Cours de Modèles de calcul

Rédaction : Laure Danthony, MIM00
d'après le cours d'Arnaud Carayol, MIM99

Cours : Marianne Delorme, janvier - mai 2001

Table des matières

Chapitre 1

Quelques modèles de calcul

On introduit dans ce chapitre les machines de Turing (voir cours de décidabilité), les machines RAM, les algorithmes de Markov (cf décidabilité), les circuits booléens.

Tous trois caractérisent les fonctions semi-calculables via la thèse de Church. On parle de fonction semi-calculables (ou ppr), de fonctions récursives à la Church-Rosser, de fonctions calculables par algorithme, ou encore de fonctions effectivement calculables. On évoquera aussi les modèles de calcul plus originaux que sont les circuits booléens et les automates cellulaires.

Par fonction, on entend fonctions de $\mathbb{N}$ dans $\mathbb{N}$ ou de Σ^* dans Σ^*, Σ étant un alphabet (fini). Ceci n'est pas restrictif dans la mesure où les n-uplets sont codés de façon canonique dans ces ensembles.

On évoquera en outre par la suite deux autres modèles de calcul un peu différents : les circuits booléens et les automates cellulaires. On introduira ensuite la notion de système de programmation acceptable, et on terminera en définissant des mesures de complexité sur les modèles présentés ici.

1.1 Machines de Turing

1.1.1 Définitions

La définition est ici d'une machine de Turing déterministe à 1 ruban. On signale l'équivalence avec les machines à plusieurs rubans, et aussi la possibilité de se restreindre à un alphabet $\{0, 1, B\}$

Définition 1.1.1 (Machine de Turing)
Une **machine de Turing** est ici un 8-uplet

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

dont les éléments s'interprètent de la façon suivante :

Q est l'ensemble (fini) des états ;

Σ est l'alphabet (fini) sur lequel agit la machine ;

Γ est l'alphabet de travail, il s'agit d'une extension de Σ contenant des caractères de contrôle supplémentaires ;

B est le caractère "blanc" ;

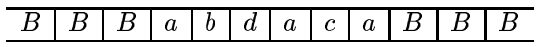
q_0 est l'état initial ;

q_a est l'état d'acceptation (état d'arrêt) ;

q_r est l'état de rejet (état d'arrêt également) ;

δ est la fonction de transition, de $Q \times \Gamma$ dans $Q \times \Gamma \times Mvt$, Mvt étant l'ensemble des mouvements de la tête, intuitivement $\{-1, 0, 1\}$.

DÉFINITION 1.1.2 (CONFIGURATION)
Une **configuration** est intuitivement la donnée d'un ruban



et de la position de la tête de lecture/écriture sur ce ruban, ainsi que de l'état dans lequel se trouve la machine. On note une configuration uqv avec u et v dans Γ^* et q dans Q, où u et v représentent respectivement la partie du ruban qui précède et qui suit la position de la tête, et où q est l'état actuel.

DÉFINITION 1.1.3 (CALCUL)
Un **calcul** de M est une suite finie de configurations $(C_i)_{i \geqslant 0}$ telle que pour tout i C_{i+1} dérive de C_i selon la fonction de transition δ.

1.1.2 Résultats classiques

On connaît le résultat fondamental selon lequel il existe une énumération $(\Phi_i)_{i \geqslant 0}$ des machines de Turing, fonction de $\mathbb{N}$ dans Ω, ensemble des mots codant une machine dans un alphabet et un codage donnés. On en *déduit* une énumération $(\varphi_i)_{i \geqslant 0}$ des fonctions semi-calculables.

On déduit de cela l'existence d'une machine de Turing φ_u qualifiée d'universelle qui vérifie pour tous i et j :

$$\varphi_u(\langle i, j \rangle) = \varphi_i(j)$$

où la notation $\langle \cdots \rangle$ désigne les bijections usuelles entre $\mathbb{N}^p$ et $\mathbb{N}$ ou entre $\mathbb{N}^*$ et $\mathbb{N}$ selon le cas.

On a également comme conséquence le théorème snm, selon lequel pour tous m et n entiers, il existe une fonction s_n^m de $\mathbb{N}^{m+1}$ dans $\mathbb{N}$ qui vérifie, pour toutes les entiers x, $y_1, \ldots, y_m$ et $z_1, \ldots, z_n$:

$$\varphi_x(\langle y_1, \ldots, y_m, z_1, \ldots, z_n \rangle) = \varphi_{s_n^m(x, y_1, \ldots, y_m)}(\langle z_1, \ldots, z_n \rangle)$$

1.2 Machines RAM

DÉFINITION 1.2.1
On dispose d'un alphabet $(a_i)_{i \leq k}$ et de registres. Les instructions autorisées sont les suivantes :

1$_j$ X add_j Y, $j \in \{1, \ldots k\}$ (concaténer au mot de Y la lettre a_j).
2 X *del* Y (effacer le premier caractère du mot contenu par Y).
3 X *clr* Y (effacer le contenu de Y, c'est à dire le remplacer par ε).
4 X Z← Y (substituer au contenu de Z le contenu de Y ; Y n'est pas modifié).

5 X $jmpN$ (aller à la ligne de nom N).
6_j X Yjmp_jN (si le mot contenu par Y commence par a_j, aller à la ligne N).
7 $CONTINUE$ (ne rien faire).
X est un numéro de ligne, Y et Z des registres, N est un nom de ligne : N_a(avant la ligne), N_b(après la ligne).

DÉFINITION 1.2.2
Un **programme RAM** est la donnée d'un nombre fini de registres et d'une liste finie d'instructions cohérentes avec une dernière instruction qui est de type 7.

DÉFINITION 1.2.3
Une fonction est **RAM-calculable** s'il existe un programme RAM qui calcule $f(u)$ (qui l'inscrit sur R_1 à la fin et termine) ssi $u \in Dom(f)$; sinon elle boucle.

PROPOSITION 1.2.1 *Les fonctions RAM calculables sont les fonctions Turing-calculables.*

1.3 Algorithmes de Markov

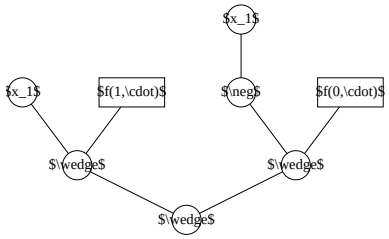
DÉFINITION 1.3.1
Un **algorithme de Markov** est la donnée d'un certain nombre de règles, données dans un ordre défini, et que l'on manipule d'une certaine façon (facteur le plus à gauche, règle la plus petite).
Plus formellement, un algorithme de Markov sur un alphabet fini Σ est une suite finie de règles (ou productions) de la forme $p_i \to q_i$ ou $p_i \to_{(t)} q_i$ (t pour terminal). On transforme un mot en son sucesseur par essai de "matching" des règles en partant par la règle de numéro plus petit. Si elle est étiquetée par (t), l'algorithme termine. Sinon, on réessaie les règles (toujours en partant de la règle 1). L'algo termine aussi si plus aucune règle ne peut plus s'appliquer.

DÉFINITION 1.3.2
Comment un tel système *calcule*-t-il une fonction f de $(\Sigma^*)^p$ dans Σ^* ? On considère pour cela un alphabet Γ qui contient Σ et un symbole ¡¡ , ¿¿ (virgule) non élément de Σ. Si M est un algorithme de Markov, on dit qu'il **calcule (ou semi-calcule)** f si et seulement si :

1. $M(x_1, x_2, \ldots, x_p)$ est défini si et seulement si $(x_1, x_2, \ldots, x_p) \in Dom(f)$
2. alors $M(x_1, x_2, \ldots, x_p) = f(x_1, x_2, \ldots, x_p)$

EXEMPLE 1.3.1 *On considère l'alphabet $\Sigma = \{a_1, a_2\}$ et la fonction S_2 de Σ^* sur lui-même définie par $S_2(u) = ua_2$. Cette fonction est alors calculée par le système de Markov suivant :*

1. $\$a_1 \to a_1\$$
2. $\$a_2 \to a_2\$$
3. $\$ \to a_2$
4. $\epsilon \to \$$



PROPOSITION 1.3.1 *Les fonctions MT-calculables sont les fonctions Markov-calculables.*

PREUVE : en exercice. On peut cependant préférer se convaincre du fait qu'il est possible de simuler toute machine de Turing par un algorithme de Markov et réciproquement. ■

1.4 Circuits booléens

Il est naturel de s'intéresser à de tels objets, étant donnée leur ressemblance évidente avec les circuits réels utilisés dans les machines. D'autre part, les circuits booléens permettent de rendre compte de phénomènes de parallélisme. Enfin, leur rapport avec la logique, et donc les mathématiques, est évident.

1.4.1 Formules et circuits

On choisit les formules contruites sur $\{\neg, \vee, \wedge\}$. On étudie les fonctions booléennes : $\{0,1\}^n \to \{0,1\}$.

DÉFINITION 1.4.1
Un **circuit booléen** est la donnée d'un graphe fini acyclique, avec V l'ensemble des portes : une porte est de type 0, 1, x_i (entrée), $\neg$, $\wedge$ ou $\vee$. La porte $n+q$ est la porte de sortie, de degré sortant 0. On associe à un circuit la fonction qu'il calcule.

DÉFINITION 1.4.2
La **taille** d'un circuit est son nombre de portes (cf complexité en espace) ; la **profondeur** d'un circuit est la longueur d'un plus long chemin (cf complexité en profondeur).

PROPOSITION 1.4.1 *Pour toute fonction booléenne n-aire, la taille minimum d'un circuit qui calcule f est strictement inférieure à 2^{n+2}.*

PREUVE : Notons f sous la forme $f(x_1, \ldots, x_n)$. On peut alors écrire

$$f(x_1, \ldots, x_n) = (x_1 \wedge f(1, x_2, \ldots, x_n)) \vee (\neg x_1 \wedge f(0, x_2, \ldots, x_n))$$

Par conséquent, le circuit 1.4.1 calcule f en supposant connus des circuits calculant $f(0,\cdot)$ et $f(1,\cdot)$: On ajoute donc quatre portes en ajoutant une variable.

Ceci prouve donc par récurrence que le nombre minimal de portes nécessaires au calcul d'une fonction booléenne à n variables est majoré par $g(n)$ où :

$$\begin{cases} g(1) = 2 \\ g(n+1) = 2g(n) + 4 \end{cases}$$

On montre alors facilement par récurrence que $g(n) < 2^{n+2}$. ■

PROPOSITION 1.4.2 *Pour tout $n \geq 2$, il existe une fonction booléenne f telle que aucun circuit calculant f ne soit de taille inférieure à $\frac{2^n}{2n}$.*

PREUVE : Désignons par m le nombre $2^n/2n$. On va comparer le nombre de fonctions booléennes de $\{0,1\}^n$ dans $\{0,1\}$, qui est 2^{2^n}, au nombre de circuits possibles de taille inférieure ou égale à m.

Une porte dans un circuit est caractérisée par son type et par les portes avec lesquelles elle est liée. Comme il y a $n+5$ étiquettes possibles et m nœuds, chaque arête étant reliée à au plus deux nœuds. Le nombre de configurations possibles pour un nœud donné est donc majoré par $(n+5)m^2$, ce qui donne un nombre maximal de circuits de $\left((n+5)m^2\right)^2$. Comme $m = 2^n/2n$, on a donc :

$$\left((n+5)m^2\right)^m = \left((n+5)\left(\frac{2^n}{2n}\right)^2\right)^{\frac{2^n}{2n}} = \left((n+5)\frac{2^{2n}}{4n^2}\right)^{\frac{2^n}{2n}}$$

$$\log\left(\left((n+5)m^2\right)^m\right) = \frac{2^n}{2n}\left(\log\left(\frac{n+5}{4n^2}\right) + 2n\right) = 2^n\left(1 - \frac{\log\frac{4n^2}{n+5}}{2n}\right)$$

Donc la comparaison de 2^{2^n} à $\left((n+5)m^2\right)^2$ peut se réduire en passant au logarithme (en base 2) à la comparaison de 2^n au dernier terme, donc à la comparaison de 1 à

$$1 - \frac{\log\frac{4n^2}{n+5}}{2n}$$

donc finalement à l'étude du signe du logarithme. Pour $n \geqslant 2$, comme $4n^2$ devient supérieur à $n+5$, le nombre de fonctions booléennes à n inconnues devient donc supérieur au nombre de circuits booléens à n variables. Le résultat est donc démontré. ■

1.4.2 Familles de circuits

Le problème c'est que pour étudier une calculabilité, on dispose de fonctions de $\{0,1\}^* \to \{0,1\}$. On va donc être amené à considérer une famille de circuits.

DÉFINITION 1.4.3
On dira qu'un langage L sur $\{0,1\}$ est **décidé par une famille de circuits** si la fonction caractéristique de L est calculable par cette famille, c'est-à-dire si pour chaque longueur de mot il existe un circuit dans la famille considérée décidant si un mot donné est ou non élément de L.

PROPOSITION 1.4.3 *Tout langage (ne contenant pas ε) sur $\{0,1\}$ est décidé par une famille de circuits booléens.*

PREUVE :
Soit $L \subseteq \{0,1\}^*$ et soit n un entier. On note $L_n = L \cup \{0,1\}^n$. Alors pour tout mot w de longueur n il existe un circuit C_w qui identifie w, c'est-à-dire qui avec un mot u en entrée décide si $u = w$. Comme L_n est un ensemble fini de mots, on peut alors rassembler les circuits C_w pour chaque w de L_n par disjonction, obtenant ainsi un circuit C_n qui décide si le mot de longueur n donné en entrée est élément de L_n. En considérant l'ensemble des C_n, on a donc une famille de circuits qui décide L. ■

REMARQUE : Le problème c'est que l'on ne veut pas récupérer les langages non récursifs.

DÉFINITION 1.4.4
On dit qu'une famille (C_n) de circuits est **polynômiale** si il existe p un polynôme tel que pour tout n, la taille de C_n soit inférieure à $p(n)$.

REMARQUE : Là encore il y a un problème car il y a des langages non décidables qui sont décidés par des familles polynômiales de circuits, comme le montre la proposition suivante :

PROPOSITION 1.4.4 *Il existe des langages indécidables qui ont des circuits polynômiaux.*

PREUVE : Soit L un langage indécidable sur $\{0,1\}$. On lui associe le langage U sur $\{1\}$ tel que 1^n soit élément de U si et seulement si n a un développement binaire dans L. U peut bien entendu être considéré comme un langage sur $\{0,1\}$, or il est clairement indécidable. Mais il est tout aussi clair qu'il existe une famille de circuits booléens de taille polynômiale qui décide U, puisqu'à une longueur donnée le circuit doit soit donner la valeur 0 soit vérifier que le mot n'est composé que de 1, et effectuant une conjonction de toutes les entrées. La taille des circuits est de plus clairement linéaire ($n-1$ opérations $\wedge$ pour une entrée de taille n), donc bornée par un polynôme. ■

Cette tentative n'a donc pas fourni le résultat attendu, qui était de retomber sur la calculabilité classique.

DÉFINITION 1.4.5
On dit qu'une famille de circuits (C_n) est **uniforme** si il existe une machine de Turing qui sur l'entrée 1^n sort le codage de C_n.

REMARQUE : Cette fois on récupère bien les fonctions Turing-calculables, mais on n'a plus une notion intrinsèque.

1.5 Automates cellulaires

On ne fera ici qu'une approche succinte. On ne considérera ici que des automates cellulaires sur $\mathbb{Z}$, donc unidimensionnels.

DÉFINITION 1.5.1 (AUTOMATE CELLULAIRE)
Un automate cellulaire est un couple $A = (Q, \delta)$, où Q est un ensemble fini (ensemble d'états) et δ une fonction de Q^3 dans Q. Cette fonction δ permet de définir une fonction G, dite fonction globale associée à l'automate A, de $Q^{\mathbb{Z}}$ dans $Q^{\mathbb{Z}}$, par

$$G(c)(i) = \delta(c(i-1), c(i), c(i+1))$$

pour toute fonction c de $\mathbb{Z}$ dans Q et tout entier relatif i.

Cette définition mérite une explication plus intuitive. L'idée est qu'un automate cellulaire agit sur une *configuration* donnée (la fonction c de la définition) pour produire une nouvelle configuration dans laquelle l'état en un point dépend de l'état du point et de ses voisins dans la configuration précédente. On peut ainsi voir l'automate comme un ensemble de *cellules* disposées sur un espace (infini) de dimension un (dans notre cas, l'extension aux dimensions supérieures étant évidente) dont chaque cellule évolue en fonction de son environnement. Le célèbre *jeu de la vie* est un cas particulier d'automate cellulaire.

On peut donc considérer que ces automates calculent des fonctions ou reconnaissent des langages en ajoutant aux données la spécification d'états d'arrêt, d'acceptation et de rejet, et éventuellement d'états dits *quiescents* qui permettent de limiter l'aire de calcul.

Envisageons plus précisément le calcul d'une fonction f de Σ^* dans Σ^* par un automate cellulaire $A = (Q, \delta, q_{\text{arrêt}}, e)$, avec e état quiescent (on dit qu'un état q est quiescent si $\delta(q, q, q) = q$) et $\Sigma \subseteq Q$. On déclare que le calcul est *terminé* si et dès que la cellule 0, à partir de laquelle on a spécifié l'entrée, entre dans l'état d'arrêt.

PROPOSITION 1.5.1 *La classe des fonctions semi-calculables par les automates cellulaires de dimension 1 est la classe des fonctions récursives.*

PREUVE : Comme toujours, le principe de la preuve est de montrer qu'il est possible de simuler toute machine de Turing par un automate cellulaire et réciproquement. ■

1.6 Complexité en temps et espace des modèles Turing, Markov et RAM

On donne les définitions pour Turing, elles seront similaires pour les autres modèles.

DÉFINITION 1.6.1 (TEMPS)
Soit u une entrée pour une MT $\mathcal{M}$, on désigne par $T_{\mathcal{M}}(u)$ (**complexité en temps sur l'entrée u** le nombre de pas de calcul sur l'entrée u augmenté de $|U|$ lorsque $\mathcal{M}$ s'arrête sur u. Sinon, $T_{\mathcal{M}}(u)$ n'est pas définie.

DÉFINITION 1.6.2 (ESPACE)
Sur l'entrée u, si $\mathcal{M}$ s'arrête, on définit $S_{\mathcal{M}}(u)$ (**complexité en espace**) comme le nombre de cases visées par la tête du calcul.

REMARQUE : On considère selon les cas que l'entrée elle-même est visée, ou bien on ne la compte pas (ex : LOGSPACE).

FAIT 1.6.1 $S_{\mathcal{M}}(u) \leq T_{\mathcal{M}}(u) \leq k_{\mathcal{M}}^{S_{\mathcal{M}}(u)}$.

PREUVE :

- La comparaison entre $S_{\mathcal{M}}$ et $T_{\mathcal{M}}$ est triviale, en effet pour visiter n cases, il faut au moins n pas de calcul.
- Pour la seconde inégalité, si M s'arrête sur le mot w, on peut évaluer le nombre de configurations possibles de la machine M. Une configuration est en effet caractérisée par l'état de la machine, choisi dans l'ensemble Q avec les notations usuelles, le mot écrit sur le ruban, qui est de longueur au plus $S_{\mathcal{M}}(u)$, et la position de la tête de la machine, qui est choisie parmi $S_{\mathcal{M}}(u)+c$ positions possibles où c est une constante. Le nombre de configurations est donc majoré par

  $$|Q| \times |\Gamma|^{S_{\mathcal{M}}(u)} \times (S_{\mathcal{M}}(u) + c)$$

  où Γ est l'alphabet de travail de la machine. On a donc l'ordre de grandeur recherché.

■

Comme l'on cherche à exprimer une hiérarchie entre les machines de Turing (c'est à dire classer des "puissances" indépendantes de l'entrée), on définit :

DÉFINITION 1.6.3
$Time_{\mathcal{M}} : \mathbb{N} \to \mathbb{N}, n \mapsto Time_{\mathcal{M}}(n) = Max\{T_{\mathcal{M}}(u) / |u| = n\}$

DÉFINITION 1.6.4
$Space_{\mathcal{M}} : \mathbb{N} \to \mathbb{N}, n \mapsto Space_{\mathcal{M}}(n) = Max\{S_{\mathcal{M}}(u) / |u| = n\}$

DÉFINITION 1.6.5
La **classe de complexité en temps** (resp. en espace) exprimée par f est l'ensemble des langages décidés par une MT (déterministe ou non) opérant en temps (resp. en espace) f : on note $DTIME(f)$ (resp. $DSPACE(f)$) pour les machines déterministes, $NTIME(f)$ (resp. $NSPACE(f)$) pour les machines non déterministes.

Chapitre 2

Isomorphisme de Rogers

DÉFINITION 2.0.6
Par **système acceptable de programmation** on entend :

1. énumération de $\mathbb{N}$ dans $\mathbb{N}$ incluant toutes les fonctions ppr.
2. existence d'un algorithme universel, c'est-à-dire u tel que :

$$\varphi_u(< i, j >) = \varphi_i(j)$$

3. existence d'un théorème s-n-m :

$$\varphi_i(< x_1, \ldots, x_m, y_1, \ldots, y_n >) = \varphi_{s_n^m(x_1,\ldots x_m)}(< y_1, \ldots y_n >)$$

Une fois cette définition posée, on veut prouver que si $(\varphi_i)_{i\geqslant 0}$ et $(\psi_i)_{i\geqslant 0}$ sont des systèmes de programmation acceptables, alors il existe une fonction f totale récursive telle que pour tout i on ait $\varphi_i = \psi_{f(i)}$ et $\psi_i = \varphi_{f^{-1}(i)}$.

Rappelons tout d'abord le théorème très important :

THÉORÈME 2.0.1 (KLEENE) *Soit $(\varphi)_i$ un sap et f une fonction partielle partiellement récursive. Alors il existe un indice e tel que $\varphi_{f(e)} = \varphi_e$.*

PREUVE : La fonction $\Psi :< x, y >\mapsto \varphi_{f(s_1^1(x,x)}(y)$ est ppr (par composition de fonctions ppr). Donc il existe un indice a tel que $\varphi_{f(s_1^1(x,x)}(y) = \varphi_a(< x, y >)$. En appliquant le théorème s-n-m, on obtient $\varphi_a(< x, y >) = \varphi_{s_1^1(a,x)}(y)$. Alors on prend $a = x$ et $n = s_1^1(< x, x >)$. Remarquer qu'un tel n est effctivement constructible. ■

PROPOSITION 2.0.1 (NON INJECTIVITÉ) *Pour tout indice k, il existe $j \neq k$, $\varphi_j = \varphi_k$.*

PREUVE : On considère la fonction suivante :

$$< x, y >\mapsto \begin{cases} \varphi_k(y) & \text{si } x \neq k \\ \varphi_{x+1}(y) & \text{sinon} \end{cases}$$

Cette fonction appartient au sap, c'est donc $\varphi_a(< x, y >)$. D'après le théorème s-n-m, il existe s_1^1 totale récursive vérifiant :

$$\varphi_a(< x, y >) = \varphi_{s_1^1(a,x)}(y)$$

Si on appelle $f : x \mapsto s_1^1(a,x)$ (a est fixé), et si on applique le théorème de Kleene : il existe un indice e tel que $\varphi_{f(e)} = \varphi_e$, et alors :
- si $e \neq k, j = e$ convient.
- si $e = k, j = e+1$ convient.

■

PROPOSITION 2.0.2 (LEMME DE REMPLISSAGE) *Il existe une fonction récursive ρ telle que :*
- $\forall i, x, x \mapsto \rho(< i, x >)$ *est injective.*
- $\forall i, x, \varphi_i = \varphi_{\rho(<i,x>)}$

PREUVE : Soit i un entier. Supposons que $\rho(\langle i,k\rangle)$ soit défini pour $0 \leqslant k < x$. On cherche alors à définir $\rho(\langle i,x\rangle)$. Considérons à cet effet la fonction

$$\langle i,y,z\rangle \longmapsto \begin{cases} \varphi_{\mathrm{Max}\{\rho(\langle i,k\rangle)\,|\,k<x\}+1}(z) & \text{si} \quad y \in \{\rho(\langle i,k\rangle)\,|\,k<x\} \\ \varphi_i(z) \,\text{sinon} \end{cases}$$

On conviendra facilement que cette fonction est semi-calculable. Elle possède donc un numéro a dans l'énumération (φ_i). Le théorème s_n^m montre alors l'existence d'une fonction s_1^2 totale récursive telle que pour tous i, y et z on ait :

$$\varphi_{s_1^2(a,i,y)}(z) = \varphi_a(\langle i,y,z\rangle)$$

et si l'on pose $f(y) = s_1^2(a,i,y)$, f est une fonction totale récursive. Il existe donc, selon le théorème de Kleene (qui est valable dans tout système de programmation acceptable puisqu'il se déduit de s_n^m), un entier e tel que $\varphi_{f(e)} = \varphi_e$. On a donc

$$\varphi_{f(e)}(z) = \varphi_e(z) = \begin{cases} \varphi_{\mathrm{Max}\{\rho(\langle i,k\rangle)\,|\,k<x\}+1}(z) \text{ si } e \in \{\rho(\langle i,k\rangle)\,|\,k<x\} \\ \varphi_i(z) \quad \text{sinon} \end{cases}$$

Deux cas de figure se présentent alors. Soit $e \notin \{\rho(\langle i,k\rangle)\,|\,k<x\}$, on pose alors $\rho(\langle i,x\rangle) = e$ et on a alors, pour tout z, $\varphi_e(z) = \varphi_{\rho(\langle i,x\rangle)}(z) = \varphi_i(z)$, soit $e \in \{\rho(\langle i,k\rangle)\,|\,k<x\}$, et on pose dans ce cas

$$\rho(\langle i,x\rangle) = \mathrm{Max}\{\rho(\langle i,k\rangle)\,|\,k<x\}+1$$

alors comme e est élément de $\{\rho(\langle i,k\rangle)\,|\,k<x\}$, il existe $k<x$ tel que $e = \rho(\langle i,k\rangle)$, et par conséquent on a à nouveau $\varphi_e(z) = \varphi_{\rho(\langle i,x\rangle)}(z) = \varphi_i(z)$ pour tout z.

Si l'on pose $\rho(\langle i,0\rangle) = i$, on a donc construit ρ par récurrence. ρ est donc bien une fonction récursive totale, et le résultat est démontré. ■

PROPOSITION 2.0.3 *Si (φ_i) et (ψ_i) sont deux sap, alors il existe des fonctions totales récursives g et k, telles que, pour tout i, on ait : $\varphi_i = \psi_{g(i)}$ et $\psi_i = \varphi_{h(i)}$.*

PREUVE :
- (existence de g) D'après la machine universelle, $\exists u, \varphi_i(j) = \varphi_u(< i,j >)$. Dans l'autre sap, il existe un autre indice k vérifiant : $\varphi_u(< i,j >) = \psi_k(< i,j >)$. Si on applique s-n-m dans le système (ψ), on obtient $\psi_k(< i,j >) = \psi_{s_1^1(k,i)}(j)$.
  Donc $\varphi_i(j) = \psi_{s_1^1(k,i)}(j)$. On prend donc pour $g : i \mapsto s_1^1(k,i)$ (k est fixé).

- l'existence de h se montre de façon symétrique.

■

PROPOSITION 2.0.4 *Soient (φ_i) et (ψ_i) deux sap. Alors il existe des fonctions g et h récursives totales telles que :*
- *$g(0) > 0$, $h(0) > 0$.*
- *g et h sont strictement croissantes.*
- *$\varphi_i = \psi_{g(i)}$ et $\psi_i = \varphi_{h(i)}$.*

PREUVE :
- (construction de g) D'après la proposition précédente, il existe une fonction totale récursive t vérifiant :

$$\forall i, \varphi_i = \psi_{t(i)}$$

  Posons :
  - $g(0) = \rho(< t(0), \mathrm{Min}\{y \mid \rho(< t(0), y >) >\})$. On a bien $g(0) > 0$.
  - soit $x > 0$. Soit $m = \mathrm{Max}\{g(0), \ldots, g(x-1)\}$. Si on pose $g(x) = \rho(< t(x), \mathrm{Min}\{y \mid \rho(< t(x), y >) > m\} >)$, alors g vérifie $g(x) > g(x+1)$ et aussi $\varphi_i = \psi_{g(i)}$.
- La construction de h est identique.

■

Ces propositions sont le cheminement vers le :

THÉORÈME 2.0.2 *Soient (φ_i) et (ψ_i) deux sap. Alors il existe une fonction f totale récursive bijective telle que :*

$$\varphi_i = \psi_{f(i)} \text{ et } \psi_i = \varphi_{f^{-1}(i)}.$$

PREUVE : D'après le résultat précédent, il existe deux fonctions g et h récursives totales, strictement croissantes, et telles que $g(0) > 0$, $h(0) > 0$, et $\varphi_i = \psi_{g(i)}$ et $\psi_i = \varphi_{h(i)}$ pour tout i. On va donc définir f à partir de g et h.

Étant donné un entier x, on considère l'ensemble des entiers obtenus à partir de x en appliquant g et h ainsi que leur réciproque un certain nombre de fois. Plus précisément, on pose :

$$\mathcal{C}_x = \left\{ \begin{array}{l} \ldots, h^{-1}\left(\left(g^{-1} \circ h^{-1}\right)^i(x)\right), \left(g^{-1} \circ h^{-1}\right)^i(x), \ldots, h^{-1}(x), x, \\ g(x), h\left(g(x)\right), \ldots, (h \circ g)^i(x), g\left((h \circ g)^i(x)\right), \ldots \end{array} \right\}$$

L'ensemble des entiers qui sont inférieurs à x est fini, or g et h sont strictement croissantes, donc $\mathcal{C}_x$, ordonné selon la définition, a un premier élément qui est soit de la forme $h^{-1}\left(\left(g^{-1} \circ h^{-1}\right)^i(x)\right)$ soit de la forme $\left(g^{-1} \circ h^{-1}\right)^i(x)$. Dans le premier cas, on dit que $\mathcal{C}_x$ termine sur ψ, et dans le second cas on dit qu'il termine sur φ. Dans le premier cas, on pose alors $f(x) = h^{-1}(x)$, et dans le second cas on pose $f(x) = g(x)$.
- Comme g et h sont strictement croissantes, g^{-1} et h^{-1} sont récursives et de domaine récursif, donc f définie ainsi est bien définie et totale récursive.

- Montrons à présent que f est bijective. Pour l'injectivité, considérons x et y tels que $f(x) = f(y)$. Étant donnée la définition de f, on a forcément $f(x) = g(x)$ ou $f(x) = h^{-1}(x)$ et de même pour $f(y)$. Par conséquent, soit $f(x)$ et $f(y)$ sont de la même forme, auquel cas on a $x = y$ par injectivité de g et h, soit on a $g(x) = h^{-1}(y)$ (ou le cas symétrique). Mais dans ce cas $g^{-1}\left(h^{-1}(x)\right)$ est défini (et vaut x), ce qui contredit la définition de f car $h^{-1}(x)$ n'est pas le premier élément de $\mathcal{C}_x$. L'injectivité est donc démontrée.
- Pour la surjectivité, considérons un entier x donné. Soit $g^{-1}(x)$ est défini, soit il ne l'est pas. S'il est défini, on considère $\mathcal{C}_x$: s'il s'arrête sur φ, on a $f(x) = g(x)$ mais aussi $f\left(g^{-1}(x)\right) = g\left(g^{-1}(x)\right) = x$ car $\mathcal{C}_x = \mathcal{C}_{g^{-1}(x)}$, donc x est atteint. Au contraire, si x n'a pas d'antécédent par g, on a forcément $f(h(x)) = h^{-1}(h(x)) = x$ car $\mathcal{C}_x = \mathcal{C}_{h(x)}$ s'arrête sur ψ. x est donc atteint dans ce cas également, donc f est bien surjective, et le théorème est démontré.

■

Chapitre 3

Classes de complexité Turing en temps

3.1 Théorèmes de compression et d'accélération linéaire

THÉORÈME 3.1.1 (COMPRESSION) *Si $c \in \mathbb{R}^{+*}$, si s est une fonction $\mathbb{N} \to \mathbb{N}$, alors $SPACE(s(m)) = SPACE(c.s(m))$.*

PREUVE : Sans perdre de généralité, on peut prendre $c < 1$.

- Trivialement, $SPACE\big(c.s(m)\big) \subseteq SPACE\big(s(m)\big)$
- Pour l'autre inclusion, posons $k = \lceil \frac{1}{c} \rceil$. Soit $L \in SPACE\big(s(m)\big)$. Il existe donc une machine de Turing $\mathcal{M}$ telle que $SPACE_{\mathcal{M}}(m) \leq s(m)$, par définition. On peut alors construire une machine de Turing $\mathcal{M}'$ qui va décider L en espace $c\big(s(m)\big)$ de la façon suivante :
  - Chaque case de $\mathcal{M}'$ "représente" k cases de la machine $\mathcal{M}$.
  - Donc la machine $\mathcal{M}'$ a un autre alphabet de travail que $\mathcal{M}$ (ce sont des triplets de lettres). La fonction de transition de $\mathcal{M}'$ se déduit facilement de celle de $\mathcal{M}$

  On se convainc facilement que la machine $\mathcal{M}'$ calcule L et par construction en temps $\dfrac{s(m)}{k} \simeq c.s(n)$

■

THÉORÈME 3.1.2 (ACCÉLÉRATION LINÉAIRE) *Soit $c \in \mathbb{R}^{+*}$. Si L est décidé par une MT (à 2 rubans au moins), en temps $t(n)$ et si $Lim \inf_{n\to\infty} \dfrac{t(n)}{n} = \infty$, alors L est décidé par une machine en temps $c.t(n)$.*

REMARQUE : $Lim \inf_{n\to\infty} \dfrac{t(n)}{n}$ désigne $\lim_{n\to+\infty} \mathrm{Inf}\{\dfrac{t(n)}{n}, \dfrac{t(n+1)}{n+1}, \ldots\}$ PREUVE : Soit $\mathcal{M}$ une machine de Turing qui décide L en temps $t(n)$. On considère la machine $\mathcal{M}_2$ qui fonctionne de la manière suivante :

- sur l'entrée u donnée sur le ruban 1,
- elle écrit sur un second ruban cette entrée u comprimée, en prenant dans une même case m caractères de $\mathcal{M}$.

– on vérifie qu'en 8 pas de calcul (mémoire : GDDG, écriture : GDDG), $\mathcal{M}_2$ réalise la simulation de m pas de calcul de $\mathcal{M}$ donc le temps du calcul en lui même est $8\lceil\frac{t(n)}{m}\rceil$.

En tout, la machine $\mathcal{M}_2$ a décidé L en un temps :

$$T = n + \lceil\frac{t(n)}{m}\rceil + 8\lceil\frac{t(n)}{m}\rceil$$

Soit $T \leq n + 8\frac{t(n)}{m} + \frac{t(n)}{m} + 2$. Or l'hypothèse $Lim \underset{n\to\infty}{\text{Inf}} \frac{t(n)}{n} = +\infty$ assure l'existence de $n_d \in \mathbb{N}, \forall n \geq n_d, \frac{t(n)}{n} \geq d$. Alors, pour $n \geq \text{Max}\{2, n_d\}$, on obtient :

$$T \leq t(n).\Big(\frac{2}{d} + \frac{8}{m} + \frac{1}{m.d}\Big).$$

Aors, si m vérifie $cm \geq 16$, donc peut trouver d qui nous permet d'affirmer que :

$$T \leq c.t(n)$$

■

REMARQUE : Ces théorèmes permettent de définir la classe de complexité d'une fonction f comme l'ensemble des langages ayant une complexité temporelle en $O(f)$.

3.2 Classe P

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

Notion de réduction polynômiale :

DÉFINITION 3.2.1
Si A et B sont deux langages sur Σ, on dit que A **se réduit polynomialement** en B et on écrit $A \leq B$ sit il existe une fonction totale récursive $f : \Sigma^* \to \Sigma^*$ calculable en temps poynomiale et telle que $u \in A \Leftrightarrow f(u) \in B$.

FAIT 3.2.1 $\leq$ *est un préordre (relation réflexive et transitive).*

PREUVE : Triviale, non ? ■

FAIT 3.2.2 *Si A et B sont deux langages de P, non vides et distincts de Σ^*, alors $A \leq B$ et $B \leq A$.*

PREUVE :

- $(A \leq B)$ Comme $B \neq \emptyset$, il existe $\omega_B \in B$ et comme $B \neq \Sigma^*$, il existe $\omega'_B \notin B$. Idem pour B. Alors la fonction $\Sigma^* \to \Sigma^*$, $u \in A \to \omega_B$ et $u \notin A \to \omega'_B$ est totale récursive et calculable en temps polynomiale et permet d'affirmer que $A \leq B$.
- $(B \leq A)$ Raisonnement identique.

■

DÉFINITION 3.2.2
Si $A \leq B$ et $B \leq A$, on note $B \sim A$ et on dit que A et B sont **équivalents** pour la réduction polynomiale.

3.3 Classe NP

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

C'est la classe des langages décidés en temps polynomial par une machine non déterministe de complexité polynomiale.

DÉFINITION 3.3.1
Une **machine de Turing non déterministe** se construit de la même façon qu'une MT classique, sauf que la fonction de transition choisit la lettre et le nouvel état dans un ensemble fini.

PROPOSITION 3.3.1 *Toute machine de Turing non déterministe M (temps $= t(n)$) peut être simulée par une machine déterministe M' (temps $2^{O(t(n))}$).*

PREUVE :
- (existence d'une MTD équivalente) Soit $\mathcal{M}$ une machine de Turing non déterministe. On utilise un parcours en largeur de son arbre de calcul. Désignons par r le plus grand nombre de transitions possibles à partir d'un état (q, a). La machine $\mathcal{M}'$ déterministe associée va avoir 3 rubans :
  - on met l'entrée sur le premier ruban,
  - sur le deuxième ruban, on énumère les mots sur $[\![1, r]\!]$ (ils représentent les chemins de taille 1, puis les chemins de taille 2, ...), dans l'ordre hiérarchique par exemple,
  - sur le troisième ruban, $\mathcal{M}'$ va exécuter le calcul de $\mathcal{M}$ sur l'entrée, suivant le chemin qui est indiqué par le mot su ruban 2 :
    - Si $\mathcal{M}'$ réalise un calcul acceptant de $\mathcal{M}$, elle s'arrête en acceptant le mot.
    - Si $\mathcal{M}'$ réalise un calcul de refus de $\mathcal{M}$, elle s'arrête en refusant le mot.
    - Si le calcul sur le troisième ruban ne mène pas à une configuration d'arrêt de $\mathcal{M}$, la machine $\mathcal{M}'$ recommence sur le mot suivant de l'énumération.

  On se convainc facilement que $\mathcal{M}'$ réalise le même travail que la machine $\mathcal{M}$ et qu'elle est déterministe.
- (temps de calcul) Le temps de calcul de $\mathcal{M}'$ est $t(n).r^{t(n)}$ (temps de calcul d'un chemin * nombre de chemins possibles), soit $2^{O(t(n))}$.

■

PROPOSITION 3.3.2 *$L \in NP$ ssi il existe un langage $L' \in P$, et un polynôme p, tels que : $L = \{x / \exists y, < x, y > \in L'$ et $|y| \leq p(|x|)\}$.*

PREUVE :
- $\Leftarrow$: Si L' et p existent, L défini par l'énoncé est dans NP. En effet, on peut considérer la machine non déterministe qui représente l'algorithme suivant :

- sur l'entrée u,
- on devine un mot y dans l'ensemble des u tel que $|u| \leq p(|x|)$,
- on cherche si $< x, y > \in L'$. Si oui, on accepte le mot, si non on le rejette.

Cet algorithme décide L.

- $\Rightarrow$: Si $L \in NP$, il existe une machine de Turing non déterministe qui décide L en temps polynomial donc un polynôme p tel que sur une entrée de longueur n, la machine ne fait que des calculs en temps $\leq p(n)$. Alors On définit L' de la façon suivante : $< x, y > \in L'$ ssi y code un calcul de la machine précédente sur l'entrée x. Alors $L' \in P$ et $L = \{x / \exists y, < x, y > \in L' \text{ et } |y| \leq p(|x|)\}$.

■

DÉFINITION 3.3.2
L est un **langage NP-complet** ssi :

1. $L \in NP$
2. $\forall L' \in NP, L' \leq L'$ au sens de la réduction polynomiale.

FAIT 3.3.1 *Si L_1 et L_2 sont NP-complets, alors $L_1 \sim L_2$.*

DÉFINITION 3.3.3
L est un langage **$\mathcal{C}$-complet** ssi :

1. $L \in \mathcal{C}$
2. $\forall L' \in C, L' \leq L'$ au sens de la réduction polynomiale.

Voici des exemples :

DÉFINITION 3.3.4
PfP est défini par : On se donne :

- Un ensemble fini C de couleurs qui contient une couleur particulière, la couleur blanche ;
- une collection de tuiles τ (contenue dans C^4) qui contient une tuile particulière, la tuile blanche ;
- un entier $n \leq |C|$.

Alors existe-t-il un pavage fini, valide (les couleurs identiques se touchent) du plan, non trivial, de taille inférieure à $n * n$?

PROPOSITION 3.3.3 *$PfP \in NPc$.*

REMARQUE : On obtient ce résultat directement à partir des machines de Turing, en obtenant des tuiles de couleur à partir des états, et un pavage à partir des configurations successives de la machine.

DÉFINITION 3.3.5
Soit une machine déterministe $\mathcal{M}$, un entier t tel que $t \leq |Q_{\mathcal{M}}|$. Existe-t-il un calcul acceptant de $\mathcal{M}$ sur l'entrée ε, en temps $\leq t$? On note L_{NDTM} le langage qui code ce problème.

PROPOSITION 3.3.4 *L_{NDTM} est dans NPc.*

PREUVE :

- $L_{NDTM} \in NP$ en considérant l'algorithme suivant : sur l'entrée $< \mathcal{M}, t >$
  - décode en $\mathcal{M}$ et t,
  - simule $\mathcal{M}$ sur ε pour un temps $\leq t$,
  - si $\mathcal{M}$ converge en ce temps, on accepte, sinon, on rejette.

  Cet algorithme est non déterministe et polynomial.
- Soit $L \in NP$. On va montrer que $L \leq L_{NDTM}$. Comme L est dans NP, il existe une machine de Turing non déterministe $\mathcal{M}_L$ et un polynôme p tel que $\mathcal{M}_L$ décide L en temps $p(n)$. On peut alors obtenir une machine de Turing non déterministe $\mathcal{M}'_L$ qui converge sur u ssi $u \in L$ et diverge sinon. Pour cela, il suffit de transformer l'état de rejet de $\mathcal{M}_L$ en un état "banal" et d'ajouter les transitions qui assurent la divergence.

  Construction :
  - (machine $\mathcal{M}'_{L,\omega}$) : sur l'entrée ε, on écrit ω sur un ruban de travail, et on simule $\mathcal{M}'(\omega)$.
  - (machine $\mathcal{M}''_{L,\omega}$) : à partir de $\mathcal{M}'_{L,\omega}$, on ajoute à l'ensemble des états de celle-ci autant d'états qu'il faut pour qu'il soit supérieur ou égal à $2\,|\omega| + p(|\omega|)$.

  Alors $\omega \in \Sigma^* \mapsto < \mathcal{M}''_{L,\omega}, 2\,|\omega| + p(|\omega|) >$ est une réduction polynomiale de L à L_{NDTM}.

■

3.4 $NP \cap CoNP$

DÉFINITION 3.4.1
L_{prem} est le langage qui code le problème :
- n entier.
- n est-t-il premier ?

PROPOSITION 3.4.1 *$L_{prem} \in NP \cap CoNP$.*

PREUVE : Trop longue ! ■

REMARQUE : On ne sait toujours pas si $L_{prem} \in P$ ou si $L_{prem} \in NPc$.

3.5 Langages P-complets

Les langages de P sont équivalents modulo la relation d'ordre induite par $\leq$. Pour établir une hiérarchie et parler de langage P-complet, on a besoin de la notion de réduction en espace logarithmique :

DÉFINITION 3.5.1
$A \leq_L B$ (A se **réduit logarithmiquement en** B) ssi il existe une fonction f totale calculable en espace logarithmique de Σ^* dans Σ^* telle que $u \in A \Leftrightarrow f(u) \in B$.

PROPOSITION 3.5.1 *$\leq_L$ est un préordre.*

PREUVE : On prend ici des MT à trois rubans (écriture, travail, lecture). La complexité est celle du ruban de travail.

- (réflexivité) prendre $f = \mathrm{Id}$.
- (transitivité) Supposons $L_1 \leq L_2$ à l'aide de la fonction f_1 et $L_2 \leq L_3$ à l'aide de la fonction L_2. Le principal problème est que l'on ne peut pas brutalement prendre la composée de ces deux focntions. On va construire $\mathcal{M}$ qui prouve cette transitivité de la façon suivante : elle a un compteur qui va permettre d'atteindre le i-ème caractère de $f_1(w)$ dont elle a besoin. $\mathcal{M}$ met son compteur à i, elle simule $\mathcal{M}_1$ sur l'entrée w pas à pas en décrémentant le compteur à chaque fois que $\mathcal{M}_1$ obtient un caractère de $f_1(w)$, et simule alors $\mathcal{M}_2$ sur le caractère en question. Comme f_1 est calculée en espace logarithmique, la machine $\mathcal{M}$ est aussi en espace logarithmique et prouve que $L_1 \leq L_3$.

■

DÉFINITION 3.5.2
L est **P-complet** si $L \in P$ et tout langage L' s'y réduit en espace logarithmique : $L' \leq_L L$.

Il existe des langages P-complets :

EXEMPLE 3.5.1 *On se donne X un ensemble fini, R une relation ($R \subseteq X \times X \times X$), et X_s, X_t deux sous-ensembles de X. La question posée est : X_t contient-t-il au moins un élément du plus petit ensemble $A \subseteq X$ tel que :*
- *$X_s \subseteq A$;*
- *si $y, z \in A$ et $< x, y, z > \in R$, alors $x \in A$.*

Le langage codant ce problème est SPS.

PROPOSITION 3.5.2 *SPS est P-complet.*

PREUVE : Voir en TD ! ■

Chapitre 4

Classes de complexité Turing en espace

4.1 Définitions

On considère les machines à 1 ruban d'entrée de pure lecture et l'espace est évalué sur les rubans de travail.

DÉFINITION 4.1.1
Etant donnée une MT $\mathcal{M}$ de caractéristiques ci-dessus, sa **complexité en espace** est le nombre de cases visitées au moins un fois par la tête.

DÉFINITION 4.1.2
Une machine $\mathcal{M}$ est de **complexité en espace bornée par** $f : \mathbb{N} \to \mathbb{N}$ si pour (presque-) tout n, $SPACE_{\mathcal{M}}(n) \leq f(n)$ (ou $O(f(n))$.

DÉFINITION 4.1.3
On dénote :
- $SPACE(f(n)) = \{L/\exists \mathcal{M}$ déterministe qui décide L et de complexité $f(n)\}$
- $NSPACE(f(n)) = \{L/\exists \mathcal{M}$ non déterministe qui décide L et de complexité $f(n)\}$
- d'où $LOGSPACE, NLOGSPACE \ldots$
- $PSPACE = \bigcup_{k \geq 0} PSPACE(n^k)$.
- $NSPACE = \bigcup_{k \geq 0} NSPACE(n^k)$.

Quelques exemples :

EXEMPLE 4.1.1 *SAT peut être décidé en espace $O(n)$, appartient donc à $PSPACE$. En effet, l'algorithme qui consiste à énumérer toutes les distributions est linéaire en espace.*

DÉFINITION 4.1.4
Soit G un graphe à n sommets. x, y sont deux sommets, existe-t-il un chemin entre x et y ? L_{att} est le langage codant ce problème. Les données sont les

sommets de 1 à n, la matrice du graphe représentée par ses lignes successives (mots sur $\{0,1\}$ de longueur n), et les deux sommets x et y.

PROPOSITION 4.1.1 $L_{att} \in SPACE(log(n)^2)$.

PREUVE : Soit le prédicat $Chem(x,y,i)$ qui est staisfait ssi il existe un chemin de longueur $\leq 2^i$ de x à y dans G. Le nombre de sommets de G étant n, la longueur d'un chemin dans G est inférieure ou égale à n. Décider si un chemin de x à y existe revient à évaluer $Chem(x,y,\lceil \log(n) \rceil)$. On va construire une machine de Turing qui permet d'évaluer cette quantité :

- sur le ruban d'entrée (lecture), on met $< G,x,y >$
- sur les 2° et 3° rubans (calcul) : sur le ruban 2 on met (x,y,i) où i est un entier. Chaque composante est codée en binaire. Sur le ruban 3 on énumère les sommets de G.

Remarquons que $Chem(x,y,i) \Leftrightarrow \exists z, Chem(x,z,i-1) \wedge Chem(z,y,i-1)$. Que va donc faire la machine ?

- si $i = 0$, il suffit de vérifier que $x = y$ ou qu'il existe une arête entre x et y.
- si $i \geq 1$ on calcule récursivement $Chem(x,y,i)$ en testant pour tout z si $Chem(x,z,i-1)$ et $Chem(z,y,i-1)$ sont satisfaites. Pour ce faire :
  - si une réponse négative est donnée, on efface $(x,z,i-1)$ du ruban et on passe à l'examen du sommet z suivant.
  - si la réponse est positive, on efface $(x,z,i-1)$ du ruban 2 que l'on remplace par $(z,y,-i-1)$. On évalue alors $Chem(z,y,i-1)$, si faux : on passe au z suivant, si vrai, on vérifie sur le ruban 2 qu'il s'agit du deuxième élément et on obtient $Chem(x,y,\lceil \log_2(n) \rceil)$.

La complexité est bien en $O(\log(n)^2)$. ■

4.2 Théorème de Savitch

THÉORÈME 4.2.1 *Soit $f : \mathbb{N} \to \mathbb{N}$ complétement constructible en espace et telle que $f(n) \geq log(n)$. Alors :*

$$NSPACE(f(n)) \subseteq SPACE(f(n)^2).$$

REMARQUE : "Complétement contructible en espace" signifie qu'il existe une machine de Turing qui sur toute entrée n occupe exactement $f(n)$ cases.
PREUVE : Soit L un langage de $NSPACE(f(n))$. Alors il existe une machine de Turing qui le décide en espace $f(n)$. Considérons le graphe des configurations de cette machine sur une entrée u de taille n. Le nombre de sommets de ce graphe est $\leq 2^{c.f(n)}$ pour une certaine constante c. On obtient le résultat en appliquant le résultat précédent sur le graphe en question avec $x = \omega$ (l'entrée) et $y =$arrêt. ■

COROLLAIRE 4.2.1 $NSPACE = PSPACE$.

A ce stade, on a :

$$LOGSPACE \subseteq NLOGSPACE \subseteq P \subseteq NP \subseteq PSPACE = NSPACE$$

4.3 Il existe des langages PSPACE-complets

DÉFINITION 4.3.1
A est $PSPACE$-complet si $A \in PSPACE$ et tout autre langage de $PSPACE$ s'y réduit polynomialement en temps.

DÉFINITION 4.3.2
Une formule QBF (totally quantified boolean formula) est une formule du type $Q_1x_1Q_2x_2 \ldots Q_nx_n\Phi(x_1, \ldots, x_n)$. Les $Q_i \in \{\exists, \forall\}$ et les variables de Φ sont toutes dans les champs des quantificateurs. $TQBF$ est le langage des fQBF "vraies".

PROPOSITION 4.3.1 *$TQBF$ est un langage $PSPACE$-complet.*

PREUVE :

- $TQBF \in PSPACE$: on considère l'algorithme suivant (récursif) :
  - sur l'entrée $< \Phi >$,
  - si Φ n'est pas quantifiée, elle n'est consituée que de constantes, on l'évalue et on accepte ou rejette selon le cas.
  - sinon : si $\Phi = \exists x\Psi$, on utilise la procédure successivement sur Ψ dans laquelle on subsitue 1 ou 0 à x. Si l'un des résultats est vrai, on répond "satisfiable" et sinon on répond "non staisfiable".
  - idem si $\Phi = \forall x, \Psi$.
- Tout langage $PSPACE$ se réduit en $TQBF$: on cherche une fonction f totale, calculable en temps polynomial qui associe à tout mot ω une formule Φ de sorte que $\omega \in L \Leftrightarrow f(\omega) = \Phi \in TQBF$. L est décidé en espace polynômial donc il existe k et une machine de Turing de complexité en espace n^k qui décide L.
  Soit $\omega \in \Sigma^*$. Alors $\Phi_{c_0, c_{arret}, 2^{c.n^k}}$ est une formule qui exprime que ω (qui définit c_0), est reconnue ou non en temps inférieur ou égal à $c.n^k$. Cette formule est équivalente à

$$\exists c_0, \exists c_{arret}, \left(\Phi'_{c_0, c_{arret}, 2^{c.n^k}} \wedge c_0 \text{ config.ini } \wedge c_{arret} \text{ config. arrêt } \right).$$

  Alors on a l'équivalence $\omega \in L \Leftrightarrow \Phi_{c_0, c_{arret}, 2^{c.n^k}}$.
  Considérons la formule $\Phi'(c_1, c_2, t)$ vraie ssi il existe un calcul dans $\mathcal{M}$ qui mène de c_1 à c_2 en au plus t pas de calcul. Alors si on substitue à $\Phi'(c_1, c_2)$ la formule :

$$\exists n, \forall c_3, \forall c_4, \left((c_3 = c_1 \wedge c_4 = n) \vee (c_3 = n \wedge c_4 = c_2) \Rightarrow \Phi'_{c_3, c_4, \lceil \frac{t}{2} \rceil} \right),$$

  qui lui est équivalente, à chaque étape de la récursion on obtient des quantités en $O(n^k)$ en espace. En admettant que la première $\Phi_{\_,\_,\_}$ ne soit pas trop grande, on obtient la réduction polynômiale souhaitée.

■

Chapitre 5

Relations entre classes de complexité

5.1 Quelques relations

THÉORÈME 5.1.1 *On a les relations générales suivantes :*

1. *Si* $L \in TIME\big(f(n)\big)$ *alors* $L \in SPACE\big(f(n)\big)$.
2. *Si* $L \in SPACE\big(f(n)\big)$, *si* $f(n) \geq log(n)$ *et si* f *est totalement constructible en espace, alors il existe une constante ne dépendant que de* L *telle que* $L \in TIME\big(c^{f(n)}\big)$.
3. *Si* $L \in NTIME\big(f(n)\big)$ *alors il existe une constante* $c_L > 0$ *telle que* $L \in TIME\big(c^{f(n)}\big)$.

PREUVE :

1. Si $L \in TIME\big(f(n)\big)$, il existe une machine de Turing déterministe qui décide L en temps borné par $f(n)$. L'espace utilisé sur une entrée de longueur n est au plus $f(n)+1$. Le résultat est donné par le théorème de compression.
2. Si $L \in SPACE\big(f(n)\big)$, on évalue le nombre de configurations distinctes possibles à partir d'une entrée :
   ce nombre $N \leq |Q|.(n+1).f(n).|\Gamma|^{f(n)}$ ($|\Gamma|^{f(n)}$ représente le nombre de mots possibles). Alors il existe une constante c, telle que $N \leq c^{f(n)}$. On construit alors la machine $\mathcal{M}'$ qui sur l'entrée x de longueur n, calcule $A(n) = c^{f(n)}$ (c'est possible car f est constructible en espace). Elle simule la machine qui décide L en espace $f(n)$ pour un temps inférieur ou égal à $c^{f(n)}$. Elle accepte ssi $\mathcal{M}$ accepte dans cette limite de temps.
   On obtient cette fois un arbre de calculs de la machine déterministe qui décide $L \in NPSPACE\big(f(n)\big)$, le nombre de chemins est inférieur ou égal à $r^{(c^{f(n)})}$, ce qui est trop. On supprime alors les configurations qui apparaissent plusieurs fois dans l'arbre. Les configurations différentes sont en nombre inférieur ou égal à $c^{f(n)}$, soit ce qu'il faut.

■

5.2 Il n'existe pas de classe "maximum"

THÉORÈME 5.2.1 *Soit f une fonction totale calculable. Il existe un langage récursif L qui n'appartient pas à $TIME(f)$ et il existe un langage récursif L' qui n'appartient pas à $SPACE(f)$.*

PREUVE :

- On se penche sur le cas de $TIME(f(n))$. Il suffit pour prouver le résultat d'exhiber un tel langage, et on procède naturellement par diagonalisation. On considère donc une énumération (M_i) des machines de Turing et une énumération (w_i) des mots sur $\{0,1\}$ donnée par l'ordre hiérarchique. Posons alors

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

  Le langage L est alors récursif, puisqu'il suffit pour déterminer si w_i appartient à L de simuler le fonctionnement de M_i sur w_i pendant au plus $f(|w_i|)$ pas. Supposons alors qu'il existe une machine de turing qui décide L en temps $f(n)$, et appelons-la M_{i_0}. On considère alors w_{i_0} :
  - si w_{i_0} est dans L, son appartenance est décidée par M_{i_0} en temps $f(|w_{i_0}|)$, donc il n'est pas élément de L par définition de L ;
  - si w_{i_0} n'est pas dans L, son appartenance est n'est alor pas décidée par M_{i_0} en temps $f(|w_{i_0}|)$, donc il est élément de L par définition de L.

  On aboutit donc à une contradiction dans les deux cas, ce qui infirme notre hypothèse. Le langage L n'est donc pas décidable en temps $f(n)$.
- Le cas $SPACE(f(n))$ se résout de façon identique.

■

5.3 Hiérarchie déterministe en espace

THÉORÈME 5.3.1 *Soit $s_2(n)$ une fonction totalement constructible en espace, et $s_1(n)$ une fonction telle que $s_2(n) \geq s_1(n) \geq log(n)$ et $Lim \inf_{n\to\infty} \frac{s_1(n)}{s_2(n)} = 0$, alors :*

$$SPACE\big(s_1(n)\big) \subsetneq SPACE\big(s_2(n)\big)$$

PREUVE : Soit $L \in SPACE\big(s_1(n)\big)$, L est décidé par une machine à plusieurs rubans en $s_1(n)$, par une machine à 1 ruban $\mathcal{M}$en $c_1.s_1(n)$.
On considère la machine $\mathcal{M}'$ qui sur l'entrée ω de taille n, réalise ce qui suit :

- $\mathcal{M}$ commence par marquer $s_2(n)$ cases (possible car s_2 est complétement constructible en espace),
- dans ce qui suit, si $\mathcal{M}'$ doit franchir la limite de ces cases marquées, elle s'arrête en rejetant l'entrée ω,
- $\mathcal{M}'$ simule $\mathcal{M}$ sur l'entrée ω, donc travaille en $c_1.s_1(n)$.
- $\mathcal{M}'$ accepte ω ssi elle réalise sa simulation en $s_2(n)$ espace et si $\mathcal{M}_\omega$ s'arrête en refusant ω.

Alors le langage décidé par $\mathcal{M}'$ vérifie :

- $L(\mathcal{M}') \in SPACE\big(s_2(n)\big)$ trivialement,

– $L(\mathcal{M}') \notin SPACE\big(s_1(n)\big)$. En effet, si $L(\mathcal{M}') \in SPACE\big(s_1(n)\big)$, il existeraît une machine $\hat{M}$ qui le décide en espace $s_1(n)$. Mais il existe certainement un mot ω de taille n vérifiant $c_1.s_1(n) < s_2(n)$ et $\hat{M} = \mathcal{M}_\omega$. Alors sur l'entrée ω, $\mathcal{M}'$ a suffisamment d'espace pour simuler $\mathcal{M}_\omega$. Or $\mathcal{M}'$ accepte ssi $\mathcal{M}_\omega$ refuse. Contradiction. Donc $L(\mathcal{M}') \neq L(\mathcal{M}_\omega)$

■

5.4 Hiérarchie déterministe en temps

THÉORÈME 5.4.1 *Soient $t_1(n)$ et $t_2(n)$ des fonctions $\mathbb{N} \to N$ telles que $t_2(n) \geq t_1(n) \geq n$ et aussi telles que $Lim \inf_{n\to\infty} \dfrac{t_1(n)\log(t_1(n))}{t_2(n)} = 0$. Alors :*

$$TIME\big(t_1(n)\big) \subsetneq TIME\big(t_2(n)\big)$$

PREUVE : Voir en TD ! ■

DÉFINITION 5.4.1
Une fonction "**super polynomiale**" est une fonction non décroissante de $\mathbb{N}$ dans $\mathbb{N}$ telle que :

$$Lim \inf_{n\to\infty} \frac{n^k}{f(n)} = 0$$

En corollaire au théorème précédent, on obtient :

PROPOSITION 5.4.1 *Pour toute fonction f super polynômiale, on a :*
- $P \subsetneq TIME\big(f(n)\big)$
- $PSPACE \subsetneq SPACE\big(f(n)\big)$

On prouve par ailleurs que $EXPTIME \neq PSPACE$, et finalement :

PROPOSITION 5.4.2
- $P \subsetneq EXPTIME = \bigcup_{k>0} TIME\big(2^{kn}\big)$
- $LOGSPACE \subsetneq PSPACE \subsetneq EXPTIME$

PREUVE : Pour prouver $EXPTIME \neq PSPACE$, on utilise $EXPTIME \subseteq TIME\big(2^{(n^{3/2})}\big) \subsetneq TIME\big(2^{n^2}\big)$ d'après précédemment. En raisonnant par l'absurde, on prouve ensuite $TIME\big(2^{n^2}\big) \subseteq PSPACE$ grâce au langage $L' = \{x\$^t \mid x \in L \text{ et } |x| + t = |x|^2\}$ si $L \in TIME\big(2^{n^2}\big)$. ■

Chapitre 6

MT avec oracle, hiérarchie polynomiale

6.1 Machine de Turing avec oracle

DÉFINITION 6.1.1 (MT AVEC ORACLE)
C'est une machine de Turing (déterministe ou non), à laquelle on ajoute 3 états : $q_?$ qui sert à interroger l'oracle, q_{oui} et q_{non} qui représenteront la réponse de l'oracle. On se donne également un langage A auquel sera réservé un ruban de la machine. L'oracle dira "oui" si le mot de ce ruban à l'instant de la question est dans le langage A, il dit "non" sinon (et ceci en 1 seul pas de calcul !)

NOTATION : on note $L(\mathcal{M}, A)$ ou $L(\mathcal{M}^A)$ le langage reconnu par la machine $\mathcal{M}$ avec A comme oracle.

Comme langage d'oracle, on prend uniquement les *langages récursifs*, sinon on pourrait récupérer des langages non récursifs, ce qui n'est pas raisonnable !

6.2 $P = NP$ ou $P \neq NP$?

THÉORÈME 6.2.1 *Il existe un oracle A tel que $P^A = NP^A$*

PREUVE : Pour la preuve, on utilise un langage $PSPACE$ complet, et on utilise le fait que $NSPACE = PSPACE$.
Soit A un langage $PSPACE$-complet. On va alors montrer que l'on a les inclusions suivantes :

$$PSPACE \subseteq P^A \subseteq NP^A \subseteq NPSPACE = PSPACE$$

- Pour montrer que $PSPACE$ est inclus dans P^A, considérons un langage L de $PSPACE$. On a alors $L \leqslant_P A$ par définition de A. Soit alors la machine qui sur un entrée w
  1. calcule $f(w)$ où f rend compte de $L \leqslant_P A$;
  2. interroge A sur $f(w)$;
  3. accepte si et seulement si l'oracle dit oui.

Alors cette machine décide L en temps polynômial.
- L'inclusion $P^A \subseteq NP^A$ est triviale.
- Considérons enfin un langage L dans NP^A. Il existe alors une machine de Turing on déterministe M qui décide L avec l'oracle A en temps polynômial. On considère alors la machine qui sur l'entrée w :
  1. simule M sur w jusqu'à ce que M entre dans l'état $q_?$;
  2. décide l'interrogation en espace polynômial, ce qui est autorisé par le fait que A est dans $PSPACE$;
  3. termine le fonctionnement de M.

  Cette machine décide bien L en espace polynômial non déterministe donc la troisième inclusion est démontrée.

■

THÉORÈME 6.2.2 *Il existe un oracle B tel que $P^B \neq NP^B$.*

PREUVE : On commence par remarquer qu'il existe une énumération (T_k) des machines de Turing dont la complexité en temps est bornée par $n^k + k$.

ça mérite réflexion...

De plus, $n \mapsto n^k + k$ est constructible en temps.

Soit X un langage sur $\{0,1\}$. On note alors

$$L_X = \{0^n \,|\, \exists x \in X, |x| = n\}$$

Alors clairement L_X est dans NP^X.

On va alors exhiber un langage B tel que $L_B \notin P^B$. On le construit comme la réunion d'une famille de langages B_i.

1. On pose $B_0 = \emptyset$ et $n(0) = 1$.
2. Pour un i donné, on pose

$$n(i+1) = \text{Min}\left\{m \,\middle|\, n(i)^i + i < m,\, m^{i+1} + i + 1 < 2^m\right\}$$

On simule T_{i+1} avec l'oracle B_i sur $0^{n(i+1)^{i+1}}$.
- s'il est accepté, on pose $B_{i+1} = B_i$;
- sinon, on pose $B_{i+1} = B_i \cup \{y\}$ où y est le plus petit mot sur $\{0,1\}$ de longueur $n(i+1)$ qui n'est pas interrogé lors du calcul de T_{i+1} sur $0^{n(i+1)}$. y existe car le nombre de mots de longueur $n(i+1)$ est $2^{n(i+1)}$.

Donc $B = \cup_{k \geq 0} B_k$ est bien défini. On remarque de plus (preuve par récurrence) :
- si $0^{n(i)}$ est accepté par T_i avec B_{i-1}, alors B_i ne contient pas de mot de longueur $n(i)$.
- si $0^{n(i)}$ est refusé, B_i contient exactement 1 mot de longueur $n(i)$.

Soit ensuite $L_B = \{0^n \mid \exists x \in B, |x| = n\}$, $L_B \in NP^B$. Mais $L_N \notin P^B$. En effet, si L_B appartenait à P^B, il existerait une machine T_k telle que $L_B = L(T_k^B)$ Alors :
- si $0^{n(k)} \in L_B$ alors $\exists x \in B, |x| = n(k)$, alors $0^{n(k)}$ est accepté par T_k, alors $B_k = B_{k-1}$ et B_{k-1} ne contient pas de mot de longueur $n(k)$! problème

– si $0^{n(k)} \notin L_B$ alors il n'est pas reconnu par T_k alors $B_k = B_{k-1} \cup \{y\}$ avec $|y| = n(k)$. Donc on a à la fois $y \in B$ et $|y| = n(k)$ ce qui est contradictoire avec $0^{n(k)} \notin L_B$.

Donc finalement une telle machine T_k n'existe pas.

Et donc $L_B \notin P^B$. ■

REMARQUE : Dommage !

6.3 Notion de réduction Turing

DÉFINITION 6.3.1
Etant donnés 2 langages A et B (sur un alphabet Σ), on dit que A est Turing-réductible en B et on note $A \leq_T B$ si il existe une machine de Turing déterministe de complexité en temps polynomiale qui décide A avec l'oracle B.

FAIT 6.3.1 *$\leq_T$ est un préordre.*

PREUVE :
- (réflexivité) évident.
- (transitivité) : si $A \leq_T B$ par la machine $\mathcal{M}$ et $B \leq_T C$ par la machine $\mathcal{M}^C$, on contruit la machine suivante :
  – sur l'entrée elle simule $\mathcal{M}$ qui décide A avec l'oracle B jusqu'à entrer dans l'état $q_?$,
  – alors elle simule la machine $\mathcal{M}^C$ sur le mot du ruban de l'oracle B. Elle répond ce que dit $\mathcal{M}^C$.

  Cette machine réalise $A \leq_T C$.

■

PROPOSITION 6.3.1 $A \leq B \Rightarrow A \leq_T B$

PREUVE : Triviale ! Attention, la réciproque est fausse ! ■

REMARQUE : Si $A \leq_T$ alors :
– B récursif $\Rightarrow$ A récursif.
– B réc. énum. $\Rightarrow$ A réc.énum.

FAIT 6.3.2 *On a $\overline{A} \leq_T A$. On ne sait pas si $\overline{A} \leq A$. Par contre, si $NP \neq Co-NP$, alors $\overline{SAT} \not\leq SAT$.*

DÉFINITION 6.3.2 (CLÔTURE)
On dit que $\mathcal{C}$ est close par $\leq_T$ si la classe vérifie :

$$\text{si } L \in \mathcal{C} \text{ et } L' \leq_T L, \text{ alors } L' \in \mathcal{C}$$

PROPOSITION 6.3.2
1. *P et $PSPACE$ sont closes pour $\leq_T$.*
2. *$NP = CoNP \Leftrightarrow NP$ est close pour $\leq_T$.*

PREUVE :

1. On va montrer que P est close pour $\leq_T$. Soit $L \in P$ et soit $L' \leq_T L$. On appelle $\mathcal{M}$ une machine qui décide L en temps polynomial, et $\mathcal{M}'$ une machine qui décide L' en temps polynomial avec l'oracle L. On considère l'algorithme suivant :
   - sur l'entrée ω, on fait agir $\mathcal{M}'$ jusqu'à ce qu'elle entre dans l'état d'interrogation,
   - alors on simule $\mathcal{M}'$ sur le ruban d'oracle,
   - $\mathcal{M}'$ continue "normalement" sur la réponse de $\mathcal{M}$.

   Cet algorithme est polynomial et décide L'.
2. • $\Rightarrow$: si $NP = Co - NP$. Soit $L \in NP$, et $L' \leq_T L$. On va montrer que $L' \in NP$. Soit $\mathcal{M}_1$ une machine de Turing qui décide L en temps $p_1(n)$. Soit $\mathcal{M}'$ une machine de Turing déterministe qui décide L' en temps polynômial avec l'oracle L, en un temps p'. Soit $\mathcal{M}_2$ qui tient compte du fait que $L \in Co - NP$ (elle décide $|L|$ en temps $p_2(n)$). Sur l'entrée ω, $\mathcal{M}'$ fonctionne jusqu'à entrer dans l'état d'interrogation. On fait alors fonctionner $\mathcal{M}_1$ et $\mathcal{M}_2$ sur le ruban d'oracle pour un temps $\text{Max}\{p_1(y), p_2(y)\}$, y étant le mot d'oracle. On continue en fonction des réponses, et on obtient un algorithme non déterministe en temps polynômial pour L'.
   • $\Leftarrow$: on montre par l'absurde. Sinon, il existe $L \in NP$, $L \notin Co - NP$, et si $L' \leq_T L$, comme NP close on obtient alors $|L| \in NP$. absurde.

■

6.4 Classes relativisées de P et de NP

DÉFINITION 6.4.1
- A étant un langage, $P^A = \{L \in P,$ décidés avec l'oracle $A\}$
- de même, $NP^A = \{L \in NP,$ décidés avec l'oracle $A\}$
- de même avec une classe $\mathcal{C}$ quelconque.

PROPOSITION 6.4.1
- $A \in NP^A$;
- $A \in P^B \Rightarrow A \in NP^B$;
- $A \in NP^B \Rightarrow A \in NP^{\overline{B}}$;
- $A \in NP^B$ *et* $B \in P^C \Rightarrow A \in NP^C$.

REMARQUE : Par abus de langage, on a noté A pour $\{A\}$ dans la proposition précédente. PREUVE : Les preuves sont identiques aux précédentes et faciles.

■

PROPOSITION 6.4.2
1. $NP^P = NP$;
2. $NP^{PSPACE} = PSPACE$.

PREUVE :
1. L'inclusion $NP \subseteq NP^P$ est évidente. Pour l'autre inclusion, prenons A un langage de NP^P. Par définition, il existe un langage $B \in P$, tel que $A \in NP^B$. Au lieu de poser les questions à l'oracle, on fait le calcul en temps polynomial. On obtient bien $A \in NP$.

2. L'inclusion $PSPACE \subseteq NP^{PSPACE}$ est évidente. Pour l'autre inclusion, prenons A dans NP^{PSPACE}. Par définition, il existe un oracle B de $PSPACE$ tel que $A \in NP^B$. Donc il existe une machine de Turing non detérministe $\mathcal{M}$ rendant compte de $A \in NP^B$. D'autre part, il existe une machine de Turing déterministe $\mathcal{M}'$ rendant compte de $B \in PSPACE$, ce qui implique l'existence de $\widetilde{M}$ qui décide A en espace polynomial. Donc $A \in PSPACE^B \subseteq PSPACE^{PSPACE} = PSPACE$.

■

6.5 Hiérarchie polynômiale, V1

Il s'agit de la famille $(\Delta_n^p, \Sigma_n^p, \Pi_n^p)_{n\geq 0}$ avec :
- $\Delta_0^p = \Sigma_0^p = \Pi_0^p = P$
- $\Sigma_{n+1}^p = NP^{\Sigma_n^p}$
- $\Pi_{n+1}^p = \text{Co-}\Sigma_{n+1}^p$
- $\Delta_{n+1}^p = P^{\Sigma_{n+1}^p}$

NOTATION : On note $PH = \bigcup_{n\geq 0} \Sigma_n^p$.

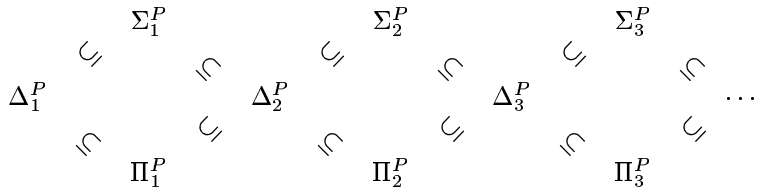
REMARQUE : $\Sigma_1^p = NP^P = NP, \Pi_1^p = \text{Co-}NP, \Delta_1^p = P$

PROPOSITION 6.5.1 $\forall n \geq 0, \Sigma_n^p \cup \Pi_n^p \subseteq \Delta_{n+1}^p \subseteq \Sigma_{n+1}^p \cap \Pi_{n+1}^p$.

PREUVE :
- Si $A \in \Sigma_n^p$, alors $A \in P^{\Sigma_n^p} = \Delta_{n+1}^p$ (car $A \in NP^A$). Si $A \in \Pi_n^p$, alors $\overline{A} \in \Sigma_n^p$. Or $\overline{A} \in P^{\overline{A}}$ et $A \in P^A = P^{\overline{A}}$ donc $A \in P^{\Sigma_n^p} = \Delta_{n+1}^p$.
- Si $A \in P^{\Sigma_n}$, alors $A \in NP^{\Sigma_n^p} = \Sigma_{n+1}^p$. Si $A \in P^{\Sigma_n^p}$, alors $\overline{A} \leq_T A \leq_T B$ pour un $B \in \Sigma_n^p$. Donc $\overline{A} \in P^B$ soit finalement $\overline{A} \in NP^B = \Sigma_{n+1}^p$.

■

REMARQUE : On obtient donc une hiérarchie que l'on visualise habituellement par des patatoïdes enlacés, ou par la figure suivante :



THÉORÈME 6.5.1 $PH = PSPACE$

PREUVE : Comme $PH = \cup_{n\geq 0}\Sigma_n^p$, on montre par récurrence sur n que $\Sigma_n^p \in PSPACE$:
- H_0 vérifiée car $P \in PSPACE$.
- $H_n \Rightarrow H_{n+1}$: Si $L \in \Sigma_{n=1}^p$ alors il existe un oracle $A \in \Sigma_n^p$, $L \in NP^A$. Par hypothèse de récurrence, $A \in PSPACE$ donc $L \in NP^{PSPACE} = PSPACE$.

■

REMARQUE : La portée de ce théorème est très importante !

6.6 Hiérarchie polynômiale, V2

Soit x un mot, $R(x)$ une propriété sur les mots, $p(n)$ un polynôme :

DÉFINITION 6.6.1
- $\exists^{p(n)} x R(x)$ signifie qu'il existe un mot x, $|x| \leq p(n)$, qui satisfait $R(x)$.
- $\forall^{p(n)} x R(x)$ signifie pour que tout mot x, tel que $|x| \leq p(n)$, on a $R(x)$.

DÉFINITION 6.6.2
Soit $\mathcal{C}$ une classe de langages, $\exists\mathcal{C}$ désigne la classe des langages A pour lesquels il existe $B \in \mathcal{C}$ tel que :

$$x \in A \Leftrightarrow \exists^{p(|x|)} y(< x, y > \in B$$

DÉFINITION 6.6.3
Soit $\mathcal{C}$ une classe de langages, $\forall\mathcal{C}$ désigne la classe des langages A pour lesquels il existe $B \in \mathcal{C}$ tel que :

$$x \in A \Leftrightarrow \forall^{p(|x|)} y(< x, y > \in B$$

THÉORÈME 6.6.1
1. $\exists P = NP = \Sigma_1^p$
2. $\forall P = Co\text{-}NP = \Pi_1^p$
3. $(k > 0), \exists\Sigma_k^p = \Sigma_k^p$
4. $(k > 0), \forall\Pi_k^p = \Pi_k^p$
5. $(k \geq 0), \exists\Pi_k^p = \Sigma_{k+1}^p$
6. $(k \geq 0), \forall\Sigma_k^p = \Pi_{k=1}^p$

REMARQUE : Donc les 2 hiérarchies sont les mêmes !
REMARQUE : La démo de ce théorème demande beaucoup de travail et l'introduction de nouvelles notions comme la notion de langage *clos par paire*.

DÉFINITION 6.6.4 ($A^{(*)}$)
Soit A un langage. On note $A^{(*)} = \{< y_1, y_2, \ldots, y_n >, n \in \mathbb{N}, y_i \in A\}$.

DÉFINITION 6.6.5 (CLÔTURE PAR PAIRE)
Une classe de langages $\mathcal{C}$ est dite **close par paire** lorsque $\forall A \in \mathcal{C}, \{< x, y > \mid x \in A\} \in \mathcal{C}$.

PROPOSITION 6.6.1 *Pour toute classe $\mathcal{C}$:*
1. $A \in \exists\mathcal{C} \Leftrightarrow \overline{A} \in \forall(Co - \mathcal{C})$.
2. *si $\mathcal{C}$ est close par paire, alors $\mathcal{C} \in \exists\mathcal{C}$ et $\mathcal{C} \in \forall\mathcal{C}$.*

PREUVE : Laisée en exercice. ■

LEMME 6.6.1 *Soit $\mathcal{C}$ l'une des classes Σ_k^p, Π_k^p, Δ_k^p. Alors pour tout langage A,*

$$A \in \mathcal{C} \Leftrightarrow A^{(*)} \in \mathcal{C}.$$

PREUVE : On fait la preuve pour Δ_k^p, les autres preuves sont identiques.

- $\Rightarrow$: soit $A \in \Sigma_k^p = P^{\Sigma_{k-1}^p}$, donc il existe $B \in \Sigma_{k-1}^p$, tel que $A \in P^B$. Ce que l'on veut montrer est $A^{(*)} \in \Delta_k^p$. Sur l'entrée $< y_1, \ldots, y_n >$, on commence par décoder en $y_1, y_2, \ldots, y_n$. Ensuite on simule la machine $\mathcal{M}$ qui rend compte de $A \in P^B$ sur les y_i. Si $\mathcal{M}$ les accepte tous, $< y_1, \ldots, y_n >$ est accepté, sinon il est rejeté. Cet algorithme décide donc $A^{(*)}$ avec l'oracle B en temps polynômial.
- $\Leftarrow$: si $A^{(*)} \in \mathcal{C}$, alors $A \in \mathcal{C}$ car $A \leq A^{(*)}$ (utiliser $x \mapsto < x >$).

■

On peut ensuite démontrer le théorème :

PREUVE : [du théorème 6.6.1]

1. Montrons $\exists P = NP$:
   - $\exists P \subseteq NP$: Soit $A \in \exists P$, il existe un langage $B \in P$ et un polynôme p tels que $x \in A \Leftrightarrow \exists^{p(|x|)} y\big(< x, y > \in B\big) \Leftrightarrow$ il existe un mot y, $|y| \leq p(|x|)$ tel que $< x, y > \in B$. Pour montrer que A est dans NP, on considère la machine suivante : sur l'entrée x, on devine y tel que $|y| \leq p(|x|)$. Ensuite, on simule la machine qui témoigne de $B \in P$ sur $< x, y >$. On accepte ssi cette machine accepte.
   - $NP \subseteq \exists P$: soit $A \in NP$, il existe donc une machine de Turing non déterministe qui décide A en temps polynômial, soit p le polynôme associé. On a alors $x \in A$ ssi il existe un chemin dans l'arbre des calculs de $\mathcal{M}$ qui mène à la configuration initiale caractérisée par x à une configuration d'acceptation, la longueur de ce chemin est inférieure ou égale à $p(|x|)$ (soit y ce chemin). On prendra pour $B = \{< x, y > \mid y$ acceptant pour x$\}$. Alors $A \in \exists P$.
2. La démonstration de $\forall P = Co - NP$ est duale de la précédente.
3. Montrons que $\exists \Sigma_k^p = \Sigma_k^p$. On commence par remarquer que Σ_k^p est close par paire. Il suffit donc de montrer que $\exists \Sigma_k^p \subseteq \Sigma_k^p$. Soit donc $A \in \Sigma_k^p$. Il existe donc un langage B et un polynôme p tels que $x \in A \Leftrightarrow \exists^{p|x|} y, < x, y > \in B$. $B \in \Sigma_k^p$ donc il existe une machine de Turing $\mathcal{M}$ et un langage $C \in \Sigma_{k-1}^p$ tels que B soit décidé en temps polynômial par $\mathcal{M}$ avec l'oracle C. On considère alors l'algorithme suivant : sur l'entrée x, on devine y tel que $|y| \leq p\,|x|$. On code $< x, y >$ que l'on teste sur $\mathcal{M}$, on accepte ssi $\mathcal{M}$ accepte. Cet algorithme décide A en temps polynômial non déterministe avec l'oracle $C \in \Sigma_{k+1}^p$. Donc $A \in NP^{\Sigma_{k-1}^p} = \Sigma_k^p$.
4. La démonstration de $\forall \Pi_k^p = \Pi_k^p$ est duale de la précédente.
5. La démonstration de $\exists \Pi_k^p = \Sigma_{k+1}^p$ est laissée en exercice.
6. La démonstration de $\forall \Sigma_k^p = \Pi_{k+1}^p$ est laissée en exercice.

■

THÉORÈME 6.6.2

1. *$A \in \Sigma_k^p$ ssi il existe $B \in P$ et $p(n)$ un polynôme tels que :*

$$x \in A \Leftrightarrow \exists^{p|x|} y_1 \forall^{p|x|} y_2 \dots Q^{p|x|} y_k (< x, y_1, \dots, y_k > \in B),$$

où $Q = \forall$ si k pair, $\exists$ sinon.

2. *résultat analogue pour Π_k^p.*

PREUVE : On prouve simultanément 1. et 2. par récurrence.

- Le cas $k = 1$ est trivial.
- On suppose 1. et 2. vraies au rang $k-1$. Comme :

$$A \in \sigma_k^p \Leftrightarrow A \in \exists \Pi_{k-1}^p,$$

alors $A \in \Sigma_k^p$ ssi il existe un langage $B \in \Pi_{k-1}^p$ et un polynome p tels que A soit décidé à l'aide de l'oracle B.
On obtient donc l'équivalence $x \in A$ ssi $\exists^{p|x|} y, (< x, y > \in B)$. On peut appliquer l'hypothèse de récurrence à B à savoir qu'il existe un polynome r et un langage $C \in P$ tel que :

$$< x, y > \in B \Leftrightarrow \forall^{r|x|} y_1 \exists^{r|x|} y_2 \dots Q_k^{r|x|} y_k < x, y_1, \dots y_k > \in C.$$

D'où le résultat.

■

6.7 Résultats généraux sur les 2 hiérarchies

THÉORÈME 6.7.1 *Si il existe k tel que $\Sigma_k^p = \Pi_k^p$, alors $\forall j \geq 0, \Sigma_{k+j}^p = \Pi_{k+j}^p$*

PREUVE : Ce théorème se montre par récurrence sur j. $\Sigma_{k+j+1}^p = \exists \Pi_{k+j}^p$ et l'hypothèse de récurrence donne : $\exists \Pi_{k+j}^p = \exists \Sigma_{k+j}^p = \Sigma_{k+j}^p$. ■

COROLLAIRE 6.7.1 $\exists k > 0, P = \Sigma_0^p \neq \Sigma_k^p \Rightarrow P \neq NP$

COROLLAIRE 6.7.2 $P \neq NP \Leftrightarrow P \neq PH$.

THÉORÈME 6.7.2 $PH = PSPACE \Rightarrow \exists k, \Sigma_{k+1}^p = \Sigma_k^p$.

PREUVE : Soit A un langage $PSPACE$-complet. Donc $A \in PSPACE$, donc $A \in PH$. Il existe donc $k, A \in \Sigma_k^p$. Soit $B \in PSPACE$, alors $B \leq A \Rightarrow B \in \Sigma_k^p$ car Σ_k^p close. Donc $PSPACE \subset \Sigma_k^p$ donc $\Sigma_{k+1}^p \subseteq \Sigma_k^p$. ■

6.8 Intérêts (! ?) de la hiérarchie polynômiale

6.8.1 Sa grande ressemblance avec la hiérarchie mathématique

- Le problème de l'arrêt donne un exemple de langage récursivement énumérable non récursif : $\varphi^{(1)}$.
- le problème de l'arrêt des MT avec l'oracle $\varphi^{(1)}$, donne l'existence d'un langage énumérable avec $\varphi^{(1)}$ mais non récursif avec $\varphi^{(1)}$: $\varphi^{(2)}$.

D'où une hiérarchie que l'on peut présenter autrement :

DÉFINITION 6.8.1
- Une relation R ($\subseteq (\Sigma^*)^k$) est dite récursive si $L_R = \{< x_1, \ldots, x_k > / R(x_1, \ldots, x_k)\}$ est récursif.
- $\forall k \geq 0$, on définit Σ_k comme la classe des langages L pour lesquels $\exists$ une relation $(k+1)$-aire R récursive telle que : $L = \{x / \exists x_1, \forall x_2, \ldots Q_k x_k R(x, x_1, \ldots, x_k)\}$.

REMARQUE : Σ_0 est l'ensemble des fonctions récursives; Σ_1 l'ensemble des fonctions récursivement énumérables.

PROPOSITION 6.8.1 *Si on note Π_k = Co-Σ_k, on a :*
- $\forall k \geq 0, \exists$ *un langage dans* $\Pi_k \setminus \Sigma_k$;
- $\forall k \geq 0, \exists$ *un langage dans* $\Sigma_k \setminus \Pi_k$;
- $\forall k \geq 0, \Sigma_k \cup \Pi_k \subseteq \Sigma_{k+1} \cup \Pi_{k+1}$;
- $\forall k \geq 0, \Sigma_k \neq \Sigma_{k+1}$.

6.8.2 Il existe des problèmes "naturels" dont les langages associés sont dans les différents niveaux de la hiérarchie

EXEMPLE 6.8.1 *CIRCUIT MINIMUM :*
- *Soit C un circuit booléen.*
- *Est-il minimum, au sens qu'un circuit ayant moins de portes ne peut calculer la même fonction.*

PROPOSITION 6.8.2 *$CM \in \Pi_2^p$. On ne sait pas s'il appartient à un niveau inférieur, ni s'il est Π_2^p-complet.*

EXEMPLE 6.8.2 *GRN :*
On dit qu'un graphe $G = (V, E)$ est k-coloré si il est associé à une fonction $E \to \{1, \ldots, k\}$. Pour $k > 0$, on définit R_k (nombre de Ramsey) comme l'entier n minimum tel que tout graphe complet 2-coloré de taille n contienne une clique monocolore de taille k. On sait que R_k existe.
GNR : problème des nombres de Ramsey généralisés. Soit un graphe complet $G = (V, E)$ partiellement 2 coloré : $C : E \to \{0, 1, *\}$, et soi $k > 0$. Est-il vrai que $\forall C' : E \to \{0, 1\}$ tel que $C'(e) = C(e)$ si $c(e) \neq *$, il existe une clique monocolore de taille k ?

PROPOSITION 6.8.3 *GRN est Π_2^p-complet.*

REMARQUE : On le démontre en le réduisant à partir de $3 - SAT_2$.

EXEMPLE 6.8.3 *SAT_k :*
Pour $k \geq 1$, généralisons SAT. Si X est un ensemble de variables, $\tau : X \to \{0, 1\}$.
Soit $X_1, \ldots, X_k$ des ensembles de variables, et φ une formule booléenne. Est-il vrai que :

$$\exists \tau_1 : X_1 \to \{0,1\}, \forall \tau_2 : X_2 \to \{0,1\}, \ldots Q_k \tau_k : X_k \to \{0,1\}, \varphi_{|\tau_i} = 1?$$

THÉORÈME 6.8.1 *$\forall k \geq 1$, SAT_k est Σ_k^p-complet.*

COROLLAIRE 6.8.1
- $\forall k \geq 1$, $2 - CNF - SAT_k$ *est* Σ_k^p*-complet pour* k *impair ;*
- $\forall k \geq 1$, $3 - CNF - SAT_k$ *est* Π_k^p*-complet pour* k *impair ;*

DÉFINITION 6.8.2
Quelques nouvelles classes de complexité :
- $2 - EXP = \bigcup_{k \geq 1} TIME\left(2^{(2^{n^k})}\right)$
- $ELEM$ est la réunion des langages sont la complexité s'écrit sous forme d'une tour finie d'exponentielles.

PROPOSITION 6.8.4 *Les problèmes d'équivalence des expressions rationnelles sont partout dans la hiérarchie :*
- *l'équivalence sur* $(+, \cdot, *)$ *est* $PSPACE$*-complet ;*
- *l'équivalence sur* $(+, \cdot)$ *est Co-*NP*-complet ;*
- *l'équivalence sur* $(+, \cdot, *, ^2)$ *est* $EXPSPACE$*-complet ;*
- *l'équivalence sur* $(+, \cdot, ^2)$ *est Co-*$NEXPSPACE$*-complet ;*
- *l'équivalence sur* $(+, \cdot, *, ^2, \neg)$ *où* $\neg$ *est le complémentaire appartient à* $ELEM$ *;*

Chapitre 7

Théorèmes généraux - théorie axiomatique

Dans ce chapitre on donne des idées de preuves pour le cas de l'espace déterministe , mais des résultats similaires existent pour le cas du temps, et aussi pour le cas non déterministe.

7.1 Théorème de la lacune

THÉORÈME 7.1.1 (LACUNE) *Soit $g : \mathbb{N} \to \mathbb{N}$ une fonction totale récursive, vérifiant $\forall n, g(n) \geq n$. Alors il existe une fonction s totale récursive telle que*

$$SPACE\big(s(n)\big) = SPACE\Big(g\big(s(n)\big)\Big).$$

REMARQUE : Cela veut dire qu'il existe un fossé entre les limites de la hiérarchie.

7.2 Théorème d'accélération de Blum

THÉORÈME 7.2.1 *Soit r une fonction totale récursive. Il existe un langage L récursif tel que pour toute machine de Turing $\mathcal{M}_i$ qui le décide, il existe une machine $\mathcal{M}_j$ qui le décide et telle que : $r\big(s_j(n)\big) \leq s_i(n)$ presque partout.*

7.3 Théorème de l'union

THÉORÈME 7.3.1 *Soit $\big(f_i(n)\big)_{i\geq 1}$ une famille récursivement énumérable de fonctions récursives (c'est-à-dire qu'il existe une machine de Turing qui énumère les machines $\mathcal{M}_i$ qui calculent les f_i). On suppose aussi que $\forall i, \forall n, f_i(n) < f_{i+1}(n)$. Alors il existe une fonction récursive s telle que*

$$SPACE\big(s(n)\big) = \bigcup_{i\geq 1} SPACE\big(f_i(n)\big).$$

REMARQUE : En particulier $PSPACE$ est une véritable classe de complexité, *i.e.* qu'il existe une fonction dont elle est la classe de complexité...

7.4 Mesures de complexité abstraites

DÉFINITION 7.4.1
Soit (φ_i) un système acceptable de programmation. Une **énumération** (Φ_i) de fonctions Turing-calculables définit une mesure de complexité pour le système $({}_v phi_i)$ lorsque :

1. $\forall i, \forall x, \Phi_i(x)$ est définie ssi $\varphi_i(x)$ l'est ;
2. l'ensemble $\{< i, x, y > | \Phi_i(x) \leq y\}$ est récursif.

EXEMPLE 7.4.1 (CANONIQUE) *Soit (φ_i) un système acceptable de programmation. Il a une fonction universelle u. On considère l'ensemble $A = \{< i, j > \mid u(< i, j >)$ est défini$\}$. A est récursivement énumérable donc il existe B récursif tel que $< i, j > \in A$ ssi $\exists t, < i, j, t > \in B$.*
Alors $\Phi_i(j) = \text{Min}\{\ t \mid < i, j, t > \in B\}$ est une mesure de complexité sur (φ_i).

THÉORÈME 7.4.1 (RELATIVISATION) *Soient (φ_i) et (ψ_i) deux systèmes acceptables de programmation. Soit t une fonction totale récursive qui permet la traduction de l'un dans l'autre (il en existe une bijective). Soient (Φ_i) et(Ψ_i) des mesures de compelxité associées. Alors il existe une fonction r totale récursive telle que pour tout i on ait presque partout :*

$$\Phi_i(x) \leq r\big(< x, \Psi_{t(i)}(x) > \big) \text{ et } \Psi_{t(i)}(x) \leq r\big(< x, \Psi_i(x) > \big)$$

REMARQUE : Par exemple, on sait comparer polynômialement les machines de Turing et les machines RAM.