temps constant en mode CRCW consistant avec respectivement :

```
m←faux
for i←1 to n in parallel
    if A[i] then m←vrai
```

et

```
m ←vrai
for i ← 1 to n in parallel
    if not A[i] then m ←faux
```

Plus généralement, une PRAM CRCW de n processeurs est capable selon le même principe de calculer en temps constant $\bigoplus_{1\leq i\leq n} e_i$ de n éléments $e_1, e_2, \cdots, e_n$ où $\oplus$ est une loi associative : appelons ce problème le *problème du produit itéré*.

L'algorithme du maximum précédent montre par ailleurs que les PRAM CRCW sont strictement plus puissantes que les PRAM CREW : en effet, une telle opération ne peut pas s'effectuer en moins de $\mathcal{O}(\log n)$ étapes sur une PRAM CREW : avec ce dernier modèle, en temps constant on peut fusionner au plus qu'un nombre constant d'éléments en une seule valeur.

D'autre part, les PRAM CREW sont strictement plus puissantes que les PRAM EREW : on peut en effet avec les premières déterminer si un élément e donné se trouve parmi n éléments $e_1, e_2, \cdots, e_n$ distincts deux à deux en temps constant : on initialise un booléen res à $faux$, ensuite en parallèle chaque processeur compare l'élément e à e_i (il y a donc des lectures simultanées) et effectue $res \leftarrow vrai$ s'il y a égalité. Comme les e_i sont distincts deux à deux, au plus un processeur va écrire la variable partagée res, et donc cela s'implémente bien dans le modèle CREW. Par contre, avec une PRAM EREW, les processeurs ne peuvent pas accéder simultanément à l'élément e. Pour que les comparaisons puissent se faire en temps constant, il faut un nombre de copies de e de l'ordre de n ce qui ne peut pas se faire en moins de $\mathcal{O}(n)$ étapes de calcul.

Plus généralement, sur une PRAM EREW, la diffusion d'une information à n processeurs nécessite un temps $\mathcal{O}(\log(n))$ car le nombre de processeurs informés ne peut pas faire plus que doubler à chaque étape.

10.2.3 Théorèmes de simulation

Nous venons de séparer les modèles en exhibant un algorithme pour lequel un facteur $\mathcal{O}(\log(p))$, où p est le nombre de processeurs, sépare les temps d'exécutions des variantes PRAM. Ce facteur $\mathcal{O}(\log(p))$ est maximal :

Théorème 10.1 *Tout algorithme sur une machine* PRAM CRCW *à p processeurs ne peut pas être plus de $\mathcal{O}(\log(p))$ fois rapide que le meilleur algorithme* PRAM EREW *à p-processeurs pour le même problème.*

Démonstration: On supposera que la PRAM CRCW opère en mode consistant. On va montrer comment simuler les écritures concurrentes en modes écritures exclusives, le cas des lectures se traitant de manière similaire.

Considérons un pas de l'algorithme CRCW à p-processeurs : on va le simuler en $\mathcal{O}(\log(p))$ étapes d'un algorithme EREW. La puissance de calcul étant la même, on se concentre sur les accès mémoire. L'algorithme EREW utilise un tableau auxiliaire A de taille p. Quand le processeur P_i de l'algorithme EREW

écrit une donnée x_i à l'adresse l_i en mémoire, le processeur P_i de l'algorithme EREW effectue l'écriture exclusive $A[i] \leftarrow (l_i, x_i)$. On trie alors le tableau A suivant la première coordonnée en temps $\mathcal{O}(\log(p))$, du moins si l'on suppose disposer d'un algorithme de tri de type EREW qui trie p données avec $\mathcal{O}(p)$ processeurs en temps $\mathcal{O}(\log(p))$, ce que nous admettrons. Une fois le tableau A trié, chaque processeur P_i de l'algorithme EREW inspecte les deux cases adjacentes $A[i] = (l_j, x_j)$ et $A[i-1] = (l_k, x_k)$, où $0 \leq j, k \leq p-1$. Si $l_j \neq l_k$, ou si $i = 0$, le processeur P_i écrit la valeur x_j à l'adresse l_j, sinon il ne fait rien. Comme le tableau est trié selon la première coordonné, l'écriture est bien exclusive. □

On a donc admis le résultat suivant [Cole, 1988], qui relève d'un joli exercice non trivial de programmation parallèle.

Proposition 10.1 *On peut trier p données avec $\mathcal{O}(p)$ processeurs en temps $\mathcal{O}(\log(p))$ avec une machine* PRAM *de type* EREW.

10.2.4 Théorème de Brent

Le théorème de Brent permet de discuter l'effet d'une réduction du nombre de processeurs, et en particulier la transformation d'un algorithme parallèle en algorithme séquentiel.

Théorème 10.2 *Soit A un algorithme comportant un nombre total de m opérations et qui s'exécute en temps t sur une* PRAM *(avec un nombre de processeurs indéterminé). Alors on peut simuler A en temps $\mathcal{O}(\frac{m}{p} + t)$ sur une* PRAM *de même type avec p processeurs.*

Démonstration: A l'étape i, l'algorithme A effectue $m(i)$ opérations. Par définition, $m = \sum_{i=1}^{t} m(i)$. On simule l'étape i avec p processeurs en temps $\lceil \frac{m(i)}{p} \rceil \leq \frac{m(i)}{p} + 1$. On obtient le résultat en sommant sur les étapes i. □

Le théorème de Brent permet de prédire les performances quand on réduit le nombre de processeurs. Prenons par exemple le calcul du maximum sur une PRAM EREW. On peut agencer ce calcul en temps $\mathcal{O}(\log(n))$ à l'aide d'un arbre binaire : à l'étape 1, on procède paire par paire avec $\lceil n/2 \rceil$ processeurs, puis on continue avec les maximum des paires deux à deux, etc. C'est à la première étape que l'on a besoin du plus grand nombre de processeurs, il en faut donc $\mathcal{O}(n)$. Formellement, si $n = 2^m$, si le tableau A est de taille $2n$, et si l'on veut calculer le maximum des n éléments de A en position $A[n], A[n+1], \cdots, A[2n-1]$, on obtient le résultat dans $A[1]$ après exécution de l'algorithme

```
for k ←0 to m − 1 do
  for 2^{m−1−k} ≤ j ≤ 2^{m−k} in parallel
      A[j] ←max(A[2j], A[2j + 1])
```

Que se passe t'il si l'on dispose de moins de $\mathcal{O}(n)$ processeurs ? Le théorème de Brent nous dit qu'avec p processeurs on peut simuler l'algorithme précédent en temps $\mathcal{O}(\frac{n}{p} + \log n)$, puisque le nombre d'opérations total est $m = n-1$. Si l'on

choisit $p = \frac{n}{\log n}$, on obtient à une constante près le même temps d'exécution, mais avec moins de processeurs !

10.2.5 Travail et efficacité

Donnons quelques définitions usuelles : soit L un problème de taille n à résoudre, et $T_{seq}(n)$ le temps du meilleur algorithme séquentiel connu pour résoudre L. Soit maintenant un algorithme parallèle PRAM qui résout L en temps $T_{par}(p)$ avec p processeurs.

Définition 10.5
- *Le* facteur d'accélération *(*speedup *en anglais) est défini comme* $S_p = \frac{T_{seq}(n)}{T_{par}(p)}$,
- *l'efficacité comme* $e_p = \frac{T_{seq}(n)}{p * T_{par}(p)}$.
- *Enfin, le* travail de l'algorithme *est* $W_p = p * T_{par}(p)$.

Intuitivement le travail est une surface rectangulaire de taille $T_{par}(p)$, le temps d'exécution, multiplié par le nombre de processeurs, et est maximal si à chaque étape de calcul tous les processeurs sont utilisés à faire des choses utiles, c'est-à-dire des opérations présentes dans la version séquentielle.

Le résultat suivant montre que l'on peut conserver le travail par simulation :

Proposition 10.2 *Soit A un algorithme qui s'exécute en temps t sur une* PRAM *avec p processeurs. Alors on peut simuler A sur une* PRAM *de même type avec $p' \leq p$ processeurs en temps $\mathcal{O}(\frac{t*p}{p'})$.*

Démonstration: Avec p' processeurs, on simule chaque étape de A en temps proportionnel à $\frac{p}{p'}$. On obtient un temps total de $\mathcal{O}(\frac{p}{p'} * t) = \mathcal{O}(\frac{t*p}{p'})$. □

On observera en particulier que l'algorithme de tri de la proposition 10.1 est optimal, car on ne peut pas trier en séquentiel en moins que $\mathcal{O}(n \log(n))$ dans un modèle avec comparaisons.

10.2.6 Algorithmes efficaces et optimaux

Une question de base en complexité parallèle est de comprendre les problèmes intrinsèquement parallèles, c'est-à-dire qui peuvent être résolus beaucoup plus rapidement avec plusieurs processeurs plutôt qu'un seul.

On s'intéresse beaucoup en particulier aux algorithmes qui fonctionnent en temps parallèle *poly-logarithmique*, c'est-à-dire en temps $\mathcal{O}(\log^k(n))$, où k est un entier. Puisqu'on s'attend à ce qu'un algorithme séquentiel lise au moins son entrée (et donc soit au moins en $\mathcal{O}(n)$) un tel algorithme fonctionne donc plus vite que tout algorithme séquentiel.

Définition 10.6 *Un algorithme parallèle est dit* efficace *s'il fonctionne en temps parallèle poly-logarithmique et si son travail est le temps du meilleur algorithme séquentiel connu multiplié par un facteur poly-logarithmique.*

Un algorithme parallèle efficace permet donc d'obtenir un temps de résolution impossible à atteindre en séquentiel, mais avec un travail plus important mais raisonnable par rapport au meilleur algorithme séquentiel.

Définition 10.7 *Un algorithme parallèle est dit* optimal *s'il est efficace et que son travail est du même ordre que le travail du meilleur algorithme séquentiel connu.*

10.2.7 Rendre optimal un algorithme efficace

Il est parfois possible de rendre optimal un algorithme efficace.

Nous allons l'illustrer sur un exemple. Le produit itéré se calcule séquentiellement en $T_{seq}(n) = \mathcal{O}(n)$ si chacune des opérations produit $\oplus$ se calcule en temps constant. Un premier algorithme parallèle est naturel : on regroupe les entrées deux par deux, et on effectue les opérations selon un arbre binaire équilibré. Le coût de cet algorithme est un temps parallèle $\mathcal{O}(\log(n))$ avec $\mathcal{O}(n)$ processeurs sous une PRAM EREW. Cet algorithme est efficace, mais n'est pas optimal.

Il est cependant possible de le transformer pour obtenir un algorithme optimal, en faisant un bon compromis entre l'algorithme efficace et le meilleur algorithme séquentiel.

Lorsqu'un algorithme n'est pas optimal, il doit y avoir des instructions vides effectuées par les processeurs à certains moments de l'exécution. L'idée est d'occuper à tout moment les processeurs au mieux : à certaines phases de l'exécution, l'algorithme parallèle est remplacé par l'exécution en parallèle pour chaque processeur de l'algorithme séquentiel. Le temps de l'algorithme parallèle reste du même ordre, mais le nombre de processeurs utilisés est plus faible.

Dans l'exemple du produit itéré, on remarque que le travail parallèle est d'un facteur $\mathcal{O}(\log(n))$ fois supérieur au travail séquentiel. Nous allons essayer de garder ce temps en réduisant le nombre de processeurs : l'obtention d'un algorithme optimal nécessite donc d'utiliser que $\mathcal{O}(\frac{n}{\log(n)})$ processeurs : cette diminution du nombre de processeurs peut-être utilisée en groupant différentes opérations sur un même processeur, sur lequel sera exécuté de manière optimal l'algorithme séquentiel.

On groupe donc les n entrées en $\mathcal{O}(\frac{n}{\log(n)})$ groupes ayant chacun $\mathcal{O}(\log(n))$ éléments. A chaque groupe, on associe un processeur qui calcule le produit itéré de son groupe par l'algorithme séquentiel : il le fait donc en temps $\mathcal{O}(\log(n))$ et on utilise $\mathcal{O}(\frac{n}{\log(n)})$ processeurs.

Puis le produit itéré des $\mathcal{O}(\frac{n}{\log(n)})$ produits partiels peut être calculé par l'algorithme parallèle initial en temps $\mathcal{O}(\log(n))$ avec $\mathcal{O}(\frac{n}{\log(n)})$ processeurs. On a obtenu un algorithme au final qui fonctionne en temps parallèle $\mathcal{O}(\log(n))$, avec $\mathcal{O}(\frac{n}{\log(n)})$ processeurs. Il est donc optimal.

10.2.8 Classification des algorithmes

En théorie de la complexité, on s'intéresse surtout au temps parallèle.

La classe NC, formalisée par Nicholas Pippenger dans [Pippenger, 1979], puis nommée par Cook NC pour "Nick's class" dans [Cook, 1985], est la classe des problèmes qui peuvent être résolus en temps parallèle poly-logarithmique (c'est-à-dire en temps $\mathcal{O}(\log^k(n))$, où k est un entier) avec un nombre polynomial de processeurs.

Nous la formaliserons plus tard, et nous la verrons indépendante des modèles. On verra que c'est une sous-classe de P.

Définition 10.8 *On notera* $\mathrm{EREW}(f(n), g(n))$ *la classe des problèmes solvables avec un nombre de processeurs en* $\mathcal{O}(f(n))$ *et temps parallèle* $\mathcal{O}(g(n))$ *avec une* EREW-PRAM.

On note $\mathrm{CREW}(f(n), g(n))$ *et* $\mathrm{CRCW}(f(n), g(n))$ *les classes similaires pour les autres variantes de modèles* PRAM. *Pour* CRCW *on fixe*[2] *la convention* prioritaire *: en cas de conflit d'écriture, la valeur retenue est celle du processeur d'indice le plus petit.*

On distingue en particulier généralement les sous-classes :

Définition 10.9
- EREW^i *est la classe des problèmes qui peuvent être résolus en temps parallèle* $\mathcal{O}(\log^i(n))$ *avec un nombre polynomial de processeurs avec une* EREW-PRAM *:*

$$\mathrm{EREW}^i = \bigcup_{k\in\mathbb{N}} \mathrm{EREW}(n^k, \log^i(n)).$$

- *De même :*

$$\mathrm{CREW}^i = \bigcup_{k\in\mathbb{N}} \mathrm{CREW}(n^k, \log^i(n)).$$

$$\mathrm{CRCW}^i = \bigcup_{k\in\mathbb{N}} \mathrm{CRCW}(n^k, \log^i(n)).$$

Il découle du théorème 10.1, et des discussions précédentes que l'on :

Proposition 10.3 *Pour tout entier* k*,*

$$\mathrm{EREW}^k \subset \mathrm{CREW}^k \subset \mathrm{CRCW}^k \subset \mathrm{EREW}^{k+1}.$$

10.3 Modèles basés sur les circuits

10.3.1 Famille de circuits uniformes

Nous avons introduit dans le chapitre 7, la notion de circuit booléen. Nous avons par ailleurs introduit dans le chapitre 8 la notion de famille de circuits et de langage accepté par une famille de circuits.

On a pour cela utilisé (implicitement) une notion d'uniformité basée sur le temps polynomial. Nous allons la renforcer en une notion d'uniformité basée sur l'espace logarithmique.

[2] Même si cela ne change rien pour ce qui suit.

Définition 10.10 (L-uniformité) *Soit $(C_n)_{n\in\mathbb{N}}$ une famille de circuits booléens, où le circuit C_n possède exactement n entrées et 1 sortie. La famille est dite* L*-uniforme si la fonction qui à $\mathbf{1}^n$ associe la description $< C_n >$ du circuit C_n est calculable en espace logarithmique en n.*

Le théorème 8.3 peut en fait se renforcer en le résultat suivant (fait que nous avons déjà utilisé au chapitre précédent pour la P-complétude).

Théorème 10.3 *Un problème est dans* P *si et seulement si*

1. *il est dans* $\mathbb{P}$
2. *la famille de circuits booléens correspondante est* L*-uniforme.*

Démonstration: Le sens direct est simplement la remarque que dans la preuve du théorème 8.3, la famille de circuits obtenue n'est pas seulement calculable en temps polynomial, mais aussi en espace logarithmique en n. Le sens indirect résulte de la version précédente, puisqu'un espace logarithmique implique temps polynomial. □

On introduit alors les classes suivantes.

Définition 10.11 (Notation $\mathrm{CIRCUIT(f(n), p(n))}$**)** *On dira qu'un langage ou un problème L est dans* $\mathrm{CIRCUIT(f(n), p(n))}$ *si L est reconnu par une famille de circuits* L*-uniforme avec une profondeur en $\mathcal{O}(p(n))$ et une taille en $\mathcal{O}(f(n))$.*

Le théorème précédent peut se reformuler ainsi en

Corollaire 10.1

$$\mathrm{P} = \bigcup_{k\in\mathbb{N}} \mathrm{CIRCUIT}(\mathrm{n^k}, \mathrm{n^k}).$$

Autrement dit, le temps séquentiel est finement relié à la taille des circuits.

10.3.2 Classes NC^i et AC^i

On s'attend à ce que le temps parallèle des circuits soit relié à leur profondeur.

On introduit pour cela les classes suivantes :

Définition 10.12 (Classe NC^i**)** *Soit $i \geq 1$. NC^i désigne la classe des problèmes qui sont reconnu par une famille de circuits de taille polynomiale et de profondeur $\mathcal{O}(\log^i(n))$. Autrement dit,*

$$\mathrm{NC}^i = \bigcup_{k\in\mathbb{N}} \mathrm{CIRCUIT}(\mathrm{n^k}, \log^{\mathrm{i}}(\mathrm{n})).$$

Définition 10.13 (Classe NC**)** *On note*

$$\mathrm{NC} = \bigcup_{i\in\mathbb{N}} \mathrm{NC}^i.$$

La possibilité d'effectuer par des machines CRCW des *ou* logiques et *et* logiques en temps constant invite à considérer la variante suivante : AC^i désigne la classe similaire à NC^i mais où l'on autorise les portes portes $\vee$ et $\wedge$ à être appliquées à plus que deux bits.

Définition 10.14 (Notation $\mathrm{UCIRCUIT(f(n), p(n))}$**)** *On dira qu'un langage ou un problème L est dans* $\mathrm{UCIRCUIT(f(n), p(n))}$ *si L est reconnu par une famille de circuits* L*-uniforme avec une profondeur en $\mathcal{O}(p(n))$ et une taille en $\mathcal{O}(f(n))$, où les portes $\vee$ et $\wedge$ dans les circuits peuvent être de fanin*[3] ≥ 2.

Autrement dit :

Définition 10.15 (Classe AC^i**)** *Soit $i \geq 1$. AC^i désigne la classe*

$$\mathrm{AC}^i = \bigcup_{k \in \mathbb{N}} \mathrm{UCIRCUIT}(\mathrm{n^k}, \log^{\mathrm{i}}(\mathrm{n})).$$

On a

Proposition 10.4 *Pour tout i,* $\mathrm{NC}^i \subset \mathrm{AC}^i$ *et* $\mathrm{AC}^i \subset \mathrm{NC}^{i+1}$.

Démonstration: La première relation est par définition. Maintenant, toute porte $\vee$ (respectivement $\wedge$) d'un circuit AC^i de taille $f(n)$ possède au plus un fanin de $f(n)$. On peut remplacer chaque telle porte par un arbre binaire de portes $\vee$ (respectivement $\wedge$). En faisant cela systématiquement, on transforme un circuit de $\mathrm{CIRCUIT(f(n), p(n))}$ en $\mathrm{CIRCUIT}(\mathcal{O}(\mathrm{f(n)}), \mathcal{O}(\mathrm{p(n)} * \log(\mathrm{f(n)})))$. La transformation se fait bien espace logarithmique. □

10.3.3 Relations avec les PRAM

Théorème 10.4 *On peut simuler un circuit de fanin non-borné par des machines* PRAM CRCW *:* $\mathrm{UCIRCUIT(f(n), p(n))} \subset \mathrm{CRCW}(f(n)^2, p(n))$.

Démonstration: Initialement l'entrée est codée dans les registres partagés 1 à n et un registre $n+i$ est affecté à la porte numéro i du circuit. Ces derniers registres sont initialisés à 0 pour une porte $\vee$ et à 1 pour une porte $\wedge$. On associe d'autre part à chaque arc du circuit un processeur. A chaque instant, chaque processeur détermine la valeur courante de son arête (u, v), en lisant le registre $n+u$, et si v est une porte $\vee$ (respectivement $\wedge$) écrit sa valeur dans l'emplacement $n+v$ si celle-ci vaut 1 (resp. 0). Si cette suite d'opération prend k opérations élémentaires sur une RAM, Au temps $k * p(n)$ chaque registre $n+i$ aura la valeur de la porte qu'il représente. □

Réciproquement, on peut montrer :

[3]Autrement dit, on relâche simplement la contrainte sur le fanin des portes étiquetées par $\vee$ ou $\wedge$ dans la définition des circuits 7.1.

Théorème 10.5 *On peut simuler une machine* PRAM CRCW *par un un circuit de fanin non-borné : supposons $f(n)$ et $p(n)$ de complexité propre alors,* $\mathrm{CRCW}(f(n), p(n)) \subset \mathrm{UCIRCUIT}(\mathrm{poly(f(n)), p(n)})$*, où poly désigne un polynôme.*

Démonstration: On montre que chaque type d'opération élémentaire autorisé dans les instructions d'une machine PRAM (c'est-à-dire essentiellement additionner 1 et réaliser la soustraction $\oplus$) se réalise par un circuit qui travaille sur les représentations binaires de fanin non-borné de profondeur 2, de taille polynomiale : voir la discussion qui va suivre.

On simule alors l'évolution globale du système par un circuit comme dans la preuve de la proposition 7.6. □

Remarque 10.7 *Il en suit que si l'on veut garder ce théorème vrai, si l'on souhaite étendre les types d'opérations autorisées dans les instructions des machines* PRAM *et RAM, il ne faut pas seulement les supposer admissible au sens de la définition 10.2, mais aussi calculable par un circuit qui travaille sur les représentations binaires de fanin non-borné de profondeur constante, et de taille polynomiale.*

Par une construction astucieuse, basée sur l'idée de l'algorithme d'Huffman pour produire des codes préfixe de longueur minimale (la compression de Huffman), on peut prouver [Hoover et al., 1984].

Théorème 10.6 *Tout circuit de* $\mathrm{CIRCUIT(f(n), p(n))}$ *est équivalent à un circuit de* $\mathrm{CIRCUIT(f(n), p(n))}$ *de fanout* 2*.*

On en déduit assez facilement :

Théorème 10.7 *Pour tout entier k,* $\mathrm{NC}^k \subset \mathrm{EREW}^k$.

10.3.4 Résumé

On obtient donc :

Théorème 10.8 *Pour tout entier k,*

$$\mathrm{NC}^k \subset \mathrm{EREW}^k \subset \mathrm{CREW}^k \subset \mathrm{CRCW}^k = \mathrm{AC}^k \subset \mathrm{NC}^{k+1}.$$

Et

Corollaire 10.2

$$\mathrm{NC} = \bigcup_{i \in \mathbb{N}} \mathrm{NC}^i = \bigcup_{i \in \mathbb{N}} \mathrm{AC}^i.$$

10.4 Quelques problèmes parallélisables

10.4.1 Addition

Proposition 10.5 *Étant donnés a et b en binaire sur n bits, on peut produire le codage binaire de leur somme sur $n+1$ bits par un circuit* AC^0*, et donc* NC^1*.*

Démonstration: Supposons que a s'écrit $a = a_{n-1}a_{n-2}\dots a_0$, et que b s'écrit $b = b_{n-1}b_{n-2}\dots b_0$. Soit c_j la retenue de la $j$ème position. Soit $g_j = a_j \wedge b_j$ (appelé *bit de génération de retenue*) et $p_j = a_j \vee b_j$ bit de *propagation de retenue*). En posant $c_{-1} = 0$, on a les récurrences $c_j = g_j \vee p_j c_{j-1}$ et $z_j = a_j \oplus b_j \oplus c_{j-1}$, duquel on tire $c_j = \bigvee_{i\leq j} g_i p_{i+1}\cdots p_j$. On calcule ainsi dans une première étape tous les g_j et p_j puis tous les produits $g_i p_{i+1}\cdots p_j$, puis c_j comme leur union de tous ces produits, et enfin les $z_j = a_j \oplus b_j \oplus c_{j-1}$. Cela donne un circuit AC^0. □

10.4.2 Multiplication

Proposition 10.6 *Étant donnés a et b en binaire sur n bits, on peut produire le codage binaire de leur produit sur $2n$ bits par un circuit* NC^1*.*

Démonstration: L'algorithme classique pour réaliser une multiplication en binaire ramène le problème à celui d'additionner n nombres de taille $2n$. Ce dernier problème peut se résoudre par un circuit de fanin borné par l'astuce "3 pour 2", qui consiste à ramener l'addition de trois nombres sur n-bits à l'addition de deux nombres sur $n+1$-bits : Soient $a = a_{n-1}a_{n-2}\dots a_0$, $b = b_{n-1}b_{n-2}\dots b_0$ et $c = c_{n-1}c_{n-2}\dots c_0$ les trois nombres à additionner. $a_i+b_i+c_i$ s'écrit en binaire sur deux bits $u_i v_i$. Les u_i et v_i s'obtiennent par un circuit de taille linéaire et de profondeur constante. Alors $a+b+c = u+v$, où $u = u_{n-1}u_{n-2}\cdots u_0 0$ et $v = v_{n-1}v_{n_2}\cdots v_0$. L'addition de n nombres, chacun de longueur $2n$ peut alors se résoudre par $\mathcal{O}(\log(n))$ itérations chacune utilisant cette astuce "3 pour 2" pour réduire le nombre de nombres d'un facteur de $2/3$, suivi d'une étape final qui somme deux nombres de $\mathcal{O}(n)$ bits. □

On sait prouver que la multiplication n'est pas dans AC^0.

10.4.3 Multiplication de matrices booléennes

Proposition 10.7 *Soient $A = (a_{i,j})$ et $B = (b_{i,j})$ des matrices $n \times n$ à coefficients dans $\{0,1\}$. On peut calculer leur produit (booléen) C avec un circuit dans* AC^0 *et donc dans* NC^1*.*

Démonstration: C s'écrit $C = (c_{i,j})$ avec $c_{i,j} = \bigvee_{k=1}^{n} a_{i,k}b_{k,j}$. Il suffit de construire un circuit qui à son premier niveau calcule les produits $a_{i,k}b_{k,j}$ puis à son second niveau fait un $\vee$ de tous ces résultats. □

10.4.4 Clôture transitive

Soit $A = (a_{i,j})$ une matrice $n \times n$ à coefficients dans $\{0, 1\}$. A peut se voir comme la matrice d'adjacence d'un graphe. On note A^0 pour la matrice identité I, et A^{k+1} comme le produit $A^k A$ pour $k \geq 0$: l'entrée i, j de la matrice A^k vaut 1 si et seulement s'il existe un chemin de longueur k entre i et j dans le graphe codé par A.

Proposition 10.8 *La clôture transitive de A, définie par $A^* = \bigvee_{k=0}^{+\infty} A^k$ se calcule par un circuit* AC^1*, et donc* NC^2*.*

Démonstration: On vérifie facilement que $A^* = (I \cup A)^{2^{\lceil \log(n) \rceil}}$. On peut donc calculer A^* en partant de $B = A \cup I$, et en élevant au carré B $\lceil \log(n) \rceil$ fois. Puisque la multiplication de matrices est dans AC^0, la clôture transitive est dans AC^1. □

Corollaire 10.3 *On peut donc déterminer s'il existe un chemin dans un graphe donné par sa matrice d'adjacence par un circuit* AC^1*, et donc* NC^2*.*

10.4.5 Inversion matricielle et déterminant

L'algorithme de Csanky donne un moyen de calculer l'inverse d'une matrice carrée, et de calculer son déterminant.

Théorème 10.9 *L'inversion matricielle, et le calcul du déterminant d'une matrice carrée peut se calculer par un circuit* NC^2*.*

En fait, on peut montrer que les problèmes
- résolution d'un système linéaire
- inversion matricielle
- calcul d'un déterminant
- calcul d'une puissance A^n

sont solvables par un circuit NC^2, et réductibles entre eux à l'aide d'une réduction de la classe NC^1. Ces problèmes sont dans la classe appelée DET, c'est-à-dire la classe des problèmes réductibles au calcul du déterminant.

10.5 Quelques résultats

10.5.1 Relations avec les autres classes

Théorème 10.10 *On a*

$$\mathrm{NC}^1 \subset \mathrm{LOGSPACE}.$$

Démonstration: Sur une entrée w, on peut produire le circuit $C_{|w|}$ correspondant à cette entrée en espace logarithmique, et utiliser ensuite une recherche en profondeur récursive partant de la sortie du circuit pour déterminer $C_{|w|}(w)$.

Cela nécessite seulement de mémoriser le chemin entre la sortie et le sommet que l'on est en train d'explorer, et de mémoriser les résultats intermédiaires que l'on a déjà pu obtenir le long de ce chemin. Puisque le circuit est de profondeur logarithmique, un espace logarithmique est suffisant. □

Plus généralement, selon exactement le même principe :

Théorème 10.11 *Soit $i \geq 1$. On a*

$$\mathrm{NC}^i \subset \mathrm{SPACE}(\log^{\mathrm{i}}(\mathrm{n})).$$

D'autre part, on a :

Théorème 10.12 *On a*

$$\mathrm{NLOGSPACE} \subset \mathrm{AC}^1.$$

Démonstration: Soit L un langage de NLOGSPACE. En utilisant les notations du chapitre précédent, déterminer si un mot w est accepté revient à déterminer s'il existe un chemin dans un graphe G_w entre $X[w]$ et X^*. On a vu que cela se ramenait au calcul de la clôture transitive de la matrice d'adjacence du graphe G_w, et que cela se réalisait par un circuit AC^1. □

Théorème 10.13 *On a*

$$\mathrm{NC} \subset \mathrm{P}.$$

Démonstration: Soit L un problème de NC. L est reconnu par une famille de circuits $(C_n)_{n\in\mathbb{N}}$. Sur une entrée w, on peut produire le circuit $C_{|w|}$, et ensuite simuler $C_{|w|}(w)$, ce qui se fera en temps polynomial puisque le circuit est de taille polynomiale. □

En résumé

Corollaire 10.4

$$\mathrm{NC}^1 \subset \mathrm{LOGSPACE} \subset \mathrm{NLOGSPACE} \subset \mathrm{AC}^1.$$

10.5.2 Résultats de séparation

On peut prouver le résultat suivant :

Théorème 10.14 *Le problème* PARITY*, où l'on se donne $a_1, \ldots, a_n, p \in \{0,1\}$ et l'on veut déterminer si $p = \sum_i a_i$ modulo 2, est dans* NC^1 *mais n'est pas dans* AC^0.

10.6 Problèmes intrinsèquement non-parallélisables

10.6.1 Réductions et parallélisme

On peut prouver le résultat suivant : on peut s'y attendre, mais cependant, il faut prouver formellement que la réduction en espace logarithmique préserve la complexité parallèle.

Théorème 10.15 *Si $L \leq_L L'$, et que le problème L' est dans* NC *alors* $L \in$ NC

Démonstration: Soit f la réduction en espace logarithmique de L vers L'. On peut facilement se convaincre qu'il existe un algorithme A' qui décide le langage constitué des couples $< w, < i >>$, où i est un entier inférieur à $|f(w)|$, tel que le $i$ème bit de $f(w)$ vaut 1. Maintenant, en résolvant le problème REACH sur le graphe G_w correspondant à cet algorithme A', on peut calculer le $i$ème bit de $f(w)$. Si on résout tous ces problèmes par des circuits NC^2 en parallèle, on peut donc calculer tous les bits de $f(w)$. Une fois que l'on a $f(w)$, on peut utiliser le circuit pour L' pour décider si $w \in L$. □

En fait, on déduit de cette preuve :

Corollaire 10.5 *Si $L \leq_L L'$, et que le problème L' est dans* NC^i*, pour $i \geq 2$, alors* $L \in \mathrm{NC}^i$.

10.6.2 Problèmes non-parallélisables

La question NC = P ? est une question ouverte (on sait cependant que NC ≠ PSPACE, puisque NC ⊂ LOGSPACE).

Si NC ≠ P (comme cela est généralement conjecturé), les problèmes de P les plus difficiles à paralléliser doivent contenir les problèmes P-complets : on dit qu'un problème est *non-parallélisable* s'il est P-difficile.

Nous avons vu que les problèmes CIRCUITVALUE, SATVALUE ainsi que MONOTONECIRCUITVALUE sont dans ce cas.

En fait, on pourrait formaliser cela encore plus en introduisant une notion de réduction NC^1, et observer que ces problèmes sont bien complet pour cette notion de réduction : notre lecteur devrait en avoir compris l'idée, et éviter de radoter une fois de plus à ce stade.

MONOTONECIRCUITVALUE peut aussi se reformuler ainsi : "Étant donné une séquence de n équations booléennes du type $e_1 = 0$, $e_1 = 1$, et $e_k = e_i \wedge e_j$ ou $e_k = e_i \vee e_k$, pour $1 \leq j < k \leq n$, calculer la valeur de e_n". Formulé ainsi, on voit qu'il s'agit intrinsèquement d'un problème séquentiel, dont la parallélisation n'est pas facile (précisément est la question NC = P).

Voici un autre exemple de problème P-complet.

Théorème 10.16 *Étant donné un graphe orienté, deux sommets distingués s et t, et une fonction c qui associe à chaque arc un entier positif ou nul, le problème de déterminer si le flot maximum de s vers t est un entier impair est* P*-complet.*

10.7 Thèse du parallélisme

10.7.1 Circuits non-polynomiaux

Avec la notion d'uniformité considérée jusque là, on impose que la fonction qui à $\mathbf{1}^n$ associe la description du circuit C_n soit calculable en espace logarithmique en n, ce qui implique en temps polynomial en n. On est donc incapable de produire un circuit avec un nombre exponentiel de portes, puisqu'en temps polynomial on ne peut écrire qu'un nombre polynomial de symboles.

En fait, on peut remplacer la notion de circuit L-uniforme par la notion de circuit BC-uniforme : la différence est qu'on parle d'un espace logarithmique en la taille du circuit C_n au lieu d'un espace logarithmique en n.

Définition 10.16 (BC**-uniformité)** *Soit $(C_n)_{n\in\mathbb{N}}$ une famille de circuits booléens , où le circuit C_n possède exactement n entrées et 1 sortie. La famille est dite* BC*-uniforme si la fonction qui à $\mathbf{1}^n$ associe la description $< C_n >$ du circuit C_n est calculable en espace logarithmique en la taille du circuit C_n.*

Trivialement (puisqu'un logarithme d'un polynôme en n est un logarithme en n) :

Proposition 10.9 *Pour les circuits de taille polynomiale, et donc pour tout ce qui précède, les notions de* L*-uniformité, et* BC*-uniformité coïncident.*

On aurait donc pu directement utiliser cette notion. Nous ne l'avons pas fait, car la plupart des ouvrages considèrent la première, à l'exception par exemple de [Balcàzar et al., 1990] qui consacre une bonne part à la thèse du parallélisme.

10.7.2 Généralisations

Cependant, avec cette notion, on peut généraliser les constructions, sans se restreindre à des circuits de taille polynomiale.

En particulier, le théorème 10.12 se généralise en :

Proposition 10.10 *Soit $f(n) \geq \log(n)$ de complexité propre. Tout langage de* SPACE(f(n)) *est reconnu par une famille de circuits* BC*-uniforme de taille $\mathcal{O}(c^{f(n)})$, pour un entier c, et de profondeur $\mathcal{O}(f(n)^k)$ pour un entier k.*

Démonstration: On construit une famille de circuits qui sur chaque entrée w calcule la fermeture transitive du graphe G_w, comme dans la preuve du théorème 10.12. La famille obtenue est BC-uniforme. □

Les théorèmes 10.10 et 10.11 se généralisent en :

Proposition 10.11 *Pour $f(n) \geq \log(n)$, tout langage reconnu par une famille de circuits* BC*-uniforme de taille $\mathcal{O}(c^{f(n)})$, pour un entier c, et de profondeur $\mathcal{O}(f(n))$ est dans* SPACE($\mathcal{O}$(f(n))).

Démonstration: Sur une entrée w, on peut produire le circuit $C_{|w|}$ correspondant à cette entrée, et utiliser ensuite une recherche en profondeur récursive partant de la sortie du circuit pour déterminer $C_{|w|}(w)$. Cela nécessite seulement de mémoriser le chemin entre la sortie et le sommet que l'on est en train d'explorer, et de mémoriser les résultats intermédiaires que l'on a déjà pu obtenir le long de ce chemin. Cela se fait donc en espace $\mathcal{O}(f(n))$. □

10.7.3 Thèse du parallélisme

On obtient au final une conséquence assez surprenante, connue sous le nom de la thèse du parallélisme.

Théorème 10.17 (Thèse du parallélisme) *Le temps parallèle polynomial (sans contrainte sur le nombre de processeurs, c'est-à-dire possiblement un nombre exponentiel de processeurs) correspond à l'espace polynomial.*

En d'autres termes :

$$\begin{aligned} \text{PSPACE} &= \text{CIRCUIT}(2^{\text{n}^{\mathcal{O}(1)}}, \text{n}^{\mathcal{O}(1)}) \\ &= \text{UCIRCUIT}(2^{\text{n}^{\mathcal{O}(1)}}, \text{n}^{\mathcal{O}(1)}) \\ &= \text{EREW}(2^{n^{\mathcal{O}(1)}}, n^{\mathcal{O}(1)}) \\ &= \text{CREW}(2^{n^{\mathcal{O}(1)}}, n^{\mathcal{O}(1)}) \\ &= \text{CRCW}(2^{n^{\mathcal{O}(1)}}, n^{\mathcal{O}(1)}) \end{aligned}$$

où la notion d'uniformité utilisée dans chaque membre droit est la BC*-uniformité.*

Observons en particulier qu'il est facile de se convaincre que le problème QSAT (PSPACE-complet) se résout bien en temps parallèle polynomial avec un nombre polynomial de processeurs.

10.7.4 Discussions

Les constructions du chapitre précédent relatives à l'espace étaient essentiellement basées sur des liens entre programmation récursive et espace des algorithmes séquentiels.

On voit donc ici qu'il y a en fait un lien fort aussi entre parallélisme, programmation récursive, et espace des algorithmes séquentiels.

10.8 Notes bibliographiques

La partie sur les machines PRAM est essentiellement reprise des cours d'algorithmique parallèle de [Legrand et al., 2003] et de [Roch, 1995]. La partie sur les circuits est essentiellement reprise du panorama [Karp and Ramachandran, 1991]. Nous nous sommes aussi inspirés à divers endroits de [Papadimitriou, 1994] et [Lassaigne and de Rougemont, 1996]. La thèse du parallélisme est longuement discutée dans [Balcàzar et al., 1990], que nous avons partiellement utilisé.

Chapitre 11

Algorithmes et techniques probabilistes

Dans ce chapitre, nous allons présenter quelques algorithmes probabilistes.

Nous supposerons que notre lecteur a suivi un cours d'introduction aux probabilités, comme le cours MAP311 de l'école polytechnique. Cependant, cela sera l'occasion pour nous de rappeler différents concepts et résultats, en se focalisant sur les principaux résultats utilisés pour l'algorithmique informatique. Nous suivons essentiellement l'excellent ouvrage [Mitzenmacher and Upfal, 2005].

11.1 Probabilités élémentaires

11.1.1 Notions de base

On modélise les probabilités par le concept d'espace de probabilité.

Définition 11.1 (Espace de probabilité) *Un* espace de probabilité *est donné par un triplet* $(\Omega, \mathcal{A}, P)$*, où*

- Ω *est un ensemble, qui modélise l'ensemble des résultats possibles.*
- $\mathcal{A}$ *est une* tribu *: $\mathcal{A}$ est une famille de parties de Ω qui contient l'ensemble vide, qui est close par union dénombrable, et qui est close par passage au complémentaire. Les éléments de $\mathcal{A}$ sont appelés des* événements.
- $\Pr : \mathcal{A} \to [0,1]$ *est une fonction de probabilité : c'est-à-dire une fonction qui vérifie* $\Pr(\Omega) = 1$*, et pour toute suite d'éléments $A_1, A_2, \cdots A_n \in \mathcal{A}$ deux à deux disjoints,*

$$\Pr(\bigcup_{i=1}^{n} A_i) = \sum_{i=1}^{n} \Pr(A_i). \tag{11.1}$$

On utilise en informatique (et on utilisera) essentiellement des probabilités sur des espaces discrets : Ω sera soit fini, soit dénombrable. Dans ce cas, on peut

toujours prendre $\mathcal{A}$ comme l'ensemble des parties de Ω, et la fonction Pr est complètement définie par sa valeur sur les singletons.

Exemple 11.1 *Pour modéliser le tirage d'un dé, on peut prendre*

$$\Omega = \{1, 2, \cdots, 6\}$$

comme espace des résultats possibles, on peut prendre $\mathcal{A} = \mathcal{P}(\Omega)$*, l'ensemble des parties de* Ω *comme espace des événements, et* $\Pr(\{i\}) = \frac{1}{6}$*.* Pr *s'étend à l'ensemble des événements* $U \in \mathcal{A}$ *par* $\Pr(U) = \frac{cardinal(U)}{6}$*.*

On peut alors assez facilement prouver le résultat suivant.

Proposition 11.1 (Union bound) *Pour toute suite finie ou dénombrablement infinie d'événements* $E_1, E_2, \cdots$

$$\Pr(\bigcup_{i \geq 1} E_i) \leq \sum_{i \geq 1} \Pr(E_i).$$

Notons que la différence avec l'équation (11.1) est que nous ne supposons pas les événements disjoints deux à deux.

Définition 11.2 (Indépendance) *Deux événements* E_1 *et* E_2 *sont dit* indépendants *si* $\Pr(E_1 \cap E_2) = \Pr(E_1)\Pr(E_2)$*.*

Plus généralement, les événements $E_1, E_2, \cdots E_k$ *sont dits* mutuellement indépendants *si et seulement si pour tout* $I \subset \{1, 2, \cdots, k\}$*,*

$$\Pr(\bigcap_{i \in I} E_i) = \prod_{i \in I} \Pr(E_i).$$

Lemme 11.1 (Principe de la décision diffèrée) *Choisir selon une loi uniforme* $\mathbf{x} = (x_1, x_2, \cdots, x_n)$ *dans* $\{0,1\}^n$ *est équivalent à choisir chaque* x_i *de façon indépendante et uniforme dans* $\{0, 1\}$*.*

Démonstration: Dans les deux cas, la probabilité de choisir chacun des 2^n vecteurs possibles est 2^{-n}. □

Ce genre de remarques, mène à ce que l'on appelle le *principe de la décision différée*, que nous allons illustrer par un raisonnement pour un problème simple.

11.1.2 Application : Vérification d'identités

Supposons que l'on se donne trois matrices $\mathbf{A}, \mathbf{B}$ et $\mathbf{C}$ de taille $n \times n$. Pour simplifier la discussion, nous supposerons que ces matrices sont à coefficients dans $\mathbb{Z}_2$. Supposons que l'on veuille vérifier si $\mathbf{AB} = \mathbf{C}$.

Une façon de faire consiste à multiplier $\mathbf{A}$ par $\mathbf{B}$, puis à tester l'égalité avec $\mathbf{C}$. L'algorithme de multiplication le plus simple nécessite $\mathcal{O}(n^3)$ opérations. Il existe des algorithmes plus efficaces en $\mathcal{O}(n^{2.37})$ opérations.

Voici un algorithme randomisé plus rapide qui permet d'aller plus vite, mais au prix de possiblement se tromper.

On choisit un vecteur aléatoire $\mathbf{x} \in \{0,1\}^n$. On calcule alors $\mathbf{ABx}$ en calculant $\mathbf{Bx}$ puis $\mathbf{A}(\mathbf{Bx})$. On calcule ensuite $\mathbf{Cx}$. Si $\mathbf{ABx} \neq \mathbf{Cx}$ on répond que $\mathbf{AB} \neq \mathbf{C}$. Sinon, on répond que $\mathbf{AB} = \mathbf{C}$.

On a besoin uniquement de trois multiplications matrice/vecteur, ce qui peut se réaliser en temps $\mathcal{O}(n^2)$ par l'algorithme évident.

La probabilité d'erreur est donnée par :

Lemme 11.2 *Si* $\mathbf{AB} \neq \mathbf{C}$*, et si* $\mathbf{x}$ *est choisi uniformément dans* $\{0,1\}^n$ *alors*

$$\Pr(\mathbf{ABx} = \mathbf{Cx}) \leq \frac{1}{2}.$$

Démonstration: Soit $\mathbf{D} = \mathbf{AB} - \mathbf{C}$. $\mathbf{ABx} \neq \mathbf{Cx}$ implique que $\mathbf{Dx} \neq 0$, et puisque $\mathbf{D}$ n'est pas la matrice nulle, la matrice $\mathbf{D}$ doit avoir au moins un coefficient non nul. Supposons sans perte de généralité que cela soit $d_{1,1}$. Pour que $\mathbf{Dx} = 0$, on doit avoir $\sum_{j=1}^{n} d_{1,j}x_j = 0$, et donc

$$x_1 = -(\sum_{j=2}^{N} d_{1,j}x_j)/d_{1,1}. \tag{11.2}$$

Selon le lemme 11.1 (principe de la décision différée), on peut voir choisir $\mathbf{x} \in \{0,1\}^n$ comme choisir chacune de ses composantes. Considérons la situation juste avant que x_1 soit choisi : à ce moment là le membre droit de (11.2) est fixé, et il y a au plus une possibilité pour x_1 qui rend l'égalité vraie. Puisqu'il y a deux choix pour x_1, cela se produit avec probabilité $\leq \frac{1}{2}$. □

11.1.3 Techniques de réduction de l'erreur

On appelle un tel algorithme un algorithme de *Monte Carlo* : l'algorithme est efficace, mais peut parfois se tromper.

Il s'agit d'un algorithme à erreur unilatérale : Si l'on trouve un $\mathbf{x}$ tel que $\mathbf{ABx} \neq \mathbf{Cx}$, on est certain de la réponse : $\mathbf{AB} \neq \mathbf{C}$. Sinon, il y a une possibilité de se tromper.

La probabilité d'erreur est au pire cas de 1/2. Cependant, on peut la rendre assez facilement aussi petite que l'on veut.

En effet, puisque l'erreur est unilatéral, il est facile de réduire l'erreur en répétant l'algorithme. Supposons que l'on recommence alors k fois le test, tant que l'on n'est pas certain de la réponse, en testant à chaque fois un nouveau $\mathbf{x}$ tiré uniformément dans $\{0,1\}^n$. La probabilité que l'on se trompe à l'issu de k tests successifs est en 2^{-k}. Si l'on prend $k = 100$, cela donne une probabilité d'erreur 2^{-100}, un nombre astronomiquement petit, avec un temps $\mathcal{O}(kn^2) = \mathcal{O}(100n^2) = \mathcal{O}(n^2)$.

11.1.4 Loi conditionnelle

Rappelons la notion de probabilité conditionnelle :

Définition 11.3 *La probabilité de E sachant F est définie par*

$$\Pr(E|F) = \frac{\Pr(E \cap F)}{\Pr(F)}.$$

11.1.5 Quelques inégalités utiles

On utilise très souvent les inégalités suivantes :

Lemme 11.3 *Soit $p \in [0, 1]$.*

$$1 - p \leq e^{-p}.$$

Lemme 11.4 *Soit $p \in [-1, 1]$.*

$$1 + p \leq e^{p}.$$

Lemme 11.5 *Soit $p \in \mathbb{R}$.*

$$\frac{e^p + e^{-p}}{2} \leq e^{p^2/2}.$$

Démonstration: Chacune s'obtient en utilisant le développement en série de Taylor de e^p et de e^{-p} et de $e^{p^2/2}$. □

11.1.6 Application : Coupures minimales

Une coupure d'un graphe est un ensemble d'arêtes telles que les enlever coupe le graphe en deux ou plus composantes connexes. Étant donné un graphe $G = (V, E)$, le problème de la coupure minimale consiste à déterminer une coupure de cardinalité minimale du graphe.

Voici un algorithme randomisé simple : on opère $n - 2$ itérations, où n est le nombre de sommets. A chaque itération, on choisit aléatoirement selon une loi uniforme une arête $\{u, v\}$ du graphe, et on la contracte : c'est-à-dire, on fusionne u et v en un unique sommet, on élimine toutes les arêtes entre u et v et on garde toutes les autres arêtes. Le nouveau graphe obtenu peut avoir des multi-arêtes (plusieurs arêtes entre deux mêmes sommets) mais pas de boucle.

Chaque itération réduit le nombre de sommet de 1. Après $n - 2$ itérations, il ne reste donc plus que deux sommets avec un ensemble d'arêtes C entre ces sommets. On retourne alors cet ensemble d'arête C.

Il est facile de se convaincre que toute coupure d'un graphe obtenu à l'une des itérations est une coupure du graphe initial. Par conséquent, la réponse C est toujours une coupure du graphe initiale, mais elle peut être de taille non minimale.

Proposition 11.2 *L'algorithme produit une coupure minimale avec probabilité au moins* $2/n(n-1)$.

Il s'agit donc encore ici d'un algorithme de Monte Carlo.

Démonstration: Soit k la taille de la coupure minimale. Le graphe peut avoir plusieurs coupures C de taille minimale. On calcule la probabilité d'en trouver une.

Puisque C est une coupure, supprimer C partitionne les sommets en deux ensembles S et $V-S$, tel qu'aucune arête connecte un sommet de S à un sommet de $V-S$. Supposons que pendant l'exécution, on contracte des arêtes de S et de $V-S$ mais jamais de C. Dans ce cas, après $n-2$ itérations, l'algorithme retourne un graphe avec deux sommets connectés par les arêtes de C. On en déduit que si l'algorithme ne choisit jamais une arête de C dans ses $n-2$ itérations, alors il retournera C.

Cela donne l'intuition de l'algorithme : en choisissant les arêtes uniformément, si C est petit, la probabilité de choisir une arête de C est faible.

Soit E_i l'événement que l'arête contractée à l'itération i n'est pas dans C. Et soit $F_i = \bigcap_{j=1}^{i} E_j$ l'événement qu'aucune arête de C n'a été contractée pendant les premières i itérations.

Puisque la coupure minimale est de taille k, tout sommet doit être de degré au moins k : en effet, si on enlève toutes les arêtes incidentes à un sommet, alors ce sommet devient sa propre composante connexe, et donc les arêtes incidentes en chaque sommet constituent une coupure du graphe de cardinalité le degré du sommet.

Si chaque sommet est adjacent à au moins k arêtes, le graphe doit donc avoir au moins $nk/2$ arêtes. La probabilité que l'on ne choisisse pas k arêtes parmi ces $nk/2$ arêtes vérifie donc

$$\Pr(E_1) = \Pr(F_1) \geq 1 - \frac{2k}{nk} = 1 - \frac{2}{n}.$$

Supposons que la première contraction n'a pas éliminé une arête de C : autrement dit, on conditionne sur F_1. Après la première itération, on a un graphe avec $n-1$ sommets, de coupure minimale de taille k. On peut donc utiliser exactement le même raisonnement sur ce nouveau graphe, pour obtenir :

$$\Pr(E_2|F_1) \geq 1 - \frac{2k}{(n-1)k} = 1 - \frac{2}{n-1}.$$

De même

$$\Pr(E_i|F_{i-1}) \geq 1 - \frac{2k}{(n-i+1)k} = 1 - \frac{2}{n-i+1}.$$

On écrit alors

$$\begin{aligned}
\Pr(F_{n-2}) &= \Pr(E_{n-2}|F_{n-3})\Pr(E_{n-3}|F_{n-4}) \cdots \Pr(E_2|F_1)\Pr(F_1). \\
&\geq \prod_{i=1}^{n-2} \left(1 - \tfrac{2}{n-i+1}\right) \\
&= \prod_{i=1}^{n-2} \left(\tfrac{n-i-1}{n-i+1}\right) \\
&= \tfrac{n-2}{n}\tfrac{n-3}{n-1}\tfrac{n-4}{n-2} \cdots \tfrac{4}{6}\tfrac{3}{5}\tfrac{2}{4}\tfrac{1}{3} \\
&= \tfrac{2}{n(n-1)}.
\end{aligned}$$

□

Puisque l'algorithme est à erreur unilatérale, on peut réduire l'erreur facilement comme précédemment : supposons que l'on fasse tourner l'algorithme $n(n-1)\log n$ fois, et que l'on produise en sortie la plus petite coupure trouvée dans toutes les itérations. La probabilité que l'on ne produise pas une coupure minimale est bornée par

$$\left(1 - \frac{2}{n(n-1)}\right)^{n(n-1)\log n} \leq e^{-2\log n} = \frac{1}{n^2},$$

où l'on a utilisé le fait que $1 - x \leq e^{-x}$.

11.2 Variables aléatoires et moyennes

Définition 11.4 (Variable aléatoire) *Une* variable aléatoire *sur un espace de probabilité Ω discret est une fonction $X : \Omega \to \mathbb{R}$.*

Une variable aléatoire discrète *est une variable aléatoire qui prend un nombre fini ou dénombrable de valeurs.*

Définition 11.5 (Moyenne) *La moyenne d'une variable aléatoire discrète X, notée $E[X]$, est définie par*

$$E[X] = \sum_i i \Pr(X = i).$$

On peut assez facilement prouver le résultat suivant :

Théorème 11.1 (Linéarité de la moyenne) *Pour toute famille finie de variables aléatoires $X_1, X_2, \cdots, X_n$ discrète de moyennes finies*

$$E[\sum_{i=1}^{n} X_n] = \sum_{i=1}^{n} E[X_i].$$

Observons que ce résultat ne nécessite aucune hypothèse sur l'indépendance des variables aléatoires.

On va voir deux applications de ce théorème : observons qu'à chaque fois le principe sera de décomposer un problème de moyennes en celui de l'évaluation d'une somme de moyennes d'événements simples.

11.2.1 Application : Problème du collectionneur

Supposons que des boites de céréales contiennent chacune un coupon parmi n coupons possibles. Supposons que l'on veuille posséder au moins un exemplaire de chacun des coupons. Combien faut-il acheter en moyenne de boites de céréales pour cela ?

Ce problème, qui s'appelle *le problème du collectionneur*, apparaît dans de nombreux contextes en informatique.

Soit X le nombre de boîtes achetées avant de posséder tous les n coupons. On s'intéresse donc à $E[X]$. Si X_i désigne le nombre de boîtes achetées en ayant exactement $i-1$ coupons, alors clairement nous devons avoir

$$X = \sum_{i=1}^{n} X_i.$$

Lorsque l'on a exactement $i-1$ coupons, la probabilité d'obtenir un nouveau coupon en achetant une boite est $p_i = 1 - \frac{i-1}{n}$. Par conséquent, X_i est une variable aléatoire géométrique de paramètre p_i et

$$E[X_i] = \frac{1}{p_i} = \frac{n}{n-i+1}.$$

En utilisant la linéarité de la moyenne, on obtient

$$E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{n}{n-i+1} = n \sum_{i=1}^{n} \frac{1}{i}.$$

Le résultat suivant est connu.

Lemme 11.6 *Le nombre $H(n) = \sum_{i=1}^{n} \frac{1}{i}$, connu sous le nom du nème* nombre harmonique*, vérifie $H(n) = \log(n) + \Theta(1)$.*

Par conséquent, la réponse au problème du collectionneur de coupon est $n \log(n) + \Theta(n)$.

11.2.2 Application : Tri Bucket Sort

Le tri *Bucket Sort* est un exemple de méthode de tri qui brise la borne inférieure de $\Omega(n \log(n))$ opérations par comparaisons : supposons que l'on ait $n = 2^m$ éléments à trier, et que chaque élément est un entier choisit uniformément dans $[0, 2^k[$, avec $k \geq m$. Avec ce tri, on peut trier en temps moyen $\mathcal{O}(n)$.

Le tri fonctionne ainsi : dans une première étape, on place les éléments dans n emplacements : l'emplacement j contient tous les éléments dont les m premiers bits correspondent au nombre j. En supposant que placer un élément dans un emplacement se fait en temps $\mathcal{O}(1)$, cette étape nécessite un temps $\mathcal{O}(n)$. Avec ces hypothèses, le nombre d'éléments qui tombe dans chaque emplacement suit une loi binomiale $B(n, 1/n)$. Les emplacements peuvent s'implémenter par des listes par exemple. Dans une deuxième étape, l'algorithme trie chaque emplacement, avec un algorithme de complexité quadratique. L'algorithme produit alors en sortie le résultat de la concaténation des listes triées.

Soit X_j le nombre d'éléments qui tombe dans l'emplacement j. Le temps pour trier chaque emplacement est de la forme $c(X_j)^2$ pour une constante c. Le temps total pour la deuxième phase est donné par

$$E[\sum_{j=1}^{n} c(X_j)^2] = c \sum_{j=1}^{n} E[X_j^2] = cnE[X_1^2],$$

où l'on a utilisé la linéarité de la moyenne et le fait que chaque emplacement joue un rôle symétrique. Puisque chaque X_i est une variable binomiale,

$$E[X_1^2] = 1 + \frac{n(n-1)}{n^2} = 2 - \frac{1}{n} < 2.$$

11.2.3 Application : Temps moyen du tri rapide

L'algorithme *Quicksort* est un algorithme de tri récursif, qui consiste, étant donné une liste $S = \{x_1, \cdots, x_n\}$ d'éléments distincts,

- à retourner S si S ne possède qu'un ou zéro élément ;
- à choisir sinon un élément x de S, appelé *pivot* ;
- à comparer chaque élément à x, pour diviser S en deux sous-listes : S_1, ceux qui sont inférieurs à x, et S_2 sont qui sont plus grands.
- à utiliser récursivement Quicksort sur chacune des listes S_1 et S_2 pour les trier ;
- et à retourner le résultat S_1, x, S_2 dans cet ordre.

Il est facile de voir que dans le pire des cas (par exemple si la liste est dans l'ordre décroissant et si l'on prend comme pivot systématiquement le premier élément), l'algorithme nécessite $\mathcal{O}(n^2)$ comparaisons.

Proposition 11.3 *Supposons que dans Quicksort on choisisse le pivot systématiquement selon une loi uniforme et des tirages indépendants parmi les possibilités. Alors pour toute entrée, l'algorithme effectue un nombre moyen de comparaisons donné par* $2n \log n + \mathcal{O}(n)$.

Démonstration: Soient $y_1, y_2, \cdots, y_n$ les mêmes valeurs que les valeurs en entrée $x_1, x_2, \cdots, x_n$ mais dans l'ordre trié. Pour $i < j$, soit X_{ij} la variable aléatoire qui prend la valeur 1 si y_i et y_j sont comparés par l'algorithme, et 0 sinon. Le nombre total de comparaisons est donné par $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$. On a donc par linéarité de la moyenne

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}].$$

Puisque X_{ij} prend les valeurs 0 ou 1, $E[X_{ij}]$ est la probabilité que y_i soit comparé à y_j.

Cela se produit si et seulement si y_i ou y_j est le premier pivot choisit parmi l'ensemble $Y^{ij} = \{y_i, y_{i+1}, \cdots, y_j\}$. En effet, si y_i (ou y_j) est le premier pivot choisit dans cette liste, alors y_i et y_j seront dans la même liste et donc seront comparés. Symétriquement, si aucun des deux n'est le premier pivot choisit dans cette liste, alors y_i et y_j seront séparés dans deux listes distinctes et donc ne seront jamais comparés.

Puisque les pivots sont choisis de façon uniforme et indépendante, la probabilité que cela se produise est $2/(j - i + 1)$. En posant $k = j - i + 1$, on

obtient

$$\begin{aligned} E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\ &= \sum_{k=2}^{n} \sum_{i=1}^{n+1-k} \frac{2}{k} \\ &= \sum_{k=2}^{n} (n+1-k)\frac{2}{k} \\ &= (2n+2)\sum_{k=1}^{n} \frac{1}{k} - 4n \\ &= 2n\log(n) + \Theta(n). \end{aligned}$$

□

Le même type de preuve montre que :

Proposition 11.4 *Supposons que dans Quicksort on choisisse systématiquement le premier élément. Alors si les entrées sont choisies de façon uniforme parmi les permutations de* $\{1, 2, \cdots, n\}$*, l'algorithme effectue un nombre moyen de comparaisons de l'ordre de* $2n\log n + \mathcal{O}(n)$*.*

11.3 Moments et déviations

On va maintenant introduire un ensemble d'inégalités, qui visent toutes à mesurer de combien une variable peut s'écarter de sa moyenne.

Nous irons des inégalités les plus grossières aux plus fines. Ces inégalités sont omniprésentes dans l'étude des algorithmes.

11.3.1 Inégalité de Markov

Théorème 11.2 (Inégalité de Markov) *Soit* X *une variable aléatoire à valeurs positive ou nulles.*

Alors pour tout $a > 0$*,*

$$\Pr(X \geq a) \leq \frac{E[X]}{a}.$$

Démonstration: Pour $a > 0$, posons

$$I = \begin{cases} 1 & \text{si } X \geq a, \\ 0 & \text{sinon.} \end{cases}$$

Puisque $X \geq 0$,

$$I \leq \frac{X}{a}. \tag{11.3}$$

Puisque I est une variable à valeur dans $\{0, 1\}$, $E[I] = \Pr(I = 1) = \Pr(X \geq a)$. En passant à la moyenne dans (11.3), on obtient

$$\Pr(X \geq a) = E[I] \leq E\left[\frac{X}{a}\right] = \frac{E[X]}{a}.$$

□

Observons que l'on a égalité par exemple pour une loi telle que $\Pr(X = a) = 1$.

Remarque 11.1 *Une façon qui peut être plus intuitive de comprendre l'inégalité est d'écrire*

$$\Pr(X \geq \mu a) \leq \frac{1}{a},$$

pour tout $a > 0$, *où* $\mu = E[X]$.

Exemple 11.2 *Considérons une suite de tirage de pièces.*
Posons

$$X_i = \begin{cases} 1 & \text{si la ième pièce est pile} \\ 0 & \text{sinon.} \end{cases}$$

Notons par $X = \sum_{i=1}^{n} X_i$ *le nombre de piles parmi les* n *tirages.*
On a $E[X] = \sum_{i=1}^{n} E[X_i] = \frac{n}{2}$. *L'inégalité de Markov donne, pour* $\lambda > 0$,

$$\Pr(X \geq \lambda n) \leq \frac{E[X]}{\lambda n} = \frac{n}{2\lambda n} = \frac{1}{2\lambda}.$$

Ou encore

$$\Pr(X \geq \lambda \frac{n}{2}) \leq \frac{1}{\lambda}.$$

11.3.2 Inégalité de Tchebychev

Rappelons la définition suivante :

Définition 11.6 (Variance) *La* variance *d'une variable aléatoire* X *est définie par*

$$\mathrm{Var}[X] = E[(X - E[X])^2] = E[X^2] - E[X]^2.$$

Lorsque l'on possède une information sur la variance, on peut utiliser l'inégalité suivante, qui est plus fine.

Théorème 11.3 (Inégalité de Tchebyshev) *Pour tout* $a > 0$,

$$\Pr(|X - E[X]| \geq a) \leq \frac{\mathrm{Var}[X]}{a^2}.$$

Démonstration: Observons tout d'abord que

$$\Pr(|X - E[X]| \geq a) = \Pr((X - E[X])^2 \geq a^2).$$

Puisque $(X - E[X])^2$ est une variable aléatoire à valeurs positives ou nulles, on peut lui appliquer l'inégalité de Markov pour obtenir,

$$\Pr((X - E[X])^2 \geq a^2) \leq \frac{E[(X - E[X])^2]}{a^2} = \frac{\mathrm{Var}[X]}{a^2}.$$

□

Remarque 11.2 *On a égalité par exemple pour une loi telle que* $\Pr(X \in \{\mu - a, \mu + a\}) = 1$, *puisque* $E[X] = \mu$, $\Pr(|X - E[X]| \geq a) = 1$, $\mathrm{Var}[X] = 1$.

Remarque 11.3 *Une façon qui peut être plus intuitive de voir l'inégalité est d'écrire*

$$\Pr(|X - \mu| \geq a\sigma) \leq \frac{1}{a^2},$$

pour tout $a > 0$, où $\mu = E[X]$, $\sigma^2 = \mathrm{Var}[X]$.

Exemple 11.3 *Reprenons l'exemple des tirages de pièces. Puisque X_i est une variable aléatoire à valeur $0-1$, $E[X_i^2] = \Pr(X_i = 1) = \frac{1}{2}$. On a*

$$\mathrm{Var}[X_i] = E[X_i^2] - E[X_i]^2 = \frac{1}{2} - \frac{1}{4} = \frac{1}{4}.$$

Puisque les X_i sont indépendants,

$$\mathrm{Var}[X] = \sum_{i=1}^{n} \mathrm{Var}[X_i] = \frac{n}{4}.$$

L'inégalité de Tchebychev donne pour $\lambda > 0$

$$\begin{array}{rcl} \Pr(|X - \frac{n}{2}| \geq \lambda n) & \leq & \frac{\mathrm{Var}[X]}{\lambda^2 n^2} \\ & \leq & \frac{n/4}{\lambda^2 n^2} \\ & \leq & \frac{1}{4\lambda^2 n}. \end{array}$$

Elle est donc plus fine que le résultat obtenu avec l'inégalité de Markov.

11.3.3 Application : Problème du collectionneur

Revenons sur le problème du collectionneur. Rappelons que le nombre moyen de coupons est donné par X de moyenne nH_n, où $H_n = \sum_{i=1}^{n} 1/n = \log(n) + \mathcal{O}(1)$.

L'inégalité de Markov donne donc

$$\Pr(X \geq 2nH_n) \leq \frac{1}{2}.$$

Pour utiliser l'inégalité de Tchebychev, il nous faut la variance de X. Rappelons que $X = \sum_{i=1}^{n} X_i$, où chaque X_i est une variable aléatoire géométrique de paramètre $(n - i + 1)/n$. Les variables X_i sont indépendantes puisque le temps nécessaire pour collecter le $i$ème coupon ne dépend pas du temps utilisé pour le $i - 1$. Par conséquent,

$$\mathrm{Var}[X] = \mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{Var}[X_i].$$

La variance d'une variable aléatoire de loi géométrique de paramètre p est $(1-p)/p^2$.

On obtient

$$\mathrm{Var}[X] = \mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] \leq \sum_{i=1}^{n} \left(\frac{n}{n-i+1}\right)^2 = n^2 \sum_{i=1}^{n} \left(\frac{1}{n}\right)^2 \leq \frac{\pi^2 n^2}{6},$$

où l'on a utilisé

$$\sum_{i=1}^{\infty} \left(\frac{1}{i}\right)^2 = \frac{\pi^2}{6}.$$

L'inégalité de Tchebychev donne alors

$$\Pr(|X - nH_n| \geq nH_n) \leq \frac{n^2\pi^2/6}{(nH_n)^2} = \frac{\pi^2}{6(H_n)^2} = \mathcal{O}\left(\frac{1}{\log^2(n)}\right).$$

Une fois encore, l'inégalité de Tchebychev donne un meilleur résultat que l'inégalité de Markov.

11.3.4 Argument d'union-bound

Cependant, on peut faire mieux, par un argument d'union-bound pour ce problème. Cela sera l'occasion de présenter de tels types d'arguments.

La probabilité de ne pas obtenir le $i$ème coupon après $n\ln(n) + cn$ étapes est donné par

$$\left(1 - \frac{1}{n}\right)^{n(\ln n + c)} \leq e^{-\ln(n)+c} = \frac{1}{e^c n}.$$

En notant E_n cet événement, et en utilisant la proposition 11.1 (union-bound), on obtient que la probabilité qu'au moins un des coupons ne soit pas collecté se majore par

$$\Pr(\bigcup_{i\geq 1} E_i) \leq \sum_{i\geq 1} \Pr(E_i) = n\frac{1}{e^c n} = \frac{1}{e^c}.$$

En particulier, en prenant $c = \ln n$, la probabilité que tous les coupons ne soient pas collectés après $2n\ln n$ étapes est au plus $1/n$, ce qui donne une borne plus fine que l'inégalité de Tchebychev.

11.3.5 Application : Calcul de la médiane

Nous présentons maintenant un autre exemple d'algorithme randomisé dont l'étude se fait en utilisant les inégalités sur l'écart entre une variable aléatoire et sa moyenne.

Étant donné un ensemble $S = \{s_1, s_2, \cdots, s_n\}$ de n éléments parmi un univers totalement ordonné, on appelle *médiane* de S un élément m de S tel que au moins $\lfloor n/2 \rfloor$ éléments de S soit inférieurs ou égaux à m, et au moins $\lfloor n/2 \rfloor + 1$ éléments de S soit supérieurs à m. Autrement dit, si S est trié, m est l'élément d'indice $\lceil n/2 \rceil$.

Bien entendu, l'idée est d'utiliser un algorithme plus efficace que $\mathcal{O}(n\log(n))$, ce qui peut être atteint en triant les données, puis en retournant cet élément.

On connaît une solution déterministe en $\mathcal{O}(n)$ opérations. On présente ici une version randomisée simple de même complexité.

L'idée de l'algorithme est d'utiliser de l'échantillonnage : on essaye de trouver deux éléments d et u tels que $d \leq m \leq u$, et tel que le nombre d'éléments entre d et u soit faible, c'est-à-dire en $o(n/\log(n))$.

Notons $C = \{s | d \leq s \leq u\}$ les éléments entre d et u.

Si l'on arrive à trouver deux éléments comme cela, il est facile de trouver la médiane en temps linéaire :

- on parcours la liste S, et
- on compte le nombre ℓ_d d'éléments inférieurs à d,
- et on trie l'ensemble C : puisque $|C| = o(n/\log(n))$, trier C se fait en temps $o(n)$ par n'importe quel algorithme de tri qui fonctionne en temps $\mathcal{O}(m\log(m))$ pour m éléments.
- L'élément d'indice $\lfloor n/2 \rfloor - \ell_d + 1$ de C est alors m.

Pour trouver d et u, on échantillonne avec remplacement un (multi-)ensemble R de $\lceil n^{3/4} \rceil$ éléments : c'est-à-dire que l'on pioche au hasard uniformément ce nombre d'éléments parmi S.

Remarque 11.4 *Pour simplifier l'analyse, on autorise à piocher plusieurs fois le même élément (on pioche avec remise), ce qui fait que R n'est pas un ensemble mais un multi-ensemble : il peut contenir plusieurs fois le même élément.*

On souhaite que chaque étape fonctionne avec *grande probabilité*, c'est-à-dire avec une probabilité au moins $1 - \mathcal{O}(1/n^c)$ pour une constante c.

Pour garantir qu'avec grande probabilité m soit entre d et u, on fixe d et u comme étant respectivement les $\lfloor n^{3/4}/2 - \sqrt{n} \rfloor$ème et $\lfloor n^{3/4}/2 + \sqrt{n} \rfloor$ ème éléments de R. L'analyse qui suit va éclairer ces choix.

L'algorithme final est donc le suivant :

1. Choisir $\lceil n^{3/4} \rceil$ éléments dans S, avec un tirage uniforme et avec remise. Soit R les éléments obtenus.
2. Trier R.
3. Soit d le $\lfloor n^{3/4}/2 - \sqrt{n} \rfloor$ élément de R.
4. Soit u le $\lfloor n^{3/4}/2 + \sqrt{n} \rfloor$ élément de R.
5. En comparant chaque élément de S à d et u, calculer les ensembles $C = \{x \in S | d \leq x \leq u\}$, $\ell_d = |\{x \in S | x < d\}|$ et $\ell_u = |\{x \in S | x > u\}|$.
6. Si $\ell_d > n/2$ ou $\ell_u > n/2$ alors échouer.
7. Si $|C| \leq 4n^{3/4}$ alors trier S sinon échouer.
8. Retourner le $\lfloor n/2 \rfloor - \ell_d + 1$ élément de C.

L'algorithme termine donc soit en retournant une valeur, soit en échouant. Lorsqu'il retourne une valeur, les derniers tests permettent de garantir que l'on se trouve dans le cas de la discussion précédente avec $d \leq m \leq u$, et donc la réponse est correcte.

Il reste à majorer la probabilité d'échouer.

Proposition 11.5 *La probabilité que l'algorithme échoue est en* $\mathcal{O}(n^{-1/4})$.

Démonstration: On considère les trois événements

$$E_1 : Y_1 = |\{r \in R | r \leq m\}| < n^{3/4}/2 - \sqrt{n}$$

$$E_2 : Y_2 = |\{r \in R | r \geq m\}| < n^{3/4}/2 - \sqrt{n}$$

$$E_3 : |C| > 4n^{3/4}.$$

L'algorithme termine si au moins l'un des trois événements E_1, E_2 ou E_3 se produit : en effet, un échec à l'étape 7 correspond à l'événement E_3. Un échec à l'étape 6 implique $\ell_d > n/2$ ou $\ell_u > n/2$. Chacun de ces cas est équivalent à E_1 ou E_2.

On a

$$\Pr(E_1) \leq \frac{1}{4} n^{-1/4}.$$

En effet, considérons la variable aléatoire X_i qui vaut 1 si le $i$ème échantillon est inférieur ou égal à la médiane m, et 0 sinon. Les X_i sont indépendants, puisque l'on procède avec remplacement. Puisqu'il y a $(n-1)/2 + 1$ éléments inférieurs ou égaux à l, on a

$$\Pr(X_i = 1) = \frac{(n-1)/2 + 1}{n} = \frac{1}{2} + \frac{1}{2n}.$$

L'événement E_1 est équivalent à

$$Y_1 = \sum_{i=1}^{n^{3/4}} X_i < \frac{1}{2} n^{3/4} - \sqrt{n}.$$

Y_1 est la somme de tirages de Bernoulli, il suit donc une loi binomiale de paramètres $n^{3/4}$ et $1/2 + 1/2n$. L'inégalité de Tchebychev donne alors

$$\begin{array}{rcl} \Pr(E_1) & = & \Pr(Y_1 < \frac{1}{2} n^{3/4} - \sqrt{n}) \\ & \leq & \Pr(|Y_1 - E[Y_1]| > \sqrt{n}) \\ & \leq & \frac{\mathrm{Var}[Y_1]}{n} \\ & < & \frac{1/4 n^{3/4}}{n} \\ & = & \frac{1}{4} n^{-1/4}. \end{array}$$

Symétriquement pour E_2.

D'autre part,

$$\Pr(E_3) \leq \frac{1}{2} n^{-1/4}.$$

En effet, si E_3 se produit, soit au moins $2n^{3/4}$ éléments sont plus grands que m, soit au moins $2n^{3/4}$ sont plus petits que m.

Bornons le premier, puisque l'autre est symétrique. S'il se produit, l'ordre de u dans l'ensemble S trié est au moins $\frac{1}{2}n + 2n^{3/4}$, et donc R possède au moins $\frac{1}{2}n^{3/4} - \sqrt{n}$ éléments parmi les $\frac{1}{2}n - 2n^{3/4}$ éléments les plus grands de S.

Soit X le nombre d'éléments parmi les $\frac{1}{2}n - 2n^{3/4}$ éléments les plus grands de S. X s'écrit $X = \sum_{i=1}^{n^{3/4}} X_i$, où X_i vaut 1 si le $i$ème élément pioché est parmi les $\frac{1}{2}n - 2n^{3/4}$ éléments les plus grands de S, et 0 sinon. X, somme de lois de Bernoulli, suit une loi binomiale, et l'inégalité de Tchebychev permet de majorer la probabilité de cet événement par

$$\begin{aligned} \Pr(X \geq \tfrac{1}{2}n^{3/4} - \sqrt{n}) &\leq \Pr(|X - E[X]| \geq \sqrt{n}) \\ &\leq \frac{\mathrm{Var}[X]}{n} \\ &< \frac{1/4n^{3/4}}{n} \\ &= \tfrac{1}{4}n^{-1/4}. \end{aligned}$$

En sommant par une union-bound, on obtient le résultat. □

Observons qu'en répétant cet algorithme jusqu'à ce qu'il réussisse, on obtient un algorithme de *Las Vegas* : il retourne toujours une réponse correcte, mais son temps de réponse est aléatoire.

En fait, le nombre d'itérations de l'algorithme serait alors donné par une loi géométrique. On peut assez facilement se convaincre que l'algorithme obtenu fonctionnerait en temps moyen linéaire.

11.3.6 Bornes de Chernoff

Les bornes de Chernoff, sont en fait une famille d'inégalités obtenues sur un même principe.

Rappelons le concept suivant : la fonction génératrice des moments d'une variable aléatoire X est $M_X(t) = E[e^{tX}]$. La propriété élémentaire que l'on va utiliser est la suivante :

Théorème 11.4 *Si X et Y sont des variables indépendantes, alors*

$$M_{X+Y}(t) = M_X(t)M_Y(t).$$

Démonstration:

$$M_{X+Y}(t) = E[e^{t(X+Y)}] = E[e^{tX}e^{tY}] = E[e^{tX}]E[e^{tY}] = M_X(t)M_Y(t).$$

□

Les bornes de Chernoff sont alors obtenues en appliquant l'inégalité de Markov sur e^{tX} pour un t bien choisi. En effet, par l'inégalité de Markov, on obtient : pour tout $t > 0$,

$$\Pr(X \geq a) = \Pr(e^{tX} \geq e^{ta}) \leq \frac{E[e^{tX}]}{e^{ta}}.$$

En particulier,

$$\Pr(X \geq a) \leq \min_{t>0} \frac{E[e^{tX}]}{e^{ta}}.$$

De façon similaire, pour $t < 0$,

$$\Pr(X \leq a) = \Pr(e^{tX} \geq e^{ta}) \leq \frac{E[e^{tX}]}{e^{ta}}.$$

Donc

$$\Pr(X \leq a) \leq \min_{t<0} \frac{E[e^{tX}]}{e^{ta}}.$$

On obtient par exemple (remarquons que l'on doit supposer les variables indépendantes) :

Théorème 11.5 *On considère des variables $X_1, X_2, \cdots, X_n$ à valeurs dans $\{0,1\}$ indépendantes telles que $\Pr(X_i) = p_i$. Soit $X = \sum_{i=1}^{n} X_i$, et $\mu = E[X]$. Alors on a les bornes de Chernoff suivantes.*

- *Pour tout $\delta > 0$*

$$\Pr(X \geq (1+\delta)\mu) \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}.$$

- *Pour tout $0 < \delta \leq 1$*

$$\Pr(X \geq (1+\delta)\mu) \leq e^{-\mu\delta^2/3}.$$

- *Pour $R \geq 6\mu$,*

$$\Pr(X \geq R) \leq 2^{-R}.$$

Démonstration:
Écrivons

$$M_{X_i}(t) = E[e^{tX_i}] = p_i e^t + (1-p_i) = 1 + p_i(e^t - 1) \leq e^{p_i(e^t-1)}.$$

Par conséquent,

$$M_X(t) = \prod_{i=1}^{n} M_{X_i}(t) \leq \prod_{i=1}^{n} e^{p_i(e^t-1)} = exp(\sum_{i=1}^{n} p_i(e^t-1)) = e^{(e^t-1)\mu}.$$

Par l'inégalité de Markov, pour tout $t > 0$, on a

$$\begin{array}{rcl} \Pr(X \geq (1+\delta)\mu) & = & \Pr(e^{tX} \geq e^{t(1+\delta)\mu}) \\ & \leq & \frac{E[e^{tX}]}{e^{t(1+\delta)\mu}} \\ & = & \frac{M_X(t)}{e^{t(1+\delta)\mu}} \\ & \leq & \frac{e^{(e^t-1)\mu}}{e^{t(1+\delta)\mu}}. \end{array}$$

Fixons $t = \ln(1+\delta) > 0$ pour obtenir

$$\Pr(X \geq (1+\delta)\mu) \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu},$$

c'est-à-dire la première inégalité recherchée.

La deuxième s'obtient en montrant que

$$\left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} \leq e^{-\mu\delta^2/3}$$

pour $0 < \delta \leq 1$ par étude de fonction.

La troisième, consiste à poser $R = (1+\delta)\mu$. Pour $R \geq 6\mu$, on a $\delta = R/\mu - 1 \geq 5$. Donc,

$$\left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} \leq \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu} \leq \left(\frac{e}{6}\right)^{R} \leq 2^{-R}.$$

□

On peut aussi obtenir des bornes sur la déviation en dessous de la moyenne, par un principe similaire :

Théorème 11.6 *On considère des variables $X_1, X_2, \cdots, X_n$ à valeurs dans $\{0,1\}$ indépendantes telles que $\Pr(X_i) = p_i$. Soit $X = \sum_{i=1}^{n} X_i$, et $\mu = E[X]$. Alors on a les bornes de Chernoff suivantes.*

- *Pour tout $0 < \delta < 1$*

$$\Pr(X \leq (1-\delta)\mu) \leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu};$$

- *Pour tout $0 < \delta < 1$*

$$\Pr(X \leq (1-\delta)\mu) \leq e^{-\mu\delta^2/2}.$$

Par un "union-bound", on obtient :

Corollaire 11.1 *On considère des variables $X_1, X_2, \cdots, X_n$ à valeurs dans $\{0,1\}$ indépendantes telles que $\Pr(X_i) = p_i$. Soit $X = \sum_{i=1}^{n} X_i$, et $\mu = E[X]$. Pour tout $0 < \delta < 1$,*

$$\Pr(|X - \mu| \geq \delta\mu) \leq 2e^{-\mu\delta^2/3}.$$

Exemple 11.4 *Reprenons l'exemple des tirages de pièces. Pour $\lambda < \frac{1}{2}$*

$$\Pr(|X - \frac{n}{2}| \geq \lambda n) \leq 2\exp(-\frac{1}{3}\frac{n}{2}4\lambda^2) = 2e^{-2/3n\lambda^2},$$

ce qui est infiniment mieux que Tchebychev et Markov.

En fait, on peut aussi voir que X est fortement centré autour de sa moyenne.

$$\Pr(|X - \frac{n}{2}| \geq \lambda\sqrt{n\ln n}) \leq 2\exp(-\frac{1}{3}\frac{n}{2}\frac{4\lambda^2\ln n}{n}) = 2n^{-2/3\lambda^2} = \frac{2}{n^{2/3\lambda^2}}$$

Ce qui donne $2/n$ pour $\lambda = \sqrt{6}/2$.

11.3.7 Application : Test d'hypothèses

Supposons que l'on veuille évaluer une probabilité p de mutation de virus dans une population. Étant donné un virus, on peut déterminer s'il a muté, mais le test est coûteux. Supposons que l'on fasse le test sur une population de taille n, et que l'on observe que parmi cet échantillon que $X = \overline{p}n$ virus ont muté. Si n est suffisamment grand, on peut espérer que $\overline{p}$ est proche de p. Formellement, cette intuition est capturée par la notion d'intervalle de confiance.

Définition 11.7 (Intervalle de confiance) *Un $1-\gamma$-*intervalle de confiance *pour un paramètre p est un intervalle $[\overline{p}-\delta, \overline{p}+\delta]$ tel que*

$$\Pr(p \in [\overline{p}-\delta, \overline{p}+\delta]) \geq 1-\gamma.$$

Théorème 11.7 *On peut déterminer un $1-\gamma$-intervalle de confiance en utilisant un échantillon de taille*

$$n = \lceil 3\frac{1}{\delta^2}\ln\frac{2}{\gamma}\rceil.$$

Démonstration: On considère $X = \sum_i X_i$, avec $\Pr(X_i = 1) = p$, $\Pr(X_i = 0) = 1-p$. Sur n expériences, on aura $E[X] = np$, et donc on va considérer $\overline{p} = X/n$ comme estimation de p. Les bornes de Chernoff disent que

$$\Pr(X/n > p+\delta) = \Pr(X > np + n\delta) = \Pr(X > np(1+\delta/p)) \leq e^{-np\delta^2/(3p^2)}$$

Soit

$$\Pr(X/n > p+\delta) \leq e^{-n\delta^2/(3p)}$$

Symétriquement

$$\Pr(X/n < p-\delta) \leq e^{-n\delta^2/(2p)}$$

On ne connaît pas p, mais on sait qu'il est plus petit que 1.
Donc

$$\Pr(X/n > p+\delta) \leq e^{-n\delta^2/(3)}$$

$$\Pr(X/n < p-\delta) \leq e^{-n\delta^2/(2)}$$

Soit, par un union bound,

$$\Pr(p \notin [X/n-\delta, X/n+\delta]) \leq 2e^{-n\delta^2/3}.$$

Il suffit de prendre

$$n \geq \lceil 3\frac{1}{\delta^2}\ln\frac{2}{\gamma}\rceil,$$

pour que cette quantité soit inférieure à γ.

□

11.3.8 Meilleures bornes pour certains cas

Il est possible d'obtenir de meilleures bornes dans certains cas : on va utiliser le fait que $\frac{e^p+e^{-p}}{2} \leq e^{p^2/2}$, ce qui est plus fin que $1+p \leq e^p$, en traitant le cas de quelque-chose de symétrique.

Théorème 11.8 *Soit $X_1, \cdots, X_n$ des variables aléatoires indépendantes avec*

$$\Pr(X_i = 1) = \Pr(X_i = -1) = \frac{1}{2}.$$

Soit $X = \sum_{i=1}^n X_i$. Pour tout $a > 0$,

$$\Pr(X \geq a) \leq e^{-a^2/2n}.$$

Démonstration: Écrivons

$$M_{X_i}(t) = E[e^{tX_i}] = \frac{1}{2}e^t + \frac{1}{2}e^{-t} \leq e^{t^2/2}.$$

Par conséquent,

$$M_X(t) = \prod_{i=1}^n M_{X_i}(t) \leq \prod_{i=1}^n e^{t^2/2} \leq e^{t^2 n/2}.$$

$$\begin{array}{rcl} \Pr(X \geq a) & = & \Pr(e^{tX} \geq e^{ta}) \\ & \leq & \frac{E[e^{tX}]}{e^{ta}} \\ & = & \frac{M_X(t)}{e^{t(1+\delta)\mu}} \\ & \leq & \frac{e^{t^2 n/2}}{e^{ta}} \\ & = & e^{t^2 n/2 - ta} \end{array}$$

Si l'on considère $t = a/n$, on obtient

$$\Pr(X \geq a) \leq e^{-a^2/2n}.$$

□

Par symétrie, on a aussi

$$\Pr(x \leq -a) \leq e^{-a^2/2n}.$$

En utilisant la transformation $Y_i = (X_i + 1)/2$, on obtient.

Corollaire 11.2 *Soient $Y_1, \cdots, Y_n$ des variables indépendantes avec*

$$\Pr(Y_i = 1) = \Pr(Y_i = 0) = \frac{1}{2}.$$

Soit $Y = \sum_{i=1}^n Y_i$, et $\mu = E[Y] = n/2$.

– *Pour tout $a > 0$,*

$$\Pr(Y \geq \mu + a) \leq e^{-2a^2/n}.$$

– *Pour tout* $\delta > 0$,

$$\Pr(Y \geq (1+\delta)\mu) \leq e^{-\delta^2 \mu}.$$

Remarquons que le gain est la constante 1 plutôt que 1/3 dans la dernière inégalité.

De façon symétrique :

Corollaire 11.3 *Soient* $Y_1, \cdots, Y_n$ *des variables indépendantes avec*

$$\Pr(Y_i = 1) = \Pr(Y_i = 0) = \frac{1}{2}.$$

Soit $Y = \sum_{i=1}^{n} Y_i$, *et* $\mu = E[Y] = n/2$.

– *Pour tout* $a > 0$,

$$\Pr(Y \leq \mu - a) \leq e^{-2a^2/n}.$$

– *Pour tout* $0 < \delta < 1$,

$$\Pr(Y \leq (1-\delta)\mu) \leq e^{-\delta^2 \mu}.$$

11.3.9 Application : Équilibrage d'ensembles

On considère une matrice $\mathbf{A}$ de taille $n \times m$ avec des coefficients dans $\{0, 1\}$. On cherche un vecteur $\mathbf{b}$ de taille m à coefficients dans $\{-1, 1\}$ qui minimise $||\mathbf{Ab}||_\infty$.

L'algorithme est extrêmement simple : on choisit les coefficients du vecteur b au hasard, indépendemment avec $\Pr(b_i = 1) = \Pr(b_i = -1) = 1/2$. Il s'avère que $||\mathbf{Ab}||_\infty$ a de forte chances d'être de l'ordre de $\mathcal{O}(\sqrt{m \ln n})$, indépendemment des coefficients de $\mathbf{A}$.

En effet, considérons la $i$ème ligne $\mathbf{a}_i = a_{i1} a_{i2} \cdots a_{im}$ de $\mathbf{A}$, et soit k le nombre de 1 dans cette ligne. Si $k \leq \sqrt{4m \ln n}$, alors clairement $|\mathbf{a}_i.\mathbf{b}| \leq \sqrt{4m \ln n}$. D'un autre coté, si $k > \sqrt{4m \ln n}$, on observe que les k termes non nuls dans la somme $Z_i = \sum_{j=1}^{m} a_{i,j} b_j$ sont des variables indépendantes ayant chacune la probabilité 1/2 d'être -1 ou $+1$. En utilisant les bornes de Chernoff, et le fait que $m \geq k$,

$$\Pr(|Z_i| > \sqrt{4m \ln n}) \leq 2e^{-4m \ln n / 2k} \leq \frac{2}{n^2}.$$

Par une union-bound, la probabilité que cela ne soit pas vrai pour au moins une ligne est donc au plus $2/n$.

11.4 Quelques autres algorithmes et techniques

11.4.1 Tester la nullité d'un polynôme

On va présenter un test probabiliste pour lequel aucun algorithme déterministe aussi efficace n'est connu : le but de ce test est de déterminer si un

polynôme $p(x_1, x_2, \cdots, x_n)$ de degré faible à coefficients entiers est identiquement nul. On suppose que p est donné sous la forme d'un circuit arithmétique sur la structure $(\mathbb{R}, +, -, *)$. On peut bien entendu vérifier cette propriété en développant le polynôme p et en vérifiant que chacun des termes du développement du polynôme est bien nul, mais cela peut prendre un temps exponentiel dans le pire cas.

De façon alternative, on peut évaluer p en des points $a_1, a_2, \cdots, a_n$ choisis aléatoirement. Si p est identiquement 0, alors $p(a_1, a_2, \cdots, a_n) = 0$. Sinon, on s'attend à ce que $p(a_1, a_2, \cdots, a_n) \neq 0$ avec grande probabilité. Cela marche en fait aussi si l'on travaille sur un corps fini, si le corps est suffisamment grand.

Proposition 11.6 (Lemme de Schwartz-Zippel) *Soit $\mathbb{F}$ un corps, et $S \subset \mathbb{F}$ un sous-ensemble arbitraire. Soit $p(\mathbf{x})$ un polynôme non nul des n variables $\mathbf{x} = (x_1, x_2, \cdots, x_n)$ de degré total d à coefficients dans $\mathbb{F}$. Alors l'équation $p(\mathbf{x}) = 0$ possède au plus $d|S|^{n-1}$ solutions dans S^n.*

Corollaire 11.4 *Soit $\mathbb{F}$ un corps. Soit $p(x_1, \cdots, x_n)$ un polynôme non nul de degré total d à coefficients dans $\mathbb{F}$. Si p est évalué sur un élément choisi uniformément parmi un point de S^n, alors*

$$\Pr(p(s_1, \cdots, s_n) = 0) \leq \frac{d}{|S|}.$$

Ce résultat possède de nombreuses applications, toutes basées sur des techniques d'empreinte (*finger printing* en anglais) : en effet, on se ramène à un problème de polynôme, et à son évaluation sur un corps fini. Observons que le passage par les corps finis sert à éviter le problème de l'explosion des coefficients qui peut se produire si l'on travaille directement sur les entiers.

11.4.2 Application : Couplage parfait

Nous allons l'illustrer sur le problème du *couplage parfait* : un couplage parfait dans un graphe biparti est un sous-ensemble d'arêtes M tel que

1. les arêtes de M ne partagent pas de sommets ;
2. chaque sommet est l'extrémité d'une arête de M.

On connaît un algorithme polynomial pour déterminer si un graphe biparti possède un couplage parfait. On ne sait pas si le problème est dans NC. Voici un algorithme qui donne un algorithme NC randomisé.

On affecte à chaque arête (i, j) de G une variable $x_{i,j}$ et on considère la matrice d'adjacence X de taille $n \times n$ avec ces indéterminées plutôt que 1.

Le déterminant $\det(X)$ est un polynôme de degré n en les variables $x_{i,j}$, avec un terme (non nul) pour chaque couplage parfait : cela est en fait assez facile à voir en revenant à la définition du déterminant.

Par conséquent, G possède un couplage parfait si et seulement si le polynôme $\det(X)$ ne s'annule pas. Cela est difficile à vérifier de façon déterministe, car $\det(X)$ peut être de grande taille. On peut le vérifier de façon probabiliste en

choisissant des valeurs pour les $x_{i,j}$ dans un corps $\mathbb{F}$ de taille suffisamment grande (par exemple dans $\mathbb{F}_p$, avec p un nombre premier plus grand que $2n$). On calcule alors le déterminant, et on teste s'il est nul. Si le graphe ne possède pas de couplage parfait, cela se produira à coup sûr. Si le graphe possède un couplage parfait, alors cela se produira avec probabilité au plus $\frac{n}{2n} = \frac{1}{2}$.

Une fois que l'on sait détecter l'existence d'un couplage parfait, on peut en fait en construire un, en supprimant les arêtes et les sommets un à un en testant à chaque étape s'il existe un couplage parfait sans cette arête.

11.4.3 Chaînes de Markov et marches aléatoires

Soit $G = (V, E)$ un graphe non-orienté. On le suppose fini, et connexe, et non biparti.

Une marche aléatoire sur G consiste à partir d'un sommet u_0 au temps $t = 0$, et à chaque temps discret t, étant en u_t à choisir un de ses voisins u_{t+1} de façon uniforme : si $d(u_t)$ désigne le degré de u_t, on choisit l'un des $d(u_t)$ voisins uniformément.

Nous admettrons les résultats suivants, qui relèvent de la théorie des chaînes de Markov.

Théorème 11.9 *Une marche aléatoire sur G converge vers une distribution stationnaire π telle que le probabilité d'être en le sommet v est donnée par*

$$\pi(v) = \frac{d(v)}{2|E|}.$$

Théorème 11.10 *Le temps moyen nécessaire pour visiter tous les sommets d'un graphe est borné supérieurement par $4 \times |V| \times |E|$.*

Cela donne l'algorithme suivant pour déterminer s'il y a un chemin entre u et v dans un graphe G non-orienté : on part de u, et on suit une marche aléatoire. Si l'on atteint v en moins de $8n^3$ étapes alors on répond qu'il y a un chemin entre u et v. Sinon, on répond qu'il n'y a pas de chemin.

Par l'inégalité de Markov, la probabilité d'erreur est au plus 1/2.

11.5 Notes bibliographiques

Nous avons essentiellement utilisé l'excellent ouvrage [Mitzenmacher and Upfal, 2005].

Nous nous sommes parfois inspiré de [Papadimitriou, 1994], [Sipser, 1997], [Motwani and Raghavan, 1995], [Lassaigne and de Rougemont, 1996], [Kozen, 2006], [Balcázar et al., 1988], et [Arora and Barak, 2009].

Chapitre 12

Classes probabilistes

Dans le chapitre précédent, nous avons présenté quelques algorithmes et quelques techniques probabilistes. Dans ce chapitre, nous discutons de la puissance obtenue par l'utilisation de choix probabilistes, en introduisant les principales classes de complexité probabilistes, et en cherchant à les positionner par rapport aux autres classes de complexité.

12.1 Notion d'algorithme probabiliste

On peut voir un algorithme probabiliste comme un algorithme classique, si ce n'est qu'à certaines étapes, il est autorisé à lancer à tirer une pièce (non-biaisée) à pile ou face, et à prendre une décision basée sur le résultat.

Formellement, on peut considérer qu'un algorithme probabiliste A qui travaille sur l'entrée w prend en fait deux entrées : d'une part, w, l'entrée, et, d'autre part, un suite (un mot) $y \in \{\mathbf{0}, \mathbf{1}\}^*$ qui donne le résultat des tirages aléatoires successifs utilisés. On note $A(w, y)$ le résultat de l'algorithme sur l'entrée w avec les tirages aléatoires y.

Bien entendu :

1. On suppose que $y \in \{\mathbf{0}, \mathbf{1}\}^*$ est de longueur suffisante, c'est-à-dire de longueur au moins égale au nombre de tirages probabilistes effectués sur l'entrée w, pour donner le résultat de chacun des tirages : la lettre numéro i de y donne le résultat du tirage aléatoire numéro i.
2. On suppose d'autre part, que le mot y est en lecture uniquement : on suppose que l'on ne contrôle pas les résultats des tirages aléatoires.

On ne s'intéressera dans la suite essentiellement qu'à des problèmes de décision : $A(w, y)$ sera donc toujours soit le résultat "accepter" soit "rejeter".

On dit bien entendu que l'algorithme fonctionne en temps $f : \mathbb{N} \to \mathbb{N}$, si pour toute entrée w de longueur n, et pour toute suite de tirages aléatoires $y \in \{\mathbf{0}, \mathbf{1}\}^*$, l'algorithme termine en temps au plus $f(n)$ étapes : autrement dit, comme on s'y attend on ne fait pas intervenir la longueur de y dans la mesure de l'entrée.

Avec ces hypothèses, cela rend le calcul des probabilités assez facile : la probabilité qu'une entrée w soit acceptée est donnée par

$$\Pr(A \text{ accepte } w) = \frac{|\{y \in \{\mathbf{0},\mathbf{1}\}^k | A(w,y) \text{ accepte}\}|}{2^k},$$

où k est un entier suffisamment grand (par exemple plus grand que le temps de calcul de l'algorithme A sur l'entrée w de façon à ce que y couvre bien tous les choix probabilistes).

12.2 Classe PP

On peut alors introduire les principales classes de complexité probabilistes.

La classe suivante est peut être celle qui vient le plus rapidement à l'esprit, même si elle ne se comporte pas très bien en pratique.

Définition 12.1 *Un langage L est dans* PP *s'il existe un algorithme probabiliste A qui fonctionne en temps polynomial tel que pour tout mot w,*

1. *si $w \in L$,* $\Pr(A \text{ accepte } w) > 1/2$ *;*
2. *si $w \notin L$,* $\Pr(A \text{ accepte } w) \leq 1/2$.

Théorème 12.1 *On a*

$$\text{NP} \subset \text{PP} \subset \text{PSPACE}.$$

Démonstration: Un mot w est dans un langage L de NP si et seulement s'il existe un certificat $u \in M^*$ tel que $< w, u > \in A$, pour A un langage reconnaissable en temps polynomial par un certain algorithme A'

Considérons un algorithme probabiliste B qui est obtenu à partir de cet algorithme A', et qui se comporte de la façon suivante : sur l'entrée w, l'algorithme tire à pile ou face. S'il tire pile, alors il accepte. Sinon, il simule l'algorithme A' sur w, en remplaçant chaque choix non-déterministe par un tirage à pile ou face. On a $\Pr(B \text{ accepte } w) = 1/2$ si $w \notin L$, puisque le seul moyen d'accepter pour B dans ce cas, est d'avoir tiré pile à la première étape. On a $\Pr(B \text{ accepte } w) > 1/2$ pour $w \in L$, puisque avec probabilité $1/2$ on accepte à la première étape, et pour au moins une suite de tirage aléatoire (correspondant à un certificat) on accepte à une étape ultérieure.

Prouvons maintenant que PP $\subset$ PSPACE : Soit A l'algorithme probabiliste qui accepte $L \in$ PP. Il suffit de considérer l'algorithme B qui simule A sur tous les tirages possibles y des choix aléatoires, et compte le nombre de tirages y tels que $A(w,y)$ accepte, et qui accepte si ce nombre est supérieur à $1/2$. □

Proposition 12.1 *La classe* PP *est close par passage au complémentaire.*

Démonstration: Il suffit de montrer que tout langage L de PP est reconnu par un algorithme probabiliste tel que la probabilité d'acception n'est jamais exactement $1/2$. En effet, il suffit alors d'inverser l'état d'acceptation et de rejet pour obtenir le résultat.

Soit A un algorithme probabiliste qui fonctionne en temps $p(n)$ pour un polynôme $p(n)$, avec $p(n) > 1$. On construit un algorithme probabiliste qui sur une entrée w simule d'abord A pendant $p(n)$ étapes, et fait ensuite $p(n)$ tirages à pile ou face supplémentaires. Il accepte si A a accepté, et si l'un de ces tirages a produit pile. Sinon, il rejette. Cette opération multiplie la probabilité d'acceptation de A par $2^{p(n)} - 1$: si A accepte, alors les au moins $2^{p(n)-1} + 1$ calculs acceptants sont ainsi multipliés, ce qui en fait bien plus que la moitié, en utilisant le fait que $p(n) > 1$. Si A rejette, on a bien toujours moins de la moitié de calculs acceptants, puisque $2^{p(n)-1}(2^{p(n)} - 1) < 2^{2p(n)-1}$. □

Corollaire 12.1

$$\mathrm{NP} \cup \mathrm{coNP} \subset \mathrm{PP}.$$

12.3 Classe BPP

La classe suivante se comporte beaucoup mieux en pratique, et capture la notion d'algorithme de *Monte Carlo*.

Définition 12.2 *Un langage L est dans* BPP *s'il existe un $0 < \epsilon < 1/2$, et un algorithme probabiliste qui fonctionne en temps polynomial tels que pour toute entrée w,*

1. *si $w \in L$,* $\Pr(A \text{ accepte } w) \geq \frac{1}{2} + \epsilon$ *;*
2. *si $w \notin L$,* $\Pr(A \text{ rejette } w) \geq \frac{1}{2} + \epsilon$.

Autrement dit, une autre façon de formuler cette définition est de dire qu'il existe une constante $0 < \epsilon'$ telle que pour toute entrée w,

$$\Pr(A \text{ se trompe sur } w) \leq \epsilon' :$$

on dit qu'un algorithme se trompe s'il accepte un mot qu'il ne devrait pas accepté, ou rejette un mot qu'il devrait accepter.

Les techniques de réduction d'erreurs du chapitre précédent, permettent d'obtenir le résultat suivant :

Proposition 12.2 *On peut fixer l'erreur à $\epsilon' = 1/4$ (ou toute autre constante entre 0 et 1/2 strictement) sans changer la classe : un langage L est dans* BPP *s'il existe un algorithme probabiliste qui fonctionne en temps polynomial tel que pour toute entrée w,*

$$\Pr(A \text{ se trompe sur } w) \leq 1/4.$$

Démonstration: Il nous faut montrer qu'avec une erreur ϵ', on peut rendre l'erreur inférieure à ϵ'' pour tout $0 < \epsilon'' < 1$.

Soit L dans BPP reconnu par l'algorithme A. Soit k un entier. On considère l'algorithme B qui sur une entrée w de longueur n simule successivement et de façon indépendante $2k$ fois l'algorithme A. B accepte l'entrée w si majorité des réponses de chacune des simulations accepte, et refuse sinon.

La probabilité que B se trompe est la probabilité qu'au moins k simulations parmi les $2k$ simulations de A sur l'entrée w donne une réponse erronée. La probabilité de cet évènement est donnée par

$$\sum_{i=k}^{2k} \Pr[A \text{ se trompe exactement } i \text{ fois sur les } 2k \text{ simulations}]$$

$$= \sum_{i=k}^{2k} C_{2k}^{i} \delta^i (1-\delta)^{2k-i},$$

où δ est la probabilité d'erreur de A sur w, que l'on sait telle que $\delta \leq \epsilon' < 1/2$.

Puisque $\delta/(1-\delta) < 1$ lorsque $\delta < 1/2$, aucun terme de cette somme ne diminue si l'on fixe $i = k$. On peut donc borner cette probabilité par

$$(k+1)C_{2k}^{k}\delta^k(1-\delta)^k \leq (k+1)2^{2k}\delta^k(1-\delta)^k \leq (k+1)(4\delta(1-\delta))^k.$$

On peut majorer cela par $(k+1)(4\epsilon'(1-\epsilon'))^k$. Il suffit de choisir k assez grand pour que cette expression soit plus petite que l'erreur désirée. □

La même preuve montre qu'on peut en fait aussi écrire :

Proposition 12.3 *Dans la définition précédente de* BPP*, on peut aussi imposer que l'erreur soit au plus* $(\frac{1}{2})^{p(|w|)}$ *pour n'importe quel polynôme* p *sans changer la classe.*

Remarque 12.1 *On aurait pu obtenir ces résultats directement par des bornes de Chernoff. Nous avons privilégié ici une preuve directe et indépendante.*

En inversant les états d'acceptation et de rejet, on a clairement :

Proposition 12.4 *La classe* BPP *est close par passage au complémentaire.*

12.4 Classes RP et ZPP

Pour formaliser la notion d'algorithme de *Las Vegas*, on est amené à introduire les classes suivantes :

Définition 12.3 *Un langage* L *est dans* RP *s'il existe un algorithme probabiliste* A *qui fonctionne en temps polynômial telle que pour toute entrée* w

1. *si* $w \in L$, $\Pr(A \text{ se trompe sur } w) < 1/2$;
2. *si* $w \notin L$, $\Pr(A \text{ se trompe sur } w) = 0$.

Définition 12.4 *Un langage* L *est dans* coRP *s'il existe un algorithme probabiliste qui fonctionne en temps polynômial telle que pour toute entrée* w

1. *si* $w \in L$, $\Pr(A \text{ se trompe sur } w) = 0$;
2. *si* $w \notin L$, $\Pr(A \text{ se trompe sur } w) < 1/2$.

Comme auparavant, la constante $1/2$ dans les deux définitions précédentes est arbitraire (la preuve est plus simple).

Théorème 12.2 *Dans les définitions de* RP *et* coRP *on peut remplacer* $1/2$ *par n'importe quelle constante* $0 < \epsilon \leq 1/2$ *sans changer la classe.*

Démonstration: Soit L dans RP reconnu par l'algorithme A. Soit k un entier. Considérons l'algorithme B qui sur une entrée w simule k fois A et accepte si et seulement si au moins l'une des k simulations de A accepte. L'algorithme B vérifie $\Pr(B \text{ rejette } w) \leq 1/2^k$ pour $w \in L$, et $\Pr(B \text{ accepte w}) = 0$ pour $w \notin L$. Il suffit de choisir k tel que $1/2^k < \epsilon$. La preuve pour coRP est symétrique. □

La même preuve montre qu'on peut en fait aussi écrire :

Proposition 12.5 *Dans la définition précédente, on peut imposer que l'erreur soit au plus* $(\frac{1}{2})^{p(|w|)}$ *pour n'importe quel polynôme* p *sans changer la classe : par exemple, pour* RP*, pour un polynôme* p*,*

1. *si* $w \in L$*,* $\Pr(A \text{ se trompe sur } w) < (1/2)^{p(|w|)}$;
2. *si* $w \notin L$*,* $\Pr(A \text{ se trompe sur } w) = 0$.

Définition 12.5 *On définit*

$$\text{ZPP} = \text{RP} \cap \text{coRP}.$$

Théorème 12.3 ZPP *correspond aux problèmes qui sont reconnus par un algorithme qui termine toujours avec une réponse correcte et qui termine en temps moyen polynomial en la taille de l'entrée.*

Démonstration: Sens $\Rightarrow$: soit $L \in$ ZPP. Soient A et A' les algorithmes qui attestent que L est respectivement dans RP et dans coRP. Sur une entrée w, avec une probabilité $> 1/2$, A et A' donnent la même réponse. Lorsque cela ce produit, cette réponse est nécessairement correcte. Considérer l'algorithme A^* qui simule de façon alternée A et A' sur son entrée, et recommence jusqu'à ce que les deux machines soient d'accord. A^* donne toujours une réponse correcte. La moyenne du temps de réponse est plus faible que $(temps(A) + temps(A')(1 * 1/2 + 2 * 1/4 + 3 * 1/8 + \dots)$ soit $\mathcal{O}(temps(A) + temps(A'))$.

Sens $\Leftarrow$: Soit t le temps d'exécution de l'algorithme probabiliste qui reconnaît le langage L. Soit A l'algorithme qui accepte un entrée w si et seulement si l'algorithme randomisé accepte l'entrée w en un temps $\leq 3*t$, rejette sinon. Pour $w \notin L$, l'algorithme A rejette toujours. Pour $w \in L$, par l'inégalité de Markov, avec une probabilité $\geq 2/3$, l'algorithme A a le temps de simuler complètement l'algorithme et donc ne se trompe pas. L est donc dans RP.
L est dans coRP par l'argument symétrique. □

Par définition, puisque le complémentaire de RP et coRP et réciproquement :

Proposition 12.6 ZPP *est close par passage au complémentaire.*

12.5 Relations entre classes

Théorème 12.4 P $\subset$ RP *et* P $\subset$ coRP*.*

Démonstration: Un algorithme non probabiliste est un algorithme probabiliste particulier. □

Corollaire 12.2 P $\subset$ ZPP*.*

Par définition, on a immédiatement :

Théorème 12.5 RP $\subset$ BPP*,* coRP $\subset$ BPP*,* BPP $\subset$ PP*.*

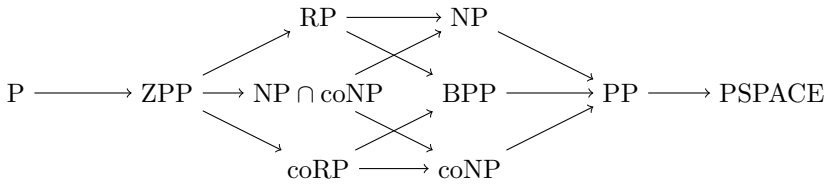
Théorème 12.6 RP $\subset$ NP*,* coRP $\subset$ coNP*.*

Démonstration: Considérer l'algorithme probabiliste comme un algorithme non-déterministe : la suite des tirages aléatoires correspond à un certificat. □

Corollaire 12.3 ZPP $\subset$ NP *et* ZPP $\subset$ coNP*.*

12.6 Résumé

On peut résumer la situation par le dessin suivant, où chaque flèche correspond à une inclusion.



12.7 Quelques résultats

12.7.1 Sur l'utilisation de pièces biaisées

Notre modèle d'algorithme probabiliste suppose l'existence d'une opération qui permet de tirer une pièce non-biaisée à pile ou face.

Que se passerait il s'il on utilisait une pièce biaisée, c'est-à-dire une pièce qui serait pile avec probabilité ρ et face avec probabilité $1-\rho$, avec $\rho \neq 1/2$?

Obtiendrait-on plus de puissance ?

La première étape est de se convaincre que l'on en a au moins autant.

Théorème 12.7 *On peut simuler un algorithme qui utilise une pièce non-biaisée par un algorithme qui utilise une pièce biaisée. Cela introduit un surcoût de temps dont la moyenne est en* $\mathcal{O}(\frac{1}{\rho(1-\rho)})$.

Démonstration: On construit un algorithme qui utilise une pièce biaisée pour simuler chaque tirage d'une pièce non-biaisée : pour simuler un tel tirage, l'algorithme tire une paire de pièce de biais ρ jusqu'à obtenir deux résultats différents sur ces pièces, c'est-à-dire face et pile, ou pile et face. A ce moment là, si la première pièce est pile, l'algorithme répond pile, et si la première pièce est face, l'algorithme répond face.

La probabilité qu'une paire de pièces biaisées fassent apparaître face et pile est $\rho(1-\rho)$, alors que la probabilité de faire apparaître pile et face est $\rho(1-\rho)$. Par conséquent, chaque étape permet de répondre avec la probabilité $2\rho(1-\rho)$. En temps moyen $\mathcal{O}(\frac{1}{\rho(1-\rho)})$, on finira donc pas répondre. Si l'on conditionne sur le fait que l'un de ces deux événements s'est produit à une étape donnée, la probabilité de répondre pile ou face est bien équiprobable. □

Remarque 12.2 *Observons qu'il n'y a aucun besoin de connaître ρ pour appliquer cette procédure.*

Maintenant, avons nous plus de puissance ?

Théorème 12.8 *Supposons que l'écriture en binaire de ρ soit*

$$\rho = 0.p_1 p_2 \cdots ,$$

et telle que l'on puisse produire le $i$ème bit de ρ en temps polynomial en i. Alors, on peut simuler un algorithme avec une pile de biais ρ par un algorithme probabiliste au sens précédent (sans biais) en temps moyen $\mathcal{O}(1)$.

Démonstration: L'algorithme produit par tirages non biaisés une suite de bits aléatoires $b_1, b_2, \cdots$, un à un, où le bit b_i est produit à l'étape i. Si $b_i < p_i$, alors l'algorithme produit la réponse pile, et s'arrête. Si $b_i > p_i$, la machine produit la réponse face et s'arrête. Sinon, l'algorithme passe à l'étape $i+1$. Clairement, l'algorithme atteint l'étape $i+1$ si et seulement si $b_j = p_j$ pour tout $j \leq i$, ce qui se produit avec probabilité $1/2^i$. La probabilité de répondre pile est donc $\sum_i p_i 1/2^i$, ce qui vaut exactement ρ. D'autre part, le temps moyen, de la forme $\sum_i i^c 1/2^i$ pour une constante c, est bien borné par une constante. □

12.7.2 BPP et circuits polynomiaux

Rappelons que $\mathbb{P}$ désigne les langages reconnus par une famille de circuits de taille polynomiale.

Théorème 12.9 (Sipser-Gâcs) BPP $\subset \mathbb{P}$.

La preuve est non constructive, et découle d'un argument existentiel.

Démonstration: Par les techniques de réduction d'erreur discutées auparavant, on sait que pour un langage L de BPP, on peut construire un algorithme A qui sur les entrées de taille n utilise m tirages aléatoires, et tel que pour tout w de longueur n, $\Pr(A \text{ se trompe sur } w) \leq 2^{-n-1}$.

Qualifions un mot $y \in \{0,1\}^m$ (correspondant à une suite de tirages aléatoires) de *mauvais* si pour une entrée w, A se trompe sur une entrée w de longueur n, et de *bon* sinon.

Pour toute entrée w, il y a au plus $\frac{2^m}{2^{n+1}}$ mots qui sont mauvais pour w. En additionnant sur tous les mots w de longueur n, il y a au plus $2^n.\frac{2^m}{2^{n+1}} = 2^m/2$ mots qui sont mauvais pour *au moins un mot* w.

Les mots qui ne sont pas mauvais pour au moins un mot w, doivent donc être bons pour tous les mots w : en particulier, il doit exister un mot y qui est bon pour *toute* entrée w de taille n.

On peut alors construire pour chaque taille n d'entrée un circuit pour les mots de taille n qui contient codé "en dur" dans le circuit ce mot y. □

12.8 Notes bibliographiques

Ce chapitre contient des résultats classiques. Leur rédaction est inspirée de leur présentation dans [Balcázar et al., 1988], [Arora and Barak, 2009] ainsi que [Sipser, 1997].

Chapitre 13

Hiérarchie polynômiale

Nous nous restreignons volontairement dans ce chapitre au strict minimum, renvoyant aux PC pour des compléments, ou à d'autres sources pour des présentations des preuves, et l'équivalence de cette présentation avec la présentation de la hiérarchie par oracles, ou par machines alternantes.

13.1 Motivation

La hiérarchie polynomiale PH est une hiérarchie de problèmes entre P et PSPACE.

Pour motiver l'étude de PH, nous allons présenter quelques problèmes qui semblent ne pas être capturés par la notion de NP-complétude.

Rappelons qu'un *stable d'un graphe* $G = (V, E)$ est un sous-ensemble $V' \subset V$ ne contenant aucune paire de sommets voisins dans G.

Nous avons vu dans le chapitre 8 qu'étant donné un graphe G et un entier k, le problème de savoir s'il admet un stable de taille $\geq k$ est NP-complet.

Pour déterminer si une paire $< G, k >$ doit être acceptée, il suffit de savoir s'il existe un certificat, à savoir un stable de taille $\geq k$.

Supposons qu'étant donné un graphe G et un entier k, on souhaite savoir si le plus grand stable de G soit de taille exactement k.

Cette fois, il ne semble pas y avoir de certificat simple. En fait, pour déterminer si une paire $< G, k >$ doit être acceptée, il suffit de savoir s'il existe un certificat, à savoir un stable de taille $\geq k$, tel que toute autre stable soit de taille inférieure.

De façon similiaire, étant donnée une formule booléenne en forme normale disjonctive, le problème de savoir si elle est satisfiable est NP-complet.

Pour savoir si elle est satisfiable, il suffit de savoir s'il existe un certificat, à savoir une affectation des variables qui la rend vraie.

Supposons qu'étant donnée une formule booléenne en forme normale disjonctive, on souhaite savoir si elle est de taille minimale. Cette fois, il ne semble pas y avoir de certificat simple. En fait, pour déterminer si ϕ est de taille minimale,

il suffit de vérifier que pour toute autre formule ψ de taille inférieure, il existe une affectation $\overline{x}$ des variables sur lesquelles $\phi(\overline{x}) \neq \psi(\overline{x})$.

13.2 Hiérarchie polynomiale

13.2.1 Définitions

La définition de la hiérarche polynomiale généralise NP, coNP. Cette classe est constituée de tous les langages qui peuvent se définir par une combinaison de prédicats calculables en utilisant un nombre polynomial de quantificateurs $\forall$ et $\exists$.

Définition 13.1 (Classe Σ_i^P) *Soit $i \geq 1$. Un langage L est dans Σ_i^P s'il existe un polynôme q et un problème A dans P tel que*

$$w \in L$$

$$\Leftrightarrow$$

$$\exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \cdots Q_i u_i \in \{0,1\}^{q(|x|)} < w, u_1, u_2, \cdots, u_i > \in A,$$

où Q_i est $\forall$ ou $\exists$ selon si i est pair ou impair respectivement.

Définition 13.2 (Classe Π_i^P) *Soit $i \geq 1$. Un langage L est dans Π_i^P s'il existe un polynôme q et un problème A dans P tel que*

$$w \in L$$

$$\Leftrightarrow$$

$$\forall u_1 \in \{0,1\}^{q(|x|)} \exists u_2 \in \{0,1\}^{q(|x|)} \cdots Q_i u_i \in \{0,1\}^{q(|x|)} < w, u_1, u_2, \cdots, u_i > \in A,$$

où Q_i est $\exists$ ou $\forall$ selon si i est pair ou impair respectivement.

On obtient immédiatement :

Corollaire 13.1 $\Sigma_1^P = \text{NP}$.

Corollaire 13.2 *co*$\Sigma_i^P = \Pi_i^P$ *et co*$\Pi_i^P = \Sigma_i^P$.

Corollaire 13.3 $\Pi_1^P = \text{coNP}$.

Corollaire 13.4 *Pour tout i,* $\Sigma_i^P \subset \Pi_{i+1}^p$ *et* $\Pi_i^P \subset \Sigma_{i+1}^p$.

Définition 13.3 $\text{PH} = \bigcup_i \Sigma_i^P = \bigcup_i \Pi_i^p$.

13.2.2 Premières propriétés

Puisque chacune des classes Σ_i^P et Π_i^P est clairement dans PSPACE, on a

Théorème 13.1

$$\mathrm{P} \subset \mathrm{PH} \subset \mathrm{PSPACE}.$$

On pense généralement que P $\neq$ NP et NP $\neq$ coNP. On peut être tenté de généraliser cela en disant que pour tout entier i, Σ_i^P est strictement inclus dans Σ_{i+1}^P, et que Π_i^P est strictement inclus dans Π_{i+1}^P.

Cette conjecture est souvent formulée en théorie de la complexité sous la terminologie de *la hiérarchie polynomiale ne s'effondre pas* : la hiéarchie polynomiale serait dite *effondrée* s'il existe un entier i tel que $\Sigma_i^P = \Pi_i^P$.

Dans ce cas, on aurait

Théorème 13.2
- *Si pour un entier i, $\Sigma_i^P = \Pi_i^P$ alors* PH $= \Sigma_i^P$ *: la hiérarchie est dite* effondrée *au niveau i, et la propriété est vraie alors pour tous les niveaux supérieurs.*
- *Si* P = NP *alors* PH = P *: la hiérarchie s'effondre en* P.

13.2.3 Problèmes complets

Théorème 13.3 *Pour tout $i \geq 1$, les classes Σ_i^P et Π_i^P possèdent des problèmes complets.*

Théorème 13.4 *Si* PH *possède un problème complet, alors la hiérarchie polynomiale s'effondre.*

Corollaire 13.5 *Si* PH = PSPACE*, alors la hiérarchie polynomiale s'effondre.*

13.3 Quelques propriétés

Théorème 13.5

$$\mathrm{BPP} \subset \Sigma_2^P \cap \Pi_2^P.$$

Théorème 13.6 *Si* NP $\subset \mathbb{P}$ *alors* PH $= \Sigma_2^p$.

13.4 Notes bibliographiques

Nous avons rédigé ce chapitre en utilisant [Balcázar et al., 1988] et [Arora and Barak, 2009].

Chapitre 14

Protocoles interactifs

14.1 Preuves interactives

14.1.1 Discussion

Les systèmes de preuve interactifs donnent un moyen de définir une version probabiliste de la classe NP, comme les algorithmes probabilistes permettent de définir l'analogue probabiliste de la classe P.

L'étude des systèmes de preuve interactifs a eu de nombreuses conséquences en théorie de la complexité.

Les langages de NP sont ceux pour lesquels, étant donné un mot, son appartenance au langage peut être testée facilement par l'existence d'un certificat court : reformulons cette affirmation en considérant deux entités : un *prouveur* qui produit une preuve d'appartenance, et un *vérificateur* qui la vérifie. Considérons que le prouveur cherche à convaincre le vérificateur de l'appartenance d'un mot donné w au langage. On suppose que le vérificateur correspond à un algorithme en temps polynomial, sinon il pourrait trouver la réponse lui-même. Par contre, on n'impose aucune contrainte sur le prouveur, puisque produire une preuve peut être difficile.

Considérons le problème SAT par exemple. Un prouveur peut convaincre un vérificateur qu'une formule ϕ est satisfiable en fournissant une affectation de ses variables sur laquelle la formule s'évalue en vrai.

Un prouveur peut-il convaincre un vérificateur qu'une formule n'est pas satisfiable ?

Le complémentaire de SAT n'est pas connu comme étant dans NP (et est coNP-complet) et donc on ne peut pas s'attendre à utiliser l'idée de l'existence d'un certificat court (sauf si P = NP). Cependant, il s'avère que oui, si l'on ajoute au prouveur et au vérificateur deux caractéristiques supplémentaires : premièrement, ils travaillent en fait en dialoguant. Deuxièmement, le vérificateur peut être un algorithme *probabiliste* qui doit simplement produire la bonne réponse avec forte probabilité, plutôt qu'avec certitude.

Un tel système de prouveur et de vérificateur constitue un *système de preuve interactif*.

14.1.2 Illustration

Comme exemple intuitif de la puissance de la combinaison de l'utilisation de randomisation et d'interactions, considérons le scénario suivant : Maria possède une chaussette rouge et une chaussette jaune, mais son ami Arthur, qui est daltonien, ne croit pas que ses chaussettes sont de couleur distincte. Comment peut-elle le convaincre ?

Voici une méthode : Maria donne les deux chaussettes à Arthur, lui dit laquelle est rouge, laquelle est jaune, et Arthur prend la chaussette rouge dans sa main droite, la chaussette jaune dans sa main gauche. Alors Maria tourne le dos à Arthur et Arthur lance une pièce. Si la pièce dit "pile", alors Arthur garde les chaussettes comme elles sont. Si c'est "face", il les échange entre main droite et main gauche. Il demande alors à Maria de deviner s'il a échangé les chaussettes ou pas. Bien entendu, il suffit à Maria de regarder si les chaussettes sont différentes, mais si elles sont identiques, alors Maria ne peut mieux faire qu'avoir une réponse correcte avec une probabilité $1/2$. En répétant l'expérience, disons 100 fois, si Maria répond toujours correctement, Arthur peut se convaincre que les chaussettes sont bien de couleur distincte.

Cette idée est celle derrière le fait que l'on peut résoudre le problème du non-isomorphisme de graphe avec un protocole interactif.

14.1.3 Définition formelle

On voit le *vérificateur* comme une fonction V qui calcule le message suivant à transmettre au prouveur à partir de l'historique de la communication jusque là. La fonction V possède trois entrées :

- l'entrée w, dont on cherche à déterminer l'appartenance au langage ;
- un mot y qui code une suite aléatoire : même si cela est équivalent, pour des raisons de commodités dans les définitions, on préfère voir un vérificateur comme prenant en seconde entrée un mot sur $\{\mathbf{0}, \mathbf{1}\}$ aléatoire, plutôt qu'un algorithme probabiliste qui ferait des choix aléatoires lors de son exécution ;
- l'historique des échanges de messages jusque là : si les messages m_1, m_2, $\cdots$, m_i ont été échangés jusque là, on note $m_1 \# m_2 \cdots \# m_i$ le mot qui code les messages m_1 jusqu'à m_i.

La sortie de la fonction V est soit le message suivant m_{i+1} dans l'interaction, soit "accepter" soit "rejeter", ce qui termine l'interaction. Autrement dit, $V : M^* \times M^* \times M^* \to M^*$ où M est l'alphabet, et $V(w, y, m_1 \# m_2 \cdots \# m_i) = m_{i+1}$ signifie que sur l'entrée w, et la suite de tirages aléatoires y, l'historique de messages $m_1, \cdots, m_i$, le prochain message du vérificateur est m_{i+1}.

Le *prouveur* est lui sans aucune limite de puissance : on le voit comme une fonction P qui prend deux entrées

- l'entrée w

– l'historique des échanges de message jusque là.

$P(w, m_1\#m_2\cdots\#m_i) = m_{i+1}$ signifie que le prouveur envoie le message m_{i+1} au vérificateur après avoir échangé les messages $m_1, \cdots, m_i$ sur l'entrée w.

On définit ensuite l'interaction entre le prouveur et le vérificateur. Pour une entrée w, on note $V \leftrightarrow P(w, y) = \textit{accepter}$ s'il existe une suite de messages $m_1 m_2 \cdots m_k$ telle que

– pour $0 \leq i < k$, où i est pair, $V(w, y, m_1\#\cdots\#m_i) = m_{i+1}$

– pour $0 < i < k$, ou i est impair, $P(w, m_1\#\cdots\#m_i) = m_{i+1}$

– le dernier message m_k est "accepter".

Pour simplifier la définition de la classe IP, on suppose que la longueur de l'argument y qui code les choix aléatoires et la longueur de chacun des messages m_i échangés restent inférieure à $p(n)$ pour un polynôme p qui ne dépend que du vérificateur. On suppose d'autre part que le nombre total de messages échangés reste inférieur à $p(n)$.

Pour une entrée w de taille n, on définit

$$\Pr(V \leftrightarrow P \text{ accepte } w) = \Pr(V \leftrightarrow P(w, y) = \textit{accepter}).$$

Définition 14.1 (Classe IP**)** *On dit qu'un langage L est dans* IP *s'il existe une fonction V calculable en temps polynomial, et une fonction P arbitraire telle que pour toute fonction $\overline{P}$, et pour toute entrée w*

– *$w \in L$ implique que* $\Pr(V \leftrightarrow P$ *accepte* $w) \geq 2/3$.

– *$w \in L$ implique que* $\Pr(V \leftrightarrow \overline{P}$ *accepte* $w) \leq 1/3$.

Remarque 14.1 *En fait, comme on peut voir* NP *comme l'ensemble des théorèmes qui admettent une preuve courte, on peut voir les protocoles interactifs comme ceux qui admettent une* preuve interactive *courte.*

Remarque 14.2 *On fait l'hypothèse que les choix probabilistes du vérificateur sont privés, et pas accessibles au prouveur.*

Remarque 14.3 *On peut voir P comme quelqu'un qui essaye de convaincre V que l'entrée vérifie une certaine propriété. En acceptant, V indique qu'il est convaincu que la propriété est vraie. V ne doit pas accepter si un intrus $\overline{P}$ s'insère.*

On peut amplifier les probabilités de succès d'un protocole interactif par répétition, comme nous l'avions vu pour les classes probabilistes RP et BPP, pour rendre les probabilités d'erreur exponentiellement faibles.

Par définition, on obtient directement.

Corollaire 14.1 NP $\subset$ IP.

Démonstration: Soit $L \in$ NP. On a $w \in L$ si et seulement s'il existe un certificat $u \in M^*$ tel que $< w, u > \in A$, pour un problème polynomial A. Il suffit d'un protocole en un échange : sur l'entrée w, le prouveur fournit le certificat u

pour une entrée w dans le langage, et n'importe quoi pour une entrée qui n'est pas dans le langage. Puisque le prouveur n'est pas supposé avoir une puissance restreinte, il peut toujours trouver un certificat u tel que $< w, u > \in A$ lorsque $w \in L$. Le vérificateur détermine alors si $< w, u > \in A$. Si c'est le cas, il accepte, sinon, il rejette. Le vérificateur accepte donc les mots de L avec probabilité 1, et les mots du complémentaire de L avec probabilité 0, puisqu'aucun prouveur ne peut convaincre le vérificateur. □

Trivialement :

Corollaire 14.2 BPP $\subset$ IP.

On verra en fait que IP = PSPACE.

14.1.4 Exemple : non-isomorphisme de graphes

On présente un exemple de problème dans IP qui n'est pas connu pour être dans NP. Étant donné un graphe G, pour le représenter, on a souvent besoin de numéroter ses sommets. On dit que deux graphes G_1 et G_2 sont isomorphes s'ils sont identiques à une renumérotation des sommets près. Si l'on préfère, s'il existe une permutation π des numéros des sommets de G_1 telle que $\pi(G_1) = G_2$, où $\pi(G)$ désigne le graphe obtenu en remplaçant le numéro en chaque sommet par son image par la permutation π. Le problème *de l'isomorphisme de graphes* consiste, étant donnés deux graphes G_1 et G_2 à déterminer s'ils sont isomorphes.

Clairement, le problème de l'isomorphisme de graphe est dans NP, puisque l'on peut prendre comme certificat la description de la permutation π. La question de savoir si ce problème est NP-complet est une question ouverte, qui a été relativement abondamment discutée dans la littérature.

En fait, on peut montrer que le non-isomorphisme de graphe est dans IP, puisqu'on peut déterminer si deux graphes ne sont pas isomorphes.

Le protocole IP est le suivant :

- V choisit $i \in \{1, 2\}$ au hasard de façon uniforme. On permute les sommets de G_i pour obtenir un nouveau graphe H. V envoie H à P.
- P identifie lequel de G_1 et de G_2 a été utilisé pour produire H. Soit G_j ce graphe. P envoie j à V.
- V accepte si $i = j$. Rejette sinon.

Pour se convaincre qu'il s'agit d'un protocole interactif correct, observons que si G_1 n'est pas isomorphe à G_2, alors il existe un prouveur tel que V accepte avec probabilité 1, parce que les graphes ne sont pas isomorphes.

Sinon, si G_1 et G_2 sont isomorphes, alors le mieux qu'un prouveur puisse faire est de deviner parce qu'une permutation de G_1 ressemble exactement à une permutation de G_2. Donc dans ce cas, pour tout prouveur la probabilité que V accepte est $\leq 1/2$.

14.1.5 IP = PSPACE

Théorème 14.1

IP = PSPACE.

14.2 Vérification probabiliste de preuve

14.2.1 Définition

On considère un modèle qui est essentiellement le même que précédemment, si ce n'est que qu'on n'autorise pas les erreurs dans le cas positif.

Définition 14.2 (Classe PCP**)** *On dit qu'un langage L est dans* PCP *s'il existe une fonction V calculable en temps polynomial, et une fonction P arbitraire telle que pour toute fonction $\overline{P}$, et pour toute entrée w*
- *$w \in L$ implique que* $\Pr(V \leftrightarrow P \text{ accepte } w) = 1$.
- *$w \in L$ implique que* $\Pr(V \leftrightarrow \overline{P} \text{ accepte } w) \leq 1/2$.

Encore une fois, en utilisant des répétitions (réduction de l'erreur), la constante $1/2$ est arbitraire, on pourrait prendre n'importe quel $\epsilon > 0$.

On peut supposer que le vérificateur inclue l'entrée w et les précédents messages $m_1, \cdots, m_i$ dans chacune de ses requêtes à P, et que la réponse de P est un unique bit : en faisant comme cela, on peut voir P comme un *oracle*, c'est-à-dire comme un langage.

On présente dans la littérature souvent PCP en présentant l'oracle comme détenant une preuve que l'entrée w est dans L, et les requêtes comme l'extraction de bits de cette preuve.

On place par ailleurs généralement des bornes sur le nombre de bits aléatoires que V utilise, et le nombre de requêtes à l'oracle qu'il peut faire.

On dit qu'un protocole interactif est $(r(n), q(n))$-borné si sur toute entrée de taille n, le vérificateur utilise moins de $r(n)$ bits aléatoires, et fait au plus $q(n)$ requêtes à son oracle. On note $\text{PCP}(r(n), q(n))$ la famille des langages reconnaissables par un protocole $(\mathcal{O}(r(n)), \mathcal{O}(q(n)))$-borné.

14.2.2 $\text{NP} = \text{PCP}(\log n, 1)$

Le théorème suivant est très puissant, et est la conséquence de toute une lignée de travaux scientifiques :

Théorème 14.2

$$\text{NP} = \text{PCP}(\log n, 1).$$

La preuve du théorème n'est pas facile.

Ce théorème dit que les problèmes de NP possèdent des protocoles interactifs qui utilisent au plus $\mathcal{O}(\log n)$ bits aléatoires, et qui utilisent au plus un nombre constant, indépendant de la taille de l'entrée, de bits de la preuve.

14.2.3 Conséquences sur la théorie de l'approximation

Le théorème possède toutefois des conséquences directes sur la théorie de l'approximation. On a vu des problèmes variés de décision comme par exemple 3SAT qui sont NP-complets. Souvent à ces problèmes est associé un problème d'optimisation qui peut être un problème de maximisation ou de minimisation.

Par exemple, le problème MAX3SAT demande de produire une affectation des variables d'une formule ϕ en forme normale conjonctive avec trois littéraux par clause (en $3CNF$) qui maximise le nombre de clauses satisfaites.

Un α-algorithme d'approximation pour un problème d'optimisation est un algorithme qui produit une réponse qui reste dans un facteur constant α de l'optimal. Pour un problème comme MAX3SAT, cela signifie produire une réponse qui rend le nombre de clauses satisfaites au moins α multiplié par le maximum, pour $0 < \alpha \leq 1$, indépendemment de la taille de l'entrée. La constante α se nomme le rapport d'approximation.

Proposition 14.1 *Il existe un algorithme en temps polynomial qui, étant donné une formule ϕ en $3CNF$ avec n variables et m clauses, telle que chaque clause possède trois variables distinctes, produit une affectation qui satisfait $7m/8$ clauses.*

Théorème 14.3 *Il existe un α-algorithme polynomial d'approximation pour* MAX3SAT *si et seulement si $P =$ NP.*

14.3 Notes bibliographiques

Ce chapitre a été rédigé en reprenant essentiellement [Sipser, 1997], [Kozen, 2006] et [Arora and Barak, 2009].

Bibliographie

[Arora and Barak, 2009] Arora, S. and Barak, B. (2009). *Computational Complexity : A Modern Approach*. Cambridge University Press.

[Balcàzar et al., 1990] Balcàzar, J., Dìaz, J., and Gabarrò, J. (1990). *Structural Complexity II*, volume 22 of *EATCS Monographs on Theoretical Computer Science*. Springer.

[Balcázar et al., 1988] Balcázar, J. L., Diáz, J., and Gabarró, J. (1988). *Structural Complexity I*, volume 11 of *EATCS Monographs on Theoretical Computer Science*. Springer.

[Blum et al., 1998] Blum, L., Cucker, F., Shub, M., and Smale, S. (1998). *Complexity and Real Computation*. Springer-Verlag.

[Blum et al., 1989] Blum, L., Shub, M., and Smale, S. (1989). On a theory of computation and complexity over the real numbers ; NP completeness, recursive functions and universal machines. *Bulletin of the American Mathematical Society*, 21(1) :1–46.

[Börger and Stärk, 2003] Börger, E. and Stärk, R. (2003). *Abstract State Machines*. Springer.

[Carton, 2008] Carton, O. (2008). Langages formels, calculabilité et complexité.

[Cole, 1988] Cole, R. (1988). Parallel merge sort. *SIAM Journal on Computing*, 17 :770.

[Cook, 1985] Cook, S. (1985). A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1-3) :2–22.

[Cori and Lascar, 1993] Cori, R. and Lascar, D. (1993). *Logique mathématique. Volume I*. Mason.

[Dershowitz and Gurevich, 2008] Dershowitz, N. and Gurevich, Y. (2008). A natural axiomatization of computability and proof of Church's Thesis. *The Bulletin of Symbolic Logic*, 14(3) :299–350.

[Dowek, 2008] Dowek, G. (2008). *Les démonstrations et les algorithmes*. Polycopié du cours de l'Ecole Polytechnique.

[Goode, 1994] Goode, J. B. (1994). Accessible telephone directories. *The Journal of Symbolic Logic*, 59(1) :92–105.

[Gurevich, 2000] Gurevich (2000). Sequential abstract-state machines capture sequential algorithms. *ACMTCL : ACM Transactions on Computational Logic*, 1.

[Hoover et al., 1984] Hoover, H., Klawe, M., and Pippenger, N. (1984). Bounding fan-out in logical networks. *Journal of the ACM (JACM)*, 31(1) :13–18.

[Hopcroft et al., 2001] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition.

[Jones, 1997] Jones, N. (1997). *Computability and complexity, from a programming perspective*. MIT press.

[Karp and Ramachandran, 1991] Karp, R. and Ramachandran, V. (1991). Parallel algorithms for shared-memory machines. In *Handbook of theoretical computer science (vol. A)*, page 941. MIT Press.

[Koiran, 1994] Koiran, P. (1994). Computing over the reals with addition and order. *Theoretical Computer Science*, 133(1) :35–47.

[Kozen, 1997] Kozen, D. (1997). *Automata and computability*. Springer Verlag.

[Kozen, 2006] Kozen, D. (2006). *Theory of computation*. Springer-Verlag New York Inc.

[Lassaigne and de Rougemont, 1996] Lassaigne, R. and de Rougemont, M. (1996). *Logique et complexité*. Hermes.

[Legrand et al., 2003] Legrand, A., Robert, Y., and Robert, Y. (2003). *Algorithmique parallèle : cours et exercices corrigés*. Dunod.

[Matiyasevich, 1970] Matiyasevich, Y. (1970). Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR*, 191(2) :279–282.

[Michaux, 1989] Michaux, C. (1989). Une remarque à propos des machines sur $\mathbb{R}$ introduites par blum, shub et smale. *Comptes Rendus de l'Académie des Sciences de Paris, Série I*, 309 :435–437.

[Mitzenmacher and Upfal, 2005] Mitzenmacher, M. and Upfal, E. (2005). *Probability and Computing. Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press.

[Motwani and Raghavan, 1995] Motwani, R. and Raghavan, P. (1995). *Randomized Algorithms*. Cambridge University Press.

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

[Pippenger, 1979] Pippenger, N. (1979). On simultaneous resource bounds. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 307–311.

[Poizat, 1995] Poizat, B. (1995). *Les petits cailloux : Une approche modèle-théorique de l'Algorithmie*. Aléas Editeur.

[Roch, 1995] Roch, J.-L. (1995). Complexité parallèle et algorithmique pram.

[Sipser, 1997] Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing Company.

[Turing, 1936] Turing, A. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2) :230–265. Reprinted in Martin Davis. The Undecidable : Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions. Raven Press, 1965.

[Wolper, 2001] Wolper, P. (2001). *Introduction à la calculabilité : cours et exercices corrigés*. Dunod.