Magistère d'Informatique et Modélisation
deuxième semestre 1999/2000

cours de

Modèles de calcul et complexité

cours : Marianne Delorme
TD : Natacha Portier
rédaction : Emmanuel Beffara et Arnaud Carayol

Introduction

La premiers modèles de calcul datent des années 1930. La motivation principale est de formaliser les notions intuitives de calcul, procédure, algorithme, etc.

La première idée, pour obtenir une classification des problèmes selon leur complexité, est la *hiérarchie de Chomsky*. Elle date des environs de 1955/1960. On se base ici sur différents modèles de puissance différente :

automates finis : on dispose d'un « centre de contrôle », évoluant dans un nombre fini d'états, et d'un ruban muni d'une tête de lecture qui ne peut que lire un caractère et avancer. Ils sont associés aux *langages rationnels* (en anglais *regular languages*) ;

automates à pile : on ajoute à l'automate une mémoire sous forme de pile où l'on peut empiler et dépiler des éléments, et où l'on ne peut lire que l'élément situé au sommet. Ces automates sont associés aux *langages algébriques* (qualifiés en anglais de *context-free*). Le terme d'automate à pile se traduit en anglais par *pushdown automaton*, mais désigne également la notion de *stack automaton*, où l'on s'autorise à lire à n'importe quel endroit de la pile ;

machines de Turing : ce modèle est le plus évolué, et il reconnait les langages dits *récursifs* et *récursivement énumérables*. On en fera une description précise dans le premier chapitre.

Mais très vite il apparaît que cette classification n'est pas satisfaisante. Hartmanis et Stearn introduisent alors une nouvelle vision de la complexité : la complexité « computationnelle », c'est-à-dire la complexité en temps ainsi qu'en espace. Cette classification utilise les machines de Turing. On aboutit à une nouvelle classification, celle des problèmes *P*, *NP*, *PSPACE* et *NSPACE*.

Il existe également une vision axiomatique au tavers de la complexité abstraite, mais les résultats obtenus ne sont pas réellement satisfaisants.

Dans la recherche sur le célèbre problème de savoir si $P = NP$, on a aussi introduit la notion d'oracle, mais les résultats n'ont pas été ceux espérés, puisqu'on obtenait des situations avec $P = NP$ avec certains oracles et des situation avec $P \neq NP$ avec d'autres.

Il existe enfin des théories plus classiques de la complexité :
- complexité algébrique (voir TD de Natacha)
- complexité descriptive basée sur les modèles finis (voir cours de complexité de Mazoyer)
- complexité algorithmique, avec la complexité de Kolmogorov, etc.

Table des matières

1 Modèles de calcul **1**
1.1 Machines de Turing . . . 1
1.1.1 Définition . . . 1
1.1.2 Résultats classiques . . . 2
1.2 Algorithmes de Markov . . . 2
1.3 Machines RAM . . . 3
1.4 Circuits booléens . . . 5
1.4.1 Fonctions booléennes . . . 5
1.4.2 Circuits booléens . . . 5
1.4.3 Familles de circuits . . . 7
1.4.4 Familles uniformes de circuits . . . 8
1.5 Automates cellulaires . . . 8
1.6 Systèmes de programmation acceptables . . . 9
1.7 Mesures de complexité . . . 13
1.7.1 Machines de Turing . . . 13
1.7.2 Autres modèles . . . 14

2 Complexité abstraite **15**

3 Complexité en temps via Turing **17**
3.1 Classes de complexité en temps . . . 17
3.2 La classe P . . . 18
3.3 Machines non déterministes . . . 18
3.4 La classe NP . . . 19

4 Classes de complexité en espace via Turing **21**

5 Quelques propriétés générales des classes de complexité **23**
5.1 (titre) . . . 23
5.2 Théorème de hiérarchie . . . 24
5.3 Théorème de hiérarchie en temps . . . 25
5.4 Lemme de translation . . . 26
5.5 Théorème de la lacune . . . 27

6 Hiérarchie polynômiale, machines de Turing avec oracle **29**
6.1 Machines de Turing avec oracle . . . 29
6.2 Relativisation, application à P et NP . . . 30
6.3 Turing-réduction . . . 32

6.4 La hiérarchie polynômiale 33
6.5 Hiérarchie polynômiale, le retour 34

A Solution des exercices **37**

B Listes **43**
B.1 Liste des définitions 43
B.2 Liste des théorèmes 43
B.3 Liste des exercices 43

premier chapitre

Modèles de calcul

On étudie ici d'abord trois modèles de calcul :
- les machines de Turing ;
- les algorithmes de Markov ;
- les machines RAM.

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 $\mathbf{N}$ dans $\mathbf{N}$ 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

On suppose connues les bases de la théorie des machines Turing, c'est pourquoi on ne reviendra que rapidement dessus.

1.1.1 Définition

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 $\{-1, 0, 1\}$.

Une configuration est intuitivement la donnée d'un ruban

$$\cdots \boxed{B \mid B \mid B \mid a \mid b \mid d \mid a \mid c \mid a \mid B \mid B \mid B} \cdots$$

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.

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 $\mathbf{N}$ 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 $\mathbf{N}^p$ et $\mathbf{N}$ ou entre $\mathbf{N}^*$ et $\mathbf{N}$ 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 $\mathbf{N}^{m+1}$ dans $\mathbf{N}$ qui vérifie, pour toutes les entiers x, $y_1, \ldots, y_m$ et $z_1, \ldots, z_n$:

$$\varphi_x(\langle y_1, \ldots, y_m, z_1, \ldots, z_n \rangle) = \varphi_{s_n^m(x, y1, \ldots, y_m)}(\langle z_1, \ldots, z_n \rangle)$$

1.2 Algorithmes de Markov

On considère ici des systèmes de réécriture particuliers.

définition 1.1 (*algorithme de Markov*)

un algorithme de Markov est un système fini ordonné *de règles $p_i \rightarrow_{(t)} q_i$, où p_i et q_i sont des mots sur un alphabet fini Σ. On a deux types de règles : $p_i \rightarrow q_i$ et $p_i \rightarrow_t q_i$.*

À partir de ces règles, on définit une relation de dérivation sur l'ensemble Σ^* des mots sur Σ :

- une dérivation simple est $u \vdash v$ où v est obtenu par substitution, au facteur p_i de u le plus à gauche dans u, du facteur q_i de la première règle possible ;
- on dit qu'une telle est terminale si $p_i \rightarrow_t q_i$;
- on en déduit une relation de dérivation $\vdash^*$ de la façon suivante :
  - $u \vdash^0 v$ si et seulement si $u = v$

- $u \vdash^{m+1} v$ si et seulement s'il existe un mot w tel que $u \vdash^m w$ et $w \vdash v$
- $u \vdash_t^{m+1} v$ si et seulement s'il existe un mot w tel que $u \vdash^m w$ et $w \vdash_t v$

et $\vdash^*$ signifie $\vdash^n$ pour un n donné ;
- si M désigne un système de Markov, on a $M(u) = v$ si et seulement s'il existe une dérivation *finie* de v à partir de u, c'est-à-dire soit $u \vdash^* v$ soit $u \vdash_t^* v$ et aucune règle n'est plus applicable à v.

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 $(x_1, x_2, \ldots, x_p) \in \mathrm{dom}\, f$
2. alors $M(x_1, x_2, \ldots, x_p) = f(x_1, x_2, \ldots, x_p)$

exemple : 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. $\varepsilon \to \$$

Une fois ces définitions posées, on obtient le résultat fondamental suivant, selon lequel les algorithmes de Markov calculent les mêmes fonctions que les machines de Turing exposées précédemment.

théorème 1.1

Une fonction est semi-calculable par machine de Turing si et seulement si elle l'est par algorithme de Markov.

La preuve sera laissée 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.

Exercice 1.1

Donner un algorithme de Markov qui calcule le successeur d'un entier écrit en binaire.

solution page 37

Exercice 1.2

Donner un algorithme de Markov qui calcule la fonction prédécesseur d'un entier écrit en binaire.

solution page 37

1.3 Machines RAM

Les machines RAM sont des systèmes de calcul beaucoup plus proches du fonctionnement réel d'ordinateurs, et de ce fait présentent un intérêt évident.

Une machine RAM est la donnée d'un certain nombre de registres, qui ont un nom $(R_1, R_2, \ldots, R_n, \ldots)$. Ces registres contiennent des mots sur un alphabet

fini Σ. On manipule les mots dans les registres via un programme dit *programme RAM* qui est une suite « convenable » d'instructions de l'un des types suivants :

type	forme	fonction
1_j	$X : \mathrm{add}_j\, Y$	concaténer a_j à droite du mot contenu dans Y
2	$X : \mathrm{del}\, Y$	effacer le premier caractère de Y
3	$X : \mathrm{clr}\, Y$	effacer le contenu de Y
4	$X : Y \leftarrow Z$	substituer le contenu de Z à celui de Y
5	$X : \mathrm{jmp}\, X'$	aller à la ligne X'
6_j	$X : Y\, \mathrm{jmp}_j\, X'$	aller en X' si la première lettre de Y est a_j
7	X : continue	ne rien faire

où X est le *nom* de la ligne portant l'instruction et où Y et Z sont des noms de registres. Un programme RAM est alors une suite finie d'instructions de l'un des types énoncés, qui se termine par une instruction de type 7, et qui est *cohérent*.

Une fonction f de $(\Sigma^*)^p$ dans (Σ^*) est dite semi-calculable par machine RAM, ou RAM-semi-calculable, s'il existe une machine RAM $R_1, \ldots, R_n$ et un programme RAM tels que si l'on range $x_1, \ldots, x_p$ dans $R_1, \ldots, R_p$, le programme termine si et seulement si $(x_1, \ldots, x_p) \in \mathrm{dom}\, f$, les autres registres contenant ε initialement, et alors l'image de $(x_1, \ldots, x_p)$ est le mot figurant dans R_1.

théorème 1.2

Une fonction est semi-calculable par machine RAM si et seulement si elle l'est par machine de Turing.

Exercice 1.3 — *équivalence RAM / Turing*

La démonstration de l'équivalence entre machines de Turing et machines RAM passera par l'intermédiaire des SRM (machines à un seul registre). On montrera ici seulement que toute fonction calculable par RAM l'est par machine de Turing. Soit donc un alphabet $A_k = \{a_1, \ldots, a_k\}$. Une SRM sur A_k est une machine qui possède un unique registre pouvant contenir n'importe quel mot de A_k^*. Ses instructions sont de trois sortes :

X **add** j qui ajoute a_j à droite du registre

X **del** qui efface la lettre la plus à gauche du registre s'il n'est pas vide

X **jmp** j, X' qui saute au premier label X' dans le sens précisé si la première lettre du registre est a_j

où X est un label ou rien et où X' est un label suivi de (a) ou (b) selon que le label doit être recherché avant ou après la position actuelle.

Si Φ est une fonction partielle à n variables sur A_k^*, une SRM sur $A_{k+1} = A_k \cup \{,\}$ calcule Φ si lorsqu'elle prend $x_1, \ldots, x_n$ comme valeur initiale de son registre elle s'arrête si et seulement si $\Phi(x_1, \ldots, x_n)$ est défini et si dans ce cas elle s'arrête avec cette valeur dans son registre.

1) Donner une SRM qui sur l'entrée $x_1, \ldots, x_n$ calcule $x_2, \ldots, x_n, x_1$

2) Montrer que toute fonction calculable par RAM est calculable par SRM.

3) Montrer que toute fonction calculable par SRM est calculable par machine de Turing et conclure.

solution page 38

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 Fonctions booléennes

On appelle fonction booléenne toute fonction de $\{0,1\}^n$ dans $\{0,1\}$. On sait déjà que toute fonction booléenne peut être représentée par une formule du calcul propositionnel. Réciproquement, une formule du calcul propositionnel définit une fonction booléenne. On parle ici de formules construites à partir des connecteurs $\neg$, $\vee$ et $\wedge$.

1.4.2 Circuits booléens

On a un certain nombre $n+p$ de « portes ». Elles sont étiquetées soit par des variables, soit par des opérateurs pris parmi $\{\neg, \vee, \wedge\}$, soit par des constantes 0 ou 1. Les portes sont reliées par des fils, et chaque fois on a des portes d'entrée desquelles on ne peut que sortir et, et une porte de sortie dans laquelle on ne peut qu'entrer. On ne peut entrer que par un fil que dans une porte $\neg$ et par deux dans une porte $\vee$ ou $\wedge$.

Il est possible également de donner une définition en termes de graphes :

définition 1.2 (*circuit booléen*)

Un circuit booléen est un graphe orienté $C = (V, E)$, avec

$$V = \{1, \dots, n, \dots, n+q\}$$

tel que :

- *chaque sommet est de degré entrant 0, 1 ou 2*
- *à chaque sommet i est associé un type et une étiquette $t(i)$ de telle sorte que*
  - *$t(i) \in \{x_1, \dots, x_n, 0, 1\}$ pour $1 \leqslant i \leqslant n$*
  - *$t(i) \in \{\wedge, \vee, \neg, 0, 1\}$ pour $n+1 \leqslant i \leqslant n+q$*
- *les nœuds d'étiquette $y_i \in \{0, 1, x_1, \dots, x_n\}$ ont 0 pour degré entrant*
- *les nœuds marqués par $\neg$, $\vee$ ou $\wedge$ sont de degré entrant 1, 2, 2 respectivement*

Les nœuds d'un circuit booléen sont appelés portes.

Un tel circuit définit une fonction f_{n+q} de $\{0,1\}^n$ dans $\{0,1\}$ de façon intuitive. On dit que le circuit C *calcule* f_{n+q}.

Une fois cette définition posée, on obtient quelques propriétés élémentaires mais importantes.

P 1.3

our toute fonction booléenne f de $\{0,1\}^n$ dans $\{0,1\}$, on a

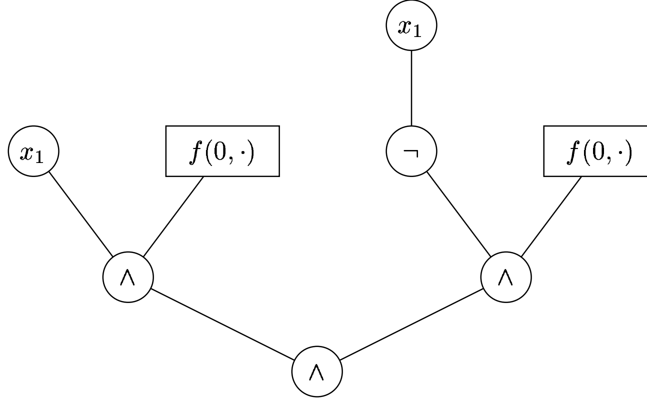
$$min\,\{q \mid t(C) = q, C \text{ calcule } f\} < 2^{n+2}$$

où $t(C)$ est la taille du graphe C, c'est-à-dire son nombre de nœuds.

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 suivant 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 $g(n) < 2^{n+2}$. ■

On peut également établir un résultat inverse, c'est-à-dire minorer le nombre minimal de portes nécessaire au calcul de la fonction nécessitant le plus grand nombre de portes. Plus précisément, on a la proposition suivante :

P 1.4

Pour tout n au moins égal à 2, il existe une fonction booléenne n-aire telle qu'aucun circuit de taille inférieure ou égale à $2^n/2n$ ne peut la calculer.

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. ■

Cependant, pour rendre compte d'ensembles de fonctions, ou pour parler de langages décidables, il est nécessaire de considérer des familles de circuits, et non des circuits en eux-mêmes.

1.4.3 Familles de circuits

On va donc considérer des familles de circuits et étudier les ensembles de fonctions booléennes associés. En particulier, 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.

On a alors le résultat évident suivant :

T 1.5

T *out langage sur $\{0,1\}$ est décidable par une famille de circuits.*

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. ■

I 1.6 (*corollaire*)

I *l existe des langages indécidables par machine de Turing qui sont décidables par famille circuits booléens.*

Ces résultats sont passablement décevants, et pour tenter de retrouver le terrain de la calculabilité classique, on va donc chercher à imposer des conditions sur les familles de circuits considérées.

La première idée est de considérer des circuits dont la complexité est soumise à une contrainte particulière. On dira par exemple qu'un langage sur $\{0,1\}$ a des circuits polynômiaux s'il existe une famille de circuits $(C_i)_{i \geqslant 0}$ qui le décide et s'il existe un polynôme p tel que pour tout n la taille de C_n soit inférieure à $p(n)$. On a bien alors une classe de langages décidables, mais on a tout de même le résultat suivant :

I 1.7

l 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.

1.4.4 Familles uniformes de circuits

définition 1.3 (*famille uniforme de circuits booléens*)

Une famille $(C_i)_{i\geqslant 0}$ de circuits booléens est dite uniforme s'il existe une machine de Turing M qui sur l'entrée 1^n produit un codage du circuit C_n.

Cette notion dépend donc *a priori* du codage utilisé, mais on se référera à un codage « canonique ».

1.5 Automates cellulaires

On ne considérera ici que des automates cellulaires sur $\mathbf{Z}$, donc unidimensionnels.

définition 1.4 (*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 $Q^{\mathbf{Z}}$ dans $Q^{\mathbf{Z}}$, par

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

pour toute fonction c de $\mathbf{Z}$ 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.

L 1.8

a classe des fonctions semi-calculables par les automates cellulaires de dimension 1 est la classe des fonctions récursives.

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 Systèmes de programmation acceptables

On passe maintenant à une notion plus abstraite de modèle de calcul, dans laquelle on pourra montrer des résultats plus généraux. Il s'agit de la notion de système de programmation acceptable. On aboutira en particulier au résultat fondamental qu'est le théorème de l'isomorphisme de Rogers.

définition 1.5 (*système de programmation acceptable*)

Par système de programmation acceptable, ou spa, on entend toute énumération $(\varphi_i)_{i \geqslant 0}$ surjective des fonctions semi-calculables (ou ppr) de **N** *dans* **N** *telle que*

1. *il existe une fonction universelle φ_u telle que $\varphi_u(\langle i, j\rangle) = \varphi_i(j)$ pour tous i et j ;*
2. *le théorème s_n^m est vérifié*[1].

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 $\psi_i = \varphi_{f^{-1}(i)}$.

On peut tout d'abord énoncer une première version, simplifiée bien entendu, de ce résultat :

S 1.9

i (φ_i) et (ψ_i) sont des systèmes de programmation acceptables, il existe une fonction totale récursive t telle que pour tout i on ait $\varphi_i = \psi_{t(i)}$

preuve : Par hypothèse, il existe une fonction universelle φ_u dans (φ_i), qui vérifie donc par définition $\varphi_u(\langle i, j\rangle) = \varphi_i(j)$ pour tous i et j. Or φ_u est récursive puisqu'elle fait partie de (φ_i), donc elle a un indice a dans l'énumération (ψ_i), et pour tous i et j on a

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

L'application du théorème s_n^m pour (ψ_i) montre alors l'existence d'une fonction totale récursive s_1^1 telle que pour tous i et j on ait

$$\psi_a(\langle i, j\rangle) = \psi_{s_1^1(a,i)}(j)$$

La fonction $t : i \mapsto s_1^1(a, i)$ convient donc, car elle est totale récursive et vérifie la propriété voulue. ■

Afin d'étendre cette proposition au résultat initialement évoqué, on utilisera le théorème suivant, dit *lemme de remplissage* :

théorème 1.10 (*lemme de remplissage*)

Soit $(\varphi_i)_{i\geqslant 0}$ une système de programmation acceptable. Il existe alors une fonction totale récursive ρ injective telle que pour tous i et x on ait

$$\varphi_i = \varphi_{\rho(\langle i,x\rangle)}$$

Ceci signifie que la fonction ρ permet d'associer à tout « programme » i une infinité de « programmes » qui calculent la même fonction φ_i.

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

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

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

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

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

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

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. ■

Exercice 1.4 — *lemme de remplissage fort*

Étendre le résultat précédent de montrant que si $(\varphi_i)_{i\geqslant 0}$ est un système de programmation acceptable, alors il existe une fonction récursive totale ρ' telle que pour tout i et pour tout x, on ait $\varphi_i = \varphi_{\rho'(\langle i,x\rangle)}$, et telle que si $i \neq i'$ ou $x \neq x'$, alors $\rho'(\langle i,x\rangle) \neq \rho'(\langle i',x'\rangle)$

solution page 40

L'étape suivante utilise le lemme précédent pour renforcer le résultat en faisant de la fonction de correspondance entre codages une fonction strictement croissante :

S 1.11

Soient (φ_i) et (ψ_i) deux systèmes de programmation acceptables. Il existe alors une fonction g totale récursive et strictement croissante telle que $g(0) > 0$ et telle que pour tout i on ait $\varphi_i = \psi_{g(i)}$.

preuve : On sait déjà qu'il existe une fonction totale récursive t telle que $\varphi_i = \psi_{t(i)}$ pour tout i. On va utiliser t et le lemme de remplissage pour définir g. On commence par poser

$$g(0) = \rho\left(\langle t(0), \min\{y \mid \rho(\langle t(0),y\rangle) > 0\}\rangle\right)$$

Ensuite, si $g(k)$ est défini pour $0 \leqslant k < x$, on pose successivement :

$$m = \max\{g(0), g(1), \ldots, g(x-1)\}$$

$$g(x) = \rho\left(\langle t(x), \min\{y \mid \rho(\langle t(0),y\rangle) > m\}\rangle\right)$$

On a alors bien $\varphi_i = \psi_{g(i)}$ et g est strictement croissante par construction. ■

On dispose à présent de tous les outils nécessaires à la démonstration du résultat fondamental attendu :

théorème 1.12 (*isomorphisme de Rogers*)

Soient $(\varphi_i)_{i\geqslant 0}$ et $(\psi_i)_{i\geqslant 0}$ deux systèmes de programmation acceptables. Il existe alors une fonction totale récursive f, bijective, telle que pour tout i on ait

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

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. ■

Exercice 1.5

Soit (Φ_i) un système de programmation acceptable. Montrer qu'il existe une fonction récursive primitive p telle que pour tous entiers i et j, $p(\langle i, j\rangle)$ soit un indice tel que $\Phi_{p(\langle i,j\rangle)} = \Phi_i + \Phi_j$.

solution page 40

Exercice 1.6 — *caractérisation des s.p.a.*

Le but de cet exercice est de montrer la caractérisation suivante des systèmes de programmation acceptables :

> Un système de programmation est acceptable si et seulement si il existe une fonction c totale récursive telle que pour tous i et j on ait $\Phi_{c(i,j)} = \Phi_i \circ \Phi_j$.

1) Montrer le sens direct.

2) Soit (Φ_i) un système de programmation acceptable tel qu'il existe une fonction totale récursive c telle que $\Phi_{c(i,j)} = \Phi_i \circ \Phi_j$. On définit les fonctions P, Q, R et les entiers p et q par

$$P(y) = \langle 0, y\rangle \quad \Phi_p = P \quad Q(\langle x, y\rangle) = \langle x+1, y\rangle \quad \Phi_q = Q$$

$$R(0) = p \quad \forall x, R(x+1) = c(q, R(x))$$

a) Montrer que pour tous x et y, on a $\Phi_{R(x)}(y) = \langle x, y\rangle$.

b) Conclure.

solution page 40

Exercice 1.7 — *théorème du point fixe*

Soit (Φ_i) est un système de programmation acceptable. Montrer que pour toute fonction totale récursive α, il existe un entier i tel que $\Phi_i = \Phi_{\alpha(i)}$.

solution page 40

1.7 Mesures de complexité

On cherche à présent naturellement à établir une notion de complexité sur le modèles considérées ici. On introduit donc des notions de complexité en temps et en espace.

1.7.1 Machines de Turing

Soit M une machine de Turing sur un alphabet Σ. on associe à M deux fonctions :

- TMTIME_M, de Σ^* dans $\mathbf{N}$, qui à tout mot w associe la somme de la longueur de w et du nombre de pas de calcul menant à l'arrêt, si M s'arrête sur w. Si M ne s'arrête pas, $\text{TMTIME}_M(w)$ n'est pas défini ;
- TIME_M, de $\mathbf{N}$ dans $\mathbf{N}$, qui à tout entier n associe

$$\text{TIME}_M(w) = \max_{|w|=n} \text{TMTIME}_M(w)$$

  si M s'arrête sur tout mot de longueur n, et qui n'est pas définie s'il existe un mot de longueur n sur lequel M de termine pas.

On admettra sans difficulté que ces deux fonctions sont récursives. De la même façon, on définit les fonctions TMSPACE et SPACE qui ne font pas référence au nombre de pas de calcul jusqu'à l'arrêt mais au nombre de cases distinctes visitées jusqu'à l'arrêt. Ces deux fonctions sont également récursives.

Une fois ces définitions posées, on a un résultat de comparaison entre les deux mesures de complexité obtenues :

P 1.13

our toute machine de Turing M et pour toute entrée w, on a

$$\text{TMSPACE}_M(w) \leqslant \text{TMTIME}_M(w) \leqslant {k_M}^{\text{TMSpace}_M(w)}$$

où k_M est une constante ne dépendant que de la machine M.

preuve : La comparaison entre TMSPACE et TMTIME 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 $\text{TMSPACE}_M(w)$, et la position de la tête de la machine, qui est choisie parmi $\text{TMSPACE}_M(w) + c$ positions possibles où c est une constante. Le nombre de configurations est donc majoré par

$$|Q| \times |\Gamma|^{\text{TMSpace}_M(w)} \times (\text{TMSPACE}_M(w) + c)$$

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

1.7.2 Autres modèles

On a montré que tous les modèles présentés (en dehors des circuits booléens) sont équivalents par simulation. On peut donc comparer les complexités en temps et en espace de ces différents modèles.

Exercice 1.8

Les comparer !

deuxième chapitre

Complexité abstraite

Pas encore...

troisième chapitre

Complexité en temps via Turing

On considère ici des machines de Turing qui s'arrêtent sur toute entrée, et l'on considère des problèmes de décision. Ces problèmes sont représentés par des problèmes de reconnaissance de langages dans un codage donné.

M étant une machine de Turing, on définit $\mathrm{TMT}\textsc{ime}_M(w)$ comme étant la somme de la longueur du mot w et du nombre de pas de calcul jusqu'à l'arrêt. On en déduit une fonction de $\mathbf{N}$ dans $\mathbf{N}$ qui est la complexité en temps de M :

$$\textsc{Time}_M(n) = \max\left\{\mathrm{TMT}\textsc{ime}_M(w) \,\middle|\, |w| = n\right\}$$

C'est à partir de cette notion que l'on va définir réellement les classes de complexité en temps.

3.1 Classes de complexité en temps

Avant de définir et d'étudier des classes de complexité particulières, il nous faut dans un souci de généralité définir une notion générale de classe de complexité en temps. On se référera toujours à la complexité en temps relativement à une machine de Turing.

définition 3.1 (*classe de complexité en temps*)

Soit t une fonction de $\mathbf{N}$ dans $\mathbf{N}$. La classe de complexité en temps $t(n)$ est l'ensemble des langages L décidés par une machine de Turing M dont la complexité est un $\mathcal{O}(t(n))$, c'est-à-dire :

$$\textsc{Time}\big(t(n)\big) = \{L \mid L \text{ est décidé par } M \text{ telle que } \textsc{Time}_M(n) = \mathcal{O}(t(n))\}$$

exemple : Considérons le langage $L = \left\{a^k b^k \,\middle|\, k \geqslant 0\right\}$. Il est bien entendu décidable par machine de Turing. Si l'on considère des machines à un seul ruban, il est de la classe de complexité en temps $\mathcal{O}\left(n^2\right)$ par méthode benête, et $\mathcal{O}(n \log n)$ par une méthode adaptée. Si l'on considère des machines à deux rubans, au contraire, le langage se reconnaît en complexité $\mathcal{O}(n)$.

Cet exemple laisse penser qu'il peut exister une méthode générale permettant de relier la complexité en temps sur les machines à différents nombres de rubans.

On a en fait le résultat suivant, surprenant à première vue mais en fait assez naturel :

S 3.1

i M est une machine de Turing à k rubans et de complexité $t(n)$, alors il existe une machine de Turing qui décide le même langage en $\mathcal{O}\left(t(n)^2\right)$.

preuve : Pour toute preuve, on se contentera d'expliquer un méthode permettant d'obtenir le résultat. L'idée est que si les rubans contiennent les mots $a_1, \ldots, a_k$, alors sur la machine à un seul ruban correspondante on écrit $\sharp a_1 \sharp \ldots \sharp a_k \sharp$ en remplaçant dans chaque mot la lettre a sur laquelle se trouve la tête par une lettre a'. On obtient donc une correspondance de configurations en travaillant sur l'alphabet $\Sigma \cup \{a' \mid a \in \Sigma\} \cup \{\sharp\}$. À une étape donnée, on parcourt donc le ruban en entier pour faire évoluer la configuration. Comme la taille de chaque mot est bornée par le nombre total d'étape $t(n)$, et comme le nombre de parcours est constant, une transition se fait en $\mathcal{O}(t(n))$, ce qui donne bien une complexité en $\mathcal{O}\left(t(n)^2\right)$ au total. ■

3.2 La classe P

définition 3.2 (*classe* P)

La classe de complexité P *est la classe des langages reconnus en temps polynômial, c'est-à-dire*

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

Il y a tellement d'exemples de langages appartenant à la classe P qu'on n'en citera pas ici.

3.3 Machines non déterministes

Afin de définir la classe des problèmes NP, on définit à présent la notion de machine de Turing non déterministe.

définition 3.3 (*machine de Turing non déterministe*)

Une machine de Turing non déterministe est un octuplet

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

où Q est l'ensemble (fini) des états de M, Σ l'alphabet d'entrée, Γ l'alphabet de travail dont B est le caractère blanc (donc $\Sigma \subsetneq \Gamma$). Les états de départ, d'acceptation et de rejet sont respectivement q_0, q_a et q_r. La fonction de transition est δ, c'est une fonction de $Q \times \Gamma$ dans $\mathcal{P}(Q \times \Gamma \times Mvts)$ où Mvts est l'ensemble des mouvements, soit intuitivement $\{-1, 0, 1\}$.

La signification de cette définition aride est simplement qu'une transition n'est plus d'un état vers un autre état mais vers un ensemble d'états possibles. On dit alors qu'un langage L est décidé par une machine M lorsque pour tout mot w il *existe* un calcul de M sur w qui mène à un état d'acceptation si et seulement si w est élément de L.

La complexité en temps se définit alors pour une telle machine de la même façon que pour une machine déterministe, en considérant une machine qui termine toujours et en considérant comme mesure de complexité la longueur *maximale* des calculs qui terminent.

S 3.2

Soit M une machine de Turing non déterministe de complexité en temps $t(n)$, il existe alors une machine de Turing déterministe S qui décide le langage décidé par M en complexité $2^{\mathcal{O}(t(n))}$.

preuve : Pour toute preuve, on se limitera à l'intuition selon laquelle on peut simuler M avec une machine déterministe S et on laissera à la sagacité du lecteur la joie montrer que S tourne en complexité voulue. ■

3.4 La classe NP

Cette classe se définit comme P :

définition 3.4 (*classe* NP)

Il s'agit de l'ensemble des langages décidés par une machine de Turing non déterministe en temps polynômial.

Il est facile de voir que P est inclus dans NP, mais on voudrait des résultats plus forts.

quatrième chapitre

Classes de complexité en espace via Turing

Exercice 4.1 — *constructibilité en espace*

Une fonction f est dite constructible en espace s'il existe une machine de Turing M à un ruban d'entrée et un ruban de travail qui, sur l'entrée 1^n, termine avec le mot $1^{f(n)}$ sur le ruban de travail sans avoir visité d'autre case. On ne conisdère pas la lecture des blancs en bout de mot comme la visite d'une case.

Montrer que les fonctions suivantes sont constructibles en espace :

1) $n \mapsto n$

2) $n \mapsto \lceil n/2 \rceil$

3) $n \mapsto n^2$

4) $n \mapsto 2^n$

solution page 41

cinquième chapitre

Quelques propriétés générales des classes de complexité

On s'intéresse toujours aux classes de complexité relativement au modèle des machines de Turing, et on cherche à présent à établir des résultats généraux, c'est-à-dire qui ne soient pas spécifiques à des classes particulières. C'est ainsi qu'on établira des théorèmes de hiérarchie, ainsi que les théorèmes dits de la lacune, de l'accélération et de l'union.

On a jusqu'à maintenant introduit les notions de complexité en temps et en espace et on a montré les relations suivantes, pour une fonction f donnée :

$$\begin{aligned}
\textsc{Space}(f(n)) &\subseteq \textsc{NSpace}(f(n)) \\
\textsc{Time}(f(n)) &\subseteq \textsc{NTime}(f(n)) \\
\textsc{NTime}(f(n)) &\subseteq \textsc{Space}(f(n)) \\
\textsc{NSpace}(f(n)) &\subseteq \textsc{Time}\left(2^{cf(n)}\right) \\
\textsc{NSpace}(f(n)) &\subseteq \textsc{Space}\left(f(n)^2\right)
\end{aligned}$$

5.1 (titre)

théorème 5.1

Soit f une fonction totale récursive de $\mathbf{N}$ dans $\mathbf{N}$ non décroissante. Il existe alors un langage L' qui n'appartient pas à $\textsc{Time}(f(n))$ et un langage L'' qui n'appartient pas à $\textsc{Space}(f(n))$.

preuve : On se penche sur le cas de $\textsc{Time}(f(n))$. 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 résoud de façon identique. ■

Il résulte de ce théorème des résultats rassurants, comme par exemple le fait qu'il existe des fonctions f et g pour lesquelles

$$\text{TIME}(f(n)) \subsetneq \text{SPACE}(f(n))$$

5.2 Théorème de hiérarchie

On va à présent renforcer le résultat précédent pour permettre de créer un véritable hiérarchie des fonctions récursives selon leur complexité. On étabilt donc le théorème suivant :

théorème 5.2 (*théorème de hiérarchie*)

Soit f une fonction constructible en espace. Il existe alors un langage décidé et espace $\mathcal{O}(f(n))$ qui ne l'est pas en $o(f(n))$.

preuve : Rappelons que f est constructible en espace si $f(n) \geqslant \log n$ pour tout n et s'il existe une machine de Turing qui sur l'entrée 1^n rend l'écriture binaire de $f(n)$ en utilisant un espace de $\mathcal{O}(f(n))$.

À nouveau, on va alors considérer une énumération (M_i) des machines de Turing, et on va considérer un langage contenu dans

$$\left\{ \langle M_i \rangle 10^k \,\middle|\, i \geqslant 0, k \geqslant 0 \right\}$$

Soint $\hat{M}$ la machine de Turing qui a le fonctionnement suivant sur une entrée w :

1. elle calcule $|w| = n$;
2. elle calcule $f(n)$;
3. si w n'est pas de la forme $\langle M_i \rangle 10^k$, elle le rejette ;
4. sinon elle simule M sur w en espace inférieur à $f(n)$, et rejette w si et seulement si M l'accepte dans cet espace.

Le langage L décidé par $\hat{M}$ l'est donc en espace $\mathcal{O}(f(n))$. Si L était décidé en espace $o(f(n))$, il existerait une machine M qui le déciderait en espace $g(n)$ avec $g(n) = o(f(n))$. Dans ce cas, lorsque $\hat{M}$ simule le calcul de M sur $\langle M \rangle 10^k$, elle simule un calcul de M en espace $dg(n)$. Or il existe un entier n_0 tel que pour tout n supérieur à n_0 on ait $dg(n) < f(n)$. Considérons alors le mot $\langle M \rangle 10^k$. Il est de longueur strictement supérieure à n_0.

- S'il est élément de L, M l'accepte en espace $g(n)$, donc en espace inférieur à $f(n)$ donc $\hat{M}$ le rejette ;
- S'il n'est pas élément de L, M le rejette donc $\hat{M}$ l'accepte.

On aboutit donc à une contradiction, donc L ne peut pas être pas reconnu en espace $o(f(n))$. ■

Les quelques conséquences qui suivent, et que nous ne démontrerons pas explicitement, découlent immédiatement ce théorème.

corollaire 5.3

Pour toutes fonctions f_1 et f_2 de $\mathbf{N}$ dans $\mathbf{N}$ telles que $f_1(n) = o(f_2(n))$ et que $f_2(n)$ soit constructible en espace, on a

$$\text{SPACE}(f_1(n)) \subsetneq \text{SPACE}(f_2(n))$$

corollaire 5.4

$\text{NL} \subsetneq \text{PSPACE}$

On a en fait le résultat plus précis

$$\text{NL} = \text{NSPACE}(\log n) \subseteq \text{SPACE}\left(\log^2\right) \subsetneq \text{SPACE}(n)$$

corollaire 5.5

$\text{PSPACE} \subsetneq \text{EXPSPACE}$

Rappelons la définition

$$\text{EXPSPACE} = \bigcup_k \text{SPACE}\left(2^{n^k}\right)$$

Ici encore, on a en fait un résultat plus fin, à savoir :

$$\text{SPACE}\left(n^k\right) \subseteq \text{SPACE}\left(n^{\log n}\right) \subsetneq \text{SPACE}\left(2^n\right)$$

5.3 Théorème de hiérarchie en temps

Il est naturel de rechercher à présent des résultats équivalents sur la complexité en temps. On a également un théorème de hiérarchie dans ce cas.

théorème 5.6 (*théorème de hiérarchie en temps*)

Pour toute fonction t constructible en temps, il existe un langage décidé en temps $\mathcal{O}(t(n))$ qui ne l'est pas en temps $o(t(n)/\log n)$.

preuve : La méthode sera similaire au cas de la complexité en espace. On va donc considérer une machine $\hat{M}$ avec un certain nombre de rubans qui devra simules M sur l'entrée $\langle M\rangle\, 10^k$. Le problème est que M peut avoir plus de rubans que $\hat{M}$.

Or on a le résultat suivant :

J'ai pas suivi !!!

Considérons alors la machine $\hat{M}$ qui, sur l'entrée w :

1. calcule $|w| = n$;
2. calcule $t(n)$;
3. calcule $\lceil t(n)/\log n \rceil$ sur un compteur puis le décrémente avant chacune des opérations qui suivent et rejette w si le compteur atteint 0 ;
4. si w n'est pas de la forme $\langle M \rangle\, 10^k$, elle le rejette ;
5. sinon elle simule M sur w, et refuse w si et seulement si M l'accepte.

Par construction, cette machine convient donc et le résultat est démontré. ■

On a de nouveau les quelques conséquences remarquables suivantes, dont nous nous passerons de la démonstration :

corollaire 5.7

Pour toutes fonctions t_1 et t_2 telles que $t_1(n) = o\left(t_2(n)/\log t_2(n)\right)$ et que t_2 soit constructible en temps, on a

$$\text{TIME}(t_1(n)) \subsetneq \text{TIME}(t_2(n))$$

Et en conséquence immédiate :

corollaire 5.8

$\text{P} \subsetneq \text{EXPTIME}$

5.4 Lemme de translation

Le résultat suivant montre une certaine structure dans les classes de complexité telles que nous les considérons ici. Il permet en effet de généraliser une relation entre deux classes données à d'autres classes plus larges. Il s'agit du lemme suivant, que nous énonçons ici pour la complexité en espace.

lemme 5.9 (*lemme de translation*)

Soient s_1, s_2 et f trois fonctions constructibles en espace telles que pour tout n on ait $s_1(n) \geqslant n$ et $f(n) \geqslant n$. Si l'on suppose de plus que l'on a

$$\text{NSPACE}(s_1(n)) \subseteq \text{NSPACE}(s_2(n))$$

alors on a également

$$\text{NSPACE}\big(s_1(f(n))\big) \subseteq \text{NSPACE}\big(s_2(f(n))\big)$$

preuve : Supposons les conditions du lemme vérifiées et considérons un langage L_1 décidable en espace non déterministe $s_1(f(n))$, et soit M_1 une machine qui le décide dans cet espace. Considérons alors un nouveau caractère \$ et posons

$$L_2 = \left\{ w\$^i \,\middle|\, M_1 \text{ décide } w \text{ en espace } s_1(|w| + i) \right\}$$

Alors L_2 est décidé par le machine M_2 qui sur l'entrée $w\$^i$:

1. marque l'espace $s_1(|w|+i)$;
2. simule M_1 sur w ;
3. accepte si et seulement si M_1 accepte w en espace $s_1(|w|+i)$.

Alors d'après les hypothèses L_2 est dans NSPACE($s_2(n)$), et il existe donc une machine M_3 qui le décide en espace $s_2(n)$ non déterministe.

On construit alors la machine M_4 qui sur une entrée w commence par calculer $n=|w|$ et marquer un espace $s_2(f(n))$, puis simule M_3 sur $w\$^i$ pour des valeurs croissantes de i. Lorsque la tête de M_3 pénètre dans la zone en $\$^i$, M_4 le retient sur un compteur en espace $\log i$.

Enfin, si M_3 accepte durant le calcul, M_4 également, sinon M_4 augmente le comteur sur i, en continuant le processus jusqu'à atteindre la limite de $s_2(f(n))$. M_4 décide donc

$$L_3 = \{w \mid \exists i, |w|+i \leqslant s_2(f(|w|)), M_3 \text{ accepte } w\$^i\}$$

or M_3 décide $w\$^i \in L_2$ en espace $s_2(|w|+i)$ ■

5.5 Théorème de la lacune

Un petit pipeau sur ce théorème dû à Borodin.

théorème 5.10 (*théorème de la lacune*)

Soit g une fonction totale récursive talle que $g(n) \geqslant n$ pour tout n, alors il existe une fonction totale récursive s telle que

$$\text{SPACE}(s(n)) = \text{SPACE}\big(g(s(n))\big)$$

preuve : On va considérer une énumération (M_i) des machines de Turing. On note alors $s_i(n)$ le nombre maximum de cases visitées par M_i dans un calcul sur un mot de longueur n. Si M_i s'arrête sur toute entrée, s_i sera donc sa complexité en espace.

On cherche alors s telle que, pour tout k, on ait
- soit $s_k(n) \leqslant s(n)$ presque partout ;
- soit $s_k(n) \geqslant g(s(n))$ infiniment souvent.

Si une telle fonction existe, alors on a

khrâde

et donc

$$\text{SPACE}\big(g(s(n))\big) \subseteq \text{SPACE}(s(n))$$

■

sixième chapitre

Hiérarchie polynômiale, machines de Turing avec oracle

Au cours des précédents chapitres, on a donc établi la hiérarchie suivante :

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PS}\textsc{pace} = \mathrm{NS}\textsc{pace} \subseteq \mathrm{E}\textsc{xp}\mathrm{T}\textsc{ime}$$

On a aussi montré que certaines de ces inclusions sont strictes, plus précisément on a les résultats suivants :

$$\mathrm{NL} \neq \mathrm{PS}\textsc{pace} \quad \text{et} \quad \mathrm{P} \neq \mathrm{E}\textsc{xp}\mathrm{T}\textsc{ime}$$

Enfin, si $\mathrm{P} \neq \mathrm{NP}$, on a une hiérarchie entre les classes P et NP_c.

6.1 Machines de Turing avec oracle

On appelle oracle n'importe quel langage formel. C'est simplement son utilisation qui le fait qualifier d'oracle. On introduit alors la définition précise suivante :

définition 6.1 (*machine de Turing avec oracle*)

Une machine de Turing avec oracle est une machine de Turing qui contient en plus :
- *trois états particuliers, $q_?$, q_{oui} et q_{non} ;*
- *un ruban supplémentaire nommé* ruban d'oracle *;*
- *un langage A sur l'alphabet de travail, nommé* oracle.

Notons $\mathcal{M}^A$ une telle machine. Le fonctionnement est celui d'une machine de Turing normale, avec comme règle supplémentaire le fait que lorsque $\mathcal{M}^A$ passe dans l'état $q_?$, elle passe immédiatement dans l'état q_{oui} si le mot écrit sur le ruban d'oracle est dans A et dans l'état q_{non} sinon.
exemple :

1. Considérons le langage L défini par

$$L = \left\{ 0^i 1^i 0^i 1^j 0^j 1^j \,\middle|\, i \geqslant 1, j \geqslant 1 \right\}$$

Ce langage est par exemple décidé par un machine de Turing avec l'oracle

$$\{a^n b^n c^n \,|\, (a,b,c) \in \Sigma^3, n \geqslant 1\}$$

Il suffit en effet de recopier tous les 0 qui commencent, puis tous les 1 qui suivent, et enfin tous les 0 qui suivent, puis d'interroger l'oracle sur la partie copiée, et de recommencer avec la fin du mot.

2. Le langage défini par

$$L = \left\{a^i b^j c^k \,\middle|\, (i+j) \wedge (i+k) = 1\right\}$$

est avantageusement décidé avec l'oracle

$$\left\{1^i \,\middle|\, i \text{ est premier}\right\}$$

3. Ces exemples montrent certes le fonctionnement d'une machine avec oracle, mais ils ne permettent pas de décider quelque chose de plus qu'une machine de Turing classique.

On peut décider avec un oracle un langage qui est indécidable. Considérons en effet le langage L_u défini comme étant l'ensemble des codages $\langle M, w\rangle$ où M est une machine de Turing et w une entrée acceptée par la machine M. Le langage L_u est clairement récursivement énumérable mais indécidable. Posons alors

$$L_v = \{\langle M\rangle \,|\, L(M) = \emptyset\}$$

L_v est alors décidé par un machine de Turing avec L_u comme oracle. Considérons en effet la machine qui, sur l'entrée $\langle M\rangle$:

- détermine la machine N qui, sur l'entrée w :
  - énumère les couples (i,j) ;
  - simule M sur i pour un nombre de pas de calcul inférieur ou égal à j ;
  - accepte si M s'arrête dans ce laps de temps ou passe au couple suivant sinon.
- interroge l'oracle sur $\langle N, \emptyset\rangle$;
- rejette si l'oracle répond oui et accepte sinon.

Cette machine fournit alors la réponse voulue.

Une fois cette notion posée, on peut alors définir une hiérarchie de langages en posant :

$$S_1 = \{\langle M\rangle \,|\, L(M) = \emptyset\} \qquad S_{n+1} = \left\{\langle M\rangle \,\middle|\, L\left(M^{S_n}\right)\right\} = \emptyset$$

6.2 Relativisation, application à P et NP

Si A est un langage, on note P^A la classe des langages décidés en temps polynômial par un machine de Turing avec l'oracle A. On définit de même NP^A en utilisant des machines de Turing non déterministes avec l'oracle A, dont la définition est la même que pour les machines déterministes.

On a alors immédiatement le résultat suivant :

proposition 6.1

NP *et* co-NP *sont inclus dans les classe* $\mathrm{P}^{\mathrm{sat}}$.

preuve : C'est évident dans puisque si un langage L est dans NP, comme SAT est N-complet, L est réductible à SAT en temps polynômial. Comme la consultation de l'oracle est instantanée, la décision de L devient polynômiale. Pour CO-NP, il suffit de passer au complémentaire. ■

L'intérêt des oracles est qu'il permet de chercher de nouvelles méthodes pour préciser les hiérarchies de classes de complexité. On a en effet le résultat important suivant :

théorème 6.2

Il existe un langage A tel que $\mathrm{P}^A = \mathrm{NP}^A$.

preuve : Soit A un langage PSpace-complet. On va alors montrer que l'on a les inclusions suivantes :

$$\mathrm{PSpace} \subseteq \mathrm{P}^A \subseteq \mathrm{NP}^A \subseteq \mathrm{NSpace} = \mathrm{PSpace}$$

- Pour montrer que PSpace est inclus dans P^A, considérons un langage L de PSpace. On a alors $L \leqslant_{\mathrm{P}} A$ par définition de A. Soit alors la machine qui sur un entrée w
  1. calcule $f(w)$ où f rend compte de $L \leqslant_{\mathrm{P}} A$;
  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 $\mathrm{P}^A \subseteq \mathrm{NP}^A$ 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. ■

Ce résultat est certes intéressant, mais le résultat suivant limite nos espoirs :

théorème 6.3

Il existe un langage A tel que $\mathrm{P}^A \neq \mathrm{NP}^A$.

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

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

Alors clairement L_X est dans NP^X.

On va alors exhiber un langage B tel que $L_B \notin \mathrm{P}^B$. 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

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

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 $2^{n(i+1)}$.

Or les mots interrogés sur l'entrée $0^{n(i+1)}$ par T_{i+1} sont de longueur inférieure à $n(i+1)^{i+1} + i + 1$.

je suis a l'ouest une fois de plus

Et donc $L_B \notin \mathrm{P}^B$. ■

6.3 Turing-réduction

On définit à présent une notion de réduction analogue à la réduction polynôminale mais qui se fonde sur les oracles.

définition 6.2 (*Turing-réduction*)

Soient A et B deux langages, on dit que A est Turing-réductible à B et on écrit $A \leqslant_T B$ lorsque A est décidable en temps polynômial déterministe avec l'oracle B.

On définit de même la Turing-réduction non déterministe en utilisant un machine non déterministe dans la définition précédente, et on note dans ce cas $A \leqslant_{NT} B$.

On a alors plus qu'une analogie entre la Turing réduction at la réduction polynômiale :

proposition 6.4

- *La relation $\leqslant_T$ est un préordre sur l'ensemble des langages d'un alphabet donné ;*
- *Si $A \leqslant_P B$, alors $A \leqslant_T B$.*

On définit alors naturellement une notion de difficulté et une notion de complétude relativement à cette réduction :

définition 6.3 (*$\mathcal{C}_T$-dur, $\mathcal{C}_T$-complet*)

Soit $\mathcal{C}$ un classe de complexité. Un langage A est dit $\mathcal{C}_T$-dur s'il se

réduit par Turing-réduction à un langage de $\mathcal{C}$. Il est dit $\mathcal{C}_T$-complet s'il est lui-même dans $\mathcal{C}$.

On a alors les relations suivantes entre ces nouvelles propriétés et celles liées à la réduction polynômiale :

proposition 6.5

1. *Si A est $\mathcal{C}_T$-dur et si $A \leqslant_P B$, alors B est $\mathcal{C}_T$-dur.*
2. *Si A est $\mathcal{C}_T$-complet, si $A \leqslant_P B$ et si B est dans $\mathcal{C}$, alors B est $\mathcal{C}_T$-complet.*
3. *Si $A \leqslant_T B$ et $B \in \mathrm{P}$, alors $A \in \mathrm{P}$.*
4. *Si $A \leqslant B$ et $B \in$ PSPACE, alors $A \in$ PSPACE.*

Si $\mathcal{C}_T$ est une classe de complexité, on définit alors les classes de complexité suivantes :

$\mathbf{P}^{\mathcal{C}}$ est la classe des langages L tels qu'il existe un langage A pour lequel on a $A \in \mathcal{C}$ et $L \leqslant_T A$;

$\mathbf{NP}^{\mathcal{C}}$ est défini de même avec la condition $L \leqslant_{NT} A$.

6.4 La hiérarchie polynômiale

On désigne par *hiérarchie polynômiale* la structure

$$\left(\Sigma_k^P, \Pi_k^P, \Delta_k^P\right)_{k \geqslant 0}$$

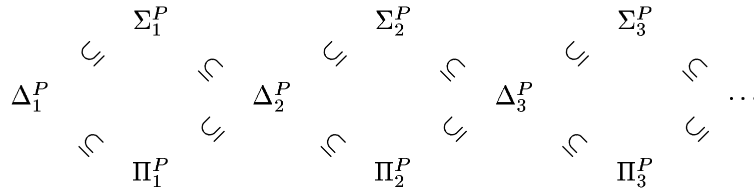
définie par les règles suivantes :

$$\Sigma_0^P = \Pi_k^P = \Delta_k^P = \mathrm{P}$$

et pour tout entier k :

$$\Sigma_{k+1}^P = \mathrm{NP}^{\Sigma_k^P} \qquad \Pi_{k+1}^P = \text{co-}\Sigma_{k+1}^P \qquad \Delta_{k+1}^P = \mathrm{P}^{\Delta_k^P}$$

Étant donnée cette définition, on a alors la hiérarchie suivante :



On introduit alors la définition suivante :

$$\bigcup_{k \geqslant 0} \Sigma_k^P = \bigcup_{k \geqslant 0} \Delta_k^P = \bigcup_{k \geqslant 0} \Pi_k^P = \mathrm{PH}$$

La classe PH ainsi définie est incluse dans PSPACE, ce qui se montre sans peine par récurrence.

6.5 Hiérarchie polynômiale, le retour

Introduisons tout d'abord les notations suivantes :

notation 6.6

Étant donnés un mot x, un entier n, un polynôme p et une propriété R sur les mots, on désigne par

$$\exists^{p(n)} x\, R(x)$$

le fait qu'il existe un mot x qui satisfasse R et qui soit de longueur au plus $p(n)$. De même, on désigne par

$$\forall^{p(n)} x\, R(x)$$

le fait que tout mot x de longueur au plus $p(n)$ qui satisfasse R.

notation 6.7

Soit $\mathcal{C}$ une classe de langages. On dénotera par $\exists\mathcal{C}$ la classe des langages A pour lesquels il existe un langage B de $\mathcal{C}$ et un polynôme p tels que

$$x \in A \iff \exists^{p(n)} y\ (\langle x, y\rangle \in B)$$

On définit de même la notation $\forall\mathcal{C}$.

On définit ensuite $A^{(*)}$ comme étant l'ensemble des n-uplets de mots de A, ou l'ensemble des suites finies sur A.

Une classe $\mathcal{C}$ est dite close par paire si pour tout langage A de $\mathcal{C}$ on a

$$\{\langle x, y\rangle \mid x \in A, y \in A\} \in \mathcal{C}$$

proposition 6.8

Pour toute classe de langages $\mathcal{C}$, on a

1. *$A \in \exists\mathcal{C}$ si et seulement si $\bar{A} \in \forall$co-$\mathcal{C}$;*
2. *si $\mathcal{C}$ est close par paire, alors $\mathcal{C} \subseteq \exists\mathcal{C}$ et $\mathcal{C} \subseteq \forall\mathcal{C}$.*

Avant de passer aux résultats généraux reliant ces définitions à celles de la hiérarchie polynômiale telle que nous l'avons vue précédemment, énonçons le lemme suivant :

lemme 6.9

Soit $\mathcal{C}$ l'une des classes Σ_k^P, Π_k^P et Δ_k^P. Alors pour tout langage A, on a

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

preuve : On énonce ici la preuve dans le cas où $\mathcal{C} = \Delta_k^P$. L'une des deux implications est triviale.

Soit A un langage de $\mathcal{C} = \mathrm{P}^{\Sigma_{k-1}^P}$. Il existe alors un langage B dans Σ_{k-1}^P tel que A soit élément de P^B. Ceci signifie qu'il existe une machine

de Turing déterministe M qui décide A en temps polynômial avec l'oracle B.

Pour montrer que $A^{(*)}$ est élément de Δ_k^P, on considère la machine M' correspondant à l'algorithme suivant :

1. sur l'entrée $\langle y_1, \dots, y_n \rangle$, M' décode son entrée en $y_1, \dots, y_n$;
2. elle simule M sur les y_i successivement ;
3. elle accepte si et seulement si M accepte toutes ces entrées.

La machine M' décide alors A en temps polynômial avec l'oracle B, puisque M marche en temps polynômial, ce qui montre que $A^{(*)}$ est dans P^B. ■

On a alors les liens suivants avec la hiérarchie précédemment définie :

proposition 6.10

1. $\exists\mathrm{P} = \mathrm{NP} = \Sigma_1^P$
2. $\forall\mathrm{P} = \text{co-NP} = \Pi_1^P$
3. *pour* $k > 0$, *on a* $\exists\Sigma_k^P = \Sigma_k^P$
4. *pour* $k > 0$, *on a* $\forall\Pi_k^P = \Pi_k^P$
5. *pour* $k \geqslant 0$, *on a* $\forall\Pi_k^P = \Sigma_{k+1}^P$
6. *pour* $k \geqslant 0$, *on a* $\exists\Sigma_k^P = \Pi_{k+1}^P$

annexe A

Solution des exercices

Exercice 1.1

énoncé page 3

On a en entrée un mot w sur l'alphabet $\{0, 1\}$. On ajoute deux caractères de contrôle : $\sharp$, et $*$. Le principe sera d'abord d'ajouter le symbole $*$ à droite de w, puis d'utiliser ce symbole de contrôle comme curseur pour l'addition. Comme on ne commence qu'avec w sans caractère de contrôle, et qu'on ne peut ajouter de caractère qu'en tête de mot lorsqu'il n'y a encore aucun repère, il faut commencer par ajouter le symbole $\sharp$ en tête de mot, puis le faire glisser en fin de mot.

On ne pourra ajouter le $\sharp$ qu'à l'aide d'une règle agissant sur le mot vide, et on placera donc cette règle en fin de liste. On fait ensuite passer le $\sharp$ initial en fin de mot par les règles suivantes :

$$\sharp 0 \longrightarrow 0\sharp$$

$$\sharp 1 \longrightarrow 1\sharp$$

Une fois le symbole en fin de mot, on initialise l'incrémentation en transformant le $\sharp$ en $*$:

$$\sharp \longrightarrow *$$

On décrit ensuite le mécanisme d'incrémentation, qui termine dès que l'on trouve le chiffre 0. Si l'on ne trouve pas de chiffre 0, on ajoute un 1 en tête de mot :

$$0* \longrightarrow_t 1$$

$$1* \longrightarrow *0$$

$$* \longrightarrow_t 1$$

Et on ajoute donc la dernière règle :

$$\varepsilon \longrightarrow \sharp$$

Exercice 1.2

énoncé page 3

Le principe sera le même que pour le successeur, à cela près qu'il faut maintenant prendre en compte le cas où le nombre en entrée est nul, donc composé uniquement du chiffre 0. Pour cela, on profite du parcours initial servant à placer un marqueur $*$ en fin de mot, en ajoutant un troisième symbole de contrôle $\sharp'$, qui indiquera qu'un 1 a été trouvé :

$$\sharp 0 \longrightarrow 0\sharp$$

$$\sharp' 0 \longrightarrow 0\sharp'$$

$$\sharp 1 \longrightarrow 1\sharp'$$

$$\sharp' 1 \longrightarrow 1\sharp'$$

Ensuite, si on n'a pas trouvé de 0, donc si on trouve un $\sharp$ en fin de mot, on laisse le mot inchangé en terminant immédiatement, sinon on place le marqueur $*$:

$$\sharp \longrightarrow_t \varepsilon$$

$$\sharp' \longrightarrow *$$

Lorsque le marqueur $*$ se trouve en fin de mot, on parcourt le mot en mettant les 0 à 1 et en s'arrêtant dès que l'on atteint un 1 en le mettant à 0 :

$$0* \longrightarrow *1$$

$$1* \longrightarrow_t 0$$

Et come le seul cas où l'on ne rencontre pas de 1 est éliminé dès le premier parcours, l'algorithme termine toujours correctement.

Exercice 1.3 — *équivalence RAM / Turing*

énoncé page 4

question1 – On comence par définir quelques abréviations. La notation « Jump X » désignera le programme

```
jmp   0, X
jmp   1, X
      ...
jmp   k, X
```

en appelant a_0 le caractère correspondant à la virgule. La notation « Rotate » désignera ensuite

```
        jmp    0, copy_0(b)
        ...
        jmp    k, copy_k(b)
copy_0  add    0
        Jump   suite(b)
        ...
copy_k  add    k
        Jump   suite(b)
suite   del
```

On pourra alors utiliser le programme suivant, dont on conviendra qu'il fournit le bon résultat sans démonstration plus explicite.

	add	0
loop	jmp	$0, \mathrm{end}(b)$
	Rotate	
	Jump	$\mathrm{loop}(a)$
end	del	

question2 – Ajoutons une lettre $\star$ à l'alphabet, par exemple en lui donnant l'indice -1. On va simuler une RAM par une SRM en considérant que l'on connaît l'indice maximal n utilisé dans l'accès au registres. On représente alors un état de la machine RAM $(R_1, \ldots, R_n)$ par l'état $\star, R_1, \ldots, R_n$ de la SRM correspondante. Si le précédent programme est désigné par RotateReg, on définit le programme Init suivant :

loop	jmp	$-1, \mathrm{end}(b)$
	RotateReg	
	Jump	$\mathrm{loop}(a)$
end		

Ce programme a pour rôle de rétablir la situation initiale avec le repère $\star$ en début de mot. On définit alors « Reg k » comme le programme qui place en tête de mot la représentation du k-ième registre. Ce programme s'obtient en plaçant successivement Init et $k-1$ occurrences de RotateReg.

Pour passer d'un programme RAM au programme SRM correspondant, on remplacera alors les instructions comme suit :

add j, R_p devient

Reg	$p+1$	(ou rien si $p = n$)
add	j	

del R_p devient

Reg	p
del	

jmp j, R_p, X devient

	Reg	p
	jmp	j, doJump
	Jump	noJump
doJump	Init	
	Jump	X
noJump	Init	

Les autres instructions s'en déduisent de façon intuitive.

question3 – Considérons une SRM sur un alphabet A_k. On construit à partir du programme SRM une machine de Turing sur $A_k \cup \{\Diamond, \Box\}$ où les deux symboles représentent respectivement le début et la fin du mot. On remplace alors chaque instruction SRM par un couple d'états de la machine de Turing, le premier état servant à rechercher le début du mot (dans le cas du saut et de l'effacement) ou la fin du mot (dans le cas de l'ajout d'une lettre), et le second état, sortie de la boucle de recherche, servant à effectuer l'opération voulue sur la lettre concernée. On relie ensuite les états de façon appropriée pour respecter le déroulement du programme ou les sauts.

On ajoute enfin des états au début pour ajouter le symbole $\Diamond$ en début de mot et le symbole $\Box$ en fin de mot, et des états à la fin pour les effacer, et on obtient une machine de Turing qui se comporte de la même façon que la SRM initiale.

On a donc montré que toute fonction calculable par RAM est calculable par machine de Turing.

Exercice 1.4 — *lemme de remplissage fort*

énoncé page 10

Pas encore de solution ... mais l'idée est de construire ρ' à partir du ρ précédent.

Exercice 1.5

énoncé page 12

L'existence d'un machine universelle Φ_u implique immédiatement que pour tousentiers i, j et x on a

$$\Phi_i(x) + \Phi_j(x) = \Phi_u(\langle i, x\rangle) + \Phi_u(\langle j, x\rangle)$$

or la fonction qui à $\langle i, j, x\rangle$ associe $\Phi_u(\langle i, x\rangle) + \Phi_u(\langle j, x\rangle)$ est récursive, par définition des fonctions récursives, elle a donc un indice u^+. Par conséquent on a, pour tous i, j et x :

$$\Phi_i(x) + \Phi_j(x) = \Phi_{u^+}(\langle i, j, x\rangle) = \Phi_{s_2^1(u^+,i,j)}(x)$$

par conséquent on peut poser $p(i, j) = x_2^1(u^+, i, j)$, ce qui fait de p une fonction récursive primitive.

Exercice 1.6 — *caractérisation des s.p.a.*

énoncé page 12

question1 – Soit (Φ_i) un système de programmation acceptable et u l'indice d'une fonction universelle. Pour tout triplet d'entiers (i, j, x), on a :

$$\begin{aligned}\big(\Phi_i \circ \Phi_j\big)(x) &= \Phi_i\big(\Phi_j(x)\big)\\ &= \Phi_i\big(\Phi_u(\langle i, x\rangle)\big)\\ &= \Phi_u\big(\,\langle i, \Phi_u(i, x)\rangle\,\big)\end{aligned}$$

En tant que composée de fonctions récursives, cette fonction est récursive en tant que fonction $\langle i, j, x\rangle$. Notons alors u° l'indice de cette fonction. On a donc

$$\big(\Phi_i \circ \Phi_j\big)(x) = \Phi_{s_2^1(u^\circ,i,j)}(x)$$

Donc on peut poser $c(i, j) = s_2^1(u^\circ, i, j)$ et c est bien récursive totale.
question2.a –

Exercice 1.7 — *théorème du point fixe*

énoncé page 12

On cherche à montrer l'existence d'un entier i tel que pour tout x on ait :

$$\Phi_i(x) = \Phi_{\alpha(i)}(x)$$

On peut reformuler cette égalité et écrire :

$$\Phi_u(i, x) = \Phi_u\big(\alpha(i), x\big)$$

Exercice 4.1 — *constructibilité en espace* *énoncé page 21*

question1 – Il suffit de considérer une machine qui commence par se positionner au début du mot d'entrée puis le copie linéairement jusqu'à parvenir à son extrémité.

question2 – On commence à nouveau par se positionner en début de mot puis on lit le ruban d'entrée deux lettres par deux lettres. En terme d'états, on peut définir la machine par ses états, en considérant l'entrée en lecture seule et la sortie en écriture seule. La notation $\cdot$ désigne l'abscence d'action :

commentaire	état	lecture d'un blanc	lecture de 1
recherche du début de mot	0	$\cdot, +, 1$	$\cdot, -, 0$
lecture d'une lettre impaire	1	$\cdot, \cdot, \text{stop}$	$\cdot, +, 2$
lecture d'une lettre paire	2	$1, \cdot, \text{stop}$	$1, +, 1$

question3 – Définissons le fonctionnement de la machine dans ce cas de façon algorithmique. On suppose qu'on dispose d'une lettre 0 en plus du blanc et de la lettre 1.

On a d'abord une procédure COPIE qui se déplace au début du mot d'entrée, le copie sur le ruban de sortie, puis remplace le dernier 1 par un 0 sur la sortie.

On a ensuite une procédure DÉCOMPTE qui commence par se déplacer au début des rubans d'entrée et de sortie, puis parcourt le ruban de sortie jusqu'au bout en avançant la tête du ruban d'entrée chaque fois qu'il rencontre un 0 sur la sortie. Elle finit sur la première case libre du ruban de sortie.

L'algorithme général sera alors le suivant :

placer la tête du ruban d'entrée en début de mot

tant que la tête d'entrée est placée sur un 1 **faire**

exécuter COPIE

exécuter DÉCOMPTE

fin tant que

placer la tête de sortie en début de mot

parcourir le mot de sortie en remplaçant tous les 0 par des 1

Si l'entrée est 1^n, la boucle s'exécute bien n fois et elle ajoute n lettres à la sortie à chaque étape, donc on a bien 1^{n^2} sur le ruban de sortie en fin de processus.

question4 – On procède à nouveau de façon algorithmique. Le principe sera de conserver sur la sortie un compteur en binaire (en supposant donc que l'on dispose du caractère 0) que l'on incrémentera en ajoutant une copie de l'entrée à chaque étape. L'algorithme sera donc le suivant sur l'entrée 1^n :

placer la tête d'entrée en début de mot

écrire un 0 pour chaque 1 sur l'entrée

répéter

placer la tête de sortie en début de mot

incrémenter le nombre représenté par les n premiers caractères

si le nombre obtenu contient un 0 **alors**

placer la tête d'entrée en début de mot
placer la tête de sortie en fin de mot
copier le mot d'entrée sur la sortie

fin si

jusqu'à ce que le mot obtenu ne contient que des 1

On a donc fait une copie de l'entrée par mot sur $\{0,1\}$ de longueur n contenant un 0, ce qui représente $2^n - 1$ copies.

annexe B

Listes

B.1 Liste des définitions

1.1 - algorithme de Markov 2
1.2 - circuit booléen 5
1.3 - famille uniforme de circuits booléens 8
1.4 - automate cellulaire 8
1.5 - système de programmation acceptable 9
3.1 - classe de complexité en temps 17
3.2 - classe P 18
3.3 - machine de Turing non déterministe 18
3.4 - classe NP 19
6.1 - machine de Turing avec oracle 29
6.2 - Turing-réduction 32
6.3 - $\mathcal{C}_T$-dur, $\mathcal{C}_T$-complet 32

B.2 Liste des théorèmes

1.6 - corollaire 7
1.10 - lemme de remplissage 10
1.12 - isomorphisme de Rogers 11
5.2 - théorème de hiérarchie 24
5.6 - théorème de hiérarchie en temps 25
5.9 - lemme de translation 26
5.10 - théorème de la lacune 27

B.3 Liste des exercices

1.3 - équivalence RAM / Turing 4
1.4 - lemme de remplissage fort 10
1.6 - caractérisation des s.p.a. 12
1.7 - théorème du point fixe 12
4.1 - constructibilité en espace 21